Beyond a Centralized Verifier: Scaling Data Plane Checking via Distributed, On-Device Verification

Qiao Xiang†, Chenyang Huang†, Ridi Wen‡, Yuxin Wang†, Xiwen Fan†, Zaoxing Liu‡, Linghe Kong*, Dennis Duan‡, Franck Le♦, Wei Sun‡,
†Xiamen Key Laboratory of Intelligent Storage and Computing, Xiamen University,
‡University of Maryland, *Shanghai Jiao Tong University, ♦Yale University, ‡IBM Research, ♢UT Austin

ABSTRACT
Centralized data plane verification (DPV) faces significant scalability issues in large networks (i.e., the verifier being a performance bottleneck and single point of failure and requiring a reliable management network). We tackle this scalability challenge by introducing Tulkun, a distributed, on-device DPV framework. Our key insight is that DPV can be transformed into a counting problem on a directed acyclic graph, which can be naturally decomposed into lightweight tasks executed at network devices, enabling fast data plane checking in networks of various scales and types. With this insight, Tulkun consists of (1) a declarative invariant specification language, (2) a planner that employs a novel data structure DPVNet to systematically decompose global verification into on-device counting tasks, (3) a distributed verification messaging (DVM) protocol that specifies how on-device verifiers efficiently communicate task results to jointly verify the invariants, and (4) a mechanism to verify invariant fault-tolerance with minimal involvement of the planner. Extensive experiments with real-world datasets (WAN/LAN/DC) show that Tulkun verifies a real, large DC in 41 seconds while other tools need minutes or up to tens of hours, and shows an up to 2355-fold speed up on 80% quantile of incremental verification with small overhead on commodity network devices.

CCS CONCEPTS
• Networks → Protocol testing and verification; Network reliability.

KEYWORDS
Network verification, Distributed verification

1 INTRODUCTION
There has been a long line of research on data plane verification [3, 4, 7, 34, 37, 43–45, 53, 55, 61, 74, 82, 83, 85, 86, 90, 92, 93]. Earlier tools analyzed a snapshot of the complete data plane of the network to identify network errors (e.g., blackholes, waypoint violation and forwarding loops) [3, 4, 34, 44, 53, 55, 61, 74, 75, 82, 83, 85, 86, 90, 92]; and recent solutions focus on incremental verification (i.e., verifying forwarding rule updates) [7, 37, 41, 43, 45, 92, 93]. State-of-the-art DPV tools (e.g., [93]) can achieve an incremental verification time of tens of microseconds per rule update. Centralized DPVs do not scale. Despite the substantial progress in accelerating DPV, existing tools employ a centralized architecture, lacking the scalability needed for deployment in large networks. Specifically, they use a centralized verifier to collect the data plane from each network device and verify the invariants. This verifier becomes the performance bottleneck and the single point of failure (PoF) of DPV tools, e.g., our test shows that it takes APKeep [93] ~1 hour to verify a 48-ary fattree (§9.3). More importantly, this design requires a management network to provide reliable, low-latency connections between the server and network devices, which itself is hard to build for large-scale networks [22].

Some studies [7, 34, 41, 90] have attempted to tackle these limitations of centralized DPV. Libra [90] partitions the IP-prefix based data plane into disjoint packet spaces to achieve parallel verification in a cluster, but it cannot efficiently partition a data plane that forwards on an arbitrary mix of headers. Azure RCDC [41] partitions the data plane by device and verify the availability of all shortest paths in parallel in a cluster, but it can only verify this specific invariant. Flash [34] proposes to process massive data plane rules in batch to accelerate the computation of equivalence classes, but it is slow in incremental verification. To relax the need of a reliable, low-latency management network, Flash [34] proposes an early detection mechanism to detect data plane violations with incomplete information. However, our test using its open-sourced prototype [33] shows that even if the verifier misses the updated rules of only three randomly chosen devices, in 9 out of 11 LAN/WAN datasets, Flash detects zero errors in 80% of the experiment cases.

In this paper, we systematically tackle the important problem of how to scale DPV to be applicable in large networks. Not only can a scalable DPV tool quickly find errors in large networks, it can also support novel routing services (e.g., convergence-free routing [48, 69], real-time control plane repair [27], fast switching among multiple data planes [16, 49, 72], and interdomain DPV [17, 84]) to respond to network errors quickly to improve network availability. Proposal: Offload DPV to distributed computations on network devices. Instead of continuing to squeeze incremental performance improvements out of centralized DPV, we embrace a...
distributed design to circumvent the inherent scalability bottleneck of centralized design. Azure RCDC [41] takes the first step in this direction by partitioning verification into local contracts of devices. It gives an interesting analogy between local contracts and program verification using annotation with inductive loop invariants, but stops at communication-free local contracts for the particular all-shortest-path availability invariant and validating them in parallel on a centralized cluster. In contrast, we go beyond and show that for a wide range of invariants (e.g., reachability, multicast and anycast), with lightweight tasks running on commodity network devices and limited communication among them, we can verify these invariants in a compositional way, achieving scalable DPV in generic settings.

Key insight: Transform DPV to distributed counting. The fundamental challenge in realizing distributed verification is how to allocate lightweight tasks running on commodity network devices because they have little spare computation power. While our position paper suggested the promise of distributed DPV [81], it fell short in answering several important questions, including (1) how to specify and verify generic, common invariants efficiently, (2) how to verify data planes with packet transformations, (3) how to minimize the information exchange between devices to reduce the overhead, and (4) how to efficiently verify the fault-tolerance of invariants. To this end, we design Tulkun, a generic, distributed, on-device DPV framework, with a key insight: the problem of DPV can be transformed into a counting problem in a directed acyclic graph (DAG) representing all valid paths in the network; the latter can be decomposed into lightweight tasks at nodes on the DAG that are distributively executed at corresponding devices, enabling fast DPV in networks of various scales with scalability approximately linear to the network diameter. As depicted in Figure 1, Tulkun has four key designs (D1-D4):

D1: A declarative invariant specification language (§3). This language abstracts an invariant as a tuple of packet space, ingress devices and behavior, where a behavior is a predicate on whether the paths of packets match a pattern specified in a regular expression. It allows operators to flexibly specify common invariants studied by existing DPV tools (e.g., reachability, blackhole-freeness, and waypoint), and more advanced, yet understudied invariants (e.g., multicast, anycast, no-redundant-delivery, and all-shortest-path availability).

D2: A verification planner to allocate tasks to devices (§4). Given an invariant, the planner leverages the automata theory [50] to multiply its path pattern regular expressions and the network topology to compute DPVNet, a DAG compactly representing all paths in the network that satisfies the path patterns in the invariant, and transforms the DPV problem into a counting problem on DPVNet. The latter can be solved by a reverse topological traversal along DPVNet. In its turn, each node in DPVNet takes as input the data plane of its corresponding device and the counting results of its downstream nodes to compute for different packets, how many copies of them can be delivered to the intended destinations along downstream paths in DPVNet. This traversal can be naturally decomposed into on-device counting tasks, one for each node in DPVNet, and distributed to the corresponding network devices. We design optimizations to compute the minimal counting information of each node in DPVNet to send to its upstream neighbors, and prove that for invariants like all-shortest-path availability, their minimal counting information is an empty set, i.e., the local contracts in Azure RCDC [41] is a special case of Tulkun.

D3: On-device verifiers equipped with a DVM protocol (§5). On-device verifiers execute the counting tasks specified by the planner and share their results with neighbor devices to collaboratively verify the invariants. We are inspired by vector-based routing protocols [56, 64] to design a DVM protocol that specifies how neighboring on-device verifiers communicate counting results in an efficient, correct way.

