# Fault Isolation in SDN Networks

João Sales Henriques Miranda
joaoshmiranda@tecnico.ulisboa.pt

Instituto Superior Técnico
(Advisor: Professor Luís Rodrigues)

**Abstract.** Software Defined Networking (SDN) is a novel paradigm for managing the computer networks. By supporting a logically centralized point of control, it has the potential for simplifying network management. For instance, it makes easier to use tools that analyse SDN configurations and automatically validate that these configurations can preserve target invariants on the data plane. However, such tools cannot cope with unpredictable hardware and software faults that occur in run-time. Thus, tools that help to localize faults in SDN networks are still required. This work makes a survey of the state of the art in debugging tools for SDN networks and proposes a research plan that aims at improving those tools.

## 1 Introduction

Software Defined Networking (SDN) [1] is a novel paradigm for managing computer networks. It provides a clear separation between the data plane (in charge of forwarding packets among devices) and the control plane (which defines how packets should be forwarded). Furthermore, it defines an architecture that allows the control functions on each device to be configured by a logically centralized controller. The availability of a single central point of control, that holds a specification of the control-plane configuration, has the potential to strongly simplify network management [2–4].

Because of the simple ways to add or change network functionalities introduced by SDN, controlling networks using software also makes it easy to introduce new bugs. On the other hand, SDN also has potential to simplify the testing, verification and debugging tasks. There are two main sources of errors in an SDN network. One consists of faulty SDN configurations, that may generate routing loops or black holes when deployed. A significant amount of research has been performed in recent years in tools that aim at detecting and helping in correcting this type of faults. Tools such as NICE[5] or VeriFlow[6] can analyse an SDN program by manipulating models of network configurations, producing possible configurations and verifying that a number of target invariants are not violated under these configurations. The other source of errors are faults in the software that runs on the networking devices or in the hardware itself. These faults are typically unpredictable and occur at run-time. To help network managers to detect and to correct these faults, tools such as OFRewind[7] or ndb[8] typically resort to instrumentation, logging of events, and replay mechanisms. Unfortunately, as it will be clear later in the text, fault localization may be difficult even with these tools.

In this report we make a survey of the most relevant tools that have been designed to support the validation and debug of SDN networks. Based on the advantages and disadvantages of these systems, we propose the construction of a tool that combines features of verification tools and debugging tools, with the aim of simplifying the task of finding the root cause for a network misbehaviour. This would be achieved by producing a differential analysis between the expected paths to be taken by packets in a correct network and the observed paths in a faulty network.

The rest of the report is organized as follows. Section 2 briefly summarizes the goals and expected results of our work. In Section 3 we present all the background related with our work. Section 4 describes the proposed architecture to be implemented and Section 5 describes how we plan to evaluate our results. Finally, Section 6 presents the schedule of future work and Section 7 concludes the report.

## 2 Goals

The Software Defined Networking (SDN) paradigm has several advantages, including the potential for simplifying network management. Part of these benefits result from having a single point of control, which simplifies the task of validating the correctness of the network configuration. Still, transient or permanent faults at the hardware level can cause networks to fail, and tools are needed to help system managers to find the root cause for the observed errors. In particular, transient faults may create unforeseen misconfigurations that may be hard to pinpoint. With this goal in mind, several tools have been developed to log and replay both data and control traffic in SDN networks.

> *Goals:* Our work aims at helping system managers to find the root cause for errors that may result from misconfigurations in SDN networks caused by transient faults. For this purpose we plan to combine tools that are able to automatically infer the traffic flows that are expected to result from a concrete SDN configuration and compare them with the flows observed in production. We expect that the differential analysis between the expected and observed flows can help in pinpointing which component is faulty or misconfigured.

Our approach to achieve this goal leverages on recent tools for the management and debugging of SDN networks. Some of these tools are able to compute offline the expected path of any packet in a given network configuration, and other tools are able to record the paths followed by packets in production. We aim to detect the point of divergence between these two paths. In the presence of faulty or misconfigured components, our tool should be able to identify the root cause for the network misbehaviour.

The project will produce the following expected results.

*Expected results:* The work will produce i) a specification of the algorithms to compute the expected path of a packet in a given configuration, to record the actual path, and to compare these two paths in order to find a point of divergence; ii) an implementation of a prototype of the system, and iii) an experimental evaluation resorting to the injection of permanent and transient faults in network components.

## 3 Related Work

Our approach to achieve the goals presented in section 2 is based on ideas presented in previous work in the areas of testing, verification, and debugging of SDN networks. Testing and verification approaches try to validate programs in respect to previously specified target invariants. Debugging approaches are tailored to solve problems as they appear.

