Network Can Check Itself: Scaling Data Plane Checking via Distributed, On-Device Verification

Qiao Xiang[†‡], Ridi Wen[†], Chenyang Huang[†], Yuxin Wang[†], Franck Le[◇],
[†]School of Informatics, Xiamen University, [‡]IKKEM, [◇]IBM Research

ABSTRACT

Current data plane verification (DPV) tools employ a centralized architecture, where a server collects the data planes of all devices and verifies them. This architecture is inherently unscalable (*i.e.*, requiring a reliable management network, incurring a long control path and making the server a single point of failure). In this paper, we tackle this scalability challenge of DPV from an architectural perspective. In particular, we circumvent the scalability bottleneck of centralized design and advocate for a distributed, on-device DPV framework. Our key insight is that DPV can be transformed into a counting problem on DAG, which can be naturally decomposed into lightweight tasks executed at network devices, enabling scalability. Evaluation shows that a prototype of this framework achieves scalable DPV under various settings, with little overhead on commodity network devices.

CCS CONCEPTS

• **Networks** → **Protocol testing and verification**; **Network reliability**;

KEYWORDS

Network verification, Distributed verification

ACM Reference Format:

Qiao Xiang, Ridi Wen, Chenyang Huang, Yuxin Wang, Franck Le. 2022. Network Can Check Itself: Scaling Data Plane Checking via Distributed, On-Device Verification . In *The 21st ACM Workshop on Hot Topics in Networks (HotNets '22), November 14–15, 2022, Austin, TX, USA*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3563766.3564095

1 INTRODUCTION

There has been a long line of research on data plane verification (DPV). Earlier tools analyzed a snapshot of the complete network data plane [1, 2, 14, 17–19, 23, 24, 28, 32, 33, 35, 36, 38–40]; and more recent solutions focus on incremental verification [14, 16, 17, 19, 41]. State-of-the-art DPV tools (*e.g.*, [41]) can achieve incremental verification times of 10s of μs per rule update. Despite such progress, existing tools use a centralized design, which lacks the scalability needed for deployment in large networks. They use a centralized server to collect the data plane from each network device and verify the invariants. This requires a management network to provide reliable, low-latency connections between the server and devices, which is hard to build for large-scale networks [10]. Moreover, the server becomes the performance bottleneck and single point of failure, *e.g.*, in our test, it takes APKeep [41] ~1 hour to verify a 48-ary fattree.

To scale up DPV, some studies propose to divide DPV into different packet spaces [17, 40] or network partitions [16] to achieve parallel verification in a cluster of servers. For example, RCDC [16] partitions DPV by device to verify the availability of all shortest paths in parallel in a cluster. However, they are still centralized designs with the limitations above, and RCDC can only verify that particular requirement.

This paper aims to tackle how to scale DPV to be applicable in real, large networks. Not only can a scalable DPV tool quickly find network errors in large networks, it can also support novel services such as convergence-free routing [20, 30], real-time control plane repair [13], fast rollback and switching among multiple data planes [7, 21, 31], and data plane verification across domains [8, 37].

Proposal: offloading verification to distributed computations on network devices. Instead of continuing to squeeze incremental performance improvements out of centralized DPV, we tackle the scaling issue of DPV from an architecture perspective. In particular, we embrace a distributed design to circumvent the inherent scalability bottleneck of centralized design. RCDC [16] takes a first step along this direction by partitioning DPV into local contracts of devices. It gives an interesting analogy between such local contracts and program verification using annotation with inductive loop invariants, but stops at designing communication-free local contracts for the particular all-shortest-path availability invariant and validating it in parallel on a centralized cluster. In contrast, we go beyond this and show that a wide range of requirements can be verified in a compositional way by running lightweight tasks distributively on commodity network devices, achieving scalable DPV in generic settings. To our best knowledge, this is the first step toward distributed, on-device DPV.

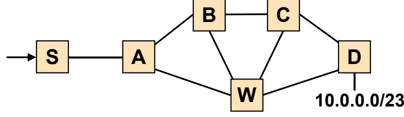
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

HotNets '22, November 14–15, 2022, Austin, TX, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9899-2/22/11…$15.00
https://doi.org/10.1145/3563766.3564095

Requirement: all packets entering the network from *S* with a destination IP in 10.0.0.0/23 must be delivered to *D* in a simple path waypointing *W*.

(S.*W.*D) ∩ $\cap_{\text{device X}}$ (([^X]*) ∪ ([^X]*[X][^X]*))

Topology:



(a) An example topology and requirement.

