Fault Isolation in Software Defined Networks

João Sales Henriques Miranda

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor:
Prof. Luís Eduardo Teixeira Rodrigues

Examination Committee

Chairperson:	Prof. João António Madeiras Pereira
Supervisor:	Prof. Luís Eduardo Teixeira Rodrigues
Member of the Committee:	Prof. Fernando Manuel Valente Ramos

November 2016

Acknowledgements

First I want to thank my supervisor, Professor Luís Rodrigues, and Nuno Machado for their direct influence in this work, being always available to guide me throughout this past year (and performing this job extremely well!). I am also very grateful to all professors and teachers that contributed to my education until now.

All my friends and my two brothers also deserve a place in the spotlight for helping me to transform the college years into a funny journey. And last but not least, my parents, at the center of this spotlight, for their continuos support in all levels, throughout my whole life.

This work was supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) with reference UID/CEC/50021/2013. VeriFlow software was developed in the Department of Computer Science at the University of Illinois at Urbana-Champaign.

Lisboa, November 2016

João Sales Henriques Miranda

For my family, friends, professors
and teachers,

Resumo

As redes definidas por software (do Inglês, *Software Defined Networks*, SDNs) têm vindo a afirmar-se como uma das mais promissoras abordagens para simplificar a configuração e gestão de equipamentos. No entanto, as SDNs não são imunes a erros tais como ciclos no encaminhamento, buracos negros, encaminhamento sub-óptimo, entre outros. Estes erros são tipicamente causados por falhas na especificação ou por erros nos equipamentos. Se as primeiras podem ser, em grande parte, eliminadas através da utilização de ferramentas que fazem a validação automática de uma especificação antes da sua instalação, os erros (e/ou avarias) nos equipamentos (muitas vezes de natureza não determinista) geralmente só conseguem ser detectados em tempo de execução.

Esta dissertação propõe uma nova técnica para facilitar o isolamento de falhas nos equipamentos em redes SDN. Esta técnica combina a utilização de ferramentas de validação formal (para obter os caminhos esperados para os pacotes) e ferramentas de registo de pacotes (para obter os caminhos observados) para realizar uma análise diferencial que permite identificar com forte precisão qual o equipamento onde ocorreu a falha que causou a desconfiguração da rede. Construímos um protótipo desta ferramenta e avaliámo-lo no MiniNet. Os nossos resultados mostram que o nosso sistema é capaz de apontar o comutador em falha ou, no pior caso, pares de comutadores em que um deles está em falha, e que o sistema consegue também categorizar o erro da rede dentro de cinco tipos diferentes de erro.

Abstract

Software Defined Networking (SDN) has been emerging as one of the most promising approaches to simplify network configuration and management. However, SDNs are not immune to errors such as forwarding loops, black holes, suboptimal routing and access control violations. These errors are typically caused by errors in the specification or by bugs in the equipment. While the former may be, mostly, eliminated by using tools that automatically validate specifications before their installation, firmware or hardware bugs in the switches (many times of non deterministic nature) can only be detected in execution time, in most cases.

This dissertation proposes a new technique to facilitate the fault isolation in SDN equipments. The described technique combines the usage of formal validation tools (to obtain the expected paths of the packets) and packet recording tools (to obtain the observed paths) to perform a differential analysis that allows the precise identification of which equipment had failed, causing the network misconfiguration. We built a prototype and evaluated it on MiniNet. Our results show that our system is able to pinpoint either the faulty switch or, in the worst case, pairs of switches in which one is the faulty, and that it can also categorize the error within five different error types.

Palavras Chave
Keywords

Palavras Chave

Redes Definidas por Software

Fiabilidade

Monitorização

Diagnóstico de redes

Correção de erros

Depuração

Keywords

Software Defined Networking

Reliability

Monitoring

Network diagnostics

Troubleshooting

Debugging

Contents

List of Figures

List of Tables

Acronyms

ACL Access Control List

API Application Programming Interface

ARP Address Resolution Protocol

ATPG Automatic Test Packet Generation

EC Equivalence Class

ECMP Equal Cost Multi Path

HSA Header Space Analysis

IP Internet Protocol

FTSR Flow Table State Recorder

MAC Media Access Control

MCS Minimal Causal Sequence

SDN Software Defined Network(ing)

SOFT Systematic OpenFlow Testing

STS SDN Troubleshooting System

TCP Transmission Control Protocol

UDP User Datagram Protocol

VM Virtual Machine

1 Introduction

Software Defined Networking (SDN) is a novel paradigm for managing computer networks. It provides a clear separation between the data plane (in charge of forwarding packets among devices) and the control plane (which defines how packets should be forwarded). Furthermore, it defines an architecture that allows the control functions on each device to be configured by a logically centralized controller, through a standardized interface. This architecture has the potential to strongly simplify network management (Scott, Wundsam, Raghavan, Panda, Or, Lai, Huang, Liu, El-Hassany, Whitlock, Acharya, Zarifis, and Shenker 2014; Khurshid, Zou, Zhou, Caesar, and Godfrey 2013; Ball, Bjørner, Gember, Itzhaky, Karbyshev, Sagiv, Schapira, and Valadarsky 2014; McKeown 2011; Shenker 2011), and has been adopted by reference companies in the area of computer networking such as Google, Microsoft and AT&T (Jain, Kumar, Mandal, Ong, Poutievski, Singh, Venkata, Wanderer, Zhou, Zhu, Zolla, Hölzle, Stuart, and Vahdat 2013; TechTarget 2016; Shenker 2011).

This thesis addresses the problem of identifying misbehaving switches in Software Defined Networking (SDN). Particularly, we aim to find inconsistencies between the states of the devices, configured by the controller, and their respective behavior observed at hardware level, and identify the switch were we found these inconsistencies. Our system, called NetSheriff, compares the expected paths of packets with their respective observed paths and performs a differential analysis in order to identify the misbehaving switches. We implemented a prototype that leverages outputs resulting from one system that verify network invariants in execution time of the controller program (VeriFlow (Khurshid, Zou, Zhou, Caesar, and Godfrey 2013)) and one system that records information of packets as they traverse the network (NetSight (Handigol, Heller, Jeyakumar, Mazières, and McKeown 2014)), to produce the expected paths and the observed paths, respectively.

1.1 Motivation

One of the improvements that SDN has brought is the possibility of simplifying network testing, verification and debugging. This happens because it is possible to leverage SDNs' architecture by intercepting commands issued by controllers in order to create models of the network configuration or instrument switches to apply record and replay techniques for these purposes.

There are two main sources of errors in a SDN. One consists of faulty controller programs, that may generate network misconfigurations such as routing loops or black holes when deployed. A significant amount of research has been performed in recent years in tools that aim at detecting and helping in correcting this type of faults. Tools such as NICE (Canini, Venzano, Perešíni, Kostić, and Rexford 2012) or VeriFlow (Khurshid, Zou, Zhou, Caesar, and Godfrey 2013) can analyse a controller program by manipulating models of network configurations, producing possible configurations and verifying that a number of target invariants are not violated under these configurations. The other source of errors are faults in the software that runs on the networking devices or in the hardware itself. Unlike the former faults, these are often unpredictable and occur at run-time, and thus cannot always be detected through test or verification. To help network managers to detect and to correct these faults, tools such as OFRewind (Wundsam, Levin, Seetharaman, and Feldmann 2011) or ndb (Handigol, Heller, Jeyakumar, Maziéres, and McKeown 2012) typically resort to instrumentation, logging of events, and replay mechanisms. Although, despite these advances in debugging SDNs, it is still generally a daunting and time consuming task, as it will become clear later in the text. With this in mind, we defined the goal of this work, which is to create a mechanism that helps network administrators to isolate error causes by automatically identifying the device responsible by a network error.

1.2 Contributions

To achieve the goal we just mentioned, we leverage techniques used in testing and verification tools as well techniques used in debugging tools, combining them in order to produce a differential analysis between expected paths of packets and the respective observed paths. By doing so, this work brings the following contributions.

- A system that automatically identifies misbehaving switches in a SDN without any modification to its components, helping network administrators to detect and troubleshoot network errors that happen in production;
- An evaluation of the system that shows its efficacy in detecting misbehaving switches and classifying forwarding errors into categories that can further help narrowing down the causes of such error.

1.3 Results

This work has produced the following results.

- A description of an architecture that is able to achieve our goal.
- A prototype of this system, where some components where adapted from other systems.
- An experimental evaluation of this prototype in an emulated environment showing that it can effectively identify multiple types of errors and make distinctions among them, automatically.

1.4 Research History

This work benefited from the fruitful collaboration of Nuno Machado, who was always available for listening and giving ideas, during all stages of the project.

A paper that describes part of this work was published and rewarded as best paper in Actas do Oitavo Simpósio de Informática, Inforum 2016 (Miranda, Machado, and Rodrigues 2016).

1.5 Structure of the Document

The rest of this document is organized as follows. For self-containment, Chapter 2 provides an introduction to SDN and testing, verification and debugging tools. Chapter 3 describes the architecture and the algorithms used by our system, which we call NetSheriff. Then, Chapter 4 presents the results of the experimental evaluation study. Finally, Chapter 5 concludes this document by summarizing its main points and future work.

2 Related Work

Our approach to achieve the main goal, which is to automatically identify the switch responsible for an error in an SDN, is based on ideas presented in previous work in the areas of testing, verification, and debugging of SDNs. Testing and verification approaches try to validate SDN control software in respect to previously specified target invariants, aiming to find problems (and possibly their causes) proactively. Conversely, debugging approaches are tailored to help finding the cause of problems after they are discovered.

Before overviewing the work in areas of testing, verification and debugging, this chapter introduces the concept of Software Defined Networking (SDN) (Section 2.1). Then we will present OpenFlow which is one of the existing Application Programming Interfaces (APIs) to manage switches in SDN (Section 2.2). After that, we briefly describe common types of errors that can appear in an SDN and their causes (Section 2.3). Finally, we conclude the chapter by presenting a set of tools that help the SDN development cycle, including testing, verification and debugging tools (2.4).

2.1 Software Defined Networking

Traditional networks are based on specialized components bundled together in the networking devices. Examples of these devices are routers, switches, firewalls or Intrusion Detection Systems (IDS). Each of these devices has a specific role and requires individual configuration (that is vendor specific). A network can have multiple devices that, despite being configured independently, are expected to cooperate to achieve a common goal. To ensure a consistent configuration across all devices in a traditional network can be extremely challenging (Scott, Wundsam, Raghavan, Panda, Or, Lai, Huang, Liu, El-Hassany, Whitlock, Acharya, Zarifis, and Shenker 2014; Khurshid, Zou, Zhou, Caesar, and Godfrey 2013; Ball, Bjørner, Gember, Itzhaky, Karbyshev, Sagiv, Schapira, and Valadarsky 2014; McKeown 2011).

Furthermore, traditional network protocols, in particular routing protocols, have been designed to be highly decentralized and to be able to operate without a unique point of failure. This decentralization has advantages but also makes hard to reason about the behavior of the system in face of dynamic changes to the workload, re-configurations, faults, and asynchrony. Finally, traditional networking components

are *vertically integrated*, i.e., they combine data forwarding functions with control functions in a way that makes hard to analyze each of these aspects in isolation. In addition to the complexity referred above, this vertical integration leads to high upgrade cost (because updating software implies buying new and expensive hardware) and very slow pace of innovation due to larger hardware development cycles (when compared to software development cycles) (Ball, Bjørner, Gember, Itzhaky, Karbyshev, Sagiv, Schapira, and Valadarsky 2014; McKeown 2011).

A key step to solve the complexity problem identified above is to break the vertical integration. This can be achieved by splitting the network functionality in two main planes: the control plane and the data plane. The control plane is responsible for deciding how traffic should be handled, in other words, it is in charge of populating the devices' forwarding tables. The data plane is responsible for forwarding the traffic according to the rules installed by the control plane.

Software Defined Networking (SDN) is an emerging paradigm that makes this separation possible. Given the strong potential of SDN to simplify network management, the paradigm has gain significant support and adoption from many large companies such as Google, AT&T and Yahoo. In an SDN network, the network devices can be simplified, becoming simple packet forward elements with a common open interface to upper layers (e.g. OpenFlow), and all the control logic is pulled to a logically centralized point of the network. Software Defined Networking is defined by three main abstractions (Shenker 2011): the *forwarding abstraction*, the *distribution abstraction*, and the *configuration abstraction*. To illustrate these abstractions and the SDN planes we use Figure 2.1 that is extracted from (Kreutz, Ramos, Verissimo, Esteve Rothenberg, Azodolmolky, and Uhlig 2014). The interfaces used and offered by the control plane are referred, respectively, as southbound API and northbound API.

- The goal of the *forwarding abstraction* is to hide vendor specific details of each individual equipment from the upper abstractions of the SDN architecture. This can be achieved by having all devices exporting a common and open API (a southbound API) to be used by the upper layer, the Network Operating System. OpenFlow is a typical example of this API.
- The goal of the *distribution abstraction* is to hide from the upper layer the number and locations of SDN controllers, giving the illusion that a logically centralized controller is in charge of implementing the control plane. The choice of whether or not make it distributed through multiple controllers (and how to distribute it) in order to meet specific requirements (e.g. scalability or fault tolerance) is up to the programmer instead of an inherent characteristic of a network.
- Finally, the goal of the *configuration abstraction* (also refered as specification abstraction, or network abstraction in the figure) is to simplify network configuration by converting the global network view that is managed by the controller into an abstract model. By doing so, the control program should be able to specify its desired behavior in an abstract network topology.

