Scalable Simulation-based Configuration Verification of DCNs via Destination-Independent Compression

Mengrui Zhang[⋆], Xiaoqiang Zheng[⋆], Letian Zhu[⋆], Xing Fang[⋆], Lizhao You[⋆†], Zhang Liu[⋆], Ziyang Yao[♢], Yang Wang[♢], Rui Wen[♢], Zhi Zhang[♢], Ronghua Sun[♢], Yuanhui Zhong[♢], Haihua Li[♢], Fei Yuan[⋆], Qiao Xiang[⋆]
[⋆]School of Informatics, Xiamen University, China [†] Shenzhen Research Institute of Xiamen University, China
[♢]Huawei Technologies Co. Ltd., China

***Abstract*—Modern cloud data center networks (DCNs) often span tens of thousands of switches, challenging configuration verification, even for the state-of-the-art simulation-based configuration verification tools. While prior work has accelerated individual stages of the verification pipeline, these fragmented optimizations fall short of addressing the end-to-end scalability bottlenecks inherent in hyper-scale networks. To address this, we present *sNetVeri*, a scalable simulation-based configuration verifier for hyper-scale DCNs, built on the key insight that networks are often highly structured and compressible. Unlike prior compression methods tied to specific destinations, *sNetVeri* introduces a destination-independent one-shot compression strategy with formal correctness guarantees. *sNetVeri* further improves scalability through two designs: (1) a Dijkstra-based BGP simulation algorithm augmented with a withdrawal mechanism to support both monotonic and non-monotonic configurations, and (2) a highly parallel data-plane verification framework that operates directly on the shared compressed topology. Extensive evaluations on both production DCNs from Huawei Cloud and synthetic topologies demonstrate that *sNetVeri* scales efficiently to networks with over 10,000 switches while maintaining accuracy in control-plane simulation and data-plane verification.**

I. Introduction

Data center networks (DCNs) underpin modern cloud services, where even minor configuration errors can trigger large-scale outages and substantial financial losses. To mitigate such risks, configuration verifiers [1]–[16] are widely used to validate network configurations before deployment. By comparing realized network behavior against operator intents (i.e., expected behavior), these tools help identify anomalies early and improve operational reliability. However, as production DCNs have grown to tens of thousands of switches, existing configuration verification tools increasingly struggle to scale to hyper-scale topologies.

Inefficiency of Existing Tools. Existing verifiers broadly fall into two categories. The first is *SMT-based* configuration verification [1]–[3], which directly abstracts routing configurations into logical models and leverages SMT-style solvers to prove properties. Minesweeper [1] pioneered this direction, but its scalability is fundamentally limited by the complexity of the network features and the network size. Subsequent systems such as ACORN [2] and ShapeShifter [3] improve scalability via abstractions over paths or route attributes; however, these abstractions often over-approximate network behavior and may introduce false positives. In addition, the one-shot workflow typically does not materialize intermediate routing/forwarding state (e.g., RIB/FIB), making results less transparent and less aligned with operators' troubleshooting practice.

The second category is *simulation-based* configuration verification [4]–[7], which first simulates the control plane to compute routing/forwarding tables and then checks data-plane properties on the derived forwarding state. This workflow is widely adopted in practice [8], [9], due to its interpretability and operational friendliness. Nevertheless, simulation-based verifiers also face severe scaling bottlenecks in hyper-scale DCNs: both control-plane simulation and subsequent data-plane verification can become prohibitively expensive as network size and configuration complexity increase.

Prior work has improved different stages of the simulation-based pipeline. Several systems accelerate control-plane simulation via parallelization of routing convergence [8] or deterministic algorithms under restricted configuration classes [10]. Other systems [11], [12] accelerate data-plane verification using fine-grained equivalence classes (ECs) and efficient EC maintenance. Despite these advances, achieving hyper-scale verification remains challenging on a single machine. Recent efforts such as S2 [13] and new Hoyan [14] scale beyond 10K devices by distributing computation across multiple servers; while effective, these multi-server deployment introduces additional operational cost and complexity, and falls short of fully exploiting the computational capacity of a single server.

Our Approach. In this paper, we tackle the challenge of scaling *simulation-based* configuration verification for hyper-scale DCNs on a *single server*. The core difficulty stems from the sheer number of devices and links in modern networks. Our key idea is to exploit the structural regularity that is common by design in DCNs (e.g., redundancy and repeated roles) to *compress* the network and perform verification on the compressed topology.

Different from prior compression techniques that are constructed *per-destination* (e.g., Bonsai [15] and

Lizhao You and Qiao Xiang are co-corresponding authors. This work is supported in part by the National Natural Science Foundation of China (Grant Nos. 62532010, 62572408, and 62302409), the National Key R&D Program of China (Grant No. 2024YFB2906900), the Fujian Provincial Science and Technology (Grant No. 2025H6021), the Shenzhen Science and Technology Program (Grant No. JCYJ20250604123034006), and the Key Project for Technological Innovation and Industrialization in the Software Industry of Fujian Province (Project "Research, Development, and Industrial Application of Key Technological Innovations for Intelligent Hybrid Cloud Conferencing Software Based on a Domestic Software and Hardware Ecosystem").

979-8-3195-1703-6/26/$31.00 ©2026 IEEE

symmetry/surgery-based DPV [16]), we seek a *one-shot* compression that can be reused across *multi-destination* properties (e.g., all-pairs reachability) without rebuilding an abstraction for each destination. Figure 1 shows an example of compressing a Fattree topology. Moreover, despite strong topological symmetry, practical DCNs often contain *local* configuration differences, such as per-ToR AS numbers (§III). These devices are locally distinct but still share the same global role and policy structure. To capture this phenomenon, we introduce *parameterized equivalence*: two devices are parameterized-equivalent if their forwarding-relevant configuration is identical up to a finite set of local parameters (e.g., AS number), which can be recorded as *route context* during simulation and later used to reconstruct exact per-device routing/forwarding tables. Parameterized equivalence enables compressing beyond strict syntactic/semantic equivalence while preserving verification accuracy through lossless reconstruction.

Building on these insights, we present *sNetVeri*, a scalable simulation-based configuration verifier for BGP-based hyper-scale DCNs. While *sNetVeri* is primarily designed for DCNs, its core verification framework and compression mechanisms are fundamentally generic, making it readily applicable to Wide Area Networks (WANs). It introduces three key designs.

D1: One-shot Network Compression (§III). We propose a destination-independent one-shot compression technique that preserves forwarding behavior between the compressed and original networks. To handle parameterized-equivalent devices, we introduce a new route representation that carries both *local-device* and *remote-device* context for each advertisement, including the relevant local parameters. This enables compressing such devices while still supporting accurate simulation on the compressed topology. After simulation, *sNetVeri* reconstructs exact routing/forwarding state for each device using the retained context. We formulate compression using the routing algebra and provide formal correctness guarantees.

D2: Accelerated Simulation on Compressed Networks (§IV). On the compressed topology, *sNetVeri* performs control-plane simulation with substantially reduced problem size. In addition, we adopt a Dijkstra-based deterministic route computation (inspired by FastPlane [10]) to avoid the slow convergence of traditional Bellman–Ford iterations. However, Dijkstra-based computation assumes monotonic configurations, which may not hold in practice (especially in transit area of DCNs). We therefore augment the Dijkstra algorithm with a lightweight withdrawal mechanism that guarantees correctness under non-monotonic configurations, enabling fast route computation while remaining compatible with real-world deployments.

D3: Accelerated DPV on Compressed Networks (§V). To efficiently process many-to-many queries (e.g., all-pairs reachability), we design and implement an accelerated data-plane verification (DPV) framework over a shared compressed network topology. Specifically, we partition the overall verification workload into multiple orthogonal sub-networks (subtasks) and employ a Breadth-First Search (BFS) algorithm to traverse each sub-network. Given the inherent nature of compressed networks, the algorithm permits repeated visits to compressed nodes during the BFS traversal. To resolve the resulting state ambiguity, we introduce a *compressed-device* context mechanism. This mechanism precisely tracks and differentiates the data forwarding states of the underlying physical devices, ultimately computing the final and accurate verification results.

