CoSEL: Control-plane Only Scalable Efficient and Lightweight SDN Debugger

Nihal Srivastava (nsnihal960@gmail.com), Gaurav Singh (gauravsingh0110@gmail.com), Huzur Saran (saran@cse.iitd.ac.in), Vinay Joseph Ribeiro (vinay@cse.iitd.ac.in), Suresh C Gupta (scgupta@cse.iitd.ac.in)

Indian Institute of Technology Delhi, India

Abstract- Network debugging has always been a challenging task primarily because the original design of the Internet architecture gave little importance to debugging and management. In addition, the Internet makes forwarding decisions in a distributed manner, which is hard to track. With the advent of Software Defined Networks (SDNs), network debugging has potentially become easier because solely the control plane makes forwarding decisions, which is often centrally located. In this paper, we propose Control-plane Only Scalable Efficient and Lightweight SDN Debugger (CoSEL), which is to the best of our knowledge the first control-plane only, network debugger for SDN. Unlike earlier proposed debuggers, which rely on information specially obtained from the data-plane or a combination of the data and control planes, CoSEL does not require the use of any additional network bandwidth for its operation and hence outperforms existing schemes in terms of scalability. In terms of completeness, however, CoSEL does not perform as well as some existing schemes, although this loss in completeness does not prevent it from capturing most of the bugs in the network, which hav[1]e been observed to be caused mainly by the control plane. We implement CoSEL on the Floodlight controller as a network application and present results of CoSEL in action over a network emulated by mininet. However, CoSEL is controller independent and utilizes a common programming API provided by POX, NOX and OpenDaylight. In addition, the control plane debugging of CoSEL enables it to be an effective debugging tool for network applications that are built on top of controllers utilizing Northbound APIs and interacting with network by injecting rules.

I. Introduction

SDNs (Software Defined Networks) have caused a paradigmatic shift in the way networks are configured, managed, and operated [9]. A SDN decouples the data plane and control planes by treating routers and switches as dumb forwarding boxes whose forwarding rules can be set by a centralized controller. This changes the nature of network from a distributed to a centralized one where the whole network is dependent on a software program in the controller. SDN also provides a platform for network applications and services to be developed on top of the controller. As a result, software bugs at the controller can cause faults in the entire network, thereby creating a need for an effective network debugger.

Systematic debugging of a network corresponds to determining errant behavior of switches and packet flows in the network. The debugger should be able to understand a given query and respond appropriately. There can be two modes in which a debugger may operate: Active and Passive. In the active mode, the network administrator gives specific queries to the debugger, such as asking it to trace route a packet, to which it responds. In passive mode, the debugger continuously inspects the network for an error, such as a routing loop, without receiving any specific user request.

NDB was one of the first systematic debuggers for SDNs [10]. It records forwarding information from the network, which it later utilizes for debugging purposes. Now, a large and complex network may carry a huge amount of data, and hence recording metadata based on all forwarded packets in network will itself require a considerable amount of network resources. Hence it is necessary for a debugger to be effective while at the same time not consume large fractions of network resources. To compare the performance of a debugger we propose the following metrics:

1. *Completeness:* This means all packet forwarding events are logged by the debugger.
2. *Soundness:* This means that no information stored in the debugger leads to a faulty diagnosis or to an incorrect conclusion about the occurrence of past events.
3. *Scalability:* This refers to how large a network the debugger can effectively operate in.
4. *Efficiency:* This refers to how low requirements of networking resource such as bandwidth are for effective operation.

Debuggers typically comprise of the following two modules.

1. *Network module*: This module is responsible for interacting with the network and recording all the relevant information that can be later used by the verification module.
2. *Verification module:* This module takes as input a query from the troubleshooter, utilizes the information provided by network module; formulates the response and responds to the troubleshooter in the required format.

In the literature, there are works that aim to develop either one of the network module [8] or the verification module [13]. In this paper, we focus more on the network module, as efficiency is more dependent on the network module, which consumes network resources. However, we design the network module so as to record all control plane events in a standard database, which can be independently accessed by the verification module. This results in decoupling of both modules, which is discussed in detail in §3.2. A recent study shows that control plane traffic shares only 1% of total traffic but is responsible for 95-99% of located bugs [6]. Thus we rely on the control plane for network debugging, where no deterministic guarantee [14] is provided for locating bugs. Our aim in this paper is to develop a tool, which locates most of the bugs in software-defined networks while using few network resources.

