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

GUILHERME BUENO DE OLIVEIRA

Managing and Securing Programmable Virtual Switches with PvS

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

Advisor: Prof. Dr. Weverton Cordeiro
Coadvisor: Prof. Dr. José Rodrigo Azambuja

Porto Alegre
March 2021

CIP — CATALOGING-IN-PUBLICATION

Bueno de Oliveira, Guilherme

Managing and Securing Programmable Virtual Switches with PvS / Guilherme Bueno de Oliveira. – Porto Alegre: PPGC da UFRGS, 2021.

84 f.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2021. Advisor: Weverton Cordeiro; Coadvisor: José Rodrigo Azambuja.

1. Software Defined Networking (SDN). 2. Programmable Data Planes (PDP). 3. Virtualization. 4. Information Flow Control (IFC). 5. Data Provenance. I. Cordeiro, Weverton. II. Rodrigo Azambuja, José. III. Título.

"If I have seen farther than others,
it is because I stood on the shoulders of giants."
— SIR ISAAC NEWTON

ACKNOWLEDGMENTS

I would like to thank my mother Marisa who has always been supporting me, caring, educating, and helping me. Without her, I would not be who I am today. Many other family members also showed their support, and I am grateful.

Also, I would like to thank my advisor Weverton Cordeiro and coadvisor José Azambuja, who allowed me to work in the project, and guided me throughout the entire course even when there were difficult moments in life. In addition, I would like to thank them, along with professor Kirill Levchenko, for the enriching experience of visiting the University of Illinois at the Urbana-Champaign (UIUC) campus to present our work. I would also like to show my gratitude for Ivan Lamb and Willian Reichert who helped construct this work. Thank you to several other friends like Mateus Saquetti, and others, that showed their support and I was able also to learn from them.

ABSTRACT

Virtualization has become an important enabler of several concepts, like cloud computing, network function virtualization, and virtual networks, helping foster innovation and tackle the *network ossification* that lasted for decades. With programmable data planes following the path of virtualization, existing solutions to deliver the notion of virtual programmable switches fall short in providing effective abstractions of switches that could be managed independently and securely. To bridge this gap, we present PvS, a system for running multiple Programmable Virtual Switches that satisfies these requirements. In our work, we focus on the control engine abstraction, responsible for managing virtual switches running in an underlying hardware (e.g., NetFPGA) and for providing compatible management interfaces with the control plane of a Software Defined Network (SDN). With PvS, we also concentrate on a potential security vulnerability regarding virtual switches, which is the “poisoning” between control plane applications (Cross-App Poisoning, or CAP, attacks) by a malicious control plane app, using virtual switches as proxy for the attack. To this end, we devise an Information Flow Control (IFC) enforcement solution to virtual switches (vIFC), to detect information flow violations from a malicious application to legitimate ones in the control plane through virtual switches. We experimented PvS using virtual switches running in a NetFPGA SUME, and assessed its effectiveness to securely manage virtual instances and prevent information flow violation in the control plane. We analyzed the operational impact of CAP attacks and the protection capabilities that vIFC provides by defending virtual switches considering two use cases: a Reactive Forwarding app, and the Inband Telemetry app. Our evaluation provides evidence that PvS is effective in providing secure manageability and detect attacks like the Cross-App Poisoning (CAP), while not incurring significant overhead.

Keywords: Software Defined Networking (SDN). Programmable Data Planes (PDP). Virtualization. Information Flow Control (IFC). Data Provenance.

Gerenciando e Protegendo Switches Virtuais Programáveis com PvS

RESUMO

A virtualização se tornou um importante habilitador de vários conceitos em rede, como computação em nuvem, virtualização de função de rede e redes virtuais, ajudando a promover a inovação e enfrentar a “ossificação de redes” que durou décadas. Com planos de dados programáveis seguindo o caminho da virtualização, nota-se que as soluções existentes para entregar a noção de *switches* programáveis virtuais não fornecem abstrações eficazes de switches que possam ser gerenciados de forma segura e independente. Para preencher essa lacuna, apresentamos o PvS, um sistema para executar vários *switches* virtuais programáveis e que satisfaz esses requisitos. Em nosso trabalho, nos concentramos na abstração do mecanismo de controle, responsável por gerenciar *switches* virtuais em execução em um *hardware* subjacente (por exemplo, NetFPGA) e por fornecer interfaces de gerenciamento compatíveis com o plano de controle de uma Rede Definida por Software (*Software Defined Networking*, SDN). Com o PvS, também nos concentramos em uma vulnerabilidade de segurança potencial em relação aos switches virtuais, que é o “envenenamento” de uma *app* do plano de controle por outra *app* maliciosa (ataques *Cross-App Poisoning*, ou CAP), usando *switches* virtuais como *proxy* para o ataque. Para este fim, desenvolvemos uma solução de aplicação de Controle de Fluxo de Informações (*Information Flow Control*, IFC) para *switches* virtuais (vIFC), para detectar violações de fluxo de informações de um app malicioso para *apps* legítimos no plano de controle via *switches* virtuais. O PvS foi avaliado considerando *switches* virtuais em execução em um NetFPGA SUME e avaliou-se sua eficácia para gerenciar com segurança as instâncias de *switches* virtuais e evitar a violação do fluxo de informações no plano de controle. Analisamos o impacto operacional dos ataques CAP e os recursos de proteção que o vIFC fornece ao defender *switches* virtuais considerando dois casos de uso: um *app* de encaminhamento reativo e o app de telemetria *in-band*. A avaliação realizada oferece evidências de que o PvS é capaz de fornecer capacidade de gerenciamento segura e detectar ataques cibernéticos como o *Cross-App Poisoning* (CAP), sem incorrer em sobrecarga significativa para o plano de controle ou para os *switches* virtuais.

Palavras-chave: Redes Definidas por Software (SDN), Plano de Dados Programáveis (PDP), Virtualização, Controle de Fluxo de Informação (IFC), Proveniência de Dados.

LIST OF ABBREVIATIONS AND ACRONYMS

API	Application Programming Interface
ARP	Address Resolution Protocol
BMv2	Behavioral Model version 2
CAP	Cross-App Poisoning
CDF	Cumulative Distribution Function
CDPI	Control-Data-Plane Interface
CLI	Command Line Interface
CPU	Central Processing Unit
DDoS	Distributed Denial-of-Service
DSL	Domain Specific Language
FPGA	Field-Programmable Gate Array
fwd	Reactive Forwarding
gRPC	google Remote Procedure Calls
IaaS	Infrastructure as a Service
IFC	Information Flow Control
INT	Inband Telemetry
IP	Internet Protocol
IvSI	Input virtual Switch Interface
MAC	Media Access Control
MRI	Multi-Hop Route Inspection
NFV	Network Function Virtualization
OF	OpenFlow
ONF	Open Networking Foundation
ONOS	Open Network Operating System

OvSI	Output virtual Switch Interface
P4	Programming Protocol-Independent Packet Processors
PaaS	Platform as a Service
PCIe	Peripheral Component Interconnect Express
PDP	Programmable Data Planes
PI	Protocol Independent
PISA	Protocol Independent Switch Architecture
POF	Protocol Oblivious Forwarding
PvS	Programmable virtual Switches
RPC	Remote Procedure Calls
SDN	Software-Defined Networking
SaaS	Software as a Service
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UDP	User Datagram Protocol
vIFC	virtual Information Flow Control
VLAN	Virtual Local Area Network
VM	Virtual Machine
vSDN	Virtual Software Defined Networking
VSMA	Virtual Switch Management API
vS	Virtual Switch
VSS	Very Simple Switch
XaaS	Everything as a Service

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

The networking community has long taken advantage of virtualization to realize a multitude of services dubbed XaaS (*anything-as-a-service*) (BARI et al., 2012; JAIN; PAUL, 2013). Popular examples include *infrastructure*, *platform*, and *software* as a service (IaaS, PaaS, and SaaS, respectively). In this context, virtualization has been used to leverage abstractions of several networking components, e.g. links (BARI et al., 2012; BELT; AHMADI; DOYLE, 2017), forwarding elements (switches, routers, etc.) (PFAFF et al., 2015; SHAHBAZ et al., 2016; KUMAR et al., 2019), servers (KOPONEN et al., 2014a), management units (JIN et al., 2015; BLENK et al., 2015; AFOLABI et al., 2018), among various others.

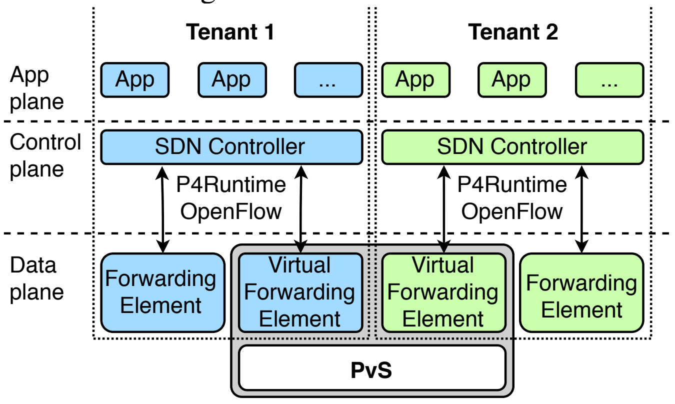
With the emergence of Software Defined Networking (SDN) (MCKEOWN et al., 2008) and Programmable Data Planes (PDPs) (BOSSHART et al., 2013), and the possibility to redefine the behavior of forwarding elements through home-brewed software (BOSSHART et al., 2014; SONG, 2013; GAO et al., 2020; Broadcom Inc., 2019), researchers have been investigating how to leverage abstractions of virtual programmable forwarding elements (HANCOCK; MERWE, 2016; ZHANG et al., 2019; ZHENG; BENSON; HU, 2018; KRUDE et al., 2019; SAQUETTI et al., 2019; SAQUETTI et al., 2020). The ability to achieve virtualization in programmable forwarding planes follows the research trend of devising a *flexible* and *extensible* Internet (MCCAULEY et al., 2019), facilitating the design, deployment, interoperability, and co-existence of various network protocols in the existing infrastructure.

Despite the potentialities provided by PDP virtualization, existing solutions do not cope well with the forwarding device management abstractions required in the context of SDN. Solutions like HyPer4 (HANCOCK; MERWE, 2016), and HyperVDP (ZHANG et al., 2019), for example, do not provide an abstraction to enable network operators to manage virtual switch instances independently. In other words, a network operator managing a virtual switch instance could easily tamper with the other switch instances running on top of the same virtualization solution, or hypervisor. Equally importantly, existing solutions lack strict performance and isolation guarantees and/or impose significant overhead to network flow processing and network management. They also lack proper mechanisms to prevent attacks against the control plane of an SDN using virtual switches as a proxy. According to (DACIER et al., 2017), attacks against SDN controllers and malicious SDN apps are the most severe threats to a Software-Defined Network. Considering a virtualization solution for PDPs, attacks can affect virtual abstractions' integrity and availability despite still not being much explored. For instance, a malicious tenant might try to overload or even poison virtual switch instances using poorly-designed management abstractions provided by existing PDP virtualization solutions.

In this work, we argue in favor of a PDP virtualization solution that enables network operators to deploy and manage multiple virtual switch instances. The management abstractions must guarantee secure and compatible management with open control-data plane interfaces and ensure that the abstraction does not pose significant overhead to the network operation and protect against attacks. To guarantee these principles within such a solution, we defend that a hypervisor should offer an abstraction of a single control engine and management channel to network apps running in the app plane while maintaining consistent security levels.

We provide such solution abstractions in PvS, a system for running multiple Programmable virtual Switches in the PDP. PvS supports the ability to manage the virtual instances offering security guarantees, privacy, and support to open southbound interfaces. Figure 1.1 provides an overview of PvS, in which multiple network operators (or tenants) manage the virtual switches through a secure control channel using a management interface of one's choice, for example, P4Runtime or OpenFlow, which allows the possibility that each tenant may choose a different management interface. Also, the network operator can use several network applications to define the virtual switch's behavior, deploy new virtual switches, and determine its pipeline configuration, with an efficient security level through authentication and access control methods provided by PvS. Considering the topology of the overview in Figure 1.1, the tenant may manage the forwarding element and, with the use of PvS, manage another completely independent virtual element.

Figure 1.1: PvS Overview



Source: From the Author

To secure virtual switch instances from malicious apps running in the app plane, we enforce information flow control (IFC) between virtual switches (vIFC) and apps in the control plane. Such enforcement seeks to prevent a type of attack known as Cross-App Poisoning (CAP) (UJCICH et al., 2018), a control plane integrity attack in which malicious apps poisons other apps (through shared objects) to perform bogus actions, using other apps' permissions. Our goal is to prevent malicious apps in the control plane from using virtual switches as a proxy for launching CAP attacks against other apps running in the control plane. Our solution, vIFC, uses a data provenance graph to analyze if the information flow (i.e., a management operation like packet-in, flow rule insertion, etc.) between apps and virtual switches is consistent with the enforcement of information flow control policies determined by a network manager. As a result, a bogus management operation initiated by a malicious app can be detected and stopped before "poisoning" any legitimate app.

1.1 Research Challenges

Devising a solution in which multiple tenants can manage several virtual abstractions while guaranteeing consistent security levels and low overhead is challenging. A malicious tenant, controlling a single virtual switch, may intend to interfere with other virtual switches' operation or poison the control plane apps from other tenants, to rephrase it, network operators cannot manage and deploy the virtual instances independently and securely. Therefore, in this proposal, we address the main research question, which consists of whether is possible to deliver a management interface suitable for virtual switches and that is independently of the virtualization solution. This research challenge defends that the interface may allow any virtualization solution to manage the virtual switch instances independently while posing only a marginal overhead. We also delineate this proposal's scope considering specific virtual switches' management and security challenges.

Security challenges were present in both the control engine and vIFC abstractions regarding how to avoid unauthorized tenants' access or correlation of information flows of applications and virtual switches. Therefore, the challenges faced may include: (i) how to ensure that tenants are not able to interfere in virtual abstractions that they do not possess the determined permissions, (ii) how to guarantee a secure management channel that the controller may connect to the control engine, (iii) how to correlate information flow events between apps and virtual switches, and (iv) How to trace flow events in the data plane to originating/related apps in the control plane.

Particular management challenges in designing the abstractions include: (i) what hypervisor management design may enable tenants to operate their virtual switch instances transparently, using known standards like P4Runtime? (ii) How to manage multiple virtual abstractions through a single communication channel.

1.2 Contributions

In this proposal, we present the PvS solution, a system capable of running multiple programmable virtual switches in parallel on top of a hardware substrate, in which the tenants may control and manage the virtual switch abstractions with a consistent security level. PvS is divided in the forwarding engine slicing (already presented by (TIRONE, 2020)) and the control engine abstraction. Our first main focus is in the control engine abstraction featuring the system's design, implementation, and other aspects. This work's other main focus is the vIFC abstraction, featuring the threat model and case study of how a CAP attack may infiltrate the network, the design to mitigate it, and the implementation.

Since the state of the art solutions lack in providing a management abstraction that allows virtual instances to be managed independently and securely. Also, they lack in providing a mechanism to avoid certain type of poisoning attacks that use virtual switches as proxies. Therefore, we intend to contribute to these virtualization solutions by providing the PvS control engine abstraction to allow secure and independent virtual switch management by the network operator with any virtualization solution that he may chooses. In addition, we intend to contribute to them with the vIFC abstraction to protect virtual switches and SDN application against poisoning attacks like the CAP attack. More specifically, the following contributions made by this proposal are:

- **The design and implementation of the Control Engine abstraction.** The abstraction of independent switch management interfaces to support virtual switch control from multiple tenants over the (shared) hypervisor interface. Our abstraction ensures secure access control, shared policy management among virtual switches, and compatibility with existing management specifications. Then, the network operator may deploy and manage the virtual instances using any virtualization solution and with isolation between the instances, avoiding tampering from unauthorized users. Also, allows the network operator to use specific southbound interface protocols, and not cause a significant overhead to the virtualization solution.

- **Defending Virtual Switches from Cross-App Poisoning Attacks with vIFC.** An abstraction solution for information flow control (IFC) policy enforcement with virtualization solutions. The solution ensures data provenance into the virtualization solution, confirming no information flow violations by analyzing the provenance graph with the stored policies. Since the state of the art solutions contained vulnerabilities in tracking flows between virtual switches, we bring this contribution with the vIFC abstraction, allowing to monitor the data provenance with any PDP virtualization solutions in the scenario, which may happen information flow between virtual switch instances. Therefore, we contribute by increasing the SDN network security state for users, by tracking and mitigating this type of poisoning attack, and causing marginal overhead.
- **A public, open-source, implementation of PvS and vIFC on GitHub.** The PvS Control Engine repository contains the control engine implementation modules, along with several SDN applications that are connected directly with P4Runtime modules, or with ONOS SDN controller. The vIFC repository has the information flow control policy enforcement code and the control engine, also there are SDN apps that exploit the CAP attack and vIFC can tackle it. Tutorials to manage the virtual switches or exploit the attack are also available in the repositories.

1.3 Outline

This dissertation is organized with the following structure. In Chapter 2 we discuss the theoretical background that form the basis of this work, ranging from basic concepts to operational functions. In Chapter 3 we cover related work. In Chapter 4 we present an overview of PvS, focusing on its design principles and associated information. Then, we describe the PvS Control Engine abstraction, mentioning specific challenges and requirements to build it and describing the abstraction modules. In Chapter 5, we discuss our solution for virtual Information Flow Control (vIFC) enforcement to tackle CAP attacks in a Software-Defined Network. In Chapter 6, we firstly discuss an integration of the control engine with a NetFPGA SUME. Then, we approach the results gathered in terms of the impact of the control engine on the management of the virtual switches, along with results of CAP attack operational impact, and the efficiency of vIFC to stop them. We close the dissertation in Chapter 7, presenting concluding remarks and discussing directions for future research.

2 BACKGROUND

This chapter aims to discuss relevant concepts to provide a better understanding of the dissertation's proposal. In Section 2.1, we discuss the Software-Defined Networking (SDN) architecture, mentioning important concepts within it. Then, following the SDN architecture, we characterize and exemplify concepts and technologies that may be related to a certain part of the architecture throughout the next sections. In Section 2.2, several concepts are discussed, like programmable data planes, and the P4 language. In Section 2.3, we mention the protocols used in the southbound interface that connect the control and data plane, like the P4Runtime. Then, in Section 2.4, we show some concepts related to the control plane exemplifying important SDN controllers. At last, in Section 2.5, we discuss virtualization by approaching a general view of how it works, and virtualization in programmable data planes which is important for the proposal.

