Preventive Model-based Verification and Repairing for SDN Requests*

Igor Burdonov[1], Alexandre Kossachev[1], Nina Yevtushenko[1,2] Jorge López[3,4], Natalia Kushik[3] and Djamal Zeghlache[3]

[1]*Ivannikov Institute for System Programming of the Russian Academy of Sciences, Moscow, Russia*

[2]*National Research University Higher School of Economics, Moscow, Russia*

[3]*Télécom SudParis, Institut Polytechnique de Paris, Palaiseau, France*

[4]*Airbus Defence and Space, Issy-Les-Moulineaux, France*

Keywords: Software Defined Networking, Systems Engineering, Verification, Repairing, Graph Paths.

Abstract: Software Defined Networking (SDN) devices (e.g., switches) route traffic according to the configured *flow rules*, and thus a set of virtual paths gets implemented in the data plane. We propose a novel preventive approach for verifying that no misconfigurations (e.g., infinite loops), can occur given the requested set of paths. Such verification is essential since when configuring a set of data paths, other *not requested and undesired* paths (including loops) may be unintentionally configured. We show that for some cases the requested set of paths cannot be implemented without adding such undesired behavior, i.e., only a superset of the requested set can be implemented. We present a verification technique for detecting such issues of potential misconfigurations and estimate the complexity of the proposed method. Finally, we propose a technique for debugging and repairing a set of paths in such a way that the *corrected* set does not induce undesired paths into the data plane, if the latter is possible.

1 INTRODUCTION

Traditional networks have currently evolved largely due to the incorporation of software engineering approaches. One of the technologies that contributes to this evolution is the Software Defined Networking (SDN) paradigm, that allows implementing various *data paths* employing common resources. When using SDN technology, the network entities are managed through the controller that works independently of the network equipment and is 'responsible' for installing the necessary rules to the forwarding devices (e.g., switches) (Sezer et al., 2013). As a result, SDN provides agile controllability and observability by separating the control and data planes. SDN is a new technology entering novel domains (for example, the IoT domain) (Mohammed et al., 2020) with novel applications actively being developed. To guarantee the requested network is configured correctly, SDN components and compositions need to be thoroughly tested and verified. For example, one can be interested in verifying the absence of loops and packet loss, or the security and access control issues. Such data plane verification has been largely investigated, especially in the past decade.

Related Work. Recent works devoted to verification and testing of an SDN data plane and related data paths can be intuitively split into several groups. The first group focuses on the application of formal verification and model checking approaches to data plane verification or forwarding devices in isolation; in this case, classical networks (not necessarily SDN) with related access control, security and other network properties are considered. Correspondingly, these techniques mostly differ in the underlying formalism utilized for describing the specified behavior and related properties. Boolean functions and their satisfiability, symbolic model checking / execution and SMT solving (Mai et al., 2011), (Canini et al., 2012), (Dobrescu and Argyraki, 2013) as well as algebra of sets (Boufkhad et al., 2016) have been considered for checking for example, reachability issues, absence of loops, etc. The problem can also be solved via corresponding static analysis when the network device is implemented in the programming language (for example, P4) (Stoenescu et al., 2018).

Approaches of the second group tend to focus on

*This work was partly supported by the Ministry of Science and Higher Education of Russian Federation (grant number 075-15-2020-788).

Burdonov, I., Kossachev, A., Yevtushenko, N., López, J., Kushik, N. and Zeghlache, D.
Preventive Model-based Verification and Repairing for SDN Requests.
DOI: 10.5220/0010494504210428
In *Proceedings of the 16th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2021)*, pages 421-428
ISBN: 978-989-758-508-1

active testing of a data plane via corresponding traffic generation and monitoring of the forwarding behavior of switches of interest (Zeng et al., 2012), (Fayaz et al., 2016), (David et al., 2014). In automatic traffic generation, the packets / flows to be sent through the switches are generated at hosts in an active mode such that specific network failures can be captured when monitoring the data plane.

The third group of techniques relies on model based testing. Existing model based testing techniques either consider a given SDN component, such as for example an SDN enabled switch (Yao et al., 2014) or an SDN framework as a whole can be tested (Berriri et al., 2018), (Yevtushenko et al., 2018).

