NetSheriff: Sheltering Software-Defined Networks from Rogue Switches*

Paolo Laffranchini[1,2][0000−0003−1059−2129],
João Miranda[1], Nuno Machado[3],
Luís Rodrigues[1], Etienne Rivière[2], and Ramin Sadre[2]

[1] INESC-ID, IST, ULisboa, Portugal
{paolo.laffranchini, joaoshmiranda, ler}@tecnico.ulisboa.pt
[2] ICTEAM, UCLouvain, Belgium
{paolo.laffranchini, etienne.riviere, ramin.sadre}@uclouvain.be
[3] Teradata
nuno.machado@teradata.com

Abstract. We present NetSheriff – a system to automatically isolate faulty switches in Software-Defined Networks. To pinpoint the devices responsible for network misbehaviors, NetSheriff performs a differential analysis between expected paths of packets (obtained from a formal model of the network forwarding specification) and the corresponding observed paths taken by flows (obtained through network monitoring). We have built a prototype of NetSheriff supporting both OpenFlow and P4 Programmable devices and evaluated it on different network topologies, simulating real traffic behavior following recent data center studies. Our results show that NetSheriff is able to accurately identify the switch(es) responsible for different types of errors.

1 Introduction

The Software-Defined Networking (SDN) paradigm has emerged as an appealing and powerful approach for simplified network management [23, 28] and troubleshooting [11, 16], by allowing a logically centralized view of the network, through which network operators can apply fine-grained routing policies to network traffic. It provides a clear separation between the *data plane*, a collection of switches in charge of forwarding packets, and the *control plane*, which defines the actual network behavior by installing routing rules specifying how the packet-handling is performed at each switch. The controller entity communicates these rules using a standard API such as OpenFlow [24]. Recent advances

* This work was supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) via projects COSMOS (via the OE with ref. PTDC/EEI-COM/29271/2017 and via the "Programa Operacional Regional de Lisboa na sua componente FEDER" with ref. Lisboa-01-0145-FEDER-029271) and UIDB/50021/2020. Paolo Laffranchini was supported by a fellowship from the Erasmus Mundus Joint Doctorate in Distributed Computing (EMJD-DC) program funded by the European Commission (EACEA) (FPA 2012-0030)

in programmable switching hardware also allow programming custom forwarding behavior and monitoring algorithms via a high-level language such as P4 [6].

The well-defined architecture of SDN along with a clear-cut semantics of the control protocol offers new capabilities for automatic testing and verification of different layers of the SDN stack [12]. A large body of recent work in SDN has been devoted to the development of such tools [1, 7, 10, 15–17, 25, 26, 29]. Despite these advances, debugging SDNs still remains a daunting task. In fact, errors such as forwarding loops, black holes, suboptimal routing, and access control violations can potentially stem from firmware and hardware bugs in the equipment [14, 30] that prevent the forwarding devices from operating as expected. The root causes of these errors are not easily identifiable [13, 21]. Verification tools are not adequate for this type of situations, because they are based on high-level models, derived from static configuration analysis, and rely on predictable models of the network to detect problems. Due to their non-deterministic nature, errors caused by hardware faults can only be detected at runtime. To help network operators identify misbehaving devices, tools such as OFRewind [29], ndb [10] and EverFlow [33] typically resort to instrumentation, event logging, and replay mechanisms. However, inspecting the event trace in order to isolate the faulty component(s) is usually time-consuming [8, 26].

In this paper, we propose NetSheriff, a system that combines features from both verification and debugging tools to automatically isolate faulty switches in SDNs. NetSheriff achieves its goal by comparing the expected path of a flow (computed statically) against the actual route followed in the network (captured dynamically). Concretely, NetSheriff leverages its knowledge of the network model and exploits the forwarding rules issued by the controller to compute the path expected to be followed by the packets as they traverse the network. Then, at runtime, NetSheriff records the actual sequence of switches a packet traverses and performs an analysis between the expected and the observed paths to check whether they match. If a mismatch is detected, NetSheriff searches for the location where the two paths began to diverge and reports the faulty components responsible for the error. Since issues are characterized by peculiar differences between the expected and observed paths, NetSheriff is also able to indicate the nature of the error (*e.g.,* suboptimal routing or black hole). This helps network operators diagnose and fix problems in a more timely manner. We evaluated our prototype of NetSheriff on networks with different topologies and facing different bugs. The results of our experiments show that NetSheriff is able to accurately and efficiently identify faulty switches.

NetSheriff is built upon existing tools and systems, namely VeriFlow [16], NetSight [11], VeriDP [32], and MAFIA [20]. Each of these systems have limitations when used in isolation. VeriFlow cannot detect run-time errors; NetSight and MAFIA can extract debug information from the switches but leave up to the network administrator the work of analyzing the logs to detect the root cause of a bug; VeriDP can only detect a limited number of bugs. The main contribution of our work is to show that, by combining these tools, it is possible to create

a system that detects bugs in a more efficient and comprehensive manner than with previous work.

2 Related Work

Over the past few years a number of solutions have been proposed to improve the reliability of Software Defined Networks. In this section, we overview some of the prior efforts on this topic that are most related to our work.