2.1 Software Defined Networking

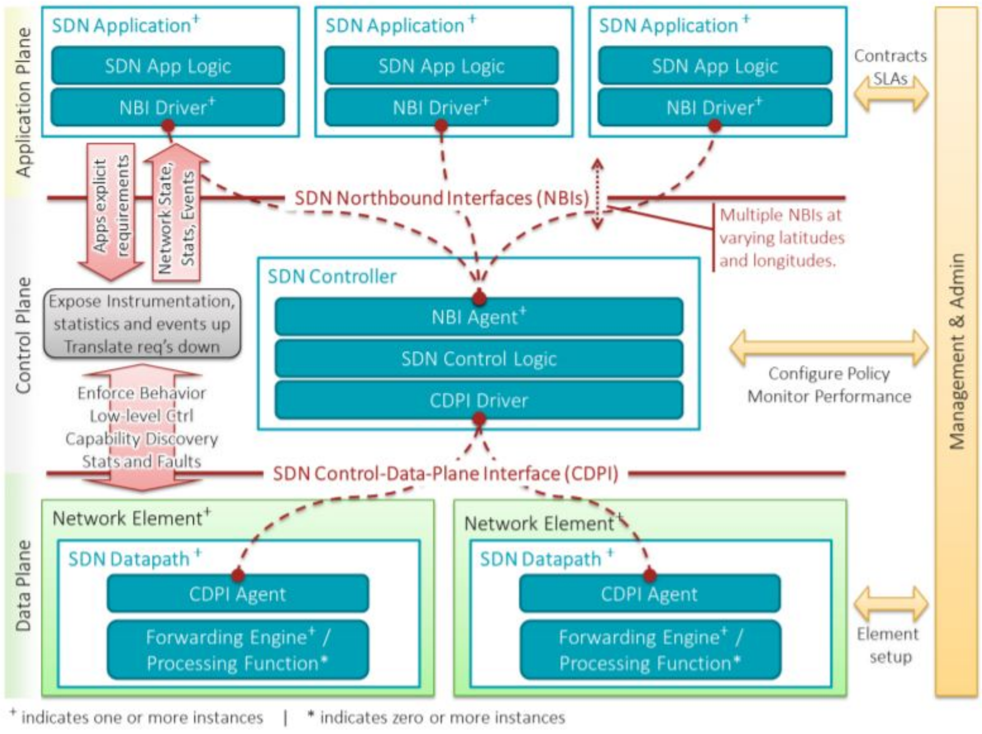
Software-Defined Networking, as cites (ONF, 2013), has the purpose of providing open interfaces that enable software development that controls the connectivity provided by network resources and the flow of network traffic through them, along with possible inspection and modification of traffic that may be performed in the network.

An SDN network is organized into application, control, and data planes, where these interfaces and components are placed. The application plane consists of the layer where SDN applications perform their functionalities and communicate with the control using Northbound interfaces. The control plane receives the information and relays them to the networking elements while maintaining communications with the SDN apps sending information about network topology, statistics, and more. The data plane consists of the networking components and a Control-Data-Plane Interface (CDPI) Agent that manages the connection of the network elements with the information received/sent in the Southbound Interface using a management interface like P4Runtime or Openflow.

Figure 2.1 shows a high-level view of the entire SDN architecture, where the application, data, and control plane meet. Some major concepts were discussed in the beginning of Section 2, however, the entire architecture has several specific components that have been addressed in (ONF, 2013) and they consist of:

SDN Applications. They consist of programs that intend to program the behavior

Figure 2.1: Overview of Software-Defined Networking Architecture


Application Plane
Control Plane
Data Plane
SDN Application+
SDN App Logic
NBI Driver+
Contracts SLAs
Apps explicit requirements
Network State, Stats, Events
SDN Northbound Interfaces (NBIs)
Multiple NBIs at varying latitudes and longitudes.
SDN Controller
NBI Agent+
SDN Control Logic
CDPI Driver
Expose Instrumentation, statistics and events up Translate req's down
Enforce Behavior Low-level Ctrl Capability Discovery Stats and Faults
Configure Policy Monitor Performance
Management & Admin
SDN Control-Data-Plane Interface (CDPI)
Network Element+
SDN Datapath+
CDPI Agent
Forwarding Engine+ / Processing Function*
Element setup
+ indicates one or more instances | * indicates zero or more instances

Source: (ONF, 2013)

of the network elements. The information that they create, is sent to the SDN controller through the Northbound interface and also receives network topology information, statistics, and more data. For instance, a simple application that gets information about the switch and installs a flow rule to enable forwarding.

SDN Controller. It is an entity that is responsible for receiving the commands of the application plane, translate them to the device in the data plane with the Southbound interface, communicate information of the data plane to the application plane, and lastly offer a logical view of the network topology to the apps. An example of this type of instance is the ONOS controller, which is an open-source controller that provides high availability, performance, and scalability (BERDE et al., 2014).

SDN Datapath. It involves the network component residing in the data plane, where the device has control over the forwarding and data processing capabilities. A layer 2 switch can be an example of the network element running in the data plane. Besides, the datapath also has a CDPI agent that will be discussed next.

CDPI Agent. It consists of an interface that makes the connection and exchange information from the SDN Controller and the network element, providing control over the forwarding functions, event notification, and other functionalities. The gRPC P4Runtime Server is an example of a CDPI agent that gets requests using the open standard P4Runtime and performs the specific operation according to the device running.

Management and Admin. Has certain static tasks that are better performed outside the application, control, and data plane. Certain examples may include business management between client and provider, physical equipment setup, or other desired configuration of the physical entity.

To better understand some of the concepts that are yet to be discussed and where they belong in the SDN architecture, we are characterizing them in the next sections divided by control, data plane, or the southbound interface. Also, some concepts may involve more than one plane. In that case, it will be mentioned later and involve each part.

2.2 SDN Programmable Data Planes

Several topics can be discussed when involving data planes, yet the most important to know regarding this dissertation include Programmable Data Planes, the P4 language, Virtualization, and others.

2.2.1 Programmable Data Planes

The ability to achieve programmability of the network has been explored with Software Defined Networks allowing the network operator to define the network's behavior easily. The first example of SDN programmability was the standard OpenFlow, which intends to propose a management interface for control plane programmability as cites (HANCOCK; MERWE, 2016).

The ability to write applications that can use OpenFlow to interact with the device fostered innovation in networking, enabling practitioners to disaggregate the network intelligence from the underlying hardware. However, the data plane was fixed since each OpenFlow specification supported only a small set of header protocols (BOSSHART et al., 2014), with novel protocols requiring updates to the OpenFlow specification. This severe limitation of Openflow paved the way for network programmability and Programmable Data Planes (PDPs).

With PDPs, network operators may define an arbitrary set of packet fields to be matched, and arbitrary actions to be performed in response to a packet match. The possibility of programming the data plane required Domain Specific Languages (DSLs) one could use to define the packet parsing and processing semantics of programmable switches. One successful example of such language is P4 (BOSSHART et al., 2014).

2.2.2 P4 Language

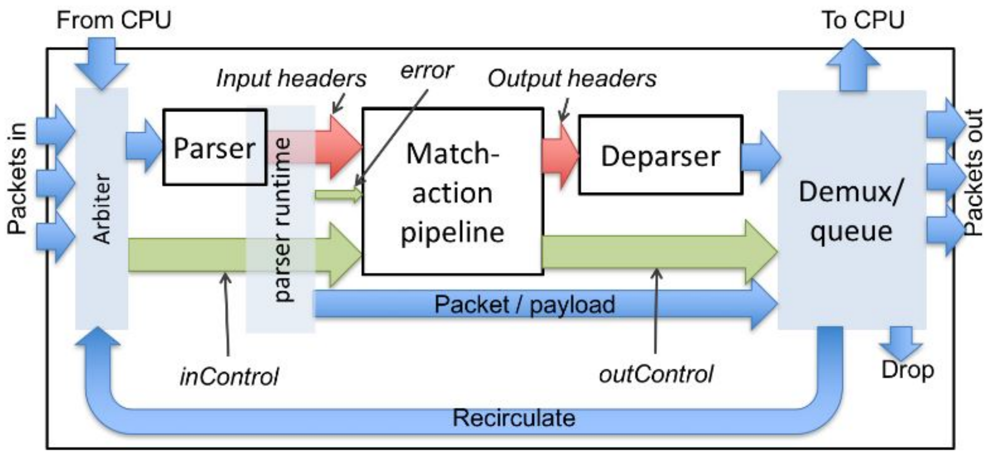
Programming Protocol Independent Packet Processors (P4) consists of a high-level language that provides a model for defining the programming of the data plane of the network devices. According to (BOSSHART et al., 2014), the major goals of P4 consists of: (i) Reconfigurability, (ii) Protocol Independence, and (iii) Target Independence. The reconfigurability defines that the controller has the ability to define how packet parsing and processing goes. The Protocol Independence consists of guaranteeing that the switch is not tied to any specific packet format, and that the controller can extract specific header fields and which match+action tables should be used. At last, Target Independence ensures that switch unrelated information is not needed.

To define how a switch must behave, one may use the P4 language to program the various stages of a switch pipeline, as defined in the PISA (Protocol Independent Switch Architecture) (BOSSHART et al., 2014), from the moment the packet arrives in the switch and is parsed, to the triggering of a certain action in a forwarding table, until the packet is deparsed and sent back to the network. Therefore, when constructing a P4 code to define the switch behavior, these set of elements are used in a program:

- **Headers.** Contain certain names and widths that identify the network protocols that the program intends to work on.
- **Metadata.** Has information generated by an execution of a P4 program, containing packet-specific state.
- **Parsers.** Can extract information from the packet, identifying them, and determining the sequence of extractions within the packet.
- **Match+Action Tables.** Define a key that the packet may match with a value, allowing to perform several actions due to the response of the match.
- **Control Flow Logic.** Contains the series of table executions, describing the packet processing when the packet arrives at the switch in the Ingress control block, and when it leaves the switch in the Egress control block.
- **Deparser.** Which reconstructs the packet again, mentioning the headers.

Figure 2.2 shows an example of a very simple switch that uses the described P4

Figure 2.2: The Very Simple Switch (VSS) Architecture



Source: (CONSORTIUM, 2017)

elements to define its behavior. The packet flow that follows when it arrives in the switch, goes through the arbiter that receives packets from the control plane, CPU, or from the recirculate method. Then, it sends the packet forward to the parser block where packet information is extracted and the match-action pipeline is called along with the metadata information, where if there is a case of a match in the table, a certain action is performed. Finally, the deparser constructs the packet and sends to the queue to be sent to the control plane, CPU, or drop. The recirculate method can occur in the case of a virtual network.

2.3 SDN Southbound Interface

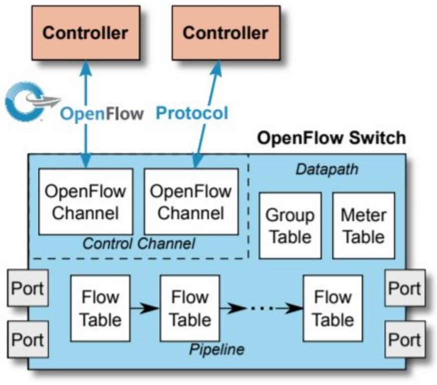
Regarding the communication between the control and data plane, the southbound interface makes itself present. The two main topics discussed in this proposal, the P4Runtime, and the OpenFlow standard. For this work, our focus is on the P4Runtime.

2.3.1 OpenFlow

The OpenFlow standard consists of a communication protocol used for controlling the network devices through an SDN controller that accepts the use of the OpenFlow protocol as the Southbound interface. With OpenFlow, there is the possibility to manipulate the flow entries available at the OpenFlow switch. With this ability to program the route of the incoming packets and how to be processed, network operators can experiment deploying new security models, routing protocols, alternatives to the IP protocol, and other functionalities as is described in (MCKEOWN et al., 2008).

With the use of the OpenFlow protocol, there is the possibility to add, update or remove the flow entries contained in the OpenFlow switch flow table, by the possibility of a reactive mode, where it happens due to the response of packets, or a proactive mode (ONF, 2015). The switch's table flow may contain table entries that perform some sort of action when matched. Considering the OpenFlow communication protocol, which is protected by TLS encryption, for the packet to arrive at the OpenFlow switch, there is the necessity of the OpenFlow channel that consists of the interface that makes the connection between both the controller and the switch. Then, packets can be sent from the controller to the switch, which the match-action tables are going to try to perform a match, and then an instruction for that packet, for example, if it matches port 22 drops the packet.

Figure 2.3: Main components of an OpenFlow switch



Source: (ONF, 2015)

2.3.2 P4Runtime

The P4Runtime consists of a control plane specification to manage the data plane elements defined by a P4 program. The goal is to be a runtime control standard for controlling P4 built-in objects (tables and value sets) and externs (counters, registers), where it is Program Independent (PI), therefore it does not matter which P4 program instance is running, and allows for local and remote control plane applications, making it scalable.

The P4Runtime standard is implemented with a program defined as P4Runtime Server (GROUP, 2019) that uses google Remote Procedure Calls (gRPC) of a service where client-server communication can be performed with remote procedure calls, which will be explained further. The P4Runtime server listens for connections at a certain port in the host, and when a client makes the connection, the server performs several message requests with the P4Runtime API for various operations ranging from setting up the connection with the data plane device to writing a table entry in the switch.

These message requests defined in the P4Runtime standard are defined as several remote procedure calls. The main RPCs used in the P4Runtime API were described in (GROUP, 2019) and they consist of:

- **Write RPC.** Proposes to update one or more P4 entities on the device. The client sends a write request to the P4Runtime server defining the device ID related to the device, The field of election ID is defined due to the requirement of knowing if the controller operating is master for the device. Then, the field of type in update dictates whether the controller intends to perform an insertion, removal, or modification in the switch table entries. In the update field, the request writes the fields to do match and actions inside the table entry. In response, the server does not need to provide information.
- **Read RPC.** Intends to retrieve information of one or more P4 entities. The difference between the request message from the write request is that it does not require election ID since the operation does not mutate any state of the switch. Also, the message requests use entities to define which fields are going to be read, for example, which table entry in the switch. The read response returns the entity information containing in the device.
- **SetForwardingPipelineConfig RPC.** Consists of defining a new P4 configuration pipeline for the device. The configuration field inside the message requests contains the P4Info of the pipeline, in other words, has specific state information of the pipeline (tables, actions, counters). Also, certain operations can be deployed with this RPC, for example, only verifying that the device can instantiate the pipeline, or verifying and commit that saves the pipeline information. The SetForwardingPipelineConfig does not return information to the control plane.
- **GetForwardingPipelineConfig RPC.** Proposes to return the P4 pipeline configuration. The message defines the device ID to get the pipeline's information, along with a response type field to make the data plane populate the correct fields in the response.
- **StreamChannel RPC.** Consists of a bi-directional stream, where both client and server perform message requests to define arbitration updates to define master con-

trollers or to perform Packet I/O, which a SDN application may perform a packet-out to the virtual switch, and a virtual switch may do a packet-in request to the SDN controller.

The P4Info, briefly mentioned before, plays an important role in the P4Runtime. This file provides information about the device's P4 entities, which is accessed by the control plane to obtain the pipeline configuration deployed in the switch when a GetForwardingPipelineConfig is called, or the control plane pushes the P4Info through the SetForwardingPipelineConfig. This P4Info file contains a structure of P4info objects, as cites (GROUP, 2019): (i) Tables, that specify all match+action tables information sent to the control plane, where it has device ID, name, aliases, match fields information, and reference actions associated; (ii) Actions, that determine all actions possible of all Match-Action forwarding tables, containing device IDs, names, aliases, and the runtime parameters to act (action IDs, bitwidths). (iii) Counters, where is divided between direct counters (are associated with a match+action table) or indexed counters (a determined number of counter values that may change); (iv) Controller Packet Metadata, that has data concerning packet I/Os, for example, the ingress port used in the switch. And, (v) Registers, where data can be read or written in the stateful memory. There are other objects in the P4Info, yet these shown are the most important for this dissertation.

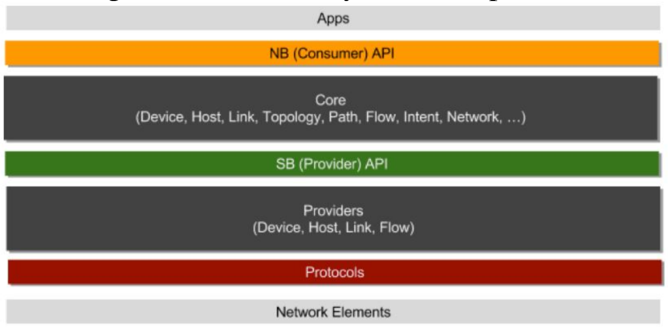
2.4 SDN Control Plane

The control plane plays a vital role in the SDN architecture, with several relevant applications and topics. Regarding this proposal and the control plane topic, we focus on discussing the Open Network Operating System (ONOS), which consists of an SDN controller example that can control P4 data plane devices with the P4Runtime standard. Besides, we mention the P4Runtime controllers that use gRPC clients to connect directly with the data plane through P4Runtime.

The ONOS consists of high performance and scalable SDN controller as cites (BERDE et al., 2014), where the network operator may build applications to define how the network behaves. ONOS already has several applications ready to use when the system is installed, approaching since routing systems to monitoring. The ONOS controller is important for this proposal due to having the ability to control P4-based switches at runtime using the standard protocol P4Runtime, and for being an open-source software.

An SDN controller, for instance, ONOS plays a vital role when deployed in an SDN network, considering that the SDN applications perform operations that are translated by ONOS into the specific functions available in the Core of ONOS as shown in fig 2.4. After ONOS performs a function originated in the SDN App, the controller must translate the operation into the Southbound API to reach the data plane, using, for example, P4Runtime or OpenFlow. According to (WIKI, 2016), there are several components available at the core in ONOS, some of them important for this proposal are:

Figure 2.4: ONOS System Components


Apps
NB (Consumer) API
Core
(Device, Host, Link, Topology, Path, Flow, Intent, Network, ...)
SB (Provider) API
Providers
(Device, Host, Link, Flow)
Protocols
Network Elements

Source: (WIKI, 2016)

- **Device Subsystem.** Contains and manage the infrastructure of all devices.
- **Link Subsystem.** Possess and manage the links available at the infrastructure.
- **Host Subsystem.** Has information about hosts and their respective location.
- **FlowRule Subsystem.** Manages the match-action flow rules installed on the devices and important flow metrics.
- **Packet Subsystem.** Allow applications to receive Packet-In from switches and perform Packet-Out into the network.

For ONOS to connect with the P4-based switches in the data plane, it must use the standard API P4Runtime to connect with the P4Runtime server mentioned in Section 2.3. The ONOS controller utilizes certain drivers and protocols for translating the actions performed by SDN apps into P4Runtime message requests, therefore allowing support to manage the P4-based devices. Considering an example of SDN apps which intend to install a flow rule write into a switch, ONOS receives the information and maps into the FlowRule Subsystem. Then, the drivers map the flow rule and transform into a WriteRequest message request, which is a RPC used by the P4Runtime. At last, when ONOS is connected with the P4Runtime server, the write request is sent to the data plane.

Another type of controller may consist of applications developed to interact directly by managing the P4Runtime message requests. These controllers consist of gRPC clients that create the message requests necessary to perform the same operation as if it were in ONOS, for instance, the P4Runtime controller directly creates the WriteRequest RPC instead of creating a flow rule.

gRPC consists of a high-performance framework to attend remote procedure calls. gRPC allows the programmer to define a service, specifying the methods that will be used between the client and server, with support to multiple programming languages. The gRPC server listens for connections in a TCP port, allowing the use of secure or insecure connections, and then the client starts the handshake by sending the message requests. There are certain types of RPCs that the user may define: (i) Unary RPC, which consists of a single request and response from client and server respectively; (ii) server streaming, that the client sends a single request, and the server uses a stream that may send multiple responses; (iii) client streaming, that the client sends a stream of message requests, yet the server sends a single response; at last (iv) bi-directional streaming, that both client and server initiates stream message requests. To define how the message requests of the service needed for gRPC can capture the data, the serialization message method called Protobuf is necessary. The protocol buffers get data in the service file defined by the programmer and structures it as messages where each one contains a small logical record of information containing several fields (GRPC, 2020).