We discuss in §2, the basic motivation behind our work followed by proposing our solutions in §3, where we also provide a comprehensive comparison of CoSEL with a data plane based debugger and discuss the implementation details of CoSEL's network module as well as a few primitive verification modules for the Floodlight controller. We discuss CoSEL's functionality in §4 followed by a compilation of the most recent related work in §5, and then conclude our paper in §6.

II. MOTIVATION

NDB [10] defines its network module as a post-card collector where a post-card contains information about a packet and the state of the switch on which the packet resides. Every packet at every hop generates post cards and all this information is logged in the database using either in-band or out-of-band connections. In other words, all information related to packet forwarding in the data plane is recorded in the database, which uses up precious extra bandwidth. We utilize various capabilities of SDN to reduce the amount of data that needs to be recorded. Also, this data should contain sufficient information for debugging various network errors retrospectively.

SDN provides a clear separation between the control plane and the data plane which in turn leads to different choices for debugging: data plane debugging, control plane debugging, or a combination of both. Data plane debugging involves recording packet flows in a network, such as what NDB does, or creating periodic snap shots of a network and later using these information for debugging. Control plane debugging involves debugging based solely on information collected from the controller over time. Control plane debugging is powerful as well as efficient because solely the controller makes packet-forwarding decisions and also, the overall control information is much smaller than that of the data plane.

In this paper, we propose a paradigm where we place the debugger adjacent to the controller which records information solely from the control plane, thereby achieving high scalability, and removing the need for in-band or out-of-band connections, although simultaneously reducing soundness to some extent.

III. PROPOSED SOLUTION

We propose a controller-centric network debugger, which could be developed as a tool/application over the controller. We use the fact that whenever a packet is forwarded at a switch, a corresponding match-action pair needs to exist on the switch which in turn must have been forwarded by the controller. Recording these flow entries with some additional information, like timestamps etc., in a database directly is the main essence of such a debugger. Note, CoSEL records all switch-controller communications, which includes "*send-to-controller rules*".

In this section, we discuss what classes of queries are supported by such a debugger, followed by an architectural design of CoSEL, and a description of the basic working of CoSEL. We then describe how to recreate network state from the information stored by CoSEL and how to expand its capabilities and improve soundness. We finally compare CoSEL with generic data plane oriented debuggers.

A. Queries supported: CoSEL supports queries, which intrinsically can be answered with the help of routing paths between different nodes of the network. Examples are:

1. What happens to a packet that was sourced from a node and destined for some other node?
2. Node x was not expected to receive this packet from node y but why does it still do so?
3. Are there cycles in the network?

In an experiment in §3.6.2, we develop a trace route verification module, which traces the path of a packet sourced from a particular source to a particular destination to demonstrate the efficacy of CoSEL. Then, we also depict how a service added in CoSEL could locate packet loops in the network.

B. *Architecture:* We developed the CoSEL debugger for the Floodlight [1] controller and we discuss its fine details in §3.6. Figure 1 depicts the generic architecture of CoSEL. The network module in CoSEL utilizes a common programming API provided by many popular controllers (say JAVA API for Floodlight, Python API for POX[3] etc.) so as to interact with the network and record flow information in the database obtained by listening to messages being forwarded by the controller to switches or vice-versa. Later, verification module uses this information to formulate a response for a given query. Thus this architecture provides complete decoupling of network and verification modules of CoSEL. This enables one to record information with a timestamp, later filter useful details from the database and recreate network forwarding state at any instant of time as described in the user query.

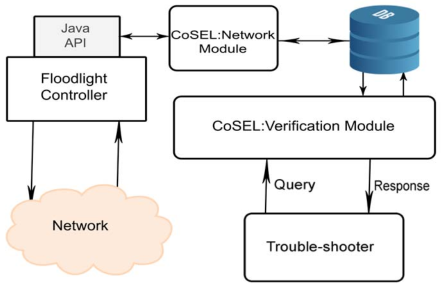


Figure 1: CoSEL: Generic Architecture

C. Methodology: CoSEL uses a property that all the flow forwarding information in the network is generated and sent to switches by the controller. CoSEL records such interaction between the switch and controller, which later is filtered out to create a network state at the instant specified by a query. Note that this information does not include switch port statistics, which may be of interest to a network operator. Such port statistics are altered on the arrival of a burst traffic of packets and can be requested via Switch statistics as specified in OpenFlow 1.1[2] onward. Trying to recreate switch states only from switch-controller interactions may lead to some soundness problems that are discussed in detail in §3.5.