S

Match	Action
10.0.0.0/23	fwd(ALL, {A})

A

Match	Action
10.0.0.0/24	fwd(ANY, {B, W})
10.0.1.0/24	fwd(ALL, {W})

B

Match	Action
10.0.0.0/23	fwd(ALL, {C})

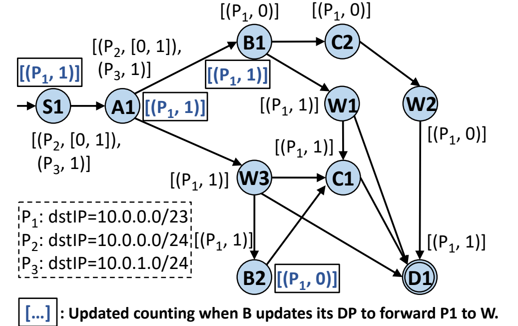
W

Match	Action
10.0.0.0/23	fwd(ALL, {C})

C

Match	Action
10.0.0.0/23	fwd(ALL, {D})

(b) The network data plane.



(c) The *DPVNet* and the counting process.

Figure 1: An illustration example to demonstrate the workflow of distributed, on-device DPV.

To be concrete, we design our distributed, on-device DPV framework with a key insight: the problem of DPV can be transformed into a counting problem on a DAG representing all valid paths in the network; the latter can be decomposed into small tasks at nodes on the DAG and distributively executed at corresponding network devices, enabling scalability.

Scope of invariants. Our proposal can verify "single-path" invariants that require packets' traces should satisfy certain patterns (*e.g.*, reachability, waypoint, blackhole freeness, isolation, multicast and anycast). We discuss how to efficiently verify path-comparison invariants (*e.g.*, node disjointness and route symmetry) and other open questions in §4.

Evaluation. We implement a prototype called Coral and evaluate it using real-world datasets in both testbed and simulations. We find Coral consistently outperforms state-of-the-art DPV tools under various networks (WAN/LAN/DC) and DPV scenarios, *i.e.*, up to 813× speed up in burst update, and up to 243× speed up on 80% quantile of incremental verification, with little overhead on commodity network devices. Coral achieves scalable DPV for two reasons: (1) by decomposing DPV into lightweight tasks executed on devices, its performance achieves a scalability approximately linear to the network diameter; (2) when a rule update happens, only devices whose task results may change need to incrementally update their results, and send them to needed neighbors. As such, its verification time can be substantially shortened.

2 DISTRIBUTED, ON-DEVICE DPV

We demonstrate the basic design of distributed, on-device DPV by considering the network in Figure 1a and the following requirement: for all packets entering the network from S and destined to 10.0.0.0/23, they must reach D via a simple path passing W. Such a requirement can be expressed using a regular expression based language such as FlowExp [17].

2.1 Verification Decomposition

From requirement and topology to *DPVNet*. The core challenge to realize distributed, on-device DPV is how to allocate lightweight tasks to be executed on devices, because a device runs multiple protocols (*e.g.*, SNMP, OSPF and BGP) on a low-end CPU, with little computation power to spare.

To this end, we leverage the automata theory [22] to take the product of the regular expression *path_exp* in the requirement and the topology, perform state minimization on the product, and get a DAG called *DPVNet*. Similar to the product graph in network synthesis [6, 15, 29], a *DPVNet* compactly represents all paths in the topology that match the pattern *path_exp*. It is decided only by *path_exp* and the topology, and independent of the actual data plane. Figure 1c gives the computed *DPVNet*. Devices in the network and nodes in *DPVNet* have a 1-to-many mapping. Each node u in *DPVNet* is assigned a unique identifier composed of $u.dev$ and an integer. For example, device C is mapped to two nodes $C1$ and $C2$ in *DPVNet*, because the regular expression allows packets to reach D via $[C, W, D]$ or $[W, C, D]$.

Backward counting along *DPVNet*. A DPV problem is transformed into a counting problem on *DPVNet*: *given a packet p, can the network deliver a satisfactory number of copies of p to the destination node along paths in the DPVNet consistently?* "Consistently" means that should p enter the network multiple times, a satisfactory number of p's copies must be delivered each time. This notion is an extended version of multipath consistency in Batfish [11], and allows us to verify data planes with random forwarding actions (*e.g.*, vendor-specific select-type group table [12]). In our example, the problem of verifying whether the data plane (Figure 1b) conforms to the requirement is transformed to the problem of counting whether at least 1 copy of each p destined to 10.0.0.0/23 is delivered to $D1$ in Figure 1c consistently.

This counting problem can be solved by traversing *DPVNet* in reverse topological order. At its turn, each node u takes as input (1) the data plane of $u.dev$ and (2) for different packet spaces, the number of copies that can be delivered from each of u's downstream neighbors to the destination, along *DPVNet*, by the network data plane, to compute the number of copies that can be delivered from u to the destination along *DPVNet* by the network data plane. In the end, the source node of *DPVNet* computes the final counting result.