Considering the P4Runtime standard RPCs, the P4Runtime controller connects with the P4Runtime server by using the StreamChannel RPC, sending master arbitration updates and other requests. After the connection is active, the network operator may perform other RPCs available, like GetForwardingPipelineConfig to get the pipeline in the P4 switch or write request to write a table entry into the switch.

2.5 Virtualization

Virtualization consists of the process of allowing multiple instances of a computer to run on top of a physical hardware. With virtual machines, operational systems and applications can be run on the computer at the same time, while having isolation between them. The hypervisor also allows the process to be done by making the host computer offer support to several guest virtual machines sharing the host resources, like memory or the processing capabilities.

There are several reasons for having virtualized environments. The ability to provide abstractions for the virtual instances can be beneficial in several domains, like in public and private cloud environments, where having the data centers can come with a high price, and with virtualization, several instances can be run simultaneity in one server. Therefore, several tenants can share one physical substrate using the determined slice, and manage the determined part that it belongs to without interfering with other ones.

One great advantage of virtualization has been network virtualization, wherewith Network Function Virtualization (NFV), there is the possibility to run multiple networking functions in a virtualized mode, where they can create services for communication or other several activities.

Also, there is the virtualization of programmable data planes, which is extremely relevant. With the ability to achieve virtualization in programmable data planes, several virtual data planes may be placed in a physical substrate. In this scenario, the virtual instances run on top of the physical data plane, ideally each running independently with isolation and other levels of security and management, to avoid tampering. Concerning the method that the solution may achieve PDP virtualization, it may consist of emulating the virtual instances using a general program, or composition of the several instances into one program, or other methods.

Considering this union of virtualization with programmable data planes, the network operator may use several use cases, like network slicing, in which the virtual instances are sliced along with the resources available. For example, to reduce risks with huge capital expenditure, companies may intend to hire slices of the service, like telecommunication companies that may opt to use slices of radio-base stations to work with 5G making it easier to interface the provider network with that of the telecom company.

3 RELATED WORK

Although the research work reported in this dissertation focuses on the management abstractions for virtual switches, we review in Section 3.1 the state of the art on virtualization of programmable forwarding planes in general. In summary, existing solutions on the software substrate to materialize multiple virtual switch instances, and use a variety of techniques to this end (emulation of virtual switches, merging of switch code for running as a single program, and software switches). We then cover in Section 3.2 prior investigation on information flow control policy enforcement in SDN. At last, in Section 3.3, we summarize the proposals by identifying if they perform IFC or have support to southbound interfaces.

3.1 Switch Virtualization

HyPer4 (HANCOCK; MERWE, 2016) and HyperVDP (ZHANG et al., 2019) have the goal to achieve switch virtualization through a general role P4 program to emulate the other P4 entities. Hyper4 became the first solution to explore virtualization in the data plane, where its emulation focused on the persona creating several match-action entries that emulated the configuration behavior needed for the P4 programs. This solution makes heavy use of the P4 primitive resubmit and creates several match-action tables to perform the emulation, incurring a high overhead to the solution. HyperVDP followed HyPer4 by proposing a solution based also on emulating the P4 programs with the use of a P4 model. The number of match-action tables needed was decreased, yet due to the emulation process, there is still a significant number of tables needed, and the performance is harmed.

SR-PVX (LI et al., 2017) follows a similar approach to Hyper4, but focuses on Protocol Oblivious Forwarding (POF) (SONG, 2013). SR-PVX enables instantiating virtual POF switches over designated substrate switches, and uses source routing to realize virtual links in virtual SDN (vSDN) slicing. The deployment flow of virtual switches involves deployment of tenant-defined flow-table(s) in the substrate switch and configuration of match tables so that packets belonging to a specific vSDN are processed by the correct tenant. In a follow-up work PVFlow (LI et al., 2018), the authors unify flow-tables that contain arbitrary matching fields with various lengths, enabling them to share the TCAM of a matching stage in a substrate switch.

The switch composition method is a virtualization strategy that relies on merging of the codebase of switches to form a single switch program, that aggregates the functionalities of the original switches. Then, with the composed programs several functions can be performed. One solution that performs this type of action is P4Visor (ZHENG; BENSON; HU, 2018). Unlike HyPer4 and HyperVDP, P4Visor does provide a feasible performance and memory footprint. However, P4Visor does not provide support to hot-swap the multiple virtual switch instances without stopping the operation and lacks in the isolation between them.

There are also solutions for running virtual switches based on software substrates like PISCES (SHAHBAZ et al., 2016). The authors of PISCES defend the idea of software switches should be able to change how they must forward packets through a DSL language, where it would be possible to specify how the packets may be processed (SHAHBAZ et al., 2016). PISCES is derived from OpenVSwitch (PFAFF et al., 2015), a hard-wired hypervisor that does not allow protocol independence. Therefore, to achieve protocol specification independence, PISCES intends to support custom protocol specification using a DSL. However, if one may run in parallel multiple software switches, there is a vulnerability of not having a controlled isolation mechanism between the virtual instances to avoid unwanted flows.

Krude et al. (KRUDE et al., 2019) discussed directions towards *Programmable Switches as a Service*, including switch hot-pluggability. These directions have been partially addressed in our prior work P4VBox (SAQUETTI et al., 2020), which supports multiple switch instances on a NetFPGA board. In this paper, we advance our prior work by showing how we realize the architectural requirements discussed in (KRUDE et al., 2019) to deliver a hypervisor for programmable virtual switches.

With regard to hardware switches available on the market and their support to virtualization, the Arista 7170 switch series, based on the Barefoot Tofino chip, enable network managers to load up to four distinct “forwarding profiles”. A forwarding profile can be regarded as a distinct switch program running in a hypervisor. The network manager may write each profile in P4, and load into the switch using the switch package manager. According to the white paper, toggling between profiles is almost hitless. However, the system is limited to four profiles and, therefore, four “switch programs” that can run in parallel. The white paper also mentions that each profile a predefined number of match-action stages, potentially leading to under-optimal resource usage if a profile uses fewer stages.

3.2 Information Flow Control

Enforcing information flow control can help prevent applications to use malicious flows or unwanted ones, which may be performed by cyber attacks. Cross-App Poisoning has been recently identified as a critical class of control plane integrity attacks in Software Defined Networking (SDN) (DACIER et al., 2017). Ujcich et al. (UJCICH et al., 2018) described in detail the *modus operandis* of the attack, which involves installing a compromised app (which could be downloaded from a public SDN app repository or marketplace (Aruba Networks, 2020; Open Network Operating System, 2020)) on the SDN controller. The malicious app's goal is to make legitimate apps perform bogus actions, through shared objects, in the control and data plane that the malicious app itself cannot perform due to insufficient privileges.

Considering Cross-App Poisoning attacks in Software Defined Networks, the authors of (UJCICH et al., 2018) also studied the CAP model for Security-Mode ONOS (YOON et al., 2017) and demonstrated that Role-Based Access Control (RBAC) solutions alone are insufficient to prevent such attacks, as they do not track information flow or enforce information flow control (IFC) through data provenance.

Data provenance consists of records that describe entities and actions from where the data came from, who influenced it as cites (W3C, 2010). Then, the data provenance's goal is to check the history of data (which entity created, which process modify or accessed it) and then can be used from validating the next entity that the data goes to forensics analysis for checking possible violations. Consider an example of data centers, as described in (BATES et al., 2014), an administrator may use network data provenance to detect if a suspicious routing table entry was the result of a routing attack by a malicious entity or that it may just be a simple misconfiguration.

Information flow control uses data provenance to enforce a security design by defining a set of policies that the flow must obey. Considering that there is a policy to block a flow from least privileged applications to higher ones, and there is such a scenario the information flow control system would block the malicious flow. To define the integrity level of the entities involved, there are concepts of tags that consist of metadata that may be associated with the entity's data, and there is the concept of labels which consists of a subset of the integrity tags. Therefore, if the application example has the tags and labels mapped, there is the possibility of defining the least privileged application and the higher one by counting the number of labels that it possesses, and then defining if the flow is malicious or benign by checking with the stored policies. Considering a scenario example that the IFC policy is to block every flow from least privileged application to a higher one, and there are two SDN applications exchanging packets. Then, if the integrity label of one of these applications is lower than the other, the flow gets detected and stopped by the IFC solution due to the violation of the policy.

Therefore, UJCICH et al. proposed ProvSDN, a solution that uses data provenance (W3C, 2010) to track information flow and intercept app requests that violate predefined IFC policies. ProvSDN gets the API requests, tracks if the operation is made to write or read in the data store of the controller, and then stores the information in a provenance graph that contains all the data about the control plane history to avoid possible violations according to the policies stored.

However, ProvSDN assumes that switches are *standalone* entities, and with recent advances in lightweight programmable forwarding planes virtualization solutions (HANCOCK; MERWE, 2016; ZHANG et al., 2019; ZHENG; BENSON; HU, 2018; SAQUETTI et al., 2020) ProvSDN cannot track information flow control among the SDN applications and virtual switches. Hence, an SDN app may poison a virtual switch with a CAP attack without being detected by ProvSDN.

The literature presents manly the ProvSDN (UJCICH et al., 2018) as the first solution to discuss and deal with Cross-App Poisoning (CAP) attacks in the SDN network using data provenance. Other related state of the art solutions intended to focus on data provenance, like PROVDETECTOR (WANG et al., 2020) that uses the provenance graph with the goal of detecting stealthy malware, or EVENTSCOPE (UJCICH et al., 2020) which creates flow graphs that record events and control flows to detect vulnerabilities in the control plane. None of the lightweight solutions have tried to enforce IFC.

3.3 Summary

The solutions described in the entire Chapter 3 may either be available in the data plane or control plane, each with its own method and functionalities. Having solutions focused on providing virtualization in the data plane, other providing information flow control in the control plane, and other proposals.

In Subsection 3.1, the solutions discussed intended to perform virtualization of the PDP through emulation, composition, or other methods. However, despite some specific issues like the high overhead that they may cause to the virtualization solution, these proposals do not offer support to an abstraction that the virtual switches can be managed independently, therefore if the network operator has access to a virtual switch, he can affect the other instances. In Subsection 3.2, solutions that use IFC policy enforcement are described to protect the SDN network against a specific poisoning attack. The solution does contain a significant contribution to mitigate this type of cyber attack, yet it does have a vulnerability when there are scenarios of PDP virtualization, in other words, since ProvSDN (UJCICH et al., 2018) sees switches as standalone entities, it cannot track flows that may happen between virtual switches.

With our PvS control engine abstraction, we address the issue that the PDP virtualization solutions cannot by allowing the network operator to deploy and manage the virtual instances independently and securely while having the support to open management interfaces, and having a marginal management overhead to the virtualization solution deployed. Also, with our vIFC abstraction, we address the issue that ProvSDN (UJCICH et al., 2018) does not, which is tracking and mitigating the CAP attack that uses virtual switches as a proxy, by using IFC to track the malicious flow and mitigate the attack regardless of the virtualization solution deployed.

Table 3.1: Existing work on virtualization, standards and IFC.

Solution	Plane location	Virtualization	Model	IFC	Southbound support
Hyper4 (HANCOCK; MERWE, 2016)	Data Plane	Yes	Emulation	No	-
HyperVDP (ZHANG et al., 2019)	Data Plane	Yes	Emulation	No	-
P4Visor (ZHENG; BENSON; HU, 2018)	Data Plane	Yes	Emulation	No	-
P4VBox (SAQUETTI et al., 2020)	Data Plane	Yes	Emulation	No	-
ProvSDN (UJCICH et al., 2018)	Control Plane	No	-	Yes	OpenFlow
PvS	Data Plane	Yes	Virtualization	Yes	P4Runtime

Table 3.1 provides a summary of the most important existing work on virtualization of forwarding elements. Observe that existing work either emulate switch pipelines through a set of match-action stages, compose switch functions into a single program, or lack abstractions for virtual switch control, in addition to lacking isolation guarantees and imposing severe performance penalties. Also, prior work has not focused on the potential vulnerability that a shared virtualization environment may bring to switches in case a malicious control plane app attempts to tamper with the operation of switches.

4 PVS VIRTUAL SWITCH MANAGEMENT ABSTRACTIONS

In this dissertation, we focus on the control management abstractions required for operating virtual switches by devising a solution for switch management that is compatible with open SDN interfaces and enables secure virtual switch instances. Then, we approach defenses required for preventing information flow violations between virtual switches. The actual virtualization of programmable switches is left out of this dissertation's scope, as it has been approached in prior investigations like by TIRONE.

In this chapter, we first provide an overview of the PvS, showing its several components, design principles involved, and abstractions for switch management. The overview of PvS is shown along with its several components, allowing the virtualization of programmable virtual switches in shared network contexts with secure management of tenants with the virtual instances. In the overview, we organize the discussion of PvS considering its forwarding engine (which is materialized by prior work on virtualization of programmable switches) and control engine (focus of this dissertation). We cover the PvS control engine's discussion with each part's main functionalities while discussing their design principles. Therefore, the outline of the chapter starts in Section 4.1, which we provide an overview of PvS. In Section 4.2, we discuss the PvS main design principles and their relations with the specific PvS components. We discuss the virtual switch management abstractions provided by PvS in Section 4.3. Finally, we discuss in Section 4.4 the PvS Control Engine with its components, requirements, and operations.

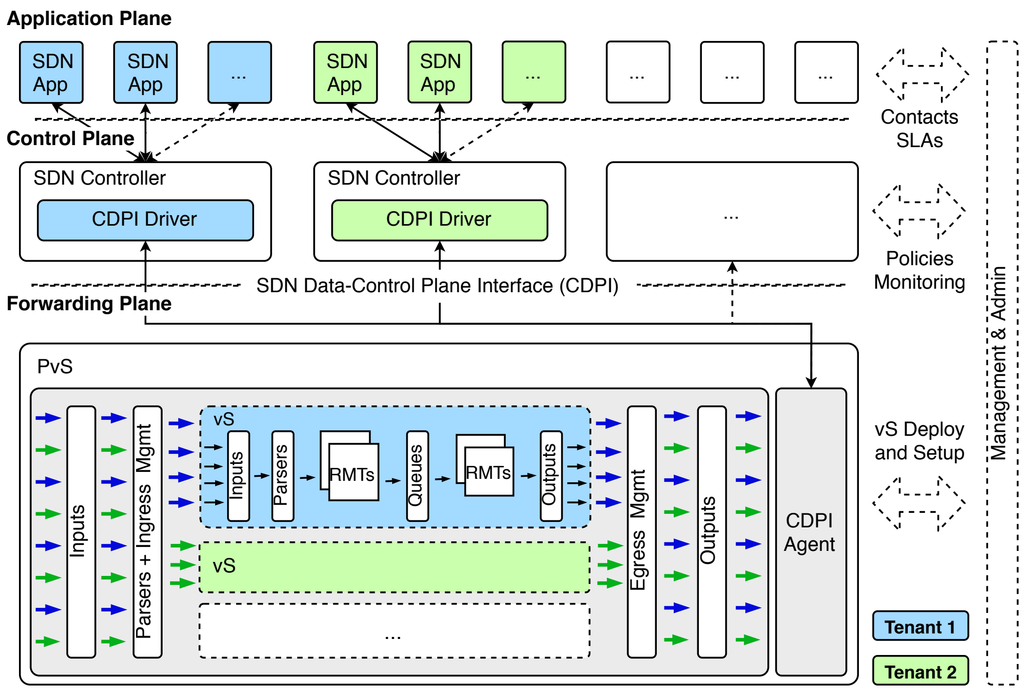
4.1 PvS Overview

The PvS design follows the idea that network operators may manage proper abstractions of the programmable virtual devices securely in the data plane provided by hypervisors. Therefore, considering a hardware substrate like a NetFPGA board having the necessary abstractions with the connection to an SDN controller, several tenants may manage the virtual switch instances with the correct access control to the entities and enforce information flow control.

Considering a physical switch, like the NetFGPA board, there are certain aspects related to the (i) available memory for storing the switches and their information, (ii) the number of network ports, (iii) throughput capability, (iv) latency overhead, (v) and the available management interfaces that it can use. The virtual abstraction follows the pattern by using some of the features like the management interfaces needed, the memory capacity, or the virtual switch ports available.

To provide the virtual abstractions running in parallel on top of a hardware substrate with consistent management and security levels, the PvS design has several components that provide the whole PvS solution. Figure 4.1 exhibits the SDN architecture along with the PvS. The PvS is divided between the *Forwarding Engine* and the *Control Engine*. The forwarding engine is the module responsible for accommodating the virtual switch instances. The control engine contains functions of the CDPI agent, allowing tenants to interact securely with the virtual switches using standard southbound interfaces, like P4Runtime or OpenFlow.

Figure 4.1: An overview of a conceptual architecture of a programmable virtual switch hypervisor, and its relationship with key elements that compose a high-level Software Defined Networking (SDN) architecture.



Source: From the Author

The forwarding engine enables supporting the multiple virtual switch instances running at the hardware substrate, with defined information of the storage needed for the match-action table entries, counters, and packet processing capacities. Furthermore, the forwarding engine is responsible for guaranteeing that the incoming packet is forwarded to the right switch virtual instance, and the output packet goes to a certain output port. These modules also shall provide the correct information (virtual ports, ingress/egress buffers) to the placeholder, hence the programmable virtual switch instance can be deployed as if it were in a physical switch.

A proper abstraction of a virtual switch control engine is necessary to support data plane management channels and manage the virtual switches' security. Then, we present the control engine abstraction, which intercepts message requests originated from the SDN apps sent from the control plane, and guarantees the message request's operation only if the authentication is made with the correct credentials sent from the controller. The control engine should secure tenants' operations within the scope of its control, like write requests to the certain match-action table and with the correct permissions to do so. If a malicious tenant performs certain operations, the control engine must also check information flow control to block CAP attacks that may happen, and with the hypervisor guarantee that it is not reprogrammed by the malicious action taken by the tenant, which may gain access to privileged memory blocks.

There are certain management aspects that are in our PvS control engine abstraction, like the possibility to possibility for a network operator to define how the switch behaves, or manage the switch's match-action tables to add, modify, or delete the table entries to control the data plane switches. Also, our abstraction is agnostic of the virtualization solution, resources, and protocols associated, for instance, our solution is not tied to specific languages like P4, it can be deployed in any other protocols or virtualization solutions.

4.2 PvS Design Principles

The entire PvS solution has several design principles that may belong to the forwarding engine or the control. Therefore, Table 4.1 divides the design principles according to the specific module.

Table 4.1: Design principles relation with PvS modules.

Design Principle	Engine	Related to
Switch Bytecode for any DSL	Forwarding Engine	PvS Forward Engine
Support Same Switch Bytecode	Forwarding Engine	PvS Forward Engine
Allocation of Physical Resources to vSs	Forwarding Engine	PvS Forwarding Engine
vS Performance Equal to Standalone Switch	Forwarding Engine	PvS Forwarding Engine
Virtual Switch Hot-swapping	Forwarding Engine	PvS Forward Engine
Virtual Switch Authentication	Control Engine	CDPI Agent
Single Management Channel for Virtual Switches	Control Engine	CDPI Agent
Virtual Switch IFC Policy Enforcement	Control Engine	vIFC
Multi Controller IFC Enforcement	Control Engine	vIFC
Virtual Networking	Forward and Control Engine	PvS Forward Engine and CDPI Agent

Considering the PvS Forwarding Plane, certain design principles that allow supporting hot-swapping of virtual instances appear, from guaranteeing the switch bytecode's intellectual property to allocating the physical resources needed for allowing several switches to be deployed in the NetFPGA. The design principles related to the forwarding engine consist of (i) supporting switch bytecodes whether it is written using P4 or other DSL, like POF; (ii) allow the same switch bytecode designed for a certain target that is running on, like a NetFPGA bytecode should be used for a hypervisor running on a NetFPGA; (iii) efficiently allocate the physical resources needed for most virtual switches to be deployed without affecting the performance; and (iv) the tenant should be able to add or remove virtual switch instances without interrupting other switches.

