UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

IVAN PETER LAMB

SDN Control Plane with Information Flow Control

Work presented in partial fulfillment
of the requirements for the degree of
Bachelor in Computer Science

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

Porto Alegre
November 2021

“We can only see a short distance ahead,
but we can see plenty there that needs to be done.”
— Alan Turing

ACKNOWLEDGEMENTS

Foremost, I give my thanks to both of my parents, Ivan Lamb and Marlene Nunes, for guiding me not only to the completion of this work, but also to at its very beginning by establishing a high quality education as a goal in my life.

This work was also made possible by my teachers, specially my advisors José Rodrigo Azambuja and Weverton Cordeiro, and my colleagues Guilherme Buenno and Matheus Saquetti that supported me throughout my undergraduate jorney at UFRGS.

ABSTRACT

This work describes the development of a control plane for a Software Defined Network with an emphasis on security through Information Flow Control. The developed control plane was implemented in Python with a modular design, making it easy to read, maintain and extend, should it be necessary. A wide variety of environments, in terms of the data plane and application plane, are supported by the control plane, which was tested in multiple SDN controllers and data plane abstractions. In regards to security, a state-of-the-art solution for Information Flow Control was employed and extended to address a specific class of network integrity attacks on virtual switches.

Keywords: Software Defined Networking. Information Flow Control. Control Plane.

Plano de Controle de SDNs com Controle de Fluxo de Informação

RESUMO

Esse trabalho descreve o desenvolvimento de um plano de controle para uma Rede Definida por Software com uma ênfase em segurança através de Controle de Fluxo de Informação. O plane de controle desenvolvido foi implementado em Python com um design modular, tornando-o fácil de ler, manter e extender, caso seja necessário. Uma grande variedade de ambientes, em termos de plano de dados e plano de aplicações, são suportados pelo plano de controle, que foi testado em multiplos controladores SDN e abstrações de plano de dados. Em relação à segurança, uma solução estado-da-arte para Controle de Fluxo de Informação foi empregada e extendida para lidar com uma classe específica de ataques à integridade da rede em switches virtuais.

Palavras-chave: Redes Definidas por Software, Controle de Fluxo de Informação, Plano de Controle.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

API	Application Programming Interface
APP	Application
ARP	Address Resolution Protocol
ASIC	Application-Specific Integrated Circuit
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
DSL	Domain Specific Language
FPGA	Field-Programmable Gate Array
GUI	Graphical User Interface
IFC	Information Flow Control
IP	Internet Protocol
MAC	Media Access Control
MRI	Multi-Hop Route Inspection
NBI	Northbound Interface
NFV	Network Function Virtualization
ONF	Open Networking Foundation
ONOS	Open Networking Operating System
P4	Programming Protocol Independent Packet Processors
PDP	Programmable Data Plane
PISA	Protocol-Independent Switch Architecture

RBAC	Role-Based Access Control
RFC	Request For Comments
RPC	Remote Procedure Call
SBI	Southbound Interface
SDN	Software Defined Networking
SSL	Secure Socket Layer
TCP	Transmission Control Protocol
TTL	Time to Live
UDP	User Datagram Protocol
vIFC	Virtual Information Flow Control
VSMA	Virtual Switch Management API
VM	Virtual Machine

CONTENTS

1 INTRODUCTION

With recent advances in the computer networks area, emerging paradigms are challenging the traditional approach in terms of network architecture and device implementation. Innovative ideas made possible a multitude of services with infrastructure advancements, such as data center networks and switch virtualization. Those advancements reduce the equipment and maintenance cost for companies and improve performance for the final users.

However, control plane functionalities regarding management and security of virtual forwarding devices in the programmable data plane did not advance at the same pace, as recent vulnerabilities discovered demonstrates. Those vulnerabilities must be addressed along with other technological improvements to ensure that SDNs achieve industry quality standards.

The presented work's goal is to develop a Software Defined Networking (SDN) control plane that utilizes modern technology to achieve flexibility, such as the P4 Domain Specific Language (BOSSHART et al., 2014) and the P4 Runtime switch configuration protocol. SDNs and related technologies contrast with the traditional computer networks industry mainly in separating the control logic and the forwarding logic from the devices and creating flexibility to the implementation of novel protocols. This flexibility allows the data plane of the network to be implemented with different paradigms, be it ASIC devices, FPGA switches, software-emulated switches, etc. and also allows for multiple SDN controllers to interact simultaneously with the control plane.

Moreover, the security of the whole network is a core principle of the developed control plane, with a Role Based Access Control authentication model implemented. Additionally, a module for protecting against a novel class of attacks, named *Cross-App Poisoning*, is designed to enhance network resilience using Information Flow Control to detect malicious applications. Security on SDNs is a highly discussed topic (DACIER et al., 2017), but it still needs more time to evolve to a point where it reaches industry standards.

While flexibility is a focus of this work, the control plane was designed to be integrated with the PvS (Programmable Virtual Switches) with P4VBox as a Programmable Data Plane (PDP). PvS is a project that allows multiple virtual forwarding devices to be deployed on the same physical hardware, such as an FPGA, and that allows partial reconfiguration during runtime of said hardware. It allows multiple tenants to split the physical

hardware and manage its devices independently, reducing the individual operational cost. Tenants can also run custom applications on the SDN application plane with permissions to interact with their own devices. In the next chapters, this integration will be noticed in performance tests, and a comparison with other data planes (mainly software-emulated) will be addressed.

This work is divided as follows: Chapter 2 presents the core concepts for the understanding of this work's background, as well as previous works on the same area; Chapter 3 contains the development of the control plane, with implementation details and the technologies used, including programming languages, APIs, modules, etc.; A link to a public GitHub repository is also provided for further understanding of the implemented functionalities. Chapter 4 presents a security module of the developed control plane, vIFC, that focuses on Information Flow Control in a scenario with virtual switches, presenting efficacy and performance results in multiple test cases. Lastly, Chapter 5 concludes with a summary of the project and possible extensions and research opportunities for future works.

2 BACKGROUND

In this chapter, the main concepts for this work are presented. The key concepts for the understanding of this work's control plane and information flow control will be explained in a greater level of detail, but references are provided for all the topics mentioned. It is encouraged to make use of those references for further understanding of specific topics that may spark an interest.

2.1 Software Defined Networking

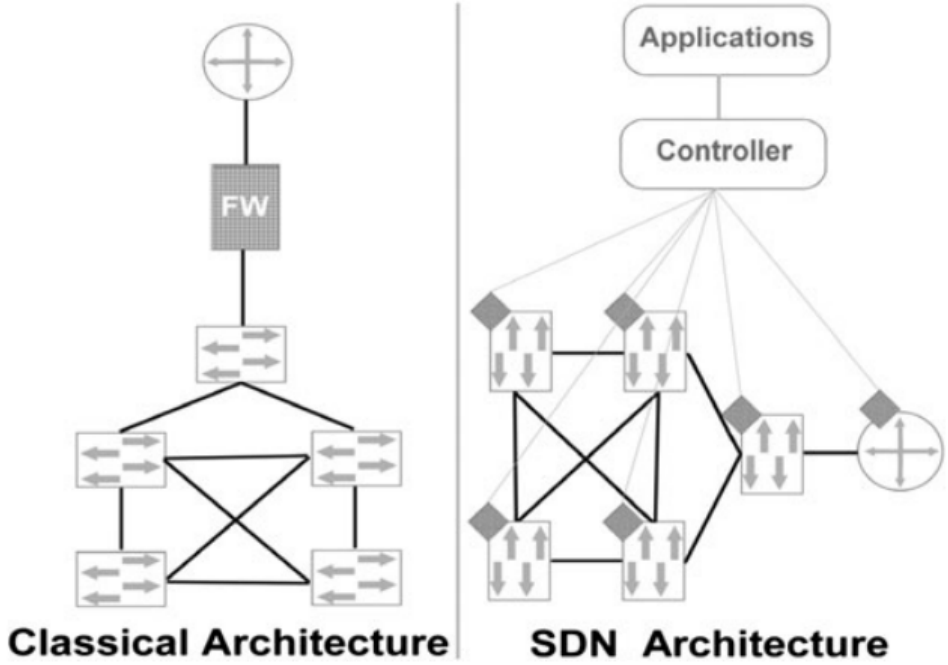
Software Defined Networking (SDN) is a flexible approach to the implementation of network systems. According to Benzekki, Fergougui and Elalaoui (2016), "SDN is an innovative approach to design, implement, and manage networks that separate the network control (control plane) and the forwarding process (data plane) for a better user experience.".

Traditionally, each individual network element (layer 2 and 3 forwarding devices such as switches and routers) implements both control and forwarding logic on its own hardware. This creates a scenario where those device's hardware's complexity increases for them to be able to implement the required network functionalities, such as routing (NETO; BEZERRA, 2002) and forwarding. This complexity increases the individual implementation and maintenance cost of each element and lowers the reliability of the network structure in case of frequent failings.

SDNs reduce the device's complexity by transferring the responsibility of routing and forwarding rules definitions of the network elements to a centralized control plane, and the data plane responsibility becomes only dealing with the traffic itself according to the control plane implemented logic. More specifically, SDNs separate this infrastructure into three layers: application plane, control plane and data plane. Figure 2.1 compares both approaches, on the left the classic approach with decentralized network elements and on the right the SDN approach with a centralized control logic layer.

For the purpose of establishing how the three layers of an SDN network communicate between them, protocols such as OpenFlow (MCKEOWN et al., 2008) emerged and became widely adopted. Those protocols are generally maintained by organizations such as the Open Networking Foundation, which specifies the OpenFlow protocol (OPEN-NETWORK-FOUNDATION, 2015).

Figure 2.1: Classic network and SDN comparison.



