S2: A Distributed Configuration Verifier for Hyper-Scale Networks

Dan Wang
Xi'an Jiaotong University
dan-wang@stu.xjtu.edu.cn

Peng Zhang
Xi'an Jiaotong University
p-zhang@xjtu.edu.cn

Wenbing Sun
Xi'an Jiaotong University
wblingsun@gmail.com

Wenkai Li
Xi'an Jiaotong University
wkli24@stu.xjtu.edu.cn

Xing Feng
NorthWest University
fengxing@stumail.nwu.edu.cn

Hao Li
Xi'an Jiaotong University
hao.li@xjtu.edu.cn

Jiawei Chen
ByteDance
chenjiawei.cpv@bytedance.com

Weirong Jiang
ByteDance
weirong.jiang@bytedance.com

Yongping Tang
ByteDance
yongping.tang@bytedance.com

ABSTRACT

Network configuration verifiers can proactively reason about a network's correctness to prevent network outages. However, even recent efforts have proposed algorithms to "scale up" the verification to several thousand switches, these algorithms still cannot be used for networks with more than 10K switches or 1000M routes, which is common for large service providers. In this paper, instead of further scaling up the verification limited to a single server, we study how to "scale out" the verification using the resources of multiple servers. To achieve this, we propose S2, a distributed verifier for network configurations. S2 partitions the network model and distributes the verification tasks, *i.e.*, control plane simulation and data plane verification, to run on multiple servers in parallel. Additionally, S2 uses prefix sharding during control plane simulation to further reduce the memory footprint on each server. We implement a prototype of S2 based on Batfish, the state-of-the-art network verifier. Based on real datacenter topologies of a large service provider and synthetic FatTree topologies, we show that S2 can verify networks with 10K routers and 1000M routes within 2 hours.

CCS CONCEPTS

• **Computer systems organization** → *Reliability*;

KEYWORDS

network verification, distributed system, control plane simulation, data plane verification

ACM Reference Format:
Dan Wang, Peng Zhang, Wenbing Sun, Wenkai Li, Xing Feng, Hao Li, Jiawei Chen, Weirong Jiang, and Yongping Tang. 2025. S2: A Distributed Configuration Verifier for Hyper-Scale Networks. In *ACM SIGCOMM 2025 Conference (SIGCOMM '25), September 8–11, 2025, Coimbra, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3718958.3750516

SIGCOMM '25, September 8–11, 2025, Coimbra, Portugal

ACM ISBN 979-8-4007-1524-2/25/09…$15.00
https://doi.org/10.1145/3718958.3750516

1 INTRODUCTION

Datacenter networks (DCNs) of large service providers can easily go beyond tens and even hundreds of thousands of switches [15, 26, 54]. Managing such hyper-scale networks is quite challenging and error-prone, especially considering the networks are constantly evolving with non-standard configurations, and switches of different vendors coexist with different vendors having their specific protocol behaviors [21, 32].

Network verification provides valuable means for operators of such hyper-scale networks to ensure the correctness of configurations. In the recent decade, tens of network configuration verifiers have been proposed [5, 10–12, 19, 22–24, 34, 42, 52]. Unfortunately, existing verifiers have limited scalability, preventing their use in hyper-scale networks. To show this, we roughly group existing verifiers into two classes, *i.e.*, simulation-based verifiers and analysis-based verifiers.

Simulation-based verifiers, *e.g.*, Batfish [19], build a model that mimics the routing and forwarding behaviors of switches, and based on the model, simulate the route computation and packet forwarding process to check properties. These verifiers can provide relatively faithful results (routing tables, packet forwarding paths), but have limited scalability. For example, it was reported that Batfish cannot scale to 2K switches with 100GB memory [34], and the largest size of FatTree Plankton experiments with is 2205 switches, which cost 170GB memory [38]. Based on our synthesized FatTree configurations, we found the latest Batfish can handle 2K switches (§5.4). To simulate even larger networks, previous efforts tried to *scale up* the performance of simulation-based CPVs (Control Plane Verifiers) on a single machine by trading off some accuracy [11, 12, 34]. For example, FastPlane [34] designed a customized algorithm to simulate BGP for 2K switches, but needs to assume BGP route preferences are monotone, *i.e.*, preferences do not decrease during route propagation, which may not hold in all networks. Moreover, such a scale (2K switches) is still far below the 10K switches and 1000M routes.

Analysis-based verifiers check properties based on a control plane model (graph [5, 22], SMT constraints [10], etc.), without simulating the route computation. They can efficiently reason about arbitrary link failures, but are less faithful compared to simulation-based verifiers, due to the lower coverage of protocol features. Moreover, they also face scalability problems, often limited to hundreds of switches [10].

Therefore, none of existing configuration verifiers can scale to hyper-scale networks with more than 10K switches and 1000M routes.

This paper proposes S2, a new verifier that can *scale out* the performance of network verification with more compute and memory resources. At a high level, S2 partitions the network into smaller segments, and assign them to multiple servers for distributed configuration verification, including simulating the routing protocols to generate the data plane state and based on it compute the packet forwarding behaviors for property checking. Since each server only executes the above tasks for a subset of switches, and holds a subset of all routes, the per-server computation and memory load can be reduced, proportional to the number of servers.

Making the verification distributed is the key to achieve scalability, but we find it is still not efficient enough to work on networks with a huge number (say >1000M) of routes. The reason is that switches need to hold all the routes during the route computation, therefore the memory cost can overwhelm each single server. To further reduce the memory cost during the distributed control plane simulation, we propose *prefix sharding*. It leverages the fact that route computations for different prefixes are mostly independent, and partitions the prefixes into several shards for multi-round simulation. At each round, only routes for a subset of prefixes reside on the servers, and when this round ends, we write it to persistent storage and start a new round for the next shard. We find prefix sharding is essential for S2 to scale to hyper-scale networks without speed penalty.

However, building a distributed verifier from scratch is hard, due to the labor-intensive work of building accurate switch models, which have multiple vendor-specific behaviors (VSBs) beyond the RFC standards. Therefore, in order to be easy to develop, deploy, and evolve, we choose to decouple the design of S2 from the verification logic as much as possible, and realize the verification logic with existing verifiers built by the community. Such a decoupled design enables us to easily adapt relatively mature verifiers to work in the framework of S2, and keep evolving with the collective effort from the community.

We implement S2 with 12K LOC of Java for the distributed verification framework, and reuse the configuration parsers, control plane computation, data plane verification of Batfish, with around 500 LOC modifications to its original code base [18]. Using both real and synthesized datacenter networks, we show that S2 can scale to datacenter networks with over 10K switches with 16 logical servers. In contrast, the vanilla Batfish runs out of memory for FatTrees with more than 2K switches on a single logical server (§5).

Contributions. In summary, this paper makes the following contributions:

- We motivate the need to verify hyper-scale DCNs based on experiences from a real DCN of a large service provider, and present a general way to make network configuration verifiers scale out with more servers.
- We design and implement a prototype of S2, a distributed control plane verifier on top of Batfish [18] that can scale to large datacenter networks with more than 10K switches and 1000M routes.
- We evaluate the performance of S2 using the configurations of a real datacenter network and various sizes of synthesized FatTrees [39]. The results show that S2 is the only one that can scale to large topologies (>10K switches and 1000M routes).

2 MOTIVATION

In this section, we first motivate the need and challenges to verify hyper-scale datacenter networks, then we share the experiences from a real hyper-scale DCN.

2.1 Hyper-Scale DCNs are Error-Prone

DCNs of large service providers can have tens and even hundreds of thousands of switches [15, 26, 40]. One may think the configurations of DCNs are quite standard, *e.g.*, automatically configurations with some synthesizers [13, 43], and hard to go wrong. However, our experiences indicate that misconfigurations are still common to DCNs, due to the following reasons.

Vendor-Specific Behaviors (VSBs). The switches in a single DCN can be from different vendors, which have different implementations of routing protocols and exhibit different behaviors (*i.e.*, vendor-specific behaviors, VSBs). Taking the `remove-private-AS` command for example, switches of some vendors will remove all private AS numbers, while those of other vendors only remove those private AS numbers preceding the first non-private one [52]. Due to the VSBs of different vendors, operators can make mistakes when configuring them. For example, a large service provider reports that 30% of their incidents are due to VSBs [21].

Nonstandard Configurations. Datacenters of large service providers often constantly evolve to meet new business demands. As a result, multiple generations of network architecture often co-exist in their datacenters. In addition, the DCNs may use different types of policy, generated with different templates. Finally, operators may manually change configurations for failure mitigation or planned updates, making the configuration styles diverge. The above nonstandard configurations increase the complexity of the network, and make the network error-prone.