The control engine has design principles to allow the consistent management and security level of the virtual switches, having them either from managing virtual switches through a single management channel to having information flow control policies to define if a flow might be a malicious attempt from an SDN app to gain privilege actions. Therefore, the control engine design principles defined for PvS are:

- **Virtual Switch Authentication.** This design principle indicates that SDN applications running in the application plane must send valid credentials to perform actions in the virtual switch. Without this principle, malicious applications can easily tamper with or even disrupt virtual switches' entire operation.
- **Simple Management Channel for Virtual Switches.** Differing from normal communication channels using P4Runtime that create multiple management channels for the data plane, PvS defends the principle of having a single management channel for multiple virtual switches, since allows more efficiency in distributing security policies and permissions for variables in the system, like users, and applications.

4.3 PvS Virtual Switch Abstractions

The virtual switch characterization was briefly mentioned in Section 4.1 where it was discussed the need for open management interfaces, and the virtual ports that the virtual switch may use. All of them consist of aspects that the hypervisor must support for allowing multiple virtual switches to be deployed in an SDN-enabled network. Given the design principles, we correlate them with the abstractions characterizations to demonstrate that PvS supports the requirements.

Open management interfaces. consist that the hypervisor must support open standard management interfaces to the virtual switch instances, therefore allowing them to be used in an SDN network while guaranteeing security against unprivileged actions from a malicious tenant that may intend to affect certain virtual instances. Also, the hypervisor must ensure that the malevolent tenant may not succeed in performing privilege escalation operations using the management interface. Accordingly, PvS uses the CDPI agent for making the abstraction needed for the management channel of the virtual switches with the tenants. Message requests are received from tenants using SDN applications, identifies the virtual switch associated, and perform certain actions if the correct permissions are defined. Also, through this shared management abstraction interface PvS can detect CAP attacks with information flow control violations that a malicious tenant may perform.

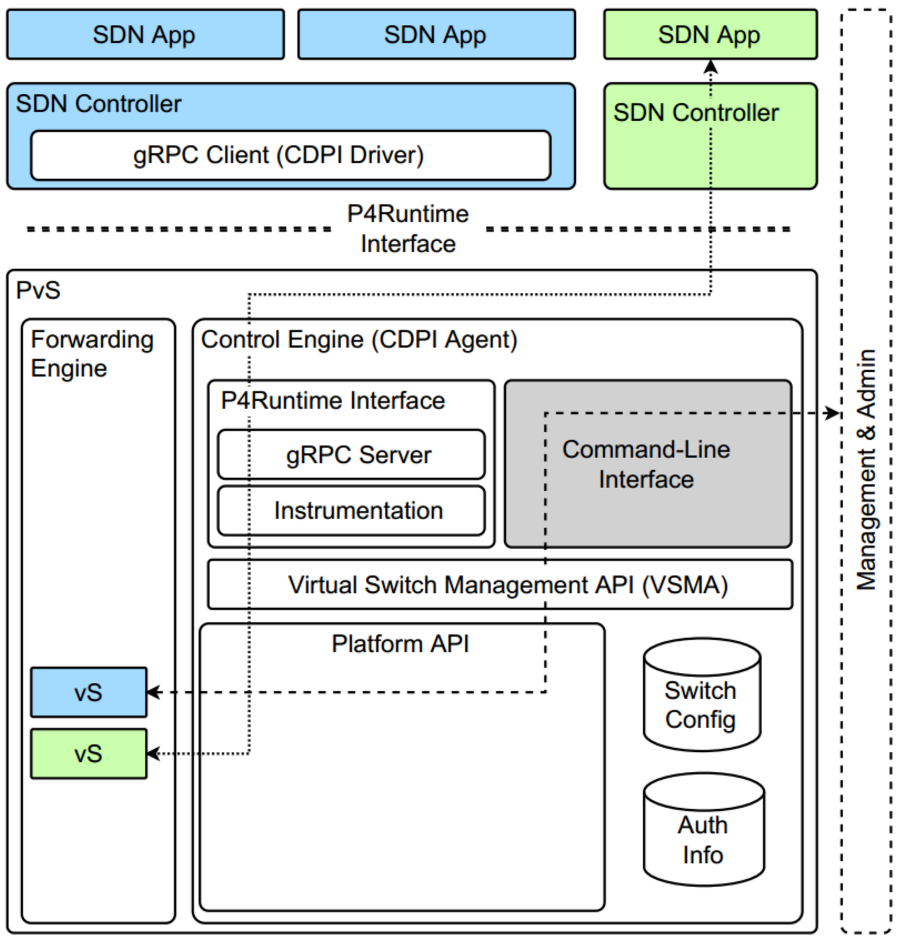
Virtual Ports. are important for correlating with the virtual switch instances, that have a physical port associated and may map an arbitrary number of virtual ports. When a packet arrives in PvS, it is delivered to the correct virtual port, where PvS identifies which tenant sent the packet and to which virtual switch. Then, when the packet is processed, the PvS egress module determines the output port necessary so that the packet is copied to, ensuring that it is the correct tenant that the packet is going to.

4.4 Control Engine Implementation

To build the PvS Control Engine, we focused on the following specific research questions: (i) what hypervisor management design may enable tenants to operate their virtual switch instances transparently, using known standards like P4Runtime? And (ii) how to share the hypervisor interface among multiple tenants while providing switch independent and secure management abstractions? We note that an effective answer to these questions must simultaneously satisfy two requirements. A bottom-up requirement is that the hypervisor interface must expose independent abstractions of virtual programmable switches, agnostic of their internal logic and available resources. The interface must support, for example, virtual switches having a diverse number of virtual ports (which may change on the fly), distinct pipelines of match-action stages, etc. A top-down requirement is that the hypervisor interface must ensure access control per virtual switch (e.g., through authentication), and provide access policy management per virtual switch resource (e.g., enable access to the tables of a virtual switch, but not to its counters).

Addressing the requirements, the PvS Control Engine aggregates the functionalities of the SDN Control-Data-Plane Interface (CDPI) Agent, being responsible for interfacing the virtual switches running in the Forwarding Engine with the SDN Control and Application planes. It does so by providing a software interface that enables network

Figure 4.2: PvS Control Engine Architecture



Source: From the Author

administrators and operators to manipulate a set of components available in the virtual switch, like counters, or match-action tables.

Fig. 4.2 presents an overview of the Control Engine. It currently supports the P4Runtime and Command-Line as management interfaces, both equivalent in terms of functionalities exposed. In this Figure, green and blue boxes illustrate the virtual switch and respective drivers belonging to the same tenant. The P4Runtime interface enables the communication between network apps running in the application plane and virtual switches implemented on PvS. The command-line interface can be used by a privileged network admin to directly deploy/manage virtual switches in the forwarding engine, and configure the mapping between virtual and/or physical switch ports.

The PvS P4Runtime interface follows the P4Runtime reference architecture (CONSORTIUM, 2017). It contains two sub-modules: a gRPC Server, which provides an entry point for SDN controllers that support P4Runtime to connect and exchange RPC message requests with the PvS P4Runtime server, and an instrumentation module, responsible for translating message requests and switch events down and up the architecture.

The Virtual Switch Management API (VSMA) provides the functionalities used by the PvS P4Runtime and Command-Line interfaces. More specifically, it handles processes like deploying virtual switches, configuring traffic steering from/to virtual switch ports, creating users and establishing virtual switch access control, reading and writing match-action table entries, etc. To this end, it interfaces the message request with the Platform API, along with switch configuration and authentication control information stored in the `SwitchConfig` and `AuthInfo` databases.

The last module before the message request arrives in the virtual switch is the Platform API, which receives the message request from the VSMA (originally from the SDN applications), and interfaces with the virtual switch instances to perform the message operation into the virtual switch. To make the connection with the virtual switches successful, device management methods are executed, for instance, a method to write a table entry into the switch. In the control engine implementation process, we addressed the following aspects: (i) how to handle message requests to virtual switches? (ii) how to extend P4Runtime for supporting virtual switch access control? and (iii) how to transfer packets between virtual switches and controllers. Next, we discuss those aspects in detail.

4.4.1 P4Runtime Support for Virtual Switches.

The PvS control engine provides a single entry point – through a single running P4Runtime server instance – for apps running on different SDN controllers to manage any PvS virtual switches. The instrumentation module receives the message requests from the P4Runtime server and calls methods in the Virtual Switch Management API (VSMA) to process them. Our PvS P4Runtime interface currently supports the following remote procedure calls: `ReadRequest`, `WriteRequest`, `SetForwardingPipeline-Config`, `GetForwardingPipelineConfig`, and `StreamChannel`.

To determine the target switch of a given message request, we use a numeric identifier and rely on the `device_id` field in P4Runtime protobuf message request. Virtual switch identifiers are (i) assigned by PvS upon deployment of virtual switches, (ii) stored in the `SwitchConfig` data base, and (iii) returned to the network administrator. These switch ids must then be used by apps running in the Application Plane to determine to which switch a given message request refers to.

Handling P4Runtime `Read` and `Write` message requests also require detailed information about the virtual switch pipeline – more specifically, its match-action tables (their fields and actions) and respective numeric identifiers. To this end, during the virtual switch deployment process, we gather information about the virtual switch in the `SwitchConfig` data base.

After receiving a `Read` or `Write` message request, the instrumentation module queries the `SwitchConfig` data base (through the VSMA module) to translate the numeric identifiers in the message into the switch, table, field, and action names. The instrumentation module then determines which method from the VSMA must be called to handle the request. The VSMA currently supports reading, writing, and deleting a table entry from a virtual switch.

The Platform API serves as a proxy for VSMA to handle the virtual switches, using methods provided by specific modules of a device to manage the virtual instances. Through these modules, a network operator may execute the specific methods of the target device to perform the actions supported by the VSMA.

4.4.2 Virtual Switch Access Control

The P4Runtime specification does not provision for multiple switches being configured through a single communication channel, neither provide user-based access control to managed switches. To bridge this gap, we extend the P4Runtime specification in the following manner.

The PvS P4Runtime server only allows secure connections with SSL/TLS authentication. Upon connecting to the P4Runtime server, a client enters the `CONNECTED` state. Before issuing any message requests, handling packet in and out from/to virtual switches, performing master arbitration, or receiving switch notifications, the client must first move to an `AUTHENTICATED` state. In case a request other than authentication is received while in the `CONNECTED` state, the message response is replied with a `grpc.StatusCode.UNAUTHENTICATED` code, as defined in the RPC API[1]. The client can terminate the connection at any time, following the current P4Runtime specification.

To authenticate in PvS, a client must use the `StreamChannel` to send an `update StreamMessageRequest` to the PvS Control Engine. We extend the `other` field (type `.google.protobuf.Any`) so that the client can send an "auth" message. Our `Auth` message implementation follows the proto3 syntax, and contains two needed fields:

[1]gRPC GitHub repo: <https://github.com/grpc/grpc/blob/master/doc/statuscodes.md>

`user_name` and `passwd`. The client populates it with the user credentials required to access the desired virtual switches. Upon receiving a `StreamMessageRequest` with an `Auth` message, the instrumentation module extracts the user credentials and delegates the authentication request to the VSMA. The VSMA then queries the `AuthInfo` data base for verifying supplied credentials. In case they are valid, the connection then moves to the `AUTHENTICATED` state.

In our prototype, a client can authenticate only one user per connection to the P4Runtime server. After moving to the `AUTHENTICATED` state, the VSMA creates a client context data structure, saves in it the connection state and the user info retrieved from the `AuthInfo` data base, and stores it in a connection array. We use the `context.peer()` for indexing the VSMA connection array. The saved context is destroyed upon client disconnect.

In addition to user credentials, the `AuthInfo` data base also maintains virtual switch access permissions. Prior to handling each message request, the VSMA must first verify if the authenticated user has access permissions to the virtual switch having the `device_id` informed. In case the user does not have the permissions for the requested message, the server replies with a `grpc.StatusCode.PERMISSION_DENIED` code.

Observe that the model above enables fine-grained access control. For example, a user may have specific permissions (e.g., `Read` and `Write`) to some virtual switches. A user may also belong to a user group that has specific access permission to certain switches. A switch may also be part of a switch group, to which specific permissions may be granted to a user or user group. We envisage this access control model to ensure that only authorized users have access to virtual switches belonging to a same tenant.

4.4.3 Packet I/O Between Virtual Switches and Controllers

PvS also enables handling packet-in and packet-out messages between virtual switches and controllers. For handling packet-in, i.e., sending a data plane packet to the controller for further inspection, the PvS control engine utilizes the Platform API which reads packets that come from the virtual switch, and forwards them to the VSMA. The VSMA then looks for active client connections authenticated for a user with packet I/O permissions for that virtual switch. For each connection, the VSMA sends a message through the persistent stream channel, containing a PacketIn update, following the standard P4Runtime specification. The control engine also sends a metadata structure containing the source switch device id and input port of the packet. In case there is no user authenticated for that device, the PacketIn update is discarded.

For handling packet out, i.e., sending a control/application plane packet to the data plane, the app running in the application plane sends a message request with a PacketOut update following the P4Runtime specification. The metadata must contain, at least, the device id of the target virtual switch and a virtual input port. Once received, the VSMA forwards the PacketOut update down to the Platform Interface, which copies the packet prepended with its metadata to the host network interface attached to the virtual switch.

5 DEFENDING SWITCHES FROM CAP ATTACKS WITH VIFC

Using the PvS Control Engine to manage and secure the virtual switches is necessary to guarantee virtual instances' consistent security levels. Nevertheless, the control engine alone cannot detect if management information is flowing among control plane apps and virtual switches in a way that violates predefined network security policies, making any virtual switch vulnerable to CAP attacks.

The Cross-App Poisoning (CAP) attack (UJCICH et al., 2018) consists of a control plane integrity attack in which malicious apps poisons other apps (through shared objects) to perform bogus actions, using other apps' permissions. To deal with CAP attacks, Ujcich et al. (UJCICH et al., 2018) proposed ProvSDN (discussed in Section 3.2), to mitigate this type of attack in SDN networks by enforcing the information flow control through data provenance directly in the control plane using the ONOS controller.

ProvSDN assumes a scenario in which switches are *standalone* entities, i.e., have a single pipeline of match-action stages, and are managed through a single controller. However, recent advances in lightweight programmable forwarding planes virtualization solutions (HANCOCK; MERWE, 2016; ZHENG; BENSON; HU, 2018; ZHANG et al., 2019; SAQUETTI et al., 2020) dramatically changed that scenario. In summary, a physical switch may either run a composition of pipelines of match-action stages from distinct programs (ZHENG; BENSON; HU, 2018) or emulate virtual switches through a general-purpose switch program (HANCOCK; MERWE, 2016); in both cases, each virtual switch instance may be managed independently, by various tenants, through different controllers. Under such a scenario, shown in Fig. 5.1, a single controller can no longer maintain a complete view of the information flow between apps and virtual switches. Consequently, apps that have permissions for a single virtual switch may end up controlling/poisoning every virtual switch composed/emulated in the target switch. Also, ProvSDN may not prevent CAP attacks that a malicious app performs to another one running in a different controller. We tackle these issues with vIFC, a solution for IFC policy enforcement in lightweight Programmable Data Planes (PDP) virtualization solutions.

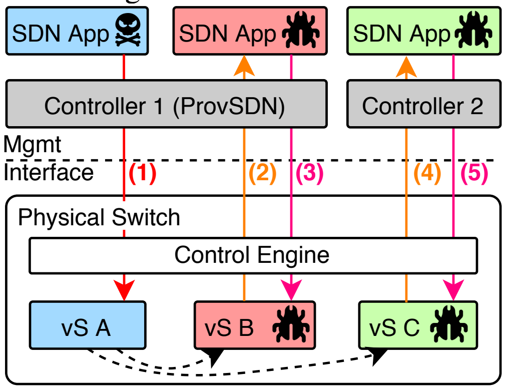
Throughout this chapter, we demonstrate in Section 5.1 the threat model used by ProvSDN and a case study of the CAP attacks that may affect the virtual switches. Then, we exhibit in Section 5.2 the vIFC discussing each component. In Section 5.3 we demonstrate the same use case shown with ProvSDN but with vIFC enforcing IFC, also demonstrating how the provenance graph works. Throughout Section 5.4, we explain the

CAP attack model for known use cases in an SDN network. In Section 5.5, we formalize the policies used in both PvS control engine abstraction and the vIFC abstraction, while discussing the labels that the SDN applications receive, and how that process works. At last, in Section 5.6, we discuss some security models that our model inherits some aspects to guarantee access control.

5.1 Threat Model and Case Study

Ujcich et al. (UJCICH et al., 2018) assume that (i) the SDN controller is trusted and secured, but it may provide services to a malicious app, (ii) the attacker controls a malicious app that has least-privileged permissions, and (iii) apps have principal identities and cannot forge actions to make them appear as made by another app. In their policy model, the authors also count switches as apps, though they only consider control plane apps as potentially malicious.

Figure 5.1: CAP Attack



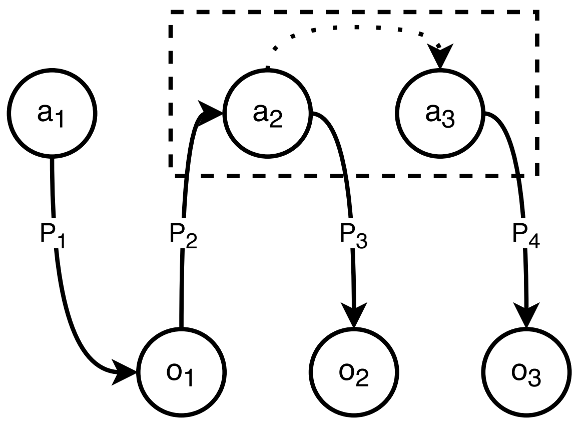
Source: From the Author

We show in Fig. 5.1 a scenario in which a malicious app performs a CAP attack against a physical switch that emulates multiple virtual switches. ProvSDN is depicted running in SDN Controller 1. First, the malicious app performs a packet-out to virtual switch (vS) A, to which the app has PACKET_⋆ permissions (flow 1 in Fig. 5.1). However, the malicious packet could be carefully crafted to be sent "out of band" to vS B (dotted arrow from vS A to vS B)[1]. The malicious packet would then trigger a packet-in event from vS B to a legitimate SDN app (2), which has both PACKET_⋆ and FLOWRULE_⋆ permissions to vS B. In response, the legitimate app installs a bogus flow rule in vS B (3). Note that, in this case, the attack is successful because ProvSDN is unable to track that a

[1]The malicious packet, although sent through a packet-in event to vS A, may reach vS B because of a logical link between vS A and B, packet (metadata) spoofing, etc.

packet-out to vS A triggered the vS B packet-in. Observe also in Fig. 5.1 that the malicious app may even launch a CAP attack against a legitimate one running in a different controller (flows 4 and 5). In this case, since the information flow graph only captures information flow between principals associated to the same controller, ProvSDN cannot prevent the malicious app from subverting, through the virtual switches, apps running on other controllers.