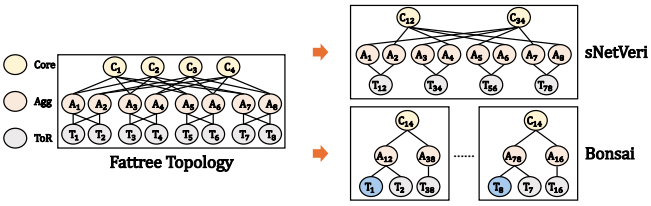


Fig. 1: Bonsai compression versus ours on a Fattree topology under the default configuration using shortest AS-path routing.

Evaluation (§VI). We extensively evaluate *sNetVeri* on a Huawei Cloud DCN with over 10,000 nodes, synthetic Fattree topologies (from Fattree 40 to Fattree 90) and synthetic WAN topologies with the following results: 1) *sNetVeri* scales to hyper-scale Huawei Cloud DCN, completing full-network simulation in 74.05s with 13,807.92 MB memory usage and verifying all-pair reachability in 420.37s with 50,071.09 MB memory usage, while other tools fail to do so; 2) *sNetVeri* achieves $42\times$ to $336\times$ speedup over a state-of-the-art (SOTA) simulation tool, Bonsai compression [15] combined with Batfish [9], for Fattree topologies; 3) *sNetVeri* outperforms the SOTA data-plane verification tools, APKeep [11] and Flash [12], by up to $993\times$ and $364\times$, respectively, in verifying all-pair reachability for Fattree topologies; and 4) *sNetVeri* outperforms the SOTA end-to-end verification tool, Bonsai compression [15] combined Batfish [9], by up to $136\times$.

To facilitate future research, we make our implementation publicly available in https://github.com/sngroup-xmu/sNetVeri.

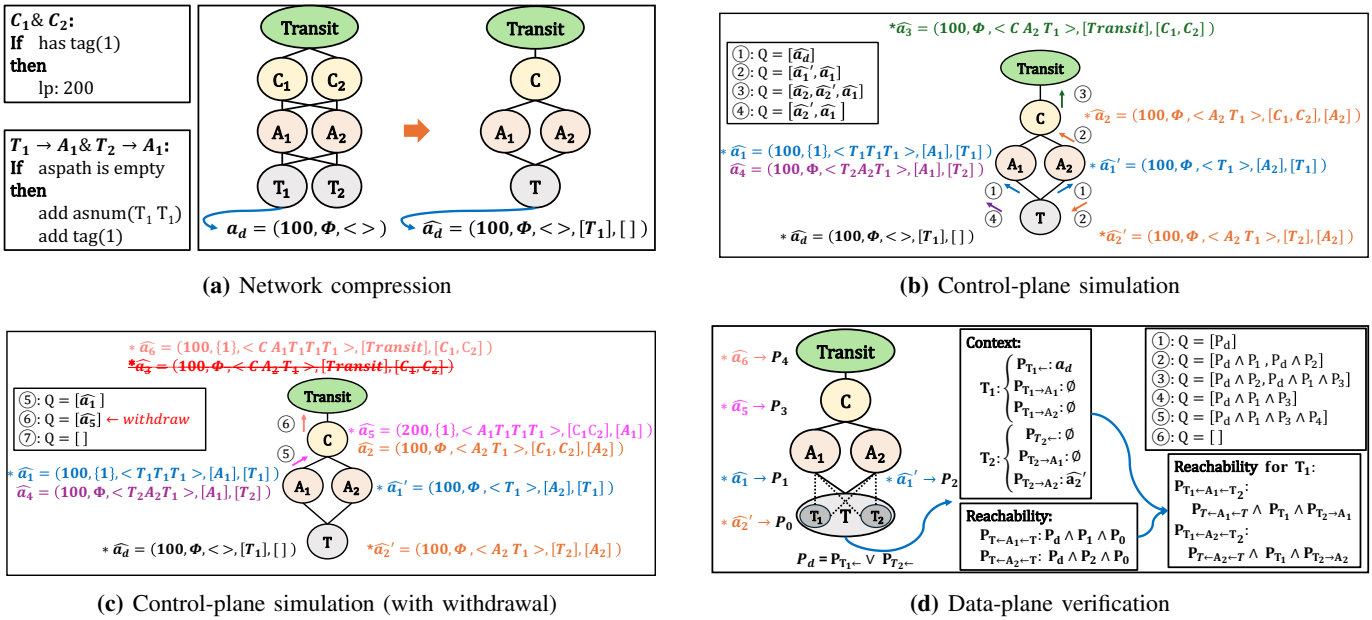
II. System Overview

This section introduces the basic concepts behind *sNetVeri* and walks through its workflow using a Border Gateway Protocol (BGP) example.

A. Basic Concepts

Border Gateway Protocol. We focus on BGP because it is both widely deployed and policy-rich, making it a representative (and challenging) control-plane for verification. In BGP, routers apply policy-based selection and propagation rules to determine which routes to advertise. A BGP route includes a destination prefix, an AS path, a local preference, and optional attributes (e.g., communities). Unlike link-state or distance-vector protocols, BGP carries explicit path information: the AS path enables routers to detect loops by checking the same AS numbers. Currently, we focus on eBGP. The extension to other routing protocols (e.g., iBGP and OSPF) can be found in the appendix [17].

Network Model. We model a network as a directed graph: nodes are switches, and edges are communication links. The control plane consists of router-local rules that govern how



(a) Network compression

(b) Control-plane simulation

(c) Control-plane simulation (with withdrawal)

(d) Data-plane verification

Fig. 2: An example network with simplified BGP configurations illustrating the *sNetVeri* workflow. Each node is annotated with the BGP routes it has received: $*$ marks the selected best route, and a crossed-out route indicates a withdrawn route.

routing information is imported, selected, and exported. We abstract each routing information item as a vector of *attributes*, independent of its concrete encoding. Along an edge, routing information is transformed by the policies at the two endpoints; this rule-driven exchange can be captured as a transfer function that maps an incoming attribute vector to an outgoing one.

B. Workflow

The workflow of *sNetVeri* consists of three stages: network compression, control-plane simulation, and data-plane verification. In the compression stage, *sNetVeri* merges devices that have the same neighbors and policies, which we refer to as parameterized-equivalent devices, and introduces a route representation that carries both *local-device* and *remote-device* context to capture subtle forwarding differences within each compressed node. In the simulation stage, *sNetVeri* runs a Dijkstra-based route computation algorithm on the compressed network, augmented with a withdrawal mechanism to handle non-monotonic routing behavior. In the verification stage, *sNetVeri* performs a BFS over the compressed network, records the packet space associated with each reachable path, and intersects the resulting packet spaces for each physical source device to determine reachability for every physical device pair.

Next, we illustrate the workflow of *sNetVeri* using the example network in Figure 2a, which is a small subset of switches from Huawei Cloud DCN. The topology contains two ToR switches T_1 and T_2, two aggregation switches A_1 and A_2, two core switches C_1 and C_2, and one transit switch. Each ToR exports routes to the left aggregation switch using the same special policy: it prepends two additional AS numbers to the AS path and attaches community tag 1. We intentionally prepend two AS numbers to create a non-monotonic configuration pattern, which will be useful for illustrating our control-plane simulation algorithm. Meanwhile, core switches prefer routes tagged with 1. For simplicity, we assume that (i) C_1 and C_2 share the same AS number C, and (ii) all other nodes use distinct AS numbers equal to their device identifiers (e.g., A_1 uses A_1). The workflow proceeds in three steps.

1) Network Compression

In the example, the core switches C_1 and C_2 have identical policies and the same neighbors, so they are merged into a single compressed node C. In contrast, the two aggregation switches A_1 and A_2 apply different export policies to routes learned from the ToRs and therefore cannot be compressed. Similarly, T_1 and T_2 have the same neighbors and nearly identical configurations except for their AS numbers. Although their AS numbers differ, our algorithm still merges them into a single compressed node. As a result, the original topology can be compressed into the one shown on the right of Figure 2a.