2.2 Verifying Hyper-Scale DCNs is Challenging

Verifying hyper-scale DCNs is memory intensive. In datacenter networks, each switch (*e.g.*, TOR) may announce multiple prefixes, and the announcement for most of these prefixes will arrive at every other switch in the network. This makes the total number of routes quadric to the number of switches or more. Taking FatTree60 (4500 switches) for example, there are $O(4 \times 10^8)$ routes in total. For control plane simulation, this indicates a huge memory cost due to the need to keep those routes in memory during the simulation[1]. For data plane verification, which extensively

[1]Since the DCN topologies are symmetric, the routes show great similarity. We tried to leverage this similarity to maintain only one copy for each unique route. However,

use BDD (Binary Decision Diagram), such a large number of routes can easily overflow the BDD node table, whose size is bounded by $O(2^{32})$ [57].

Verifying hyper-scale DCNs is compute intensive and lacks parallelism. To achieve flexibility and scalability, modern datacenter networks widely use BGP as the routing protocol. This means that the best routes are not always those of shortest paths, and to compute the best routes, network simulators need to mimic the routing process of each switch (applying import route policies, selecting best routes, and applying export policies). To speedup this process, one can let multiple switches compute their routes in parallel, as in Batfish. Even though such parallel execution can speedup the simulation process, the scalability is still limited since the number of cores of a single server is far less than the number of switches in a hyper-scale network. For data plane verification, which needs to simulate the forwarding of symbolic packets (encoded with BDDs) in the network, there will be a lot of expensive BDD operations, *e.g.*, conjunctions, disjunctions, and negations. In addition, since all switches share a single BDD data structure, which only allows one operation at the same time, the parallelism is rather limited. That is, even though theoretically different switches can forward symbolic packets in parallel, they may still be blocked due to BDD operations.

2.3 Experiences from a Hyper-Scale DCN

We share some experiences from a large service provider's DCN, highlighting the importance and challenges of verifying hyper-scale DCNs.

Scale. The DCN has 16K+ switches running BGP (*i.e.*, eBGP). The ASNs (Autonomous System Numbers) are assigned to ensure switches at the same layer (*e.g.*, TOR) have the same ASN, while those at different layers have different ASNs. Each of the switches has around 10K lines of configurations on average. To reduce the total number of routes, route aggregation policies are configured on switches at layer 3 or above (layer 0 is the bottom layer). However, there are still $O(2 \times 10^8)$ IPv4 routes and $O(3 \times 10^8)$ IPv6 routes in total. We have only parsed the IPv4 related configurations for experiments, and find that Batfish times out after 2 hours when simulating the computation of IPv4 routes. We hypothesize that after including IPv6 routes, the verification task can be more challenging.

Routing behaviors. To prevent route drops due to repetitive ASNs, switches are configured with `AS_PATH overwrite` policies, which overwrite the AS_PATH of matched routes to its own ASN. In addition to `AS_PATH overwrite`, there are route aggregation policies at core layers, which aggregates the VLAN interface addresses (used for business) and loopback addresses (used for management) received from switches of lower layers. The policies also tag the aggregated routes with specific communities, which will be used by the top-layer switches to filter routes when exchanging routes with each other, or the backbone switches that connect the DCN to the outside world. Finally, since the switches of this DCN are from 5+ different vendors, each with different behaviors, operators may need to configure the above routing policies in different ways.

we found this optimization was not effective since each switch still needs a pointer to the unique route, which is still a huge memory cost.

Nonstandard configurations. Even if all clusters in the DCN follow the Clos topology [16], different clusters may have different number of layers, with larger clusters having 5 layers, while smaller ones only having 3 layers. The other nonstandard configuration is ECMP (Equal-Cost Multi-Path): even for switches at the same layer, they may be configured with different maximum numbers of equal-cost paths. As a result, the number of routes on different switches can vary a lot and not quite predictable without simulation. Finally, some switches are configured with a dual stack of IPv4 and IPv6, while others only support IPv6. After talking with the operators, we know that the architecture DCN is still evolving, with new ASN assignment mechanisms and route aggregation policies in the near future.

3 S2 DESIGN OVERVIEW

We have the following goals when designing S2:

- *Scalability.* It should be able to scale to DCNs with 10K+ switches and 1000M+ routes.
- *Efficiency.* It should finish within a reasonable amount of time, say <2 hours.
- *Easiness.* It should be easy to *develop*, *deploy*, and *evolve*.

3.1 Key Ideas

"Scale out" instead of "scale up". To overcome the resource bottleneck observed in §2.2, we partition the task of configuration verification into smaller sub-tasks that can be run on multiple servers in a distributed way, so each server has a smaller memory cost and increased computation parallelism. S2 splits the network model into segments, each containing a smaller number of switches, and assigns these segments to run on multiple workers.

- For control plane simulation, the sheer amount of routes overwhelms the memory of a network verifier. By making the simulation distributed, each worker only needs to hold those routes for a subset of switches running on it. Therefore, the per-worker memory cost can be reduced, by a ratio proportional to the number of workers. In addition, since each worker only needs to compute routes for a subset of switches, the computation cost at each worker can also be reduced.
- For data plane verification, we can use different BDD data structures for symbolic packet forwarding on different workers (requiring serialization and deserialization when packets are forwarded across workers). This not only reduces the number of BDD nodes, alleviating the burden on BDD node table, but also increases parallelism. That is, BDD operations performed by a switch on one worker will not block the BDD operations performed by another switch on a different worker, and therefore switches on different workers can forward symbolic packets in parallel.

Even though we have distributed the verification tasks to multiple workers, the memory cost can still be too high for a single worker. Especially during control plane simulation, each worker needs to hold routes for all prefixes in the network, leading to a high peak memory cost. For example, when simulating FatTree80 (8000 switches), the per-worker peak memory usage reached around 86GB, approaching the memory capacity of a single worker (100GB),

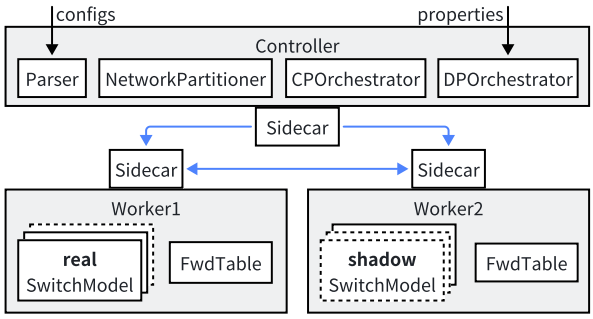

Figure 1: System architecture of S2.

and when simulating FatTree90 (10125 switches), all workers run out of memory (Figure 8).

Reduce memory cost with prefix sharding. We leverage the observation that route computations of different prefixes are mostly independent. That is, the computations of routes for different prefixes do not interfere with each other. Based on this observation, we divide the prefixes into multiple *shards*, each of which has only a subset of all prefixes. Then, we divide the route computation into multiple rounds, one for each shard. When the computation for a shard is finished, the resultant routes are written to disk. Thus, at any time during simulation, only routes of a single prefix shard reside in each worker's memory, thereby reducing the peak memory cost of workers. To ensure correctness, shards should be computed and scheduled with respect to route dependency. We show how S2 achieves this in §4.5.

Decouple the distributed framework from the switch model. Building an accurate switch model (especially for the control plane) is key to verification, but hard and labor-intensive due to the vendor-specific behaviors (VSBs) beyond the RFC standards. To be easy to develop and evolve, we choose to decouple the design of S2 from the verification logic as much as possible, such that we can adapt existing switch models (*e.g.*, Batfish's) to S2 and keep evolving with little manual effort. Towards such a decoupling, on each worker, S2 creates switch nodes running on a different worker as *shadow nodes*, which behaves exactly the same as real nodes, except that when a method of the shadow node is called, the shadow node relays the call to its real node via a proxy termed *sidecar*. In this way, a node is fully agnostic of whether its neighbor is on the same worker or not, allowing S2 to use existing switch models without any modification.

3.2 System Architecture

Figure 1 shows the system architecture with one *Controller* and multiple *Workers*, which communicate through their respective *Sidecars*.

Controller. The controller consists of *Parser*, *Partitioner*, and two *Orchestrators*.

- *The parser* converts the vendor-specific configuration files into vendor-independent models (VIs).
- *The partitioner* splits the network model into several *segments*, each of which holds a subset of switches and runs on a dedicated worker. The partitioner should ensure balanced workload across workers, while minimizing the inter-worker communication costs.
- *The control plane orchestrator (CPO)* schedules workers to compute routes for one or multiple routing protocols (*e.g.*, OSPF, BGP). For each protocol, CPO divides the prefixes into independent *shards* and orchestrates route computation for only one shard a time (§4.2).
- *The data plane orchestrator (DPO)* schedules workers to reason about the forwarding behaviors based on the computed routes, in a distributed way. It controls the general workflow of data plane verification, *i.e.*, first let all workers compute FIBs and forwarding/ACL predicates, then let all workers execute packet forwarding to check properties (*e.g.*, loop-free).

Workers. Each worker consists of a set of *Nodes*, each corresponding to a switch.

