Fast Configuration Change Impact Analysis for Network Overlay Data Center Networks

Lizhao You, *Member, IEEE,* Jiahua Zhang, Yili Jin, Hao Tang, and Xiao Li

***Abstract*—This paper presents the first network configuration verifier that provides fast all-pair reachability analysis of incremental configuration changes for network overlay data center networks (DCNs). Network overlay DCNs leverage distributed routing protocol on edge leaf switches to disseminate overlay routes and establish overlay tunnels. In addition, network overlay DCNs use access control lists, microsegmentation policy, policy-based routing and firewall policy to control east-west and north-south traffic. Although some incremental verification approaches have been proposed, they either do not support certain forwarding features of the network, or are not efficient. Our configuration verifier addresses these issues through the following components: 1) a port predicate based forwarding model that is general to support all features; 2) fine-grained association technique to index possibly affected reachable pairs by changed interfaces in the original network; and 3) required waypoint path computation that finds all reachable pairs related to changed interfaces in the new network. Based on these components, our verifier presents two incremental verification algorithms that are specially designed for different service update cases. Experiment results show that our incremental verification algorithms are accurate and fast. For all-pair reachability, our verifier performs change-impact analysis within 15s for networks with 200 leafs (4000 subnets and 16 million pairs), outperforming existing approaches by up to 10x.**

***Index Terms*—network verification, configuration verification, incremental verification, data center networks**

I. Introduction

MANAGEMENT of Data Center Networks (DCNs) is difficult due to its support of multi-tenant cloud. The network overlay solution [1], [2] adopted by major device vendors makes the management more complicated. Different from the centralized software-defined network (SDN) solution that controller makes forwarding decisions, network overlay solution leverages distributed Border Gateway Protocol (BGP) on edge leaf switches to advertise overlay routes and establish virtual extensible local area network (VXLAN) tunnels [3]. In addition, various policies such as access control lists (ACLs), microsegmentation (MCS) and policy-based routing are enforced by switches inside the network to control east-west and north-south traffic.

In this paper, we consider the incremental configuration verification problem in network overlay DCNs. When new services are to be deployed, incremental configurations of network switches are generated, and are to be verified so that they do not violate intended reachability requirements. In particular, we focus on the all-pair reachability change impact analysis: at the beginning, all endpoint interfaces that generate overlay traffic have a mutual reachability relationship; after incremental configurations are generated (but before deployment), the verifier returns changed all-pair reachability caused by incremental configurations so that users can check whether the impact follows their intent.

Given that overlay services are frequently deployed, the computation overhead of change impact analysis should be low. In addition, the pre-deployment analysis should be as fast as possible compared with post-deployment offline verification. Re-computing all pairs' reachability [4] is not scalable given the potential large amount of endpoint interfaces in real networks. Therefore, it is more practical to find pairs whose reachability are affected, and re-compute these pairs only. Although some incremental verification approaches have been proposed, they have limitations for network overlay DCNs.

Existing EC-based incremental approaches are not general for network overlay DCNs. Equivalence class (EC) approaches [5]–[10] pre-compute the global forwarding equivalence classes that have the same forwarding behavior in all devices, and represent the edge of network graph model as EC labels. Then they can compute incremental ECs when facing configuration changes, and have demonstrated a fast incremental verification performance in IP or ACL forwarding scenarios. However, the forwarding features in commercial network overlay DCNs are quite complicate: there is a long-chained forwarding pipeline for overlay packets, and the pipeline goes beyond sequential. Single-modular abstraction for a device is not adequate. Moreover, some packet forwarding features are closely related to devices' local behavior (e.g., firewall security policy), and their forwarding constraints can not be represented by the traditional packet header. Even if the constraints can be represented by an extended EC model, extensive packet rewrites (e.g., tunnels) and distributed processing (e.g., microsegmentation) can make global EC computation inefficient. A general network model without computing ECs is preferable when handling complex forwarding features in commercial network overlay DCNs.

Existing indexing-based incremental approaches are not efficient for network overlay DCNs. Indexing approaches are an alternative way to compute differential reachability, and do not rely on the EC model. They first compute all-pair reachability in the forwarding graph model, and associate each node with reachability information such as arrived flows or pairs (i.e., establish reachability index) when initializing the all-pair reachability matrix. When nodes that change forward-

L. You is with School of Informatics, Xiamen University, Xiamen 361005, China (e-mail: lizhaoyou@xmu.edu.cn). J. Zhang, H. Tang and X. Li are with Huawei Technologies Co., Ltd, Shenzhen 518129, China (e-mail: {zhangjiahua4, tanghao15, lixiao91}@huawei.com). Y. Jin is with The Chinese University of Hong Kong (Shenzhen), Shenzhen 518172, China. This work was done when Yili was an intern at Huawei Technologies Co., Ltd.

ing behavior are identified, simply looking up the table can find possibly affected flows or pairs, and re-computation of these flows or pairs can find reachability changes. However, the existing approaches [11], [12] either use a large amount of memory due to the per-node flow storage, or invoke many inefficient computation due to inaccurate association and per-pair re-computation.

To overcome these limitations, this paper presents a new real-time configuration verifier, **P**ort **P**redicate **V**erifier (PPV). PPV adopts the modular design principle that decouples the forwarding pipeline into blocks, and builds a port (interface) forwarding graph model that connects basic blocks and is expressive for all kinds of forwarding behavior in network overlay DCNs. Moreover, PPV defines an extended symbolic packet header that includes some local fields to model device-specific forwarding features. These fields are meaningful only inside a device, and are cleaned up before leaving the device. Then PPV defines the forwarding constraints as predicate on the symbolic packet header, and performs reachability analysis on the raw predicate presentation instead of EC labels.

Furthermore, based on the port predicate model, our verifier first presents an improved depth-first search (DFS) algorithm to compute reachability of one endpoint to all endpoints in one round, avoiding expensive per-pair computation. Then PPV presents two novel incremental verification algorithms to compute differential reachability based on the DFS algorithm. The first algorithm leverages a reachability table that associates interfaces (more accurate than [12]) with reachable pairs in the old graph, and augments the indexing approach with a required waypoint path computation procedure to find all reachable pairs related to the changed interfaces in the new graph. In particular, the procedure treats these changed interfaces as required waypoints, calls the DFS algorithm in the forward direction and backward direction to find reachability paths, and combines them to find all reachable pairs. The first algorithm performs well for small-scale networks and for some service update cases. To further improve the performance and avoid storing reachability information, PPV presents a new algorithm that does not rely on the reachability table, and uses two rounds of required waypoint path computation (with smaller header space) to find reachability changes. The second algorithm performs better for some cases that changed interfaces are not at edge and for large-scale networks.

Overall, our contributions are summarized as follows:

1) To the best of our knowledge, we are the first to design and demonstrate incremental configuration verifier for network overlay DCNs.
2) We present a general port predicate model, and design two incremental verification algorithms that use a new indexing method and a required waypoint path computation method to find all-pair reachability changes.
3) We show that our verifier analyzes reachability changes within 15s for studied service update cases, outperforming existing approaches by 0.8x-10x.

The rest of this paper is organized as follows. Section II introduces the background and motivations. Section III overviews our designs and the architecture of our verifier. Section IV and Section V present the design details. Experimental results are given in Section VI. Section VII summarizes related works. Section VIII concludes this paper.

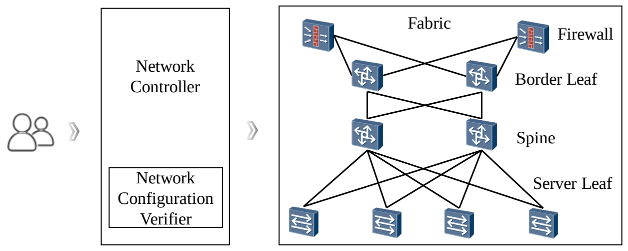


Fig. 1: A typical network overlay DCN system

II. Background and Motivations

A. Network Overlay DCN

System Overview: Figure 1 shows a typical network overlay DCN system, including a network controller and a DCN fabric. A network overlay DCN fabric is actually a data center (similar to an Availability Zone for the public cloud), and is made up with network switches and firewalls. It usually adopts the two-layer Spine-Leaf architecture: the physical servers and their hosting virtual machines (VMs) connect with top-of-rack (ToR) server leaf switches, and each server leaf connects with two spine switches. Network traffic are evenly distributed over the two links, and the traffic can be redirected to the other link if one link fails. A DCN fabric also owns two special leaf switches — border leafs. On one hand, border leafs serve as edge switches that connect with the external network (e.g., provider edge routers). On the other hand, firewalls are attached to the border leafs, and policies are configured on firewalls to filter east-west and north-south traffic. We focus on verifying the overlay service[1] in the DCN fabric, where all controls are enforced in the fabric devices (e.g., switches).

In this system, network administrators usually use a network controller (e.g., Cisco APIC [1] or Huawei Agile-Controller [2]) to manage the fabric. The network controller provides a portal to deploy services, where users can specify network parameters and network policy. It then converts the inputs to configurations applied to network devices. The generated configurations include protocol parameters and policy parameters, which are to be elaborated below. Note that these configurations are not the real forwarding plane as in SDN, and network switches still need to leverage distributed protocols to generate the forwarding plane.

VxLAN: VxLAN is the de facto overlay protocol to support network virtualization in DCN [3]. VxLAN uses a MAC-in-UDP packet encapsulation mode, where the original overlay Layer 2 frame is encapsulated into an outer UDP-IP header. There is also a VxLAN header that contains some overlay network information such as virtual network identifier (VNI) and group ID.

Virtual tunnel end points (VTEPs) are responsible for encapsulation and decapsulation of the outer header and the VxLAN header. Different from the host overlay DCNs, where VTEPs

[1] More accurately, we are verifying the implementation of virtual private clouds within data center networks, including both VxLAN tunnel forwarding (underlay) and the service traffic forwarding over the tunnels (overlay).

are created on virtual switches (vSwitch) in the server, the server leafs and border leafs hold the VTEP interfaces (called network virtual interface, NVE) in network overlay DCNs. To enable VxLAN tunnels, the underlay transport network needs to be configured first: IGP protocols (e.g., OSPF, BGP) are usually used to ensure mutual reachability of NVEs in ToR leafs and border leafs. BGP Ethernet Virtual Private Network (EVPN) [13] is usually used as the control plane to distribute overlay routes.

VRF: In overlay DCNs, virtual private cloud (VPC) is the basic resource for tenants. A tenant can own several VPCs, and can configure VPC peering to enable inter-VPC communication. A VPC is usually implemented using virtual routing and forwarding (VRF). A VRF can be treated as a virtual router that owns private subnets and a private forwarding information base (FIB). The VNI field in VxLAN header is used to isolate traffic belonging to different VPCs. Each VPC uses a unique VNI to label its traffic. When receiving a VxLAN packet, destination VTEP parses the VNI value, and delivers the packet to the corresponding VRF. VNIs are advertised in the overlay routes.

The VxLAN tunnel also allows inter-VPC traffic if source VTEP pads the destination VRF's VNI in the VxLAN packet. However, in practice, to further control the inter-VPC accessible subnets, the inter-VPC access can be configured in border leafs by *cross-VRF static routes*, and these static routes are imported into BGP EVPN. In this way, inter-VPC traffic are exchanged at border leafs, and go through two VxLAN tunnels. For example, VM_A is attached to server leaf L_1, and VM_B is attached to server leaf L_2. They belong to different VPCs. We can configure two cross-VRF static routes in border leaf B_1 to let them communicate. In this way, traffic from VM_A to VM_B follow the path of L_1, B_1 and L_2. The traffic go through two VxLAN tunnels (i.e., L_1 to B_1, and B_1 to L_2) with corresponding VNI labels that are rewritten in B_1.

Firewall: Firewalls are mainly used to examine all north-south traffic, protecting a data center from attack from the outside Internet. Firewalls also have independent VRFs (aka. *vsys*) to simulate virtual firewall instances for tenants. Firewalls provide network address translation (NAT) function to north-south traffic. A special packet forwarding feature that firewall NAT introduces is the bidirectional NAT, where a bidirectional NAT policy is composed of a source NAT and a destination NAT. The processing in bidirectional NAT is *stateful*: a packet matches destination NAT will definitely apply the corresponding source NAT when it is to be forwarded out of the device. Moreover, different from switches, firewalls leverage security zones that group interfaces to classify security areas, and policy rules are defined on zones.

