Fakultät für Elektrotechnik und Informatik
Institut für Softwaretechnik und Theoretische Informatik
Lehrstuhl für Security in Telecommunications

From Threats to Solutions in Data Center Networks

vorgelegt von
M. Sc.
Kashyap Thimmaraju

von der Fakultät IV - Elektrotechnik und Informatik
der Technische Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Ingeniuerwissenschaften
- Dr. -Ing. -
genehmigte Dissertation

Promotionsausschuss:

Vorsitzender:	Prof. Georgios Smaragdakis, Ph.D., Technische Universität Berlin
Gutachter:	Prof. Dr. Jean-Pierre Seifert, Technische Universität Berlin
Gutachter:	Prof. Dr. Stefan Schmid, Universität Wien, Österreich
Gutachter:	Prof. Arvind Krishnamurthy, Ph.D, University of Washington, Seattle
Gutachter:	Prof. Eric Keller, Ph.D, University of Colorado, Boulder

Tag der wissenschaftlichen Aussprache: 17. Januar 2020

Berlin 2020

Ich versichere von Eides statt, dass ich diese Dissertation selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

Datum

Abstract

In this dissertation we adopt a threat model where the data center network infrastructure is potentially malicious. To describe practical threats and solutions related to malicious switches, we draw our attention to multi-tenant data center networks that i) consolidate control over the (hardware and software) switches to a logically centralized controller and ii) use virtualization techniques for multi-tenancy.

Our extensive security analyses and evaluations of the design, specifications and systems of logically centralized data center network controllers reveals the following. Malicious switches can covertly bypass network-wide security policies and mechanisms via the controller. We identify three reasons for the existence of such covert channels: i) malicious switches share the logical controller, ii) lack of authentication and authorization of switches to the controller and iii) introduction of automation and programmability of the network. These channels can be reliable (TCP-based) and fast (10 Mbps). As a result malicious switches can launch several network-based attacks in the data center, e.g., to circumvent firewalls to access unauthorized data. Furthermore, our state transition and delay model of the switch-controller handshake allows us to design, implement and evaluate a covert timing channel that uses a frame-based transmission scheme for accurate and low bandwidth (20 bps) communication, e.g., to exfiltrate private keys. We also initiate the discussion of practical countermeasures, e.g., coupling TLS with the switch-controller handshake for authentication.

Next, our security analysis of network virtualization architectures that use virtual switches—a key system for enforcing network isolation in multi-tenant data center networks—sheds light on the following. Increasing network functionality in the virtual switch coupled with co-locating it with the hypervisor and the lack of appropriate threat models among other reasons has resulted in an insecure design. An attacker can escape host and network virtualization and compromise the entire data center as a worm. By fuzzing the packet parser of a popular virtual switch (OvS), we discovered 3 exploitable memory corruption vulnerabilities. We use just one of them in a popular cloud management system (OpenStack) to demonstrate our point: From a virtual machine (VM) we could take down hundreds of servers in a few minutes. Our measurements of the impact of software-based countermeasures that could have prevented the discovered vulnerabilities from being exploited for OvS show that maximum packet processing throughput is reduced by half in the kernel whereas the overhead in user-space is minimal (1-15%).

Finally, we continue our previous work by first surveying the security landscape of 23 virtual switches and conclude that nearly all of them lack security in their design. Hence, we introduce four secure design principles for virtual switches and accordingly build a scalable prototype that prevents the virtual switch from being a liability to the (multi-tenant) data center network. The key insights from our system and performance evaluations are as

follows. We can isolate and scale the virtual switches and their respective virtual networks by placing them in containers in VMs. Using Single-Root I/O Virtualization allows us to i) reduce the trusted computing base of virtual networking, ii) provide cloud operators an easy upgrade path and iii) increase the tenants' network application (e.g., web servers and key-value stores) performance.

Zusammenfassung

In dieser Dissertation nehmen wir ein Bedrohungsmodell an, bei dem die Netzwerkinfrastruktur eines Rechenzentrums potentiell bösartig ist. Zur Beschreibung praktischer Bedrohungen und Lösungen im Zusammenhang mit böswilligen Switches, fokussieren wir uns auf Multi-Mandanten-Rechenzentrumsnetzwerke, i) bei denen die Kontrolle über die (Hard- und Software-) Switches einem (logisch) zentralisierten Controller unterliegt und ii) die Virtualisierungstechniken für Multi-Mandanten-Fähigkeit verwenden.

Unsere umfangreichen Sicherheitsanalysen und Bewertungen des Designs, der Spezifikationen und der Systeme von Controllern für zentralisierte Rechenzentrumsnetzwerke zeigt, dass bösartige Switches die netzwerkweiten Sicherheitsrichtlinien und -mechanismen über den Controller verdeckt umgehen können. Wir identifizieren drei Gründe für die Existenz solcher verdeckten Kanäle. i) Der zentralisierte Controller wird von bösartige Switches mitbenutzt; ii) Switches benötigen keine Authentifizierung oder Autorisierung gegenüber dem Controller; sowie iii) die Einführung von Automatisierung und Programmierbarkeit des Netzwerks. Diese Kanäle können verlässlich (TCP-basiert) und schnell (10 Mbps) sein. Damit können bösartige Switches verschiedene netzwerkbasierte Angriffe im Rechenzentrum durchführen und beispielsweise zur Umgehung von Firewalls oder den unberechtigten Zugriff auf Daten benutzen. Darüber hinaus können wir mit Hilfe unseres Zustandsübergangs- und Verzögerungsmodell des Switch-Controller-Handshakes einen verdeckten, Zeit-basierten Kommunikationskanal entwerfen, implementieren und evaluieren. Dieses Frame-basierte Übertragungsschema für bandbreitenarme (20 bps) Kommunikation mit niedriger Fehlerrate erlaubt es uns z.B. private Schlüssel zu exfiltrieren. Als Abwehrmaßnahme diskutieren wir unter anderem die Kopplung von TLS mit dem Switch-Controller-Handshake zur Authentifizierung.

Eine weitere Schlüsseltechnologie zur Durchsetzung von virtuellen Netzwerk-Architekturen in Multi-Mandanten-Rechenzentrumsnetzwerken ist die Verwendung von virtuellen Switches. Unsere Sicherheitsanalyse dieser Architekturen zeigt, dass die Erhöhung der Funktionalität im virtuellen Switch in Verbindung mit der die Einbettung in den Hypervisor, das Fehlen geeigneter Bedrohungsmodelle neben anderen Gründen zu einem unsicheren Design geführt hat. Ein Angreifer kann der Host- und Netzwerkvirtualisierung entkommen und damit das gesamte Rechenzentrum als Wurm kompromittieren. Durch das Fuzzing des Paketparsers eines populären virtuellen Switch (OvS) entdeckten wir drei ausnutzbare Schwachstellen. Eine davon nutzen wir in einem beliebten Cloud-Management-System (OpenStack) um unseren Befund nachzuweisen: Von einer virtuellen Maschine (VM) aus könnten wir hunderte von Servern in wenigen Minuten kompromittieren. Unsere Messungen zeigen, dass die

Auswirkungen von softwarebasierten Gegenmaßnahmen, die hätten verhindern können, dass die in OvS entdeckten Schwachstellen ausgenutzt werden können, den maximalen Paketdurchsatz im Kernel um die Hälfte reduzieren, während der Overhead im User Space minimal ist (1-15%).

Abschließend bemerken wir dass in unserer Studie von 23 virtuellen Switches fast keiner Sicherheit als Design-Ziel verfolgt. Daher schlagen wir vier sichere Designprinzipien für virtuelle Switches vor und entwerfen einen skalierbaren Prototyp, der verhindert, dass der virtuelle Switch eine Gefahr für das Multi-Mandanten-Netzwerk darstellt. Die wichtigsten Erkenntnisse aus unseren System- und Leistungsbewertungen sind wie folgt. Wir können die virtuellen Switches und ihre jeweiligen virtuellen Netzwerke durch die Verwendung von VMs isolieren und skalieren. Die Verwendung von Single-Root-I/O-Virtualisierung ermöglicht es uns, i) die trusted computing base virtueller Netzwerke zu reduzieren, ii) Cloud-Betreibern eine einfachen Upgrade-Pfad zu bieten und iii) die Netzwerkdurchsatz der Mandanten-Maschinen zu erhöhen, beispielsweise für Webserver oder Key-Values-Datenbanken.

Bibliographic Note

Partial results of the work presented in Part I have been published in the following peer-reviewed proceedings.

1. Kashyap Thimmaraju, Liron Schiff, and Stefan Schmid. „Outsmarting Network Security with SDN Teleportation“. In: *Proceedings of the IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2017, pp. 563–578. See Chapter 4/ [224].

2. Robert Krösche, Kashyap Thimmaraju, Liron Schiff, and Stefan Schmid. „I DPID It My Way! A Covert Timing Channel in Software-Defined Networks“. In: *Proceedings of the IFIP Networking Conference (IFIP Networking) and Workshops*. IEEE. 2018, pp. 217–225. See Chapter 5/ [112].

Partial results of the work presented in Part II have been published in the following peer-reviewed proceedings.

1. Kashyap Thimmaraju, Bhargava Shastry, Tobias Fiebig, Felicitas Hetzelt, Jean-Pierre Seifert, Anja Feldmann, and Stefan Schmid. „Taking Control of SDN-based Cloud Systems via the Data Plane“. In: *Proceedings of the Symposium on SDN Research (SOSR)*. ACM. 2018, pp. 1–15. See Chapter 6/ [226].

2. Kashyap Thimmaraju, Bhargava Shastry, Tobias Fiebig, Felicitas Hetzelt, Jean-Pierre Seifert, Anja Feldmann, and Stefan Schmid. „The vAMP Attack: Taking Control of Cloud Systems via the Unified Packet Parser“. In: *Proceedings of ACM Cloud Computing Security Workshop (CCSW)*. ACM. 2017, pp. 11–15. See Chapter 6/ [229].

3. Kashyap Thimmaraju, Gábor Rétvári, and Stefan Schmid. „Virtual Network Isolation: Are We There Yet?“ In: *Proceedings of the Workshop on Security in Softwarized Networks: Prospects and Challenges (SECSON)*. ACM. 2018, pp. 1–7. See Chapter 7/ [230].

4. Kashyap Thimmaraju, Saad Hermak, Gábor Rétvári, and Stefan Schmid. „MTS: Bringing Multi-Tenancy to Virtual Networking“. In: *Proceedings of the USENIX Annual Technical Conference (ATC)*. USENIX. 2019, pp. 521–536. See Chapter 7/ [223].

Acknowledgements

This dissertation would not have been possible without the support from my near and dear ones. I'd like to express my sincere gratitude to Jean-Pierre Seifert for including me in his research group, connecting me with Stefan Schmid and supporting me throughout my PhD. To Stefan Schmid, who has constantly been there for me throughout my PhD experience, so much of this work is a result of his guidance. To Arvind Krishnamurthy and Eric Keller for accepting to review and serve on my dissertation and defense committee. To Georgios who really motivated and advised me towards the end of my PhD. To Claudi who was always there to help out with purchases, reimbursements, holidays and all the other things that enabled me to work at the university. I have got to work with and learn from so many interesting people throughout my time at TU Berlin, and for that I am grateful. I would like to share some memorable moments rather than list out many people. The many white board sessions with Stefan in 2015-2016 when you were still in Berlin and also when Liron, Nati and Nir visited Berlin and we went to the Einstein Kaffee Cafe. The moment JP shared his advise with me in the corridor on the 16th floor about how to design experiments. When Liron and I were on Skype and we discovered a new covert channel with ONOS was exciting and something I want to continue experiencing. Shinjo and Altaf who got me home safely after the T-Labs Xmas party. Debugging those MPLS parsing bugs in OvS with Bhargava was really the start of something much bigger. I think we both found our way forward after that fuzzy(ng) experience. Working with Felicitas on developing and testing the exploit for OvS was really nice, it was impressive to see how fast she got it working. Working with Robert on his master thesis and our timing covert channel was a great experience for me. I owe Tobias Fiebig a big thanks for introducing me to JP and instilling the "Scientific Method" in me. Anja Feldmann instilled integrity into my research when we met at JPs office and also motivated me when we met at the SecSoN workshop in Budapest. I'm thankful to Saad for opening up my mind to Geopolitics with his great book recommendation. His experimental framework was so cool, I could receive plots of the measurements (for the ATC paper) made via email. The seminar students were more often than not a great batch to work with, as it was interactive and we learnt from each other. For me, lunch has always been a social affair and I enjoyed every one of them with the people I was with. And not to forget the coffee/tea discussions after that. It was usually around noon. The usual suspects were Bhargava, Felicitas and Julian, with Robert, Shinjo, Altaf, Dominik and Nils joining frequently. It was fun to go bouldering and climbing with Robert. I liked the competitiveness we had which I think was implicit. Dominik's birthday parties were always a blast with so many people having a good time! Thilo, Anna, Nils, Robert, Altaf and Saad who helped us move flats. I owe all the people who designed my awesome defense hat a big thanks! Working with Lukas towards the end of my PhD was a nice learning experience and also I am grateful to him for helping me out with the EIT project so that I could concentrate on getting my dissertation and defense. Gabor, who supported and guided my work on `MTS` via the many skype calls we had in 2018 and 2019. My visits to the US (California and Washington) and India to meet

friends and family as well when my friends and family visited me in Berlin were fun and memorable. Meeting Julia changed my life and she has always been there for me during my PhD. To her parents and family who were like my family throughout. Dad, Mom and Preeti, when they visited me in Berlin for my 30th and my 33rd in Ireland, was really memorable in addition to them visiting us for my defense and to see Lia. To Ruth Wisharth for letting me participate in her writing without boundaries course and exposing me to the different styles of creative non-fiction. I sought inspiration from Muhammad Shahbaz's dissertation for Figure 2.1 in my dissertation and used the `clean-thesis` latex template to write this dissertation. A special thanks to Nils Wisiol for translating the abstract into German.

Parts of the work conducted in this dissertation were financially supported by the following: Federal Ministry of Education and Research of Germany Grant KIS1DSD032 (Project Enzevalos), Software Campus 2.0 project nos. 01IS17052; Helmholtz Research School on Security Technologies scholarship at the German Aerospace Center and the Technical University Berlin; and the "API Assistant" activity of EIT Digital.

To my first daughter,
Lia.

Contents

1 Introduction 1

2 The Multi-Tenant Data Center 5

2.1 Overview . . . 5

2.2 Software-Defined Centralized Control . . . 6

2.3 Network Virtualization . . . 7

3 Threat Model 11

I Covert Channels via the Control Plane 13

4 Outsmarting Network Security with Teleportation 15

4.1 Preliminaries . . . 16

4.2 Modeling Teleportation . . . 17

4.3 Teleportation Techniques . . . 18

4.4 Switch- and Host-based Attacks . . . 22

4.5 Out-of-Band Forwarding Performance . . . 32

4.6 Countermeasures . . . 36

4.7 Related Work . . . 38

4.8 Conclusions . . . 39

5 Peekaboo! I DPID You: A Novel OpenFlow Covert Channel 41

5.1 A Covert Channel using Teleportation . . . 42

5.2 Design and Performance Challenges . . . 49

5.3 Evaluation . . . 51

5.4 Discussion . . . 56

5.5 Conclusions . . . 57

II Network Isolation via the Data Plane 59

6 Reins to the Cloud via the Virtual Switch 61

6.1 Security Analysis . . . 62

6.2 Case Study: OvS in OpenStack . . . 66

6.3 Software Countermeasures . . . 72

6.4 Related Work . . . 75

6.5 Conclusion . . . 76

7 Bringing Multi-Tenancy to Network Virtualization 77

7.1 Securing Virtual Switches . . . 78

7.2 Threat Model 80
7.3 Design Principles and Security Levels 80
7.4 The `MTS` Architecture 82
7.5 Evaluating Tradeoffs 86
7.6 Workload-based Evaluation 91
7.7 Scaling `MTS` 94
7.8 Discussion 96
7.9 Related Work 97
7.10 Conclusion 99

8 Future Work 101

9 Conclusion 103

Bibliography 105

1 Introduction

This dissertation describes practical threats and solutions that pertain to an untrusted or malicious network infrastructure in (multi-tenant) data centers. A plausible scenario: national security agencies have bugged network equipment [207], networking vendors have left backdoors open [183, 80, 141, 34] and attackers have repeatedly demonstrated exploits on switches and routers [227, 222, 215, 121].

This is worrying and problematic for our critical infrastructure [87, 84] (which includes the Internet) for two reasons. First, several organizations (including governments) are either designing and building their own data centers [83] or migrating to public (multi-tenant data center) cloud providers [132, 6]. These data center networks typically outsource the control of the hardware and software switches (data plane) to a logically centralized controller (control plane), which could be a single point of failure. Second, these data centers are meant to host many users/tenants, applications and databases, oftentimes private [31] and personally identifying [12]. To host multiple tenant workloads and isolate them from each other, virtualization techniques are used [110], which if not done correctly could leak private information. Unfortunately, our understanding of the security implications of a malicious network infrastructure in such data centers is polarized.

Similar to Internet security research [146, 25, 114, 140], a lot of research effort has gone into studying the security of the centralized control plane (controller) [17, 174, 203, 108, 111]. The data plane has been assumed to be trusted and attacks arise from malicious end-hosts [76]. As a result, several authors have pointed out the weaknesses of a centralized controller: denial-of-service [29, 17, 48, 241], topology poisoning [76, 46], timing and reconnaissance attacks [210, 2, 123], as well as the lack of secure identifier bindings [95].

However, if the data plane is compromised, it has direct access to all traffic passing through it. Furthermore, its direct access to the centralized control plane, serves as a stepping stone to compromising the entire network [227]. Therefore, only recently have researchers begun to study a threat model [108, 46, 94, 224] where the data plane can be malicious. Dhawan et al. [46] and Hong et al. [76] demonstrated how malicious switches can poison the control plane's view of the physical network topology. Sasaki et al. [192] shed light on the lack of accountability in the data plane and Klöti et al. [108] conceptualized several attacks arising from a malicious data plane as well.

Given the limited grasp of such a pertinent threat model, we are motivated to cast light on what else attackers are capable of doing in today's data center networks and how we can defend ourselves against such a threat model. Therefore, for a deeper understanding of designing, building and operating a data center with potentially malicious network equipment we posit the following hypothesis.

Thesis Statement: Network-wide security policies or mechanisms in multi-tenant data center networks can be circumvented by malicious data plane systems. Hence, appropriate security measures need to be taken to tolerate compromised data planes in the network.

Methodology: Our approach to prove our hypothesis is to first understand how multi-tenant data center networks and their systems are designed, e.g., from specifications, designs and implementations. We then carefully analyze how the design and implementation decisions introduce vulnerabilities that enable an attacker to bypass security policies or enforcement mechanisms. Accordingly, we verify if the identified vulnerabilities can be discovered and demonstrated in real systems by manual, e.g., security analysis, or automated methods, e.g., fuzzing. We then address the uncovered security vulnerabilities by recommendation, design or implementation and evaluate whether the tradeoff between security, performance and resources is practical.

Contributions: Adopting the above methodology, this dissertation makes several contributions to i) the specifications, design and implementation of logically centralized controllers and ii) the design and implementation of multi-tenant network virtualization architectures. An overview of the scientific contributions are described below.

1. **Covert communication via the control plane:** Our theoretical and practical security analyses of logically centralized controllers and their protocols (e.g., OpenFlow and P4Runtime) uncovers three reasons why malicious switches can covertly communicate via the control plane and therefore bypass network-wide security policies and mechanisms: i) the malicious switches share the same logical controller ii) lack of authentication and authorization of switches to the controller in OpenFlow iii) automation and programmability of the network. We introduce four techniques to construct storage and timing covert channels and use them to demonstrate several simple and sophisticated attacks on state-of-the-art controllers, e.g., to circumvent firewalls or physically disconnected switches. We evaluate the performance and resource consumption of our storage channel and find that it can be reliable (using TCP) and fast (10 Mbps) with a small impact on the controller's CPU. By constructing a state transition and delay based model of OpenFlow's switch-controller handshake, we design, implement and evaluate a covert timing channel that uses a frame-based transmission scheme for accurate and low bandwidth communication (20 bps) e.g., to exfiltrate RSA private keys. Finally, we discuss several practical countermeasures.

2. **Improving isolation in the data plane:** Our systematic security analysis of multi-tenant network virtualization architectures that use virtual switches to enforce multi-tenant network isolation casts light on the following critical security issues. First is the trend to include increasingly complex network functionality in the virtual switch. Second, the virtual switch is co-located with the virtualization layer of the Host. Third, existing threat models for network virtualization do not sufficiently account for an attacker to compromise the data plane and the data center network. Using a case study (OpenStack and Open vSwitch), we fuzz the packet parsing logic of Open vSwitch and discover 3 memory corruption vulnerabilities. By exploiting one of them, we provably demonstrate how the state-of-the-art design is insecure: an attacker can not only escape virtualization but also take down an entire cloud in a

few minutes by targeting, compromising and propagating via the virtual switch. Our measurements of the impact of existing memory corruption protection mechanisms on network forwarding performance makes it clear that user-space forwarding is barely impacted (1-15% overhead) however, protecting the kernel is not practical for high performance requirements (reduces the maximum throughput by half).

By surveying the security posture of 23 virtual switches we conclude that nearly all of them lack security in their design: basic secure design principles such as least common mechanism, least privilege, small trusted computing base are absent. Therefore, we introduce four secure design principles that can prevent the virtual switch from being a liability to virtualization in the data center. To that end, we present `MTS`, it is scalable (using virtual machines and container), built from off-the-shelf hardware (Single-Root I/O Virtualization) and software (QEMU/KVM, Open vSwitch, DPDK) and incrementally deployable. `MTS` comes with the following benefits: i) reduces the trusted computing base of virtual networking, ii) inexpensive upgrade path for cloud operators and iii) increased security and performance. Our evaluation of the security, performance and resource tradeoffs on real workloads reveals a noteworthy improvement (1.5-2x) in throughput compared to the Baseline, with similar or better latency for an extra CPU.

The code developed to conduct our analyses, scripts to emulate the environments and data to back our experimental conclusions are available at the following URL:

`www.github.com/securedataplane/`

Tab. 1.1: List of assigned CVEs presented in this dissertation

Vendor	Vulnerability Description	CVE ID
ONOS	ONOS before 1.5.0 when using the ifwd app allows remote attackers to cause a denial of service (NULL pointer dereference and switch disconnect) by sending two Ethernet frames with ether_type Jumbo Frame (0x8870).	CVE-2015-7516
Open vSwitch	Buffer overflow in lib/flow.c in ovs-vswitchd in Open vSwitch 2.2.x and 2.3.x before 2.3.3 and 2.4.x before 2.4.1 allows remote attackers to execute arbitrary code via crafted MPLS packets, as demonstrated by a long string in an ovs-appctl command.	CVE-2016-2074
	In Open vSwitch (OvS) 2.5.0, a malformed IP packet can cause the switch to read past the end of the packet buffer due to an unsigned integer underflow in 'lib/flow.c' in the function 'miniflow_extract', permitting remote bypass of the access control list enforced by the switch.	CVE-2016-10037
OpenFlow	OpenFlow version 1.0 onwards contains a Denial of Service and Improper authorization vulnerability in OpenFlow handshake: The DPID (DataPath IDentifier) in the features_reply message are inherently trusted by the controller. that can result in Denial of Service, Unauthorized Access, Network Instability. This attack appear to be exploitable via Network connectivity: the attacker must first establish a transport connection with the OpenFlow controller and then initiate the OpenFlow handshake.	CVE-2018-1000155

Ethical Considerations and Practical Impact: To avoid disrupting production systems and businesses, all the findings reported in this dissertation have been verified on our own infrastructure. We also responsibly disclosed all the vulnerabilities we discovered as a result of this work (see Table 1.1 for the CVEs). The responsible/affected stakeholders have also acknowledged the severity and relevance of our findings e.g., via security advisories [190, 45] announcements [157, 40, 41] and press releases [149, 148]. Software vendors, e.g., RedHat, Citrix, Ubuntu, OpenStack, Mirantis, Suse, Open vSwitch, ONOS, Open DayLight, RYU and others have acknowledged our findings.

Dissertation Structure. The remainder of this dissertation is structured as follows. **Chapter 2** describes the architecture and relevant system components of multi-tenant data center networks that use a logically centralized controller and virtual switches. The threat model adopted in this dissertation is described in **Chapter 3**.

We then split the technical work of this dissertation into two major parts. In **Part I**, which includes **Chapter 4** and **Chapter 5**, we elaborate on our work that led to our contributions on covert channels via the control plane. Next, in **Part II**, which includes **Chapter 6** and **Chapter 7**, our work on isolation in network virtualization is extensively discussed. We highlight potential areas for future work in **Chapter 8** and draw our final conclusions in **Chapter 9**.

The Multi-Tenant Data Center 2

This chapter begins with a high-level architecture of a typical multi-tenant data center (MTDC) network, also commonly known as the *cloud* or *cloud networks*, followed by key components that make up and manage the network. We introduce relevant terminology and limit the scope of this chapter to the networking aspects of such a data center. Further information on specific topics are provided in the respective chapters of this dissertation.

2.1 Overview

A typical network topology for a multi-tenant data center is depicted in Figure 2.1. It is composed of hardware and software systems working together to host a variety of tenant workloads, e.g., MapReduce, Web servers, Key-Value Stores, ML (machine learning) applications and so on.

There are hardware servers (rectangles at the bottom) and switches (circles and the vswitch) that form the physical network topology. The switches are typically organized as three layers of leaf, spine and core switches. Additionally, a border switch provides connectivity between the internal and external network (e.g., the Internet). As shown in Figure 2.1, the physical servers connect to the leaf switches, which are in turn connected to the spine and core switches. For capacity (e.g., high bisection bandwidth) and availability (link failures) reasons, multiple links exist between the leaf, spine and core switches.

Multi-tenancy in this setting is typically offered via virtualization software (and hardware) running in the server known as the Hypervisor or Host operating system (OS). Tenants are allocated virtual machines (VMs) to share the underlying compute, memory and I/O resources (shown as different coloured boxes in the server). The respective tenant VMs form a virtual network that are isolated from other tenants via virtual switches that reside in the hypervisor. The virtual switch is the first and last hop along a tenant's network path within the data center network.

The switches (hardware and virtual) and servers are typically managed and configured from a logically-centralized controller. The centralized controller can be viewed as a software system that maintains a global view of the (physical and virtual) network and runs applications (e.g., traffic engineering, authentication, routing, firewalls) that configure the hardware and software switches according to the network and security policies specified by the operator and tenants.

In the next two sections we elaborate on centralized control and network virtualization.

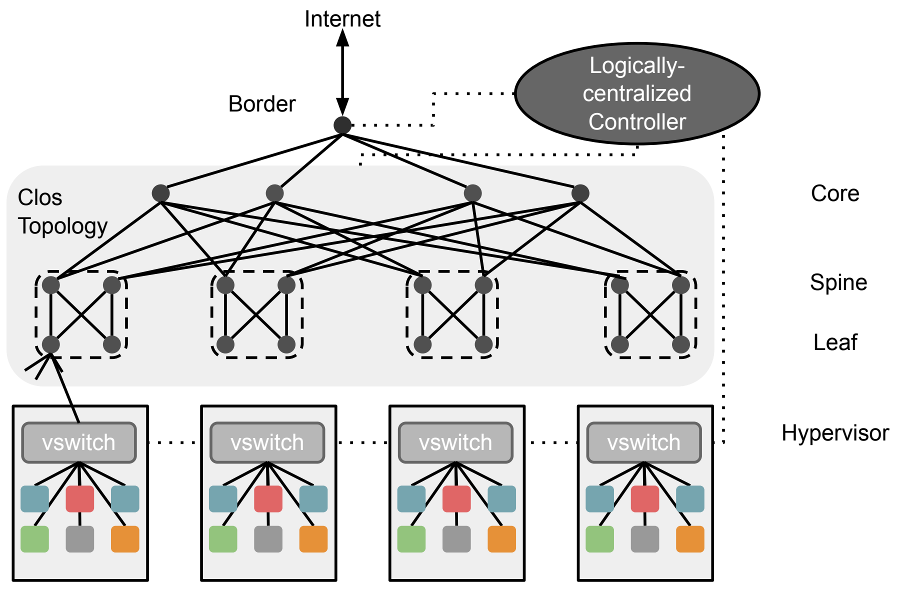

Fig. 2.1: Abstract representation of an MTDC network topology and the key components: (i) centralized controller to operate the physical and virtual network infrastructure, (ii) vswitch for network virtualization.

2.2 Software-Defined Centralized Control

Centralized network control and network-wide view are key design goals for state-of-the-art cloud operating systems such as OpenStack [243] and other commercial cloud providers, e.g., Google Cloud Platform [235] and Microsoft Azure [58]. The centralized control is attractive as it reduces the operational cost and complexity of managing cloud networks [231]. It also provides flexibility for managing and using cloud services, including VM migration.

This centralized network control is popularly known as *Software-Defined Networking* (SDN) and networks built using this design are called *Software-Defined Networks*. Unlike traditional networking wherein each switch/router has a data plane and control plane, in SDN, all the switches now *share* a logically centralized control plane as illustrated in Figure 2.1.

In SDN, OpenFlow [127, 150] is the de facto protocol used by the controller and switch for communication and programming e.g., forwarding rules. This interface is also known as the "south-bound" interface. OpenFlow follows a match-action paradigm: The controllers install rules on the switches which consist of a match and an action part; the packets (i.e., flows) matching a rule are subject to the corresponding action. That is, each switch stores a set of (flow) tables which are managed by the controllers, and each table consists of a set of flow entries which specify expressions that need to be matched against the packet headers, as well as actions that are applied to the packet when a given expression is satisfied. Possible actions include dropping the packet, sending it to a given egress port, or modifying its header fields, e.g., adding a tag. The match-action paradigm is attractive as it simplifies formal reasoning and enables policy verification. Via the OpenFlow API, the controller can add, remove, update, and monitor flow tables and their flows in the switches across the network.

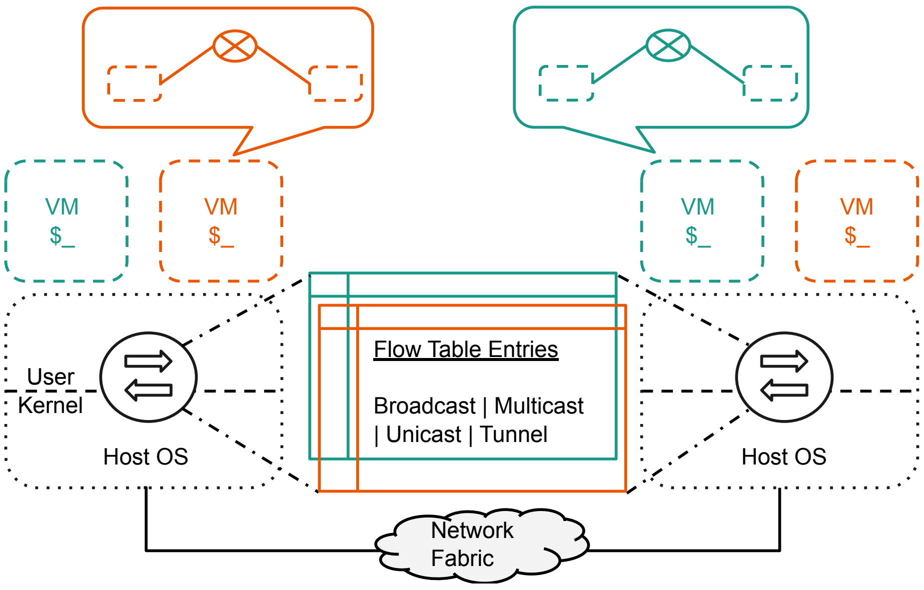

Fig. 2.2: High-level illustration of network virtualization using virtual switches in an MTDC. The vswitch isolates the green and orange tenant networks via tenant-specific flow table entries thereby giving each tenant the illusion of a single network.

While SDNs are logically centralized, the control plane can be physically distributed, e.g., for fault-tolerance or performance reasons. Accordingly, OpenFlow supports multiple controllers [150] for a single switch. The controllers and switch exchange `Role-request` and `Role-reply` messages respectively to assert the various roles (`Master`, `Equal` and `Slave`). There may be only one `Master` controller for a given switch while multiple `Equal` and `Slave` controllers are permitted. The OpenFlow standard [151] specifies basic security mechanisms. For example, the transport layer (TCP/UDP) communication between the controller and switch can be authenticated and encrypted, using TLS.

2.3 Network Virtualization

In a similar vein as compute resources are shared among the different tenants of the cloud, the physical network, in particular the data plane, is also a resource that is shared by the tenants. Tunneling protocols, e.g., VXLAN, GENEVE, NVGRE, etc., enable the cloud provider to create virtual networks (e.g., L2 virtual networks using VXLAN) that are then allocated to the tenants. The key component to network virtualization that implements the tunneling protocol, establishes VM connectivity and enforces isolation is the virtual switch, which is described next.

2.3.1 Virtual Switches

The virtual network's data plane(s) can either be distributed across virtualized servers or across physical (hardware) switches. Open vSwitch (OvS), VMware vSwitch, and Cisco Nexus 1000V are examples of the former and are commonly referred to as *virtual switches*

Tab. 2.1: List of virtual switches surveyed in this dissertation. `MTS` is introduced in this dissertation.

Name	Reference	Year	Emphasis
OvS	[169]	2009	Flexibility
Cisco NexusV	[236]	2009	Flexibility
VMware vSwitch	[239]	2009	Centralized control
Vale	[182]	2012	Performance
Research prototype	[98]	2012	Isolation
Hyper-Switch	[176]	2013	Performance
MS HyperV-Switch	[133]	2013	Centralized control
NetVM	[81]	2014	Performance, NFV
sv3	[211]	2014	Security
fd.io	[219]	2015	Performance
mSwitch	[75]	2015	Performance
BESS	[19]	2015	Programmability, NFV
PISCES	[198]	2016	Programmability
OvS with DPDK	[184]	2016	Performance
ESwitch	[137]	2016	Performance
MS VFP	[58]	2017	Performance, flexibility
Mellanox BlueField	[129]	2017	CPU offload
Liquid IO	[163]	2017	CPU offload
Stingray	[70]	2017	CPU offload
GPU-based OvS	[234]	2017	Acceleration
MS AccelNet	[59]	2018	Performance, flexibility
Google Andromeda	[44]	2018	Flexibility and performance
Slim	[251]	2019	Flexibility, Deployability and Security
MTS	[223]	2019	Performance, Security

(abbreviated as vswitch), and is what we focus on in this dissertation. Cisco VN-Link [35] and Virtual Ethernet Port Aggregator (VEPA) [101] are examples of the latter.

Multi-tenant virtual networks is typically provided in this design by (i) deploying the vswitches with the server's hypervisor (e.g, Open vSwitch aka OvS [171]), (ii) using *flow-table-level isolation*: the vswitch's flow tables are divided into per-tenant logical datapaths that are populated with sufficient flow table entries to link tenants' data-center-bound resources into a common interconnected workspace [110, 91, 168] and (iii) overlay networks using a tunneling protocol, e.g., VXLAN [237], to connect tenants' resources into a single virtual network/workspace. Alternatives to this host-based vswitch model [171], e.g., NIC-based vswitch solutions [99, 70] and FPGA-based designs [59], share the main trait that the logical datapaths have a common networking substrate (the vswitch).

The position of the virtual switch in the data center network is undeniably advantageous, it processes *all VM related network traffic*. Hence, much effort has gone into enhancing the performance, functionality and programmability of the virtual switch as illustrated in the Emphasis column in Table 2.1. For example, firewalling functionality has been introduced into OvS [89], the performance of OvS has been improved using DPDK [184] and Microsoft Azure introduced their hardware programmable virtual switch, VFP [58] and AccelNet [59] that offers high performance and programmability using FPGAs and extensible functionality in software.

Virtual switches are typically designed for centralized control, e.g., OvS, and support OpenFlow [147, 235, 44]. This enables management and configuration of the switch, e.g., configuring ports, policies, etc. from a logically centralized (OpenFlow) controller. The packet processing and forwarding functionality of the switch can be spread across the system running the virtual switch. The virtual switch can, but does not have to be separate processes. Moreover, it can either fully reside in user- or kernel-space, or be split across them.

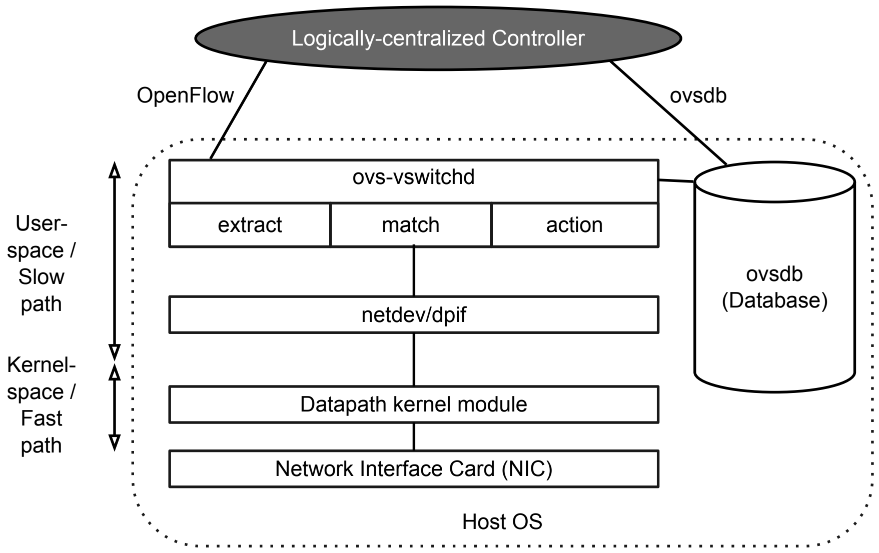

Fig. 2.3: High-level architecture of Open vSwitch.

Packet forwarding is usually based on a sequential (or circular) packet processing pipeline. The pipeline starts by parsing the packet's header to extract the information that is required, e.g., MAC and IP addresses, for a lookup of the forwarding instructions for that packet. The lookup is typically a (flow) table lookup—the second stage of the pipeline. The final stage uses this result to either forward the packet, drop it, or send it back to the first stage. These three steps are show as part of the components of `ovs-vswitchd` in Figure 2.3.

2.3.2 Open vSwitch

Open vSwitch [30, 171, 170, 216] is a popular open source SDN and multi-platform virtual switch. A high-level overview of its architecture is shown in Figure 2.3. The logically centralized controller can manage OvS using OpenFlow or `ovsdb`, a protocol designed for OvS that uses a database to store configuration information that is then used by `ovs-vswitchd` to update it's configuration.

OvS uses two forwarding paths: the slow path—a user-space daemon (`ovs-vswitchd`) and the fast path—a datapath kernel module (`openvswitch.ko`). A `dpif` (datapath interface) is used to communicate between the user-space and kernel module as OvS supports multiple OSes. `ovs-vswitchd` installs rules and associated actions on how to handle packets in the fast path, e.g., forward packets to ports or tunnels, modify packet headers, sample packets, drop packets, etc. When a packet does not match a rule of the fast path, the packet is sent to `ovs-vswitchd`, which then determines, in user-space, how to handle the packet. It then passes the packet back to the datapath kernel module to execute the action. If `ovs-vswitchd` does not know how to handle the packet, it can send it to the controller which will then send OvS instructions on how to handle the packet.

This concludes the necessary background information for us to introduce our threat model in the next chapter.

3 Threat Model

Our threat model is in the context of multi-tenant data center networks that use a logically centralized controller and virtual switches for virtual networking as described in the previous chapter. The main objective of the attacker is to carry out different attacks based on her requirements, e.g,. exfiltrate sensitive data, modify network data, launch denial of service attacks or even backdoor operating system images in data center server networks. We assume the attacker can be resource constrained however, this is not a strict requirement. The attacker could also be an insider, i.e., an authorized user who intends to subvert his/her current organization.

We assume the attacker has managed to compromise one or more switches (hardware/-software) used in the data center network, e.g., exploiting vulnerabilities in the switch software or via the supply-chain. In Chapter 6 we concretely describe how a (virtual) switch can be compromised by an attacker with limited resources by exploiting a buffer-overflow vulnerability in the MPLS protocol parsing logic. In order to collude, the attacker has programmed the switches to recognize special data and/or timing patterns to trigger covert communication. To control the compromised switches and compromise benign switches, we assume the attacker has network access to the victim switch via a physical or software link. Note that such connectivity is still via the data plane and not the management plane.

After the switch is compromised by the attacker we do not place any restrictions on what a malicious switch can and cannot do. For example, a malicious switch can fabricate and transmit any type of OpenFlow message, it can arbitrarily deviate from the SDN (e.g., OpenFlow, P4, P4Runtime) specification, and it can even use multiple identities, all at the risk of being detected. Similarly, we do not place any restrictions on what a malicious host can and cannot do. For example, a malicious host may masquerade its Media Access Control (MAC) and/or Internet Protocol (IP) addresses, use an incorrect gateway, falsify Address Resolution Protocol (ARP) requests/responses, and so on.

The controller and its applications on the other hand are trusted entities and are available to the switches: For example, they are based on static and dynamic program analyses, or developed in-house and run on trusted hardware. The controller-switch transport channel is reliable and may be encrypted (using TLS), however, authentication in the controller-switch protocol (e.g., OpenFlow) may not necessarily be specified and/or enforced.

Part I

Covert Channels via the Control Plane

In Part I of the dissertation, we demonstrate how malicious switches (and hosts) in a data center network can secretly communicate with each other via the *centralized* controller and bypass important data plane security policies and mechanisms, e.g., firewalls. Such *covert* communication is *fundamental* to networks with centralized controllers.

In Chapter 4, we describe *Teleportation*, a new vulnerability introduced by centralized control planes: Malicious switches (and hosts) can covertly communicate with each other by exchanging messages (e.g., OpenFlow messages) with the logically centralized controller. We identify four teleportation techniques that enables a broad spectrum of attacks for malicious switches: Circumventing data plane security mechanisms and isolation, attack coordination, exfiltration and eavesdropping. Via theoretical and practical analyses we explain how design, specification (OpenFlow) and implementation weaknesses can be leveraged by malicious switches for fast (10 Mbps) and reliable (TCP) covert communication using real software controllers (ONOS, OpenDaylight, RYU) and switches (Open vSwitch). Along the way, we also report on a denial-of-service attack on ONOS (`CVE-2015-7516`).

In Chapter 5, we extend work from the previous chapter by theoretically modelling switch identification teleportation using timing delays to construct a novel covert channel based on the OpenFlow switch-controller handshake. We then present the details on the design and implementation of our sender and receiver state transition diagrams, algorithms and implementations. Additionally, we evaluate the design, performance and accuracy of our covert channel experimentally. Our prototype implementation using ONOS and Open vSwitch, validates the feasibility of our channel: even under load at the controller, we can achieve 20 bits per second with a one-way communication accuracy of 90%. This work led to `CVE-2018-1000155` for the OpenFlow specification and a security patch to the P4Runtime specification. OpenFlow controller vendors have been alerted on the lack of authentication and authorization within the OpenFlow handshake and hence, require implementing those mechanisms with TLS.

4 Outsmarting Network Security with Teleportation

In this chapter we show how a (logically) centralized control plane—as it lies at the heart of the SDN paradigm, introduces an opportunity for *teleportation*: Malicious SDN switches may transmit information via the logically centralized software control plane, *completely* bypassing data plane elements (such as other switches, middleboxes, etc.). By violating logical and even physical separations, teleportation can constitute a serious security threat. For example, teleportation could be used by one malicious switch to discover (and communicate information to) other malicious switches, bypassing security checks in the data plane. As we will show in this chapter, teleportation can also be exploited by malicious hosts, triggering (benign) switches to teleport information for them.