D4: Minimizing planner-verifiers communication (§6). To avoid the planner becoming the scalability bottleneck, we design a mechanism to let on-device verifiers check the fault-tolerance of invariants with minimal involvement of the planner. Specifically, the planner precomputes a fault-tolerant DPVNet representing the union of all valid paths in operator-specified failure scenes and sends tasks to verifiers. When failures happen, verifiers adaptively adjust their tasks to count along paths in the DPVNet corresponding to the updated topology, without contacting the planner.

Implementation (§8). We implement a prototype of Tulkun and release it as an open source project [79] with a set of demos [78]. Tulkun is being evaluated by a couple of major vendors to integrate into their commodity switches. Our proposal to integrate Tulkun as a feature of SONiC is also under review by the community [65].

Evaluation results (§9). We evaluate Tulkun extensively using real-world datasets, in hardware testbed and simulations. Tulkun consistently outperforms centralized DPV tools under various networks (WAN/LAN/DC) and DPV scenarios: (1) Verifying a real, large DC in less than 41 seconds while the state-of-the-art DPV tools take minutes and the classic ones take tens of hours; (2) Achieving an up to 2355× speedup on 80% quantile of incremental verification, with little resource overhead.

## 2 OVERVIEW

This section introduces some key concepts in Tulkun, and illustrates its workflow using an example.

### 2.1 Basic Concepts

**Data plane model.** For ease of exposition, given a network device, we model its data plane as a match-action table, where the entries are ordered in descending priority. Each entry has a match field to match packets on packet headers (e.g., TCP/IP 5-tuple) and an action field to perform packet actions. Possible actions include modifying the headers of the packet and forwarding the packet to a group of the next-hops [25, 41]. An empty group means the action is to drop the packet. If an action forwards the packet to all next-hops in a
non-empty group, we call it an ALL-type action. If it forwards the packet to one of the next-hops in a non-empty group, we call it an ANY-type action. Given an ANY-type action, we do not assume any knowledge on how the device selects one next-hop from the group. It is because this selection algorithm is vendor-specific, and sometimes a blackbox [25].

Packet traces and universes. Inspired by NetKAT [4], we introduce the concept of packet trace to record the state of a packet as it travels from device to device, and use it to define the network behavior of packet forwarding. When a packet enters a network from an ingress device $S$, a packet trace of $p$ is defined as a non-empty sequence of devices visited by $p$ until it is delivered to the destination device or dropped.

However, due to ALL-type actions, a packet may not be limited to one packet trace each time it enters a network. For example, in Figure 2a, the network forwards a packet $p$ with a destination IP 10.0.0.0 along a set of two traces $\{[S, A, B], [S, A, W, D]\}$ because $A$ forwards it to both $B$ and $W$. We denote this set to be a universe of packet $p$ from ingress $S$. In addition, with the existence of ANY-type actions, a packet may traverse one of a number of different sets of packet traces (universes) each time it enters a network. In the same example, consider a packet $q$ with a destination IP 10.0.1.0 and a destination port 80. When it enters the network in different instances, the network may forward $q$ according to the universe $\{[S, A, B, D]\}$ or the universe $\{[S, A, W, D]\}$ because $A$ forwards $q$ to either $B$ or $W$. These universes (each being a set of traces) can be thought of as a “multiverse” - should the packet enter the network multiple times, it may experience different fates each time.

The notion of universes is a foundation of Tulkun. We are inspired by multipath consistency [24], where a packet is either accepted on all paths or none at all, but go beyond. For each invariant, we verify whether it holds in all universes.

### 2.2 Workflow

We demonstrate Tulkun’s workflow with the network in Figure 2a and an invariant: for all packets destined to 10.0.0.0/23, when entering the network at $S$, they must reach $D$ via a simple path passing $W$. Tulkun verifies it in three phases.

#### 2.2.1 Invariant Specification.

In Tulkun, operators specify verification invariants using a declarative language. An invariant is specified as a (packet, packet, behavior) tuple. The semantic means: for each packet $p$ in packet entering the network from any device in ingress set, the traces of $p$ in all its universes must satisfy the constraint specified in behavior, which is specified as a tuple of a regular expression of valid paths path_exp and a match operator.

#### 2.2.2 Verification Decomposition and Distribution.

Given an invariant, Tulkun uses a planner to decide the tasks to be executed distributively on devices to verify it. The core challenge is how to make these on-device tasks lightweight, because a network device typically runs multiple protocols (e.g., SNMP, OSPF and BGP) on a low-end CPU, with little computation power to spare. To this end, the Tulkun planner employs a data structure called DPVNet to decompose the problem into small on-device verification tasks, and distribute them to on-device verifiers for distributed execution.

**From invariant and topology to DPVNet.** The planner leverages the automata theory [50] to multiply the regular expression path_exp and the topology and get a DAG called DPVNet. Similar to the product graph [11, 39, 66], a DPVNet compactly represents all paths in the topology that match path_exp. It is decided only by path_exp and the topology, and is independent of the actual data plane of the network.

Figure 2c gives the DPVNet in our example. Devices in the network and nodes in DPVNet have a 1-to-many mapping. Each node $u$ in DPVNet has a concatenation of $u$.dev and an integer as its identifier. For example, device $B$ in the network is mapped to $B1$ and $B2$ in DPVNet, because the regular expression allows packets to reach $D$ via $[B, W, D]$ or $[W, B, D]$.

**Backward counting along DPVNet**. With 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 in each universe? In our example, the problem of verifying whether the data plane of the network (Figure 2b) satisfies the invariant is transformed into 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 2c in all of $p$’s universes.
This counting problem can be solved by traversing DPVNet in reverse topological order. In its turn, each node \( u \) takes as input (1) the data plane of \( u.\text{dev} \) and (2) for different \( p \) in packet_space, 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 result.

Figure 2c illustrates this process. We use \( P_1, P_2, P_3, P_4 \) to represent the packet spaces \( \{\text{destIP} = 10.0.0.0/23\} \), \( \{\text{destIP} = 10.0.0.0/24\} \), \( \{\text{destIP} = 10.0.1.0/24, \text{destPort} = 80\} \), and \( \{\text{destIP} = 10.0.1.0/24, \text{destPort} \neq 80\} \), respectively. \( P_2, P_3 \) and \( P_4 \) are disjoint and \( P_1 = P_2 \cup P_3 \cup P_4 \). Each \( u \) in DPVNet initializes a \( (\text{packet space, count}) \) 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). We traverse all the nodes in DPVNet in reverse topological order to update their mappings. Each node \( u \) checks the data plane of \( u.\text{dev} \) to find the set of next-hop devices \( u.\text{dev} \) will forward \( P_1 \) to. If the action of forwarding to this next-hop set is of \( \text{ALL}-\text{type} \), 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 \( u.\text{dev} \) for forwarding \( P_1 \). For example, node \( W1 \) updates its mapping to \( (P_1, 1) \) because \( W \) forwards \( P_1 \) to \( D \). \( B2 \) updates to \( [(P_2, 0), (P_3 \cup P_4, 1)] \) because \( B \) forwards \( P_3 \cup P_4 \) to \( D \), but drops \( P_2 \). However, \( B1 \) does not update its mapping because \( B \) does not forward to \( W \). Similarly, although \( W2 \) has two downstream neighbors \( B2 \) an \( D1 \), each with an updated mapping \( (P_1, 1) \), in its turn, \( W2 \) updates its mapping to \( (P_1, 1) \) instead of \( [(P_2, 1), (P_3 \cup P_4, 2)] \) because \( W \) only forwards \( P_1 \) to \( D \), not \( B \).