Policy-Based Routing: Policy-based routing (PBR) is a mechanism that forwards packets based on defined policy. In particular, PBR is to first define a group of packets (e.g., through ACLs), and enforce network devices to forward the defined traffic according to the defined actions (e.g., *permit, deny, redirect*), bypassing the original data plane forwarding. Moreover, the *redirect* action is to look up the FIB to find the final outgoing interface for the defined traffic. It is commonly used with firewalls to control east-west traffic in network overlay DCNs, where some east-west traffic are redirected to firewalls, and access policy on firewalls further control the redirected traffic.

Microsegmentation: Another notable feature in network overlay DCNs is the use of microsegmentation (MCS). MCS is an alternate way to control the east-west traffic, which does not forward the traffic to firewalls, and relies on local policies to perform access control. In particular, MCS defines groups, that are essentially a set of VMs defined on switches locally, and group-based policies [14] such as *permit* or *deny*. Compared with conventional ACLs that define policies on individual IPs, MCS is more scalable since a group aggregate IPs that have the same policy.

For network overlay DCNs, MCS is implemented by source and destination VTEPs in a distributed manner: the source VTEP first computes a group ID by source IPs, and puts it into the VxLAN header; when the destination VTEP decapsulates the VxLAN packet to get source group ID, it computes the destination group ID by destination IPs, and processes the packet according to the specified group policies on group IDs.

Examples: To help understand these new forwarding features in network overlay DCNs, we provide some examples in Appendix A, including supported FIB rules, firewall NAT/security policies and PBR/MCS configurations.

B. Challenges

Existing approaches that support real-time incremental verification can be classified into two categories: EC approaches [5]–[10], [15] and indexing approaches [11], [12]. However, they do not completely address new challenges introduced by network overlay DCNs.

(a) Model Expressiveness: Existing EC approaches [5]–[9] only consider traditional FIB or ACL forwarding scenarios, and do not support tunnels essential in network overlay DCNs, where overlay packets are encapsulated/de-encapsulated over the VxLAN tunnel. APT [15] supports tunnels (and general packet rewrite) by introducing auxiliary flag variables and using predicate existential quantification, and adopts a device-based forwarding model where a device is equipped with several interfaces. However, the single-module abstraction for a device is designed for traditional FIB or ACL forwarding, where the forwarding pipeline is simple (FIB plus ACL) and sequential. In commercial network overlay DCNs, the processing can be quite complicated, even within a single interface. The forwarding pipeline may be a long chain, where a packet may go through several steps (e.g., PBR, MCS, overlay encap and de-encap, cross-VRF routing) before forwarded out of an interface, and the pipeline may be conditional branch (e.g., PBR bypasses the FIB forwarding).

APKeep [10][2] overcomes the model expressiveness problem of devices' pipelined forwarding behavior via a modular network model. In particular, the model splits the pipeline into several elements, and each element represents a processing step. Elements have virtual ports, and they are connected via edges with packet header space constraints. The header space

[2] APKeep does not support tunnels, but it can be extended to support tunnels as in APT [15].

is defined over a symbolic packet header that denotes fields in a real packet. ECs are then computed based on all edges' packet header space.

However, EC approaches rely on a fundamental assumption that all forwarding constraints can be represented by the symbolic packet header, and the representation is *globally* applicable for EC computation. In commercial network overlay DCNs, especially in firewalls, some processing is *locally* meaningful. For example, the constraints of zone security policy are not only specified on IPs but also on zones and interfaces belonging to the zone. Another example is the packet forwarding times constraint within the firewall: a packet is not allowed to travel over two VRFs [16]. Both constraints are hard to represent using the conventional packet header.

(b) Model Efficiency: Even though there may be a model to tackle the model expressiveness problem for EC approaches, they may encounter an efficiency problem. The device splitting method (i.e., the modular design) may lead to excessive elements and edges (note that each step corresponds to a new element, and the NAT element defines a new edge for each NAT rule [10].). It may impact the EC computation performance (for both full or incremental). In addition, supporting tunnels in network overlay DCNs involves with intensive packet rewrites, and APT [15] has shown that the number of packet rewrite has a strong impact on the performance.

Another drawback is that the EC computation framework may over-compute the number of ECs for the distributed processing in network overlay DCNs (e.g., distributed group ID definitions in MCS). For example, switch *L1* defines *192.168.1.0/24* in *vpn1* with source group ID *1*, and switch *L2* defines *192.168.2.0/24* in *vpn2* with destination group ID *2*. They belong to different VRFs, and may not be reachable if there are no cross-VRF routes. However, current EC approaches will generate three ECs by intersecting their header space, although they are not reachable.

(c) Algorithm Efficiency: Indexing approaches [11], [12] are explored to support incremental verification. The key idea is to record some information (e.g., arrived flows or pairs) on each interface, extract the stored information by network changes, and perform incremental computation.

NetPlumber [11] is the first paper to use such technique. In particular, each rule node in the graph model stores all arrived flows. If a rule changes in the new graph, fetching stored flows of the rule node and computating from the changed point can find reachability changes. However, NetPlumber suffers from extensive memory usage due to its rule-based model, and performs badly for large-scale networks [10].

TenantGuard [12] is the state-of-the-art indexing approach to support incremental verification. For N endpoint interfaces, it computes the reachability of N^2 pairs, and associates each device with reachable or unreachable pairs that have visited the device. For devices that have delta changes, TenantGuard extracts the potentially changed pairs indexed from the table, re-computes these pairs' reachability results and compares them with the original reachability results to identify changes. However, the per-pair computation of the all-pair reachability matrix is quite inefficient, since N endpoints lead to N^2 pairs. In addition, the device-level association is coarse-grained, which may lead to unnecessary re-computations. For example, the border leafs may carry many inter-VPC traffic. If a change only affects one inter-VPC pair, due to device-level association, all pairs that visit border leafs are re-computed.

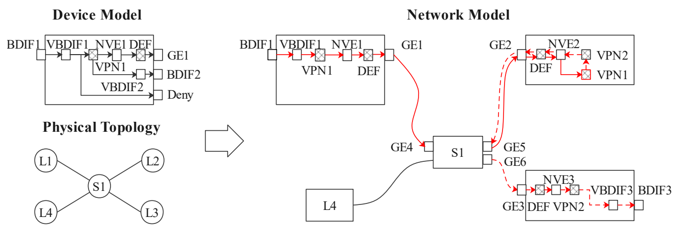


Fig. 2: An example of our port predicate graph model

III. Design Overview

A. Port Predicate Model

To tackle challenge (a), we follow the modular network model design as in [10] that abstracts each processing step as a block, and build a port predicate graph model that can accommodate all kinds of forwarding behavior in network overlay DCNs. In particular, we follow the device's forwarding behavior that interfaces are the basic blocks for packet forwarding, and define interfaces (ports) as graph nodes. A directional edge connects two interfaces that can forward packets from one to the other. We also introduce some virtual ports that represent middle states of the packet processing. Moreover, each edge is associated with a header space that represents the forwarding constraint, and the header space is represented by a predicate on a symbolic packet.

Figure 2 shows an example of our model. In this example, layer-2 overlay interfaces (i.e.., bridge-domain interfaces, BDIFs) are endpoints. The left side shows a simplified device model. BDIFs first connect with corresponding layer-3 overlay interfaces (i.e., virtual BDIFs, VBDIFs). Inside the VBDIF interface, packets may be dropped by configured ACLs, or forwarded according to the FIB in the VRF that the VBDIF interface belongs to (e.g., *VPN1*). Note that, the outbound interfaces can be NVE (e.g., *NVE1*) or local interfaces (e.g., *VBDIF2*). If former, it means the packet will enter another VxLAN tunnel: the packet will be first encapsulated, and then enter the default VRF (i.e., *DEF*) to find the outgoing interface (e.g., physical interface *GE1*).

The right side shows the network model composed of device models and physical topology. In particular, it shows the complete path of a cross-VRF overlay packet from *BDIF1* to *BDIF3*. The packet starts from *BDIF1* in server leaf *L1*, and then enters associated *VBDIF1*. After checking the *VPN1*'s FIB, it matches the VxLAN route to server leaf *L2*. *L1*'s NVE adds outer IP header and VxLAN header (with VNI *VPN1*), and forwards it to spine *S1*. Following underlay routes in FIBs, it can reach server leaf *L2* and is de-encapsulated by *L2*'s NVE. By checking the packet's VNI, it enters the *VPN1* FIB, and gets forwarded to *VPN2*. The match route in *VPN2* is another VxLAN tunnel. Following similar processing, it finally arrives *L3*, and gets forwarded to the outbound interfaces *VBDIF3* and *BDIF3* according to the new VNI value (i.e., *VPN2*) inside the packet.

To avoid computing excessive ECs (i.e., challenge (b)) and handle the local forwarding constraint representation problem, we choose not to compute ECs, and use raw header space for reachability computation. In particular, we introduce some new fields in the symbolic packet header to represent local processing: these fields are useful only inside a device, and are cleaned up just before packets leave the device. For example, to address the zone security policy problem mentioned in Section II-B, we add a new field in the symbolic packet header to denote zones. When a packet enters an interface, the field is rewritten to the zone number that the interface belongs to. After the packet is to be forwarded out of an interface, the destination zone number is extracted, and zone security policy applies. Moreover, if the packet is not dropped, the field is erased before it is forwarded to the next device. Note that locally unique is enough for the number, and there is no need for global uniqueness. Similarly, we can add a counting field to represent the *vsys* forwarding times.

B. Algorithms for Differential Reachability

To support incremental all-pair reachability computation without EC model, we follow the indexing method to compute incremental all-pair reachability, and put forth some new designs to tackle challenge (c).

As discussed in Section II-B, existing approach [12] needs to compute N^2 rounds to find all-pair reachability and establish a reachability table that associates visited *devices* with reachable or unreachable pairs. If a changed device is identified, associated pairs are indexed, and their reachability is re-computed one by one. The per-pair computation is expensive but is necessary since unreachable pairs need to be associated to trigger re-computation. Otherwise, if the reachability table only records reachable pairs, we may miss some pairs that not reachable originally but become reachable after changes.

To avoid expensive per-pair computation, we first present a general reachability computation algorithm with improved performance to find reachable pairs only and establish a reachability table that associates visited *interfaces* with them. In particular, we treat all endpoints as destination endpoints in each round, and hence only need to invokes N rounds to find initial all-pair reachability. Experiment results also show superior performance due to the aggregate computation (see Section V-A). Then we augment the general reachability algorithm with a new mechanism named *required waypoint path computation* to find all reachable pairs related to the changed interfaces in the new graph. In particular, we treat changed interfaces as required waypoints, call the general reachability algorithm in the forward direction and backward direction to find reachability paths, and combine them to find all reachable pairs that visit the changed interfaces. Then we look up the all-pair reachability table to find the original reachable pairs, and compare them to find changed reachability.

Another new design avoids computing the all-pair reachability matrix and maintaining the reachability table, and further improves the performance of incremental computation in some cases. The above algorithm is not efficient when the waypoint path computation finds large number of paths (and pairs) in the new network. It can happen if the changed interfaces are critical interfaces in the network (e.g., NVE interface in border leafs). The inefficiency roots in the complete header space used during the required waypoint path computation. In fact, those header space that are the same between the new network and the original network do not cause reachability changes. Therefore, we present a new algorithm that first computes differential header space of the interface in the new and old graphs, and then issues two required waypoint path computations in the two graphs to find reachable pairs. Then, we compare their results to get reachability changes. Experiment results show that this algorithm is especially useful for large-scale networks (see Section VI-C).

Figure 2 also shows an example on how to compute differential reachability. Assume that *BDIF1* could not reach *BDIF3* originally, and we add a cross-VRF static route in VRF *VPN1* of *L2* to enable their reachability. Then *VPN1* node is the changed interface in our graph (with FIB changes). For the first incremental algorithm, it issues the required waypoint path computation, and finds a reachable path from *BDIF1* to *BDIF3* (the solid line denotes the backward path, and the dashed path denotes the forward path). Since the original all-pair reachability table does not associate *VPN1* with any pairs, (*BDIF1*, *BDIF3*) is the added reachability. For the second incremental algorithm, the differential header space of the old graph is *ZERO*, and that of the new graph is *BDIF2* subnets. Then the reachability path in the old graph is null, and that of the new graph is (*BDIF1*, *BDIF3*). We can also get the added reachability (*BDIF1*, *BDIF3*).

C. Architecture

Equipped with the aforementioned designs, we build PPV, a network configuration verifier. PPV is a component of the network controller. The network controller collects complete network device configurations (in CLI format), and generate incremental configurations (in YANG XML format) for service updates. PPV is mainly used for pre-deployment verification: before incremental configurations are deployed, the zero-input intent verification that returns all-pair reachability changes, or the standard intent verification that returns all possible paths of provided endpoints can be performed.

