Track: Tracerouting in SDN Networks with Arbitrary Network Functions

Yuxiang Zhang*, Lin Cui*‡, Fung Po Tso†, Yuan Zhang*
*Department of Computer Science, Jinan University, Guangzhou, China
†Department of Computer Science, Loughborough University, LE11 3TU, UK
‡Guangdong Key Laboratory of Big Data Analysis and Processing, Guangzhou, P.R.China
samuelzyx0924@gmail.com; tcuilin@jnu.edu.cn; p.tso@lboro.ac.uk; michellinyuan@gmail.com

***Abstract*—The centralization of control plane in Software defined networking (SDN) creates a paramount challenge on troubleshooting the network as packets are ultimately forwarded by distributed data planes. Existing path tracing tools largely utilize packet tags to probe network paths among SDN-enabled switches. However, network functions (NFs) or middleboxes, whose presence is ubiquitous in today's networks, can drop packets or alter their tags – an action that can collapse the probing mechanism. In addition, sending probing packets through network functions could corrupt their internal states, risking of the correctness of servicing logic (e.g., incorrect load balancing decisions). In this paper, we present a novel troubleshooting tool, *Track*, for SDN-enabled network with arbitrary NFs. *Track* can discover the forwarding path including NFs taken by any packets, *without* changing the forwarding rules in switches and internal states of NFs. We have implemented *Track* on RYU controller. Our extensive experiment results show that *Track* can achieve 95.08% and 100% accuracy for discovering forwarding paths with and without NFs respectively, and can efficiently generate traces within 3 milliseconds per hop.**

***Keywords*—Network Diagnostics, Network Function, Software-Defined Networking, Traceroute**

I. Introduction

Software Defined Networking (SDN) has seen unprecedented adoption in recent years due to its ability in enabling control plane programmability for networks. SDN has opened up a profound opportunity for network operators to dynamically provision their networks and services in response to the ebb and flow of user demand as well as the rapid changing business environment. While the development of SDN is still at its infancy, coming with its numerous benefits are as many challenges. Among them, the most prominent one is troubleshooting as there has been a lack of effective tools for diagnosing anomaly events when traffic does not behave as expected [1].

SDN traceroute [1] is the first tool for querying the current path taken by *any types of packet*, e.g., Ethernet frames, IP, TCP/UDP packets and so on, in an SDN-enabled network. This tool works by installing several highest-priority rules into SDN switches for forwarding tagged probing packets (using a VLAN priority tag) to the SDN controller for keeping track of the ordered list of SDN switches traversed by the path-probing packets.

However, *SDN traceroute* cannot work correctly in a network with network functions (or middleboxes)[1] because some NFs, such as proxy and load balancer [4], could modify packet headers and/or payload. Once the tag fields (or header fields for matching) are modified by the NFs, subsequent switches would fail to recognize probing packets from normal packets and no longer forward them to the controller.

In comparison, *SFC Path Tracer* [5] is able to trace paths consisting of NFs. However, it also relies on tags probing packets (by flagging 2-bit Explicit Congestion Notification (ECN) in IP header) that must not be modified/dropped to discover NFs along the path. In addition to tagging, *SFC Path Tracer* identifies the type of NFs that have forwarded tagged packets to the controller NFs through looking up their device IDs from predefined topology. This will greatly limit its usability when a person has only partial or no access to the topology information. Worse still, since considerable number of NFs are stateful, sending probing packets through them may corrupt their internal states, jeopardising the correct servicing of ongoing production traffic.

In this paper, we present *Track*, an efficient and effective tool for querying paths in an SDN-enabled network with NFs. *Track* treats the whole path as several sub-paths joined by NFs. *Track* injects a probing packet with user-defined header fields into network to trace each sub-path. In the meantime, it runs a correlation procedure to infer behaviors of NFs and concatenate all sub-paths in correct order according to correlation results. Thus, *Track* does not send probing packets through NFs. Our correlation approach also eliminates the requirement of look-up of NF's ID from pre-defined topology information. Better still, *Track* does not modify the forwarding rules of production flows so that it does not affect network performance.