In the example, we assume that only T_1 originates a route a_d. For ease of exposition, we visualize a route using three key attributes. For example, $a_d = (100, \Phi, \langle\rangle)$ is a 3-tuple where the first element is the local preference (default 100), the second is the set of community tags (with Φ denoting the empty set), and the third is the AS path. In the compressed network, we augment each compressed route with two additional attributes: *LocalDevices* and *RemoteDevices*. These lists record the physical devices that receive and advertise the route, respectively. Accordingly, a_d is replaced by $\widehat{a}_d$, represented as $\widehat{a}_d = (100, \Phi, \langle\rangle, [T_1], [\,])$, where the fourth field lists the local devices (receivers) and the fifth field lists the remote devices (senders). Since a_d is originated at T_1, the only local device is T_1, and there are no associated remote devices.

2) Control-Plane Simulation

Control-plane simulation is executed on the compressed network, as illustrated in Figures 2b and 2c. Following the

simulation algorithm in Section IV, we maintain a global priority queue Q of candidate routes, ordered by route priority (in this example, local preference is compared first, followed by AS-path length). At each step, we dequeue the highest-priority route in Q and propagate it. For readability, we omit propagations triggered by routes that are ultimately dropped.

① Insert the originated route $\widehat{a}_d$ into Q. Pop $\widehat{a}_d$ and export it to A_1 and A_2, producing $\widehat{a}_1$ and $\widehat{a}'_1$ (where $\widehat{a}_1$ carries tag 1 and has a longer AS path). Each logical route records *LocalDevices* (receivers) and *RemoteDevices* (senders). For example, $\widehat{a}_1$ has *LocalDevices*= $[A_1]$ and *RemoteDevices*= $[T_1]$, meaning that A_1 learns $\widehat{a}_1$ from T_1 and currently selects it.

② Process $\widehat{a}'_1$ first: A_2 exports it to C and T, yielding $\widehat{a}_2$ and $\widehat{a}'_2$. Since the AS path of $\widehat{a}'_1$ contains T_1, node T drops the route at T_1 but accepts it at T_2; therefore, *LocalDevices* of $\widehat{a}'_2$ contains only T_2.

③ C sends $\widehat{a}_2$ to $Transit$, yielding $\widehat{a}_3$.

④ T exports $\widehat{a}'_2$ to A_1, yielding $\widehat{a}_4$. Here, $\widehat{a}_4$ ties with $\widehat{a}_1$. We assume A_1 keeps the first received route $\widehat{a}_1$ as the best route.

⑤ A_1 exports $\widehat{a}_1$ to C, yielding $\widehat{a}_5$. Since C prefers routes with tag 1, $\widehat{a}_5$ becomes the new best route and triggers a withdrawal of the previous best route $\widehat{a}_2$. The withdrawal is applied recursively to all dependent routes; in particular, $\widehat{a}_3$ at $Transit$ is withdrawn. After all outdated dependent routes are removed, we insert $\widehat{a}_5$ into Q.

⑥ C exports $\widehat{a}_5$ to $Transit$, yielding $\widehat{a}_6$.

⑦ Q becomes empty, and route propagation terminates.

3) Data-Plane Verification

Data-plane verification is performed on the compressed network, as illustrated in Figure 2d. Following the verification algorithm in Section V, ***sNetVeri*** first encodes the packet space of each logical node as Binary Decision Diagrams (BDDs), with one BDD for each output port. For example, route $\widehat{a}_1$ is encoded as packet space P_1 associated with port $\langle A_1 \to T \rangle$; for a destination node, ***sNetVeri*** encodes its destination packet space as P_d, which is the union of the destination spaces of T_1 and T_2. For each queried target node, ***sNetVeri*** further separates packet spaces by physical device. For example, to check reachability from T_2 to T_1, ***sNetVeri*** separately encodes the relevant device-level spaces: T_1 has destination space $P_{T_1\leftarrow}$, while T_2 has packet space $P_{T_2\to A_2}$, representing the packets that T_2 can forward to A_2, and an empty destination space $P_{T_2\leftarrow}$ because the destination route $\widehat{a}_d$ is originated only by T_1. ***sNetVeri*** then maintains a priority queue Q of reachable packet spaces and performs BFS over the compressed network.

① Insert the destination space P_d into Q. After popping P_d, ***sNetVeri*** traverses the neighbors of T, A_1 and A_2, and intersects P_d with the corresponding per-port packet spaces P_1 and P_2. Since both intersections are non-empty, $P_d \wedge P_1$ and $P_d \wedge P_2$ are inserted into Q.

② Pop $P_d \wedge P_1$ and traverse the neighbors of A_1, T and C. For Node T, ***sNetVeri*** obtains $P_d \wedge P_1 \wedge P_0$ and records it at T as reachable packet space on path $< T \leftarrow A_1 \leftarrow T >$. Since T is the target node, this packet space is not inserted into Q. For C, ***sNetVeri*** obtains $P_d \wedge P_1 \wedge P_3$; because it is non-empty, it is inserted into Q.

③ Pop $P_d \wedge P_2$ and traverse the neighbors of A_2, T and C. The traversal to T is handled similarly, so the reachable packet space recorded at T on path $< T \leftarrow A_2 \leftarrow T >$ becomes $(P_d \wedge P_2 \wedge P_0)$. For C, the intersection is empty because P_3 only corresponds to port $\langle C \to A_1 \rangle$.

④ Pop $P_d \wedge P_1 \wedge P_3$ and traverse the neighbors of C, A_1, A_2, and $Transit$. The packet spaces for ports $\langle A_1 \to C \rangle$ and $\langle A_2 \to C \rangle$ are empty, whereas the traversal to $Transit$ yields $P_d \wedge P_1 \wedge P_3 \wedge P_4$, which is inserted into Q.

⑤ Pop $P_d \wedge P_1 \wedge P_3 \wedge P_4$ and traverse the neighbor of $Transit$, C. The packet space for port $\langle C \to Transit \rangle$ is empty, so no new packet space is inserted.

⑥ Q becomes empty, and the BFS terminates.

For each physical target pair, ***sNetVeri*** intersects each recorded reachable packet space with the destination space and the corresponding per-port packet space of the physical devices. For example, to check whether T_2 can reach T_1, ***sNetVeri*** first considers the path through A_1, whose recorded reachable packet space is $P_{T\leftarrow A_1\leftarrow T}$. It intersects this space with T_1's destination space P_{T_1} and T_2's per-port packet space $P_{T_2\to A_1}$. The result is empty because $P_{T_2\to A_1}$ is empty, indicating that no path from T_2 to T_1 traverses A_1. ***sNetVeri*** then considers the path through A_2, whose recorded reachable packet space is $P_{T\leftarrow A_2\leftarrow T}$. Intersecting it with P_{T_1} and $P_{T_2\to A_2}$ yields a non-empty result, showing that T_2 can reach T_1 through A_2.

III. One-Shot Network Compression

In this section, we present our network compression approach; a detailed correctness proof is provided in the appendix [17].

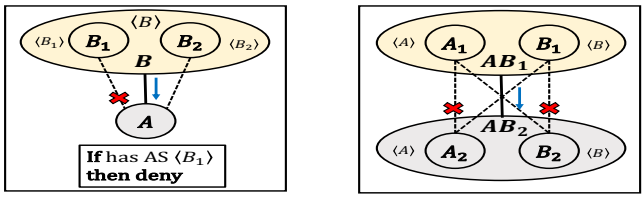
A. Network Model

We adopt the basic network model defined in Bonsai [15].