We argue that teleportation can be seen as a flexible communication channel which constitutes a threat in various situations, for example (see Figure 4.1):

1. **Bypassing critical network components:** By implicitly communicating information via the control plane, it is possible to circumvent critical network components, such as switches, middleboxes or policy enforcers located in the data plane. For example, teleportation can in principle be used to bypass middleboxes performing security checks (e.g., network intrusion detection systems (NIDS)), middleboxes in charge of billing (e.g., a Radius server), or QoS enforcers (e.g., a leaky bucket policer).

2. **Rendezvous and attack coordination:** While already a single malicious switch, for example located *inside* a data center network, may cause significant harm and violate basic security policies, the situation becomes worse if multiple malicious switches cooperate [55]. Malicious or Trojan switches (e.g., switches containing a hardware/-software backdoor) may use teleportation as a rendezvous protocol, to discover each other, and subsequently coordinate an attack.

3. **Exfiltration:** Teleportation can also be used to exfiltrate sensitive information between networks that have no physical data plane connectivity.

4. **Eavesdropping and data tampering:** Particularly serious threats are introduced if malicious hosts and switches collude. For instance, in a scenario with collusion, teleportation can be used for eavesdropping. We show that a malicious switch and host can carry out a man-in-the-middle attack that serves benign hosts with malicious web pages.

Teleportation can be difficult to detect: The teleported information follows the normal traffic pattern of control communication, not between switches directly but *indirectly* between

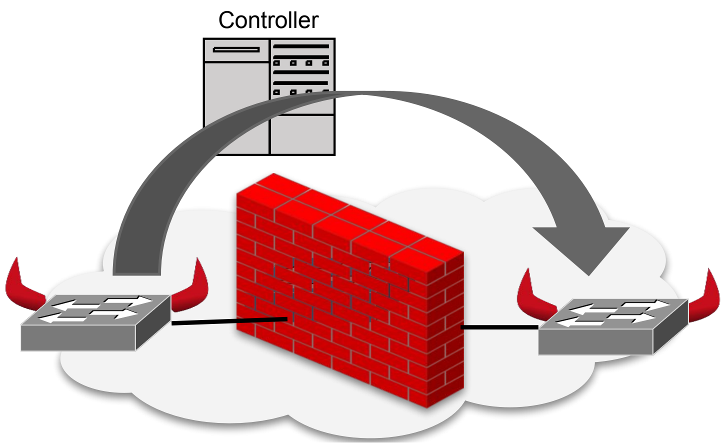

Fig. 4.1: Illustration of teleportation: Malicious switches (with *red horns*) exploit the controller for hidden communication, possibly bypassing data plane security mechanisms such as a firewall.

any switch and the controller. Moreover, the teleportation channel is *inside* the (typically encrypted) OpenFlow channel. Accordingly, it cannot easily be detected with modern NIDSs, even if they operate in the control plane.

The remainder of this chapter is organized as follows. Section 4.1 introduces the necessary background on OpenFlow and SDN. Section 4.2 characterizes possible teleportation channels. Section 4.3 describes teleportation techniques; based on these techniques, we demonstrate and discuss different attacks in Section 4.4. Section 4.5 describes our performance evaluation of the out-of-band forwarding channel. Section 4.6 initiates the discussion of countermeasures. After reviewing related work in Section 4.7, we conclude this chapter in Section 4.8.

4.1 Preliminaries

This chapter considers networks which outsource and consolidate the control over the network switches to a logically centralized software controller, e.g., in data center networks (recall Section 2.2). If a packet arrives at a switch and does not match an existing rule, the packet (usually without payload if the switch supports packet buffering) is forwarded to the controller. This event is called a `Packet-in`. Upon a `Packet-in` event, the controller can decide how to react to packets of the corresponding type by issuing `Flow-mod` messages to the switch (and maybe to other switches proactively on this occasion as well). For example, to add a flow rule, a `Flow-add` type Flow-mod message would be sent, and to delete a flow rule, a `Flow-del` type Flow-mod would be sent. The controller can also decide to send out a packet explicitly from a switch, issuing a so-called `Packet-out` command to the switch.

An attractive alternative to the hop-by-hop installation of new flows, i.e., reacting to a new packet repeatedly along the path (multiple `Packet-ins`), is the so-called *pave-path technique*: Once the controller receives a first `Packet-in` event from some switch, it proactively updates the other switches along the path. Such an *intent-based* controller behavior can render the reaction to network events and set up of host-to-host/network connectivity (according to current policies) more efficient.

Finally, we note that although some of our techniques are generally applicable in networks separating the control plane and the data plane, while others exploit OpenFlow specific features, when clear from the context, in this chapter we will treat SDN and OpenFlow as synonyms.

Threat Model. As described in Section 3, in this chapter we consider a threat model where OpenFlow switches, hosts (e.g., VMs, servers, PCs), or both, may not behave correctly but are malicious.

4.2 Modeling Teleportation

With these concepts in mind, we now model and characterize a novel threat called *teleportation* which targets the heart of SDNs: The outsourcing and consolidation of control over multiple data plane elements. In particular, we argue that we can see an OpenFlow controller as a "reactor": It reacts (in a best-effort and timely manner) to events generated by the network operator, the OpenFlow switches, and timeouts; as a response, the controller sends OpenFlow messages to switches. Accordingly, we argue that the following 3-stage functionality is fundamental in the SDN paradigm.

1. **Switch to controller:** A source switch intentionally or unintentionally sends modulated information to the controller (e.g., by adding specific events, delaying existing events, etc.).

2. **Controller to switches:** The controller reacts to the received events, by sending messages to one or multiple other switches.

3. **Destination processing:** A destination switch processes incoming messages from the controller. In case of a malicious switch, the switch may search for some message properties, temporal patterns, etc., and hence infer the information modulated by the source, or by simply forwarding the information (to a potentially malicious host).

Based on this controller model, we can identify two kinds of teleportation channels [1]:

- **Explicit teleportation:** Information actually appears in the messages exchanged. The message may for example contain steganographic contents.

- **Implicit teleportation:** Relies on modulating information implicitly. For example, it is based on timing (e.g., message transmissions are delayed according to some pattern) or it is based on shared resources, whose availability is changed over time (e.g., leveraging mutual exclusion).

[1] We note that our terminology of teleportation can be viewed as analogous to covert channels. Explicit teleportation is analogous to covert storage channels and implicit teleportation is analogous to covert timing channels.

4.3 Teleportation Techniques

Having established a conceptual model of teleportation, we next present techniques that can realize teleportation in today's SDNs. In particular, we have identified the following three fundamental SDN functionalities which can be exploited for teleportation:

1. **Flow (re-)configurations:** A controller needs to react to various data plane events (such as so-called `Packet-ins` in OpenFlow or link failures), and configure and reconfigure flows and paths accordingly. Triggering and exploiting such events can be used for teleportation.

2. **Switch identification:** In OpenFlow, switches are responsible for introducing and uniquely identifying themselves to the controller. This is required as policies are often specific to the switch. Unique switch identifiers are also necessary to correctly construct and enforce policies on the switches and in the controller. We will show that such switch identification mechanisms can be exploited for teleportation.

3. **Out-of-band forwarding:** An SDN controller must not only be able to receive events and control packets from switches, but also to instruct switches to forward specific messages. This basic functionality in SDNs can be exploited by a malicious switch or host to forward entire packets via the controller.

In the remainder of this section, we will discuss these teleportation techniques in more detail in turn.

4.3.1 Flow (Re-)Configurations

We distinguish between two types of flow re-configuration events: path update and path reset.

Path Update. Our first teleportation technique is based on *path updates*. Path updates are a fundamental controller functionality, and come in the form of different controller features such as *Mobility*, *VM Migration* or simply *MAC Learning*. The basic scheme is as follows: A controller typically maintains some mapping of which hosts (MAC addresses) are connected to which ports (on the switch). If a host suddenly appears on another switch, the controller installs new flows for the host on the new switch, and also deletes the corresponding flow rules on the old switch. We define this *programmatic* installation and deletion of flows by the controller on switches as path update. Specifically, a path update involves the use of `Packet-in`, `Flow-mod` and `Packet-out` messages. Malicious switches can use path update for implicit teleportation.

For teleportation with path update, a switch triggers the deletion of rules at other switches. Malicious switches can teleport information between themselves by prompting path updates for the same host using `Packet-ins`. Note that during a path update, the `Packet-out` is to be sent to the destination reported in the `Packet-in` which may generate data plane

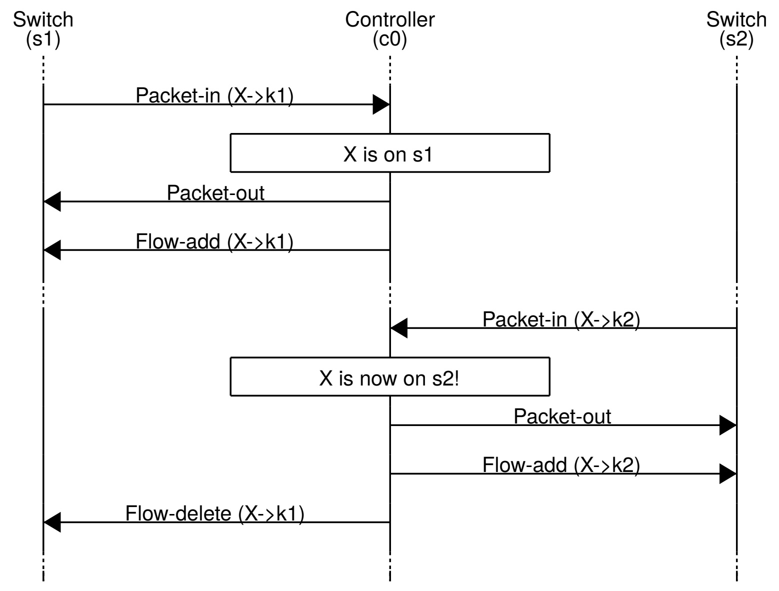

Fig. 4.2: Message sequence pattern for path update teleportation. Switch $s2$ teleports information to $s1$ when $s1$ receives the Flow-delete message from controller $c0$.

Algorithm 1: Generalized pseudo-code executed by switch s_i to teleport information using path update.

```
1 process incoming OpenFlow message :
2     on start teleportation :
3         - announce all $\{X_{i,j}\}_{j\in[m]}$ and $\{X_{j,i}\}_{j\in[m]}$
4     on received Flow-delete for $X_{i,j}$ for some $j \in [m]$ :
5         - announce $X_{i,j}$
6         - add $j$ to $Discovered\_Switches$
```

traffic. To prevent data plane traffic, the malicious switch can use a destination host that is connected to itself (so that the `Packet-out` is sent back to it). The message sequence pattern for path update teleportation is shown in Figure 4.2.

We can summarize the scheme presented so far with the following abstract steps: A switch $s1$ announces X, shortly thereafter a switch $s2$ announces X thereby stealing X from $s1$, where stealing is detected by the “victim" ($s1$). Also note that announcing is possible once some host which is connected to the malicious switch (e.g., $k1$ at $s1$) is learned by the controller.

Based on these basic steps, we can generalize our scheme for m malicious switches. Each malicious switch, s_i, with id $i \in [m]$, should implement an event handler (see Algorithm 1), in addition to the normal (non-malicious) behavior. We assume that all switches are programmed with the same list of m^2 special MAC addresses $\{X_{i,j}\}_{i,j\in[m]}$ (the *pre-shared secrets*). Note that once switch i discovers switch j, it can contact it by sending packets with source $X_{i,j}$ and destination address $X_{j,i}$.

Path Reset. We next discuss a second flow reconfiguration called path reset. Recall that at the heart of any SDN controller lies the functionality to set up host-to-host/network

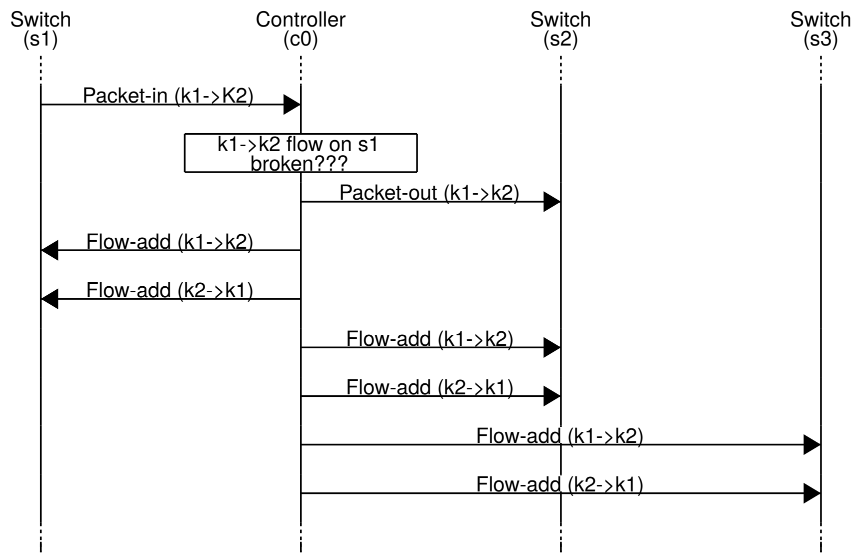

Fig. 4.3: Message sequence pattern for path reset teleportation. Switch $s1$ teleports information to $s2$ and $s3$ via `Flow-add` messages sent by the controller $c0$.

connectivity, according to the network policy (e.g., defining constraints such as bandwidth, link type and waypoints), which is translated into device level configurations (e.g., flow rules). The "pave-path technique", a proactive and programmatic approach to network configuration is an attractive alternative to the hop-by-hop installation of new flows: Once the controller receives a first `Packet-in` event from some switch, it proactively updates the other switches along the path of the pair of hosts that want to communicate.

For high availability, a controller also monitors the network state and makes necessary changes, such as rerouting or resetting flows on switches, when needed (e.g., due to a link failure). For example, triggered by a `Packet-in` event, a controller may learn that (parts of) the path may no longer be available, and hence initiates the reconfiguration/repair of the path. We define the reinstallation of flows by the controller on switches along a path as path reset. Accordingly, the path reset technique involves `Packet-in`, `Flow-mod` and `Packet-out` messages.

Malicious switches may use path reset for implicit teleportation: If the controller resets the complete path between hosts when it receives a `Packet-in` from a switch that ignores the flow rule, then information can be communicated. By doing this at multiple and specific times, a malicious switch can teleport information to other malicious switches along the path. Figure 4.3 illustrates the message sequence pattern for teleportation using path reset.

4.3.2 Switch Identification

This teleportation type exploits the fact that a switch typically must uniquely identify itself whenever it connects to the controller. For example, in OpenFlow this is usually done using the Datapath-ID (DPID) field in the `Features-reply` message. We define two switches attempting to use the same DPID to connect to the same logical controller as switch identification. The outcome can be used for implicit teleportation. In the chapter that follows

Algorithm 2: Generalized pseudo-code executed by switch s_i for switch identification teleportation when the controller denies the second switch a connection.

```
1 process connect to OpenFlow controller :
2     on Features-request message from controller :
3         - announce DPID $\{X_{i,j}\}_{j \in [m]}$ in Features-reply message
4     on Controller denies connection to announced DPID $X_{i,j}$ for some $j \in [m]$ :
5         - add $j$ to $Discovered\_Switches$
```

(Chapter 5), we describe how this technique can be used to transfer data between two switches.

Four basic ways an OpenFlow controller can react to using the same DPID are as follows:

1. The controller denies the second switch a connection.

2. The controller terminates the first switch and connects to the second.

3. The controller accepts both switches.

4. The controller accepts both switches but sends them different `Role-request` messages.

Only in the first, second and fourth outcomes can the malicious switches infer if the DPID it used is already in use by another switch. By using a-priori configured single or multiple DPID values, a pair of malicious switches can establish teleportation. For example, consider the message sequence pattern in Figure 4.4, and assume that first switch $s1$ tells controller $c0$ that its DPID is 1. At a later time, switch $s2$ tells $c0$ that its DPID is 1. At this point, $c0$ does not allow $s2$ to connect with DPID 1. Since $c0$ denied $s2$ to connect with DPID 1, $s1$ teleported information to $s2$ via $c0$. With a similar message sequence pattern, the second outcome can be used for teleportation as well.

Interestingly, switch identification is not limited to scenarios with a single controller: We have found additional threats in the presence of *distributed control planes*. The message sequence pattern that uses the Role-Request messages for multiple controllers is shown in Figure. 4.5. Moreover, we can generalize the first switch identification outcome to scenarios with m malicious switches, see the event-handler algorithm, Algorithm 2. The other two outcomes discussed can also be seen as event-handler algorithms.

4.3.3 Out-of-band Forwarding

The fourth and potentially most powerful teleportation technique is called out-of-band forwarding. It is an example of explicit teleportation. Out-of-band forwarding exploits the fact that an SDN controller is typically connected to multiple switches: Accordingly, a packet from one switch can potentially reach multiple other switches in the network via the control plane. Out-of-band forwarding involves a `Packet-in` from one switch and a `Packet-out` message at another switch, with the possible side effect of `Flow-mod` messages on the

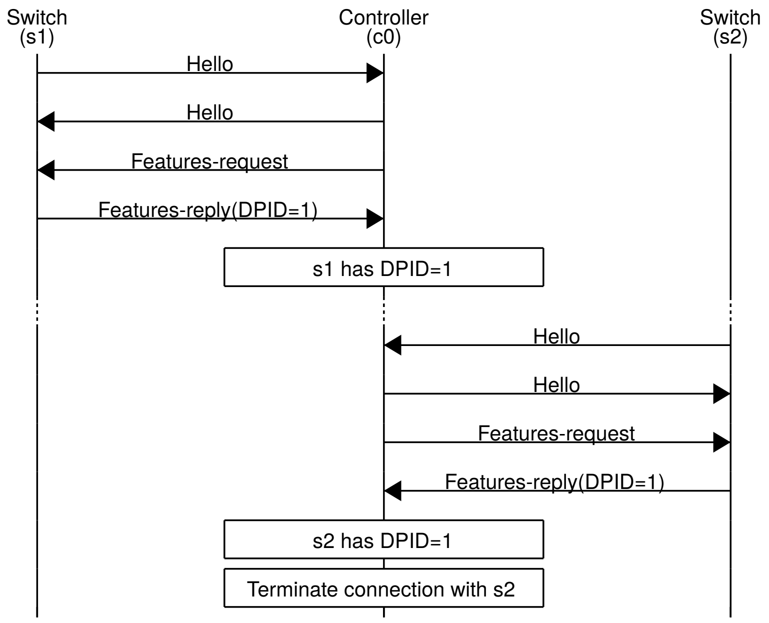

Fig. 4.4: Message sequence pattern for switch identification teleportation when the controller denies the second switch a connection. When $s2$'s connection is terminated, $s1$ successfully teleports information to $s2$.

Tab. 4.1: Summary of teleportation techniques, types, messages and associated threats.

Technique	Type	Messages	Threat
Flow (Re-)Configuration	Implicit	Packet-in, Flow-add, Flow-del, Packet-out	Covert communication and coordination.
Switch Identification	Implicit	Hello, Features-request, Features-reply, TCP FIN, Role-request, Role-reply	Covert communication and coordination.
Out-of-band Forwarding	Explicit	Packet-in, Packet-out	Exfiltration, firewall/NIDS bypass and man-in-the-middle.

switch that sent the `Packet-in` message. Out-of-band forwarding could for example include the complete Ethernet frame (typically 1500 bytes), and can even serve as a "multicast service". Out-of-band forwarding can be a serious threat to network security, not only because malicious traffic can bypass critical security functions in the data plane, but also because it can be exploited by switches and hosts. Figure 4.6, illustrates the message sequence pattern for teleportation using out-of-band forwarding.

Summary. We summarize the three teleportation techniques we just explained along with their type, OpenFlow messages and potential threats each of them pose to network security in Table 4.1.

4.4 Switch- and Host-based Attacks

We now demonstrate how the identified teleportation techniques can be exploited to carry out specific attacks. In particular, we show how teleportation may be exploited:

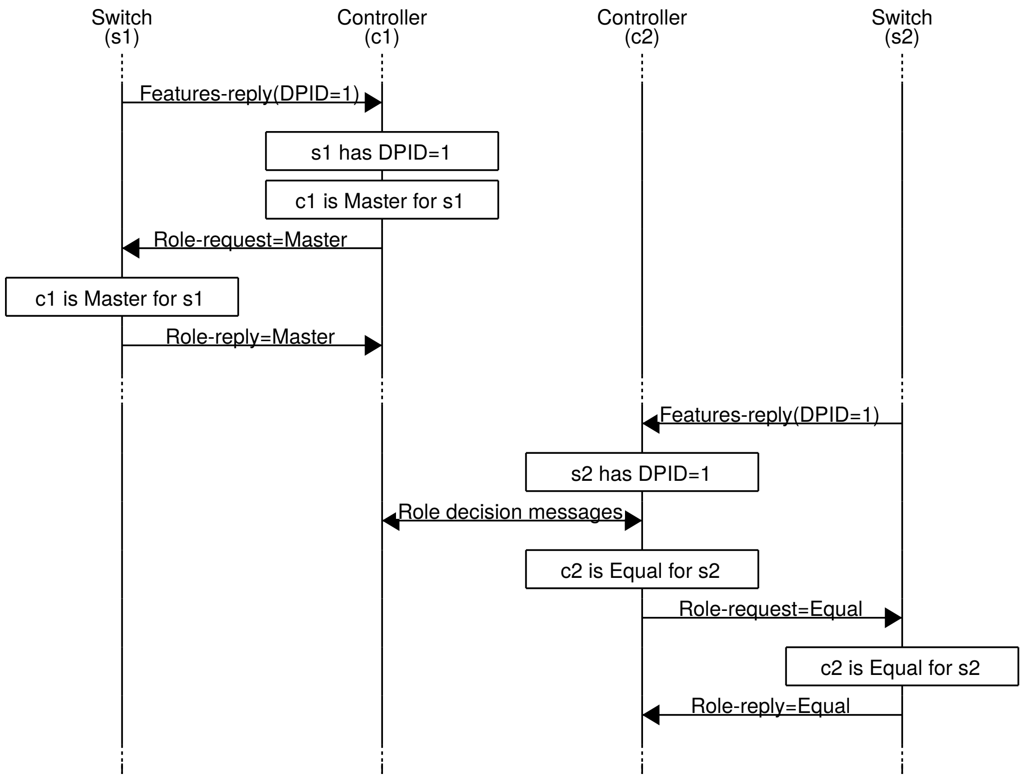

Fig. 4.5: Message sequence pattern for switch identification teleportation when controllers $c1$ and $c2$ send different Role-request messages to $s1$ and $s2$ respectively. When $s1$ receives the `Role-request=Master` message whereas $s2$ receives the `Role-request=Equal` message. In this manner $s1$ teleports information to $s2$ when $s2$ received the `Role-request=Equal` message.

1. To bypass security critical network functions such as firewalls and NIDSs;
2. As a rendezvous protocol for malicious switches;
3. To exfiltrate sensitive data from remote locations;
4. To conduct a man-in-the-middle (mitm) attack.

Along the way, we also present a novel *Denial of Service*(*DoS*) attack (published as `CVE-2015-7516` [144]). Before presenting the attacks in more detail, we report on the setup we used to verify the attacks. Our evaluation environment is available online at the following URL:

`www.github.com/securedataplane/teleportation`

Setup. We verified all our attacks in a virtual machine, using Mininet-2.2.0 and Open vSwitch-2.0.1 for the data plane. For the control plane we used ONOS-1.1.0 as it was the state-of-the-art. At the time of our experimentation Floodlight, OpenDaylight Lithium-SR2, and RYU v3.27, still did not support the intent based framework. Indeed our experiments showed that they were only vulnerable to a subset of the attacks (e.g., switch identification,

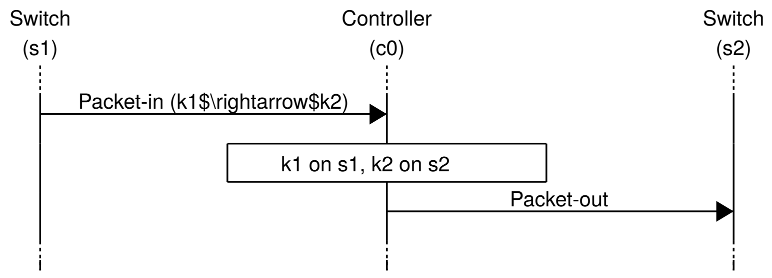

Fig. 4.6: Message sequence pattern for out-of-band forwarding teleportation. The controller $c0$ receives the `Packet-in` from $s1$ and accordingly sends a `Packet-out` to $s2$, successfully teleporting packets from $s1$ to $s2$.

out-of-band-forwarding[2]). For packet generation we use ping and nmap-6.40. We use `ebtables` v2.0.10-4 (December 2011) as our transparent firewall and `Snort` version 2.9.6.0 GRE (Build 47) as our NIDS. We modified code developed by `austinmarton` [197] to set the Ethertype field in an Ethernet frame. `ettercap` 0.8.0 was used with a custom HTTP filter for the mitm attack.

4.4.1 Bypassing Critical Network Functions

We believe that the possibility to bypass network elements is a serious threat in modern computer networks. For example, many network policies today are defined in terms of adjacency matrices or big switch abstractions, specifying which traffic is allowed between an ingress port s and an egress network port t [103]. In order to enforce such a policy, traffic from s to t needs to traverse a middlebox instance (waypoint) inspecting and classifying the flows. The location of every middlebox may be optimized, but is subject to the constraint that the route from s to t should always go via the waypoint.

Firewall and NIDS. In order to demonstrate how a firewall may be circumvented by hosts (or switches), we set up Mininet and ONOS as shown in Figure 4.7. The switches do not have flow rules for $k1$ and $k2$ to communicate. The firewall $fw1$ prevents hosts on the left to communicate with hosts on the right and vice-versa. ONOS has the *Intent Reactive Forwarding* (`ifwd`) application enabled. `ifwd` uses the reactive "pave-path technique" (recall Path Reset from Section 4.3.1) to install flows in the switches. By default, the `ifwd` application establishes host-to-host connectivity when it receives a `Packet-in` for which no flows exist.

We send a ping packet from $k1$ to $k2$. Despite the presence of the firewall, $k1$ receives the reply from $k2$ using out-of-band forwarding teleportation. In the absence of out-of-band forwarding teleportation, the packet would have been dropped by $fw1$.

Indeed, in this case, out-of-band forwarding teleportation has the side effect of installing flows on $s1$ and $s2$ for $k1$ and $k2$ to communicate, preventing further out-of-band forwarding teleportation. By masquerading its MAC address, $k1$ can teleport more data to $k2$ via out-of-band forwarding teleportation.

[2] `https://goo.gl/FN9ULQ`

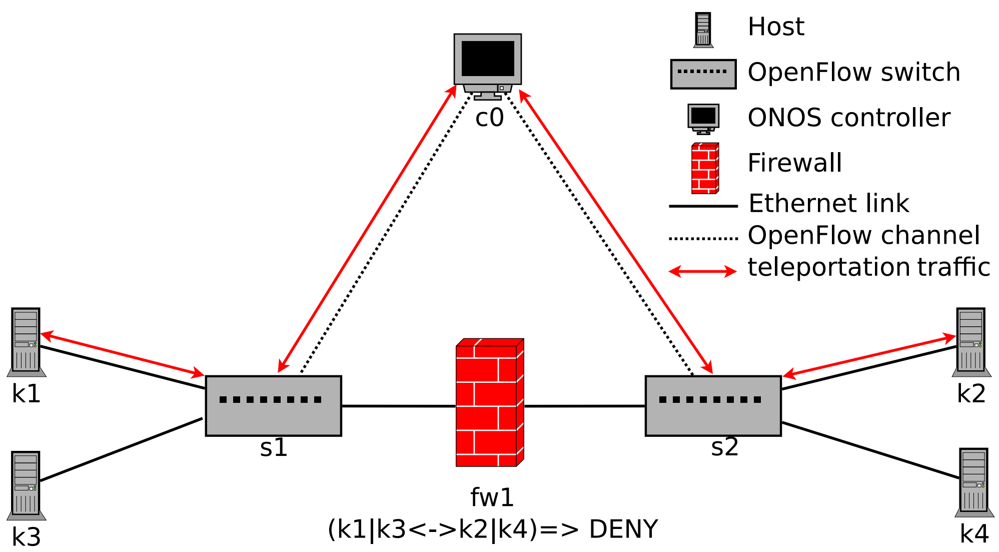

Fig. 4.7: An SDN topology with OpenFlow switches $s1$ and $s2$ and an OpenFlow controller $c0$ (ONOS). $k1$ and $k3$ are connected to $s1$ while $k2$ and $k4$ are connected to $s2$. $s1$ and $s2$ are separated by a firewall $fw1$ that denies hosts on $s1$ to communicate with hosts on $s2$ and vice-versa. $k1$ can use out-of-band forwarding teleportation to transfer data to $k2$, bypassing $fw1$.

Similar to the firewall scenario, we can also use out-of-band forwarding teleportation in the presence of `Snort`, an NIDS. In particular, we can generate attack traffic using nmap to conduct TCP flag attacks or even port scans. Indeed, by masquerading the source MAC address, one can effectively carry out a wide enough port scan without having the scan pass through the firewall and being detected by the latter.

By replacing the firewall we previously described with `Snort`, we use nmap from $k1$ to carry out a TCP port scan on $k2$ using out-of-band forwarding teleportation. By inspecting the alerts in `Snort` we verified that no alerts were generated for the port scan.

Note that the host-to-host connectivity setup involves a `Packet-in` and `Flow-mod` messages whereas the out-of-band forwarding teleportation only involves `Packet-in` and `Packet-out` messages with the side effect of `Flow-mod` messages. Therefore, security policy enforcers that do not inspect and correlate `Packet-in` with `Packet-outs`, will miss out-of-band forwarding teleportation based attacks. Of course, violating `Flow-mods` may eventually be detected, but only after the data has been teleported.

As illustrated in Fig. 4.7, it is not always possible for the controller to manage the firewall, e.g., firewalls can run inside tenant VMs. If the vswitch or the hardware switches (e.g., leaf, spine, core) are compromised, they can use such a technique to bypass tenant-specific security policies and mechanisms.

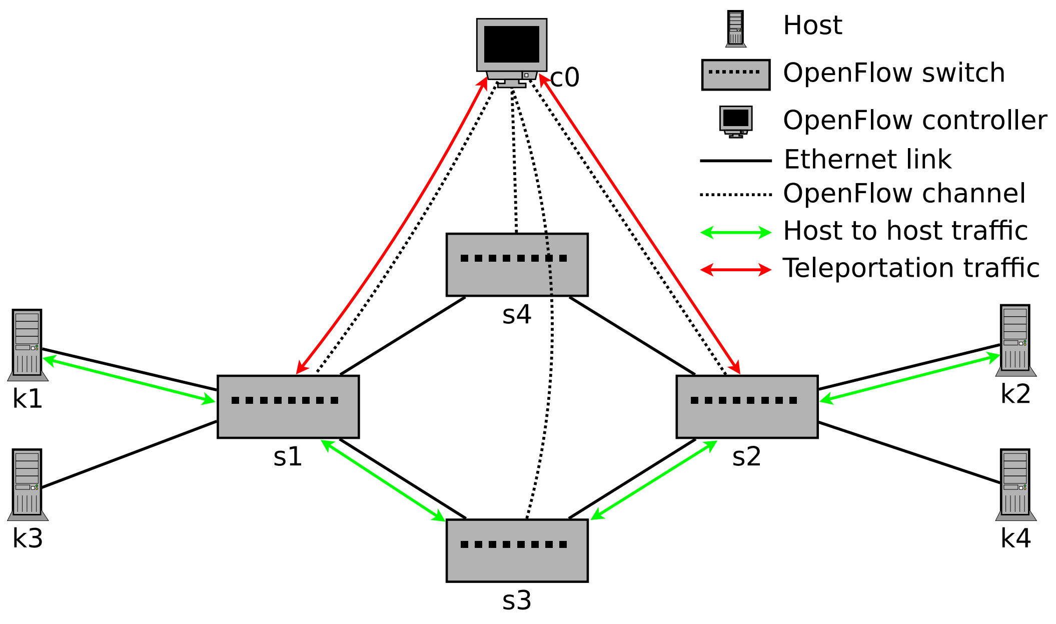

Fig. 4.8: An SDN topology of OpenFlow switches $s1$, $s2$, $s3$ and $s4$, OpenFlow controller $c0$ (ONOS). Hosts $k1$ and $k3$ are connected to $s1$ and $k2$ and $k4$ are connected to $s2$. $c0$ has installed flows on $s1$, $s3$ and $s2$ so that $k1$ and $k2$ can communicate bi-directionally. Teleportation traffic is via $c0$.

4.4.2 Rendezvous and Malicious Switch Discovery

We next consider a rendezvous protocol in which malicious switches wish to discover one another. A rendezvous or discovery protocol can be also seen as a precursor to a much more damaging attack such as a *DoS*, man-in-the-middle (*mitm*) or exfiltration. A rendezvous protocol can rely on steganography, i.e., embedding patterns in teleported benign information or modulating patterns in legitimate messages. Without teleportation, by going through the data plane directly, the malicious switches risk detection. We show how three of our techniques, namely path update, path reset and switch identification teleportation may be used as a rendezvous protocol for malicious switches.

4.4.2.1 Path Update

To demonstrate a rendezvous with path update teleportation, we set up Mininet and ONOS as shown in Figure 4.8. Instead of instrumenting code for the malicious switches, we keep them as simple Open vSwitches and we defined dedicated Mininet hosts ($k3$ and $k4$) for each of them. We use the dedicated hosts ($k3$ and $k4$) to generate the packet that the malicious switch sends as a `Packet-in` to the controller. The `host mobility` and `ifwd` applications are enabled on ONOS. The controller has already installed flows for $k1$ to $k2$ and vice-versa. Accordingly, we use $k4$ connected to $s2$, to send $k2$ a packet using $k1$ as the source MAC address. This triggers the controller to issue `Flow-mod` commands to $s1$, $s2$ and $s3$. $s2$ thereby teleported its presence to $s1$.

By inspecting the flows on the switches, we verified the successful path update teleportation: $s2$ was able to cause a flow deletion in $s1$ without exchanging any packets with $s1$ directly (except for a normal flow in the past).

Note that path update may trigger alerts in systems that keep track of moving MAC addresses by inspecting `Packet-in` and `Flow-mod` messages. In such cases, many moving MAC addresses may introduce suspicious activity within the network. Also worth noting is that port-based security (that associates MAC addresses with specific ports) may not be applicable in the presence of malicious switches (recall our threat model in Chapter 3).

4.4.2.2 Path Reset

To demonstrate that path reset teleportation can be used as a rendezvous protocol, we consider the same setup as outlined in Section 4.4.2.1. We modulate traffic between $k1$ and $k2$ using ping packets with 100 microsecond intervals. Instead of manipulating the Open vSwitch code for sending a `Packet-in` for an existing flow from $s1$, we simply remove the flow for $k1$ to $k2$ on $s1$, using the `ovs-ofctl del-flow` command. This causes $s1$ to send $c0$ a `Flow-removed` message which triggers the controller to add the flow back onto $s1$. But due to the high rate of ping traffic, at least one packet triggers a table-miss before $s1$ adds the flow and a `Packet-in` is sent to $c0$.

When $c0$ receives the `Packet-in` it sends the packet to $s2$ directly as a `Packet-out`, bypassing $s3$, and then sends `Flow-mods` to $s1$, $s2$ and $s3$ resetting the bi-directional path between $k1$ and $k2$. By checking the lifetime of the flow rules on $s1$, $s2$ and $s3$ we verified that path reset teleportation succeeded. In this manner, $s1$ teleported its presence to $s2$ by having the controller send `Flow-mod` commands for existing flow rules.

Note that such an attack works in the presence of topology spoofing defenses [46, 77] as the `Packet-in` and `Flow-mod` messages generated do not alter the existing topology. Indeed, receiving a `Packet-in` for a flow that exists in the switch is suspicious but we are not aware of any work that keeps track of such events.

4.4.2.3 Switch Identification

We now demonstrate how two malicious switches may teleport their presence using switch identification. In the next chapter we describe how this technique can also be used for data exfiltration. We set up Mininet and ONOS as shown in Figure 4.8 with only $s1$, $s3$ and $s4$ having connected to $c0$ with DPID 1, 3 and 4 respectively. Also, there are no flows installed on the switches for hosts to communicate. We modified the Mininet script to configure $s2$ with the same DPID as $s1$.

When $s2$ tries to connect to $c0$ with DPID 1 after $s1$ has connected to $c0$, it is denied a connection. This way, $s1$ teleports its presence to $s2$.

In Floodlight and OpenDaylight, when $s2$ attempts to connect to $c0$ with DPID 1 after $s1$ has connected, Floodlight terminates the connection with $s1$ and accepts $s2$'s connection. $s2$

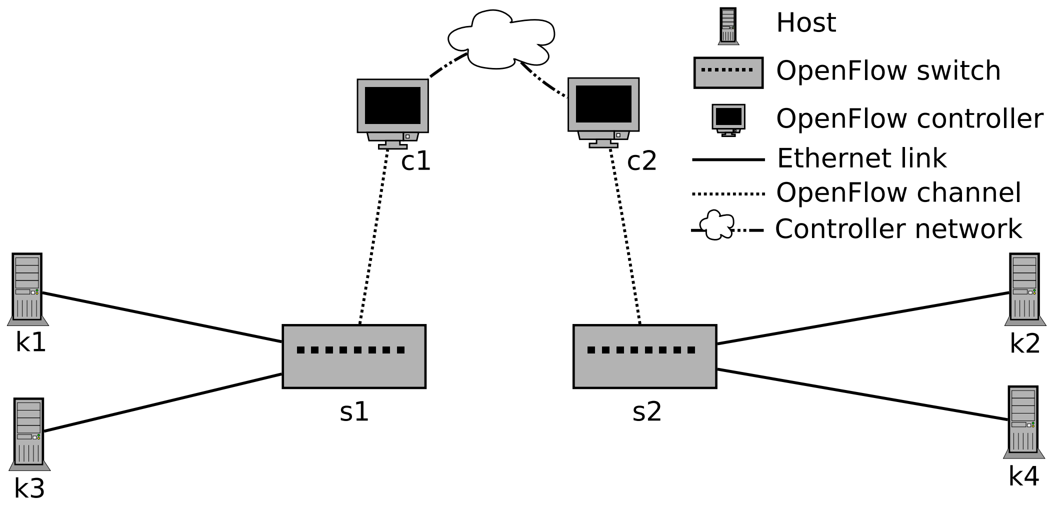

Fig. 4.9: An SDN topology with independent OpenFlow switches controlled by independent OpenFlow controllers (ONOS). $c1$ and $c2$ share and synchronize state information via an independent controller network. $s1$ is controlled by $c1$ and $s2$ is controlled by $c2$ respectively.

thereby teleports its presence to $s1$. Interestingly RYU allowed switches with the same DPID to co-exist which potentially introduces additional issues.

Switch identification teleportation is also possible when multiple controllers manage independent switches. We set up Mininet and ONOS as shown in Figure 4.9. Initially $s1$ connects to $c1$ with DPID 1. $c1$ then declares itself as the `Master` for $s1$. At a later time, $s2$ connects to $c2$ and claims to have DPID 1. $c2$ then sends $s2$ the `Equal` role. In this manner, $s1$ teleports its presence to $s2$. By inspecting the OpenFlow channels, we verified the different `Role-request` messages sent by the respective controllers to their respective switches.

Here we can see that when using automation and programmability, if the controller is not aware of global isolation policies then it cannot correctly enforce the desired isolation.

4.4.3 Exfiltration

Our next attack is related to data exfiltration. This is a key concern for many organizations that own intellectual property, personal data or any kind of sensitive information. Once an attacker gets into a network, one possible goal of the attacker is to stealthily exfiltrate sensitive data.

We demonstrate exfiltration by considering a scenario where a small number of hosts are networked together in a remote location. The data plane isolation is meant to improve security. However the data plane elements are managed by a controller that handles other similar remote locations. We show that in such a network, not only malicious switches can exfiltrate data using out-of-band forwarding teleportation but even malicious hosts.

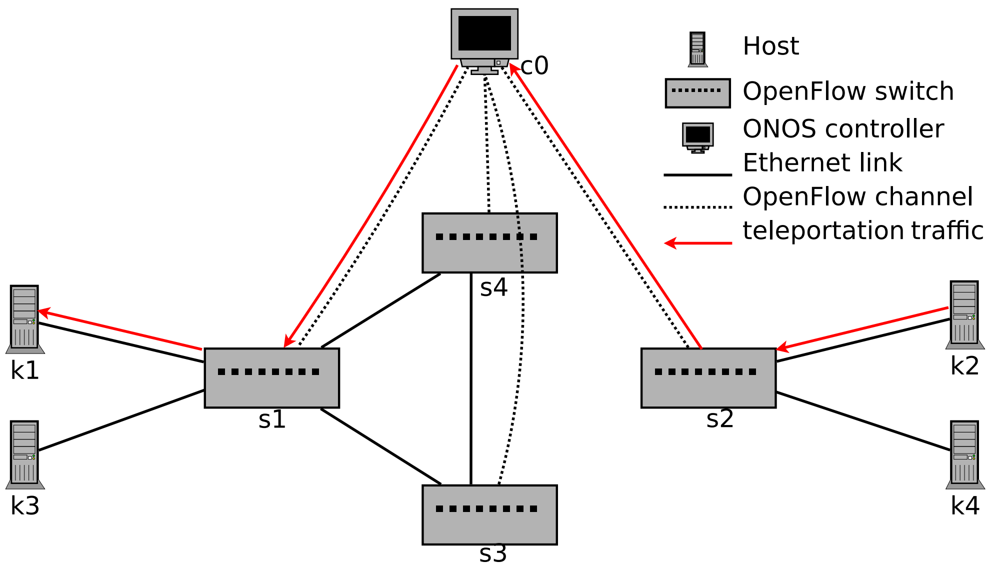

Fig. 4.10: An SDN topology with OpenFlow switches $s1$, $s2$, $s3$ and $s4$ and an OpenFlow controller $c0$ (ONOS). $k1$ and $k3$ are connected to $s1$ while $k2$ and $k4$ are connected to $s2$. Note that $s2$ is not connected to the other switches, and thereby is isolated in the data plane. $k2$ can still exfiltrate data to $k1$ using out-of-band forwarding teleportation circumventing the data plane isolation.

We set up Mininet and ONOS as shown in Figure 4.10. ONOS has the `ifwd` application activated. By showing how $k2$ can exfiltrate data to $k1$, we also demonstrate how $s2$ can exfiltrate data to $k1$ or $s1$.

Given that $s1$ and $s2$ do not have flow rules for traffic from $k2$ to $k1$ (as they are located in disconnected data planes), $k2$ can exfiltrate data to $k1$ by simply sending a packet (e.g., UDP packet) to $k1$ thereby exploiting out-of-band forwarding teleportation. The controller will receive the packet from $s2$ and send it to $s1$ which will then forward the packet to $k1$.

By inspecting the OpenFlow channels, we can see the out-of-band forwarding teleportation, first as a `Packet-in` and then as a `Packet-out`.

4.4.4 Evading Policy Conflicts

For an attacker, remaining stealthy is key to persistent existence. One of the side effects of using the out-of-band forwarding teleportation is the `Flow-mod` messages issued by the controller. The `Flow-mod` messages may generate policy conflicts (unauthorized/conflicting flow rules), alerting the administrator. A stealthier version of using the out-of-band forwarding teleportation would be to prevent the `Flow-mod` side effect. This would not only prevent policy conflicts, but also leave minimal traces on the source and sink switches.

In order to demonstrate this attack, we set up Mininet and ONOS with `ifwd` activated as shown in Figure 4.10. $k2$ can exfiltrate data to $k1$ using out-of-band forwarding teleportation without triggering `Flow-mod`'s on $s2$ and $s1$ by masquerading its source MAC address *and* ETHER_TYPE (e.g., Jumbo frame: 0x8870).