Figure 5.2: Attack Vector



Source: From the Author

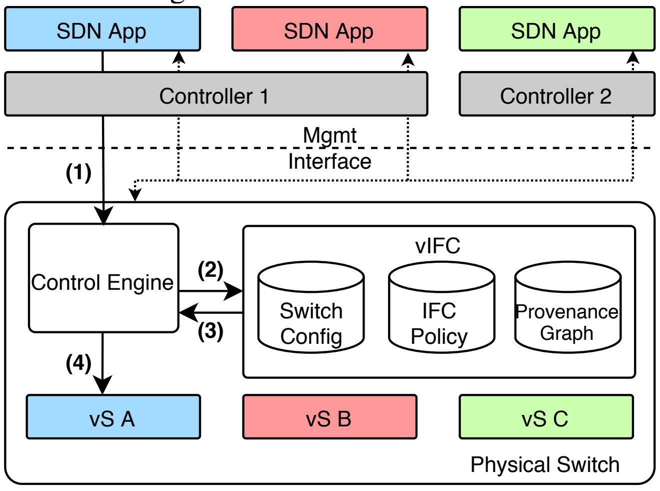
Fig. 5.2 illustrates an attack vector following Ujcich et al. (UJCICH et al., 2018)'s model. Observe that, while ProvSDN can prevent app a_1 from poisoning object o_2 using virtual switch a_2's permission p_3, it cannot track the "out of band" flow (dotted arrow) between virtual switches a_2 and a_3, allowing a_1 to use a_3's permission p_4 to poison o_3.

5.2 vIFC Implementation

Addressing CAP attacks that explore lightweight switch virtualization is challenging. In addition to tracking packet flows between multiple virtual switches, detecting information flow violations involving SDN apps and virtual switches requires addressing the following issues: (i) How to correlate information flow events between apps and virtual switches? And (ii) How to trace flow events in the data plane to originating/related apps in the control plane?

Our solution, Virtual Switch Information Flow Control (vIFC), addresses these issues by mediating IFC between apps and virtual switches. In our work, we extend Ujcich et al. (UJCICH et al., 2018)'s model by considering that virtual switches share a single control engine, which is also trusted and cannot be tampered with.

Figure 5.3: vIFC Architecture.



Source: From the Author

Fig. 5.3 presents an overview of vIFC conceptual architecture. It interfaces with the virtualization solution's control engine to indicate whether a given request from an app should be allowed or blocked. Before processing an app request arriving from the controller and sending it to the virtual switch (flow 1), the control engine forwards it to vIFC (2). The vIFC then queries the `Switch Config` database to obtain further information about the virtual switch target of the request (for example, whether the requesting app has the necessary permissions). Then, similarly to ProvSDN, vIFC updates a provenance graph (MISSIER; BELHAJJAME; CHENEY, 2013; UJCICH et al., 2018) in the `Provenance Graph` database, which tracks the origin of data and their flow. After an update of the provenance graph, vIFC verifies whether the resulting graph is consistent with the set of policies stored in the `IFC Policy` database. This process is done similarly as in ProvSDN. In case the app request does not violate established IFC policies, then vIFC signals the control engine (3) that the request is legitimate and may proceed; otherwise, the request is blocked. The control engine forwards the request to the virtual switch if allowed (4).

To support vIFC, we extend PvS to contemplate the following additional components:

- **Virtual Switch IFC Policy Enforcement.** Using policies to define if an information flow may be accepted or rejected is a design principle that vIFC uses to define the response that the flow will have. This design principle is extremely important for guaranteeing the information flow control enforcement since it may allow vulnerabilities that SDN applications may exploit if there is a lack of defined IFC policies.

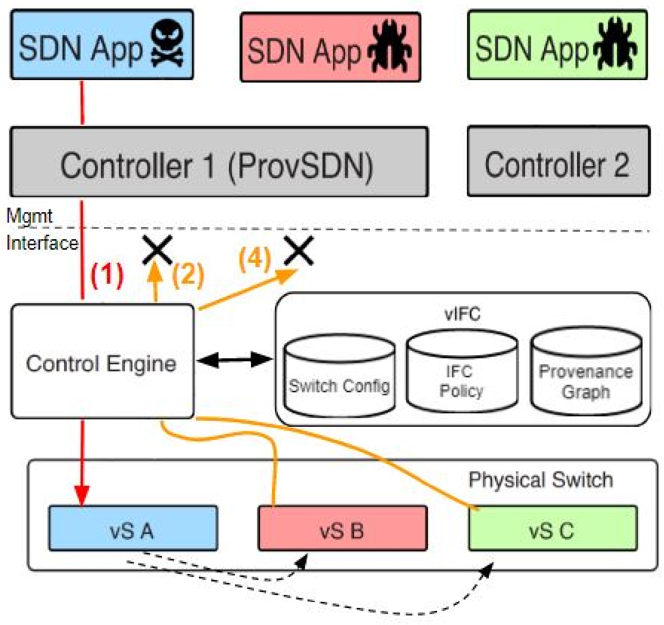
- **Multi Controller IFC Enforcement.** Using multiple controllers may exploit vulnerabilities and remain undetected, like in the state of the art solutions ProvSDN (UJCICH et al., 2018) that act directly in the control plane, and with several controllers, it is not able to detect information flow control violations in such a scenario. Therefore, connecting with multiple controllers, the vIFC must be able to detect CAP attacks with the defined topology.

5.3 vIFC Case Study

The case study mentioned before in Section 5.1 concerned the vulnerability that ProvSDN's method with data provenance to ensure information flow control is not sufficient for protecting virtual switch instances against CAP attacks. Therefore, we discuss the same case study demonstrating how the vIFC abstraction can efficiently detect and mitigate the malicious operation.

Fig. 5.4 presents the same use case except for using the vIFC to stop the malicious activity. In the scenario, the malicious application still performs the packet-out operation to the virtual switch (vS) A, that possesses `PACKET_*` permissions (flow 1 in Fig. 5.4).

Figure 5.4: vIFC Study Case.

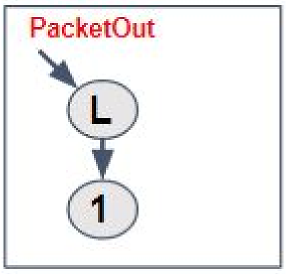


Source: From the Author

After the packet arrives in the control engine, the same forwards the packet information to the vIFC, which queries the Switch Config database searching for the permissions, like if the user in the application has permission to do such an activity. The provenance graph is then updated with the packet-out information containing the data provenance data of the entire operation, for instance, who sent the packet-out request and the permissions associated. The “out of band” flow (dotted arrow) occurs, resulting in the malicious packet-in operation intended to arrive in the SDN application. However, the control engine forwards that packet-in operation to the vIFC, which queries the switch config database to see permissions, and updates the provenance graph with the packet-in information. Considering that a CAP attack intends to gain privilege escalation by using a least privileged application to use a higher integrity one, the IFC policy database in vIFC must contain the policy to stop this attack, using a policy to block flows that come from an application with fewer privileges and aims to arrive at a higher one. At last, the vIFC checks the provenance graph with the same policy stored, and checks that there was a violation. At this moment, the packet-in operation is blocked (flow 2 in Fig. 5.4) and will not arrive at the SDN controller, yet the vIFC sends message information to the controller mentioning that there was a CAP attack attempt. When using multiple controllers, vIFC is also efficient in mitigating this attack, with the same *modus operandis* from before, the malicious packet-in flow is blocked by vIFC (flow 4 in Fig. 5.4)

Our vIFC abstraction is capable of detecting the information flows through the provenance graph, in which we use the users connected to the SDN applications and the virtual switches as the graph nodes. These nodes contain important metadata information that assists in mapping if a flow is coming from a least privileged application to a higher one, like the labels (set of integrity tags combined for checking permission level, later explained in Subsection 5.5). Therefore, considering the study case shown when the SDN application performs the packet-out request (flow 1 in Fig. 5.4), the provenance graph starts being created with the information of which user sent the flow, and to which switch the flow is intended. As is shown in Figure 5.5, the user (L for lower integrity) performs the packet request to the virtual switch.

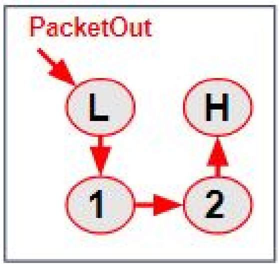
Figure 5.5: Provenance Graph Updated.



Source: From the Author

At the moment the information flow is intended for the SDN controller trying to perform the packet-in request (flow 2 in Fig. 5.4), the control engine interfaces that message with the vIFC abstraction, which checks the information associated with the packet and updates the provenance graph accordingly, for instance, vIFC can detect that the flow originated in the lower integrity application to the virtual switch “1", however it was designed to go to the virtual switch “2" and tried to perform the packet-in to the higher integrity application. Then, the provenance graph shown in 5.6 denotes the state of the graph containing the entire information flow path, including between the virtual instances, and then with the IFC policies, vIFC may check for violations.

Figure 5.6: Provenance Graph Completed.



Source: From the Author

5.4 CAP Attack Modeling and Use Cases

The case study demonstrated in Section 5.3, exhibits how a CAP attack may impact the SDN network with virtual switches to prove the concept. Yet, there are several known use cases that the operational impact of a CAP attack that may have drastic consequences. Therefore, we model two important use cases (Reactive Forwarding App, and Inband Telemetry App) showing the operational impact that this type of cyber attack may have on the SDN network, and demonstrate how vIFC can mitigate CAP attacks on them.

The Reactive Forwarding App (fwd) consists of an SDN application available in the ONOS controller. The application works by intercepting miss packets that arrive from the data plane and responding with a flow rule that allows the original switch connection. For example, host A intends to send a packet to host B. However, there is no entry in the match-action table that allows this flow to happen. Thus, the packet is sent to the controller with the intent of creating the flow rule. The Reactive forwarding app intercepts this packet and creates the following flow rule that is populated in the switch, causing the connection of host A to host B available.

The Inband Telemetry App (INT) case consists of an SDN application created for monitoring the network state of packets arrived from the data plane, explicitly checking the ID of each switch that the packet goes through and the queue depth of the switch. In case the application monitors an unusual behavior, actions are taken to mitigate the issue.

The attack vector for both use cases is the same, a network operator utilizes an SDN App from the SDN App store that possesses several networking functions, however, inside the application there is a fragment of malicious code that intends to do the CAP attack. Considering the Reactive Forwarding App (fwd), a malicious ARP reply packet is sent by the Trigger app, which only has `PACKET_*` permissions. This SDN application works by intercepting miss packets that arrive from the data plane and responding by creating a flow rule according to the packet sent. Then, the corrupt flow rule installs the switch's malicious table entry resulting in a DoS in the data plane. The Inband Telemetry case may also be explored by the CAP attacks, which a malicious trigger application with `PACKET_*` permissions can explore it by performing a malicious packet-out request containing fake telemetry data to simulate a scenario that does not exist in the data plane, causing the INT app to respond accordingly and install a malicious flow rule to mitigate the fake scenario caused by the packet with false telemetry information.

The vIFC can efficiently detect and mitigate both CAP attacks by using data provenance, collecting information into the provenance graph about which SDN app acted, the user permissions related to the application, where the flow goes through, and where the flow is supposed to go. Then, checking with the IFC policies, vIFC can accurately detect a violation and stop malicious flows.

Given both the Reactive Forwarding App and the Inband Telemetry CAP attack scenarios, vIFC stores in the provenance graph information about the Trigger app, like the user and the permissions he has. At the moment the packet-in request is performed by the virtual switch, the provenance graph is updated with information of the legitimate SDN app that the flow is trying to go to (fwd or INT app), like the user permissions for the legitimate application, from which switch the packet-in was requested, and other data. By checking the information flow control policies, vIFC can detect violations with the provenance graph, for instance, there is a policy to block flow from least-privileged permission applications to higher integrity ones, then since the Trigger application does have fewer permissions than the Reactive Forwarding or the INT app, vIFC sees the violation and blocks the flow from reaching the legitimate application.

The operational impact of the CAP attack on these use cases is shown in the evaluation Section 6.3, while also demonstrating that vIFC can successfully detect and mitigate this type of cyber attack incurring a marginal overhead.

5.5 Formalizing the Policies and Labels

Throughout this proposal, we use policies in both the control engine abstraction and the vIFC abstraction, which must be addressed along with examples for both abstractions. Therefore, we formalize how these policies are defined. Also, we must define how the SDN applications get the associated labels.

We define two types of security policies: (i) the policies for switch access control, and (ii) the policies for information flow control. The policies for switch access control cover requests allow the possibility to enforce fine-grained access control For example, a user may have specific permissions (e.g., Read and Write) to some virtual switches. A user may also belong to a user group that has specific access permissions to certain switches. A switch may also be part of a switch group, to which specific permissions may be granted to a user or user group. These policies are independent of the southbound protocol, like the P4Runtime, and allow for the network operator to manage read/write permissions for virtual switch configuration, device events, flow rules, packet I/O events, and registers/counters. The second type of policy that we formalize is the IFC policies, which can be expressed containing the integrity labels for each of the SDN applications that the information flow goes through, the enforcement check to detect violations related to the specific operation, like a write request. Finally, we define the response to perform when there is a violation of the IFC policy. For instance, the network operator may define a policy to block packets from a least privileged application that may only perform a packet-out request, and that wishes to arrive in a higher integrity application that may perform a write request into the virtual switch. Therefore, the policy would be defined as: $p = (\{packet\}, \{packet, flowrule\}, WRITE, BLOCK)$.

Regarding how to assign the labels associated with the SDN applications, we follow the same model as the ProvSDN (UJCICH et al., 2018) utilizes, which a network operator labels apps with integrity tags, resulting in the applications having their integrity label composed of a subset of integrity tags. These integrity tags may range from permissions to perform packet operations, like Packet I/O, or permissions to make a write request to the switch. In our model, the network operator also may define the IFC policies, yet we update the model focusing on the user connected to the SDN application instead of the application itself. Therefore, we define a relation of users' integrity labels to switches as the mapping of the permissions a user has to perform actions on some switch. For example, a user u_1 can perform packet events (packet in/out) in the switch s_1 only if that user has the *packet* integrity tag for that switch, i.e., $\{u_1, (\text{packet}, s_1)\}$.

5.6 Existing Security Models

There are existing security models that may enforce access control, like the Bell-LaPadula (BLP) model (LAPADULA; LAPADULA; BELL, 1996) which intends to bring a mandatory access control in the information of the systems. The BLP model supports labels for determining the level of access of certain information, focusing on avoiding leakage of data, then, the focus of the BLP model is confidentiality. BLP model expresses certain principles to achieve the level of security, like “no write down" and “no read ups" principles according to the permission that the user has to that information. Regarding integrity security models, there is the Biba model (BIBA, 1977), which compares the level of integrity levels associated with the information to determine if the information flow is corrupted.

These type of security models can be associated with the vIFC abstraction since our abstraction inherit some aspects to detect and mitigate malicious information flows that may end up affecting the virtual switches. Our security model also utilizes labels as the models described to classify the level of access control of the users and virtual switches, we have support to methods like the write and read which are used, and our abstraction may block the information flows with the same behavior as, for instance, the “no write down" in case there is an IFC policy to perform it. However, our abstraction does differ in some aspects, like more levels of labels available in the system, and the ability to dynamically alter the IFC policy if it must be changed by the network operator, which considering the described models is static, in other words, the policy must be known in advance.

6 EVALUATION

In this chapter, we show that the abstractions implemented satisfy the proposed goals, maintaining significant results while guaranteeing low overhead. Firstly, we introduce a study case involving the integration of PvS with a NetFPGA SUME, involving how important concepts of the PvS were performed. Then, to evaluate both the PvS Control Engine and vIFC, two main evaluation sections were created to discuss the proposed methodology, testbed topologies, and results. In Section 6.2 we discuss the evaluation of the PvS Control Engine, in an experiment using a NetFPGA board to assess the operational impact that virtual switch management abstractions cause to network flows and to SDN apps, as well as measuring the latency of actions like Packet I/O. In Section 6.3, we provide evaluation results of vIFC, which demonstrates the methodology of the CAP attacks along with their operational impact in the SDN network, along with the vIFC results for detecting and mitigating this type of attack.

6.1 Integrating PvS with the NetFPGA SUME

To allow the integration of the PvS with the NetFPGA SUME, we demonstrate how this combination were performed, in which we highlight important concepts like Packet I/O or traffic steering along with the NetFPGA.

6.1.1 P4Runtime Support for Virtual Switches with the NetFPGA

The PvS control engine still supports the same following remote procedure calls described in Subsection 4.4.1 regarding the integration with the NetFPGA. Yet, some changes were performed to make this support available.

To determine the target switch of a given message request, we use a numeric identifier and rely on the `device_id` field in P4Runtime protobuf message request. Virtual switch identifiers are (i) assigned by PvS upon deployment of virtual switches, (ii) stored in the `SwitchConfig` database, and (iii) returned to the network administrator. Regarding the integration with the NetFPGA, we reserve the switch id 0 for configuring the Ingress and Egress match-action tables on the Forwarding Engine. These switch ids must then be used by apps running in the Application Plane to determine to which switch a given message request refers to.

Handling P4Runtime Read and Write message requests also require detailed information about the virtual switch pipeline – more specifically, its match-action tables (their fields and actions) and respective numeric identifiers. To this end, during the virtual switch deployment process, we extract from its switch.dat information about the switch pipeline, including the match-action tables and registers available, and import it to the SwitchConfig data base. This file is important for this integration since it is used by the NetFPGA to map the virtual switch pipeline information.

Then, the main difference of this integration from the process described in Subsection 4.4.1 comes in the Platform API which uses methods provided by the P4 Tables API and P4 Regs API modules to this end, to rephrase it, specifically to the NetFPGA SUME. These modules (i) query the SwitchConfig data base to identify which shared libraries handle the target virtual switch, (ii) load the identified library, and (iii) call the library method that forwards the request to the virtual switch in the PvS Forwarding Engine. For example, for a write request to add a match-action table entry to the green virtual switch, the Platform API delegates the request to the table_cam_add_entry method of the P4 Tables API, which in turn queries the SwitchConfig data base for the libcam library of the green switch. The P4 Tables API then loads the identified module and calls the cam_read_entry from it, passing the field values, action name, and action parameters received in the write request.

6.1.2 Traffic Steering Between Virtual Switches

After deploying a virtual switch, the network administrator must determine flow steering from/to (and between) virtual switches. As mentioned earlier, we use VLAN tagging and device_id to support flow steering. For determining flow steering, the administrator must configure the Ingress and Egress tables in the forwarding engine. The configuration of these tables follows the same philosophy of writing P4 match-action table entries and therefore can be done either using P4Runtime (using device_id = 0 in the message request) or the Command-Line Interface.

The administrator may configure the IvSI Ingress table to match the RX port from which the packet entered (from sume_metadata.src_port) and the packet VLAN tag (e.g., pkt_in.vlan_id). Conversely, the administrator may configure the Egress table (hosted in the Output vS Interface) for determining the output physical port of a packet exiting a virtual switch. For steering packets between virtual switches (virtual networking within PvS), the network administrator must provide as parameter to the `Egress.forward` action the index of a vTX as destination port. According to the P4-NetFPGA Workflow Overview[1], valid indexes are 2 (`0x00000010`, for `nf0_dma`), 8 (`0x00001000`, for `nf1_dma`), and 32 (`0x00100000`, for `nf2_dma`); `nf3_dma` is reserved for packet-in and packet-out messages, as discussed next. The Virtual Networking daemon in the Platform API will then read that packet from the host network interface attached to that virtual port, and send the packet again through that same interface. As a result, the packet will loopback to the respective vRX input virtual port.