Stable Routing Problem (SRP). An SRP formally captures the set of all routing behaviors a network can exhibit. An SRP instance is defined as a tuple $\langle G, A, a_d, \prec, trans \rangle$. Here, $G = (V, E, d)$ is a directed graph, where V is the set of nodes (routers), $E \subseteq V \times V$ is the set of directed edges (links), and $d \in V$ is the designated destination node. A is the set of routing attributes, which describe the content of routing messages (*e.g.*, in BGP: local preference, community tags, and AS path). $\perp$ denotes the absence of a valid route (*i.e.*, no routing information). The a_d represents the original route originated by the destination node d. The partial order $\prec$ over A defines how routers compare route attributes to select the best route (e.g., BGP compares local preference, then AS path length, and so on). $trans$ is a transfer function that models how attributes are transformed as they traverse edges. In BGP, this captures the application of export policies and import policies and the loop prevention.

B. Conditions for Compression

By analyzing the network configuration, we enforce the following constraints when compressing nodes and edges:



(a) Example for AS-Constraint 1 **(b)** Example for AS-Constraint 2

Fig. 3: Illustration examples of AS-Constraints

Agg-constraint: All devices within the same logical node must announce the same aggregated prefix.

AS-constraints: (1) If an AS number is explicitly referenced in any routing policy (e.g., AS-path filtering), then all routers merged into the same logical node must share that AS number, and the logical node must be assigned this AS number. This preserves consistent policy enforcement. (2) For any logical edge, at least one of its endpoint logical nodes must be AS-equivalent. This avoids inconsistent behavior caused by partially connected meshes.

Edge-constraint: Each device in a logical node must have the same neighbors, and the export policies for these neighbors and their import polices must also be identical.

We specifically explain the necessity of introducing AS constraints. Figure 3a shows an example of the first AS constraint. Consider B_1 and B_2 with different AS numbers. $\langle B_1 \rangle$ and $\langle B_2 \rangle$ are compressed into a single node B. Suppose node A is configured to deny any routes containing $\langle B_1 \rangle$. Although the *remoteDevices* field of the logical route allows us to inspect whether any device in B used AS number $\langle B_1 \rangle$, doing so complicates route processing and makes the logical AS number meaningless. To avoid this, we require that all routers use the AS number that appears in any routing policy, then it can only be compressed with other routers that share the same AS number, preserving as-equivalence. Thus, B_1 should only be compressed with other routers that share AS number $\langle B_1 \rangle$, and their logical AS number must also be $\langle B_1 \rangle$.

Figure 3b illustrates an example of the second AS constraint. Routers A_1 and B_1 are compressed into node AB_1, A_2 and B_2 are compressed into node AB_2. Suppose A_1 and A_2 share AS number $\langle A \rangle$, and B_1 and B_2 share AS number $\langle B \rangle$. When node AB_2 receives a logical route $\widehat{a}$ from AB_1, A_2 will only add B_1 in $\widehat{a}.remoteDevices$, B_2 will only add A_1 in $\widehat{a}.remoteDevices$. However, $\widehat{a}.localDevice = [A_2, B_2]$ and $\widehat{a}.remoteDevices = [A_1, B_1]$—implying a full-mesh exchange. To ensure consistency in such scenarios, we require that at least one of the two logical nodes (*i.e.*, either AB_1 or AB_2) maintains as-equivalence. This guarantees that all devices on one side either uniformly accept or drop the route.

C. Compression Algorithms

We first describe how *sNetVeri* compresses nodes and edges. We then present the solution algorithm, which computes the best-route *set* for each logical node (since different physical devices within the same logical node may select different best routes). Finally, we introduce the transfer algorithm, which correctly models route propagation and loop prevention in the presence of heterogeneous physical AS numbers.

1) Network Compression Algorithm

Our compression pipeline has two stages: *grouping* and *compression*. In the grouping stage, the algorithm orders nodes by degree and groups physical nodes subject to the constraints in §III-B, proceeding layer by layer from leaves toward the root. In the compression stage, it merges devices into logical nodes according to the grouping results while preserving interface and destination information, and it merges all edges that connect the same pair of groups into a single logical edge.

2) Solution Algorithm

For BGP, the solution algorithm computes the best-route set for each logical node v in the compressed network. It maintains (i) a device set D, which tracks the physical devices in v that have already been assigned a best route, and (ii) a result set R, which stores the selected logical routes. The candidate routes in $rib(v)$ are first sorted according to the comparison relation $\prec$. For each candidate route a_i, the algorithm computes the uncovered devices $D_i = a_i.localDevices \setminus D$. If D_i is non-empty (i.e., a_i is the best route for at least one additional device), the algorithm adds a_i to R and updates $D \leftarrow D \cup D_i$. The procedure terminates once every physical device in v is either assigned a best route or has no route ($\perp$).

Because a logical route may be learned from multiple neighbors, the algorithm also updates the *RemoteDevices* field accordingly, and sets the route's physical AS number to match the selected next hop.

3) Transfer Algorithm

For BGP, the transfer algorithm models route propagation and loop prevention under heterogeneous physical AS numbers. Under our AS constraints, each logical route falls into one of two cases: (1) all devices in *LocalDevices* share the same AS number, while devices in *RemoteDevices* may have different AS numbers; or (2) all devices in *RemoteDevices* share the same AS number, while devices in *LocalDevices* may have different AS numbers.

In case (1), to determine whether a particular physical device drops the logical route due to a loop, we instantiate the AS path using the physical AS number corresponding to each device in *RemoteDevices*, and then apply loop prevention for the route received from that device. In case (2), since all remote devices share the same AS number, the logical route's AS path is identical across the corresponding physical routes. We therefore only need to check loop prevention using each local device's AS number.

IV. Control-Plane Simulation Acceleration

Based on the compressed network, *sNetVeri* further accelerates control-plane simulation with an enhanced route computation method. Because prefixes are typically originated by different destinations, their simulations are largely independent. We therefore begin with per-prefix control-plane simulation (considering route aggregation), and then present the complete multi-prefix simulation algorithm.

A. Single-Prefix Control-Plane Simulation

In BGP, a route carries the AS numbers of traversed nodes during propagation, and each node typically prefers routes with

Algorithm 1: Single-Prefix Control-Plane Simulation

```
1  Function SIMULATION(G, d, a_d, T):
2      dist(d) ← {a_d};
3      queue ← {(a_d, d)};
4      while queue ≠ ∅ do
5          (a, u) ← arg min_{(r,v)∈queue} (r under ≺);
6          queue ← queue \ {a, u};
7          foreach v ∈ E(u) do
8              a' ← trans(u → v, a);
9              TRIGGEREDAGG(a', v, T);
10             if dist(v) = ∅ then
11                 dist(v) ← {a'};
12                 queue ← queue ∪ {v, a'};
13             else
14                 RECONCILEROUTES(a', v);
15     return dist;
16 Function RECONCILEROUTES(a',u):
17     hasWithdraw ← False;
18     foreach r ∈ dist(u) do
19         if (a'.localDevices ∩ r.localDevices ≠ ∅ ∧ a' ≺ r)
           then
20             WITHDRAW(r, u);
21             hasWithdraw ← True;
22     if hasWithdraw then
23         foreach r ∈ Solution(u) do
24             if r ∉ dist(u) then
25                 dist(u) ← {r};
26                 queue ← queue ∪ {r, u};
27     return dist;
28 Function WITHDRAW(a,v):
29     dist(v) ← dist(v) \ {a};
30     queue ← queue \ {a, v};
31     foreach u ∈ E(v) do
32         a' ← trans(v → u, a);
33         if a' ∈ rib(u) then
34             rib(u) ← rib(u) \ {a'};
35         RECONCILEROUTES(r, u);
36     return dist;
37 Function TRIGGEREDAGG(p, v,T):
38     foreach q ∈ P_agg(v) do
39         if IFCONTAINS(q, p)∧AGGPOLICY(v, p) then
40             T ← T ∪ {q};
41     return T;
```

Algorithm 2: Multi-Prefix Control-Plane Simulation

```
1  Function PARALLEL SIMULATION(G, P):
2      P_sorted ←SORTBYSPECIFICITY(P);
3      P_reg ← [];
4      P_aggSorted ← [];
5      foreach p ∈ P_sorted do
6          if p ∈ P_agg then
7              P_aggSorted ← P_aggSorted ‖ [p];
8          else
9              P_reg ← P_reg ‖ [p];
10     T ← ∅ ;        // Triggered aggregate prefixes
11     foreach p ∈ P_reg in parallel do
12         T, dist_p ←SIMULATION(G, d, a_d(p), T);
13     foreach p ∈ P_aggSorted do
14         if p ∈ T then
15             dist_p ←SIMULATION(G, d, a_d(p), T);
16     return dist;
```

shorter AS paths. As a result, the route's weight decreases as it propagates through the network, meaning each node can simply select the shortest path as its optimal route. Sobrinho [18] defines monotonic behavior as the property where a route's weight decreases consistently as it propagates. However, in practice, the AS path can be altered during propagation, and nodes prioritize the local-preference attribute, which can be arbitrarily modified at each hop, as illustrated in Figure 2. This leads to situations where a route's weight may increase or decrease during propagation. Such cases, though rare in DCNs, break the monotonic property [10].