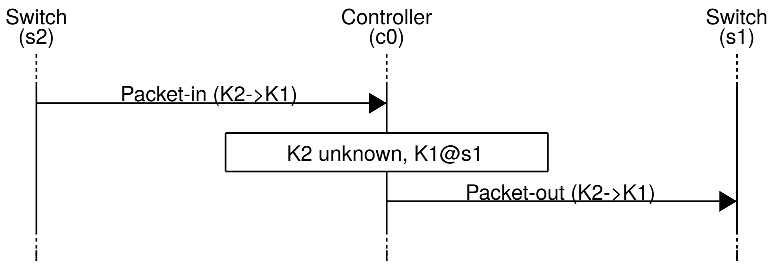

Fig. 4.11: The message sequence pattern for evading policy conflicts using out-of-band forwarding teleportation. The side effect of `Flow-mod` messages are avoided when Jumbo frames are used from a masqueraded MAC address; only `Packet-ins` and `Packet-outs` are used.

If the packet processor and intent framework cannot correctly identify a packet, their behavior may violate security policies. Note that it is enough if the ETHER_TYPE is set to a value that ONOS does not recognize, and we are not restricted to Jumbo frames only. The message sequence pattern for out-of-band forwarding teleportation without the `Flow-mod` side effect is shown in Figure 4.11.

By inspecting the OpenFlow channels, we can verify that the packet was indeed teleported via out-of-band forwarding teleportation first as a `Packet-in` and then as a `Packet-out`. By inspecting the flows on the switches, we can verify that no new flows are present.

Remark on a Denial-of-service Attack. Interestingly, we observed that a side effect of our out-of-band forwarding teleportation is a novel denial-of-service attack. If in our evading policy conflicts example, the host sends the same packet (Jumbo frame) again, then `ifwd` encounters a null-pointer exception and disconnects the switch that sent it the packet. This shows how a malicious host can cause the switch it is connected to, to be disconnected from the controller even when a packet it sends is not corrupted.

We emphasize that this is a side effect of out-of-band forwarding teleportation only, and not a teleportation issue in itself. Fortunately, the issue has been resolved by the ONOS community after we contacted them (published as `CVE-2015-7516` [144]).

4.4.5 Man-In-The-Middle

While we have so far focused on attacks where either only switches or only hosts are malicious, we now detail an attack that involves a malicious switch and a malicious host. The damage of such a collaboration can be severe, for example, the attackers could serve benign hosts with malicious web pages. In order to exemplify the attack we use HTTP rather than HTTPS.

For this attack, we set up Mininet and ONOS with `ifwd` activated as shown in Figure 4.12. $s1$ and $k2$ are both malicious while the others are not. $k3$ is a benign web server. $s1$ teleports specific HTTP traffic towards $k2$. $k2$ modifies the HTTP traffic and teleports it back to $s1$ who then forwards it to $k1$. In order to emulate the malicious switch, we introduced a flow rule (shown in Listing 4.1) that rewrites the destination MAC address for TCP traffic

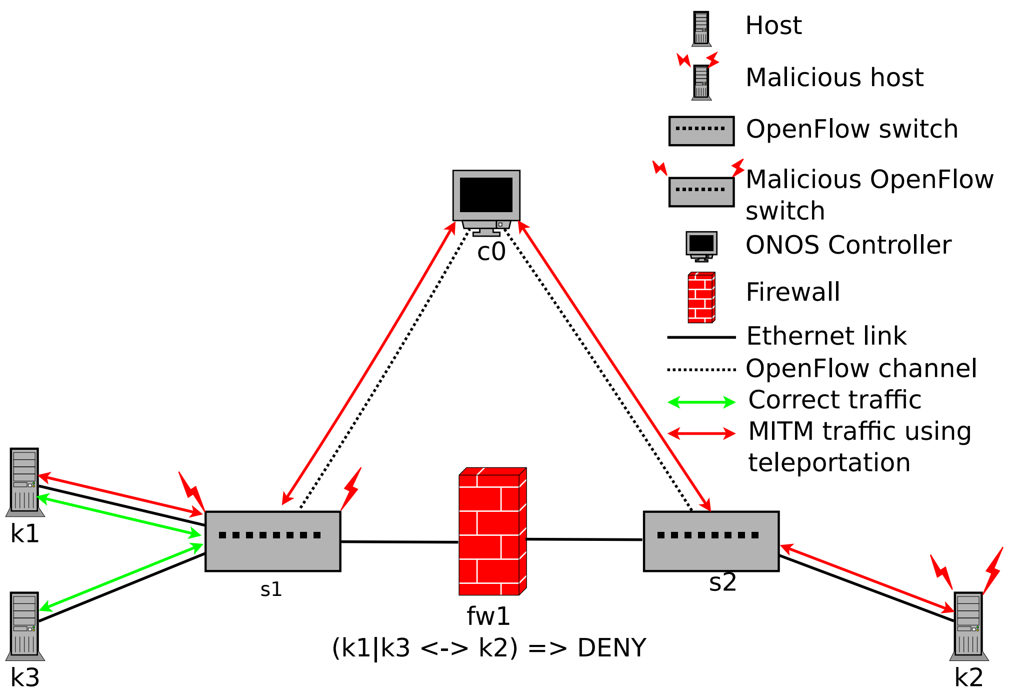

Fig. 4.12: An SDN topology with OpenFlow switches $s1$ and $s2$ with $c0$ the OpenFlow controller (ONOS). $k1$ and $k3$ are connected to $s1$ while $k2$ is connected to $s2$. $fw1$ denies $k2$ to communicate with $k1$ and $k3$ and vice-versa via the data plane. $s1$ and $k2$ being malicious, exploit the out-of-band forwarding teleportation to eavesdrop and modify communication data between $k1$ and $k3$ bypassing $fw1$.

with PSH and ACK flags sent from $k3$ to $k1$, to $k2$. This modified packet is then passed through the flow table lookup again by using the `resubmit` action in Open vSwitch. $k2$ runs `ettercap` to modify the TCP/HTTP payload and forwards the packet to the correct destination. Specifically, we created an `ettercap` filter that looks inside HTTP responses from $k3$ for the word “good", replaces it with “evil", and sends it to $k1$. The firewall $fw1$ is meant to block traffic between hosts on the right and the left.

When $k1$ requests the `index.html` page from $k3$, based on the flow rule installed on $s1$, only HTTP responses from $k3$ are teleported to $s2$ and forwarded to $k2$, through the out-of-band forwarding teleportation. Subsequently, $k2$ modifies only the `index.html` web page and has $s2$ teleport it back to $s1$ via out-of-band forwarding teleportation. Indeed, the side effect is `Flow-mod` messages to $s1$ and $s2$.

By viewing the `index.html` file received at $k1$ we verified that the mitm attack was successful. The benign and malicious web pages are shown in Listing 4.2 and Listing 4.3 respectively. By inspecting the flow counters on the switches we verified that necessary packets did not pass through the data plane.

Note that we did not introduce code into the Open vSwitches to handle the mitm, therefore once the flows are installed on the switches, the firewall will block all traffic between $s1$ and $s2$ and vice-versa.

```
priority=50001,tcp,in_port=2,
dl_src=00:00:00:00:00:03,
dl_dst=00:00:00:00:00:01,
tp_src=80,tcp_flags=+psh+ack
actions=mod_dl_dst:00:00:00:00:00:02,
resubmit:0
```

Listing 4.1: An Open vSwitch flow rule that was introduced into the malicious switch ($s1$) to teleport HTTP traffic with the PSH and ACK flags to the benign switch $s2$. The matching packets have the destination MAC address modified and resubmitted to the flow-table lookup which results in Out-of-Band Forwarding teleportation.

```
root@Mininet-vm:~# curl http://10.0.0.2
<html>
<head>
<title>Welcome page</title>
<body>
good
</body>
</html>
```

Listing 4.2: HTML code from the benign web server. Note the word “good” is present in the body of the HTML code.

4.4.6 Summary

We can summarize the demonstrated attacks using the four teleportation techniques as follows. Malicious switches can indeed exploit the fundamental design of data center networks for covert communication that can result in at least five different attacks. Automation and programmability (used in intent frameworks) combined with the lack of important information (network security policies) at the controller also enables malicious switches to bypass firewalls, coordinate attacks, evade security policies, exfiltrate data and conduct mitm attacks. Finally, the capability for switches to connect to the controller and spoof their DPIDs allows them to discover other malicious switches and also be used for covert communication (as we will see in the next chapter).

4.5 Out-of-Band Forwarding Performance

Having identified and demonstrated the various attacks in this section we describe our evaluation of the out-of-band forwarding channel. In the following chapter we describe our evaluation of the switch identification teleportation channel. In particular we measure the throughput, jitter and packet loss of the channel, and the resource footprint of this channel in terms of CPU usage and memory consumption at the controller.

4.5.1 Setup

In order to measure the throughput, jitter and packet loss of the out-of-band forwarding channel, we set up three dedicated systems: one system (64 bit Intel Core i7-3517U CPU @ 1.90 GHz with 4GB of RAM) running ONOS-1.5, another system (Intel Core 2 CPU @ 2.13

```
root@Mininet-vm:~# curl http://10.0.0.2
<html>
<head>
<title>Welcome page</title>
<body>
 evil
</body>
</html>
```

Listing 4.3: HTML code modified by the malicious switch $s1$ and host $k2$. Note the word "evil" is present in the body of the HTML code.

GHz with 4GB of RAM) running Mininet for the switches, and a third system (Intel Core i5-5200U CPU @ 2.20GHz with 16GB of RAM) running OFCProbe [92] for load generation. Only the three systems are networked together via a Netgear 100Mbps switch. On the Mininet system, we use a simple line topology consisting of two hosts and two switches, where, host1 is connected to switch1 which is connected to switch2; switch2 in turn is connected to host2. The switches accordingly connect to ONOS as their controller.

4.5.2 Methodology

To emulate the malicious switch, we simply install a flow rule on switch1 with the highest priority so that the out-of-band forwarding applies by default, i.e., Packet-Ins are sent to ONOS and forwarded accordingly.

We measure the throughput, jitter and packet loss using iPerf3 running on the hosts in Mininet using UDP packets. We consider UDP packets with a payload of 512 bytes to be teleported. Note that for the 512 bytes to be teleported, the overhead in bytes for encapsulation (in the following order: Ethernet, IP, TCP, OpenFlow, Ethernet, IP, UDP) is 110 bytes (for a Packet-in) and 108 bytes (for a Packet-out). Therefore, a 10 Mbps teleportation channel corresponds to approximately 2009 packets (Packet-ins) per second. For the CPU and memory usage on the controller, we use `taskset` to pin ONOS to a single CPU and use `top` to measure the CPU and memory usage. For the load generation: OFCProbe, emulates 20 switches that trigger Packet-Ins to the controller following a Poisson distribution (λ=1). The throughput, jitter, packet-loss, CPU and memory usage is sampled every second for 600 seconds.

4.5.3 Evaluation

We first study the throughput of the UDP-based teleportation channel, then consider the packet loss and jitter characteristics, and finally examine the resource footprint in terms of CPU and memory in turn.

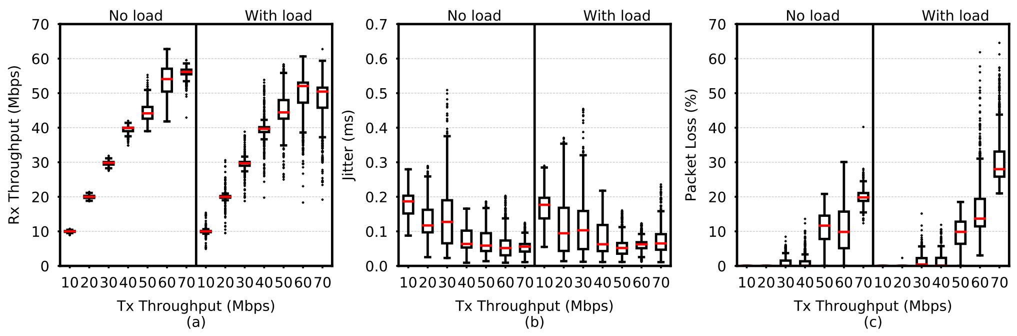

Fig. 4.13: Out-of-Band Forwarding performance characteristics without and with load on the controller. (a) Received throughput, (b) Jitter and (c) Packet loss.

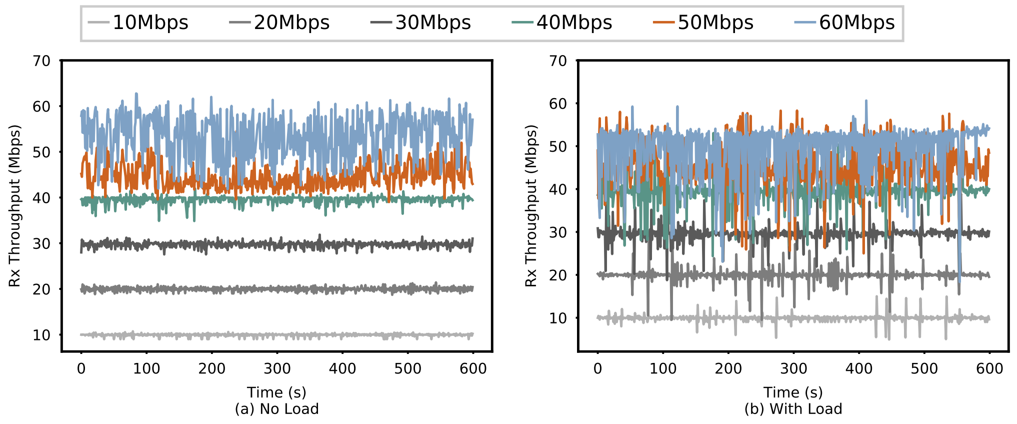

Fig. 4.14: Received throughput using Out-of-Band Forwarding (a) without load (b) with load on the controller.

4.5.3.1 Throughput

In Fig. 4.13 (a) we visualize the throughput of the teleportation channel as box plots without and with load on the controller. In Fig. 4.14, we visualize the throughput of the teleportation channel without and with load resp. as scatter plots.

We first observe that the teleportation channel can indeed sustain very high transmission rates (Tx), of up to 40Mbps in both scenarios. In the scenario without load, we see that the channel becomes saturated around rates slightly higher than 60Mbps, after which the throughput suffers. In the scenario with load the variance of the throughput is naturally higher, but nevertheless it can only sustain rates upto 50 Mbps.

In conclusion, our results show that the performance of teleportation can go far beyond a small number of packets per second, which underlines the relevance (and potential threat) of such channels.

4.5.3.2 Jitter and Packet Loss

We plot the jitter without load resp. with load in Fig. 4.13 (b) [3]. In terms of this metric, we can see that the teleportation channel offers a good quality also for high rates. The load on the controller again introduces some variance to the jitter, however, it does not influence the median value by much.

Figure 4.13 (c) shows the packet loss for the scenario without load and with load. The experiments confirm the quality of the considered teleportation channel: Up to 40 Mbps, the packet loss is small despite some variance, and naturally increases beyond 10% above 50 Mbps. Indeed, we can see a direct correlation between the packet loss and the drop in throughput.

4.5.3.3 Resource Footprint

To better understand the resource requirements of the teleportation channel, as well as the reasons behind the throughput drop at high rates, we measured the CPU load and memory footprint on the controller.

Fig. 4.15 (a) visualizes the CPU usage as a box plot, while Fig. 4.16 visualizes the CPU loads over time. We observe that for a 10 Mbps channel, the CPU utilization has a median value of 55, which is fairly high, but not alarming. We also observe that at rates around 20 Mbps, the additional CPU load introduced for an extra 10 Mbps (20 Mbps vs 30 Mbps channels) is small. The influence of the load on the controller is discernible by the variance introduced and a slight increase in the utilization. However, again at around 50 Mbps, the effects become larger: We can clearly see the relationship between the throughput and CPU load, and when the CPU consumption begins to climb, the throughput begins to drop. Indeed, for transmission rates beyond 50 Mbps, the CPU utilization is so high that it can easily be detected. This is also the time around which the jitter tends to increase by a small amount.

Hence, we draw the following conclusions. As the transmission rates increase, the receive and transmit queues at the controller fill up quickly. Since the controller needs to parse, decapsulate, make a forwarding decision, encapsulate the packet (as a packet-out) and then transmit it, packet processing takes longer as the controller CPU is interrupted by the increasing rate at which packets arrive either from OFCProbe and/or the switch.

With respect to the memory consumption, Fig. 4.15 (b) shows that between 10 and 50 Mbps the memory consumption is within a close range (13-15 MB) regardless of whether the load is induced or not. For 60Mbps and above, the memory consumption is higher. Nonetheless, the impact of teleportation on the memory is negligible.

[3] Due to noise in our measurement setup, we obtained some outliers in the jitter experiments. Therefore, we followed the median absolute deviation [82] method, with a tolerance of 3.5 to remove such outliers.

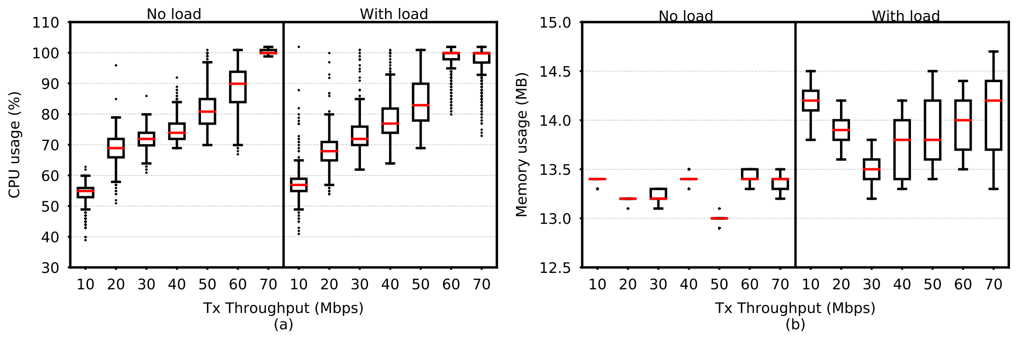

Fig. 4.15: (a) CPU load using Out-of-Band Forwarding without and with load on the controller. (b) Memory usage using Out-of-Band Forwarding without and with load on the controller.

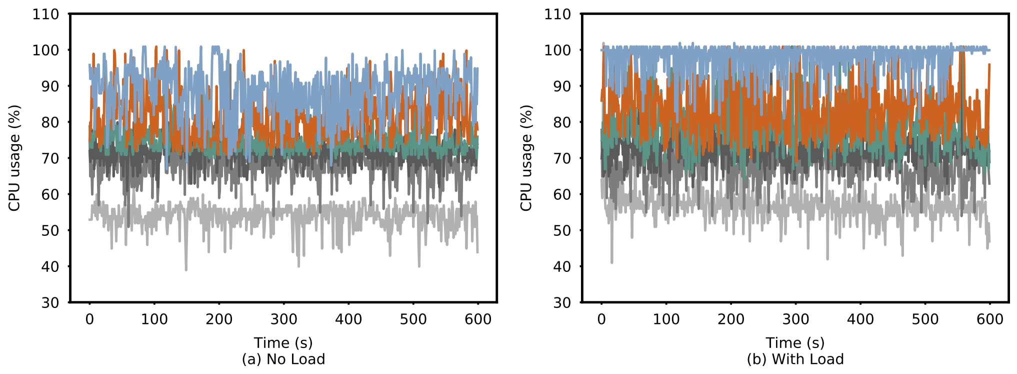

Fig. 4.16: Scatter plot of the CPU usage on the controller for Out-of-Band Forwarding. (a) Without load on the controller. (b) With load on the controller. Legend as in Fig. 4.13.

4.5.4 Summary

Our first experiments show that teleportation channels in the order of 10 Mbps are feasible, providing low packet loss and low jitter. Moreover, these channels introduce a moderate resource overhead in terms of CPU and a low overhead in terms of memory. Hence, such traffic may go unnoticed given the normal traffic patterns (even if the regular traffic rate is orders of magnitude smaller). However, we also observe that beyond a certain teleportation rate, the CPU load will increase and become the bottleneck for teleportation, limiting the throughput, introducing high packet loss rates, and jitter. Therefore, we expect a sophisticated attacker to target teleportation rates resulting in resource footprints which are obfuscated by the regular load.

4.6 Countermeasures

Having showcased a variety of attacks using teleportation, we now start exploring possible countermeasures. Although we have demonstrated all the attacks using ONOS we believe that these issues are likely to become more general in nature. They are becoming important with the shift towards automated and intent aware controller frameworks allowing for simpler and agnostic controller applications. Based on our experiments we have also seen that the resources required and utilized for teleportation, even at high rates are moderate. Therefore, it may be difficult to distinguish the attack traffic from the benign traffic. Accordingly we believe that, with the separation of the control and data plane, it is now important to monitor and police the communication channel between the separated planes due to the increased attack surface [228].

4.6.1 Packet-in-Packet-out Watcher

In order to prevent the out-of-band forwarding teleportation, we strongly advise the use of a `Packet-in` and `Packet-out` watcher. It can either exist as a controller application or as an application that resides between the controller and switches akin to hypervisors. It would involve tracking and enforcing security policies for `Packet-ins` and their corresponding `Packet-outs`. Existing security enforcement kernels, hypervisors and security applications must account for `Packet-ins` and `Packet-outs` in addition to `Flow-mods` to detect and prevent out-of-band forwarding teleportation.

Note that the out-of-band forwarding teleportation could also be used by malicious controller applications. In a non-adversarial scenario, the order in which a packet's fate is decided upon by various applications can inadvertently teleport the packet. Therefore, verifying that the `Packet-out` does not reach an undesired switch/host can prevent out-of-band forwarding teleportation.

4.6.2 Audit-Trails and Accountability

We propose controllers to introduce secure audit-trail capabilities, and accounting, that enable network administrators to thoroughly investigate events in their networks. For example, controllers must log and alert sensitive events such as a moving MAC addresses, or, receiving a `Packet-in` when a flow has not yet timed out. Such capabilities can aid detection and prevention mechanisms. It is also useful for investigating security incidents. We recommend administrators to frequently view controller logs, investigate failed events and suspicious identities in the network.

4.6.3 Enhanced IDS with Waypoint Enforcement

Network intrusion detection systems are an important means to detect and limit cyber attacks today, and accordingly intrusion detection systems constitute an integral part of most networks. We strongly suggest the use of an IDS application on top of or before the controller, that can inspect `Packet-ins` and `Packet-outs` and alert on suspicious traffic. Indeed, some controllers today already offer basic functionality for waypoint enforcement. In particular, we suggest waypoint enforcement and coordinating intrusion detection systems from the control plane with the data plane. This is non-trivial, but vital for network security.

4.7 Related Work

While researchers have already pointed out several interesting novel challenges in providing a correct operation of networks with separate data and control planes [159, 178, 249], it is generally believed that SDN has the potential to render computer networking more verifiable [105, 107] and even secure [128, 138, 174, 203, 204].

Only recently researchers have started discovering security threats in SDN. Klöti et al. [108] report on a STRIDE threat analysis of OpenFlow, and demonstrate data plane resource consumption attacks. Kreutz et al. [111] survey several threat vectors that may enable the exploitation of SDN vulnerabilities. Benton et al. [18] analyze vulnerabilities in OpenFlow. In particular they point out the lack of TLS adoption/implementation in OpenFlow switches and controllers. In addition, they correctly identify the possibility of dos attacks on the centralized control plane. Another key challenge arising from the separation of the control and data planes, is the potential loss of network visibility. It has been shown that the network view of the controller may even be poisoned [46, 77]. Thimmaraju et al. [228], point out that threat models for the virtualized data plane need to account for a malicious/compromised data plane in SDNs, and cloud operating systems such as OpenStack.

While much research went into designing more robust and secure control planes [27, 28], less published work exists on the issue of malicious switches. A notable exception is the work by Antikainen et al. [8], who consider the possibility of a malicious relay node for a man-in-the-middle attack. Interestingly, in this chapter, we have shown that the relay node can be the benign controller itself.

To the best of our knowledge, our work is the first to point out and characterize the fundamental problem of SDN teleportation. More generally, while most prior studies about malicious switches focus on (indirect) *attacks targeting the controller*, we in this chapter demonstrate new kinds of attacks which merely exploit the controller for *directly* attacking (e.g., the confidentiality or availability) of network services.

However, there are a number of interesting approaches proposed in the literature which have implications for our scenarios as well. For example, the pre- and post-conditions of TopoGuard [77] can defend against our path update attack. However, if the switches are malicious, these conditions can be spoofed by the malicious switches. Also, TopoGuard cannot detect teleportation using path reset, switch identification and out-of-band forwarding teleportation.

Sphinx [46] can alert on the path update teleportation. However, it cannot detect the path reset as the flow graph remains the same. Additionally, Sphinx assumes that switches cannot use the same DPIDs, therefore, we believe that our switch identification teleportation will not be detected by Sphinx. Also, our out-of-band forwarding relies on `Packet-in` and `Packet-out` messages, while `Packet-outs` are not considered by Sphinx[4]. Therefore the suggested out-of-band forwarding teleportation can evade Sphinx, until topology altering flows are installed.

[4] Unfortunately, the source code of Sphinx is not available.

Tab. 4.2: Summary of teleportation attacks and involved entities.

Attack	Teleportation technique	Exploited by
Bypass Firewall	Out-of-band forwarding	Switch and Host
Bypass NIDS	Out-of-band forwarding	Switch and Host
Exfiltration	Out-of-band forwarding	Switch and Host
Evading policy conflicts	Out-of-band forwarding	Switch and Host
Man-in-the-middle	Out-of-band forwarding	Switch and Host
	Path update	Switch
Rendezvous	Path reset	Switch
	Switch identification	Switch

Porras et al. [175] propose a security mediator that comprises of Rule Conflict Analysis, Role-based Source Authentication, State Table Manager and a Permission Mediator. We admit that the path update can be detected using this approach, however, our path reset does not introduce any conflicting rules. The `Features-reply` messages are not a part of their solution, therefore, we believe that switch identification teleportation can succeed. With respect to out-of-band forwarding teleportation, unless the mediator investigates the destination switch or MAC address in the `Packet-out`, the teleportation can bypass the security mediator given sufficient permissions.

SDN Hypervisors such as CoVisor [97], Flowvisor [201], FortNOX [174] depend on policies maintained in the hypervisor. Therefore, we believe that all our teleportation mechanisms hold unless a specific policy blocks it. Dover Networks [48] discovered the behavior of Floodlight with switches using the same DPID, which we exploit for teleportation. While Security-Mode ONOS [195] can enhance the security in many scenarios, by introducing roles and permissions, at least today, it does not help against teleportation: Once `ifwd` has the permission to write intents and emit packets, our teleportation succeeds. These permissions are bare necessities for `ifwd` to function.

4.8 Conclusions

As OpenFlow networks transition from research to production, new levels of reliability and performance are necessary [117]. This chapter has identified and demonstrated a novel security threat introduced by software-defined networks separating the control plane from the data plane. In the presence of an unreliable south-bound interface (containing malicious switches): We have shown that state-of-the-art controller(s) are vulnerable to teleportation. Teleportation has numerous applications (cf. the summary in Table 4.2): It can be exploited to bypass security-critical network elements (e.g., to exfiltrate confidential information), as a discovery protocol for malicious switches, to evade policy conflicts as well as for man-in-the-middle attacks. Based on our preliminary evaluation, we can say that a teleportation channel of over 10 Mbps can easily masquerade inside a loaded control channel.

Our work can also be seen as a first security analysis of the increasingly popular intent-based network mechanisms [86]: While intent-based mechanisms are attractive for allowing (cloud) network operators resp. SDN applications to focus on *what to connect* rather than *how*, we have shown that controller managed intents need to be used with care. Indeed, our experiments with controllers that are only starting to introduce an intent based mechanism are not yet vulnerable to all the specific attacks presented in this chapter. Moreover, while intent mechanism implementations can vary across controllers, we believe that the underlying issues are fundamental, such as switch identification teleportation, the subject of the next chapter.

5 Peekaboo! I DPID You: A Novel OpenFlow Covert Channel

We saw how *teleportation* is inherent to networks designed with a centralized control plane in the previous chapter. In this chapter we delve deeper into the switch identification teleportation technique initially described as a rendezvous protocol. We also explain why switch identification teleportation is fundamental to OpenFlow networks.

We then describe how it can also be used for covert communication: malicious switches can transfer a 2048 byte *RSA private key file* in $\sim$13 minutes. In particular, we design, develop and evaluate a time-based covert channel using the switch identification teleportation in this chapter.

Novelty and Related Work. To the best of our knowledge, this is the first publicly known covert timing channel in an SDN, and OpenFlow-based network in particular. We are only aware of one other paper dealing with covert channels in SDN, which is however very different in nature: Hu et al. [79] proposed to use SDN to improve the detection of storage covert channels that use the TCP flags for covert communication. More generally, the study of covert channels dates back to the 80's when Simmons [205] introduced the "Prisoners Problem" and the *subliminal channel*. Network based covert channels in local area networks were introduced by Girling [68], wherein a covert channel based on the inter frame delay was proposed. Handel et al. [73] conducted an extensive study on viable covert channels within the OSI networking model. A covert channel based on sending an IP packet or not in a time interval was demonstrated by Cabuk et al. [26]. More recently, Tahir et al. [217], designed and developed Sneak-Peek, a high speed covert channel in data center networks. Their covert channel also utilizes a delay mechanism wherein the sender's *flow* introduces a delay into the receivers *flow* over the same network link thereby covertly communicating information based on the delay measured by the receiver. In 2019, Ovadia et al. [156], demonstrated how users connected to the same router can covertly communicate with each other even when they are placed in logically isolated networks, and Xing et al. [245] proposed to use P4 to detect network covert channels (TCP based) to avoid forwarding performance loss.

Chapter Organization. In Section 5.1 we briefly recap switch identification teleportation and then elaborate on our covert channel. We describe the key challenges we encountered in Section 5.2 followed by our evaluation in Section 5.3. After a discussion is Section 5.4, we conclude in Section 5.5.

Threat Model. Similar to Chapter 3, in this chapter we consider a threat model where OpenFlow switches, hosts, or both, may not behave correctly but are malicious.

5.1 A Covert Channel using Teleportation

Covert channels are communication channels that were not designed with the intention for communication [21]. They can be used to bypass security policies, thereby leading to unauthorized information disclosure [118]. A covert timing channel is one wherein a sender and receiver "use an ordering or temporal relationship among accesses to a shared resource" [21] to covertly communicate with each other. In the following we recapitulate switch identification teleportation and then describe how it can be used as a covert timing channel in a software-defined network using the OpenFlow protocol.

Switch Identification Teleportation. In an OpenFlow network, the switch typically initiates a TCP connection with the OpenFlow controller as shown in Fig. 5.1. If TLS/SSL is configured, the connection is further authenticated and subsequent messages exchanged are encrypted as well. Once the transport connection is established, the switch sends the controller an OpenFlow `Hello` message. The controller responds with a `Hello` message. These messages are used to negotiate the OpenFlow version to be used. Next, the controller sends the switch a `Features-Request` message. The switch replies with a `Features-Reply` message. The `Features-Reply` message includes a `Datapath ID` (DPID) field that uniquely identifies the switch to the controller. After processing the `Features-Reply` message, the OpenFlow connection is considered established, and ready for operation [151].

A fundamental requirement of an SDN is for the controller to uniquely identify the switches in the network which is achieved by the switch providing "identity" information, e.g., DPID in the `Features-Reply` message, to the controller. *Switch identification teleportation* is the outcome of two switches connecting to the same logical controller using the same DPID (recall Section 4.3.2). We have identified 4 possible outcomes when this occurs in OpenFlow: i) The controller denies a connection with the second switch; ii) The controller accepts the connection with the second switch, and terminates the first switch's connection; iii) The controller accepts connections for both switches; iv) The controller accepts connections for both switches, however, each switch receives a different `Role-request` message. Only in outcomes i, ii and iv can the malicious switches infer if the DPID it used is already in use by another switch. The message sequence pattern for the OpenFlow handshake and outcome i is shown in Fig. 5.1.

Lack of Authentication and Authorization. We analyzed the security of the OpenFlow handshake and identified two important security features missing: authentication and authorization. First, the OpenFlow specification does not require the controller to authenticate the DPID announced by the switch. This lets the switch spoof its DPID during the handshake. Second, the specification does not require the controller to connect to only an authorized list of DPIDs. This lets malicious switches use arbitrary DPIDs in the `Features-relpy` message. The lack of these two security mechanisms combined with the fact that switches can initiate the (transport, e.g., TCP) connection to the controller contribute to the possibility of switch identification teleportation *by design* in OpenFlow. These findings have been reported in CVE-2018-1000155. In Section 5.4 we discuss a practical solution to counter this problem in OpenFlow.

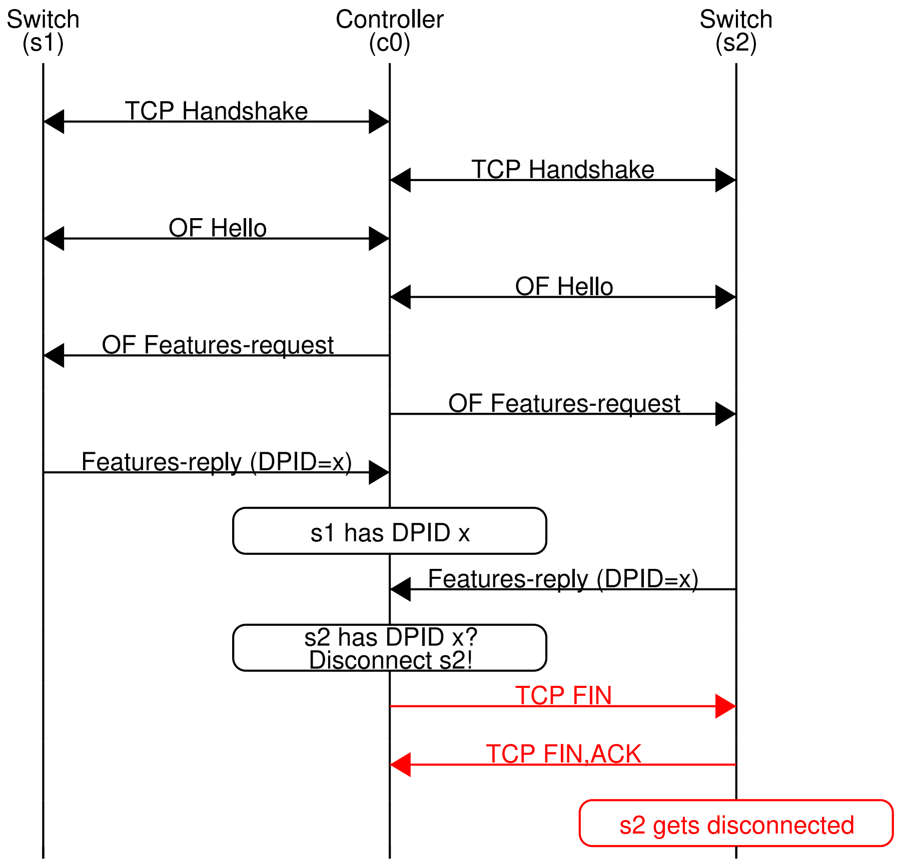

Fig. 5.1: Message sequence pattern for the OpenFlow handshake and *switch identification* teleportation when the controller denies the second switch a connection. In this way, s2 can infer a bit value after it gets disconnected.

5.1.1 Single Bit Transfer

From the message sequence pattern in Fig. 5.1, switch $s2$ can infer a binary value of 1 if it gets disconnected, and a binary value of 0 if it is able to connect, thereby it receives one bit of data. We can precisely describe the states and transitions to transfer one bit value as finite state machines for the sender and receiver resp. Additionally, we can precisely describe a time-based model to transfer one bit value that can be leveraged to design a channel to transfer multiple bits. We now turn our attention to describe the state transition model followed by the time model to transfer one bit. Then, we describe our algorithms to transfer multiple bits.

5.1.1.1 State Transition Model

The state transition model for switch identification involves a *sender* and *receiver*. As the names imply, the sender sends a binary bit value by either connecting to the controller or not. Similarly, the receiver receives a binary bit value by detecting whether its OpenFlow connection to the controller is allowed or denied.

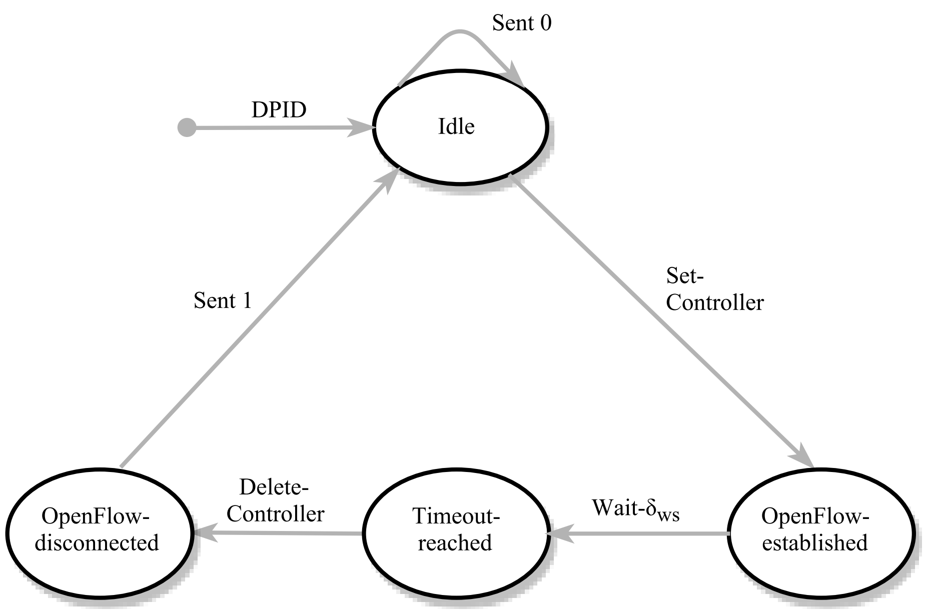

Fig. 5.2: State machine for the sender to send one binary value.

In our model, we make the following assumptions. We assume that the sender and receiver use an a priori agreed upon DPID (one that is not used in the network), a time to connect to the same OpenFlow controller and a time interval Δ. Δ, is the total time the sender and receiver use to send and receive resp. a bit value. The sender and receiver have synchronized their clocks. We discuss synchronization further in Sec. 5.2.1. The receiver in particular, is always able to connect to the controller a short time δ_{offset} after the sender. The controller, behaves according to outcome i (see Switch Identification Teleportation). The receiver infers a binary bit value of 1, if its OpenFlow connection is denied, i.e., the sender connected to the controller before the receiver. The receiver infers a binary bit value of 0, if its OpenFlow connection is accepted, i.e., the sender did not connect to the controller.

The sending and receiving of bit information can be described in more detail by defining a set of states and transitions for the sender and receiver resp., as shown in Fig. 5.2 and 5.3.

Sender. The sender *starts* data transmission with an agreed upon DPID, by entering into the *Idle* state. To send a 0, it simply remains in the *Idle* state. To send a 1, it transitions to the *OpenFlow-established* state via the *Set-Controller* transition. *Set-Controller* involves initializing internal objects, e.g., `rconn` and `vconn` data structures in Open vSwitch, in order to initiate a transport (e.g., TCP) connection to the controller at a specific IP and port address. It also involves establishing the TCP and OpenFlow connection with the controller. Once the OpenFlow connection is established, the sender waits for a timeout δ_{ws}, to move into the *Timeout-reached* state. From there, the sender enters into the *OpenFlow-disconnected* state by tearing down the TCP and OpenFlow connection, and deleting its controller information. From thereon, the sender completes a bit transfer by entering back into the *Idle* state. The sender's state diagram is depicted in Fig. 5.2.

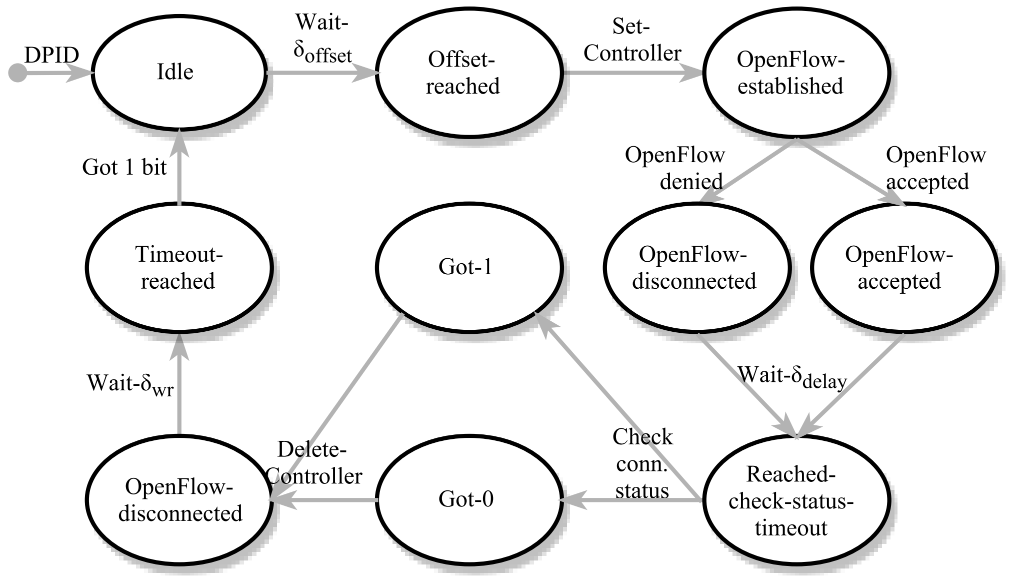

Fig. 5.3: State machine for the receiver to receive one binary value.

Receiver. The receiver also starts with the same DPID to enter into the *Idle* state. Unlike the sender, the receiver must always attempt to connect to the controller to receive a 0 or a 1. It waits for δ_{offset} time to enter the *Offset-reached* state before it sets the controller to enter into the *OpenFlow-established* state, similar to the sender. If the receiver's OpenFlow connection is denied, it will enter into the *OpenFlow-disconnected* state resulting in its OpenFlow and transport connection being terminated. If the receiver's OpenFlow connection is accepted, it will enter into the *OpenFlow-accepted* state resulting in its OpenFlow connection being sustained. Regardless of the outcome, the receiver waits δ_{delay} time, thereby transitioning to the *Reached-check-status-timeout* state. From there, the receiver checks the OpenFlow connection status. It enters the *Got-1* state if it was disconnected, i.e., it got a 1. It enters the *Got-0* state if it was accepted, i.e., it got a 0. From there on the receiver deletes its controller information, resulting in the OpenFlow and transport connection being torn down if it is still present. Depending on the value of Δ, there may still be time left, hence the receiver waits δ_{wr}, till the end of the time interval, to enter the *Timeout-reached* state. It then completes the reception by moving back into the *Idle* state. The state diagram for the receiver is shown in Fig. 5.3.

5.1.1.2 Transition Delays

To leverage switch identification as a covert timing channel we must first establish the time it takes for the sender to send a 1—as sending a 0 requires the sender to remain in the *Idle* state—and the receiver to receive a bit value. We define a *time interval* Δ, as the time the sender and receiver use to send and receive resp. a binary bit value.

Δ comprises of the several state transitions described for the sender and receiver (Sec. 5.1.1.1). We can construct a time-based model by considering the transitions as delays or timeouts for the sender and receiver that can be used to analyze the performance of our covert channel.

In the following we define the various delays and timeouts for the sender and receiver state transitions.