Figure 3 shows the overall flowcharts of our configuration verifier. For CLI complete configurations, it first parses them, and then converts them into network object models. Then, PPV simulates the control-plane routing protocols, and generates a converged data plane (i.e., FIB) for reachability analysis. The port predicate graph model is built using configurations and a FIB (Section IV). Based on the graph model, PPV can compute the initial all-pair reachability matrix and establish the reachability table (Section V-A). For YANG incremental configuration changes, PPV merges the changed objects, and updates existing network models. Then it simulates the control plane again to get the latest FIB. By comparing configurations and FIBs, delta interfaces that have policy or forwarding changes can be found). Plus the original all-pair reachability matrix and the reachability table, our verifier can find the reachability changes, and update the all-pair reachability matrix (Sections V-B and V-C).

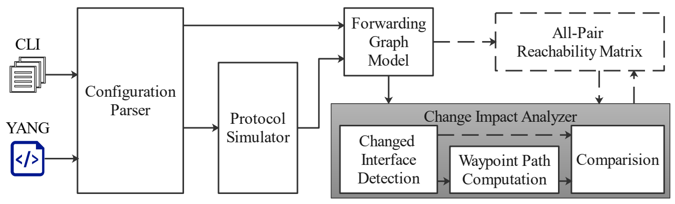


Fig. 3: Flowcharts of our configuration verifier

SrcVtepIP	DstVtepIP	VNI	GroupID	SrcIp	DstIp	SrcPort	DstPort	Protocol	Reserved
32bits	32bits	24bits	16bits	32bits	32bits	16bits	16bits	8bits	16bits

Fig. 4: The symbolic packet used in our model

IV. Model Design

A. Forwarding Graph Model

Inspired by the actual forwarding behavior in network devices, we use interfaces as the basic blocks to build the forwarding graph. An interface (or called port) is a node in the graph. For example, layer-2 BDIFs and layer-3 VBDIFs are nodes. Physical interfaces that connect network devices (e.g., Gigabit Ethernet interfaces) are also nodes. Depending on the forwarding states, a port may be further split into several nodes. For example, deny ACLs configured on an interface can lead to an interface ACL deny state, and we create a new node to denote it in the interface. Moreover, there may be some special nodes such as a VRF node that represents the forwarding state in the VRF, a *NoRoute* node that means no match routes in FIBs, and a *NullRoute* node that means the *null0* interface match.

Edges connect nodes that have potential forwarding relationship. They can represent associations of BDIFs and VBDIFs. They can also represent the forwarding relationship by FIBs: packets enter an inbound interface, and are forwarded to outbound interfaces according to their VRF's FIBs. Moreover, edges also connect two interfaces that are in the physical topology. Each edge carries a packet header space, which are the allowed packets that the edge can forward. For example, the edge from L_1's VBDIF1 to NVE only allows packets that match the VxLAN routes in VRF1's FIB. Since there may be several match rules, the edge constraint is essentially a space that combines these rules. We refer to the header space as edge constraint.

Furthermore, we develop a framework to let each edge compose constraints. In particular, the edge constraint is designed to be a general statement (i.e., an abstract object in our implementation) that represent the overall constrainted header space. It accepts an input header space and outputs the processed header space. The format of the statement can be a simple object, a list of objects or *if-then-else* objects. For example, for simple FIB forwarding without ACL definition, the statement is just a concrete object that represents the packets that the associated interface can forward. For ACL plus FIB forwarding, the statement is two objects where the first object denotes the allowed header space of the ACL, and the second object is the allowed forwarding header space. For conditional processing (e.g., multiple NAT address groups that are transformed to different range of IPs), the statement allows us to define conditions (i.e., address group) and corresponding statements (i.e., transformed IPs). In this way, when a packet header space enters the edge, it follows the if-then-else process, and generates two new header space. In the end, the final header space is the union of the two header space belonging to the two branches. To provide more flexibility, our framework also supports recursive definition (i.e., the statement can be another *if-then-else* statement).

B. Header Space Model

The next issue is how to represent header space in the graph model. The representation is crucial to the efficiency of a network verifier, since computations are just operations (e.g., AND, OR, DIFF) on the header space. Early works [11], [17] use tenary bit vector (TBV) to represent header space, and use lazy evaluation to improve the computation efficiency. Latter works [9], [10], [15] introduce logic predicate (Boolean expression) as the representation, and have shown better performance than the TBV computation.

We also use logical predicate to encode the packet header space. In particular, we first define a symbolic packet (Figure 4) that contains all reachability-relevant fields in the header. Since there is at most one tunnel in our scenario, we only model one encapsulation layer. For VxLAN packets, we add an additional IP field and a VxLAN header field, and for non-VxLAN packets, we only use the inner IP fields. In this way, tunnel encapsulation and de-encapsulation can be regarded as a kind of packet rewrites: encapsulation is just rewriting values of the outer header, and de-encapsulation is just erasing values of the outer header. In addition, we define a reserved field with 16 bits to represent some local variables discussed in Section III-A, including 2-bit zone index number (there are at most four kinds of zones), 2-bit packet forwarding count (there are at most two times).

Each bit in the symbolic packet has a corresponding variable x_i. x_i means the bit is one ($true$), $\bar{x_i}$ means the bit is zero ($false$), and non-existence of x_i means the bit is either one or zero. Non-existence of x_i can also mean the field does not exist. Then, each FIB (ACL) rule can be converted into a Boolean formula in terms of x_i, $i \in [0, len)$, where len is the length of the packet. Furthermore, according to the FIB longest-prefix match and the ACL first match principles, the header space that each rule represents can be computed by applying the *DIFF* operation on these Boolean formulas. We use Binary Decision Diagram (BDD) [18], an efficient data structure that enables fast logic operations of Boolean formulas.

The predicate model is flexible, and it can support network overlay DCNs. As we elaborated in Section IV-A, each edge represents an action. Therefore, we only need to consider how to represent different actions on predicate:

Normal Forwarding: For the FIB forwarding and ACL forwarding, we need to aggregate several packet header space. For example, several FIB rules may point to the same outbound interface. The final header space should be the union of each rule's header space. Moreover, an interface can be configured with multiple ACLs (each ACL has several permit or deny rules), and these ACLs can be applied in the *AND* or *OR* fashion. Then, the final header space is the intersection or union of each ACL's header space. It is easy to aggregate them in the predicate model using conjunction or disjunction of predicate.

Encapsulation and De-encapsulation: Overlay packet encapsulation and de-encapsulation are implemented using *packet rewrite*, and packet rewrite can be implemented using the existential quantification and conjunction on predicate as in [15]. The whole process can also be called *erase-and-set.*

Let p be a predicate, and x be one of the Boolean variables. The existential quantification of x is defined as:

$$\exists x.p = p|_{x=true} \vee p|_{x=false},$$

where $p|_{x=true/false}$ evaluates the value of p by substituting x with $true$ or $false$.

Assume the input predicate is P, and the relevant fields are $x_1, \dots, x_k$. The new value to be set is specified by predicate V. Then the whole process is denoted by

$$T(P) = (\exists x_1 \dots \exists x_k.P) \wedge V.$$

Type Check: We can check whether it is a VxLAN packet or a non-VxLAN packet using the universal quantification on predicate. The universal quantification of x is defined as $\forall x.p$. We can assert its value to check whether x has been assigned with a value. In particular, we use the full VNI field bits, and check the value of $\forall x_1 \forall x_2 \dots \forall x_k.p$, where k is the length of the VNI field. If the value is $false$, the VNI field is defined, and it is a VxLAN packet. Otherwise, it is a non-VxLAN packet.

Another type check is to check whether a property of the packet is smaller than a particular value (e.g., the *vsys* forwarding times are not allowed to be larger than two). We use the reserved field (variable) to express the property. We rely on the fact that there are limited number of values (e.g., 2-bit can represent $\{0,1,2\}$), and convert the check problem to the existence problem. For example, smaller than two means that *AND* with $\{0,1\}$ is true for the 2-bit variable.

Local Processing: To support local processing, we use the reserved field and a new variable. The length of the variable depends on the forwarding type. Initially, the variable does not have any value. When a packet enters a device, if the processing is to set a value, we use *erase-and-set* to set it. If the processing is *self-increment*, we can use *type check* and *erase-and-set* to implement it. For example, if the value is checked to be *1*, we erase the variable and set *2*. After the packet leaves the device, the variable is erased using the existential quantification.

C. Backward Graph Model

Another novel part of our model is that we also build reverse (or backward) edges in the graph. They are essential for required waypoint path computation (see the definition in Section III-B). The forward edge is mainly used to find reachable endpoints starting from an endpoint. A symmetric problem is to find start endpoints that can reach a given endpoint. If the graph has backward edges, we can call the same DFS algorithm to find all start endpoints.

Then we define the constraint of the backward edge. Note that our target is to find reachable start endpoints instead of finding the original header space of start endpoints (actually it is difficult to infer the input header space given the output header space and the forward edge's constraint). Thus, we can reuse the constraint of forward edges. In particular, for the normal forwarding action and the check action, the constraint in the backward edge is the same with the forward edge. For the *erase-and-set* actions, all values are possible for the erased fields, and we should set them to *ONE*. Therefore, assuming the input predicate is P and the edge constraint is V, the output of the reverse edge can be denoted by $\exists x_1 \dots \exists x_k.(P \wedge V)$.

V. Algorithm Design

Provided the aforementioned models, we design algorithms to compute the initial all-pair reachability matrix, and support incremental computation of the matrix. In particular, we present a general reachability computation algorithm that is fast and tailored for our network model. Then, we present two incremental algorithms based on the general algorithm, one algorithm that leverages a novel search strategy and stored information to reduce computation and the other algorithm that further reduces computation for some scenarios. The proof of algorithm correctness are provided in Appendix B.

The time complexity of all proposed algorithm is polynomial (and scalable) with respect to the network size and the number of changed interfaces under practical DCN topology. The space complexity is hard to characterize, but we have successfully demonstrated that at most 8G memory is needed for 200 leafs, a lightweight implementation for commercial use. Detailed analysis can be found in Appendix C.

A. General Reachability Computation Algorithm

We use all-pair reachability matrix computation for example to explain the algorithm. The all-pair reachability matrix is computed when the network is first uploaded to our verifier. Then it is updated incrementally. Actually, the algorithm can be used to compute incremental elements in the matrix as well (i.e., incremental verification in Section V-B and Section V-C).

To compute the initial all-pair reachability matrix, a naïve approach is to compute N^2 pairs' reachability for a network with N endpoints. In particular, we leverage the classical depth-first search (DFS) algorithm[3] (with some modifications) to find reachable header space. For a pair (S, D), we can construct the input header space (without VxLAN fields) using destination's subnet information, and then start to traverse the graph. After passing through an edge, a new header space is got by conjunction of the original header space and the edge's header space. If the conjunction leads to $false$ or this search reaches the destination endpoint, this path terminates (the latter case means the pair is reachable). Due to the possibility of multiple paths[4], even if our algorithm has found a path, it still needs to backtrack to find other paths. The traversal stops when all possible paths are explored. To avoid infinite computations,

[3] Depth-first search is superior to breadth-first search since it can also return paths explicitly, which benefits display and debug.

[4] For a pair (S, D), some packets may go from S to D directly, and other packets may visit other devices, making other path be $(S, \dots, D)$.

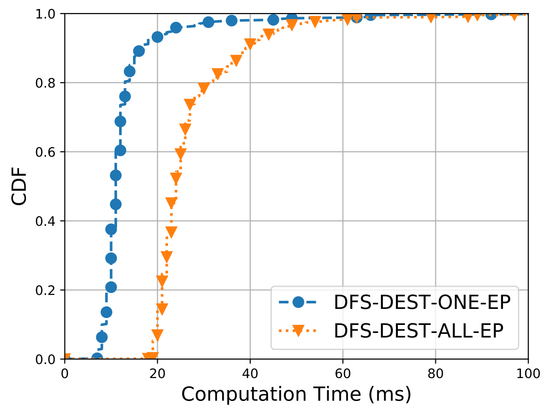


Fig. 5: Cumulative distribution function of DFS-DEST-ONE-EP and DFS-DEST-ALL-EP computation time for a network with 20 leafs (400 endpoints).

our algorithm also incorporates a loop detection routine that breaks the current search and backtracks, if a loop is found. A loop is defined as a revisit of an interface that appears in the current path with smaller header space.