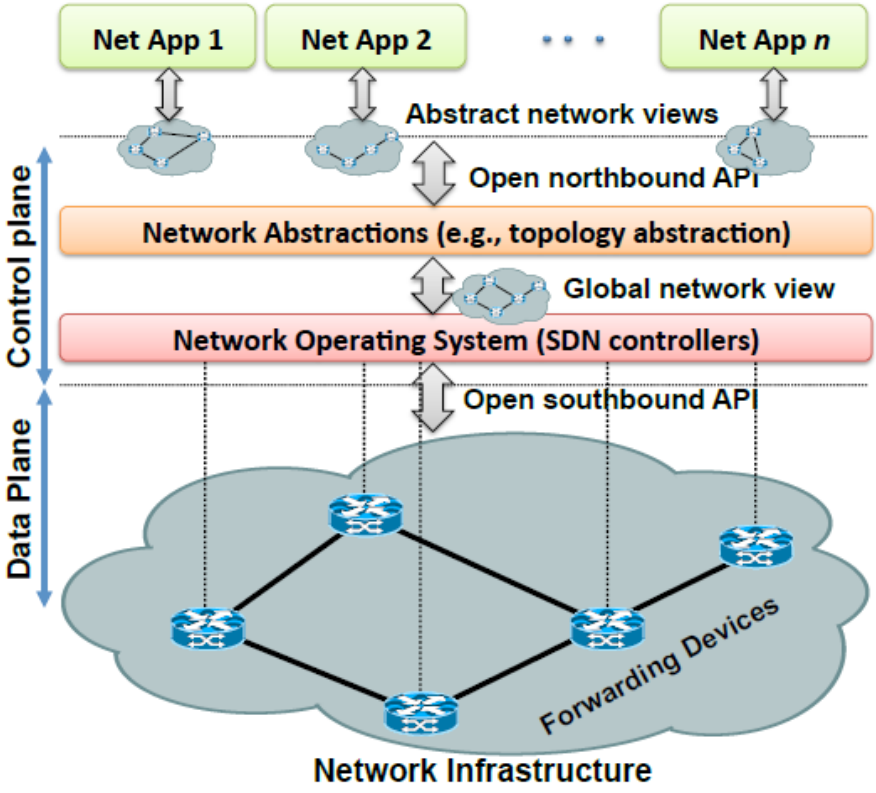

Figure 2.1: SDN architecture and its fundamental abstractions (picture taken from (Kreutz, Ramos, Verissimo, Esteve Rothenberg, Azodolmolky, and Uhlig 2014))

2.2 OpenFlow

OpenFlow is one of the most used southbound APIs (see Figure 2.1). As stated in (McKeown, Anderson, Balakrishnan, Parulkar, Peterson, Rexford, Shenker, and Turner 2008), when the OpenFlow idea was developed, the main goal was to allow researchers to experiment new ideas for network protocols in existing network equipment (typically from different vendors) by providing a standardized interface to manipulate the devices' forwarding tables. Vendors could provide this interface without exposing the internal working of their devices. This fact is relevant because vendors would not easily open their algorithms and protocols since just by opening them they would lower the barrier to entry for new competitors, for example.

Not only the goal previously described was achieved, and SDN has emerged as a very useful research aid, but it has also emerged in production networks as well, as it was mentioned in the previous section.

An OpenFlow switch is a switch that can be managed by a remote controller through the OpenFlow

protocol. First we will briefly describe how the data plane functionality is achieved and after that we introduce the OpenFlow protocol.

The data plane functionality is implemented through a set of flow tables and a group table. Each entry in a flow table has a matching rule and a set of actions. Each entry in a group table has a group identifier and multiple sets of actions. Examples of actions are forwarding the packet to a given output, or modify some header field. When a packet reaches the switch, it is matched against the entries of the first flow table. If there is a match, the corresponding actions are applied, and the packet may continue being matched to other flow tables (also specified in the entry). One of the possible actions specifies that the packet should be processed through a specified group in the group table. In this case, sets of actions are selectively applied, depending on the group type of the table entry. *e.g.,* multicasts are implemented by applying all sets of actions, where each of these sets will modify the header fields accordingly so that the packet eventually reaches the desired destination, and forward the modified packet to the respective port. If at some point the packet does not match any entry of the flow table being matched, a default action is applied. Typically this default action is either (1) sending this packet to the controller, asking what actions to apply, or (2) dropping the packet.

In order to be managed remotely by a controller, an OpenFlow switch must implement a a secure channel connecting itself to the controller, and also implement the OpenFlow protocol. This channel has the purpose of exchanging OpenFlow messages, *i.e.,* messages that belong to the OpenFlow protocol. This protocol specifies a number of events that the switches should report (*e.g.,* modification of the status of a port, or receiving a packet that does not match any rule) and commands that the controller can issue to the switch. The protocol specifies how the switch should react to these commands (*e.g.,* some commands specify that the switch should add a new rule to a forwarding table). Figure 2.2 (taken from (McKeown, Anderson, Balakrishnan, Parulkar, Peterson, Rexford, Shenker, and Turner 2008)) represents this model of an OpenFlow switch. These switches may be either dedicated OpenFlow switches or OpenFlow-enabled switches. We now describe these two types of switch.

A *Dedicated OpenFlow switch* is a dumb device that has only data link layer functionality implemented, i.e. it routes packets according to a *flow table*. Each entry of this table is a pair (rule, action). If a packet matches the rule, the respective action will be applied to it. Examples of possible actions are described below. The flow table is populated by means of commands sent to switches from a *remote controller*, that implements the functionality of the above layers of the network protocols stack. Communication between the controller and the switch is done by a secure channel, using the OpenFlow Protocol. The required actions in a dedicated OpenFlow switch are:

- drop the packet, *e.g.,* for security reasons;
- forward the packet to a given port. Particularly, there is one specific logical port (required by the

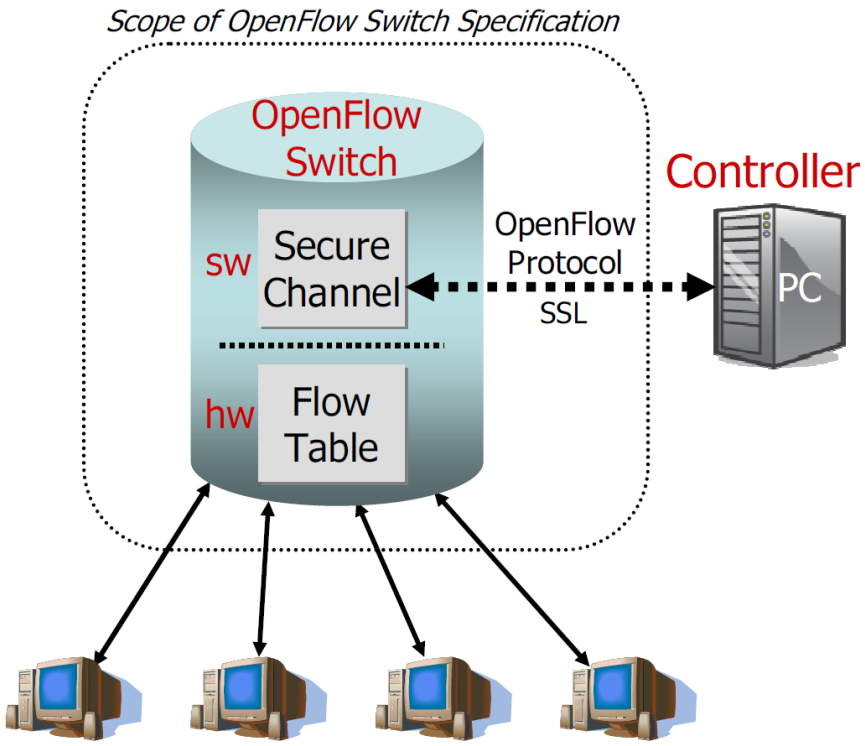

Figure 2.2: Idealized OpenFlow switch (figure taken from (McKeown, Anderson, Balakrishnan, Parulkar, Peterson, Rexford, Shenker, and Turner 2008)). For simplicity, only one flow table is represented.

OpenFlow specification) that is used to encapsulate the packet and forward it to the controller as an OpenFlow message. This is typically used as the default action, when a packet does not match any rule and corresponds to asking the controller where the packet should be sent to;

- process the packet through a specified group of the group table.

An *OpenFlow-enabled switch* is a regular switch that has extended functionality to implement the behavior of a dedicated OpenFlow switch, but maintaining the upper layers of network protocols stacks (and thus being smarter devices). These switches should provide the additional functionality without losing the previous ones. This allows traffic from a production network to be routed using standard and tested protocols, and research traffic to be routed with experimental protocols. In order to achieve this, switches should have an additional action:

- forward packet to the normal processing pipeline (layers above data link layer in the network protocols stack). This is also achieved by the action previously described as “forward to a given port”, using a specific logic port. However, this is an optional port in the OpenFlow switch specification, since not all switches must implement it.

The actions that were briefly described here are detailed in the OpenFlow switch specification (Consortium et al. 2009), along with the OpenFlow protocol. We now describe the most relevant types of messages of the protocol:

- *packet_in*: sent from switches to the controller when a received packet that does not match any rule, or the matched rule specifies an output action to the controller;
- *packet_out*: sent from the controller to switches when the controller has to send a packet through the data plane;
- *flow_mod*: sent from the controller to switches. Used to update a flow table entry (add, modify or delete);
- *port_status*: sent from switches to controller to inform about a port that was added, modified or removed.

By using these messages, the controller can be programmed to react to switch events by performing changes in its own state and responding with commands that manipulate their flow tables, or with individual packets that will be sent through the network.

2.3 Common errors in SDN and their causes

We list below the main errors that can affect the operation of SDN networks. These errors can be caused by a combination of misconfigurations, software, and hardware faults. While misconfigurations may potentially be avoided by using offline validation tools, that check the correctness of the configurations before they are applied, software and hardware faults may occur in run-time and the resulting errors require online techniques to be detected.

- **Forwarding loops:** are always undesirable because it means that packets are traveling around and will not reach any destination. There are some situations, though, that networks might tolerate loops during a state transition.
- **Black holes:** are bugs that lead to packet loss because the packets do not match any rule.
- **Access control violations:** happens when a host can connect to another host in a separate VLAN.
- **Anomalous forwarding:** these errors are characterized by the fact that packets are not being correctly forwarded according to the specification. This might occur, for example, because a forwarding rule is not correctly installed. A special case of this error is one called suboptimal routing.

It happens when packets are reaching their destination, but being incorrectly forwarded, traversing different paths than the expected ones.

While these errors are not exclusive of SDN, we can leverage the layered architecture of SDN to identify the causes of errors. Figure 2.3 further divides the SDN planes mentioned in Section 2.1 into layers of the SDN control stack and presents examples of bugs that can make each pair of layers incoherent. The figure is borrowed from (Heller, Scott, McKeown, Shenker, Wundsam, Zeng, Whitlock, Jeyakumar, Handigol, McCauley, Zarifis, and Kazemian 2013), which describes how one can use multiple tools to systematically find the layer of a bug and then localize it within the layer. Each state layer of the SDN control stack fully specifies network behavior. In a correct network, every layer is always correctly mapped to every other layer. Most errors in the SDN control stack result in mistranslations between these layers. If this is not the case and all layers match while there is an error, either the policy or the hardware is broken.

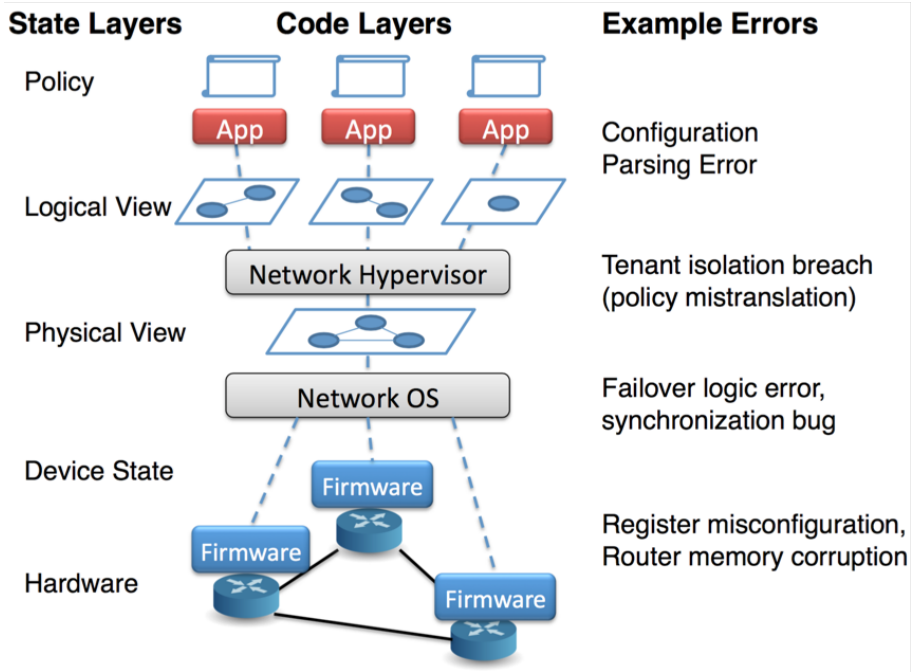

Figure 2.3: SDN stack and examples of errors that cause mismatch between the layers (picture taken from (Heller, Scott, McKeown, Shenker, Wundsam, Zeng, Whitlock, Jeyakumar, Handigol, McCauley, Zarifis, and Kazemian 2013))

2.4 Helper tools for SDN development

Prototyping, testing, verification, and debugging play important roles in software development, and this is no different in the development in SDN. These activities are so important and recurrent that there are multiple tools available to automate parts of them. Examples of these tools will be described in detail in Sections 2.4.1 to 2.4.3. The remainder of this section will briefly introduce each of these activities.

Prototypes are faster and cheaper ways to validate ideas than full implementations. To evaluate SDN program prototypes, it would be better in many situations to use an emulated environment rather than a real network, because it is faster to configure software in a single computer than setup hundreds of real devices, and also because these devices might be running production software that operators will probably not like to stop. Note that using emulated networks is useful for evaluating prototypes of network applications but also prototypes of testing tools, online verification tools and debugging tools. Emulated networks are also useful for performing the tests, verifications and debugging themselves, for the same reasons mentioned above. Consider, for example, that a bug is reported in production: if the network is still operational, debugging could be performed in a separate (emulated) network without having to shutdown the production network or use additional network devices. Section 2.4.1 will present an example of such emulation tool.

Testing is unavoidable, since there is no way to know if a system works as expected without testing it. Testing can target functional requirements or system qualities (such as performance or availability). Despite being helpful, typically testing cannot guarantee the absence of bugs (even if it is exhaustive). When software testing is not enough, it is possible to apply verification, *i.e.,* verify if the implementation satisfies the specification. In the particular case of SDN, each translation of state layers of the SDN control stack is implemented by a code layer. The specification states that every layer must always map to every other layer. This is particularly relevant for testing and verification tools, because each state layer of the SDN control stack (Figure 2.3) fully specifies network behavior, and having a logically centralized controller facilitates the observation of these layers. Thus, testing and verification tools end up searching for mistranslations between distinct layers automatically, in order to locate the code layers responsible for errors, *i.e.,* the code layer that performs this wrong translation.

Section 2.4.2 will present correctness testing and verification of SDN control programs. Testing and verification can be performed either in offline mode or in online mode. Offline tools check target network invariants before deployment, in a set of possible topologies, or in all admissible topologies. Online tools perform this check in run time, before each forwarding rule is updated (added, modified or deleted).

