Fault Isolation in Software Defined Networks

(extended abstract of the MSc dissertation)

João Sales Henriques Miranda
Departamento de Engenharia Informática
Instituto Superior Técnico

Supervisor: Professor Luís Rodrigues

***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.**

We propose 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.

I. Introduction

Software Defined Networking (SDN) has recently emerged as an appealing and powerful paradigm for managing networks. First, SDN 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). Second, SDN allows the packet-handling rules of each switch to be configured by a logically-centralized controller, using a standard API (*e.g.,* OpenFlow [1]). The existence of a single central point of control, maintaining a global view of the network and holding a specification of the control-plane configuration, has the potential to strongly simplify network management [5], [4], [2].

Like traditional networks, software defined networks (SDNs) are prone to errors (or *bugs*), such as forwarding loops, black holes, suboptimal routing, and access control violations. One of the advantages brought by SDN is the easiness of designing tools to perform or simplify the tasks of testing, verification and debugging. Recent work has been showing that SDNs' architecture can be leveraged to create such tools [3], [4], [5], [6], [7]. As these networks rely on a software based, logically centralized controller, they can benefit from automatic testing and verification tools used in other fields of computer science to increase controllers' reliability [3], [4] or check whether invariants hold among different layers of the SDN stack [8]. They can also benefit from debugging tools that help network operators to replicate errors in order to identify the causes from errors triggered in execution time [6], [9], [7], [5]. These tools intercept controllers' commands in production or test networks (possibly recording them) and instrument switches to record data packets in order to reproduce the network behavior, possibly in other network, experimenting different executions with the objective of isolating traffic that causes the errors.

Despite these advances, debugging SDNs still remains a daunting task. Hardware and firmware bugs on equipments that make them behave differently than specified continue to generate errors that can only be detected at execution time. Testing and verification tools are not adequate for this type of errors, because they rely on predictable models of the network to identify bugs. To help network operators diagnose and debug this kind of bugs, tools such as OFRewind [6] and ndb [7] typically resort to instrumentation, event logging, and replay mechanisms. Unfortunately, inspecting the event trace in order to isolate the faulty component(s) is usually a time-consuming task [10], [5].

In this paper, we propose NetSheriff, a system that automatically isolates misbehaving switches in SDNs. NetSheriff combines features from both verification tools and debugging tools, with the aim of simplifying the task of finding the root cause of a network error. For each equivalence class of packets traversing the network, NetSheriff uses a model of the network and the commands provided by the controller to first generate the paths that are expected to be followed by packets of the different equivalence classes. Then, at execution time, NetSheriff records the observed paths. Finally, NetSheriff applies a differential analysis to both the expected and the observed paths and checks whether they match or not, identifying the misbehaving switch in case of divergence. Moreover, NetSheriff helps the network operator to diagnose the root cause of the fault and fix the problem in a more timely manner by categorizing it accordingly to patterns in the differences between these two paths.

II. Related Work

Although SDN is still an emerging technology, over the past few years, a number of systems have been proposed to help improve the reliability of SDNs. In this section, we overview some of the prior efforts on this topic that are most related to our work.

Testing and Model Checking: Several tools aim at verifying the correctness of SDN applications. Some take a static approach, performing this tests or verification prior to deployment. For instance, NICE [3] combines model checking and symbolic execution to automatically find errors in SDNs and output a sequence of packets that triggers the error. VeriCon [11] 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, given that the network topology specification and the correctness conditions are written as first-order logic formulas. VeriCon has the advantage of providing the guarantee that a given SDN program is indeed free of errors. In turn, SOFT [12] is a tool designed to find bugs in OpenFlow implementations or interoperability problems among different switch vendors.

Conversely, Veriflow [4], ATPG [13] and Monocle [14] perform dynamic checks. VeriFlow verifies if the data plane holds the invariants specified in the input policies, intercepting the rules from the controlling and performing checks in the network model as they arrive. Monocle uses a similar technique to verify if the switches forwarding tables are being correctly updated as new rules are intercepted. ATPG frequently generates probe packets that verify reachability policies and performance health in the data plane.

NetSheriff contrasts to the aforementioned tools in that it does not strive to check for predictable invariant violations. Instead, NetSheriff focuses on reporting unexpected and incorrect behavior of forwarding devices (potentially due to hardware transient faults).

Post-mortem analysis: The advent of SDN facilitated the development of better tools for network debugging, since the control software is logically centralized, has a global network view and is hardware transparent. As prominent systems for debugging and diagnosis SDN errors, we highlight NetSight [9], OFRewind [6], and STS [5].

NetSight registers the packet histories and offers an interactive network debugger (dubbed *ndb*) that eases the navigation through different states of the network in order to help network operators to identify which sequences of events reproduce the incorrect network behavior.