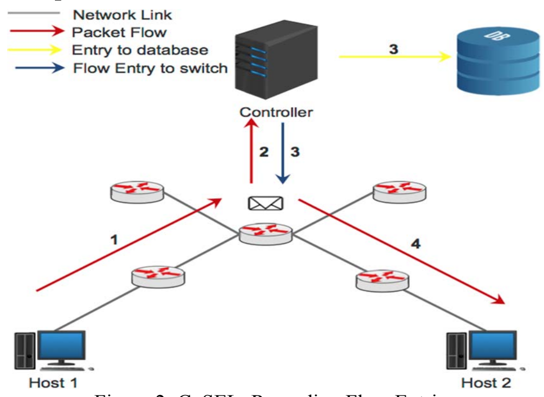


Figure 2: CoSEL: Recording Flow Entries

This methodology is depicted in Figure 2 where a switch upon receiving a packet for which it has no matching flow entry is forwarded to the controller, which in turn is responsible now not only for formulating a corresponding match-action pair and sending it to the switch but also to forward the same entry along with a timestamp and idle/hard timeouts to the database. In the case of all packets corresponding to same flow following this one, the switch does have a match action pair, and so forwards them without consulting the controller. Hence CoSEL makes no record of such packets. After the timeout of the match-action pair, however, the process repeats. In contrast, NDB makes an entry for all packets of a flow which leads to a large amount of possibly redundant data being stored in the database. Note that similar to flow insertion, flow removal messages also trigger an event which is recorded in the database.

The next task of the debugger is to recreate the network forwarding state at the instant specified in a user's query. To achieve this, we also store topology information in the database. Now, we can recreate network state by aggregating the topology and the forwarding state information (For implementation details, see §3.6).

D. *Expanding capabilities of CoSEL:* Our current implementation of CoSEL captures all flow entries at every switch. While debugging offline, if presented with a data packet that was input to the network at some switch, CoSEL assumes that the packet was forwarded successfully according to its match-action pair at all switches starting from this switch. This may not be true, however, as the packet may have been dropped because of buffer overflow at some switch along its potential path in the network. This reduces CoSEL's soundness. In order to improve CoSEL's soundness, future versions will request the controller to record statistical details provided by Port, Flow and Table Statistics of OpenFlow [2]. The controller can periodically broadcast such a request to switches and gather their responses in the database. The interval at which one needs to gather such information will be crucial. A finer interval will lead to better soundness but will also lead to more overheads in collecting information. We note that collecting such statistics does not change the control-plane-only nature of CoSEL. The addition of such functionality in CoSEL will allow it to support a larger class of queries. For example, the addition of port statistics information will enable CoSEL to better answer a typical query such as: "One consumer was supposed to get a bandwidth of 1 Gbps but instead got only 10 Mbps around mid noon. Why?"

E. *Comparison: CoSEL vs. NDB:* We now compare CoSEL with NDB over various evaluation metrics. Although there is a large set of network debuggers available in literature, like OFRewind[14], SDN Trace route[5] , Negative Provenance[13] etc., we compare CoSEL with NDB because the class of queries supported by NDB and CoSEL are closely related. Because other works answer different classes of queries, we cannot make a fair comparison with them. However, we discuss the working of these debuggers and compare them to CoSEL in §5.

1. *Completeness:* NDB is complete as it records all data plane events, provided no packet was lost while being transmitted to NDB's database. Since CoSEL does not record all packet-forwarding events, it is not strictly complete. However, CoSEL can recreate all routing state information at any past time instant. We term such a capability routing completeness. CoSEL is hence capable of debugging many routing-related queries.
2. *Soundness:* Only in the case of packet arrival, a switch can send a record to the database. Unless there is link error between switches and database, NDB is sound. In §3.4 we showed that CoSEL is not perfectly sound and describe ways to improve its soundness. Packet loss caused due to hardware malfunction or resource constraints at switches will not be recreated by the current version of CoSEL. However, CoSEL's soundness can be increased if the controller collects more state information from switches, such as port statistics (see §3.4).
3. *Scalability:* As Figure 1 depicts, CoSEL's database is connected directly to the controller and thus there is no requirement of either in-band or out-of-band connections. Hence CoSEL faces few scaling constraints. In contrast, NDB scales poorly because it needs more in-band or out-of-band bandwidth as the network grows larger, which may not be feasible to provision for.
4. *Efficiency:* Clearly the placement of the database adjacent to the controller also offers efficiency. Our debugger does not use any network bandwidth; it just needs a dedicated connection between the controller and database (or even an inbuilt database in the controller), whereas in the case of NDB, about 31%[10] of network bandwidth is required solely for debugger operations.