In short, our main contributions are as follows:

Corresponding author: Lin Cui (tcuilin@jnu.edu.cn)

This work is partially supported by Chinese National Research Fund (NSFC) Project No. 61402200; the UK Engineering and Physical Sciences Research Council (EPSRC) grants EP/P004407/1 and EP/P004024/1; the Fundamental Research Funds for the Central Universities (21617409); the Opening Project of Guangdong Province Key Laboratory of Big Data Analysis and Processing (2017009).

[1]Network functions (or middleboxes) [2] are very common in today's networks to improve security and performance. Across all network sizes, the number of middleboxes is on par with the number of routers in a network [3].

1) We design an effective *Track* scheme which can discover the whole path, including NFs of service chains, in an SDN-enabled environment.
2) We introduce a correlation procedure to infer the type of NFs in question rather than sending probing packets through them, preserving their internal states.
3) We implement *Track* on RYU and our extensive experiments on Mininet show that it can return the correct paths 100% and 95.08% of the time when it runs on an SDN-enabled network without and with NFs respectively.

The remainder of this paper is organized as follows. Section II describes detailed design and implementation of *Track*, followed by Section III, which extensively evaluates the performance of *Track*. Section IV outlines related work on SDN troubleshooting and policy enforcement in SDN network. Finally, Section V concludes the paper.

II. System Design and Implementation

A. Design Principles

Track is a diagnosing tool for debugging in SDN environment with NFs. In designing *Track*, we target a solution adhered to following principles:

- **Do not corrupt NF states:** NFs are states sensitive. The outputs of the function processing have strong correlations with these states (such as load balancers, proxies, and etc.). If probing packets modify these states, it would not only impact the performance of these NFs but also the correctness of the output.
- **Do not modify NF service logic:** There are many types of NFs and each has many forms of implementations. It is difficult, if not impossible, to understand each form of implementation or establish a standard API for each commodity implementation.
- **Do not modify production rules:** Forwarding rules of production flows are not allowed to be affected during the probing process. This will make *Track* readily deployable in production network environment.

In addition to these design principles, we also make a practical assumption that the controller has a global view of the network. More specifically we assume that: 1) the controller knows the topology of a given network, and 2) the controller knows which switch has an NF attached to it. For the ease of description, we call these switches connected to NFs as *NF-switches*.

B. System Architecture

Track, as demonstrated in Figure 1, has two main components: Correlation Module and Tracing Module. Correlation Module runs the correlation procedure which is detailed in Section II-C while Tracing Module applies a tracing procedure which is described in Section II-D and this procedure is similar to the way of *SDN traceroute* getting the routes.

The test-bed prototype of *Track* is implemented as a component of RYU controller [6] and it communicates with SDN switches using OpenFlow 1.3 protocol.

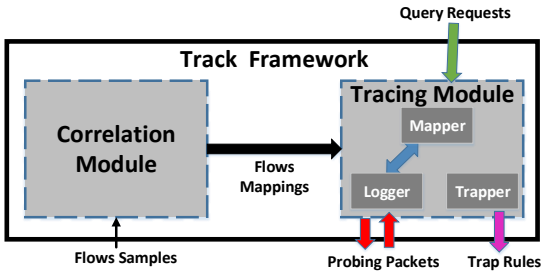


Fig. 1. Overview of the *Track*. Red arrows depict probing packets; green arrow depicts client query.