Figure 1c illustrates this algorithm. We use P_1, P_2, P_3 to represent the packet spaces with destination IP prefixes of 10.0.0.0/23, 10.0.0.0/24, and 10.0.1.0/24, where $P_2 \cap P_3 = \emptyset$

and $P_1 = P_2 \cup P_3$. Each u in *DPVNet* initializes a (packet space, count) mapping, $(P_1, 0)$, except for $D1$ that initializes the mapping as $(P_1, 1)$ (*i.e.*, one copy of any packet in P_1 will be sent to the correct external ports). Afterwards, we traverse all the nodes in *DPVNet* in reverse topological order to update their mappings. Each node u checks the data plane of $u.dev$ to find the set of next-hop devices $u.dev$ will forward P_1 to. If the action of forwarding to this next-hop set is of ALL-type, it means $u.dev$ will forward P_1 to all next-hop devices in this set. The mapping at u can be updated by adding up the count of all downstream neighbors of u whose corresponding device belongs to the set of next-hops of $u.dev$ for forwarding P_1. For example, $C1$ updates its mapping to $(P_1, 1)$ because C forwards to D, but $W2$'s mapping is still $(P_1, 0)$ because W does not forward P_1 to D. Similarly, although $W1$ has two downstream neighbors $C1$ an $D1$, each with an updated mapping $(P_1, 1)$. At its turn, we update its mapping to $(P_1, 1)$ instead of $(P_1, 2)$, because W only forwards P_1 to C, not D.

If a node u in *DPVNet* has a rule whose forwarding action is of ANY-type, it means $u.dev$ will forward packets matching this rule to one of the next-hops in this rule. In this case, u may have different counts for the same packet, depending on how $u.dev$ selects the next hop. We do not assume any knowledge on the selection algorithm because it is sometimes blackbox. Instead, we update the mapping at u to record all distinct counts, to verify whether the counting result is consistently satisfactory or not. Consider the mapping update at $A1$. A would forward P_2 to either B or W. If A forwards P_2 to B, the mapping at $A1$ is $(P_2, 0)$, because $B1$'s mapping is $(P_1, 0)$ and $P_2 \subset P_1$. If A forwards P_2 to W, the mapping at $A1$ is $(P_2, 1)$ because $W3$'s updated mapping is $(P_1, 1)$. As such, the mapping for P_2 at $A1$ is $(P_2, [0, 1])$, indicating both counts can happen. In the end, the mapping of $S1$ $[(P_2, [0, 1]), (P_3, 1)]$ reflects the final counting results. This means that the data plane does not satisfy the requirement consistently, *i.e.*, the network data plane is erroneous.

Decomposing into distributed counting. This centralized counting algorithm in *DPVNet* allows a natural decomposition into on-device counting tasks. Specifically, for each node u in *DPVNet*, an on-device counting task: (1) takes as input the data plane of $u.dev$ and the results of on-device counting tasks of all downstream neighbors of u whose corresponding devices belong to the set of next-hop devices $u.dev$ forwards packets to; (2) computes all possible numbers of copies that can be delivered from u to the destination along *DPVNet* by the network; and (3) sends the result to devices where its upstream neighbors in *DPVNet* reside in. In the end, the device of the source node can easily verify the requirement.

Computing consistent counting results. Counting tasks are event-driven. Given node u, when an event happens (*e.g.*, a rule/port change at $u.dev$, or a count update received from its downstream neighbor) $u.dev$ updates u's counting result, and sends it to u's upstream neighbors if the result changes. As such, assuming the network becomes stable at some point, the device of source node of *DPVNet* will eventually update its count result to be consistent with the network data plane.

Why not forward propagation? We let devices back propagate local counting results along *DPVNet*. One may wonder whether they could be forward propagated and achieve a better performance. Both can provide correct verification results. However, we choose back propagation because it allows each device to have counting results from itself to the final destinations, which can be used by routing services (*e.g.*, convergence-free routing [20, 30] and fast switching among multiple data planes [31]) to respond to network errors quickly to improve network availability. In contrast, forwarding propagation cannot provide such information.

Handling topology and requirement changes. The operator specifies requirements in the form that packets should be forwarded along a predefined set of paths in the original topology. After devices receive tasks, they run independently without a centralized controller. Devices handle unplanned topology changes (*e.g.*, link failures) setting corresponding counting results as 0 and propagating to upstream for consistency. If the failures lead the network to select a path that is not in the predefined set, our tool is no longer complete, *i.e.*, it might signal an error even when the property (*e.g.*, shortest path under failure) holds. When the operator makes planned topology changes or specifies new requirements, we need to recompute and send devices the updated tasks.

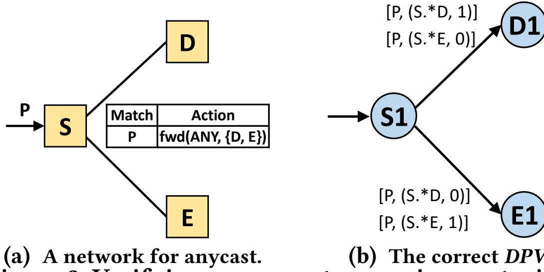
2.2 Distributed DPV Protocol

We design a distributed DPV protocol that specifies how devices incrementally update their on-device tasks, as well as how they communicate task results, efficiently and correctly.