So CoSEL clearly outperforms NDB in case of scalability as well as efficiency.

F. Implementation: We implemented CoSEL[1] on the Floodlight-0.90 [1] controller and tested it on a network emulated by mininet [12]. We implemented the architecture presented in Figure 1 by creating the modules in Figure 3. However, we emphasize that CoSEL is largely controller independent as all APIs utilized by CoSEL are common to many popular controllers besides Floodlight, such as POX, NOX, and OpenDaylight etc. We made no modifications to the actual code of Floodlight other than altering timeout values and allowing matching with more fields than the default MAC address matching provided.[2]

a) Prototype: We now discuss each module depicted in Figure 3. In the network module, the FlowMod Listener listens to any flow insertion message sent from the controller to any switch and records it in the database. The FlowRem Listener records the flow removal notifications forwarded by any switch to the controller.

PortTracker and Topology have the responsibility of storing the topology information of the network in the database; PortTracker recognizes the nodes connected to various ports of the switches and Topology stores the network topology. In the database, the DBFlowMod and DBFlowRem tables store flow entries and flow removal messages that were forwarded by the network module. DBResult is used as an auxiliary table that stores the network state temporarily required at the time of debugging. The DBTopology table stores all the topology information along with timestamp values. The verification module implemented here is a primitive one mainly for illustrative purposes. Here, we implemented a simple trace route and a loop detector as described in §3.6.

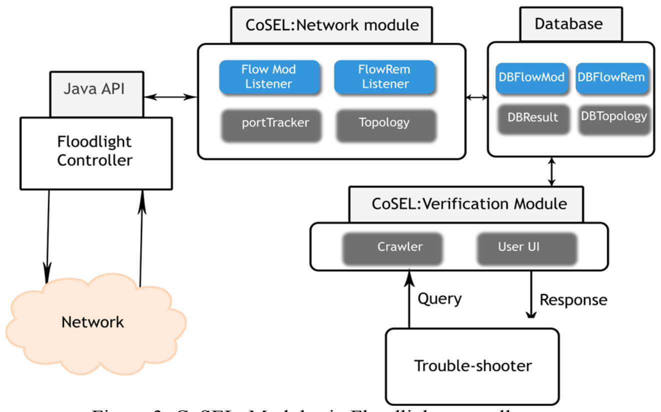


Figure 3: CoSEL: Modules in Floodlight controller

To get a clearer view of how CoSEL filters the flow entries to retain only those relevant to a particular query, we demonstrate the filtering process using Figure 4 where the number on each block represents a unique flow entry. First, we filter out all the flow event entries, which appeared later than the timestamp in the request, thus we remove 9 and 6 from our consideration in the DBFlowMod and DBFlowRem tables respectively. Then we filter out the flows that were received earlier than the hard timeout, like flow entry 0, because they would have been dropped anyway. Now, at last we filter the flow entries that were recorded as flow entries that were removed in DBFlowRem. Note, every flow removal entry will cancel out one flow modification entry, and so flow 1 still exists in the DBResult. The remaining flow entries are, thus ones, which exist actually at the instant of the timestamp in the query.

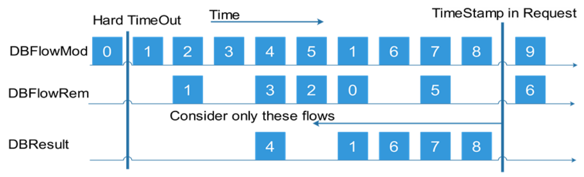


Figure 4: CoSEL: Filtering Flow Entries

After recreating the network state at a particular timestamp, one can easily obtain each and every flow that was matched by every switch along the path of the packet. Note, here we solely discussed micro-flows for easier demonstration; obviously CoSEL also records macro flows and its responsibility of verification module to utilize this information. Here, macro-flows represent an aggregated flow, such as using wild cards to combine various micro-flows.