An important line of research is represented by approaches whose goal is to capture and replay sequences of events in order to reproduce an issue. Interesting systems that follow in this category are NetSight [11], OFRewind [29], STS [26] and EverFlow [33]. OFRewind is a debugging tool that allows the record and replay of packets flowing in the network. A network operator can then reproduce the bug multiple times in an attempt to isolate the root cause. NetSight records the packet histories and offers an interactive debugger (*ndb* [10]) that eases the navigation through different states of the network in order to help network operators identify which sequences of events cause the incorrect behavior. Note that both NetSight and OFRewind offer the means to inspect a sequence of packets that lead to a failure, but do not provide any clue about the switch(es) that actually caused the error. As a consequence, network operators still have to progressively inspect these sequences in order to isolate the problem. EverFlow permits fine-grained recording of specific packets by marking them with a "debug" flag. EverFlow is then able to inject the recorded packet as an exact copy of the original and trace its behavior in the network. Although the granularity of the recording allows for precise identification of errors, operators still need to detect the issues before being able to trace the behavior of the probe in the network. STS [26], 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*, contains the smallest amount of events responsible for triggering the bug. STS is, however, unable to detect errors outside the control software (for instance, hardware problems in switches).

Several tools focus on verifying the correctness of Software Defined Networks [3, 7]. Some follow a *static approach*, usually prior to deployment, performing either symbolic execution of controller and switch implementations or thorough testing against a model of the network. For instance, NICE [7] combines model checking and symbolic execution to automatically discover errors in SDN controller programs, by generating a sequence of carefully crafted packets that triggers the flaw. VeriCon [3] verifies SDN programs at compilation time, validating their correctness not only for any admissible topology but also for all possible sequences of network events. VeriCon has the advantage of guaranteeing that a given SDN program is indeed free of errors. In turn, SOFT [17] is a tool designed to find bugs in OpenFlow implementations or interoperability problems among switches from different vendors. VeriFlow [16] is a real-time approach that intercepts commands issued by the controller and certifies, by

building a static model of the forwarding policy, whether the forwarding rules installed do not violate network invariants. VeriFlow is able to detect flawed configurations in real-time but its outcome only highlights issues intrinsic to the policies, as it does not take into account the actual switch runtime behavior.

Conversely to static tools, systems that employ a *dynamic approach* perform verification of the forwarding behavior of the network at runtime. ATPG [30] frequently generates probe packets that verify reachability policies and performance health in the data plane, however, it does not check the trajectory of probes. In contrast, SDN Traceroute [1] is a tool to collect information about the path taken by probes forged with specific header fields that mimick real data packets in the network. However, it does not provide automatic checking for the whole network model, thus not relieving the user from the burden of checking the correctness of the network configuration. Monocle [25] takes similar concepts a step forward and aims at verifying whether the switch actions over the packets are consistent with the ones intended by the set of rules installed. It does so by periodically generating probes whose goal is to test the appropriate matching of all rules installed in each switch. However, the probes may be treated differently from real packets. PathletTracer [31] and CherryPick [27], instead, perform tagging of switch identifiers directly in the packets to be able to reconstruct their paths. However, they do not perform any verification that packets actually respect the routing policy specified from the controller. VeriDP [32] checks if the routing policies are being applied by having the switches tag packets with their identifier (and input/output port) as they flow. The path a packet takes is directly encoded in its header by means of a Bloom filter updated at each hop. When the packet leaves the network, a report is sent to a centralized collector which then verifies the path. However, it may suffer limitations in the presence of a switch hardware fault which prevents a packet from completing the processing pipeline. In this case, the report for the packet may never be generated and the problem might remain indefinitely in the network.

3 Motivation and Fault Scenarios

To motivate our work, we illustrate how hardware and/or software faults can cause forwarding errors in current networks. These bugs are common reasons for network failures according to a survey of network operators [30].

Inconsistent routing. Due to limited hardware resources (and in particular memory), modern SDN switches typically split forwarding tables between hardware and software [14]. While hardware innovations will eventually enable larger memory capacity, there still exists a fundamental trade-off between the size of the rules table, power consumption, and costs. A hybrid hardware-software implementation of forwarding tables requires, however, the usage of rule eviction policies which may fail to respect dependencies among multiple forwarding rules [14]. Furthermore, some implementations lack features such as rule priority ordering [19, 32]. All of these issues may cause a switch to handle a packet us-

ing an incorrect forwarding rule, thus causing a mismatch between the routing policy known by the controller and its actual execution in the network.

Connectivity loss. The process of reliably updating the network data plane is challenging in SDN [18]. Switches might fail in correctly installing or updating forwarding rules [25], causing a mismatch between the routing behavior and the controller's policy. This issue typically stems from an incorrect execution of the acknowledgment protocol between the switch and the controller [19,32]. An overloaded switch might even indefinitely delay the execution of controller commands [9]. These problems could inevitable cause packet losses or reachability issues, hence degrading performance and the overall quality of service.

4 NetSheriff

NetSheriff is a system that automatically detects incorrect traffic paths and pinpoints the network devices responsible for the network misconfiguration. NetSheriff relies on features from model checking to compute the expected path of a packet, as well as tracing mechanisms to efficiently obtain the respective observed path. NetSheriff supports two different tracing mechanisms. The first, only requiring standard SDN switches, is based on an extension of NetSight [11] and relies on the online generation of *postcards*. The second is designed to work with programmable switches and uses the MAFIA [20] interface to implement a robust P4-based variant of VeriDP [32]. While the first implementation is more general, the second is able to trace paths more efficiently.

NetSheriff comprises four components: (1) the *seer*, (2) the *instrumenter* proxies, (3) the *collector*, and (4) the *checker*. These components, depicted in Figure 1, are described in the following paragraphs.

