MED: The Monitor-Emulator-Debugger for Software-Defined Networks

Quanquan Zhi
Institute for Interdisciplinary Information Sciences
Tsinghua University
Beijing, P.R.China
Email: zq_ax@163.com

Wei Xu
Institute for Interdisciplinary Information Sciences
Tsinghua University
Beijing, P.R.China
Email: weixu@tsinghua.edu.cn

***Abstract*—Software-Defined Networks (SDN) greatly improves programmability but brings in extra challenges for debugging. We propose an SDN debugging framework, Monitor-Emulator-Debugger (MED). It closely monitors the physical network, and automatically creates an emulator that can be set to the network state at any given point of time. In the emulator, MED synchronously constructs a virtual SDN that is identical to the physical SDN and replays real packet samples. The emulator also handles the non-determinism due to packet reordering. On top of the emulator, we provide fast and efficient debugging tools including loop and reachability detector, race condition detector and forwarding table checker. All the tools run on the emulator without adding any additional overhead to the physical SDN.**

We implement MED for an OpenFlow-based SDN in a data center network employing 20 switches. Using a combination of micro-benchmarks and real debugging case studies, we show that MED is both fast and useful in SDN debugging. During the evaluation, we reveal two physical switch bugs that have been confirmed by the vendor.

I. Introduction

Software-Defined Networks (SDNs) [1] introduce a new architecture and offer flexible network functionality: custom programmability and centralization of the control plane. Unfortunately, as with all new emerging technologies, bugs are common in SDN. In particular, SDN systems are inherently distributed and asynchronous, with events happening at different switches and the controller. Bugs can happen at any level in the SDN stack (e.g. controller logic, switches, and individual SDN applications). [2] and [3] report three types of common bugs, which we also see in our production network:

1. Controller logic bugs. Buggy controllers can cause user-visible failures such as reachability problems, forwarding loops and broadcast storms. As [4] points out, most SDN bugs are related to the controller logic.

2. Switches software bugs and performance disturbances. Since most of the current SDN software is relatively new with very limited production use, it is not surprising to find more bugs in these implementations than in traditional switch software. In addition, SDN switches may have high variance in CPU load that may cause transcient failures [5]. We provide two real case studies in Section VI-F.

3. Race conditions. Although the logical control plane is centralized, the system contains multiple switches, forming a distributed system with asynchronous events. These events may not happen in deterministic order across multiple switches. Race conditions may occur if switches process events in different orders. Examples of race conditions are discussed in references [6], [7], and [8]. Higher frequency of network updates in SDN [9] makes the race condition problem even worse.

With SDN, the network operation is turning into a development-operation (DevOp) task. Operating an SDN is similar to operating a distributed software system that involves constant updates. Therefore, new tools in network debugging is critical for the SDN DevOp efforts. Researchers have proposed multiple approaches in this direction. Some have developed modern domain-specific languages such as Frenetic [10] and Nettle [11] to simplify SDN programming and reduce the chances of bugs occurring. For debugging SDN, people have developed both static checking tools like NICE [6], Anteater [12], and Header Space Analysis [13], and runtime monitoring-and-replay based tools like OFRewind [5]. We compare our work to these projects in Section II.

Unfortunately, we are still missing the key features needed for SDN debugging. In the Test-Driven Development (TDD), a widely used software engineering practice [14], repeatable test cases are emphasized in order to drive development as well as bug fixes. Minimally, TDD requires tools to create a test environment, as well as testing data input (aka fixtures).

This paper describes our experience in creating such a tool, dubbed the Monitor-Emulator-Debugger (MED). MED can create a software emulator to automatically emulate a network's state at any given time in the monitored history repeatably. Specifically, it can emulate the non-determinism in the networking caused by race conditions. MED also provides a series of automatic debugger programs that detect network anomalies, such as forwarding failures and loops, as well as incorrect switch software implementations.

MED contains three major components: monitor, emulator, and debuggers. To set up the testing environment, the *monitor* captures every state change events in the control plane of the physical SDN in real time. The captured information is then used to construct a virtual SDN *emulator* that is identical to the physical SDN at a given point in time. The developer can either use this emulator interactively, or use automatic *debuggers*. Figure 1 shows an overview of MED's architecture.

978-1-4673-9953-1/16/$31.00 ©2016 IEEE

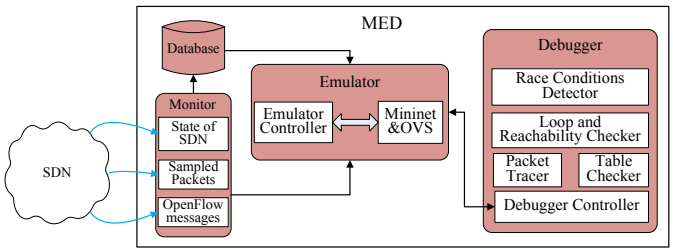


Fig. 1. MED overview. MED is a a debugging system for SDNs that captures the configurations of the physical network fabric to construct an emulator, which provides a testing and debugging environment

To gather proper test data, MED samples packets from the physical SDN and automatically replays these samples in the emulator. It traces the forwarding paths for each individual packet in the emulator, and detects suspicious forwarding (e.g. dropped along the way or trapped in a loop) automatically.

MED does not require modifications to the physical network. Other than the optional *sflow*-based [15] packet sampling, MED brings no extra performance overhead to production networks.

We implement an MED prototype to support the OpenFlow-based SDN [16], and deploy the prototype in our production cloud environment with 20 switches and 200 physical servers. The experimental results are promising: MED can quickly and accurately capture packet forwarding paths with complex forwarding logic, detecting the three types of bugs mentioned above. MED also reveal two bugs in our physical switch software that are confirmed by the switch vendor.