Contribution. The existing approaches often rely on a current network configuration, i.e., the rules have been already pushed to the switches while in this paper, we propose a novel preventive model-based approach for verifying programmable network properties. Indeed, given the set of paths to be implemented on the data plane for connecting appropriate hosts, if this set is not consistent or can lead to potential loops then its implementation should be avoided (the software should not be reconfigured as per the request). Let P be a set of paths which should be implemented on the data plane for packets of a given traffic type. The set P should be 'inspected' before its actual implementation, checking whether it is possible to precisely implement the set P on the data plane or there will be additional (unintended) paths implemented? In the latter case, it can happen that there are implemented paths which are not edge simple and thus, a loop for packets of a given traffic type can occur. In this paper, we show that given a traffic type which is defined by the packet headers (packets with the same traffic type follow the same data paths) and a set of (requested) paths P, the implementation of P can induce new paths appearing on the data plane, and moreover, if all the paths of P are edge simple (no loops should occur) it does not guarantee the absence of potential cycles on the data plane. We establish the criterion for the absence of those that relies on the property of the set P to be arc closed (Section 3). This criterion and the preventive verification method on its basis form the main contributions of the paper. Experimental results built over well-known SDN components (the Onos controller and Open vSwitches) confirm the necessity of such preventive verification; otherwise, in the implemented network, the packets generated at a certain host can go into infinite loops, i.e., can simply flood the network. These results can be used when developing any SDN application or when engineering SDN controllers, in order to implement safe SDN controllers. Another contribution of the paper is a technique for an automatic debugging and repairing of a set P of paths that did not pass the verification in such a way, that the resulting set of paths becomes arc closed (and thus safe to implement). For both, verification and debugging / repairing approaches, their related complexity is discussed.

This paper is a short version and thus, it omits several fundamental and experimental details, the interested reader can refer to of (Burdonov et al., 2019).

The structure of the paper is as follows. Section 2 presents the necessary background. Section 3 discusses the possibility of inducing undesired paths on the data plane that can cause, for example, infinite cycles. The proposed preventive verification approach for the set of paths P, together with the criterion for the absence of undesired links and the related complexity analysis is presented in Section 4. Automatic debugging and repairing of the set of paths for which the verification failed, is proposed in Section 5. Section 6 concludes the paper.

2 PRELIMINARIES

SDN is a networking paradigm that consists in separating the control and data plane layers (OpenNetworkingFoundation, 2012); this paradigm relies in software components: controller, forwarding devices, and applications. With a centralized SDN controller, SDN applications can reconfigure the SDN data plane (the forwarding devices). SDN-enabled forwarding devices steer the incoming network packets based on so-called flow rules installed (through the controller) by the SDN applications. A flow rule consists of three main (functional) parts: a packet matching part, an action part and a location / priority part. The matching part describes the values which a received network packet should have for a given rule to be applied. The action part states the required operations to perform to the matched network packets, while the location / priority part controls the hierarchy of the rules using tables and priorities. We focus on the resulting *implemented* data paths (produced by the rules installed at the forwarding devices); more precisely, we focus on the analysis of such data paths and the potentially unintended additional data paths resulting from a configuration.

The SDN *resource* topology (data plane) or *resource network connectivity topology (RNCT)* is represented as an undirected graph $G = (V,E)$ where $E \subseteq \{\{a,b\} | a \in V \ \& \ b \in V\}$ without multiple edges and loops. The set $V = H \cup S$, $H \cap S = \emptyset$, of nodes represents network devices such as *hosts* (the set H) and *switches* (the set S). Edges of the graph (the set

E) represent connections (links) between two nodes in G and each link can transmit packets in both directions. Correspondingly, we write (a,b) if a packet is transmitted from a to b and (b,a) when it is transmitted from b to a. We reasonably assume that each host is connected exactly with one switch and that G is connected; otherwise, each (connected) component can be treated as a separate network.

Data paths[1] are sets of paths which carry packets, i.e., those paths can have appropriate parameters according to which the packets are then forwarded; in other words, each packet belongs to an appropriate *traffic type*. When a forwarding rule is installed on an SDN-enabled switch, a data link from and to other node (-s) adjacent to the switch is created, i.e., a packet accepted from adjacent nodes is forwarded to a (corresponding) set of ports that are connected to appropriate ports of other nodes.

A host can generate packets that are forwarded to a single switch connected with this host. A switch can only forward packets; moreover, we assume that a switch does not modify the packet header, i.e., the packet's traffic type and payload are not changed through the network. A switch can forward a packet to several ports, and the set of ports depends on the traffic type as well as on the input port from which it arrives. Every node a of the graph G (a host or a switch) has a set of *ports* which can be input as well as output and each such port corresponds to some edge at the node a and vice versa, each edge at the node a is associated with a corresponding port. Thus, there is one-to-one correspondence between edges at the node a and the set of its ports, i.e., there is one-to-one correspondence between the set of ports of a and the set of neighbor nodes of a, since G has no multiple edges nor node (self) loops. Therefore, without loss of generality, we can use a neighbor node instead of the port number.

A path is a sequence of neighboring nodes of G, i.e., a path is a sequence[2] of nodes such that there is an edge between neighboring sequence nodes. A path $\pi = x_1 \cdot \ldots \cdot x_n$ starts at the node x_1, is finished at the node x_n, has length $n-1$, and passes via an arc (x_i, x_{i+1}) for $i \in \{1,\ldots,n-1\}$. The path is *edge simple* if it passes via each arc at most one time; the path is *node simple* if all its nodes are pairwise different. A path is *complete* if its head and tail nodes are hosts and there are no hosts as intermediate nodes.