- *Control plane.* Each worker wraps each local switch hosted by it as a "real" node (solid switch model in Figure 1), and wraps each remote switch hosted on other workers with a "shadow" node (dotted switch model in Figure 1). The real node is the same as off-the-shelf switch models as in Batfish, while for the shadow node, we override the functions for sending and receiving route advertisements to use the *sidecars*.
- *Data plane.* For each "real" node running on a worker, the worker converts the routes of the node into forwarding rules, and then into a set of predicates (boolean formulas), each representing a set of packets that will be processed in the same way (forwarded to a specific port, permitted by port, etc.). Then, the workers construct symbolic packets (also boolean formulas), and forward them through the network. At each node, the worker applies logical operations on the symbolic packets with the predicates, and when a symbolic packet is forwarded to a node on a different worker, it uses the sidecar to serialize and send the symbolic packet.

Sidecars. We realize communications among the controller and workers through *sidecars*. Similar to sidecars in microservices, each sidecar of S2 is a separate process running on the same server hosting the controller or a worker, and uses remote procedure call (RPC) to communicate with each other. Each sidecar maintains a map from nodes to workers, so that when a node sends a route or packet to another node hosted on a different server, it forwards the route or packet to the sidecar, which will route it to the corresponding sidecars; when a sidecar receives a route or packet from other sidecars, it can route the route or packet to the corresponding node.

3.3 Workflow

Network Partition. Given user-provided vendor-specific configuration files, the parser first parses them into vendor-independent configurations. After that, the *Network-Partitioner* splits the network into several segments, and dispatches them to run on workers. When assigning segments to workers, we inform the overall assignment (*i.e.*, the assignment of every segment) to each worker, in order to let sidecars route requests among workers. Given assignments, workers will set up switch models. Specifically, a worker will set up

a "real" node for each switch assigned to it, and a "shadow" node for each other switch.

Control Plane Simulation. After setting up network partitions, the CPO will orchestrate the workers to execute a distributed, fix-point route computation algorithm. The algorithm is round-based, where in each round, workers let real nodes on them exchange routes with their neighbors (either real or shadow) and update their RIB. Benefits from our design, when a node exchanges routes with its neighbors, it does not need to differentiate between local (real) neighbors and remote (shadow) neighbors, since the shadow neighbor will delay the messages to the real neighbor on a different worker through RPC handled by sidecars.

Data Plane Verification. After the control plane simulation, routes of different nodes reside on their respective workers. Instead of gathering all routes onto a single server and utilize existing DPVs, the DPO schedules workers to reason about the forwarding behaviors in a distributed way. Firstly, on each worker, "real" nodes convert their RIBs into FIBs, and compute for each port the predicates (*i.e.*, the set of packets that will be filtered inbound/outbound, or forwarded to it) (*i.e.*, port predicates [51, 57]). Then, each worker executes packet forwarding to check properties (*e.g.*, loop-free). While packet transformations among workers incur additional overhead (*i.e.*, BDD serialization and deserialization), we found it negligible compared to the improved efficiency caused by improved parallelism of BDD-based packet forwarding.

4 DESIGN DETAILS

This section introduces the design details of S2.

4.1 Network Partition

Before control plane simulation, S2 partitions the network model into multiple *segments*, each of which has a subset of all nodes. Then, S2 assigns each segment to a different worker for distributed and parallel execution. The partition should optimize towards the following two goals:

- Balancing the workload across multiple workers, in terms of computation and memory.
- Minimizing the inter-worker communication overhead.

These two goals are general for distributed network simulation or emulation, but other works focus mainly on minimizing inter-server communications [21, 53, 58]. However, given our observation that memory is the primary bottleneck, when designing the partition algorithm, we place a higher priority on balancing workload among workers.

Algorithm. The network partitioning problem can be formulated as a graph partitioning problem, which is shown to be NP-hard [7]. Fortunately, some heuristic algorithms like METIS [3] yield good results in practice. To use these algorithms, we have to specify the load of each node and edge. We observe that the workload of each node is closely related to the number of routes on the node. Since the number of routes on each node is unknown before simulation, we estimate the number of routes for each node. For standard FatTree with k pods, every core, aggregation, and edge router is estimated to process approximately $k^3/2$, $k^3/2$, and $k^3/4$ routes, respectively. For nonstandard networks (like our DCN) where the number of routes is not easy to estimate, we assume uniform node loads. A more general and precise load estimation algorithm is left as one of our future works.

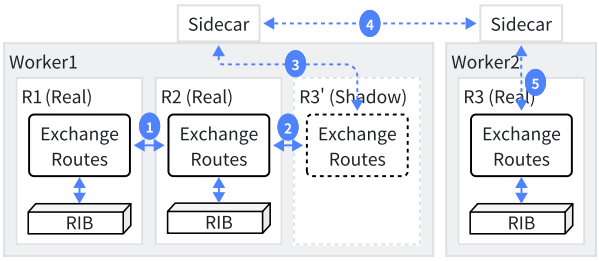

Figure 2: Illustration of the distributed control plane simulation in S2.

Algorithm 1: `RouteComputation(`W`)`

Input: W: workers.

```
// On Controller, fix-point route computation
   orchestrated by CPO.
converged ← false;
while ¬converged do
    converged ← true;
    foreach w ∈ W do
        converged ← RPC(w, Execute) ∧ converged ;
// On Worker, execute one route computation round.
Function Execute():
    foreach node ∈ w.assigned_nodes do
        foreach neighbor ∈ node.neighbors do
            updates ← neighbor.ExchangeRoutes(node);
            node.UpdateRIB(updates);
// Inside a Node, exchange route updates with a
   specific neighbor.
Function ExchangeRoutes(neighbor):
    if node ∈ w.assign_nodes then
        // This is a real node, call the wrapped model's
           function.
        node.model.ExchangeRoutes(neighbor);
    else
        // This is a shadow node, make RPC to the real
           node on another Worker.
        RPC(node, ExchangeRoutes, neighbor);
```

4.2 Distributed Control Plane Simulation

In this section, we show how S2 executes control plane simulation distributively. We also show how S2 partitions prefixes to further enhance the scalability of control plane simulation.

After setting up the network partition, the CPO orchestrates the workers to simulate the control plane in a parallel and distributed way, as shown in Algorithm 1.

First, if there are multiple routing protocols configured in the network, CPO schedules each protocol in sequence, with IGP protocols (*e.g.*, OSPF, RIP, IS-IS) before EGP (*e.g.*, BGP) protocols. For

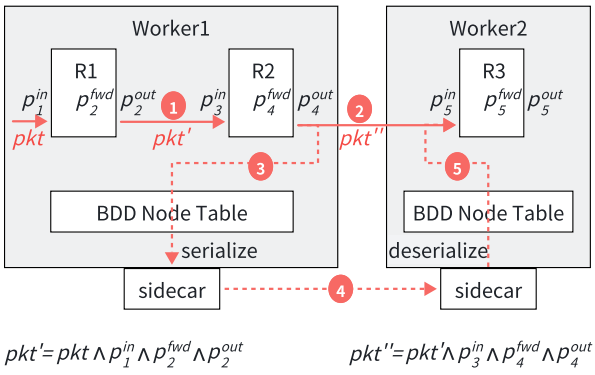

Figure 3: Illustration of the distributed data plane verification in S2.

each protocol, CPO orchestrates workers to run a distributed fix-point algorithm to compute the routes. The fix-point algorithm runs in multiple rounds. At each round, all nodes on a worker pull route updates from their neighbors (line 9) and merge the update into their RIBs in parallel (line 10). This process continues until all workers report the fix point is reached, *i.e.*, there are no more route updates.

Note in our design, each node A pulls route updates from all of its neighbors in the same way, fully agnostic of whether the neighbor is on the same worker or not. This allows S2 to avoid modifying the logic of control plane simulation shipped with existing verifiers like Batfish. We achieved this by creating a real node for each switch hosted on the worker, and a shadow node for each switch not hosted on the worker. If the neighbor node is on the same worker, then the real node's function will be called and the function will return the results; otherwise, the shadow node's function is called, the function will relay the call to the real node on another worker through RPC. line 13 and 15 shows the details.

Figure 2 shows an example of distributed route exchange, where R_1 and R_2 is hosted on $Worker_1$, R_3 is hosted to $Worker_2$. When R_2 on $Worker_1$ requests routes from R_1, which also runs on $Worker_1$, R_2 calls the corresponding function of R_1 directly (1). On the contrary, when R_2 on $Worker_1$ requests routes from R_3, which runs on $Worker_2$, R_2 calls the corresponding function of R_3' (the shadow node for R_2) in the same way as if R_3 is also on $Worker_1$ (2). When the function of R_3' is called, it relays the call to R_3 on $Worker_2$ through the the sidecar on $Worker_1$ (3 → 4 → 5).

With the above design, our distributed framework is decoupled from the simulation logic. That is, S2 does not need to model how switches selects the best routes, or apply route policies. but rather override how switches pull routes from neighbors. Consequently, when the switch model evolves, we only need to make minor adjustments to keep pace.

4.3 Distributed Data Plane Verification

Packet. We represent packets with headers. We use $\mathcal{H}$ to denote the set of all valid packet headers. Each header $h \in \mathcal{H}$ is a bit vector of length $104 + m$, where 104 bits are for the 5-tuple, and m bits are metadata used for checking path-related properties, *e.g.*, waypoint.