We offer the following three contributions:

1. We design MED, an SDN emulation framework. MED can automatically and repeatably create a software emulator to emulate network state at any point of time, making it a practical testing and debugging tool.

2. We implement a series of debuggers in MED which analyzes and automatically report abnormal behaviors such as forwarding loops and race conditions. MED also helps detect buggy switch software implementations.

3. We implement an MED prototype in a real, 20-switch SDN environment and demonstrate its effectiveness using several real bug case studies.

The remainder of this paper is organized as follows: We introduce the related work in Sections II. Section III presents the general architecture and design considerations of MED. Section IV discusses about the detailed design and implementation of MED's core component, the emulator. Section V provides details in our four debugger programs. Section VI presents an evaluation of the MED prototype. We conclude and propose some future directions in Section VII.

II. Related Work

As is typical in the software world, existing work on SDN debugging falls in two categories: 1) Using static analysis to discover network problems, and 2) Using runtime-monitoring data to debug. Both of these approaches have pros and cons. MED falls in the latter category.

A. Runtime monitoring based approaches

OFRewind [5] helps network operators to localize and troubleshoot network events that could cause anomalies. It records network events and replays these events in a physical network, either using a separate set of equipment or the production network during off-peak time. MED also monitors these data, as well as additional topology information. Different from OFRewind, MED also tries to do automatic testing and diagnoses. Also, MED then takes advantage of a software emulator to perform the replay. Eliminating the dependency on the physical network is essential: it not only makes the debugging faster and cheaper, but also allows us to have much better control over the replay. For example, we are able to revert the network state to any given point of time, "slow down" time in the network, and collect detailed runtime information from the network. Of course, MED does not have the capability of OFRewind to debug data plane and performance problems, but as [4] points out, 90% of the bugs in a typical SDN are control plane bugs, which MED can help debug. In addition, MED is able to help check switch software implementations while OFRwind does not.

Packet tracing is an important topic in SDN monitoring. Ndb [17] and NetSight [18] modify the existing rules in the network and allows the physical switches to send "postcards," which can be used to reconstruct the path and sequence of forwarding actions for a subset of traffic. Paper [19] installs special, high priority rules on the switches that matches special markers in packet headers. [20] leverages the network topology information and uses calling context encoding to record the path information into packet headers. Comparing to these projects, MED takes advantage of the software emulator and simplifies the packet tracing process (details in Section V). MED does not modify the physical network, and thus introduce no extra overhead or risks to the production network.

Monitoring is the foundation of debugging. There are also projects aiming to improve SDN monitoring efficiency. SDN-RADAR [21] uses distributed monitoring to collect traffic information for debugging. [22] computes an optimal or near-optimal number of static forwarding rules on switches to locate link failures and verifies the topology connectivity.

B. Static analysis based approaches

Header Space Analysis (HSA) [13] uses a geometric model to analyze SDN behavior. HSA constructs the transfer functions for the switch and topology, which is based on the ordered set of forwarding rules. It also analyzes switch configurations to statically check connectivity and isolate errors in control-plane. By viewing the packet headers as a flat sequence of ones and zeros, HSA simulates the forwarding behavior. However, these models are time-consuming to create [13]. Also, in an SDN with frequent state changes, it is even more time consuming to update the model. In comparison, MED uses an emulator to actually "exercise" test packets and detect problems, which is faster, although may not cover all the possible cases. This is a common tradeoff between static verification and runtime analysis.

Automatic Test Packet Generation (ATPG) [2] and NetPlumber [23] are extensions of HSA. ATPG automatically generates a minimum set of test packets, which people can use to actively test for potential errors. NetPlumber improves HSA by constructing a "plumbing graph" that is updated in real-time on network state changes. The tradeoff is between better "test coverage" and computation cost. Instead, MED use lots of sampled packets running in fast and scalable emulator to improve the test coverage.

NICE [6] applies model checking and symbolic execution by automating OpenFlow applications testing in the control plane with simplified switch/host models. Anteater [12] models network devices's (firewalls, routers and switches) behavior in the data plane as instance of Boolean satisfiability problems (SAT), and then uses an SAT solver to analyze and check certain network invariants.

Static model-based debuggers analyzes tables on switches to infer forwarding paths and detect problems. However, even if the configuration is correct, the actual switch may not follow the configuration due to switch software bugs or runtime failures such as resource limitations. We believe at least during the early years of SDN, it is still important to be able to debug the actual implementation, including both controller and the switch control plane code, which MED focuses on.

III. MED ARCHITECTURE

The SDN control logic is implemented as a centralized controller program. The controller computes forwarding rules for every switch. Using a standard protocol such as OpenFlow, the controller directly manipulates the forwarding tables on each switch. Thus, by observing all communications between the network equipment and the controller, we can determine most of the state changes in the network.

Based on this observation, we propose Monitor-Emulator-Debug (MED), which collects messages from the controller-network communications and the data plane traffic information in the physical SDN, as well as the status information of all the components. Using this information, MED emulates an SDN network that is identical to the real network, using Mininet [24]. MED then uses the emulator to detect and analyze the issues of the real network by replaying traffic, which is more convenient and flexible than using the actual network. Specifically, we take snapshots of the SDN switch state, including flow tables and physical links, but we only observe the behavior of the SDN controller while treating the controller logic as a black box. Thus, our approach works on any OpenFlow-based SDN fabric.

Figure 1 illustrates the three major components of MED: the monitor, the emulator, and the debugger.