Given a node \( u \) in DPVNet, if the action of forwarding is of \( \text{ANY}-\text{type} \), the count may vary at different universes. As such, we update the mapping at \( u \) to record these distinct counts. For example, \( A \) would forward \( P_3 \) to either \( B \) or \( W \). As such, in one universe where \( A \) forwards \( P_3 \) to \( B \), the mapping of \( P_3 \) at \( A \) is \( (P_3, 0) \), because \( B1 \)'s updated mapping is \( (P_1, 0) \) and \( P_3 \subset P_1 \). In the other universe where \( A \) forwards \( P_3 \) to \( W \), the mapping of \( P_3 \) at \( A \) is \( (P_3, 1) \) because \( W3 \)'s updated mapping is \( (P_1, 1) \). Therefore, the updated mapping of \( P_3 \) at \( A \) is \( (P_3, 0, 1) \), indicating the different counts at different universes. In the end, the updated mapping of \( S1 \) \( [(P_2 \cup P_4, 1), (P_3, 0, 1)] \) is the final counting results, indicating that Figure 2a does not satisfy the invariant in Figure 2b in all universes, i.e., the network data plane is erroneous.

Counting decomposition and distribution. This counting algorithm allows a natural decomposition into on-device counting tasks to be executed distributively on network devices. For each node \( u \) in DPVNet, an on-device counting task: (1) takes as input the data plane of \( u.\text{dev} \) 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 \( u.\text{dev} \) forwards packets to; (2) computes the number of copies that can be delivered from \( u \) to the destination along DPVNet, by the network data plane in each universe; and (3) sends the computed result to devices where its upstream neighbors in DPVNet reside in. After the decomposition, the planner sends the counting task of each \( u \) and the lists of \( u \)'s downstream and upstream neighbors to device \( u.\text{dev} \).

Minimizing planner-verifiers communication. One hurdle that may make the planner the scalability bottleneck is fault tolerance, because an invariant may have different sets of valid paths under different failure scenarios (e.g., shortest-path reachability under k-link-failure). To this end, we design a mechanism consisting of fault-tolerant DPVNet precomputation and online recounting to allow on-device verifiers to verify the fault-tolerance of invariants with minimal involvement of the planner. The communication between the planner-verifiers is restricted to the cases when (1) the operator makes planned topology changes or specifies new invariants; (2) a data plane error is found by on-device verifiers; and (3) on-device verifiers find failure scenes that are not pre-specified by operators.

2.2.3 Distributed, Event-Driven Verification using DVM Protocol. On-device verifiers execute the tasks sent from the planner in a distributed, event-driven way. When events (e.g., rule update and the arrival of neighbors’ updated results) happen, on-device verifiers update the results of their tasks, and send them to neighbors if needed. We design a DVM protocol that specifies how verifiers incrementally update and communicate their task results efficiently and correctly.

Consider a scenario in Figure 2, where \( B \) updates its action to forward \( P_3 \cup P_4 \) to \( W \), instead of \( D \). The changed mappings of different nodes are circled with boxes in Figure 2c. \( B \) locally updates the results of \( B1 \) and \( B2 \) to \( [(P_2, 0), (P_3 \cup P_4, 1)] \) and \( [(P_3, 0)] \), and sends the updates to \( A \) along \( (B1, A1) \) and \( W \) along \( (B2, W2) \). Upon receiving the update, \( W \) does not update the mapping of \( W2 \) because \( W \) does not forward any packet to \( B \). As such, \( W \) sends no update to \( A \) along \( (W3, A1) \). In contrast, \( A \) updates its task result of node \( A1 \) to \( (P_1, 1) \) because (1) the count of \( P2 \) and \( P4 \) at \( A \) does not change; (2) no matter whether \( A \) forwards \( P3 \) to \( B \) or \( W \), 1 copy of each packet will be sent to \( D \), and (3) \( P2 \cup P3 \cup P_4 = P_1 \). Finally, \( S \) updates its local result for \( S1 \) to \( (P_1, 1) \), i.e., the invariant is satisfied after the update.

3 SPECIFICATION LANGUAGE
Tulkun provides a declarative language for operators to specify verification invariants based on the concepts of traces and universes. Figure 3 gives its simplified grammar.

Language overview. On a high level, an invariant is specified by a \( (\text{packet space, ingress set, behavior}, \text{fault scenes}) \) tuple, with semantics as explained in §2.2.1. Operators can also include an optional field \( \text{fault scenes} \) in the tuple to specify fault tolerance of invariants (see §6 for details). To specify behaviors, we use the building block of \( (\text{match op}, \text{path exp}) \) entries. The basic syntax provides two \( \text{match op} \) operators. One is \( \text{exist count exp} \), which requires that in each universe, the number of traces matching \( \text{path exp} \) satisfies
Invariants | Tulkun specifications
---|---
Reachability ([24, 33, 55]) | \((P, [S], \text{exist} >= 1.S.D)\)
Isolation ([24, 53, 55]) | \((P, [S], \text{exist} <= 1.S.D)\)
Loop-freeness [55] | \((P, [S], \text{exist} >= 0.S.D)\) and \((\text{not}(not X))\) or \((\text{not}(not X'))\) and \((\text{not}(Y))\) or \((\text{not}(Y'))\) or \((\text{not}(Y'Y))\)
Black hole freeness [55] | \((P, [S], \text{exist} <= 0.S.D)\)
Waypoint reachability (DPVNet) | \((P, [S], \text{exist} >= 1.X.W.D)\)
Reachability with limited path length [43] | \((P, [X, Y], \text{exist} >= 1.X.D(\text{finite}))\)
Different ingress/egress reachability [43, 55] | \((P, [S], \text{equal}(S, D, (\text{finite})))\)
All-shortest-path reachability [51] | \((P, [S], \text{exist} >= 1.S.D)\)
Non-redundant reachability (Tulkun) | \((P, [S], \text{exist} >= 1.S.D)\)
Multicast (Tulkun) | \((P, [S], ((\text{exist} >= 1.S.D) \text{and (exist} >= 1.S.E))\)
Anycast (Tulkun) | \((P, [S], ((\text{exist} >= 1.S.D) \text{and (exist} >= 0.S.E)) \text{or (exist} >= 0.S.D) \text{and (exist} >= 1.S.E))\)

Table 1: Tulkun specifications for selected invariants.

count_exp. For example, exist >= 1 specifies at least one trace should match path_exp in each universe, and can be used to express reachability invariants. The other operator is equal, which specifies an equivalence behavior: the union of universes for each \(p\) in pkt_space from each ingress in ingress_set must be equal to the set of all possible paths that match path_exp [41]. Operators specify path_exp as a regular expression over the set of devices, with an optional field length_filters to filter it with length constraints. For example, \((S.D, (\text{shortest} \geq 1))\) represents all paths that match \(S.D\) and have a hop count no more than that of the shortest one plus 1. Behaviors can also be specified as conjunctions, disjunctions, and negations of \((\text{match}_\text{op}, \text{path}_\text{exp})\) pairs.

These two operators can be used to form a wide range of invariants in DPV. Table 1 provides examples of invariants that can be specified and verified in Tulkun, and the corresponding specifications in the Tulkun language. For example, using exist count_exp, operators can express simpler invariants (e.g., reachability, waypoint reachability, and loop-freeness) that are well studied by existing DPV tools [43–45, 83, 93], and more advanced invariants (e.g., multicast, anycast and no-redundant-delivery routing). Another example is an invariant given in Azure RCDC [41], which requires that all pairs of ToR devices should reach each other along a shortest path, and all ToR-to-ToR shortest paths should be available in the data plane. This can be formulated as an equal behavior on all shortest paths across all universes (row 9 in Table 1).

Note that once an invariant is specified, Tulkun checks whether it is consistently satisfied across all universes. As such, the multipath consistency [24, 53] is expressed separately as reachability and isolation invariants.

Convenience features. Tulkun builds and provides operators with a (device, IP_prefix) mapping for network devices with external ports (e.g., a ToR switch or a border router), where each tuple means that IP_prefix can be reached via an external port of device. If an invariant is submitted with inconsistencies between the destination IPs in packet_space and the destination devices in its corresponding path_exp, Tulkun will raise an error to operators.