The remaining of the section is organized as follows. Section 3.1 introduces the concept of Software Defined Networking (SDN). Section 3.2 presents OpenFlow which is one of the existing APIs to manage switches in SDN. Section 3.3 introduces the concepts of testing, verification, debugging and emulation in the context of SDN. Section 3.4 identifies common errors in SDN. Section 3.5 presents the testing and verification approaches to find such errors. Section 3.6 presents debugging approaches. Section 3.7 presents the emulation approach. Finally, section 3.8 compares the previously presented solutions.

### 3.1 Software Defined Networking

Traditional networks are based on specialized components bundled together in the networking devices. Examples of these devices are routers, switches, firewalls or Intrusion Detection Systems (IDS). Each of these devices has a specific role and requires individual configuration (that is vendor specific). A network can have multiple devices that, despite being configured independently, are expected to cooperate to achieve a common goal. To ensure a consistent configuration across all devices in a traditional network can be extremely challenging[2].

Furthermore, traditional network protocols, in particular routing protocols, have been designed to be highly decentralized and to be able to operate without a unique point of failure. This decentralization has advantages but also makes hard to reason about the behavior of the system in face of dynamic changes to the workload, re-configurations, faults, and assynchrony. Finally, traditional networking components are *vertically integrated*, i.e., they combine data forwarding functions with control functions in a way that makes hard to analyse each of this aspects in isolation. In addition to the complexity refered above, this vertical integration leads to high upgrade cost (because updating software implies buying new and expensive hardware) and very slow pace of innovation due to larger hardware development cycles (when compared to software development cycles)[2, 3].

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

Software Defined Networking (SDN) is an emerging paradigm that makes this separation possible. Given the strong potential of SDN to simplify network management, the paradigm has gain significant support and adoption from many large companies such as Google, AT&T and Yahoo[3]. In an SDN network, the network devices can be simplified, becoming simple packet forward elements with a common open interface to upper layers (e.g. OpenFlow), and all the control logic is pulled to a logically centralized point of the network. Software Defined Networking is defined by three main abstractions[4]: the *forwarding abstraction*, the *distribution abstraction*, and the *configuration abstraction*. To illustrate these abstractions and the SDN planes we use Figure 1 that is extracted from [9]. The interfaces used and offered by the control plane are refered, respectively, as southbound API and northbound API.

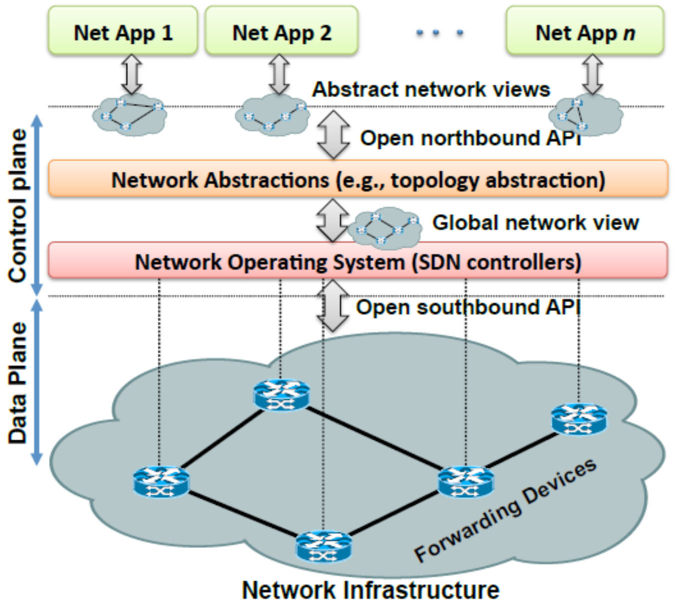

**Fig. 1.** SDN architecture and its fundamental abstractions (picture taken from [9])

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

### 3.2 OpenFlow

OpenFlow is one of the most used soutbound APIs (see Figure 1). As stated in [10], when the OpenFlow idea was developed, the main goal was to allow researchers to experiment new ideas for network protocols in existing network equipment (typically from diferent vendors) by providing a standardized interface to manipulate the devices' forwarding tables. Vendors could provide this interface without exposing the internal working of their devices. This fact is relevant because vendors would not easily open their algorithms and protocols since just by opening them they would lower the barrier to entry for new competitors, for example.