A. Monitor

The OpenFlow protocol defines the standard interface for interaction between the switches and the controller, and allows the querying of switch and traffic states. The monitor leverages these OpenFlow features to collect the following four types of information.

1. *Physical topology.* Including MAC and IP addresses of end hosts, ports configurations, connections, and link capacities from the switches.

2. *Flow table dumps.* The debugger uses the information collected to construct a global snapshot of SDN states. We detail this process in the next section.

3. *Network events.* The monitor observes the controller port and monitors all OpenFlow messages between controllers and switches, including packets-in, packets-out, rule installation/removal, and ports up/down events. In the following discussion, we call these messages *network events*.

4. *Data packet samples.* While it is impossible to collect and record all packets, the monitor uses *sflow* [15] to sample some packets. It further reduces the amount of data required by only collecting the packet header.

The monitor stores all information in its internal database. It provides both pull and push APIs, allowing the emulator to access the database.

B. Emulator

The emulator is the core component of MED. It uses the monitor database to automatically create an emulator. The key feature of the emulator is to emulate network states at any given point in time in the monitored history. We leverage Mininet to build the data plane, while focusing on the emulation of the control plane. The emulation process includes generating a snapshot of the physical network using the monitor data, using the snapshot to initialize the emulator, and replaying events to keep track of the state changes that occur in the physical network.

The main challenge of building such an emulator is finding a way to emulate non-determinstic behaviors in the network such as race conditions. To make the emulator powerful yet easy to use, we provide two different emulation functions: `set_to_stable` and `set_to_nondeterministic`, allowing users to see different levels of event reordering and non-determinism. We provide details of the emulator design in Section IV.

C. Debugger

The debugger is based on Mininet and thus supports all the debugging tools built into Mininet. We implement an extensible debugger-emulator interface, making it easy to manipulate the emulated network. The debugger API is extensible, and as examples of the paper, we provide the following debug functionalities:

1) The Packet Tracer (PT) automatically replays data packets from the real network, and records the forwarding paths and the matching flow table entry on each switch.

2) The Loop and Reachability Checker (LRC) that is built on top of the PT, detects loops and reachability problems.

3) The Table Checker (TC) automatically compares all of the flow table entries in the physical and emulated switches, and alerts the operator if there is any disagreement.

4) The Race Conditions Detector (RCD) takes advantage of the non-deterministic state feature of the emulator, and automatically discovers potentially harmful race conditions.

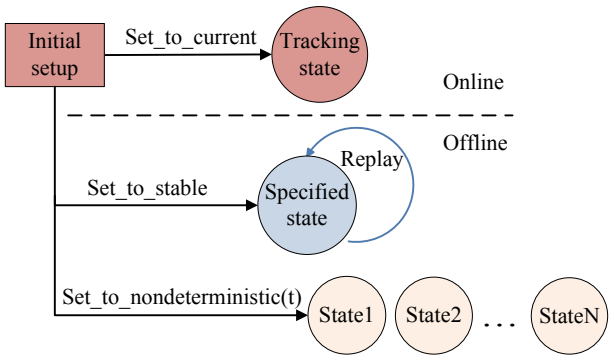


Fig. 2. The functionalities of the emulator. It works both in online or offline mode. In the offline mode, the user can choose whether to observe the non-determinism in the network. The text provides more detailed discussion.

We discuss the detailed implementation of these debuggers in Section V. The debugger not only allows operators or debugger algorithms to passively observe the SDN, but also, by using an emulator, it helps to replay and debug race conditions, which are very hard to reproduce in physical network.

IV. Emulator Design and Implementation

Our emulator is based on Mininet. Mininet uses Open vSwitch (OVS) [25], the widely used OpenFlow implementation, to emulate the data plane. We choose Mininet and OVS because of their popularity. We use Mininet and OVS to perform all data plane emulation, which is not our focus in this paper. We focus on control plane emulation in this section. Figure 2 shows the major functionalities of the emulator, which we discuss in detail in this section.

We define the network state at a given time t, S_t, as the physical topology (links, ports, switches, hosts etc.) of the entire fabric and all flow table entries from all switches. S_t determines the forwarding path of any packets arriving at t. Note that the controller might also have some internal states that have not affected physical switches yet, and we do not include those in S_t, as they are not affecting packets at time t. The key functionality of the emulator is to take a user input t, read the monitor's history database, and create an emulated network with S_t that is identical to the physical network.

Note that solely using the information from the monitor, sometimes we cannot uniquely determine S_t. This is usually because of the possible reordering of event processing, or potential race conditions. We discuss about the details of these issues in Section IV-D.

A. Physical SDN snapshot and emulator initialization

We allow user to start the MED process at any time while the physical network is running. Thus, we need to mirror all physical network states in the emulator first.

At a high level, MED uses the topology from the monitor to generate a Mininet configuration file. It then uses the flow table dumps to initialize emulated flow tables. In order to set the emulator to a specified state S_t, we first install the latest snapshot before time t, and then replay messages between the snapshot time and time t.

Theoretically, the initial setup requires a consistent snapshot of the physical network, which is difficult to achieve, as states may change during the snapshot. We use an optimistic approach: during the snapshot process, we keep monitoring the network events, and after the snapshot we replay these messages to allow the switches to "catch up" the most recent changes. After obtaining a snapshot, we check its accuracy by comparing the emulated state with the physical state. We retry the snapshot if they do not agree. Experiments in Section VI-B show that we can obtain a consistent snapshot within a short period of time.

B. Tracking state changes in physical network

The typical use of the emulator is allowing the emulator to track all physical network state changes, so that we can perform live testing in an online fashion.

