Published: 28 November 2023

RESEARCH-ARTICLE

Modular Data Plane Verification for Compositional Networks

XU LIU, Xi'an Jiaotong University, Xi'an, Shaanxi, China

PENG ZHANG, Xi'an Jiaotong University, Xi'an, Shaanxi, China

HAO LI

WENBING SUN

Proceedings of the ACM on Networking, Volume 1, Issue CoNEXT3 (November 2023)
https://doi.org/10.1145/3629145
EISSN: 2834-5509

Modular Data Plane Verification for Compositional Networks

XU LIU, Xi'an Jiaotong University, China
PENG ZHANG, Xi'an Jiaotong University, China
HAO LI, Xi'an Jiaotong University, China
WENBING SUN, Xi'an Jiaotong University, China

Modern networks are increasingly using layering and bridging to form a compositional architecture. Layering protocols like VXLAN create multiple overlay networks on top of a single underlay network infrastructure. This makes network configurations even more complex, and error-prone. To check the correctness of such compositional networks, one needs to model the dependency across multiple layers (underlay and overlay) and multiple domains (different VPNs/VPCs). Existing verifiers, which are optimized to scale in single-layer single-domain networks, exhibit scalability limitations when applied to compositional networks. This paper proposes MNV, a modular network verifier that scales to large compositional networks. At its core is a new verification method termed decompose-merge reasoning, which decomposes the network into self-contained modules, verifies each module independently, and merges the verification results. Our experiments show that for a typical data center network virtualized with VXLAN, to check reachability for more than 100 million pairs of subnets, MNV is at least 100x faster than state-of-the-art tools.

CCS Concepts: • **Networks** → **Network reliability**.

Additional Key Words and Phrases: modular verification, compositional network

ACM Reference Format:
Xu Liu, Peng Zhang, Hao Li, and Wenbing Sun. 2023. Modular Data Plane Verification for Compositional Networks. *Proc. ACM Netw.* 1, CoNEXT3, Article 23 (December 2023), 22 pages. https://doi.org/10.1145/3629145

1 INTRODUCTION

Unlike classical networks where various protocols are stacked into a fixed number of layers, today's networks are composed in a much more flexible way, forming a *compositional architecture* [39, 40]. For example, cloud provider [2, 12, 25] virtualizes its data center networks so that each tenant can create their virtual private clouds (VPCs) over the physical IP network, and bridge those VPCs by the user intent, using techniques like network virtualization over layer 3 (NVO3) [6]; campus network operators isolate the traffic of departments by creating multiple virtual private networks (VPNs) using VXLAN [23]; ISPs connect multiple geographically separated sites of the same customer by creating MPLS/SRv6 tunnels in its backbone network [8, 28].

Despite its flexibility, such a compositional architecture also makes networks even more complex and error-prone. The reason is that compositional networks often build virtual networks with encapsulation protocols including IP-in-IP [26], NVGRE [10], VXLAN [23], GENEVE [14], etc., which make the protocol stack more complex. Moreover, the encapsulations can be applied many times. For example, AT&T's backbone network applies as many as eleven times of encapsulation [39].

Authors' addresses: Xu Liu, x.liu.reason@outlook.com, Xi'an Jiaotong University, Xi'an, China; Peng Zhang, p-zhang@xjtu.edu.cn, Xi'an Jiaotong University, Xi'an, China; Hao Li, hao.li@xjtu.edu.cn, Xi'an Jiaotong University, Xi'an, China; Wenbing Sun, swb2511138592@outlook.com, Xi'an Jiaotong University, Xi'an, China.

2834-5509/2023/12-ART23 $15.00
https://doi.org/10.1145/3629145

In addition, virtualized networks allow users to apply fine-grained access control, e.g., group-based policies [17], which introduces even more header fields.

To ensure a compositional network's correctness, network verification becomes indispensable. One of the most mature network verification techniques is *data plane verification* [4, 16, 20, 21, 24, 36, 37, 41], which can catch faults of forwarding rules, and misconfigured access control lists (ACLs). Over a decade, researchers from both the industry and academy have made tremendous efforts in scaling data plane verification with network *sizes*. For example, data plane verifiers [4, 16, 21, 36, 37, 41] can verify a large network with millions of forwarding rules, by computing a relatively small number of equivalence classes (ECs), satisfying that packets of the same EC have the same forwarding behavior.

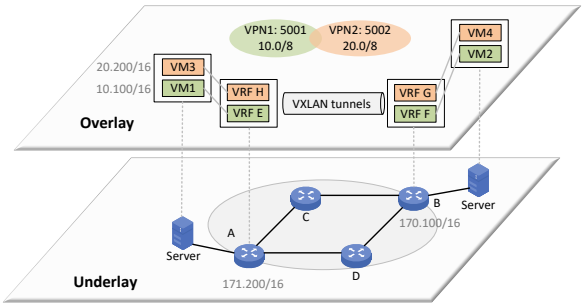
In contrast, much less effort has been made to scale data plane verifiers with network *complexity* due to network compositions. As a result, the number of ECs computed by existing data plane verifiers [16, 21, 36, 37, 41] can grow exponentially, due to frequent packet encapsulations and decapsulations in compositional networks. As the state-of-the-art data plane verifier for multi-layer networks, Katra [4] computes a much smaller number of ECs compared to previous work, while the number of ECs still reaches over 260,000 on a network with 20 routers (See §6.4).

A major reason for the above is due to the *monolithic* network model adopted by these verifiers. That is, they model the forwarding behaviors of packets across multiple layers and bridges as a whole. Some verifiers [18, 34] decompose the network into smaller components for more efficient verification. However, such decomposition requires a regular topology (e.g., data centers [13, 22]) and/or ignores the interactions of modules [5]. Recent work on modular control plane verification [1, 31, 32] decomposes a property to verify into several local component-specific properties that are easier to verify. However, this is not applicable to data plane verifications in general topologies. For example, suppose we want to check whether there is a path from source to destination that is located in different components, with multiple possible paths between them. The local properties for one component to check will depend on the specific paths chosen by another component, and cannot be determined beforehand for general topologies.

To address these problems, this paper introduces ***modular network verifier (MNV)*** to verify compositional networks. MNV applies a novel *bottom-up* verification scheme, termed as *decompose-merge reasoning*. This scheme partitions a compositional network into network modules leveraging the network's inherent modularity: *packet forwarding behaviors are closely-coupled inside the same component, but are loosely-coupled among different components.* As such, the whole verification process can be viewed as an iteration of verifying each module locally and merging the local verification results. Since ECs from different modules are relatively independent, each module can compute its own ECs, based on which to verify the reachability inside the module. As a result, MNV significantly reduces the total number of ECs, while respecting the dependency among modules. In addition, MNV uses *call graph* and *virtual rule* to ensure correctness, i.e., making the local verification raise no false positives or false negatives, and efficiency, i.e., propagating only useful results up to the upper-layer modules.

In sum, our contributions are as follows:

- We introduce *decompose-merge reasoning*, a bottom-up approach that can verify large compositional networks in a scalable way.
- We realize the decompose-merge reasoning with MNV, a new data plane verifier that uses off-the-shelf verifiers for local verification.
- We show MNV achieves a second-level verification time for synthesized updates in large data center networks virtualized with VXLAN, at least 100x faster than state-of-the-art tools.



(a) Layered topology of the network.

```
//----------Configuration of A----------
1  cts sgt-map vrf vrfE 10.0/8 sgt 10
2  vrf definition vrfE
3  vrf definition vrfH
4  vrf vrfH apply TP_A outbound
5  vlan vpn1 member vni 5001
6  vlan vpn2 member vni 5002
7  interface 10ge1/0.10
8  interface 10ge1/0.20
9  interface vlan vpn1 vrf vrfE
10 interface vlan vpn2 vrf vrfH
11 interface vlan 10 ip 10.0/9 vrf vrfE
12 interface vlan 20 ip 20.128/9 vrf vrfH
13 interface nve ip 171.200/16
14 ip route vrf vrfE 30.0/8 nexthop firewall
15 prefix-list PL_A source 10.128/9
16 traffic-policy TP_A match PL_A behavior deny
17 ospf network 171.0/8 metric 1

//----------Configuration of B----------
1  access-list role-based PL_A deny ip
2  cts permissions from 20 PL_A
3  vrf definition vrfF
4  vrf definition vrfG
5  vlan vpn1 member vni 5001
6  vlan vpn2 member vni 5002
7  interface 10ge1/0.10
8  interface 10ge1/0.20
9  interface vlan vpn1 vrf vrfF
10 interface vlan vpn2 vrf vrfG
11 interface vlan 10 ip 10.128/9 vrf vrfF
12 interface vlan 20 ip 20.0/9 vrf vrfG
13 interface nve ip 170.100/16
14 ip route vrf vrfF 20.0/8 nexthop vrfG
15 ospf network 170.0/8 metric 1
```

(b) Configuration snippets for router *A* and *B*.

Fig. 1. An example of compositional network based on VXLAN.

Limitations. The current version of MNV still has several limitations. First, the decomposition of networks still requires some domain knowledge of how specific protocols (VXLAN, NVGRE, etc.) work, and requires users to write parsers for specific vendors to extract a network model. In addition, the current MNV cannot support control plane verification. Finally, we have not evaluated networks whose size is even larger, say >1, 000 routers, which may better motivate the need for modular verification.

This work does not raise any ethical issues.