Seer. The *seer* component in NetSheriff is responsible for computing in real-time the expected path of a packet. The *seer* is built upon VeriFlow [16], which models the network's behavior as a set of *forwarding graphs*. A forwarding graph is a representation of how packets belonging to the same *equivalence class* should traverse the network. Any two packets $p1$ and $p2$ are said to belong the same equivalence class (EC) if and only if, for any network device R, the forwarding action at R is identical for $p1$ and $p2$. Forwarding graphs indicate, therefore, how the traffic is expected to flow across the network. Vertices in a forwarding graph represent switches, while edges model network links and indicate forwarding decisions between pairs of switches. To maintain the network model consistent, VeriFlow intercepts OpenFlow messages exchanged between the controller and the switches and updates forwarding graphs according to the newly installed rules. Results are then reported to the NetSheriff checker, which will store them to perform the differential analysis between expected and observed packet flows.

Instrumenter. After being processed by the seer, OpenFlow messages are then intercepted by the *instrumenter*. The role of the instrumenter is to setup actions in the switches that allow NetSheriff to trace the actual path followed by packets of a given flow. These actions force switches to create, in certain conditions, a

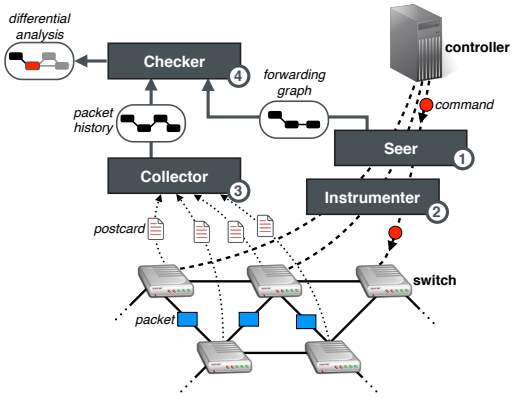

Fig. 1: Overview of NetSheriff building blocks

postcard for an incoming packet that matches an installed rule. These postcards are forwarded to the NetSheriff's collector for analysis. Currently, NetSheriff supports two different technologies to generate postcards when packets of a target flow are forwarded by a given switch. The first is based on NetSight [11] and works with any SDN compliant switch. The second uses the MAFIA [20] interface and only works with programmable switches.

Collector. The collector component consists of a server (centralized or distributed) that receives the postcards sent from the switches and reorganizes them in order to create multiple distinct collections called *packet histories* [11]. The collector can run on the same host as the SDN controller, assuming it is capable to handle the additional load of processing packet histories. A packet history corresponds to the set of all postcards generated by a packet while traversing the network. The packet history allows to (1) reconstruct the path taken by a given packet, and (2) understand which switches performed any header modifications. The propagation of postcards can be performed in two different modes: *in-band* or *out-of-band*. With the former, postcards share the same network used by normal traffic and therefore consume part of the available bandwidth; with the latter postcards are instead routed via dedicated links that connect each switch to the collector, avoiding bandwidth reservation at the expense of additional hardware dedicated to this purpose.

Checker. The checker component is responsible for comparing the expected and observed graphs of packets and signaling unexpected forwarding decisions, pinpointing the switch that misbehaved. To this end, the checker leverages both the forwarding graphs generated by the seer and the packet histories assembled by the collector to perform a differential analysis. The differential analysis consists in projecting the expected path of a packet (given by the forwarding graph) against its actual path (indicated by the packet history) and in checking for potential divergences. If the projection of the two paths yields a perfect match, then the packet was correctly forwarded across the network and NetSheriff does not report any anomaly. On the other hand, when a mismatch is

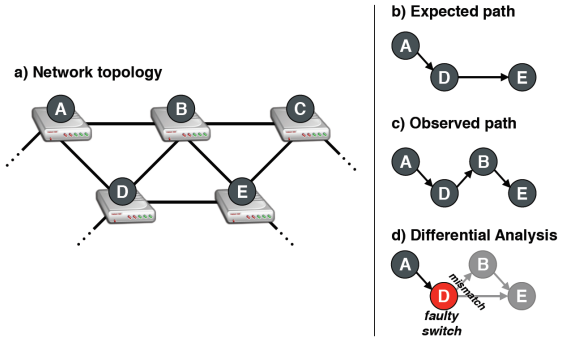

Fig. 2: Differential analysis performed by NetSheriff's checker.

detected NetSheriff reports the issue indicating the switch where the divergence occurred. This switch is then deemed as responsible for the fault. As an example, consider the scenario in Figure 2, which depicts a network of five switches: A, B, C, D, and E and assume that a given packet is expected to be routed from A to E via $A \rightarrow D \rightarrow E$ (Figure 2b). If the actual trajectory of the packet ends to be $A \rightarrow D \rightarrow B \rightarrow E$ (Figure 2c), the checker would be able to notice the mismatch while performing the differential analysis between the two paths and, as a result, report an error identifying switch D as the source of the problem.

Depending upon the pattern of the mismatch in the path projection, NetSheriff is able to report additional information regarding the type of the error, *e.g.,* a black hole or suboptimal routing. Section 5 further details the algorithm allowing to infer the type of error according to the projection. NetSheriff can also handle scenarios where there are multiple correct paths for a given flow (*e.g.,* when using load balancing); however, in the current implementation, we only assume that a single correct path exists. An extension to cover the use of multiple paths is planned for future work.