Leveraging the fact that DCNs are largely monotonic, we compute the best routes by treating the problem as finding globally optimal paths. For the rare non-monotonic cases, we incorporate a recursive withdrawal mechanism that retracts incorrectly selected routes and recomputes the optimal routes. Algorithm 1 takes as input the network topology G, a destination vertex d, the destination's original route a_d, and a set $\mathcal{T}$ for recording triggered aggregate routes. The algorithm initializes the distance map $dist$ (line 2), similar to shortest-path distances, to store each node's best routes. A global priority $queue$ (line 3) holds candidate routes ordered by $\prec$. The main loop (line 4) pops the highest-weight route (a, u) (line 5) and propagates it to each neighbor v (line 7) using the transfer function $trans(u \rightarrow v, a)$ (line 8). It then checks whether the received route a' triggers any aggregate route at v (line 9). If v has no route (line 10), a' is assigned to $dist(v)$ and enqueued (lines 11–12). Otherwise, if a' has higher priority than the current routes in $dist(v)$ (lines 14, 19–21), the recursive $\mathrm{Withdraw}(a, v)$ function (lines 28–36) is triggered to remove outdated routes. If withdrawals occur, the algorithm recomputes v's best routes via $\mathrm{Solution}(v)$, adds new routes to $dist(v)$, and enqueues them for further propagation (lines 22–26). The process repeats until the queue is empty, ensuring a stable routing solution.

B. Multi-Prefix Control-Plane Simulation

In BGP, manually configured aggregate routes are not advertised independently; instead, they are triggered by the presence of more specific contributing routes. During parallel computation, we record which aggregate prefixes are triggered (Algorithm 1, lines 9 and 37–41). After finishing the computation for all non-aggregate prefixes, we sequentially process each triggered aggregate route from the most specific to the least specific prefix (Algorithm 1, line 2 and lines 13–15). This ordering preserves dependencies among aggregates.

For the complete simulation algorithm, we first derive $\mathcal{P}_{aggSorted}$ and $\mathcal{P}_{reg}$ from the current configuration (line 5–9). We then run the per-prefix simulation in parallel for prefixes in $\mathcal{P}_{reg}$ (Algorithm 2, lines 11–12). For prefixes involved in aggregation, we process them sequentially according to $\mathcal{P}_{aggSorted}$ (Algorithm 2, lines 13–15).

V. DATA-PLANE VERIFICATION ACCELERATION

In this section, we describe how *sNetVeri* accelerates data-plane reachability verification on the compressed network.

Dataset		Compression		End-to-End Running Time (s)												Memory Usage (MB)	
Topology	Node/Edge	Bonsai	sNetVeri	Batfish			Bonsai+Batfish				sNetVeri					sNetVeri	
		Compressed Node/Edge	Compressed Node/Edge	Simu.	Veri.	All	Comp.	Simu.	Veri.	All	Comp.	Simu.	Veri.	All		Simu.	Veri.
Fattree 40	2000/32000	6/5	860/1600	504.24	>2h	>2h	13.61	1220.80	745.52	1979.93	3.68	3.63	7.25	14.56		7783.39	2841.68
Fattree 50	3125/62500	6/5	1325/2500	1262.14	>2h	>2h	41.01	1892.50	1382.75	3316.26	9.23	7.39	19.77	36.39		12105.67	6771.63
Fattree 60	4500/108000	6/5	1890/3600	>2h	>2h	>2h	91.42	2019.60	2422.44	4533.46	11.90	15.75	33.39	61.04		12851.38	11716.50
Fattree 70	6125/171500	6/5	2555/4900	>2h	>2h	>2h	195.99	3682.35	2331.67	6210.01	22.70	29.74	65.45	117.89		13731.80	21581.95
Fattree 80	8000/256000	6/5	3320/6400	>2h	>2h	>2h	379.05	4310.40	4101.76	>2h	27.16	60.28	79.98	167.42		18915.99	29476.13
Fattree 90	10125/364500	6/5	4185/8100	>2h	>2h	>2h	665.23	5431.05	6836.81	>2h	39.63	130.47	179.57	349.67		24993.43	52121.11
Huawei Cloud DCN	13325/117791	1157.97/2903.73	1191/2986	>2h	>2h	>2h	2373.66	>2h	>2h	>2h	25.86	74.05	420.37	520.28		13807.92	50071.09
Bell South	47/62	7.77/3.66	29/39	2.00	641.67	643.67	0.0939	39.26	87.68	127.03	0.0104	0.4651	0.0290	0.5045		139.76	30.93
FCCN	23/25	7.09/3.91	9/10	1.53	144.27	145.80	0.0345	19.99	33.43	53.46	0.0043	0.4203	0.0086	0.4332		101.71	31.25
GRnet	36/41	9.14/5	31/36	1.86	366.37	368.23	0.0752	30.32	74.45	104.84	0.0077	0.4212	0.0283	0.4572		118.64	31.16
GTS Hungary	27/28	7.41/4.63	17/18	1.51	206.75	208.26	0.0474	22.64	41.27	63.96	0.0055	0.4195	0.0128	0.4378		107.77	31.25
GTS Slovakia	32/34	7.03/3.78	14/16	1.24	289.89	291.13	0.0497	26.81	47.93	74.79	0.0060	0.4278	0.0155	0.4493		108.13	31.26
LITNET	42/42	6.88/4.36	10/10	1.91	514.06	515.97	0.0875	34.88	61.94	96.90	0.0067	0.4518	0.0180	0.4765		121.96	31.20
RoEduNet	41/45	6.34/5.22	14/18	1.31	486.07	487.38	0.0774	33.94	53.53	87.55	0.0076	0.4595	0.0176	0.4847		126.35	31.36
Tecove	70/70	9.09/5.94	36/36	2.04	1374.40	1376.44	0.1905	58.90	144.41	203.50	0.0124	0.4766	0.0451	0.5341		154.34	31.33
ULAKNET	79/79	5.54/3.19	14/14	2.73	1776.80	1779.53	0.1858	67.53	87.17	154.89	0.0126	0.4952	0.0694	0.5772		143.66	31.35
VinaREN	22/24	8.73/5.18	15/17	1.24	93.14	94.38	0.0436	18.33	42.86	61.23	0.0043	0.3966	0.0101	0.4110		104.02	31.46

TABLE I: End-to-end running time (with a breakdown by major components), compression results for different configuration verifiers, and memory usage of *sNetVeri* on Huawei Cloud DCN, synthesized DCNs, and WANs.