2 MOTIVATION AND RELATED WORK

2.1 Compositional network architecture

The classical Internet builds on the layered architecture, where a network consists of *fixed* number of layers, and each layer provides services to the layer directly on top of it. Today's Internet, however, has a scale and complexity way beyond the classical ones and follows a *compositional architecture* according to [39, 40]: a network is often composed of a *flexible* number of smaller or simpler *self-contained* networks, each with its own namespace, protocol stack, topology, routing, and forwarding. According to [40], there are two ways to compose networks:

- Layering. Link of an upper network is implemented by a session in a lower network;
- Bridging. Two networks communicate through their gateways.

For example, many data center and campus networks use virtualization techniques (e.g., VXLAN [23]) to create multiple logically-isolated L2/L3 virtual networks on top of a single physical network. A link of a virtual network is implemented by a session in the physical network (layering), and the communication between two virtual networks is enabled by static routes at virtual gateways (bridging). As another example, ISPs virtualize their backbone network into several Layer-3 VPNs, one for each customer [28]. Links between two geographically-separated sites of the same VPN are implemented by a tunnel (e.g., MPLS [28], SRv6 [8]) in the backbone network.

In this paper, we will term the multiple self-contained networks as *component networks* or simply *components*.

Motivating Example. Fig. 1a shows an example compositional network, adapted from real data center networks but oversimplified for ease of understanding. There are two physical servers connected by four routers. Each physical server hosts two virtual machines (VMs) of two different tenants. For each tenant, the operators use VXLAN to create a virtual network to connect its VMs, i.e., a virtual network connecting VM1 and VM2, and another virtual network connecting VM3 and VM4. In addition to isolating the traffic of the two tenants, the operators have an additional intent that VM1 should be able to reach VM3. As shown in Fig. 1a, there are at least three components here, one physical (underlay) network, and two virtual (overlay) networks. Fig. 1b shows the configuration snippets of Router *A* and *B*. loosely in the Cisco IOS language.

2.2 Composition makes networks even more complex

Indeed, the compositional architecture makes networks more flexible, since operators can compose large complex networks with smaller or simpler networks as building blocks. The downside, however, is that the network becomes even more complex and error-prone.

To get a feel of the complexity of the compositional network, we use the above example to show how a packet is forwarded from VM1 to VM3 according to the configuration snippets in Fig. 1b. First, *A* and *B* establish a tunnel between two logical interfaces named Network Virtual Edges (NVEs) (Line 13). When VM1 sends a packet to VM3, the server hosting VM1 will tag the packet with a VLAN ID of 10 that locally identifies the tenant. Since the packet has a VLAN ID 10, Router *A* will receive the packet from its Layer-2 sub-interface (Line 7), a virtual L2 interface for a specific VLAN. According to the binding between the VLAN and virtual routing and forwarding (VRF), router *A* finds the packet should go into VRF *E* (Line 11). The packet enters a VXLAN tunnel at the *nve* interface of *A*, where the VRF inserts into the packet an outer destination IP for the *nve* interface of *B* (Line 13), a virtual network identifier (VNI) 5001 (Lines 5 and 9), and a group ID of 10 (Line 1). The packet finds its way to the *nve* interface of *B*, according to routes computed via OSPF (Line 17/15 of A/B). Router *B* peels off the outer destination IP, and delivers the packet to VRF *F* associated with the VNI 5001 (Lines 5 and 9). The packet matches a group-based policy (GBP) permitting packets with group ID 10 (Lines 1-2), and then a static route (Line 14) which forwards it to VRF *G*. After that, the packet follows a similar process and finally reaches VM3.

As shown above, operators not only need to configure VLANs and routing protocols for the underlay (physical) network, but also need to configure VNIs, VRFs, GBPs, and NVEs for the overlay (virtual) networks. To enable routing among different virtual networks, the operators further need to configure the BGP-EVPN protocol [29], which is not shown in this example. Interested readers can refer to [7] for more details.

In this simple example, there is a misconfiguration: the static route at *B* (Line 14) not only permits traffic from VM1 to VM3, but also VM1 to VM4 and VM2 to VM4 by accident. Even the misconfiguration is not hard to find, while in real networks consisting of many virtual networks, network verifiers become necessary.

2.3 Limitations of existing verifiers

Researchers have made remarkable progress in building tools to verify network data planes with thousands of nodes. A key technique underlying most state-of-the-art verifiers is *equivalence classes (ECs)* [4, 16, 21, 35–37, 41]. Specifically, data plane verifiers take the forwarding rules (FIBs), filtering rules (ACLs), and encapsulation rules, as input, and based on them to partition the space of all packet headers into equivalence classes (ECs), satisfying that packets in the same EC have the same forwarding behavior. Then, the verifiers can check the desired properties against these ECs.

Even though data plane verifiers can already scale well with network *size*, they cannot scale well with network *complexity*, as observed in the compositional network. As a result, when checking

large networks with complex compositions, their performance seriously degrades. Returning to the motivating example, there would be 72 ECs, each of which is a conjunction of the following 5 matching fields:

VNI:	⟨1⟩	5001	⟨2⟩	5002		
Virtual source IP:	⟨1⟩	10.0/9	⟨2⟩	10.128/9	⟨3⟩	¬10.0/8
Virtual destination IP:	⟨1⟩	20.0/9	⟨2⟩	20.128/9	⟨3⟩	30.0/8
Physical destination IP:	⟨1⟩	170.0/8	⟨2⟩	171.0/8		
Group ID:	⟨1⟩	20	⟨2⟩	¬20		

In real compositional networks, the number of ECs can be quite large. For example, on a network with only 20 routers, even the state-of-the-art verifier computes over 260,000 ECs (§6.4).

In the following, we examine several workaround options to overcome the above limitations.

Workaround option 1. Not computing equivalence classes. A straightforward option is not to compute any ECs, but rather use predicates (Boolean formulas representing packet sets). PPV [38] takes such an approach to encode all the packet headers, including 5-tuple for the underlay, 5-tuple for the overlay, VNI, Group ID, which accounts for 256 bits per packet. Even though PPV avoids the EC explosion and can run to complete for networks up to 200 routers, it does not scale well to larger networks. According to our experiments, for networks with 500 routers, PPV takes up to tens of seconds to verify the all-pair reachabilities. This is because of the large number of Boolean variables (due to packet headers) makes the logical operations over BDDs [3] rather expensive.

Workaround option 2. Checking each component separately. Another option is to check each component separately. This requires that the verification of one component must make assumptions about the forwarding behaviors of other components. For example, to verify a two-layer network, e.g., a VXLAN network, the verification of the overlay component must assume that the underlay component satisfies all-pair reachability so as to implement every tunnel. Such a decomposition is problematic because it ignores the interaction among components [5]. For example, the underlay network may filter part of the overlay packets due to misconfigured ACLs. Since it doesn't violate the all-pair reachability of the underlay component, the verification of the overlay component will never catch this bug.

There are also several methods for checking properties for virtualized networks [18, 34]. However, they assume the networks have a specific structure, e.g., fat trees [30], VL2 [13], and therefore cannot generalize to other networks, like campus networks, wide area networks, etc.

In sum, directly applying data plane verifiers to the compositional network either encounters EC explosion or becomes quite inefficient; simple workarounds are not general, or cannot scale to large networks. How to check the correctness of the data plane for compositional networks is still an open problem.

3 DESIGN OVERVIEW

This section overviews the design of MNV. We will first introduce the basic idea and then show the workflow of MNV with an example.

3.1 Basic idea

We have the following observations when aiming to verify the correctness of compositional networks.

Observation 1: packet forwarding behaviors are closely coupled inside the component. For a certain component, changes on one node in it could affect a large portion, sometimes all, of other nodes. Taking a virtual private network/cloud (VPN/VPC) as an example, when a VPN/VPC

is created, all nodes inside it are reachable by default. Changing the configurations of any node, could affect the forwarding behaviors (paths) at all other nodes in the same component. For Fig. 1, if VRF H has another ACL rule to deny TCP destination port 22, all SSH sessions originating from VM3, and VM4 will be affected.

Observation 2: packet forwarding behaviors are loosely coupled between components. The dependencies between modules are quite limited: the forwarding behaviors in one component are only affected by a small portion of properties in the other component. For the example in Fig. 1, the forwarding behaviors from E to F in VPN1 are only affected by the reachability of packets 170.100/16 from A to B in the underlay network; changes to the forwarding behaviors of packets other than 170.100/16 in underlay will not affect the forwarding behavior within VPN1. As another example, the cross-VPN communication between VM1 and VM3, whose IP address is 20.200/16 as shown in Fig. 1a, is only affected by the reachability of 20.200/16 between VRF G and VM3 in the VPN2 network, and the reachability between VM1 and F in the VPN1 network.

Idea: decompose-merge reasoning. The above observations motivate us to adopt a *decompose-merge reasoning* approach to the verification of compositional networks. The decompose-merge reasoning works in a *bottom-up* way: it first decomposes the network into a hierarchy of *modules*, each corresponding to a component of the compositional network, and then recursively performs the following two steps: (1) verify the forwarding behavior locally within a module m, and (2) merge the local verification results of a module m into all modules that depend on m. This process continues until all modules have been verified. In our example network, we can decompose the compositional network into separate modules, such as the VPN1 module, VPN2 module, and Underlay module. The underlay module only needs to merge the reachability of packets 170.100/16 from A to B up to VPN1, so that VPN1 can perform an efficient local verification without considering irrelevant forwarding behaviors of other components.

It is helpful to draw an analogy to `quick-sort` and `merge-sort`. The assume-guarantee reasoning (a classic compositional reasoning approach that decomposes the properties that should hold globally into a set of properties that can be verified locally [11, 15, 19, 27]) works in a way like `quick-sort`: if the local properties (sub-sequences) are guaranteed (ordered), then the global properties (whole-sequence) hold (is ordered); while the proposed decompose-merge approach works in a way like `merge-sort`: it computes local properties (sort sub-sequences) from the bottom, and recursively merges the local properties up into a global property (ordered sequence).