We demonstrate the operation of various modules with an example in Figure 5, where the numbers on the switches represent the decimal form of its DPID (Data Path ID). Here, suppose burst traffic is directed from 10.0.0.4 to 10.0.0.12, then when the first of these packets reaches switch 4 where there is no matching entry, the packet is forwarded to controller that formulates a flow entry and forwards it to the switch. Now, this forwarding triggers an event called a flow modification event that is caught by the FlowMod listener and a copy of the message is stored in the database.

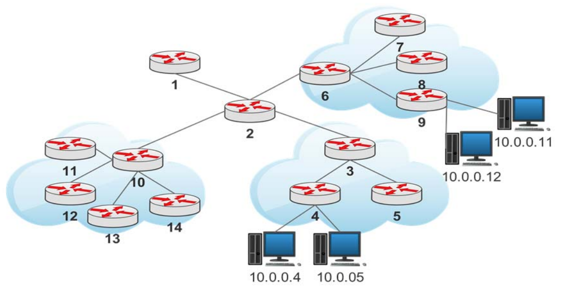


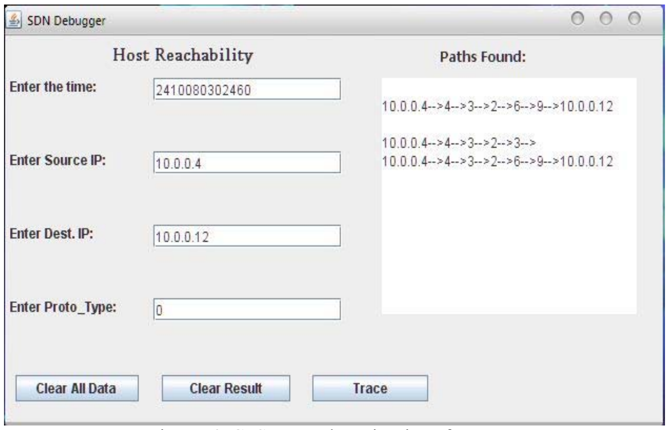
Figure 5: Example topology

Similar to the case for switch 4, entries are made in the database by a FlowMod listener for flow entry insertion in switches 3, 2, 6 and 9 along with timestamps. Now, when a timeout for a flow entry occurs and a Flow removal flag is set, the switch deletes the entry and also sends a flow removal message to the controller, which triggers a flow removal event. The FlowRem listener now catches this event and an entry is made in the database.

To utilize the stored information we can just look at flow entry records in the database. Suppose the network administrator receives a debugging request corresponding to disconnection between two nodes at a particular time. He/she can use the User Interface to fire a query on the trace route to detect what happens to a packet between those two nodes.

[2] The source code for CoSEL is openly available at https://github.com/gauravsingh0110/CoSEL.git

Now, trace route filters the network state and creates a pseudo network, utilizing forwarding state information from DBResult and link information from the DBTopology table, where it fires a crawler algorithm recursively to trace the path of a packet in the network and thus one can locate the faulty path. A faulty path clearly indicates the faulty flow entry and its resident switch thereby identifying the bug.



Enter the time: 2410080302460

Enter Source IP: 10.0.0.4

Enter Dest. IP: 10.0.0.12

Enter Proto_Type: 0

```
10.0.0.4-->4-->3-->2-->6-->9-->10.0.0.12

10.0.0.4-->4-->3-->2-->3-->
10.0.0.4-->4-->3-->2-->6-->9-->10.0.0.12
```

Figure 6: CoSEL: Debugging interface

b) *CoSEL in action*

To validate CoSEL in the active mode, we implemented trace route as a verification module, which offers tracing a packet in the network at any timestamp provided by a network administrator. We emulated a topology depicted in Figure 5 containing 20 hosts and 14 switches, which are identified by IP addresses and DPIDs (DataPathIDs) respectively. Each switch is connected to two hosts. To create a scenario of a buggy network, we use a static flow pusher API[4] to manually insert flows in the OpenFlow network. Note that although we insert flows manually, the static flow pusher is an API provided by the Floodlight controller and so this entry is indirectly sent as a flow modification message by the controller only, thus making a record in the database.

We tested if trace route functions properly by tracing the path of a packet in a normal as well as buggy network. A snapshot of the tool is depicted in Figure 6, which shows the interface provided to a network administrator to input the source IP, destination IP, timestamp and network protocol.