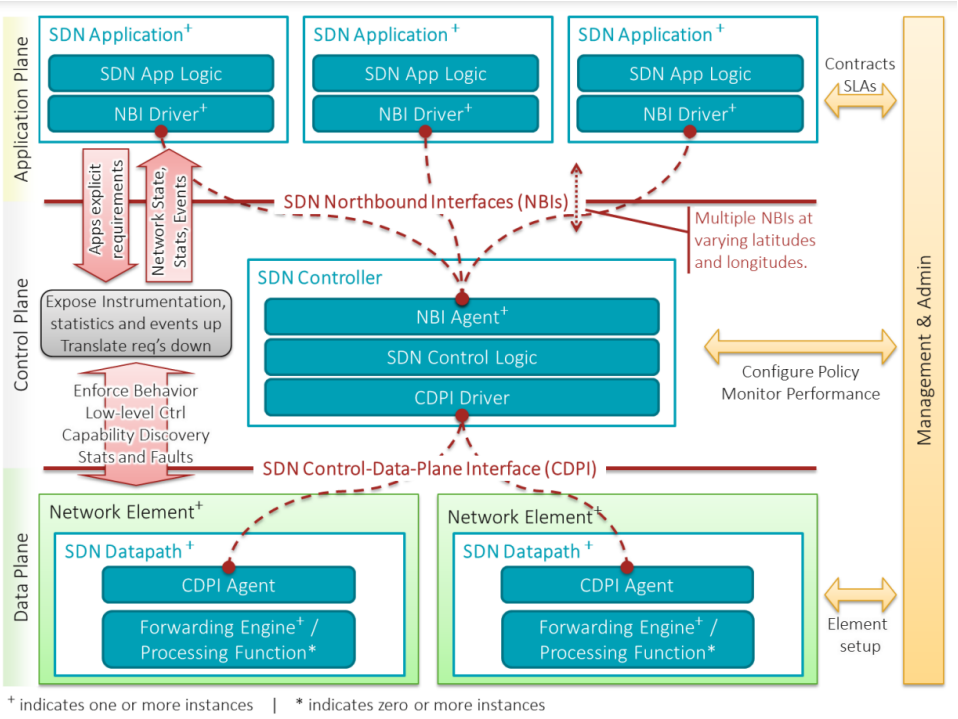
Source: (BENZEKKI; FERGOUGUI; ELALAOUI, 2016)

The Open Networking Foundation also provides an overview of the SDN architecture (OPEN-NETWORK-FOUNDATION, 2013), as seen in Figure 2.2. From top to bottom: (i) the application plane consists of applications, which are programs that communicate with the SDN through an interface with the control plane called "Northbound-interface" (NBI), receiving information about the network state and performing requests according to its own logic; (ii) the controller is the centralized entity[1] that translates the requirements from the SDN Application layer to the SDN data plane and provides the applications with a logical view of the network. In summary, it consists of interfaces with the application layer (NBIs), the control logic, and an interface with the data plane, called Control-Data-Plane Interface (CDPI); (iii) the data plane, where the network elements are located. As mentioned before, those devices are "simpler" than classical network elements, being responsible for the actual processing of packet traffic instead of the network routing logic, which they receive from the control plane through the CDPI Agent. Having a uniform CDPI Agent on each device provides a powerful abstraction tool since the control plane does not have to know the exact specifications of each device (for instance, in Figure 2.2, the network elements might have been different devices from different manufacturers, but the CDPI Agent is uniform).

[1]The abstraction of the control layer as a centralized entity does not exclude the possibility of multiple controllers running on the control plane of the network. The implementation of this work, for instance, considers that multiple controllers may be running on the control plane.

The Management & Admin entity depicted in Figure 2.2 is responsible for business management between the network provider and clients (tenants), such as deploying the devices and implementing authentication. While this work's main focus is the implementation of the control plane, the Management & Admin is also implemented to provide an interface to establish user authentication and device deployment.

Figure 2.2: SDN Architecture Overview.



Source: (OPEN-NETWORK-FOUNDATION, 2013)

2.1.1 Programmable Data Planes

Programmable Data Planes (PDPs) are a key concept of SDNs for this work. While protocols such as OpenFlow (MCKEOWN et al., 2008) established the foundation of the communication between SDN layers, it also imposed a limitation on the data plane, since these protocol specifications supported a small set of header protocols (BOSSHART et al., 2014). Therefore, innovative ideas in terms of protocols and headers would be only supported through OpenFlow specification updates. This limitation created a collective effort to develop alternatives in terms of flexibility, such as PDPs.

The goal of PDPs is to allow network operators to define an arbitrary number of packet header fields to be matched and to define the actions to be performed when those fields are matched. For that end, Domain Specific Languages (DSLs) were introduced as

a standard way of describing the packet processing logic and to allow data plane programming. Those languages allow the specification of header fields, match table definitions, implementation of actions performed, checksum verification, security protocols, etc. One such DSL is P4 (BOSSHART et al., 2014).

2.1.2 P4 Domain Specific Language

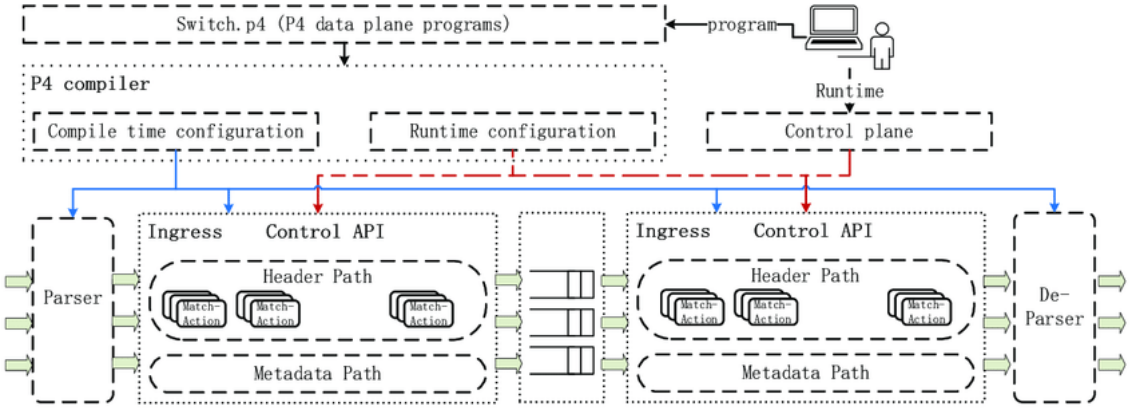
The P4 language (Programming Protocol Independent Packet Processors) is a high-level language to implement PDPs (BOSSHART et al., 2014). The motivation for the creation of P4 was to increase the flexibility of manufactures to create and implement new network headers and protocols, a scenario which the language creators described as "showing no signs of stopping", citing NVGRE, VXLAN and STT as new packet encapsulation advances of the time. More specifically, the design principles of the P4 language are, citing Bosshart et al. (2014):

- **Reconfigurability.** The controller should be able to re-define the packet parsing and processing in the field.
- **Protocol independence.** The switch should not be tied to specific packet formats. Instead, the controller should be able to specify (i) a packet parser for extracting header fields with particular names and types and (ii) a collection of typed match+action tables that process these headers.
- **Target independence.** Just as a C programmer does not need to know the specifics of the underlying CPU, the controller programmer should not need to know the details of the underlying switch. Instead, a compiler should take the switch's capabilities into account when turning a target-independent description (written in P4) into a target-dependent program (used to configure the switch).

The **Target independence** principle is an interesting one in particular since it allows this work's control plane to manage a wide array of switch implementations. Because of this flexibility, ASIC based devices, an FPGA switch implementation (SAQUETTI; BUENNO; AZAMBUJA, 2020), a software-emulated switch (MININET, 2017), etc. are all supported in the data plane of this work's control plane implementation.

Figure 2.3 shows an example of a P4-defined protocol-independent switch architecture (HANG et al., 2019). In the switch, the parser and deparser are configured to process user-defined packet headers. The ingress and egress pipelines implement match-

Figure 2.3: The architecture of a protocol-independent switch architecture (PISA).



Source: (HANG et al., 2019)

action tables to process the packets in arbitrary user-defined stages, matching the packet headers with rules to perform the corresponding actions[2]. Those actions use language primitives to modify the packet's metadata and headers.

It is also depicted on Figure 2.3 the process to deploy the source code (Switch.p4) to the switch, which involves the compilation (P4 Compiler) and configuration (blue lines). The runtime configuration (red lines) allows the Control Plane to insert match-action rules on the switches after the initial deployment. Section 2.2 describes runtime configuration of the switches in more detail.

From the programmer's point-of-view, a P4 program consists of:

- **Headers:** Headers define the structure of a set of fields, including width and constraints on the values of those fields. Figure 2.4 shows an example of an Ethernet header definition.
- **Parsers:** A definition of a parser specifies how to identify headers and valid header sequences within packets.
- **Tables:** The main mechanism for packet processing. Match-action tables apply match actions on the packet's defined headers and perform the specified action for the match.
- **Actions:** Complex actions can be built from the P4 protocol-independent primitives. Those actions are available within match-action tables.
- **Control Programs:** The control program determines the order of match-action tables to be applied to the packets. It is a simple imperative program to describe this flow of control.

[2]Actions may be decreasing TTL, calculating checksum, dropping invalid packets, sending the packet to the control plane, etc.

Figure 2.4: Example of an Ethernet header definition on the P4 language.

```
header ethernet {
    fields {
        dst_addr : 48; // width in bits
        src_addr : 48;
        ethertype : 16;
    }
}
```

Source: (BOSSHART et al., 2014)

2.2 P4 Runtime

P4 Runtime is a framework for the control / application planes of a SDN to manage devices in the data plane that are defined by a P4 program. It inherits the abstraction advantages of having a common interface with the devices, regardless if they are ASICs, FPGA based, Software based, etc. allowing it to control multiple types of switches.

As McKeown (2017) explains: "In the past, switch chips were controlled by closed, fixed and proprietary APIs. A fixed API written to the target chip covered the needs, and there was little or no need to extend the API over time." An inflexible API disallows innovation from network operators in the implementation of new headers and protocols.

Before P4 Runtime, OpenFlow (MCKEOWN et al., 2008) was created to provide certain vendor-independent flexibility, as discussed in Section 2.1.1. However, OpenFlow was never designed to be extended, it was only a standard way to implement operations on protocols used at the time. P4 Runtime, on the other hand, is by design protocol independent. For this work, the control plane manages the switches with P4 Runtime requests it receives from the application plane, inheriting great flexibility to the data plane.

The P4 Runtime specification provided by the Open Networking Foundation (ONF, 2020) provides a set of requests the applications may send to the control plane to interact with the data plane. Those requests are implemented with Remote Procedure Calls (explained on Section 2.3) to a P4 Runtime Server running on the control plane. Several methods are defined on the specification, but the most important ones for this work's control plane are:

- **Stream Channel.** Creates a bi-directional connection between an application (client) and the control plane (server). This connection is used to receive / send packets to

the devices[3] and to perform any other required communication with the control plane (on this work, for instance, Stream Channel is used to perform application authentication, granting permissions to perform read / write actions on the data plane).

- **Write.** Writes data to a P4 entity on a certain device. P4 entities may be match-action tables, counters or registers. The data written may be of types UPDATE, DELETE or MODIFY.
- **Read.** Retrieves information about a P4 entity on a certain device. With this RPC it is possible to read table entries and the corresponding actions or counters and registers on the device.