We treat the physical SDN controller as a blackbox. The controller may compute the forwarding logics in the SDN fabric based on current network traffic. As with OFRewind [5], since the controller logic is eventually turned into OpenFlow messages to network devices, we only need to replay these messages to emulate the network state changes. Specifically, the emulator replays all OpenFlow messages (Flow-mod, Packet_in, Packet_out, Port_mod) from the monitor database to the Mininet.

Strictly speaking, we need to assume that the observed event ordering is the same as the actual processing order. It is not always true even on a single switch, if the controller does not use `barrier` messages correctly. In realtime tracking, we assume any processing order results the same state, which is generally true. If the user needs to debug packet reordering, she can use the non-deterministic function in Section IV-D.

We take advantage of the software nature of the Mininet emulator. Instead of using OpenFlow protocol to install the flow table entries, we chose to use standard OVS tools such as *ovs-vsctl* and *ovs-ofctl* to manipulate the flow table entries. In this way, we can still leave the OpenFlow controller interface available for debugging controllers such as the Packet Tracer discussed in Section V.

C. Offline emulation: stable states

Network operators often want to set the network into a certain state to take a closer look at a bug. MED provides a function `set_to_stable` to support this. We define a stable network state to be a state in which there are no control plane events in the network that have not been processed. The system will return the closest *stable* state after t. For a network that does not change too often, or if the user is not debugging timing-related issues, examining the stable state is often sufficient.

The `set_to_stable` algorithm is straight-forward: First, we reuse the initialization and replay functionality to bring the network state to time t. Then we check if there are any pending events in the network at time t. We detect pending events by looking for controller message without an ACK. Notice that the ACK message is optional in OpenFlow specification,

and we require users to enable the feature, which is already a common practice. Finally, we replay these pending events until all of them are processed.

D. Offline emulation: non-deterministic states

Given the distributed nature of the physical SDN, we cannot determine the actual order in which different switches process network events. Previous work such as OFRewind uses a logical clock to track the causal ordering, so that the system can provide the "true" specific ordering. In this sense, MED goes beyond just "replaying" events. MED can provide the users with all possible states as the results of different ordering, using the `set_to_nondeterministic(t)` function. This feature can be crucial for people debugging time-related issues. We built a race condition detector based on this feature, and we provide a concrete example in Section V-D.

The emulator implements this functionality by first detecting concurrent events. It then forks multiple emulators and uses each one to replay one possible event ordering. Of course, the number of possible states can be factorial to the number of pending messages.

V. Debugger Implementations

In this section, we provide some details regarding the major debugger plugins we provide in MED and further demonstrate the power of the MED emulator.

A. Packet Tracer (PT)

The goal of Packet Tracer (PT) is to discover the exact path in which a packet is forwarded in the network, at a given network state S_t. PT also discovers matching flow table entries on each switch, and enables many higher-level debuggers.

We implement PT as an SDN controller, which is an example of the *debugger controller* in Figure 1. Note that the debugger controller is different from the emulation controller, although they both manipulate the flow tables in the Mininet. The debugger controller interfaces with Mininet using the standard OpenFlow protocol, and thus it will receive all `PACKET_IN`s from emulated switches.

Figure 3 illustrates the workflow of PT, after the emulator is set to network state S_t.

On initialization, PT modifies flow tables on each emulated switch, adding an extra action `TO_CONTROLLER` to each entry. In other words, when the packet matches any entry, in addition to normal the forwarding, we ask the switch to mirror the packet to the controller at the same time.

To start a packet tracing, we choose not to use the host emulation feature in Mininet in order to lower the resource utilization. Instead, we inject the packet directly into the system by constructing a `PACKET_OUT` message and sending it to the switch with the injection port. The data area in the message is the layer 2 header of packets, in which in-port is set to the injection port, and the action field is set to `TABLE`. In this case, the emulated switch will treat the packet as though it had been received from the host.

When the switch receives the injected message, it takes actions based on the matching flow table entry. There are at least two actions on the entry: the normal entry that decides the packet forwarding, and the extra `TO_CONTROLLER` action added at the initialization step. This extra action causes the switch to package the incoming packet into a `PACKET_IN` message and forward it to the controller. Thus, PT receives the packet and knows where it comes from.

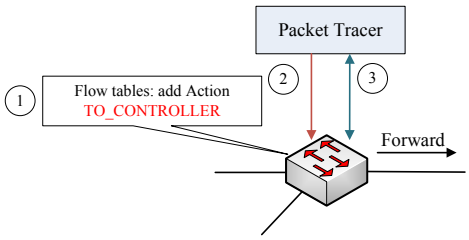


Fig. 3. The Packet Tracer (PT) workflow. The `PACKET_OUT`, `PACKET_IN`, `FLOW_STATS_REQUEST` and `FLOW_STATS_REPLY` message track exchanges between OVS and PT to trace the path of data packets.

The final step is to find the matching flow table entry, which is often useful for finding the root cause of a bug. For each `PACKET_IN` that PT receives, PT send out a `FLOW_STATS_REQUEST` message, using information in the `PACKET_IN` to construct the `MATCH` field. On receiving the `FLOW_STATS_REQUEST` message from PT, the switch looks up the table to find the matching forwarding rule and replies to PT with a `FLOW_STATS_REPLY` message.

PT repeats the entire process on every switch on the forwarding path of the packet. When the packet reaches its destination (when no more forwarding rules can be found), the process ends and PT reports the results.

We implement the PT as an application within the POX [26] framework. PT works in two modes. We can automatically sample and replay a message side by side with the live network and detect problems automatically (the online mode), or we can use it to replay messages from recorded traces or traces manually created by the developer (the offline mode).