TABLE I
A taxonomy of different network functions

Network functions	Actions	Info needed	Approach	Type
Monitor	No change	None	-	1
IDS	No change	None	-	1
Firewall	Drop?	None	-	4
IPS	Drop?	None	-	4
NAT	Rewrite header	Header mapping	Payload match	2
Load balancer	Rewrite header & reroute	Session mapping	Payload match	2
Proxy	Map session	Session mappings	Similarity detector	3
WAN Opt.	Map session	Session mappings	Similarity detector	3

C. Correlation Module

1) NFs Classification: NFs provide security and performance guarantees. Typically, when a packet traversing a service chain, NFs may drop this packet or dynamically modify its headers and contents. For example, firewalls may drop packets according to specific rules while NAT (or proxy) may change packet header fields (and modify packet's content). According to these behaviors, we roughly classify NFs into 4 types, as tabulated in Table I. Particularly, *type 1* NF is the simplest case which neither change the packet headers and nor multiplex/spawn flows. While *type 4* NF does not modify packet, it may drop the packet. Both *type 2* and *type 3* NF may modify packets, but *type 2* only modifies packet header while *type 3* may modify packet content.

In addition to aforementioned classification, some NFs are stateful. For example, if the number of SYN message sent from particular host exceeds a threshold in IPS, the flows from this host may be recognized as intrusion. These states are crucial for NFs which is the main factor influencing the NFs' processing.

2) Correlation Procedure: Rather than modeling NFs or asking network administrators to specify the dynamic behaviors of NFs, we treat NFs as blackboxes and infer their relevant input-output behaviors [7], as classified in Table I. We name this phase as *correlation procedure*.

Note that we do not need access to the internal proprietary logic of the NFs. We only need to reason about the NFs behaviors pertinent to packets' forwarding. That is, we only

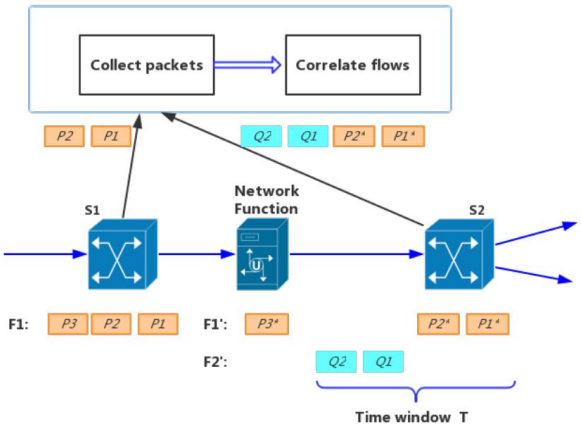


Fig. 2. Similarity based correlation of incoming and outgoing flows through an NF.

need to identify how the incoming and outgoing flows at the NFs are correlated.

As we summarized in Table I, for *type 1* NFs, we can directly map the incoming and outgoing flows (which we marked as None in the information needed column in the table). Consider that *type 2* NFs merely rewrite packet headers, we can simply perform an exact payload match between the incoming and outgoing packets to detect the flow correlations (which is labeled as payload match in the TABLE I). The most challenging case is to infer the behaviors in *type 3* NFs because they may create new sessions or merge existing sessions. For these NFs, we cannot directly match the payloads of individual packets. Instead, we have observed that even though the traffic is not identical after it traverses the NFs, the payloads will still have a significant amount of partial overlap. In light of this, we utilize Rabin-Karp algorithm [8][9] to calculate the similarities across flows. Given these insights, correlation procedure is comprised of four steps, as depicted in Figure 6:

i. Collecting packets – When a new flow arrives at an NF-switch, the switch sends the first P packets of the new flow to the controller. Similarly, we collect the first P packets for all the flows going out of the NF within a time window T. The controller reconstructs the payload stream from the P packets collected for each flow.
ii. Flows mapping – For *type 1*, we can directly map the incoming and outgoing flows which traversed this NF. Moreover we can do an exact payload match between the incoming and outgoing packets of *type 2* NFs.
iii. Calculate payload similarity – As discussed earlier, *type 3* NFs may modify or reorder part of the traffic flows, and we use Rabin-Karp algorithm to compute a similarity scores which represent the amount of overlap between every pair of flows [7].
iv. Identify the most similar flows – We identify the flow going out of the NF that has the highest similarity with the new incoming flow. We set a threshold for determining whether this correlation is right. If the similarity is lower than the threshold [8], we consider this is a correlation mistake and recognize this is a *type 4* NF which drop this incoming flow. Otherwise, we correlate these two flows. Note that *type 4* NFs can be easily mis-identified as *type 1* NFs if specified flows are not dropped by them. But this has little impact on later tracing procedure.