3.2 Challenge and our Approach

To fully reveal the advantages of the decompose-merge reasoning, our approach must realize two goals: (1) ensuring the correctness of the merging process, i.e., the verification results must be consistent with the ones in the monolithic model; and (2) maximizing the decomposition of the modules, i.e., the complexity of each module must be largely tamed to accelerate the local verification. For the first goal, one tends to pack more properties into one module such that its behaviors could hold through the decomposition. Consider the example network where VPN1 and VPN2 are bridged together by a static route. A straightforward approach would pack VPN1 and VPN2 into a single module, because directly decomposing VPN1 and VPN2 into separate modules would lead to false negatives, i.e., missing the reachability from VPN1 to VPN2. However, this in turn makes the module more complex, and thus violates the second goal, since the local verification of that cross-VPN module would slow down the whole process.

The above dilemma raises a critical challenge in realizing the decompose-merge reasoning: how to ensure the correctness of verification while maximumly decomposing the compositional network? Our approach addresses this challenge by the following two designs.

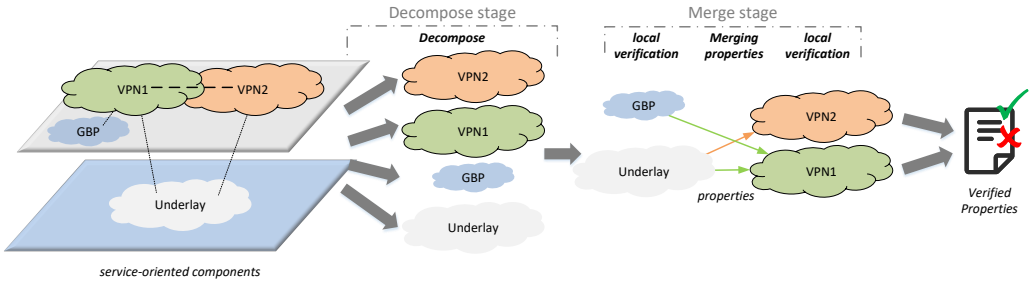


Fig. 2. Workflow of decompose-merge reasoning for the walk-through example.

First, one key reason that direct decomposing would fail is that it may miss important composition relationships between the modules, e.g., the separate VPN1 and VPN2 modules miss the bridging relationships between them. To this end, we advocate the *call graph* to reveal the composition relationships, i.e., *dependencies*, between the modules. The call graph creates a directed edge $A \longrightarrow B$ for each A depending on B. In order to capture a layering relationship, the call graph creates an edge that starts from the overlay module and points to the underlay module. For the bridging relationship, we explicitly create a bridge module, and create edges from the bridge module to the bridged modules. In the example network, we would create a bridge module and its dependencies on VPN1 and VPN2 in the call graph.

Second, given the call graph, the key to addressing the dilemma becomes how to ensure that the local verification results are correctly merged without computation redundancy. To this end, our approach composes *virtual rules* to relay the local verification results to the dependent modules. In our example network, the underlay would extract the reachability between tunnel endpoints A and B to virtual rules for VPN1 and VPN2 modules, which permit packets to pass the VPN tunnel. VPN1 and VPN2 modules would extract the reachability involving nodes representing the gateway (referred to as gateway node), such as VRF E and VRF G, to virtual rules for the bridge module, which forward packets from/to the gateway nodes. We have proved that using virtual rules ensures the correctness of decompose-merge reasoning (§4.5).

3.3 Modular Network Verifier (MNV)

Based on the decompose-merge approach, we introduce the Modular Network Verifier (MNV), a modular data plane verifier for compositional networks. Fig. 2 illustrates the workflow of MNV, which consists of a decomposition stage, and a series of local verification stages and merge stages. Before walking through the example, we first define the rule that is used to model the forwarding behaviors.

Rule. A rule is defined as $@location : match \mapsto action$, where *location* denotes the node that holds the rule, *match* specifies a set of packets, and *action* specifies the forwarding decision that should be taken on the matched packets. We use $f_V(S)$ to represent *match*, meaning that the values for fields V are contained in a set S, i.e., $f_V(S) = \{pkt | pkt.V \subseteq S\}$. Specially, we will use the notation $f_V(*)$ to represent all the packets with fields V. In our example VXLAN network, we have denoted as $V \subseteq \mathcal{V} = \{vni, vsrc, vdst, gid, dst\}$. The *action* can be one of the actions $\{fwd, deny, permit, \phi\}$: fwd is to forward the packet to the next hop such that $fwd\ X$ is to forward packets to node X, *permit* and *deny* are actions that decide whether the packets can pass the node, ϕ is the transformation so that $\phi_V(S)$ is to set the fields V to the value of S. In the following, we distinguish between two types of rules: *physical* rules which model the forwarding behavior of the network, and *virtual* rules which are created by MNV to merge local verification results.

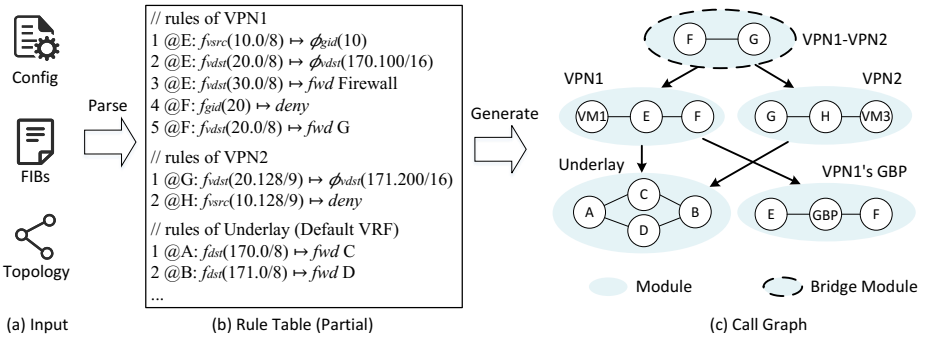


Fig. 3. Decomposition of the network in the walk-through example.

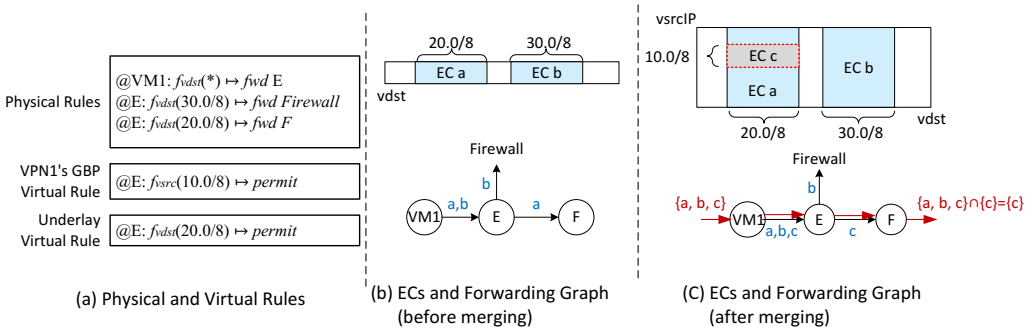


Fig. 4. The local verification for the VPN1 module.

Step 1: Network decomposition. In this step, MNV takes the configurations, topology, and FIBs as input, and generates a *rule table* (Fig. 3a, 3b). Here, we only show the physical rules that are derived based on the configuration in Fig. 1b. Based on the rule table, MNV generates a *call graph* (Fig. 3c). Here, four *modules* (Underlay, VPN1's GBP, VPN1, and VPN2) are parsed from the configurations, and one special *bridge module* (VPN1-VPN2) is a virtual one that MNV adds to model the cross-VPN reachability. As shown in the call graph, both VPN1 and VPN2 depend on the physical network module, while the VPN1 module also depends on a GBP module, which applies filtering policies. The bridge module VPN1-VPN2 depends on both VPN1 and VPN2 because the inter-VPN forwarding requires the intra-VPN reachability properties.

Step 2: Local verification for Underlay and GBP. This step is much the same as the verification of all-pair reachabilities as in existing data plane verifiers [41]. MNV directly leverages off-the-shelf data plane verifiers. The result is *reachability properties* or simply *reachability*, which are defined by 3-tuples $\langle src, dst, pkt \rangle$, meaning the module can deliver pkt from node src to node dst. In this example, the Underlay module computes the reachability $\langle A, B, f_{dst}(170.0/8) \rangle$, $\langle B, A, f_{dst}(171.0/8) \rangle$; The GBP module computes the reachability $\langle E, F, f_{gid}(\neg 20) \rangle$.