5 Implementation

We have built NetSheriff's by leveraging upon the features provided by previous works: VeriFlow [16], NetSight [11], VeriDP [32], and MAFIA [20]. In particular, the seer component is built upon VeriFlow, adding the necessary code to communicate the computed graphs to NetSheriff. One version of the instrumenter is based on NetSight. The other is a variant of VeriDP [32] that uses MAFIA [20][4]. The NetSheriff collector and checker are built on top of NetSight's server. Since both VeriFlow and NetSight have been implemented by introducing a proxy layer between the SDN controller and the switches, our implementation follows this approach and connects the two proxies together. Forwarding rules issued by the controller are thus first intercepted by VeriFlow, which uses them

[4] Hybrid setups can also be supported by instrumenting P4 compliant switches using MAFIA code, whereas the remaining switches would be configured using NetSight.

to compute the expected graphs that will then be notified to NetSheriff. Then, the rules are relayed to the instrumenter, that uses NetSight or MAFIA to create the actions required to generate postcards. Only the packet headers are sent to the collector and the payload is stripped. In the current implementation, we install postcard generation rules to track all flows. Clearly, the system is flexible to allow fine-grained monitoring of a selection of flows and reducing network overheads. Postcards generated by switches are sent to the collector which will perform the differential analysis and identify possible forwarding mismatches. In the following, we describe the implementation of these steps.

5.1 NetSight-based Instrumentation

The first implementation of the instrumenter is an extension of NetSight's *Flow Table State Recorder* [11] and has the goal of enabling the capture of the information required to reconstruct the path that packets follow at runtime. To achieve this, it intercepts forwarding rule modifications sent to the switches and augments them with two new actions. The first action instructs the switch to create a *postcard* for any incoming packet that matches the newly installed rule. The postcard consists of a copy of the original packet extended to carry additional information, namely the identity of the switch, an identifier indicating the version of the switch's flow table, and the input port and then truncated to the minimum network packet size. The second action, in turn, instructs the switch to forward the postcard to the NetSheriff's collector. In this case, postcards are created for every packet of the flow being analyzed, and at every switch. The advantage of this implementation is that it works with any SDN compliant switch, although with a non-negligible signaling overhead (discussed in the evaluation).

5.2 MAFIA-based Instrumentation

The second implementation of the instrumenter uses the MAFIA [20] interface to implement a robust variant of VeriDP [32]. This method significantly reduces the number of postcards generated, but only works with programmable switches. This implementation requires switches to have the ability to inspect and change packet fields on the fly, something that is supported by P4 [6] compliant switches.

In this implementation, the instrumenter instructs the ingress switch to *sample* packets of the flow. A fraction of these packets are tagged to be processed by other switches in a horizon that is set to a value from 1 to the maximum path length (different sampled packets use different horizons). Packets that have been tagged will be processed by other switches in their path until the horizon is reached. When the horizon is reached, a postcard is generated and sent to the collector. This simple mechanism allows to implement a simplified version of traceroute in the SDN context [1], without requiring much involvement from the SDN controller and generating way less probing traffic.

In order to setup the action in each switch, we leverage the MAFIA [20] language, that has been designed to simplify the development of monitoring solutions for SDN networks. MAFIA, that stands for Measurements As FIrst-class

```
1  // Code executed at ingress switch
2  flowid = key(ip.src,ip.dest,tcp.src,tcp.dest,ip.proto)
3  // Current horizon values
4  horizons = HashMap(key=flowid, size=1024, type=Counter(width=4));
5  // Per-flow horizon values. Configured by the controller
6  horizons_max = HashMap(key=flowid, size=1024, type=Counter(width=4));
7
8  pkts
9    >> tag(pkt.mafia.monitor, 0x1)
10   >> match(horizons.read() == horizons_max.read()) >> horizons.reset()
11   >> horizons.set(horizons.read() + 1)
12   >> tag(pkt.mafia.horizon, horizons.read()) )
```

Fig. 3: MAFIA code for ingress switches.

```
1  // Code executed at all switches (including the ingress)
2  location = key(pkt.input_port, switch.id, pkt.output_port)
3  trajectory = BloomFilter(alg="membership", nhash=4, key=location, size=16);
4  horizon = Counter(width=4);
5
6  pkts
7    >> match(pkt.mafia.monitor== 0x1)
8    >>
9           (
10            >> trajectory.insert()
11            >> tag(pkt.mafia.path_bf, pkt.mafia.path_bf | trajectory.read()))
12            >> trajectory.reset() )
13          +
14          (  >> horizon.set(pkt.mafia.horizon - 1)
15             >> tag(pkt.mafia.horizon, horizon.read())
16             >> match(pkt.mafia.horizon == 0) >> duplicate(reports) )
17 reports
18     >> collect()
```

Fig. 4: MAFIA code to generate the path Bloom filter.

Artifacts, defines a set of reusable primitive building blocks that can be composed to express measurement tasks in a concise way. MAFIA code associated with the NetSheriff instrumentation described above is presented in Figures 3 and 4. It consists of a brief state declaration and the operations to be executed for each incoming packet. The packets that need to be traced are tagged with two control fields: a Bloom filter that captures the path followed by the packet and a *horizon* field that indicates when a postcard needs to be generated.