2.3 Remote Procedure Calls and gRPC

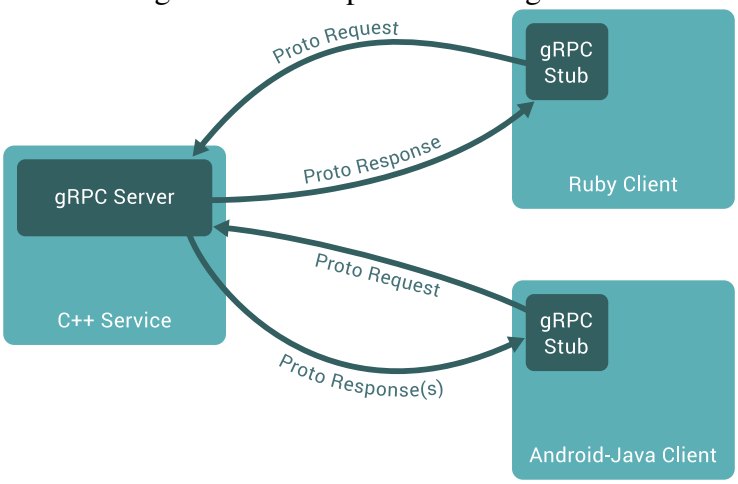
Remote Procedure Calls (RPC) is a protocol to implement communication between two processes, and its main concept is that a process can call procedures (methods or functions) on the other process (which may be running on another host). The protocol is defined in Request For Comments (RFC) 1831 (SRINIVASAN, August 1995). A phrase from the RFC describes this main concept: "The caller process first sends a call message to the server process and waits (blocks) for a reply message. The call message includes the procedure's parameters, and the reply message includes the procedure's results".

One of the uses of RPCs is to implement a client-server model, where all the message exchanges between processes happen through the defined procedure calls. The protocol itself does not address some of the classical network problems, such as reliable data transfer and network latency, but most concrete implementations are developed with those issues in mind.

gRPC is a framework to implement RPCs (GRPC, 2021) available in multiple programming languages, allowing the programmer to define a service and specify the procedures that can be called by the clients and the server. It implements the connection between the clients and the server with the TCP transport layer protocol, offering reliable data transfer, and also implements transport layer security with SSL. Figure 2.5 illustrates RPCs with gRPC, using clients and server programs implemented in multiple programming languages.

[3] Applications may send and receive data packets from the devices. A "Packet-In" is a packet sent by a data plane device to an application plane app, and a "Packet-Out" is a packet sent by an app to a device.

Figure 2.5: RPC protocol with gRPC.



Source: gRPC Documentation (GRPC, 2021).

Citing the gRPC documentation (GRPC, 2021), four types of RPCs are available for the programmer to define with gRPC:

- **Unary:** Where the client sends a single request to the server and gets a single response back, just like a normal function call.
- **Server Streaming:** Where the client sends a request to the server and gets a stream to read a sequence of messages back. The client reads from the returned stream until there are no more messages. gRPC guarantees message ordering within an individual RPC call.
- **Client Streaming:** Where the client writes a sequence of messages and sends them to the server, again using a provided stream. Once the client has finished writing the messages, it waits for the server to read them and return its response. Again gRPC guarantees message ordering within an individual RPC call.
- **Bidirectional Streaming:** Where both sides send a sequence of messages using a read-write stream. The two streams operate independently, so clients and servers can read and write in whatever order they like: for example, the server could wait to receive all the client messages before writing its responses, or it could alternately read a message then write a message, or some other combination of reads and writes. The order of messages in each stream is preserved.

Of the four RPC types, the most important for this work are the **Unary** and **Bidirectional** ones. The unary RPCs will be used by the control plane to implement the **Write** and **Read** P4 Runtime requests, and the bidirectional RPCs will be used to implement the **Stream Channel** connection, which is described in Section 2.2.

2.4 Switch Virtualization

Virtualization is a technique to split a resource (in most cases, hardware) of a host between multiple tenants (guests). It is a broadly discussed topic by many authors. TANAEMBAUM; BOS (2014), for instance, lists some of the advantages of Virtual Machine[4] virtualization, such as service isolation to increase fault tolerance without incurring additional costs of independent hardware. An entity called "hypervisor" is responsible for this division of the host's resources between the guests.

The computer networks area has also seen advances in regards to virtualization recently. Network Function Virtualization (NFV), for instance, is an emergent technology to run Internet's core high-volume packet-processing functions on commodity hardware through virtualization (JOSH; BENSON, 2016). For this work, the most relevant form of network virtualization involves switch virtualization on the data plane.

With the virtualization of programmable data planes, many forwarding devices may be deployed on a physical substrate (SAQUETTI; BUENNO; AZAMBUJA, 2020), performing its defined functions independently of each other and allowing more efficient use of hardware, which reduces the overall network cost. PDP virtualization is not limited to hardware resource sharing. Research and development of emulation of P4 programs with a general-purpose program as in Hyper4 (HANCOCK; MERWE, 2016) and HyperV (ZHANG et al., 2017), and composition of several instances of P4 programs in a single program as in P4Visor (ZHENG; BENSON; HU, 2018) are also active topics.

2.5 Mininet

Mininet (MININET, 2017) is a tool to create a software-emulated network that can run custom switch application code, such as a P4 compiled program, on a single machine or VM. It is intended to be easy to use for prototyping, development, teaching and research; situations where high-level performance is not a requirement.

It is specially useful for running P4 switches and using P4 Runtime in a controlled environment, therefore it is one of the most important development tools in this work. Many examples on Chapters 3 and 4 are executed with Mininet managing the data plane of the SDN.

[4] One of the most common type of virtualization employed are Virtual Machines, in which many logical hosts may operate sharing the same physical hardware.

2.6 CAP Attacks

Cross-App Poisoning (CAP) has been recently identified as a critical class of control plane integrity attacks in SDNs (DACIER et al., 2017). Ujcich et al. (UJCICH et al., 2018) described in detail the workflow of the attack, which involves installing a malicious application (app) on the SDN controller (that could be downloaded from a repository or public SDN app marketplace (Aruba, 2021; ONOS, 2021)). The malicious app's goal is to cause legitimate apps to perform actions in the control and data plane that the malicious app itself cannot perform due to insufficient permissions, such as writing flow rules to specific device's tables. To that end, the malicious app may write control data on shared objects among the other apps in memory, which causes legitimate apps to perform the desired actions when consumed (similar to the classic *confused deputy* problem) (HARDY, 1988).

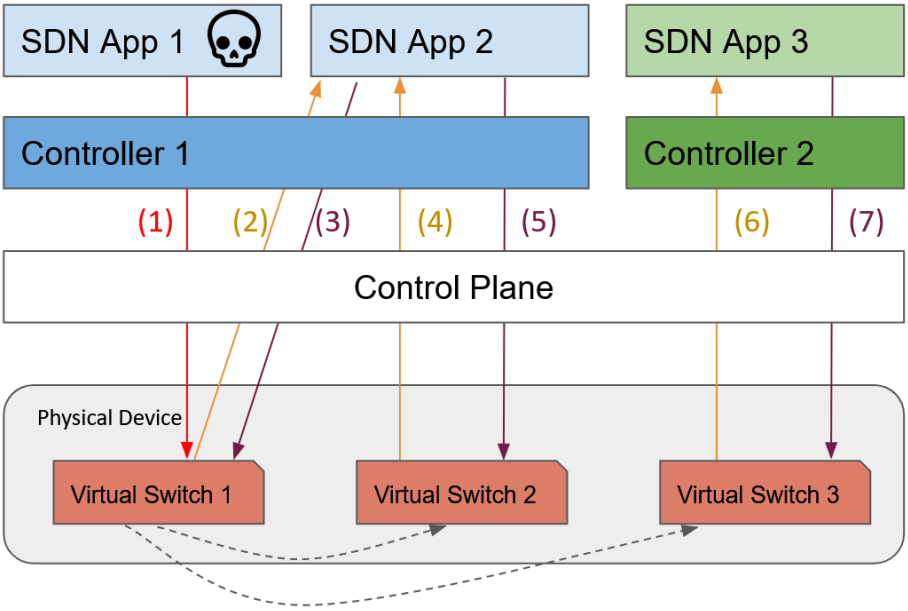
It was demonstrated that Role-Based Access Control (RBAC) solutions alone are insufficient to prevent such attacks, as they do not track information flow or enforce information flow control (IFC). The use of data provenance and a data provenance graph (a core concept in the prevention of those attacks), was also proposed by Ujcich (UJCICH et al., 2018) as ProvSDN, a solution that intercepts app requests that violate predefined IFC policies.

The main limitation of ProvSDN is that it 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; SAQUETTI; BUENNO; AZAMBUJA, 2020) dramatically changed that scenario. In summary, a physical switch may either run a composition of pipelines of match-action stages from distinct programs or emulate virtual switches through a general-purpose switch program; in both cases, each virtual switch instance may be managed independently, by multiple tenants, through different controllers. Under such scenario, a single controller can no longer maintain a complete view of the information flow between apps and virtual switches. Consequently, apps that have control of a single switch may poison every virtual switch composed or emulated in the target switch.

Preventing CAP attacks that explore switch virtualization is challenging. It involves tracking the packet flow between multiple virtual switches and detect violations of the IFC policies on those packet flows.

2.6.1 CAP Attack Example

Figure 2.6: CAP Attack example.



Source: The Author

Figure 2.6 illustrates an example of a conceptual CAP attack. In this scenario, suppose the SDN applications 1, 2 and 3 are installed and running on the control plane after being downloaded from a public SDN repository, such as Aruba Networks (Aruba, 2021) and that Apps 1 and 2 are managed by Controller 1 and App 3 is managed by a different one, Controller 2. Controller 1 manages virtual switches 1 and 2 and Controller 2 manages virtual switch 3.

However, SDN App 1 is a malicious application that intends to explore vulnerabilities in the control plane to launch a CAP attack on the network. The seeming purpose of this application is to monitor packets received from the data plane through *Packet-In* events (packets flowing from the devices to the applications) and create diagnostic information of the network to the system administrator. With that in mind, the administrator gives this application only the `PACKET_EVENT` permission, which allows it to send and receive packets from the switches. The obscure purpose of SDN App 1 is to manipulate the forwarding tables on virtual switches to cause disruption on the network (or to forward data flows to an attacker's controlled computer). To perform those malicious actions, SDN App 1 would need the `FLOWRULE_WRITE` permission. Other applications, such as SDN App 2 and 3 have both `PACKET_EVENT` and `FLOWRULE_WRITE` permissions, so they can receive packets from the data plane and also modify the forwarding

tables on switches.