1. δ_s: The time the sender takes to send a binary bit value.
2. δ_r: The time the receiver takes to receive a binary bit value.
3. δ_{sc}: The time to transition from the *Idle* state to the *OpenFlow-established* state.
4. δ_{dc}: The time to move from the *OpenFlow-established* state to *OpenFlow-disconnected* state.
5. δ_{offset}: A timeout value the receiver waits before it sets the controller.
6. $\delta_{of\text{-}deny}$: The time to move from *OpenFlow-established* to *OpenFlow-disconnected* when the connection is denied.
7. δ_{delay}: A timeout value the receiver waits before it checks the OpenFlow connection status.
8. $\delta_{chk\text{-}conn}$: The time the receiver takes to determine a 0 or 1 by checking the OpenFlow connection status.
9. $\delta_{ws} = \Delta - \delta_s$: A timeout value the sender waits before moving from the *OpenFlow-established* state to *OpenFlow-disconnected*.
10. $\delta_{wr} = \Delta - \delta_r$: A timeout value the receiver waits before moving from the *OpenFlow-disconnected* state to the *Idle* state.

Using the above definitions, we can now compute the time to send and receive a 0 or 1. The total time to send a 0 or 1 is shown in Eq. 5.1. As we can see, it takes more time to send a 1 ($\delta_{sc} + \delta_{dc}$) compared to a 0. In Eq. 5.2, we can see the time it takes to receive a 0 or a 1. In particular, the different delay is $\delta_{of\text{-}deny}$ for the 1. For the sender and receiver to operate correctly, we require the inequality shown in Eq. 5.3 to hold, i.e., the time interval Δ must not be less than the total time to send or receive a binary bit value.

Additionally, for the receiver to correctly detect a 0 and 1, we require the inequalities as shown in Eq. 5.4 and 5.5 to hold. The former equation states that δ_{offset} must be greater than the time it takes for the sender to enter the *OF-established* state. This is to ensure that the receiver does not connect before the sender when the sender wants to send a 1. The latter equation states that the minimum amount of time it can wait before checking the OpenFlow connection status is 0, and the maximum time it can wait depends on the time interval, the time elapsed so far, and the time for the remaining transitions to complete. The δ_{delay} gives the receiver the flexibility of waiting for some amount of time before checking the status of the OpenFlow connection. For example, checking the connection status at $\Delta/2$, i.e., at the middle of the time interval, may be better than checking it at $\Delta/4$. Hence, the receiver can set δ_{delay} such that, the OpenFlow connection status is checked at a point where the connection is determined to be most stable.

$$\delta_s = \begin{cases} 0, & \text{to send } 0 \\ \delta_{sc} + \delta_{dc}, & \text{to send } 1 \end{cases} \tag{5.1}$$

$$\delta_r = \begin{cases} \delta_{offset} + \delta_{sc} + \delta_{delay} + \delta_{chk\text{-}conn} + \delta_{dc}, & \text{to get } 0 \\ \delta_{offset} + \delta_{sc} + \delta_{of\text{-}deny} + \delta_{delay} + \delta_{chk\text{-}conn} + \delta_{dc}, & \text{to get } 1 \end{cases} \tag{5.2}$$

$$\delta_s \leq \delta_r \leq \Delta \tag{5.3}$$

$$\delta_{offset} \geq \delta_{sc} \tag{5.4}$$

$$0 \leq \delta_{delay} \leq \Delta - (\delta_{offset} + \delta_{sc} + \delta_{of\text{-}deny} + \delta_{chk\text{-}conn} + \delta_{dc}) \tag{5.5}$$

5.1.2 From One Bit to Multiple Bits

Until now, we have described how the sender can transmit only a single bit value to the receiver. To receive the single bit value, the sender and receiver need to be synchronized, i.e., the sender and receiver must know the exact time at which the time interval Δ begins and ends. To this end, we assume the sender and receiver synchronize their clocks using the same network time protocol (NTP) time server. Furthermore, we assume the sender and receiver a priori agree upon specific times at which they will initiate their covert communication.

In order to be useful, a covert channel should provide a sender with the ability to transmit several kilobytes of data, e.g., an RSA private key file. Accordingly, in the following, we extend our discussion from a single bit transmission to multiple bits. First, the sender and receiver must agree upon an encoding/decoding scheme, e.g., ASCII. Second, they must also agree upon a method to signal the *start* and *end* of a message. To do so, we use a frame-based transmission method. In particular, the sender encodes a message M into into *frames* F, of length Fl, and transmits the frames. The receiver, decodes each frame received to obtain the sent message.

For *simplicity*, we consider a frame with at least one *SoF* (Start of Frame) bit, and at least seven *data* bits (e.g., ASCII characters can be represented in 7 bits). The SoF bit is used by the sender to signal the receiver that a frame transmission begins which is followed by data bits. We assume that the *SoF* bit is a binary 1, and if the receiver gets this value at the agreed upon time and time interval, it will begin receiving data bits. The data bits can be 0 or 1 depending on how the message is encoded. To indicate the end of a message, the sender sends a frame with all the data bits as 0. When the receiver receives such a frame, it will

```
Algorithm 3: To send binary data as frames.
Input: Message $M$, Frame-length $Fl$, Frames $F$, Time-interval $\Delta$, Start-time $t$,
       Encode-scheme $S$
initialize(sender)
encode($M$)
for $frame \in F$ do
    $set\text{-}controller$                          ▷ Send SoF bit
    Wait $\delta_{ws}$
    for $bit \in frame$ do
        if ($bit$==0) then
            $delete\text{-}controller$               ▷ Send 0
        else
            $set\text{-}controller$                  ▷ Send 1
        end
        Wait $\delta_{ws}$
    end
    $delete\text{-}controller$
end
```

terminate execution. The above steps are specified as algorithms for the sender and receiver in Alg. 3 and 4 respectively.

Sender and Receiver. The sender's algorithm (Alg. 3) requires several inputs. M is the message to be transmitted, Fl is the frame length, e.g., 8, F is the list of frames that are to be sent, Δ is the time interval, t is the transmission start-time and S is the encoding scheme. The input values for the receiver (Alg. 4) are the same frame length, time interval, start-time and encoding as the sender.

For every frame to be sent, the sender first sends an *SoF* bit for that frame by connecting to the controller. Similarly the receiver waits for δ_{offset} time before attempting to receive the *SoF* bit. If its connection is denied, it will begin receiving data bits. After sending the *SoF* bit, the sender sends data bits: if sending a 0, it disconnects from the controller, if sending a 1, it connects to the controller. It then waits till the end of the timing interval before sending the next data bit. The receiver detects the data bits in a frame by connecting to the controller, and waiting for δ_{delay} time before checking whether its OpenFlow connection was allowed or not. If the connection was accepted, it will append a 0 to the data bits received in the frame, otherwise it will append a 1. The receiver then deletes the controller, and then waits δ_{wr}, i.e., till the end of the time interval before connecting to the controller again.

Once the sender has sent the data bits of a frame, it will wait δ_{ws} time, i.e., for the next time interval to send the next frame. The receiver detects the end of a message when it has received a frame with all the data bits zeroed, thereby terminating the while loop at the receiver. The receiver can then decode the binary data to reveal the message sent.

Algorithm 4: To receive binary data as frames.

```
Input: Frame-length $Fl$, Time-interval $\Delta$, Start-time $t$, Decode-scheme $S$
initialize(receiver) while End of message not received do
    Wait $\delta_{offset}$
    $set$-$controller$                                 ▷ Receive SoF bit
    Wait $\delta_{delay}$
    Check OpenFlow connection state
    if OpenFlow denied then                                              ▷ Got SoF bit
        Wait $\delta_{wr}$
        for $bit \in Fl$ do
            $set$-$controller$                         ▷ Get data bit
            Wait $\delta_{delay}$
            Check OpenFlow connection state
            if OpenFlow accepted then
                $frame$ += "0"                         ▷ Got $0$
            else
                $frame$ += "1"                         ▷ Got $1$
            end
            $delete$-$controller$
            Wait $\delta_{wr}$
        end
        if $frame$ ==“0000000” then
            End of message received
            Break                                      ▷ Terminate reception
        else
            $M+=frame$                                 ▷ Append frame to message
        end
end
decode($M$)
```

5.2 Design and Performance Challenges

Our covert channel design requires us to overcome several non-trivial challenges. Hence, we discuss the most important challenges that affects our design in this section before transitioning to our implementation. We also cast light on factors that affect the performance of our design.

5.2.1 Synchronization

One of the main problems in designing a covert timing channel is synchronization. Lack of synchronization can lead to the receiver obtaining inaccurate information, thereby reducing the accuracy of the channel. The sender and receiver must share a reference clock to ensure that the algorithms start at the same time. To this end, we use NTP (as it is easily available for today's popular operating systems) and the same NTP server to synchronize the clocks of the sender and receiver to achieve at least millisecond accuracy [135]. Since the sender and receiver clocks can slowly drift apart their clocks must be periodically synchronized with the same NTP server.

When the clocks are synchronized, the *SoF* bit(s) in each frame sent synchronizes the receiver with the sender enabling the receiver to obtain the data bits. During the transmission of a frame, we introduce the δ_{ws} and δ_{wr} times for the sender and receiver resp. at the end of a time interval for synchronization across time intervals in a frame. Furthermore, between frames the sender and receiver can synchronize again by waiting, for example for the next second. This *inter frame delay* adds another layer of synchronization to enable the sender and receiver to send and receive resp. the *SoF* bit(s) accurately.

5.2.2 Determining the Time Interval Δ and Delays

The time interval in which the sender and receiver send and receive a bit leads to the achievable throughput of the channel. As the time interval reduces, the probability of an error occurring increases, e.g., the receiver may check the connection status before receiving the TCP FIN from the controller. Furthermore, system and network artefacts can non-deterministically influence the state transitions resulting in errors. Hence, the challenge here is to determine a time interval as small as possible within an acceptable level of accuracy ($\geq 95\%$). We empirically identify suitable time intervals in Sec. 5.3 based on our prototype implementation. However, in the real-world, the channel would have to start with a programmed value, e.g., 1s, and later be negotiated.

Recall Sec. 5.1.1.2, there are several delays involved in our timing channel. The delays for one network system, may not be applicable elsewhere. Delays such as δ_{sc}, δ_{dc}, $\delta_{of\text{-}deny}$, and $\delta_{chk\text{-}conn}$, depend on the system and network conditions. Moreover, they are not under the control of the sender/receiver. The timeouts δ_{offset} and δ_{delay} although bounded (see Eq. 5.4 and 5.5 resp.) can be tuned by the receiver. Hence, we evaluate 3 different δ_{delay} values in Sec. 5.3.

5.2.3 Frame-based Transmission

Our design uses a frame-based method to transfer data from the sender to the receiver. The smallest frame size we consider is 8 bits long: 1 *SoF* bit and 7 data bits. The size of this frame can change, e.g., we can send 14, 28 or more data bits as well. Sending more data bits in a frame reduces the overhead of sending the *SoF* bit. We can also increase the number of *SoF* bits to ensure the receiver can get the data bits. However, increasing the number of bits in a frame increases the probability of errors within a frame. We do not consider error correction in our design although it can be introduced, e.g., using Hamming codes. However, we do include a minimal set of error detections at the receiver which we describe next.

Receiver misses the start bit of the frame. Several reasons can affect the receiver from missing the *SoF* bit of a frame. In such cases the receiver simply remains idle for the remainder of the time that is necessary to transmit an entire frame.

End of Transmission. For simplicity, the sender indicates the end of transmission via a special *EoM* (End of Message) frame. This design choice comes with a couple of challenges for the receiver to correctly terminate. First, if the receiver misses the *SoF* bit of the *EoM* frame, then it will continue to expect to receive frames. To address this problem, we define a threshold number of consecutive frames, e.g., 5, the receiver does not receive beyond which the receiver terminates reception. Second, the receiver can incorrectly detect a 1 as a 0 due to synchronization issues for example. As a result, the receiver may detect the *EoM* prematurely and stop receiving data even though the sender continues to send data. We cannot address this case as it is a limitation of our design to not include the length of the message to be received.

5.2.4 Influence of the Controller

The OpenFlow controller that is used to covertly communicate is beyond the control of the sender and receiver. Hence, the accuracy and performance of our channel is limited by the controller that operates the OpenFlow network.

Load on the Controller. Typically, there are more switches connected to the controller than just the sender and the receiver of the covert channel. If the communication between the benign switches and the controller is frequent and voluminous, the sender and receiver will experience non-deterministic delays in connecting/disconnecting (δ_{sc}, δ_{dc} and $\delta_{of\text{-}deny}$) to the controller, thereby reducing the performance (throughput and accuracy) of the channel.

Controller Architecture. The system and software architecture of the controller also influences our design. For example, the controller could be single threaded or multi-threaded. The former can lead to long delays, whereas the latter can lead to non-determinism due to the scheduler.

Path to the Controller. Network paths not under the control of the sender and receiver can influence the performance of our channel. For example, buffers in switches can be filled up by other network packets resulting in packet loss and hence errors in the received bits.

5.3 Evaluation

To obtain deeper insights and validate our expectations of our covert channel, we prototyped our design using Open vSwitch [147] and ONOS [145]. Furthermore, we designed a set of experiments based on the challenges described in the previous section to characterize the performance of our channel. We begin with a brief description of our implementation, and then describe the experiments.

Implementation. We used Open vSwitch (OvS) as our sender and receiver OpenFlow switches. We only modified the (OpenFlow) connection handling of OvS so that after it disconnects from the controller, it waits for 4 seconds to reconnect. To set/delete controller information, and configure the DPID, we used the `ovs-vsctl` tool that ships with OvS. We then implemented the sender and receiver algorithms (Alg. 3 and 4) as python scripts. In doing so, we traded performance for *simplicity* which we consider acceptable for the sake of prototyping and evaluation. Our implementation is only meant to demonstrate the *feasibility* of our attack. We synchronized the system clocks of the sender and receiver using our university's NTP time server. To encode and decode the messages, we used the ASCII scheme. We implemented an adaptive *inter-frame delay* synchronization scheme in which the sender sends a frame only at the start of the next second.

Setup. Our evaluation setup comprised of three (sender, receiver and controller) Dell PowerEdge 2950 servers with 4 core Intel(R) Xeon(TM) CPU 3.73GHz processors and 16 GB of RAM each. The sender and receiver were directly connected to the controller. For OpenFlow load generation, we used a fourth server running directly connected to the controller. All these servers used dedicated ports to connect to a management switch that was used for orchestration from a fifth server to conduct the evaluation. All systems ran Ubuntu 14.04.5 LTS. For the sender and receiver, we used Open vSwitch 2.7. For the controller, we used ONOS 1.10.2.

Objectives. Based on our covert timing channel design the objectives of the evaluation are the following. First, we want to establish time intervals that achieve high accuracy and throughput (Sec. 5.2.2. Second, we want to determine the influence the frame length has on the accuracy (Sec. 5.2.3), e.g., do shorter frames have fewer errors than longer frames? Third, we want to measure the influence of δ_{delay} on the accuracy and throughput of our channel (Sec. 5.2.2), e.g., is there a δ_{delay} value for which the time interval can be smaller? Finally, we want to measure the accuracy of our channel when there is load on the controller (Sec. 5.2.4).

Methodology. The general methodology we undertake is the following. The controller runs ONOS with the default applications activated. We program the sender and receiver with a specific start time t, time interval Δ, offset $\delta_{offset} = 5\ ms$, check the connection status at $\Delta/2\ ms$ and frame length Fl. The sender then sends a 64 byte message M_s and the receiver receives a message M_r. We then restart ONOS and OvS, and clean up the OvS database before we repeat the measurement. We collect ten such measurements for the configured values. We measure accuracy as the similarity between M_r and M_s using the *edit distance* or *Levenshtein distance* [67]. For load on the controller, we use OFCProbe [92] as our OpenFlow topology and packet generator. We configure OFCProbe to emulate 20 switches that trigger Packet-Ins to the controller following a Poisson distribution (λ=1). After OFCProbe has started the Packet-in generation, we wait for one minute before we start the sender and receiver, to avoid any warm-up effects from OFCProbe and ONOS.

5.3.1 Experiments

Following the aforementioned methodology, we now describe the experiments and their results.

Effect of Timing Interval Δ. We set the frame length $Fl = 7$, and measure the accuracy for time intervals from 30 ms up to 100 ms. The results are shown in Fig. 5.4.

The results depict that our channel can achieve nearly 100% accuracy for time intervals greater than 60 ms when there is no load on the controller. For $\Delta = 60\ ms$, we have a

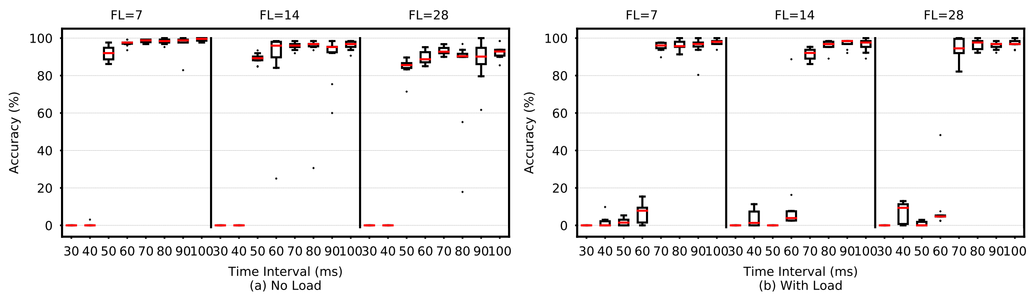

Fig. 5.4: Channel accuracy for time intervals 30-100 ms, and frame lengths 7, 14 and 28 when $\delta_{offset} = 5ms$, OpenFlow status is checked at $\Delta/2$, with no load and with load on the controller.

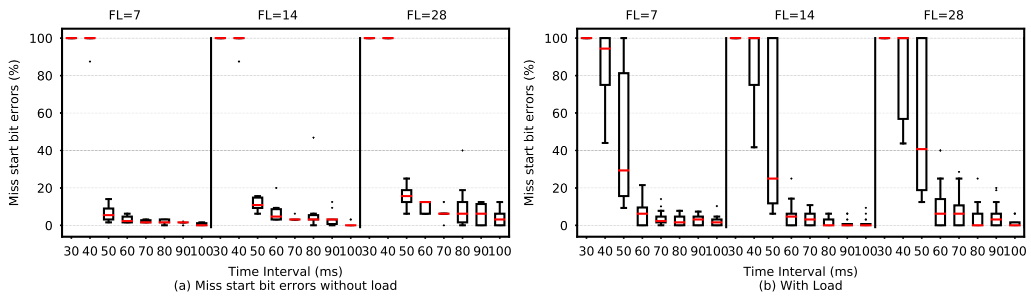

Fig. 5.5: Missed start bit errors for time intervals 30-100 ms, and frame lengths 7, 14 and 28 when $\delta_{offset} = 5ms$, OpenFlow status is checked at $\Delta/2$, and there is no load on the controller.

throughput of approximately 16.67 bps. What we can also see is that as the time interval increases the accuracy increases, which is what we expected. Another distinct observation is that for the values configured, our channel cannot operate below 40 ms because the receiver gets the *EoM* prematurely, it detects only 0 in the data bits as we can see in Figure 5.5 where the missed start bit error (MSB) rate is 100% for time intervals 30 and 40 ms.

Effect of Frame Length Fl. To measure the influence of the frame length on the accuracy we chose the following values: 7, 14 and 28. Note that these values represent the number of data bits in the frame, i.e., 1, 2, and 4 ASCII characters resp. We use only one *SoF* bit in the frame. We repeat the measurements for time intervals from 30-100 ms. The results from this experiment are also depicted in Fig. 5.4.

Indeed, the frame lengths we used show us that as the frame length increases the accuracy drops. Longer frame lengths result in fewer frames but more data per frame being sent. Hence, if the receiver misses the *SoF* bit for $Fl = 14$, it misses twice as many characters compared to $Fl = 7$. Moreover, the chance of incorrect bit detection (bit-flips) increases with larger frames. We analyzed the number of missed *SoF* bits shown in Fig. 5.5 (a) and (b) and we can see that as the frame length increases the missed start bit rate increases from roughly 10% to 20% for time interval 50 ms. To address the problem of missing the start bit we can introduce redundant *SoF* bits.

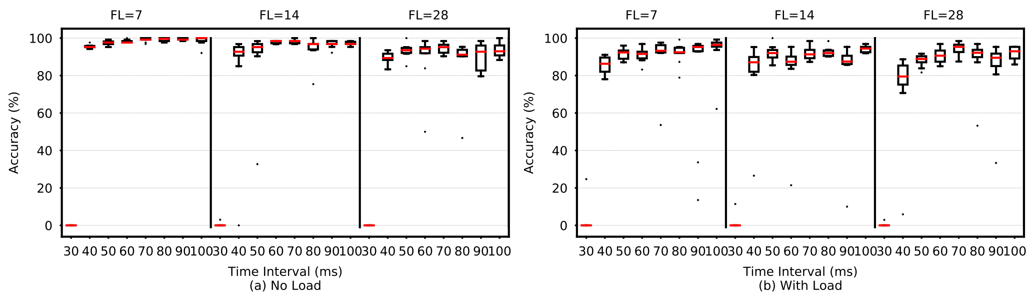

Fig. 5.6: Channel accuracy for time intervals 30-100 ms, and frame lengths 7, 14 and 28 when $\delta_{offset} = 5ms$, OpenFlow status is checked at $2\Delta/3$, with no load and with load on the controller.

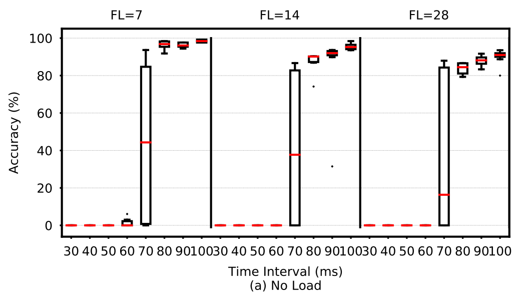

Fig. 5.7: Channel accuracy for time intervals 30-100 ms, and frame lengths 7, 14 and 28 when $\delta_{offset} = 5ms$, OpenFlow status is checked at $\Delta/3$, with no load and with load on the controller.

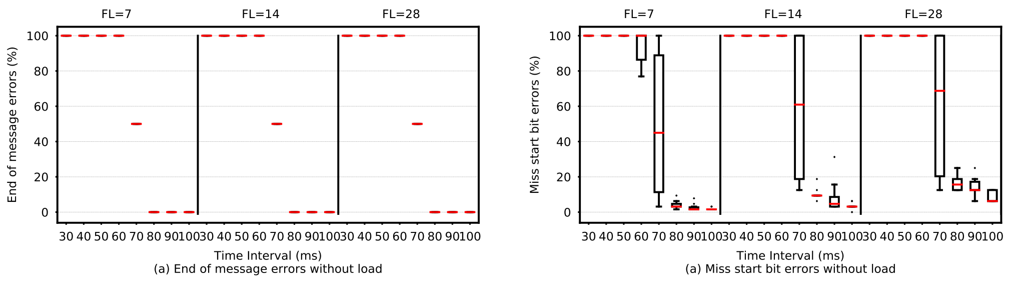

Fig. 5.8: End of message errors and Missed start bit errors for time intervals 30-100 ms, and frame lengths 7, 14 and 28 when $\delta_{offset} = 5ms$, OpenFlow status is checked at $\Delta/3$, with no load.

Effect of δ_{delay} in Checking Connection Status. We now investigate how δ_{delay} influences the throughput and accuracy of our channel. Recall from Sec. 5.1.1.2 that this value is the time the receiver waits before it checks the status of the OpenFlow connection. Until now, we checked the connection status at $\Delta/2$. Hence, in this experiment we check the connection status at $2\Delta/3$ and $\Delta/3$ for frame lengths 7, 14 and 28, and time intervals 30-100 ms. The results for $2\Delta/3$ and $\Delta/3$ are shown in Fig. 5.6 and 5.7 resp.

When we check the status at $2\Delta/3$, we are able to achieve a 25% increase in the throughput. the 40 ms time interval operates at nearly 100 % accuracy. Moreover, the accuracy for this δ_{delay} value performs better compared to our baseline value of $\Delta/2$. When we check the status at $\Delta/3$, we observe a negative influence on the channel, i.e., time intervals 50-70 ms are not effective. In particular, we note that the 70 ms time interval is the operational edge when δ_{delay} is at $\Delta/3$. The reason for these marked changes is the following: The time at which the receiver checks the OpenFlow connection status is crucial. Done too soon, it is likely to detect a zero, and done too late, it is likely to detect a one.

Based on our design, detecting a 1 as a 0 reduces the accuracy more than detecting a 0 as a 1: detecting zeros for all the data bits results in the *EoM* (see Fig. 5.8 (a)) and missing the *SoF* bit (1) can lead to missing the entire frame (see Fig. 5.8 (b)). The two combined can drastically bring down the accuracy which is evidenced when we check the status at $\Delta/3$.

Effect of Message Length $|M|$. To ensure that our channel can sustain longer messages, we measured the accuracy of sending 512 and 1024 byte messages with and without load. The accuracy in each case was very close to the 64 byte message as depicted in Figure 5.9. However, there is slightly more variance when there is load on the controller at the 40 ms time interval.

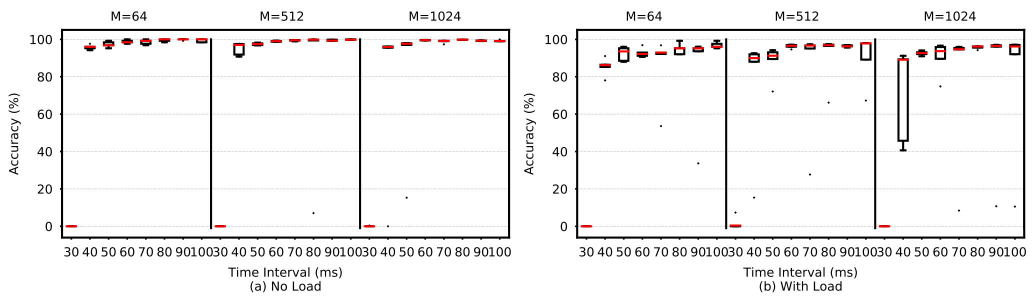

Fig. 5.9: Channel accuracy for message lengths 64, 512 and 1024 for time intervals $30 - 100$ ms, and frame length 7, when OpenFlow status is checked at $2Delta/3$, with no load and with load on the controller.

Effect of Load on the Controller. Having determined time intervals, frame lengths and δ_{delay} values with close to 100 % accuracy, we compare them with measurements when the controller is under load, as real OpenFlow network can operate with more than two switches. Fig. 5.4, 5.6 and 5.9 illustrate the results from this experiment.

Naturally, load on the controller reduces the accuracy of our channel. Other switches trigger events at the controller which introduces queuing and processing delays for the sender's and receiver's messages. This introduces errors for time intervals that were previously highly accurate, e.g., 60 ms and checking the OpenFlow connection at $\Delta/2$ (Fig. 5.4) drops to roughly 10% when the controller is under load. Although there is a drop in the accuracy when we check the connection at $2\Delta/3$ (Fig. 5.6), the smaller time intervals, e.g., 50 ms can still operate at or above 90% accuracy.

5.4 Discussion

Our evaluation demonstrated that switch identification teleportation can be a highly accurate channel for low throughput covert communication in our setup. We also showed that it depends on several factors, e.g., Δ, δ_{delay}, and the system and network conditions. Nonetheless, techniques to detect teleportation in general, and a covert timing channel such as the one presented in this chapter are crucial for networks with high security demands. Hence, we briefly discuss detection possibilities. We also describe some limitations and possible improvements for our design and implementation.

Detection and Mitigation. To the best of our knowledge, firewalls and intrusion detection systems do not monitor the OpenFlow sessions. Even if they do, detecting teleportation attacks are non-trivial as they follow the *normal* pattern of (encrypted) OpenFlow sessions. Preventing switch identification teleportation is exacerbated by the fundamental requirement that switches need to uniquely identify themselves to the controller, and that the controller must allow only a single DPID in the network.

The attack however, can be deterred if OpenFlow connections are secured via the following hardened authentication scheme: unique TLS certificates for switches, white-list of switch DPIDs at controllers [71] which also includes the switches' respective public-key certificate identifier, and lastly a controller mechanism that verifies the DPID announced in the OpenFlow handshake is over the TLS connection with the associated (DPID) certificate.

In lieu of our findings, ONOS has patched their controller [60] and other controllers have also acknowledged the vulnerability: RYU [177] and Redhat's OpenDaylight [24].

Other SDN Protocols. We have also evaluated switch identification with P4Runtime [158] as described by Gbur [65] in his Bachelor's Thesis. We concluded that the attack described in this chapter is not feasible and valuable with P4Runtime for the following reasons.

- The network connection establishment begins from the controller unlike in OpenFlow wherein the switch can initiate the network connectivity. In P4Runtime, the handshake can be viewed as a two step process. First, the controller receives a so-called *device configuration* file that specifies the switch ID, IP address, port number, etc. After processing the file, the second step is taken i.e., *the controller initiates the connection to the switch* at the address and port mentioned in the device configuration file.

- The trigger for the connection from the controller to the switch is determined by the device configuration file. Hence, it is critical to secure this interface. However, if an attacker has already gained access to this interface, then there is little value in using switch identification teleportation, as the attacker can directly manipulate the controller to perform actions at the switches.

- The second observation has been included as a patch to the P4Runtime specification to emphasize the requirement for securing the interfaces that receive the configuration as well as between the switch and controller [194].

Limitations and Improvements. Indeed, our prototype implementation achieves throughput rates in the order of tens of bits per second. However, it is reasonable to assume that the throughput can be increased by, implementing our algorithms in OvS which is programmed in 'C', or using another controller. Consequently, the delays, e.g., δ_{sc}, will be reduced as the response time to events will be faster, e.g., we will not have to rely on `vsctl` and `ovsdb` to set/delete the controller. A novel approach to increase the throughput which we have not measured is for the sender and receiver to initiate several concurrent connections to the controller using unique DPIDs for each connection. In this manner, the sender can send as many bits as connections are made, thereby increasing the throughput by the number of connections. Our channel also comes with some system and network level limitations that are difficult to overcome, e.g., time to establish a TCP connection, packet loss along the path to the controller, etc. Furthermore, our design is for uni-directional communication and does not include error correction. A channel from the receiver back to the sender where the receiver acknowledges, e.g., every frame, can boost the accuracy of the channel.

5.5 Conclusions

In this chapter, we described how switch identification is fundamental to SDN not only due to the requirement for switches to uniquely identify themselves to the controller but also due to the lack of authentication and authorization of DPIDs in the OpenFlow specification and controller implementations. We then elaborated on the design, implementation and evaluation of a novel covert timing channel based on the switch identification teleportation technique.

Our prototype implementation of our design can achieve throughput rates of up-to 20 bits per second, with an accuracy of approximately 90% even when there is load at the controller. This means that a 2048 byte RSA private key file can be transferred in nearly 13 minutes. Although our *proof-of-concept* implementation is a low bandwidth channel, we discussed techniques to increase the throughput and practically mitigate the problem.

This chapter concludes Part I of this dissertation where we saw how the logically centralized controllers can be used in unintended ways by malicious switches (and hosts) to covertly bypass data plane security policies and mechanisms.

Part II

Network Isolation via the Data Plane

The previous two chapters highlighted how malicious switches in the data plane can covertly communicate as well as cause denial of service attacks due to the shared controller, programmability of the network (e.g., via intent networking) and lack of authentication and authorization in specifications and implementations. In Part II of the dissertation, we focus on the design and implementation of data plane systems for virtual networking in the cloud. This involves i) how attackers can break out of network (and VM) isolation and ii) how we can maintain network isolation even under attack.

In Chapter 6, we raise the alarm on the security implications of state-of-the-art virtual switches, a key mechanism for multi-tenant network virtualization in the cloud. In particular, our security analysis shows that existing designs and implementations for network virtualization not only *increase the attack surface* of the cloud, but virtual switch vulnerabilities can also lead to attacks of much *higher impact* compared to mere hardware or software switches. Our findings motivate us to qualitatively analyze existing threat models for network virtualization architectures, and accordingly introduce a new attacker model for virtual switches. We demonstrate the accuracy and practical relevance of our analyses using a case study with Open vSwitch and OpenStack. Our security analysis allowed us to precisely target the packet processing code of OvS to look for vulnerabilities. By fuzzing that specific component, we discovered multiple exploitable memory corruption issues (reported in `CVE-2016-2074` and `CVE-2016-10377`). Using just one vulnerability we were able to create a worm that can compromise hundreds of servers in a matter of minutes. Finally, we find that existing software countermeasures for memory corruption may not always be used by default even though their overhead in user-space has minimal impact (1-15%) on forwarding throughput and latency. However, the overhead of protecting the kernel, e.g., using `grsec`, is prohibitive, it reduces the maximum throughput by nearly half making it unlikely to be used in production networks.

Motivated by the above discoveries, in Chapter 7, we dive deeper into the design of virtual switches. Our qualitative analysis of 23 virtual switches reveals that most virtual switches are designed for performance and flexibility. They have a large trusted computing base, co-located with the Host and perform privileged packet processing which essentially violates basic secure design principles. Hence, we present, implement, and evaluate a scalable virtual switch architecture, `MTS`, which brings four secure design principles to the context of multi-tenant virtual networking: least-privilege vswitch, complete mediation of tenant-Host communication, extra security boundary between the tenant and Host, and least common mechanisms. We build `MTS` from commodity components, providing an incrementally deployable and inexpensive upgrade path to cloud operators using Single-Root I/O Virtualization, VMs and containers. Our extensive experiments, extending to both micro-benchmarks and cloud applications, show that, depending on the way it is deployed, `MTS` may produce 4x isolation and 1.5-2x the throughput compared to state-of-the-art, with similar or better latency and modest resource overhead (1 extra CPU).

6 Reins to the Cloud via the Virtual Switch

At the heart of an efficiently operating datacenter lies the idea of resource sharing and *multi-tenancy*: independent instances (e.g., applications or tenants) can utilize a given infrastructure concurrently, including the compute, storage, networking, and management resources deployed at the data center, in a physically integrated but logically *isolated* manner [110, 62].

The multi-tenant network architecture (recall Chapter 2.3) enables sharing of the network resources among the different users/tenants of the data center network. This comprises of a centralized controller (control plane) and a set of programmable software and hardware switches (data plane). These components form the underlying network infrastructure that are then multiplexed to isolate the different tenants' networks from each other.

Key to network virtualization is the *virtual switch*, a network component located in the virtualization layer of the (edge) servers that connects tenants' compute and storage resources (e.g., VMs, storage volumes, etc.), provisioned at the server, to the rest of the data center and the public Internet [110, 91, 168].

Virtual switches are typically not limited to provide *traditional switching* but support an increasing number of network and middlebox functionality [58, 89], e.g., routing, firewalling, network address translation and load-balancing. Placing such functionality at the edge of the network (i.e., the Host OS/Hypervisor) is attractive, as it allows to keep the network fabric simple and as it supports scalability [170, 58]. A few prominent virtual switches today are: Open vSwitch [171], Cisco Nexus 1000V [236], VMware vSwitch [239] and Microsoft VFP [58].

The general tendency to move functionality from the network fabric to the edge also comes at the price of increased complexity. For example, the number of protocols that need to be parsed and supported by virtual switches (Open vSwitch and Cisco Nexus 1000v) and OpenFlow [127] have been growing steadily over the last years [229] (see Fig. 6.1).

The trend towards more complex virtual switches is worrisome as it may increase the attack surface of the virtual switch. For example, implementing network protocol parsers in the virtual switch is non-trivial and error-prone [193, 200, 57]. These observations motivate us to conduct a security study of network virtualization architectures that use virtual switches.

Structure. Section 6.1 introduces and discusses our security analysis of virtual switches and existing threat models. Based on this analysis we propose a new attacker model. Section 6.2 presents a proof-of-concept case study attack on OvS in OpenStack. Subsequently, we discuss

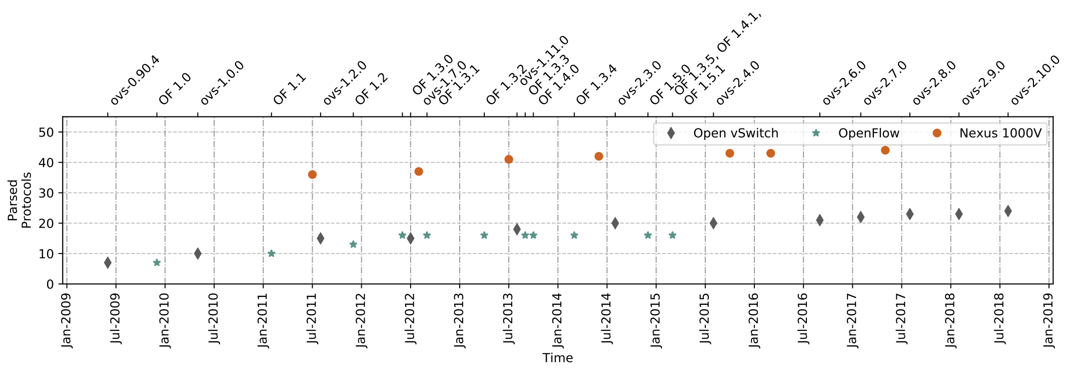

Fig. 6.1: The total number of parsed network protocols in two popular virtual switches (OvS and Cisco's Nexus 1000V) and OpenFlow from 2009-2019.

possible software mitigations and their performance impact in Section 6.3 After discussing related work in Section 6.4, we conclude in Section 6.5.

6.1 Security Analysis

In this section, we present a systematic security analysis of the network virtualization architecture introduced in Chapter 2. Based on these insights, we investigate existing threat models for virtual switches and then construct an attacker model against which virtual switches must be resilient.

6.1.1 Attack Surface and Vulnerabilities

In the following we characterize the attack surface and vulnerabilities of state-of-the-art multi-tenant network virtualization architectures which make them feasible, attractive, and exploitable targets. An overview of the security analysis and the implications are illustrated in Fig. 6.2.

Attacker Facing Component. From an attacker's perspective targeting a component that it can directly communicate with is an excellent choice as it provides control and feedback over the data sent/received. Recall (Section 2.3.1) that the virtual is the last hop switch for the VMs. Hence, for an attacker controlled VM, the virtual switch is a highly suitable target as illustrated in Fig. 6.2 ①.

Centralized Control via Direct Communication. Multi-tenant network virtualization systems are designed for centralized control over all the assigned data plane elements, e.g., software and/or hardware switches (recall Section 2.2 and 2.3). Fig. 6.2 depicts this with the Compute Nodes (OpenStack nomenclature for the hypervisor) connected to the Controller Node. The centralized controller uses its "southbound interface", today most often "OpenFlow", to exchange messages with all the data plane elements. Following datacenter best practises [155] this is often implemented using a trusted management network that is shared by all the data plane elements. This implies that a (compromised) data plane,

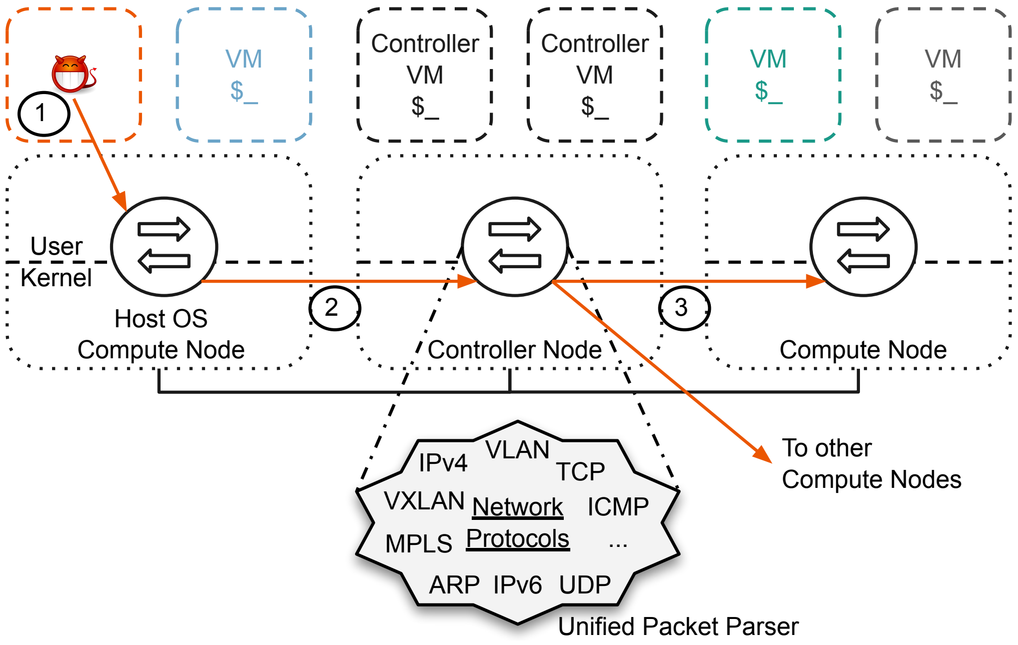

Fig. 6.2: An overview of the security implications of current multi-tenant network virtualization architectures that use virtual switches. An attacker can exploit Host OS co-location, centralized and virtualized control, and complex packet processing (in the Unified packet parser) of untrusted data to launch an attack from a VM on the virtualization layer ((1)). From there, the attacker can propagate to the controller node ((2)) and then compromise other servers in the cloud ((3)).

e.g., virtual switch, can *directly* send packets to the *controller* and/or *all* other *data plane elements*. Management networks, containing only trusted components, are commonly not protected with an additional intrusion detection system. Furthermore, as reported in `CVE-2018-1000155` authentication and authorization are absent in OpenFlow [41].

Virtualized Controllers. The control software for network virtualization and cloud management, for high availability run in VMs on one or more servers [238, 152]. These servers also run and operate a virtual switch that is directly connected to other tenant servers as shown in the Controller Node in Fig. 6.2. This exposes the virtualized controllers to vulnerabilities from the virtual switch. Note that the consequence of this is huge because an attacker from a VM can now exploit a vulnerability in a virtual switch and then compromise the centralized controller server by exploiting the same vulnerability.

Unified Packet Parser. Once a virtual switch receives a packet it parses its headers to determine if it already has a matching flow rule. If this is not the case it will forward the packet to an intermediate data path (slow path) that processes the packet further in order to request a new flow table entry [171, 58] (also see Section 2.3.1. In this step, the virtual switch commonly extracts all header information from the packet, e.g., MPLS and transport layer information, before requesting a flow table entry from the controller. Parsing is the switch's responsibility as centralizing this task would not scale. The additional information from higher-level protocols is used for advanced functionality like load balancing, deep packet inspection (DPI), and non-standard forwarding [58, 89] to support the growing workload demands of data centers [58, 89]. However, with *protocol parsing in the data plane* the virtual switch is as susceptible to security vulnerabilities as any daemon for the parsed

Tab. 6.1: Design characteristics of virtual switches surveyed in this dissertation. `MTS` is presented in this dissertation.

Name	Ref.	Year	Emphasis	Co-Location	Kernel	User
OvS	[169]	2009	Flexibility	✓	✓	✓
Cisco NexusV	[236]	2009	Flexibility	✓	✓	✗
VMware vSwitch	[239]	2009	Centralized control	✓	✓	✗
Vale	[182]	2012	Performance	✓	✓	✗
Research prototype	[98]	2012	Isolation	✗	✓	✓
Hyper-Switch	[176]	2013	Performance	✓	✓	✓
MS HyperV-Switch	[133]	2013	Centralized control	✓	✓	✗
NetVM	[81]	2014	Performance, NFV	✓	✗	✓
sv3	[211]	2014	Security	✓	✗	✓
fd.io	[219]	2015	Performance	✓	✗	✓
mSwitch	[75]	2015	Performance	✓	✓	✗
BESS	[19]	2015	Programmability, NFV	✓	✗	✓
PISCES	[198]	2016	Programmability	✓	✓	✓
OvS with DPDK	[184]	2016	Performance	✓	✗	✓
ESwitch	[137]	2016	Performance	✓	✗	✓
MS VFP	[58]	2017	Performance, flexibility	✓	✓	✗
Mellanox BlueField	[129]	2017	CPU offload	✗	✓	✓
Liquid IO	[163]	2017	CPU offload	✗	✓	✓
Stingray	[70]	2017	CPU offload	✗	✓	✓
GPU-based OvS	[234]	2017	Acceleration	✓	✓	✓
MS AccelNet	[59]	2018	Performance, flexibility	✓	✓	✗
Google Andromeda	[44]	2018	Flexibility and performance	✓	✗	✓
Slim	[251]	2019	Flexibility, Deployability and Security	✓	✓	✓
MTS	[223]	2019	Security and Performance	✗	✗|✓	✓

protocol. Thus, the attack surface of the data plane increases with any new protocol that is included in parsing [193, 14].