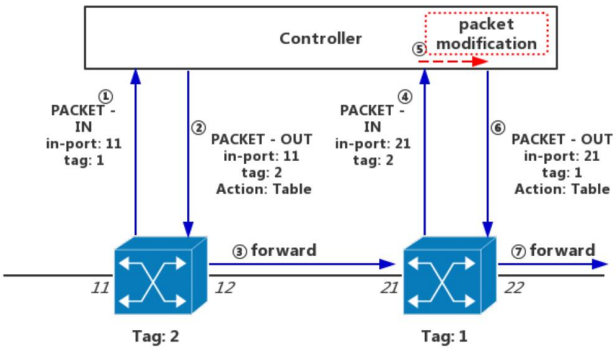


Fig. 3. Example for *Track* working procedure.

3) Implementation of the Correlation Module: The controller installs rules at NF-switches to retrieve the first few packets for each new flow. We use a custom implementation of the Rabin-Karp algorithm configured with an expected chunk size of 16 bits [7]. Note that if a flow is forwarded to multi-paths, *Track* just correlates one of these paths to the original flow.

D. Tracing Module

1) Pre-installed rules: Before sending any probing packets, *Track* must install rules that allow it to selectively trap probing packets. The rules must support two different tasks: i) matching the incoming probing packets so the hop can be logged at the controller, and ii) forwarding the controller returned probing packets as normal packets. In this section, we outline what rules are installed into the switches in order to achieve the first task. In the next section we will show how to modify those rules to perform the second task.

Similar to *SDN traceroute*, *Track* begins by applying a graph coloring algorithm to the topology and this coloring algorithm assigns each switch a color such that no two adjacent switches (switches directly connected via a link) are assigned the same color. These colors serve as tags that are an integral part of the rules required by *Track*. *Track* requires all probing packets to carry a tag so that switches can differentiate probing packets and normal packets and our implementation uses the VLAN ID field [2] as the probing tag. Our goal is to have switches log all packets except packets tagged with the default tag (production traffic) and those tagged with their own color. To meet this goal, *Track* installs rules in each switch to match the color of all adjacent switches. These rules are assigned the highest priority and forward the matched packet to the controller. Normal flows that do not carry this tag can be forwarded along the service chains. Note that these rules must be installed before *Track* first running.

[2]When VLAN is used in the network (e.g. the VLAN service chaining architecture), we can use the MPLS label field as the probing tag.

2) Tracing procedure: Once the network is configured in the manner discussed above, it is ready to accept user requirements for route tracing. The whole tracing procedure is explained in Figure 3. Users need to specify packet header (e.g. source/destination IP address, source/destination port and so on). Then *Track* constructs the packet with user specified packet header fields and tag, and identify the injection point which is the switch connected to the source specified by user. Meanwhile, *Track* identify the end point switch according to user-specified destination. After performing the trace route, the program returns an ordered list of switch_dpid and NF point corresponding to nodes which the packet traversed in the network.

Track would send the probing packet to injection point with the input port set to the source host and begin tracing service chain's forwarding path. Since each switch is configured to trap all packets matching the neighboring switches colors, the probing packet would be sent to the controller as a PACKET_IN (step 1). *Track* receives the packet at the controller and logs the switch-dpid which forwarded the packet to the controller as one hop in the forwarding path.

If current hop is not an NF-switch, *Track* would modify the probing packet by rewriting the reserved tag field to the bits corresponding to the color of the current switch. It then sends the probing packet back as a PACKET_OUT to the same switch that had sent the PACKET_IN message (step 2). The input port in the PACKET_OUT is set to the input port where the packet was received at the switch. The switch receives the probing packet from the controller and forward it to neighbor switch according to the flow-tables (step 3). This ensures that the actual forwarding rules in the switch are used to route the packet even though it is a probing packet and not production traffic.