The language provides syntax sugar to simplify the expression of invariants. For example, it allows users to specify a device set and provides device iterators. It provides shortcuts of behaviors, e.g., loop_free, and length filters, e.g., shortest. It also provides a third match_op called subset, which requires for packet \(p\) entering the network from ingress \(S\), the set of traces of \(p\) in each universe is a non-empty subset of path_exp. A behavior subset path_exp is a shortcut of \((\text{match} >= 1 \text{ path}_\text{exp})\) and \((\text{match} == 0 \cdot a \text{ and (not path}_\text{exp})\). We omit their details for the sake of simplicity.

**4 VERIFICATION PLANNER**

We introduce DPVNet and how to use it for verification decomposition assuming an invariant has one regular expression, and then describe how to handle more complex invariants.

4.1 **DPVNet**

Given a path_exp and a network, DPVNet is a DAG representing all paths in the network that matches path_exp. DPVNet can be constructed in different ways (e.g., graph dual variables). We are inspired by network synthesis [11, 39, 66] and leverage the automata theory [50] for DPVNet construction.

Specifically, given a path_exp, we first convert its regular expression into a finite automaton \((\Sigma, Q, F, q_0, \delta)\). \(\Sigma\) is the alphabet whose symbols are network device identifiers. \(Q\) is the set of states. \(q_0\) is the initial state. \(F\) is the set of accepting states. \(\delta : Q \times \Sigma \rightarrow Q\) is the state transition function. For example, Figure 4 shows the finite automaton of \(S.W.D\).

After converting path_exp to a finite automaton, the planner multiplies it with the topology and gets a product graph \(G' = (V', E')\). Each node \(u \in V'\) has an attribute dev representing a device in the network and an attribute state representing its state in the finite automaton of path_exp. Given two nodes \(u, v \in V'\), there exists a directed link \((u, v) \in E'\) if (1) \((u.dev, v.dev)\) is a link in the network, and (2) \(\delta(u.state, v.dev) = v.state\). If path_exp has length filters, we trim \(G'\) to only keep paths satisfying the filters. Finally, the planner performs state minimization on \(G'\) to remove redundant nodes [36], and assigns each remaining node \(u\) a unique identifier to get the DPVNet. An example of DPVNet was given in Figure 2c. We refer readers to [50] for a comprehensive tutorial on automata multiplication.

4.2 **Verification Decomposition**

Our key insight is to transform DPV to a counting problem on DPVNet and decompose it into on-device counting tasks. Specifically, an invariant on \(p\) in the form of \((\text{exist count_exp, path}_\text{exp})\) can be verified by counting whether the network can deliver a satisfactory number of copies of \(p\) to the destination along paths in the DPVNet in each universe. It can be solved by a reverse topological traversal of DPVNet (Algorithm 1), during which each node \(u\)
counts the number of copies of $p$ in all $p$’s universes that can reach the destination from $u$.

**Counting at nodes.** Each $u_i$ only keeps unique counting of different universes to avoid information explosion. If $u_i$ is a destination in DPVNet, its count is 1. Denote the downstream neighbors of $u_i$ in DPVNet as $N_d(u_i) = \{v_j\}_j$, and their counting results as sets $(c_{v_j})_j$. Let $b_{ij} = 1$ if the group of next-hops for $p$ on $u_i$.dev includes $v_j$.dev, and 0 otherwise. Define $\otimes$ as the cross-product sum operator for sets, i.e., $c_i \otimes c_j = (a \oplus b | a \in c_i, b \in c_j)$. If $u_i$.dev’s forwarding action for $p$ is of type $\text{ALL}$, the count of $p$ at $u_i$ is,

$$c_{u_i} = \otimes_{j} b_{ij}=1(c_{v_j}). \tag{1}$$

For example, in Figure 2c, for packets in $P_1$, the count at $W$ is $[1]$, the result of $D_1$, because $W$ forwards $P_1$ to only $D_1$.

Define $\oplus$ as the union operator for sets. Let $\delta = 1$ if $u_i$.dev forwards $p$ to at least one device that does not have a corresponding node in $N_d(u_i)$, and 0 otherwise. If $u_i$’s forwarding action for $p$ is of type $\text{ANY}$, the count of $p$ at $u_i$ is,

$$c_{u_i} = \left\{ \begin{array}{ll}
\oplus_{j} b_{ij}=1(c_{v_j}), & \text{if } \delta = 0, \\
(\otimes_{j} b_{ij}=1(c_{v_j})) \oplus 0, & \text{if } \delta = 1.
\end{array} \right. \tag{2}$$

Still in Figure 2c, for packets in $P_3$, the count at $A_1$ is $[0,1]$, the union of $[0]$ from $B_1$ and $[1]$ from $W_2$ because $A_1$’s device $A$ forwards packets in $P_3$ to either $B$ or $W$. The proof sketch of this counting algorithm’s correctness is in Appendix A.1.

**Distributed counting.** This algorithm can be naturally decomposed into lightweight tasks, one for each node $u$ in DPVNet, to enable distributed counting. The planner sends $u$.dev the task of $u$ and its lists of downstream and upstream neighbors. $u$.dev receives the counts from $v_j$.dev, where $v_j \in N_d(u)$, computes $c_u$ using Equations (1)(2), and sends $c_u$ to the corresponding devices of all $u$’s upstream neighbors in DPVNet. In the end, the counts at the source node of DPVNet (e.g. $c_{S1}$ at $S1$ in Figure 2c) are the numbers of copies of $p$ delivered to the destination of DPVNet in all $p$’s universes. The device of the source node can then easily verify the invariant.

**Optimizing counting result propagation.** If there are a huge number of paths in DPVNet, $c_u$ can be large due to ANY-type actions at devices (e.g., a chained diamond topology). Letting $u$.dev sends the complete $c_u$ to the devices of $u$’s upstream neighbors may result in large communication and computation overhead. Given an invariant, we define the **minimal counting information** of $u$ as the minimal set of elements in $c_u$ that needs sending to its upstream nodes so that the source node in DPVNet can correctly verify the invariant, assuming arbitrary data planes at devices and $u$ not knowing the network topology.

For exist countExp operation, suppose two sets $c_1, c_2$ with all non-negative elements. For any $x \in c_1$ and $y \in c_2$, $a = x+y \in c_1 \oplus c_2$ satisfies $a \geq x$ and $a \geq y$. We then have:

**Proposition 1.** Given an invariant with exist countExp operation, the minimal counting information of node $u$ is $\min(c_u)$ (max($c_u$)) if countExp is $\geq N$ or $> N$ ($\leq N$ or $< N$), and the first $\min(c_u)$, 2 smallest elements in $c_u$ if countExp is $== N$. The proof is in Appendix A.2.

For an invariant with an equal operator, we prove that the minimal counting information of any $u$ is 0. Specifically, no node $u$ even needs to compute $c_u$. It only needs to check if $u$.dev forwards any packet specified in the invariant to all the devices corresponding to the downstream neighbors of $u$ in DPVNet. If not, a network error is identified, and $u$.dev can immediately report it. This design enables local verification on generic equivalence invariants, making the local contracts on all-shortest-path availability in RCDC [41] a special case.

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

### 4.3 Compound Invariants

We introduce how to decide on-device tasks for invariants with a logic combination of (exist countExp, pathExp) pairs since the equal operator can be verified locally. Because an invariant with pathExp of different sources can be handled by adding a virtual source device connected to all the sources, we focus on the destinations of pathExp.

**Regular expressions with different destinations.** A natural strawman is to build a DPVNet for each pathExp, let devices count along all DPVNet $s$ and cross-multiply the results at the source. However, it is incorrect. Consider an anycast invariant for $S$ to reach $D$ or $E$, but not both (Figure 5a). It is satisfied in the network. But if we build two DPV Nets, $S1 \rightarrow D1$ and $S2 \rightarrow E1$, one for each destination. After counting on both DPVNet, $S1$ and $S2$ each get $[0,1]$ for $D1$ and $E1$, respectively. The cross-product is $\{0,0,0.1,0.1,0,1,1\}$, raising a false-positive network error.