Hypervisor Co-Location. The design of virtual switches co-locates them with the hypervisor, e.g., the Host OS's user- and kernel-space, see Figure 6.2 and Table 6.1. Components of the virtual switch often run with elevated privileges in the Host, e.g., using `sudo` in user-space, kernel-space or both. From a performance perspective this is a sensible choice. However, from a security perspective this *co-location* and *elevated privilege* puts all VMs of the Host at risk once an attack against the virtual switch is successful. Recall, such VMs include those that run critical cloud software, e.g., the VM hosting the network controller.

Summary. In combination, the above observations demonstrate that compromising multi-tenant data center networks (including the network virtualization architecture) is a *feasible* threat via the virtual switch. By renting a VM and exploiting a protocol parsing vulnerability an attacker can start her attack by taking over a single virtual switch shown in ① of Fig. 6.2. Thus, she also takes control of the physical machine on which the virtual switch is running due to hypervisor co-location. Next (②), she can take control of the Host OS where the VM running the network virtualization—and in most cases cloud—controller is hosted due to the direct communication channel. Thus, she now controls the whole data center (network and servers). From the controller (③), the attacker can leverage the logically centralized design to rein all other servers, e.g., manipulate flow rules to violate essential network security policies. Alternatively, the attacker can change other cloud resources, e.g., modify the identity management service or change a boot image for VMs to contain a backdoor.

6.1.2 Attacker Models for Virtual Switches

With these vulnerabilities and attack surfaces in mind, we revisit existing threat models. We particularly focus on work starting from 2009 when virtual switches emerged into the network virtualization market [170]. We find that virtual switches are not appropriately accounted for in existing threat models, which motivates us to subsequently introduce a new attacker model.

Existing Threat Models. Virtual switches intersect with several areas of network security research: Data plane, network virtualization, SDN, and the cloud. Therefore, we conducted a qualitative analysis that includes research we identified as relevant to attacker models for virtual switches in the cloud.

Qubes OS [189] in general assumes that the networking stack can be compromised. Similarly, Dhawan et al. [46] assumed that the software-defined network data plane can be compromised. Jero et al. [94] base their assumption on a malicious data plane in an SDN on Pickett's BlackHat briefing [172] on compromising an SDN hardware switch.

A conservative attacker model was assumed by Paladi et al. [161] who employ the Dolev-Yao model for network virtualization in a multi-tenant cloud. Grobauer et al. [72] observed that virtual networking can be attacked in the cloud without a specific attacker model.

Jin et al. [98] accurately described two threats to virtual switches: Virtual switches are co-located with the hypervisor; and guest VMs need to interact with the hypervisor. However, they stopped short of providing a concrete threat model, and underestimated the impact of compromising virtual switches. Indeed at the time, cloud systems were burgeoning. However, only recently Alhebaishi et al. [4] proposed an updated approach to cloud threat modelling wherein the virtual switch was identified as a component of cloud systems that needs to be protected. However, in both cases, the authors overlooked the severity, and multitude of threats that apply to virtual switches.

Motivated by a strong adversary, Gonzales et al. [3], and Karmakar et al. [104] accounted for virtual switches, and the data plane. Similarly Yu et al. [247], Thimmaraju et al. [224] and Feldmann et al. [56] assumed a strong adversarial model, with an emphasis on hardware switches, and the defender having sufficiently large resources.

Hence, we posit that previous work have either assumed a generic adversary model for the SDN data plane, stopped short of an accurate model for virtual switches, undervalued the impact of exploiting virtual switches, or assumed strong adversaries. Given the importance and position of virtual switches in general, and in multi-tenant data center in particular, we describe an accurate, and suitable attacker model for virtual switches in the following.

A New Attacker Model. Given the shortcomings of the above attacker models, we now present a new attacker model for virtual switch based cloud network setups that use a logically centralized (and virtualized) controller. We note here that this attacker model is a finer version of the one described in Chapter 3. Contrary to prior work we identify the virtual switch as a critical core component which has to be protected against direct attacks, e.g., malformed packets. Furthermore, our attacker does not have to be supported by a major organization (she is a “Lone Wolf”) nor does she have access to special network vantage points. The attacker’s knowledge of vulnerability discovery and crafting exploits is not exceptional, with some effort she can discover and create software exploits. In addition, the attacker controls a computer that can communicate with the cloud under attack.

The attacker’s target is a cloud infrastructure that uses virtual switches for network virtualization. We assume that our attacker has only limited access to the cloud. Specifically, the attacker does not have physical access to any of the machines in the cloud. Regardless of the cloud delivery model and whether the cloud is public or not, we assume the attacker can either rent a single VM, or has already compromised a VM in the cloud, e.g., by exploiting a web-application vulnerability [37].

We assume that the cloud *provider* follows security best-practices [155]. Hence, at least three isolated networks (physical/virtual) dedicated towards management, tenants/guests, and external traffic exist. Furthermore, we assume that the same software stack is used across all servers in the cloud.

We consider our attacker successful, if she obtains full control of the cloud. This means that the attacker can perform arbitrary computation, create/store arbitrary data, and send/receive arbitrary data to all nodes including the Internet.

6.2 Case Study: OvS in OpenStack

Based on our analysis, we conjecture that current virtual switch implementations are not robust to adversaries from our attacker model. To test our hypothesis, we conducted a case study. We evaluate the virtual switch Open vSwitch in the context of the cloud operating system OpenStack against our attacker model. We opted for this combination as OpenStack is one of the most prominent cloud systems, with thousands of production deployments in large enterprises and small companies alike. Furthermore, according to the OpenStack Survey 2016 [231], over 60% of OvS deployments are in production use and over one third of 1000+ core clouds surveyed use OvS.

Attack Methodology. We conduct a four step methodology targeting the attack surface we previously described. We first validate the attack surface in OvS and OpenStack in Section 6.2.1, followed by discovering vulnerabilities in OvS’s unified packet parser in Section 6.2.2. In the third step described in Section 6.2.3, we exploit the vulnerability. Finally in Section 6.2.4, we demonstrate the potential of a large-scale compromise using OpenStack and OvS by exploiting the discovered vulnerabilities.

6.2.1 Attack Surface Analysis

The first step in our analysis is validating that OvS processes packets from VMs in OpenStack. We verify this by looking at the configuration and architecture of OpenStack [154]. It is clear that if packets pass through the firewall (`iptables`), OvS will process the packet from the

VM. However, Linux kernel patches [214], RedHat [64], OpenStack [153] and OvS [166] introduced support for firewalling in OvS using `conntrack` which eliminates iptables and thereby giving attackers *direct* access to OvS.

Next, we validate that OvS is indeed co-located with the Host OS of the compute and controller nodes [154]. It is co-located with the host OS's user- and kernel-space [169]. Additionally, the user-space daemon (`ovs-vswitchd`) by default runs with root (`sudo`) privileges as it requires capabilities to communicate with the network card driver. By inspecting the packet processing source code of OvS, we confirm that OvS implements a unified packet parser in its `key_extract` and `flow_extract` functions in the fast-path and slow-path respectively.

Third, OvS supports centralized control via OpenFlow and/or `ovsdb`. In OpenStack this is achieved via the Controller node coordinating and communicating with the Neutron agent and OvS ML2 plugin. Furthermore, we validate that OpenStack control software can run in VMs with OvS co-located with the Host OS of that server [136].

6.2.2 Vulnerability Discovery

Based on our security analysis, we expect to find vulnerabilities in the unified packet parser of OvS. A suitable methodology to discover new vulnerabilities in programs that accept user input is randomized program testing, e.g., *fuzzing*. Hence, we used an off-the-shelf coverage-guided fuzz tester, namely American Fuzzy Lop (AFL) [218], on OvS's unified packet parser in the slow-path.

Having identified the target functions that represent the unified packet parser in OvS, we used the `test_flows` test case that ships with OvS as our program input to AFL. The `test_flows` program requires two input files: a pcap file that represents packets in the pcap format, and a flows file that represents OvS/OpenFlow flow rules. We used the `PERL` script [240] that ships with OvS to generate the flows and pcap files that can be given as input to `test_flows`. With the inputs available, the pcap that was generated was also used as a seed to AFL. This aided AFL in randomly generating new packets (in the pcap format) that would then exercise the `flow_extract` function.

When this work was conducted, we fuzzed three different versions of OvS: OvS-2.3.2 (Long Term Support (LTS)), OvS-2.4.0 (latest stable) and OvS-2.5.0 (LTS). For OvS-2.3.2 the AFL system executed for a little more than 1.5 days, for OvS-2.4.0 AFL executed for about 3 months and for OvS-2.5.0 AFL executed for 3 hours.

AFL reported several thousand crashes which were manually triaged. Upon analysis we observed thousands of duplicates, hence we developed heuristics to reduce the set of crashes to only unique ones which we describe next.

`CVE-2016-2074`. Using the above methodology, we identified 3 unique memory corruption vulnerabilities in the unified packet parser of OvS (`ovs-vswitchd`). In this section we focus on one of the vulnerabilities reported in `CVE-2016-2074` [43] we found in the then LTS

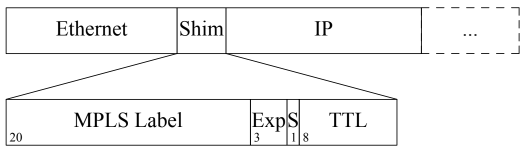

Fig. 6.3: MPLS label stacks are placed in the Shim header between the Ethernet and IP headers.

stable branch (v2.3.2), as it was the only one that gives an attacker remote control over the switch. Further vulnerabilities discovered during our study include exploitable parsing errors leading to denial of service (DoS) also reported in `CVE-2016-2074` in OvS-2.4.0 and an ACL bypass vulnerability in `CVE-2016-10377` [42] in the unified packet parser of OvS-2.5.0.

The vulnerability is a stack buffer overflow introduced by the MPLS parsing code of the OvS slow-path. We acknowledge that stack buffer overflows and how they are exploited are well understood. However, we fully document it here to: (i) Underline how such vulnerabilities can occur, especially in software handling network packets, and, (ii) To make our work more accessible in the context of networking research outside the security community.

To understand the finer details of the vulnerability, we make briefly digress by summarizing the MPLS protocol. MPLS is often deployed to address the complexity of per packet forwarding lookups, traffic engineering, and advanced path control. MPLS uses "Forwarding Equivalence Classes" (FECs) to place a "label" in the *shim header* between the Ethernet and the IP header [187] of a packet as shown in Figure 6.3). This label is then used for forwarding. In addition, labels can be stacked via *push* and *pop* operations.

An MPLS label is 20 bits long, followed by the Exp field of 3 bits reserved space. This is followed by the 1 bit S field, which, if set to 1, indicates that the label is the bottom of the label stack. It is a critical piece of "control" information that determines how an MPLS node parses a packet. The TTL field indicates the Time-To-Live of the label. MPLS labels should be under the providers' administration, e.g., offering L2/L3 VPNs, and are negotiated using protocols such as LDP (Label Distribution Protocol) [7]. As per RFC 3032, MPLS labels are inherently trusted.

Now we return to the vulnerability description. When a packet arrives, the function `flow_extract` is eventually called (see Listing 6.1), which creates a local structure `m` (line 8) that contains a local buffer `buf` (line 7). Based on the contents of the packets, the buffer is initialized in line 12, and then filled up accordingly by calling `miniflow_extract` in line 13. Since the `ETHER_TYPE` is set to `0x8848` for MPLS packets, the function `parse_mpls` in invoked on the packet (shown in the call stack in Listing 6.2). The function parses *all* the labels to count the number of labels present in the label stack, and returns the maximum number between what it counted and a static variable `FLOW_MAX_MPLS_LABELS` (which is set to 3). When there are hundreds of labels, e.g., 200, the comparison yields 200, which results in copying 800 bytes of labels from the packet into the structure `m` (line 8 from Listing 6.1), overflowing `buf` which is supposed to be the size of `FLOW_U32S` and thereby overwriting the return address to the function that called `flow_extract`.

```
 1  void flow_extract(struct ofpbuf *packet,
 2                    const struct pkt_metadata *md,
 3                    struct flow *flow)
 4  {
 5      struct {
 6          struct miniflow mf;
 7          uint32_t buf[FLOW_U32S];
 8      } m;
 9
10      COVERAGE_INC(flow_extract);
11
12      miniflow_initialize(&m.mf, m.buf);
13      miniflow_extract(packet, md, &m.mf);
14      miniflow_expand(&m.mf, flow);
15  }
```

Listing 6.1: Source code of the function in which the buffer `buf` (line 7) overflowed.

```
flow_extract(struct ofpbuf *packet,
             const struct pkt_metadata *md,
             struct flow *flow)
...
miniflow_extract(packet, md, &m.mf)
...
count = parse_mpls(&data, &size);
miniflow_push_words(mf, mpls_lse, mpls, count);
miniflow_push_words_(MF, offsetof(struct flow, FIELD), VALUEP, N_WORDS)
memcpy(MF.data, (VALUEP), (N_WORDS) * sizeof *MF.data);
```

Listing 6.2: The call stack that leads to the buffer overflow

To prevent attackers from exploiting such overflows, Cowan et al [38] introduced stack canaries (known as StackGuard in their paper). The compiler inserts a random value, the canary, before the return instruction of a function during compilation. Then at run time, when the function returns, it first ensures that the canary is as expected, if not, the program aborts, hence preventing an attacker from controlling the instruction pointer (i.e., by jumping to another function/address). In the case of OvS-2.3.2 and Ubuntu-14.0.4, `gcc-4.8.4` is the default compiler which by default did not insert a canary before the `flow_extract` stack frame. Compiling OvS with `gcc-5.4.0` however does insert a canary over `flow_extract`. We did not investigate this discrepancy in compiler behavior. However, we did ascertain that with the `-fstack-protector-all` option, gcc does insert a canary.

Remark on Parsing MPLS. Indeed, we note that the specification of MPLS, see RFC 3031 [188] and RFC 3032 [187] does not specify how to parse the whole label stack. Instead, it specifies that when a packet with a label stack arrives at a forwarding component, only the top label must be popped to be used to make a forwarding decision. Yet, OvS parses all labels of the packet even beyond the supported limit and beyond the pre-allocated memory range for that stack. If MPLS is to handled as per the RFC, only the top label should be popped, which has a static, defined size. Thus, there would be no opportunity for a buffer overflow.

6.2.3 Exploiting CVE-2016-2074 as a Worm

The pure presence of a vulnerability is not sufficient to state that OvS is not robust against our threat model. We have to demonstrate that the vulnerability does enable a large-scale compromise. Thus, we need to turn the vulnerability into an exploit. Here, we use a common exploit technique, namely Return Oriented Programming (ROP) [185], to realize a worm that can fully compromise an OpenStack setup within minutes.

We implement the ROP [185] attack in an MPLS packet payload as that it is in the MPLS label stack processing that the vulnerability exists. By now, ROP attacks are well documented and can be created by an attacker who has explored the literature on implementing ROP attacks, e.g., using RopGadget [186]. Nonetheless, we briefly describe ROP here and suggest the reader to refer to Roemer et al. [185] for further details.

Return Oriented Programming. ROP attacks re-combine instruction sequences (called gadgets) from a (target) binary to execute (arbitrary) code. A gadget typically consists of one or more operations followed by a return instruction. After executing each gadget, the return will pop the address of the next gadget into the instruction pointer. The sequence of gadgets that facilitate the execution of (desired) code is called a ROP chain. Given a stack buffer overflow, the attacker overwrites the stack frame with such a ROP chain. Restoration of the overwritten return instruction pointer diverts the control flow of the target program to the first gadget. Once control reaches a return instruction, the next attacker controlled address will be loaded into the instruction pointer.

Exploit. As per RFC 3032 [187] MPLS label processing terminates if the *S* bit is set to 1. Hence, the gadgets that make up the ROP chain must be selected such that the *S* bit is set to 0. This ensures that we can successfully overflow the buffer with the necessary instructions (that redirects a shell back to the attacker) before terminating the mpls label stack processing. Hence, we select appropriate gadgets by customizing Ropgadget and modify the shell command string. To handle change in packet size due to tunneling protocols, our exploit also implements a NOP sleigh: the initial gadgets don't do anything meaningful (no operations).

The ROP chain in our exploit packet starts with the Ethernet header and padding, followed by the MPLS labels. Our example ROP payload connects a shell on the victim's system (the server running *ovs-vswitchd*) to a listening socket on the remote attacker's system. To spawn the shell the payload triggers the execution of the `cmd` `bash -c "bash -i >& /dev/tcp/<IP>/<PORT> 0>&1"` through the *execve* system call (0x3b). This requires the following steps:

1. Set-up the shell command (`cmd`) string in memory;
2. construct the argument vector `argv`;
3. place the address of the command string in the register `%rdi`;
4. place the address of `argv` in `%rsi`;
5. place the address of `envp` in `%rdx`;
6. place the system call number `0x3b` in `%rax`; and finally
7. execute the system call, `execve`.

In summary, our exploit could also have been created by an attacker familiar with the tools and literature with this kind of technique. This is in accordance with our attacker model, which does not require an uncommonly skilled attacker.

Worm Implementation. To propagate the worm, multiple steps need to be automated. These are visualized in Figure 6.2. In Step ①, the worm originates from an attacker-controlled (guest) VM within the cloud and compromises the host operating system of the server via the vulnerable packet processor of the virtual switch. Once she controls the server, she patches `ovs-vswitchd` on the compromised host, as otherwise the worm packet cannot be propagated. Instead the packet would trigger the vulnerability in OvS yet again.

With the server under her control the remote attacker, in Step ②, propagates the worm to the server running the controller VM and compromises it via the same vulnerability. The centralized architecture of OpenStack requires the controller to be reachable from all other servers via the management network and/or guest network. By gaining access to one server we gain access to these networks and, thus, to the controller. Additionally, the virtualization of the control software (in VMs) necessitates its Host OS to also have a virtual switch. Hence, if we compromise the virtual switch on the controller, we compromise the entire controller server. Network isolation using VLANs and/or tunnels (GRE, VXLAN, etc.) does not prevent the worm from spreading once the server is compromised.

With the controller's server also under the control of the remote attacker, the worm again patches `ovs-vswitchd` and can then taint the remaining uncompromised server(s) (Step ③). Thus, finally, after Step ③, all servers are under the control of the remote attacker. We automated the above steps using a shell script.

6.2.4 Attack Evaluation

Rather than evaluating the attack in the wild we chose to create a test setup in a lab environment. More specifically, we use the Mirantis 8.0 distribution that ships OpenStack "*Liberty*" with OvS version 2.3.2. On this platform we set up multiple VMs. The test setup consists of a server (the fuel master node) that can configure and deploy other OpenStack nodes (servers) including the OpenStack controller, compute, storage, network. Due to limited resources, we created one controller and one compute node with multiple VMs in addition to the fuel master node using the default Mirantis 8.0 configuration. Virtual switching was handled by OvS.

The attacker was given control of one of the VMs on the compute server and could deploy the worm from there. Since, we are sending MPLS packets, it will bypass `iptables` and reach OvS. It took less than 20 seconds until the worm compromised the controller. This means that the attacker has root shell (ovs-vswitchd runs as root) access to the compute node as well as the controller. This includes 3 seconds of download time for patching `ovs-vswitchd` (OvS user-space daemon), the shell script, and the exploit payload. Moreover, we added 12 seconds of sleep time for restarting the patched `ovs-vswitchd` on the compute node so that attack packets could be forwarded.

Next, we added 60 seconds of sleep time to ensure that the network services on the compromised controller were restored. Since all compute nodes are accessible from the controller, we could compromise them in parallel. This takes less time than compromising the controller, i.e., less than 20 seconds. Hence, we conclude that the compromise of a standard cloud setup can be performed in less than two minutes.

6.2.5 Summary

Our case study demonstrates how easily an amateur attacker can compromise the virtual switch, and subsequently take control of the entire cloud in a matter of minutes. This can have serious consequences, e.g., amateur attackers can exploit virtual switches to launch ransomware attacks in the cloud. This is a result of complex packet parsing in the unified packet parser, co-locating the virtual switch with the virtualization layer, centralized and virtualized control, and inadequate attacker models.

6.3 Software Countermeasures

There exist many mitigations for attacks based e.g., on buffer overflows, including MemGuard [38], control flow integrity [1], position independent executables (PIEs) [164], and Safe (shadow) Stack [116]. Any one of these severely reduces the impact of crucial, frequently occurring vulnerabilities like the one used as an example in this chapter. However, due to their assumed performance overhead, especially on latency, they are commonly not deployed for virtualized network components.

Hence, while these mitigations are widely available, we find that they are not enabled by default for OvS. Furthermore, virtual switch solutions presented in the literature commonly do not discuss these techniques. One possible downside of these mitigations is their performance overhead. Past work reported that MemGuard imposes a performance overhead of 3.5–10% [38] while PIEs have a performance impact of 3–26% [164]. Furthermore, prior evaluations did not focus on the systems' network performance. Instead, their main focus was on the systems' process performance, e.g., kernel context switches and the size of compiled binaries with the applied mitigations. However, in the context of OvS, network related metrics are far more relevant: Forwarding latency and forwarding throughput.

In order to investigate the potential performance penalty of such countermeasures, we showcase two variants of these mitigation techniques that are supported by the Gnu cc compiler `gcc` out of the box. Namely, stack protector and position independent executables. To determine the practical impact of these mitigations, we designed a set of experiments to evaluate the performance impact on OvS's forwarding latency and throughput.

Evaluation Setup. The test setup is chosen to ensure accurate one-way delay measurements. Thus, for our tests, we use three systems, all running Linux kernel (v4.6.5) compiled with gcc (v4.8). The systems have 16GB RAM, two dual-core AMD x86_64 2.5GHz, and four Intel Gigabit NICs. The systems are interconnected as follows: One system serves as the Load Generator (LG) and replays packet traces according to the specific experiments. This system is connected to the Device Under Test (DUT), configured according to the different evaluation parameters. The data is then forwarded by OvS on the DUT to a Load Receiver (LR), a third system.

The connections between LG and DUT, and, LR and DUT respectively are monitored via a passive taping device. Both taps are connected to our measurement system. This system has two dual-core Intel(R) Xeon(TM) CPUs running at 3.73GHz with hyperthreading enabled and 16GB RAM. We use an ENDACE DAG 10X4-P card to capture data. Each line (RX/TX) of the tapped connections is connected to one interface of the DAG 10X4-P. Each interface has its own receive queue with 1GB. This ensures accurate one-way delay measurements with a high precision, regardless of the utilization of the measurement host.

Evaluation Parameters. We evaluate forwarding latency and throughput for eight different combinations of traffic composition and software mitigations. We compare a vanilla Linux kernel (v4.6.5) with the same kernel integrated with `grsecurity` patches (v3.1), which protects the in-kernel fast-path by preventing kernel stack overflow attacks using stack canaries, address space layout randomization and ROP defense. For both kernels, we evaluate two versions of OvS-2.3.2: The first one compiled with `-fstack-protector-all` for unconditional stack canaries and `-fPIE` for position independent executables; the second one compiled without these two features. Since `gcc`, the default compiler for the Linux kernel, does not support Safestack (safe and unsafe stack) we did not evaluate this feature, even though it will be available with `clang`, another compiler, starting with version 3.8. The selected mitigations increase the total size of `ovs-vswitchd` from 1.84 MB to 2.09 MB (+13.59%) and `openvswitch.ko` from 0.16 MB to 0.21 MB (+31.25%). However, apart from embedded systems, the size changes are not relevant on modern systems with several hundred gigabytes of memory.

One important feature in virtual switches, recall Section 2, is, whether traffic is handled by the slow or the fast path. We decided to focus on the corner cases where traffic is either handled exclusively by the fast or by the slow path. By isolating the two cases we can assess if and to what extent the software security options impact each path. Hereby, we follow current best practices for OvS benchmarking, see Pfaff et al. [171]. To trigger the slow path for all packets in our experiments, we disable the *megaflows cache* and replay a packet trace in which each packet has a new source MAC address (via sequential increments). For measuring fast path performance, we pre-establish a single flow rule on the DUT, a wildcard-one, that matches all packets entering from the LG. The rule instructs the virtual switch to process these packets via the fast path and forward them on the interface connected to the LR. Therefore, for the sake of consistency, we can replay the same traces as used for the slow path experiments. Additionally, to reduce the uncertainty in our setup, we pin `ovs-vswitchd` to a single core.

Latency Evaluation. For the latency evaluation, we studied the impact of packet size on OvS forwarding. We selected the following packet sizes from the legacy MTU range: $64B$ (minimum IPv4 UDP packet size), $512B$ (average packet), and $1500B$ (maximum MTU) packets. In addition, we also select the following jumbo frames: $2048B$ packets (small jumbo frame) and $9000B$ (maximum jumbo frame). For each experimental run, i.e., packet size and parameter set, we continuously send 10,500 packets from the LG to the LR via the

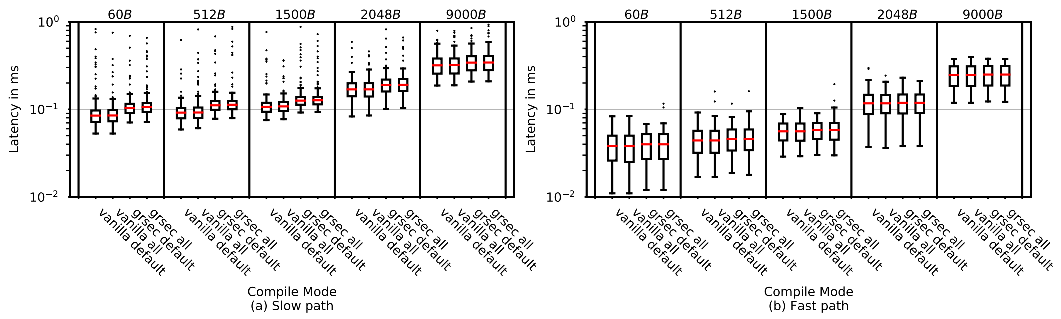

Fig. 6.4: Forwarding latency of OvS with and without countermeasures on a vanilla kernel and `grsecurity` enabled kernel in the slow and fast path.

DUT at a rate of 10 packets per seconds (pps). To eliminate possible build-up or pre-caching effects, we only evaluate the last 10,000 packets of each experiment.

The results for the latency evaluation are depicted in Figures 6.4 (a) and (b) for the slow path and fast path resp. We find that `grsecurity` (grsec default and grsec all) imposes a minimal increase in latency for all packet sizes in the slow and fast path. We observe that user-land protection mechanisms (Fig. 6.4 (a).) impose a small overhead (1-5%) in the slow path for a vanilla and `grsecurity` enabled kernel. Naturally, there is no impact of the user-land protection mechanisms in the fast path, see Fig. 6.4 (b).

Throughput Evaluation. For the throughput evaluation we use a constant stream of packets replayed at a specific rate. We opted for small packets to focus on the packets per second (pps) throughput rather than the bytes per second throughput. Indeed, pps throughput indicates performance bottlenecks earlier [90] than bytes per second. As in the latency experiments, we opted to use packets that are $64B$ long.

Each experimental run lasts for 1000 seconds and uses a specific replay rate. Then we reset the system and start with the next replay rate. Our evaluation focuses on the last 900 seconds. For the slow path, the replay rates start from $10k$ to $40k$ packets per second, in steps of $1k$ pps. For the fast path, the replay rates start from $300k$ to $900k$ packets per second, in steps of $10k$ pps. For better readability we show the slow path plot from $10k$ to $35k$ pps.

An overview of the results for the slow and fast path throughput measurements are depicted in Figures 6.5 (a) and (b) resp. In the slow path, packet loss for the vanilla kernel first sets in just after $18k$ pps, while the experiments on the `grsecurity` enabled kernel already exhibit packet loss at $14k$ pps. In the fast path, grsec exhibits packet loss from $350k$ pps whereas the vanilla kernel starts to drop packets at $690k$ pps. Hence, we note that the `grsecurity` kernel patch does have a measurable impact on the forwarding throughput in the slow and fast path of OvS: it reduces the maximum throughput by half. With respect to the user-land security features, we observe an overhead only in the slow path of approximately 4-15%.

Key takeaways. Our measurements demonstrate that user-land mitigations do not have a large impact on OvS's forwarding performance. However, `grsecurity` kernel patches do cause a performance overhead for latency as well as throughput. Given that cloud

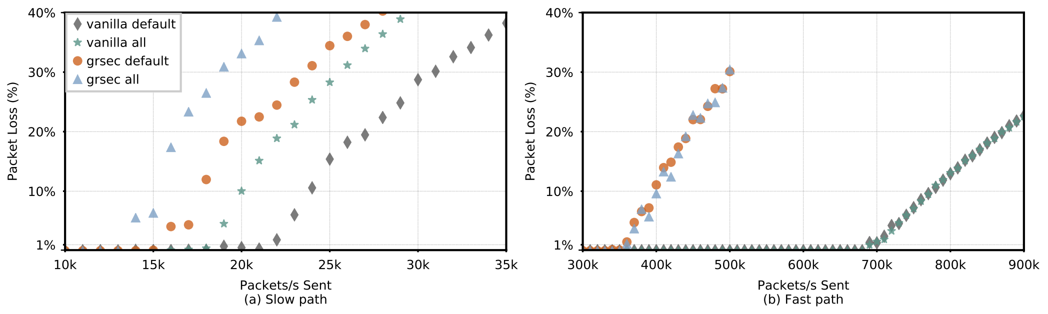

Fig. 6.5: Forwarding throughput of OvS with and without countermeasures on a vanilla kernel and `grsecurity` enabled kernel in the slow and fast path.

systems support a variety of workloads, e.g., low latency or high throughput, kernel-based mitigations are unlikely to be used. However, cloud systems such as the one studied by Pfaff et al. [171] can adopt the user-land and kernel software mitigations described in this paper.

It is only a question of time until the next wormable vulnerability in a virtual switch is discovered. As software mitigations can be more easily deployed than a fully re-designed virtual switch ecosystem, we strongly recommend the adoption of software countermeasures, until a more securely designed virtual switch platform can be rolled out (the subject of the next chapter).

6.4 Related Work

Cloud systems. In the past, various attacks on cloud systems have been demonstrated. Ristenpart et al. [179] showed how an attacker can co-locate her VM with a target VM to obtain secret information. Costin et al. [37] found vulnerabilities in web-based interfaces operated by cloud providers. Wu et al. [244] assessed the network security of VMs in computing clouds. They pointed out what sniffing and spoofing attacks a VM can carry out in a virtual network. Ristov et al. [180] investigated the security of a default OpenStack deployment and show that it is vulnerable from the inside rather than the outside. Indeed, the OpenStack security guide [155] mentions that OpenStack is inherently vulnerable to insider threats due to bridged domains (Public and Management APIs, Data and Management, etc.).

SDN security. Several researchers have pointed out security threats for SDN. For example, Klöti et al. [108] report on STRIDE, a threat analysis of OpenFlow, and Kreutz et al. [111] survey several threat vectors that may enable the exploitation of SDN vulnerabilities. So far, work on how to handle malicious switches is sparse. Sonchack et al. describe a framework for enabling practical software-defined networking security applications [209] and Shin et al. [204] present a flow management system for handling malicious switches. Work on compromised data planes is sparse as well. For example, Matsumoto et al. [125] focus on insider threats. Dhawan et al. [46] investigated methods to detect topology based attacks from the data plane. Their proposal, SPHINX, aims at detecting attacks on the control and data plane by validating updates to network topology. It relies on a majority of uncompromised network entities to maintain a valid model of the network topology and flags divergence from this state based on a set of general as well as custom policies. Hong et al. [76] focus on how the controller's view of the network (topology) can be compromised. They identify topology based attacks in an SDN that allow an attacker to create false links to perform man-in-the-middle and blackhole attacks. Although they discovered novel SDN attacks, their threat model does not account for a compromised data plane.

Data plane security. Lee et al. [119] investigated how malicious routers can disrupt data plane operations, while Kamisinski et al. [102] demonstrate methods to detect malicious switches in an SDN. In addition, Chasaki et al. [33, 32] uncover buffer overflows in Click a software switch. To mitigate this, they propose a secure packet processor, which can preserve its own control flow. In contrast to our work, they do not identify the inherent attack surface in the conceptual design of virtual switches. Porez-Botero et al. [165] characterize possible hypervisor vulnerabilities and identify Network/IO as one. In contrast to our work, they omit a deep analysis on the challenges introduced by co-located data planes. Hence, they did not find any network based vulnerabilities. Dobrescu et al. [47] develop a data plane verification tool for the *Click* software. They prove properties such as crash-freedom, bounded execution, or filtering correctness for the switch's data plane. Although software verification can ensure the correctness and security of greenfield software data plane solutions, they currently fall short of ensuring this for legacy software. In such a scenario, coverage guided fuzz testing is a more appropriate approach. More recently, Shin-Yeh et al. [232] described their security analysis of a popular data plane protocol RDMA and their hardware implementations. They identified many security issues, e.g., lack of accountability, denial of service and predictable hardware managed keys. Furthermore, they were able to design and implement side channel attacks [233] that enables an attacker to decipher access patterns to key-value stores of victims in the cloud.

6.5 Conclusion

In this chapter we presented our study of the attack surface of today's virtual switches as they are the linchpins in multi-tenant network virtualization architectures in data centers. We demonstrated that the network virtualization architecture is susceptible to attack via the virtual switch by *design*. Furthermore, we pointed out that existing threat models for virtual switches are insufficient. Accordingly, we derived a new attacker model for virtual switches and underlined this by demonstrating a successful attack against OpenStack. We discovered multiple vulnerabilities in OvS by fuzzing the packet parsing logic which we could then exploit to break out of VM and network isolation and spread across the data center.

To counter the memory corruption vulnerabilities we discovered, we evaluated the forwarding performance impact of readily available software countermeasures (stack canaries and PIE). The key takeaway message here is to use such mechanisms in the user-space as the security-performance tradeoff is pragmatic. To address the deeper issue namely, insecure vswitch designs, we propose a practical solution next.

7 Bringing Multi-Tenancy to Network Virtualization

In the previous chapter, we uncovered a serious multi-tenancy problem with a popular virtual switch (OvS). An adversary could not only break out of the VM and attack all applications on the Host, but could also manifest as a worm, and compromise an entire datacenter in a few minutes. Along similar lines, Csikor et al. [39] uncovered a performance isolation violation also in OvS, resulting in a cross-tenant denial-of-service attack on the virtual networking system.

Such attacks may exacerbate concerns surrounding the security and adoption of public clouds (that is already a major worry across cloud users [196]). Hence, we argue that despite the wide-scale deployment [44, 59, 99], the level of (logical and performance) isolation provided by vswitches is not well-understood. Indeed, a closer look at the cloud virtual networking best-practice, whereby *per-tenant logical datapaths are deployed on a single host-based vswitch using flow-table-level isolation* [110, 91, 168], reveals that the current state-of-the-art violates basically all relevant secure system design principles [191, 20].

First, the principle of *least privilege* would require that any system component should be given only the minimum set of privileges necessary to complete its task, yet we see that vswitch code typically is *co-located with the Host* and executes with administrator, or what is worse, with full kernel privilege [239, 75], even though this would not be absolutely necessary (see Sec. 7.4). Second, untrusted user code directly interacts with the vswitch and hence with the Host OS, e.g., it may send arbitrary packets from VMs, query statistics, or even install flow table entries through side channels [39], which violates the secure design principle of *complete mediation*. But most importantly, the shared vswitch design goes directly against the principle of the *least common mechanism*, which would minimize the amount of resources common to more than one tenant.

These design flaws motivates us to revisit the fundamental design of vswitches. Hence, in this chapter, we present, implement, and evaluate a scalable multi-tenant (virtual) switch architecture, `MTS`, which extends the benefits of multi-tenancy to the vswitch in a secure manner, without imposing prohibitive resource requirements or jeopardizing performance.

Organization. We dive deeper into designing a secure vswitch in Section 7.1. In Section 7.2 we highlight the nuances in our threat model followed by the design principles and security levels of `MTS` in Section 7.3. In Section 7.4 we elaborate on `MTS` and report on two evaluations in Sections 7.5 and 7.6. In Section 7.7 we introduce our approach to scaling `MTS` using Containers as well as a preliminary evaluation. We enter a discussion of `MTS` in Section 7.8, review related work in Section 7.9 and finally draw conclusions in Section 7.10.

Tab. 7.1: Design characteristics of surveyed virtual switches in this dissertation. `MTS` is presented in this chapter.

Name	Ref.	Year	Emphasis	Monolithic	Co-Location	Kernel	User
OvS	[169]	2009	Flexibility	✓	✓	✓	✓
Cisco NexusV	[236]	2009	Flexibility	✓	✓	✓	✗
VMware vSwitch	[239]	2009	Centralized control	✓	✓	✓	✗
Vale	[182]	2012	Performance	✓	✓	✓	✗
Research prototype	[98]	2012	Isolation	✓	✗	✓	✓
Hyper-Switch	[176]	2013	Performance	✓	✓	✓	✓
MS HyperV-Switch	[133]	2013	Centralized control	✓	✓	✓	✗
NetVM	[81]	2014	Performance, NFV	✓	✓	✗	✓
sv3	[211]	2014	Security	✗	✓	✗	✓
fd.io	[219]	2015	Performance	✓	✓	✗	✓
mSwitch	[75]	2015	Performance	✓	✓	✓	✗
BESS	[19]	2015	Programmability, NFV	✓	✓	✗	✓
PISCES	[198]	2016	Programmability	✓	✓	✓	✓
OvS with DPDK	[184]	2016	Performance	✓	✓	✗	✓
ESwitch	[137]	2016	Performance	✓	✓	✗	✓
MS VFP	[58]	2017	Performance, flexibility	✓	✓	✓	✗
Mellanox BlueField	[129]	2017	CPU offload	✓	✗	✓	✓
Liquid IO	[163]	2017	CPU offload	✓	✗	✓	✓
Stingray	[70]	2017	CPU offload	✓	✗	✓	✓
GPU-based OvS	[234]	2017	Acceleration	✓	✓	✓	✓
MS AccelNet	[59]	2018	Performance, flexibility	✓	✓	✓	✗
Google Andromeda	[44]	2018	Flexibility and performance	✓	✓	✗	✓
Slim	[251]	2019	Flexibility, Deployability and Security	✓	✓	✓	✓
MTS	[223]	2019	Performance, Security	✗	✗	✗|✓	✓

7.1 Securing Virtual Switches

As demonstrated in previous work [227, 39], the current state-of-the-art in virtual switch design can be exploited to not only break network isolation, but also to break out of a virtual machine. This motivates us to identify requirements and design principles that make virtual switches a dependable component of the data center.

7.1.1 State-of-the-Art

Virtual networks in cloud systems using virtual switches typically follow a *monolithic* architecture, where a single controller programs a *single vswitch* running in the Host OS with per-tenant logical datapaths in the vswitch. Isolation between tenants is at the level of flow-tables [110, 91, 168]: the controller populates the flow tables in each per-tenant logical datapath with sufficient flow rules to connect the tenant's Host-based VMs to the rest of the data center and the public Internet. Those sets of flow rules are complex: with a small error in one rule potentially having security consequences, e.g., making intra-tenant traffic visible to other tenants.

As shown in Table 7.1, 95% of the surveyed vswitches are *monolithic* in nature. A single vswitch is installed with flow rules for all the tenants hosted on the respective server. This increases the trusted computing base (TCB) of the single vswitch, as it is responsible for Layer 2-7 of the virtual networking stack. Next, nearly 80% of the surveyed vswitches are co-located with the Host virtualization layer. This increases the TCB of the server since a vswitch is a complex piece of software, consisting of tens of thousands of lines of code. The complexity of network virtualization is further increased by the fact that packet processing

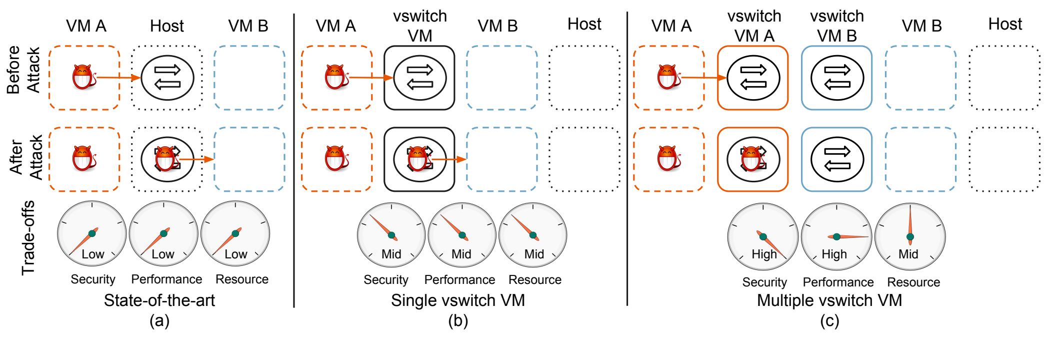

Fig. 7.1: A high-level view depicting the security-performance-resource tradeoffs for the state-of-the-art and MTS network virtualization architectures.

for one-third the vswitches is spread across user space *and* kernel. Close to 70% of the surveyed virtual switches have implemented their packet processing in kernel only or user space and kernel (see last two columns in Table 7.1). These concerns are partially addressed by the current industry trend towards offloading vswitches to *smart NICs* [99, 163, 70, 129]. Indeed consolidating the vswitch into the NIC can improve the security as it reduces the TCB of the Host. These burgeoning architectures, however, share the main trait that the per-tenant logical datapaths are monolithic, often with full privilege and direct access to the Host OS, which when compromised can break network isolation and be used as a stepping-stone to the Host.

7.1.2 Secure vswitch Design

Fig. 7.1 illustrates the key idea underlying the MTS design, along with the security-performance-resource tradeoffs for different architectures. The current vswitch architecture is shown in Fig. 7.1(a), whereby per-tenant logical datapaths share a common (physical or software) switch component deployed at the Host hypervisor layer (in the rest of this chapter, we shall sometimes refer to this design point as the “Baseline”). As we argued at the beginning of this chapter, this design is fundamentally insecure [227, 39] as it violates basic secure design principles, like least privilege, complete mediation, or the least common mechanism. In MTS, we address the least privilege principle by the *compartmentalization* of the vswitches (Fig. 7.1(b)): by moving the vswitches into a dedicated vswitch VM, we can prevent an attacker from compromising the Host via the vswitch [98]. Then, we establish a secure communication channel between the tenant VMs and the vswitch VM via a trusted hardware technique, Single Root Input/Output Virtualization, or SR-IOV, a common feature implemented in most modern NICs and motherboards [11, 99]. Thus, all tenant-to-tenant and tenant-Host networking is *completely mediated* via the SR-IOV NIC. Adopting Google’s *extra security layer* design principle [5] which requires that between any untrusted and trusted component, there have to be at least two distinct security boundaries [16, 62], we introduce a second level of isolation by moving the vswitch, deployed into the vswitch VM, to the user space. Hence, at least two independent security mechanisms need to fail (user-kernel separation and VM-isolation) for the untrusted tenant code to gain access to the Host.