Information storage. A device X stores two types of information: (1) a table of local equivalence classes (LECs), where an LEC is a set of packets whose actions are identical at X; and (2) a counting information base (CIB), a table of $(packet\ space, count)$ mapping of each $X.node$ in *DPVNet*.

Information exchange and handling. Devices share their CIB with the devices of upstream neighbors along the opposite of links in *DPVNet*, using UPDATE messages. No loop-prevention is needed because messages flow along a DAG. When device X receives an UPDATE message, it updates its CIB with the latest downstream counts in the message, and sends only the delta (*i.e.*, the changed $(packet\ space, count)$ mapping) to its upstream neighbors. Internal events (*e.g.*, rule update or link down) are handled in a similar way.

Consider a scenario in Figure 1, where B updates its rule to forward P_1 to W, instead of to C. Figure 1c circles the changed mappings of different nodes with boxes. B locally updates the results of $B1$ and $B2$ to $[(P_1, 1)]$ and $[(P_1, 0)]$, respectively, and sends them to the devices of their upstream neighbors, *i.e.*, $[(P_1, 1)]$ sent to A following the opposite of



(a) A network for anycast. (b) The correct *DPVNet*.

Figure 2: Verifying an anycast, a requirement with multiple *path_exp* with different destinations.

$(A1, B1)$ and $[(P_1, 0)]$ sent to W. Upon receiving the update, W need not update its mapping for $W3$, because W does not forward any packet to B. So W sends no update to A along the opposite of $(A1, W3)$. In contrast, A needs to update the task result for $A1$ to $[(P_1, 1)]$ because (1) no matter A forwards packets in P_2 to B or W, 1 copy of each packet will be sent to D, and (2) $P_2 \cup P_3 = P_1$. After updating the result, A sends the update to S. Finally, S updates the result for $S1$ to $[(P_1, 1)]$, *i.e.*, the requirement is satisfied after the update.

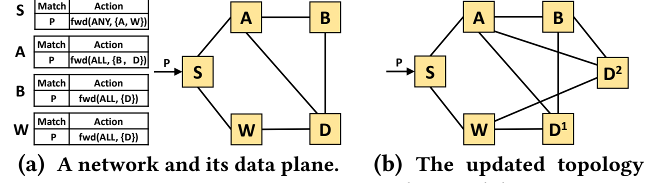
3 VERIFY COMPLEX REQUIREMENTS

We next discuss how to verify more complex requirements.

Anycast/Multicast. Consider an anycast requirement for S to reach D or E, but not both (Figure 2a). This is satisfied in the network. One may think a natural solution is to build a *DPVNet* for each destination, respectively, let devices count along all *DPVNet*s, and cross-produce the results at the source. However, this is incorrect. In this example, if we build two *DPVNet*s, one for each destination, we get two chains $S1 \rightarrow D1$, and $S2 \rightarrow E1$. After counting on both *DPVNet*s, $S1$ gets a set $[0, 1]$ for reaching $D1$, and $S2$ gets $[0, 1]$ for reaching $E1$. The cross-product computed by device S would be $[(0, 0),$ $(0, 1)$, $(0, 1)$, $(1, 1)]$, raising a false positive of network error.

Our solution is to construct a single *DPVNet* representing all paths in the network that reach at least one destination, by multiplying the union of all regular expressions (*e.g.*, $S.^*D$ and $S.^*E$) with the topology, and specify one counting task for one regular expression, at all nodes in *DPVNet*, including all destination nodes. Consider the same anycast example. We compute one *DPVNet* in Figure 2b. Each node counts the number of packets reaching both D and E. The count of $D1$ is $[(S.^*D, 1), (S.^*E, 0)]$ and $E1$ is $[(S.^*D, 0), (S.^*E, 1)]$. Such results are sent to $S1$. After $S1$ processes it, it determines that a packet is always sent to D or E, but not both, *i.e.*, the network is correct. This design extends to multicast as well.

Redundant delivery or waypoint inspection. We use a slightly odd requirement to demonstrate how to verify a complex requirement that has multiple regular expressions of the same destination. Consider Figure 3a and a requirement that specifies at least two copies of each packet in P should be sent to D along a loop-free path, or at least one copy should



(a) A network and its data plane. (b) The updated topology with virtual destinations.

Figure 3: Verifying a requirement with multiple *path_exp*s with the same destination.

be sent to D along a loop-free path passing W for inspection. We observe that the data plane satisfies this requirement.

Following the previous design, one may want to handle this case by also constructing a single *DPVNet* for the union of two *path_exp*s (*i.e.*, $S.^*D$ and $S.^*W.^*D$). However, because these *path_exp*s have the same destination, the counting along *DPVNet* cannot differentiate the counts for different *path_set*s, unless the information of paths are collected and sent along with the counting results. This would lead to large communication and computation overhead across devices.