A drawback of the above algorithm is an intensive computation of N^2 pairs. If we only care about reachable pairs, we can merge the destination endpoints and invoke N rounds of computation only. That is, for each round of computation, the target pair is $(S, \{D_1, \ldots, D_N\})$. Meanwhile, we use the full header space *ONE* for traversal[5], since we consider a set of destinations. The other parts of the depth-first algorithm are the same. The overall process is summarized in Algorithm 1.

Below, to compare these two algorithms, we call the first algorithm that runs N rounds to find the reachability $(S, \{D_1, \ldots, D_N\})$ as DFS-DEST-ONE-EP, and the second algorithm that one round to find the reachability $(S, \{D_1, \ldots, D_N\})$ as DFS-DEST-ALL-EP. Actually, TenantGuard [12] adopts DFS-DEST-ONE-EP, since it needs to associate devices with each pair. For PPV, it only needs to associate interfaces with reachable pairs, and thus PPV adopts DFS-DEST-ALL-EP.

Figure 5 shows the computation time of these two search algorithms (excluding the model construction time). The used dataset is a network with 20 leafs and 400 endpoints. The details of the network can be found in Section VI-B. As we can see, the performance of DFS-DEST-ALL-EP is better than that of DFS-DEST-ONE-EP, due to aggregated computation of N destination endpoints, although per-destination of DFS-DEST-ONE-EP is faster than DFS-DEST-ALL-EP. The result also explains why TenantGuard performs poorly for various incremental update cases (see Section VI), because TenantGuard needs to call DFS-DEST-ONE-EP for each possibly affected pair.

Indexing: When a reachability pair is found, we can parse the reachability path, and extract the visited interfaces. Then we can build an reachability table that links interfaces and reachability pairs. The reachability table is used in incremental computation to find possibly affected pairs in Section V-B.

[5] In fact, we can use a union of each destination's header space. A narrower input may prune more branches and terminate earlier during the search. For simplicity, we adopt the full header space *ONE*.

Algorithm 1 General Reachability Computation

Input:
- $graph$: forwarding graph model
- $startInf$: start endpoint
- $infSet$: target endpoints

Output:
- $matrix$: reachability matrix

```
1:  $path \leftarrow \{\}$
2:  $DFS(startInf, ONE, path)$
3:
4:  procedure DFS($inf$, $p$, $path$)
5:      $path.append\{inf\}$
6:      $nxtInfs \leftarrow graph.getNextNode(inf)$
7:      foreach $nxtInf \in nxtInfs$ do
8:          $p_0 \leftarrow graph.getEdgeHeaderSpace(inf, nxtInf)$
9:          if $p \wedge p_0 == false$ or $nxtIface \in path$ then
10:             continue
11:         if $nxtInf \in infSet$ then
12:             $matrix(inf, nxtInf) = \{p \wedge p_0, path\}$
13:             continue
14:         $p \leftarrow p \wedge p_0$
15:         $DFS(nxtInf,\ p,\ path)$
16:         $path.pop()$
```

However, simply recording visited interfaces does not produce the optimal performance in network overlay DCNs. Consider NVE interfaces that carry all possible VxLAN traffic. The traffic are to other VTEPs, and even to different VRFs in the same VTEP (i.e., with different VNI). If NVE is used for indexing, the coarse grained association will lead to many false positives of affected pairs. Therefore, we split the NVE interface into *NVE:NODE1:VRF1:NODE2:VRF2* using path information, and record corresponding reachable pair.

In addition, we need to differentiate the inbound and outbound direction, since policy can be applied in either direction. The direction can be extracted from the reachability path: for an edge, the outgoing interface is the outbound direction, and the incoming interface is the inbound direction.

B. Incremental Algorithm with Reachability Table

Our configuration verifier performs incremental computation of all-pair reachability in three steps, as shown in Algorithm 2. The key idea is to use **R**equired **W**aypoint **P**ath computation and a reachability table. We refer to this incremental verification algorithm as **RWP-I**.

Changed Interface Detection: We use interfaces as the basic element to find all changed nodes and their associated changed edges in the graph. To identify FIB forwarding changes, we need to first find changed forwarding rules in FIBs based on incremental configurations. For the current being, we have not implemented an incremental protocol simulator. Thus we use the full protocol simulation to generate a new FIB, and compare these two FIBs to find outbound interfaces that change their forwarding behavior. Moreover, to reduce

Algorithm 2 Incremental All-Pair Reachability Analysis I

Input:

- $indexTable$: reachability table that associates interfaces with reachable pairs
- $oldMatrix$: original reachability matrix
- $newGraph$: forwarding graph in the new network

Output:

- $addReachPairs$: pairs that are newly reachable
- $delReachPairs$: pairs that are no longer reachable
- $modReachPairs$: pairs whose reachability changed

```
1:  $infSet \leftarrow$ findChangedInfs()
2:
3:  $newReachPairs = \{\}$, $newMatrix = \{\}$
4:  foreach $inf \in infSet$ do
5:      $p_n \leftarrow newGraph.getNodeHeaderSpace(inf)$
6:      $npaths \leftarrow$ waypointPaths($inf$, $p_n$, $newGraph$)
7:      foreach $p \in npaths$ do
8:          $newReachPairs.add((p.start, p.end))$
9:          $newMatrix((p.start, p.end)).add(p)$
10:
11: $oldReachPairs = indexTable(infSet)$
12: $addReachPairs = newReachPairs - oldReachPairs$
13: $delReachPairs = oldReachPairs - newReachPairs$
14: $modReachPairs = \{\}$
15: foreach $pair \in oldReachPairs \cap newReachPairs$ do
16:     if $oldMatrix(pair) \neq newMatrix(pair)$ then
17:         $modReachPairs.add(pair)$
```

false positives that forwarding rules change but the actual forwarding remains unchanged, we re-compute forwarding edge constraints, and compare the edge predicates. If the two predicates are the same, although the FIB rules change, the interface is not marked as modified since its forwarding behavior does not change.

For policy changes such as ACLs or MCS, we iterate over all interfaces in the configuration model, and compare their associated policies. Similarly, we convert the policies to predicate in the forwarding graph, and compare the corresponding graphs to determine changed interfaces. Overall, changed nodes and edges are divided into three categories:

- **ADD:** interfaces only exist in the new network;
- **DEL:** interfaces only exist in the original network;
- **MOD:** interfaces exist in both networks, but their forwarding behavior changes.

Required Waypoint Path Computation: To compute reachability changes, a straightforward idea is to lookup the reachability table, and re-compute reachability paths of these possibly affected pairs. By comparing reachability results of these pairs, the algorithm can identify reachability changes. However, this algorithm may ignore new reachable pairs that are induced by changes. For example, originally two endpoints could not communicate, and cross-VRF static route changes on border leafs are deployed to open a new connection. In this case, no relationship between interfaces and the pair is established, and thus the approach cannot issue re-computation of the two endpoints' reachability.

Therefore, we propose a new algorithm that is able to find new reachable pairs. The key idea is to treat these changes interfaces as required waypoints, and compute all reachable paths (pairs) that visit these changed interfaces, including new reachable pairs. There is no need to consider other reachable pairs whose reachable paths do not visit changed interfaces, since their reachability are the same in the original and new networks. The argument can be proved by contradiction: if the reachability of a pair changes, there must be an interface (edge) that has a different forwarding constraint. Then the interface must be a changed interface, leading to a contradiction.

To compute required waypoint paths, previous approaches [4], [9], [19] propose to compute all possible paths, and eliminate paths that do not include the required waypoints. It is inefficient since it requires N rounds of DFS traversal for N endpoints. Therefore, we propose to use a combination of forward traversal and backward traversal to compute required waypoint paths: 1) we start from the waypoint interface, and compute all reachable paths (endpoints) via forward traversal; and 2) we use backward traversal to find all reachable paths that terminate at the interface (i.e., all endpoints that can reach the waypoint). Given all forward reachable paths and backward reachable paths (and their allowed header space), we can do conjunction to filter infeasible paths that have empty header space, and get all paths that visit the required waypoints. In our algorithm, we perform the required waypoint path computation for changed interfaces marked as **ADD** and **MOD**.

Comparison: Required waypoint path computation generates the new reachability results related to changed interfaces. Using changed interfaces with all types (including type **DEL**), we can also lookup the reachability table to extract reachable pairs in the original network, and get original reachability results from the all-pair reachability matrix. By comparing the two sets of reachable pairs and their reachability results, we can find reachability changes. Meanwhile, we can update the all-pair reachability matrix by reachability changes. In this way, we can keep the all-pair reachability matrix and the reachability table up to update.

C. Incremental Algorithm without Reachability Table

The algorithm presented in Section V-B is not efficient if the required waypoints are critical nodes that are associated with many reachability pairs in the network (e.g., NVE in border leafs). If a required waypoint path computation is performed, all related reachability paths (pairs) are computed. Even worse, different from the single-rule change setup in previous data-plane incremental verification algorithms [5]–[10], configuration changes always involve with several devices (consider a route update propagates a prefix to the whole network). It may lead to several waypoint path computation in different devices, further lowering down the performance.

The problem is caused by using the complete header space of changed interfaces in the new network. In fact, with the header space information in the original network, we can differentiate them to get the *diff* header space that is added or removed. Then, we use the diff header space for required

Algorithm 3 Incremental All-Pair Reachability Analysis II

Input:

- *oldGraph*: forwarding graph in the old network
- *newGraph*: forwarding graph in the new network

Output: same with Algorithm 2

```
1: $infSet \leftarrow$ findChangedInfs()
2: foreach $inf \in infSet$ do
3:     $h_i^o \leftarrow oldGraph.getNodeHeaderSpace(inf)$
4:     $h_i^n \leftarrow newGraph.getNodeHeaderSpace(inf)$
5:     $\delta_i^o \leftarrow h_i^o - h_i^o \wedge h_i^n$
6:     $\delta_i^n \leftarrow h_i^n - h_i^o \wedge h_i^n$
7:     $opaths \leftarrow$ waypointPaths($inf$, $\delta_i^o$, $oldGraph$)
8:     $npaths \leftarrow$ waypointPaths($inf$, $\delta_i^n$, $newGraph$)
9:     foreach $p \in opaths.paris \cup npaths.pairs$ do
10:        if $p \in opaths.pairs$ and $p \notin npaths.pairs$ then
11:            $delReachPairs.add(p)$
12:        if $p \notin opaths.pairs$ and $p \in npaths.pairs$ then
13:            $addReachPairs.add(p)$
14:        if $opaths.get(p) \neq npaths.get(p)$ then
15:            $modReachPairs.add(p)$
```

waypoint path computation in the new network. Instead of using reachability table, we further use the *diff* header space for required waypoint path computation in the original network to find related reachability pairs. Then, we compare their results to get reachability changes. The new incremental verification algorithm is referred to as **RWP-II**.

However, such optimization does not always lead to improvement, and may lead to performance degradation, since the new incremental algorithm requires two rounds of required waypoint path computation. Consider a small network. Even if changed interfaces trigger full-scale computation, the per-round computation time does not make a big difference given complete header space or diff header space. We also note that this technique cannot be used to improve RWP-I, because RWP-I needs to compute all reachable pairs in the new network. Otherwise, the comparison may produce false positives, since the indexing step finds all reachable pairs in the original network.

Overall, RWP-II is more appropriate when the network size is large (e.g., >100 leafs as shown in Section VI-C). Another advantage of RWP-II is that it does not require computation of the initial all-pair reachability matrix (and the reachability table), since RWP-II does not count on the reachability table to find reachability changes. It may save some computation when initializing the network and memory resource, and is more efficient.

VI. Performance Evaluation

A. Implementation

We have implemented the configuration verifier with full components as shown in Figure 3 as a microservice. The verifier was integrated into a network controller for pre-deployment network validation, and is to be deployed in production environment.

Our configuration verifier is based on Batfish [4], an open-source configuration verifier. In terms of the configuration and protocol models, we have the following enhancements: 1) we support the CLI parsing and modeling of Huawei configurations; 2) we support the YANG XML inputs and 3) we support BGP EVPN protocol simulation of Huawei devices.

For the verification algorithms, we reuse the forwarding graph model in Batfish, and simplify some parts to fit our port predicate model introduced in Section IV. We develop our own all-pair reachability analyzer, including the initial and differential reachability matrix computation, with around 4K lines of Java code. For the BDD library, we use JavaBDD [20]. Our implementation is single-threaded since JavaBDD does not support multi-threads for a created BDD factory. Meanwhile, we also limit the memory usage of our verifier to 8G[6] to enable lightweight deployment.