An SDN application configures sets of paths (through the controller) which should transport corresponding packets, i.e., those paths can have appropriate parameters (which define their traffic type) according to which the packets are then forwarded (OpenNetworkingFoundation, 2015). The flow rules of a switch can be written as a mapping of input ports into subsets of output ports. If the subset of output ports is empty then the switch will 'drop' a packet that arrived at a corresponding input port.

We assume that an SDN application configures the switch tables in such a way that each rule determines the set of output ports depending on the traffic type and an input port. As G has no multiple edges it implies that a rule determines the set of neighboring nodes where a packet has to be forwarded. We also assume that all the switches have in their tables only the information sent by the controller, i.e., no default rules or external interfaces are considered. For the sake of simplicity and in fact, without loss of generality for our purpose, we assume that all the rules have the same priority. For packets belonging to the same traffic type, we can consider every rule as a triple $(a, s, b) \in V \times S \times V$ where a and b are neighbors of s. This rule says that getting a packet with the corresponding traffic type from neighbor a, switch s should send it to the neighbor b. If there are several rules which differ only in the neighbor b, then switch s performs *cloning*, i.e., the incoming packet is transmitted to several neighbors. The set of rules of all switches is called *configuration* (for the given traffic type).

3 IMPLEMENTING THE GIVEN SET OF COMPLETE PATHS

The set of complete paths that should be implemented on the data plane is based on a user request or predefined configuration (by a given application). Correspondingly, before setting a switch configuration according to a set of paths, it would be useful to verify whether a given set of paths can be eventually implemented. Note that hereafter we assume that the requested set of paths P does not contradict the RNCT G. A trivial check that P forms a sub-graph of G can be performed beforehand, if necessary.

When implementing a set of paths P, three options are possible. 1) P can be implemented as it is and in this case, the edge simplicity should be verified for the set P. 2) P cannot be implemented without implementing unintended paths, i.e., a superset of P is implemented. In this case, the condition of the edge simplicity should be checked for this superset. If the minimal superset of P that can exist on the data plane has cycling paths, then the set P cannot be implemented (packet loops may flood the network) in the given data plane. 3) P cannot be implemented but the minimal superset of P that can be implemented satis-

[1] They are directed, differently from the topology itself.

[2] We use '$\cdot$' for denoting the sequence concatenation.

fies the edge simplicity property. We further discuss how given a set P of paths, a corresponding switch configuration is specified and given a switch configuration, which paths are induced by this configuration.

Complete paths induce switch rules. When implementing rules for a complete path (for the given traffic type) $\alpha \cdot a \cdot b \cdot c \cdot \beta$ where $a, b, c \in V, \alpha, \beta \in V^*$, we need a rule (a, b, c), i.e., a switch b once getting a packet belonging to this traffic type from the neighbor a has to send it to the neighbor c. Formally, the set P of paths induces the set $P\downarrow$ of rules:

$\forall a \in V, b \in S, c \in V, \alpha \in V^*, \beta \in V^*$

$\alpha \cdot a \cdot b \cdot c \cdot \beta \in P$ implies that there is a rule $(a, b, c) \in P\downarrow$.

Switch rules induce paths. The rule (a, b, c) induces a path $a \cdot b \cdot c$ of length 2. If there is a path $\alpha \cdot x \cdot y$ and there is a rule (x, y, z) then there is a path $\alpha \cdot x \cdot y \cdot z$. Formally, a switch configuration $P\downarrow$ induces the set of complete paths, written $P\downarrow\uparrow$:

$\forall b_j \in V$

$(a_1, b_1, b_2), (b_1, b_2, b_3), \ldots, (b_{n-1}, b_n, a_2) \in P\downarrow$ where a_1 and a_2 are the only hosts, there is a path $a_1 \cdot b_1 \cdot b_2 \ldots \cdot b_{n-1} \cdot b_n \cdot a_2$ in $P\downarrow\uparrow$.

By definition, the set $P\downarrow\uparrow$ has only complete paths and the following statement holds.

Proposition 1. *Given a switch b, for each rule $(a, b, c) \in P\downarrow$ of this switch, there is a path $\alpha \cdot a \cdot b \cdot c \cdot \beta \in P\downarrow\uparrow$ for some α and β.*

We now establish the conditions when two paths $\alpha \cdot x \cdot y \cdot \beta$ and $\alpha' \cdot x \cdot y \cdot \beta'$ in the set $P\downarrow\uparrow$ induce another two paths in this set.