Another strawman is to construct one *DPVNet* for one regular expression, count separately and aggregate the result at the source via cross-producing. However, false positives again can arise. Suppose we construct a *DPVNet* for each *path_exp*, and perform counting separately. S will receive a counting result $[1, 2]$ for reaching D with a simple path, and a counting result $[0, 1]$ for reaching D with a simple path passing W. The cross-product results $[(1, 0)$, $(1, 1)$, $(2, 0)$, $(2, 1)]$ indicate that a phantom violation is found.

To address this issue, we add virtual destination devices. Suppose a requirement has m *path_exp*s with the same destination D. We change D to D^1 and adds $m-1$ virtual devices D^i $(i = 2, \ldots, m)$. Each D^i has the same set of neighbors as D does. We then rewrite the destination of $path_exp_i$ to D^i $(i = 1, \ldots, m)$. Figure 3b gives the updated topology of Figure 3a. Next, we take the union of all *path_exp*s, intersect it with an auxiliary *path_exp* specifying no any two D^i, D^j should co-exist in a path. We then multiply the result with the new topology to get one single *DPVNet*. Counting can then proceed as the case for regular expressions with different destinations, by letting each device treat all its actions forwarding to D as forwarding to *all* D^is.

All-shortest-path availability. This invariant in Azure [16] requires all pairs of ToR devices to reach each other along a shortest path and all ToR-to-ToR shortest paths to be available in the data plane. Other than *DPVNet* construction and on-device decomposition, we prove that for each u in *DPVNet*, it does not need to send anything to its upstream neighbors, reducing distributed verification to local verification. Specifically, no u needs to compute the number of copies of packets that can delivered from u to destination. Instead, each u only checks if $u.dev$ forwards any packet to all the devices corresponding to the downstream neighbors of u in *DPVNet*.

If not, a network error is found and $u.dev$ can immediately report it to the operators. This makes local contract [16] a special case of distributed, on-switch DPV.

4 DISCUSSION

We discuss open research questions in regarding to distributed, on-device DPV and use cases it can benefit.

How to handle path-comparison invariants? To verify invariants that require the comparison between the paths of different packet spaces (*e.g.*, route symmetry and node-disjointness), one may construct the reachability *DPVNet* for each packet space, let on-device verifiers collect the actual downstream paths and send them to their upstream neighbors, and eventually perform the comparison with the complete paths of the two packet spaces as input. One potential downside, however, is efficiency, as more information than counting is collected and propagated. Distributed graph computation [27] may shed light for more efficient solutions.

How to handle packet transformations? Suppose a device has packet transformation rules, it can use BDD to encode such actions [39], and extend the CIB and the protocol UPDATE message to record and share the count results of packet transformation actions. Recent progress from model checking of pushdown systems [5] could also be helpful for encoding such rules more efficiently.

How to handle a huge number of valid paths? One concern is that *DPVNet* may be too large to generate in large networks with a huge number of valid paths. First, our survey and private conversations with operators suggest that they usually want the network to use paths with limited hops, if not the shortest one. The number of such paths is small even in large networks. Second, if a network wants to verify requirements with a huge number of valid paths, a potential solution is to partition the network into abstract one-big-switches, construct *DPVNet* on this abstract network, and perform intra-/inter-partition distributed verifications.

Does a company have to upgrade all its devices to use this framework? Our framework can be deployed incrementally in two ways. The first is to assign an off-device instance (*e.g.*, VM) for each device without an on-device verifier, who plays as a verifier to collect the data plane from the device and exchange messages with other verifiers based on *DPVNet*. This is a generalization of the deployment of RCDC, whose local verifiers are deployed in off-device instances. The second is the divide-and-conquer approach above. We deploy one verifier on one server for each partition. The verifier collects the data planes of devices in its partition to perform intra-partition verification, and exchanges the results with verifiers of other partitions for inter-partition verification. The two approaches are not exclusive.

What use cases can benefit from distributed, on-device DPV? In convergence-free routing [20, 30], devices can use their own counting results toward destinations to decide the next-hops to forward packets. In a network where SDN and distributed routing coexist [7, 31], devices can use such information to decide which data plane to use. Fast control plane repair [13] can use the verification results on devices to quickly find network errors and rollback to the correct configurations. This framework can also enable interdomain DPV [8, 37] to improve interdomain network reliability.

5 PERFORMANCE EVALUATION

We implement a prototype called Coral to evaluate its capability (§5.1), performance (§5.2, §5.3) and overhead (§5.4).

5.1 Functionality Demonstrations

We assemble a network of 6 switches in Figure 1a: 4 Mellanox [25], 1 Edgecore [9] and 1 Barefoot [4]. We run demos to verify (1) loop-free, waypoint reachability from S to D in Figure 1a, (2) loop-free, multicast from S to C and D, (3) loop-free, anycast from S to B and D, (4) different-ingress consistent loop-free reachability from S and B to D, and (5) all-shortest-path availability from S to C [16]. We run each demo with correct and erroneous data planes. The network always computes the right results. Details can be found at [3].