B. Setup

Datasets: We use both real network datasets and synthetic network datasets to evaluate the performance of our configuration verifier. Each dataset contains base CLI configurations and YANG XMLs corresponding to service update. The dataset includes both underlay and overlay configurations. OSPF is used as the underlay IGP protocol, and BGP EVPN is used in overlay.

For the real network, we use eight Huawei CloudEngine 6800 switches and one Huawei USG6000V virtual firewall. Meanwhile, we use Huawei AC controller to control devices and deploy new services. Before service update, we use the controller to collect full CLI configurations as base. After each service update, we use the controller to collect full CLI configurations as delta.

To evaluate the scalability of our algorithms, we use synthetic network datasets. We follow the traffic pattern in real networks to simulate the synthetic networks. The base dataset simulates a network with several VPCs. There are some inter-VPC traffic through cross-VRF static routes, and some intra-VPC traffic with MCS control. In addition, the number of VPC increases as network scales, while the scale of each VPC remains unchanged. In particular, the generation rules are as follows:

- We fix the number of VPCs per server leaf to twenty, and each VPC has ten different subnets distributed over different server leafs (to emulate that VMs belonging to the same VPC are up at different server leafs). Each subnet belongs to a layer-3 VBDIF (and layer-2 BDIF), and each VBDIF belongs to a VRF.
- Cross-VRF static routes are generated on border leafs to enable inter-VPC traffic, and the route number is equal to the number of VPCs.
- MCS policies are configured to explicitly allow access of subnets that belong to the same VPC. Each subnet belongs to a unique group ID, and MCS policy is specified on group IDs. The number of MCS policy rules is equal to the number of VPCs.

For each dataset, we also have two spines, two border leafs and one firewall. In the following, we use BDIFs as endpoints,

[6] Note that open-source Batfish [4] recommends at least 32GB RAM to verify real networks, four times of our implementation.

Dataset	Leafs (VPCs)	Subnets (EPs)	MCS Rules	Static Routes	FIB Rules
DS0	20	400	40	40	13,428
DS1	50	1000	100	100	59,808
DS2	100	2000	200	200	144,508
DS3	200	4000	400	400	388,908

TABLE I: Base network datasets

since VMs are first attached to them. Table I summarizes our base datasets. We vary the number of server leafs from 20 (i.e., 400 endpoints) to 200 (i.e., 4000 endpoints).

Service Update: For incremental configurations, we consider the following typical service updates:

- **Case A (ADD SUBNET):** We randomly choose a VRF, and add one new subnet to allow more intra-VPC subnet access. The configuration changes happen in server leafs only.
- **Case B (ADD VPC):** We create a new VPC, and distribute ten subnets over existing server leafs randomly. The configuration changes happen in server leafs only.
- **Case C (ADD INTER VPC):** We randomly choose one VPC pair, and add two cross-VRF static routes to allow more inter-VPC subnet access. The configuration changes happen in border leafs only.
- **Case D (ADD MCS POLICY):** We randomly choose one subnet pair (same VPC), and add a new MCS policy rule to allow more intra-VPC subnet access. The configuration changes happen in server leafs only.
- **Case E (ADD PBR POLICY):** We randomly choose one subnet pair (same VPC), and add PBR policy rules to redirect the traffic to firewall and then get back. The configuration changes happen in server leafs, border leafs and firewall.

Methods: We compare our algorithms (RWP-I in Section V-B and RWP-II in Section V-C) with the following algorithms:

- **TenantGuard [12].** The algorithm computes the initial all-pair reachability matrix, and establish the device-level association reachability table. Then, the algorithm lookups possibly affected pairs through changed interfaces, and re-compute their reachability. We only measure the time for incremental verification.
- **Baseline algorithm.** The algorithm re-computes the all-pair reachability matrix for incremental configurations using DFS-DEST-ALL-EP in Section V-A, and compare it with the original reachability matrix. This algorithm provides benchmark results, and is used to check correctness of other algorithms.

Metric: we measure the incremental verification time, including new forwarding graph construction time, changed interface detection time, required waypoint path computation time, and comparison time. We do not measure the protocol simulation time for incremental configurations, since they are the same for all methods.

All experiments are run on Huawei RH2288H V3 Server with Intel Xeon CPU E5-2600 v3@2.60GHz and 128G RAM. Note that our program is single-threaded, and does not use the full memory (we limit the maximum memory usage of Java Virtual Machine to 8G). For each point, we run experiments five times, and present the average result with 95% confidence interval.

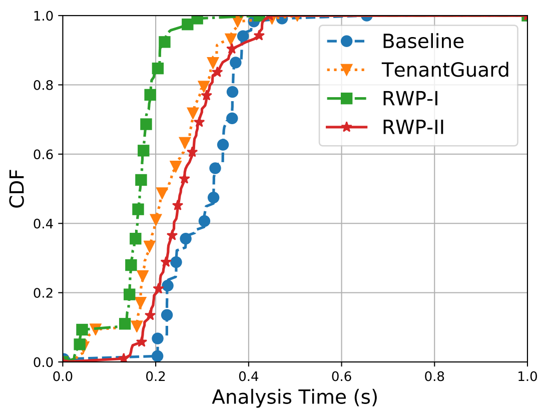


Fig. 6: Cumulative distribution function of change impact analysis time for various service updates in a real network (8 leafs with around 10 endpoints).

C. *Experiment Results*

1) Verification Time for Real Networks: We first perform experiments to see our verifier's performance in practical use, especially for small-scale networks. Figure 6 shows the overall analysis time of all algorithms for different service update cases. We generate 120 use cases, including adding/removing/deleting VPCs, BDIFs, cross-VRF static routes, ACLs and PBR policy.

For most cases (90%), RWP-I finishes the analysis within 0.25s, demonstrating a fast response time of our configuration verifier. Meanwhile, RWP-I outperforms TenantGuard (about 80%) due to its narrowed recomputation scope (changed interfaces versus changed interfaces). Furthermore, RWP-I is faster than RWP-II for the small-scale network, as explained in Section V-C. RWP-II performs worse than TenantGuard due to its heavyweight recomputation in small-scale networks, but it outperforms the baseline algorithm most of time.

2) Verification Time for Synthetic Networks: Figures 7 to 11 shows the performance of different algorithms for studied cases. We only show the results for ADD scenarios, and the results for other scenarios are similar.

Impact of different updates: For TenantGuard, the performance approaches the baseline algorithm for cases A, B and C in networks with 20 and 50 leafs. The reason is that new routes are propagated to almost all devices for A, B and C, and these devices are associated with almost all pairs, leading to almost re-computation. For case E in networks with 20 and 50 leafs, the policy changes happen in NVE at border leafs, and border leafs are also associated with almost all pairs. For other cases (e.g., cases A to E in networks with 100/200 leafs), the route propagation may not reach all devices, or the policy changes are not associated with all pairs (for larger networks, some traffic may not go through border leafs). Thus, there is a gap between them. In contrast, the performance of our algorithms is better than TenantGuard since our algorithms can filter irrelevant pairs using required waypoint path computation. The

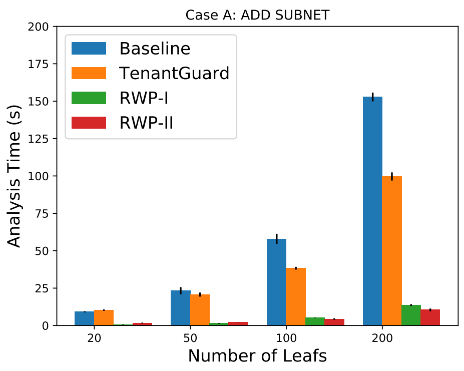


Fig. 7: Algorithm performance comparision for case A.

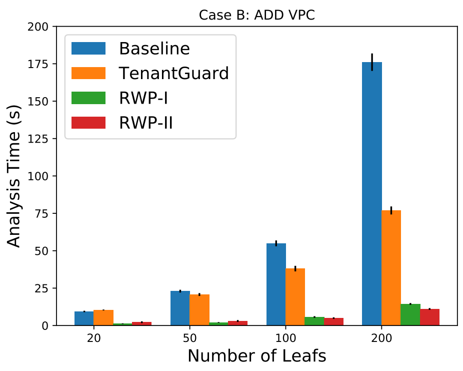


Fig. 8: Algorithm performance comparision for case B.

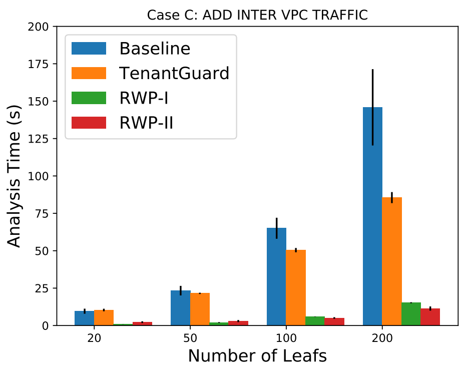


Fig. 9: Algorithm performance comparision for case C.

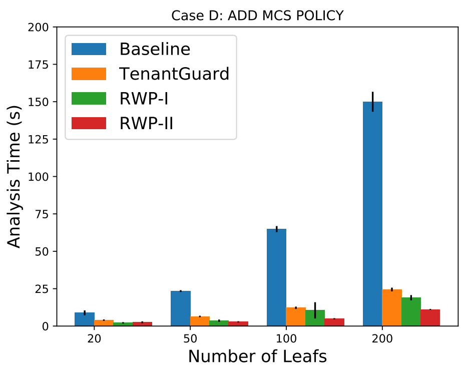


Fig. 10: Algorithm performance comparision for case D.

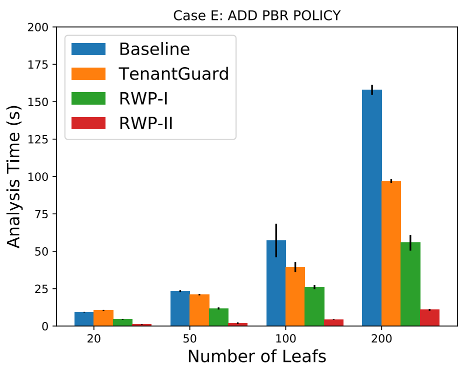


Fig. 11: Algorithm performance comparision for case E.

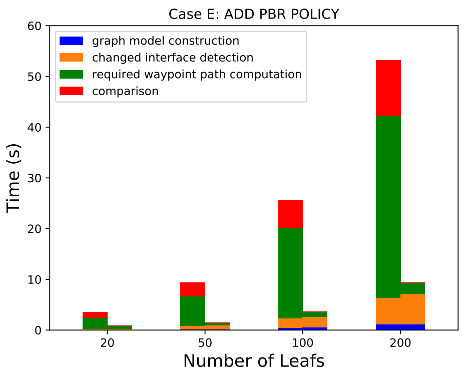


Fig. 12: Comparison of different components' computation time for our algorithms (left: RWP-I, right: RWP-II).

improvement ranges from 50% to 900% (i.e., for cases A and E with 200 leafs).

Moreover, our algorithm RWP-II is better than RWP-I for cases D and E, especially for large networks with 100 and 200 leafs. The reason is that the number of possibly affected pairs that RWP-I computes is large, since changed interfaces in case D and E are associated with many reachability pairs. Instead, RWP-II uses the differential header space, and improves the accuracy of possibly affected pairs. Thus, the performance of RWP-II is better. For cases A to C, the performance is almost the same since the effectiveness of possibly affected pairs is almost the same. In some cases, RWP-I may perform better than RWP-II, since RWP-I only invokes one round of required waypoint path computation while RWP-II invokes two rounds.

Scalability: Another message is that RWP-II is scalable. More accurately, the performance of RWP-II depends on the scale of reachability changed pairs, and is almost irrelevant to the network scale. For cases A to E, the analysis time of RWP-II increases slightly for different number of leafs (almost linear), while that of RWP-I increases a lot. To investigate the impact of different components to our algorithms, we plot the time consumed by different components for case E for different number of leafs in Figure 12 (for case E, there is only one pair that changes reachability). As we can see, the time consumed by required waypoint computation is almost the same, and the comparison time is negligible, meaning RWP-II is accurate to find relevant pairs that change reachability. However, the graph model construction time and the changed interface detection time mainly contribute to the linear increase. It is due to that our algorithms compare the whole network model to find changed interfaces, while the model becomes more complex as network scales. Meanwhile, the required waypoint path computation time mainly contributes to the time increase of RWP-I, since using current header space leads to many false positives of possibly affected pairs as network scales.