Symbolic Packet. A symbolic packet is a set of packets. By assigning one Boolean variable for every bit of the packet header, a symbolic packet can be represented as a Boolean formula over those Boolean variables. That is, for every $i \in [0, 104 + m)$, we use b_i to represent the value (*i.e.*, 0 or 1) of the i-th bit of the header. BDD is the mostly adopted data structure to encode symbolic packets (*e.g.*, [18, 23, 24, 48, 51, 55–57]). Given the huge header space (*i.e.*, 2^{104+m}), state-of-the-art DPVs forward symbolic packets instead of concrete packets to efficiently analyze network forwarding behaviors.

Pre-computing predicates. Before forwarding packets, for each node, S2 assumes the verifier can compute a set of predicates based on the FIB (forwarding information base) and ACLs of the node. There are at least two types of predicates:

- Forwarding predicate. For each port p, it has a forwarding predicate p^{fwd}, representing the packets that can be forwarded out of p.
- ACL predicate. For each port p, it has two predicates p^{in} and p^{out}, representing the packets that are permitted when received at and sent out to port p, respectively.

Forwarding symbolic packets. To check properties, the DPO orchestrates workers to forward symbolic packets through the network. When a switch receives a symbolic packet pkt at port p_1, it forwards a symbolic packet out of each of its ports p_2, with the symbolic packet transformed as:

$$pkt \leftarrow pkt \wedge p_1^{in} \wedge p_2^{fwd} \wedge p_2^{out}. \quad (1)$$

The forwarding of a symbolic packet continues until one of the following *final states* is reached.

(1) Arrive: the packet arrives at the assigned destination node or the node that holds the destination prefix.
(2) Exit: the packet is sent out by another edge port.
(3) Blackhole: the packet matches a rule that drops it.
(4) Loop: the packet traverses the maximum number of hops (*i.e.*, exceeding TTL).

Distributed forwarding of symbolic packets. During traversal, the next hop of a symbolic packet can be on a different worker (*e.g.*, 2 in Figure 3, from p_4 on R_2 to p_5 on R_3), Since the symbolic packet encodes a set of concrete packets with Binary Decision Diagram (BDD), we need to ensure these workers settle on a common BDD encoding. There are two options.

(1) All workers share a logically centralized BDD node table. Since BDD operations need to read or write the BDD node table, only one switch is allowed to perform BDD operations each time. This is similar to the centralized DPV, except we use need to synchronize different copies of the BDD node table.

(2) Each worker has its own BDD node table. In this case, when symbolic packets are forwarded across different workers, the workers need to perform BDD serialization and deserialization (*e.g.*, 3 and 5 in Figure 3), similar to serializing the BDD as a boolean formula on one side, and re-encoding it with BDD on the other side.

We tried both of these options and adopted the latter one since it achieves a higher parallelism and lower memory cost.

- *Higher parallelism.* Symbolic packet forwarding incurs a lot of BDD operations, and each operation needs to read or write the BDD node table. On the contrary, if each worker has its own BDD table, when a switch is performing BDD operation, other switches not on the same worker are not blocked, and can proceed in parallel.
- *Lower memory cost.* BDD node table is memory intensive since each BDD operation may create new nodes, and when the number of nodes exceeds the limit of BDD node table (a threshold at most 2^{32}, due to the maximum integer value), and may eventually saturate the node table. On the contrary, if using multiple BDD engines, one per worker, the size of the node table of each worker can be reduced for a lower memory cost. The reduction of memory cost can also be translated into speedup, due to less garbage collections or table resizing for BDD node table (which can be quite time-consuming).

4.4 Property Checking

S2 assumes the verifier provides a mechanism to let users specify queries, where a query is a 4-tuple (H, V_s, V_d, V_t). $H \subseteq \mathcal{H}$ specifies the checked header space, V_s, V_d and V_t is the set of source, destination, and transit nodes, respectively. Given a query, S2 first converts the header space H into a symbolic packet (*i.e.*, a BDD). Then, it injects the symbolic packet into every source node $v_s \in V_s$. From those source nodes, the symbolic packets will be forwarded through the network until all symbolic packets enter a final state, as introduced in §4.3. For each destination node $v_d \in V_d$, we denote the symbolic packets arriving at it as P_{v_d}.

Currently, S2 supports the following 5 types of queries.

Reachability. After packet forwarding, for each destination node $v_d \in V_d$, the symbolic packet that ARRIVE at v_d (*i.e.*, $pkt \in P_{v_d} \wedge pkt.fs = \text{ARRIVE}$) represents the set of packets that can reach this destination node from one of the source nodes.

Waypoints. Before packet forwarding, for each switch $v_t \in V_t$, we use one bit of the m-bit metadata in the packet header, say b_{v_t}, to indicate whether the packet has visited v_t ($b_{v_t} = 1$) or not ($b_{v_t} = 0$). We also add a "write" rule for switch v_t, which set packets received by it with $b_{v_t} = 1$. When a packet pkt reaches a final state, we check whether $b_{v_t} = 1$ holds, *i.e.*, $pkt \wedge bdd_{v_t} = pkt$, where bdd_{v_t} is the BDD encoding $b_{v_t} = 1$.

Multi-path Consistency. This is a property first introduced by Batfish [19], saying that traffic along all paths from a source should be treated the same. A violation of multi-path consistency may indicate traffic arrives at the destination along one path but encounters a loop along the other. Suppose V_s contains only one source node v_s, and a series of packets $\{pkt_1, \ldots, pkt_n\}$ reach the final state. If there exist two packets pkt_i and pkt_j ($i, j \in [1, n]$) which overlap (*i.e.*, $pkt_i \wedge pkt_j \neq \emptyset$), but have different final states, then we say the multi-path consistency property is violated by packets $pkt_i \wedge pkt_j$ from node v_s.

Loop/Blackhole. If there exists a packet reaching a final state of LOOP or BLACKHOLE, we say the *loop-free* property or the *blackhole-free* property is violated.

4.5 Prefix Sharding

Sometimes, the memory of each server is still insufficient after network partition. Fortunately, we observe that for each protocol, the route computations for different prefixes are *mostly independent*. In such cases, we turn to prefix sharding to further decrease the memory usage during route computation. That is, we divide all prefixes to be computed into multiple independent shards, *i.e.*, the route computations for prefixes in different shards are independent. Then, we can let workers compute routes for one single shard a time, which lowers the peak memory usage during route computation.

To guarantee the correctness of prefix sharding, we have to capture all prefixes of a protocol. The prefixes of a protocol can originate either from internal announcement (*e.g.*, using BGP `network` command), or through redistribution from other protocols. Therefore, when collecting prefixes for protocols, we first collect the self-originated prefixes for each protocol, then add the prefixes of protocol A to those of protocol B, if A is configured to redistribute its routes to B.

To guarantee the correctness of prefix sharding, we also have to account for dependencies among prefixes. That is, the partition should ensure that dependent prefixes are always in the same shard. *Prefix dependency* refers to the route computation of one prefix depends on those of other prefixes. During route computation, prefix dependency behaves as the activation of a route depends on the presence/absence of other routes. Taking BGP as an example, aggregate routes for general prefixes (*e.g.*, 10.1.0.0/16) are activated only when routes for specific prefixes (*e.g.*, 10.1.2.0/24) exist, therefore, these prefixes should fall in the same shard[2].

Prefix Sharding Algorithm. First, we construct a directed prefix dependency graph (DPDG), where each node represents a prefix. On a DPDG, an edge from $prefix_1$ to $prefix_2$ exists if and only if the computation of $prefix_1$ depends on $prefix_2$. That is, either $prefix_1$ is an aggregation prefix covering the specific prefix $prefix_2$, or the announcement of $prefix_1$ relies on the presence or absence of $prefix_2$ in the RIB [1]. Once the DPDG is constructed, we obtain all weakly connected components (CCs) from the DPDG and distribute them into m (the expected number of) shards using a greedy algorithm. Specifically, we arrange the CCs in descending order of size (*i.e.*, the number of prefixes they contain), and initialize m empty shards. Then, we iteratively assign each CC to the currently smallest shard (*i.e.*, the shard that has the fewest number of prefixes). Note that when sorting the CCs by size, we shuffle those with identical sizes to ensure balanced memory usage among workers. This prevents shards from being dominated by prefixes originating from switches assigned to the same worker. Without the shuffling, we observe uneven memory usage across workers when testing S2.

5 EVALUATIONS

In this section, we show the evaluations on S2. We are interested in answering the following questions:

(1) Can S2 verify the configurations of networks with 10K+ switches and 1000M+ routes?
(2) How well can S2 scale out to even larger networks by adding more servers?

[2]In the worst case, if there is a route aggregation that covers all prefixes in the network, then we are not able to do any sharding.

(3) How well is S2's network partition algorithm on balancing workloads among workers while minimizing inter-worker communications?
(4) How do prefix sharding and distributed DPV contribute to the scalability of S2?