To address this issue, for such an invariant, we first construct a single DPVNet representing all paths in the network that match...
at least one regular expression in path_exprs by multiplying the union of all regular expressions with the topology. We then specify one counting task for one regular expression at every node in DPVNet, including all destination nodes. Consider the anycast example. The planner computes one DPVNet in Figure 5b. Each node counts the number of packets reaching both D and E. The count of D1 is \(\{S \cdot D, 1\}\) and E1 is \(\{(S \cdot D, 0), (S \cdot E, 1)\}\). After S1 receives these results and processes them using Equation (2), it determines that in each universe, a packet is sent to D or E, but not both, i.e., the invariant is satisfied.

Regular expressions with the same destination. Following the case of different destinations, one strawman is to construct a single DPVNet for the union of such path_exprs. However, because they have the same destination, the counting along DPVNet cannot differentiate the counts for different path_exprs, unless the information of paths is collected and sent along with the counting results. That would lead to large communication and computation overhead at devices.

Another strawman is to construct one DPVNet for one path_expr, count separately and aggregate the result at the source in cross-product. But false positives can arise again. Consider Figure 6a and an invariant \((P, \{S\}, (\text{exist} \geq 2, (S \cdot D \text{ and loop_free}) \text{ or } (\text{exist} \geq 1, S \cdot W \cdot D \text{ and loop_free}))\), which specifies at least two copies of each packet in P should reach D along a simple path, or at least one copy should reach D along a simple path passing W. Figure 6a satisfies this invariant. But if we construct a DPVNet for each path_expr and perform counting separately, S will receive a count \([1, 2]\) for reaching D with a simple path, and a count \([0, 1]\) for reaching D with a simple path passing W. The cross-product \([1, 0], (1, 1), (2, 0), (2, 1)\) raises a phantom error.

We add virtual destination devices to handle such invariants. Suppose an invariant has \(m\) \((\text{exist count_expr, path_exprs})\) pairs where path_exprs have the same destination D. We change D to \(D^1\) and add \(m - 1\) virtual devices \(D^i\) \((i = 2, \ldots, m)\). Each \(D^i\) has the same set of neighbors as D does in the network topology. We then rewrite the destination of path_expr to \(D^1\) \((i = 1, \ldots, m)\). Figure 6b gives the updated topology to handle the invariant above.

Afterward, we take the union of all path_exprs, and intersect it with an auxiliary path_expr specifying any two \(D^i, D^j\) should not co-exist in a path. We then multiply the resulting regular expression with the new topology to generate 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 \(D^i\)'s, and adjust Equations (1)(2) accordingly.

5 DVM PROTOCOL

Given link \((u, v)\) in DPVNet, DVM defines the format and order of messages u.devo sends to u.dev, and the actions u.devo takes when receiving the messages. DVM is inspired by vector-based routing protocols [56, 64]. One distinction is that it needs no loop-prevention mechanism. It is because the messages are sent along the reverse direction in the DAG DPVNet. As such, no message loop will be formed. For ease of presentation, we introduce DVM assuming a single destination.

5.1 Information Storage

Each device stores two types of information: LEC (local equivalence class) and CIB (counting information base). Given a device X, a LEC is a set of packet copies whose actions are identical at X. X stores its LECs in a "packet_space, action" mapping called the LEC table. We choose to encode packet sets as predicates using binary decision diagram (BDD) [14], and use BDD-based DPV tools [83, 93] to maintain a table of minimal number of LECs at devices. It is because in DVM, devices perform packet set operations \((e.g., \cup \text{ and } \cap)\), which can be realized efficiently using logical operations on BDD.

Given a device X, CIB stores for each X.node in DPVNet (i.e., nodes with a device ID X), for different packet sets, the number of packet copies that can reach from X.node to the destination node in DPVNet. For each X.node, X stores three distinct types of CIB:

- \(\text{CIBIn}(u)\) for each of X.node's downstream neighbors v: it stores the latest, unprocessed counting results received from v in a \((\text{predicate}, \text{count})\) mapping;
- \(\text{LocCIB}(X.\text{node})\): it stores for different predicates, the latest number of packet copies that can reach from X.node to the destination node in \((\text{predicate}, \text{count, action, causality})\) tuples, where the causality field records the input to get the count field \((i.e., \text{the right-hand side of Equations } 1(2))\);
- \(\text{CIBOut}(X.\text{node})\): it stores the count results to be sent to the upstream nodes of X.node in \((\text{predicate, count})\) tuples.

Figure 7a gives an example DPVNet, with the counts of node v, z, the LEC table of u.dev, and CIBIn(v), CIBIn(z), and LocCIB(u) at node u. Specifically, the causality field is \([v, P_1, 1], [z, P_2, 1]\) because the count 2 of predicate P1 is computed via the results of both v and z \((i.e., 2 = 1 + 1)\).

5.2 Message Format and Handling

Messages in DVM are sent over TCP connections to ensure in-order message delivery and processing. DVM defines control messages to manage the connections between devices. We focus on the UPDATE message that is used to transfer counting results between devices.

UPDATE message format. An UPDATE message has three fields: (1) intended link: along which link in DPVNet the result is propagated oppositely \((\text{e.g., } (W_1, D_1) \text{ or } (W_2, D_1) \text{ in Figure } 2c))\); (2) withdrawn predicates: a list of predicates whose counting results are obsolete; and (3) incoming counting results: a list of predicates with their latest counts.

UPDATE message principle. DVM maintains an important principle: for each UPDATE, the union of withdrawn predicates equal to the union of the predicates of incoming counting results. It ensures a node always receives the latest, complete counting results from its downstream neighbors, guaranteeing the eventual consistency between the verification result at the source of DPVNet and a stable data plane.
UPDATE message handling. Consider link \((u, v)\) in \(DPVNet\). Suppose \(u\).\texttt{dev} receives an UPDATE message whose intended link is \((u, v)\). \textit{u.dev} handles it in three steps.

\textbf{Step 1: updating \(CIBIn(u)\).} \textit{u.dev} updates \(CIBIn(u)\) by removing entries whose predicates belong to withdrawn predicates and inserting all entries in incoming counting results.

\textbf{Step 2: updating \(LocCIB(u)\).} To update \(LocCIB(u)\), \textit{u.dev} first finds all affected entries, i.e., the ones that need to be updated. To be concrete, an entry in \(LocCIB(u)\) needs to be updated if its \textit{causality} field has one predicate from \(v\) and belongs to the withdrawn predicates of this message. It then updates the counting results of all affected entries one by one. Specifically, for each pair of an affected entry \(r\) and an entry \(r'\) from the incoming counting results, \textit{u.dev} computes the intersection of their predicates. If the intersection is not empty, a new entry \(r'_{\text{new}}\) is created in \(LocCIB(u)\) for predicate \(r.\text{pred} \land r'.\text{pred}\). The count of \(r_{\text{new}}\) is computed in two steps: (1) perform an inverse operation of \(\otimes\) or \(\oplus\) between \(r.\text{count}\) and \(v\)'s previous counting result in \(r.\text{causality}\), to remove the impact of the latter; and (2) perform \(\otimes\) or \(\oplus\) between the result from the last step and \(r'.\text{count}\) to get the latest counting result. The \textit{action} field is the same as \(r\). The \textit{causality} of this entry inherits from that of \(r\), with a tuple \((u, r')\) replacing \(v\)'s previous record. After computing all new entries, all affected entries are removed from \(LocCIB(u)\).

Figure 7b shows how \(u\) in Figure 7a processes an UPDATE message from \(v\).\texttt{dev} to update its \(CIBIn(v)\) and \(LocCIB(u)\).