Testing/Verification and debugging are closely related since the former are means to find software failures (also called bugs, defects, problems, errors, and other terms) and the latter is the process of finding the cause(s) of these failures in order to fix them. Debugging happens during development and

during production, since deploying software without bugs is, in most cases, an utopic goal, specially when software starts growing in complexity. Section 2.4.3 will present examples of tools that aid the debugging activity by recording, processing and replaying traces of events and packet flows in an SDN network.

2.4.1 Emulation

Emulation plays an important role in SDN programs' development. This is also true for developing tools that aid in testing and debugging. An example where emulation shows its importance in debugging is the ability to replay a buggy trace in an emulated network, maintaining live the (partially) functioning production network. The replay debugging process can be executed without interfering with the production network. As for the tools development, or more generically, the SDN programs' development, emulation is important because it might allow for faster prototyping iterations. Because configuring an emulated environment is faster than configuring a physical one, especially when it has multiple nodes.

MiniNet (Lantz, Heller, and McKeown 2010) was designed aiming to enable a *prototyping workflow* that is flexible (adding new functionality and changing among network topologies should be simple), deployable and realistic (there should be no need to change a functional prototype that runs on an emulated network to run it on a physical network), interactive (network should be managed and ran in an interactive manner just like a physical network), scalable (it should allow to emulate networks with thousands of nodes), and shareable (collaboration should be easy so that other developers could use one's experiments and modify them).

Previously available solutions are not affordable by most developers or not realistic. A network of Virtual Machines (VMs) would meet almost all previously mentioned goals, but since using system virtualization for every switch and host uses significant resources, this approach becomes too heavy (does not scale). MiniNet emulates the whole network instead, sharing resources between virtualized nodes to achieve a scalable solution. To accomplish this, MiniNet uses software OpenFlow switches, and operating-system-level virtualization to virtualize the hosts. Controllers can be placed anywhere as long as the machine where the switches are running has IP-level connectivity to them.

Each host is a shell process placed in a different linux network namespace, which is a container for network state. A network namespace has its own exclusive set of network interfaces and routing tables. With this approach, different hosts could be running servers listening to `eth0` interface, both on the same port, because these interfaces are private to each network namespace. However, resources that do not need to be virtualized (like the filesystem) are shared among hosts, thus massively reducing the overhead when compared to virtualizing a full machine for each host. The links between nodes are realized through virtual Ethernet pairs, which act like wires connecting two virtual interfaces.

Network devices and the topology they form can be managed through a command line interface and python scripts. In order to interact with hosts through a command line interface, the process that runs it is connected to the host processes using pipes.

MiniNet has helped the development of many useful prototypes, as reported in (Lantz, Heller, and McKeown 2010). Benchmarks performed by its authors show that a single laptop can be used to emulate networks with thousands of hosts, and even in these largest topologies each host or switch starts in less than one second each, which is reasonable, and faster than booting hardware devices. The software under test can be deployed on real networks without modification and the network can be shared with others using a single VM image, facilitating peer collaboration. It has been also extended to improve aspects such as device isolation and monitoring (Handigol, Heller, Jeyakumar, Lantz, and McKeown 2012). Currently, MiniNet cannot exceed the resources of a single machine, *i.e.,* we cannot use multiple machines to distribute the load of emulating a huge network. However, this is a limitation of the implementation and not of the design.

2.4.2 Testing and Verification of Software Defined Networks

Ideally, one would like to deploy a network that would not present any failures in production. Unfortunately, this is an almost utopian goal to achieve in practice, due to the enormous state space of SDN applications. In addition to the state of the controller program, one has also to take into account the state of the devices encompassed by the network, namely the switches and the end hosts. As such, the state space of an SDN application can grow along three dimensions: *i)* space of input packets (SDN applications must be able to handle a large variety of possible packets), *ii)* space of switch and end hosts states (e.g. switches have their own state that includes packet-matching rules, as well as counters and timers), and *iii)* space of network event orderings (e.g. events such as packet arrivals and topology changes that can occur in the network in a non-deterministic fashion).

Despite the challenge of dealing with the growth of the state space, testing and verifying SDNs' correctness is of paramount importance to reduce the likelihood of failures after deployment. In fact, performing tests and verification allow not only to improve the reliability of the SDN, but also help developers to fix the underlying bug in a more timely manner, by providing concrete traces of packets that trigger errors. Therefore, different tools have been recently developed, targeting different types of errors and therefore approaching this state explosion issue in different ways.

Recall that testing can refer to multiple properties such as correctness, performance, availability or security. We now overview some tools that aim at testing SDN control software for correctness. Essentially, these tools check the consistency between layers of state layers of the SDN control stack (represented in Figure 2.3).

2.4.2.1 Static checking

Part of the testing and verification tools operate prior to the deployment of the network. These are called *static checking* or *offline checking* tools. We now describe some of the recent work in this area.

2.4.2.1.1 NICE

NICE (Canini, Venzano, Perešíni, Kostić, and Rexford 2012) has the goal of finding errors in SDN applications via automatic testing. As input, it receives a controller program, a network topology (with switches and end hosts) and a specification of correctness properties for this network.

NICE views the network (i.e. the devices and the controller program) as a unique system, and applies model checking to systematically explore the possible states of this system while checking correctness properties in each state. The state of the network is modeled as the union of the individual states of its *components*. In turn, the state of a component is represented by an assignment of values to a given set of variables. Components change their state via *transitions* (e.g. send or receive a message) and, at any given state, each component keeps a list of its possible transitions. A system execution is, thus, a sequence of transitions among component states.

For each state, NICE checks if a given correctness property is violated. If there is such violation, the tool can, at this point, output the sequence of transitions that triggers the violation. Otherwise, if no such sequence is found, the search ends without finding a bug and the system passes the test.

In terms of scalability issues, NICE addresses the state space of input packets by applying symbolic execution to identify relevant inputs in the controller program. Symbolic execution allows executing a program with some variables marked as symbolic, meaning that they can have any value. In a branching point, the symbolic execution engine will assign these variables with values that will allow the program to explore both branch outcomes and exercise multiple code paths.

Since the controller program consists of a set of packet-arrival handlers, NICE leverages symbolic execution to find equivalence classes of packets (i.e. sets of packets that exercise the same code path in the controller program). In this way, NICE can model check different network behaviors by simply adding a state transition that injects a representative packet from each class in the network.

NICE copes with the state space of the switches and end hosts by using simplified models for these components and the transitions between them. The simplified models ignore unnecessary details of these devices, therefore reducing their state space.

Finally, to efficiently search the space of network event orderings, NICE employs a number of domain-specific heuristics that favor the exploration of event interleavings that are more likely to expose bugs. For instance, one of the heuristics focuses on discovering race conditions by installing rules in

switches with unusual or unexpected delays. In turn, another heuristic imposes a bound on the number of packets sent by end hosts, with the goal of reducing the search space of possible transitions.

As a remark, to specify these correctness properties, the programmer can either use NICE's library of common correctness properties or write their custom properties. *NoForwardingLoops* and *NoBlackHoles* are two examples of properties already offered by NICE.

2.4.2.1.2 VeriCon

VeriCon (Ball, Bjørner, Gember, Itzhaky, Karbyshev, Sagiv, Schapira, and Valadarsky 2014) verifies SDN programs at compile time, validating their correctness not only for any admissible topology, but also for all possible (infinite) sequences of network events. VeriCon has the advantage of providing the guarantee that a given SDN program is indeed free of errors. This contrasts to NICE (and other finite state model-checking tools), which are able to identify bugs, but not their absence. Furthermore, VeriCon scales better than NICE to large networks because it encodes the network and the correctness conditions as first-order logic formulas, which can be efficiently solved by theorem solvers.

VeriCon receives as input: an SDN program and first-order logic formulas describing both the constraints on the network topology and the correctness condition (expressed as network invariants). To help defining these formulas and write easily-verifiable SDN programs, the authors of VeriCon designed a simple imperative language called *Core SDN* (CSDN). CSDN programs operate on a single kind of data structures – relations. Relations in CSDN represent the state of the network, by modeling the network topology, switch flow tables and the SDN program internal state. CSDN commands are then used to query and manipulate relations (by adding or removing tuples from them) as a response to events such as a packet arrival. Since updates to relations are expressible using Boolean operations, CSDN programs ease significantly VeriCon's verification task.

VeriCon verifies correctness by validating if the invariants are preserved after arbitrary events on switches and on the controller, on an arbitrary topology that respects the topology constraints. In order for an invariant to hold, it must *i)* be satisfied at the initial state of the network, and *ii)* be inductive, i.e. the invariant must hold after an arbitrary sequence of events.

Writing inductive invariants is not straightforward because one has to specify the sets of states after an arbitrary sequence of events. However, VeriCon provides an utility that uses iterated weakest preconditions to produce inductive invariants from simpler invariants specified by the developer. This approach works as follows. Given an invariant and an event handler, a new condition will be added to the invariant. This condition is the weakest precondition that guarantees that the previous invariant is true after the execution of the event handler. According to the authors, at least for simple SDN programs, this approach can produce inductive invariants with few iterations.

Having the inductive invariants, VeriCon performs the verification by sending the first-order logic formulas (the invariants, the constraints on the topology and the CSDN program transformed in first-order logic formulas) to a theorem prover. The solver will then either output a concrete example that violates the invariants or confirm that the program is correct.

A drawback of VeriCon is that it only supports the verification of safety invariants and assumes that rules are installed atomically, thus ignoring out-of-order installation problems.

2.4.2.1.3 Systematic OpenFlow Testing (SOFT)

SOFT (Kuzniar, Peresini, Canini, Venzano, and Kostic 2012) is a tool designed to find bugs in OpenFlow implementations or interoperability problems among different switch vendors. Unlike the first testing and verification tools referenced, it does not take for granted that all OpenFlow switches work the same way, targeting firmware bugs or differences in the behaviors of switches from different vendors, caused by ambiguities in the OpenFlow specification. The approach taken by SOFT is to combine symbolic execution with constraint solving to generate sets of inputs (*e.g.,* fields in OpenFlow messages) that generate different outputs from one OpenFlow implementation to another.

2.4.2.1.4 Header Space Analysis / Hassel

The Header Space Analysis (HSA) framework (Kazemian, Varghese, and McKeown 2012) is a general framework, not specific to SDN. We present it in this section because the first implementation of this framework was a library of static analysis tools (called Hassel). HSA models the network behavior in a geometric model. Packets are just points in the $\{0,1\}^L$ space (the header space), where L is the header length. The network space is the cross product of the header space with the switch-port space (the space of ports in all switches), *i.e.,* $\{0,1\}^L \times \{0,..,P\}$ where P is the number of different ports in the network. The networking boxes such as Network Address Translation (NAT) boxes are modeled by transfer functions that convert points or subspaces of the network space into other points or subspaces. Multihop packet traversal is modeled through composition of transfer functions. HSA defines an header space algebra that can be used to mathematically analyze the network with respect to policies such as reachability.

Hassel retrieves information from routers such as MAC-Address table, ARP table and IP forwarding table to generate the network transfer functions. Then, using this network algebra and optimizations such as lazy evaluation of transfer functions and lazy subtraction it performs reachability tests, loop detections, and slice isolation checks.

2.4.2.2 Dynamic checking

Tools that check network programs at run-time are often called *dynamic/online checking/monitoring tools*. We will now head to the description of examples of such tools.

2.4.2.2.1 VeriFlow

VeriFlow (Khurshid, Zou, Zhou, Caesar, and Godfrey 2013) analyses the data plane state of a live SDN network and checks for invariant violations before every forwarding rule is inserted, updated, or deleted. While previous tools that perform offline invariant checks would be closer to the goal of not deploying buggy control software, the execution times of those tools are at timescales of seconds to hours. Moreover, they cannot detect or prevent bugs as they arise.

To perform online verification, VeriFlow intercepts the communications between the controller and network devices to monitor all network state update events (i.e. events that insert, modify or delete rules). However, to cope with the problem of state space explosion, VeriFlow confines the checks for invariant violations solely to the subpart of the network where these events will produce forwarding state changes.

VeriFlow computes equivalence classes (EC) of packets in order to find the subparts of network that are affected by a given rule. Here, just like in NICE, an EC corresponds to a set of packets that are equally forwarded across the network. For each EC, VeriFlow generates a forwarding graph where each vertex represents the EC in a given network device, and each edge represents the forwarding decision taken by this device for packets belonging to that EC. Thus, this graph gives us the expected flows in a physical network view (rather than in an abstract view) as it is constructed directly from information retrieved from the data plane through the OpenFlow API. After generating these graphs, VeriFlow will run queries that will be used to verify network invariants after a rule update is intercepted. If the invariants are not violated, the rule will be sent to the devices. Otherwise, VeriFlow will perform a pre-defined action, such as blocking the rule and firing an alarm or just fire the alarm without blocking the rule, in case the invariant violation is not dramatically severe (e.g. packet loss might be tolerable for a while, but ACL violations might not).

VeriFlow was designed to be transparent for OpenFlow devices. The authors have implemented two prototypes: one that is a proxy between the controller and the network, being controller independent. the second one is integrated with the NOX OpenFlow controller (Gude, Koponen, Pettit, Pfaff, Casado, McKeown, and Shenker 2008) for performance improvement reasons.

The downside of performing online invariant checks approaches with respect to offline approaches is that they degrade the performance of the network, regarding rule updates. In particular, the results in (Khurshid, Zou, Zhou, Caesar, and Godfrey 2013) show that VeriFlow incurs 15.5% performance

overhead on average (delay on rule update), which the authors argue to be tolerable in practice. Even though one of the components of our prototype is based on VeriFlow, it does not have the same overhead necessarily, because we do not need to check invariants. We only need the forwarding graphs.

2.4.2.2.2 Automatic Test Packet Generation (ATPG)

ATPG (Zeng, Kazemian, Varghese, and McKeown 2012) is a framework that automatically generates test packets from network configurations (using the Header Space framework (Kazemian, Varghese, and McKeown 2012)), proactively testing reachability policies and performance health. It generates the minimal set of packets to test the liveness of the underlying topology, and the congruence between data plane state and configuration specifications, *i.e.,* if the policies specified by the network operators correspond to the actual network behavior. ATPG can exercise every link in the network (minimally) or exercise every rule in the network (maximally), therefore reducing the scope of a bug to a single link or forwarding entry. However, because this tool generates test packets for certain classes of packets, it does not target errors in which different packets of the same class are forwarded differently (which is an error). *e.g.,* lets say that for a given class, EC, ATPG generates a test packet T. If a switch incorrectly forwards two packets A and B belonging to EC to different ports, this is an error. Although, because ATPG only tested the generated test packet T, the error remains undetected. Error like this are caused by certain faults in the hardware or software running on the switches.