Step 3: Merging the local verification results into VPN1 and VPN2. This step abstracts the local reachability of a module into virtual rules that can be used by other modules that depend on this module. For VPN1, MNV abstracts the reachability between the tunnel endpoints, i.e., $\langle A, B, f_{dst}(170.0/8) \rangle$, of the Underlay module, into a virtual rule $@E : f_{vdst}(20.0/8) \mapsto permit$ for the VPN1 module, indicating the reachability in the Underlay "permits" packets $f_{vdst}(20.0/8)$ to reach F from E in VPN1. The reason for creating this virtual rule is because VRF E encapsulates packets $f_{vdst}(20.0/8)$ with a destination IP address $f_{dst}(170.100/16)$ at A, which has been verified

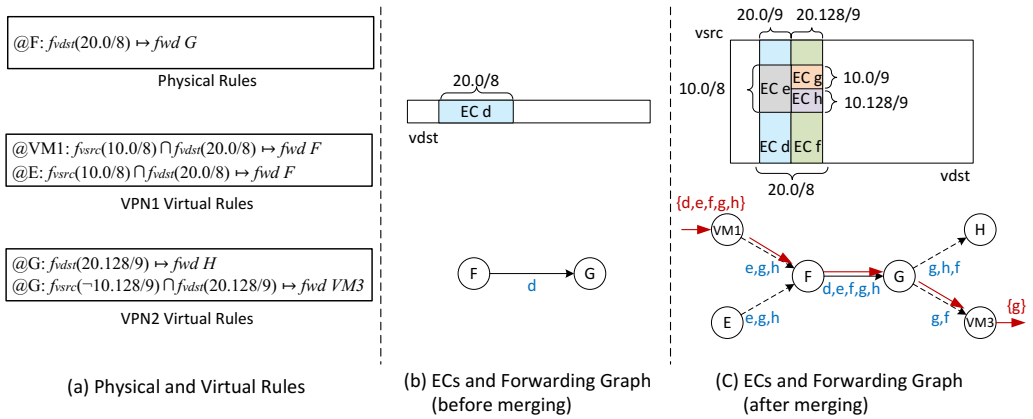


Fig. 5. The local verification for the bridge module.

reachable between A and B in Underlay. Therefore, VRF F can receive those packets at B after removing the outer IP address. Similarly, MNV abstracts the reachability $\langle E, F, f_{gid}(\neg 20)\rangle$ of the GBP module into $@E : f_{vsrc}(10.0/8) \mapsto permit$, and abstracts $\langle B, A, f_{dst}(171.0/8)\rangle$ of the Underlay module into a virtual rule $@G : f_{vdst}(20.128/9) \mapsto permit$ for the VPN2 module. These virtual rules will participate in the local verification of VPN1 and VPN2.

Step 4: Local verification for VPN1 and VPN2. Fig. 4 shows the process for VPN1. Similar to the Underlay module, MNV directly uses existing verifiers. The difference is that we should merge the verification results into the two VPN modules, using *virtual rules*. As shown in Fig. 4a, MNV inserts the two virtual rules on edge (E, F). After that, MNV updates the ECs and forwarding graph (Fig. 4b, 4c), and computes the all-pair reachability for both VPN1 and VPN2. In order to verify the cross-VPN reachability in the bridge module, VPN1 and VPN2 provide the bridge module with the reachability properties involving gateway nodes (e.g., $\langle VM1, F, f_{vsrc}(10.0/8) \cap f_{vdst}(20.0/8)\rangle$ in VPN1 and $\langle G, H, f_{vdst}(20.128/9)\rangle$ in VPN2).

Step 5: Merging the local verification results into VPN1-VPN2. In contrast to the packet transformation when merging for layering, the merging process for bridging directly sets the reachable packets as the set of match packets of the virtual rules. Specifically, MNV converts the VPN1 property $\langle VM1, F, f_{vsrc}(10.0/8) \cap f_{vdst}(20.0/8)\rangle$ to the rule $@VM1 : f_{vsrc}(10.0/8) \cap f_{vdst}(20.0/8) \mapsto fwd\ F$, and converts the VPN2 property $\langle G, H, f_{vdst}(20.128/9)\rangle$ to rule $@G : f_{vdst}(20.128/9) \mapsto fwd\ H$.

Step 6: Local verification for VPN1-VPN2. Fig. 5 illustrates the local verification for the bridge module VPN1-VPN2. Similar to the verification for VPN1, the VPN1-VPN2 inserts the virtual rules on the corresponding edges (the left of Fig. 5). The difference is that the bridge module needs to create these edges, indicating the intra-VPN reachability (the dashed arrow in the right of Fig. 5).

After the third iteration of local verification, MNV computes the all-pair reachability for every module. One benefit of decompose-merge reasoning is scalability. MNV achieves better scalability by maintaining fewer ECs per module. In the example network, as shown in Fig. 4c and Fig. 5c, VPN1 maintains 3 ECs, and VPN1-VPN2 maintains 5 ECs. Similarly but not shown, either Underlay or GBP maintains 2 ECs, and VPN2 maintains 4 ECs. Compared to the 72 ECs maintained by the monolithic methods, our modular model computes 16 ECs in total.

4 DESIGN DETAILS

This section presents the design details of MNV. We first show how MNV realizes the *decompose-merge reasoning* for a snapshot of the data plane on a single machine, and then show how MNV can support incremental verification. Finally, we discuss how to make MNV distributed.

4.1 Building the data plane model

Rule table. A rule table $\mathcal{R}$ is a table of rules that are used to module the forwarding behaviors of a network, where each entry in the rule table has the form defined in §3.3. In addition to rules that are directly parsed from FIBs and configuration files, there are two special rules that are used to capture network composition relationship:

- *Layering rules* are rules constructed to represent layering composition. Typically, a layering rule is either an encapsulation rule that matches fields from one component and inserts fields from another component, or a decapsulation rule which works reversely.
- *Bridging rules* are rules constructed to represent bridging composition. Typically, a bridging rule is a static route that matches packets from one component and forwards them to another component.

Rule table generation. MNV generates the rule table $\mathcal{R}$ based on the configuration files and FIBs, in the following three steps.

(1) Extracting rules from the FIBs. Let a FIB entry be a 4-tuple $(loc, match, nxthop, intf)$, where loc is the node where this rule is installed, $match$ and $nxthop$ are prefixes with some fields V_1 and V_2 respectively, and $intf$ is the output interface for the matched packet. If $intf$ is a physical interface, MNV creates a rule $@loc : f_{V_1}(match) \mapsto fwd\ X$, where X is the node connected to the $intf$. Otherwise if $intf$ is a tunnel (e.g., an NVE interface), MNV creates an encapsulation rule $@loc : f_{V_1}(match) \mapsto \phi_{V_2}(nxthop)$.

(2) Extracting rules from policies in configuration files. Let ply be a policy, defined as a tuple of *classifier* and *behavior*, where the classifier is a set of prefixes for fields V, and the behavior is one of the two actions: *permit* and *deny*, MNV creates a rule for each prefix p in the $classifier$, denoted as $r = @ply : f_V(p) \mapsto behavior$. For example, router A defines an ACL-based policy that denies packets with virtual source IP 10.128/9 (Line 15-16 of A, Fig. 1b), therefore, MNV creates a rule $@H : f_{vsrc}(10.128/9) \mapsto deny$.

(3) Creating encapsulation rules based on configuration fields. For each configuration that requires the routers to tag the packets, MNV creates an encapsulation rule. In our example network, router A will add a group ID tag into a packet (Line 1 of A, Fig. 1b), therefore, MNV creates an encapsulation rule $@E : f_{vsrc}(10.0/8) \mapsto \phi_{gid}(10)$ to model this. For each encapsulation rule, MNV adds the corresponding decapsulation rule at the other side of the tunnel, and the decapsulation rule can be constructed in a similar way and thus omitted here.

4.2 Decomposition of the model

This step decomposes the model of the data plane into a set of *modules*, and a *call graph* that describes the dependencies among modules.

Module. A module is defined as a tuple of an identifier and a forwarding graph $\langle id, G \rangle$. MNV will create a module for each component in the compositional network, e.g., an underlay network, or a virtual network. In addition, to model the bridging composition, MNV will also create a virtual *bridge module*.

Forwarding graph. A forwarding graph G is defined as a 4-tuple $\langle N, E, L, V \rangle$, where

(1) N is the set of nodes,

(2) $E \subseteq N \times N$ is the set of directed edges,
(3) $L : E \rightarrow 2^C$ is the map from each edge to the set of equivalence classes. C denotes the set of equivalence classes.
(4) V is the set of fields that the rules of this module match,

Creating modules. First, MNV identifies all components of the network according to *component identifiers*, which is a concept that depends on protocols. Taking VXLAN as an example, a virtual L2 network is identified by a broadcast domain ID, and a virtual L3 network is identified by a VPN instance name. For NVGRE and MPLS, the identifiers would be Virtual Subnet Identifier (VSID) and Route Distinguisher (RD), respectively. Therefore, depending on the protocols that are used for composing the network, MNV builds a corresponding parser to identify the components.

Once the components are identified, MNV creates a module for each of them. The forwarding graph of the module is constructed as follows. (1) V is the set to the fields that this component uses. For example, it will be set to 5-tuple for VXLAN which forwards packets using UDP/IP. (2) N is set to the locations that the component binds to. For example, the VPN1 module has three nodes, VRF E, VRF F, and VM1. The reason is VPN1 binds to VRFs E and F on router A and B, respectively (according to Lines 5, 7, 9, 11 of A, and lines 5, 9 of B, Fig. 1b). (3) E is set to the physical links, virtual tunnels, and an application of policy. For example, the VPN1 module has an edge $(VM1, E)$ which is a physical link, an edge (E, F), which is tunnel. The GBP module has an edge (GBP, F), which models the fact that the policy GBP is applied to the inbound of VRF F (Lines 1-2 of B, Fig. 1b). (4) L is set to an empty set for each edge, indicating the edge initially denies every packet.

Call graph. A call graph $\mathcal{G}$ is a directed acyclic graph $\langle V, E \rangle$ where V is the set of modules, and E is a set of edges between nodes in V capturing the dependency between modules. Specifically, if module m_i depends on module m_j, then we have an edge $m_i \rightarrow m_j \in E$. We will use the notation $Impl(m_i) = \{m_j | m_i \rightarrow m_j\}$.