Overall, the results show that our algorithms are significantly better than existing incremental approaches. In addition, RWP-I performs almost better than RWP-II for small-scale networks, while RWP-II is preferable if the service update leads to many critical interfaces that change their forwarding behavior (especially for large-scale networks). Moreover, although we only simulate networks up to 200 leafs, we believe that our incremental verification algorithms (especially RWP-II) are scalable for larger networks, since the current results show almost linear increases as network size increases.

3) Protocol Simulation Time: So far, we have assumed the use of a complete protocol simulation even for incremental configurations. The assumption does not invalidate our conclusions, since all algorithms need simulation first. A question is whether protocol simulation slows down the whole change impact analysis procedure. Figure 13 shows the simulation time for all synthetic datasets. We group results of different service updates for the same number of leafs since they all

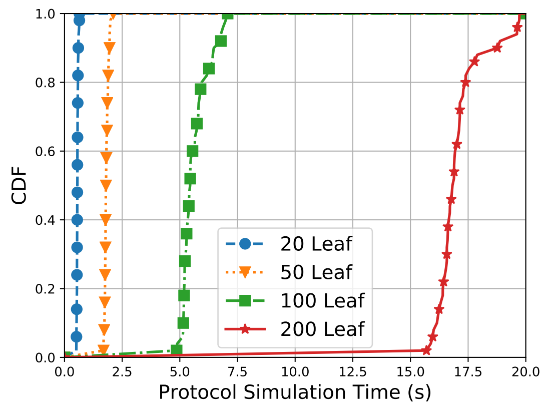


Fig. 13: Cumulative distribution function of protocol simulation time for synthetic networks.

use a complete simulation. As we can see, for the studied cases, the simulation time increases almost linearly as the number of leafs increases. For the 200 leaf case, the maximum simulation time is at most 20s, meaning that the end-to-end time for change impact analysis is less than 35s (i.e., the overall improvement is around 4.3x), acceptable for practical usage. In future, a protocol simulator that supports incremental simulation (e.g., RealConfig [21]) could benefit the end-to-end change impact analysis. Furthermore, an incremental forwarding graph model with faster changed interface detection time is also preferable.

VII. Related Work

Data-plane Network Verifiers: ConfigChecker [22] uses symbolic model checking to model the data-plane forwarding and verify network reachability, where state is the packet IP header and the current location of the packet. Anteater [23] models the data-plane forwarding as an instance of Boolean satisfiability problems (SAT), and use a SAT solver to find counterexamples if reachability intent violations are found. HSA [17] is the first paper that proposes the header space analysis framework for data-plane verification. NOD [24] uses network optimized Datalog to encode data plane, and leverages the build-in Datalog solvers to check network invariants. However, they are all offline, and could not meet the real-time requirement of incremental verification. Furthermore, their frameworks do not support incremental verification.

Real-time data-plane verifiers [5]–[12], [15] are the closest work to us. Although NetPlumber [11] supports incremental computation, its high memory usage (due to per-rule node model and per-flow caching mechanism for all rule nodes) makes it unpractical for large-scale overlay networks. EC approaches [5]–[10], [15] either lack the generality to support new network features in network overlay DCNs, or may have efficiency problem as explained in Section II-B. TenantGuard [12] also uses indexing approach for incremental computation, but targets overlay networks without in-network policy control. Its all-pair reachability computation is inefficient and its association mechanism is suboptimal for network overlay DCNs. We present a new indexing technique, and complement the approach with required waypoint path computation.

Data-plane state consistency is quite challenging for large-scale DCNs due to unavoidable hardware faults [25], which affects the verification accuracy. To address this issue, RCDC [26] exploits the regular topology structure of DCNs, and performs local checks to identify data-plane forwarding anomaly. It is fast but it targets underlay networks and could not return end-to-end paths. Plotkin et. al. [27] also leverages the topology regularities, and uses network transformation techniques to accelerate reachability query. Although it does not solve the incremental verification problem, the transformation idea can be incorporated into our verifier to further improve performance.

Control-plane Network Verifiers: Batfish [4], [28] is the first verifier that performs real-time configuration verification for a concrete environment, and converts the control-plane verification problem to the data-plane verification problem. Batfish [28] uses Datalog to encode control-plane semantic, and relies on a fixed point computation to get a converged data plane. Open-source Batfish [4] writes customized codes to perform protocol simulation. We follow the same approach to build PPV. However, they both do not support change impact analysis in an incremental manner. Instead, PPV presents two incremental algorithms for all-pair reachability analysis, and achieves better performance than the baseline algorithm.

Most control-plane verifiers [19], [29]–[32] target multi-environment control-plane verification problem, where reachability invariants are checked under any link failures or external route advertisements. They do not perform protocol simulation, and conduct verification on control-plane network model directly. Hoyan [33] adopts the symbolic protocol simulation approach to tackle the k-failure verification problem. Their target scenarios are orthogonal to us. Bonsai [34] is the control-plane verification tool that compresses large networks into smaller ones with similar behavior. ShapeShifter [35] further provides an abstract interpretation of network control planes that achieves a tradeoff between verification scalability and accuracy. These new models can be used by PPV to improve performance.

NUV [36] is the first paper to consider configuration change impact analysis from control-plane model. The key idea is to build a table that associates configuration change type with possibly affected pairs. It first uses Bonsai [34] to simplify the model, and then leverages the table to infer possibly affected pairs. It targets IP or ACL forwarding scenarios, and thus its association guidance may not apply to network overlay DCNs. Furthermore, even if PPV uses the data-plane approach, PPV's performance (35s for 200 leafs) is comparable to that of NUV (441.16s on average for studied networks with 2 to 500 routers).

Other Network Verifiers: AWS Tiros [37] focuses on the tenant-level VPC network verification, where the forwarding behavior of VPC networks are different from network overlay DCNs. Symnet [38] and Netdiff [39] focus on data plane equivalence verification of designed VPC networks and their implementation.They use symbolic execution techniques. VMN [40] and NetSMC [41] focus on stateful network verification where policies are enforced on stateful network functions (e.g., proxies, load balancers).

VIII. Conclusion

We have presented and experimentally evaluated the first configuration verifier that supports incremental configuration verification and outperforms existing approaches by 10x for network overlay DCNs. Thanks to the new indexing technique and the new required waypoint path computation technique, our verifier achieves fast change-impact analysis of delta configurations. Going forward, an incremental protocol simulator, and supporting host overlay DCNs (including OpenFlow rules and stateful packet filtering rules) and larger networks (e.g., multi-fabric DCNs) are interesting directions of future work.

References

[1] Cisco Application Policy Infrastructure Controller (APIC). https://www.cisco.com/c/en/us/products/cloud-systems-management/application-policy-infrastructure-controller-apic/index.html.

[2] Huawei Agile Controller. http://huaweigpon.cz/wp-content/uploads/HUAWEI-Agile-Controller-Brief-Brochure.pdf.

[3] Internet Engineering Task Force (IETF). Virtual eXtensible Local Area Network (VXLAN). https://tools.ietf.org/html/rfc7348.

[4] Batfish. https://github.com/batfish/batfish.

[5] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey, "Veriflow: Verifying network-wide invariants in real time," in *USENIX NSDI*, 2013.

[6] N. Bjørner, G. Juniwal, R. Mahajan, S. A. Seshia, and G. Varghese, "ddnf: An efficient data structure for header spaces." in *Haifa Verification Conference*, 2016.

[7] A. Horn, A. Kheradmand, and M. Prasad, "Delta-net: Real-time network verification using atoms," in *USENIX NSDI*, 2017.

[8] A. Horn, A. Kheradmand, and M. R. Prasad, "A precise and expressive lattice-theoretical framework for efficient network verification," in *IEEE ICNP*, 2019.

[9] H. Yang and S. S. Lam, "Real-time verification of network properties using atomic predicates." in *IEEE ICNP*, 2013.

[10] P. Zhang, X. Liu, H. Yang, N. Kang, Z. Gu, and H. Li, "APKeep: Realtime verification for real networks," in *USENIX NSDI*, 2020.

[11] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte, "Real time network policy checking using header space analysis," in *USENIX NSDI*, 2013.

[12] Y. Wang, T. Madi, S. Majumdar, Y. Jarraya, A. Alimohammadifar, M. Pourzandi, L. Wang, and M. Debbabi, "TenantGuard: Scalable runtime verification of cloud-wide VM-level network isolation." in *NDSS*, 2017.

[13] Internet Engineering Task Force (IETF). BGP MPLS-based Ethernet VPN. https://tools.ietf.org/html/rfc7432.

[14] M. Smith and L. Kreeger, "VXLAN Group Policy Option," Internet Engineering Task Force, Internet-Draft, Oct. 2018.

[15] H. Yang and S. S. Lam, "Scalable verification of networks with packet transformers using atomic predicates," *IEEE/ACM Transactions on Networking*, vol. 25, no. 5, pp. 2900–2915, 2017.

[16] Huawei USG6000 Command Reference (firewall forward cross-vsys). https://support.huawei.com/hedex/hdx.do?docid=EDOC1100013380&tocURL=resources%2Fcli%2Ffirewall_forward_cross-vsys.html.

[17] P. Kazemian, G. Varghese, and N. McKeown, "Header space analysis: Static checking for networks." in *USENIX NSDI*, 2012.

[18] H. R. Andersen, "An introduction to binary decision diagrams," in *Lecture Notes*, 1997.

[19] S. Prabhu, K. Y. Chou, A. Kheradmand, B. Godfrey, and M. Caesar, "Plankton: Scalable network configuration verification through model checking," in *USENIX NSDI*, 2020.

[20] J. Whaley. Java library for manipulating bdds. http://javabdd.sourceforge.net/.

[21] P. Zhang, Y. Huang, A. Gember-Jacobson, W. Shi, X. Liu, H. Yang, and Z. Zuo, "Incremental network configuration verification," in *ACM HotNets*, 2020.

[22] E. Al-Shaer, W. Marrero, A. El-Atawy, and K. ElBadawi, "Network configuration in a box: towards end-to-end verification of network reachability and security," in *IEEE ICNP*, 2009.

[23] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King, "Debugging the data plane with anteater," in *ACM SIGCOMM*, 2011.

[24] N. P. Lopes, N. Bjørner, P. Godefroid, K. Jayaraman, and G. Varghese, "Checking beliefs in dynamic networks," in *USENIX NSDI*, 2015.

[25] P. Gill, N. Jain, and N. Nagappan, "Understanding network failures in data centers: Measurement, analysis, and implications," in *ACM SIGCOMM*, 2011.

[26] K. Jayaraman, N. Bjørner, Padhye, and et.al., "Validating datacenters at scale," in *ACM SIGCOMM*, 2019.

[27] G. D. Plotkin, N. Bjørner, N. P. Lopes, A. Rybalchenko, and G. Varghese, "Scaling network verification using symmetry and surgery," *SIGPLAN Not.*, vol. 51, no. 1, p. 69–83, Jan. 2016.

[28] A. Fogel, S. Fung, L. Pedrosa, M. Walraed-Sullivan, R. Govindan, R. Mahajan, and T. Millstein, "A general approach to network configuration analysis," in *USENIX NSDI*, 2015.

[29] A. Gember-Jacobson, R. Viswanathan, A. Akella, and R. Mahajan, "Fast control plane analysis using an abstract representation." in *ACM SIGCOMM*, 2016.

[30] S. K. Fayaz, T. Sharma, A. Fogel, R. Mahajan, T. D. Millstein, V. Sekar, and G. Varghese, "Efficient network reachability analysis using a succinct control plane representation." in *USENIX OSDI*, 2016.

[31] R. Beckett, A. Gupta, R. Mahajan, and D. Walker, "A general approach to network configuration verification." in *ACM SIGCOMM*, 2017.

[32] A. Abhashkumar, A. Gember-Jacobson, and A. Akella, "Tiramisu: Fast multilayer network verification," in *USENIX NSDI*, 2020.

[33] F. Ye, D. Yu, E. Zhai, H. H. Liu, B. Tian, Q. Ye, C. Wang, X. Wu, T. Guo, C. Jin, D. She, Q. Ma, B. Cheng, H. Xu, M. Zhang, Z. Wang, and R. Fonseca, "Accuracy, scalability, coverage: A practical configuration verifier on a global wan," in *ACM SIGCOMM*, 2020.

[34] R. Beckett, A. Gupta, R. Mahajan, and D. Walker, "Control plane compression," in *ACM SIGCOMM*, 2018.