5.1 Implementation

We implement a prototype of S2 on top of Batfish with ~12K LOC of Java. Batfish is an excellent open-source verifier that can simulate the control plane of around 10 vendors, including Cisco, Juniper, and Arista. It has been widely used and tested by researchers and operators. Its contributors are still actively improving the accuracy of existing switch models, as well as adding accurate switch models for other vendors. To enjoy improved accuracy, we adapt Batfish to our distributed framework in a non-intrusive way. Specifically, we use sub-classing to minimize the modifications to the original Batfish code base (see §3.2).

For the controller, we use the parser of Batfish to convert configuration files into VI (vendor-independent) representations. We use METIS [3], a well-known multi-level graph partition algorithm, to partition the network. For workers, we use the route computation model in Batfish [18]. Specifically, we extended the BGP and OSPF classes in Java, to realize the real and shadow BGP and OSPF nodes. In total, we modified ~500 LOC of the original Batfish. For communication among the controller and workers, we use gRPC [25]. Instead of using ProtoBuf [4], we use Java to serialize and deserialize messages, since classes in Batfish are serializable with Java without modifications. We expect a further improvement by adopting ProtoBuf. For the serialization of BDD, we use the BDDIO of JDD library [46].

5.2 Datasets and Setup

Datasets. We use both real and synthesized configurations.

- *The real DCN configurations* are from a large service provider. The network has $O(16K)$ nodes running BGP, producing $O(200M)$ IPv4 routes in total. More details about the DCN can be found in §2.3.
- *The synthesized FatTree configurations* are constructed using a script from ACORN [39]. We modified the IP assignments in the script so as to generate large FatTrees (*e.g.*, FatTree90). Using the script, we synthesized different sizes of FatTrees running BGP, where every switch has a unique AS number and forms eBGP peer sessions with its connected switches. The FatTrees are ECMP enabled, where every switch can have up to 64 equal-cost paths for a prefix.

Setup. We compare S2 with Batfish [18] and Bonsai [11]. For Bonsai, it was originally implemented on top of an old version of Batfish, and we adapted it to the Batfish version that S2 is built on. The experiments are run on five physical Linux servers, each with two 32-core Intel Xeon Platinum 8336C CPUs @ 2.30GHz and 500G memory. We use the command `taskset` and the JVM option `-Xmx` to divide each server into four logical servers, each with 15 cores and 100GB memory. For Batfish and Bonsai, we use one logical server. For S2, we use one logical server for the controller, and up to 16 logical servers for workers, The logical server running the controller resides on a separate physical server other than those four servers for workers.

Unless otherwise specified, we check all-pair-reachability to verify a network, and the peak memory usages reported in the following refer to per-worker peak memory usages.

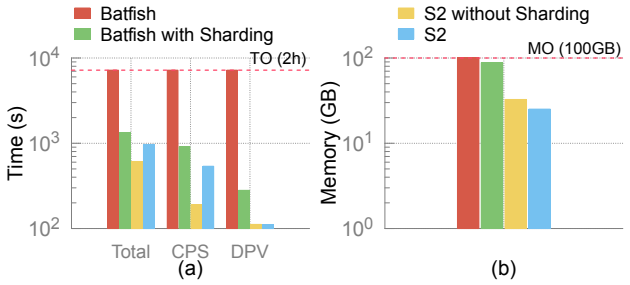

Figure 4: (a) running time and (b) peak memory usage of using Batfish, Batfish with prefix sharding, S2 without prefix sharding, and S2 to verify the real DCN.

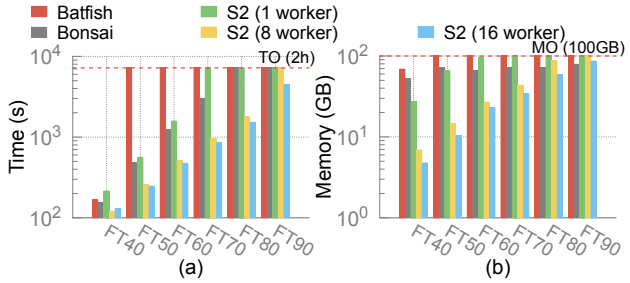

Figure 5: (a) running time and (b) peak memory usage when using Batfish, Bonsai, and S2 (with 1, 8, and 16 workers) to verify different sizes of FatTrees.

5.3 Results for Real Datacenter Network

We run both S2 and Batfish on the real DCN (*i.e.*, using real configurations on the real topology), and they output the same set of RIBs, which consists of $O(2 \times 10^8)$ routes in total. Figure 4 shows the performance of Batfish and S2 to verify the real datacenter network. It shows that vanilla Batfish runs out of memory during route computation. Enabling prefix sharding (20 shards) helps Batfish finish the verification, but the memory is still approaching the limit. In comparison, S2 can finish the verification in 16 minutes with 35GB memory,

As shown in Figure 4(a), prefix sharding can slow down the control plane simulation of S2. This is because prefix sharding has some overhead in simulation time due to the sequential simulation of multiple shards. When the memory is the bottleneck, prefix sharding can avoid the time-consuming garbage collections, and thereby can speedup the simulation time. However, when the memory is sufficient, the overhead due to prefix sharding is not paid off, making the simulation even slower.

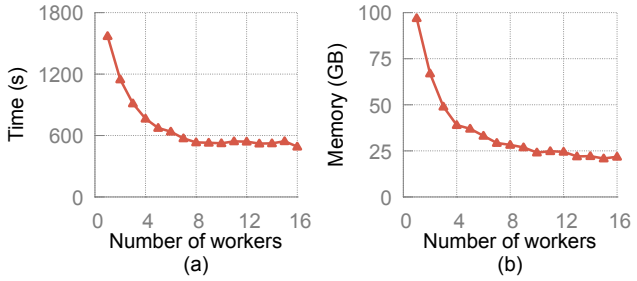

Figure 6: (a) time and (b) peak memory usage of S2 to verify FatTree60 (4500 switches), with different numbers of workers.

5.4 Results for Synthesized FatTrees

We evaluate the performance of S2 (1, 8, and 16 workers) on different sizes of FatTrees, and compare the results with those of Bonsai [11] and Batfish. For S2, we divide the prefixes into 20 prefix shards for each size of FatTrees. Bonsai compresses networks based on the destination. If given a specific destination (*i.e.*, the prefix announced by an edge switch), Bonsai can compress a synthesized FatTree of an arbitrary k into only 6 nodes[3]. However, if given a wildcard destination (*i.e.*, 0.0.0.0/0), Bonsai cannot compress it. Therefore, to check all-pair-reachability on a synthesized FatTree, we apply Bonsai to compress the topology for each prefix, and run Batfish on the compressed topology to check the reachability for that prefix, in a parallel way.

As shown in Figure 5, Batfish can only scale to somewhere between FatTree40 (2000 switches) and FatTree50 (3125 switches), limited by the memory resource of a single logical server. Bonsai scales better than Batfish, but only to somewhere between FatTree70 (6125 switches) and FatTree80 (8000 switches), limited by the compute resource (*i.e.*, number of cores), rather than the memory resource, of a single logical server. This is because Bonsai is memory efficient by compressing FatTree into smaller size (6 nodes for all sizes of synthesized FatTree). However, it times out on hyper-scale FatTrees since the compression time for each destination increases with the FatTree size, and the number of destinations far exceeds the number of cores of a single logic server. S2 can scale to FatTree60 (4500 switches) with 1 worker, FatTree80 (8000 switches) with 8 workers, and FatTree90 (10125 switches) with 16 workers. Even with a single worker, S2 scales better than Batfish, mainly due to the use of prefix sharding. Besides being more scalable, S2 is also faster and more efficient in memory usage.

We can see that the time and peak memory usage of S2 are much higher for the FatTree90 (10125 switches) topology than our DCN (Figure 4), even though they have similar number of switches. This is because in the DCN, the operators have configured route aggregations, such that the total number of routes ($O(2 \times 10^8)$) is only 1/10 that of FatTree90 ($O(3 \times 10^9)$). In terms of the number of routes, the DCN is closer to FatTree50 with $O(2 \times 10^8)$ routes, but in terms of running time and memory cost, the performance for DCN is closer to that of FatTree70, perhaps due to rich routing policies in the DCN. The above comparisons indicate that the verification cost for S2 is influenced by three factors: (1) the number of switches, (2) the number of routes, and (3) configuration complexity.

[3]The 6 nodes include: (1) one edge switch for the destination, (2) one core switch, (3) one edge switch and one aggregation switch in the same pod as the destination; (4) one edge switch and one aggregation switch in a different pod as the destination.

5.5 Scaling Out with More Workers

We continue to evaluate how S2 can scale out with more workers. We use FatTree60 (4500 switches), and vary the number of workers from 1 to 16.

Figure 6 shows the result. In general, the running time and peak memory usage decrease when we add more workers. However, they only decrease rapidly when there are less than 8 workers. This is because when the number of workers is relatively small compared to the FatTree size, the computation and memory resource on each worker is insufficient, therefore the addition of every worker matters. In contrast, when the number of workers is relatively large compared to the FatTree size, the computation and memory resource on each worker is sufficient, thus S2 only benefits a little from adding more workers.