OFRewind is a debugging tool that allows the record and replay of packets in SDNs. A network operator can then reproduce the error multiple times and, throughout these repeated replays, isolate the component or traffic causing the error. OFRewind also allows partial recording (*e.g.,* track only the control messages or the packet headers), which helps reduce the recording overhead.

Both NetSight and OFRewind provide the means to inspect a sequence of packets that lead to a failure, but offer no clues on what events actually caused the error. The network operators are the ones that have to progressively reduce these sequences in order to isolate the events that cause the error. STS, on the contrary, aims at reducing the effort spent on troubleshooting SDN control software by automatically eliminating from buggy traces the events that are not related to the error. This curated trace, denoted *minimal causal sequence* (MCS), contains the smallest amount of events responsible for triggering the bug. However, STS is not able to detect errors outside the control software (for instance, hardware problems in switches), whereas NetSheriff is. In fact, we believe that NetSheriff can be a useful system to complement current debugging tools. By quickly pinpointing faulty components of the SDN and categorizing the problem accordingly to patterns in the differences between the expected and observed paths for packets, NetSheriff allows network operators to direct their efforts to analyze devices that are indeed relevant to the network misconfiguration.

III. NetSheriff

This section describes NetSheriff. We start by providing an overall description of NetSheriff's architecture and components (Section III-A). We then show how NetSheriff uses features from model checking tools to compute the expected path of a packet (Section III-B), as well as tracing mechanisms to efficiently obtain the respective observed path (Section III-C). We conclude the section by with a description of the main steps for building our prototype (Section III-D).

A. Overview

NetSheriff is a system that detects incorrect traffic paths and automatically pinpoints the network device responsible for the network misconfiguration. NetSheriff comprises four components: the *seer*, the *instrumenter*, the *collector*, and the *checker*. These components, depicted in Figure 1, are described as follows.

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 *forwarding graphs* [4]. 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

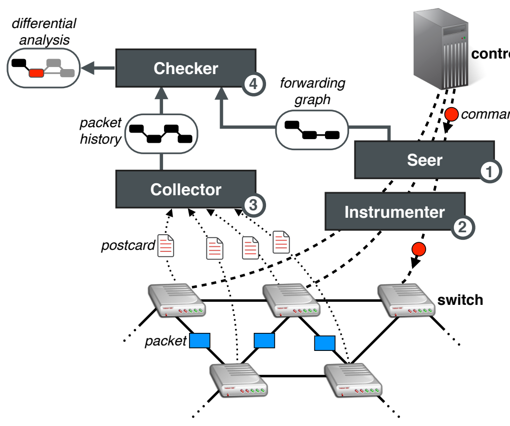

Figure 1. **Overview of NetSheriff.** 1) The *seer* builds forwarding graphs, which are a representation of how packets should be forwarded in the network. 2) The *instrumenter* intercepts the controller's commands and modifies them, so that they additionally instruct switches to create postcards of 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.

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 III-B, we describe how NetSheriff addresses this challenge.

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 [9], 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. We take into account that the postcards themselves can be incorrectly dropped by misbehaving switches, but we will assume they are not until Section IV-D, where we explain how we deal with this problem.

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 [9]. 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.

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 forward 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 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 $A \rightarrow B \rightarrow C \rightarrow D$ (Figure 2b). However, the actual path of the packet ends to be $A \rightarrow D \rightarrow E$ (Figure 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 III-D.

B. 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

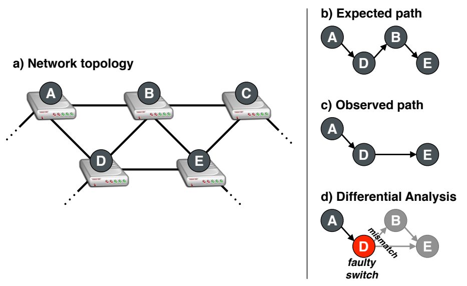

Figure 2. **Differential analysis performed by NetSheriff's checker.** The checker projects the expected path (3.b) against the observed path (3.c) in order to find potential mismatches in the packet flow and pinpoint the faulty switch (3.d).

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 [4], 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.

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 [4].

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

C. 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 [9]. 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.

Generation and capture: NetSight implements the mechanism for packet duplication described in Section III-A. 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.

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 III-A.

D. Implementation

As it is mentioned in Sections III-B and III-C, 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 [15], 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 $\langle$*equivalence class, forwarding graph*$\rangle$, 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.

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 order 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 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 there 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 word, 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 Table I.

IV. Evaluation

This section 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 IV-A). We will then analyze five simple case studies that serve the purpose of demonstrating that NetSheriff can locate the faulty switch in errors from the five categories, presented in Table I, and identify which are the categories of the error (Section IV-B). After that, we will head to slightly more complex cases, yet more realistic (Section IV-C). Finally, we discuss the system's accuracy (Section IV-D) and network overhead (Section IV-E).