While current hop is an NF-switch, *Track* check the flows correlation mappings in this switch to infer the actions in NF (step 6). If it is a *type 1* NF, *Track* would not modify the packet. If it is a *type 2* or *type 3* NF, *Track* would modify the probing packet as the NFs do (according to the mappings) and preserve the probe tag. If the correlation procedure infer that this packet would be dropped at this NF, *Track* would terminate the tracing procedure and output the routes it traced. Otherwise, including former three type NFs, *Track* would rewrite the tag field and send it back to the same switch (with modification of headers) as a PACKET_OUT (step 5).

This process (step 2 to step 4, sometimes step 6) repeats for each hop in the path. The process terminates when the probing packet arrives at end point switch then output the route *Track* traced. Notice that *Track* only log down the information of PACKET_IN messages with probe tag. This allows for scenarios where regular packet processing at a switch may itself initiate a PACKET_IN to the controller, such as in reactive rule installation.

3) Implementation of Tracing Module: Tracing Module uses the interfaces of RYU to construct probing packets, send probing packets to switches via an ofp_packet_out message and receive them from a switch via an ofp_packet_in OpenFlow message. When receiving a probing packet with tag, Tracing module record this switch which forwarded the packet to the controller and modify the probing packet if needed (according to the mappings Correlation Module provided). Among three sub-modules in Tracing Module, *Trapper* is responsible for installing "trap" rules while *Logger* is used for probes construction, sending/receiving probes and log the routes and *Mapper* does the packet modification.

III. Evaluation

A. Evaluation Setup

We have implemented a prototype of *Track* on RYU controller. And we have extensively evaluated its performance in Mininet [10] on a server with a 4-core Intel i7-4790 3.6GHz CPU and 16 GB memory as well as Ubuntu 14.04 Linux operating system. The experiment adopted the Internet2 [11] advanced layer2 service topology, which contains nearly 70 nodes including switches, hosts and NFs. Each link has an emulated bandwidth of 100 Mbps.

We deployed Bro [12], PRADS [13], Squid [14], and iptables [15] to act distinct types NFs in the network, i.e., *IDS*, *Monitor*, *Proxy*, *Firewall*, *NAT* [16]. Each NF is directly attached to a switch through a network link. Network traffic among hosts are generated by *iperf* [17]. Each traffic flow can be identified by port numbers and IP addresses of the sender and receiver. Each flow is subjected to at most on service chain contains one or multiple NFs above. Totally 325 service chains are installed in the network with different order or number of these five NFs and every pair of hosts has at least 2 service chains.

The time window T is set to be 200ms which is a better setting to get a trade-off between running time and accuracy [7].

In addition to *Track*, we also implement the *SDN traceroute* [1] in our controller as a comparison. We did not implement SFC Path Tracer [5] because the paper does not reveal the implementation details. Since *SDN traceroute* failed to work with the presence of some NFs, e.g., Proxy, it may only return a partial resulting path in some scenarios.

Two metrics are considered: *accuracy*, which is obtained through repeatedly installed random routes and verified that *Track* can correctly discover them, and *latency*, including the end-to-end time of conducting traces on various network paths and time to discover one single hop (e.g., switches or NFs).

B. Performance of Tracing

We first evaluate the latency performance of both *Track* and *SDN traceroute*. We triggered both *Track* and *SDN Traceroute* to trace the forwarding paths in an SDN environment with/without NFs. Figure 4 summarized the latency of *Track* and *SDN traceroute* tracing paths of varying length. A hop is defined as an intermediate switch and one hop contains one NF at most. We measured the latency of a variety of paths which range from one to ten hops in Figure 4. Experiment results show that the performance of *Track* is better than *SDN traceroute's* when tracing service chains. The average per-hop latency of *Track* is 2.4583ms while *SDN traceroute's* is

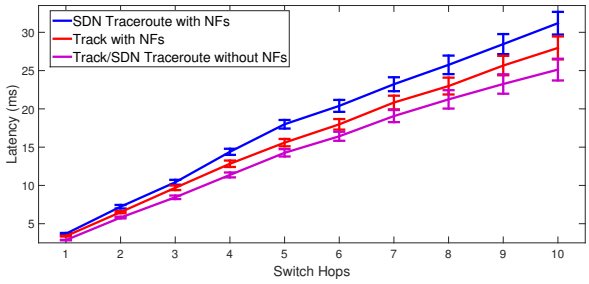


Fig. 4. Latency to trace a path with different number of hops in SDN environment with/without NFs.