Figure 3 depicts the relevant code that runs at the ingress switch. Two hashmaps indexed by the flow 5-tuple (Lines 4 and 6) are used as persistent state about the flow horizons. The former will store the latest value used while the latter will hold its max value as configured by the controller (*e.g.,* the expected path length). The switch sets the *monitor* flag to 1 and sets the *horizon* counter to a value in the interval $[1, path\_length]$. The current horizon value is incremented for each packet (Line 11). Each packet is tagged, therefore, with a different horizon in a round-robin manner.

Figure 4 depicts the relevant code to generate the Bloom filter representation of the packet's path. It is executed at all switches, including the ingresses. Persistent state includes a Bloom filter (Line 3) which is constructed on a per-packet basis and a counter (Line 4) to handle the packet's horizon. Only packets that have the *monitor* flag set to 1 are processed (Line 7), by executing the follow-

ing sequence of actions: first, the trajectory Bloom filter is computed using the switch identifier and the current packet's input and (expected) output port(Line 10); then, its value written back in the packet into the *path_bf* field (Line 11) and finally reset (locally). Afterwards, the *horizon* counter is decremented (Line 14) and, if the horizon has reached the value 0, a postcard is generated (Line 16). Postcards are finally sent to a pre-configured collector (Lines 17–18).

5.3 Postcard Consolidation and Construction of $\mathcal{G}_O$

NetSheriff works by comparing a graph $\mathcal{G}_E$ representing its expected path $\mathcal{P}_E$ and the graph $\mathcal{G}_O$ representing its observed path $\mathcal{P}_O$. The observed path is constructed by consolidating multiple postcards associated with multiple individual packets. Consolidation is mandatory in the MAFIA-based version of NetSheriff because postcards are generated for different packets at different points in the path (depending on the horizon parameter, as described above). Thus multiple postcards need to be combined to create a complete observed path. However, even with the NetSight implementation postcards can be lost in the network and by consolidating postcards generated by multiple individual packets it is possible to obtain a more reliable observation of the actual path.

Postcard consolidation is performed by the collector as follows. First, postcards produced by the MAFIA instrumentation are *unfolded* by creating a separate postcard for every switch included in the Bloom filter, *as if* an individual postcard has been sent by each of these switches. This allows to process MAFIA postcards in the same way as NetSight postcards. Then, postcards associated with the same packet are grouped. The sets of postcards from the same packet are then fed to the algorithm that constructs the observed path. The graph $\mathcal{G}_O$ representing the observed path $\mathcal{P}_O$ is constructed as follows. Let v_i and v_j be two adjacent vertices of the physical network graph (i.e., they represent switches that are directly connected to each other). If in the set of consolidated postcards there is a postcard from v_i *and* one from v_j, such that both postcards belong to the same packet, then both v_i, v_j, and the edge between these vertices are added to $\mathcal{G}_O$. This process is performed for all adjacent vertices.

5.4 Differential Analysis

Depending upon the divergences between the expected and the observed paths, NetSheriff categorizes different forwarding errors. Routing flaws appear as peculiar patterns in the observed graphs when compared to the expected one. Consider, for a packet of a given EC, a graph $\mathcal{G}_E$ representing its expected path $\mathcal{P}_E$ and the graph $\mathcal{G}_O$ representing its observed path $\mathcal{P}_O$. We can distinguish four macro-situations:

1. $\mathcal{G}_E \equiv \mathcal{G}_O$: if the expected and observed graphs match, then the packet traversed the network topology correctly and no error is reported.
2. $\mathcal{G}_E \supset \mathcal{G}_O$: if the observed path is a subgraph of the expected, then packets have been incorrectly dropped at some switch. This typically captures the existence of a black hole in the network.

3. $\mathcal{G}_E \subset \mathcal{G}_O$: if the expected path is a subgraph of the observed, then a switch might have forwarded a packet to a switch it was not expected to. This error may cause an access control violation as packets may reach a protected network segment and/or network congestion since switch and link resources may be used by unexpected flows.
4. $\mathcal{G}_E \not\equiv \mathcal{G}_O$: if the observed path is not equivalent to the expected, meaning that they diverge in some vertex, then the packet traversed the network via switches it was not supposed to use. The packet might however reach its destination, although not along the path that was defined by the rules defined by the controller. This may cause suboptimal routing in the network, possibly affecting its overall performance and efficacy.

6 Evaluation

We evaluated NetSheriff's prototype in terms of its efficacy in identifying faulty switches and differentiate multiple types of errors. The experiments were performed on an Intel i7-720QM with 8GB RAM DDR3, 250 GB SSD and Ubuntu 14.04, using Mininet [22]. To evaluate NetSheriff's efficiency in detecting errors, we randomly injected different faults in switches.

We evaluate the impact of NetSheriff's instrumentation overhead, *i.e.,* how long it takes to compute expected paths and configure the switches tables and actions, in terms of the delay to setup a TCP connection. Finally, we measure the efficiency at which NetSheriff is able to process packet histories to detect forwarding errors. Unless otherwise specified, we employ simple linear topologies for the microbenchmarks since the number of hops in a path is the only factor affecting the experiments. Different topologies variate features such as the number of back-up paths or the average network diameter, but NetSheriff mechanisms are independent of these characteristics.