[35] ——, "Abstract interpretation of distributed network control planes," *Proc. ACM Program. Lang.*, vol. 4, no. POPL, Dec. 2019.

[36] Y. Li, Z. Wang, X. Yin, X. Shi, J. Wu, F. Ye, J. Yao, and H. Zhang, "Assisting reachability verification of network configurations updates with nuv," *Computer Networks*, vol. 177, p. 107326, 2020.

[37] J. Backes, S. Bayless, B. Cook, and et.al., "Reachability analysis for aws-based networks," in *CAV*, 2019.

[38] R. Stoenescu, M. Popovici, L. Negreanu, and C. Raiciu, "Symnet: Scalable symbolic execution for modern networks." in *ACM SIGCOMM*, 2016.

[39] D. Dumitrescu, R. Stoenescu, M. Popovici, L. Negreanu, and C. Raiciu, "Dataplane equivalence and its applications." in *USENIX NSDI*, 2019.

[40] A. Panda, O. Lahav, K. Argyraki, M. Sagiv, and S. Shenker, "Verifying reachability in networks with mutable datapaths," in *USENIX NSDI*, 2017.

[41] Y. Yuan, S.-J. Moon, S. Uppal, L. Jia, and V. Sekar, "NetSmc: A custom symbolic model checker for stateful network verification," in *USENIX NSDI*, 2020.

[42] E. Al-Shaer and S. Al-Haj, "Flowchecker: Configuration analysis and verification of federated openflow infrastructures," in *ACM SafeConfig*, 2010.

[43] S. Wallin and C. Wikström, "Automating network and service configuration using netconf and yang," in *USENIX LISA*, 2011.

Lizhao You received the Ph.D. degree from The Chinese University of Hong Kong, China, in 2016, and the B.S. and M.E. degrees from Nanjing University, China, in 2009 and 2013, respectively. He joined Huawei Technologies Co., Ltd. after his graduation until 2021. He is currently an Assistant Professor at Xiamen University, China. His research interests include wireless networks, computer networks and network verification.

Jiahua Zhang recieved the B.E. and M.S. degrees from Peking University, China, in 2012 and 2015, respectively. He joined Huawei Technologies Co., Ltd. in 2017, as a software development engineer. His research interests include network verification, network automation, container networks and wireless networks.

Yili Jin received his B.E. degree from Sun Yat-Sen University in 2021. He worked as a software development engineer intern at Huawei Technologies Co., Ltd. since October 2020. He is now with The Chinese University of Hong Kong, Shenzhen. His research interests include network verification and network automation.

Hao Tang is a developer at Huawei Technologies Co., Ltd.. He obtained the M.S. degree in Computer Science from the University of Manchester in 2016. His current research interests cover cloud data center networks, network verification and network automation.

Xiao Li graduated from Zhejiang University in 2000 and is now a senior cloud network technology expert of Huawei Technology Co., Ltd.. His research interests include large-scale data center network, public cloud infrastructure network, and WAN acceleration.

Appendix A: New Forwarding Feature Examples

FIB Rules: Below shows an example FIB of Huawei devices in network overlay DCNs, including a local route, a VxLAN route and a cross-VRF static route. The routing table name *vpn1* means it belongs to the VPC that is implemented as VRF *vpn1*. The special outgoing interface *VXLAN* indicates it is a VxLAN route, and the next hop is the IP address of target NVE. The cross-VRF static route uses target VRF's name as the outgoing interface, and the special IP address *0.0.0.0* as the next hop.

```
Routing Table: vpn1

Destination/Mask    Protocol    Pre    Cost    Flags    NextHop
       Interface
192.168.1.0/24      Connected   0      0       NA       192.168.1.1  Vbdif1
192.168.2.1/32      IBGP        255    0       NA       10.1.1.1     VXLAN
192.168.3.0/24      Static      60     0       NA       0.0.0.0      vpn2
```

Firewall Security Policy: Below shows a concrete configuration of a zone security policy in a firewall. In addition to the IP constraint, the policy also has a zone constraint, where a zone is mapped to its associated interfaces. For a packet, the source zone is determined when it enters an interface, the destination zone is determined when it is to be forwarded out of an interface, and then the security policy takes effect.

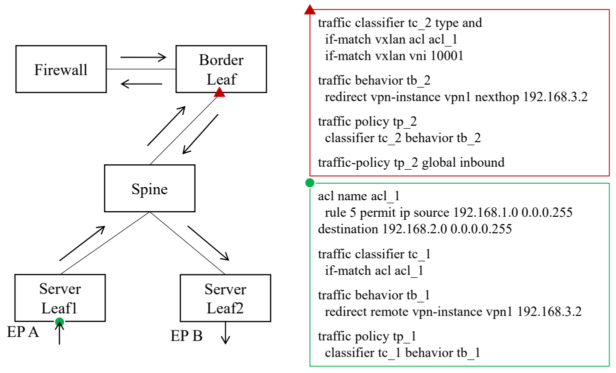
```
firewall zone trust
add interface GigabitEthernet1/0/0

firewall zone untrust
add interface GigabitEthernet1/0/1

security-policy
rule name policy_sec_01
source-zone untrust
destionation-zone trust
source-address 192.168.1.0 24
destination-address 192.168.2.0 24
action permit
```

Firewall NAT Policy: Below shows a concrete configuration of a bidirectional NAT policy in a firewall. Bidirectional NAT is usually used to simplify internal routing configurations where the originally external source IPs are replaced by internal firewall IPs, and thus there is no need to introduce routes to external IPs inside the network (but there are routes to the replaced internal IPs). In the bidirectional NAT example, destination NAT (DNAT) and source NAT (SNAT) are bonded together. If a packet matches DNAT, SNAT will be surely applied when it is to be forwarded out of the device. This forwarding feature introduces stateful packet processing: we need to remember whether the packet matches DNAT in the bidirectional NAT. In this way, we can ensure applying SNAT by checking the state. Note that it is difficult to directly use the original destination IP address (i.e., *1.1.10.10*) as the SNAT constraint, since the destination IP has been replaced by DNAT (and may be further changed) during the pipilined forwarding.

```
nat address-group ag1
section 0 10.2.0.10 10.2.0.15

destination-nat address-group ag2
section 1 10.2.0.7 10.2.0.8

nat-policy
rule name policy_nat_01
destination-address 1.1.10.10 32
action source-nat address-group ag1
action destination-nat static address-to-address address-group ag2
```

PBR: Figure 14 shows an example network with PBR configurations for Huawei devices. In this network, there are two server leafs, one border leaf, one spine and one firewall. The PBR configurations are on server leaf1 and border leaf. As we can see, the PBR configurations are to redirect a flow from endpoint A to endpoint B through the firewall. In server leaf1, we need to configure a traffic policy *tp_1* with *redirect* behavior to the interface that is used by firewall to connect with border leaf (i.e., *192.168.3.2* is the interface IP). The traffic policy should be applied to the incoming endpoint interface. Meanwhile, we need to configure another traffic policy *tp_2* on border leaf. The condition that combines IP constraints and VxLAN constraints (i.e., *10001* is the VNI value of VRF *vpn1*) matches the redirected traffic, and the traffic is forwarded to the firewall following the specified behavior. After firewall's processing, the traffic is forwarded back to border leaf, and may follow the normal FIB forwarding to server leaf2.

Given the above configurations, we can easily construct the service update case E in Section 6. In particular, the flow from endpoint A to endpoint B goes through spine originally. After applying PBR configurations, the flow is redirected through firewall. Furthermore, it is difficult to design an efficient incremental verification algorithm for this incremental configurations. Note that, the *tp_2* traffic policy is usually applied in the *global* mode, where all interfaces are linked with the new policy. That is, all interfaces on border leafs are marked as **MODIFY**, making required waypoint path computation output many reachability pairs. To tackle the challenge, we specially design the *RWP-II* algorithm (see Section 5.3) that achieves a good performance, as shown in Section 6.



```
traffic classifier tc_2 type and
  if-match vxlan acl acl_1
  if-match vxlan vni 10001

traffic behavior tb_2
  redirect vpn-instance vpn1 nexthop 192.168.3.2

traffic policy tp_2
  classifier tc_2 behavior tb_2

traffic-policy tp_2 global inbound
```

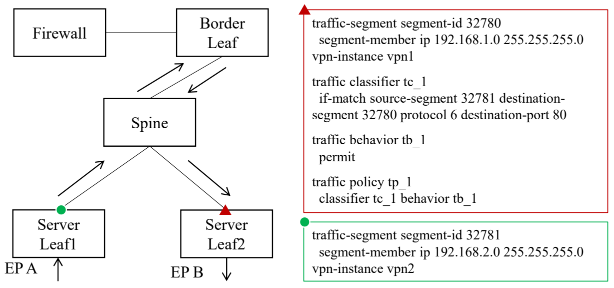
```
acl name acl_1
  rule 5 permit ip source 192.168.1.0 0.0.0.255
destination 192.168.2.0 0.0.0.0.255

traffic classifier tc_1
  if-match acl acl_1

traffic behavior tb_1
  redirect remote vpn-instance vpn1 192.168.3.2

traffic policy tp_1
  classifier tc_1 behavior tb_1
```

Fig. 14: An example network with PBR configurations

MCS: Figure 15 shows an example network with MCS configurations for Huawei devices. In this network, there are two server leafs, one border leaf, one spine and one firewall. The MCS configurations are on server leaf1 and server leaf2. As we can see, the group is defined on IPs, and the port/protocol constraints are specified through policy rules (i.e., classifier *tc_1*). In this example, the policy rule allows packets with source group ID *32781*, destination group ID *32780*, protocol *6* (TCP) and destination port *80*. Server leaf1 maps the packet's destination IP *192.168.1.0/24* in VRF *vpn1* to group *32780*, while the source group ID *32781* was generated by server leaf2 if the packet's source IP is *192.168.2.0* in VRF *vpn1*. Note that, since the source IP and the destination IP belong to two VRFs, the flow is first routed to the border leaf.

It is easy to extend the base configurations to support new services. To construct the service update case D in Section

6, we can add a new group definition in server leaf1, and a new policy rule that allows the two groups to communicate in server leaf2.



```
traffic-segment segment-id 32780
  segment-member ip 192.168.1.0 255.255.255.0
vpn-instance vpn1

traffic classifier tc_1
  if-match source-segment 32781 destination-
segment 32780 protocol 6 destination-port 80

traffic behavior tb_1
  permit

traffic policy tp_1
  classifier tc_1 behavior tb_1
```

```
traffic-segment segment-id 32781
  segment-member ip 192.168.2.0 255.255.255.0
vpn-instance vpn2
```

Fig. 15: An example network with MCS configurations

Appendix B: Proof of Algorithm Correctness

It is easy to check that Algorithm 1 is correct. So we focus on Algorithms 2 and 3.

A. Algorithm 2

Proof. It is obvious that required waypoint path computation of an interface finds all reachable pairs (paths) that visit the interface in the new graph. Looking up the reachability table can find all reachable pairs (paths) that visit the interface in the old graph. Comparing them can find reachability changes related to the interface. Since Algorithm 2 calls required waypoint path computation for all changed interfaces, we are to prove that all reachability changes are due to changed interfaces. That is, there are no pairs that changed their reachability and are not related to any changed interfaces. We prove it by contradiction.

Assume that a pair (S, D) changes its reachability, and is not related to any changed interfaces. Without loss of generality, let me assume the pair has only one reachable path. The reachable path in the old graph is $(S, V_1^o, \ldots, V_k^o, D)$, and the reachable path in the new graph is $(S, V_1^n, \ldots, V_l^n, D)$. Since there are no changed interfaces, the nexthop of S must be the same (i.e., $V_1^o = V_1^l$), and their head space intersection is the same (i.e., $h_1^o = h_1^n$). Otherwise, V_1^l must be the changed interface. Following the argument, $V_i^o = V_i^n$, $h_i^o = h_i^n$ and $l = k$. The two reachable paths are the same, and their path header space is the same. The reachability does not change, leading to a contradiction. □

B. Algorithm 3

Proof. We first show that if RWP-II uses the complete header space, the results are correct. The argument is similar to the proof of Algorithm 2: changed reachability must be related to changed interfaces.

Without loss of generality, let us focus on a particular node V_k and its associated edge. Assume the edge has header space h_k^o in the old graph, and header space h_k^n in the new graph. In the new graph, we can use the complete header space h_k^n to find all reachable paths; in the old graph, we can use the complete header space h_k^o to find all reachable paths; by comparing them, we can find rechability changes due to changed node V_k. Note that we run the algorithm for all changed interfaces, and the computed reachable pairs may overlap, since a path may visit several interfaces that change their behavior. Although there may be some overlapped computation, the results are correct.