6.1.3 Packet I/O Between Virtual Switches of the NetFPGA and Controllers

PvS also enables handling packet-in and packet-out messages between virtual switches and controllers. For handling packet-in, i.e., sending a data plane packet to the controller for further inspection, the virtual switch sets `sume_metadata.dst_port` to 128 (`0x10000000`, for `nf3_dma`). This is a reserved port attached to a host network interface that handles packet I/O. Upon receiving a packet with the determined destiny port `sume_metadata.dst_port = 128`, the OvSI will copy the packet to the `nf3_dma` port, and write in a reserved register the `device_id` of the virtual switch that generated the packet-in, the virtual port from which the packet was received, its timestamp, and any other information required to assemble a `ControllerPacket-Metadata` structure as discussed in the P4Runtime specification.

The Virtual Networking daemon in the Platform API will read the packet from the host network interface attached to `nf3_dma`, and the metadata from the register using the `libsume` driver, and forward them to the VSMA. The VSMA will then look for active client connections authenticated for a user with packet-in permissions for that virtual switch. For each connection, the VSMA will send a `StreamMessageResponse`, through the persistent `StreamChannel`, containing a `PacketIn update`. In case there is no user authenticated for that device, the packet-in is discarded.

For handling packet-out, the app running in the control plane must send a certain type of request defined as a `StreamMessageRequest` message with a `PacketOut` update. The metadata must contain the `device_id` of the virtual switch that will receive the packet-out, input virtual port, etc. The VSMA will forward the packet to the Platform

[1]https://github.com/NetFPGA/P4-NetFPGA-public/wiki/Workflow-Overview

API down to the Virtual Networking daemon, which will copy the packet-out to the host network interface attached to the `nf3_dma` port, and the associated metadata to the reserved register (the latter, written using the `libsume` driver). The IvSI will receive the packet-out and metadata and determine, based on the metadata, to which virtual switch the packet-out will be forwarded to.

6.2 Control Engine Evaluation

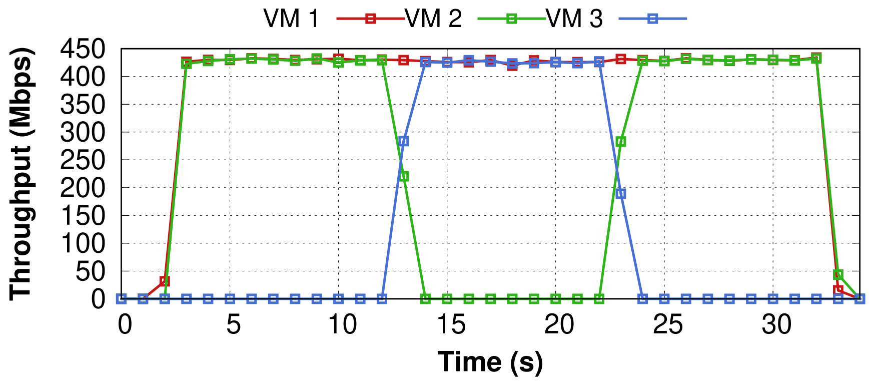
To measure the efficiency that the control engine has to allow tenants to manage the virtual switches, we discuss in this experiment the impact that the SDN applications have in the virtual switches by performing packet I/O, writing and modifying table entries of the match-action tables, to support the SDN management in a NetFPGA SUME.

To perform this experiment, we are utilizing the P4VBox (SAQUETTI et al., 2020) as the virtualization solution, and we deploy our PvS control engine abstraction in Ubuntu 16.04.3 with kernel 4.10.0-28-generic, Intel Core i7-7700K (8 cores), and 32 GB of RAM. The virtualization solution was using a NetFPGA-SUME Virtex 7 board (SKU 410-301) at 200 MHz operating frequency, installed on an Asus H110M-CS with PCI-e Gen3 x16 bus, BIOS v4210. We used SFP+ 10G transceivers to connect the NetFPGA-SUME to our testbed.

To express that the control engine allows secure SDN management using open standard protocols, like P4Runtime, we experimented PvS using a network app connected directly to P4Runtime, and another running on top of ONOS 2.4.0 SDN controller. As discussed in (TIRONE, 2020), the experiment was explained focusing on the reconfiguration times in the NetFPGA. However, there is the need to talk through the control engine behavior in the experiment, showing how the SDN apps have managed the match-action table information securely using P4Runtime. Our main objective with this experiment is to demonstrate functional use of the PvS control engine abstraction, allowing to manage the virtual switch instances securely and independently, which the SDN application must send its credentials and check the permissions for the message request operation, and also the efficiency in performing the message request operation.

The experiment began with one SDN application and three virtual machines, each connected to one port in the NetFPGA. Firstly, the SDN app connects directly using P4Runtime by using the StreamChannel RPC, sending the correct credentials to the control engine. Specifically in this case, the SDN app has admin permissions to perform packet I/O, Write requests, and other operations. At instant t = 0, an iperf3 flow was created from VM1 to VM2 by configuring the l2 switch to steer incoming flows to VM2 in the match-action tables. When t = 10, the SDN app sends a Write request RPC to modify the table entry available in the dmac table, setting the output port to the VM3's port, therefore forwarding the iperf3 flow from VM1 to VM3. Then, at t = 20, the SDN app sends another Write Request RPC to update the dmac table entry again, by returning to the original flow order VM1 to VM2.

Figure 6.1: vS Management Impact.



Source: From the Author

This experiment is demonstrated in Figure 6.1, which exhibits the throughput of the iperf3 flow with the impact caused by the message requests made by the SDN application. The throughput of VM1 remained stable during table update, where the other ones were affected. The Write Request RPC took over 200 ms to take effect, and there were 909 packets lost during the traffic steering. We may conclude that the SDN application can effectively manage the virtual switch by inserting and modifying the table entries available in the switch, while checking for the credentials and permissions, and incurring a small overhead to the virtualization solution for the RPC to take effect.

We also built an ONOS SDN application to manage the L2 switch in the data plane, aiming to prove the concept that the PvS control engine works with a well-known SDN controller solution. This application running in ONOS can perform Packet I/O and has the L2 switch mapping information of match-action tables that makes available for the application to create several operations to manage the switch. For instance, the app creates a flow rule in the controller to insert a table entry in the switch, which then results in a Write Request RPC to the data plane.

We also analyzed the impact of virtual switch management to packet input/output. Observe that five stages contribute to packet I/O latency: sending the packet from the forwarding to the control engine back and forth, through the PCI-e bus (two stages), processing the packet I/O in the control engine (including permission verification) back and forth, and processing the packet I/O in the SDN app. We observed an average of 10 ms packet I/O round-trip latency in successive experiments, with the PvS control engine and SDN app (directly connected to P4Runtime) contributing in around 40% and 60%, respectively. The contribution of the forwarding engine was on the order of nanoseconds, thus negligible. That latency could be reduced by migrating the control plane implementation from Python to C and handling packet I/O with DPDK (DPDK Project, 2020).

6.3 vIFC Evaluation

6.3.1 Cross-App Poisoning Attacks

Considering that an application may perform cross-app poisoning attacks in the SDN network, the entire control-data plane state can be affected, harming the operation of the switches in the data plane and poisoning the view that the control plane has. Therefore, we intend to evaluate how this type of attack may influence the SDN network with the two conceptual CAP attacks discussed in Section 5.4, demonstrating the importance of using vIFC to mitigate this attack.

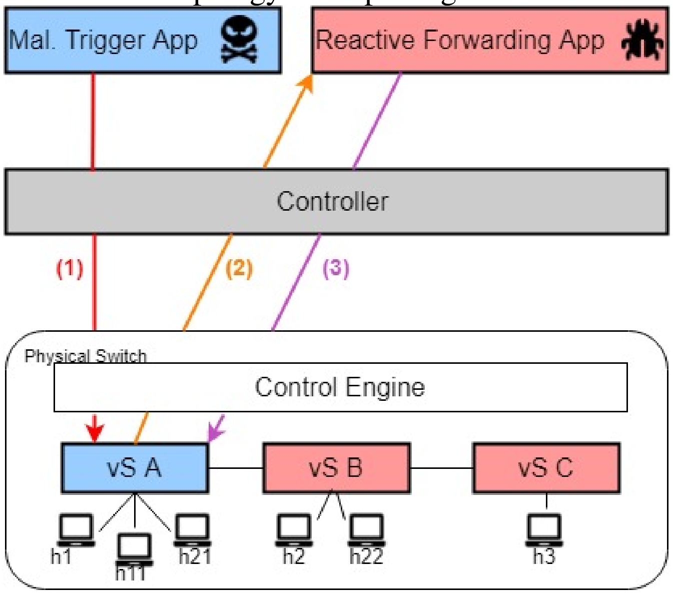
The previous experiment used simple SDN applications to prove the concept, however, the SDN applications scope is much wider. Thus, we are evaluating two known use cases that a network operator may utilize: (i) an SDN application available in the ONOS controller called the Reactive Forwarding App (fwd), which listens to miss packets from the data plane and accordingly installs flow rules; (ii) an Inband Telemetry App (INT), which collects information about the network state and reacts according to it.

To do this experiment, we emulate the virtual switches using Mininet (TEAM, 2017) and bmv2 (CONSORTIUM et al., 2018). All evaluations regarding the Subsection 6.3.1 and Subsection 6.3.2 were carried out in a VM with Ubuntu 18.04 containing the ONOS 2.4.0 SDN controller and Mininet (TEAM, 2017). The VM was allocated three 2.6 GHz Intel Core i7-9750H CPUs and 8 GB of memory. With this evaluation of the attacks, one can measure the time it takes for the attacks to impact the SDN network and the determined operational impact of the attack.

6.3.1.1 Reactive Forwarding CAP Attack

To perform this experiment to see the impact of the CAP attack with the Reactive Forwarding App [2], we have created an SDN application `Trigger` that sends a malicious ARP reply with content that the valid IP address is at the malicious MAC address as is conceptually discussed in Section 5.4. In the data plane, we are using a simple L2 switch p4 code in the bmv2 switches. As shown in Figure 6.2, the testbed topology for the experiment is set where the malicious application is present and tries to explore the CAP attack vulnerability to affect legitimate SDN App.

Figure 6.2: Testbed topology for exploring CAP attack in fwd app.



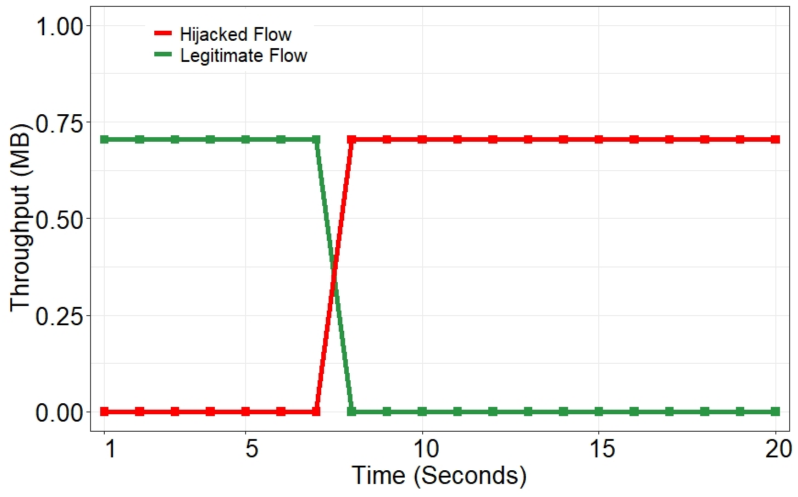
Source: From the Author

The *modus operandis* of the attack consists of the Trigger app sending a malicious ARP reply packet-out request to virtual switch A, which sends the packet information back to the controller, and the Reactive Forwarding App intercepts this packet. Then, the legitimate application creates the bogus action, in this case, is a malicious flow rule that modifies a flow entry in the match-action table of virtual switch A, which sets the output port of a host to a malicious entity, therefore allowing legitimate flows to be hijacked.

In Figure 6.3, we demonstrate the operational impact that the CAP attack has when exploring this use case. The green line consists of an iperf3 legitimate flow, which is deeply affected by the attack and ends up hijacked in a matter of seconds, showed by the red line. Considering the entire time captured, the Trigger application was initiated by the attacker at the five seconds mark, and the entire time needed for the attack to

[2]Reactive Forwarding GitHub code: <https://github.com/opennetworkinglab/onos/blob/master/apps/fwd/src/main/java/org/onosproject/fwd/ReactiveForwarding.java>

Figure 6.3: CAP attack impact on network flows.



Source: From the Author

be successful was around 1.22 seconds measured by when the packet-out was sent from ONOS and when the flow was hijacked. In Table 6.1, each operation that the CAP attack uses it measured by its latency.

Table 6.1: Latency of each operation performing the fwd CAP Attack.

Flow	**Time(s)**
Packet-Out ONOS => vIFC	0.0016
Out-of-band	1.033
Packet-In vIFC => ONOS	0.0047
ONOS Packet-In => Write Request	0.02
ONOS Write Request => vIFC Write Request	0.062
vIFC Write Request => bmv2 Switch Table	0.048
bmv2 Switch => iperf3 flow hijacked	0.0525
Total	**1.22**

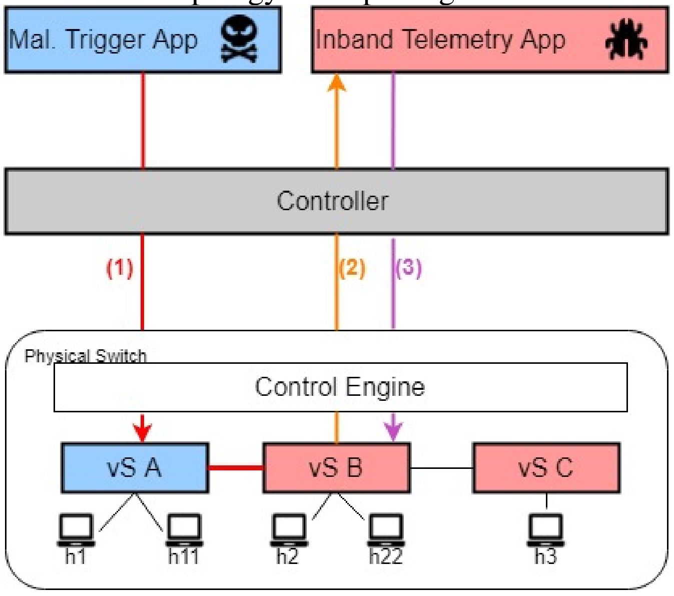
6.3.1.2 Inband Telemetry CAP Attack

To explore CAP attacks using the Inband Telemetry App, we have created an SDN application `Trigger` which executes a malicious packet-out every ten seconds to the switch as is conceptually discussed in Section 5.4. In the data plane, we are emulating the switches with the Multi-Hop Route Inspection (MRI) [3] that tracks the path and length of queues that the packet goes through. The packet-out from the Trigger application was manually crafted to do the “out of band” flow to reach virtual switch B, and also contains fake telemetry to explore the vulnerability using the Inband Telemetry App.

Figure 6.4 exhibits the testbed topology and *modus operandis* of the CAP attack when targeting the Inband Telemetry scenario. The Trigger application sends the mali-

[3]MRI GitHub code: <https://github.com/p4lang/tutorials/tree/master/exercises/mri>

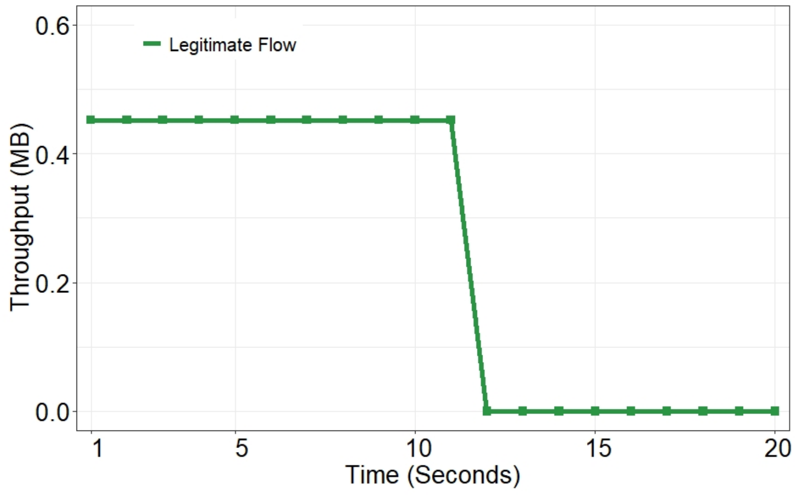
Figure 6.4: Testbed topology for exploring CAP attack in INT app.



Source: From the Author

cious packet-out every ten seconds with fake telemetry data signaling that the queue depth of the virtual switch A is equal to 4,000. The packet-out was also designed to go to the virtual switch B through "out of band", and a resulting packet-in is made to the controller. The Inband Telemetry application intercepts the packet and monitors the switch ID and queue depth existing of each packet, considering a normal scenario of the MRI p4 code the queue depth of the switch would be in a mean of 40, however with the CAP attack impact the queue depth is above 4,000. The legitimate application identifies the value as an outlier, i.e., higher than than usual. In this case, a bogus action is made by the application due to the attack, in this case, a malicious flow rule is created to disrupt normal flows in the virtual switch.

Figure 6.5: CAP attack impact on the network flow.



Source: From the Author

Figure 6.5 indicates how this CAP attack severely impacts the data-plane operation. The green line shows a legitimate flow between host `h11` of the virtual switch A and the host `h22` of the virtual switch B. The CAP attack, which is launched at the five seconds mark, deeply affects the existing network flow by disrupting a matter of seconds. More specifically, the application needs 5.09 seconds to disrupt the data-plane operation. Compared with the Reactive Forwarding CAP attack experiment, the time was higher due to the Trigger application performing several packet-out requests with an interval of time, and the "out of band" flow between virtual switches, as is shown in Table 6.2.

Table 6.2: Latency of each operation performing the INT CAP Attack.

Flow	**Time(s)**
Packet-Out ONOS => vIFC	1.484
Out-of-band	3.39
Packet-In vIFC => ONOS	0.05
ONOS Packet-In => Write Request	0.107
ONOS Write Request => vIFC Write Request	0.051
vIFC Write Request => bmv2 Switch Table	0.0069
Total	**5.09**

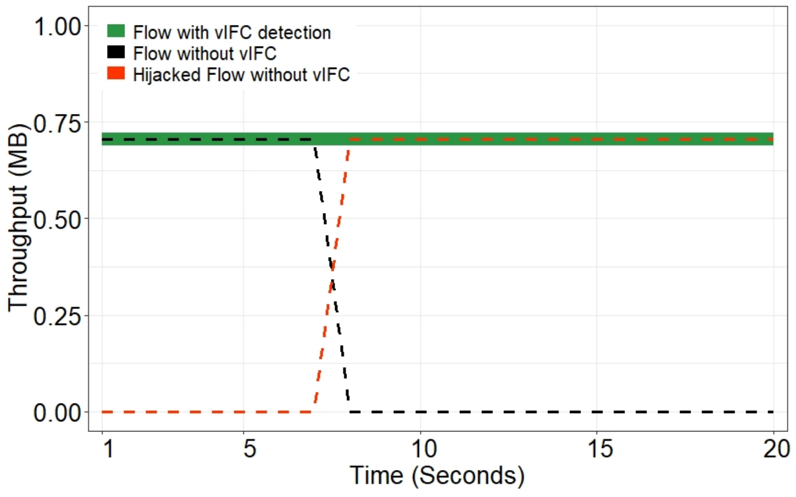
6.3.2 vIFC Efficiency

To determine that vIFC can mitigate known scenarios in which CAP attacks may occur, we evaluate the efficiency that vIFC has considering both CAP attacks shown in Subsection 6.3.1. The experiment intends to measure that vIFC efficiently detects the CAP attack that happened, mitigates it without incurring too much overhead into the virtualization solution, and rate potential false positives or negatives that may take place.