A. 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 [15], 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 Table I 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 way, 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 IV-B. By applying different faults on these topologies,, we recreated the different error categories listed in Table I. 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.

B. 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 IV-D

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 can seen in Table I, 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

Name	**Description**	**Examples of possible problems raised**	**Detection**
Total unexpected forwarding	Switch forwards a packet that should be dropped	Access control violations	One or more unexpected edges have origin in vertices without expected exiting edges
Partial unexpected forwarding	Switch forwards a packet to additional ports other than the expected in the configuration	Network congestion, access control violations	At least one expected edge exits from vertices with at least one marked expected edge
Unexpected partial drop	Switch forwards a packet only to a (non-empty) smaller subset of the expected ports	Availability downgrade (less fault tolerance)	A source vertex contains one or more marked expected edges, alongside one or more unmarked expected edges
Unexpected total drop	Switch unexpectedly drops a received packet	No reachability	All the expected edges exiting from a vertex are unmarked
Suboptimal routing	Combinations of the above cases, but the packets reach their correct destinations	Network congestion	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

Table I
CATEGORIES OF ERRORS IDENTIFIED BY NETSHERIFF, ALONGSIDE THEIR DESCRIPTION, POSSIBLE PROBLEMS RAISED BY THEM AND HOW NETSHERIFF DETECTS THEM

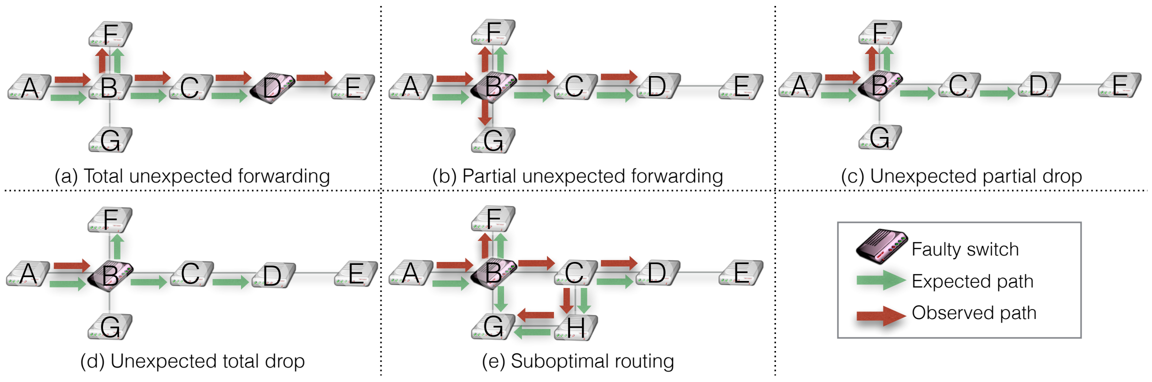
F
A
B
C
D
E
G
H
(a) Total unexpected forwarding
(b) Partial unexpected forwarding
(c) Unexpected partial drop
(d) Unexpected total drop
(e) Suboptimal routing
Faulty switch
Expected path
Observed path

Figure 3. Simple case studies, used in the first part of our evaluation.

locate the error automatically in both situations and also if they are correctly identified.

Total unexpected forwarding: Figure 3a) 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.

Partial unexpected forwarding: This case is represented by Figure 3b). 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.

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 section, 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 **??**), 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).

Unexpected partial drop: In Figure 3c) 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.

Unexpected total drop: Figure 3d) 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.

3) Suboptimal routing: Figure 3e) 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.

C. *Case studies in Fat-Trees*

We now present two network configurations on a very common baseline for datacenter topologies, the *fat-tree* [16], 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 IV-D.

Ignored commands in ECMP routing: The first fat-tree configuration, represented in Figure 4, 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. 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).

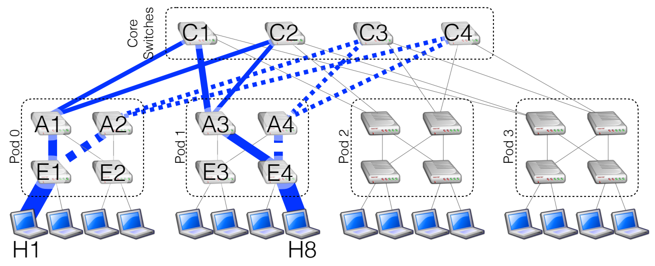

Figure 4. **Load balancing in a fat-tree.** Ideally, the traffic would be equally (in ECMP) splited among the different equal (lowest) cost paths. Between hosts H1 and H8, these paths (with hosts excluded) are: $E1 \rightarrow A1 \rightarrow C1 \rightarrow A3 \rightarrow E4$; $E1 \rightarrow A1 \rightarrow C2 \rightarrow A3 \rightarrow E4$; $E1 \rightarrow A2 \rightarrow C3 \rightarrow A4 \rightarrow E4$ 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.