2.4.2.2.3 Monocle

Monocle (Perešíni, Kuzniar, and Kostić 2015) targets bugs in switch software and hardware specifically regarding OpenFlow rule updates, using automatically generated probes to check if switches are correctly updating their forwarding tables. For this purpose, Monocle formulates switches' forwarding tables logic using a Boolean satisfiability (SAT) problem, and dynamically updates this model as network state changes. It first creates probes in the abstract space to test newly modified rules and then generates the real probes, comparing the output in the model with the output observed in actual behavior. In order to obtain this result, prior to sending probes, Monocle uses specific bits in a specific packet header only for probes (to avoid misinterpreting production packets with probes and vice-versa) and installs probe-catching rules in neighbor switches that forward probes back to the controller. Doing this, it identifies the outcome of each rule and can then compare it with the expected outcome in the model.

Monocle suffers from the same problem as ATPG because it uses generated test packets (probes). We have no knowledge of testing or verification tools that can guarantee that all packets of a class will actually be forwarded to the same port, if we allow switch arbitrary faults in our fault model.

2.4.3 Debugging SDN networks

Testing and verification tools for SDN applications have shown promising results and are undoubtedly useful to improve the overall quality of these applications by reducing the likelihood of errors. However, all these tools rely on models of the network behavior, hence cannot handle firmware and hardware unpredictable bugs, that occur at run-time, and produce non-deterministic behavior or behavior that cannot be related to network models in any way. *e.g.,* because we have no knowledge of the internal working of a switch, and it varies depending on the model, we do not have a guarantee that packets from the same equivalence class will be forwarded equally as expected. To address problems stemming from this kind of bugs (as well as other bugs that might have not been uncovered during the testing/verification phase), SDN debugging tools are still required.

Typical tools used to debug traditional networks include *ping*, *traceroute*, *tcpdump* and *SNMP statistics* (Handigol, Heller, Jeyakumar, Mazières, and McKeown 2014; Handigol, Heller, Jeyakumar, Maziéres, and McKeown 2012). Unfortunately, these tools are often not enough to find the root cause of bugs, or provide little aid in this task, because they are still very time consuming.

SDN facilitates the development of better tools for network debugging, since the control software is logically centralized and hardware transparent. Furthermore, these centralized control programs have a global network view, and write the network state changes directly into the switch forwarding tables using a standard API. (Handigol, Heller, Jeyakumar, Maziéres, and McKeown 2012)

In this section, we overview some tools designed to help developers debug errors in SDN networks.

2.4.3.1 NetSight

NetSight (Handigol, Heller, Jeyakumar, Mazières, and McKeown 2014) is an extensible platform designed to capture and reason on packet histories. Packet histories can be extremely helpful to debugging because one can ask questions such as where the packet was forwarded to and how it was changed, instead of having to manually inspect forwarding rules. The authors have implemented on top of it four network analysis tools for network diagnosis showing NetSight's usefulness. Among these tools, we are particularly interested in *ndb* (Handigol, Heller, Jeyakumar, Maziéres, and McKeown 2012) which is an interactive debugger for networks, with functionalities similar to *gdb*.

NetSight offers an API to specify, receive, and act upon packet histories of interest. The packet histories are obtained by means of a regular-expression-like language, called *Packet History Filters* (PHFs). More concretely, PHFs consist of regular expressions used to process and filter *postcards*. Postcards, in turn, are records created whenever a packet passes by a switch.

Postcards contain the following information: the switch id, output ports, and the version of switch forwarding state (i.e. a counter that is incremented every time a flow modification message is received). NetSight generates postcards by duplicating each packet that enters a switch and truncating the duplicated packet to the minimum packet size. This packet, i.e. the postcard, is then forwarded to a predetermined NetSight server, which are responsible for processing the postcards and generate packet histories.

To monitor the behavior of the switches, NetSight uses a process called *flow table state recorder* (or recorder, for short) placed in between the controller and the switches. The recorder intercepts all flow modification rules sent to the switches and stores them in a database. In addition, NetSight augments each rule with instructions to create the corresponding postcard, namely the destination of the NetSight server which the postcard should be forwarded to.

Naturally, recording postcards for every packet hop and processing them in order to build packet histories imposes scalability challenges. To address these challenges, NetSight uses one or more dedicated hosts to receive and process postcards, employs aggressive packet header compression and relies on a carefully optimized code.

The authors have implemented a prototype of NetSight and tested it on both physical and emulated networks, with multiple unmodified OpenFlow controllers and diverse topologies. According to the results reported in (Handigol, Heller, Jeyakumar, Mazières, and McKeown 2014), NetSight scales linearly with the number of servers.

2.4.3.2 OFRewind

OFRewind (Wundsam, Levin, Seetharaman, and Feldmann 2011) is a debugging tool that allows the record and replay of packets in SDN networks. As stated before in this section, standard tools used for network debugging are in many situations not enough to find the root cause of a problem. The behavior of black-box components (switches with proprietary code) cannot be understood by analytical means alone. In such situations, repeated experiments are needed. Therefore, it would be useful for developers to replay the sequence of events that led to an observed problem. OFRewind was designed to achieve this goal. The evaluation results shown in (Wundsam, Levin, Seetharaman, and Feldmann 2011) show that OFRewind can be enabled by default in production networks.

One of the key factors that allows OFRewind to be always on is the possibility of partial recording: even though it is possible to record the full set of packets, this is infeasible in an entire production network. Most of the times, it is enough to record just a small subset of the traffic, such as just the control messages or just the packet headers. The ability to replay different subsets of traffic, multiple

times, allows to reproduce the error and, throughout these repeated replays, isolate the component or traffic causing the error.

2.4.3.3 SDN Troubleshooting System (STS)

STS (Scott, Wundsam, Raghavan, Panda, Or, Lai, Huang, Liu, El-Hassany, Whitlock, Acharya, Zarifis, and Shenker 2014) aims at reducing the effort spent on troubleshooting SDN control software, and distributed systems in general, by automatically eliminating from buggy traces the events that are not related to the bug. This curated trace, denoted *Minimal Causal Sequence* (MCS), contains the smallest amount of inputs responsible for triggering the bug.

The minimization process consists of two tasks: *i)* searching through subsequences of the logged external events (e.g. link failures) and *ii)* deciding when to inject external events for each subsequence so that, whenever possible, an invariant violation is triggered again, during replay.

For the first task, STS applies delta debugging, which is an algorithm that takes as input a sequence of events and iteratively selects subsequences of it. STS also receive as an input the invariant that was violated by the execution of this sequence of events. If a subsequence successfully triggers the invariant violation, the other subsequences are ignored and the algorithm keeps refining that one until it has found the MCS. Note that, using this approach, an MCS is not necessarily globally minimal.

In order to keep the network state consistent with the events in the subsequence, STS treats failure events and its corresponding recovery events as pairs and prunes them simultaneously. Also, STS updates the hosts' initial positions when pruning host migration events to avoid inconsistent host positioning.

For the second task (deciding when to inject external events), STS uses an interposition layer that allows delaying event delivery to make sure that the replayed sequence of events obeys to the original "happens-before" order.

STS has the advantage of creating an MCS without making assumptions about the language or instrumentation of the software under test.

In terms of limitations, STS is not guaranteed to always find an MCS due to partial visibility of internal events, as well as non-determinism (although not finding is a hint for the type of bug). Also, performance overhead from interposing on messages may prevent STS from minimizing bugs triggered by high message rates. Similarly, STS's design may prevent it from minimizing extremely large traces. Finally, bugs outside the control software (for instance, misbehaving routers or link failures) are considered to be out of the scope of this system.

2.4.3.4 DiffProv

DiffProv (Chen, Wu, Haeberlen, Zhou, and Loo 2016), similarly to STS, aims at automatically minimizing the event trace that is known to trigger a bug in a distributed system. DiffProv is based on the concept of network provenance (Zhou, Sherr, Tao, Li, Loo, and Mao 2010), which is a way to describe the causal relationships between network events. Having logged the faulty event, the key idea is to use a reference event to improve the diagnosis. This reference event is one that, unlikely the faulty one, has produced the desired outcome for the network operator. Aside from that, a good reference is as similar as possible to the faulty event. This way, DiffProv algorithm will likely identify a small set of changes needed to transform the "bad" event into the "good" event, *i.e.,* the set of changes to fix the network misconfiguration.

Regarding SDN, DiffProv accepts as input a controller program and two packets: one that was forwarded as intended, and one that was not. DiffProv then uses a replay mechanism so that the algorithm can find a local minimum of events that triggers the difference between these two packets. Thus, like STS, bugs outside the control software are considered to be out of the scope of this system.

Summary

In this chapter we introduced the concept of Software Define Networking, its objectives and architecture. We also introduced the mostly well known API for switch configuration in SDN, OpenFlow. After, we started describing some of the tools that leverage SDNs' architecture to improve network reliability, which is our objective.

Regarding reliability, we approached these systems dividing them in two big categories: SDN testing and verification (Section 2.4.2) and SDN debugging (Section 2.4.3). In one hand we have tools that strive for finding networks errors before they arise, *e.g.,* checking for congruence between layers in the SDN stack and present either counter-examples of packets for network invariants, the whole set of packets that violates the invariants or problematic forwarding rules. On the other hand we have tools that are tailored to help network operators to find bugs after they occur, either in cases where the previous set of tools was not able to fulfill its job or in cases that they cannot guarantee correctness (non deterministic bugs or unpredictable behavior). These tools rely in instrumentation and event logging, typically so that one can either analyze them or reproduce them repeatedly and partially in order to consecutively narrow down the event list and isolate the root cause of the error. Since this manual inspection is usually very time-consuming, it is usually preferred to take the latter approach, but even this one is often a daunting task, and is not adequate for non-deterministic bugs.

With these ideas in mind, we propose a tool that could help network operators, helping to automate

part of the network debugging task (finding the faulty switch) and can deal with unpredictable and non deterministic bugs. The next chapter will introduce the architecture and implementation details of our system.

3 NetSheriff

This chapter describes the design of our system - NetSheriff and the implementation of our prototype. We start by providing an overall description of NetSheriff's architecture and components (Section 3.1). We then show how NetSheriff uses features from model checking tools to compute the expected path of a packet (Section 3.2), as well as tracing mechanisms to efficiently obtain the respective observed path (Section 3.3). After, we describe the main steps for building our prototype (Section 3.4). We conclude the chapter by presenting a list of categories of errors identified by NetSheriff (Section 3.5).

3.1 Overview

NetSheriff is a system that detects mismatches between the expected path of a packet and its observed path, and automatically pinpoints the network device responsible for the network misconfiguration. NetSheriff comprises four components: the *seer*, the *instrumenter*, the *collector*, and the *checker*. The seer produces a forwarding graph for each equivalence class of packets. The instrumenter appends actions to controller's commands, instrumenting switches to create a *postcard* when a packet traverses it. The collector assembles postcards generated by the same packet, creating a packet history. The checker receives both the packet histories and the forwarding graphs and produces the differential analysis, finding the faulty switch, if there is one. Our implementation uses a component of NetSight as instrumenter, and modified versions of another component of NetSight as collector and VeriFlow as seer. The modifications introduced were made to communicate with our new component, the checker. Before detailing each component, let us note that we have considered the possibility of postcard drops, but we will abstract away from this scenario until Section 4.4, that describes how we deal with this problem.

The aforementioned components, depicted in Figure 3.1, are described as follows.

3.1.1 Seer

The seer component in NetSheriff is responsible for computing the expected path for each equivalence class of packets in real time. To this end, the seer models the network's behavior as a set of

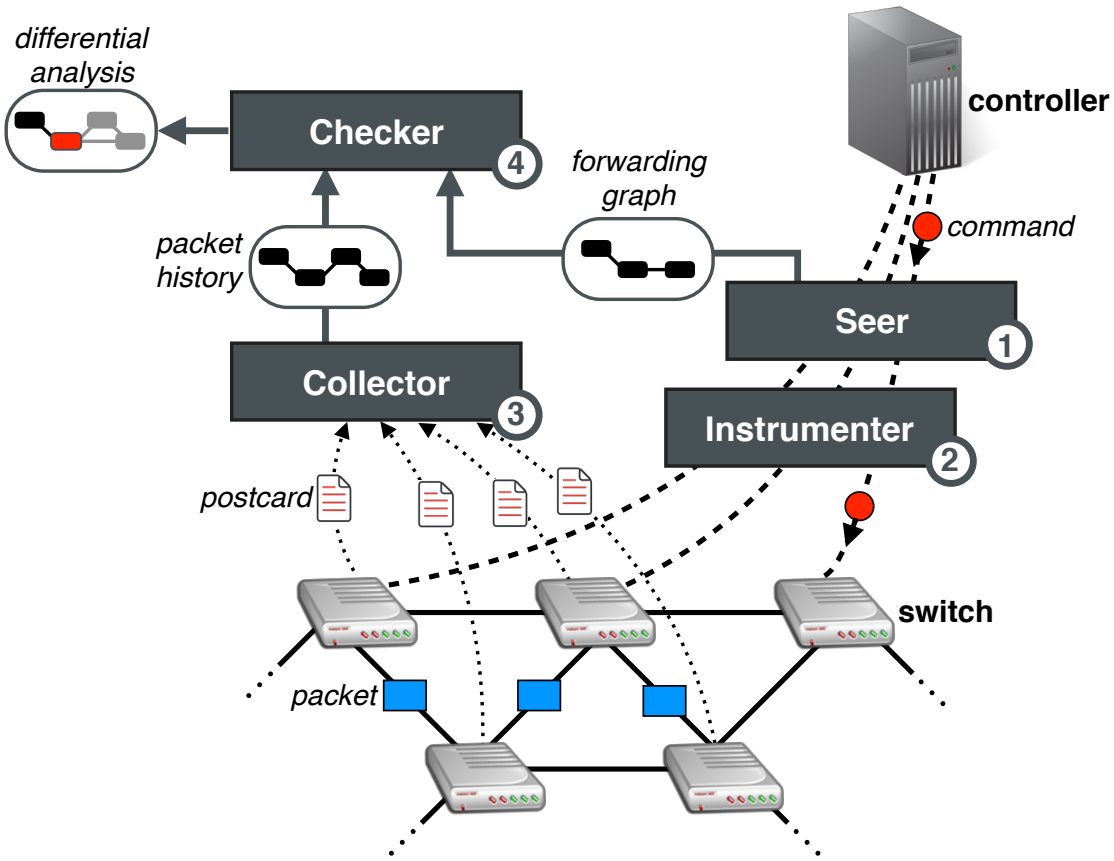

Figure 3.1: **Overview of NetSheriff.** 1) The *seer* builds forwarding graphs, which are a representation of how packets should be forwarded in the network. 2) The *instrumenter* intercepts the controller's commands and modifies them, so that they additionally instruct switches to create postcards of packets traversing them. 3) The *collector* collects postcards from the switches, and assembles them into packet histories, which correspond to the actual packet flows observed at runtime. 4) Finally, the *checker* is responsible for comparing the expected forwarding behavior of a packet (indicated by the forwarding graph) against the observed path (indicated by the packet history), in an effort to find inconsistencies, which reveal errors in the switches.

forwarding graphs (Khurshid, Zou, Zhou, Caesar, and Godfrey 2013). A forwarding graph is a representation of how packets belonging to the same *equivalence class* (*i.e.,* packets that exhibit the same forwarding action for any network device) should traverse the network. In other words, it represents the expected path of these packets. Vertices in forwarding graphs represent switches, and edges indicate forwarding decisions between two switches. For instance, let $\mathcal{F}$ be the forwarding graph of an equivalence class *EC*. An edge $A \rightarrow B$ in $\mathcal{F}$ indicates that switch A forwards all packets within *EC* to switch B.