Proposition 2. *Given a switch configuration $P\downarrow$, $P\downarrow$ induces the set of complete paths $P\downarrow\uparrow$ with the following features:*

$\forall \alpha, \alpha', \beta, \beta' \in V^*$

$\alpha \cdot x \cdot y \cdot \beta \in P\downarrow\uparrow \ \& \ \alpha' \cdot x \cdot y \cdot \beta' \in P\downarrow\uparrow \implies \alpha \cdot x \cdot y \cdot \beta' \in P\downarrow\uparrow$.

According to Proposition 2, the set of data paths on the data plane induced by the given set P is exactly $P\downarrow\uparrow$, and in fact, it is the actual set of paths that gets implemented when requesting to implement the set P.

The set P of complete paths is *closed with respect to a given arc* (x, y) if for each two paths $\alpha \cdot x \cdot y \cdot \beta$ and $\alpha' \cdot x \cdot y \cdot \beta'$ of the set P which have a common arc (x, y), paths $\alpha \cdot x \cdot y \cdot \beta'$ and $\alpha' \cdot x \cdot y \cdot \beta$ are also in P. The set P of paths is *arc closed* if P is closed w.r.t. each arc over the set E. Given a set P of complete paths, the arc closure of P is the smallest arc closed set of complete paths that contains P. According to the definition of an arc closed set and Proposition 2, the following statement can be established.

Proposition 3. *Given a set P of complete paths, the set $P\downarrow\uparrow$ is the arc closure of P.*

Corollary 1. *The set $P\downarrow\uparrow$ coincides with P if and only if P is arc closed.*

The above corollary establishes necessary and sufficient conditions for the precise implementation of set P on the data plane (without additional 'undesired' paths).

Corollary 2. *If P has only edge simple paths and is arc closed then $P\downarrow\uparrow$ has only edge simple paths.*

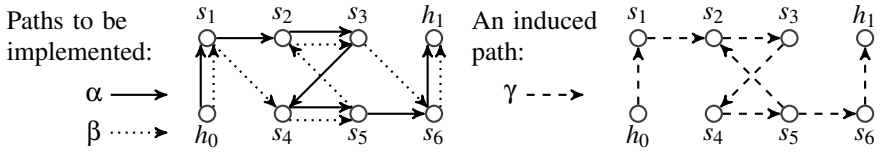
If P is not arc closed then P cannot be implemented on the data plane up to the equality relation. Moreover, sometimes P cannot be implemented on the data plane at all as its arc closure has some cycling paths. Figure 1 shows an example when the set P has two edge simple paths α and β from initial host h_0 to the final host h_1 (left of the figure), the set of rules induced by this set is shown at the bottom and an induced path γ of the set $P\downarrow\uparrow$ is illustrated at the right. The path is not edge simple, and this example illustrates that cycles can occur even when paths of the set P are edge simple.

Similar to P, all the paths of the set $P\downarrow\uparrow$ are complete paths. However, if $P\downarrow\uparrow$ is a proper superset of P then we have to check whether all the paths of the set $P\downarrow\uparrow$ are edge simple. If it is the case then the set P can be implemented on the data plane up to the set $P\downarrow\uparrow$ (i.e., with additional unspecified paths from $P\downarrow\uparrow \setminus P$). If it is not the case then the set P should be modified and this issue is discussed in Section 5.

From a practical point of view, perhaps the most interesting application is when some set $P\downarrow\uparrow$ of paths is already implemented on the data plane and a new request arrives; either a request A to add new paths $(P \cup A)$ or a request R to remove paths $(P \setminus R)$ to / from the original set. In this case, the same check should be performed on $((P \cup A) \setminus R)\downarrow\uparrow$ before implementing / removing paths, guaranteeing the implementability of the augmented set of paths. Algorithm 1 summarizes the necessary verification steps (Section 4) and returns the corresponding verdict about the implementability of a given set of paths.

Practical / Experimental Motivation. In order to verify if our (fundamental) findings can occur in real SDN framework implementations, an experimental evaluation was performed. Experiments were carried in a virtual machine running GNU/Linux CentOS 7.6 with 8 vCPUs and 16GB of RAM. The Onos (Berde et al., 2014) SDN controller (version 4.2.8) was installed via a Docker (Merkel, 2014) container. To emulate the SDN data plane, the Containernet (Peuster et al., 2018) was also installed through a Docker container.