Algorithm 3: Parallel Data-Plane Verification

```
1  Function PARALLEL VERIFICATION(G,Nodes,Targets,X):
2      M_ps^phy, M_ps^n, M_ds^n, M_ds^phy ←
         ENCODING(G, Nodes, Targets, X);
3      reachable ← ∅;
4      foreach dst ∈ Target in parallel do
5          Net ←INITTOPOLOGY(M_ps^n, M_ds^n[dst]);
6          VERIFICATION(Targets, Net, M_ds^n[dst]);
7          foreach phy_dst ∈ dst do
8              foreach src ∈ Targets do
9                  foreach phy_src ∈ src do
10                     foreach key_port ∈ src.reachPred do
11                         if src.reachPred[key_port] ∧
                             M_ds^phy[phy_dst] ∧
                             M_ps^phy[src][phy_src][key_port] ≠ ∅
                           then
12                             reachable ← {(phy_dst, phy_src)};
13                             break;
14     return reachable;
```

We first present the parallel verification framework, which which constructs independent and destination-specific graphs to compute all-to-one reachability. We then explain how *sNetVeri* performs BFS on each such graph while preserving enough compressed-device context to recover physical-device-level reachability.

A. Parallel Verification Framework

All-to-all reachability verification is notoriously susceptible to state-space explosion, with its computational overhead scaling intractably as the network grows. While a naive approach—constructing a separate directed graph for every source-destination pair—is prohibitively expensive, *sNetVeri* mitigates this by building a single, destination-anchored graph to concurrently compute reachability from all potential sources.

sNetVeri first defines a base compressed network *BaseNet* as a directed graph $G_b = (V, E, \mathcal{P}, \mathcal{P}^{phy})$, where:

- V is the set of verification nodes. Each $v \in V$ represents a node in the compressed network topology.
- E is the set of directed edges. Each edge $e = (u, v) \in E$ represents a connection from node u to node v in the compressed topology.
- $\mathcal{P}$ denotes the set of BDD-encoded per-port forwarding packet spaces. Each $p \in \mathcal{P}$ represents the packet header space that can be forwarded through a port.
- $\mathcal{P}^{phy}$ denotes the set of per-device, per-port BDD constraints. Each $p \in \mathcal{P}^{phy}$ represents the packet header space that a physical device can forward through a specific port.

To construct *BaseNet*, *sNetVeri* first encodes packet spaces into BDDs. It collects all route prefixes and builds a prefix-to-BDD cache. Then, for each compressed node, *sNetVeri* groups routes by next-hop neighbor, which corresponds to an output port, and unions the packet spaces associated with each port to obtain $\mathcal{P}$. For each target node, *sNetVeri* also records per-physical-device forwarding spaces in $\mathcal{P}^{phy}$, so that the final reachability result can be mapped back to individual physical devices.

Then, *sNetVeri* defines a destination-specific compressed network *Net* with destination node d as a directed graph $G_d = (G_b, \mathcal{P}_d, \mathcal{P}_d^{phy})$, where:

- $\mathcal{P}_d$ denotes the BDD-encoded destination packet space for destination node d.
- $\mathcal{P}_d^{phy}$ denotes the set of per-physical-destination BDD constraints. Each $p \in \mathcal{P}_d^{phy}$ represents the destination packet space of a physical device within destination node d.

As illustrated in Algorithm 3, for each destination, *sNetVeri* constructs a single *Net* that covers all source nodes and uses the destination packet space $\mathcal{P}_d$ to prune unreachable packet spaces during graph construction (line 5). Different destination-specific *Net* instances are independent, so they can be verified in parallel (line 6). After verification, *sNetVeri* determines reachability for each physical destination–source pair by intersecting the reachability records of each target node with the physical destination space in $\mathcal{P}_d^{phy}$ and the physical source forwarding space in $\mathcal{P}^{phy}$ (line 7–13).

This design reduces the space complexity of reachability verification from $O(|V_{\text{phy\_dst}}| \times |V_{\text{phy\_src}}|)$ to $O(|V_{\text{node\_dst}}|)$, where $|V_{\text{phy\_src}}|$ and $|V_{\text{phy\_dst}}|$ denote the numbers of physical source and destination devices, respectively, and $|V_{\text{node\_dst}}|$ denotes the number of destination nodes in the compressed network.

B. Verification on Compressed Network

Having described how verification tasks are organized across destinations, we now explain how ***sNetVeri*** verifies reachability within one destination-specific *Net*. The detailed pseudocode is provided in the appendix [17].

Under parameterized-equivalent compression, physical devices within the same logical node may still exhibit subtle differences in forwarding behavior. Therefore, a reachable path between two logical nodes does not necessarily imply that every physical device in the source node can reach every physical device in the destination node. The key challenge is to preserve enough per-device information while still exploiting the compactness of the compressed topology.

Recording all relevant paths. A logical path may correspond to reachability for only a subset of physical devices. To avoid losing such distinctions, ***sNetVeri*** records the reachable packet space for each path and each output port during traversal. Specifically, BFS starts from the destination node with its destination packet space, *i.e.*, the packet space of all original routes originated by that destination. Instead of stopping once a target node is reached, ***sNetVeri*** continues traversing all relevant edges and records the packet space that can reach the destination from each port. A traversal is pruned only when the newly reached packet space does not expand the packet space already recorded for that port. In this way, ***sNetVeri*** captures all reachable paths at port granularity. For example, in Figure 2d, $P_{T\leftarrow A_2\leftarrow T}$ records the packet space that can reach the destination logical node T through A_2.

Checking reachability per device. After recording reachable packet spaces at the logical-node level, ***sNetVeri*** maps the result back to physical device pairs. For destination devices, ***sNetVeri*** records destination packet spaces separately; for example, $P_{T_1\leftarrow}$ denotes the destination space of T_1. Intersecting this space with a recorded reachable path filters out packets that cannot reach the specific destination device. Similarly, for source devices, ***sNetVeri*** records per-port packet spaces separately; for example, $P_{T_2\rightarrow A_2}$ denotes the packet space that T_2 can forward to A_2. Given the path packet space $P_{T\leftarrow A_2\leftarrow T}$, where the first T denotes the destination logical node and the last T denotes the source logical node, ***sNetVeri*** intersects it with $P_{T_2\rightarrow A_2}$ and $P_{T_1\leftarrow}$. The remaining packet space is exactly the traffic that can reach T_1 from T_2 through A_2. This per-port, per-device intersection provides the bridge from compressed-node reachability to physical-device reachability.

Specifically, after encoding, ***sNetVeri*** performs BFS over the compressed topology, starting from the destination node. It maintains a queue of nodes together with their currently reachable packet-space predicates, as well as a visited map that records the packet space already explored on each port. For each popped node n and predicate $pred$, ***sNetVeri*** traverses each neighbor v. It first checks whether port (n, v) has already been visited with a superset of the current predicate, in which case the traversal can be pruned. Otherwise, ***sNetVeri*** computes the packet space that can be forwarded to v. If v is a target node, ***sNetVeri*** records the reachable packet space for later physical-device-level checks. If the remaining packet space is non-empty, ***sNetVeri*** pushes v and its reachable predicate into the queue, and then updates the visited map for the port. The algorithm terminates when the queue becomes empty.

VI. Evaluation

We implement a prototype of ***sNetVeri*** in 4K LoC in Java and evaluate its performance extensively to study the following three questions: (1) Can ***sNetVeri*** support WANs and scale to hyper-scale production DCNs? (2) How does ***sNetVeri*** perform compared to the state-of-the-art configuration verification tools (including CPS and DPV)? (3) How does ***sNetVeri*** degrade under non-monotonic configurations?

A. Experimental Setup

Environment. All experiments are conducted on a server equipped with two 20-core Intel(R) Xeon(R) Gold 6230 2.10GHz CPUs and 256 GB of DDR4 DRAM. If not specified, we assume that all CPUs and memory are usable.

Dataset. We use Huawei Cloud DCN, a suite of synthetic DCNs based on Fattree topologies, and 10 synthetic WANs based on topologies from Topology Zoo [20]. The real production DCN consists of 13,325 devices and 117,791 links. Each device in the network originates over 140,000 initial routes, and a wide range of complex routing policies. Furthermore, the configurations of Huawei Cloud DCN are monotonic.