\textbf{Step 3: updating \(CIBOut(u)\).} \textit{u.dev} puts the predicates of all entries removed from \(LocCIB(u)\) in the withdrawn predicates. For all inserted entries of \(LocCIB(u)\), it strips \textit{action} and \textit{causality}, merges entries with the same \textit{count} value, and puts the results in the incoming counting results.

After processing the UPDATE message, for each upstream neighbor \(w\) of \(u\), \textit{u.dev} sends an UPDATE messaging consisting of an intended link \((w, u)\) and \(CIBOut(u)\).

Internal event handling. If \textit{u.dev} has an internal event (e.g., rule update or link down), we handle it similarly to an UPDATE message. For example, if a link is down, we consider predicates forwarded to that link update their counts to 0. The predicates whose forwarding actions are changed by the update are considered withdrawn predicates and the predicates in incoming counting results of an UPDATE message. Different from regular UPDATE messages, no \(CIBIn(v)\) needs updating. The counts of newly inserted entries in \(LocCIB(u)\) are computed by inverting \(\otimes/\oplus\) and reading related entries in different \(CIBIn(v)\).s. Predicates with new counts are included as withdrawn predicates and incoming counting results in \(CIBOut(u)\).

Handling packet transformation. Suppose device \(X\) needs to compute the counting for \(predicate_1\) and it has a rule that transforms packets in \(predicate_1\) to packets in \(predicate_2\) before forwarding them. In DVM, for each \(X.\text{node}\) in \(DPVNet\), \(X\) sends a SUBSCRIBE message \(\text{sub}(\text{predicate}_1, \text{predicate}_2)\) to all \(v\).\texttt{devs}, where \(v\) is a downstream node of \(X.\text{node}\), to specify that \(v\) should send the counting result of \(\text{predicate}_2\) not \(\text{predicate}_1\), to \(X.\text{node}\). \textit{u.dev} then follows this message to send the counting result of \(\text{predicate}_2\) in UPDATE messages. \(X\) uses this received result to update the counting result of \(\text{predicate}_1\), and sends it to the upstream neighbors of \(X.\text{node}\). If \(X\)'s packet transformation rule is updated later, \(X\) needs to send new SUBSCRIBE messages accordingly.

6 MINIMIZING PLANNER-VERIFIERS COMMUNICATION

We design a mechanism for on-device verifiers to check the fault tolerance of invariants with minimal involvement with the planner, avoiding the latter becoming the bottleneck.

Basic idea: precomputing fault-tolerant \(DPVNet\) and online recounting. Given an invariant with specified fault tolerance (e.g., shortest-path reachability under 2-link-failure), the planner computes a \(DPVNet\) to represent the union of all valid paths in all fault scenes, decomposes it into on-device tasks labeled with different scenes, and sends them to on-device verifiers. Verifiers first perform counting along paths corresponding to the original topology. When a fault scene happens, verifiers detecting link failures flood them using a link state synchronization protocol [31, 32]. After synchronization, the destinations recount along paths in the \(DPVNet\) corresponding to this scene. If an unspecified fault scene or one with no valid path in \(DPVNet\) happens, any device finding this during flooding reports it to the planner.

Specifying fault-tolerance. Operators use the \textit{fault_scenes} field to specify the fault-tolerance of invariants. It is a set of fault scenes \(f_1, f_2, \ldots\), each expressed as a set of failed links. For example, (\(P, [S, (\text{exist} \geq 1, (S.\text{D})), ((A, B), (B, W), (B, D))])\) requires that \(S\) should reach \(D\) not only when all links are up, but also when \((A, B)\) is down and when both \((B, W)\) and \((B, D)\) are down. Syntax sugars are provided to simplify the expression (e.g., \textit{any_two} for all 2-link-failures).

Relating fault-tolerant \(DPVNet\) with \textit{length} \_\textit{filters}. Given an invariant, we compute its fault-tolerant \(DPVNet\) based on the \textit{length} \_\textit{filters} in its \textit{path_exp}. A length filter is concrete if it stays the same in all fault scenes as in the original topology (e.g., \(< 5\) hops), and is symbolic if it may change by fault scenes (e.g., \(==\) \textit{shortest}). Given a network \(G\) and an invariant \(\Psi\), denote the set of valid paths
Why not forward propagation? Although forward 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 [48, 69] and fast switching among data planes [49, 72]) to respond to network errors to improve availability. Forward propagation cannot provide such information.

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 ones. The number of such paths is small even in large networks. Second, for invariants with a huge number of valid paths, Tulkun verifies them via divide-and-conquer: divide the network into abstracted one-big-switches, construct DPVNet on this abstract network, and perform intra-/inter-partition distributed verifications.

Incremental deployment. Tulkun can be deployed incrementally in two non-exclusive ways. One is to assign an off-device instance (e.g., VM) for each device without an on-device verifier, to play as a verifier to collect the data plane from the device and exchange messages with others based on DPVNet. It is a generalization of RCDC, whose local verifiers are deployed in off-device instances. The other is the divide-and-conquer above. We assign one instance for each partition to perform intra-/inter-partition verification.

Verifying transient data planes. Tulkun currently guarantees the eventual consistency between the verification result and the network data plane. To verify transient data planes in networks where the data plane frequently changes, we may extend Tulkun’s DVM protocol to capture and verify stable snapshots of the network data plane by leveraging Libra’s design on taking stable snapshots [90].

Local verification of invariants with exist operators. Consider such an invariant, given a node u in a DPVNet, if we assume u knows the network topology (e.g., through pre-configuration), under certain conditions, the minimal counting information of u could also be Φ, the same as that for invariants with equal operators we proved in §4.2. One such condition is u.dev is a cut of the network (e.g., A in the example network in Figure 2a). A systematic exploration of such conditions is an interesting future research question.

Multi-path comparison. To support “multi-path” invariants that compare the packet traces of two packet spaces (e.g., route symmetry and node-disjointness), Tulkun can extend its language with an id keyword to refer to different packet spaces and allow users to define trace comparison operators. It then constructs the DPVNet for each packet space, lets on-device verifiers collect the actual downstream paths and send them to upstream neighbors, and performs user-defined comparison operations on the collected complete paths.

Security and privacy risks. The on-device verifiers of Tulkun may suffer from security vulnerabilities if their residing network devices are breached. Preventing these breaches from happening is an orthogonal research topic [46]. Tulkun currently has no privacy issue because it operates in a single network. How to extend Tulkun to an interdomain setting while preserving the privacy of different networks is another open research question.

8 IMPLEMENTATION

Our prototype has ~9K lines of Java and Python code, including a verification planner and on-device verifiers (Figure 9). The planner computes the DPVNet based on the invariant and topology, and decides the on-device counting tasks.

In addition to security modules (i.e., authentication and authorization interfaces) like those in other protocols (e.g., SNMP, OSPF and BGP), an on-device verifier has (1) a LEC builder that reads the data plane of the device to maintain a LEC table of a minimal number of LECs, and (2) a verification agent that maintains TCP
connections with the verifiers of neighbor devices, takes in the LEC table and the DVM protocol UPDATE messages from neighbor devices to update the on-devices CIBs, and sends out UPDATE messages with latest counting results to neighbor devices, based on counting tasks. For the verification agent, we use a thread pool implementation, where a thread is assigned for a node in a DPVNet. To avoid creating too many threads and hurting the system performance, we design an opportunistic algorithm to merge threads with similar responsibilities (e.g., invariants with different source IP prefixes but same destination IP prefixes) into a single thread. A dispatcher thread receives events (e.g., a LEC table update or a DVM protocol UPDATE message), and dispatches events to the corresponding thread. A LEC table update is sent to all threads whose invariants overlap with the update, and an UPDATE message is dispatched based on the intended link field of the UPDATE message. For predicate operation and transmission, we adapt and modify the JDD [71] library to support the serialization and deserialization between BDD and the Protobuf data encoding [30], so that BDDs can be efficiently transmitted between devices in UPDATE messages.