5.6 Comparison of Different Network Partition Schemes

We evaluate S2 with different network partition schemes, *i.e.*, "random", "expert", and "metis". For the random scheme, we shuffle all switches evenly into different segments. For the expert scheme, we use different strategies for different topologies. Taking FatTrees for example, the expert scheme will always assign the aggregation and edge switches of the same pod into the same segment, and assign the core switches evenly to different segments. For the real DCN topology, we sort all switches according to their name, and then evenly assign them into segments. This is based on the experience that switches whose names have similar prefixes are more likely to be adjacent on the topology. For the metis scheme, we use the METIS algorithm [3] to divide the topology, with the balancing of workers' load as the primary goal and the minimization of inter-worker communication overhead as the secondary goal.

As shown in Figure 7, the running times and memory usages using these three schemes only differ slightly. Especially, the most communication-heavy partition scheme (*i.e.*, random) only leads to slightly worse performance, compared to the other two partition schemes. We hypothesize that this is because the performance of S2 is not sensitive to inter-worker communication overhead, but mostly depends on whether the loads among workers are balanced. We further confirm this by testing two extreme partitions. One scheme is to partition the network into *load-imbalanced* segments (*i.e.*, 3/4 switches of the network in one segment, while the remaining switches evenly distributed to other segments), which performs far worse than the above three schemes. Another scheme is to partition the network into *communication-heaviest* segments (*i.e.*, for FatTrees, core and edge switches in some segments, while aggregate switches on the other segments), which performs slightly worse than the random scheme.

5.7 Effectiveness of Prefix Sharding

We evaluate whether prefix sharding is necessary for verifying extremely large networks. As shown in Figure 8, for FatTrees smaller than FatTree80 (8000 switches), S2 uses less time and memory by

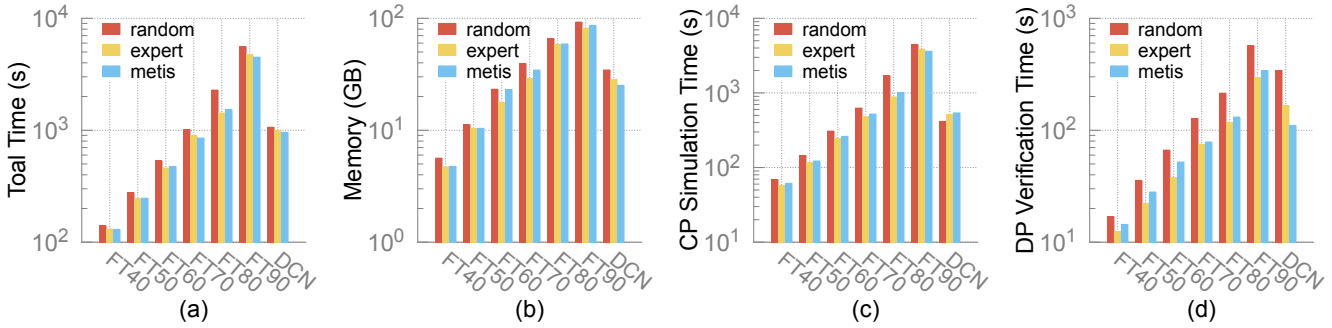

Figure 7: (a) total running time; (b) peak memory usage; (c) control plane simulation time; and (d) data plane verification time of S2 to verify different networks using different network partition schemes.

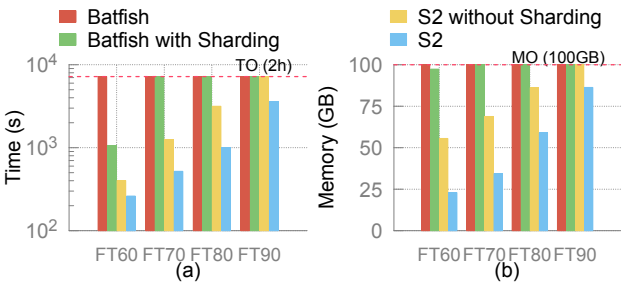

Figure 8: (a) time and (b) peak memory usage to simulate FatTree60, 70, 80, and 90 with different schemes.

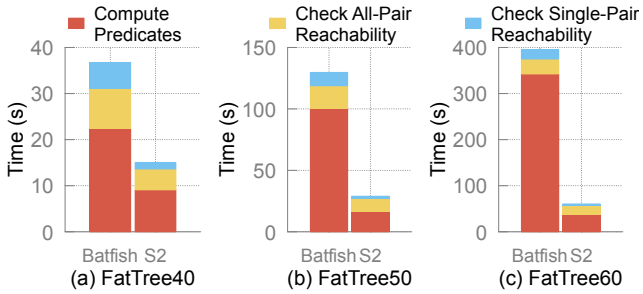

Figure 10: Time to verify all-pair-reachability and single-pair-reachability on (a) FatTree40 (b) FatTree50, and (c) FatTree60 using Batfish and S2.

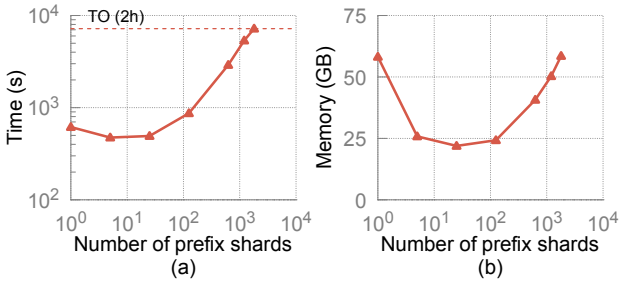

Figure 9: (a) time and (b) peak memory usage of S2 to simulate FatTree60 (4500 switches) with different number of prefix shards.

enabling prefix sharding; For FatTrees larger than FatTree90 (10125 switches), enabling prefix sharding becomes necessary to finish the simulation.

We continue to evaluate the scalability of prefix sharding in control plane simulation. We run S2 to simulate FatTree60 (4500 switches), with different numbers of prefix shards. Figure 9 shows the result. When memory is insufficient (*i.e.*, <25 shards), the simulation time decreases as the number of shards increases. The reason is that as the number of prefixes decreases in each shard, the peak memory usage drops, avoiding many costly garbage collections. When the memory is sufficient (*i.e.*, >25 shards), the simulation time increases with the number of shards.

5.8 Effectiveness of Distributed DPV

To evaluate the effectiveness of distributed packet forwarding in data plane verification, we compare the time to check all-pair-reachability and single-pair-reachability on FatTree40 (2000 switches) to FatTree60 (4500 switches) using Batfish and S2. Note that to evaluate the DPV of Batfish on FatTree50 and FatTree60, we enable prefix sharding on it to let it successfully generate FIBs.

As shown in Figure 10, S2 is faster than Batfish for both all-pair-reachability and single-pair-reachability checking. The checking time consists of two parts, *i.e.*, the time to compute forwarding and ACL predicates, and the time to forward packets then check all-pair reachability or single-pair-reachability. We can see that S2 is faster than Batfish in total and separate phases. Moreover, the speedup of S2 compared to Batfish increases with the FatTree sizes. This result aligns with our claim that maintaining one BDD node table on each worker can greatly improve the BDD operation parallelism and reduce the times of BDD node table garbage collections.

In detail, S2 speeds up the first phase (*i.e.*, compute predicates) the most (up to 10 times), since switches on different workers can compute their predicates in parallel. However, it may be counterintuitive that S2 can also speed up the reachability checking so much. This is because even checking single-pair reachability between

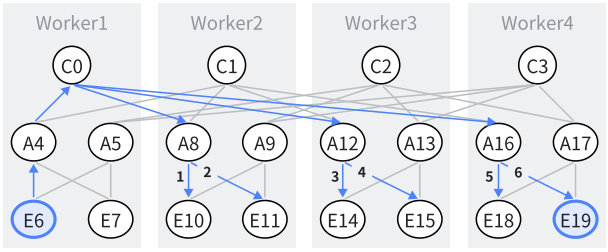

Figure 11: Packet forwarding when checking reachability from E_6 to E_{19} on FatTree4. Numbers on the edges indicate indices for forwarding steps.

two edge switches from two different pods will trigger packet forwarding on all workers. Take Figure 11 for example, which shows the process of checking reachability from edge switch E_6 to edge switch E_{19} on four workers[4]. After arriving at the core switch C_0, the packet will be copied and forwarded to all other workers to exhaustively find all possible paths from E_6 to E_{19}. While Batfish can only execute Step 1-6 sequentially, S2 is capable of executing steps on different workers (*e.g.*, Step 1 and 3) in parallel, thus accelerating the reachability checking process. One may also restrict C_0 to only forward to worker4, knowing that E_{19} resides on it. However, we think of finding all available forwarding paths valuable to find path-specific forwarding anomalies (*e.g.*, forwarding valley such as $E_6 \rightarrow A_4 \rightarrow C_0 \rightarrow A_8 \rightarrow E_{10} \rightarrow A_9 \rightarrow C_3 \rightarrow A_{17} \rightarrow E_{19}$)

6 RELATED WORK

6.1 Network Emulators

[2, 21, 27, 32] run vendor-specific switch software in emulated environments and generate the corresponding FIBs. However, emulating switch images is resource-intensive, making network emulation not scalable. Therefore, some emulators seek ways to reduce the number of switches to emulate. Unfortunately, there is no panacea for all networks and configurations; for example, the heuristic of CrystalNet [32] to find a safe emulation boundary is inapplicable to Crescent [21].