Synthetic DCNs include Fattree topologies with $k \in \{40, 50, 60, 70, 80, 90\}$, where each ToR node is assigned a destination IP prefix. We generate three three variants: monotonic (M), non-monotonic (NM), and highly non-monotonic (HNM), totaling 18 topologies. Variants are constructed by manipulating route preferences using *AS-path-length*, *BGP-community-tag*, and *local-preference*. Specifically, we reduce route preferences by increasing AS path length and adding community tags to denote these path-length-increased routes. Then, we define import policies against these tags to create different levels of monotonicity violations: in NM, Aggregation nodes are configured to increase the local preference of these tagged routes, causing non-monotonic behavior; in HNM, this strategy is applied to all Aggregation and ToR nodes.

We adopt the same set of WAN topologies from Topology Zoo and BGP configurations as in ACORN [2] for direct comparison. These topologies contain 22–79 nodes, and we assign one destination IP prefix to each node.

Benchmark Tools. We evaluate ***sNetVeri*** on all-pairs reachability. For end-to-end verification, we compare ***sNetVeri*** against Batfish [8], Bonsai [15] and ACORN [2]. For Bonsai, we reimplement its compression procedure and run control-plane simulation (CPS) and data-plane verification (DPV) on the resulting compressed network using the latest Batfish. For ACORN, we construct synthetic Fattree instances for single-

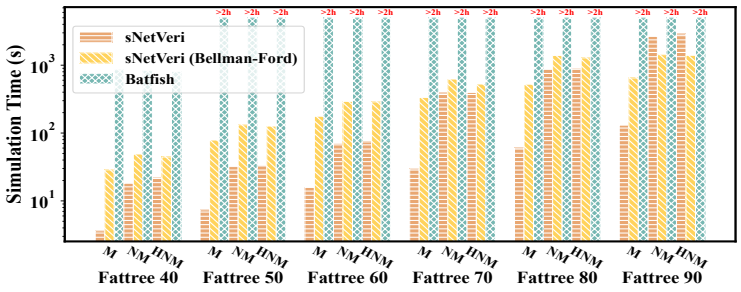


Fig. 4: Runtime of different control-plane simulation tools on synthetic DCNs considering both monotonic and non-monotonic configurations.

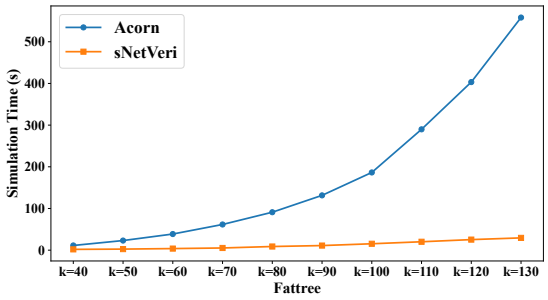
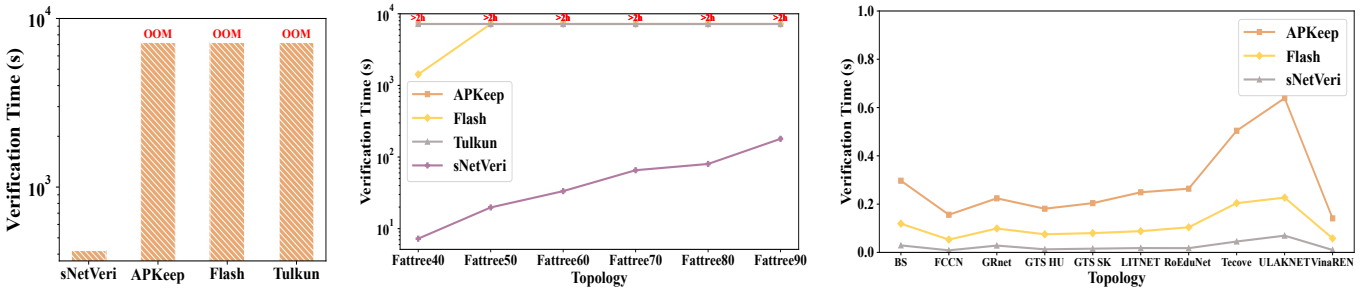


Fig. 5: End-to-end runtime of Acorn [2] and *sNetVeri* on synthetic DCNs.



(a) Huawei Cloud DCN **(b)** Synthetic DCNs **(c)** WANs

Fig. 6: Runtime of different DPV tools (APKeep [11], Flash [12], Tulkun [19], and *sNetVeri*) on DCNs and WANs.

destination shortest-path queries ($k \in [40, 130]$) and compare *sNetVeri* against the publicly reported ACORN results [21].

For the CPS comparison, we include Batfish (its coloring-based Bellman–Ford), *sNetVeri* with the simulation algorithm in §IV, and *sNetVeri* with a Bellman–Ford-based simulator. These experiments primarily use synthesized DCNs, with both monotonic and non-monotonic configurations.

For the DPV comparison, we include APKeep [11], Flash [12], and Tulkun [19]. All three are open-source implementations from public codebases [22]–[24]. We evaluate them on both DCN and WAN topologies.

B. Experimental Results

End-to-End Performance. Table I summarizes the end-to-end results. On synthetic DCNs, *sNetVeri* remains efficient as the topology scales (e.g., Fatree-90), whereas Batfish fails to complete within a reasonable time. On WAN topologies, although the networks are smaller, *sNetVeri* still outperforms Bonsai and Batfish in end-to-end runtime. On Huawei Cloud DCN, *sNetVeri* finishes end-to-end verification within 520.28 s (<9 min) with 50,071.09 MB memory usage, while Batfish again fails to complete within a reasonable time. Bonsai spends substantial time generating per-destination compressions and, in our experiments, does not complete subsequent simulation and verification on the compressed network. Figure 5 compares the end-to-end performance of ACORN and *sNetVeri* on synthetic DCNs. As the topology scales from Fattree-40 to Fattree-130, *sNetVeri* consistently outperforms ACORN, finishing in 1.89–29.45 s versus 11.26–558.09 s for ACORN, corresponding to a 5.96×–18.95× speedup and indicating better scalability.

Compression Performance. As shown in Table I, although Bonsai achieves a high compression ratio on Fattree topologies, it spends substantial time generating per-destination compressions. On Huawei Cloud DCN, Bonsai attains a compression ratio similar to *sNetVeri* but incurs significantly higher compression runtime, whereas *sNetVeri* compresses the topology once in 25.86 s. On WANs, *sNetVeri* also consistently outperforms Bonsai in compression time.

CPS Performance. As shown in Table I, *sNetVeri* outperforms Bonsai and Batfish in simulation performance across topology types and scales. To evaluate whether *sNetVeri*'s simulation algorithm remains effective under non-monotonic configurations, we further compare *sNetVeri* with Batfish across multiple degrees of non-monotonicity. As illustrated in Figure 4, our algorithm consistently outperforms Batfish across different topology sizes, even in highly non-monotonic settings. Notably, as the network scales to larger topologies (*i.e.*, Fattree-90), *sNetVeri* remains efficient, whereas Batfish fails to produce results within a reasonable time.

DPV Performance. As shown in Figure 6a, on Huawei Cloud DCN, *sNetVeri* completes verification in 420.37 s, whereas APKeep, Flash, and Tulkun fail due to out-of-memory (OOM) errors. On synthetic DCNs, *sNetVeri* remains efficient as the topology scales (e.g., Fattree-90), while the other three tools do not finish within a reasonable time except that Flash can work under Fattree-40. For smaller WANs, *sNetVeri* is slightly slower than APKeep and Flash, but the gap is minor.

VII. Related Work

Control-Plane Verification. Control-plane verifiers (CPVs) [1]–[3], [25] check network properties by directly encoding configurations and routing semantics into formal models.

To improve scalability, ShapeShifter [3] and ACRON [2] introduce abstractions over paths or route attributes, but these abstractions can over-approximate network behavior and thus yield false positives. Bonsai [15] improves scalability via network abstraction, yet its abstraction is constructed *per destination* and is therefore not well-suited for multi-destination properties such as all-pairs reachability. In contrast, *sNetVeri* introduces a *one-shot* compression that can be reused across destinations while preserving forwarding equivalence, enabling scalable verification of multi-pair properties.