To assess that vIFC does not pose significant overhead to mitigate the proposed CAP attacks, the experiment used two data sets of 10 malicious packet-out requests from least-privileged applications that later arrive in a higher priority app to perform a certain bogus action, characterizing as a CAP attack. From the first data set, 10 packet-out requests target the Reactive Forwarding app, while the other data set has 10 packet-out requests targeting the INT app.

Considering CAP attacks targeting the Reactive Forwarding application as is shown in Figure 6.3, vIFC accurately detects the attempt and the impact caused by the CAP attack does not affect the existing flow. Demonstrated by Figure 6.6, there are two scenarios: (i) the vIFC is not deployed and the CAP attack occurs (showed by the dashed lines), and

Figure 6.6: vIFC mitigation of CAP attack in fwd app.

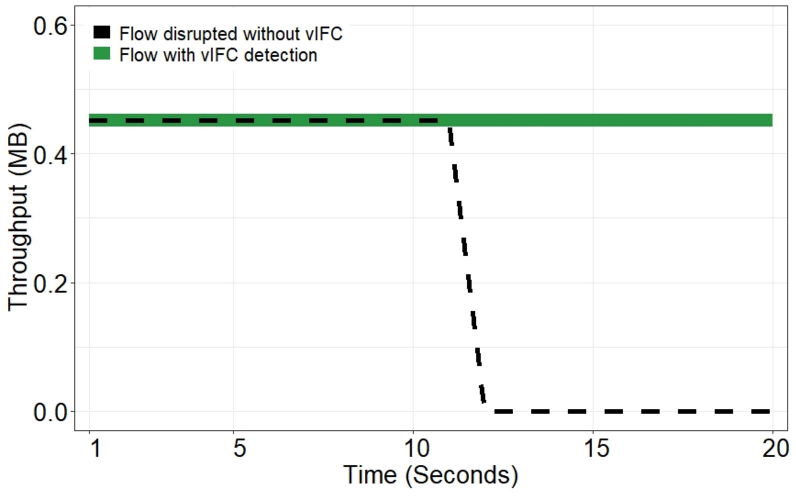


Source: From the Author

(ii) the vIFC is deployed and mitigates the CAP attack not allowing the iperf flow to be hijacked (showed by the green solid line).

When the CAP attack targets the INT application, vIFC also efficiently detects the attempt, and the attack's impact does not disrupt the legitimate flow. Figure 6.7 exhibits the same two scenarios (with and without vIFC), and when vIFC is deployed the legitimate flow is not disrupted by the attack (showed by the green solid line).

Figure 6.7: vIFC mitigation of CAP attack in INT app.



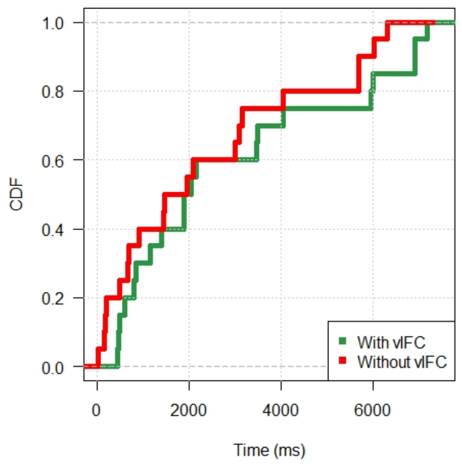
Source: From the Author

6.3.2.1 vIFC Incurring Latency

Utilizing the defined data sets that contain information about both the Reactive Forwarding CAP attack and the Inband Telemetry one, we evaluate how much latency is posed by vIFC considering these complex scenarios. In this experiment, we check the time that the packet-out request arrived in the control engine to the moment that the packet-in request is blocked or not by vIFC. Then, vIFC's latency is compared with the latency when information flow control is not enforced.

Figure 6.8 exhibits the result of the experiment demonstrating the flow latency of both without an IFC solution and one with the enforcement of information flow control to the virtualization solution for both CAP attack cases. 20 trials were run to detect the time of the scenarios.

Figure 6.8: Flow latency macro-benchmark.



Source: From the Author

The average latencies were 1,107 ms, and 1,431 ms respectively. Considering the experiment with the INT application, the average latencies were 4,179 ms and 4,656 ms respectively. The obtained values do pose a significant latency in the order of ms due to several operations performed in the entire process, like the "out of band" flow. Considering the entire CAP attack scenarios, vIFC posed a lightly overhead to the virtualization solution, more specifically, 324 ms and 477 ms respectively for the Reactive Forwarding case and the INT case. This slight increase is given due to the information flow control enforcement that gathers the information in the provenance graph, checks the labels of applications, and checks for violations with the stored policies.

6.3.2.2 Precision and Recall

In this experiment, we intend to analyze the accuracy that vIFC has, by detecting true positives, false positives, and other variables of two given data sets: (i) the first data set was used in the Figure 6.8, which contains malicious packet-out requests (from both the Reactive Forwarding CAP attack and the INT attack scenario) that violate the information flow control policies of vIFC, and (ii) a second data set of malicious packet-outs that do not infringe the policies stored in vIFC. This second data set explores the scenario in which the Trigger application has the same integrity permissions as the SDN app to determine if vIFC may detect as a violation. Besides, we explore a third scenario where the attacker utilizes the virtual switch as the attack vector, such a factor was left out of the scope, however, will be open for future work. For such a scenario, we issue 5 malicious packets to do the "out of band" flow.

With this obtained result, precision and recall can be measured. The precision mentioned in (CHEN et al., 2020) consists of the fraction of the data set that successfully detected all positive cases, while the recall measures the successfully detected with true positive cases. More specifically, to calculate the precision we use true positives and false positives, with the following formula: $P = TP \div TP + FP$. The recall differs by using the fraction true positives divided by true positives plus false negatives, with the formula: $P = TP \div TP + FN$.

Table 6.3: Performance of vIFC in different permission scenarios.

Permission Scenarios	**TP**	**TN**	**FP**	**FN**
SDN App lower integrity => Higher	20	0	0	0
SDN App higher integrity => Higher	0	20	0	0
vS => Higher	0	0	0	5
Total vIFC	**20**	**20**	**0**	**5**

Table 6.3 exhibits the number of detected CAP attacks in each scenario by analyzing if the violation occurred as if should. During the first scenario, the least-privilege integrity application performed a CAP attack to send the flow to a higher integrity application. vIFC was effective in these true positives that violated the stored policy to avoid flows from a lower integrity app to a higher one, not detecting any false positives or false negatives. In the second scenario, the Trigger application had the same set of integrity labels as the destination app, without a specific policy to block these types of scenarios in vIFC. Even though the CAP attack is happening in this scenario, we defend that vIFC must detect the violations according to policies stored. Therefore, the vIFC successfully detected the packet-out requests as true negatives. At last, the third scenario was out of the scope when first constructing vIFC, where the attacker explores the virtual switch directly and does behave as a CAP attack since the flow is not originated in the SDN application in the application plane. However, there may be the case of attack that may poison the virtual

switches, but which vIFC detects as a false negative, since that the malicious packet-out request did not originate all the flow in the SDN app. This and other attack vectors will be explored in future work.

The scenarios demonstrated some possibilities of several variables that may be explored in an SDN network. Therefore there is a limitation in these results that they do not cover all scenarios. Considering only the three permission scenarios, the precision of the system is 1.00 due to no false positives, and the recall consisted of 0.8 since there were five false negatives of the malicious switch scenario case.

7 FINAL CONSIDERATIONS

This chapter presents the final considerations of this dissertation. We first summarize in Section 7.1 the main contributions of this work, provide an overview of obtained results, and enumerate the challenges faced during the proposal's development. Then, in Section 7.2, we discuss future research to help make SDN networks more manageable and secure, highlighting aspects that this dissertation approached, like the information flow control enforcement that may open several precedents for future proposals.

7.1 Summary

Achieving data plane programmability provides several opportunities for network operators to define their behavior. With virtualization, the pattern is followed, beginning precedents to have several virtual entities running on top of a physical substrate. Accomplishing such a model requires addressing several challenges that range from managing the virtual instances to guaranteeing consistent security levels avoiding possible vulnerabilities and attacks. Therefore, in this dissertation, we addressed these research challenges by firstly presenting the PvS Control Engine to support management and compatibility with open control-data interfaces. Then, we presented the vIFC, an information flow control enforcement solution to provide security against known attacks, like the Cross-App Poisoning (CAP) attacks.

The PvS Control Engine is presented in this work contributing with an abstraction of a single control engine and management channel for multiple network operators using SDN applications. The PvS control engine's goal is to enable management of virtual switch instances using open southbound interfaces like P4Runtime, while maintaining consistent security levels, with proper credentials of the applications and the set of permissions being checked. It also ensures the tenant's actions are legitimate events, avoiding malicious actions to switches beyond their control. Then, we proposed the vIFC, to mitigate a type of attack focused on gaining privilege escalation by poisoning SDN apps with a malicious application through shared objects. The vIFC contributed to PvS by implementing information flow control enforcement through data provenance to check for violations that might occur with the IFC stored policies, guaranteeing that such attack does not impact the virtual switches' operation.

Proofs-of-concept were performed to ensure the correct secure manageability of

SDN apps with virtual instances with the Control Engine, and the effectiveness of vIFC to mitigate this type of attack. Then, we evaluated the operational impact that a CAP attack may cause in an SDN network with known use cases, and proposed the vIFC to defend the network with the same use cases. Results exhibited that both the Control Engine and the vIFC were successful in their proposals, ensuring the design principles that each posed, and guaranteeing low overhead to the virtualization solution.

7.2 Future Work

This dissertation presented important contributions to a more secure and manageable SDN architecture, however, several future directions can be explored to increase security in SDN networks. We discuss some approaches in this section.

Compatibility with other management interfaces. The PvS Control Engine currently supports P4Runtime as the southbound interface for communicating with the SDN control plane. Future directions would be to explore adding compatibility with OpenFlow, which would broaden the possibilities for network operators to manage the virtual switch instances, and allow them to control the virtual switches with several other controllers like Ryu (TOMONORI, 2013). To open precedents in supporting OpenFlow, the PvS Control Engine abstraction would need to have the interface that translates the message request into the data plane device configuration.

Attack vulnerabilities. Several types of attack may happen in an SDN network, and since proposals for ensuring the security of virtual switches are still being explored, vulnerabilities might be exploited by these threats. Therefore, we intend to analyze possible attack vectors and other attacks that may pose a risk to the virtual instance. As it was shown in the Subsection 6.3.2, there are several possibilities that an attacker may try to perform a attack, for instance, a malicious virtual switch controlled by the attacker creates a malicious flow aiming to reach the legitimate SDN application to perform a bogus action that may cause disruption or another type of malicious activity in the data plane.

Forensics. With vIFC, the data provenance contains information about events that occurred, and then check for violations which the forensics may able to analyze. Future investigations in this path may improve the analysis of the data provenance, for instance, indicating an early intrusion detection by the early actions of the same application.

REFERENCES

AFOLABI, I. et al. Network slicing and softwarization: A survey on principles, enabling technologies, and solutions. **IEEE Communications Surveys & Tutorials**, IEEE, v. 20, n. 3, p. 2429–2453, 2018.

ARBETTU, R. K. et al. Security analysis of opendaylight, onos, rosemary and ryu sdn controllers. In: IEEE. **2016 17th International telecommunications network strategy and planning symposium (Networks)**. [S.l.], 2016. p. 37–44.

Aruba Networks. **SDN Apps - Airheads Community**. 2020. Retrieved July 24, 2020 from https://community.arubanetworks.com/t5/SDN-Apps/ct-p/SDN-Apps.

BACON, J. et al. Information flow control for secure cloud computing. **IEEE Transactions on Network and Service Management**, IEEE, v. 11, n. 1, p. 76–89, 2014.

BARI, M. F. et al. Data center network virtualization: A survey. **IEEE Communications Surveys & Tutorials**, IEEE, v. 15, n. 2, p. 909–928, 2012.

BATES, A. et al. Let sdn be your eyes: Secure forensics in data center networks. In: **Proceedings of the NDSS workshop on security of emerging network technologies (SENT'14)**. [S.l.: s.n.], 2014.

BELT, J. van de; AHMADI, H.; DOYLE, L. E. Defining and surveying wireless link virtualization and wireless network virtualization. **IEEE Communications Surveys & Tutorials**, IEEE, v. 19, n. 3, p. 1603–1627, 2017.

BERDE, P. et al. Onos: towards an open, distributed sdn os. In: **Proceedings of the third workshop on Hot topics in software defined networking**. [S.l.: s.n.], 2014. p. 1–6.

BIBA, K. J. **Integrity considerations for secure computer systems**. [S.l.], 1977.

BLENK, A. et al. Survey on network virtualization hypervisors for software defined networking. **IEEE Communications Surveys & Tutorials**, IEEE, v. 18, n. 1, p. 655–685, 2015.

BOSSHART, P. et al. P4: Programming protocol-independent packet processors. **ACM SIGCOMM Computer Communication Review**, ACM New York, NY, USA, v. 44, n. 3, p. 87–95, 2014.

BOSSHART, P. et al. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. In: **ACM SIGCOMM 2013**. New York, NY, USA: ACM, 2013. p. 99–110. ISBN 9781450320566.

Broadcom Inc. **Broadcom's new Trident 4 and Jericho 2 switch devices offer programmability at scale**. 2019. Available from Internet: <https://www.broadcom.com/blog/trident4-and-jericho2-offer-programmability-at-scale>.

BUNEMAN, P.; KHANNA, S.; WANG-CHIEW, T. Why and where: A characterization of data provenance. In: SPRINGER. **International conference on database theory**. [S.l.], 2001. p. 316–330.

CHEN, J. et al. Chaperone: Real-time locking and loss prevention for smartphones. In: **29th {USENIX} Security Symposium ({USENIX} Security 20)**. [S.l.: s.n.], 2020. p. 325–342.

CONSORTIUM, P. L. et al. Behavioral model (bmv2). **URL: https://github.com/p4lang/behavioral-model [cited 2020-01-21]**, 2018.

CONSORTIUM, T. P. L. **P416 Language Specification**. [S.l.], 2017. Available from Internet: <https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html>.

CORDEIRO, W. L. da C.; MARQUES, J. A.; GASPARY, L. P. Data plane programmability beyond openflow: Opportunities and challenges for network and service operations and management. **Journal of Network and Systems Management**, Springer, v. 25, n. 4, p. 784–818, 2017.

CUGINI, F. et al. P4 in-band telemetry (int) for latency-aware vnf in metro networks. In: OPTICAL SOCIETY OF AMERICA. **Optical Fiber Communication Conference**. [S.l.], 2019. p. M3Z–6.

DACIER, M. C. et al. Security challenges and opportunities of software-defined networking. **IEEE Security & Privacy**, IEEE, v. 15, n. 2, p. 96–100, 2017.

DPDK Project. **Data Plane Development Kit**. 2020. Retrieved June 29, 2020 from https://community.arubanetworks.com/t5/SDN-Apps/ct-p/SDN-Apps.

DRIDI, L.; ZHANI, M. F. Sdn-guard: Dos attacks mitigation in sdn networks. In: IEEE. **2016 5th IEEE International Conference on Cloud Networking (Cloudnet)**. [S.l.], 2016. p. 212–217.

GAO, J. et al. Lyra: A cross-platform language and compiler for data plane programming on heterogeneous asics. In: **ACM SIGCOMM 2020**. New York, NY, USA: Association for Computing Machinery, 2020. (SIGCOMM ’20), p. 435–450. ISBN 9781450379557. Available from Internet: <https://doi.org/10.1145/3387514.3405879>.

GROUP, T. P. A. W. **P4Runtime Specification**. [S.l.], 2019. Available from Internet: <https://p4.org/p4runtime/spec/v1.0.0/P4Runtime-Spec.pdf>.

GRPC. **Introduction to gRPC**. [S.l.], 2020. Available from Internet: <https://grpc.io/docs/what-is-grpc/introduction/>.

HANCOCK, D.; MERWE, J. Van der. Hyper4: Using p4 to virtualize the programmable data plane. In: **Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies**. [S.l.: s.n.], 2016. p. 35–49.

JAIN, R.; PAUL, S. Network virtualization and software defined networking for cloud computing: a survey. **IEEE Communications Magazine**, IEEE, v. 51, n. 11, p. 24–31, 2013.

JIN, X. et al. Covisor: A compositional hypervisor for software-defined networks. In: **USENIX NSDI 15**. Oakland, CA: USENIX Association, 2015. p. 87–101. ISBN 978-1-931971-218.

KANDOI, R.; ANTIKAINEN, M. Denial-of-service attacks in openflow sdn networks. In: IEEE. **2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)**. [S.l.], 2015. p. 1322–1326.

KOPONEN, T. et al. Network virtualization in multi-tenant datacenters. In: **USENIX NSDI 14**. Seattle, WA: USENIX Association, 2014. p. 203–216. ISBN 978-1-931971-09-6.

KOPONEN, T. et al. Network virtualization in multi-tenant datacenters. In: **11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)**. [S.l.: s.n.], 2014. p. 203–216.

KRUDE, J. et al. Online reprogrammable multi tenant switches. In: **ACM CoNEXT Workshop on ENCP**. New York, NY, USA: ACM, 2019. p. 1–8. ISBN 9781450370004.

KUMAR, P. et al. Picnic: Predictable virtualized nic. In: **Proceedings of the 2019 ACM SIGCOMM**. New York, NY, USA: ACM, 2019. p. 351–366. ISBN 9781450359566.

LAPADULA, L.; LAPADULA, L. J.; BELL, D. E. Secure computer systems: A mathematical model. Citeseer, 1996.

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

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

MARTINEZ-YELMO, I. et al. Arp-p4: A hybrid arp-path/p4runtime switch. In: IEEE. **2018 IEEE 26th International Conference on Network Protocols (ICNP)**. [S.l.], 2018. p. 438–439.

MCCAULEY, J. et al. Enabling a permanent revolution in internet architecture. In: **Proceedings of the ACM Special Interest Group on Data Communication**. [S.l.: s.n.], 2019. p. 1–14.

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

MISSIER, P.; BELHAJJAME, K.; CHENEY, J. The w3c prov family of specifications for modelling provenance metadata. In: **Proceedings of the 16th International Conference on Extending Database Technology**. New York, NY, USA: ACM, 2013. p. 773–776.

ONF. **SDN Architecture Overview**. [S.l.], 2013. Available from Internet: <https://opennetworking.org/wp-content/uploads/2013/02/SDN-architecture-overview-1.0.pdf>.

ONF. **OpenFlow Switch Specification**. [S.l.], 2015. Available from Internet: <https://opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf>.

Open Network Operating System. **Apps and Use Cases**. 2020. Retrieved July 24, 2020 from https://wiki.onosproject.org/display/ONOS/Apps+and+Use+Cases.

PFAFF, B. et al. The design and implementation of open vswitch. In: **USENIX NSDI 15**. Oakland, CA: USENIX Association, 2015. p. 117–130. ISBN 978-1-931971-218.

SAQUETTI, M. et al. Virtp4: An architecture for p4 virtualization. In: **2019 RAW workshop – Short Paper Introductions**. [S.l.]: IEEE, 2019. (RAW ’19), p. 1–4.