5.2 Testbed Experiments

We add 3 Barefoot switches to mimic the 9-device Internet2 WAN [26]. We inject propagation latencies between switches, based on Internet2 topology [34]. We verify the conjunction of loop-freeness, blackhole-freeness and all-pair reachability between switches along paths with ($\leq$x+2) hops, where x is the smallest-hop-count for each pair of switches.

Experiment 1: burst update. We first evaluate Coral in the scenario of burst update, *i.e.*, all forwarding rules are installed to corresponding switches all at once. Coral finishes the verification in 0.99 seconds, outperforming the best centralized DPV in comparison by 2.09× (Figure 4a).

Experiment 2: incremental update. After the burst update, we randomly generate $10K$ rule updates distributed evenly across devices and apply them one by one. After each update, we incrementally verify the network. For 80% of the updates, Coral finishes the verification $\leq 5.42ms$, outperforming the best centralized DPV in comparison by 4.90×.

5.3 Large-Scale Simulations

We implement an event-driven simulator to evaluate Coral in various real-world networks.

Datasets and comparisons. We use 5 datasets. We assign link latencies for WAN based on topologies [34], and $10\mu s$ latency for each link in LAN/DC. We compare Coral with centralized DPV tools: AP [36], APKeep [41] and Veriflow [19].

Requirements. We verify the same reachability requirement in §5.2 for WAN/LAN and the all-ToR-pair shortest path reachability for DC. For all datasets, the latency to compute on-device tasks is between 0.07 and 338 seconds.

Network				Verification Time			
Name	#Device	#Rules	Type	AP	APKeep	VeriFlow	Coral
Internet2	9	7.74×10^6	WAN	2.07	2.88	11.20	exp: 0.99 (2.09×) simu: 0.95 (2.18×)
Stanford	16	3.84×10^3	LAN	0.49	0.29	0.12	**0.06 (2.00×)**
Airtel1	16	9.64×10^4	WAN	0.82	2.56	1,771.14	**0.72 (1.14×)**
Airtel2	68	4.56×10^5	WAN	5.93	18.23	39,979.64	**1.60 (3.71×)**
Fattree (k=48)	2880	3.31×10^6	DC	40,608.79	3,293.30	46,103.03	**4.05 (813.16×)**

(a) Verification time of burst update (seconds).

Network	Percentage < 10 ms				80% quantile (ms)			
	AP	APKeep	VeriFlow	Coral	AP	APKeep	VeriFlow	Coral
Internet2	0%	29.29%	0.03%	exp: 81.55% simu: 83.63%	2782.10	26.55	345.62	5.42 (4.90×) 8.83 (3.01×)
Stanford	1.23%	99.93%	22.55%	**99.76%**	10,522.66	0.27	11.22	**0.06 (4.5×)**
Airtel1	4.89%	13.20%	0%	**85.50%**	1,488.84	122.25	336.88	**4.99 (24.50×)**
Airtel2	1.49%	10.73%	0%	**73.38%**	924.10	127.78	1,127.78	**73.64 (1.74×)**
Fattree (k=48)	0%	0%	0%	**96.81%**	29.21	86.17	18,962.00	**0.12 (243.42×)**

(b) Verification time of 10K incremental updates.

Figure 4: Verification time of experiments and large-scale simulations.

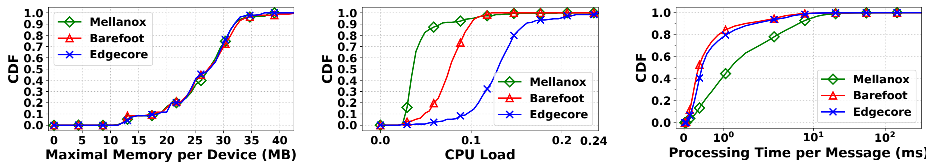
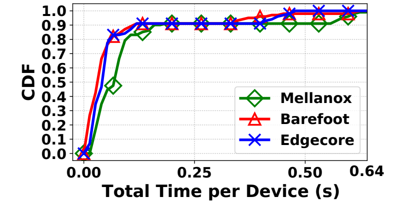
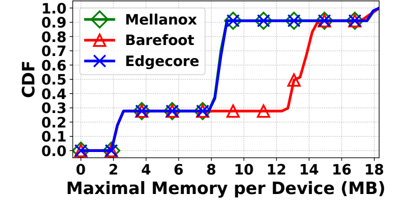


Figure 5: UPDATE message processing overhead.





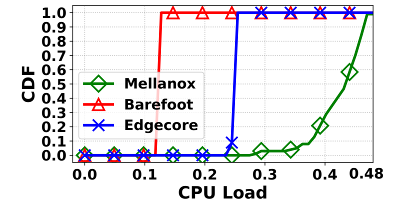


Figure 6: Initialization overhead.

Metric. We study the verification time. It is computed as the period from the arrival of data plane updates at devices to the time when all requirements are verified, including the propagation delays. For centralized DPV, we randomly assign a device as the location of the server, and let all devices send data planes to the server along lowest-latency paths.