9 PERFORMANCE EVALUATION

We conduct extensive evaluations on Tulkun. Specifically, we study four questions: (1) What is the capability of Tulkun in verifying generic invariants? (§9.1) (2) What is the performance of Tulkun in a testbed with different types of network devices, mimicking a real-world WAN? (§9.2) (3) What is the performance of Tulkun in various real-world, large networks under various DPV scenarios? (§9.3) (4) What is the overhead of running Tulkun on commodity network devices? (§9.4)

9.1 Functionality Demonstrations

We build a network of 5 switches in Figure 2a: 3 Mellanox [57], 1 Edgecore [19] and 1 UbiSpace [6], equipped with SONIC [58] or ONL [63]. We run demos to verify (1) loop-free, waypoint reachability from S to D in Figure 2b, (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 [41]. We run each demo with the correct and erroneous data planes. The network always computes the correct results. We also provide an interactive demo in [78].

9.2 Testbed Experiments

We add 1 Mellanox switch and 3 UbiSpace switches to mimic the 9-device INet2 WAN [59]. We install public dataset rules [59] on switches and inject propagation latencies between switches based on INet2 topology [77]. We verify the loop-free, blackhole-free, all-pair reachability along paths with (≤ shortest + 2) hops.

Experiment 1: burst update. We first evaluate Tulkun in the scenario of burst update, i.e., all forwarding rules are installed to corresponding switches all at once. Tulkun finishes the verification in 0.99 seconds, outperforming the best centralized DPV in comparison by 2.09× (Figure 11a).

Experiment 2: incremental update. After the burst update, we randomly generate 10k rule updates and apply and verify them one by one. For 80% of the updates, Tulkun finishes the incremental verification ≤ 5.42ms, outperforming the best centralized DPV in comparison by 4.90× (Figure 11c). This is because in Tulkun, when a rule update happens, only devices whose 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 these affected devices is small [80].

9.3 Large-Scale Simulations

We implement an event-driven simulator to evaluate Tulkun in various networks on a server with 2 Xeon 4210R CPUs.

9.3.1 Simulation Setup. We first introduce the settings.

Datasets. We use 13 datasets in Figure 10. Four are public ones and the others are synthesized with public topologies [35, 40, 47, 67]. FT-48 is a 48-ary fattree [2]. NGDC is a real, Clos-based DC. For WAN, we assign link latencies based on topologies [77]. For LAN and DC, we assign a 10μs link latency.

Comparison methods. We compare Tulkun with five state-of-the-art centralized DPV tools: AP [83], APKeep [93], Delta-net [37], Veriflow [45] and Flash [34]. We also compare Tulkun with APT [86] and Katra [7], two DPV tools designed to support packet transformation, in our technical report [80]. We reproduce Katra, and use the open-sourced version of other tools.

Invariants. We verify the all-pair loop-free, blackhole-free, (≤ shortest + 2)-hop reachability in §9.2 with 3-link-failure for WAN/LAN and all-ToR-pair shortest path reachability for DC. Tulkun also verifies the local contracts of all-shortest-path availability of DC, as RCDC does, in our technical report [80].

Metrics. In all simulations, Tulkun successfully finds all the errors we injected. We compute the verification time as the period from the arrival of rule updates at devices to the time when all invariants are verified, including the propagation delays. For centralized DPV, we randomly assign a device as the location of the verifier, and let all devices send it their data planes along lowest-latency paths. We also study Tulkun’s message overhead [80] and the latency of Tulkun planner to compute DPVNet with different k-link-failures. Figure 13 shows that in 10 out of 11 topologies (removing AT1-2 and AT2-2 for deduplication), Tulkun computes 2-link-failure (3-link-failure) tolerant DPVNet in <95s (<1440s).
9.3.2 Results: Burst Update. Figure 11a gives the verification time of Tulkun, and its acceleration ratio over other tools. For WAN/LAN, Tulkun completes the verification in ≤ 1.60s and achieves an up to 6.21× speedup than the fastest centralized DPV. For DC, Tulkun finishes verifying NGDC in 40.45s, outperforming AP, APKeep and Veriflow (10s of hours) by three orders of magnitude. In AT1-1 and AT2-2 whose topologies are the same scale networks, Tulkun is still 7.4× faster. It is because Tulkun 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). On-device verifiers achieve a very high level of parallelization, enabling scalability. The verification time of all tools is in our technical report [80].

Note that Tulkun is slower than AP and Flash in AT1-1 and AT2-1, but faster in AT1-2 and AT2-2 whose topologies are the same pairwise. It is because the latter two have a much higher number of rules (3.99x and 11.97x). The bottleneck of AP and Flash is to transform rules into equivalence classes (EC), whose time increases linearly with the number of rules. In contrast, Tulkun only computes LEC on devices in parallel, and is not a bottleneck [80]. As such, with more rules, Tulkun becomes faster than AP and Flash.

9.3.3 Results: Incremental Update. We evaluate Tulkun for incremental verification using the same methodology as in §9.2. The 80% quantile verification time of Tulkun is up to 2355× faster than the fastest centralized DPV (Figure 11c). Among all datasets, Tulkun finishes verifying at least 72.72% rule updates in less than 10ms, while this lower bound of other tools is < 1% (Figure 11b). It is for the same reason as in experiments (§9.2), and proves that Tulkun enables scalable DPV under various networks and DPV scenarios.

9.3.4 Results: Fault-Tolerance. For each LAN/WAN, we generate 50 fault scenes of ≤ 3 link failures based on the statistic of Microsoft’s WAN [95]. For each scene, we measure the verification time of recounting along DPVNet with failure flooding (Figure 12a); and generate 1K random rule updates after that to measure the incremental verification time (Figure 12b and 12c). Tulkun consistently outperforms others as in §9.3.3 and §9.3.2. It shows that by computing a fault-tolerant DPVNet and online recounting, Tulkun efficiently verifies fault-tolerant invariants without involving the planner. We observe that Delta-net slightly outperforms Tulkun in verifying the complete network with fault scenes in several datasets. It is because in Tulkun, devices need to update their LECs after fault scenes happen. In contrast, when there is no rule update in fault scenes (i.e., the setting in Figure 12a), centralized DPVs do not need to update their ECs. This observation shows that the EC data structure of Delta-net (i.e., atom) is more effective than those of other centralized DPVs in invariant checking. However, atom only works for destination IP-prefix-based data planes.
**Figure 13**: DPVNet computation latency.

**Figure 14**: Initialization overhead.

**Figure 15**: DVM UPDATE message processing overhead.

When there are rule updates in fault scenes (i.e., the setting in Figure 12b and 12c), centralized DPVs provide comparable performances as Tulkun does only in STFD, the campus network of Stanford. It is because STFD has a much smaller scale than other datasets, in terms of the number of devices, geo-locations and the number of rules. This again demonstrates the scalability of Tulkun.

### 9.4 On-Device Microbenchmarks

We measure the overhead of Tulkun on-device verifiers on four models of commodity switches. The fourth one is a Centec switch using an ARM-based CPU and SONIC.

**Initialization overhead.** For each of 414 devices from WAN / LAN and 6 devices from NGDC/Fattree (one edge, aggregation and core switch, respectively), we measure the overhead of its initialization phase in burst update (i.e., computing the initial LEC and CIB), in terms of total time, maximal memory and CPU load, on all four switch models. The CPU load is computed as CPU time / (total time × number of cores). Figure 14 plots their CDFs. On all four switches, all devices in the datasets complete initialization in \( \leq 1.75s \), with a CPU load \( \leq 0.48 \), and a maximal memory \( \leq 19.6MB \). The Centec switch has the worst time performance because it uses an ARM-based CPU while other sue x86-based CPUs.