SAQUETTI, M. et al. P4vbox: Enabling p4-based switch virtualization. **IEEE Communications Letters**, v. 24, n. 1, p. 146–149, Jan 2020. ISSN 2373-7891.

SHAHBAZ, M. et al. Pisces: A programmable, protocol-independent software switch. In: **Proceedings of the 2016 ACM SIGCOMM**. New York, NY, USA: ACM, 2016. p. 525–538. ISBN 9781450341936.

SIMMHAN, Y. L.; PLALE, B.; GANNON, D. A survey of data provenance in e-science. **ACM Sigmod Record**, ACM New York, NY, USA, v. 34, n. 3, p. 31–36, 2005.

SONG, H. Protocol-oblivious forwarding: Unleash the power of sdn through a future-proof forwarding plane. In: **ACM SIGCOMM HotSDN 13**. New York, NY, USA: ACM, 2013. p. 127–132. ISBN 978-1-4503-2178-5.

TAN, L. et al. In-band network telemetry: A survey. **Computer Networks**, Elsevier, p. 107763, 2020.

TEAM, M. Mininet: An instant virtual network on your laptop (or other pc)-mininet. **Mininet. org**, 2017.

THAPA, B. K.; DIKICI, B.; SCHÖNWÄLDER, J. Reactive forwarding applications in onos. **no. January**, 2018.

TIRONE, M. S. P. d. C. Programmable virtual switches: design and implementation of the forwarding engine and supporting features. 2020.

TOMONORI, F. Introduction to ryu sdn framework. **Open Networking Summit**, p. 1–14, 2013.

TU, N. V.; HYUN, J.; HONG, J. W.-K. Towards onos-based sdn monitoring using in-band network telemetry. In: IEEE. **2017 19th Asia-Pacific Network Operations and Management Symposium (APNOMS)**. [S.l.], 2017. p. 76–81.

UJCICH, B. E. et al. Cross-app poisoning in software-defined networking. In: **Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security**. New York, NY, USA: ACM, 2018. (CCS ’18), p. 648–663. ISBN 9781450356930.

UJCICH, B. E. et al. Automated discovery of cross-plane event-based vulnerabilities in software-defined networking. In: **Network and Distributed System Security Symposium**. [S.l.: s.n.], 2020.

W3C. **Provenance XG Final Report**. [S.l.], 2010. Available from Internet: <https://www.w3.org/2005/Incubator/prov/XGR-prov-20101214/#What_is_provenance>.

WANG, Q. et al. You are what you do: Hunting stealthy malware via data provenance analysis. In: **Symposium on Network and Distributed System Security (NDSS)**. [S.l.: s.n.], 2020.

WIKI, O. **ONOS System Components**. [S.l.], 2016. Available from Internet: <https://wiki.onosproject.org/display/ONOS/System+Components>.

YOON, C. et al. A security-mode for carrier-grade sdn controllers. In: **Proceedings of the 33rd Annual Computer Security Applications Conference**. New York, NY, USA: ACM, 2017. (ACSAC 2017), p. 461–473. ISBN 9781450353458.

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

ZHENG, P.; BENSON, T.; HU, C. P4visor: Lightweight virtualization and composition primitives for building and testing modular programs. In: **Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies**. [S.l.: s.n.], 2018. p. 98–111.

APPENDIX A — RESUMO EXPANDIDO EM PORTUGUÊS

Nesse capítulo, apresentamos um resumo expandido da dissertação. Primeiramente, na Seção A.1 é discutida uma contextualização sobre os principais assuntos que envolvem o trabalho, junto com os desafios de pesquisa existentes e objetivos do projeto. Então, uma visão geral do PvS é demonstrada na Seção A.2, com as devidas discussões abrangendo as abstrações que satisfazem os aspectos mencionados. Posteriormente, abrangemos na Seção A.3 os experimentos durante e resultados feitos envolvendo as abstrações do PvS. Por fim, na Seção A.4, considerações finais e projeções futuras são exploradas.

A.1 Introdução

A arquitetura da internet contém várias falhas que a deixam deficiente, abrangendo problemas de design de segurança que possibilitam ataques cibernéticos e grandes barreiras em atualizações na arquitetura da internet. Então, soluções tem sido desenvolvidas para deixar a internet mais extensível, como as Redes Definidas por Software e Plano de Dados Programáveis que seguem o conceito de diminuir a “ossificação das redes”. A habilidade de ter virtualização no plano de dados segue o mesmo padrão oferecendo o uso de várias funções de composição de redes, além dos benefícios para o plano de controle em uma rede SDN na qual possibilita várias aplicações SDN fazerem o gerenciamento das abstrações virtuais dos switches. Porém, existem vários desafios para ter um gerenciamento eficaz e seguro dos switches virtuais. Ataques cibernéticos são um exemplo, em que ataques focados em plano de dados programáveis virtualizados são uma realidade, onde os mesmos podem ser afetados na integridade ou disponibilidade de instâncias virtuais. Em específico, ataques cibernéticos que fazem envenenamento entre aplicações (CAP attack) podem afetar as instâncias virtuais através de uma aplicação maliciosa que pretende envenenar uma aplicação legítiman SDN através de objetos compartilhados. Soluções do estado da arte começaram a explorar o tópico de virtualização em plano de dados programáveis, porém falham em garantir as abstrações necessárias para gerenciamento seguro das instâncias de forma independente, além de causar custos excessivos.

Nesse trabalho, defendemos a apresentação de uma solução de virtualização no plano de dados programáveis que oferece abstrações para gerenciamento seguro das instâncias virtuais dos switches pelos operadores de redes que utilizam aplicativos SDN no controlador. Para entregar uma solução desse porte, diversos desafios de pesquisa devem ser tratados, então dividimos os desafios em duas categorias: gerenciamento e segurança. O gerenciamento contém os desafios de como o design deve ser para suportar usuários que desejam gerenciar as instâncias virtuais de forma transparente com o uso de protocolos de comunicação conhecidos como P4Runtime e OpenFlow, e como fazer esse gerenciamento usando um único canal de comunicação para as diversas instâncias. Em relação a segurança, desafios de pesquisa relacionados variam de como ter garantias de evitar acesso não permitido de usuários a como correlacionar informações entre aplicações SDN e instâncias de switches virtuais.

A.2 PvS

Então, apresentamos o PvS, um sistema para rodar múltiplos switches virtuais programáveis em um contexto compartilhado em rede, que satisfaz os desafios de pesquisa propostos, oferecendo abstrações necessárias para ter as garantias necessárias de segurança, privacidade, e suporte para protocolos de comunicação, por exemplo, o P4Runtime. O PvS é dividido entre o Mecanismo de Encaminhamento (PvS Forwarding Engine) e o Mecanismo de Controle (PvS Control Engine). O foco desse trabalho é descrever e discutir o mecanismo de controle, em que ele possui duas principais abstrações: a abstração de mecanismo de controle, e a abstração do vIFC.

A abstração do Mecanismo de Controle contém as interfaces responsáveis por garantir o gerenciamento seguro com as aplicações SDN utilizando módulos que fornecem ponto de entrada para conexões, tradução das requisições para o dispositivo no plano de encaminhamento, autenticação, controle de acesso para as instâncias, e execução de comandos como leitura e escrita na tabela do switch virtual. Então, com a abstração do mecanismo de controle as aplicações SDN conseguem gerenciar os múltiplos switches virtuais utilizando P4Runtime como protocolo de comunicação e com segurança de controle de acesso para evitar possíveis ações de usuários não autorizados. Já a abstração do vIFC, tem o intuito usar proveniência de dados para garantir que não há violações nas políticas de controle de fluxo de informações existentes, possibilitando evitar ataques de envenenamento entre aplicações SDN. A abstração do vIFC contém interfaces que junto com a abstração do mecanismo de controle garantem que as aplicações têm as devidas permissões, e coletam as informações necessárias em um grafo de eventos ocorridos, qual usuários fez a atividade, e outras informações. Posteriormente, se há violação das políticas existentes no grafo, a abstração do vIFC mitiga o fluxo malicioso.

A.3 Experimentação

Para a abstração do Mecanismo de Controle, experimentos foram demonstrando a prova de conceito que aplicações SDN podem seguramente gerenciar as instâncias virtuais em uma NetFPGA SUME, na qual é possível demonstrar a inserção, leitura, e remoção de entradas na tabela dos switches virtuais. Além de poder ser utilizados diferentes tipos de controlador SDN, como, por exemplo, o controlador ONOS que tem suporte ao P4Runtime. Foi possível verificar a latência existente em pacotes I/O, e medimos os impactos operacionais que os ataques de envenenamento entre aplicações podem fazer em uma rede SDN e o tempo que necessitam para fazer efeito, demonstrando que a mitigação dos mesmos é imprescindível para o funcionamento seguro do estado de rede SDN, e com a abstração do vIFC é possível detectar esses ataques e extinguir os mesmos. Os experimentos voltados ao vIFC demonstram uma prova de conceito que é possível parar os ataques com um mínimo custo na solução de virtualização, além de demonstrar casos de cenários de aplicações amplamente utilizadas em SDN: o encaminhamento reativo, e a telemetria em banda.

A.4 Considerações Finais e Direções Futuras

As conclusões achadas foram que as abstrações criadas são extremamente importantes para o gerenciamento e a segurança de redes SDN, com demonstrações de como as aplicações podem manusear os switches virtuais de forma segura, como os ataques cibernéticos que buscam envenenar as aplicações podem afetar uma rede SDN, e como a abstração do vIFC pode eficiemente detectar as ameaças e mitigá-las. O uso das abstrações nas soluções de virtualizam representaram um custo baixo de latência para as mesmas. Em resumo, as principais contribuições apresentadas foram: (i) fornecer uma abstração de gerenciamento de switch virtuais com múltiplas aplicações SDN interagindo, e com compatibilidade com especificações de gerenciamento existentes, (ii) uma abstração do vIFC que garante o uso de controle de fluxo de informação garantindo que fluxos maliciosos que busquem envenenar aplicações legítimas de SDN não sejam possíveis, e (iii) código público de ambas abstrações disponível no GitHub que podem contribuir com trabalhos na área e em possíveis pesquisas, contento vários tutoriais de demonstração de gerenciamento dos switches, exploração dos ataques de envenenamento, e de mitigação com o vIFC.

Com esse trabalho de dissertação, avanços importantes para ter redes SDN mais seguras e gerenciáveis foram feitos. Porém, há diversas direções possíveis que podem ser exploradas, como, por exemplo, adicionar novos protocolos de comunicação com o design para aumentar a possibilidade de meios de gerenciamento, junto com possíveis controladores SDN que utilizam outros meios do que o P4Runtime. Outra direção de pesquisa é avaliar possíveis vetores de ataque que essas ameaças de envenenamento podem explorar, por exemplo, um switch malicioso que inicia o fluxo para atacar a aplicação SDN legítima. Por fim, outro tópico futuro consiste em oferecer soluções voltadas a computação forense que poderiam beneficiar-se do grafo montado na abstração do vIFC, e possibilitar por exemplo, possíveis detecções de intrusões dessas aplicações SDN pelas ações já tomadas pelas mesmas.

APPENDIX B — PVS CONTROL ENGINE TUTORIAL

In this chapter, we present the tutorial for the PvS Control Engine which the user may run SDN applications that interact directly with P4Runtime or using the ONOS controller. There are examples of SDN apps that use P4Runtime directly available in the [1]examples/sdnapps folder that you can use to create a stream channel with PvS Control Engine, authenticate the app, perform read/write table entries from/into virtual switches, process packet in/out from/to virtual switches, and read switch registers and counters. The “clientl2.py” file handles the L2 switch; the “clientrouter.py” file handles the Router switch; and the “clientint.py” file handles the switch that does in-band telemetry. Also, there are SDN apps that can be used with the ONOS controller, which the network operator may interact with the L2 switch, performing write requests, packet I/O, and other stream message requests.

B.1 SDN Apps running direcly with P4Runtime

Firstly, install Python, pip and Git and clone this repository:

```
sudo apt install -y python python-pip git
git clone https://github.com/guilherme6041/
p4runtime-grpc-v3.git
```

Then, go to the project’s root directory, make the script executable and run it:

```
cd p4runtime-grpc-v3
chmod +x ./scripts/install_p4runtime.sh
./scripts/install_p4runtime.sh
```

To run the PvS server, issue the following commands:

```
chmod +x ./run_p4runtime_server.sh
sudo ./run_p4runtime_server.sh
```

Run the SDN apps:

```
export PYTHONPATH=.
python examples/sdn_apps/client_[switch].py
```

[1]https://github.com/ComputerNetworks-UFRGS/p4runtime-grpc-v3

B.2 SDN Apps running with ONOS

Firstly, install the ONOS controller as is defined in the developer quick section. [2]

In the main directory example [3], make sure the folders (protobuffs, and protocols/p4runtime) in pvs−onos/onos-drvs/ are set correctly in ONOS respectively:

```
onos/
onos/protocols/
```

Starting the tutorial, execute ONOS in the first terminal:

```
bazel build onos
bazel run onos-local -- clean debug
```

On the terminal 2, enter the ONOS CLI:

```
ssh -p 8101 karaf@localhost
```

Deactivate the link provider, and start the bmv2 driver:

```
onos> app deactivate org.onosproject.lldpprovider
onos> app activate org.onosproject.drivers.bmv2
```

Now, we activate the L2 example Pipeconf:

```
onos> app activate org.onosproject.p4tutorial.pipeconf
```

On the terminal 3, we're instantiating the server:

```
./run_p4runtime_server.sh
```

On terminal 4, we're informing ONOS of the devices in the data plane.

```
onos-netcfg localhost netconfig.json
```

Perform packet I/O:

```
python example/packetin/packet_in_onos.py
```

Activate the L2 switch SDN application:

```
onos> app activate org.onosproject.p4tutorial.l2_switch
```

[2]https://wiki.onosproject.org/display/ONOS/Developer+Quick+Start
[3]https://github.com/ComputerNetworks-UFRGS/p4runtime-grpc-v3/tree/master/examples/pvs-onos

APPENDIX C — CAP ATTACKS AND VIFC TUTORIAL

In this chapter, we present the tutorial firstly for the CAP attacks, which the user may intend to send a malicious packet-out request that results in a Cross-App Poisoning (CAP) Attack. The user may explore two scenarios: the Reactive Forwarding App in which a malicious ARP reply packet is sent, and the Inband Telemetry App in which a malicious packet-out request with fake telemetry data is sent to the data plane. Then, we explain how to activate the vIFC to stop these attacks.

In the main directory [1], start with the Reactive Forwarding CAP attack. Set the system files (maliciousapp/trigger, usecase) in the following ONOS folders respectively:

```
/home/user/onos/apps
/home/user/onos/apps/p4-tutorial
```

Alter the file “modules.bzl” located at:

```
/home/user/onos/tools/build/bazel/
```

In the APPMAP of the file, add the following if they do not exist:

```
``//apps/p4-tutorial/trigger:onos-apps-p4-
tutorial-trigger-oar'': [],
``//apps/p4-tutorial/pipeconf:onos-apps-p4-
tutorial-pipeconf-oar'': [],
``//apps/p4-tutorial/int:onos-apps-p4-
tutorial-int-oar'': [],
```

Starting the tutorial, we firstly execute ONOS in terminal 1:

```
bazel build onos
bazel run onos-local -- clean debug
```

On the terminal 2, we’re going to enter ONOS CLI:

```
ssh -p 8101 karaf@localhost
```

Deactivate the link provider, and start the bmv2 driver:

```
onos> app deactivate org.onosproject.lldpprovider
onos> app activate org.onosproject.drivers.bmv2
```

[1] https://github.com/ComputerNetworks-UFRGS/p4runtime-grpc-v3/tree/master/examples/attacks

Now, we activate the L2 example Pipeconf:

```
onos> app activate org.onosproject.p4tutorial.pipeconf
```

On the terminal 3, build the L2 bmv2 exercise by emulating the Mininet:

```
cd /home/user/p4runtime-grpc-v3/examples/-
attacks/bmv2/exercises/l2/
make all
```

On the terminal 4, we're instantiating the server:

```
./run_p4runtime_server.sh
```

In the ONOS CLI, we need to activate the reactive forwarding app:

```
onos> app activate org.onosproject.fwd
```

Activate the malicious application Trigger:

```
onos> app activate org.onosproject.p4tutorial.trigger
```

For the Inband Telemetry CAP attack scenario, paste the usecase [2] (containing the INT SDN app, and the INT pipeconf) folder in the following ONOS:

```
/home/user/onos/apps/p4-tutorial/
```

Starting the tutorial, we firstly execute ONOS in terminal 1:

```
bazel build onos
bazel run onos-local -- clean debug
```

On the terminal 2, we're going to enter ONOS CLI:

```
ssh -p 8101 karaf@localhost
```

Deactivate the link provider, and start the bmv2 driver:

```
onos> app deactivate org.onosproject.lldpprovider
onos> app activate org.onosproject.drivers.bmv2
```

Now, we activate the MRI example Pipeconf:

[2]Use case: <https://github.com/ComputerNetworks-UFRGS/p4runtime-grpc-v3/tree/pvs_bmv2/examples/attacks/int_attack/use_case>

```
onos> app activate org.onosproject.p4tutorial.pipeconf
```

On the terminal 3, build the MRI bmv2 exercise by emulating the Mininet:

```
cd /home/user/p4runtime-grpc-v3/examples/-
attacks/bmv2/exercises/mri/
make all
```

On the terminal 4, we're instantiating the server:

```
./run_p4runtime_server.sh
```

In the ONOS CLI, we need to activate the reactive forwarding app:

```
onos> app activate org.onosproject.p4tutorial.int
```

Activate the malicious application Trigger:

```
onos> app activate org.onosproject.p4tutorial.trigger
```

To mitigate both these performed CAP attacks, simply turn on the solution in the PvS and following the attack tutorial for each case, which vIFC is going to detect the information flow control violation performed by the Trigger application and the CAP attack will not occur:

```
vim /home/user/p4runtime-grpc-v3/config/ServerConfig.py
vIFC = True
```

APPENDIX D — PUBLICATIONS

D.1 Journals

- **P4VBox: Enabling P4-based Switch Virtualization.** M. Saquetti, G. Bueno, W. Cordeiro, J.R. Azambuja. IEEE Communications Letters. Nov. 2019. doi: 10.1109/ LCOMM.2019.2953031

D.2 Conferences

- **Defending Lightweight Virtual Switches from Cross-App Poisoning Attacks with vIFC.** G. Bueno, M. Saquetti, W. Cordeiro, J.R. Azambuja. ACM SIGCOMM 2020 Conference Posters and Demos
- **Hard Virtualization of P4-based switches with VirtP4.** M. Saquetti, G. Bueno, W. Cordeiro, J.R. Azambuja. In proceedings of the ACM SIGCOMM 2019 Conference Posters and Demos 2019 Aug 19 (pp. 80-81). doi: 10.1145/3342280.3342314.
- **VirtP4: An Architecture for P4 Virtualization.** M. Saquetti, G. Bueno, W.Cordeiro, J.R. Azambuja. In 2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW) 2019 May 20 (pp. 75-78). IEEE. doi: 10.1109/ IPDPSW.2019.00021.

D.3 Submited, Awaiting Review

- **Defending Lightweight Virtual Switches from Cross-App Poisoning Attacks with vIFC.** G. Bueno, I. Lamb, M. Saquetti, W. Cordeiro, J.R. Azambuja. IEEE/IFIP DISSECT 2021