Concretely, the seer intercepts all the commands sent by the controller to the switches (*e.g.,* rule insertion or deletion). Then, it generates new forwarding graphs for the equivalent classes that are affected by these commands. Finally, the seer sends the updated forwarding graphs to the checker, which will later use them to perform the differential analysis between the expected and the observed

paths of packets. Quickly computing these graphs in execution time in order to allow their analysis in a short time span might be a challenge. In Section 3.2, we describe how NetSheriff addresses this challenge.

3.1.2 Instrumenter

The instrumenter component has the goal of triggering the record of the necessary information to rebuild the paths taken by each packet in the network. For this purpose, the instrumenter leverages the SDN architecture, intercepting messages between the controller and every switch (note that both the seer and the instrumenter are transparent for the switches and the controller).

In particular, the instrumenter extends each rule sent by the controller, appending actions that first modify the packet so that it carries additional information (detailed below) and after that forwards this duplicate packet so that it will reach the collector. The original packet is sent unmodified to the specified output port in the rule. The duplicated packet, which we will call *postcard* just like in (Handigol, Heller, Jeyakumar, Mazières, and McKeown 2014), contains the essential packet information as well as the id of the switch that has recorded it, the version of the switch flow table (a counter incremented when this table is modified), and the input and output ports.

3.1.3 Collector

The collector component consists in a server (centralized or distributed) that receives the postcards sent from the switches and reorganizes them with the purpose of creating multiple distinct collections, designated by *packet histories*, as in (Handigol, Heller, Jeyakumar, Mazières, and McKeown 2014). Each packet history corresponds to the set of all postcards generated while a packet traverses the network. The packet history thus allows to build the path taken by this packet and infer the possible header modifications performed by each switch.

3.1.4 Checker

The checker component is responsible for signaling an unexpected network behavior regarding forwarding decisions, pinpointing the faulty switch that caused such misbehavior. To this end, the checker leverages both the forwarding graphs generated by the seer and the packet histories assembled by the recorder.

The checker performs the differential analysis by projecting the histories against the graphs and detecting possible divergences. If the projection of the expected path and the actual path yields a perfect

match, then the packet was correctly forwarded across the network and NetSheriff does not report any anomaly. Conversely, when there is a mismatch between the expected path and the observed path, NetSheriff reports the incorrect forwarding and indicates the switch where the divergence first occurred, *i.e.,* the switch responsible for the fault. In this latter case, different patterns in the projection can also give further information about the switch problem.

As a simple example, consider the scenario in Figure 3.2, which depicts an SDN with five switches: A, B, C, D, and E. Now let us consider that one wants to send a packet from A to E and that the packet is expected to be forward along the route $A \rightarrow D \rightarrow B \rightarrow E$ (Figure 3.2b). However, the actual path of the packet ends to be $A \rightarrow D \rightarrow E$ (Figure 3.2c). When the checker performs the differential analysis between these two paths, it verifies that there is a mismatch in switch D. As a result, NetSheriff reports a fault in the SDN and identifies switch D as the source of the problem.

In more complex scenarios, there are the possibilities of *multicasts* or *broadcasts*, which means that, in certain points, the paths can be split in multiple branches. Furthermore, switches might modify the packet headers (*e.g.,* in presence of Network Address Translation (NAT) boxes). These modifications must be taken into account when performing the differential analysis, because when the header is modified, the "new" packet might belong to a different EC, therefore having a different forwarding graph. NetSheriff deals with this problem by merging the multiple possible graphs for a given packet history. The merged graph is then projected against the postcards of the corresponding packet histories. If there is a divergence between the expected and the observed paths, the error can be categorized according to the characteristics of the projection. The differential analysis algorithm is described in detail in Section 3.4.

3.2 Computing Forwarding Graphs

As mentioned in the previous section, NetSheriff's Seer component is based in forwarding graphs, which represent the paths that each equivalence class of packets is expected to follow. NetSheriff computes forwarding graphs while the network state is being altered, recomputing the graphs only for equivalence classes that are affected by the intercepted rules issued by the controller, since in the presence of large and complex networks, it becomes impractical to compute forwarding graphs for the whole network every time a new forwarding rule is changed.

NetSheriff's seer component is built on top of part of VeriFlow (Khurshid, Zou, Zhou, Caesar, and Godfrey 2013), leveraging its ability to efficiently compute the forwarding graphs. In particular, we do not need to verify network invariants, but we need the rest of its functionality to compute the graphs. We modified it so it sends to the checker component the links of the forwarding graphs that are modified, along with the respective EC. The checker will then reconstruct the forwarding graphs using this information.

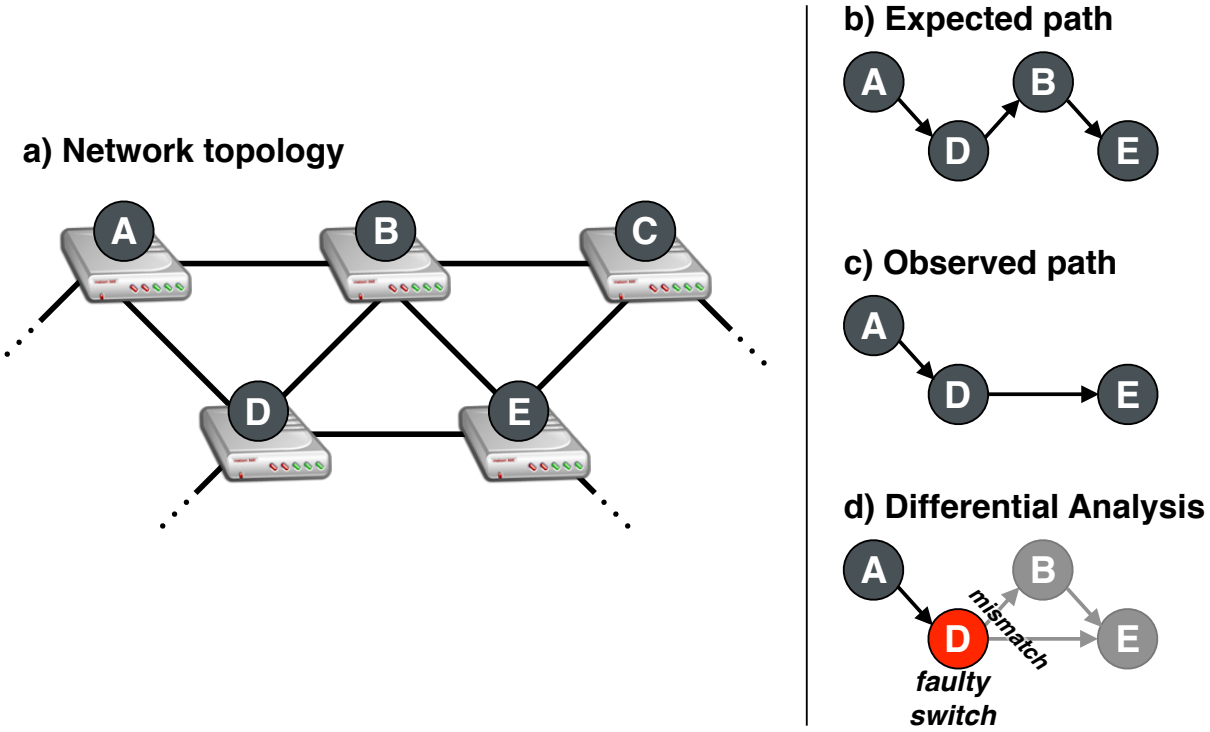

Figure 3.2: **Differential analysis performed by NetSheriff's checker.** The checker projects the expected path (3.b) against the observed path (3.c) in order to find potential mismatches in the packet flow and pinpoint the faulty switch (3.d).

The seer maintains a *trie* (*i.e.,* a prefix tree) that associates the forwarding rules installed in the switches to the prefixes of the packet header fields that they match. Using this structure, the n-th level in the trie represents the n-th bit of the packet header field matched by a forwarding rule. Nodes in the trie (apart from the root and the leaves) contain one of the three possible values for that specific bit in the packet header, namely 0, 1, and *wildcard* (represented by $*$). Hence, each node in level $n-1$ spawns three child nodes, corresponding respectively to the three values that the n-th bit of the header can have. Leaves in the trie store pairs of type *(s, r)*, meaning that the switch s contains a forwarding rule r that matches packets with prefixes given by the path between that leaf and the root of the tree.

When the controller issues a command to one of the switches, the seer traverses each level of the trie to find all packet headers that are affected by the incoming rule. In particular, the search outputs the set of leaves whose rules overlap with the new forwarding decision. Note that, by having each dimension representing a bit of the packet header, the trie allows the seer to perform the lookup very efficiently (as it only searches along the branches that fall within the address range of the new rule).

Next, the seer computes affected ECs (*i.e.,* ECs affected by the updated rule) as follows. For the set of overlapping rules, the seer computes a group of disjoint ranges (starting from the most generic rule to the least generic one), such that no range can be further divided. Each EC will then be defined by a unique, individual range. As an example of this procedure, let us consider a switch with a rule matching

packets having IP addresses with prefix 10.1.1.0/24. Now consider that the switch installs a new rule (with higher priority) affecting packets within the address space 10.1.0.0/16. Since the two rules overlap (the latter rule is more restrict than the former and has higher priority), NetSheriff's seer will identify three different ECs for this case, corresponding to the three following ranges: [10.1.0.0, 10.1.0.255], [10.1.1.0, 10.1.1.255], and [10.1.2.0, 10.1.255.255]. A more detailed description on how to compute ECs can be found in Veriflow's paper (Khurshid, Zou, Zhou, Caesar, and Godfrey 2013).

Finally, NetSheriff's seer generates the forwarding graphs for the new ECs and sends them to the checker, as referred in Section 3.1.

3.3 Computing the Observed Path

NetSheriff's obtains packet histories by assembling the postcards generated by the switches which the packets passed through. To this end, the collector has to gather all postcards corresponding to each individual packet, which may correspond to a large amount of data. To build packet histories in an efficient and scalable way, we used two components of NetSight (Handigol, Heller, Jeyakumar, Mazières, and McKeown 2014). NetSight is a tool for generating, capturing and processing postcards. For large networks, it allows the usage of multiple *servers* that collect postcards. Our instrumenter is NetSight's Flow Table State Recorder (FTSR), and we our collector is a NetSight server that sends all the packet histories to our checker component.

3.3.1 Generation and capture

NetSight implements the mechanism for packet duplication described in Section 3.1. The propagation of postcards can be performed in two different modes: using the production network (*in-band*) and therefore consuming part of the available bandwidth but avoiding the use of additional switches only for this purpose, or opposingly, using a different subnetwork to avoid bandwidth overhead at the cost of having dedicated switches for this purpose, that will connect the remaining switches to NetSight servers. As it was already stated, in either of these two situations it is possible to use multiple servers, dispersed in the network. This will minimize the traffic that is essential for this phase. Using multiple servers requires another phase that will guarantee that postcards generated by the same packet will eventually be stored in the same server.

3.3.2 Processing

To ensure load balancing, NetSight distributes the packets across the servers. Servers periodically exchange batches of postcards among them, such that postcards corresponding to the same packet are assembled by the same server. To guarantee packet locality, NetSight uses the packet ID (which is an hash of the packet contents) as index, and associates servers with packet index ranges. Moreover, to reduce both the storage space and the network bandwidth, postcards are compressed before being exchanged by the servers.

Each server maintains a *path table*, where it stores postcards belonging to different packet flows. The path table is a key-value data structure that maps a unique packet ID to the group of postcards observed for that packet.

NetSight's coordinator receives user queries, called *packet history filters*, for filtering user's packet histories of interest. This allows network operators to look at packet histories that match specific patterns which they want to analyze, *e.g.,* packets that traverse two specific switches (or any other number). Matched packet histories become available to applications built on top of NetSight, along with the respective matched filter. NetSheriff performs the task of setting filters automatically, setting filters for affected equivalence classes after the seer component computes these classes.

NetSheriff requires packet histories to be ordered in order to be able to perform the differential analysis. However, due to delays in the network, postcards from switches can arrive to the servers in an out-of-order fashion. To cope with this issue, the collector relies on topology information (namely information regarding the switch IDs and output ports) to correctly sort the postcards. A packet history thus corresponds to a sorted list of postcards. Once assembled, packet histories are sent to the checker, as described in Section 3.1.

3.4 Implementation

As it is mentioned in Sections 3.2 and 3.3, NetSheriff's seer and collector components are small modification of VeriFlow and of NetSight server, respectively. The modifications add the necessary logic to project packet histories in forwarding graphs, by sending the data to our checker component. Our instrumenter component is NetSight's *flow table state recorder (FTSR)*. This NetSight component originally acts as a proxy between an OpenFlow controller and the switches of the network, as it happens with VeriFlow. Thus, we connected these two proxies to each other so that VeriFlow acts as a proxy between the controller and FTSR, and this latter acts as a proxy between VeriFlow and the switches. NetSight servers listen to specific ports of host machines called logger hosts. In the specific case of

our prototype, we launched a server listening to a port created by the network emulator MiniNet (Lantz, Heller, and McKeown 2010), which was the chosen environment to test and evaluate our prototype.