Results: burst update. In Figure 4a, for WAN/LAN, Coral completes the verification in $\leq 1.60s$ and achieves an up to 3.71× speedup than the fastest centralized DPV. For DC, this speedup is up to 813.16×. This is because Coral decomposes verification into on-device tasks, which have a dependency chain roughly linear to the network diameter. A DC has a small diameter (*e.g.*, 4 hops). As such, Coral achieves a very high level of parallelization, enabling high scalability.

Results: incremental update. Figure 4b shows that, for 10K incremental verification in each network, the 80% quantile verification time of Coral is up to 243× faster than the fastest centralized DPV. In all datasets, Coral finishes verifying at least 73.38% rule updates in $< 10ms$, while this lower bound of other tools is less than 1%. This is for the same reason as in experiments (§5.2). When a rule update happens, only devices whose on-device task results are affected need to incrementally update their results, and only these changed results are sent to neighbors incrementally. For most rule updates, the number of affected devices is small (2 devices at 75% quantile in all simulations). In addition, all UPDATE messages during simulations have a size less than 150 KB.

5.4 On-Device Microbenchmarks

Initialization overhead. For each of 98 devices from WAN / LAN and 3 devices from Fattree (one edge, aggregation and core switch), we measure its initialization overhead in burst update (*i.e.*, computing the initial LEC and CIB), in terms of total time, maximal memory and CPU load, on the 3 switch models. The CPU load is computed as $CPU\ time / (total\ time \times \#cores)$. On all 3 switches, all devices in the datasets complete initialization in $\leq 0.64s$, with a CPU load ≤ 0.48, and a maximal memory $\leq 18.3MB$ (Figure 6).

UPDATE message processing overhead. For each of the same set of 101 devices, we collect their received UPDATE messages during burst and incremental update, replay them consecutively on each of the three switches, and measure the message processing overhead, *i.e.*, maximal memory, CPU load and processing time/message. (Figure 5). For 90% of devices, all 3 switches process all UPDATE messages in $\leq 2.37s$, with a maximal memory $\leq 32.37MB$, and a CPU load ≤ 0.16. And for 90% of all $83.48K$ UPDATE messages, the switches can process it in $\leq 6.11ms$. To summarize, these microbenchmarks show that Coral can be deployed on commodity network devices with little overhead.

Acknowledgments. We are extremely grateful for our shepherd, Aurojit Panda, and the anonymous HotNets reviewers for their wonderful feedback. We thank Jiwu Shu, Dennis Duan, Dong Guo, Xiwen Fan and Hanyang Shao for their help on this paper. This work is supported in part by the National Key R&D Program of China 2022YFB2901502, NSFC Award 62172345, Alibaba Innovative Research Award, Open Research Projects of Zhejiang Lab 2022QA0AB05, MoE China Award 2021FNA02008, NSF-Fujian-China 2022J01004, and IKKEM Award HRTP-2022-34.

REFERENCES

[1] E. Al-Shaer and S. Al-Haj. Flowchecker: Configuration analysis and verification of federated openflow infrastructures. In *Proceedings of the 3rd ACM workshop on Assurable and usable security configuration*, pages 37–44, 2010.

[2] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker. Netkat: Semantic foundations for networks. *Acm sigplan notices*, 49(1):113–126, 2014.

[3] A. Authors. Coral system functionality demonstration. http://distributeddpvdemo.tech/, 2022.

[4] Barefoot S9180-32X Switch. https://www.ufispace.com/uploads/able/files/productfilemanager/000045467d1fc648d792c404372956a0.pdf, 2019.

[5] R. Beckett and A. Gupta. Katra: Realtime verification for multilayer networks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 617–634, 2022.

[6] R. Beckett, R. Mahajan, T. Millstein, J. Padhye, and D. Walker. Don't mind the gap: Bridging network-wide objectives and device-level configurations. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 328–341, 2016.

[7] S. Choi, B. Burkov, A. Eckert, T. Fang, S. Kazemkhani, R. Sherwood, Y. Zhang, and H. 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.

[8] A. Dhamdhere, D. D. Clark, A. Gamero-Garrido, M. Luckie, R. K. Mok, G. Akiwate, K. Gogia, V. Bajpai, A. C. Snoeren, and K. Claffy. Inferring persistent interdomain congestion. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 1–15, 2018.

[9] Edgecore Wedge32-100X Switch. https://www.edge-core.com/productsInfo.php?cls=1&cls2=5&cls3=181&id=335, 2021.

[10] Facebook Employees Were Unable to Access Critical Work Tools During Six-Hour Outage. https://www.cnbc.com/2021/10/04/facebook-workers-lose-access-to-internal-tools-following-outage.html, 2021.

[11] A. Fogel, S. Fung, L. Pedrosa, M. Walraed-Sullivan, R. Govindan, R. Mahajan, and T. Millstein. A general approach to network configuration analysis. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 469–483, 2015.

[12] O. N. Foundation. Openflow switch specification 1.5.1. Open Networking Foundation (on-line), Mar. 2015.