The paths shown in Figure 1 were configured independently and successful communication from h_0



$$P=\{\alpha=h_0\cdot s_1\cdot s_2\cdot s_3\cdot s_4\cdot s_5\cdot s_6\cdot h_1,\beta=h_0\cdot s_1\cdot s_4\cdot s_5\cdot s_2\cdot s_3\cdot s_6\cdot h_1\}$$

$$P{\downarrow}=\{(h_0,s_1,s_2),(s_1,s_2,s_3),(s_2,s_3,s_4),(s_3,s_4,s_5),(s_4,s_5,s_6),(s_5,s_6,h_1),$$
$$(h_0,s_1,s_4),(s_1,s_4,s_5),(s_4,s_5,s_2),(s_5,s_2,s_3),(s_2,s_3,s_6),(s_3,s_6,h_1)\}$$

$$\gamma=h_0\cdot s_1\cdot s_2\cdot s_3\cdot s_4\cdot s_5\cdot s_2\cdot s_3\cdot s_4\cdot s_5\cdot\ldots\cdot s_2\cdot s_3\cdot s_4\cdot s_5\cdot s_6\cdot h_1\in P{\downarrow}{\uparrow}$$

Figure 1: Induced (cyclic) paths' occurrence.

to h_1 was discovered, i.e., there was no problem while configuring both paths independently. When both paths were configured simultaneously, the loop was effectively produced. A single packet sent from h_0 to h_1 produced infinitely many of them. In Figure 2, we show the packet dump (using the well-known utility tcpdump) as seen by h_1. Note that, the packet sent is an ICMP echo request (using the ping utility), and the sequence ID is always 1, as the single packet gets copied infinitely many times. When continuously sending the packets the network rapidly degraded until the whole infrastructure became unusable. These experiments confirm the importance of our findings. Indeed, it is important to provide SDN frameworks with verification tools before rules are pushed to the switches. One of the procedures for such verification is given in Algorithm 1.

4 CHECKING THE ARC CLOSURE

In this section, we propose an algorithm for checking if a given set of paths P induces unintended paths, i.e., a superset of P is implemented (when P is intended); likewise, we discuss how to detect potential cycles induced by the implementation of P. Algorithm 1 shows the verification steps necessary to check the arc closure of a given set of paths. Note that the algorithm always terminates due to the finite calculations in nested loops, independently if $P{\downarrow}{\uparrow}$ contains a path with a loop or not.

Proposition 4. *Algorithm 1 returns the verdict True if and only if P is arc closed.*

Consider the example in Figure 1; the graph $D(P)$ is shown in Figure 3. By direct inspection one can assure that there is a cycle $(s_2,s_3),(s_3,s_4),(s_4,s_5),(s_5,s_2)$ in the graph and thus, the number of paths from the vertex *source* to the vertex *sink* is infinite, and therefore, the set P is not arc closed.

Algorithm 1: Verifying if the set of paths P is arc closed.

Input : A set P of edge simple complete paths
Output: A verdict whether the set P is arc closed
Derive a subset $Q=\{q_1,\ldots,q_k\}$ of P that contains all the paths of length greater than two; we denote as k_j the length of a path q_j, $j\in\{1,\ldots,k\}$;
Derive a graph $D(P)=<D,E>$ for the set Q where the vertices of $D(P)$ are pairs of vertices of the paths in Q;
$D=\{source,sink\}$; $E=\emptyset$;
$j=0$;
while $j<k$ **do**
  $j++$; $D=D\cup\{(q_j(1),q_j(2)),(q_j(k_j),q_j(k_j+1))\}$;
  $E=E\cup\{(source,(q_j(1),q_j(2)),(q_j(k_j),q_j(k_j+1)),sink)\}$;
  $m=2$;
  while $m<k_j+1$ **do**
    $D=D\cup\{(q_j(m),q_j(m+1))\}$;
    $E=E\cup\{((q_j(m-1),q_j(m)),(q_j(m),q_j(m+1)))\}$;
    $m++$;
if *the number of paths in* $D(P)$ *from source to sink is greater than* k **then**
  return *False*;
return *True*;

Proposition 5. *The complexity of checking the absence of cycles for a given set of paths P is* $O(L+|V|^3)$ *where* $|V|$ *is the number of nodes in G and L is the sum of the lengths of the paths in P.*

The complexity of constructing the graph $D(P)$ is $O(L)$ where L is the sum of the lengths of the paths from P. In order to check for (infinite) loops, the absence of oriented cycles in the graph $D(P)$ needs to be checked, which is done through a topological sort (e.g., using depth first search (DFS) (Cormen et al., 2009)). DFS-algorithm can also be used for computing the number of paths from the *source* to the *sink*

```
19:13:47.938895 IP 10.0.0.1 > 10.0.0.3: ICMP echo request, id 87, seq 1, length 64
19:13:47.939142 IP 10.0.0.3 > 10.0.0.1: ICMP echo reply, id 87, seq 1, length 64
19:13:48.138850 IP 10.0.0.1 > 10.0.0.3: ICMP echo request, id 87, seq 1, length 64
19:13:48.138903 IP 10.0.0.3 > 10.0.0.1: ICMP echo reply, id 87, seq 1, length 64
19:13:48.138954 IP 10.0.0.1 > 10.0.0.3: ICMP echo request, id 87, seq 1, length 64
19:13:48.139029 IP 10.0.0.3 > 10.0.0.1: ICMP echo reply, id 87, seq 1, length 64
19:13:48.338943 IP 10.0.0.1 > 10.0.0.3: ICMP echo request, id 87, seq 1, length 64
19:13:48.338988 IP 10.0.0.1 > 10.0.0.3: ICMP echo request, id 87, seq 1, length 64
19:13:48.339010 IP 10.0.0.3 > 10.0.0.1: ICMP echo reply, id 87, seq 1, length 64
19:13:48.339012 IP 10.0.0.3 > 10.0.0.1: ICMP echo reply, id 87, seq 1, length 64
```

Figure 2: Packet capture showing an infinite loop in the experimental infrastructure.