Despite the checker not being an extension to either of these systems, but rather a component whose inputs are generated by their extensions, in our prototype the checker's code was added to NetSight server's code to avoid an extra communication flow and another program being executed, thus easing the development and tests. The changes performed to the original versions of VeriFlow and NetSight were the following.

- Upon receiving a flow_mod message, VeriFlow computes the affected Equivalence Classes of packets and the respective Forwarding Graphs. In NetSheriff, we leverage this fact to generate the expected path of a packet from the forwarding graphs created by VeriFlow. Concretely, we extended VeriFlow to send the forwarding graphs to NetSight server as a sequence of connections, which are pairs of switch identifiers. This simpler version of forwarding graph is sent through an operation that we added to NetSight server's API - `change_path_request` - that receives a pair ⟨*equivalence class, forwarding graph*⟩, in the form of text;
- NetSight server was extended to process the pair mentioned above through the following steps. When the pair is received, NetSight server reconstructs the equivalence class and the respective forwarding graph (received as text). Then it installs a packet history filter that matches packets belonging to this equivalence class. Since, by definition, packets cannot belong to more than one equivalence class, we have guaranteed that this filter results (*i.e.,* matched packet histories) are uniquely matched by one filter. In other words, the packet history returned by matching a filter will be associated to only one expected path, and is ready for differential analysis.

3.4.1 Differential analysis algorithm

Having available the packet histories and the respective expected paths (forwarding graphs), the checker can proceed to differential analysis to check if the observed paths match the expected ones.

The checker component reconstructs the forwarding graphs using the same method of ordering as the topological order arranged by NetSight. Therefore, when it receives a packet history, it can assume that as as soon as a difference in the paths is detected, this difference exists, without having to analyze the remaining postcards. For this reason, the checker performs a breadth-first search in order to detect faults as soon as possible in the path. This fact is relevant because a fault in a switch might cause that a packet would then belong to a different equivalence class, if the headers are modified differently than expected, invalidating the rest of the graph from that point onwards. For this reason, the graph should be analyzed starting in the source rather than arbitrarily.

The divergences are found by comparing the vertices from the forwarding graph and the id of the switch associated to the next postcard popped out of the packet history. Before processing each vertex, the checker analyses the data stored in the postcards to verify if there were any modifications to the packet header during its network traverse. In case of any modifications, the graphs corresponding to the equivalence classes of the different headers are merged. After that, the checker compares each neighbor vertex in the forwarding graph with the id of the switch associated to the next postcard, until it finds a mismatch, marking the edges throughout the search. If all the graph is traversed and there are not more postcards in the packet history, we have a perfect match and the check passes. Otherwise an error is reported.

This is semantically equivalent to performing a projection of two graphs (expected and observed paths). We classify edges of the projection as *expected* or *unexpected*. An expected edge is an edge that belongs to the merged forwarding graph (*i.e.,* the edge represents a portion of the expected path of the packet in the network). On the other hand, an unexpected edge is an edge that did not belong to the merged forwarding graph but is created by projection a vertex in the graph and the switch associated to the next postcard in the history. In other words, an unexpected edge means that the packet was forwarded to an unexpected switch, according to the network configuration.

In summary, when the network is behaving correctly, corresponding to its configuration, all expected edges in the projection graph are marked by the checker. Also, there are not unexpected edges in the projection graph. On the other hand, in case of errors, these conditions are not verified simultaneously. Additionally to identifying the faulty switch, we can also categorize the error, using the criteria defined in Section 3.5.

3.5 Error categories

Below we categorize switch configuration errors according to the characteristics of the projection graph when the mismatch is found and what problems they could raise in the network.

- ***Total unexpected forwarding***
  Description: Switch forwards a packet that should be dropped.
  Examples of possible problems raised: Access control violations.
  Detection: One or more unexpected edges have origin in vertices without expected exiting edges.

- ***Partial unexpected forwarding***
  Description: Switch forwards a packet to additional ports other than the expected in the configuration.
  Examples of possible problems raised: Network congestion, access control violations.

Detection: At least one expected edge exits from vertices with at least one marked expected edge.

- ***Unexpected partial drop***
  Description: Switch forwards a packet only to a (non-empty) smaller subset of the expected ports.
  Examples of possible problems raised: Availability downgrade (less fault tolerance).
  Detection: A source vertex contains one or more marked expected edges, alongside one or more unmarked expected edges.

- ***Unexpected total drop***
  Description: Switch unexpectedly drops a received packet.
  Examples of possible problems raised: No reachability.
  Detection: All the expected edges exiting from a vertex are unmarked.

- ***Suboptimal routing***
  Description: Combinations of the above cases, but the packets reach their correct destinations.
  Examples of possible problems raised: Network congestion.
  Detection: Possible combinations of the aforementioned cases where it is possible to traverse the projection graph from the source to the final vertices (*i.e.,* the ones that have no exiting edges) through expected marked edges and the unexpected edges calculated by NetSheriff.

Summary

In this chapter we have presented the design and implementation of NetSheriff. Leveraging the ability of others tools to efficiently create and maintain forwarding graphs for each equivalence class of packets (VeriFlow) and to create, store and process records of the packets' network traversals (NetSight) we built a system that is able to perform a differential analysis with the expected paths of packets and with the respective observed paths, in order to locate the responsible switch for a network misconfiguration. Our differential analysis algorithm works by creating a projection of the packet histories provided by our Instrumenter and Collector components (built on top of NetSight) in the forwarding graphs provided by the Seer components (built on top of VeriFlow). During this task, the algorithm classifies the edges in a way that it can then provide more information about the error, being able to classify it among five non disjoint categories.

In the next chapter we present the experimental evaluation made using this prototype.

4 Evaluation

This chapter describes the experimental evaluation that we performed for NetSheriff. We will start with the details of the base experimental setup for each of the evaluated case studies (Section 4.1). We will then analyze five simple case studies that serve the purpose of demonstrating that NetSheriff can identify the faulty switch in errors from the five categories presented in the end of the previous chapter (Section 4.2). After that, we will see slightly more complex cases, yet more realistic (Section 4.3). Finally, we discuss the system's accuracy (Section 4.4) and network overhead (Section 4.5).

4.1 Experimental setup

We evaluated NetSheriff's prototype mainly targeting its efficacy in identifying faulty switches, as well as its ability to differentiate multiple types of errors in an SDN. With this in mind, we performed experiments with multiple types of errors that might appear in these networks. The experiments were performed using MiniNet (Lantz, Heller, and McKeown 2010), version 2.2.1, in an Intel i7-720QM with 8GB RAM DDR3, 250 GB SSD and Ubuntu 14.04.

Fault injection: To evaluate NetSheriff's efficacy in detecting errors in SDNs, we injected different faults in switches so that the multiple types of errors enumerated in Section 3.5 were generated. More concretely, to inject faults in switches we changed the flow tables using the command:

```
sudo ovs-ofctl mod-flows <switch-id> <flow>
```

This command changes a flow table of the Software OpenFlow Switch being used by MiniNet, modifying an entry. Concretely, we use it to modify the actions associated to a given entry. For example, if we want to inject a fault of dropping a packet that should be forwarded to port 1, we copy the entry, retrieved with the command `sudo ovs-ofctl dump-flows <switch-id>`, and substitute the action `output:1` by `output:0` (which means drop).

Using this approach, NetSheriff does not change the forwarding graphs, since the forwarding rules are not being modified by the controller. We can also simulate that a switch fails to install, modify or delete one or more rules by recording entries in the flow table of that switch before the controller

issues the commands and then restore these entries using the aforementioned method. This way, the forwarding graphs will change but the switch will keep forwarding like it did not receive any commands.

Network configurations: The first experiments were performed using the simple topologies and network configurations presented in Section 4.2. By applying different faults on these topologies,, we recreated the different error categories listed in Section 3.5. We then made tests in fat-trees using more realistic network configurations.

To perform these experiments, we first modified a POX controller so that it would configure the network according to the expected paths for each case. We then manipulated the flow tables of the ”faulty” switch in order to simulate that a controller command was ignored by that switch, a command that would result in the expected configuration. We modify the corresponding entry so that packets are forwarded along the observed paths represented in the figure.

We also performed additional experiments, using the POX and NOX unmodified controllers in larger fat-trees and linear topologies of multiple sizes, to evaluate the operation of NetSheriff in multiple different situations, when possible (for example, some controllers do not allow physical loops in the network, independently of using NetSheriff or not). NetSheriff also operated correctly in these experiments.

Applications: For our tests we used ICMP pings, `iperf` and a python web server.

4.2 Simple case studies

To better understand and assess the limitations of NetSheriff, we first tested our prototype with five simple configurations that produce the categories of errors previously described when we injected specific faults. Recall that these categories are not disjoint sets.

We also tried different variations of each case and verified that NetSheriff was able to identify the faulty switch and the type of error in all cases, except when we consider dropped postcards. In this situation, there were cases where our system can extrapolate the missing part of the observed path (*i.e.,* the dropped postcard), and others where it can only identify that one of two switches is failing, or multiple pairs of adjacent switches are failing. This is not our ideal objective, but it is still very good to be able to narrow down an error to pairs of switches, specially considering the number of switches on typical datacenters (hundreds to thousands). This issue of postcard drops and accuracy is explored in Section 4.4

4.2.1 Unexpected forwarding

The first two cases are errors in which a switch forwards a certain class of packets through more ports than those that it was expected to. Recall that this type of errors is detected by NetSheriff when unexpected edges are found, and that an unexpected edge is an edge that is found in the projection of a packet history in a forwarding graph of the respective equivalence class of packets, but is not present in the original forwarding graph.

The distinction between the two cases (presented below) is relevant because, as we have seen in Section 3.5, these two types of error may have different consequences, and also because one may be harder to detect with other tools than the other. Therefore, it seems logical that we check if NetSheriff can locate the error automatically in both situations and also if they are correctly identified.

4.2.1.1 Total unexpected forwarding

Figure 4.1 represents this case. We can see that there is a difference between the expected path and the observed path for packets of a certain EC. Particularly, switch D should drop all packets of this EC, although, we can observe that it is forwarding packets through the link $D \longrightarrow E$. NetSheriff detected this case correctly.

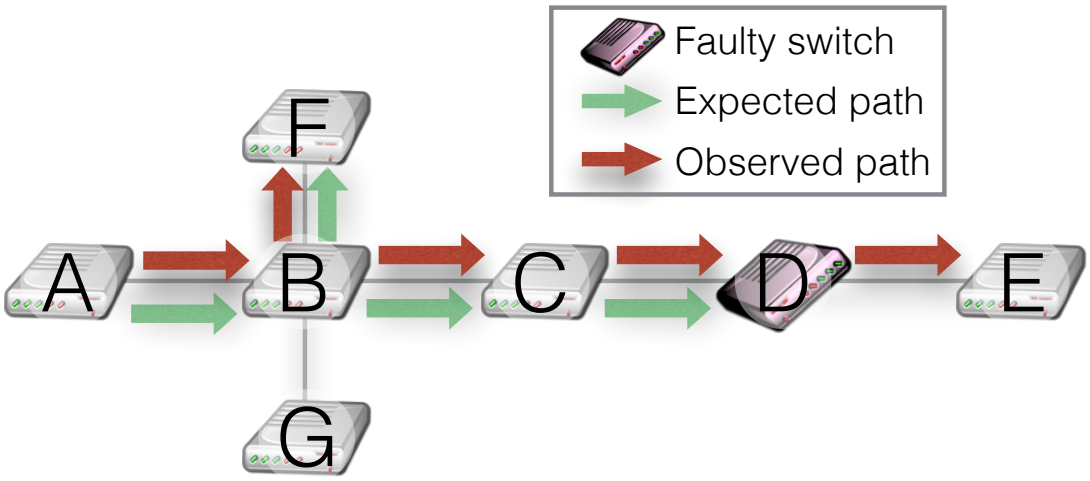

Figure 4.1: **Total unexpected forwarding.** Switch forwards a packet that should be dropped.

4.2.1.2 Partial unexpected forwarding

This case is represented by Figure 4.2. Now, comparing the represented expected and the observed paths, we are looking at a slightly different type of error: now the switch should not drop the packets.

Instead, the switch is expected to forward the packets to some ports, but the injected fault makes it forward packets through additional ports. Concretely, switch B should forward packets only to links $B \longrightarrow C$ and $B \longrightarrow F$. Despite that, not only switches C and F receive these packets, but also switch G.

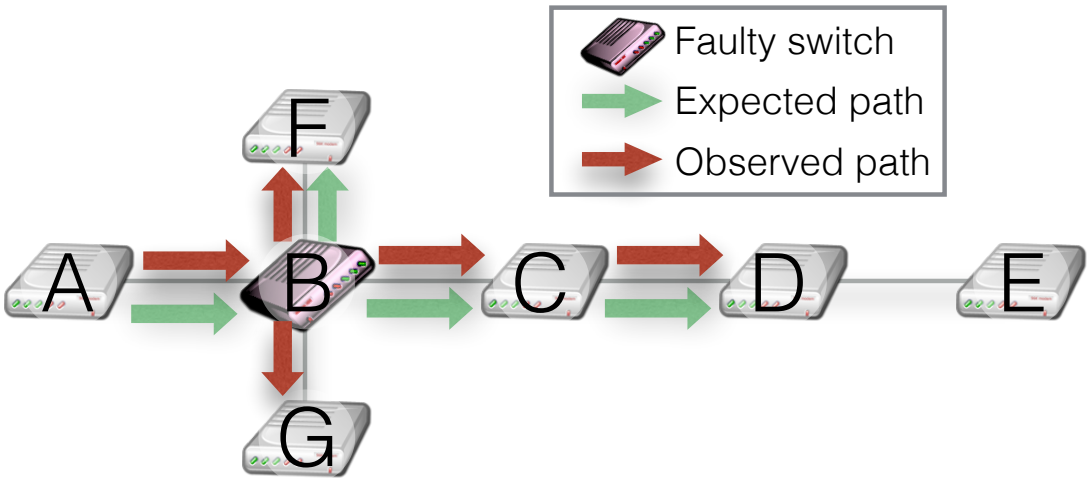

Figure 4.2: **Partial unexpected forwarding.** Switch forwards a packet to additional ports other than the expected in the configuration.