We carried out an experiment as follows:

i. Host 10.0.0.4 pings 10.0.0.12, along with the pingall command running in the background, which produces traffic between all possible pairs of nodes.
ii. We introduce an error in switch 5 using a faulty flow entry where the switch reverts a packet back to the ingress port if it was sourced from 10.0.0.4 and destined to 10.0.0.12.
iii. We now remove the erroneous flow by inserting the flow entry with a proper forwarding decision with higher priority.

We traced the path of the packet after all three events. Figure 6 shows the correct tracing of packets in the network in all three situations, and shows (10.0.0.4 -> 4 -> 3 -> 2 -> 3) even after the error was introduced in the network. This can help the network administrator determine that the error occurred at switch 2 and also can help him/her identify the flow entry responsible for it in the logs.

To verify the functioning of CoSEL in the passive mode, we develop a loop detection service in CoSEL. We set the frequency of operation of the loop detector to be 30 seconds. To devise a simple loop detector, we just filter the network state at the current instant and corresponding to all flows originated at all the ports of switches which were connected to a device, we traverse the path and check if a loop is present. We note there are various graph cycle detection algorithms[11] available in literature which could be deployed to build a more efficient verification module for the same task and only implement the loop detector in this fashion for illustrative purposes.

To introduce a loop in the network, depicted in Figure 5, first we add a link between switch 2 and switch 4, then we insert a flow in switch 2 which forwards a packet to switch 4 that was sourced from 10.0.0.4 to 10.0.0.12. The packet hence loops from switch 4 -> 3 -> 2 -> 4. In very few seconds just after we insert the flow and the ping from 10.0.0.4 to 10.0.0.12, we receive a notification that a cycle is detected in the network.

IV. DISCUSSIONS

In addition to high level functionalities provided by CoSEL, one can easily formulate various low level debugger primitives like watch, breakpoints and steps. An implementation of step can be the hopping of a packet one by one through various switches, a switch can be used as a point in breakpoints and watch can show variables like various flow entry fields, switch statistics etc. which all are stored in CoSEL database. Also, CoSEL can point out the entry that was responsible for errant behavior but cannot point out the actual line of code that was responsible for such behavior. We will need to incorporate gdb or its equivalent into CoSEL to achieve such functionality.

CoSEL is protocol independent and merely requires Open-Flow enabled switches. It is designed to be controller independent, compatible with any network utilizing OpenFlow switches and requires almost no human interaction in its functioning, other than understanding the response of a query. Also, with newer network applications that are built on top of the controllers, bug susceptibility in networks increases due to incorrect formulations of flow entries by such applications, which later are injected into the network. We believe that CoSEL can double up as an effective application debugger for debugging such network applications, as all bugs in such applications must result in few buggy flow entries, which could be located by CoSEL efficiently and promptly.

V. RELATED WORK

Other than NDB, which is probably the most comprehensive debugging tool available for generic software defined networks; there have been other attempts at debugging such as NetSight [16], AntEater [17], HSA [18], Veriflow [19], OFRewind [14] etc.

Further expansion in the capability of NDB [16] tries to make NDB scalable by reducing the overhead of 31% bandwidth requirement to 3-7%. NDB achieves this by reducing the size of data that needs to be recorded by differential compression of packet headers that are being collected. This is achieved by switch-assisted (7%) and host-assisted (3%) computations in the network, whereas CoSEL has no requirement of hosts/switches assisted computation and network devices are completely unaware of CoSEL. AntEater [17] serves as a verification module, which tries to answer various debugging queries by simulating the network invariants as SAT formulae and then solving them. It analyses the network snapshot data to create such SAT formulae and then tries to solve them for given packet header and thus providing results. It does not exploit any special architectural feature of SDN and serves to work in even in trivial network architecture. This SAT formulation and solving them requires additional computations, which is not the case in CoSEL. HSA [8] treats a network as black box and troubleshoots network for various issues. It sends special packets in network and observes their behavior in network to find various bugs in the network. HSA utilizes no SDN capability and is compatible with a trivial network architecture also. HSA triggers some additional packets in the network to monitor their trajectory and header modifications. This again consumes network bandwidth (which is high in some cases like loop detection where multiple packets from multiple ports are injected). VeriFlow [9] poses the problem that all trivial network debugger try to find bugs in network when the bug has already affected the network. VeriFlow tries to prevent such bugs from entering in the network. This is achieved by intercepting the inter communication between controller switches. VeriFlow checks network invariants in real time and thus constantly checks for error as the network evolves. OFRewind records selective traffic in both control and data plane, but differs in the mode of collection of data, where OFRewind relies on switches, also, to inject data in data store, which is distributed in contrast to the sole centralized data recording in CoSEL. In addition, CoSEL solely relies on the control plane for debugging.