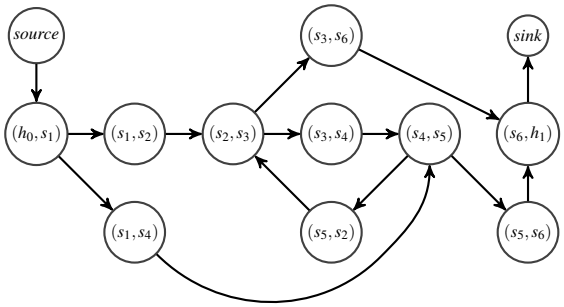


Figure 3: Graph $D(P)$ for verifying the set of paths P.

node when there are no cycles. The running time of the depth first search algorithm on the graph $D(P)$ is evaluated as $O(m)$, where m is the number of arcs of the graph $D(P)$, $m \leq |V|^3$.

5 DEBUGGING AND REPAIRING A SET OF PATHS

We discuss some possibilities of correcting / modifying the set of paths P whenever this set is not arc closed. One first needs to identify the reason, i.e., a subset of paths that destroy the corresponding property, and the set of paths P should be either augmented with new paths or on the contrary, certain paths should be deleted from the set P. In both ways, the resulting subset becomes arc closed and thus, can be implemented on the data plane without any additional links. We refer to this process as automatic *P debugging* and *repairing*. Such repairing process can have various objectives, such as for example: minimization of the number of paths to be excluded / included from / to P, maximization of a host to host connectivity in the resulting set of paths, minimization of the number of changes in the paths of the set, minimization of virtual links on the data plane, etc. We furthermore discuss some of the possibilities listed above and propose various debugging and repairing strategies.

For the following subsections, we use the following notations. Given a set P of complete paths, let $P = \{p_1, \ldots, p_k\}$, i.e., $k = |P|$, and $k_i = |p_i| - 1$, i.e., k_i is the length of p_i for all $i \in \{1, \ldots, k\}$.

5.1 Minimizing the Set of Paths to be Excluded / Included from / to P

The problems we address in this subsection are the following: how to delete / add a minimal number of paths from / to the set P, such that the resulting subset / superset becomes arc closed.

We say that two different paths p_i and p_j of P are *incompatible* if there exists a common arc, i.e., there exist $u \in \{1, \ldots, k_i - 1\}$ and $v \in \{1, \ldots, k_j - 1\}$ such that $p_i(u) = p_j(v)$ & $p_i(u+1) = p_j(v+1)$ while a path $p_i(1) \cdot \ldots \cdot p_i(u) \cdot p_j(v+1) \cdot \ldots \cdot p_j(k_j+1)$ or a path $p_j(1) \cdot \ldots \cdot p_j(v) \cdot p_i(u+1) \cdot \ldots \cdot p_i(k_i+1)$ is not in P. In this case, one can also say that p_i and p_j are incompatible w.r.t. the common arc $(a,b) = (p_i(u), p_i(u+1))$. If p_i and p_j of P are not incompatible, then they are *compatible*.

The problem of deleting a minimal number of paths can be reduced to the well known maximum independent set problem. For that matter, we propose to derive an undirected graph $G(P)$ in the following way: the nodes of the graph correspond to the paths of the set P. There is an arc between p_i and p_j, $i \neq j$, in the graph $G(P)$ if the paths p_i and p_j are incompatible. Given an undirected graph $G(P)$, a subset of nodes which are not pairwise connected is an independent subset of nodes.

Proposition 6. *An independent subset of nodes of graph $G(P)$ is an arc closed set.*

Corollary 3. *A subset of P is arc closed if and only if it is an independent subset of the graph $G(P)$.*

Therefore, the problem of minimizing the set of paths to be excluded from P is reduced to the derivation of a maximal independent subset of nodes in $G(P)$. Note that this problem is known to be NP-hard, and thus the repairing approach can be more complex than that one presented for the verification itself (Section 4).