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 5, and then injected faults to drop packets, partially or totally, reproducing a behavior similar to the unexpected packet drops in the simpler examples (Section IV-B2). 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.

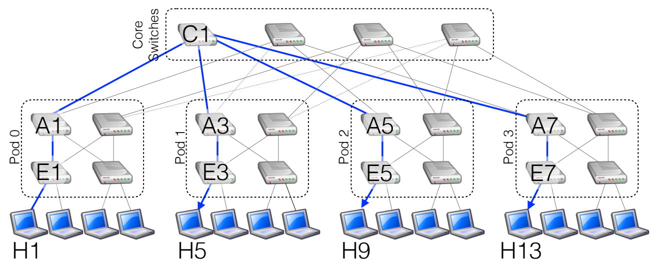

Figure 5. Replication in a fat-tree

D. 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 6. This is how NetSheriff sees two of the tested variations of the case presented in Section IV-C: 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.

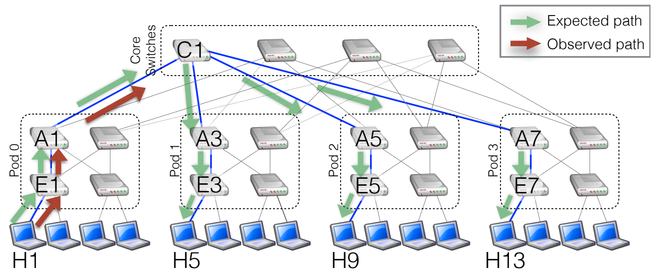

Figure 6. One total drop or three different total drops (plus postcard drops)?

Table II summarizes this section. 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 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 II
NETSHERIFF'S ACCURACY FOR DIFFERENT ERROR TYPES, CONSIDERING THE POSSIBILITY OF POSTCARD DROPS.

E. 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 [9], 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.

V. Conclusions

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.

Acknowledgments

We are grateful to N. Machado for the fruitful discussions and comments during the execution of this work.

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.

References

[1] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: Enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, Mar. 2008.

[2] N. McKeown, "How SDN will shape networking," Available at https://www.youtube.com/watch?v=c9-K5O_qYgA, Oct. 2011.

[3] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford, "A nice way to test openflow applications," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'12. Berkeley, CA, USA: USENIX Association, 2012.

[4] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey, "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: USENIX, 2013.

[5] C. Scott, 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, "Troubleshooting blackbox sdn control software with minimal causal sequences," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, Aug. 2014.

[6] A. Wundsam, D. Levin, S. Seetharaman, and A. Feldmann, "Ofrewind: Enabling record and replay troubleshooting for networks," in *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIXATC'11. Berkeley, CA, USA: USENIX Association, 2011.

[7] N. Handigol, B. Heller, V. Jeyakumar, D. Maziéres, and N. McKeown, "Where is the debugger for my software-defined network?" in *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, ser. HotSDN '12. New York, NY, USA: ACM, 2012.

[8] B. Heller, C. Scott, N. McKeown, S. Shenker, A. Wundsam, H. Zeng, S. Whitlock, V. Jeyakumar, N. Handigol, J. McCauley, K. Zarifis, and P. Kazemian, "Leveraging sdn layering to systematically troubleshoot networks," in *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN '13. New York, NY, USA: ACM, 2013.

[9] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown, "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*, ser. NSDI'14. Berkeley, CA, USA: USENIX Association, 2014.

[10] Cisco Systems Inc., "Spanning tree protocol problems and related design considerations," http://www.cisco.com/c/en/us/support/docs/lan-switching/spanning-tree-protocol/10556-16.html, Aug. 2005.

[11] T. Ball, N. Bjørner, A. Gember, S. Itzhaky, A. Karbyshev, M. Sagiv, M. Schapira, and A. Valadarsky, "Vericon: Towards verifying controller programs in software-defined networks," *SIGPLAN Not.*, vol. 49, no. 6, Jun. 2014.

[12] M. Kuzniar, P. Peresini, M. Canini, D. Venzano, and D. Kostic, "A soft way for openflow switch interoperability testing," in *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, ser. CoNEXT '12. New York, NY, USA: ACM, 2012.

[13] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown, "Automatic test packet generation," in *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, ser. CoNEXT '12. New York, NY, USA: ACM, 2012.

[14] P. Perešíni, M. Kuzniar, and D. Kostić, "Rule-level data plane monitoring with monocle," *SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 4, Aug. 2015.

[15] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: Rapid prototyping for software-defined networks," in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, ser. Hotnets-IX. New York, NY, USA: ACM, 2010.

[16] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*, ser. SIGCOMM '08. New York, NY, USA: ACM, 2008.