Creating the call graph. MNV creates the call graph by identifying dependency relationships between modules based on the rule table $\mathcal{R}$. Specifically, MNV iterates over each rule $r \in \mathcal{R}$ and checks whether r is a layering rule or a bridging rule. If r is a layering rule which matches fields of component A and inserts another field of component B, then, MNV creates a dependency $A \rightarrow B$ for the call graph. If r is a bridging rule that forwards packets from one component A to another component B, then, it seems that MNV can first create a bridge module C and add the dependencies $C \rightarrow A$ and $C \rightarrow B$ into the call graph. However, this can be problematic and raise false negatives, since a component can be bridged with multiple other components. For example, suppose B is bridged with two components A and D. If we just create two bridge modules, one for A-B and the other for B-D, then, the potential reachability between A and D will be missed since there is no bridged module for A-D, resulting in false negatives (A and D are falsely concluded to be unreachable). Therefore, MNV creates a bridge module for each connected-components (here A, B, D are in the same connected-components), rather than for each pair of bridged components. This can be efficiently implemented with the union-find data structure [9].

Initially, the fields set V of one bridge module is the same as the set of the bridged module, the nodes set N contains only the gateway nodes of the bridged modules, and the edge set E contains only the links that connect the gateway nodes. Nevertheless, new nodes and edges could be added into N and E during the merging stage, we will talk about it in §4.4.

After network decomposition, MNV iterates over the local verification and the verification results merging, based on a scheduling method. Currently, MNV uses a simple scheduling method: compute a worklist according to the topological order for $\mathcal{G}$, which is a directed acyclic graph; trigger the local computation for modules that do not depend on other modules; Merge the computed verification

results into modules that depend on the triggered module, and remove the module from the worklist. There are several optimizations for the scheduling. For example, MNV can verify modules without dependencies in parallel to speed up. We leave this as future work.

4.3 Local verification

MNV computes the local properties of a module. We focus on the reachability properties in this paper. Other properties like multi-path forwarding, and waypoints, are discussed in Section 7.

Reachability property. Let $G = \langle V, N, E, L \rangle$ be the forwarding graph to be verified. The reachability property is a 3-tuple $\langle src, dst, pkt \rangle, src, dst \in N, pkt \in f_V(*)$, representing the packet pkt can reach node dst if sent from node src. Computing reachability properties is much the same with how existing data plane verifiers compute all-pair reachability: first computing the equivalence classes (ECs) based on data plane rules, then constructing a graph where edges are labeled with the ECs that can traverse (i.e., updating L), finally, traversing the graph to compute the reachable packets between nodes, by computing conjunctions on the set of ECs with the labels.

MNV differs from existing data plane verifiers in that the data plane rules not only include *physical* rules for forwarding, filtering, rewriting, but also include *virtual* rules that abstract the reachability properties of modules (refer to §4.4 for details).

4.4 Merging

After the local verification of a module, MNV abstracts the reachability properties that are locally computed in the module into a set of *virtual rules*, and inserts them into modules that depend on this module.

Virtual rule. A virtual rule is a rule constructed to represent the properties that locally hold in a module and inserted into other modules that depend on it. For different types of dependencies, we define different types of virtual rules.

Definition 4.1. **Virtual rule for bridging dependency.** If $\langle s, t, pkt \rangle$ is a reachability property of a module, and at least one of s and t is the gateway node, then the virtual rule is $r = @s : f_V(pkt) \mapsto fwd\ t$, where V are the fields of packet pkt.

Definition 4.2. **Virtual rule for layering dependency.** Given a tunnel (s', t') and its tunnel endpoints s and t, if the tunnel is implemented by the encapsulation rule $r_{enc} = @s' : f_{V_1}(S') \mapsto \phi_{V_2}(S)$, and the reachability property $\langle s, t, S \rangle$ holds, the virtual rule for the tunnel is $r = @s' : f_{V_1}(S') \mapsto permit$.

Deriving the virtual rules. Let the local property be $\langle s, t, pkts \rangle$, MNV derives virtual rules in two steps: (1) If at least one of s and t is the gateway node, a virtual rule is derived according to the Definition 4.1. In this case, MNV identifies the bridge module containing the gateway node and adds the non-gateway node (if any) to its node set N, and adds the edge (s, t) to its edge set E. Then, the virtual rule gets inserted into the bridge module. (2) If s and t are tunnel endpoints of some tunnel (s', t'), MNV finds the encapsulation rule from the rule table that implements this tunnel, derives a virtual rule according to the Definition 4.2, and inserts the virtual rule into the module containing the tunnel.

Note that Definition 4.2 could raise false positives due to the *nondeterministic encapsulation*. For example, when a packet matches an encapsulation rule $f_{vdst}(10.1/16) \mapsto \phi_{dst}(192.168/16)$, the outer destIP can be an arbitrary IP from 192.168/16. If only part of these IPs, say 192.168.2/24, are reachable on the lower layer, Definition 4.2 will flag all packets from 10.1/16 as unreachable in the

overlay, even those some of them are reachable (false positives) MNV defines a loose version of the virtual rule for this situation:

Definition 4.3. **Virtual rule for layering dependency (loose version).** Given a tunnel (s', t') and its tunnel endpoints s and t, if the tunnel is implemented by the encapsulation rule $r_{enc} = @s' : f_{V_1}(S') \mapsto \phi_{V_2}(S)$, and the reachability property $\langle s, t, P \rangle, P \subsetneq S$ holds, the virtual rule for the tunnel is $r = @s' : f_{V_1}(S') \mapsto permit$.

If a virtual rule is derived according to this loose definition, MNV can tag this virtual rule as "unsure", and the unsure tag will further be propagated to local properties that depend on this virtual rule.

MNV supports another encapsulation pattern, i.e., *deterministic encapsulation*. For example, when a packet matches an encapsulation rule $f_{vdst}(10.1.x.y) \mapsto \phi_{dst}(192.168.x.y)$, the outer destIP has the same first two Bytes as the inner destIP. MNV supports this encapsulation with the Permutation operator of BDD. Details can be found in Appendix D.

4.5 Proof of Correctness

We prove the correctness of MNV by showing the following two theorems hold.

Theorem 4.4. ***Soundness.*** *For any module m in MNV, if a reachability property $\langle s, t, pkts \rangle$ holds locally in module m, then for any $pkt \in pkts$, it can be forwarded from s to t in the monolithic model.*

Theorem 4.5. ***Completeness.*** *For any packet pkt, if it can be forwarded from s to t in the monolithic model, then there exists one and only one module m in MNV, satisfying that reachability $\langle s, t, pkts \rangle$ holds locally in module m and $pkt \in pkts$.*

Proof. By the mathematical induction on the level n of modules in the hierarchy of the call graph. See the Appendix A for details. □

4.6 Supporting incremental verification

Like other data plane verifiers, such as AP Verifier, APKeep, etc., MNV supports incremental verification on network changes (e.g., FIB changes). The key difference is that MNV not only needs to incrementally compute differential properties (e.g., reachability), but also needs to propagate those differences up to other modules that depend on it. The schedule of propagation should obey the following principle: suppose a module m_i depends on multiple other modules, i.e., $Impl(m_i)$, and the local property of one module $m_j \in Impl(m_i)$ has changed, then, we should update the properties of m_i, only when all $m \in Impl(m_i)$ have been updated. The details can be found in Appendix B.

The propagation of differences among modules brings a benefit for MNV: the propagation may terminate early when there are no modules that have changes in properties. This makes MNV limit the update scope. As shown in our experiments, the incremental verification is quite fast: <1 second for all cases, at least 100× faster than other tools.

4.7 Supporting distributed verification

The modular nature of MNV makes it an appealing choice for distributed verification. We have not implemented a full-fledged distributed version of MNV, but outline some effort in making this happen. First, we note that the rules, configurations, topologies, and forwarding graph can be distributed to separate servers for parallel processing. A tricky part is the underlying data structure, i.e., the Binary Decision Diagram. The reason is BDD achieves a compact representation of boolean formulas, by compressing redundant edges and nodes, such that different modules may share

(a) Snapshot statistics.

Network	Leafs (VPCs)	Subnets (VRFs)	Forwarding rules	MCS rules	Static routes
6Leaf	6	36	856	12	12
10Leaf	10	100	2,508	20	20
20Leaf	20	400	13,428	40	40
50Leaf	50	1,000	59,808	100	100
100Leaf	100	2,000	144,508	200	200
200Leaf	200	4,000	388,908	400	400
500Leaf	500	10,000	1,528,538	1,000	1,000

(b) Types of synthesized changes.

ID	Update	Explanation
1	StaticRoute	Add two cross-VRF static routes to a VRF pair
2	MCSAdd	Add a MCS rule to permit access for a VRF pair
3	PBRAdd	Add a redirect policy between a VRF pair
4	SubnetAdd	Add a subnet and enable all intra-VPC access
5	VPCAdd	Add a VPC with 10 subnets to random leaves

Table 1. Snapshot statistics and synthesized changes for the DCN datasets.

common BDD nodes. Therefore, using a single BDD for all modules may incur high overhead due to synchronization.

MNV chooses another option: each module maintains its own BDD. When merging the verification results across different modules, a global *translator* converts the BDD representation of one module to the BDD representation in another module. The process is basically a serialization/deserialization process, similar to the BDD I/O function in JDD [33]. Details for the BDD translator can be found in Appendix C.

5 IMPLEMENTATION

We implemented a prototype of MNV with ~ 3K lines of Java code. For local verification, we used an extended version of APKeep [41], which was modified to support VXLAN and virtual filter rules (§4.4). Note that other data plane verifiers like Katra [4], and AP Transformer [35] can also be adapted and used for local verification. We will evaluate their performance in our future work.