Interestingly, we are able to show the resultant secure vswitch design, which we call the *single vswitch VM* design, does not come at the cost of performance; just the contrary, our evaluations show that we can considerably improve throughput and latency, for a relatively small price in resources. Finally, we introduce a "hardened" `MTS` design that we call the *multiple vswitch VMs* design (Fig. 7.1(c)), whereby, in line with the principle of the least common mechanism, we further separate the vswitch by creating multiple separate vswitch VMs, one for each tenant (or based on security zones/classes). This way, we can maintain full network isolation for multiple tenants.

7.2 Threat Model

Our threat model in this chapter is a slightly nuanced threat model compared to the one described in Chapter 3 and Section 6.1.2. In particular, we assume the attacker's goal here is to either escape network virtualization by compromising the virtual switch, or to tamper with other tenant's network traffic by controlling the virtual switch [227]. She however, does not have direct control to configure the Host OS and hardware: all configuration access happens through a dedicated cloud management system.

The defender is a public cloud provider who wants to prevent the attacker from compromising virtual network isolation; in particular, *the cloud provider wants to maintain tenant-isolation even when the vswitch is compromised*. We assume that the cloud provider already supports SR-IOV NICs [11, 99, 59] and the underlying virtualization infrastructure is trusted, including the hypervisor layer, NICs, firmware, drivers and so on.

7.3 Design Principles and Security Levels

Our `MTS` design is based on the application of the secure system design principles, established by Saltzer et al. [191] (see also Bishop [20] and Colp et al. [36]), to the problem space of virtual switches.

7.3.1 Least Privilege vswitch

The vswitch should have the minimal privileges sufficient to complete its task, which is to process packets to and from the tenant VMs. Doing so limits the damage that can result from a system compromise or mis-configuration. Current best-practice is, however, to run the vswitch co-located with the Host OS and with elevated privileges; prior work has shown the types and severity of attacks that can happen when this principle fails [227]. A well-known means to the principle of least privilege is *compartmentalization*: execute the vswitch in an isolated environment with limited privileges and minimal access to the rest of the system. In the next section, we will show how `MTS` implements compartmentalization by committing the vswitches into one or more dedicated vswitch VMs.

7.3.2 Complete Mediation of Tenant-to-Tenant and Tenant-to-Host Networking

This principle requires that the network communication between the untrusted tenants and the trusted Host is completely mediated by a trusted intermediary to prevent undesired communication. This principle, when systematically applied, may go a long way towards reducing the vswitch attack surface. By *channeling all network communication* between untrusted and trusted components *via a trusted intermediary* (a so called reference monitor), the communication can be validated, monitored and logged based on security policies. In the next section, we show how complete mediation is realized in `MTS` using a secure SR-IOV channel between the tenant VMs, vswitches and Host.

7.3.3 Extra Security Boundary Between the Tenant and the Host

This security principle, widely deployed at Google [5], requires that between any untrusted and trusted component there has to be at least two distinct security boundaries, so at least two independent security mechanisms need to fail for the untrusted component to gain access to the sensitive component [16]. We establish this extra layer of security in `MTS` by *moving the vswitch to user space*. This also contributes to implementing the "least privilege" principle: the user-space vswitch can drop administrator privileges after initialization.

7.3.4 Least Common Mechanisms

This principle addresses the amount of infrastructure shared between tenants; applied to the context of vswitches this principle requires that the network resources (code paths, configuration, caches) common to more than one tenant should be minimized. Indeed, every shared resource may become a covert channel [20]. Correspondingly, *decomposing the vswitches themselves into multiple compartments* could lead to hardened vswitch designs.

7.3.5 Security Levels

From these principles, we can obtain different levels of security:

- **Baseline:** The per-tenant logical datapaths are consolidated into a single physical or software vswitch that is co-located with the Host OS.

- **Level-1:** Placing the vswitch in a dedicated compartment provides a first level of security by protecting from malicious tenants to compromise the Host OS via the vswitch ("single vswitch VM" in Fig. 7.1b).

- **Level-2:** Splitting the vswitches into multiple compartments (based on security zones or on a per-tenant basis) adds another level of security, by isolating tenants' vswitches from each other ("multiple vswitch VMs" in Fig. 7.1c).

- **Level-3:** Moving the vswitches into user space, combined with Baseline or Level-1 or -2, reduces the impact of a compromise and further reduces the attack surface.

7.4 The MTS Architecture

We designed MTS to meet the aforementioned secure design principles. We first provide an overview and then present our architecture in details.

7.4.1 Overview

Compartmentalization. There are many ways in which isolated vswitch compartments can be implemented: full-blown VMs, OS-level sandboxes (jails, zones, containers, plain-old user-space processes [62], and exotic combinations of these [221, 220]), hardware-supported enclaves (Intel's SGX) [160, 10], or even safe programming language compilers (Rust), runtimes (eBPF), and instruction sets (Intel MPX). For flexibility, simplicity, and ease of deployment, MTS *relies on conventional VMs as the main unit of compartmentalization*.

VMs provide a rather strong level of isolation and are widely supported in hardware, software, and management systems. This in no way means that VM-based vswitches are mandatory for MTS, just that this approach offers the highest flexibility for prototyping. For simplicity, Fig. 7.2 depicts two vswitch compartments (Red and Blue solid boxes) running independent vswitches in their isolated VMs. The multiple compartments further reduce the common mechanisms between the vswitch and the connected tenants, achieving security Level-2. Security Level-1, although not depicted, would involve only a single vswitch VM.

Complete Mediation. To mediate all interactions between untrusted tenant code and the Host OS through the vswitch, we need a secure and high-performance communication medium between the corresponding compartments/VMs. In MTS *we use Single Root IO Virtualization (SR-IOV) to inter-connect the vswitch compartments* (see Figure 7.2).

SR-IOV is a PCI-SIG standard to make a single PCIe device, e.g., a NIC, appear as multiple PCIe devices that can then be attached to multiple VMs. An SR-IOV device has one or more physical functions (PFs) and one or more virtual functions (VFs), where the PFs are typically attached to the Host and the VFs to the VMs. Only the Host OS driver has privileges to configure the PFs and VFs. NIC driver in the VMs in turn have restricted access to VF configuration. Only via the Host, VFs and PFs can be configured with unique MAC addresses and Vlan tags. Network communication between the PFs and VFs occurs via an L2 switch implemented in the NIC based on the IEEE Virtual Ethernet Bridging standard [109]. This enables Ethernet communication not only from and to the respective VMs (vswitch and tenants) based on the destination VF's MAC address but also to the external networks.

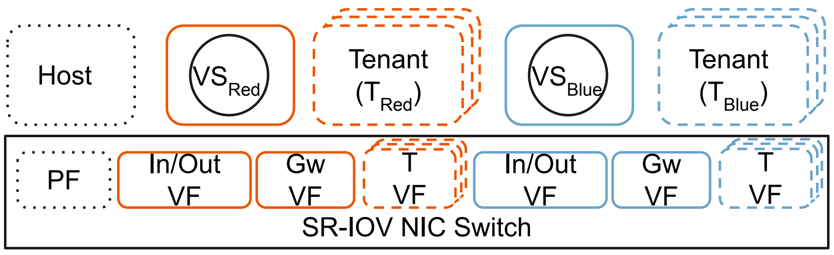

Fig. 7.2: High-level overview of `MTS` in security Level-2. The Red and Blue vswitch compartments (VMs) are allocated dedicated virtual functions (VFs) to communicate with external networks using the *In/Out VF*, their respective tenants using the *Gw VF* and *T VF*. Communication between the vswitches, tenants and the Host physical function (PF) are mediated via the SR-IOV NIC switch.

Sharing the NIC SR-IOV VF driver and the Layer 2 network virtualization mechanism implemented by the SR-IOV NIC is considerably simpler than including the NIC driver and the entire network virtualization stack (Layer 2-7) in the TCB. Tenants already share SR-IOV NIC drivers in public clouds [11, 99, 13]. Virtual networks can be built as we will see next, as per-tenant user-space applications implementing Layer 3-7 of the virtual networking stack.

Thanks to the use of SR-IOV in `MTS`, packets to and from tenant VMs completely bypass the Host OS; instead, all potentially malicious traffic is channeled through the trusted hardware medium (SR-IOV NIC) to the vswitch VM(s). Furthermore, using SR-IOV reduces CPU overhead and improves performance (see Section 7.5). Finally, SR-IOV provides an attractive upgrade path towards fully offloaded, smart-NIC based virtual networking: chip [115] and OS vendors [248, 213] have been supporting SR-IOV for many years now at a reasonable price, major cloud providers already have SR-IOV NICs deployed in their data centers [11, 99, 13], and, perhaps most importantly, this design choice liberates us from having to re-implement low-level and complex network components [98]: we can simply use any desired vswitch, deploy it into a vswitch VM, configure and attach VFs to route tenants' traffic through the vswitch, and start processing packets right away.

User-Space Packet Processing. As discussed previously, we may choose to deploy the vswitches into the vswitch VM user-space to establish an extra security boundary between the tenant and the Host OS (Level-3 design). Thanks to the advances in kernel bypass techniques, several high-performance and feature-rich user-space packet processing frameworks are available today, such as Netmap [181], FD.IO [219], or Intel's DPDK [85]. Our current design of `MTS` *leverages OvS with the DPDK datapath for implementing the vswitches* [184]. DPDK is widely supported, it has already been integrated with popular virtual switch products, and extensive operational experience is available regarding the expected performance and resource footprint [120]. Note, however, that using DPDK and OvS is not mandatory in `MTS`; in fact, thanks to the flexibility provided by our VM-compartments and SR-IOV, we can deploy essentially any user-space vswitch to support `MTS`.

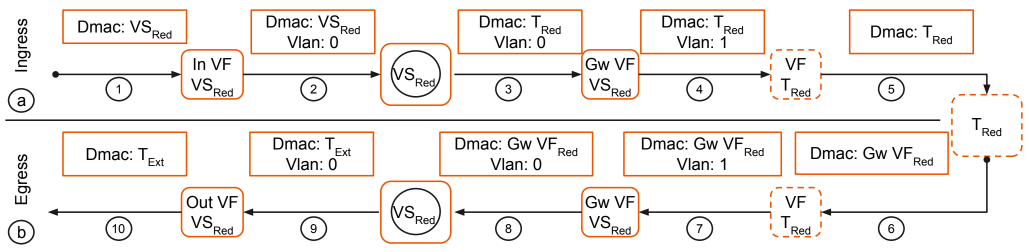

Fig. 7.3: A step-by-step illustration of how packets enter and leave the Red tenant from Figure 7.2 in MTS. ⓐ shows how ingress packets reach Tenant$_{Red}$. ⓑ shows how Tenant$_{Red}$ packets reach an external system Tenant$_{Ext}$.

7.4.2 Detailed Architecture

For the below discussion, we consider the operation of MTS for one vswitch compartment and its corresponding tenant VMs from the Level-2 design shown in Fig. 7.2. The case when only a single compartment (Level-1) is used is similar in vein: the flow table entries installed into the vswitch and the VFs attached to the vswitch compartment need to be modified somewhat; we call out the differences in-line.

Connectivity. Each vswitch VM is allocated at least one VF (In/Out VF) for external (inter-server) connectivity and another as a gateway (Gw VF) for vswitch-VM-to-tenant-VM connectivity as shown in Fig. 7.2. For Level-1, all the tenant gateways VFs are allocated to the single vswitch VM. Furthermore, isolation between the external and the tenant network (tenant VF shown as T VF in the Figure) is enforced at the NIC-level by configuring the Gw VF and the tenant VFs with a Vlan tag specific to the tenant. Different Vlan tags are used to further isolate the multiple vswitch compartments and their resp. tenants on that server.

The packets between VFs/PFs in the NIC are forwarded based on the destination MAC address and securely isolated using Vlan tags (the same security model as provided by enterprise Ethernet switches). For all packets to and from the tenant VMs to pass through the vswitch-VM, the destination MAC address of each packet entering and leaving the NIC needs to be accurately set, otherwise packets will not reach the correct destination. This can be addressed by introducing minor configuration changes to the normal operation of the tenant and the vswitches, detailed below.

Ingress Chain. Fig. 7.3ⓐ illustrates the process by which packets from an external network reach the tenant VMs. In step ① a packet enters the server through the NIC fabric port having the Red vswitch's In/Out VF MAC address as the destination MAC address (Dmac). The NIC switch will deliver the packet to the vswitch VM untagged (Vlan 0) in ②. The Red vswitch then uses the destination IP address in the packet to identify the correct tenant VM to send the packet to, changes the destination MAC address to that of the Red tenant's VF (VF T_{Red}), and emits the packet to the Gw VF in the NIC in ③. This ensures accurate packet delivery to and from tenant VMs and the complete isolation of the tenant-vswitch traffic from other traffic instances. In ④ and ⑤, the NIC tags the packet with the Red tenant's specific Vlan tag (Vlan 1 in the figure), uses the built-in switch functionality to make a lookup in the MAC learning table for the Vlan, pops the Vlan tag and finally forwards the packet to the Red tenant's VM. The NIC forwarding process is completely transparent to the vswitch and tenant VMs, the only downside is the extra round-trip to the NIC. Later we show that this round-trip introduces negligible latency overhead.

Egress Chain. The reverse direction shown in Fig. 7.3ⓑ, sending a packet from the tenant VM through the vswitch to the external network goes in similar vein. In ⑥ the Red tenant VM T_{Red} sends a packet through its VF (T_{Red}) with the destination MAC address set to the MAC address of the Red tenant's Gw VF; in the next subsection we describe two ways to achieve this. In ⑦ the NIC switch tags the packet (Vlan 1), looks-up the destination MAC address which results in sending the packet to the Gw VF. At the gateway VF ⑧, the NIC switch pops the Vlan tag and delivers the packet to the Red vswitch VM. The vswitch receives the packet, looks up the destination IP address, rewrites the MAC address to the actual (external) gateway's MAC address, and then sends the packet out to the In/Out VF in ⑨. Finally in ⑩, the NIC will in turn send the packet out the physical fabric port.

Communication between the two VMs of a single tenant inside the server goes similarly, with the additional complexity that packets now take *two* extra round-trips to the NIC: once on the way from the sender VM to vswitch, and once on the way from the vswitch to the destination VM. Again, our evaluations in the next sections will show that the induced latency overhead for such a traffic scenario is low.

System Support. Next, we detail the modifications the cloud operator needs to apply to the conventional vswitch setup to support MTS. The primary requirement is to modify the centralized controllers to appropriately configure tenant specific VFs with Vlan tags and MAC addresses, and insert correct flow rules to ensure the vswitch-tenant connectivity. Second, advanced multi-tenant cloud systems rely on tunneling protocols to support L2 virtual networks. This is also supported by MTS, by modifying the flow tables to pop/insert the appropriate headers whenever packets need to be decapsulated/encapsulated. Note that after decapsulation the tunnel id can be used in conjunction with the destination IP address to identify the appropriate tenant VM. Third, the ARP entry for the default gateway must be appropriately set in each tenant VM so that packets from the tenant VM go to the vswitch VM. To this end, the tenant VMs can be configured with a static ARP entry pointing to the appropriate Gw VF, or using the centralized controller and vswitch as a proxy-ARP/ARP-responder [139]. Finally, to prevent malicious tenants from launching an attack on the system, the cloud operator needs to deploy security filters in the NIC. In particular, source MAC address spoofing prevention must be enabled on all tenant VMs' VFs. Furthermore, flow-based wildcard filters can also be applied in the NIC for additional security, e.g., to drop packets not destined to the vswitch compartment, to prevent the Host from receiving packets from the tenant VMs, etc. Our MTS implementation, described in Section 7.5, takes care of removing the manual management burden in applying the above steps.

Resource Allocation. Additional levels of security usually come with increased resource requirement, needed to run the security/isolation infrastructure. Below, we describe two resource sharing strategies and how the VFs are allocated to the vswitch compartments. However, due to the sheer quantity and diversity in cloud setups, we restrict the discussion to plain compute and memory resources and the number of SR-IOV VFs for the different MTS security levels.

We consider two modes for compute and memory resources. A *shared* mode where tenants' vswitches share a single physical CPU core, while in the *isolated* mode each tenant's vswitch is pinned to a different core. However, we assume that each vswitch compartment gets an equal share of main memory (RAM) and this is inexpensive compared to physical CPU cores. Given the demanding compute and memory workloads in the cloud, allocating resources for vswitching is not uncommon among cloud operators [59, 44]. Note that the *shared* and *isolated* resource allocations are merely two ends of the resource allocation spectrum, different sets of vswitch VMs could be allocated resources differently, e.g., based on application or customer requirements. In the next section we will see that the resource requirement for multiple vswitch VM compartments, i.e., Level-2 alone, is not resource prohibitive in the *shared* mode, however, Level-2 and Level-3 can be.

Regarding the number of SR-IOV VFs needed, the current standard allows each SR-IOV device to have up to 64 VFs per PF. For Level-1, the total number of VFs for T tenants (see Equation 7.1) is given by the sum of 1) E, the number of VFs allocated for external connectivity (In/Out VF); 2) g_i, the total number of tenant-specific gateway VFs; and 3) t_i, tenant-specific VM VFs hosted on the server. In a basic Level-1 setup hosting 1 tenant, with 1 In/Out VF and 1 gateway VF and 1 VF for the tenant VM, the total VFs is 3. Similarly for 4 tenants, the total VFs is 9. For Level-2, the total number of VFs for T tenants (see Equation 7.2) is given by the sum of 1) e_i, the tenant-specific VFs allocated for external connectivity; 2) g_i, the tenant-specific gateway VFs; and 3) t_i, tenant-specific VM VFs hosted on the server. For a basic Level-2 setup hosting 2 tenants, with 1 In/Out VF, 1 gateway VF per tenant vswitch and 1 VF for each tenant VM, the total VFs is 6. Similarly for 4 tenants, the total VFs is 12. Level-3 is the same as Level-1 or Level-2.

$$E + \sum_{i=1}^{T}(g_i + t_i) \tag{7.1}$$

$$\sum_{i=1}^{T}(e_i + g_i + t_i) \tag{7.2}$$

7.5 Evaluating Tradeoffs

We designed a set of experiments to empirically evaluate the security-performance-resource tradeoff of `MTS`. To this end, we measure `MTS`'s performance for different security levels under different resource allocation strategies, in canonical cloud traffic scenarios [52]. The focus is on throughput and latency performance metrics, and physical cores and memory for resources. In particular, the experiments serve to verify our expectation that our design does not introduce a considerable overhead in performance. However, we do expect the amount of resources consumed to increase; our aim is to quantify this increase in different realistic setups.

Prototype Framework. We took a programmatic approach to our design and evaluation, hence, we developed a set of primitives that can be composed to configure `MTS` to conduct all the experiments described in this chapter. Hence, as a first step we do not consider complex cloud management systems (CMS) such as OpenStack; this way we can conduct self-contained experiments without the possible interference cause by a CMS. Our framework is written in Python and currently supports OvS and ovs-DPDK as the base virtual switch, Mellanox NIC, and the `libvirt` virtualization framework. Our framework and data are available on-line at the following URL:

`https://www.github.com/securedataplane/mts`

Methodology. We chose a set of standard cloud traffic scenarios (see Fig. 7.4) and a fixed number of tenants (4). For each of those scenarios, we allocated the necessary resources (Sec. 7.4) and then configured the vswitch either in its default configuration (Baseline) or one of the three security levels (Sec. 7.3). The system was then connected in a measurement setup to measure the one-way forwarding performance. Important details on the topology, resources, security levels and the hardware and software used are described next.

Traffic Scenarios. The three scenarios evaluated are shown in Fig. 7.4. *Physical-to-physical (p2p):* Packets are forwarded by the vswitch from the ingress physical port to the egress. This is meant to shed light on basic vswitch forwarding performance. *Physical-to-virtual (p2v):* Packets are forwarded by the vswitch from one physical port to a tenant VM, and then back from the tenant VM to the other physical port. Compared to the p2p, this will show the overhead to forward to and from the tenant VM. *Virtual-to-virtual (v2v):* Similar to the p2v, however, when the packets return from the tenant to the vswitch, the vswitch sends the packet to another tenant which then sends it back to the vswitch and then out the egress port. This scenario emulates service chains in network function virtualization. Since the path length increases from p2p to p2v to v2v, we expect the latency to increase and the throughput to decrease when going from p2p to p2v to v2v.

Resources. We allocated compute resources in the following two ways. *Shared:* All vswitch compartments share 1 physical CPU core and their associated cache levels. *Isolated:* Each vswitch compartment is allocated 1 physical CPU core and their associated cache levels. In case of the Baseline, we allocated cores proportional to the number of vswitch compartments, e.g., 2 cores to compare with 2 vswitch VMs. For main memory, each VM (vswitch and tenant) was allocated 4 GB of which 1 GB is reserved as one 1GB Huge page. Similarly, for the Baseline, a proportional amount of Huge pages was allocated. When using `MTS`, each vswitch VM was allocated 2 In/Out VFs (1 per physical port), and 2 appropriately Vlan tagged Gw VFs per tenant (1 per physical port). When DPDK was used in Level-3: one physical core needs to allocated for each ovs-DPDK compartment (including the Baseline), hence, only the isolated mode was used; all In/Out, gateway and tenant ports connected to OvS were assigned DPDK ports (in the case of the Baseline, the tenant port type was the `dpdkvhostuserclient`). All the tenant VMs got two physical cores and two VFs, 1 per port (these are VMs the tenant would use to run her application) so that the forwarding app (`l2fwd`) could run without being a bottleneck.

Security Levels and Tenants. For each resource allocation mode, we configured our setup either in Baseline or one of the three `MTS` security levels (Section 7.3). In the Baseline and Level-1, there were 4 tenant VMs connected to the vswitch. For Level-2, we configured 2

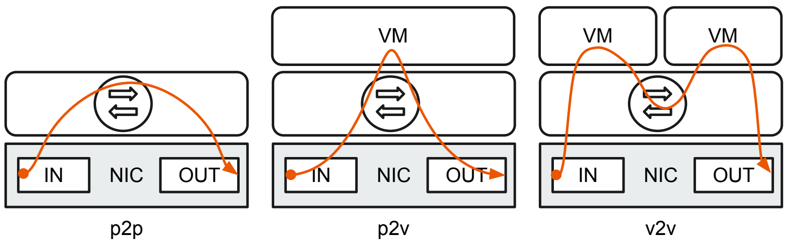

Fig. 7.4: Three canonical traffic scenarios evaluated: p2p, p2v and v2v

vswitch VMs and each vswitch had 2 tenant VMs, and then we configured 4 vswitch VMs where each vswitch VM had 1 tenant VM. We repeated Level-3 with Baseline, Level-1 and the two Level-2 configurations.

Setup. To accurately measure the one-way forwarding performance (throughput and latency), we used two servers connected to each other via 10G short range optical links. The DUT server was an Intel(R) Xeon(R) CPU E5-2683 v4 @ 2.10GHz with 64 GB of RAM with the IOMMU enabled but hyper-threading and energy efficiency disabled, and a 2x10G Mellanox ConnectX4-LN NIC with adaptive interrupt moderation and irq balancing disabled. The other server was the packet/load generator, sink and monitor, with an Endace Dag 10X4-P card (which gives us accurate and precise hardware timestamps) [53]. The link between the LG and DUT, and DUT and sink were monitored via a passive optical network tap connected to the Dag card. Each receive stream of the Dag card was allocated 4 GB to receive packets. The Host, vswitch VM and tenant VMs used the Linux kernel 4.4.0-116-generic, Mellanox OFED linux driver 4.3-1.0.1.0, OvS-2.9.0 and DPDK 17.11. Libvirt 1.3.1 was used with QEMU 2.5.0. In the tenant VMs, we adapted the DPDK-17.11 `l2fwd` app to rewrite the correct destination MAC address when using `MTS`, and used the default `l2fwd` drain-interval (100 microseconds) and burst size (32) parameters. For the Baseline, we used the default linux bridge in the tenant VMs as using DPDK in the tenant without being backed by QEMU and OvS (e.g., `dpdkvhostuserclient`) is not a recommended configuration [78]. For network performance measurements, we used Endace dag-5.6.0 software tools (dagflood, dagbits, and dagsnap).

7.5.1 Throughput

Our first performance tradeoff is evaluating the forwarding throughput. This will shed light on the packets per second processing performance of `MTS` compared to the Baseline. We measure the aggregate throughput with a constant stream of 64 B packets replayed at line rate (14 Mpps) by the LG and collected at the sink. Since we fixed the number of tenants to 4, the stream of packets comprises 4 flows, each to a respective tenant VM identified by the destination MAC and IP address. At the monitor we collect the packets forwarded to report the aggregate throughput. Each experimental run lasts for 110 seconds and measurements are made from the 10-100 second marks.

Results. The throughput measurement data for the shared mode is shown in Fig. 7.5(a). In Fig. 7.5(d) we can see the data for the isolated mode and in Fig. 7.5(g) the data for Level-3 in the isolated mode is shown. From Figures 7.5(a) and (d) we can see that nearly

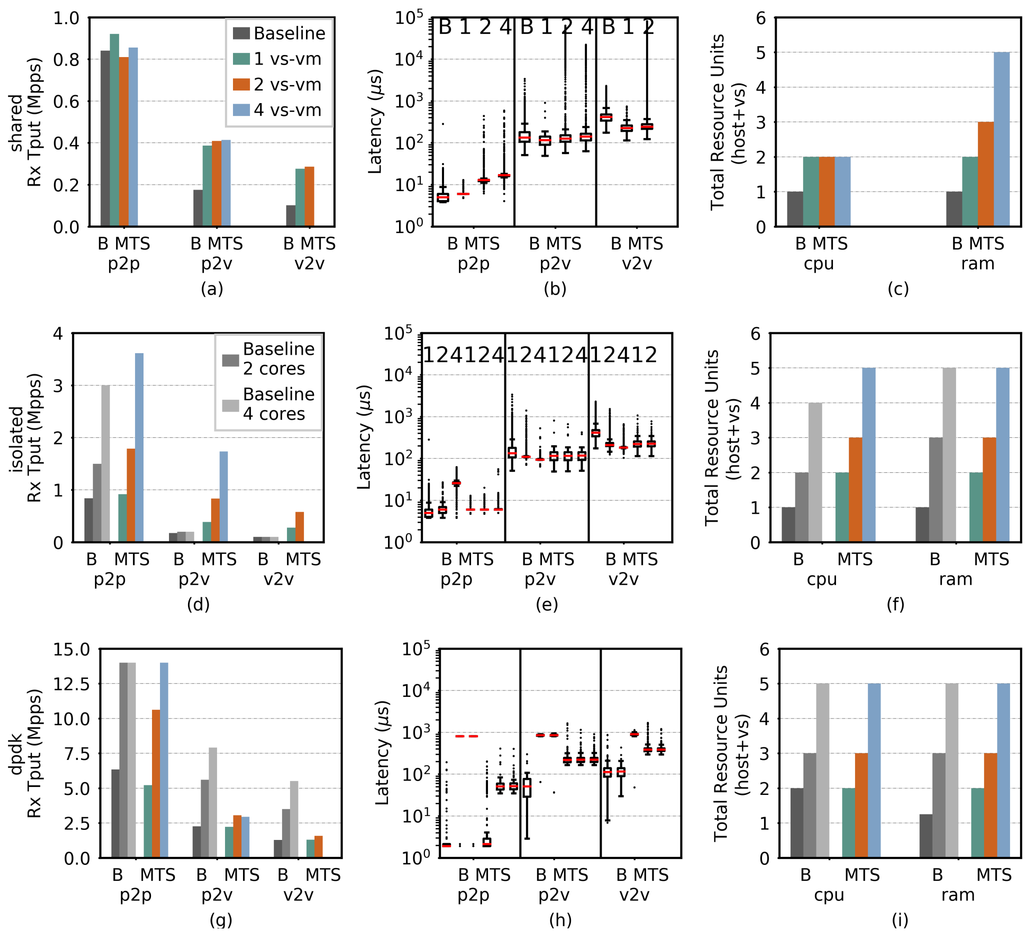

Fig. 7.5: The security, throughput, latency and resource tradeoff comparison of `MTS`. The rows indicate the resource mode. The columns are ordered as throughput, latency and resources. The security levels used are shown in the legend. Note the bottom row is for security Level-3 in the isolated resource mode combined with other security levels.

always *`MTS` had either the same or higher aggregate throughput than the Baseline*. The improvement in throughput is most obvious in the p2v and v2v topologies as vswitch-to-tenant communication is via the PCIe bus and NIC switch, which turns out to be faster than Baseline's memory bus and software approach. Sharing the physical core for multiple compartments (Fig. 7.5(a)) in the p2v and v2v scenarios can offer 4x isolation (Level-2 with 4 compartments) and a 2x increase in throughput (nearly .4 Mpps and .2 Mpps) compared to the Baseline (nearly .2 Mpps and .1 Mpps).

Fig. 7.5(d) is noteworthy as multiple cores for vswitch VMs and the Baseline functions as a load-balancer when isolating the CPU cores. In the p2p scenario, the aggregate throughput increases roughly from 1 Mpps to 2 Mpps to 4 Mpps as the number of cores increase. We observe that `MTS` is slightly more than the Baseline in the p2p, however, in the p2v and v2v scenarios `MTS` offers higher aggregate throughput. As expected, using DPDK can offer an order of magnitude better throughput (Fig. 7.5(g)). In the p2p topology, we were able to nearly reach line rate (14.4 Mpps) with four DPDK compartments as the packets were load-balanced across the multiple vswitch VMs, while the Baseline was able to saturate the link with 2 cores. With `MTS`, the throughput saturates (at around 2.3 Mpps) in the p2v and v2v topologies because several ports are polled using a single core and packets have to bounce off the NIC twice as much compared to the Baseline where we observe nearly twice the throughput for 2 and 4 cores. Nevertheless, we can see a slight increase in the throughput of `MTS` as the vswitch VMs increase, because the number of ports per vswitch VM decreases as the number of vswitch VMs increase. Due to the limited physical cores on the DUT, we could not evaluate 4 vswitch VMs in the v2v topology as it required more cores and RAM than available (as we did not want the tenant workload, e.g., `l2fwd`, to be the bottleneck).

Key Findings. The key result here is that `MTS` offers increasing levels of security with comparable, if not increasing levels of throughput in the shared and isolated resource modes, however, the Baseline's throughput with user-space packet processing (DPDK) is better than `MTS`.

7.5.2 Latency

The second performance tradeoff we evaluated was the forwarding latency, in particular, we studied the impact of packet size on forwarding. We selected 64B (minimum IPv4 UDP packet size), 512B (average packet), 1500B (maximum MTU) packets and 2048B packets (small jumbo frame). As in the throughput experiments, we used 4 flows, one to each tenant. For each experimental run, we continuously sent packets from the LG to the sink via the DUT at 10 kpps for 30 seconds. Note that is the aggregate throughput sent to the NIC and not to the vswitch VM. To eliminate possible warm-up effects, we only evaluated the packets from the 10-20 second mark.

Results. For brevity the latency distribution only for 64 B packets is reported here. Fig. 7.5(b) shows the data for the shared mode, while Fig. 7.5(e) is for the isolated mode. Level-3 latency data is shown in Fig. 7.5(h). Although the p2p scenarios shows that `MTS` increases the latency (Fig. 7.5(b), (e) and (h)), the p2v and v2v scenarios show that `MTS` is slightly faster than the Baseline. This is for two reasons. First, packets between the vswitch and the tenant VMs pass through the SR-IOV NIC (PCIe bus) rather than a software only vswitch (memory bus). Second, when using the Baseline the tenant uses the Linux bridge. The exception to this can be seen with user-space packet processing (Fig. 7.5(h)), where the Baseline with a single core for dpdk (2 in total) is always faster than `MTS`. As mentioned in Section. 7.5.1, due to resource limitations we could not evaluate the 4 vswitch VMs in v2v.

The variance in latency increases as more compartments share the same physical core (Fig. 7.5(b)). Isolating the vswitch VM cores leads to more predictable latency as seen in Fig. 7.5(e). When using DPDK (Fig. 7.5(h)) we make two observations: i) `MTS` takes longer to forward packets than without using DPDK; ii) the latency for Baseline with 2 and 4 cores for dpdk (3 and 5 in total) is unexpectedly high (around 1 ms). Regarding the former, we conclude that `MTS` with OvS and DPDK requires further tuning as we used the default OvS-DPDK parameters for the drain interval, batch size and huge pages: There is an inherent tradeoff between high throughput and average per-packet latency when using a shared memory model where a core is constantly polling [49]. For the latter, we observe that the throughput of 10 kpps is too low to drain the multiple queues on the DPDK ports.

At 100 kpps and 1 Mpps, we measured an approximately 2 microsecond latency for the p2p scenario.

Key Findings. We observe that for the shared mode, and 4x compartmentalization (Level-2), the latency is comparable to the Baseline (p2v) with a lot of variance whereas when isolated the latency is more predictable.

7.5.3 Resources

In Fig. 7.5(c), (f) and (i) we see the total CPU and memory consumption for Baseline and MTS. Note that across all the figures, one core and at least one Huge page is always dedicated for the Host OS. In the case of the (single core) Baseline, the vswitch (OvS) runs in the Host OS and hence shares the Host's core and ram. However, for the single vswitch VM in the shared, isolated and DPDK modes, the Host OS consumes one core and the vswitch VM consumes another core making the total CPU cores two. Similarly, the 2 and 4 vswitch VMs in the shared mode, also consume the same number of cores as the single vswitch VM but a linear increase in ram. In the isolated mode, MTS consumes only one extra physical core relative to the Baseline, and in DPDK, MTS and Baseline consume equal number of cores. With respect to the memory consumption, we note that MTS's and Baseline's memory consumption in the isolated and DPDK modes are the same.

Hence, we conclude that for one extra physical core, MTS offers multiple compartments, making the shared resource allocation economically attractive. The resource cost goes up when user-space packet processing is introduced or isolating cores, making it relatively expensive for multiple vswitch VMs.

Key Findings. (i) High levels (2x/4x) of virtual network isolation per server can be achieved with an increase in aggregate throughput (2x) in the shared mode; (ii) for applications that require low and predictable latency, vswitch compartments should use the isolated mode; (iii) although user-space packet processing using DPDK offers high throughput, it is expensive (physical CPU and energy costs).

7.6 Workload-based Evaluation

We also conducted experiments with real workloads, to gain insights on how cloud applications such as web servers and key-value stores will perform as tenant applications are the end hosts of the virtual networks.

Methodology. For simplicity we focus our workload-based evaluation only on TCP applications as our previous measurements dealt with UDP. In general, we use a similar methodology to the one described in Section 7.5. For all the TCP-based measurements, we configured the tenant VMs to run the respective TCP server and from the client (LG) we benchmark the server to measure the throughput and/or response time. The topologies, resources and setup used to make these measurements are slightly nuanced which we highlight next.

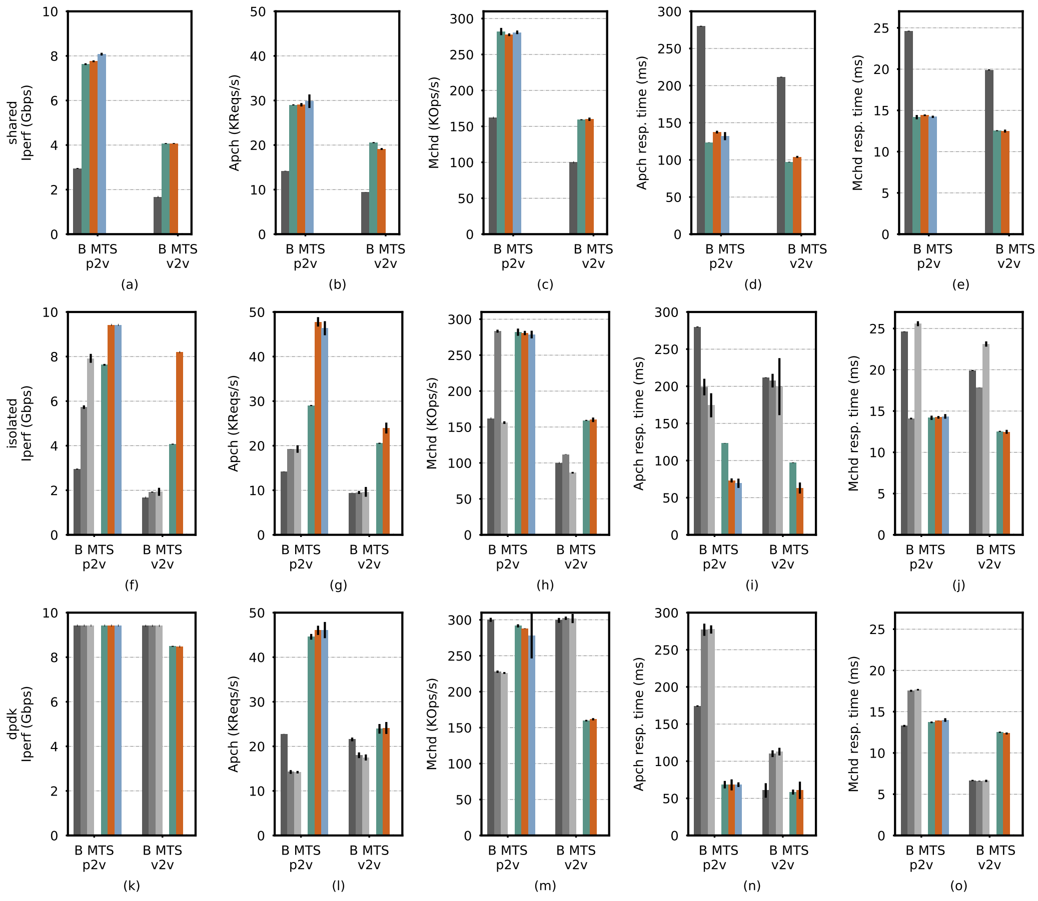

Fig. 7.6: Iperf throughput, Apache and Memcached throughput and latency (shown in the columns) comparison of MTS. The rows indicate the resource mode where the bottom row is for security Level-3 in the isolated resource mode combined with the other security levels. The legend is the same as in Figure 7.5.

Traffic Scenarios. Only the p2v and v2v patterns are evaluated with workloads as we want to understand the performance of applications hosted in the server.

Resources. The ingress and egress ports for all the traffic are on the same physical NIC port unlike in the previous section where the ingress and egress ports were on separated physical ports of the NIC. Hence, each tenant's vswitch VM was given 1 VF for In/Out and 1 tagged Gw VF. Each tenant VM was given 1 VF.

Setup. The applications generating the load are standard TCP, Apache and Memcached benchmarking tools respectively Iperf3 v3.0.11 [88], ApacheBench v2.3 (ab) [9] and libMemcached v1.0.15 (memslap) [130]. Instead of the Endace card we used a similar Mellanox card at the LG.

7.6.1 Workloads and Results

Iperf. To compare the maximum achievable TCP throughput, we ran Iperf clients for 100 s with a single stream from the LG to the respective Iperf servers in the DUT's tenant VM. The aggregate throughput was then reported as the sum of throughput for each client-server. We collected 5 such measurements for each experimental configuration and report the mean with 95% confidence.

Web Server. To study workloads from web servers (a very common cloud application), we consider the open-source Apache web server. Using the ApacheBench tool from the LG, we benchmarked the respective tenant web servers by requesting a static 11.3 KB web page from four clients (one for each web server). Each client made up to 1,000 concurrent connections for 100 s after which we collected the throughput and latency statistics reported by ApacheBench. In the v2v scenario, we used only two client-servers as one of the tenant VMs simply forwarded packets using the DPDK `l2fwd` app. We collected 5 such repetitions to finally report the average throughput and latency for each experimental configuration with 95% confidence.

Key-Value Store. Key-value stores are also commonly used cloud applications (e.g., with with web servers). We opted for the open-source Memcached key-value store as it also has an open-source benchmarking tool libMemcached-memslap. We used the default Set/Get ratio of 90/10 for the measurements. The methodology and reporting of the measurements are the same as the web server.

Results. The data for the Iperf measurements in the shared mode is shown Fig. 7.6(a). The data for the isolated mode is shown in Fig. 7.6(f) and Fig. 7.6(k) depicts the throughput for Level-3. As seen in Section 7.5.1, here too we observe that `MTS` has a higher throughput (more than 2x in the shared mode) than the Baseline except when DPDK is used in the v2v topology. `MTS` saturated the 10G link in the p2v scenario when isolated and DPDK modes were used.

The data from the throughput measurements for the Apache web server and Memcached key-value store are first reported in the shared mode in Fig. 7.6(b) and (c) respectively. For the isolated mode they are shown in Fig. 7.6(g) and (h). Level-3 throughput is shown in Fig. 7.6(l) and (m). The three main results from the throughput measurements for Apache and Memcached are the following. `MTS` can offer nearly 1.5-2x throughput and 4x isolation (Level-2) in the shared mode. In the isolated and Level-3 modes, Apache's and Memcached's throughput saturated with `MTS`: we expected the throughput to increase as the vswitch VMs increase when the compartments have isolated cores, however, we do not observe that. This is further validated when using DPDK. Apache's and Memcached's throughput are highly sensitive when the Baseline uses multiple cores in the isolated and DPDK modes which means that using 2 or more cores requires workload specific tuning to the Host: the DPDK parameters, e.g., drain interval, and the workload VMs, e.g., allocating more cores, which may not always be necessary with `MTS`.

The data from the response time measurements for the Apache web server and Memcached key-value store are first reported in the shared mode in Fig. 7.6(d) and (e) respectively. For the isolated mode they are shown in Fig. 7.6(i) and (j). Level-3 throughput is shown in Fig. 7.6(n) and (o). Regarding the latency, we again discern that `MTS` can offer multiple levels of isolation and maintain a lower response time (approximately twice as fast) than the Baseline.

Key Findings. Our web server and key-value store benchmarks reveal that application throughput and latency of real application are improved by `MTS`. However, for user-space packet processing, the resource costs go up for a fractional benefit in throughput or latency. Hence, biting the bullet for shared resources, offers 4x isolation and approximately 1.5-2x application performance compared to the Baseline.

7.7 Scaling `MTS`

Depending on the number of cores per server, the number of tenant VMs co-located on the same server will vary. Hence, our design should also accommodate servers that host more than 4 tenants. However, as we have seen, unmodified VMs for vswitches impose a resource limitation. Therefore, to scale with the number of co-located tenants per server, a lightweight isolation mechanism is necessary in addition to the VM isolation.

7.7.1 Using Containers

A suitable choice for lightweight virtualization is *containers* for three reasons. First, they are well supported by multiple OSes, e.g., Linux, FreeBSD, Windows. Second, they offer user-space and kernel-space (network namespace) isolation allowing us to run several vswitch containers in the same VM (we can run the OvS kernel module for each container). Third, they use the resources allocated to the VM: we can run several containers within a VM without necessarily increasing the compute and memory allocated to the VM.

Implementation. We extended our framework to support containers using `docker` [131]. We considered other options as well, e.g., gVisor [220] and Kata Containers [221]. However, after looking into the projects, we concluded that gVisor was still nascent and Kata Containers has too much overhead (it runs containers in lightweight VMs). Hence, we continued to use our already supported VM approach and simply integrated docker into the workflow. Using `pipeworks` [208], we were able to attach the VFs in the VM to specific containers.

7.7.2 Evaluation

We experimented with scaling `MTS` using containers in VMs. In the following we describe how we achieved this and cast light on the packet processing throughput of such a system.

Methodology. As a first step we only evaluated the throughput of our container based system. We adapted the methodology as described in Section 7.5 for measuring the raw packet processing throughput. To understand the security-performance-resource tradeoffs of using containers, we created two sets of experiments, one comparable with using four tenants and their respective VMs, and another set to measure the scaling capabilities. For the traffic scenarios, we only used the p2p and p2v. We had to allocate slightly different resources and tenants which we describe next.