[13] A. Gember-Jacobson, C. Raiciu, and L. Vanbever. Integrating verification and repair into the control plane. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, pages 129–135, 2017.

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

[15] K.-F. Hsu, R. Beckett, A. Chen, J. Rexford, P. Tammana, and D. Walker. Contra: A programmable system for performance-aware routing. *to apper at NSDI'20*, 2020.

[16] K. Jayaraman, N. Bjørner, J. Padhye, A. Agrawal, A. Bhargava, P.-A. C. Bissonnette, S. Foster, A. Helwer, M. Kasten, I. Lee, et al. Validating datacenters at scale. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 200–213. 2019.

[17] P. Kazemian, M. Chan, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real time network policy checking using header space analysis. In *NSDI*, pages 99–111, 2013.

[18] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: Static checking for networks. In *NSDI*, volume 12, pages 113–126, 2012.

[19] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. Veriflow: Verifying network-wide invariants in real time. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 15–27, 2013.

[20] K. Lakshminarayanan, M. Caesar, M. Rangan, T. Anderson, S. Shenker, and I. Stoica. Achieving convergence-free routing using failure-carrying packets. *ACM SIGCOMM computer communication review*, 37(4):241–252, 2007.

[21] F. Le, G. G. Xie, and H. Zhang. Theory and new primitives for safely connecting routing protocol instances. *ACM SIGCOMM Computer Communication Review*, 40(4):219–230, 2010.

[22] H. R. Lewis and C. H. Papadimitriou. Elements of the theory of computation. *ACM SIGACT News*, 29(3):62–78, 1998.

[23] N. P. Lopes, N. Bjørner, P. Godefroid, K. Jayaraman, and G. Varghese. Checking beliefs in dynamic networks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 499–512, 2015.

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

[25] Mellanox SN2700 Switch. https://www.mellanox.com/related-docs/prod_eth_switches/PB_SN2700.pdf, 2015.

[26] T. I. Observatory. The internet2 dataset. http://www.internet2.edu/research-solutions/research-support/observatory, 2021.

[27] G. Pandurangan, P. Robinson, and M. Scquizzato. On the distributed complexity of large-scale graph computations. *ACM Transactions on Parallel Computing (TOPC)*, 8(2):1–28, 2021.

[28] G. D. Plotkin, N. Bjørner, N. P. Lopes, A. Rybalchenko, and G. Varghese. Scaling network verification using symmetry and surgery. *ACM SIGPLAN Notices*, 51(1):69–83, 2016.

[29] R. Soulé, S. Basu, P. J. Marandi, F. Pedone, R. Kleinberg, E. G. Sirer, and N. Foster. Merlin: A language for managing network resources. *IEEE/ACM Transactions on Networking*, 26(5):2188–2201, 2018.

[30] K. Subramanian, A. Abhashkumar, L. D'Antoni, and A. Akella. D2r: Policy-compliant fast reroute. In *Proceedings of the ACM SIGCOMM Symposium on SDN Research (SOSR)*, pages 148–161, 2021.

[31] S. Vissicchio, L. Cittadini, O. Bonaventure, G. G. Xie, and L. Vanbever. On the co-existence of distributed and centralized routing control-planes. In *Computer Communications (INFOCOM), 2015 IEEE Conference on*, pages 469–477. IEEE, 2015.

[32] H. Wang, C. Qian, Y. Yu, H. Yang, and S. S. Lam. Practical network-wide packet behavior identification by ap classifier. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, pages 1–13, 2015.

[33] H. Wang, C. Qian, Y. Yu, H. Yang, and S. S. Lam. Practical network-wide packet behavior identification by ap classifier. *IEEE/ACM Transactions on Networking*, 25(5):2886–2899, 2017.

[34] WonderNetwork. Global ping statistics. https://wondernetwork.com/pings, 2021.

[35] G. G. Xie, J. Zhan, D. A. Maltz, H. Zhang, A. Greenberg, G. Hjalmtysson, and J. Rexford. On static reachability analysis of ip networks. In *Proceedings IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies.*, volume 3, pages 2170–2183. IEEE, 2005.

[36] H. Yang and S. S. Lam. Real-time verification of network properties using atomic predicates. In *2013 21st IEEE International Conference on Network Protocols (ICNP)*, pages 1–11. IEEE, 2013.

[37] H. Yang and S. S. Lam. Collaborative verification of forward and reverse reachability in the internet data plane. In *2014 IEEE 22nd International Conference on Network Protocols*, pages 320–331. IEEE, 2014.

[38] H. Yang and S. S. Lam. Real-time verification of network properties using atomic predicates. *IEEE/ACM Transactions on Networking*, 24(2):887–900, 2016.

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

[40] 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 *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 87–99, 2014.
[41] P. Zhang, X. Liu, H. Yang, N. Kang, Z. Gu, and H. Li. Apkeep: Realtime verification for real networks. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 241–255, 2020.