6.2 Control Plane Verifiers (CPVs)

Simulation-based CPVs run switch models rather than the switch images, and therefore uses much less computing and memory resources. For example, network verifiers like Batfish [14, 19], Plankton [38], and DNA [55] execute route computation models to generate routes and forwarding rules, which can then be checked with data plane verifiers (DPVs) [28, 30, 31, 36, 51, 56]. ShapeShifter [12], Hoyan [52], and SRE [57] use abstract interpretation or symbolic execution to make the simulation more efficient. FastPlane [34] significantly speeds up the simulation by assuming the routes are monotone. However, none of them are reported to scale to networks as large as 10K switches.

[4]Batfish supports both forward and backward traversing to check properties; we only show forward traversing here for simplicity.

Analysis-based CPVs [5, 10, 22, 49] directly check properties based on customized control plane models (*e.g.*, graph [5, 22], SMT constraints [10, 49], etc.), without simulating route computations and generating the data plane. Compared with simulation-based verifiers, analysis-based verifiers can efficiently reason about arbitrary link failures. However, their scalability in terms of network size is also limited [10], or they trade off some accuracy or feature coverage for scalability [5, 22].

Modular CPVs [6, 44, 45] verify network configurations by dividing the network into components that can be verified in isolation, and verifying those components in parallel. Specifically, they ask users to define *interfaces* between components that describe each component's routing behavior, then verify whether each component respects its interface in parallel. If so, we can conclude that the monolithic network satisfies the property (*e.g.*, reachability) that the interfaces imply. Kirigami [45] proposed an architecture for modular control plane verification, but restricted its interfaces to only exact routes. Lightyear [44] supports more expressive interfaces, but can only check that a network never receives a route (*e.g.*, for access control policies) — it cannot check reachability, a keen property of interest. Timepiece [6] proposed a modular technique that can verify a wide range of properties (including reachability) by defining abstract network interfaces. However, it requires much more work from users, *i.e.*, define interfaces that characterize the routes each network component may generate at each time.

6.3 Data Plane Verification

Data plane verification has been studied for years and fruitful [9, 28, 30, 31, 33, 51, 56], with earlier DPVs focusing on checking forwarding properties (*e.g.*, HSA [30] and Anteater [36]) and later DPVs focusing on being faster (*e.g.*, Veriflow [31] and AP [51]), incremental (*e.g.*, APKeep [56] and Delta-net [28]), and supporting multi-layer networks (*e.g.*, Katra [9] and MNV [33]). However, they are all unscalable due to their centralized architecture (*i.e.*, limited by the CPU and memory resources of a single machine).

Some DPVs embrace a distributed design to scale, *e.g.*, Libra [54] and Tulkun [50]. Libra exploits the scaling properties of MapReduce [17], and scales to analyze a hyper-scale network with 10K switches within a minute, using 50 servers. Note that Libra also proposes to divide forwarding rules into shards. However, the sharding of Libra is a pure partition of a large set of forwarding rules, without considering dependencies among them. In contrast, we take dependencies among prefixes into consideration when sharding prefixes and guarantee that one prefix is in the same shard of the prefixes it depends on if they exist (§4.5). Tulkun is an on-switch verifier that transforms data plane verification into a counting problem on a directed acyclic graph (DAG) called DPVNet. It can verify loop-free, blackhole-free, and all-pair reachability on a real 6K-switch DC network in 40 seconds. However, a DPVNet is essentially a DAG that contains all topological paths between two switches, whose construction is centralized and unscalable. Specifically, for a FatTree with k pods, the complexity of building a DPVNet for two edge switches in different pods is much higher than $O(k^2)$ (*i.e.*, the number of shortest paths between those two edge switches).

Some DPVs exploit the characteristics of datacenters to scale, *e.g.*, [37] uses the symmetry and surgery of datacenters to transform

large networks into smaller ones. It can scale to datacenters with about 100K switches. However, instead of proactively detecting misconfigurations before adopting them in the network, data plane verifiers focus on auditing FIBs and finding forwarding property violations. Thus, we cannot use them although they are extremely scalable.

7 DISCUSSION

Parallel and Distributed Strategies. Our current design runs switches on multiple workers in parallel, while executing prefix shards in multiple rounds. In each round, all switches execute the same prefix shard. As an alternative, we can create multiple nodes for each switch, such that different nodes of the same switch can execute different prefix shards in parallel. Such prefix parallelism is orthogonal to the existing switch parallelism, where multiple switches run in parallel. Furthermore, we can distribute nodes of the same switch across multiple workers, to reduce the running time.

Centralized simulators like Batfish have already implemented switch parallelism to speedup simulation. Specifically, Batfish lets multiple switches compute their routes in parallel, one thread for each switch. However, the maximum number of threads of a single server, compared to the number of switches of a hyper-scale network, is far from sufficient, making the scalability of the centralized simulators' parallel execution limited.

Control Plane Simulation vs. Discrete-Event Simulation. Discrete-Event Simulator [29, 35, 41, 47] simulates packet forwarding to evaluate the performance of network algorithms like congestion control and packet scheduling. Recently, DONS [20] and UNISON [8] proposed new designs like data-oriented design to make DES more scalable. Even though DES is orthogonal to control plane simulation, these efforts in designing scalable DES may shed light on scaling control plane simulation.

Correctness of Prefix Sharding. When sharding prefixes, we take dependencies among prefixes into consideration, such that a prefix will always be in the same shard as the prefixes it depends on (§4.5). However, this may miss some unforeseen dependencies that only emerge during route computation. We argue that S2 can avoid the false positives or false negatives due to unforeseen dependencies with simple extensions. Specifically, S2 can collect prefix dependencies when computing routes, and when some unforeseen prefix dependencies emerge, S2 can refine and recompute the affected shards. For example, if $prefix_1 \in shard_1$ is found to depend on $prefix_2 \in shard_2$ at runtime, S2 can merge these two shards and recompute routes for the new shard (*i.e.*, $shard_1 \cup shard_2$).

Limitations. S2 assumes the network converges, after a smaller number of rounds, to one or multiple states; otherwise, S2 cannot terminate. When there are multiple converged state, S2 can only converge to one such state. Finally, S2 now only supports IPv4, and we are still working to support IPv6.

8 CONCLUSION

This paper presents S2, a distributed network configuration verifier capable of scaling to networks with over 10K switches and 1000M routes. Instead of scaling up with a single server, S2 chooses to scale out with multiple servers. By partitioning the network into multiple segments and run them on multiple servers, S2 observes a reduced memory cost, lower computation load, and increased parallelism at each server. To validate its effectiveness, we plugged the models of Batfish into the framework of S2, with minor code modification, and showed it can verify our DCN with 16K switches in 16 minutes, and scale to FatTrees with 10K switches and 1000M routes with 16 workers. Our future work includes (1) supporting the simulation of route computation for IPv6, and (2) increasing the number of workers to further test its scalability.

Acknowledgement. We thank our shepherd and the anonymous SIGCOMM reviewers for their valuable comments and suggestions. This work is supported by the National Key Research and Development Program of China (No. 2022YFB2901403) and by the National Natural Science Foundation of China (No. 62272382 and No. 62172323). Peng Zhang is the corresponding author of this paper.

This work does not raise any ethical issues.