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

An OpenFlow network is composed by OpenFlow switches. These switches implement a flow table, a secure channel connecting the switch to a controller and the OpenFlow protocol, that allows to send events to the controller and to receive packets or commands to manipulate the flow table. Figure 2 (taken from [10]) represents this model of an OpenFlow switch. These switches may be either dedicated OpenFlow switches or OpenFlow-enabled switches. We now describe these two types of switch.

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

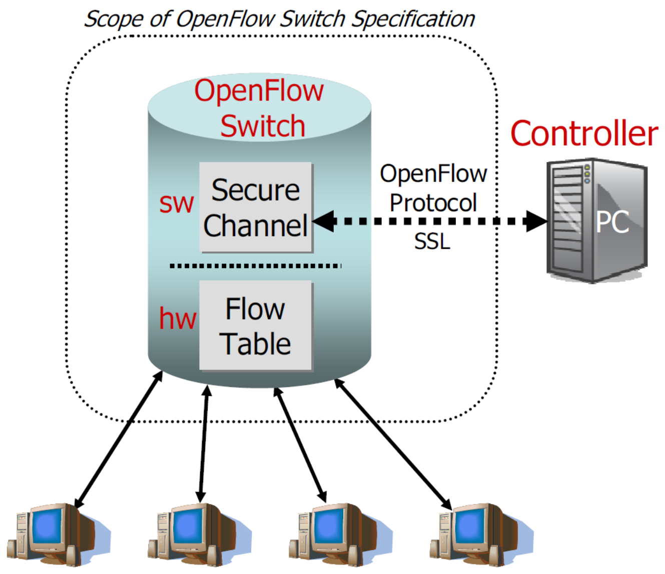

**Fig. 2.** Idealized OpenFlow switch (picture taken from [10])

is done by a secure channel, using the OpenFlow Protocol. The possible actions in a dedicated OpenFlow switch are:

– drop the packet, e.g., for security reasons
– forward the packet to a given port
– encapsulate the packet and forward it to the controller. This is typically used for the default action, when a packet does not match any rule and corresponds to asking the controller where the packet should be sent to.

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

– forward packet to the normal processing pipeline (layers above data link layer in the network protocols stack).

The four actions that were briefly described here are detailed in the OpenFlow switch specification [11], along with the OpenFlow protocol. We now describe the most relevant types of messages of the protocol:

– *packet_in*: sent from switches to the controller when a received packet that does notmatch any rule;
– *packet_out*: sent from the controller to switches when the controller has to send a packet through the data plane;
– *flow_mod*: sent from the controller to switches. Used to update a flow table entry (add, modify or delete);
– *port_status*: sent from switches to controller to inform about a port that was added, modified or removed.

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

## 3.3 Helper tools for SDN programs development

Prototyping, testing, verification, and debugging play important roles in software development, and this is no different in the development of SDN programs. These activities are so important and recurrent that there are multiple tools available to automate parts of them. Examples of these tools will be described in detail in sections 3.5 to 3.7. The remainder of this section will briefly introduce each of these activities.

Testing is unavoidable, since there is no way to know if a system works as expected without testing it. Testing can target functional requirements or system qualities (such as performance or availability). Despite being helpful, typically testing cannot guarantee the absence of bugs (even if it is exhaustive). When software testing is not enough, it is possible to apply verification, i.e. verify if the implementation satisfies the specification. This requires that the specification is formally written and it still does not guarantee correctness because the specification cannot be validated automatically - there is always human intervention in specification writing and validation. In other words, verification finds bugs that fit in the first two of the categories of bugs enumerated in [12]. These two categories are:

– The software does not do something that the product specification says it should do;
– The software does something that the product specification says it should not do.

Section 3.5 will present correctness testing of SDN control programs. It will also present tools that perform verification of these programs, either in offline mode or in online mode. Offline tools check target network invariants before deployment, in a set of possible topologies, or in all admissible topologies. Online tools perform this check in run time, before each forwarding rule is updated (added, modified or deleted).

Testing/Verification and debugging are closely related since the former are means to find software failures (also called bugs, defects, problems, errors, and other terms) and the latter is the process of finding the cause(s) of these failures in order to fix them. Debugging happens during development and during production, since deploying software without bugs is, in most cases, an utopic goal, specially when software starts growing in complexity. Section 3.6 will present examples of tools that aid the debugging activity by recording, processing and replaying traces of events and packet flows in an SDN network.