The CAP attack starts with a fabricated packet from the malicious application. Since SDN App 1 has the `PACKET_EVENT` permission it can create *Packet-Out* messages (a packet from the application to the device) for the switches managed by the SDN controller in which it is installed. In this case it sends this message to Virtual Switch 1 (flow 1 on Figure 2.6). The packet's source and destinations addresses are fabricated to cause a *table miss* on the switch, that occurs when the switch table has no entry that matches the inbound packet's destination. One possible action that a switch may take to deal with these *table misses* is to send a *Packet-In* message to the controller. SDN App 2 may be registered to receive such *Packet-In* messages from Virtual Switch 1, therefore receiving the message (flow 2). In response, the legitimate application may reconfigure the forwarding table that caused the *table miss* on Virtual Switch 1, installing a malicious flow rule with fabricated source and destination addresses (flow 3).

ProvSDN (UJCICH et al., 2018) can act on this scenario ensuring that the IFC policies are respected. In that case in particular, if ProvSDN was installed and running on Controller 1, it would notice that information (the fabricated data packet) is flowing from a least privileged application (SDN App 1) to a more privileged application (SDN App 2). Because of this, ProvSDN will block either flow 2 (the *Packet-In*) or flow 3 (the *flowrule write*), depending on the control policy and preserve the integrity of the forwarding tables on Virtual Switch 1. Although ProvSDN is effective in this scenario, it cannot maintain a complete view of the information flow between virtual switches. It may be the case that the fabricated packet is sent to Virtual Switch 2 from Virtual Switch 1 in an *out-of-band* flow (dashed flow in the figure) through a logical link between the virtual switches[5]. In this case, Virtual Switch 2 could be the one that sends the packet through a *Packet-In* event to SDN App 2 (flow 4), which may trigger the table reconfiguration of the previous scenario. In this case, the attack will be successful, since ProvSDN will not be able to trace the packet as being created by SDN App 1, since it considers Virtual Switch 1 and 2 to be distinct entities, although they are virtual switches executing in the same physical device. As a consequence, applications with permissions on a single virtual switch may poison each virtual switch composed/emulated on the target switch. Furthermore, ProvSDN does not prevent a CAP attack that a malicious application may launch against another application running on a different controller (flow 6 and flow 7) through similar methods.

[5]This flow may occur, for example, if Virtual Switch 1 is a top-level switch in a load-balancing architecture, and simply forwards every package to lower levels.

The implementation of this work's control plane contains a module that expands on the foundation of ProvSDN's data provenance analysis to deal with those challenging scenarios. It is capable of detecting the previously mentioned *out-of-band* flows, and to trace the source of the information to the malicious application. The module is described in greater detail on Chapter 4.

2.6.2 Other Poisoning Attacks

While (UJCICH et al., 2018), to the best of my knowledge, is the only discussion of *Cross-App Poisoning* attacks, many similar poisoning attacks can be launched against SDNs. (SATTOLO et al., 2019) classifies poisoning attacks (including CAP attacks) and explains the subtle differences between them. This survey also classifies each type of attack based on its outcomes, such as a *Flowrule Change* for CAP attacks or a *Data Leakage* for Host Hijacking.

The main differences between CAP attacks and the other attacks cited on that survey is that (i) they are initiated by a malicious application and (ii) they operate through information flow. Thus, a solution to deal with this type of attack must enforce Information Flow Control between the application plane and the data plane.

3 CONTROL PLANE DEVELOPMENT

This chapter presents in detail the implementation of the control plane. The control plane was implemented in Python, with a modular design to separate its functionalities. The source code for the whole project is available on GitHub[1], and Python is an easy-to-read programming language, therefore, it is encouraged to refer to the source code for a better understanding of specific implementation details, if some part of the explanation leaves gaps to be filled.

Figure 3.1 provides an overview of the control engine interacting with other SDN elements on PvS (SAQUETTI; BUENNO; AZAMBUJA, 2020). PvS is a project that allows for multiple tenants to deploy virtual switches on the same physical hardware, such as an FPGA, providing runtime partial reconfiguration of this hardware, being able to deploy / remove devices without interfering with others that are already deployed. This work's control plane is used in this bigger project[2] as the control engine that provides an interface between the applications, data plane, and management. Figure 3.1 also depicts the control flow that applications use to interact with the data plane (through the P4 Runtime Interface) and the control flow that the Management & Admin uses to interact with the data plane and update the *switch config* database and *auth info* database through the Command-Line Interface (CLI).

Section 3.1 discusses the main components of the system and how the technologies presented in Chapter 2 were used to implement those components. Section 3.2 contains a brief setup guide and a test case sample for the system on a Mininet software-emulated data plane and a simple app running on the application plane, if it is desired to execute the project.

3.1 Main Components

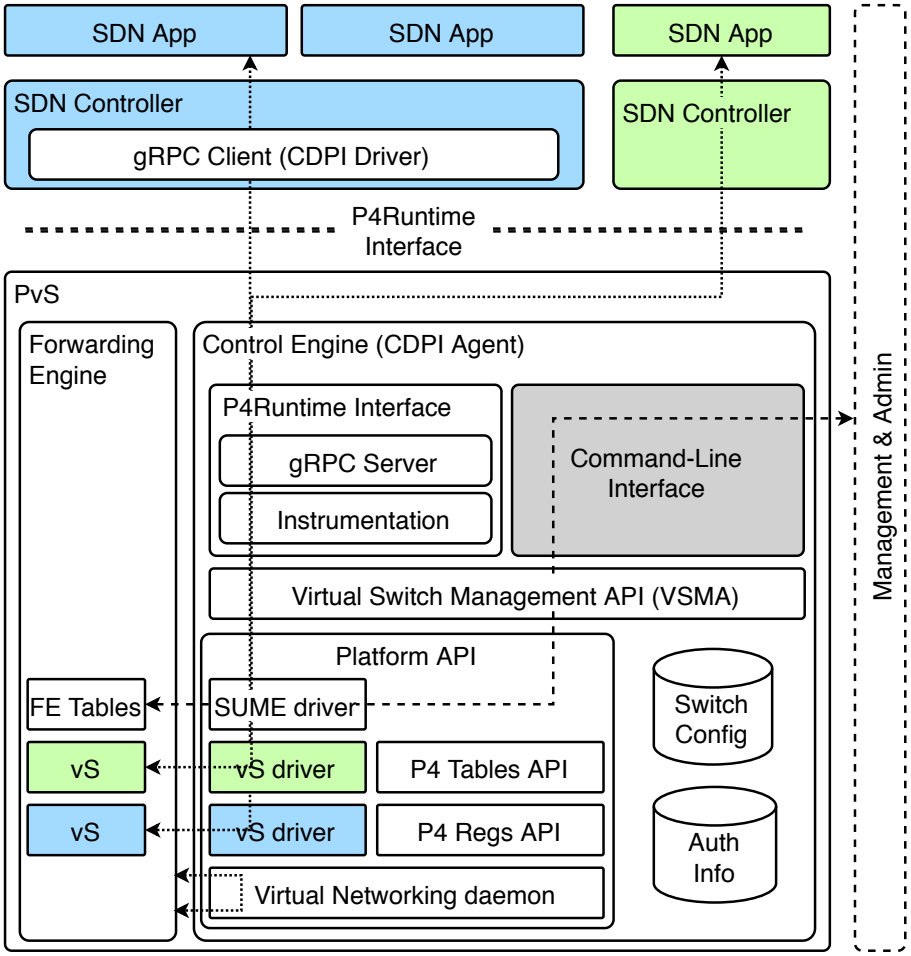
3.1.1 Server

The server script (located in *server/grpc_server.py*) is the entry point of the control plane. It starts by opening a log for debugging, loading the devices and users databases

[1] Link to the repository: <https://github.com/Ivanatorion/p4runtime-grpc-tcc>

[2] As mentioned before, the developed control plane is not limited to PvS. It can be deployed with other data plane abstractions.

Figure 3.1: Control Engine interacting with other PvS SDN elements.



Source: The Author

to the memory, creating a thread to receive *Packet-Ins* from the data plane, and starting a gRPC server to receive P4 runtime requests.

Most server parameters are configurable (on the *config/ServerConfig.py* file), such as the server port, SSL certificates, *Packet-In* / *Packet-Out* interfaces, etc. Plugins of additional modules, vIFC (discussed in Chapter 4), for instance, can also be enabled / disabled in this configuration file.

3.1.2 RPC Management

After a client connects to the server, the message exchanges are made through RPCs through the P4Runtime Inteface, as seen in Figure 3.1. The server's RPC methods, such as **Stream Channel**, **Write Request**, **Read Request** and other additional methods for device arbitration defined in the P4 runtime specification, can be called to interact directly with the data plane devices, provided that the application has the required per-

missions on the device it is communicating with.

The first message a client must send to the server upon connecting is a call to open a **Stream Channel** so that it can send an authentication message. Other RPCs cannot be called by the client until it has sent a valid authentication message with its user credentials (username and password), because the RBAC model must know if the client has permission to perform the specific requests.

The *Virtual Switch Management API* (VSMA) depicted in Figure 3.1 is responsible for converting the P4 Runtime requests that require interaction with the virtual devices (such as **Read** and **Write** requests) to function calls of the specific device. For the PvS example, those function calls are provided by the FPGA's Platform API. The system administrator can also interact with the VSMA directly through a Command Line Interface.

The *Packet-In* and *Packet-Out* operations (receiving and sending data plane packets) are also implemented on the **Stream Channel** method. The *Packet-In* is implemented by creating a packet buffer for each connection and storing the packets each client is allowed to receive according to the RBAC model. The *Packet-Out* is implemented with a socket to an interface to which the packets are forwarded along with metadata about the switch ID and ingress port, which are used on the data plane to determine the device and port that will receive the packet.

3.1.3 Database Model

A SQL database is used to store the necessary information about the forwarding devices such as table fields and register and to store user information and permissions. The database is also used to store vIFC's (Chapter 4) policies. The database contains the following tables:

- **Switches:** Contains information about the data plane devices[3], such as name, ID, API paths and management addresses.
- **Switch Tables:** Contains information of the forwarding tables on each device, including base addresses and match types.
- **Switch Registers:** Registers that are available to read / write on each device, which might serve any purpose (e.g. ingress queue size).

[3]The term "switch" is usually used to refer to a layer 2 device, but for this work it can represent any device, such as a layer 3 router.

- **Table match fields - Table actions - Table action fields:** Those three tables are used to register information of the tables required to perform P4 Runtime requests, most importantly the bit length of each field.
- **Users:** This table stores the users registered in the system. It is a simple table containing the username and password of each user.
- **Permissions:** This tables stores the permissions that each user has on each device. The permissions are defined on the *config/PermEnum.py* file. One of the design principles of the control plane is access control through this permission system, which gives control to the system administrator over each deployed data plane device. On PvS, for instance, permissions for each device may be given only to the tenants of that device.
- **Policies:** Stores the system-administrator defined policies for vIFC. These policies are better explained on Chapter 4, which addresses security functionalities of the control plane.