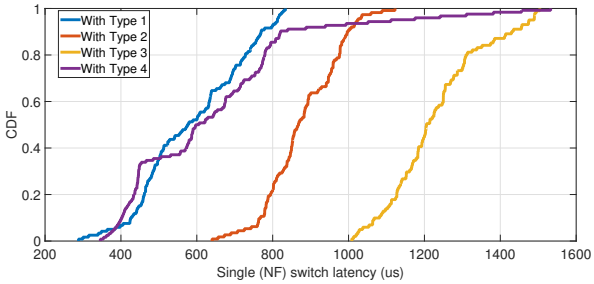
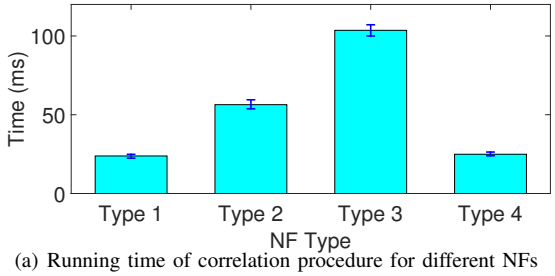


Fig. 5. The CDF of processing time for each types of NFs.

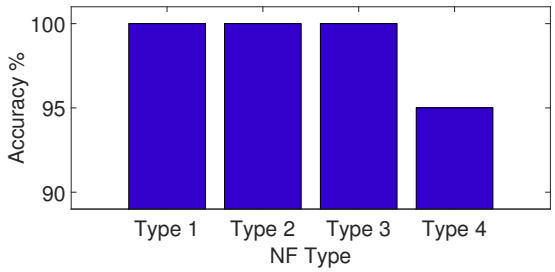
2.7582ms. Moreover, *Track/SDN traceroute* can trace paths in an SDN environment without NFs with the average 2.2279ms per hop latency. The experiments results showed that *Track* is effective tool for tracing paths under an SDN environment with/without NFs and it returns the whole paths in both scenarios with faster average response time. Each point shows the average of 30 runs with error bars showing standard deviation.

Next we verify if *Track* can correctly discover forwarding rules. As *Track* gives the correct path 100% of the time in an SDN environment without the presence of NFs, we focus on studying the accuracy performance of *Track* with the presence of NFs. We ran *Track* to trace each service chain in the network and to validate its functionality. We analyzed the accuracy of *Track*. In these 325 service chains, *Track* can correctly obtain 309 paths. The accuracy rate is 95.08%. Then, we checked the incorrect outputs of *Track* and compared them to the correlation result. We found that mis-correlating input and output of proxy is the main cause of tracing error. As we mentioned above, inferring the correlations between input and output flows of proxy is a great challenge because proxy creates/multiplexes sessions and changes packet contents.

Then we record the processing time in controller. While tracing routes, *Track* needs to log the hop or modify packets. We define the period of time from receiving a PACKET_IN to sending a PACKET_OUT as the processing time. If current hop is not an NF-switch, *Track* just needs to log this hop and sends the probing packet back. Otherwise, *Track* needs to search the correlation between the input and output of the NF which this switch connects. For *type 1* or *type 4* NF, *Track* just needs to send the probing packet back or drop it. But for *type 2* or *type 3* NF, *Track* needs to modify packet according



(a) Running time of correlation procedure for different NFs



(b) Accurate rate of the correlation procedure of different type of NFs

Fig. 6. Running time and accuracy of correlation procedure operating on each type NFs

to the outcome of correlation procedure. Figure 5 shows the CDF of controller running time in one hop. From Figure 5 we could tell that processing time in controller is lower than 1ms in most cases.