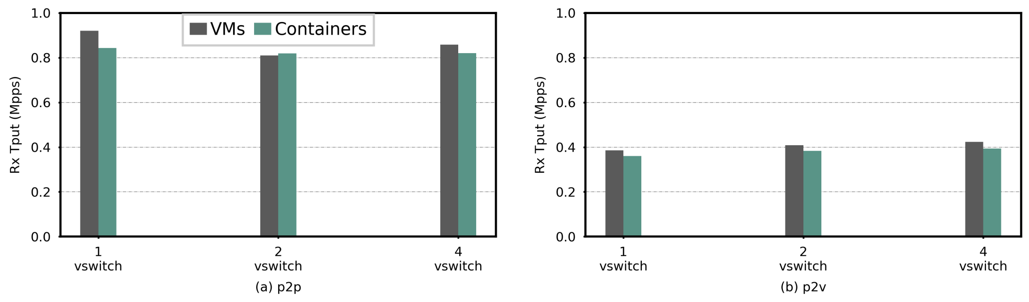

Fig. 7.7: Forwarding throughput comparison of MTS with four tenants in the p2p and p2v scenarios. The vswitch runs in VMs (gray bars) or in containers in VMs (green bars).

Resources. We allocated resources only in the *shared mode*, i.e., all the vswitch VMs shared a singly physical core, as we believe this is the most economical mode for cloud providers. The memory allocation remained the same for the vswitch VMs, when scaling and comparing with the four tenants. However when we made the scaling measurements, the tenant VM was allocated a minimum of 4 GB memory and 1 GB per tenant flow. The number of VFs allocated to each vswitch container and tenant (DPDK) application was the same as described in Section 7.5.

Security Levels and Tenants. We only measured the throughput of MTS as the Baseline by default does not support multiple containers. We first configured MTS in Level-1. Next, in Level-2 we configured MTS for comparison with four tenants, and we also configured it with upto a total of 16 containers (in a single VM and 4 containers in 4 VMs). We note that the number of tenants configured determined the maximum number of containers. In this evaluation, we used a single tenant VM with sufficient VFs to emulate several tenants, e.g., 8, 12 and 16. We did not include Level-3 for two reasons: i) we used the shared mode, hence, scaling MTS in containers with DPDK would require more resources and ii) to avoid the complexity involved with configuring and supporting DPDK in containers

Setup. We used the same setup as described in Section 7.5. We used `docker-18.09.2` and linux `cgroups` to run OvS in containers. For the single tenant VM we ran multiple instances of DPDK, e.g., if we had a total of 8 containers, we ran 8 DPDK `l2fwd` instances in the tenant VM. A bug/limitation with QEMU prevented us from attaching 37 or more VFs to a VM, hence, we could not evaluate scenarios that required more than 37 VFs per vswitch VM, e.g., 16 tenants in a single vswitch VM container.

Results. The data comparing OvS in containers in VMs with OvS only in VMs is shown in Figure 7.7. The results of the throughput of using as many supported containers in VMs with our hardware is shown in Figure 7.8 and Figure 7.9. From Fig. 7.7 we can see a minor but negligible overhead of using containers in both the p2p and p2v. However, as we scale the number of vswitch containers and tenants to a maximum of 16 tenants shown in Fig. 7.8 and 7.9, we make 3 observations. First, the impact of containers for software switching (p2p scenario) is minor. Second, in the p2v scenario, the throughput decreases as the number of vswitch containers in a given vswitch VM increases. Third, for the same number of total vswitch containers, the throughput slightly decreases as the number of vswitch VMs increase.

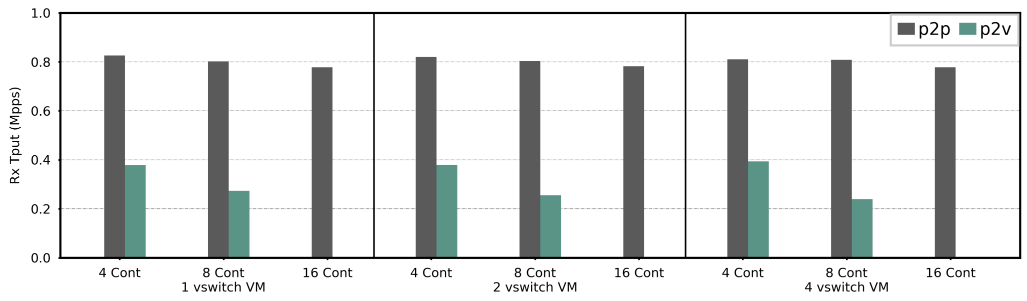

Fig. 7.8: Forwarding throughput comparison of scaling MTS using containers in multiple VMs in the p2p and p2v scenarios. Note that the number of containers is equal to the number of tenants and that they are evenly distributed across all the VMs.

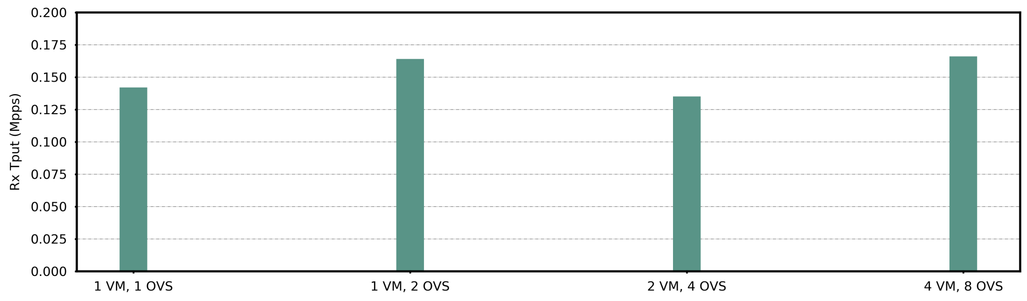

Fig. 7.9: Forwarding throughput of MTS with 16 tenants and maximum number of supported vswitch containers in VMs in the p2v scenario.

The main reasons for the drop in throughput in the p2v scenario are as follows. The kernel is unable to forward the incoming packets fast enough as the number of egress ports increase, which causes the incoming buffers to remain full resulting in packets being dropped on the ingress port of the respective vswitch container. Although not shown in the Figures, we also observed that for a fixed number of vswitch containers, as the number of tenants increase (recall this means that that the vswitch now has an extra gw VF for each new tenant), the throughput drops. Here again we notice that the bottleneck is in moving packets from the incoming buffer to the transmit buffer. Furthermore, as the number of transmit queues increase (i.e., more VFs attached to the VM and the respective vswitch containers), the kernel drops more packets.

Key Findings. On a single core we could scale MTS upto 16 containers across 4 VMs with minor loss in p2p forwarding throughput. However, in the p2v scenarios the throughput dropped by half as the number of attached VFs to containers increased. Hence, the key message here is that MTS can scale using containers, however, it requires more compute (and memory) resources to scale with the number of tenants.

7.8 Discussion

Centralized Control, Accounting and Monitoring. MTS introduces the possibility to realize multi-tenant virtual networks which can expose tenant/compartment specific interfaces to a logically centralized control/management plane. This opens up possibilities for full network virtualization, how to expose the interface, and also how to integrate MTS into existing cloud management systems in an easy and usable way. Additionally, MTS also enables the cloud provider to develop/adopt novel virtual network embedding algorithms as the vswitch can now be allocated dedicated and isolated resources. Furthermore, controllers may need to manage more device, topology and forwarding information, however, the computations (e.g., routing) should remain the same. From an accounting and billing perspective, we strongly believe that MTS is a new way to bill and monitor virtual networks at granularity more than a simple flow rule [69]: CPU, memory and I/O for virtual networking can be charged.

SR-IOV: A Double-Edged Sword. If an attacker can compromise SR-IOV, she could violate isolation and in the worst case get access to the Host OS via the PF driver. Hence, a rigorous security analysis of the SR-IOV standard, implementations and SR-IOV-NIC drivers can reduce the chance of a security vulnerability. Compartmentalizing the PF driver is a promising approach [22] to secure the driver software. However, if the attacker has compromised the SR-IOV-NIC hardware then MTS cannot guarantee isolation, as MTS assumes the NIC is trusted and operates in accordance with the cloud provider. When a vswitch VM is shared among tenants, performance isolation issues could lead to covert channels [15] or denial-of-service attacks [206, 250]. Although not yet widely supported, VM migration with SR-IOV can be introduced [134] as proposed by Xu and Davda [246]. SR-IOV NICs have limited VFs and MAC addresses which could limit the scaling properties of MTS, e.g., when using containers as compartments instead of VMs.

Evaluation Limitations. The results from our experiments are from a network and application performance perspective using a 10 Gbps NIC. For a deeper understanding of the performance improvement we obtained in this chapter using SR-IOV, further measurements are necessary, e.g., using the performance monitoring unit (PMU) to collect a breakdown of the packet processing latencies. Such an understanding is important and relevant when dealing with data center applications that require high NIC bandwidth, e.g., 40/100 Gbps.

As described by Neugebauer et al. [142], the PCIe bus can be a bottleneck for special data center applications (e.g., ML applications): A *typical* x8 PCIe 3.0 NIC (with a maximum payload size of 256 bytes and maximum read request of 4096 bytes) has an effective (usable) bi-directional bandwidth of approximately 50 Gbps. Hence, the usability of MTS with PCIe 3.0 and 8 lanes can indeed be a limitation which we did not observe in this chapter. Nevertheless, increasing the lanes to x16 is one potential workaround to double the effective bandwidth to around 100 Gbps. Furthermore, with chip vendors initiating PCIe 4.0 devices [23], the PCIe bus bandwidth will increase to support intense I/O applications.

7.9 Related Work

There has been noteworthy research and development on isolating multi-tenant virtual networks in cloud (datacenter) networks: tunneling protocols have been standardized [237, 66], multi-tenant datacenter architectures have been proposed [110], and real cloud systems have been built by many companies [59, 44]. Furthermore, considerable effort has gone into enforcing performance isolation in the cloud [113, 212, 202, 96, 173]. However, most of the previous work still co-locates the vswitch with the Host as we discussed in Section 7.1.1. Hence, here we discuss previous and existing attempts specifically addressing the security weakness of vswitches.

To the best of our knowledge, in 2012 Jin et al. [98] (see Research prototype in Table 7.1) were the first to point out the security weakness of co-locating the virtual switch with the hypervisor. However, the proposed design, while ahead of its time, (i) lacks a principled approach which this chapter proposes; (ii) has only a single vswitch VM whereas `MTS` supports multiple vswitch compartments making it more robust; (iii) is resource (compute and memory) intensive as the design used shared memory between the vswitch VM and all the tenant VMs while `MTS` uses an inexpensive interrupt-based SR-IOV network card for complete mediation of tenant-vswitch-VM and tenant-host networking; (iv) requires considerable effort, expertise and tuning to integrate into virtualization system whereas `MTS` can easily be scripted into existing cloud systems.

In 2014 Stecklina [211] followed up on this line of work and proposed sv3, a user-space switch, which can enable multi-tenant virtual switches (see sv3 in Table 7.1). sv3 adopts user-space packet processing and also supports compartmentalization, i.e., the Host can run multiple vswitches. However, it is still co-located with the Host, partially adopts the security principles outlined in this chapter, lacks support for *real* cloud virtual networking, and requires changes to QEMU. Our system on the other hand moves the vswitch out of the Host, supports production vswitches such as OvS and does not require any changes to QEMU.

Between 2016 and 2017, Panda et al. [162] and Neves et al. [143] took a language-centric approach to enforce data plane isolation for virtual networks. However, language-centric approaches require existing vswitches to be reprogrammed/annotated which reduces adoption. Hence the solution of using compartments and SR-IOV in `MTS` allows existing users to easily migrate using their existing software. Shahbaz et al. [198] reduced the attack surface of OvS by introducing support for the P4 domain specific language which reduces potentially vulnerable protocol parsing logic (see PISCES in Table 7.1.

In 2018, Pettit et al. [167] proposed to isolate virtual switch packet processing using eBPF: which is conceptually isolating potentially vulnerable parsing code in a kernel-based runtime environment. However, the design still co-locates the virtual switch and the runtime with the Host.

In early 2019 Zhuo et al. [251] (see SLIM in Table 7.1) introduced faster virtual networking for container networks in the cloud using connection-based network virtualization. Although SLIM bypasses the vswitch and is implemented in user-space with an optional module to enforce security in the kernel, it is still monolithic and co-located with the Host. By compromising SLIM, an attacker breaks into the Host's namespace and can take control of the Host. Later on, Hedayati et al. [74] introduced `Hodor`, which enables multiple (untrusted) user-space applications to share the underlying hardware, e.g., NIC, when kernel-bypass libraries such as DPDK are used. Currently, it is not possible to share the underlying hardware, e.g., NIC, when an application uses DPDK. However, even without `Hodor`, using SR-IOV does enable such sharing and furthermore Mellanox's ConnectX-4 NIC for example implements a *bifurcated* driver model that enables sharing the NIC between the user-space application and the kernel.

7.10 Conclusion

This chapter was motivated by the observation that while vswitches have been designed to enable multi-tenancy, today's vswitch designs lack strong isolation between tenant virtual networks. Accordingly, we presented a novel vswitch architecture based on 4 secure design principles which extends the benefits of multi-tenancy to the virtual switch, offering improved isolation and performance, at a modest additional resource cost. When used in the *shared* mode (only one extra core), with four vswitch compartments the forwarding throughput (in pps) is 1.5-2 times better than the Baseline. The tenant workloads (web server and key-value stores) we evaluated also receive a 1.5-2 times performance (throughput and response time) improvement with `MTS`.

We believe that `MTS` is a pragmatic solution that can enhance the security and performance of virtual networking in the cloud. In particular, `MTS` introduces a way to schedule an entire core for tenant-specific network virtualization which has three benefits: (i) application and packet processing performance is improved, (ii) this could be integrated into pricing models to appropriately charge customers on-demand and generate revenue from virtual networking for example and (iii) virtual network and Host isolation is maintained even when the vswitch is compromised.

With this we conclude Part II of this dissertation. We saw how the vswitch can be exploited by an attacker to violate isolation policies and mechanisms as well as a new network virtualization architecture namely `MTS`, that can ensure multi-tenant network isolation policies and mechanisms can be enforced for uncompromised vswitches.

8 Future Work

We believe this dissertation opens up several interesting avenues for further research which we outline in the following.

Detecting Malicious Switches. This dissertation was motivated by the possibility that the switches in the data center network can be compromised. Having seen the potential capabilities of malicious switches in a data center network in this dissertation, a follow-up research question to ask could be: How can we detect malicious switches in the network? The question is a fundamental one and may even prove to be impossible to do [63]. Nonetheless, by restricting the scope of the problem, novel detection systems can be introduced, as we have done e.g., with `Preacher` [225].

Detecting Covert Channels. To identify teleportation attacks, and similarly covert channels in production networks, detection systems and algorithms have to be devised, else they will go undetected. As we saw in Section 4.3.1, changes to the specifications may not suffice: the flow reconfiguration techniques can modulate legitimate configuration messages to covertly communicate. Furthermore we expect that even if detection systems are designed and implemented, they will need to be tuned to the operator's network and allocated sufficient resources [50, 51]. Hence, identifying suitable values and resources for various networks is also another research path.

Programmable Data Planes. With programmable data planes gaining momentum [126], researchers are revisiting functions that can be offloaded to the data plane [93, 199, 54, 122]. As we have seen in this dissertation, the data plane is shared by several users that need to be isolated from each other. Hence, the functions offloaded to the data plane need to communicate and process data in isolation, e.g., in secure enclaves/environments. Investigating systems that can offer such offload functionality *and* security can prove useful for confidential/sensitive information processing. SmartNICs that run multicore ARM processors with Linux are a potential candidate that can offer isolation via VMs to run tenant workloads in the NIC and reduce CPU usage as well as latency. However, to the best of our knowledge these interfaces are still nascent [54] and further research evaluation is necessary.

Virtual Switches. Designing and evaluating an alternative secure vswitch design is also a promising direction. For example, following the lines of Jin et al. [98], that uses shared memory instead of SR-IOV for multiple vswitches using KVM/QEMU could be an alternative approach. A clean-slate approach could also be adopted wherein safe programming languages are used for the vswitch. The concept of compartmentalization can be extended to the software architecture of the vswitch as well.

Pricing and Isolation. Cloud providers are attempting to take on more responsibility by managing and scaling resource requirements, thereby allowing developers not only to focus on writing code rather than set-up infrastructure but also to offer competitive pricing models, i.e., only pay for running your code at the granularity of seconds [61]. In such a model [100], the isolation mechanisms and policies are oblivious to the tenants, making it critical that tenant isolation is correctly enforced, and tenants only pay for what they use. Initial work on performance isolation has already been carried out by Wang et al. [242, 106], and we believe further research is necessary if the market decides to adopt the “serverless” model.

Security Metrics. Finally, developing accurate and quantitave metrics to measure network isolation is an interesting field of work. The typical metric used is static lines of code (SLOC), however, this is not meaningful when comparing designs and systems with no access to the source code.

9 Conclusion

This dissertation was motivated by the thesis statement: "Network-wide security policies or mechanisms in multi-tenant data center networks can be circumvented by malicious data plane systems. Hence, appropriate security measures need to be taken to tolerate compromised data planes in the network". Our argument was divided into two parts.

In Part I of this dissertation, we described how the design, specification and implementations of logically centralized control planes in data center networks can be exploited by malicious switches and end hosts to covertly communicate with each other (at high and low rates) using teleportation. We presented four teleportation techniques that malicious switches can use to bypass data plane isolation mechanisms meant to enforce (network-wide) security policies, e.g., firewalls. We established three reasons for the existence of teleportation: i) malicious switches share the same logical controller ii) lack of authentication and authorization in the specification of OpenFlow and controller implementations and iii) introduction of automation and programmability (e.g., intent networking) into the operation of the network. We reinforced reasons i) and ii) by modeling the OpenFlow handshake to design and implement a covert timing channel that can be used by malicious switches to exfiltrate private data. The practical significance of this work has been acknowledged by the community and hence resulted in fixes to SDN controllers (ONOS, RYU and ODL) and specifications (OpenFlow and P4Runtime).

In Part II of this dissertation, our security analysis uncovered multiple design flaws in multi-tenant network virtualization systems that use virtual switches as their isolation enforcement mechanism. The flaws enable an attacker to compromise an entire data center network (virtual switches, centralized controller and servers) in a few minutes. We attributed the findings to the lack of appropriate threat models and design decisions, e.g., co-locating the vswitch with the hypervisor and increasing complexity in the vswitch. To justify our argument, we fuzzed OvS (a widely used vswitch), discovered multiple vulnerabilities and demonstrated a data center wide compromise using OpenStack (a widely used CMS). Our deeper analysis of the design of existing virtual switches shows that the state-of-the-art violates basic secure design principles. To address the identified weaknesses and enable easy adoption in the cloud, we introduced a new network virtualization system namely `MTS`, that is based on four secure design principles. `MTS` prevents the Host from being compromised via the virtual switch, maintains multi-tenant network isolation and reduces the chance of a data center wide compromise occurring via the virtual switch. We obtained promising results from the scalable prototype we developed and evaluated using real-world workloads.

As cloud operators and operations expand, multi-tenant data center networks will also increase [124]. Hence, the work described in this dissertation should motivate cloud operators and adopters to pay more attention to potential design and programming flaws that can lead to security breaches in unintended ways, e.g., as we explained with our work on teleportation and vswitches. The ease with which we could compromise OvS (used in Google's GCP [235]), emphasizes: i) the validity of our threat model and ii) the large scale impact malicious (virtual) switches have in the data center. Compromising a hardware switch can follow a similar approach adopted in this dissertation, e.g., fuzzing the packet processing logic.

Furthermore, many countries have *digitalization* on their agenda. This means that is it highly likely that more data centers will be designed and used in the future. With only a handful of vendors capable of delivering equipment for such networks, and the increasing reliance on such critical infrastructure, attackers be it nation state or script-kiddies, will attempt to find the weakest link in the chain. We hope that this dissertation is i) a reminder that malicious switches are a real threat and ii) that we have to a degree prevented the centralized network controller and data plane systems from being weak links.

Bibliography

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. „Control-flow Integrity“. In: *Proc. ACM Conference on Computer and Communications Security (CCS)*. 2005, pp. 340–353 (cit. on p. 72).

[2] Stefan Achleitner, Thomas La Porta, Trent Jaeger, and Patrick McDaniel. „Adversarial network forensics in software defined networking“. In: *Proceedings of the Symposium on SDN Research*. ACM. 2017, pp. 8–20 (cit. on p. 1).

[3] Dan Gonzales et al. „Cloud-Trust - a Security Assessment Model for Infrastructure as a Service (IaaS) Clouds“. In: *Proc. IEEE Conference on Cloud Computing* PP.99 (2017), pp. 1–1 (cit. on p. 65).

[4] Nawaf Alhebaishi, Lingyu Wang, Sushil Jajodia, and Anoop Singhal. „Threat Modeling for Cloud Data Center Infrastructures“. In: *Intl. Symposium on Foundations and Practice of Security*. Springer. 2016, pp. 302–319 (cit. on p. 65).

[5] Tim Allclair. *Secure Container Isolation: Problem Statement & Solution Space*. `https://docs.google.com/document/d/1QQ5u1RBDLXWvC8K3pscTtTRThsOeBSts_imYEoRyw8A`. Accessed: 05-01-2019. 2018 (cit. on pp. 79, 81).

[6] Amazon AWS. *The Trusted Cloud for Government*. `https://aws.amazon.com/government-education/government/`. 2019 (cit. on p. 1).

[7] L. Andersson, P. Doolan, N. Feldman, A. Fredette, and B. Thomas. *LDP Specification*. RFC 3036 (Proposed Standard). Obsoleted by RFC 5036. Internet Engineering Task Force, 2001 (cit. on p. 68).

[8] Markku Antikainen, Tuomas Aura, and Mikko Särelä. „Spook in Your Network: Attacking an SDN with a Compromised OpenFlow Switch“. In: 2014, pp. 229–244 (cit. on p. 38).

[9] Apache. *ab - Apache HTTP server benchmarking tool*. `https://httpd.apache.org/docs/2.2/en/programs/ab.html`. Accessed: 07-01-2019 (cit. on p. 92).

[10] Sergei Arnautov, Bohdan Trach, Franz Gregor, et al. „SCONE: Secure Linux Containers with Intel SGX“. In: *Proc. Usenix Operating Systems Design Principles (OSDI)*. 2016 (cit. on p. 82).

[11] AWS. *Enhanced Networking on Linux*. `https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/enhanced-networking.html`. Accessed: 24-01-2018. 2018 (cit. on pp. 79, 80, 83).

[12] *AWS Case Study: Robinhood*. https://aws.amazon.com/solutions/case-studies/robinhood/. 2019 (cit. on p. 1).

[13] Microsoft Azure. *Create a Linux virtual machine with Accelerated Networking*. `https://docs.microsoft.com/en-us/azure/virtual-network/create-vm-accelerated-networking-cli`. Accessed: 24-01-2018. 2018 (cit. on p. 83).

[14]Julian Bangert and Nickolai Zeldovich. „Nail: A practical tool for parsing and generating data formats“. In: *Proc. Usenix Operating Systems Design Principles (OSDI)*. 2014, pp. 615–628 (cit. on p. 64).

[15]Adam Bates, Benjamin Mood, Joe Pletcher, et al. „On detecting co-resident cloud instances using network flow watermarking techniques“. In: *Springer International Journal of Information Security* (2014) (cit. on p. 97).

[16]Antoine Beaupré. *Updates in container isolation*. `https://lwn.net/Articles/754433`. Accessed: 09-01-2019. 2018 (cit. on pp. 79, 81).

[17]Kevin Benton, L. Jean Camp, and Chris Small. „OpenFlow Vulnerability Assessment“. In: *Proc. ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN)*. 2013 (cit. on p. 1).

[18]Kevin Benton, L. Jean Camp, and Chris Small. „OpenFlow Vulnerability Assessment“. In: *Proc. ACM Workshop on Hot Topics in Software Defined Networking (HotSDN)*. 2013, pp. 151–152 (cit. on p. 38).

[19]BESS Comitters. *BESS (Berkeley Extensible Software Switch)*. `https://github.com/NetSys/bess`. Accessed: 09-05-2017. 2017 (cit. on pp. 8, 64, 78).

[20]Matthew A Bishop. *Introduction to computer security*. Vol. 50. Addison-Wesley Boston, 2005 (cit. on pp. 77, 80, 81).

[21]Matthew A Bishop. *The art and science of computer security*. Addison-Wesley, 2002 (cit. on p. 42).

[22]Silas Boyd-Wickizer and Nickolai Zeldovich. „Tolerating Malicious Device Drivers in Linux.“ In: *Usenix Annual Technical Conference (ATC)*. 2010 (cit. on p. 97).

[23]*Broadcom Samples Thor, World’s First 200G Ethernet Controller with 50G PAM-4 and PCIe 4.0*. `https://www.broadcom.com/company/news/product-releases/2367107`. Accessed: 06-05-2019 (cit. on p. 97).

[24]*Bug 1578652 (CVE-2018-1000155)*. `https://bugzilla.redhat.com/show_bug.cgi?id=CVE-2018-1000155`. Accessed: 16 September 2019. 2018 (cit. on p. 56).

[25]Kevin Butler, Toni Farley, Patrick McDaniel, and Jennifer Rexford. *A Survey of BGP Security Issues and Solutions*. Tech. rep. AT&T Labs - Research, Florham Park, NJ, 2004 (cit. on p. 1).

[26]Serdar Cabuk, Carla E Brodley, and Clay Shields. „IP covert timing channels: design and detection“. In: *Proc. ACM Conference on Computer and Communications Security (CCS)*. 2004, pp. 178–187 (cit. on p. 41).

[27]M. Canini, P. Kuznetsov, D. Levin, and S. Schmid. „A distributed and robust SDN control plane for transactional network updates“. In: *Proc. IEEE INFOCOM*. 2015, pp. 190–198 (cit. on p. 38).

[28]Marco Canini, Daniele Venzano, Peter Perešíni, Dejan Kostić, and Jennifer Rexford. „A NICE Way to Test OpenFlow Applications“. In: *Proc. Usenix Networked Systems Design and Implementation (NSDI)*. 2012, pp. 127–140 (cit. on p. 38).

[29]Jiahao Cao, Qi Li, Renjie Xie, et al. „The CrossPath Attack: Disrupting the SDN Control Channel via Shared Links“. In: *Proc. Usenix Security Symp.* 2019, pp. 19–36 (cit. on p. 1).

[30]Martín Casado, Teemu Koponen, Rajiv Ramanathan, and Scott Shenker. „Virtualizing the Network Forwarding Plane“. In: *Proc. ACM CoNEXT Workshop on Programmable Routers for Extensible Services of Tomorrow*. 2010, 8:1–8:6 (cit. on p. 9).

[31]*Case Studies for Financial Services*. https://aws.amazon.com/financial-services/case-studies/. Accessed: 09-09-2019. 2019 (cit. on p. 1).

[32]D. Chasaki and T. Wolf. „Design of a secure packet processor“. In: *Proc. ACM/IEEE Architectures for Networking and Communication Systems (ANCS)*. 2010, pp. 1–10 (cit. on p. 76).

[33]Danai Chasaki and Tilman Wolf. „Attacks and Defenses in the Data Plane of Networks“. In: *Proc. IEEE/IFIP Transactions on Dependable and Secure Computing (DSN)* 9.6 (2012), pp. 798–810 (cit. on p. 76).

[34]Stephen Checkoway, Jacob Maskiewicz, Christina Garman, et al. „A systematic analysis of the Juniper Dual EC incident“. In: *Proc. ACM Conference on Computer and Communications Security (CCS)*. ACM. 2016, pp. 468–479 (cit. on p. 1).

[35]*Cisco VN-Link: Virtualization-Aware Networking*. White paper. 2009 (cit. on p. 8).

[36]Patrick Colp, Mihir Nanavati, Jun Zhu, et al. „Breaking up is hard to do: Security and Functionality in a Commodity Hypervisor“. In: *Proc. ACM Symposium on Operating System Principles (SOSP)*. 2011 (cit. on p. 80).

[37]A. Costin. „All your cluster-grids are belong to us: Monitoring the (in)security of infrastructure monitoring systems“. In: *Proc. IEEE Communications and Network Security (CNS)*. 2015, pp. 550–558 (cit. on pp. 66, 75).

[38]Crispin Cowan et al. „StackGuard: Automatic Adaptive Detection and Prevention of Buffer-overflow Attacks“. In: *Proc. Usenix Security Symp.* 1998, pp. 5–5 (cit. on pp. 69, 72).

[39]Levente Csikor, Christian Rothenberg, Dimitrios P. Pezaros, et al. „Policy Injection: A Cloud Dataplane DoS Attack“. In: *Proc. ACM SIGCOMM Posters and Demos*. 2018 (cit. on pp. 77–79).

[40]*CVE-2016-2074*. `https://nvd.nist.gov/vuln/detail/CVE-2016-2074`. Accessed: 9 September 2019. 2016 (cit. on p. 3).

[41]*CVE-2018-1000155: Denial of Service, Improper Authentication and Authorization, and Covert Channel in the OpenFlow 1.0+ handshake*. `https://www.openwall.com/lists/oss-security/2018/05/09/4`. Accessed: 9 September 2019. 2018 (cit. on pp. 3, 63).

[42]*CVE listing for CVE-2016-10377*. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-10377`. Accessed: 7-11-2019 (cit. on p. 68).

[43]*CVE listing for CVE-2016-2074*. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-2074`. Accessed: 19-07-2016 (cit. on p. 67).

[44]Michael Dalton, David Schultz, Jacob Adriaens, et al. „Andromeda: Performance, Isolation, and Velocity at Scale in Cloud Network Virtualization“. In: *Proc. Usenix Networked Systems Design and Implementation (NSDI)*. 2018 (cit. on pp. 8, 64, 77, 78, 86, 97).

[45]*Denial-of-Service (DoS) due to exceptions in application packet processors*. `https://wiki.onosproject.org/display/ONOS/Security+advisories`. Accessed: 9 September 2019. 2015 (cit. on p. 3).

[46]Mohan Dhawan, Rishabh Poddar, Kshiteej Mahajan, and Vijay Mann. „SPHINX: Detecting Security Attacks in Software-Defined Networks.“ In: *Proc. Internet Society Symposium on Network and Distributed System Security (NDSS)*. 2015 (cit. on pp. 1, 27, 38, 65, 75).

[47]Mihai Dobrescu and Katerina Argyraki. „Software Dataplane Verification“. In: *Proc. Usenix Networked Systems Design and Implementation (NSDI)*. 2014, pp. 101–114 (cit. on p. 76).

[48]Jeremy M. Dover. *A denial of service attack against the Open Floodlight SDN controller*. Tech. rep. Dover Networks, 2013 (cit. on pp. 1, 39).

[49]Dpdk. *Writing Efficient Code*. `https://doc.dpdk.org/guides/prog_guide/writing_efficient_code.html`. Accessed: 06-01-2019 (cit. on p. 90).

[50]Holger Dreger, Anja Feldmann, Vern Paxson, and Robin Sommer. „Operational Experiences with High-volume Network Intrusion Detection“. In: *Proc. ACM Conference on Computer and Communications Security (CCS)*. CCS ’04. ACM, 2004, pp. 2–11 (cit. on p. 101).

[51]Holger Dreger, Anja Feldmann, Vern Paxson, and Robin Sommer. „Predicting the Resource Consumption of Network Intrusion Detection Systems“. In: *Recent Advances in Intrusion Detection*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 135–154 (cit. on p. 101).

[52]Paul Emmerich, Daniel Raumer, Florian Wohlfart, and Georg Carle. „Performance characteristics of virtual switching“. In: *Proc. IEEE Conference on Cloud Networking*. 2014 (cit. on p. 86).

[53]*Endace DAG 10X4-P Datasheet*. `https://www.endace.com/dag-10x4-p-datasheet.pdf`. Accessed: 07-01-2019 (cit. on p. 88).

[54]Haggai Eran, Lior Zeno, Maroun Tork, Gabi Malka, and Mark Silberstein. „NICA: An Infrastructure for Inline Acceleration of Network Applications“. In: *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, 2019, pp. 345–362 (cit. on p. 101).

[55]Snorre Fagerland, Waylon Grange, and Blue Coat Systems Inc. *The Inception Framework: Cloud-Hosted APT*. `http://dc.bluecoat.com/Inception_Framework`. Accessed: 2017-02-06. 2014 (cit. on p. 15).

[56]Anja Feldmann, Philipp Heyder, Michael Kreutzer, et al. „NetCo: Reliable Routing With Unreliable Routers“. In: *IEEE Workshop on Dependability Issues on SDN and NFV*. 2016 (cit. on p. 65).

[57]Tobias Fiebig, Franziska Lichtblau, Florian Streibelt, et al. „SoK: An Analysis of Protocol Design: Avoiding Traps for Implementation and Deployment“. In: *arXiv preprint arXiv:1610.05531* (2016) (cit. on p. 61).

[58]Daniel Firestone. „VFP: A Virtual Switch Platform for Host SDN in the Public Cloud“. In: *Proc. Usenix Networked Systems Design and Implementation (NSDI)*. 2017, pp. 315–328 (cit. on pp. 6, 8, 61, 63, 64, 78).

[59]Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, et al. „Azure Accelerated Networking: SmartNICs in the Public Cloud“. In: *Proc. Usenix Networked Systems Design and Implementation (NSDI)*. 2018 (cit. on pp. 8, 64, 77, 78, 80, 86, 97).

[60]*Fix for CVE-2018-1000155*. `https://github.com/opennetworkinglab/onos/commit/f69e3e340921396004046817 98cebeefebcfa6c6`. Accessed: 16 September 2019. 2018 (cit. on p. 56).

[61]Sadjad Fouladi, Francisco Romero, Dan Iter, et al. „From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers“. In: *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, 2019, pp. 475–488 (cit. on p. 102).

[62]Jessie Frazelle. *Hard Multi-Tenancy in Kubernetes*. `https://blog.jessfraz.com/post/hard-multi-tenancy-in-kubernetes`. Accessed: 09-01-2019. 2018 (cit. on pp. 61, 79, 82).

[63]Felix C. Freiling. *Detecting Targeted Attacks Considered Impossible*. Dagstuhl Seminar 12502. Accessed on 25.10.2015. 2012 (cit. on p. 101).

[64]Marcos Garcia. *How connection tracking in Open vSwitch helps OpenStack performance*. `https://www.redhat.com/en/blog/how-connection-tracking-open-vswitch-helps-openstack-performance`. Accessed: 17-09-2019. 2016 (cit. on p. 67).

[65]Yuri Gbur. *A Feasibility Study of SDN Teleportation in P4Runtime*. Bachelor's Thesis. `https://github.com/yurigbur/publications/blob/master/Bachelorthesis.pdf`. 2018 (cit. on p. 56).

[66]*Geneve: Generic Network Virtualization Encapsulation*. `https://tools.ietf.org/html/draft-ietf-nvo3-geneve-08`. Accessed: 03-01-2019 (cit. on p. 97).

[67]Michael Gilleland and Merriam Park Software. *Levenshtein Distance, in Three Flavors*. `https://people.cs.pitt.edu/~kirk/cs1501/Pruhs/Spring2006/assignments/editdistance/Levenshtein%20Distance.htm`. Accessed: 02-01-2018. 2017 (cit. on p. 52).

[68]C. Gray Girling. „Covert Channels in LAN's“. In: *IEEE Trans. Software Engineering* 13.2 (1987), p. 292 (cit. on p. 41).

[69]*Google Compute Engine Pricing*. https://cloud.google.com/compute/pricing. Accessed: 03-01-2019. 2018 (cit. on p. 97).

[70]Andy Gospodarek. *The Rise of SmartNICs – offloading dataplane traffic to...software*. https://youtu.be/AGSy51VlKaM. Open vSwitch Conference. 2017 (cit. on pp. 8, 64, 78, 79).

[71]Nicholas Gray, Thomas Zinner, and Phuoc Tran-Gia. „Enhancing SDN Security by Device Fingerprinting“. In: *In Proc. IFIP/IEEE International Symposium on Integrated Network Management (IM)*. 2017 (cit. on p. 56).

[72]Bernd Grobauer, Tobias Walloschek, and Elmar Stöcker. „Understanding Cloud Computing Vulnerabilities“. In: *IEEE Security & Privacy Magazine* 9.2 (2011), pp. 50–57 (cit. on p. 65).

[73]Theodore G Handel and Maxwell T Sandford. „Hiding data in the OSI network model“. In: *Proc. Intl. Workshop on Information Hiding*. Springer. 1996, pp. 23–38 (cit. on p. 41).

[74]Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, et al. „Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries“. In: *Usenix Annual Technical Conference (ATC)*. 2019, pp. 489–504 (cit. on p. 98).

[75]Michio Honda, Felipe Huici, Giuseppe Lettieri, and Luigi Rizzo. „mSwitch: a highly-scalable, modular software switch“. In: *Proc. ACM Symposium on SDN Research (SOSR)*. 2015 (cit. on pp. 8, 64, 77, 78).

[76]Sungmin Hong, Lei Xu, Haopei Wang, and Guofei Gu. „Poisoning Network Visibility in Software-Defined Networks: New Attacks and Countermeasures.“ In: *Proc. Internet Society Symposium on Network and Distributed System Security (NDSS)*. 2015 (cit. on pp. 1, 76).

[77]Sungmin Hong, Lei Xu, Haopei Wang, and Guofei Gu. „Poisoning Network Visibility in Software-Defined Networks: New Attacks and Countermeasures“. In: *NDSS*. 2015 (cit. on pp. 27, 38).

[78]*HowTo Launch VM over OVS-DPDK-17.11 Using Mellanox ConnectX-4 and ConnectX-5*. https://community.mellanox.com/s/article/howto-launch-vm-over-ovs-dpdk-17-11-using-mellanox-connectx-4-and-connectx-5. Accessed: 09-01-2019 (cit. on p. 88).

[79]Yiyuan Hu, Xiangyang Li, and Xenia Mountrouidou. „Improving covert storage channel analysis with SDN and experimentation on GENI“. In: *National Cyber Summit* 16 (2016), pp. 7–9 (cit. on p. 41).

[80]*Huawei HG8245 backdoor and remote access*. http://websec.ca/advisories/view/Huawei-web-backdoor-and-remote-access. Accessed: 2017-02-06. 2013 (cit. on p. 1).

[81]Jinho Hwang, KK Ramakrishnan, and Timothy Wood. „NetVM: high performance and flexible networking using virtualization on commodity platforms“. In: *Proc. Usenix Networked Systems Design and Implementation (NSDI)*. 2014 (cit. on pp. 8, 64, 78).

[82]Boris Iglewicz and David Caster Hoaglin. *How to detect and handle outliers*. Vol. 16. Asq Press, 1993 (cit. on p. 35).

[83]*India Poised for Massive Data Center Growth*. https://www.whatsnextcw.com/india-poised-for-massive-data-center-growth/. Accessed: 28-10-2019. 2018 (cit. on p. 1).

[84]Federal Office for Information Security (BSI). *BSI: Recommendations for critical information infrastructure protection*. https://www.bsi.bund.de/EN/Topics/Criticalinfrastructures/criticalinfrastructures_node.html. Accessed: 28-10-2019. 2015 (cit. on p. 1).

[85]Intel. *Enabling NFV to Deliver on its Promise*. https://www-ssl.intel.com/content/www/us/en/communications/nfv-packet-processing-brief.html. Accessed: 28-10-2019. 2015 (cit. on p. 83).

[86]*Intent-Based Networking (IBN) Explained*. https://www.cisco.com/c/en/us/solutions/intent-based-networking.html. Accessed: 25-10-2019. 2019 (cit. on p. 39).

[87]Internet Science Working Group. *Internet as a Critical Infrastructure: Security, Resilience and Dependability Aspects (JRA7)*. http://www.internet-science.eu/groups/internet-critical-infrastructure-security-resilience-and-dependability-aspects. Accessed: 2017-02-06. 2015 (cit. on p. 1).

[88]*iPerf - The ultimate speed test tool for TCP, UDP and SCTP*. https://iperf.fr/. Accessed: 07-01-2019 (cit. on p. 92).

[89]Ethan J. Jackson, Melvin Walls, Aurojit Panda, et al. „Softflow: A middlebox architecture for open vswitch". In: *Usenix Annual Technical Conference (ATC)*. 2016, pp. 15–28 (cit. on pp. 8, 61, 63).

[90]Van Jacobson. „Congestion avoidance and control". In: *ACM Computer Communication Review (CCR)*. 1988 (cit. on p. 74).

[91]Raj Jain and Sudipta Paul. „Network virtualization and software defined networking for cloud computing: a survey". In: *IEEE Communication Magazine* 51.11 (2013) (cit. on pp. 8, 61, 77, 78).

[92]Michael Jarschel, Christopher Metter, Thomas Zinner, Steffen Gebert, and Phuoc Tran-Gia. „OFCProbe: A platform-independent tool for OpenFlow controller analysis". In: *Communications and Electronics (ICCE), 2014 IEEE Fifth International Conference on*. IEEE. 2014, pp. 182–187 (cit. on pp. 33, 52).

[93]Theo Jepsen, Daniel Alvarez, Nate Foster, et al. „Fast String Searching on PISA". In: *Proceedings of the 2019 ACM Symposium on SDN Research*. ACM. 2019, pp. 21–28 (cit. on p. 101).

[94]Samuel Jero, Xiangyu Bu, Cristina Nita-Rotaru, et al. „BEADS: Automated Attack Discovery in OpenFlow-Based SDN Systems". In: *Proc. RAID Recent Advances in Intrusion Detection*. 2017 (cit. on pp. 1, 65).

[95]Samuel Jero, William Koch, Richard Skowyra, et al. „Identifier Binding Attacks and Defenses in Software-Defined Networks". In: *Proc. Usenix Security Symp.* 2017 (cit. on p. 1).

[96]Vimalkumar Jeyakumar, Mohammad Alizadeh, David Mazières, et al. „EyeQ: Practical Network Performance Isolation at the Edge". In: *Proc. Usenix Networked Systems Design and Implementation (NSDI)*. 2013, pp. 297–311 (cit. on p. 97).

[97]Xin Jin, Jennifer Gossels, Jennifer Rexford, and David Walker. „CoVisor: A Compositional Hypervisor for Software-Defined Networks". In: *Proc. Usenix Networked Systems Design and Implementation (NSDI)*. 2015 (cit. on p. 39).

[98]Xin Jin, Eric Keller, and Jennifer Rexford. „Virtual Switching Without a Hypervisor for a More Secure Cloud". In: *Proc. USENIX Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (HotICE)*. 2012 (cit. on pp. 8, 64, 65, 78, 79, 83, 98, 101).

[99]Chen Jing. *Zero-Copy Optimization for Alibaba Cloud Smart NIC Solution*. http://www.alibabacloud.com/blog/zero-copy-optimization-for-alibaba-cloud-smart-nic-solution_593986. Accessed: 03-01-2019. 2018 (cit. on pp. 8, 77, 79, 80, 83).

[100]Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, et al. „Cloud Programming Simplified: A Berkeley View on Serverless Computing". In: *CoRR* abs/1902.03383 (2019). arXiv: 1902.03383 (cit. on p. 102).

[101]Daya Kamath et al. „Edge virtual Bridge Proposal, Version 0. Rev. 0.1". In: *Apr* 23 (2010), pp. 1–72 (cit. on p. 8).

[102]Andrzej Kamisiński and Carol Fung. „FlowMon: Detecting Malicious Switches in Software-Defined Networks". In: *Proc. ACM Workshop on Automated Decision making for Active Cyber Defense*. 2015, pp. 39–45 (cit. on p. 76).

[103]Nanxi Kang, Zhenming Liu, Jennifer Rexford, and David Walker. „Optimizing the "One Big Switch" Abstraction in Software-defined Networks“. In: *Proc. ACM CoNEXT*. 2013 (cit. on p. 24).