REFERENCES

[1] Configure and Verify the BGP Conditional Advertisement Feature. https://www.cisco.com/c/en/us/support/docs/ip/border-gateway-protocol-bgp/16137-cond-adv.html.
[2] Graphical network simulator-3 (GNS3). https://gns3.com.
[3] Metis. http://glaros.dtc.umn.edu/gkhome/views/metis.
[4] Protocol Buffers (ProtoBuf). https://protobuf.dev/.
[5] Anubhavnidhi Abhashkumar, Aaron Gember-Jacobson, and Aditya Akella. Tiramisu: Fast and general network verification. In *USENIX NSDI*, 2020.
[6] Timothy Alberdingk Thijm, Ryan Beckett, Aarti Gupta, and David Walker. Modular control plane verification via temporal invariants. *Proceedings of the ACM on Programming Languages*, 7(PLDI):50–75, 2023.
[7] Konstantin Andreev and Harald Räcke. Balanced graph partitioning. In *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 120–124, 2004.
[8] Songyuan Bai, Hao Zheng, Chen Tian, Xiaoliang Wang, Chang Liu, Xin Jin, Fu Xiao, Qiao Xiang, Wanchun Dou, and Guihai Chen. Unison: A parallel-efficient and user-transparent network simulation kernel. In *Proceedings of the Nineteenth European Conference on Computer Systems*, pages 115–131, 2024.
[9] Ryan Beckett and Aarti Gupta. Katra: Realtime verification for multilayer networks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 617–634, 2022.
[10] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. A general approach to network configuration verification. In *ACM SIGCOMM*, 2017.
[11] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. Control plane compression. In *ACM SIGCOMM*, 2018.
[12] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. Abstract interpretation of distributed network control planes. In *ACM POPL*, 2020.
[13] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. Don't mind the gap: Bridging network-wide objectives and device-level configurations. In *ACM SIGCOMM*, 2016.
[14] Matt Brown, Ari Fogel, Daniel Halperin, Victor Heorhiadi, Ratul Mahajan, and Todd Millstein. Lessons from the evolution of the batfish configuration analysis tool. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 122–135, 2023.
[15] Sean Choi, Boris Burkov, Alex Eckert, Tian Fang, Saman Kazemkhani, Rob Sherwood, Ying Zhang, and Hongyi Zeng. Fboss: building switch software at scale. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 342–356, 2018.
[16] Charles Clos. A study of non-blocking switching networks. *Bell System Technical Journal*, 32(2):406–424, 1953.
[17] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.
[18] Ari Fogel. Batfish. https://github.com/batfish/batfish.
[19] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. A general approach to network configuration analysis. In *USENIX NSDI*, 2015.
[20] Kaihui Gao, Li Chen, Dan Li, Vincent Liu, Xizheng Wang, Ran Zhang, and Lu Lu. Dons: Fast and affordable discrete event network simulation with automatic parallelization. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 167–181, 2023.
[21] Zhaoyu Gao, Anubhavnidhi Abhashkumar, Zhen Sun, Weirong Jiang, and Yi Wang. Crescent: Emulating heterogeneous production network at scale. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI*

24), Santa Clara, CA, April 2024. USENIX Association.

[22] Aaron Gember-Jacobson, Raajay Viswanathan, Aditya Akella, and Ratul Mahajan. Fast control plane analysis using an abstract representation. In *ACM SIGCOMM*, 2016.

[23] Nick Giannarakis, Devon Loehr, Ryan Beckett, and David Walker. NV: an intermediate language for verification of network control planes. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*, 2020.

[24] Nick Giannarakis, Alexandra Silva, and David Walker. Probnv: probabilistic verification of network control planes. *Proc. ACM Program. Lang.*, 5(ICFP), 2021.

[25] Google. gRPC. https://grpc.io/.

[26] Ramesh Govindan, Ina Minei, Mahesh Kallahalla, Bikash Koley, and Amin Vahdat. Evolve or die: High-availability design principles drawn from googles network infrastructure. In *ACM SIGCOMM*, 2016.

[27] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, Bob Lantz, and Nick McKeown. Reproducible network experiments using container-based emulation. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '12, page 253–264, New York, NY, USA, 2012. Association for Computing Machinery.

[28] Alex Horn, Ali Kheradmand, and Mukul R Prasad. Delta-net: Real-time network verification using atoms. In *USENIX NSDI*, 2017.

[29] Teerawat Issariyakul, Ekram Hossain, Teerawat Issariyakul, and Ekram Hossain. *Introduction to network simulator 2 (NS2)*. Springer, 2009.

[30] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *USENIX NSDI*, 2012.

[31] Ahmed Khurshid, Wenxuan Zhou, Matthew Caesar, and P Godfrey. VeriFlow: Verifying network-wide invariants in real time. In *USENIX NSDI*, 2013.

[32] Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Jiaxin Cao, Sri Tallapragada, Nuno P Lopes, Andrey Rybalchenko, Guohan Lu, and Lihua Yuan. Crystalnet: Faithfully emulating large production networks. In *ACM SOSP*, 2017.

[33] Xu Liu, Peng Zhang, Hao Li, and Wenbing Sun. Modular data plane verification for compositional networks. *Proceedings of the ACM on Networking*, 1(CoNEXT3):1–22, 2023.

[34] Nuno P Lopes and Andrey Rybalchenko. Fast BGP simulation of large datacenters. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, 2019.

[35] Zheng Lu and Hongji Yang. *Unlocking the power of OPNET modeler*. Cambridge University Press, 2012.

[36] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P Godfrey, and Samuel Talmadge King. Debugging the data plane with Anteater. In *ACM SIGCOMM*, 2011.

[37] Gordon D Plotkin, Nikolaj Bjørner, Nuno P Lopes, Andrey Rybalchenko, and George Varghese. Scaling network verification using symmetry and surgery. In *ACM POPL*, 2016.

[38] Santhosh Prabhu, Kuan-Yen Chou, Ali Kheradmand, P Godfrey, and Matthew Caesar. Plankton: Scalable network configuration verification through model checking. In *USENIX NSDI*, 2020.

[39] Divya Raghunathan. Synthesized Fattrees. https://github.com/divya-urs/ACORN_benchmarks.

[40] Sivaramakrishnan Ramanathan, Ying Zhang, Mohab Gawish, Yogesh Mundada, Zhaodong Wang, Sangki Yun, Eric Lippert, Walid Taha, Minlan Yu, and Jelena Mirkovic. Practical intent-driven routing configuration synthesis. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 629–644, 2023.

[41] George F Riley and Thomas R Henderson. The ns-3 network simulator. In *Modeling and tools for network simulation*, pages 15–34. Springer, 2010.

[42] Samuel Steffen, Timon Gehr, Petar Tsankov, Laurent Vanbever, and Martin Vechev. Probabilistic verification of network configurations. In *ACM SIGCOMM*, 2020.

[43] Yu-Wei Eric Sung, Xiaozheng Tie, Starsky H. Y. Wong, and Hongyi Zeng. Robotron: Top-down network management at facebook scale. In *ACM SIGCOMM*, 2016.

[44] Alan Tang, Ryan Beckett, Karthick Jayaraman, Todd Millstein, and George Varghese. Lightyear: Using modularity to scale bgp control plane verification. *arXiv preprint arXiv:2204.09635*, 2022.

[45] Timothy Alberdingk Thijm, Ryan Beckett, Aarti Gupta, and David Walker. Kirigami, the verifiable art of network cutting. *IEEE/ACM Transactions on Networking*, 32(3):2447–2462, 2024.

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

[47] Andras Varga. A practical introduction to the omnet++ simulation framework. In *Recent Advances in Network Simulation: The OMNeT++ Environment and its Ecosystem*, pages 3–51. Springer, 2019.

[48] Dan Wang, Peng Zhang, and Aaron Gember-Jacobson. Expresso: Comprehensively reasoning about external routes using symbolic simulation. In *Proceedings of the ACM SIGCOMM 2024 Conference*, pages 197–212, 2024.

[49] Konstantin Weitz, Doug Woos, Emina Torlak, Michael D Ernst, Arvind Krishnamurthy, and Zachary Tatlock. Scalable verification of border gateway protocol configurations with an SMT solver. In *ACM OOPSLA*, 2016.

[50] Qiao Xiang, Chenyang Huang, Ridi Wen, Yuxin Wang, Xiwen Fan, Zaoxing Liu, Linghe Kong, Dennis Duan, Franck Le, and Wei Sun. Beyond a centralized verifier: Scaling data plane checking via distributed, on-device verification. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 152–166, 2023.

[51] Hongkun Yang and Simon S Lam. Real-time verification of network properties using atomic predicates. In *IEEE ICNP*, 2013.

[52] Fangdan Ye, Da Yu, Ennan Zhai, Hongqiang Harry Liu, Bingchuan Tian, Qiaobo Ye, Chunsheng Wang, Xin Wu, Tianchen Guo, Cheng Jin, et al. Accuracy, scalability, coverage: A practical configuration verifier on a global WAN. In *ACM SIGCOMM*, 2020.

[53] Ken Yocum, Ethan Eade, Julius Degesys, David Becker, Jeff Chase, and Amin Vahdat. Toward scaling network emulation using topology partitioning. In *11th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer Telecommunications Systems, 2003. MASCOTS 2003.*, pages 242–245. IEEE, 2003.

[54] Hongyi Zeng, Shidong Zhang, Fei Ye, Vimalkumar Jeyakumar, Mickey Ju, Junda Liu, Nick McKeown, and Amin Vahdat. Libra: Divide and conquer to verify forwarding tables in huge networks. In *USENIX NSDI*, 2014.

[55] Peng Zhang, Aaron Gember-Jacobson, Yueshang Zuo, Yuhao Huang, Xu Liu, and Hao Li. Differential network analysis. In *USENIX NSDI*, 2022.

[56] Peng Zhang, Xu Liu, Hongkun Yang, Ning Kang, Zhengchang Gu, and Hao Li. APKeep: Realtime verification for real networks. In *USENIX NSDI*, 2020.

[57] Peng Zhang, Dan Wang, and Aaron Gember-Jacobson. Symbolic router execution. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 336–349, New York, NY, USA, 2022. Association for Computing Machinery.

[58] Huaiyi Zhao, Xinyi Zhang, Yang Wang, Zulong Diao, Yanbiao Li, and Gaogang Xie. Improving the scalability of distributed network emulations: an algorithmic perspective. *IEEE Transactions on Network and Service Management*, 20(4):4325–4339, 2023.