B. The Loop and Reachability Checker (LRC)

On top of Packet Tracker, we build a higher-level debugger to detect loops and reachability problems in the network. The Loop and Reachability Checker (LRC) keeps monitoring the Packet Tracker output, and alerts the user through two callback functions `loop_alert()` and `reachability_alert()`.

In particular, the loop detector simply checks for replicates in the {**switch_id**; **inport**; **outport**} from the path output in Packet Tracker. The reachability detector checks if the packet ends up at the intended destination (specified in the original packet header) at the end of the path. If it does not, LRC determines that there is a reachability issue for that packet.

C. Table Checker: detecting buggy physical switches

One important goal of MED is to detect control plane bugs in the physical switch software. As the physical switches are black boxes to us, we can only reason about their behaviors from their public interface.

The switches may install forwarding rules incorrectly for many reasons. There may be a transient failure, for example, when the CPU utilization ratio is very high on the switch, the rule installation may timeout and fail. There may also be some bugs in the OpenFlow agent software implementation causing persistent failures. Using MED, we can quickly detect incorrect flow tables by comparing the physical flow tables to those in the emulator. We raise alerts when behaviors of the physical switch deviate from those of the OVS.

D. Race Conditions Detector (RCD)

As we mentioned in Section I, race conditions are common in SDN, due to the distributed nature of the SDN fabric. As [7] reports, the most common types of race conditions are results of the controller program bugs, causing dependent actions to run concurrently. In such cases, the outcomes depend on the actual order in which these actions are performed, causing hard-to-debug issues. Even worse, the bugs are often visible only during a network state change, causing transient packet drops or loops. These transient problems may have a big impact on application performance, especially for latency sensitive applications, as [9] points out.

We design Race Conditions Detector (RCD) to help debug race conditions. RCD relies on the `set_to_nondeterministic(t)` function to work. It first generates all possible network states, using all possible orderings of these concurrent events, and then applies LRC detectors in each possible ordering to find any suspicious situations. RCD finally generates a report for the operator, showing possible anomalies (loops or packet drops, plus which ordering caused them).

In order to debug transient problems caused by race conditions, we are able to slow down or "single step through" the transition period, allowing users to test partial updates during a network transition. Section VI-F provides a concrete example.

VI. Evaluation

In this section, we demonstrate the effectiveness of MED using both micro-benchmarks and real case studies. Our experimental results show that MED not only captures and provides all states of a physical SDN accurately, but also provides insightful information and tools to help reproduce, test and debug common issues, such as loops, reachability failure, commodity switch software bugs and race conditions.

We perform all of our evaluations on a real data center network with OpenFlow-based SDN, consisting of 20 Pica8 [27] P-3298 switches connecting about 200 servers, runing OpenStack. We use 16 of these 200 servers to emulate the hosts, each of which has 12 CPU cores, 128GB of DRAM and 1Gbps Ethernet card. The switches are interconnected using 10Gbps Ethernet. Figure 4 shows the topology of this network, a typical fat-tree with redundant aggregate and core switches. As we have redundant paths in the network, we enable Spanning Tree Protocol (STP) in all the switches, in addition to OpenFlow. Aside from STP, the switches run in OpenFlow mode without any other L2/L3 features.

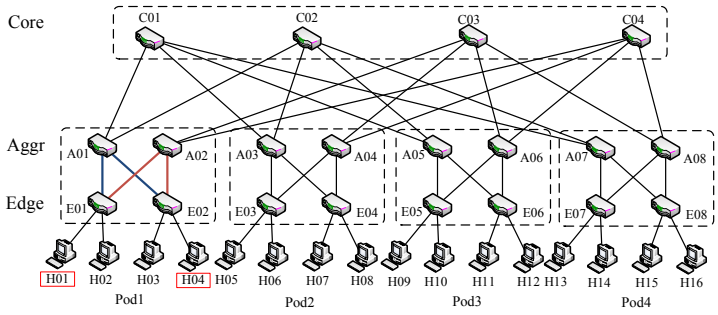


Fig. 4. The topology of the 20-switch testbed.

A. Emulator initialization

We evaluate the time required to initialize MED, during which MED collects the current state of the physical network, such as the topology, link bandwidth, and the switch configuration including the forwarding rules. The MED configures the Mininet. We perform two micro-benchmarks: a network in stable state, and also a network with constant flow table updates.

Initialization from stable state: We randomly insert over 30,000 OpenFlow rules to the 20 switches, averaging 1,500 rules per switch, and measure the time taken for MED to set up the emulator. We insert these rules so that there is at least one path between every pair of hosts. The initialization time involves three components: 1) time to discover the topology, generate the Mininet configuration file and initialize Mininet; 2) time to dump all flow tables from the physical network and 3) time to insert all the flow table entries into the emulator.

Table I shows the results with 30,000 rules. The topology discovery and setup takes about 4.9 seconds to complete on a 20 switch fabric, and about 0.54 seconds to dump all flow tables from these switches. Then, it takes about 12.2 seconds for MED to install all 30,000 flow table entries and set up the emulator. That is, on average, it takes MED about 0.405 ms to set up a forwarding rule to the emulator and the total time taken is linear to the number of rules. It takes less than 18 seconds to finish the entire initialization process and fully configure the emulator.

We further evaluate the performance in dumping flow tables with various sizes. Figure 5 shows the results of dumping different number of entries from a single switch. The flow table is small: a table with 1,500 entries is only 415KB in size. As we use a multi-threaded program to dump flow tables from all switches in parallel, it takes roughly the same time to dump one table or to dump all 20 tables. We can see that the dumping process is fast, as it is a batch operation. It is significantly slower when we have over 1300 entries, and we believe it is due to the specific implementation of our physical switch.