6 EVALUATION

6.1 Setup

Dataset We run MNV with the following two types of datasets.

- Data Center Networks (DCN). As shown in Table 1, the dataset consists of 7 configuration snapshots of different sizes, and 5 types of configuration updates for each size. The datasets are synthesized by PPV [38], based on real DCNs. The networks have a spine-leaf topology, with two spines, two border routers, and different numbers of leaf routers from 6 to 500. For the network with n leaf routers, VXLAN is used to virtualize it into n VPNs, each of which has the same number $min(n, 20)$ of subnets. That is, we bound the number of total subsets for each VPN to 20. The subnets are distributed in a way that: subnets of the same VPN are hosted at different leaf nodes, and each leaf node hosts the same number of subnets. Therefore, the network with 500 leaf routers will have 10,000 subnets, and 10^8 pairs of reachability to check.
- Multi-layer IP networks (Multi-IP). We synthesized the dataset according to Katra [4]. For each network of parameter (l, n), there are l layers, and each layer has n nodes. The first layer is the physical network, and the ith $(i > 1)$ layer is an overlay network on top of the $(i-1)th$ layer, by encapsulating the packets with an additional IP header. For each layer, packets are forwarded based on the shortest paths.

Methods to compare We compare MNV with the following data plane verifiers.

- AP Transformer [37]. We retrieve the source code from [35], and extend it to process VXLAN headers (VNI, Group ID), referred to as **APT**$^+$.
- APKeep [41]. We retrieve the source code from the author, and extend it to support packet encapsulation and VXLAN headers, refered to as APKeep$^+$.

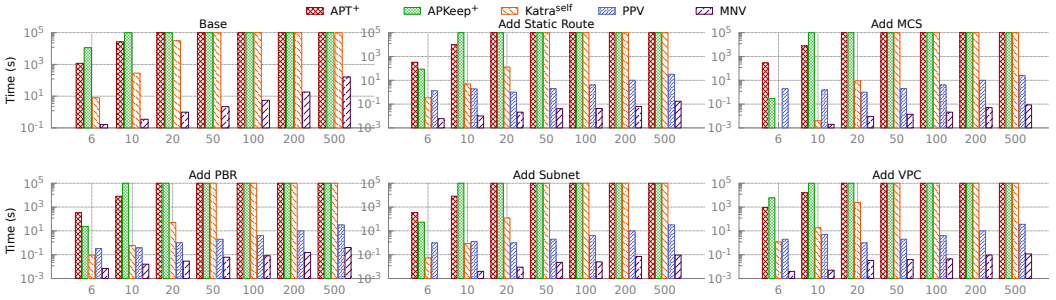


Fig. 6. The running time for DCN datasets.

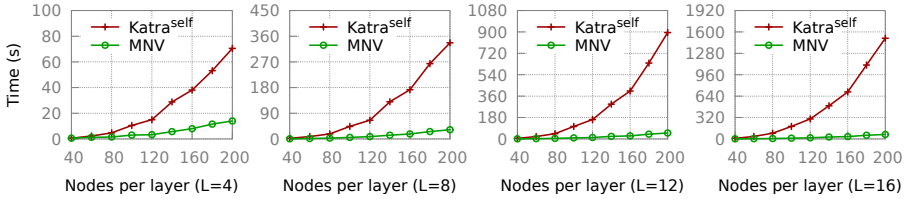


Fig. 7. The running time for Multi-IP datasets.

- Katra [4]. We re-implement Katra in Java according to the paper, and extend it to process VXLAN headers, referred to as **Katra$^{\text{self}}$**. On the Multi-IP datasets, Katra$^{\text{self}}$has a performance similar to those reported in the paper.
- PPV [38]. Since PPV is not open source, we let the authors run those experiments on their own server, and report the results.

All the experiments, except those of PPV, run on a desktop with a 3.7GHz Intel Core i5 CPU and 32GB RAM. The experiments of PPV are run by the authors on their own server, with a 2.60GHz Intel Xeon E5-2600 CPU and 128G RAM [38].

6.2 Correctness validation

We cross-validated the correctness of MNV with PPV [38] on the DCN datasets. We used both MNV and PPV to check the all-pair reachability in all networks in the DCN datasets, i.e., for each pair of endpoints, computing the packets that are reachable. Both the tools returned the same results except for a small number of pairs. For example, MNV found 16 more reachable pairs in the 20Leaf dataset. The authors confirmed that this was because the original version of the code disenabled 7 static routes for internal tests, and corrected this in their current version. After correcting this, MNV and PPV returned the same results.

6.3 Verification time

Fig. 6 shows the running time for all the methods to verify the reachability for the snapshots and updates of the DCNs. As we can see, the running time of MNV increases linearly with network size, and the time is within 1 second for all updates, which is at least 2 orders of magnitude faster than other tools. A notable thing is that no EC-based tool run to complete all datasets: APT^{+} stops at 10Leaf, APKeep^{+} stops at 6Leaf and Katra$^{\text{self}}$ stops at 20Leaf. The reason is due to EC explosion: the number of ECs is so large that the compute and/or memory is overwhelmed. On the other hand, PPV runs to completion because it uses predicates to represent packet sets without computing ECs. However, PPV is still 2 orders of magnitude slower than MNVwhen verifying updates. The reason is that using predicates to traverse the forwarding graph is slower than using sets of integers, which has been observed by AP Verifier [36].

Size	APT$^+$	APKeep$^+$	Katra^{self}	MNV
6	24,696	32,424	6,776	117
10	110,880	>92,675*	29,604	201
20	>463,092*	>17,850$^\#$	264,198	425
50	>267,697*	>15,519$^\#$	>75,378$^\#$	1,010
100	>147,864*	>7,320$^\#$	>72,457$^\#$	1,985
200	>585,312$^\#$	>2,912$^\#$	>34,629$^\#$	3,935
500	>4,140,264$^\#$	>1,371$^\#$	>13,437$^\#$	11,281

Table 2. Number of ECs for the DCN datasets. * represents the number after timeout (>24h), and $^\#$ represents the number after running out of memory (>32GB).

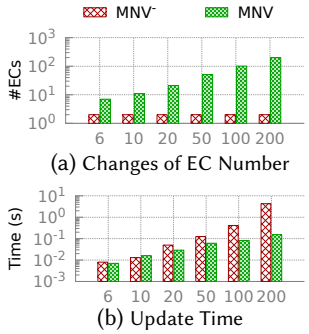


Fig. 8. Changes of EC number vs running time.

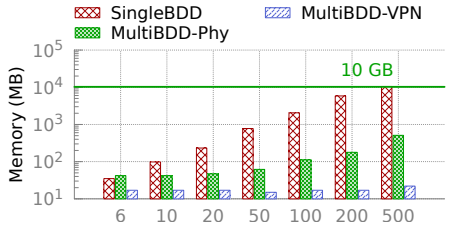


Fig. 9. The memory usage of using single BDD and multiple BDD.

Fig. 7 shows the running time of MNV and Katra^{self} to verify the Multi-IP network. MNV is faster than Katra^{self}, especially when the number of nodes per layer becomes larger. For instance, MNV is nearly 20× faster than Katra^{self}. This demonstrates that MNV does not longer scale well with the number of nodes for the two-layer DCNs, but also can scale well with the number of layers.

6.4 Number of Equivalence Classes

The reason that MNV can scale to large networks, in contrast to existing tools, is mainly owing to the significant reduction of ECs. To confirm this, we count the number of ECs computed by MNV and the other three tools for the DCN datasets. As shown in Table 2, Katra^{self}, reduces the number of ECs compared to APT$^+$ and APKeep$^+$. However, it still computes over 260,000 ECs for DCN consisting of 20 leaf nodes and then runs out of memory beyond that size. In contrast, MNV only computes 425 ECs on 20Leaf, a 99.83% reduction compared with Katra^{self}. The above results confirm that by decomposing large networks into smaller modules, MNV can significantly reduce the number of ECs, making it scale to large compositional networks.

6.5 Reduced update scope

Besides the reduction in the number of ECs, another factor for the speedup of MNV is the reduction of update scope. To show this, we build **MNV$^-$**, which is the same with MNV except it does not decompose the overlay network, e.g., modeling all VPNs with a single module. As shown in Fig. 8, MNV$^-$ computes fewer ECs compared to MNV. The reason is that MNV$^-$ computes a single set of ECs for the overlay, whereas MNV computes ECs for each VPC separately, and some ECs may be computed multiple times by different VPCs. However, MNV is still faster than MNV$^-$. This can be attributed to the fact that MNV$^-$ needs to update hundreds or even thousands of nodes during the update, while MNV only needs to update nodes within a much smaller scope (tens of nodes).

6.6 Distributed MNV

The current prototype of MNV runs on a single machine but has the potential to run in a distributed way so as to scale to even larger networks. The key challenge is the underlying data structure, i.e., Binary Decision Diagram (BDD), which consumes most of the memory but has no mature distributed version up to now. Therefore, we implement the approach in §4.7, such that each module maintains its own BDD, with an additional component responsible for translating the ECs of different modules. As shown in Fig. 9, the memory usage for each module is reduced significantly.

7 DISCUSSION

Beyond reachability properties. To verify properties like load balancing, multi-path consistency, etc., MNV should compute path-related properties. MNV extends forwarding graph to let each link also include attributes, i.e., the label becomes a tuple $\langle EC, attribute \rangle$. Each attribute is a value specific to a property. For example, if the property is there exist n equal-cost paths from s to t, then the attribute of a link is an integer denoting the number of paths carried on that link, and the property is a 4-tuple $\langle src, dst, pkt, attribute \rangle$, where the attribute represents the reachable path between s and t for packets pkt. Different operations would be defined for the attributes, depending on the properties of interest. The design is left as future work.