Control-Plane Simulation. Simulation-based tools compute routing and forwarding behavior under given configurations and are widely used in practice. Prior work accelerates control-plane simulation via custom simulation engines [7]–[9], Datalog-based formulations [4], [6], explicit-state exploration [5], generalized Dijkstra-style computation under restricted configuration classes [10], and distributed simulation across multiple servers [13], [14]. These approaches primarily speed up route computation on the *original* topology, and thus remain bottlenecked by hyper-scale network size. *sNetVeri* is complementary: it *reduces the input topology* via compression and then accelerates route computation on the compressed network, jointly improving scalability while maintaining forwarding-equivalent behavior.

Data-Plane Verification. Data-plane verifiers (DPVs) analyze forwarding tables to check properties such as reachability, isolation, and waypointing. APKeep [11] and Flash [12] are high-performance centralized verifiers that rely on equivalence classes (ECs) and efficient EC maintenance. Hetu [26] further improves BDD management to enable high-throughput parallel verification on a single machine, while distributed verification has also been explored to scale across multiple servers or network devices [13], [14], [19], [27]. Plotkin et al. [16] leverage DCN symmetry (and surgery) to reduce the verification problem size, but their reductions are typically tied to per-destination reasoning. In contrast, *sNetVeri* performs DPV on a *single* compressed/hybrid topology that is reusable across destinations, which is crucial for multi-pair queries such as all-pairs reachability. Our compression is orthogonal to advances in BDD engines (e.g., Hetu [26]) and can be combined with them to further improve performance.

VIII. Conclusion

We present *sNetVeri*, a scalable configuration verifier for hyper-scale networks. *sNetVeri* addresses the core scalability bottlenecks of existing verifiers through a novel one-shot network compression strategy, accompanied by two key designs executing on this compressed abstraction: a Dijkstra-based simulation algorithm augmented with a lightweight withdrawal mechanism, and a highly parallel verification framework that leverages device contexts to perform accurate verification without any decompression. Extensive evaluations on both WAN and production and synthetic DCN topologies demonstrate the efficiency and effectiveness of our approach.

References

[1] R. Beckett, A. Gupta, R. Mahajan, and D. Walker, "A general approach to network configuration verification," in *SIGCOMM '17*. ACM, 2017, pp. 155–168.

[2] D. Raghunathan, R. Beckett, A. Gupta, and D. Walker, "ACORN: network control plane abstraction using route nondeterminism," in *FMCAD'22*. IEEE, 2022, pp. 261–272.

[3] R. Beckett, A. Gupta, R. Mahajan, and D. Walker, "Abstract interpretation of distributed network control planes," *Proceedings of the ACM on Programming Languages*, pp. 1–27, 2019.

[4] A. Fogel, S. Fung, L. Pedrosa, M. Walraed-Sullivan, R. Govindan, R. Mahajan, and T. Millstein, "A general approach to network configuration analysis," in *NSDI '15*. USENIX, 2015, pp. 469–483.

[5] S. Prabhu, K. Y. Chou, A. Kheradmand, B. Godfrey, and M. Caesar, "Plankton: Scalable network configuration verification through model checking," in *NSDI '20*. USENIX, 2020, pp. 953–967.

[6] P. Zhang, A. Gember-Jacobson, Y. Zuo, Y. Huang, X. Liu, and H. Li, "Differential network analysis," in *NSDI '22*. USENIX, 2022, pp. 601–615.

[7] L. You, J. Zhang, Y. Jin, H. Tang, and X. Li, "Fast configuration change impact analysis for network overlay data center networks," *IEEE/ACM Transactions on Networking 2022*, vol. 30, no. 1, pp. 423–436, 2022.

[8] "Batfish," https://github.com/batfish/batfish, 2024.

[9] M. Brown, A. Fogel, D. Halperin, V. Heorhiadi, R. Mahajan, and T. Millstein, "Lessons from the evolution of the batfish configuration analysis tool," in *SIGCOMM '23*. ACM, 2023, pp. 122–135.

[10] N. P. Lopes and A. Rybalchenko, "Fast bgp simulation of large datacenters," in *VMCAI '19*. ACM, 2019, pp. 386–408.

[11] P. Zhang, X. Liu, H. Yang, N. Kang, Z. Gu, and H. Li, "$APKeep$: Realtime verification for real networks," in *NSDI '20*. USENIX, 2020, pp. 241–255.

[12] D. Guo, S. Chen, K. Gao, Q. Xiang, Y. Zhang, and Y. R. Yang, "Flash: fast, consistent data plane verification for large-scale network settings," in *SIGCOMM '22*. ACM, 2022, pp. 314–335.

[13] D. Wang, P. Zhang, W. Sun, W. Li, X. Feng, H. Li, J. Chen, W. Jiang, and Y. Tang, "S2: A distributed configuration verifier for hyper-scale networks," in *SIGCOMM'25*, 2025, pp. 796–808.

[14] Y. Yuan, F. Ye, Y. Li, J. Zhang, M. Liu, Y. Sang, R. Yang, D. She, Z. Ye, T. Guo *et al.*, "New evolution of hoyan: Enhancing scalability, usability, and accuracy for alibaba's global wan verification," in *SIGCOMM'25*, 2025, pp. 809–825.

[15] R. Beckett, A. Gupta, R. Mahajan, and D. Walker, "Control plane compression," in *SIGCOMM '18*. ACM, 2018, pp. 476–489.

[16] G. D. Plotkin, N. Bjørner, N. P. Lopes, A. Rybalchenko, and G. Varghese, "Scaling network verification using symmetry and surgery," *SIGPLAN '16*, vol. 51, no. 1, pp. 69–83, 2016.

[17] "sNetVeri Appendix: Correctness proofs for network compression, protocol extensions, and verification algorithms," https://github.com/sngroup-xmu/sNetVeri/blob/main/sNetVeri_iwqos26_appendix.pdf, 2026.

[18] J. L. Sobrinho, "Network routing with path vector protocols: Theory and applications," in *SIGCOMM '03*. ACM, 2003, pp. 49–60.

[19] Q. Xiang, C. Huang, R. Wen, Y. Wang, X. Fan, Z. Liu, L. Kong, D. Duan, F. Le, and W. Sun, "Beyond a centralized verifier: Scaling data plane checking via distributed, on-device verification," in *SIGCOMM '23*. ACM, 2023, pp. 152–166.

[20] "Topology zoo," https://topology-zoo.org/, 2013.

[21] "Acorn benchmarks," https://github.com/divya-urs/ACORN_benchmarks, 2022.

[22] "Flash," https://github.com/snlab/flash, 2024.

[23] "Tulkun," https://github.com/sngroup-xmu/ddpv-pubilc, 2024.

[24] "APKeep," https://github.com/XJTU-NetVerify/apkeep, 2020.

[25] X. Fang, F. Ding, B. Huang, Z. Wang, G. Han, R. Yang, L. You, Q. Xiang, L. Kong, Y. Liu *et al.*, "Network can help check itself: Accelerating smt-based network configuration verification using network domain knowledge," in *INFOCOM '24*. IEEE, 2024, pp. 2119–2128.

[26] P. Peng, X. Sun, Z. Shen, F. Ding, J. Chen, L. You, W. Jiang, Y. Tang, and F. Luo, "Hetu: High-performance centralized parallel data-plane verification for hyper-scale dcns," in *2025 IEEE 33rd International Conference on Network Protocols (ICNP)*. IEEE, 2025, pp. 1–11.

[27] H. Zeng, S. Zhang, F. Ye, V. Jeyakumar, M. Ju, J. Liu, N. McKeown, and A. Vahdat, "Libra: Divide and conquer to verify forwarding tables in huge networks," in *NSDI '14*. USENIX, 2014, pp. 87–99.