TABLE I
Time breakdowns of MED initialization

Topology	Dump flow tables	Install flow tables
4.9 seconds	0.54 seconds	12.2 second

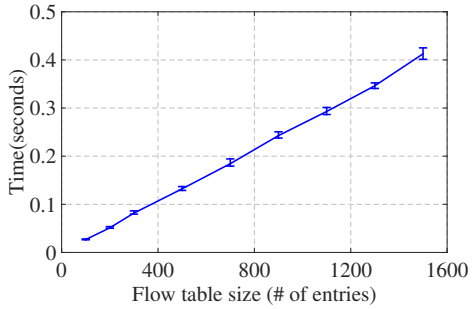


Fig. 5. Time to dump flow tables of various sizes from the physical switches.

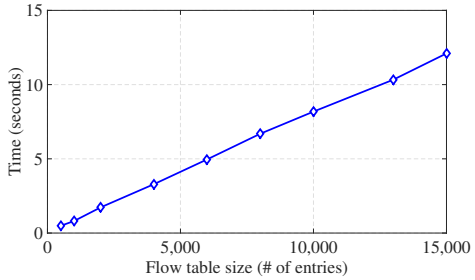


Fig. 6. Time to set up different number of flow table entries into the emulator.

It takes more time to install these tables into the emulator though, as we add these entries one by one. We evaluate that performance in Section VI-B.

The results shows that emulator initilialization is fast, even with lots of random forwarding rules. We believe this performance level is a key feature to allow users to try different network states in a short period of time.

Initialization from a running network: As we have discussed in Section III-B, MED supports snapshotting the physical network while the network is still updating. We evaluate the correctness and performance of this function using hosts A01, A02 and switches E01, E02 in Figure 4.

In order to test logics that handle network updates while taking the snapshot, we generate a Flow-mod event for each switch every 35 ms, close to the physical limit of how fast our physical switch can handle such events.

In our experiments, it takes about 910 ms for MED to snapshot and copy all of the flow tables from the physical switch into the emulator. During those 910 ms, there are 78 flow-mod events in the physical network, making the snapshot stale at some switches. By identifying and replaying these 78 events, we verify that we get the same network state as the physical network, by comparing the flow tables directly.

To simulate unexpected delays in the network during the snapshot (e.g. some slow-responding switches), we add random delays to the snapshot process, and roughly extend the snapshot time to 9.1 seconds, or 10x the original delay. As expected, this initial snapshot is even less consistent. We end up replaying about 780 events to bring the emulator to the correct state. This extreme case demonstrates the robustness of our emulator initialization process.

B. Emulator performance

In order to emulate and automatically test a large number of states and packets, the performance of the emulator is crucial. We evaluate the event replay performance and time required to change to a specified state S_t.

Replaying network event: The time required to set the emulator into the desired state depends on how many flow table entries we need to insert into the Mininet emulator. Figure 6 shows the time required versus the number of events we need to replay.

We can see that similar to the previous experiments, the state switching cost is linear to the number of table entries processed. It takes MED 230 ms to install the 500 new rules to the emulator. As a comparison, it takes 360 ms to install the same set of rules to the physical switches. The emulator is faster because OVS has a much faster CPU. We compare the flow table entries on both MED and the physical network and confirm the correctness of the emulator.

The linear processing time may slow down replaying network events over a long period of time. In order to accelerate the switching, we can choose to take periodic snapshots from the physical network, and choose the closest snapshot and start replaying events from there. In that way, we can reduce the number of events we need to replay. Also as a future work, we are extending Mininet's interface so we can improve the parallelism when insertting independent entries.

C. Packet tracing

In this section, we evaluate the performance and correctness of the Packet Tracer, using the same topology and network state as in Section VI-A.

We enable *sflow* to collect a sample of packets. For testing purposes, we feed all the sampled packets into the MED emulator with the Packet Tracer enabled. We query PT for both the traces and the flow table entries used in each hop. We use random rules because we want to have a variety of path lengths for evaluation. We have paths with lengths of 2 to 10 hops. The number of hops in the sample data is Gaussian distributed, with both mod and mean 6 hops.

As we use random forwarding rules, the packets do not necessarily use one of the shortest paths. For example, with one of the random topologies in our experiments, we have a long path between host H09 and H16, which is $H09 \rightarrow E05 \rightarrow A06 \rightarrow C03 \rightarrow A02 \rightarrow E01 \rightarrow A01 \rightarrow C01 \rightarrow A07 \rightarrow E08 \rightarrow H16$.

Table II shows the performance results on the path detection latency. If the path contains only two hops (passing through a single switch), it takes PT an average of 0.6 ms to output the path and matching rules. For paths with 10 hops, it takes about 5 ms to discover.

We also measure packet tracing throughput. Using our single node MED, we are able to analyze at a packet sample rate of 50 Mbps in real time. Note that throughput is not a major blocker for scalability, as we can trivially scale it out to more nodes, each of which running an independent emulator.

To verify the correctness of the packet tracer results, we add an extra step on the physical switches to obtain the "ground

TABLE II
PATH LENGTH V.S. PACKET TRACER PERFORMANCE

# hops	2	4	6	8	10
% of test data	10.6%	13.2%	57.9%	16.2%	2.1%
time taken (ms)	0.626	1.536	2.828	3.532	5.001

truth". We enable port mirroring on all the physical switches, and analyze the packet dump for the exact packet's path. We also dump all flow table entries and associated counters from each physical switch in order to check which rule is used for those packets. Of course, this extra work is only for the evaluation; we do not require any change to the physical network in production. All of the comparisons show that MED can correctly detect all forwarding paths.