Prototypes are faster and cheaper ways to validate ideas than full implementation. To evaluate SDN program prototypes, it would be better in many situations to use an emulated environment rather than a real network, because it is faster to configure software in a single computer than setup hundreds of real devices, and also because these devices might be running production software that operators will probably not like to stop. Note that using emulated networks is useful for evaluating prototypes of network applications but also prototypes of testing tools, online verification tools and debugging tools. They are also useful for performing the tests, verifications and debugging themselves, for the same reasons mentioned above. Consider, for example, that a bug is reported in production: if the network is still operational, debugging could be performed in a separate (emulated) network without having to shutdown the production network or use additional network devices. Section 3.7 will present an example of such emulation tool.

### 3.4 Common errors in SDN networks

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

- **Forwarding loops:** are always undesirable because it means that packets are travelling around and will not reach any destination. There are some situations, though, that networks might tolerate loops during a state transition. Loops can be detected by verifying if a packet traverses a switch more than one time.
- **Black holes:** are bugs that lead to packet loss because the packets do not match any rule.
- **Suboptimal routing:** there are (almost) always multiple possible paths to connect two devices. Even if packets are reaching their destination, they might be incorrectly forwarded if they are routed through paths that take more time than others.
- **Access control violations:** happen when a host can connect to another host in a separate VLAN.
- **Anomalous forwarding:** these errors are characterized by the fact that packets are not being correctly forwarded according to the specification. This might occur, for example, because a forwarding rule is not correctly installed.

## 3.5 Testing and Verification of Software Defined Networks

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

Despite these challenges, testing SDNs for errors is of paramount importance to reduce the likelihood of failures after deployment. In fact, tests allow not only to improve the reliability of the SDN, but also help developers to fix the underlying bug in a more timely manner, by providing concrete traces of packets that trigger errors.

Testing can refer to multiple properties such as correctness, performance, availability or security. We now overview some tools that aim at testing SDN control software for correctness. To this end, these tools rely on correctness properties specified by the user, such as no black holes or no forwarding loops.

For each system presented below, we also discuss how it copes with the aforementioned state explosion issue.

**NICE [5]** has the goal of finding errors in SDN applications via automatic testing. As input, it receives a controller program, a network topology (with switches and end hosts) and a specification of correctness properties for this network.

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

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

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

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

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

Finally, to efficiently search the space of network event orderings, NICE employs a number of domain-specific heuristics that favor the exploration of event interleavings that are more likely to expose bugs. For instance, one of the heuristics focuses on discovering race conditions by installing rules in switches with unusual or unexpected delays. In turn, another heuristic imposes a bound on the number of packets sent by end hosts, with the goal of reducing the search space of possible transitions.

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

**VeriCon [13]** verifies SDN programs at compile time, validating their correctness not only for any admissible topology, but also for all possible (infinite) sequences of network events. VeriCon has the advantage of providing the guarantee that a given SDN program is indeed free of errors. This contrasts to NICE (and other finite state model-checking tools), which are able to identify bugs, but not their absence. Furthermore, VeriCon scales better than NICE to large networks because it encodes the network and the correctness conditions as first-order logic formulas, which can be efficiently solved by theorem solvers.

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

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

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

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

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

Both NICE and VeriCon are *offline approaches*, in the sense that they operate prior to the deployment of the network. In the following, we present a solution that follows a *online approach*, meaning that it verifies network invariants in real-time, i.e. during the actual execution of the SDN application.

**VeriFlow [6]** analyses the data plane state of a live SDN network and checks for invariant violations before every forwarding rule is inserted, updated, or deleted. While previous tools that perform offline invariant checks would be closer to the goal of not deploying buggy control software, the execution times of those tools are at timescales of seconds to hours. Moreover, they cannot detect or prevent bugs as they arise.

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

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

VeriFlow was designed to be transparent for OpenFlow devices. The authors have implemented two prototypes: one that is a proxy between the controller and the network, being controller independent. the second one is integrated with the NOX OpenFlow controller [14] for performance improvement reasons.

The downside of online approaches with respect to offline approaches is that they degrade the performance of the network. In particular, the results in [6] show that VeriFlow incurs 15.5% performance overhead on average, which the authors argue to be tolerable in practice.

### 3.6 Debugging SDN networks

Testing and verification tools for SDN applications have shown promising results and are undoubtedly useful to improve the overall quality of these applications by reducing the likelihood of errors. However, all these tools rely on models of the network behavior, hence cannot handle firmware and hardware bugs. To address problems stemming from firmware and hardware bugs (as well as bugs not uncovered during the testing/verification phase), SDN debugging tools are still required.

Typical tools used to debug traditional networks include *ping*, *traceroute*, *tcpdump* and *SNMP statistics* [15, 8]. Unfortunately, these tools are often not enough to find the root cause of bugs, or provide little aid in this task, because they are still very time consuming.