C. Performance of Correlation

Next, we evaluate the correlation procedure with focuses on each type NFs and whole service chain. We first conducted experiments on validating the effectiveness and efficiency of correlation procedure upon all types of NF. Figure 6 shows the running times of correlation for each type NFs and their accurate rate and each point shows the average of 40 flows with error bars showing standard deviation. *Type 1* and *type 4* NFs cost smaller running overhead relatively during the correlation procedure with average 23.823ms and 24.92ms running time respectively. 56.39ms is the average processing time for correlating the flows through *type 2* NFs while 103.581ms is for *type 3* NFs. The accuracy of *Track* performing on all types NF reaches 100% except *type 3* NFs. For *type 3* NFs, which is the most challenging part of the correlation, *Track* can still get 95% accuracy. These experiment results confirm the effectiveness of correlation procedure.

TABLE II
RUNNING TIME AND ACCURATE RATE FOR A SERVICE CHAIN WITH VARIED NUMBER OF NFS

Number of NFs	Running time(ms)	Accurate rate
1	0.36	96.67%
2	23.13	95%
3	48.52	96.67%
4	71.79	96.67%
5	103.44	95%

We then investigate the performance of correlation procedure when inferring the behaviours of all NFs in a service

chain. To be more specific, time window is not included in running time. Because running time is the processing overhead in controller during the correlation procedure, which can be performed offline. Table II shows that for a service chain of 5 NFs, *Track* can get about 95% accuracy within 104ms processing time and most errors occurred during inferring correlation between the input and output of proxy.

IV. Related Works

Troubleshooting is an important issue in computer networks. The simplest tool that provides visibility into a network is Traceroute [18]. However, it is extremely limited in troubleshooting SDN-enabled networks because it can only provide the layer-3 (IP) path information since it relies on time-to-live (TTL) field in the IP header to trigger ICMP error messages from intermediate routers. It also assumes routing and forwarding are destination-based.

SDN traceroute [1] is a Traceroute-like tool for SDN environment. It injects tagged probe packets in to network to discover the forwarding behavior for specified flows by mirroring packets with specific tags to the controller. It will fail to work once probe packets are dropped or tags are modified.

Hybridtrace [19] can reveal the routing path taken by a packet that needs to traverse multiple legacy network islands and SDN network islands to reach its destination. When it traces routes in legacy island, it uses a traceroute-like method which triggers the routers that send back an "ICMP Time Exceeded" error message to the source host. When it tracks the forwarding paths in SDN-enabled network, it triggers SDN controller to send probe packets and log down each hop. Netography [20] is a system that troubleshoots the network leveraging packet behavior and flow rules to locate and dig root causes of network issues. Netography sends probes into networks and these probes would be copied once a time with the information about the matched flow rules and sent back to it. By analyzing the list of copies of probes, network operators can locate root causes.

In contrast to these tools, *Track* and *SFC Path Tracer* [5] are the only tools that are capable of querying network paths with presence of NFs. However, *SFC Path Tracer* sends probe packets directly through NFs and is hence error-prone if probe packets are dropped or packet tags are modified. In comparison *Track* avoids sending probe packets to NFs by inferring them.

Identifying the behaviours of NFs is a great challenge when tracing the paths in a network with NFs. Some proposals aim at solving this problem. SIMPLE [7] runs a similarity based correlation algorithm to identify how the incoming and outgoing flows at the middleboxex are correlated. Flowtags [21] adds minimal extensions to middleboxes to export the relevant contextual information, in the form of Tags embedded inside packet headers. Tracebox [22] sends IP packets containing TCP segments with different TTL values and analyses the packet encapsulated in the returned ICMP messages in order to detect any modification performed by middleboxes. In comparison, in order not to send probing packets to NFs, we use a correlation procedure to figure out the identification of modification of NFs which is similar to SIMPLE.