D. Checking loop and reachability

Paper [28] presented a POX L2_learning switch bug that causes reachability failures during a host migration. We reproduce the bug on our testbed to demonstrate the Loop and Reachability Checker functionality.

We use the Pod1 in Figure 4 to reproduce the problem. We first connect H01 to port P1 of switch E01. Then we physically move the H01 to port P3 of switch E02. After the migration, we find that H02 could no longer communicate with H01.

LRC provides two pieces of information to help diagnose the problem: First, LRC automatically reports the following unreachable path from H02 to H01: $H02 \rightarrow E01(Port1)$. Second, LRC shows the rule on switch E01 that causes the packets to go out of port P1 (the original port, instead of the new port connecting E02). This information points the operator to the root cause of the bug: the controller forgot to remove the original flow table entry after the migration. To understand the root cause, one needs to examine multiple switches along the potential path, which is time consuming. MED checks all flow tables along the path automatically, greatly reducing the human labor.

E. Detecting bugs on physical switch software

Table Checker checks whether the physical switch processes a set of events in the same way as OVS. Using TC, we reveal two bugs in our physical switch.

Switch Bug 1: An important goal of MED is to detect buggy or incompatible switch software. Again, we run the POX module l2_learning.py [26]. To simplify the discussion, we will only focus on a single switch with two connected hosts. On the switch, we run PicOS-OVS 2.3 in OVS mode [29] .

We surprisingly find that H01 cannot ping H02, even if they are connected on the same top-of-rack switch. When we query MED for the potential problems, MED raises an alert showing a disagreement between the forwarding tables on the emulated switch and the physical switch: the emulated switch has the entries while the physical switch does not have any.

The results from the emulator make sense: it installs a rule for ARP and a rule for ICMP, matching the expected behavior of a L2/L3 switch. However, the physical switch does not correctly install the flow table entry. In this case, MED provides two points: 1) the controller does compute the correct forwarding rules, and 2) which rules are missing from the switch. With such information, we have a strong reason to suspect that it is due to buggy switch software. Since we do not have access to the switch code to debug, we report the issue to the switch vendor, who quickly reproduced the bug using the information we provide and confirm it [30].

Switch Bug 2: We also find an inconsistency in different implementations of the OpenFlow protocol. In one of our experiments, we observe an error message `ofp_error_msg` from the emulator, and the emulator fail to install the flow table entry. The physical switch, however, installs the rule without any error.

After some debugging, we find the following reason for the error: the controller tries to install two rules that are exactly the same (with the same priority), using the `OFPT_FLOW_MOD` message. In all of our switches, we have set the flag `of.OFPFF_CHECK_OVERLAP` set to `true`. According to the OpenFlow specification, the switch shall refuse the operation, and raise an error, and OVS follows this specification. However, the physical switch silently ignore the error and installed the entry. The switch vendor has confirmed the inconsistency.

As the SDN standard is still evolving and the switches are not well tested, we expect to find more bugs or issues in these switches. TC allows for checking an OpenFlow agent implementation against a reference implementation (in this case, OVS), which is very useful.

F. Debugging race conditions

We demonstrate the features of Race Condition Detector (RCD) using a real race condition of a traffic engineering controller we develop in another project. As an example, consider hosts H01 and H04 in Figure 4. There are multiple paths between them. The traffic engineering controller periodically reassigns flows from H01 and H04 to an alternative path. A synchronization bug leads to packet loss sometimes, causing unpredicatable performance problems for the latency-sensitive applications.

For example, we change the path from H01→E01→A02 →E02→H04 to H01→E01→A01→E02→H04.

The controller needs to send three events to three switches A01, E02 and E01 respectively to complete the path change.

- Add r (match H01 to H04, in_port=1→Port2) at A01
- Add r (match H01 to H04,in_port=3→Port1) at E02
- Change r (match H01 to H04, in_port=1→Port3) at E01

The buggy controller issues the three events in one shot, without waiting for any operations to complete (i.e. waiting for the ACK message). Due to random delays in controller-switch communication and switch software, there are 6 possible orderings for the three operations. It is not hard to see that four of these orderings may cause packet loss during the transition.

MED correctly detects that these three operations are concurrent. It automatically retries all six possible orderings in the emulator. MED reuses existing loop and reachability detectors

on each of the ordering, between any two operations. For each ordering, MED will "single step" through the process, and after each step, it will replay the test packets and use LRC to detect abnormal forwarding paths. In this case, MED correctly output an alert showing that these operations should not be concurrent. Of course, actually fixing the bug requires more work, and there are research projects trying to prevent these situations from happening [7], [9].

VII. Conclusion and Future Work

We presented our work on MED, a new SDN debugger that combines the benefits of a software emulator, real network configuration and real packets. We show that with minimal overhead, we can capture the network states, automatically create an emulator, and run a variety of debuggers. MED supports repeatable and low-cost experiments, which are useful in testing processes. In contrast to existing work, MED combines an emulation-based debugging method with the semantics to handle non-determinism, allowing users to build powerful debuggers. We show the effectiveness of MED in a 20-switch SDN environment with real bug cases.

As future work, we would like to integrate MED more deeply with existing software development tools. We believe tools like MED can provide a convenient development environment so that developers can better integrate the SDN components into the cloud platform, providing improved flexibility and performance. Inside the emulator, we would like to further optimize the amount of states to test in non-deterministic executions by analyzing the interdependency of messages. We will also work on statistical learning based approaches for automatic probing, bug discovery and performance optimizations using the MED emulator.