6.1 Proxy Overhead

When a new path is setup, NetSheriff requires SDN commands to be intercepted by the seer and by the instrumenter, such that the expected graph is extracted and the actions to generate postcards are installed at the switches. In the case of the MAFIA implementation, postcard generation code is embedded in the P4 MAFIA program loaded on the switch but requires initial state (*e.g.,* flow horizons) to be set up by the measurement controller. We measure the delay of performing these steps by establishing a TCP connection between two hosts and monitoring the TCP handshaking phase. Figure 5 depicts the average overhead across ten runs (variance negligible) for the various approaches. As it can be seen, the overhead increases with the number of switches. This happens because, in the current prototype, the deployment of the required rules at each individual switch in the path is done sequentially.

Thus, the longer the path, the larger the impact of both a non-instrumented system and of a system using NetSheriff during routing policy changes (*e.g.,* new

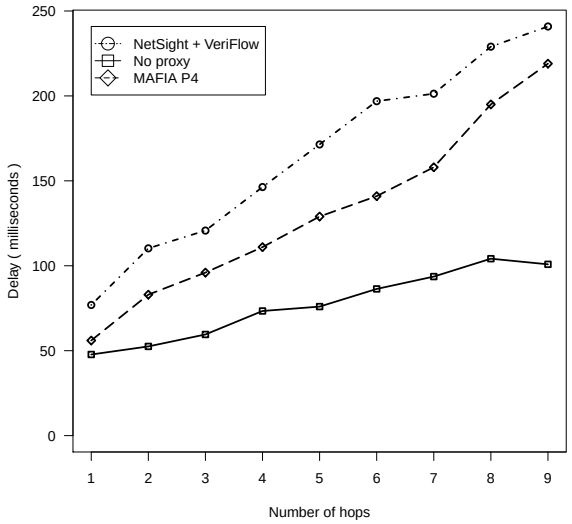

Fig. 5: The impact of NetSight and NetSheriff proxies on TCP handshaking. Both include VeriFlow's graph computation times.

flow setup or steering), because both forwarding graph and switch configuration need to be updated. Note that even in a system without instrumentation, the setup of a path also grows proportionally with the number of hops. In the figure, it is possible to see that the NetSight implementation is slower than the MAFIA implementation, because with NetSight the postcard generation actions are generated and deployed on the fly while the corresponding code is pre-installed in the MAFIA implementation.

6.2 Performance of the Differential Analysis

To evaluate the performance of NetSheriff while processing the postcards it receives, we conducted various experiments recording the time it takes to fully check the history of a packet. We vary the path length up to a maximum of ten switches, which is a fair upper bound for the average diameter of most modern networks, with topologies that can even be optimized down to a diameter of 2–3 [5]. Figure 6 shows the CDF of the processing latency of one million packets injected into the network. The history processing time stays below 6 microseconds for paths up to 5 hops, while it may require up to 10 and 15 microseconds as the length increases to 7 and 10 switches, respectively. For fat trees topologies, which experience an average path length less than 5 hops, hence, the verification throughput stays between $14 \cdot 10^6$ and $16 \cdot 10^6$ histories per second in 90% of the cases. However, as our implementation is still non-optimized and single-threaded, we expect a higher verification throughput in the future.

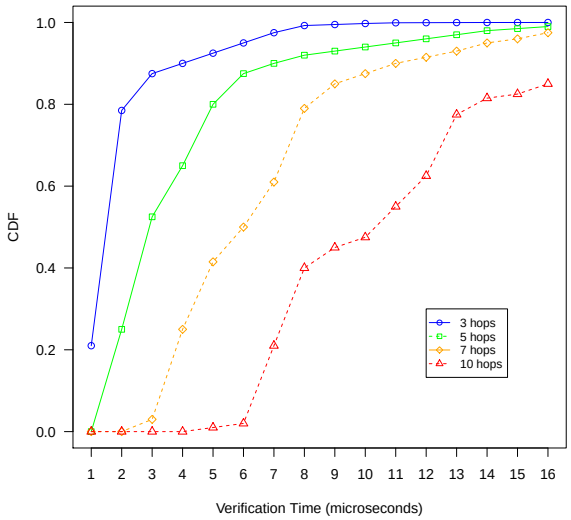

Fig. 6: CDF of the history processing delay in NetSheriff.

6.3 Error Identification

We have run an extensive evaluation of the tool, by manually injecting several types of faults in different configurations of fat-tree network topologies [2], commonly adopted in data centers. Figure 7 illustrates the forwarding errors generated by the faults (using a simplified network view). The faults were injected by modifying directly the forwarding tables in the switches. In all cases, NetSheriff was able to pinpoint the faulty switch responsible for the errors.

Table 1 shows the error detection latency for the various bugs. To detect a routing error, in a NetSight-based instrumentation, we need to wait for a single packet to traverse its path consisting of N hops (NT_d) and for the corresponding postcards to arrive at the collector (T_c). The MAFIA-based instrumentation adds an additional $N\delta$ to the latency, because it needs to collect postcards from N packets, until a complete path is formed (each packet generates a postcard at a different hop). The latency for detecting partial and total drops is dominated by a timeout (which can be configured), since postcard generation is interrupted after the misbehaving hop. If the network gets congested due to a spike in traffic, postcards might get delayed and received after the timeout is expired, causing false positives; this can be easily solved by raising an alarm after multiple, subsequent packet histories indicate the presence of a forwarding error.