SDN facilitates the development of better tools for network debugging, since the control software is logically centralized and hardware transparent. Furthermore, these centralized control programs have a global network view, and write the network state changes directly into the switch forwarding tables using a standard API. [8]

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

**NetSight [15]** is an extensible platform designed to capture and reason on packet histories. Packet histories can be extremely helpful to debugging because one can ask questions such as where the packet was forwarded to and how it was changed, instead of having to manually inspect forwarding rules. The authors have implemented on top of it four network analysis tools for network diagonsis showing NetSight's usefulness. Among these tools, we are particularly interested in *ndb* [8] which is an interactive debbuger for networks, with functionalities similar to *gdb*.

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

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

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

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

The authors have implemented a prototype of NetSight and tested it on both physical and emulated networks, with multiple unmodified OpenFlow controllers and diverse topologies. According to the results reported in [15], NetSight scales linearly with the number of servers.

**OFRewind [7]** is a debugging tool that allows the record and replay of packets in SDN networks. As stated before in this section, standard tools used for network debugging are in many situations not enough to find the root cause of a problem. The behavior of black-box components (switches with proprietary code) cannot be understood by analytical means alone. In such situations, repeated experiments are needed. Therefore, it would be useful for developers to replay the sequence of events that led to an observed problem. OFRewind was designed to achieve this goal. The evaluation results shown in [7] show that OFRewind can be enabled by default in production networks.

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

**SDN Troubleshooting System (STS) [16]** aims at reducing the effort spent on troubleshooting SDN control software, and distributed systems in general, by automatically eliminating from buggy traces the events that are not related to the bug. This curated trace, denoted *Minimal Causal Sequence* (MCS), contains the smallest amount of inputs responsible for triggering the bug.

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

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

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

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

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

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

## 3.7 Emulation

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

MiniNet [17] was designed aiming to enable a *prototyping workflow* that is flexible (adding new functionality and changing among network topologies should be simple), deployable and realistic (there should be no need to change a functional prototype that runs on an emulated network to run it on a physical network), interactive (network shoud be managed and ran in an interactive manner just like a physical network), scalable (it should allow to emulate networks with thousands of nodes), and shareable (collaboration should be easy so that other developers could use one's experiments and modify them).

Previously available solutions are not affordable by most developers or not realistic. A network of Virtual Machines (VMs) would meet almost all previously mentioned goals, but since virtualizing every switch and host uses significant resources, this approach becomes too heavy (does not scale). MiniNet emulates the whole network instead, sharing resources between virtualized nodes to achieve a scalable solution. In MiniNet's implementation, the network devices and the topology they form can be managed through a command line interface and python scripts. The software under test can be deployed on real networks without modification and the network can be shared using a single VM images.

MiniNet has helped the development of many useful prototypes, as reported in [17]. It has been also extended to improve aspects such as device isolation and monitoring [18].

### 3.8 Summary

Table 1 summarizes the main characteristics of the systems presented in sections 3.5 and 3.6, i.e. the testing, verification and debugging tools. All these systems have the objective of finding errors in SDN programs. The presented characteristics are the approach of the system, when the tool is ran[1], its input and its output/result.

## 4 Architecture

As discussed in the previous section, there are many alternatives to test and verify correctness properties of SDN control programs. These solutions validate a program by testing network invariants either in an offline mode, given a network topology or constraints in the possible topologies, or in an online mode. Unfortunately, these solutions cannot uncover all types of bugs, because they make assumptions that the network devices behave as the control programs expect them to behave. To detect this type of problems, debugging tools must be

[1] When talking about debugging tools, we use the designation of *post-mortem* to denote that the tool is ran after an error has been detected, in contrast to tools that try to find these errors offline, but before they happen in production.

| **System** | **Approach** | **When** | **Input** | **Output/Result** |
|---|---|---|---|---|
| NICE | Testing | Offline | Program + network topology + desired correctness properties | Sequence of events that triggers a bug |
| VeriFlow | Verification | Online | Desired correctness properties | Rule insertion/update/deletion is canceled or alarm is issued |
| VeriCon | Verification | Offline | SDN program + network topology constraints + desired correctness properties | Counter examples (topologies that violate invariants) |
| NetSight | Debugging (recording packet histories) | Interactive or offline (post-mortem) | *Packet History Filters (PHFs)* | Packet histories of interest |
| OFRewind | Debugging (record and replay) | Offline (post-mortem) | Record mode / Replay mode | Buggy trace |
| STS | Debugging (trace minimization) | Offline (post-mortem) | Correctness propertie that was violated + buggy trace | Minimized buggy trace (*Minimal Causal Sequence*) |

**Table 1.** Summary of the presented systems

used. Despite being helpful, debugging tools presented in previous section still cannot isolate the root cause of a problem.