Then we are to prove that, a pair (S, D) is computed to have reachability changes by RWP-II via the complete header space, if and only if the pair (S, D) is computed by RWP-II via the differential header space.

There are two cases: (a) all nodes in the old path and the new path are the same, but the final header space is different; (b) the old path and the new path are different. The latter case can also represent the no path case: we can add a virtual edge between the end node D (or the start node S) and the first node that does not have the next (or previous) node, and assign the null header space to it. Figure 16 illustrates these two cases, where the triangle and square nodes mean changed interface, the dashed lines mean null intersection, and the red bond lines means changed edges.

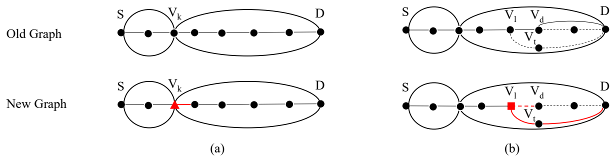


Fig. 16: Two cases that a pair (S, D) has differential reachability in the old graph and the new graph: (a) their paths have same path edges, but have different header space; (b) their paths have different edges.

Assume the old path is $(S, \ldots, V_k^o, \ldots, D)$ with length m, and the new path is $(S, \ldots, V_k^n, \ldots, D)$ with length t (m may not be equal to t). The associated edges of node V_k has header space h_k^o and h_k^n. Define $\delta_k^o = h_k^o - h_k^n$, $\delta_k^n = h_k^n - h_k^o$ and $h_k = h_k^n \wedge h_k^o$.

For **Case (a)**, their paths are the same: $V_i^o = V_i^n$. Without loss of generality, assume only V_k and its associated edge change its behavior, and other nodes remain unchanged.

Then we can define

$$h^o = h_1^o \wedge \cdots \wedge h_k^o \wedge \cdots \wedge h_m^o,$$

$$h^n = h_1^n \wedge \cdots \wedge h_k^n \wedge \cdots \wedge h_m^n,$$

and

$$\delta^o = h_1^o \wedge \cdots \wedge \delta_k^o \wedge \cdots \wedge h_m^o,$$

$$\delta^n = h_1^n \wedge \cdots \wedge \delta_k^n \wedge \cdots \wedge h_m^n.$$

Given $h_k^o = h_k + \delta_k^o$ and $h_k^n = h_k + \delta_k^n$, $h^o \neq h^n$ if and only if $\delta^o \neq \delta^n$. The two algorithms are equivalent.

For **Case (b)**, without loss of generality, assume V_l is the first node that diverts the two paths. The two paths are $(S, \ldots, V_l, V_{l-1}^o, \ldots, D)$ and $(S, \ldots, V_l, V_{l-1}^n, \ldots, D)$. Then we define

$$h^{l,o} = h_1^o \wedge \cdots \wedge h_l^o \wedge \cdots \wedge h_m^o,$$

$$h^{l,n} = h_1^n \wedge \cdots \wedge h_l^n \wedge \cdots \wedge h_t^n,$$

and

$$\delta^{l,o} = h_1^o \wedge \cdots \wedge \delta_l^o \wedge \cdots \wedge h_m^o,$$

$$\delta^{l,n} = h_1^n \wedge \cdots \wedge \delta_l^n \wedge \cdots \wedge h_t^n,$$

Case (b)-I: $h^{l,o}, h^{l,n}, \delta^{l,o}, \delta^{l,n}$ are not null. It is trivial that they are equivalent since their paths are different.

Case (b)-II: $h^{l,o}, h^{l,n}, \delta^{l,o}, \delta^{l,n}$ can be null. $h^{l,o}, h^{l,n}$ cannot be null at the same time. Otherwise, they do not have reachability changes. Without loss of generality, assume $h^{l,o}$ is null and $h^{l,n}$ is not null. Since $\delta_k^o \in h_k^o$, $\delta^{l,o}$ must be null. We are to prove that the two algorithms both can find reachability change of (S, D) (e.g., $h^{l,n}$ is not null if and only if $\delta^{l,n}$ is not null).

We take a different approach. Note that our algorithms apply to all changed nodes. We can find a new node that can find the reachability change using the two algorithms. For the new path, it diverts from V_l, and there must be some edge (V_d^n, V_{d+1}^n) in the new path that changes its header space where V_d appears later than V_l. Otherwise, there will not be a new path.

For node V_d^n, we run the algorithm. Then we have

$$h^{d,n} = h_1^n \wedge \cdots \wedge h_l^n \wedge \cdots \wedge h_d^n \wedge \cdots \wedge h_t^n,$$

and

$$\delta^{d,n} = h_1^n \wedge \cdots \wedge h_l^n \wedge \cdots \wedge \delta_d^n \wedge \cdots \wedge h_t^n.$$

For node V_d, $h_d^n = \delta_d^n$. Thus, $h^{d,n}$ is not null if and only if $\delta^{d,n}$ is not null. The two algorithms are equivalent. □

Appendix C: Algorithm Complexity Analysis

A. Time Complexity

Algorithm 1: Algorithm 1 is different from the conventional DFS algorithm that visits each node only once. In our scenario, we do not limit the number of visits. Therefore, our DFS algorithm can perform poorly in the worst case. Considering a clique of N nodes, there are at most $(N$-1)! paths, meaning exponential complexity.

The good news is that the network overlay DCNs follow a tree structure. Without loss of generality, let us consider a typical network overlay DCN shown in Figure 17(a) where there are N+1 server leafs, two spines, two border leafs and one firewall (usually there are more than one firewall, but firewalls always work in standby mode. So, we count one). The number of edges are (N+1)*2+2*2+2=2N+8. Each device has one endpoint interface. Under the practical setup, we can prove that the time complexity is $O(N)$ with respect to a network of N+6 devices.

The input of our algorithm is the forwarding graph representing Figure 17(a). Note that, as discussed in Section 4.1, each switch is further divided into several nodes (include interface nodes and state nodes) in the graph (e.g., Figure 17(b)). However, they do not affect the algorithm execution except that the traversal visits more nodes within a switch. Therefore, we treat each switch as a blackbox, and count the time complexity based on switches in the following.

Figure 17(c) shows the worst-case execution path of our DFS algorithm. In general, not all paths will be explored due to null intersection of header space and early search termination. In this graph, there are 1+(N+2)*3+2*2+1=3N+12 nodes. Next, we count the number of visited edges. We first compute the basic block in dashed square: there are 2N+2 edges. The basic block is visited five times. Therefore, edges are visited 5*(2N+2)+2*2+2+2=10N+18 times. During each edge visit, there is a header space (i.e., BDD) intersection computation. In general, the cost depends on the structure of BDDs involved. The good news is that operations used in our algorithm (i.e., conjunction, disjunction, existential quantification, universal quantification) can be implemented in polynomial time [18]. Therefore, we add a normalized cost T to each edge.

Overall, given a network with N+6 nodes and 2N+8 edges, our algorithm runs in (10N+18)*T+(3N+12)=$O(N)$ time, polynomial with respect to the input size.

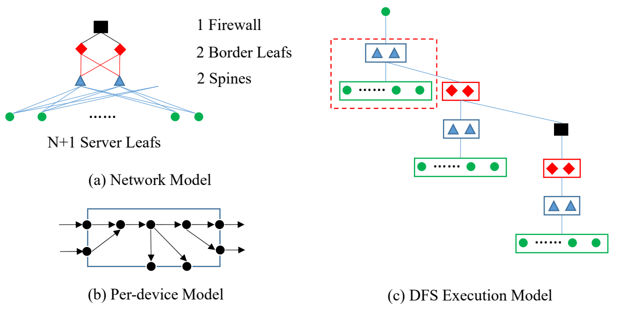


Fig. 17: (a) A typical network overlay DCN fabric; (b) An example model inside a device; (c) The DFS exection model.

Algorithm 2: Each waypoint path computation involves with one forward search and one backward search. Given the running time of Algorithm 1 is polynomial, the waypoint path computation time is polynomial too. The comparision time depends on the number of pairs, which is at most N^2. Assume that there are K ($K \le N$) changed interfaces. The overall time complexity is $K * (2O(N) + O(N^2)) = O(N^3)$.

Algorithm 3: For each changed interface, the algorithm performs two rounds of required waypoint path computation. In addition, it compares all computed reachable pairs (at most N^2) to find reachability changes. Assume that there are K ($K \le N$) changed interfaces. The overall time complexity is $K * (2 * 2O(N) + O(N^2)) = O(N^3)$.

B. Space Complexity

Algorithm 1: Similar to the conventional DFS algorithm, the algorithm uses a stack to store visited nodes. The used space is at most $O(N)$. However, we need to store all reachable paths, and establish indexes. The storage cost is proportional to the number of reachable paths, which depends on how the network is designed. In general, it is hard to characterize. In our implementation, to reduce the overall memory usage, the reachable paths are offloaded to hard-disk once computed, and only the *path* variable and the reachability table are stored in memory.

Another memory usage comes from BDD. We need to store a BDD reference table in memory for fast BDD operations. Each element in the table refers to a valid BDD variable (representing a header space). Unfortunately, it is hard to characterize the space complexity. In our implementation, we carefully manage the space used by BDD. In particular, we manually clean up all temporarily generated BDD variables. Otherwise, they will occupy the table, and increase the memory usage.

Algorithm 2: For required waypoint path computation, the used space is similar to Algorithm 1 except that the algorithm introduces a new variable *npaths*. Additional use comes from two new variables: *newReachPairs* and *newMatrix*. For each changed interface, the used space of these variables should be negligible. Overall, the space usage is proportional to the number of changed interfaces.

However, there is indeed a huge overhead related to storing the all-pair reachability matrix. For a large network with large number of endpoints (e.g., N=100K), there would be 10G endpoint pairs. Even though each pair may use a small space, the overall used space is large. Similarly, we offload the data to hard-disk once we compute the reachability results of a pair in our implementation. In this way, we can reduce the memory usage, and reuse the same memory to process all pairs.

Algorithm 3: Compared with Algorithm 2, it introduces a new round of required waypoint path computation and a new variable *opaths*. However, it avoids the use of an all-pair reachability matrix. The overall used space is smaller than Algorithm 2.

Appendix D: Discussion

Model Extension: We believe our model can be extended to support host overlay DCNs, where vSwitches use OpenFlow rules for packet forwarding and Iptables for packet filtering. OpenFlow and Iptables also follow the match-action paradigm, and match is specified on some new packet fields.We can extend the symbolic packet fields and use BDD to aggregate packets as in [42]. However, it may be difficult to support stateful rules. For example, Iptables use *conntrack* to maintain connection of incoming packets. If so, current network verifiers are inadaquate, and may need advanced techniques as in [40], [41].

Network State Consistency: State consistency (between model and reality) is critical to the accuracy of control-plane network verifiers, since they require device configurations and physical topology from the network. Fortunately, almost all modern network controllers (e.g., Cisco APIC, Huawei Agile-Controller, OpenDaylight) are equipped with NETCONF [43], a new network management protocol to manipulate configurations. It can be used to perform consistency check of configurations between devices and their models in the controller. PPV leverages the commercial controller's capability to provide a consistent view of the network state, and targets finding errors due to users' faulty policy inputs and controller's faulty implementation.

Verification under Network Faults. Network faults (e.g., link failures or hardware faults) are quite common in DCNs. Traditional distributed routing protocols (e.g., OSPF, BGP) compute new routes automatically, and thus FIB changes, when network faults happen.

PPV could support these failure scenarios as well, if network controller can capture the network state (device configurations and topology) accurately. However, PPV needs to run protocol simulation again for each failure scenario, and then triggers an incremental reachability check. As shown in Section 6, the overall re-computation time (including protocol simulation time) is around 35s for 200 leafs, which is acceptable for commercial use. Nonetheless, PPV needs to iterate over all possible failure scenarios, if we target the k-failure verification scenario as in control-plane verifiers [19], [29]–[32].

However, for overlay traffic verification in DCNs, the impact of network failures may be less critical. A typical DCN fabric usually employs several physical connections to ensure tunnel endpoint reachability. That is, even if network faults happen (and the faults do not exceed the network redundancy), the tunnel endpoints remain connected, and the overlay endpoints' reachability remain unchanged. Thus, PPV is not designed to tackle physical network failures. In contrast, PPV mainly targets finding errors due to users' faulty policy inputs or controller's faulty implementation.