6.4 Overhead

Figure 8 shows the traffic increase due to the postcard generation. We use the same metric as in the original NetSight paper [11]. The plot shows the traffic

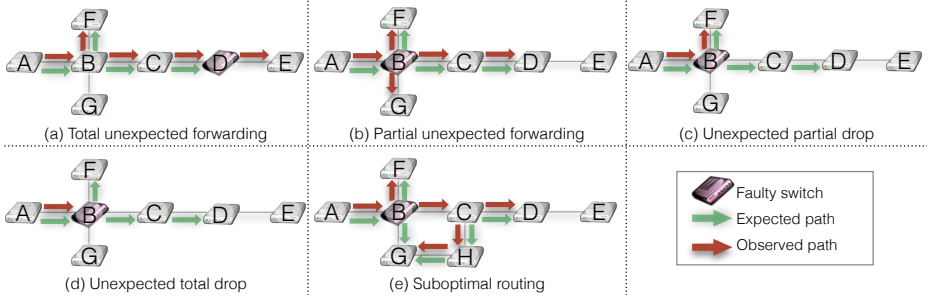

Fig. 7: Forwarding bugs detected by NetSheriff.
Green arrows depict the expected paths; red arrows the faulty observed paths.

Bug	NetSheriff	
	NetSight-based	Mafia-based
Routing errors	$N\,T_d + T_c$	$N\delta + N\,T_d + T_c$
Partial/Total drop	$N\,T_d + T_c + Timeout$	$N\delta + N\,T_d + T_c + Timeout$

Table 1: Error detection latency (N: correct path length; T_d: single hop latency; δ: inter-packet arrival time; T_c: time for a postcard to reach the collector).

increase in percentage for the two different versions of NetSheriff, namely the NetSight based implementation (lines tagged as "NS") and the MAFIA based implementation (lines tagged as "M"). Naturally, the overhead is a function of the average packet size in the network and the size of the postcard. Both implementations generate postcards that are truncated to the minimum network packet size, which in this case is 64 Bytes. We have also considered three different average network packet sizes, namely 1,031 Bytes, as reported in the NetSight paper [11], 850 Bytes, a value observed in datacenters [4], and a network with a smaller average packet size of 576 Bytes. In the x-axis we varied the number of hops of the flow path. As expected, with the NetSight implementation, traffic increase is larger for paths with larger number of hops, given that a postcard is generated at every hop for every packet. Conversely, in the MAFIA implementation, a single postcard is generated for each data packet at deterministic points in the path, as deep as the current horizon value for the flow. Thus, the overhead of the MAFIA implementation is below 12%, even in networks with a small average packet size. The overhead for the NetSight implementation is 31% for a network with average packet size of 1,031 Bytes and paths of 5 hops (as detailed by Handigol *et al.* [11]), but can be substantially larger in networks with smaller average packet size.

Table 2 shows the additional switch resources consumed by NetSheriff. If NetSight-based, instrumenting the switch to generate the postcards requires 4 additional OpenFlow actions (3 tagging, 1 sampling); we couldn't characterized the amount of stages required because it varies between OpenFlow implementa-

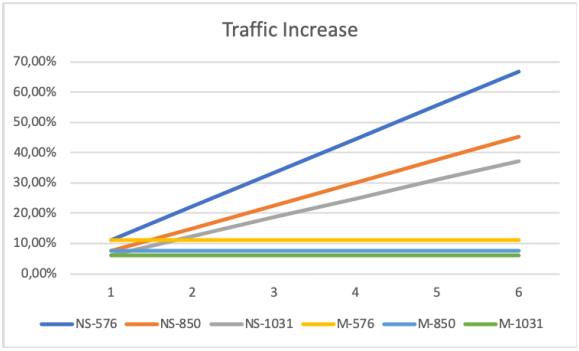

Fig. 8: Traffic increase due to postcards.

Overhead	NetSheriff	
	NetSight-based	MAFIA-based
Stateful computations	4 OpenFlow actions	4 pkt manipulations + 14 mem. read/write
Pipeline stages	NA	7

Table 2: NetSheriff processing overhead.

tions. If MAFIA-based, the resource overhead consists of 4 packet manipulations actions (same as OpenFlow), plus 14 stateful memory read/write associated to the generation of the path bloom filter and horizon updating. These actions are spread across 7 stages of the P4 switch pipeline. The resource overhead for the MAFIA case is low given its capability to drastically reduce postcard traffic.

7 Conclusions

We proposed and evaluated NetSheriff, an automatic debugging tool for SDN. NetSheriff combines formal validation techniques and packet recording mechanisms with the goal of monitoring the consistency between the forwarding behavior and the policies defined by the SDN controller. NetSheriff works with standard SDN switches and with programmable switches, offering different coverage/efficiency trade-offs. We experimentally evaluated NetSheriff with different types of errors, showing that NetSheriff was able to pinpoint the faulty switch and categorize the errors.

Acknowledgments We thank the anonymous reviewers and Elad Schiller for their constructive feedback, that allowed us to improve this paper.