VIII. Acknowledgment

Research supported in part by the National Natural Science Foundation of China grants 61361136003, 1000 Talent Plan grant, Tsinghua Initiative Research Program grants 20151080475 and a Google Faculty Research Award.

References

[1] ONF Market Education Committee.Software-defined networking: the new norm for networks, https://www.opennetworking.org/images/stories/downloads/sdn-resources/white-papers/wp-sdn-newnorm.pdf, 2015.

[2] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown, "Automatic test packet generation," in *Proceedings of the 8th international conference on Emerging networking experiments and technologies*. ACM, 2012, pp. 241–252.

[3] Troubleshooting the Network Survey, http://yuba.stanford.edu/peyman/docs/atpg-survey.pdf, 2015.

[4] G. Altekar and I. Stoica, "Focus replay debugging effort on the control plane," *Proc. USENIX HotDep*, pp. 1–9, 2010.

[5] A. Wundsam, D. Levin, S. Seetharaman, A. Feldmann *et al.*, "Ofrewind: Enabling record and replay troubleshooting for networks." in *USENIX Annual Technical Conference*, 2011.

[6] M. Canini, D. Venzano, P. Peresini, D. Kostic, J. Rexford *et al.*, "A nice way to test openflow applications." in *NSDI*, vol. 12, 2012, pp. 127–140.

[7] H. H. Liu, X. Wu, M. Zhang, L. Yuan, R. Wattenhofer, and D. Maltz, "zupdate: Updating data center networks with zero loss," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 411–422, 2013.

[8] M. Reitblatt, N. Foster, J. Rexford, and D. Walker, "Consistent updates for software-defined networks: Change you can believe in!" in *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, 2011, p. 7.

[9] X. Jin, H. H. Liu, R. Gandhi, S. Kandula, R. Mahajan, M. Zhang, J. Rexford, and R. Wattenhofer, "Dynamic scheduling of network updates," in *Proceedings of SIGCOMM'2014*, 2014, pp. 539–550.

[10] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: A network programming language," in *ACM SIGPLAN Notices*, vol. 46, no. 9. ACM, 2011, pp. 279–291.

[11] A. Voellmy, A. Agarwal, and P. Hudak, "Nettle: Functional reactive programming for openflow networks," DTIC Document, Tech. Rep., 2010.

[12] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. Godfrey, and S. T. King, "Debugging the data plane with anteater," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4, pp. 290–301, 2011.

[13] P. Kazemian, G. Varghese, and N. McKeown, "Header space analysis: Static checking for networks." in *NSDI*, 2012, pp. 113–126.

[14] J. A. Whittaker, J. Arbon, and J. Carollo, *How Google tests software*. Addison-Wesley, 2012.

[15] P. Phaal, S. Panchen, and N. McKee, "Inmon corporations sflow: A method for monitoring traffic in switched and routed networks," RFC 3176, Tech. Rep., 2001.

[16] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[17] N. Handigol, B. Heller, V. Jeyakumar, D. Maziéres, and N. McKeown, "Where is the debugger for my software-defined network?" in *Proceedings of the first workshop on Hot topics in software defined networks*. ACM, 2012, pp. 55–60.

[18] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown, "I know what your packet did last hop: Using packet histories to troubleshoot networks," in *Proc. NSDI*, 2014.

[19] K. Agarwal, E. Rozner, C. Dixon, and J. Carter, "Sdn traceroute: Tracing sdn forwarding without changing network behavior," in *Proceedings of the third workshop on Hot topics in software defined networking*. ACM, 2014, pp. 145–150.

[20] H. Zhang, C. Lumezanu, J. Rhee, N. Arora, Q. Xu, and G. Jiang, "Enabling layer 2 pathlet tracing through context encoding in software-defined networking," in *Proceedings of the third workshop on Hot topics in software defined networking*. ACM, 2014, pp. 169–174.

[21] G. Gheorghe, T. Avanesov, M.-R. Palattella, T. Engel, and C. Popoviciu, "Sdn-radar: Network troubleshooting combining user experience and sdn capabilities," in *Network Softwarization (NetSoft), 2015 1st IEEE Conference on*. IEEE, 2015, pp. 1–5.

[22] U. C. Kozat, G. Liang, and K. Kokten, "On diagnosis of forwarding plane via static forwarding rules in software defined networks," in *INFOCOM, 2014 Proceedings IEEE*. IEEE, 2014, pp. 1716–1724.

[23] P. Kazemian, M. Chan, H. Zeng, G. Varghese, N. McKeown, and S. Whyte, "Real time network policy checking using header space analysis." in *NSDI*, 2013, pp. 99–111.

[24] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: rapid prototyping for software-defined networks," in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*. ACM, 2010, p. 19.

[25] Open vSwitch: An Open Virtual Switch, http://openvswitch.org/.

[26] POX: An Operating System for Networks, http://www.noxrepo.org/pox/about-pox/.

[27] OpenFlow-enable commercial switch, http://www.pica8.com.

[28] C. Scott, A. Wundsam, B. Raghavan, A. Panda, A. Or, J. Lai, E. Huang, Z. Liu, A. El-Hassany, S. Whitlock *et al.*, "Troubleshooting blackbox sdn control software with minimal causal sequences," in *Proceedings of the SIGCOMM'2014*, 2014, pp. 395–406.

[29] Network operating system that enables customers to easily migrate from conventional networking to SDN using commodity bare metal switches, http://www.pica8.com/company/company-overview.php, 2015.

[30] Release-notes-for-picos, http://www.pica8.com/document/v2.3/html/release-notes-for-picos-2.3, 2015.