The problem addressed in this report is to design a system that isolates the device where a network misbehaviour has its origin. Section 4.1 presents an overview of the system's architecture. Then Section 4.2 presents an example of a fault in a network device which is hard to isolate with other tools and which our tool should have no trouble in isolating. Finally, Section 4.3 indicates some implementation details, namely the tools that we will adapt to build the components presented in Section 4.1.

## 4.1 Overview

Figure 3 shows a simplified view of the main components of our planned system. The system will have two main components: a *recorder* that computes the actual paths followed by certain packets in a network, and a *graph converter* that converts the forwarding graphs into expected paths, with a representation similar to those computed by the recorder. Section 4.3 explains how these graphs are obtained. The graph converter then feeds the expected paths into the recorder so that it can compare the expected and the actual paths.

The operations performed by the system can be split between two phases, that we describe as follows.

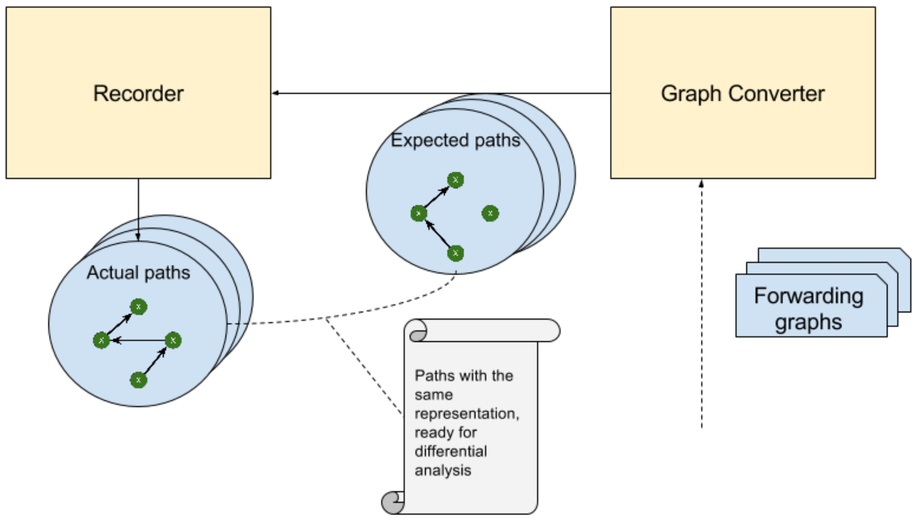
Recorder
Graph Converter
Expected paths
Actual paths
Forwarding graphs
Paths with the same representation, ready for differential analysis

**Fig. 3.** Simplified view of the system that we plan to build

- *flow modification phase*: when the controller issues commands to modify the switches' flow tables, the paths taken by different sets of packets will most likely change. At this time, the graph converter will retrieve the forwarding graphs (according to the expected forwarding behaviour) and perform the transformation to paths with representations compatible with those computed by the recorder. It will then send these results to the recorder, attaching the identifiers of these sets of packets to their respective expected paths. The recorder will then use this information to record the paths taken only by packets of interest, i.e. those whose forwarding behaviour has changed. This is achieved by setting a packet filter (computed by the recorder);
- *recording phase*: during this phase, based on the filters previously set, the recorder will receive the packet histories of interest, i.e. the ones that, in the presence of a missing flow table update, might deviate from the expected path. Computing these filters is therefore important to avoid comparing paths of packets whose expected paths did not change (because forwarding rules that were changed did not affect them). Upon receiving a packet history, the recorder has to look for the respective expected path in order to be able to perform the differential analysis and then identify the misbehaving devices, if there are any.