Other than these widely accepted tools, some recent advances have also been made in this field. SDN Traceroute [5], which traverses the path taken by a packet in the network using switch coloring and very few extra pre-installed flow entries in the switch. This tool, however, runs trace route on an existing network in real-time and is not retrospective in nature. VeriCon [7] tries to test various controller applications over all admissible topologies by using first-order logic. Negative provenance[13] discusses a new approach for a verification module where one can asks question such as: “Why a certain rule was not installed?”, or “Why a certain packet did not arrive?”, and show how such why not queries can be answered using a provenance graph. PktTrace [15] traces the path of IP packet in software defined data centers running various network services by placing various detection points in various paths in the network.

VI. CONCLUSIONS

This paper proposes a new approach towards systematic debugging of SDN by utilizing only control plane information. We propose architecture for data collection so as to eliminate overhead on network bandwidth and thereby offer higher network scalability, independent of debugger operations. We demonstrate how various trivial trouble shooting operations can be incorporated into CoSEL by implementing few verification modules like trace route and loop detection. CoSEL aims to provide a retrospective, easily deployable, scalable, efficient, and lightweight debugging tool for network administrators that has the potential to locate a vast percentage of the bugs (95-99%[6]) and simultaneously requires few network resources. CoSEL also serves as a preferred debugger for debugging various network applications, built on top of the controller, by locating erroneous flow entries in the network.

REFERENCES

[1] Floodlight controller. At http://www.projectFloodlight.org/Floodlight/.

[2] OpenFlow specification 1.1. at http://archive.OpenFlow.org/documents/ OpenFlow-spec-v1.1.0.pdf.

[3] Pox controller. At http://www.noxrepo.org/pox/about-pox/.

[4] Static flow pusher API. At http://www.OpenFlowhub.org/display/ Floodlightcontroller/Static+Flow+Pusher+API.

[5] 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, pages 145–150. ACM, 2014.

[6] G. Altekar and I. Stoica. Focus replay debugging effort on the control plane. In Proceedings of USENIX HotDep, 2010.

[7] T. Ball, N. Bjørner, A. Gember, S. Itzhaky, A. Karbyshev, M. Sagiv, M. Schapira, and A. Valadarsky. Vericon: Towards verifying controller programs in software-defined networks. In Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, page 31. ACM, 2014.

[8] B. Claise. Cisco systems netflow services export version 9. 2004.

[9] N. Feamster, J. Rexford, and E. Zegura. The road to SDN. Queue - Large-Scale Implementations, 11, December 2013.

[10] N. Handigol, B. Heller, V. Jeyakumar, D. Mazi˜Al’res, and N. McKeown. Where is the debugger for my software-defined network? In Proceedings of HotSDN ’12, pages 55–60. ACM New York, 2012.

[11] Kosaraju, S. Rao, and G. Sullivan. Detecting cycles in dynamic graphs in polynomial time. In Proceedings of the twentieth annual ACM symposium on Theory of computing. ACM, 1988.

[12] Lantz, Bob, 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.

[13] Y. Wu, A. Haeberlen, W. Zhou, and B. T. Loo. Answering why-not queries in software-defined networks with negative provenance. In Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks, page 3. ACM, 2013.

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

[15] H. Zou, A. Mahajan, and S. Pandya. Pkttrace: A packet life-cycle tracking tool for network services in a software-defined data center. VMware Technical Journal, Summer 2014 Edition, 2014.

[16] Handigol et al. I know what your packet did last hop: Using packet histories to troubleshoot networks, Proc. USENIX NSDI 2014

[17] Mai, Haohui, et al. Debugging the data plane with anteater. ACM SIGCOMM Computer Communication Review 41.4 (2011): 290-301. APA

[18] Kazemian, Peyman, George Varghese, and Nick McKeown. Header Space Analysis: Static Checking for Networks. NSDI. 2012.

[19] Khurshid, Ahmed, et al. Veriflow: verifying network-wide invariants in real time. ACM SIGCOMM Computer Communication Review 42.4(2012): 467-472