4.2.2 Unexpected drops

The next two cases are errors in which a switch drops packets that were expected to be forwarded. As it was mentioned in the previous chapter, this type of errors is detected if the differential analysis algorithm finishes with unmarked edges in its projection.

Again, the distinction between these two cases is relevant because these two types of error may have different consequences (identified in Section 3.5), and also because one may be harder to detect with other tools than the other, depending on the policies defined by the network operator. For example, if five servers are receiving the same packets, it is reasonable to consider that one would detect a total drop faster than a partial drop. The first case is a reachability problem, while the first is just an availability downgrade (some replicas receive the packets).

4.2.2.1 Unexpected partial drop

In Figure 4.3 we can see that the error corresponds to a switch dropping packets on some ports where it was expected to forward them, while maintaining the correct behavior in other ports. Namely, in this case, switch B forwards the packets only to switch F, when it was expected to forward them also to switch C.

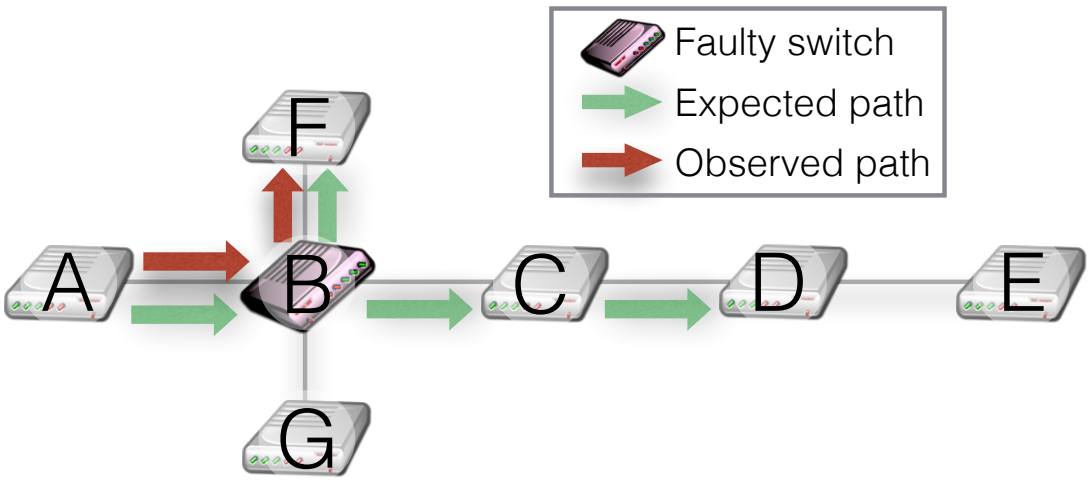

Figure 4.3: Unexpected partial drop

4.2.2.2 Unexpected total drop

Figure 4.4 represents the case where a switch just drops all the packets of an EC. In this case we can see that none of the links $B \longrightarrow C$ or $B \longrightarrow F$ is followed.

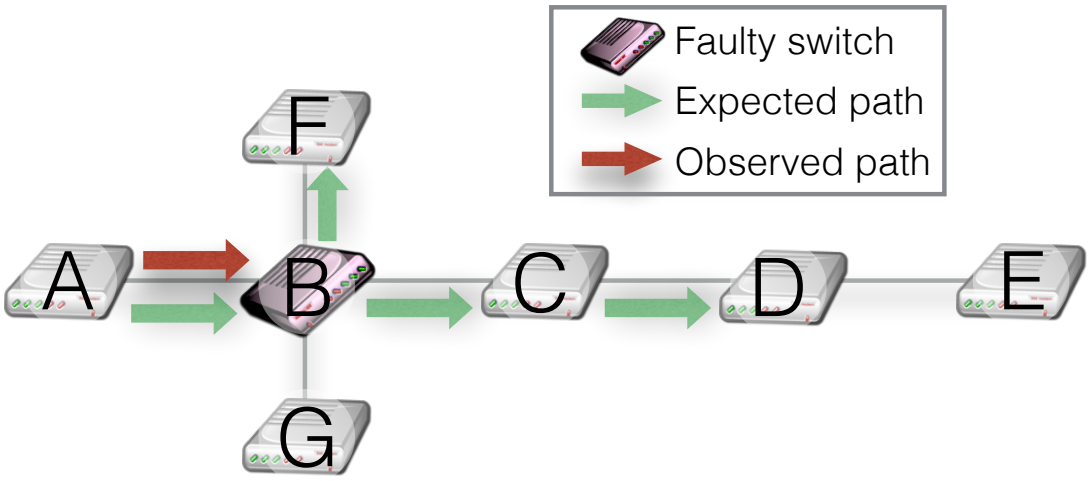

Figure 4.4: Unexpected total drop

4.2.3 Suboptimal routing

Figure 4.5 shows a case where a switch forwards the packets incorrectly, but they still reach their destination. Of course, this is still undesirable and could trigger multiple problems, being performance degradation (*i.e.,* bottlenecks in the network) the most obvious. Particularly, in this example, switch B

should forward packets to switches F and G. Instead, it forwards them to switches F and C. Because of the state of the forwarding tables of the other switches, packets will still reach F, following another path. Note that, despite switch B not being expected to forward the packets to C, we still represented the path from C to G to avoid confusion, but recall that even if the rules are not yet installed, the path could be followed without errors if the controller then sends the respective commands. The problem here is that switch B had a rule in its forwarding table to match those packets and output them to switches B and G only, so switch C should not receive it (and G should, but from B).

Errors like this could be triggered by a switch that does not install a rule or a set of rules (in this case, switch B). They are special instances of cases where the ir both unexpected forwarding and unexpected drops.

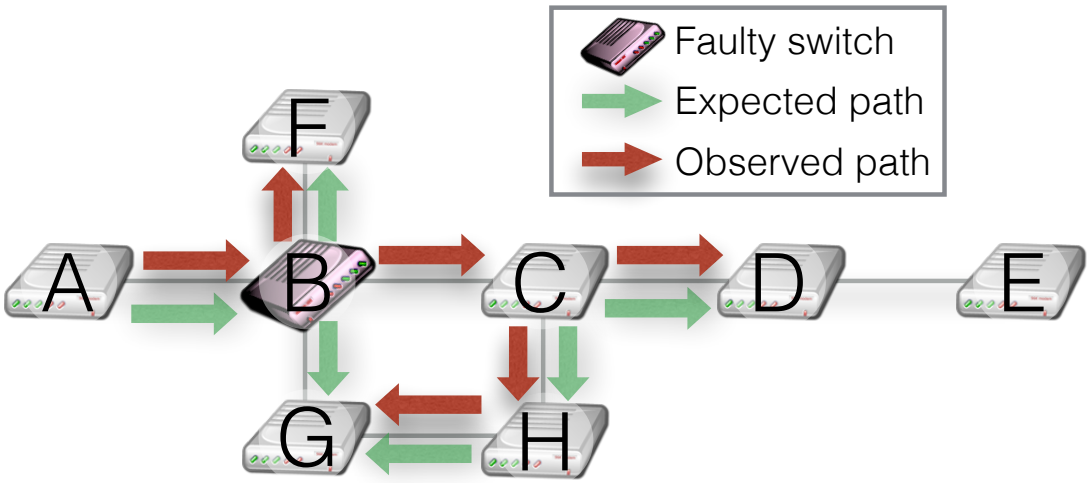

Figure 4.5: Suboptimal routing

4.3 Case studies in Fat-Trees

We now present two network configurations on a very common baseline for datacenter topologies, the *fat-tree* (Al-Fares, Loukissas, and Vahdat 2008), along with possible errors that could occur and their respective possible causes. Both topologies of the examples are fat-trees with $k = 4$ (the simplest fat-tree), since this simplifies the presentation and comprehension, but they can be adapted to larger and more realistic values of k. As with the simple case studies, we are for now ignoring the issue of dropped postcards. That will be discussed in Section 4.4.

4.3.1 Ignored commands during load balancing

The first fat-tree configuration, represented in Figure 4.6, is based on a load balancer similar to one that uses Equal-cost multi-path routing (ECMP), where traffic between two hosts is split among the best paths that have an equal cost. A specific path is chosen based on a 5-tuple hash for each packet (the protocol number, the IP addresses and the TCP or UDP source and destination port numbers).

Consider, in this case, communications between $H1$ to $H8$. Each packet can be forwarded through one of four different paths. These four paths are enumerate in the caption of Figure 4.6. Every time there are different branching possibilities, the links are represented with thiner lines, and when the traffic from two different switches is merged to only one output, the line gets thicker. For example, packets arriving to switch $A1$ from switch $E1$ could be output to switch $C1$ or to switch $C2$, depending on the hash result. Ideally, half of the packets would be routed through sub-paths represented with full lines, and the other half would be routed through the others, represented with dashed lines. Also, we will consider that the controller logic is fault tolerant against path breaks, meaning that if a switch that is used for full-line paths (but not for dashed-line paths) crashes, the corresponding traffic should be now routed through dashed-line paths. Switches that meet these conditions are $A1$ and $A3$. Only one of these needs to fail in order for the traffic to be redirected. Also, if $C1$ and $C2$ fails, the result would be the same, although this is more unlikely compared to the previous scenario. Note that in bigger topologies there would be more paths and more switches would meet these conditions. Also note that this type of crashes is not rare in big datacenters, hence the fault tolerance mechanism, which can be implemented by using heart beats or regularly checking switches' port status (ping/echo).

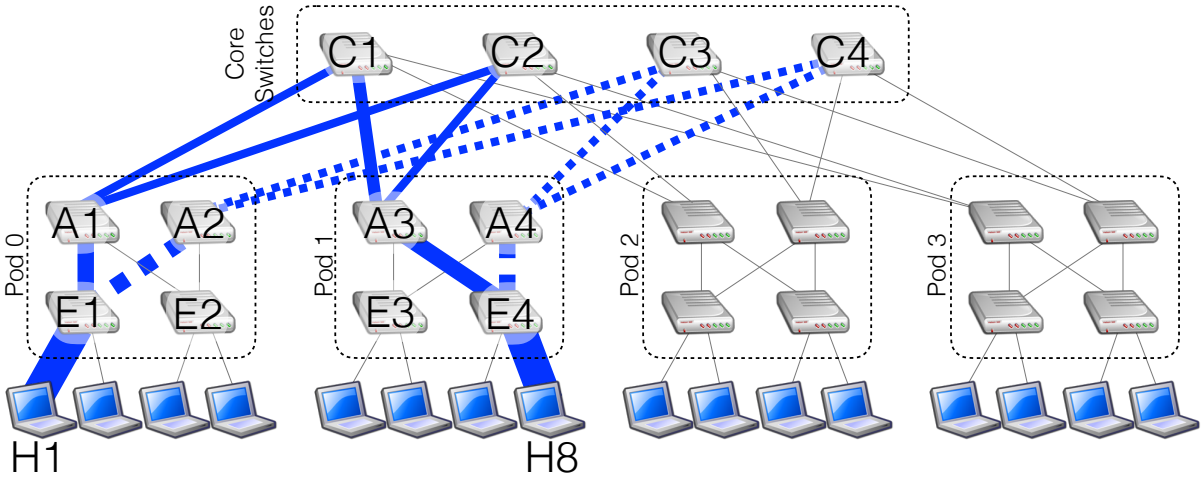

Figure 4.6: **Load balancing in a fat-tree.** Ideally, the traffic would be equally (in ECMP) splited among the different equal (lowest) cost paths. Between hosts H1 and H8, these paths (with hosts excluded) are: $E1 \rightarrow A1 \rightarrow C1 \rightarrow A3 \rightarrow E4$; $E1 \rightarrow A1 \rightarrow C2 \rightarrow A3 \rightarrow E4$; $E1 \rightarrow A2 \rightarrow C3 \rightarrow A4 \rightarrow E4$ and $E1 \rightarrow A2 \rightarrow C4 \rightarrow A4 \rightarrow E4$

The situation that might happen not so often, but is also way harder to detect, is the one where a

switch is not behaving as specified, failing in a not so clean manner. One example of such a failure is ignoring new commands from the controller. Failing to install new rules in switch $E1$ at this point would cause packets from currently installed flows to keep being routed through dashed-line paths, even when the full-line paths could be followed again, after the crashed switch is recovered. At this point, there is a misconfiguration in the network, causing an error that is not always detected, since packets are reaching their expected destination, without delays while the path is not congested. If this was a transient fault in switch $E1$ the full-line path will be used by other classes of packets and detecting the error will get even harder, let alone identifying the responsible device. NetSheriff can be used to detect such problems, as we verified experimentally with this case.

4.3.2 Anomalous forwarding in a actively replicated server

The second example on fat-trees is a configuration that implements an actively replicated server using network level packet duplication. Consider an edge switch that was configured to replicate certain packets to $k-1$ servers, each connected to a different pod. In this case, where $k=2$, this would be 3 servers. We set a network to have this behavior, represented in Figure 4.7, and then injected faults to drop packets, partially or totally, reproducing a behavior similar to the unexpected packet drops in the simpler examples (Sections 4.2.2.1 and 4.2.2.2). Concretely, $H1$ is sending packets to $H5$. Switch C1 replicates these packets as they arrive, redirecting copies to $H9$ and $H13$. The replication could be done in other switches and use multiple paths in a ECMP like fashion. In this particular case, the faults injected made switch $C1$ send the packet only through 2 or 1 ports instead of 3, or simply drop it. Again, NetSheriff was able to identify the switch responsible for the errors and determine the category of the error.