References

1. Agarwal, K., Rozner, E., Dixon, C., Carter, J.: SDN Traceroute: Tracing SDN Forwarding Without Changing Network Behavior. In: HotSDN (2014)
2. Al-Fares, M., Loukissas, A., Vahdat, A.: A scalable, commodity data center network architecture. In: SIGCOMM (2008)
3. Ball, T., Bjørner, N., Gember, A., Itzhaky, S., Karbyshev, A., Sagiv, M., Schapira, M., Valadarsky, A.: Vericon: Towards verifying controller programs in software-defined networks. SIGPLAN Not. (2014)
4. Benson, T., Anand, A., Akella, A., Zhang, M.: Understanding data center traffic characteristics. SIGCOMM Comput. Commun. Rev. **40**(1), 92–99 (Jan 2010)
5. Besta, M., Hoefler, T.: Slim fly: A cost effective low-diameter network topology. In: SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (2014)
6. Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G., Walker, D.: P4: Programming Protocol-independent Packet Processors. SIGCOMM Comput. Commun. Rev. **44**(3) (2014)
7. Canini, M., Venzano, D., Perešíni, P., Kostić, D., Rexford, J.: A NICE Way to Test Openflow Applications. In: NSDI (2012)
8. 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 (2005)
9. Curtis, A.R., Mogul, J.C., Tourrilhes, J., Yalagandula, P., Sharma, P., Banerjee, S.: DevoFlow: Scaling Flow Management for High-performance Networks. In: SIGCOMM (2011)
10. Handigol, N., Heller, B., Jeyakumar, V., Maziéres, D., McKeown, N.: Where is the Debugger for My Software-defined Network? In: HotSDN (2012)
11. Handigol, N., Heller, B., Jeyakumar, V., Mazières, D., McKeown, N.: I Know What Your Packet Did Last Hop: Using Packet Histories to Troubleshoot Networks. In: NSDI (2014)
12. Heller, B., Scott, C., McKeown, N., Shenker, S., Wundsam, A., Zeng, H., Whitlock, S., Jeyakumar, V., Handigol, N., McCauley, J., Zarifis, K., Kazemian, P.: Leveraging SDN Layering to Systematically Troubleshoot Networks. In: HotSDN (2013)
13. Hendriks, L., Schmidt, R., Sadre, R., Bezerra, J., Pras, A.: Assessing the quality of flow measurements from openflow devices. In: TMA (2016)
14. Katta, N., Alipourfard, O., Rexford, J., Walker, D.: Cacheflow: Dependency-aware rule-caching for software-defined networks. In: SOSR (2016)
15. Kazemian, P., Chang, M., Zeng, H., Varghese, G., McKeown, N., Whyte, S.: Real Time Network Policy Checking Using Header Space Analysis. In: NSDI (2012)
16. Khurshid, A., Zhou, W., Caesar, M., Godfrey, P.B.: VeriFlow: Verifying Network-wide Invariants in Real Time. In: HotSDN (2012)
17. Kuzniar, M., Peresini, P., Canini, M., Venzano, D., Kostic, D.: A SOFT Way for Openflow Switch Interoperability Testing. In: CoNEXT (2012)
18. Kuzniar, M., Peresini, P., Kostić, D.: Providing reliable fib update acknowledgments in sdn. In: CoNEXT (2014)
19. Kuźniar, M., Perešíni, P., Kostić, D.: What you need to know about sdn flow tables. In: PAM (2015)
20. Laffranchini, P., Rodrigues, L., Canini, M., Krishnamurthy, B.: Measurements as first-class artifacts. In: IEEE InfoCom (2019)

21. di Lallo, R., Gradillo, M., Lospoto, G., Pisa, C., Rimondini, M.: On the practical applicability of sdn research. In: NOMS (2016)
22. Lantz, B., Heller, B., McKeown, N.: A network in a laptop: Rapid prototyping for software-defined networks. In: HotNets IX (2010)
23. McKeown, N.: How SDN will shape networking. https://www.youtube.com/watch?v=c9-K5O_qYgA (Oct 2011)
24. McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., Turner, J.: OpenFlow: Enabling Innovation in Campus Networks. SIGCOMM Comput. Commun. Rev. (2008)
25. Perešíni, P., Kuźniar, M., Kostić, D.: Monocle: Dynamic, Fine-grained Data Plane Monitoring. In: CoNEXT (2015)
26. Scott, C., Wundsam, A., Raghavan, B., Panda, A., Or, A., Lai, J., Huang, E., Liu, Z., El-Hassany, A., Whitlock, S., Acharya, H., Zarifis, K., Shenker, S.: Troubleshooting Blackbox SDN Control Software with Minimal Causal Sequences. In: SIGCOMM (2014)
27. Tammana, P., Agarwal, R., Lee, M.: CherryPick: Tracing Packet Trajectory in Software-defined Datacenter Networks. In: SOSR (2015)
28. Wang, R., Butnariu, D., Rexford, J.: Openflow-based server load balancing gone wild. In: Hot-ICE'11 (2011)
29. Wundsam, A., Levin, D., Seetharaman, S., Feldmann, A.: OFRewind: Enabling Record and Replay Troubleshooting for Networks. In: ATC (2011)
30. Zeng, H., Kazemian, P., Varghese, G., McKeown, N.: Automatic Test Packet Generation. In: CoNEXT (2012)
31. Zhang, H., Lumezanu, C., Rhee, J., Arora, N., Xu, Q., Jiang, G.: Enabling Layer 2 Pathlet Tracing Through Context Encoding in Software-defined Networking. In: HotSDN (2014)
32. Zhang, P., Li, H., Hu, C., Hu, L., Xiong, L., Wang, R., Zhang, Y.: Mind the Gap: Monitoring the Control-Data Plane Consistency in Software Defined Networks. In: CoNEXT (2016)
33. Zhu, Y., Kang, N., Cao, J., Greenberg, A., Lu, G., Mahajan, R., Maltz, D., Yuan, L., Zhang, M., Zhao, B.Y., Zheng, H.: Packet-Level Telemetry in Large Datacenter Networks. In: SIGCOMM (2015)