As an example, consider again the paths of the set P in Figure 1. Note that the paths from P possess the necessary feature, i.e., they have a common arc (s_2, s_3) with the above property and the set P has no path $h_0 \cdot s_1 \cdot s_2 \cdot s_3 \cdot s_4 \cdot s_5 \cdot s_2 \cdot s_3 \cdot s_4 \cdot s_5 \cdot s_6 \cdot h_1$. Therefore, the corresponding vertices in $G(P)$ are connected, i.e., P

is not arc closed and only the singletons $\{\alpha\}$ or $\{\beta\}$ are arc closed.

For deriving a minimal superset of P that is arc closed, the graph $D(P)$ derived in the previous section can be used. If the graph returned by Algorithm 1 has no cycles then the set of all paths from the *source* node to the *sink* node is the smallest superset of P that is arc closed.

Proposition 7. *1. If all the paths from the source node to the sink node in $D(P)$ are edge simple then the set of all paths is the smallest superset of P that is arc closed. 2. If there is a path from the source node to the sink node in $D(P)$ that is not edge simple then there is no finite superset of P that is arc closed.*

Note that in case 2, it is not possible to add paths to the given set P; the set P can be only reduced as it is discussed at the beginning of the subsection. Indeed, it is exactly the case for the set P in Figure 1.

5.2 Minimizing the Number of Arc Changes in the Set P

Consider a set P of edge simple complete paths that is not arc closed, the question arises: can the paths of the set be minimally corrected (w.r.t. the number of arcs) in order to get an arc closed set preserving the head and tail hosts of each path? We propose a simple way for modifying a single edge or a sub-path of a path using edges of the RNCT graph G which were not utilized in the paths of P (the set N in Algorithm 2).

Proposition 8. *Given a set P of edge-simple complete paths, if Algorithm 2 returns a set $Q = \{p_1, p'_2, \ldots, p'_k\}$ then this set is arc closed and for each $j \in \{1, \ldots, k\}$, the head and tail vertices of p'_j coincide with those of p_j.*

Note that the set of repaired paths returned by Algorithm 2 has only edge simple paths and is arc closed, since every time only unused links are utilized for the replacement. In this paper, we consider only simple heuristics for repairing a path and the result significantly depends on the order of the paths in P. Additional research is needed to propose more rigorous conditions for repairing a set of initial paths. Those conditions can be related to the link load distribution and thus, could re-direct some packets, for example, for traffic optimization.

As an example, consider again the paths in Figure 1, assuming that each pair of switches is connected in the RNCT G. These paths have a common arc (s_4, s_5) that can be replaced by a path $s_4 \cdot s_6 \cdot s_5$. After this modification the paths have a common arc (s_2, s_3) that can be replaced by a path $s_2 \cdot s_4 \cdot s_3$. Thus, we obtain an arc closed set of paths $P' = \{h_0 \cdot s_1 \cdot s_2 \cdot s_3 \cdot s_4 \cdot s_5 \cdot s_6 \cdot h_1, h_0 \cdot s_1 \cdot s_4 \cdot s_6 \cdot s_5 \cdot s_2 \cdot s_4 \cdot s_3 \cdot s_6 \cdot h_1\}$.

Algorithm 2: Repairing via modifying an edge or a sub-path preserving the head and tail hosts of the path.

Input : A set P of edge-simple complete paths that is not arc closed, a non-empty set N of edges between switches of the RNCT graph G which are not used in the paths of the set P

Output: A verdict *False* if paths cannot be modified, or a modified arc closed set P where the head and tail vertices of each modified path p'_j coincide with those of the initial path p_j of P

$Q = \{p_1\}$;
$j = 2$;
while $j \leq |P|$ **do**
  $p'_j = p_j$;
  $l = 1$;
  while $l \leq |Q|$ **do**
    $p = q_l$;
    if *paths p and p'_j are incompatible w.r.t. P* **then**
      if $N = \emptyset$ **then**
        return *False*;
      else
        while *the paths p and p'_j are incompatible w.r.t. the common arc (s_1, s_2)* **do**
          if *the paths p and p'_j have a common sub-path $s_3 \cdot \alpha \cdot s_1 \cdot s_2 \cdot \beta \cdot s_4$ and (s_3, s_4) is in N* **then**
            Derive p'_j by replacing a sub-path $s_3 \cdot \alpha \cdot s_1 \cdot s_2 \cdot \beta \cdot s_4$ in p'_j by a sub-path $s_3 \cdot s_4$;
            Delete (s_3, s_4) from the set N;
          else if *there is a switch s_3 such that $(s_1, s_3), (s_3, s_2) \in N$* **then**
            Derive p'_j by replacing a sub-path $s_1 \cdot s_2$ in p by a sub-path $s_1 \cdot s_3 \cdot s_2$;
            Delete (s_1, s_3) and (s_3, s_2) from the set N;
          else
            return *False*;
    $l{+}{+}$;
  Add p'_j to the set Q;
  $j{+}{+}$;
return an arc closed set $Q = \{p_1, p'_2, \ldots, p'_k\}$