[104]Kallol Krishna Karmakar, Vijay Varadharajan, and Uday Tupakula. „Mitigating attacks in Software Defined Network (SDN)“. In: *Proc. IEEE Software Defined Systems (SDS)*. 2017, pp. 112–117 (cit. on p. 65).

[105]Peyman Kazemian, George Varghese, and Nick McKeown. „Header Space Analysis: Static Checking for Networks“. In: *Proc. Usenix Networked Systems Design and Implementation (NSDI)*. 2012, pp. 113–126 (cit. on p. 38).

[106]Junaid Khalid, Eric Rozner, Wesley Felter, et al. „Iron: Isolating Network-based CPU in Container Environments“. In: *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. Renton, WA: USENIX Association, 2018, pp. 313–328 (cit. on p. 102).

[107]Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. „VeriFlow: Verifying Network-Wide Invariants in Real Time“. In: *Proc. Usenix Networked Systems Design and Implementation (NSDI)*. 2013, pp. 467–472 (cit. on p. 38).

[108]R. Klöti, V. Kotronis, and P. Smith. „OpenFlow: A security analysis“. In: *Proc. IEEE International Conference on Network Protocols (ICNP)*. 2013, pp. 1–6 (cit. on pp. 1, 38, 75).

[109]Mike Ko and Renato Recio. *Virtual Ethernet Bridging*. `http://www.ieee802.org/1/files/public/docs2009/new-hudson-vepa_seminar-20090514d.pdf`. Accessed: 06-01-2019 (cit. on p. 82).

[110]Teemu Koponen, Keith Amidon, Peter Balland, et al. „Network Virtualization in Multi-tenant Datacenters“. In: *Proc. Usenix Networked Systems Design and Implementation (NSDI)*. 2014 (cit. on pp. 1, 8, 61, 77, 78, 97).

[111]Diego Kreutz, Fernando M.V. Ramos, and Paulo Verissimo. „Towards Secure and Dependable Software-defined Networks“. In: *Proc. ACM Workshop on Hot Topics in Software Defined Networking (HotSDN)*. 2013, pp. 55–60 (cit. on pp. 1, 38, 75).

[112]Robert Krösche, Kashyap Thimmaraju, Liron Schiff, and Stefan Schmid. „I DPID It My Way! A Covert Timing Channel in Software-Defined Networks“. In: *Proceedings of the IFIP Networking Conference (IFIP Networking) and Workshops*. IEEE. 2018, pp. 217–225 (cit. on p. ix).

[113]Praveen Kumar, Nandita Dukkipati, Nathan Lewis, et al. „PicNIC: Predictable Virtualized NIC“. In: *Proc. ACM SIGCOMM*. 2019, pp. 351–366 (cit. on p. 97).

[114]Nate Kushman, Srikanth Kandula, Dina Katabi, and Bruce M. Maggs. „R-BGP: Staying connected in a connected world“. In: *Proc. Usenix Networked Systems Design and Implementation (NSDI)*. 2007 (cit. on p. 1).

[115]Patrick Kutch. „PCI-SIG SR-IOV primer: An introduction to SR-IOV technology“. In: *Intel application note* (2011), pp. 321211–002 (cit. on p. 83).

[116]Volodymyr Kuznetsov et al. „Code-Pointer Integrity“. In: *Proc. Usenix Operating Systems Design Principles (OSDI)*. 2014, pp. 147–163 (cit. on p. 72).

[117]Maciej Kuźniar, Peter Perešíni, and Dejan Kostić. „What You Need to Know About SDN Flow Tables“. In: *Proc. Passive and Active Measurement (PAM)*. 2015 (cit. on p. 39).

[118]Butler W Lampson. „A note on the confinement problem“. In: *Communications of the ACM* 16.10 (1973), pp. 613–615 (cit. on p. 42).

[119]Sihyung Lee, Tina Wong, and Hyong S Kim. „Secure split assignment trajectory sampling: A malicious router detection system“. In: *Proc. IEEE/IFIP Transactions on Dependable and Secure Computing (DSN)*. 2006, pp. 333–342 (cit. on p. 76).

[120]T. Lévai, G. Pongrácz, P. Megyesi, et al. „The Price for Programmability in the Software Data Plane: The Vendor Perspective“. In: *IEEE J. Selected Areas in Communications* (2018) (cit. on p. 83).

[121] Felix Lindner. *Cisco IOS router exploitation*. Tech. rep. Accessed: 2017-02-06. Recurity Labs, 2009 (cit. on p. 1).

[122] Ming Liu, Tianyi Cui, Henry Schuh, et al. „Offloading Distributed Applications Onto smartNICs Using iPipe“. In: *Proc. ACM SIGCOMM*. SIGCOMM '19. ACM, 2019, pp. 318–333 (cit. on p. 101).

[123] Sheng Liu, Michael K Reiter, and Vyas Sekar. „Flow reconnaissance via timing attacks on SDN switches“. In: *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE. 2017, pp. 196–206 (cit. on p. 1).

[124] Angus Loten. *Data-Center Market Is Booming Amid Shift to Cloud*. `https://www.wsj.com/articles/data-center-market-is-booming-amid-shift-to-cloud-11566252481`. 2019 (cit. on p. 103).

[125] Stephanos Matsumoto, Samuel Hitz, and Adrian Perrig. „Fleet: Defending SDNs from malicious administrators“. In: *Proc. ACM Workshop on Hot Topics in Software Defined Networking (HotSDN)*. 2014, pp. 103–108 (cit. on p. 75).

[126] James McCauley, Aurojit Panda, Arvind Krishnamurthy, and Scott Shenker. „Thoughts on load distribution and the role of programmable switches“. In: *ACM SIGCOMM Computer Communication Review* 49.1 (2019), pp. 18–23 (cit. on p. 101).

[127] Nick McKeown, Tom Anderson, Hari Balakrishnan, et al. „OpenFlow: enabling innovation in campus networks“. In: *SIGCOMM Comput. Commun. Rev.* 38.2 (2008), pp. 69–74 (cit. on pp. 6, 61).

[128] Syed Akbar Mehdi, Junaid Khalid, and Syed Ali Khayam. „Revisiting Traffic Anomaly Detection Using Software Defined Networking“. In: *Proc. RAID Recent Advances in Intrusion Detection*. 2011, pp. 161–180 (cit. on p. 38).

[129] Mellanox. *Mellanox BlueField SmartNIC*. `https://bit.ly/2JaMitA`. Accessed: 05-06-2018. 2017 (cit. on pp. 8, 64, 78, 79).

[130] *Memcached*. `https://libmemcached.org/libMemcached.html`. Accessed: 07-01-2019 (cit. on p. 92).

[131] Dirk Merkel. „Docker: lightweight linux containers for consistent development and deployment“. In: *Linux Journal* (2014), p. 2 (cit. on p. 94).

[132] Dan Meyer. *Microsoft Scores Cloud Deal With India's Reliance Jio*. `https://www.sdxcentral.com/articles/news/microsoft-scores-cloud-deal-with-indias-reliance-jio/2019/08/`. Accessed: 28-10-2019. 2019 (cit. on p. 1).

[133] Microsoft. *Hyper-V Virtual Switch Overview*. `https://technet.microsoft.com/en-us/library/hh831823(v=ws.11).aspx`. Accessed: 27-01-2017. 2013 (cit. on pp. 8, 64, 78).

[134] Microsoft. *SR-IOV VF Failover and Live Migration Support*. `https://docs.microsoft.com/en-us/windows-hardware/drivers/network/sr-iov-vf-failover-and-live-migration-support`. Accessed: 03-01-2019. 2017 (cit. on p. 97).

[135] David L Mills. „On the accuracy and stablility of clocks synchronized by the network time protocol in the Internet system“. In: *ACM Computer Communication Review (CCR)* 20.1 (1989), pp. 65–75 (cit. on p. 49).

[136] Mirantis. *Control plane virtual machines*. `https://docs.mirantis.com/mcp/q4-18/mcp-ref-arch/infra-nodes-plan/cluster-nodes-overview.html`. Accessed: 17-09-2019. 2019 (cit. on p. 67).

[137] László Molnár, Gergely Pongrácz, Gábor Enyedi, et al. „Dataplane Specialization for High-performance OpenFlow Software Switching“. In: *Proc. ACM SIGCOMM*. 2016 (cit. on pp. 8, 64, 78).

[138]Jayaram Mudigonda, Praveen Yalagandula, Jeff Mogul, Bryan Stiekes, and Yanick Pouffary. „NetLord: A Scalable Multi-tenant Network Architecture for Virtualized Datacenters“. In: *Proc. ACM SIGCOMM*. 2011, pp. 62–73 (cit. on p. 38).

[139]Assaf Muller. *OVS ARP Responder – Theory and Practice*. `https://assafmuller.com/2014/05/21/ovs-arp-responder-theory-and-practice/`. Accessed: 06-01-2019 (cit. on p. 85).

[140]Jad Naous, Michael Walfish, Antonio Nicolosi, et al. „Verifying and Enforcing Network Paths with Icing“. In: *Proc. ACM CoNEXT*. 2011, 30:1–30:12 (cit. on p. 1).

[141]*Netis Routers Leave Wide Open Backdoor*. `http://blog.trendmicro.com/trendlabs-security-intelligence/netis-routers-leave-wide-open-backdoor/`. Accessed: 2017-02-06. 2014 (cit. on p. 1).

[142]Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, et al. „Understanding PCIe performance for end host networking“. In: *Proc. ACM SIGCOMM*. 2018 (cit. on p. 97).

[143]Miguel Neves, Kirill Levchenko, and Marinho Barcellos. „Sandboxing Data Plane Programs for Fun and Profit“. In: *Proc. ACM SIGCOMM Posters and Demos*. 2017 (cit. on p. 98).

[144]ONOS. *Security advisories*. `https://wiki.onosproject.org/display/ONOS/Security+advisories`. Accessed: 2017-02-06. 2015 (cit. on pp. 23, 30).

[145]*ONOS Wiki Home*. `https://wiki.onosproject.org/display/ONOS/Wiki+Home`. Accessed: 02-01-2018. 2017 (cit. on p. 51).

[146]P.C. van Oorschot, Tao Wan, and Evangelos Kranakis. „On Interdomain Routing Security and Pretty Secure BGP (psBGP)“. In: *ACM Trans. Inf. Syst. Secur.* 10.3 (2007) (cit. on p. 1).

[147]Open vSwitch. *Open vSwitch*. `http://openvswitch.org`. Accessed: 02-01-2018. 2018 (cit. on pp. 8, 51).

[148]*OpenFlow protocol bug to get mitigations, not a rewrite*. `https://www.theregister.co.uk/2018/05/14/onf_wont_patch_openflow_protocol_vulnerability/`. Accessed: 9-9-2019 (cit. on p. 3).

[149]*OpenFlow protocol has a switch authentication vulnerability*. `https://www.theregister.co.uk/2018/05/10/openflow_switch_auth_vulnerability/`. Accessed: 9 September 2019. 2018 (cit. on p. 3).

[150]OpenFlow Spec. In: *openflow.org* (2013) (cit. on pp. 6, 7).

[151]*OpenFlow Switch Specification*. ONF TS-009. Version 1.3.2 Wire Protocol 0x04. Open Networking Foundation. Open Networking Foundation, 2013 (cit. on pp. 7, 42).

[152]OpenStack. *Control plane architecture*. `https://docs.openstack.org/arch-design/design-control-plane.html`. Accessed: 17-09-2019. 2019 (cit. on p. 63).

[153]OpenStack. *Native Open vSwitch firewall driver*. `https://docs.openstack.org/newton/networking-guide/config-ovsfwdriver.html`. Accessed: 17-09-2019. 2019 (cit. on p. 67).

[154]OpenStack. *Provider networks with Open vSwitch*. `https://docs.openstack.org/liberty/networking-guide/scenario-provider-ovs.html`. Accessed: 17-09-2019. 2016 (cit. on pp. 66, 67).

[155]*OpenStack Security Guide*. `http://docs.openstack.org/security-guide`. Accessed: 27-01-2017. 2016 (cit. on pp. 62, 66, 75).

[156]Adar Ovadia, Rom Ogen, Yakov Mallah, Niv Gilboa, and Yossi Oren. „Cross-Router Covert Channels“. In: *Proc. Usenix Workshop on Offensive Technologies (WOOT)*. 2019 (cit. on p. 41).

[157]*[ovs-announce] CVE-2016-2074: MPLS buffer overflow vulnerabilities in Open vSwitch*. `https://mail.openvswitch.org/pipermail/ovs-announce/2016-March/000082.html`. Accessed: 9 September 2019. 2016 (cit. on p. 3).

[158]*P4Runtime Specification*. https://github.com/p4lang/p4runtime. Accessed: 16 September 2019. 2019 (cit. on p. 56).

[159]Oded Padon, Neil Immerman, Aleksandr Karbyshev, et al. „Decentralizing SDN Policies“. In: *Proc. ACM POPL*. 2015 (cit. on p. 38).

[160]Nicolae Paladi and Christian Gehrmann. „SDN Access Control for the Masses“. In: *Elsevier Computers & Security* (2019) (cit. on p. 82).

[161]Nicolae Paladi and Christian Gehrmann. „Towards Secure Multi-tenant Virtualized Networks“. In: *Proc. IEEE Trustcom/BigDataSE/ISPA*. Vol. 1. 2015, pp. 1180–1185 (cit. on p. 65).

[162]Aurojit Panda, Sangjin Han, Keon Jang, et al. „NetBricks: Taking the V out of NFV“. In: *Proc. Usenix Operating Systems Design Principles (OSDI)*. 2016 (cit. on p. 98).

[163]Manoj Panicker. *Enabling Hardware Offload of OVS Control & Data plane using LiquidIO*. https://youtu.be/qjXBRCFhbqU. Open vSwitch Conference. 2017 (cit. on pp. 8, 64, 78, 79).

[164]Mathias Payer. *Too much PIE is bad for performance*. http://e-collection.library.ethz.ch/eserv/eth:5699/eth-5699-01.pdf. Accessed: 27-01-2017. 2012 (cit. on p. 72).

[165]Diego Perez-Botero, Jakub Szefer, and Ruby B. Lee. „Characterizing Hypervisor Vulnerabilities in Cloud Computing Servers“. In: *Proc. ACM Workshop on Security in Cloud Computing*. 2013, pp. 3–10 (cit. on p. 76).

[166]Justin Pettit and Thomas Graf. *Stateful connection tracking & Stateful NAT*. Open vSwitch Fall Conference 2014. 2014 (cit. on p. 67).

[167]Justin Pettit, Ben Pfaff, Joe Stringer, et al. „Bringing platform harmony to VMware NSX“. In: *ACM SIGOPS Operating System Review* (2018) (cit. on p. 98).

[168]Justin Pettit, Ben Pfaff, Chris Wright, and Madhu Venugopal. *OVN, Bringing Native Virtual Networking to OVS*. https://networkheresy.com/2015/01/13/ovn-bringing-native-virtual-networking-to-ovs/. Accessed: 27-01-2017. 2015 (cit. on pp. 8, 61, 77, 78).

[169]Ben Pfaff. *Open vSwitch: Past, Present, and Future*. http://openvswitch.org/slides/ppf.pdf. Accessed: 27-01-2017. 2013 (cit. on pp. 8, 64, 67, 78).

[170]Ben Pfaff, Justin Pettit, Keith Amidon, et al. „Extending Networking into the Virtualization Layer.“ In: *Proc. ACM Workshop on Hot Topics in Networks (HotNETs)*. 2009 (cit. on pp. 9, 61, 65).

[171]Ben Pfaff, Justin Pettit, Teemu Koponen, et al. „The design and implementation of Open vSwitch“. In: *Proc. Usenix Networked Systems Design and Implementation (NSDI)*. 2015 (cit. on pp. 8, 9, 61, 63, 73, 75).

[172]Gregory Pickett. „Abusing software defined networks“. In: *Black Hat EU* (2014) (cit. on p. 65).

[173]Lucian Popa, Gautam Kumar, Mosharaf Chowdhury, et al. „FairCloud: Sharing the Network in Cloud Computing“. In: *Proc. ACM SIGCOMM* (2012), pp. 187–198 (cit. on p. 97).

[174]Philip Porras, Seungwon Shin, Vinod Yegneswaran, et al. „A Security Enforcement Kernel for OpenFlow Networks“. In: *Proc. ACM Workshop on Hot Topics in Software Defined Networking (HotSDN)*. 2012, pp. 121–126 (cit. on pp. 1, 38, 39).

[175]Phillip Porras, Steven Cheung, Martin Fong, Keith Skinner, and Vinod Yegneswaran. „Securing the Software-Defined Network Control Layer“. In: *Proc. Internet Society Symposium on Network and Distributed System Security (NDSS)*. 2015 (cit. on p. 39).

[176]Kaushik Kumar Ram, Alan L Cox, Mehul Chadha, Scott Rixner, and TW Barr. „Hyper-Switch: A Scalable Software Virtual Switching Architecture“. In: *Usenix Annual Technical Conference (ATC)*. 2013 (cit. on pp. 8, 64, 78).

[177]*Re: [Ryu-devel] CVE-2018-1000155*. https://sourceforge.net/p/ryu/mailman/message/36320200/. Accessed: 16 September 2019. 2018 (cit. on p. 56).

[178] Mark Reitblatt, Nate Foster, Jennifer Rexford, and David Walker. „Consistent Updates for Software-defined Networks: Change You Can Believe in!“ In: *Proc. HotNets*. 2011, 7:1–7:6 (cit. on p. 38).

[179] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. „Hey, You, Get off of My Cloud: Exploring Information Leakage in Third-party Compute Clouds“. In: *Proc. ACM Conference on Computer and Communications Security (CCS)*. 2009, pp. 199–212 (cit. on p. 75).

[180] Sasko Ristov, Marjan Gusev, and Aleksandar Donevski. *Openstack cloud security vulnerabilities from inside and outside*. Technical Report. 2013 (cit. on p. 75).

[181] Luigi Rizzo. „Netmap: a novel framework for fast packet I/O“. In: *Usenix Annual Technical Conference (ATC)*. 2012 (cit. on p. 83).

[182] Luigi Rizzo and Giuseppe Lettieri. „VALE, a Switched Ethernet for Virtual Machines“. In: *Proc. ACM CoNEXT*. 2012 (cit. on pp. 8, 64, 78).

[183] Jordan Robertson and Michael Riley. *The Big Hack: How China Used a Tiny Chip to Infiltrate U.S. Companies*. https://www.bloomberg.com/news/features/2018-10-04/the-big-hack-how-china-used-a-tiny-chip-to-infiltrate-america-s-top-companies. Accessed: 5-11-2018. 2018 (cit. on p. 1).

[184] Robin G. *Open vSwitch with DPDK Overview*. https://software.intel.com/en-us/articles/open-vswitch-with-dpdk-overview. Accessed: 27-01-2017. 2016 (cit. on pp. 8, 64, 78, 83).

[185] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. „Return-Oriented Programming: Systems, Languages, and Applications“. In: *ACM Trans. on Information and System Security (TISSEC)* 15.1 (2012), 2:1–2:34 (cit. on p. 70).

[186] *ROPGadget Tool*. https://github.com/JonathanSalwan/ROPgadget/tree/master. Accessed: 02-06-2016 (cit. on p. 70).

[187] Eric C. Rosen et al. *MPLS Label Stack Encoding*. RFC 3032 (Proposed Standard). Internet Engineering Task Force, 2001 (cit. on pp. 68–70).

[188] Eric C. Rosen, Arun Viswanathan, and Ross Callon. *Multiprotocol Label Switching Architecture*. RFC 3031 (Proposed Standard). Internet Engineering Task Force, 2001 (cit. on p. 69).

[189] Joanna Rutkowska and Rafal Wojtczuk. „Qubes OS architecture“. In: *Invisible Things Lab Tech Rep* 54 (2010) (cit. on p. 65).

[190] *[Ryu-devel]: CVE-2018-1000155: Denial of Service, Improper Authentication and Authorization, and Covert Channel in the OpenFlow 1.0+ handshake*. https://sourceforge.net/p/ryu/mailman/message/36329676/. Accessed: 9 September 2019. 2018 (cit. on p. 3).

[191] Jerome H. Saltzer and Michael D. Schroeder. „The protection of information in computer systems“. In: *Proceedings of the IEEE* 63.9 (1975), pp. 1278–1308 (cit. on pp. 77, 80).

[192] Takayuki Sasaki, Christos Pappas, Taeho Lee, Torsten Hoefler, and Adrian Perrig. „SDNsec: Forwarding accountability for the SDN data plane“. In: *2016 25th International Conference on Computer Communication and Networks (ICCCN)*. IEEE. 2016, pp. 1–10 (cit. on p. 1).

[193] Len Sassaman et al. „Security Applications of Formal Language Theory“. In: *IEEE Systems Journal* 7.3 (2013), pp. 489–500 (cit. on pp. 61, 64).

[194] *Security concerns for the P4Runtime handshake*. https://github.com/p4lang/p4runtime/pull/161. Accessed: 16 September 2019. 2019 (cit. on p. 56).

[195] *Security-Mode ONOS*. https://wiki.onosproject.org/display/ONOS/Security-Mode+ONOS. Accessed: 2017-02-06. 2015 (cit. on p. 39).

[196]SecuritytWeek. *CSA's Cloud Adoption, Practices and Priorities Survey Report*. http://www.securityweek.com/data-security-concerns-still-challenge. Accessed: 09-01-2019. 2015 (cit. on p. 77).

[197]*Send a raw Ethernet frame in Linux*. https://gist.github.com/austinmarton/1922600. Accessed: 2017-02-06. 2011 (cit. on p. 24).

[198]Muhammad Shahbaz, Sean Choi, Ben Pfaff, et al. „Pisces: A programmable, protocol-independent software switch". In: *Proc. ACM SIGCOMM*. 2016 (cit. on pp. 8, 64, 78, 98).

[199]Naveen Kr. Sharma, Ming Liu, Kishore Atreya, and Arvind Krishnamurthy. „Approximating Fair Queueing on Reconfigurable Switches". In: *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. Renton, WA: USENIX Association, 2018, pp. 1–16 (cit. on p. 101).

[200]Bhargava Shastry, Markus Leutner, Tobias Fiebig, et al. „Static Program Analysis as a Fuzzing Aid". In: *Proc. RAID Recent Advances in Intrusion Detection*. 2017 (cit. on p. 61).

[201]Rob Sherwood, Glen Gibb, Kok-Kiong Yap, et al. *Flowvisor: A network virtualization layer*. Tech. rep. OpenFlow, 2009 (cit. on p. 39).

[202]Alan Shieh, Srikanth Kandula, Albert Greenberg, Changhoon Kim, and Bikas Saha. „Sharing the Data Center Network". In: *Proc. Usenix Networked Systems Design and Implementation (NSDI)*. Boston, MA, 2011, pp. 309–322 (cit. on p. 97).

[203]S. Shin, P. Porras, V. Yegneswaran, et al. „Fresco: Modular composable security services for software-defined networks". In: *Proc. Internet Society Symposium on Network and Distributed System Security (NDSS)*. 2013 (cit. on pp. 1, 38).

[204]Seungwon Shin, Vinod Yegneswaran, Phillip Porras, and Guofei Gu. „AVANT-GUARD: Scalable and Vigilant Switch Flow Management in Software-defined Networks". In: *Proc. ACM Conference on Computer and Communications Security (CCS)*. 2013, pp. 413–424 (cit. on pp. 38, 75).

[205]Gustavus J. Simmons. „A Secure Subliminal Channel (?)" In: *Advances in Cryptology*. 1986, pp. 33–41 (cit. on p. 41).

[206]Igor Smolyar, Muli Ben-Yehuda, and Dan Tsafrir. „Securing Self-Virtualizing Ethernet Devices". In: *Proc. Usenix Security Symp.* 2015 (cit. on p. 97).

[207]*Snowden: The NSA planted backdoors in Cisco products*. http://www.infoworld.com/article/2608141/internet-privacy/snowden--the-nsa-planted\-backdoors-in-cisco-products.html. Accessed: 02-01-2018. 2014 (cit. on p. 1).

[208]*Software-Defined Networking tools for LXC (LinuX Containers)*. https://github.com/jpetazzo/pipework. Accessed: 18-09-2019 (cit. on p. 94).

[209]John Sonchack, Adam J Aviv, Eric Keller, and Jonathan M Smith. „Enabling Practical Software-defined Networking Security Applications with OFX". In: *Proc. Internet Society Symposium on Network and Distributed System Security (NDSS)*. 2016 (cit. on p. 75).

[210]John Sonchack, Anurag Dubey, Adam J Aviv, Jonathan M Smith, and Eric Keller. „Timing-based reconnaissance and defense in software-defined networks". In: *Proceedings of the 32nd Annual Conference on Computer Security Applications*. ACM. 2016, pp. 89–100 (cit. on p. 1).

[211]Julian Stecklina. „Shrinking the Hypervisor One Subsystem at a Time: A Userspace Packet Switch for Virtual Machines". In: *Proc. ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments (VEE)*. 2014 (cit. on pp. 8, 64, 78, 98).

[212]Brent Stephens, Aditya Akella, and Michael Swift. „Loom: Flexible and Efficient NIC Packet Scheduling". In: *Proc. Usenix Networked Systems Design and Implementation (NSDI)*. 2019, pp. 33–46 (cit. on p. 97).

[213]Ryan Stone. *PCI SR-IOV on FreeBSD*. https://people.freebsd.org/~rstone/BSDCan_SRIOV.pdf. Accessed: 06-01-2019 (cit. on p. 83).

[214]Joe Stringer. *OVS conntrack support*. `https://lwn.net/Articles/652967/`. Accessed: 17-09-2019. 2015 (cit. on p. 67).

[215]*SYNful Knock - A Cisco router implant - Part I*. `https://www.fireeye.com/blog/threat-research/2015/09/synful_knock_-_acis.html`. Accessed: 2017-02-06. 2015 (cit. on p. 1).

[216]T. Koponen et al. „Network Virtualization in Multi-tenant Datacenters“. In: *11th USENIX Symposium on Networked Systems Design and Implementation*. 2014 (cit. on p. 9).

[217]Rashid Tahir et al. „Sneak-Peek: High speed covert channels in data center networks“. In: *Proc. IEEE INFOCOM*. 2016, pp. 1–9 (cit. on p. 41).

[218]*Technical whitepaper for afl-fuzz*. `http://lcamtuf.coredump.cx/afl/technical_details.txt`. Accessed: 02-06-2016 (cit. on p. 67).

[219]The Fast Data Project. *What is the Fast Data Project (FD.io)?* `https://fd.io/about`. Accessed: 05-06-2018. 2017 (cit. on pp. 8, 64, 78, 83).

[220]*The gVisor project*. `https://github.com/google/gvisor`. Accessed: 09-01-2019. 2018 (cit. on pp. 82, 94).

[221]*The Kata Containers project*. `https://katacontainers.io`. Accessed: 09-01-2019. 2018 (cit. on pp. 82, 94).

[222]*The Tale of One Thousand and One DSL Modems*. `https://securelist.com/analysis/publications/57776/the-tale-of-one-thousand-and-one-dsl-modems/`. Accessed: 2017-02-06. 2012 (cit. on p. 1).

[223]Kashyap Thimmaraju, Saad Hermak, Gábor Rétvári, and Stefan Schmid. „MTS: Bringing Multi-Tenancy to Virtual Networking“. In: *Proceedings of the USENIX Annual Technical Conference (ATC)*. USENIX. 2019, pp. 521–536 (cit. on pp. ix, 8, 64, 78).

[224]Kashyap Thimmaraju, Liron Schiff, and Stefan Schmid. „Outsmarting Network Security with SDN Teleportation“. In: *Proceedings of the IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2017, pp. 563–578 (cit. on pp. ix, 1, 65).

[225]Kashyap Thimmaraju, Liron Schiff, and Stefan Schmid. „Preacher: Network Policy Checker for Adversarial Environments“. In: *38th International Symposium on Reliable Distributed Systems (SRDS) 2019*. 2019 (cit. on p. 101).

[226]Kashyap Thimmaraju, Bhargava Shastry, Tobias Fiebig, et al. „Taking Control of SDN-based Cloud Systems via the Data Plane“. In: *Proceedings of the Symposium on SDN Research (SOSR)*. ACM. 2018, pp. 1–15 (cit. on p. ix).

[227]Kashyap Thimmaraju, Bhargava Shastry, Tobias Fiebig, et al. „Taking Control of SDN-based Cloud Systems via the Data Plane“. In: *Proc. ACM Symposium on SDN Research (SOSR)*. 2018 (cit. on pp. 1, 78–80).

[228]Kashyap Thimmaraju et al. „Taking Control of SDN-based Cloud Systems via the Data Plane“. In: *Proc. ACM Symposium on SDN Research (SOSR)*. 2018 (cit. on pp. 37, 38).

[229]Kashyap Thimmaraju, Bhargava Shastry, Tobias Fiebig, et al. „The vAMP Attack: Taking Control of Cloud Systems via the Unified Packet Parser“. In: *Proceedings of ACM Cloud Computing Security Workshop (CCSW)*. ACM. 2017, pp. 11–15 (cit. on pp. ix, 61).

[230]Kashyap Thimmaraju, Gábor Rétvári, and Stefan Schmid. „Virtual Network Isolation: Are We There Yet?“ In: *Proceedings of the Workshop on Security in Softwarized Networks: Prospects and Challenges (SECSON)*. ACM. 2018, pp. 1–7 (cit. on p. ix).

[231]Heidi J. Tretheway et al. „A snapshot of Openstack users’ attitudes and deployments.“ In: *Openstack User Survey* (Apr 2016) (cit. on pp. 6, 66).

[232] Shin-Yeh Tsai and Yiying Zhang. „A Double-Edged Sword: Security Threats and Opportunities in One-Sided Network Communication“. In: *Proc. Usenix Workshop on Hot Topics in Cloud Computing (HotCloud)*. 2019 (cit. on p. 76).

[233] Shin-Yeh Tsai, Mathias Payer, and Yiying Zhang. „Pythia: Remote Oracles for the Masses“. In: *Proc. Usenix Security Symp.* 2019, pp. 693–710 (cit. on p. 76).

[234] Janet Tseng et al. „Accelerating Open vSwitch with Integrated GPU“. In: *Proc. ACM Workshop on Kernel-Bypass Networks*. 2017 (cit. on pp. 8, 64, 78).

[235] Amin Vahdat. *Enter the Andromeda zone - Google Cloud Platform's latest networking stack*. `https://cloudplatform.googleblog.com/2014/04/enter-andromeda-zone-google-cloud-platforms-latest-networking-stack.html`. Accessed: 19-10-2017. 2014 (cit. on pp. 6, 8, 104).

[236] Rick Vanover. *Virtual switching to become enhanced with Cisco and VMware announcement*. `http://www.techrepublic.com/blog/data-center/virtual-switching-to-become-enhanced-with-cisco-and-vmware-announcement`. Accessed: 27-01-2017. 2008 (cit. on pp. 8, 61, 64, 78).

[237] *Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Network*. `https://tools.ietf.org/html/rfc7348`. Accessed: 01-06-2016 (cit. on pp. 8, 97).

[238] VMware. *NSX Controller Installation and Clustering*. `https://docs.vmware.com/en/VMware-NSX-T-Data-Center/2.1/com.vmware.nsxt.install.doc/GUID-447C0417-A37B-4C2E-965E-499F52587160.html`. Accessed: 17-09-2019. 2018 (cit. on p. 63).

[239] VMware. *VMware ESX 4.0 Update 1 Release Notes*. `https://bit.ly/2sFTuTy`. Accessed: 05-06-2018. 2009 (cit. on pp. 8, 61, 64, 77, 78).

[240] Open vSwitch. *Flowgen*. `https://github.com/openvswitch/ovs/blob/branch-2.3/tests/flowgen.pl`. Accessed: 17-09-2019. 2013 (cit. on p. 67).

[241] Haopei Wang, Lei Xu, and Guofei Gu. „FloodGuard: a dos attack prevention extension in software-defined networks“. In: *Proc. IEEE/IFIP Transactions on Dependable and Secure Computing (DSN)*. 2015, pp. 239–250 (cit. on p. 1).

[242] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. „Peeking behind the curtains of serverless platforms“. In: *Usenix Annual Technical Conference (ATC)*. 2018, pp. 133–146 (cit. on p. 102).

[243] *What is Openstack?* `https://www.openstack.org/software`. 2016 (cit. on p. 6).

[244] Hanqian Wu et al. „Network security for virtual machine in cloud computing“. In: *Proc. IEEE Conference on Computer Sciences and Convergence Information Technology*. 2010, pp. 18–21 (cit. on p. 75).

[245] Jiarong Xing, Adam Morrison, and Ang Chen. „NetWarden: Mitigating Network Covert Channels without Performance Loss“. In: *Proc. Usenix Workshop on Hot Topics in Cloud Computing (HotCloud)*. 2019 (cit. on p. 41).

[246] Xin Xu and Bhavesh Davda. „SRVM: Hypervisor Support For Live Migration With Passthrough SR-IOV Network Devices“. In: *Proc. ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments (VEE)*. 2016, pp. 65–77 (cit. on p. 97).

[247] Dongting Yu, Andrew W Moore, Chris Hall, and Ross Anderson. *Security: a Killer App for SDN?* Tech. rep. Indiana Uni. at Bloomington, 2014 (cit. on p. 65).

[248] Yu Zhao. *PCI: Linux kernel SR-IOV support*. `https://lwn.net/Articles/319651/`. Accessed: 06-01-2019. 2009 (cit. on p. 83).

[249]Wenxuan Zhou, Dong (Kevin) Jin, Jason Croft, Matthew Caesar, and Philip Brighten Godfrey. „Enforcing Customizable Consistency Properties in Software-Defined Networks“. In: *Proc. Usenix Networked Systems Design and Implementation (NSDI)*. 2015, pp. 73–85 (cit. on p. 38).

[250]Zhe Zhou, Zhou Li, and Kehuan Zhang. „All Your VMs are Disconnected: Attacking Hardware Virtualized Network“. In: *Proc. ACM Conference on Data and Application Security and Privacy (CODASPY)*. 2017 (cit. on p. 97).

[251]Danyang Zhuo, Kaiyuan Zhang, Yibo Zhu, et al. „Slim: OS Kernel Support for a Low-Overhead Container Overlay Network“. In: *Proc. Usenix Networked Systems Design and Implementation (NSDI)*. Boston, MA: USENIX Association, 2019, pp. 331–344 (cit. on pp. 8, 64, 78, 98).

List of Figures

2.1 Abstract representation of an MTDC network topology and the key components: (i) centralized controller to operate the physical and virtual network infrastructure, (ii) vswitch for network virtualization. 6

2.2 High-level illustration of network virtualization using virtual switches in an MTDC. The vswitch isolates the green and orange tenant networks via tenant-specific flow table entries thereby giving each tenant the illusion of a single network. 7

2.3 High-level architecture of Open vSwitch. 9

4.1 Illustration of teleportation: Malicious switches (with *red horns*) exploit the controller for hidden communication, possibly bypassing data plane security mechanisms such as a firewall. 16

4.2 Message sequence pattern for path update teleportation. Switch $s2$ teleports information to $s1$ when $s1$ receives the Flow-delete message from controller $c0$. 19

4.3 Message sequence pattern for path reset teleportation. Switch $s1$ teleports information to $s2$ and $s3$ via `Flow-add` messages sent by the controller $c0$. . . 20

4.4 Message sequence pattern for switch identification teleportation when the controller denies the second switch a connection. When $s2$'s connection is terminated, $s1$ successfully teleports information to $s2$. 22

4.5 Message sequence pattern for switch identification teleportation when controllers $c1$ and $c2$ send different Role-request messages to $s1$ and $s2$ respectively. When $s1$ receives the `Role-request=Master` message whereas $s2$ receives the `Role-request=Equal` message. In this manner $s1$ teleports information to $s2$ when $s2$ received the `Role-request=Equal` message. 23

4.6 Message sequence pattern for out-of-band forwarding teleportation. The controller $c0$ receives the `Packet-in` from $s1$ and accordingly sends a `Packet-out` to $s2$, successfully teleporting packets from $s1$ to $s2$. 24

4.7 An SDN topology with OpenFlow switches $s1$ and $s2$ and an OpenFlow controller $c0$ (ONOS). $k1$ and $k3$ are connected to $s1$ while $k2$ and $k4$ are connected to $s2$. $s1$ and $s2$ are separated by a firewall $fw1$ that denies hosts on $s1$ to communicate with hosts on $s2$ and vice-versa. $k1$ can use out-of-band forwarding teleportation to transfer data to $k2$, bypassing $fw1$. 25

4.8 An SDN topology of OpenFlow switches $s1$, $s2$, $s3$ and $s4$, OpenFlow controller $c0$ (ONOS). Hosts $k1$ and $k3$ are connected to $s1$ and $k2$ and $k4$ are connected to $s2$. $c0$ has installed flows on $s1$, $s3$ and $s2$ so that $k1$ and $k2$ can communicate bi-directionally. Teleportation traffic is via $c0$. 26

4.9 An SDN topology with independent OpenFlow switches controlled by independent OpenFlow controllers (ONOS). $c1$ and $c2$ share and synchronize state information via an independent controller network. $s1$ is controlled by $c1$ and $s2$ is controlled by $c2$ respectively. 28
4.10 An SDN topology with OpenFlow switches $s1$, $s2$, $s3$ and $s4$ and an OpenFlow controller $c0$ (ONOS). $k1$ and $k3$ are connected to $s1$ while $k2$ and $k4$ are connected to $s2$. Note that $s2$ is not connected to the other switches, and thereby is isolated in the data plane. $k2$ can still exfiltrate data to $k1$ using out-of-band forwarding teleportation circumventing the data plane isolation. . 29
4.11 The message sequence pattern for evading policy conflicts using out-of-band forwarding teleportation. The side effect of `Flow-mod` messages are avoided when Jumbo frames are used from a masqueraded MAC address; only `Packet-ins` and `Packet-outs` are used. 30
4.12 An SDN topology with OpenFlow switches $s1$ and $s2$ with $c0$ the OpenFlow controller (ONOS). $k1$ and $k3$ are connected to $s1$ while $k2$ is connected to $s2$. $fw1$ denies $k2$ to communicate with $k1$ and $k3$ and vice-versa via the data plane. $s1$ and $k2$ being malicious, exploit the out-of-band forwarding teleportation to eavesdrop and modify communication data between $k1$ and $k3$ bypassing $fw1$. 31
4.13 Out-of-Band Forwarding performance characteristics without and with load on the controller. (a) Received throughput, (b) Jitter and (c) Packet loss. 34
4.14 Received throughput using Out-of-Band Forwarding (a) without load (b) with load on the controller. 34
4.15 (a) CPU load using Out-of-Band Forwarding without and with load on the controller. (b) Memory usage using Out-of-Band Forwarding without and with load on the controller. 36
4.16 Scatter plot of the CPU usage on the controller for Out-of-Band Forwarding. (a) Without load on the controller. (b) With load on the controller. Legend as in Fig. 4.13. 36

5.1 Message sequence pattern for the OpenFlow handshake and *switch identification* teleportation when the controller denies the second switch a connection. In this way, s2 can infer a bit value after it gets disconnected. 43
5.2 State machine for the sender to send one binary value. 44
5.3 State machine for the receiver to receive one binary value. 45
5.4 Channel accuracy for time intervals 30-100 ms, and frame lengths 7, 14 and 28 when $\delta_{offset} = 5ms$, OpenFlow status is checked at $\Delta/2$, with no load and with load on the controller. 53
5.5 Missed start bit errors for time intervals 30-100 ms, and frame lengths 7, 14 and 28 when $\delta_{offset} = 5ms$, OpenFlow status is checked at $\Delta/2$, and there is no load on the controller. 53
5.6 Channel accuracy for time intervals 30-100 ms, and frame lengths 7, 14 and 28 when $\delta_{offset} = 5ms$, OpenFlow status is checked at $2\Delta/3$, with no load and with load on the controller. 54
5.7 Channel accuracy for time intervals 30-100 ms, and frame lengths 7, 14 and 28 when $\delta_{offset} = 5ms$, OpenFlow status is checked at $\Delta/3$, with no load and with load on the controller. 54

5.8 End of message errors and Missed start bit errors for time intervals 30-100 ms, and frame lengths 7, 14 and 28 when $\delta_{offset} = 5ms$, OpenFlow status is checked at $\Delta/3$, with no load. 54
5.9 Channel accuracy for message lengths 64, 512 and 1024 for time intervals $30 - 100$ ms, and frame length 7, when OpenFlow status is checked at $2Delta/3$, with no load and with load on the controller. 55

6.1 The total number of parsed network protocols in two popular virtual switches (OvS and Cisco's Nexus 1000V) and OpenFlow from 2009-2019. 62
6.2 An overview of the security implications of current multi-tenant network virtualization architectures that use virtual switches. An attacker can exploit Host OS co-location, centralized and virtualized control, and complex packet processing (in the Unified packet parser) of untrusted data to launch an attack from a VM on the virtualization layer (①). From there, the attacker can propagate to the controller node (②) and then compromise other servers in the cloud (③). . . 63
6.3 MPLS label stacks are placed in the Shim header between the Ethernet and IP headers. 68
6.4 Forwarding latency of OvS with and without countermeasures on a vanilla kernel and `grsecurity` enabled kernel in the slow and fast path. 74
6.5 Forwarding throughput of OvS with and without countermeasures on a vanilla kernel and `grsecurity` enabled kernel in the slow and fast path. 75

7.1 A high-level view depicting the security-performance-resource tradeoffs for the state-of-the-art and `MTS` network virtualization architectures. 79
7.2 High-level overview of `MTS` in security Level-2. The Red and Blue vswitch compartments (VMs) are allocated dedicated virtual functions (VFs) to communicate with external networks using the *In/Out VF*, their respective tenants using the *Gw VF* and *T VF*. Communication between the vswitches, tenants and the Host physical function (PF) are mediated via the SR-IOV NIC switch. . . . 83
7.3 A step-by-step illustration of how packets enter and leave the Red tenant from Figure 7.2 in `MTS`. ⓐ shows how ingress packets reach Tenant$_{Red}$. ⓑ shows how Tenant$_{Red}$ packets reach an external system Tenant$_{Ext}$. 84
7.4 Three canonical traffic scenarios evaluated: p2p, p2v and v2v 88
7.5 The security, throughput, latency and resource tradeoff comparison of `MTS`. The rows indicate the resource mode. The columns are ordered as throughput, latency and resources. The security levels used are shown in the legend. Note the bottom row is for security Level-3 in the isolated resource mode combined with other security levels. 89
7.6 Iperf throughput, Apache and Memcached throughput and latency (shown in the columns) comparison of `MTS`. The rows indicate the resource mode where the bottom row is for security Level-3 in the isolated resource mode combined with the other security levels. The legend is the same as in Figure 7.5. 92
7.7 Forwarding throughput comparison of `MTS` with four tenants in the p2p and p2v scenarios. The vswitch runs in VMs (gray bars) or in containers in VMs (green bars). 95
7.8 Forwarding throughput comparison of scaling `MTS` using containers in multiple VMs in the p2p and p2v scenarios. Note that the number of containers is equal to the number of tenants and that they are evenly distributed across all the VMs. 96

7.9 Forwarding throughput of MTS with 16 tenants and maximum number of supported vswitch containers in VMs in the p2v scenario. 96

List of Tables

1.1 List of assigned CVEs presented in this dissertation 3

2.1 List of virtual switches surveyed in this dissertation. MTS is introduced in this dissertation. 8

4.1 Summary of teleportation techniques, types, messages and associated threats. 22
4.2 Summary of teleportation attacks and involved entities. 39

6.1 Design characteristics of virtual switches surveyed in this dissertation. MTS is presented in this dissertation. 64

7.1 Design characteristics of surveyed virtual switches in this dissertation. MTS is presented in this chapter. 78