8 CONCLUSION

This paper presented MNV, a modular data plane verifier for compositional networks. MNV used a new decompose-merge reasoning method, which first decomposed networks into self-contained modules, and then recursively merged the local verification results of modules, in a bottom-up way. With decompose-merge reasoning, MNV can significantly reduce the number of equivalence classes, thereby achieving a better scalability, in compositional networks. Experiments with datasets synthesized based on real DCNs showed that MNV was more than 100x faster than existing verifiers.

Acknowledgement. We would like to thank all the anonymous CoNEXT reviewers for their valuable comments. We also thank Hao Tang and Lizhao You for their help on the datasets. This work is supported by NSFC (No. 62272382 and No. 62172323).

REFERENCES

[1] Timothy Alberdingk Thijm, Ryan Beckett, Aarti Gupta, and David Walker. 2023. Modular Control Plane Verification via Temporal Invariants. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 50–75.

[2] Amazon. [n. d.]. Amazon EC2 secure and resizable compute capacity for virtually any workload. https://aws.amazon.com/ec2/.

[3] Henrik Reif Andersen. 1997. An introduction to binary decision diagrams. *Lecture notes, available online, IT University of Copenhagen* (1997), 5.

[4] Ryan Beckett and Aarti Gupta. 2022. Katra: Realtime Verification for Multilayer Networks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. 617–634.

[5] Ryan Beckett and Ratul Mahajan. 2020. A general framework for compositional network modeling. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*. 8–15.

[6] D Black, Jon Hudson, Larry Kreeger, Marc Lasserre, and T Narten. 2016. *An Architecture for Data-Center Network Virtualization over Layer 3 (NVO3).* Technical Report.

[7] Cisco. [n. d.]. BGP EVPN VXLAN Configuration Guide, Cisco IOS XE Bengaluru 17.6.x (Catalyst 9400 Switches). https://www.cisco.com/c/en/us/td/docs/switches/lan/catalyst9400/software/release/17-6/configuration_guide/vxlan/b_176_bgp_evpn_vxlan_9400_cg/bgp_evpn_vxlan_overview.html?dtid=osscdc000283.

[8] Clarence Filsfils, Pablo Camarillo, John Leddy, Daniel Voyer, Satoru Matsushima, and Zhenbin Li. 2021. Segment Routing over IPv6 (SRv6) Network Programming. RFC 8986. https://doi.org/10.17487/RFC8986

[9] Harold N Gabow and Robert Endre Tarjan. 1983. A linear-time algorithm for a special case of disjoint set union. In *Proceedings of the fifteenth annual ACM symposium on Theory of computing*. 246–251.

[10] Pankaj Garg and Y Wang. 2015. *NVGRE: Network virtualization using generic routing encapsulation.* Technical Report.

[11] Dimitra Giannakopoulou, Kedar S Namjoshi, and Corina S Păsăreanu. 2018. Compositional reasoning. *Handbook of Model Checking* (2018), 345–383.

[12] Google. [n. d.]. Google cloud: Cloud computing services. https://cloud.google.com/.

[13] Albert Greenberg, James R Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A Maltz, Parveen Patel, and Sudipta Sengupta. 2009. VL2: A scalable and flexible data center network. In *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*. 51–62.

[14] J Gross, T Sridhar, P Garg, C Wright, and I Ganga. 2016. Geneve: Generic network virtualization encapsulation. IETF draft.

[15] Thomas A Henzinger, Shaz Qadeer, and Sriram K Rajamani. 1998. You assume, we guarantee: Methodology and case studies. In *Computer Aided Verification: 10th International Conference, CAV'98 Vancouver, BC, Canada, June 28–July 2, 1998 Proceedings 10*. Springer, 440–451.

[16] Alex Horn, Ali Kheradmand, and Mukul Prasad. 2017. Delta-net: Real-time network verification using atoms. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 735–749.

[17] Huawei. [n. d.]. Understanding Microsegmentation. https://support.huawei.com/enterprise/en/doc/EDOC1100198635/8f515dd/understanding-microsegmentation.

[18] Karthick Jayaraman, Nikolaj Bjørner, Jitu Padhye, Amar Agrawal, Ashish Bhargava, Paul-Andre C Bissonnette, Shane Foster, Andrew Helwer, Mark Kasten, Ivan Lee, et al. 2019. Validating datacenters at scale. In *Proceedings of the ACM Special Interest Group on Data Communication*. 200–213.

[19] Cliff B Jones. 1983. Specification and design of (parallel) programs. In *9th IFIP World Computer Congress (Information Processing 83)*. Newcastle University.

[20] Peyman Kazemian, George Varghese, and Nick McKeown. 2012. Header space analysis: Static checking for networks. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. 113–126.

[21] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P Brighten Godfrey. 2013. VeriFlow: Verifying Network-Wide Invariants in Real Time. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. 15–27.

[22] Petr Lapukhov, Ariff Premji, and Jon Mitchell. 2016. Use of BGP for routing in large-scale data centers. *Internet Requests for Comments RFC Editor RFC* 7938 (2016).

[23] Mallik Mahalingam, Dinesh Dutt, Kenneth Duda, Puneet Agarwal, Lawrence Kreeger, T Sridhar, Mike Bursell, and Chris Wright. 2014. *Virtual extensible local area network (VXLAN): A framework for overlaying virtualized layer 2 networks over layer 3 networks*. Technical Report.

[24] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P Brighten Godfrey, and Samuel Talmadge King. 2011. Debugging the data plane with anteater. *ACM SIGCOMM Computer Communication Review* 41, 4 (2011), 290–301.

[25] Microsoft. [n. d.]. Microsoft azure: Cloud computing services. https://azure.microsoft.com/en-us/.

[26] Charles Perkins. 1996. *IP encapsulation within IP*. Technical Report.

[27] Amir Pnueli. 1985. *In transition from global to modular temporal reasoning about programs*. Springer.

[28] E. Rosen and Y. Rekhter. [n. d.]. RFC 4364: BGP/MPLS IP Virtual Private Networks (VPNs). https://datatracker.ietf.org/doc/html/rfc4364.

[29] Ali Sajassi, J Drake, N Bitar, R Shekhar, J Uttaro, and W Henderickx. 2018. *A network virtualization overlay solution using ethernet VPN (eVPN)*. Technical Report.

[30] Deszo Sima, Terence Fountain, and Peter Kacsuk. [n. d.]. Advanced Computer Architectures, a design space approach, 1997.

[31] Alan Tang, Ryan Beckett, Steven Benaloh, Karthick Jayaraman, Tejas Patil, Todd Millstein, and George Varghese. 2023. Lightyear: Using modularity to scale bgp control plane verification. In *Proceedings of the ACM SIGCOMM 2023 Conference*. 94–107.

[32] Timothy Alberdingk Thijm, Ryan Beckett, Aarti Gupta, and David Walker. 2022. Kirigami, the verifiable art of network cutting. In *2022 IEEE 30th International Conference on Network Protocols (ICNP)*. IEEE, 1–12.

[33] Arash Vahidi. [n. d.]. JDD, a pure Java BDD and Z-BDD library. https://bitbucket.org/vahidi/jdd.

[34] Yushun Wang. 2017. *Tenantguard: Scalable runtime verification of cloud-wide vm-level network isolation*. Ph. D. Dissertation. Concordia University.

[35] Hongkun Yang and Simon S Lam. [n. d.]. Scalable Network Verification Using Atomic Predicates. https://www.cs.utexas.edu/users/lam/NRL/Atomic_Predicates_Verifiers.html.

[36] Hongkun Yang and Simon S Lam. 2015. Real-time verification of network properties using atomic predicates. *IEEE/ACM Transactions on Networking* 24, 2 (2015), 887–900.

[37] Hongkun Yang and Simon S Lam. 2017. Scalable verification of networks with packet transformers using atomic predicates. *IEEE/ACM Transactions on Networking* 25, 5 (2017), 2900–2915.

[38] Lizhao You, Jiahua Zhang, Yili Jin, Hao Tang, and Xiao Li. 2022. Fast Configuration Change Impact Analysis for Network Overlay Data Center Networks. *IEEE/ACM Transactions on Networking* 30, 01 (2022), 423–436.

[39] Pamela Zave and Jennifer Rexford. 2019. The compositional architecture of the Internet. *Commun. ACM* 62, 3 (2019), 78–87.

[40] Pamela Zave, Jennifer Rexford, and John Sonchack. 2020. The Remaining Improbable: Toward Verifiable Network Services. *arXiv preprint arXiv:2009.12861* (2020).

[41] Peng Zhang, Xu Liu, Hongkun Yang, Ning Kang, Zhengchang Gu, and Hao Li. 2020. {APKeep}: Realtime Verification for Real Networks. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 241–255.

A CORRECTNESS PROOF

In order to prove our correctness, we define the forwarding equivalent (fwd-equivalent) between one abstract model and the monolithic model. Intuitively, two models are "equivalent" with respect to a packet if they can "equally" forward it from one node to another.

Definition A.1. **Fwd-equivalent.** Given a packet pkt, a virtual rule r, and a sequence of physical rules R_{phy}. We say r is fwd-equivalent with R_{phy} with respect to the packet pkt, denoted as $r \equiv_{pkt} R_{phy}$ if and only if r and R_{phy} forwards pkt from the same source to the same destination.

This definition is used to evaluate the equivalence between one rule and a sequence of physical rules, and can be extended to evaluate the equivalence between two sequences of rules.