**DVM UPDATE message processing overhead.** For all 420 devices in the datasets, we collect the trace of their received DVM UPDATE messages in all the evaluations, replay them consecutively on each switch, and measure the message processing overhead in terms of total time, maximal memory, CPU load and per message processing time (Figure 15). For 90% of devices, all four switches process all UPDATE messages in \( \leq 0.29s \), with a maximal memory \( \leq 19.57MB \), and a CPU load \( \leq 0.24 \). And for 90% of all 2895.62k UPDATE messages, the switches can process it in \( \leq 3.52ms \).

These results show that Tulkun on-device verifiers can be deployed on commodity switches with little overhead.

### 10 RELATED WORK

Network verification includes CPV that checks errors in configurations [1, 5, 8–10, 21, 23, 24, 26, 28, 29, 42, 62, 68, 73, 76, 87, 94]; and DPV that checks errors in the data plane. Tulkun is a DPV tool, and can help simulation-based CPV [24, 51, 54] verify the simulated DP.

**Centralized DP.** Existing DPV tools [3, 34, 37, 41, 43–45, 53, 55, 61, 74, 75, 82, 83, 85, 86, 90, 92, 93] use a centralized verifier to collect and analyze the data planes. Despite optimization efforts, centralized DPV does not scale due to the need for reliable verifier-network connections and the verifier being a bottleneck and single PoF. They also lack explicit support for generic invariants such as anycast, multicast, no redundant routing and 1+1 routing. Libra [90], RCDC [41] and Flash [34] focus on scale up DPV using parallelization and batch processing. However, they are still centralized designs with the limitations above. Our position paper [81] proposed the idea of distributed DPV, but left many important questions unanswered. In contrast, we design Tulkun with several key components to systematically decompose DPV into tasks executed on network devices, achieving scalable DPV on generic invariants with little overhead and minimal involvement of a centralized component.

**Verification of stateful/programmable DP.** Some studies investigate the verification of stateful DP [15, 60, 88, 89, 91] and programmable DP (e.g., P4 [13]) [18, 52]. Extending Tulkun to stateful and programmable DP is an interesting future work.

**Network synthesis.** Synthesis [11, 20, 39, 66, 70] is complementary to verification. Tulkun is inspired by some of them [11, 39, 66] to use automata theory to generate DPVNet.

**Predicate representation.** Tulkun chooses BDD [14] to represent packets for its efficiency. Recent data structures (e.g., ddNF [12] and PEC [38]) may benefit Tulkun.

### 11 CONCLUSION

We design Tulkun, a distributed DPV framework to achieve scalable DPV by decomposing verification to lightweight on-device counting tasks. Experiments demonstrate the benefits of Tulkun. This work does not raise any ethical issues.

**ACKNOWLEDGMENTS**

We are extremely grateful for the anonymous SIGCOMM reviewers for their wonderful and constructive feedback. We have incorporated their comments in the manuscript. We thank Mina Tahmashi Arashloo, Wei Bai, Mahesh Balakrishnan, Manya Ghobadi, Dong Guo, Geng Li, Srinivas Narayana, Harvey Newman, Ruzica Piskac, and Jiwu Shu for their help and suggestions during the preparation of this paper. Qiao Xiang, Chenyang Huang, Ridi Wen, Yuxin Wang, and Xiwen Fan are supported in part by the National Key R&D Program of China 2022YFB2901502, NSFC Award 62172345, Open Research Projects of Zhejiang Lab 2022QQA0AB05, Future Network Innovation Award of Ministry of Education of China 2021FNA02008, and NSF-Fujian-China 2022J01004. Linghe Kong is supported in part by NSFC Award 62141220, 61972253, and U1908212.
Appendices are supporting material that has not been peer-reviewed.

A PROOFS OF DPVNET BACKWARD COUNTING

A.1 Proof Sketch of the Correctness of the Counting Algorithm

For presentation purposes, we first summarize the backward counting algorithm in DPVNet in Algorithm 1. Given a packet \( p \) and a DPVNet, the goal of Algorithm 1 is to compute the number of copies of \( p \) that can be delivered by the network to the destination of DPVNet along paths in the DPVNet in each universe. Suppose Algorithm 1 is incorrect. There could be three cases: (1) there exists a path in DPVNet that is provided by the network data plane, but is not counted by Algorithm 1; (2) There exists a path in DPVNet that is not provided by the network data plane, but is counted by Algorithm 1; (3) Algorithm 1 counts a path out of DPVNet. None of these cases could happen because at each node \( u \), Equations (1) (2) only counts \( e_{ij} \) of \( e_j \) with \( h_{ij} = 1 \), i.e., the downstream neighbors of \( u \) whose devices are in the next-hops of \( u.dev \) forwarding \( p \) to. As such, Algorithm 1 is correct.

A.2 Proof of Proposition 1

Consider \( c_u \) of packet \( p \) at \( u \), and an upstream neighbor of \( u \), denoted as \( w \). Suppose \( u.dev \) is in the group of next-hops where \( w.dev \) forwards \( p \). Because of the monotonicity of \( \oplus \), in each universe that \( w.dev \) forwards \( p \) to \( u.dev \), the number of copies of \( p \) that can be sent from \( w \) to the destination in DPVNet is greater than or equal to the number of copies of \( p \) that can be sent from \( u \) to the destination in DPVNet. As such,

* When \( \text{count}_{\text{exp}} \) is \( \geq N \) or \( > N \), each \( u \) only sends \( \min(c_u) \) to its upstream neighbors. With such information, in the end, the source node of DPVNet can compute the lower bound of the number of copies of \( p \) delivered in all universes. If this lower bound satisfies \( \text{count}_{\text{exp}} \), then all universes satisfy it. If this lower bound does not satisfy \( \text{count}_{\text{exp}} \), a network error is found.
* When \( \text{count}_{\text{exp}} \) is \( \leq N \) or \( < N \), each \( u \) only sends \( \max(c_u) \) to its upstream neighbors. The analysis is similar, with the source node computing the upper bound.
* When \( \text{count}_{\text{exp}} \) is \( = N \), if \( c_u \) has more than 1 count, it means any action to forward \( p \) to \( u \) would mean a network error. In this case, \( u \) only needs to send its upstream neighbors any 2 counts in \( c_u \) to let them know that. If \( c_u \) has only 1 count, \( u \) sends it to \( u \)'s upstream neighbors for further counting. Summarizing these two sub-cases, \( u \) only needs to send the first \( \min(|c_u|, 2) \) smallest elements in \( c_u \) to its upstream neighbors.

With this analysis, we complete the proof of Proposition 1.

B ARTIFACT APPENDIX

Abstract

The artifact provides an implementation of Tulkun using Java and Python. It includes all key components in the paper and the necessary datasets for reproducing the evaluation results in the paper.

Scope

The artifact allows to validate the following evaluation results:

1. The planner parses the invariant specification language (§3) and generates DPVNet (§4, Figure 13)
2. The results of testbed experiments (§9.2).
3. The effects of burst update (§9.3.2), incremental update (§9.3.3) and fault-tolerance (§9.3.4).
4. The overhead of Tulkun on-device verifiers (§9.4).

Note that the exact values may vary on different machines (even with the same CPU and memory configuration). The artifact is only allowed for research purposes.

Contents

The artifact includes the following contents:

1. An implementation of Tulkun planner.
2. An implementation of Tulkun on-device verifier.
3. A simulator that allows Tulkun to be simulated on a single machine.
4. The datasets (Figure 10) include topology, FIB, and packet space.

Hosting

The artifact is hosted on GitHub.

Requirements

- The planner requires a server with at least 16GB memory and requires Java 8.
- The simulator requires a server with at least 16GB of memory and requires Python 3.9+.
- The on-device verifiers require network devices to have JDK 8 and may need to be adjusted for specific devices.