The methods for interacting with the database, which manages concurrent access to the database through mutexes, are also defined in files on this folder. For performance reasons every time an entry is queried from some of the database tables it is stored in a buffer in memory, which reduces disk access. If the network's complexity is high (in terms of number of forwarding elements), it might be required to disable this buffering functionality.

3.1.4 Switch Modules Management

The developed control plane can interact with multiple types of switches through P4 Runtime, but each device has its peculiarities when it comes to calling the read and write methods. PvS switches, for instance, run on a NetFPGA-SUME board, which comes with C APIs that implement the P4 Runtime calls, while Mininet switches implement them on multiple programming language modules (such as Python classes).

The GitHub repository contains examples of those APIs. The *bmv2_module* folder contains the required method definitions for Mininet switches and the *platform_api* contains the NetFPGA-SUME interface for the C function calls, along with sample P4 programs that implement simple layer 2 and layer 3 forwarding devices, operating with Ethernet and IP protocols.

The Switch Modules Management (file *utils/SM_mgmt.py*) is responsible for loading those modules for each device, so that when the control plane receives a RPC call (on the RPC Management class) it knows how to translate it to an actual P4 Runtime call that is sent to the data plane. Those modules are always loaded on memory when the system initiates or when a new device is registered in the system by the network administrator through the Management & Admin CLI depicted in Figure 3.1.

3.2 Installation and Test App Sample

3.2.1 Dependencies

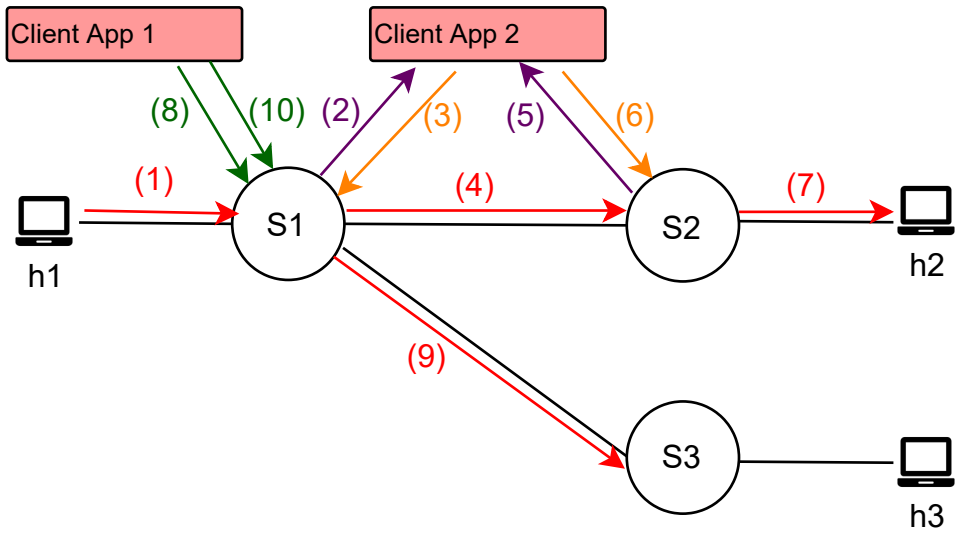
There are many dependencies required to execute the project: gRPC, bmv2, P4 compiler, etc. For that reason, a bash script is provided to install all those dependencies in *scripts/install_dependencies.sh*. It is encouraged to create a Virtual Machine with Ubuntu 16.04 (which is the system version in which the control plane was developed, and it is verified to run properly) and execute the script for a clean installation of the dependencies.

3.2.2 Executing the Example

A sample test case for executing the project is provided in the *examples/test_case* folder. It contains two client applications to connect to the control plane and manage data plane devices. It also contains a folder with a Mininet topology definition, which will be used as the data plane for the example.

Figure 3.2 illustrates the network topology for the sample scenario and the packet and control flows that occur. S1, S2 and S3 are simple layer 2 switches implemented in P4, configured to send packets that result in a "table-miss" to the control plane through *Packet-Ins*. At the beginning of the experiment, all switches start with empty forwarding tables. First host *h2* starts an iperf3 server, which will be used to create a packet flow. Host *h1* will connect to the iperf3 server with host's *h2* IP (it's ARP cache already has an entry that indicates the MAC address of *h2*, so no MAC address discovery is necessary). The packet flow will begin (1, on the figure) and reach S1, which will send the first packet to the control plane since it will not match on its forwarding table (2). The control plane will send the packet to Client App 2, which simulates an application that installs flowrules on

Figure 3.2: Sample scenario to execute the control plane.



Source: The Author

devices based on table-misses[4], and it will react by inserting a flowrule on S1 to forward the packets to S2 (3). A similar process occurs with S2 after the packet flow reaches it (4, 5 and 6). Host h2 receives the first packet (7), and a normal iperf3 session begins. After 10 seconds, Client App 1 will install a flowrule on S1 that redirects the packet flow to S3 (8), disturbing the iperf3 session for a while. Another 10 seconds later, Client App 1 will re-establish the original flow, inserting another flowrule on S1 (10). The iperf3 session will finish 30 seconds after it is started. It should be possible to visualize the moments the flowrules alter the packet flows through the iperf3 reports, which should also indicates around 33% packet loss.

The link <https://www.youtube.com/watch?v=xYyBXHVGDqA> contains a video of the execution of the experiment. If the replication of this scenario is desired, execute the following steps:

1. Open four terminals on the root directory of the project.
2. On the first terminal, enter root mode with:

```
$ sudo su
```

3. On the second terminal, navigate to the Mininet folder and start the network simulation with:

```
$ cd examples/test_case/Mininet
$ make
```

[4]This type of application is discussed in more detail in Chapter 4.

4. On the third terminal, start Client App 1 with:

```
$ source scripts/export_vars.sh
$ python examples/test_case/client_table_write.py
```

5. On the fourth terminal, start Client App 2 with:

```
$ source scripts/export_vars.sh
$ python examples/test_case/client_table_write_2.py
```

6. On the Mininet terminal, create 2 terminals for hosts *h1* and *h2* with:

```
$ xterm h1 h2
```

7. On *h2*'s terminal, start the iperf3 server:

```
$ iperf3 -s
```

8. And on *h1*'s terminal, connect to the iperf3 server:

```
$ iperf3 -c 10.0.1.2 -u -t 30 -b 500K
```

9. As mentioned before, the experiment will run the iperf3 session for 30 seconds, with some trafic steering occurring during this time. The iperf3 report will indicate the moments that the florules were installed on S1.

4 SECURITY WITH INFORMATION FLOW CONTROL

This chapter presents vIFC, a module of the developed control plane that enhances security against Cross-App Poisoning (CAP) attacks that may be launched by malicious applications trying to explore vulnerabilities in the access control policy through information flow. CAP attacks are described in this work's background chapter (Section 2.6). First, in Section 4.1 the implemented solution, vIFC, is introduced along with implementation details and examples. Then, Section 4.2 presents the results of the implementation on a few test cases, in terms of efficacy (detecting CAP attacks) and efficiency (the latency added to the control plane processing time).

4.1 Virtual Information Flow Control (vIFC)

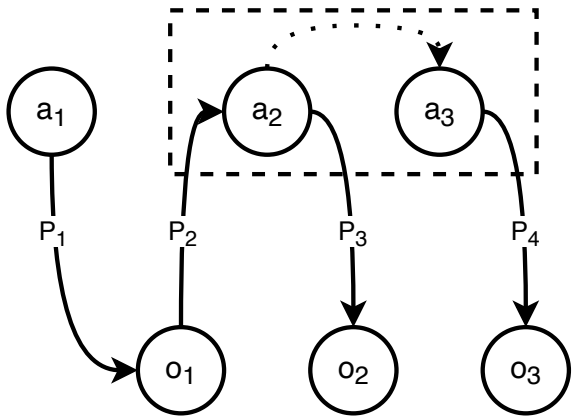
The module for the proposed control plane, *Virtual Information Flow Control* (vIFC), extends the data provenance graph proposed by ProvSDN (UJCICH et al., 2018) to register messages exchanged by virtual switches, so that the source of a message sent to a virtual switch may be tracked to the original application. Furthermore, vIFC ensures the compliance of security policies even with applications running in different controllers with user-based authentication.

4.1.1 Threat Model

When proposing ProvSDN, Ujcich et al. (UJCICH et al., 2018) consider a threat model in which (i) the SDN controller is reliable and secure, but may provide services to a malicious application, (ii) the attacker controls a malicious application that has least privileged permissions, and (iii) applications have identities and cannot fabricate actions to cause it to be seen as performed by another application. On their policy model, the authors also consider switches as applications, even though they only consider control plane applications as potentially malicious.

Using the model from Ujcich et al. (UJCICH et al., 2018), Figure 4.1 illustrates a CAP attack vector. ProvSDN can prevent application a_1 from poisoning object o_2 using permission p_3 from virtual switch a_2, but it cannot track the *out-of-band* flow between a_2 and a_3 (dashed line), which allows a_1 to use permission p_4 from a_3 to poison o_3.

Figure 4.1: CAP attack vector example.



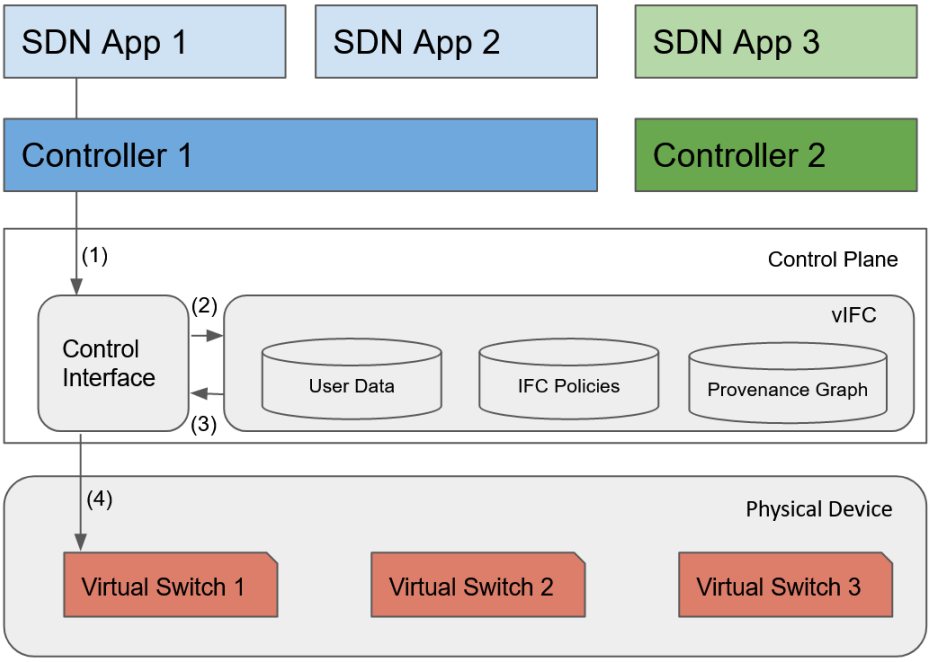
Source: The Author

For vIFC, the scope of CAP attack detection in comparison to ProvSDN is enlarged by considering that multiple virtual switches can be executed in the same physical device. In this case, the goal is to ensure that *out-of-band* information flow can be traced by comparing new messages with those previously seen by the control plane, which will allow the identification of information flow from least privileged to more privileged applications. To that end, the threat model from Ujcich et al. is extended to consider that virtual switches share the same control interface, that is also trusted and cannot be corrupted.