Definition A.2. **Extended fwd-equivalent.** Given a packet pkt and a sequence of rules $R = \{r_1, r_2, \ldots, r_n\}$ that forwards pkt from some node s to some other node t in the abstract model, and the sequence of physical rules $R_{phy} = \{h_1, h_2, \ldots, h_m\}$ that forwards packet pkt from s to t in the monolithic model. R and R_{phy} are fwd-equivalent if and only if there is a partition of R_{phy}, denoted as $\hat{R}_{phy} = \{\hat{r}_1 = \{h_1, \ldots h_k\}, \hat{r}_2 = \{h_{k+1}, \ldots, h_{k+q}\}, \ldots, \hat{r}_p = \{h_{k+q+l}, \ldots, h_m\}\}$, and the following two conditions hold: (1) $|R| = |\hat{R}_{phy}|$; (2) $\forall i \in 1 \ldots n, r_i \equiv_{pkt} \hat{r}_i$

Thus, we can check if all the sequence of rules found in the local verification is fwd-equivalent with some sequence of rules in the monolithic model (i.e., soundness), and vice-versa (i.e., completeness). We will prove this by mathematical induction on the hierarchical level of the modules.

Definition A.3. **Hierarchical level.** Given a module m, the hierarchical level (HL) of this model is a nature number n that satisfies the following conditions:

$$HL(m) = n = \begin{cases} 0 & \text{if } m \text{ depends on no modules} \\ max_{m_i \in impl(m)} HL(m_i) + 1 & \text{otherwise} \end{cases}$$

Theorem A.4. ***Soundness.*** *For any module in MNV, if a reachability property $\langle s, t, pkts \rangle$ holds locally in module m, then for any packet $pkt \in pkts$, it can be forwarded from s to t in the monolithic model.*

Proof. Let m be an arbitrary module, and $\langle s, t, pkts \rangle$ be an arbitrary property held locally. Pick an arbitrary sequence of rules R that realizes this reachability. We will prove by mathematical induction on the hierarchical level n of the module m.

Case (n=0): Evaluate R is the sequence of physical rules that forwards pkt from s to t in the monolithic model since there are no virtual rules in R.

Case k+1: To show that pkt can be forwarded from s to t in the monolithic model, we must show that there is a sequence of physical rules R_{phy} that is fwd-equivalent with R. There are two cases to consider.

(1) The first is that m is a component module. This means that the virtual rules in R are derived from the layering relationship. Pick any virtual rule $r \in R$, where $r = @x : f_V(S) \mapsto permit$, $pkt \in S$, and (x, y) is the edge r applied on. According to the Definition 4.2. We know r is derived from some encapsulation rule r_{enc} and the reachability property $\langle x', y', S' \rangle$ in the module with

the hierarchical level at most of k. From the inductive hypothesis, we know there is a sequence of physical rules R'_{phy} that forwards S' from x' to y' in the monolithic model. Let r_{denc} be the decapsulation rule of r_{enc}, then we have the sequence of rules $\hat{r} = \{r_{enc}, R'_{phy}, r_{denc}\}$ and $r \equiv_{pkt} \hat{r}$.

We repeatedly replace the virtual rule r in R with the corresponding sequence of rules $\hat{r}$, we get a replaced sequence of rules $\hat{R}$, which is the sequence of physical rules that fwd-equivalent with R.

(2) The second case is where m is a bridge module. This means that the virtual rules in R are derived from the bridging relationship. Pick any virtual rule $r \in R$, where $r = @x : f_V(S) \mapsto fwd\ y$ and $pky \in S$. Similarly, we know from the inductive hypothesis that there is a sequence of physical rules $\hat{r}$ that forwards S from x to y in the monolithic model, and $r \equiv_{pkt} \hat{r}$. Evaluate $\hat{R}$ is the fwd-equivalent sequence of physical rules by repeatedly replacing the virtual rule r in R with $\hat{r}$.

In either case, we see that R is fwd-equivalent with some sequence of physical rules in the monolithic model, completing the induction. □

Theorem A.5. ***Completeness.*** *For any packet pkt, if it can be forwarded from s to t in the monolithic model, then there exists one and only one module m in MNV, satisfying that reachability $\langle s, t, pkts\rangle$ holds locally in module m and $pkt \in pkts$.*

Proof. Let s and t be two arbitrary nodes, there are two cases to consider:

(1) The first is that s and t belong to the same component. By the bijective relationship between the component and the module, we know there is only one module that contains s and t.

(2) The second case is where s and t belong to different components. According to the definition of the bridge module, that defines the bridge module as the connected component of components, we know there is only one bridge module that contains both s and t.

Either way, we find only one module, denoted as m, that contains s and t. Let R_{phy} be the sequence of physical rules that forward pkt from s to t in the monolithic model, we will prove by mathematical induction on the hierarchical level n that m will locally hold a property $\langle s, t, pkts\rangle$ and $pkt \in pkts$.

Case (n=0): Evaluate R_{phy} is the sequence of rules that realizes the reachability property since there is no encapsulation in R_{phy}.

Case k+1: To show that the property locally holds in m, we must show that there is a sequence of rules R in m that is fwd-equivalent with R_{phy}. Similarly with the proof of Theorem A.4, we can find such R from the inductive hypothesis by repeatedly replacing the subsequence of rules $\hat{r} \subseteq R_{phy}$ with a virtual rule r in m, where $r \equiv_{pkt} \hat{r}$, completing the induction. □

B INCREMENTAL UPDATE ALGORITHM

Consider the scenario that one module has rule insertion, deletion, or modification for physical rules, making some nodes update the forwarding behaviors. Such network updates could possibly change the reachabilities of this module and the modules depending on it. MNV propagates the changes of properties between modules by updating the corresponding virtual rules. Specifically, given a module m_i and the set of modules it depends on $Impl(m_i)$, MNV propagates the changes from $Impl(m_i)$ into m_i as follows: (1) If the reachability between some nodes (s, t) are changed in some module $m_j \in Impl(m_i)$, MNV identifies the virtual rule r that derived from the previous reachability, and computes a new virtual rule r' from the changed reachability according to Definition 4.1 or 4.2. If r and r' are different, MNV creates a virtual rule deletion and insertion for m_i, denoted as $\{-r, +r'\}$. (2) MNV continually performs step (1) until the virtual rule updates are computed from all the changed reachabilities, then MNV computes the reachability changes of m_i by applying the virtual rule updates into the incremental local verification of m_i. Finally, the reachability changes of m_i will be propagated into other modules if they depend on m_i.

Algorithm 1: `TranslateBDD`($bdd_1, BDDEng_1, BDDEng_2$)

Input: bdd_1: the BDD to be translated; $BDDEng_1$: the original BDD engine; $BDDEng_2$: the target BDD engine.
Output: bdd_2: the translated BDD.

```
1  amap(0) ← 0;
2  amap(1) ← 1;
3  RecursiveTranslate (bdd_1);
4  return amap(bdd_1);
5  Function RecursiveTranslate(bdd):
6      if bdd < 2 then
7          return;
8      if amap(bdd) = Null then
9          var ← BDDEng_1.getVar (bdd);
10         low ← BDDEng_1.getLow (bdd);
11         high ← BDDEng_1.getHigh (bdd);
12         RecursiveTranslate(low);
13         RecursiveTranslate(high);
14         low ← amap(low);
15         high ← amap(high);
16         amap(bdd) ← BDDEng_2.mk (var, low, high);
```

C BDD TRANSLATION ALGORITHM

Algorithm 1 summarizes the process of translation. This algorithm uses a map to record the corresponding relation between BDDs. Initially, MNV records the mapped value of 0 and 1 as itself (Line 1-2). Then, it uses a recursive function to translate BDD (Line 3). The function first checks if the BDD is lesser than 2, which means the BDD is `True` (1) or `False` (0) (Line 5-6). If so, it terminates (Line 7). Otherwise, it processes the further translation if the current BDD is not on the map. In detail, MNV creates new BDDs in the target BDD engine, using the branch relationship in the original BDD engine (Line 8-16). In implementation, the BDD translation is used to transform the reachability in global BDD, and to transform the virtual rule after the application of inverse encapsulation rules into overlay BDD.

D SUPPORTING DETERMINISTIC ENCAPSULATION

Suppose the Permutation is defined as:

$$p.Perm_{(x_1,x_2,x_3)\rightarrow(y_1,y_2,y_3)} = p|_{x_1=y_1,x_2=y_2,x_3=y_3}$$

where x_1, x_2, x_3 are Boolean variables of Boolean formula p, y_1, y_2, y_3 are other variables. Permutation replace the x_1, x_2, x_3 with y_1, y_2, y_3 respectively. Based on Permutation, the deterministic encapsulation rule $f_{vdst}(10.1.x.y) \mapsto \phi_{dst}(192.168.x.y)$ should be defined as:

$$f_{vdst}(10.1/16) \mapsto \phi_{dst}(192.168/16) \wedge Perm_{vdst[17,32]\rightarrow dst[17-32]}$$

, where the $V[17-32]$ denotes the (17-32)-th bits of variables V. Consequently, the reverse encapsulation rule is defined as:

$$f_{dst}(192.168/16) \mapsto \phi_{vdst}(10.1/16) \wedge Perm_{dst[17,32]\rightarrow vdst[17-32]}$$

The truth is that the basic idea of using the reverse encapsulation rule is correct, while the actual reversing process varies from the types of encapsulation rule.

With this definition, we can define a general framework: construct a resolver to parse the encapsulation rules from configurations, determine which type of encapsulation rules the device uses according to the device vendor, then we create corresponding reverse encapsulation rule and apply it in virtual rule computation.

Received July 2023; revised September 2023; accepted October 2023