## 4.2 Example

Figure 4 shows a sample topology created with MiniEdit, which is a GUI to edit network configurations on MiniNet. We will use this topology to show examples of expected and actual paths and how the comparison of both can help us isolating the fault in a network. Consider that this topology is a configuration that implements in one side high bandwidth paths for video streaming (paths that traverse switch s2) and in the other side paths for regular traffic (paths that traverse switch s4). Hosts h1-h4 are servers that can use either of these paths. Clients are connected to switches s5 to s10 (omitted in the figure for simplicity). c0 is the controller. If one (and only one) of the switches s2 or s4 stops working, the traffic can still reach all the clients because of link redundancy. A problem that might occur in this configuration is switch s1 failing to install a set of forwarding rules due, for example, to ignoring an up link event issued after a brief failure of switch s2. This would cause flows to be installed to the lower bandwith path and, even when switch s2 is functional again, recent connections from video stream applications would be routed through the regular traffic path due to previous installed rules during s2's failure. New connections would be estabilished through the expected video stream path. Verification tools would not be able to identify this problem because it was a transient fault (causing a permanent error - an anomalous forwarding) and it does not even violate any invariant. Debugging tools would not be able to reproduce the error unless they have recorded the affected data packets. Even though, analyzing packet histories manually is not practical, at least taking into account that these networks may have hundreds or thousands of nodes. For this particular example, the expected path for a packet sent from h1 to a client, c, connected to s5 would be {h1, s1, s2, s5, c}. The actual path would be {h1, s1, s4, s5, c}. Comparing both sequences we can identify that the misbehaving switch is s1, the point of divergence between the two paths. We expect that this approach can aid debugging larger networks, but the first test scenario is planned be similar to this example.

## 4.3 Selected tools

To compute the expected path, we will need a tool that analyses the data plane of a network in order to produce an expected physical model of a network at a given instant. We are considering to use VeriFlow for that reason. As it is explained in Section 3.5, VeriFlow is a tool that performs online invariant validations. It operates in the data plane and uses forwarding graphs to represent network forwarding behaviour for a given *Equivalence Class* (EC) of packets. In particular, VeriFlow provides two functions in its API that will be, we believe, very useful for our approach:

- *GetAffectedEquivalenceClasses*: returns the ECs that are affected by a given forwarding rule update
- *GetForwardingGraph*: returns the forwarding graph of a given EC

To compute the actual path we need to use a recording tool rather than verification tools, because the latter assume that network devices behave as the control programs expect them to behave. We also need that this tool records the trace of a packet, i.e. the devices that the packet traversed. For this purpose we are considering extending NetSight because it is designed specifically to compute

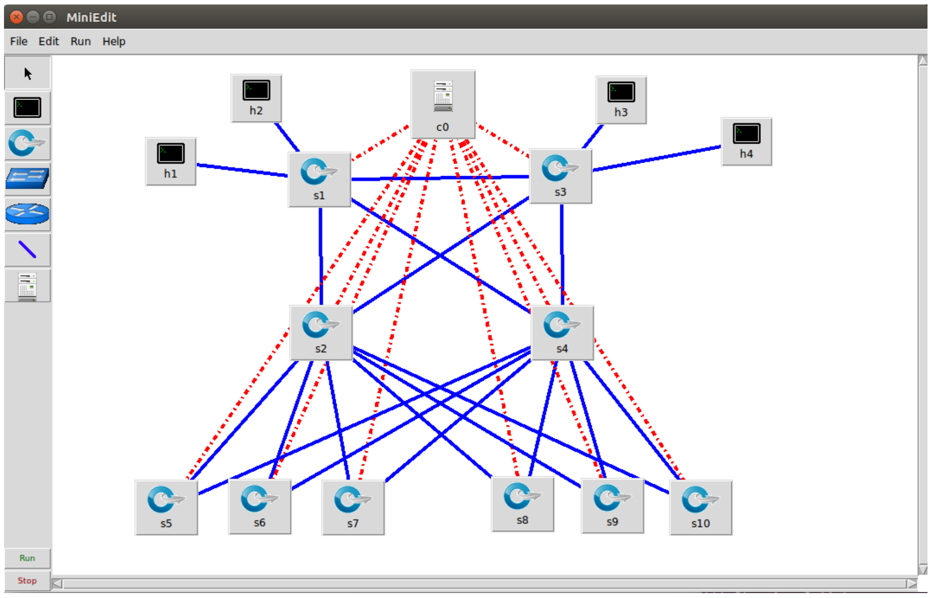

**Fig. 4.** Sample topology created with MiniEdit

the packet histories and it is optimized for that purpose. We are mainly interested in these two functions of NetSight:

- *add_filter (and delete_filter)*: allows to add (or delete) a filter of packets of interest and a respective callback function that will be called with the respective packet history

## 5 Evaluation

As described in the previous section, we plan to implement a tool that pinpoints the faulty device in presence of a network misbehaviour. To evaluate our prototype, we plan to proceed as follows.