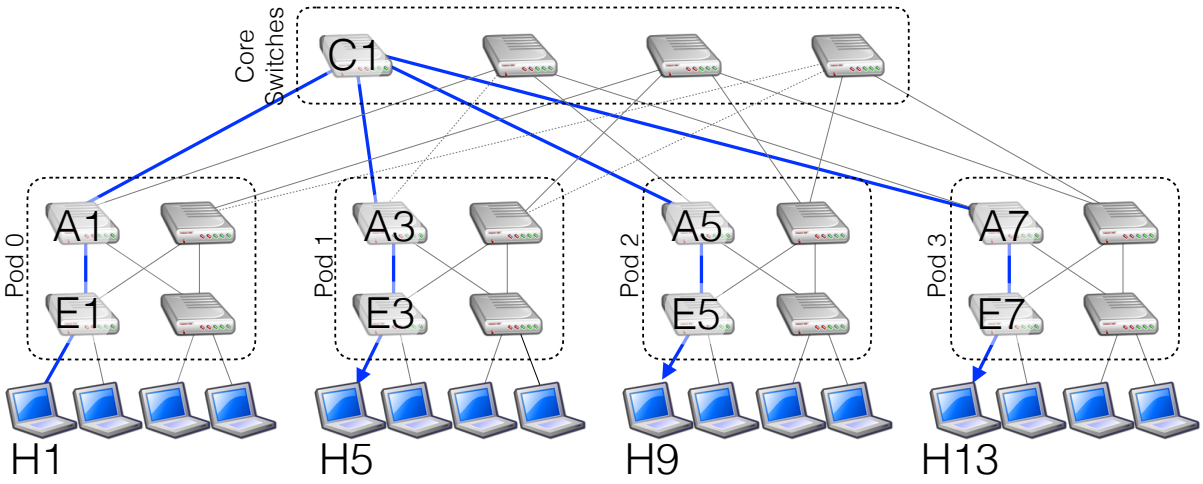

Figure 4.7: Replication in a fat-tree

4.4 Postcard drops and accuracy

The results of the experiments show that NetSheriff was able to correctly detect all the tested error scenarios, except when we injected faults to drop postcards. However, we cannot assume that postcards are not dropped, because that would defeat the purpose of our system, as we are mainly aiming at locating faulty switches, targeting hardware faults and faults in software running on switches. Thus we must find ways of detecting postcard drops or at least suspect of these drops.

We address this problem by trying to reconstruct the observed path (or the branch) when we miss a postcard. Because postcards carry information about the output ports, when we cannot proceed in the graph search we know the switch of the missing postcard (using the topology information). Postcards also carry information about the input port. Thus, we can check which switch sent the packet that generated the available postcards to the switches that generated them. If we find a match against the switch of the missing postcard, we can reconstruct the path and even proceed with the differential analysis. This worked for all tested examples of a single failing switch if the error was an unexpected forward (variations of the previous presented case studies).

The harder case is when normal packets are unexpectedly dropped. We should be able to distinguish a normal packet being dropped from a postcard drop, or at least warn the user about the possibility. In practice, in some cases we can say that a packet is being dropped: cases when we try the above method and it still can not mark all the edges, being the number of unmarked edges not very low. This is an heuristic, but we can better see this intuition by looking at Figure 4.8. This is how NetSheriff sees two of the tested variations of the case presented in Section 4.3.2: one where switch $C1$ does not forward the packets to any of the switches $A3$, $A5$ and $A7$ (but sends the postcard, hence the marked edge $A1 \rightarrow C1$), and other where all switches $A3$, $A5$ and $A7$ do not forward packets to the next switches ($E3$, $E5$ and $E7$, respectively) and also do not forward the respective postcards. Recall that observed paths are constructed from postcards, so if $C1$ drops the packet but not the postcard, we will see the path $E1 \rightarrow A1 \rightarrow C1$ (hosts excluded), and if $A3$, $A5$ and $A7$ drop the packet and the postcards we will see the same result. If they did not drop the packet, but only the postcards, we would see postcards from switches $A3$, $A5$ and $A7$ and could reconstruct the incomplete branches.

Since the accumulation of errors is way bigger in the latter case than in the former, it is safe to say that it is more likely that switch $C1$ is the faulty one. Despite that, since it is not guaranteed that the latter case will not happen, we keep the output of NetSheriff to list the unmarked edges in such cases (of when we cannot extrapolate the observed path in case of possible dropped postcards) and leave this conclusion of what is the most likely scenario for the network operator using NetSheriff.

Table 4.1 summarizes these findings. In the cases where NetSheriff presents pairs of switches instead of just one, we say it has high accuracy. This is because we are not pointing a single switch, but

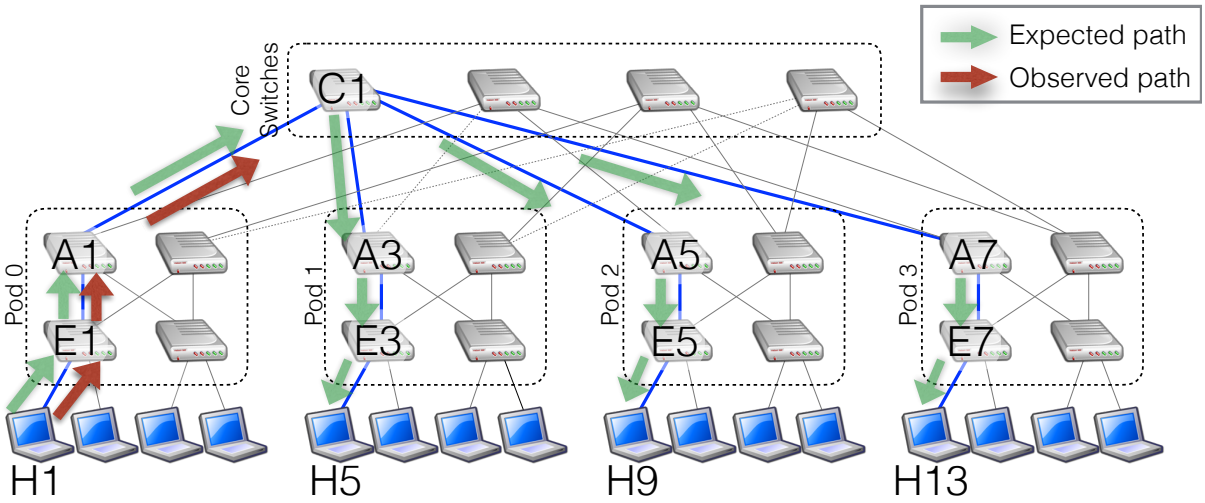

Figure 4.8: One total drop or three different total drops (plus postcard drops)?

still we know that one of those in the pair is the faulty switch, thus we can say we have high accuracy. In the other cases we say it has maximum accuracy.

Another issue is that NetSheriff currently generates false positives in multiple switches in presence of link congestions along a path, due to packet loss (either normal packets or postcards). We consider that this is a error that is easier to diagnose with other tools and therefore did not resolve the issue, but we might consider in future work the task of differentiating these cases from the other types of error.

Error	**Accuracy**
Total unexpected forwarding	Maximum
Partial unexpected forwarding	Maximum
Unexpected partial drop	High
Unexpected total drop	High
Suboptimal routing	High

Table 4.1: NetSheriff's accuracy for different error types, considering the possibility of postcard drops.

4.5 Network overhead

The network overhead induced by postcard collection, as well as the cost associated to their processing, was extensively evaluated by NetSight's authors, and can be consulted in (Handigol, Heller, Jeyakumar, Mazières, and McKeown 2014), and thus we omit that analysis in this dissertation. We let for future work the evaluation of overhead produced by introducing a proxy between the controller and the switches, since that evaluation is only relevant in real networks, and not in the simulated environment we used in this evaluation.

Summary

In this chapter we described the experimental evaluation we made to NetSheriff and its results, detailing the experiments that showed its efficacy in pinpointing the devices responsible for errors in a network. We tested our prototype in an emulated environment and created five simple case studies, each being a different type of error. These experiments have shown that NetSheriff was correctly identifying the error and the responsible switch, assuming that postcards were not dropped. We then proceeded to test more complex cases - error situations in more realistic network configurations. Again, NetSheriff passed these tests. We then relaxed the assumption of no postcard drops, and found that it is sometimes possible to extrapolate the observed path even when postcards are missing. In the remaining situations, we can still narrow down the network error to pairs of switches that have at least one misbehaving switch. Considering that datacenter networks usually have hundreds to thousands of switches, we think that this is still a useful result.

The next chapter finishes this thesis by presenting the conclusions regarding the work developed and also introduces some directions in terms of future work.

5 Conclusions

5.1 Conclusions

In this thesis we proposed and evaluated NetSheriff, an automatic debugging tool for software defined networks. NetSheriff combines formal validation techniques (to obtain the expected paths of packets) and packet recording mechanisms (to obtain the observed paths) with the goal of performing a differential analysis, that allows to identify exactly in which device a fault occurs. We experimentally evaluated NetSheriff with different types of errors, that exercise different aspects of the differential analysis. The results have shown that NetSheriff was able to pinpoint the faulty switch and categorize the error in all tested scenarios, except in some cases of packet drops. Although, even in these cases NetSheriff could detect that it was either one switch dropping packets unexpectedly or other switch(es) dropping both the packets and the respective postcards. Based in these findings, we think this work, with a few further improvements, can be useful to detect and to help in debugging unpredictable faults in hardware and software running in the switches in production networks.

5.2 Future Work

There are a few issues that we have in mind for NetSheriff's improvement in future work.

Although this was not explored in the current prototype, one can extend NetSheriff to validate properties of the paths during a reconfiguration process. As stated before, the postcards generated by NetSight carry a version number of the switch when the packet was forwarded. This allows NetSheriff to associate the observed paths to the corresponding versions of the expected paths, and by doing so we could perform the validations during reconfigurations.

- Currently, link congestions along a path are generating alerts for errors in multiple switches, due to packet loss (either normal packets or postcards). This type of error could be treated differently, leveraging NetSheriff to also deal with excessive packet loss, although it is the less prioritized task for now, since this distinction should be better dealt with other types of tools.

- A cause of network errors is that switches sometimes do not respond correctly to commands, failing to add, modify or delete rules. After detecting the faulty switch, NetSheriff could check the switch's forwarding table and automatically try to resend the command in order to not only detect the error and the faulty switch, but also correcting it.
- The most important, yet more challenging, future work would be to find a way to maintain NetSheriff functionality while reducing network overhead from postcards. After that, the prototype must be evaluated in real networks.

References

Al-Fares, M., A. Loukissas, and A. Vahdat (2008). A scalable, commodity data center network architecture. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*, SIGCOMM '08, New York, NY, USA, pp. 63–74. ACM.

Ball, T., N. Bjørner, A. Gember, S. Itzhaky, A. Karbyshev, M. Sagiv, M. Schapira, and A. Valadarsky (2014, June). Vericon: Towards verifying controller programs in software-defined networks. *SIGPLAN Not. 49*(6), 282–293.

Canini, M., D. Venzano, P. Perešíni, D. Kostić, and J. Rexford (2012). A nice way to test openflow applications. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, Berkeley, CA, USA, pp. 10–10. USENIX Association.

Chen, A., Y. Wu, A. Haeberlen, W. Zhou, and B. T. Loo (2016, August). The Good, the Bad, and the Differences: Better Network Diagnostics with Differential Provenance. In *Proceedings of ACM SIGCOMM 2016*.

Consortium, O. S. et al. (2009). Openflow switch specification version 1.0. 0.

Gude, N., T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker (2008, July). Nox: Towards an operating system for networks. *SIGCOMM Comput. Commun. Rev. 38*(3), 105–110.

Handigol, N., B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown (2012). Reproducible network experiments using container-based emulation. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '12, New York, NY, USA, pp. 253–264. ACM.

Handigol, N., B. Heller, V. Jeyakumar, D. Maziéres, and N. McKeown (2012). Where is the debugger for my software-defined network? In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, HotSDN '12, New York, NY, USA, pp. 55–60. ACM.

Handigol, N., B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown (2014). I know what your packet did last hop: Using packet histories to troubleshoot networks. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, Berkeley, CA, USA, pp. 71–85. USENIX Association.

Heller, B., C. Scott, N. McKeown, S. Shenker, A. Wundsam, H. Zeng, S. Whitlock, V. Jeyakumar, N. Handigol, J. McCauley, K. Zarifis, and P. Kazemian (2013). Leveraging sdn layering to system-

atically troubleshoot networks. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, New York, NY, USA, pp. 37–42. ACM.

Jain, S., A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat (2013, August). B4: Experience with a globally-deployed software defined wan. *SIGCOMM Comput. Commun. Rev. 43*(4), 3–14.

Kazemian, P., G. Varghese, and N. McKeown (2012). Header space analysis: Static checking for networks. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, Berkeley, CA, USA, pp. 9–9. USENIX Association.

Khurshid, A., X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey (2013). Veriflow: Verifying network-wide invariants in real time. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, Lombard, IL, pp. 15–27. USENIX.

Kreutz, D., F. M. V. Ramos, P. Verissimo, C. Esteve Rothenberg, S. Azodolmolky, and S. Uhlig (2014, June). Software-Defined Networking: A Comprehensive Survey. *ArXiv e-prints*.

Kuzniar, M., P. Peresini, M. Canini, D. Venzano, and D. Kostic (2012). A soft way for openflow switch interoperability testing. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '12, New York, NY, USA, pp. 265–276. ACM.

Lantz, B., B. Heller, and N. McKeown (2010). A network in a laptop: Rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, Hotnets-IX, New York, NY, USA, pp. 19:1–19:6. ACM.

McKeown, N. (2011, October). How SDN will shape networking. Available at `https://www.youtube.com/watch?v=c9-K5O_qYgA`.

McKeown, N., T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner (2008, March). Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev. 38*(2), 69–74.

Miranda, J., N. Machado, and L. Rodrigues (2016, September). Isolamento de falhas em redes definidas por software. In *Actas do oitavo Simpósio de Informática (Inforum)*, Lisboa, Portugal.

Perešíni, P., M. Kuzniar, and D. Kostić (2015, August). Rule-level data plane monitoring with monocle. *SIGCOMM Comput. Commun. Rev. 45*(4), 595–596.

Scott, C., A. Wundsam, B. Raghavan, A. Panda, A. Or, J. Lai, E. Huang, Z. Liu, A. El-Hassany, S. Whitlock, H. Acharya, K. Zarifis, and S. Shenker (2014, August). Troubleshooting blackbox sdn control software with minimal causal sequences. *SIGCOMM Comput. Commun. Rev. 44*(4), 395–406.

Shenker, S. (2011, October). The future of networking, and the past of protocols. Available at `https://www.youtube.com/watch?v=YHeyuD89n1Y`.

TechTarget (2016, November). Carriers bet big on open sdn. http://searchsdn.techtarget.com/news/4500248423/Carriers-bet-big-on-open-SDN.

Wundsam, A., D. Levin, S. Seetharaman, and A. Feldmann (2011). Ofrewind: Enabling record and replay troubleshooting for networks. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'11, Berkeley, CA, USA, pp. 29–29. USENIX Association.

Zeng, H., P. Kazemian, G. Varghese, and N. McKeown (2012). Automatic test packet generation. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '12, New York, NY, USA, pp. 241–252. ACM.

Zhou, W., M. Sherr, T. Tao, X. Li, B. T. Loo, and Y. Mao (2010). Efficient querying and maintenance of network provenance at internet-scale. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, New York, NY, USA, pp. 615–626. ACM.