4.1.2 Conceptual Architecture

Figure 4.2 presents an overview of the conceptual architecture for vIFC. It has an interface with the control plane to receive the requests from the applications or switches and indicate if a data flow should be allowed or blocked. Before processing a request from an application and sending it to a virtual switch (1), the control interface sends the request to vIFC (2). vIFC then querries the `User Data` database to obtain information from it (for example, if the requesting application has the necessary permissions for that switch). After that, similar to ProvSDN, vIFC updates a provenance graph (MISSIER; BELHAJJAME; CHENEY, 2013; UJCICH et al., 2018) on the `Provenance Graph` database, that tracks the source of data flows. After updating the provenance graph, vIFC verifies if the resulting graph is consistent with the set of policies kept in the `IFC Policies` database. If the request from the application does not violate the established IFC policies, vIFC will signal to the control engine interface (3) that it may proceed and send the request to the virtual switch (4), otherwise, the request is blocked.

Figure 4.2: Conceptual Architecture for vIFC.



Source: The Author

4.1.3 Policy Model

This work considers a policy model similar to Ujcich et al. (UJCICH et al., 2018), but with a few changes to help with information flow detection between virtual switches. Policies may be applied to *read* or *write* information flows. In the example from Figure 2.6, flows (2), (4) and (6) are *read* information flows, because the message is flowing from the data plane (virtual switches) to the control plane (application). Likewise, flows (3), (5) and (7) are *write* information flows, because the information is flowing from an application in the control plane to a virtual switch and this control flow is modifying the configuration on those devices. On that note, policies that are applied to *read* information flows would block flows (2), (4) and (6), and policies that are applied to *write* information flows would block only flows (3), (5) and (7), should a violation be detected.

Formally, the model for vIFC consists of:

- **U:** A set of users. All applications must authenticate with the credentials of a user. The permissions for a given application depends on the user it uses to authenticate, therefore they may change if the user changes.
- **S:** A set of switches or other forwarding devices. Virtual switches are also included in this set.

- **L:** A set of permissions (labels) to interact with switches (e.g. read/write tables or receive packets).
- **T:** A mapping that determines the permissions each user has on each switch. Mathematically, **T** is a function $T : U \rightarrow (L \times S)^2$.
- **P:** A set of policies defined by the system administrator. An example of policy is $(L_1, L_2, WRITE, WARN)$, where L_1 and L_2 are labels. This policy indicates that information flow may happen from an application with label L_1 to an application with label L_2 (on the same switch), but a warning must be issued (WARN) when an application performs an action that changes the state of the data plane due to this information flow (WRITE). The possible response types are: WARN, BLOCK and NONE, and are very useful to help the system administrator in detecting and dealing with false positives.

4.1.4 Out-of-band Flow Detection

One of the challenges in CAP attack detection is to correlate data that flows between the switches and identify if they have any relation with the requests that originated from the control plane. The employed strategy was to create metadata for each flow between the control plane and the data plane. Examples of metadata include a *timestamp*, network addresses for *Packet-In* and *Packet-Out*, data segment hash, etc.

With this approach, it is possible to infer if a packet arriving at the control plane is the same one from a previous request. In Figure 2.6, for instance, when the packet from (4) arrives in the control plane, vIFC will correlate its metadata to detect it is the same packet from (1), and also detect the information flow between Virtual Switch 1 and 2.

It is worth mentioning that the *out-of-band* flow from this previous example, between Virtual Switches 1 and 2, is only deduced by the fact that they are the switches that directly interact with the control plane, but the real path of the packet might have been different, such as Virtual Switch 1 forwards the packet to 3, 3 forwards it to 2 and then it is sent to the control plane. Although vIFC does not track the complete path, it is still sufficient to the scope of this work to know that the packet eventually was sent to the control plane through Virtual Switch 2. In future works in which a complete packet trace would be desired, it is worth considering implementing part of vIFC on the data plane, which would also reduce the workload on the control plane.

4.1.5 Data Provenance Graph

As it was previously mentioned, a core piece of vIFC and other IFC solutions is the provenance graph. vIFC's provenance graph is initialized with all users and all switches as nodes, and it is updated by adding edges between nodes when a data flow is detected between them. It is easy to detect data flow between users and switches since this flow must go through the control plane (Figure 4.2), but it is not straightforward to detect data flow between two switches as in the *out-of-band* examples, since the control plane cannot "see" them directly. Those data flows are detected with the strategies previously mentioned, such as comparing packet metadata. After updating the graph, we check if a path was created that indicates an information flow from a least privileged application to a more privileged one, or that violates one of the administrator-defined policies.

4.2 Test Case Attacks

The evaluation of vIFC is based on two test case attacks: (i) an attack that manipulates legitimate applications that manage forwarding tables on virtual switches and (ii) an attack that manipulates *In-Band* telemetry data (TU; HYUN; HONG, 2017). Those attacks were chosen due to the open-source nature of the applications, which run in the ONOS controller (ONOS, 2021). For those test cases, Mininet (MININET, 2017) was employed to create an emulated data plane, running simple layer-2 bmv2 switches[1].

A third test case that mixes concepts from the other ones was also designed to be executed with a different concrete data plane, using P4VBox (SAQUETTI; BUENNO; AZAMBUJA, 2020). The goal of this test case is to demonstrate that vIFC (as well as the developed control plane as a whole) is platform-independent, being able to perform information flow control under a variety of environments. The switches deployed for this test case are also layer-2 P4 switches, with vlan tags and a *Packet-In* implementation to comply with P4VBox requirements.

The following subsections present the test cases in more detail. For all test cases, the attacker's malicious application is considered to have fewer permissions being able only to receive and send packets to the network through *Packet-In* and *Packet-Out*, whereas

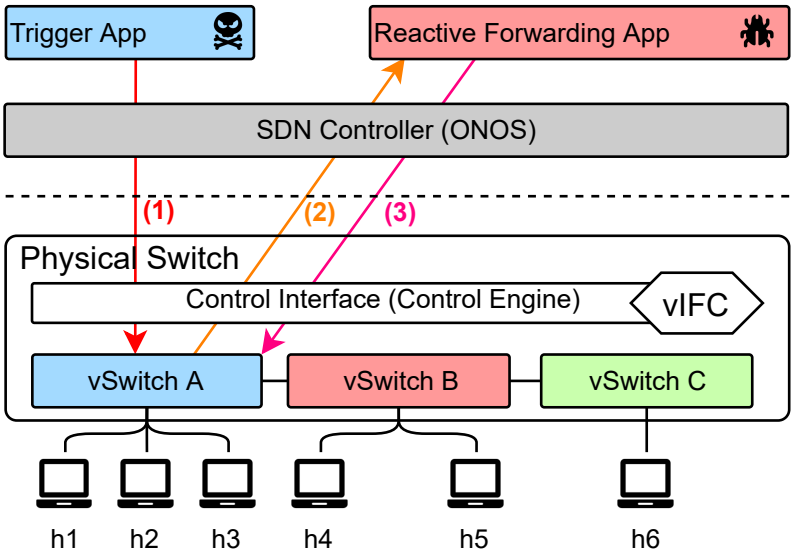
[1] bmv2 (Behavioral Model Version 2) is a software switch not intended to have production-grade performance, but offers a good data plane abstraction for debugging purposes. It is available on Github at: <https://github.com/p4lang/behavioral-model>

the legitimate applications have additional permissions to write data on the data plane devices. The goal of the attacker is to manipulate the legitimate applications to make use of those additional permissions.

4.2.1 Reactive Forwarding Attack

The Reactive Forwarding application (fwd) is a SDN app available on the ONOS controller[2]. The application captures packets that are sent to the control plane through a *Packet-In* after a "table-miss" on the data plane, and reacts by creating and inserting forwarding flow rules on switches (hence its name) to allow the packet flow between the communicating hosts. It is often the case that P4 switches have smaller tables in terms of possible match entries and old entries have to be replaced often, thus the need for constant operation of the *fwd* app.

Figure 4.3: CAP attack topology for the Reactive Forwarding test case.



Source: The Author

Figure 4.3 illustrates this scenario, where *h1* starts a packet flow to *h2*. To perform the CAP attack a malicious app (Trigger App) was implemented, which sends a fabricated ARP reply packet to the network through a *Packet-Out* (1). That ARP reply has fake information that a valid IP address (*h2*'s IP) is present at a malicious-app controlled MAC address (this packet is sent with the goal of steering the flow from this IP to the attacker's computer). This packet is then sent to the Reactive Forwarding app from vSwitch A (2).

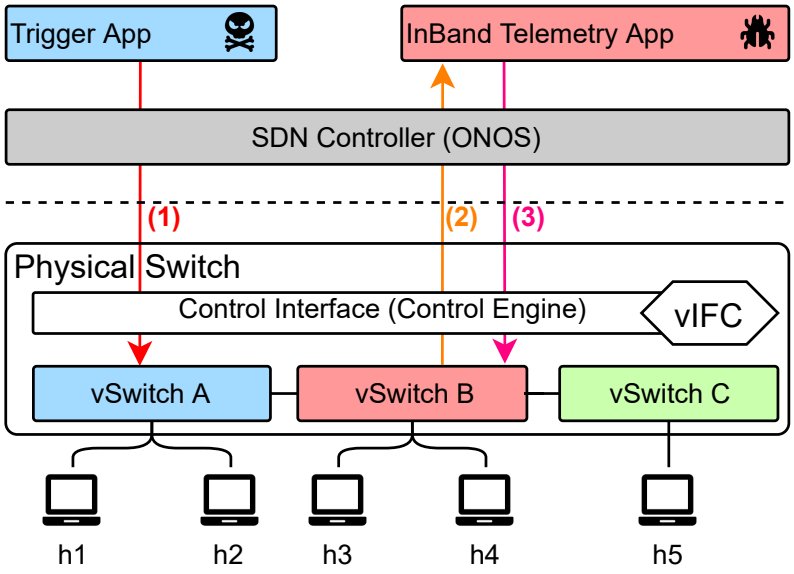
[2]Source code for the Reactive Forwarding app: <https://github.com/opennetworkinglab/onos/tree/master/apps/fwd/src/main>

The legitimate application will consume the packet and react by creating and inserting a flow rule on vSwitch A (3) that will redirect the flow from *h1* to *h2* to another host (which may be *h3* or a host linked to another virtual switch).