- **Environment:** we intend to use MiniNet's network emulation capabilities, using common network topologies and multiple buggy devices. Last paragraph of last section describes a particular scenario that we plan to use to test the first functional version of the prototype. In order to perform this evaluation we need to modify MiniNet to be able to inject faults in network devices.
- **Metrics:** we plan evaluate the system's performance by quantifying the parameters that influence the time that the tool takes to isolate a fault, by measuring this time while varying the number of *Equivalence Classes* of packets and the number of network devices, one at a time. Since the purpose

of this system is to isolate switch faults in order to aid in the debugging activity, it makes sense to compare it with ndb (the debugger built on top of NetSight) which is also appropriate to debug these kind of problems. To perform this comparison we will need other metrics. Debugging time is hard to evaluate in a fair manner, so we plan to use other metrics that are related but are easier to obtain - the *number of inputs* that are necessary to give and the *number of switches* that we must look at when analyzing a path that is different from the expected one. We expect that our system will not need any input and will identify a single switch (in cases where a single fault is present), while ndb would need the manual specification of the packet histories of interest, and the packets will typically traverse several switches.

## 6 Scheduling of Future Work

Future work is scheduled as follows:

– January 9 - March 29: Detailed design and implementation of the proposed architecture, including preliminary tests.
– March 30 - May 3: Perform the complete experimental evaluation of the results.
– May 4 - May 23: Write a paper describing the project.
– May 24 - June 15: Finish the writing of the dissertation.
– June 15: Deliver the MSc dissertation.

## 7 Conclusions

Software Defined Networking (SDN) is an emerging networking paradigm that facilitates network management but also introduces new sources of errors, despite also creating opportunities to create better tools that help finding the causes of these errors. In this work we have discussed the motivation for using SDN, the main concepts of SDN architecture and common errors that affect SDN networks. Then we presented examples of testing, verification and debugging tools, as well as one emulation tool to illustrate the current state of the art in these aspects. We have then presented a solution that combines techniques of verification with techniques of debugging to isolate faults in SDN networks. Finally, we present our plans to evaluate our solution and the schedule of future work.

**Acknowledgments** We are grateful to N. Machado for the fruitful discussions and comments during the preparation of this report.

## References

1. Open Networking Foundation: Software-Defined Networking: The New Norm for Networks. White paper, Open Networking Foundation, Palo Alto, CA, USA (April 2012)

2. McKeown, N.: How SDN will shape networking. Available at https://www.youtube.com/watch?v=c9-K5O_qYgA (October 2011)
3. : Software defined networking (SDN): This changes everything! Available at https://www.youtube.com/watch?v=H_3Lk6XbWw0 (May 2014)
4. Shenker, S.: The future of networking, and the past of protocols. Available at https://www.youtube.com/watch?v=YHeyuD89n1Y (October 2011)
5. Canini, M., Venzano, D., Perešíni, P., Kostić, D., Rexford, J.: A nice way to test openflow applications. In: Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation. NSDI'12, Berkeley, CA, USA, USENIX Association (2012) 10–10
6. Khurshid, A., Zou, X., Zhou, W., Caesar, M., Godfrey, P.B.: 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) 15–27
7. Wundsam, A., Levin, D., Seetharaman, S., Feldmann, A.: Ofrewind: Enabling record and replay troubleshooting for networks. In: Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference. USENIXATC'11, Berkeley, CA, USA, USENIX Association (2011) 29–29
8. Handigol, N., Heller, B., Jeyakumar, V., Maziéres, D., McKeown, N.: Where is the debugger for my software-defined network? In: Proceedings of the First Workshop on Hot Topics in Software Defined Networks. HotSDN '12, New York, NY, USA, ACM (2012) 55–60
9. Kreutz, D., Ramos, F.M.V., Verissimo, P., Esteve Rothenberg, C., Azodolmolky, S., Uhlig, S.: Software-Defined Networking: A Comprehensive Survey. ArXiv e-prints (June 2014)
10. 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. **38**(2) (March 2008) 69–74
11. : The openflow switch specification. Available at http://OpenFlowSwitch.org (February 2011)
12. Patton, R.: Software Testing (2Nd Edition). Sams, Indianapolis, IN, USA (2005)
13. 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. **49**(6) (June 2014) 282–293
14. Gude, N., Koponen, T., Pettit, J., Pfaff, B., Casado, M., McKeown, N., Shenker, S.: Nox: Towards an operating system for networks. SIGCOMM Comput. Commun. Rev. **38**(3) (July 2008) 105–110
15. 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: Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation. NSDI'14, Berkeley, CA, USA, USENIX Association (2014) 71–85
16. 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. SIGCOMM Comput. Commun. Rev. **44**(4) (August 2014) 395–406
17. Lantz, B., Heller, B., McKeown, N.: A network in a laptop: Rapid prototyping for software-defined networks. In: Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks. Hotnets-IX, New York, NY, USA, ACM (2010) 19:1–19:6

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