V. Conclusion

In this paper, we present *Track*, a powerful tool for tracing the path of any service chains in SDN-enabled networks with network functions. The key feature of the tool is that it can infer the behaviors of network functions upon packets without requiring prior knowledge on the types of network functions under test. With this feature, Track can correctly trace routes without polluting the internal states of NFs. We envision *Track* as an integral part of any network administrators toolkit for managing and troubleshooting the network.

References

[1] K. Agarwal, E. Rozner, C. Dixon, and J. Carter, "SDN traceroute: Tracing SDN forwarding without changing network behavior," in *Proceedings of HotSDN*. ACM, 2014, pp. 145–150.

[2] L. Cui, F. P. Tso, D. P. Pezaros, W. Jia, and W. Zhao, "PLAN: Joint policy-and network-aware VM management for cloud data centers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 4, pp. 1163–1175, 2017.

[3] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar, "Making middleboxes someone else's problem: Network processing as a cloud service," *SIGCOMM Comput. Commun. Rev.*, vol. 42, no. 4, pp. 13–24, Aug. 2012.

[4] L. Cui, R. Cziva, F. P. Tso, and D. P. Pezaros, "Synergistic policy and virtual machine consolidation in cloud data centers," in *35th Annual IEEE International Conference on Computer Communications (INFOCOM)*. IEEE, 2016, pp. 1–9.

[5] R. A. Eichelberger, T. Ferreto, S. Tandel, and P. A. P. R. Duarte, "SFC Path Tracer: A troubleshooting tool for service function chaining," in *International Symposium on Integrated Network Management*. IEEE/IFIP, 2017, pp. 568–571.

[6] "Ryu." [Online]. Available: https://osrg.github.io/ryu/

[7] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu, "SIMPLE-fying middlebox policy enforcement using SDN," *ACM SIGCOMM computer communication review*, vol. 43, no. 4, pp. 27–38, 2013.

[8] H. Pucha, D. G. Andersen, and M. Kaminsky, "Exploiting similarity for multi-source downloads using file handprints." in *NSDI*, 2007, p. 2007.

[9] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, "The rabin–karp algorithm," *Introduction to Algorithms*, pp. 911–916, 2001.

[10] "Mininet." [Online]. Available: http://yuba.stanford.edu/foswiki/bin/view/OpenFlow/Mininet

[11] "Internet2." [Online]. Available: http://www.internet2.edu/

[12] V. Paxson, "Bro: a system for detecting network intruders in real-time," in *Conference on Usenix Security Symposium*, 1998, pp. 3–3.

[13] "Passive Real-time Asset Detection System." [Online]. Available: http://prads.projects.linpro.no

[14] "Squid." [Online]. Available: http://squid-cache.org.

[15] "iptables." [Online]. Available: http://netfilter.org/projects/iptables.

[16] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella, "Opennf: Enabling innovation in network function control," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, pp. 163–174, 2015.

[17] "iPerf." [Online]. Available: https://iperf.fr/

[18] "Traceroute." [Online]. Available: http://traceroute.sourceforge.net/

[19] S.-Y. Wang, C.-C. Wu, and C.-L. Chou, "Hybridtrace: A traceroute tool for hybrid networks composed of SDN and legacy switches," in *IEEE Symposium on Computers and Communication (ISCC)*. IEEE, 2016, pp. 403–408.

[20] Y. Zhao, P. Zhang, and Y. Jin, "Netography: Troubleshoot your network with packet behavior in SDN," in *Network Operations and Management Symposium (NOMS), 2016 IEEE/IFIP*. IEEE, 2016, pp. 878–882.

[21] S. K. Fayazbakhsh, L. Chiang, V. Sekar, M. Yu, and J. C. Mogul, "Enforcing network-wide policies in the presence of dynamic middlebox actions using flowtags," in *NSDI*, 2014, pp. 533–546.

[22] G. Detal, B. Hesmans, O. Bonaventure, Y. Vanaubel, and B. Donnet, "Revealing middlebox interference with tracebox," in *IMC*. ACM, 2013, pp. 1–8.