4.2.2 In-Band Telemetry Attack

For the test case with the In-Band Telemetry (int) application, a malicious app was also implemented. The malicious app sends fabricated packets periodically to the network, which contains fake telemetry data. The *int* app monitors the switches on the data plane with Multi-Hop Route Inspection (MRI)[3], which tracks the path that the packets go through and the queue depth for the switches in that path.

Figure 4.4: CAP attack topology for the In-Band Telemetry test case.



Source: The Author

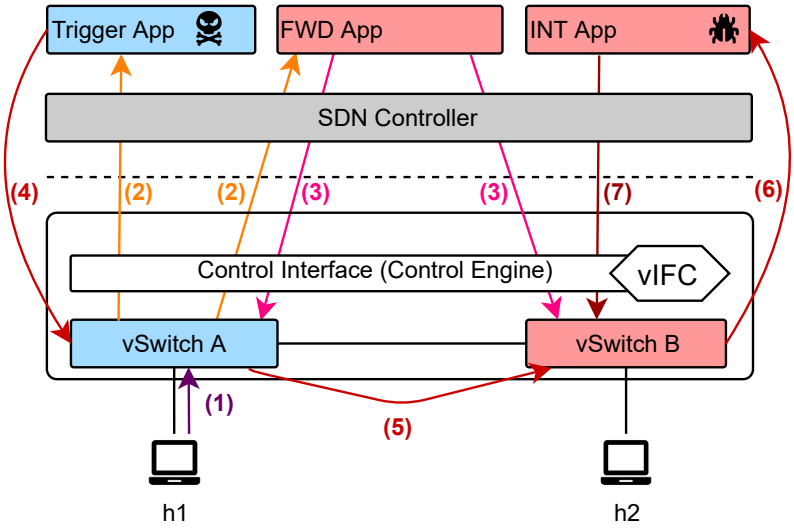
The workflow for this test case is shown in Figure 4.4. The malicious app sends the fabricated packet with fake information about the queue depth of vSwitch B (1). The packet is forwarded to vSwitch B from vSwitch A in the "out-of-band" flow and then sent to the In-Band Telemetry app (2). The legitimate application notices the high queue depth (considering the normal data plane scenario for the MRI example, an average of the queue depths for the virtual switches is 40, whereas the fabricated reports indicate a 4000 queue depth). The legitimate application blocks the flow from vSwitch A to vSwitch B (3).

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

4.2.3 Mixed Attack

This last test case was designed to be executed in a different data plane, P4VBox. Moreover, the applications on this scenario do not run in the ONOS controller, but were simulated in Python, connecting directly to the control plane. The goal is to change the environment, in terms of SDN controller and data plane implementation, to assert that vIFC is platform-independent.

Figure 4.5: CAP attack topology for the Mixed test case.



Source: The Author

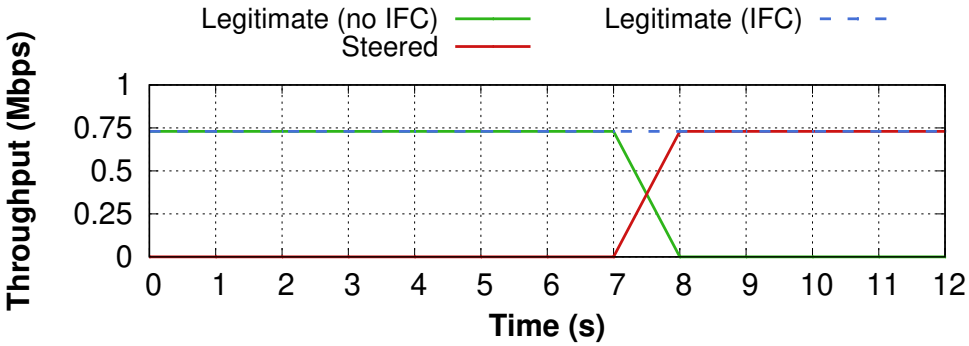
The test case consists of a mix of the previous ones, with the malicious app trying to poison the In-Band Telemetry app to block a legitimate flow. Figure 4.5 illustrates the attack flow. Initially, no forwarding rules are defined on the tables of both virtual switches. Host *h1* starts a flow with host *h2* destination address (1). Since the first packet of the flow results in a "table-miss" on vSwitch A, it is sent to the control plane through a *Packet-In* and forwarded to the applications that can receive this *Packet-In* (2). The Reactive Forwarding app will create and install appropriate flow rules on both virtual switches to allow the intended legitimate flow to occur (3). The malicious app will also trigger a routine upon receiving the *Packet-In* to create a fabricated *Packet-Out* after a short delay, aiming to poison the In-Band Telemetry app similar to the previous scenario (4). The malicious packet is sent "out-of-band" to vSwitch B (5) and then sent to the legitimate application through a *Packet-In* (6). The legitimate application blocks the legitimate flow in reaction to the malicious packet (7).

4.3 Evaluation

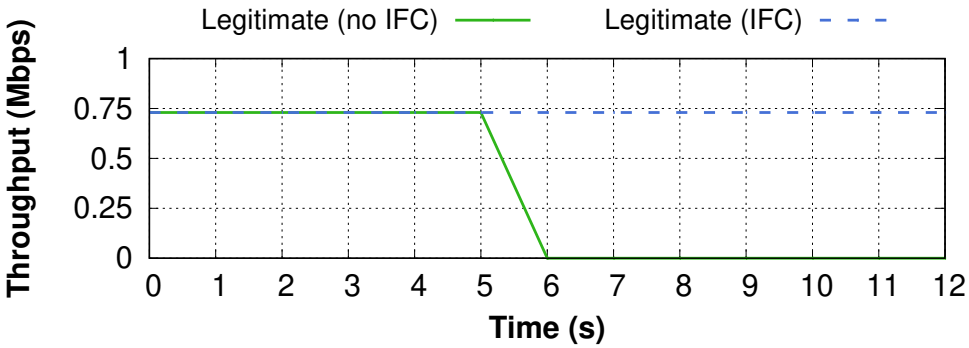
4.3.1 Efficacy

Using the CAP attack test cases of Section 4.2, vIFC's efficacy (being able to block the attacks) and efficiency (additional overhead / latency to the control plane) were evaluated to verify its viability. The expected outcome is that vIFC will block the malicious flow reaching the legitimate apps on all test cases since they violate the default policy of a less privileged application sending data to a more privileged one.

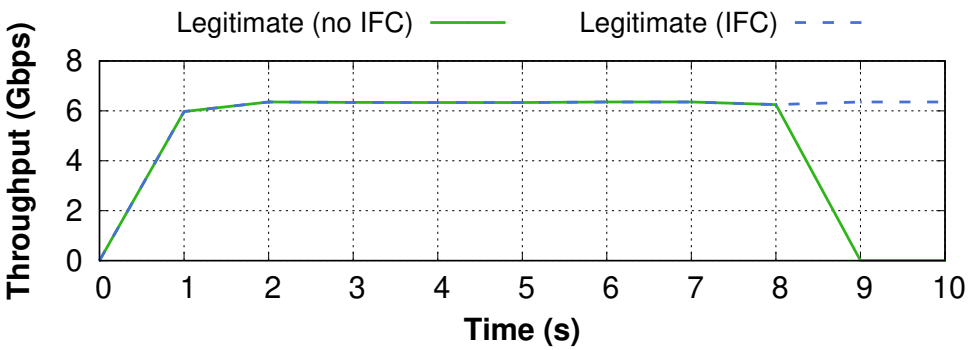
Figure 4.6: vIFC efficacy on FWD, INT and P4VBox test cases.



(a) Reactive Forwarding



(b) In-Band Telemetry



(c) P4VBox

Source: The Author

Figure 4.6 demonstrates the operational impacts of the three test cases on the legitimate flows of the network and vIFC's efficacy in mitigating said impacts. The green lines represents an iperf3 flow between two hosts that passes through a virtual switch that receives a malicious flow rule from a CAP attack without vIFC, while the dashed blue lines represents the same scenarios with vIFC used to mitigate the attacks. For the Reactive Forwarding case (Figure 4.6a), vIFC detects and blocks the malicious *Packet-In*, preventing the traffic steering that occurs around 7 seconds of the experiment. In the In-Band Telemetry case (Figure 4.6b), vIFC also prevents the malicious packet containing the fake telemetry data from reaching the legitimate app, by inferring the "out-of-band" flow after detecting that the *Packet-In* is the same *Packet-Out* that the malicious app sent before. Finally, the P4VBox case (Figure 4.6c) shows that vIFC can also detect malicious information flow on a different, more complex, test case.

4.3.2 Efficiency

For the efficiency evaluation, data from the Reactive Forwarding and In-Band Telemetry test cases was gathered by measuring the total processing time of the control plane for the *Packet-In* and *Packet-Out* operations and the fraction of this time that was employed on vIFC during the attacks. For this purpose, an additional policy was defined to allow the malicious information flow to reach the legitimate apps and generate a warning when this happens[4], so that the attack can go through but vIFC's ability to detect it also be asserted.

Table 4.1: vIFC latency on the Reactive Forwarding attack (μs).

	PacketOut			PacketIn		
	Total	vIFC	Baseline	Total	vIFC	Baseline
avg.	478.1	409.8 (85%)	69.8	425.0	396.1 (93%)	63.3
σ	78.3	74.35	10.56	69.67	67.42	59.92

Table 4.2: vIFC latency on the In-Band Telemetry attack (μs).