6 CONCLUSION

In this paper, we discussed implementability issues for a given set of paths on an SDN data plane. We showed that for a fixed traffic type, whenever the requested set contains only edge simple paths, more (unintended) paths can still be implemented on the data plane, and some of those can create cycles, i.e., infinite packet loops, and we established the necessary and sufficient conditions for a set of requested paths to be implemented without any undesired connections. Our preventive verification approach is useful for guaranteeing that new (requested) and preexisting paths form valid configurations. The estimated (polynomial w.r.t. the total paths' length) complexity of the proposed approach makes possible its applicability for large scale virtual networks. For a set of paths that cannot be implemented directly on the data plane, we proposed debugging and repairing approaches for correcting the initial request, such that the resulting set becomes arc closed. As future work, we plan to extend the proposed approaches abstracting from a given traffic type as well as considering other kinds of specifications for user requests.

REFERENCES

Berde, P., Gerola, M., Hart, J., Higuchi, Y., Kobayashi, M., Koide, T., Lantz, B., O'Connor, B., Radoslavov, P., Snow, W., et al. (2014). Onos: towards an open, distributed sdn os. In *Proceedings of the third workshop on Hot topics in software defined networking*, pages 1–6. ACM.

Berriri, A., López, J., Kushik, N., Yevtushenko, N., and Zeghlache, D. (2018). Towards model based testing for software defined networks. In *Proceedings of the 13th International Conference on Evaluation of Novel Approaches to Software Engineering ENASE*, pages 440–446.

Boufkhad, Y., De La Paz, R., Linguaglossa, L., Mathieu, F., Perino, D., and Viennot, L. (2016). Forwarding tables verification through representative header sets. *arXiv preprint arXiv:1601.07002*.

Burdonov, I. B., Kossachev, A., Yevtushenko, N., López, J., Kushik, N., and Zeghlache, D. (2019). Verifying SDN data path requests. *CoRR*, abs/1906.03101.

Canini, M., Venzano, D., Perešíni, P., Kostić, D., and Rexford, J. (2012). A NICE way to test openflow applications. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 127–140.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to algorithms*. MIT press.

David, L., Stefano, V., and Olivier, B. (2014). Towards test-driven software defined networking. In *2014 IEEE Network Operations and Management Symposium*, pages 1–9.

Dobrescu, M. and Argyraki, K. (2013). Toward a verifiable software dataplane. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, page 18. ACM.

Fayaz, S. K., Yu, T., Tobioka, Y., Chaki, S., and Sekar, V. (2016). BUZZ: Testing context-dependent policies in stateful networks. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 275–289.

Mai, H., Khurshid, A., Agarwal, R., Caesar, M., Godfrey, P., and King, S. T. (2011). Debugging the data plane with anteater. *ACM SIGCOMM Computer Communication Review*, 41(4):290–301.

Merkel, D. (2014). Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, 2014(239):2.

Mohammed, A. H., Khaleefah, R. M., k. Hussein, M., and Amjad Abdulateef, I. (2020). A review software defined networking for internet of things. In *2020 International Congress on Human-Computer Interaction, Optimization and Robotic Applications (HORA)*, pages 1–8.

OpenNetworkingFoundation (2012). Software-defined networking: The new norm for networks. *ONF White Paper*.

OpenNetworkingFoundation (2015). Openflow switch specification version 1.5. 0. *ONF Specification*.

Peuster, M., Kampmeyer, J., and Karl, H. (2018). Containernet 2.0: A rapid prototyping platform for hybrid service function chains. In *2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft)*, pages 335–337. IEEE.

Sezer, S., Scott-Hayward, S., Chouhan, P. K., Fraser, B., Lake, D., Finnegan, J., Viljoen, N., Miller, M., and Rao, N. (2013). Are we ready for sdn? implementation challenges for software-defined networks. *IEEE Communications Magazine*, 51(7):36–43.

Stoenescu, R., Dumitrescu, D., Popovici, M., Negreanu, L., and Raiciu, C. (2018). Debugging P4 programs with vera. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication SIGCOMM*, pages 518–532.

Yao, J., Wang, Z., Yin, X., Shiyz, X., and Wu, J. (2014). Formal modeling and systematic black-box testing of sdn data plane. In *The IEEE 22nd International Conference on Network Protocols (ICNP)*, pages 179–190.

Yevtushenko, N., Burdonov, I. B., Kossachev, A., López, J., Kushik, N., and Zeghlache, D. (2018). Test derivation for the software defined networking platforms: Novel fault models and test completeness. In *2018 IEEE East-West Design & Test Symposium EWDTS*, pages 1–6.

Zeng, H., Kazemian, P., Varghese, G., and McKeown, N. (2012). Automatic test packet generation. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, pages 241–252. ACM.