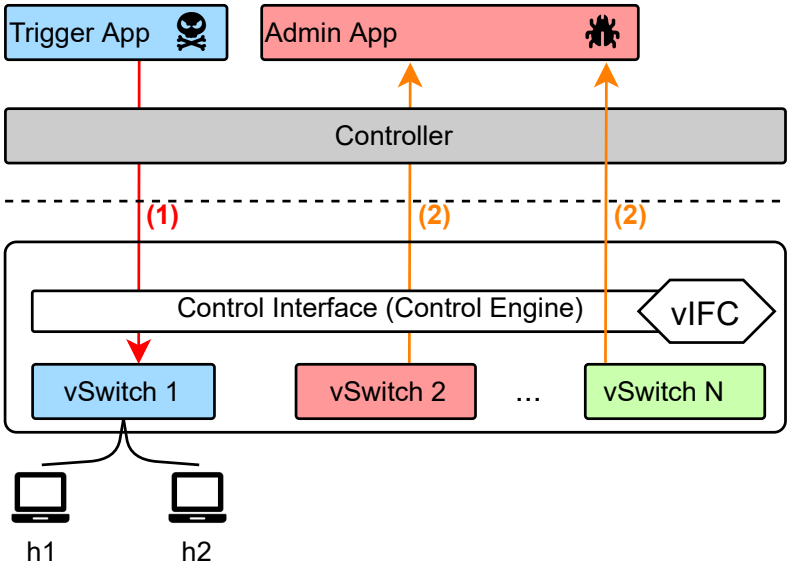
	PacketOut			PacketIn		
	Total	vIFC	Baseline	Total	vIFC	Baseline
avg.	1,095.90	905.67 (82%)	90.40	1,419.47	1,341.40 (96%)	60.23
σ	590.33	202.02	67.69	575.52	557.62	34.52

[4]Using the model from Section 4.1.3, the policy used for the latency tests is: $(PACKET, FLOWRULE, Read, Warn)$

Tables 4.1 and 4.2 contain the average results of the efficiency tests from 30 rounds of each experiment. For the Reactive Forwarding experiment, the proportion of the processing time of the control plane for vIFC on *Packet-Out* and *Packet-In* respectively was 85.71% and 93.20%, whereas for the experiment with the In-Band Telemetry application the proportion was 82.64% and 96.88%. This high fraction of time is due to the simplicity of the other operations that the control plane performs in the *Packet-In* and *Packet-Out* operations, which is basically sending the packet through a socket.

It takes more time for vIFC to verify the requests on the In-Band Telemetry scenario compared to the Reactive Forwarding scenario. This is due to the detection of the "out-of-band" flow on the former. It ultimately depends on the judgment of the system administrator if those times are acceptable for its specific scenario. In the test cases, considering that the applications only request *Packet-In*s from the control plane each 5 - 10 seconds, a latency of less than 2ms is deemed acceptable. Even so, it is certainly possible to improve vIFC's performance in future work with a more optimised code.

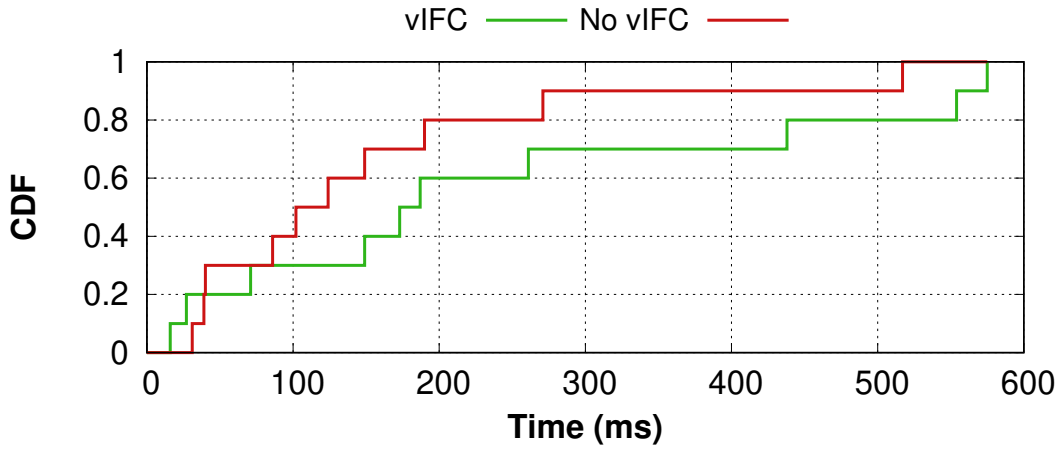
Figure 4.7: Topology for the vIFC stress test.



Source: The Author

A stress test on vIFC was also performed to assert its impact on the control plane through a Cumulative Distribution Function (CDF). Figure 4.7 illustrates the topology for the test, which was repeated 10 times, with N ranging from 5 to 50. For each switch other than vSwitch 1, the malicious app sends a *Packet-Out* that will be forwarded to that switch and then send to a higher priority app (Admin App). vIFC should be able to detect the violation in all *Packet-Ins*. Figure 4.8 shows the resulting CDF, which indicates an expected slight performance overhead when using vIFC.

Figure 4.8: CDF for the vIFC stress test.



Source: The Author

It is worth noting that vIFC has a configurable max memory limit. For the CDF test case, the memory limit was set high enough so that all attacks were blocked. However, with the memory limit set lower, some attacks will go through since vIFC will not be able to trace the information flows that it already discarded.

5 CONCLUSIONS AND FUTURE WORK

The classical approach to computer networking is still broadly employed in today's commercial and industrial environments due to legacy structures. Research and development of SDNs continue to evolve, and it will eventually surpass this barrier of adoption when the technology becomes mature enough.

This work presented the development process of a SDN control plane with a modular design, in contrast with the classical inflexible approach. The main advantages of SDNs are the separation between the functionalities of each component, leading to a complex structure with simple components, which facilitates the development of new features and the maintenance of deployed hardware and software. The modular design also makes it easier to implement new functionalities on the control, application or data planes. A switch "load balance" for a data center, for instance, could be implemented by changing the P4 code of a deployed switch or by an additional module on the control plane.

Another problem addressed in this work is the control plane security and network integrity, mainly focused on a novel, emerging class of SDN attacks known as CAP attacks. Research in this topic is still in the early stages, as UJCICH et al. 2018 firstly introduced the problem and proposed an initial solution. The developed control plane made use of the core concepts of that work, such as data provenance through Information Flow Control and adapted it to a scenario where a single switch may virtualize multiple switches that can be seen independently by the control plane.

While this work presents a fully functional control plane for a SDN that can be easily adapted to execute in multiple SDN environments, future work can certainly improve its usability so that it reaches production grade quality. An interface with a Graphical User Interface (GUI) would make it more appealing to network administrators; support for other runtime protocols such as OpenFlow in addition to P4 Runtime will make it even more flexible, etc.

In regards to security, vIFC can be improved in terms of false-positive detection. One of the main advantages of the policy system is helping the system administrator detect and override specific cases where an information flow will be seen as a violation, but it is indeed legitimate and not a malicious attack. Improving the detection of those cases is a challenging research goal that would increase the system autonomy, not having to rely on external user-defined exceptions.

REFERENCES

Aruba. **Aruba Networks. SDN Apps - Airheads Community**. 2021. Retrieved April 18, 2021 from https://www.arubanetworks.com/sdn-apps/.

BENZEKKI, K.; FERGOUGUI, A. E.; ELALAOUI, A. E. Software defined networking (sdn): a survey. **Security Comm. Networks 2016**, p. 5803–5833, 2016.

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

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.

GRPC. **gRPC: A high performance, open source universal RPC framework**. 2021. Retrieved September 16, 2021 from <https://grpc.io/>.

HANCOCK, D.; MERWE, J. van der. Hyper4: Using p4 to virtualize the programmable data plane. In: **12th International on Conference on Emerging Networking EXperiments and Technologies**. New York, NY, USA: Association for Computing Machinery, 2016. (CoNEXT '16), p. 35–49. ISBN 9781450342926.

HANG, Z. et al. Programming protocol-independent packet processors high-level programming (p4hlp): Towards unified high-level programming for a commodity programmable switch. 2019.

HARDY, N. The confused deputy: (or why capabilities might have been invented). **SIGOPS Oper. Syst. Rev.**, Association for Computing Machinery, New York, NY, USA, v. 22, n. 4, p. 36–38, oct. 1988. ISSN 0163-5980. Available from Internet: <https://doi.org/10.1145/54289.871709>.

JOSH, K.; BENSON, T. Network function virtualization. **IEEE Internet Computing**, v. 20, p. 7–9, 2016.

MCKEOWN, N. P4 runtime – putting the control plane in charge of the forwarding plane. **ONF News and Events**, 2017. Retrieved September 8, 2021 from <https://opennetworking.org/news-and-events/blog/p4-runtime-putting-the-control-plane-in-charge-of-the-forwarding-plane>.

MCKEOWN, N. et al. Openflow: Enabling innovation in campus networks. **ACM SIGCOMM Computer Communication Review**, v. 38, p. 67–79, 2008.

MININET. **Mininet, An instant virtual network on your laptop (or other PC)**. 2017. Retrieved September 6, 2021 from http://mininet.org/.

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

NETO, M. C. M.; BEZERRA, R. M. d. S. **Protocolos de roteamento RIP e OSPF**. 2002. Retrieved September 6, 2021 from <https://www.researchgate.net/publication/260637894>.

ONF. **P4Runtime Specification**. 2020. Retrieved September 8, 2021 from <https://opennetworking.org/wp-content/uploads/2020/10/P4Runtime-Specification-120-wd.html>.

ONOS. **Open Network Operating System. Apps and Use Cases**. 2021. Retrieved September 2, 2021 from <https://wiki.onosproject.org/display/ONOS/Apps+and+Use+Cases>.

OPEN-NETWORK-FOUNDATION. **SDN Architecture Overview**. 2013. Retrieved September 3, 2021 from <https://opennetworking.org/wp-content/uploads/2013/02/SDN-architecture-overview-1.0.pdf>.

OPEN-NETWORK-FOUNDATION. **OpenFlow Switch Specification**. 2015. Retrieved September 2, 2021 from <https://opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf>.

SAQUETTI, M.; BUENNO, G.; AZAMBUJA, J. r. P4vbox: Enabling p4-based switch virtualization. **IEEE Communications Letters**, v. 24, 2020.

SATTOLO, T. et al. Classifying poisoning attacks in software defined networking. **2019 IEEE International Conference on Wireless for Space and Extreme Environments (WiSEE)**, IEEE, p. 96–100, 2019.

SRINIVASAN, R. **RFC 1831, RPC: Remote Procedure Call Protocol Specification Version 2**. August 1995.

TANAEMBAUM, A. S.; BOS, H. **Modern Operating Systems**. 4th. ed. [S.l.]: Prentice Hall, 2014.

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: **2018 ACM SIGSAC Conference on Computer and Communications Security**. New York, NY, USA: ACM, 2018. (CCS '18), p. 648–663. ISBN 9781450356930.

ZHANG, C. et al. Hyperv: A high performance hypervisor for virtualization of the programmable data plane. In: IEEE. **International Conference on Computing and Communication Networks 2017**. [S.l.], 2017. p. 1–9.

ZHENG, P.; BENSON, T.; HU, C. P4visor: Lightweight virtualization and composition primitives for building and testing modular programs. In: **International Conference of Emerging Networking EXperiments and Technologies**. [S.l.]: ACM, 2018. (CoNEXT '18), p. 98–111. ISBN 978-1-4503-6080-7.