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

Paper ID: #348, Number of Pages: 12 pages + References (3 pages) + Appendices (7 pages)

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). In this paper, we tackle the scalability challenge of DPV 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 *DVNet* 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 less than 41 seconds while other tools need several minutes or up to tens of hours, and shows an up to 2355× speed up on 80% quantile of incremental verification with small overheads on commodity network devices.

1 INTRODUCTION

There has been a long line of research on data plane verification [1–21]. Earlier tools analyzed a snapshot of the complete data plane of the network to find network errors (*e.g.*, blackholes, waypoint violation and forwarding loops) [1–6, 8, 10–17]; and recent solutions focus on incremental verification (*i.e.*, verifying forwarding rule updates) [7–9, 18–21]. State-of-the-art DPV tools (*e.g.*, [20]) 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 [20] ~1 hour to verify a 48-ary fattree (§8.3). 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 [5–7, 21] have tried to tackle these limitations of centralized DPV. Libra [5] 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 [21] 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 [6] 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 [6] proposes an early detection mechanism to detect data plane violations with incomplete information. However, our test using its open-sourced prototype [23] shows that even if the verifier misses the updated rules of only three devices, Flash detects zero errors in 9 out of 11 LAN/WAN datasets (Appendix A).

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 [24, 25], real-time control plane repair [26], fast rollback and switching among multiple data planes [27–29], and interdomain DPV [30, 31]) 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 [21] 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 a previous position paper by Xiang et al., also suggested the promise of distributed DPV [32], 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 uses it and the network topology to compute *DVNet*, 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 *DVNet*. The latter can be solved by a reverse topological traversal along *DVNet*. In its turn, each node in *DVNet* 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 *DVNet*. This traversal can be naturally decomposed into on-device counting tasks, one for each node in *DVNet*, and distributed to the corresponding network devices. We design optimizations to compute the minimal counting information of each node in *DVNet* 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 [21] is a special case of Tulkun.

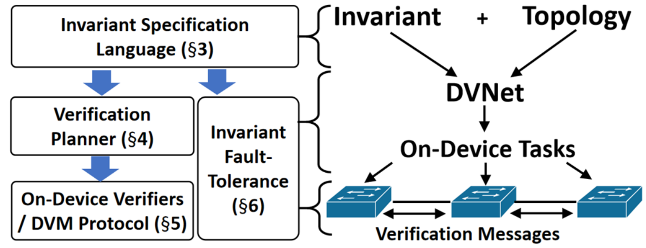


Figure 1: The architecture and workflow of Tulkun.

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 [33, 34] 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 *DVNet* representing the union of all valid paths in all operator-specified failure scenes and sends tasks to verifiers. When failures happen, verifiers adaptively adjust their tasks to count along paths in the *DVNet* corresponding to the updated topology, without contacting the planner.

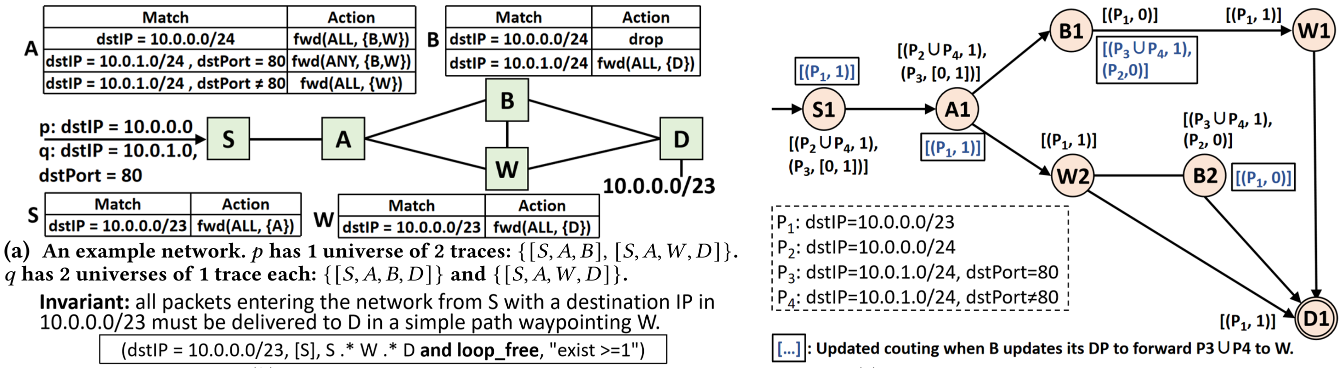
Evaluation results (§8). Tulkun is being evaluated by a major vendor to integrate into its commodity switches. It is also being reviewed as a feature of a major open-source network OS. We will also release the prototype of Tulkun upon the publication of this paper. A preliminary demo can be found at [35]. 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 [21, 36]. An empty group means the action is to drop the packet.



(a) An example network. p **has 1 universe of 2 traces:** $\{[S, A, B], [S, A, W, D]\}$. q **has 2 universes of 1 trace each:** $\{[S, A, B, D]\}$ **and** $\{[S, A, W, D]\}$.

(b) An example invariant.

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

Figure 2: An illustration example to demonstrate the workflow of Tulkun.

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 [36].

Packet traces and universes. Inspired by NetKAT [17], 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 p 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 [37], 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 ($packet\_space$, $ingress\_set$, $behavior$) tuple. The **semantic** means: for each packet p in $packet\_space$ 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. Figure 2b gives the program of the example invariant, where **loop_free** is a shortcut in the language for a regular expression that accepts no path with a loop. It specifies that when any p destined to 10.0.0.0/23 enters from S, at least 1 copy of it will be delivered to D along a simple path waypointing W.

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 *DVNet* to decompose the DPV problem into small on-device verification tasks, and distribute them to on-device verifiers for distributed execution.

From invariant and topology to ***DVNet*****.** The planner leverages the automata theory [38] to multiply the regular expression $path\_exp$ and the topology and get a DAG called *DVNet*. Similar to the product graph [39–41], a *DVNet* 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 *DVNet* in our example. Devices in the network and nodes in *DVNet* have a 1-to-many mapping. Each node u in *DVNet* 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 *DVNet*, because the regular expression allows packets to reach D via $[B, W, D]$ or $[W, B, D]$.

Backward counting along *DVNet*. With *DVNet*, a DPV problem is transformed into a counting problem on *DVNet*: *given a packet p, can the network deliver a satisfactory number of copies of p to the destination node along paths in the DVNet 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 *DVNet* in reverse topological order. In its turn, each node u takes as input (1) the data plane of $u.dev$ 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 *DVNet*, by the network data plane, to compute the number of copies that can be delivered from u to the destination along *DVNet* by the network data plane. In the end, the source node of *DVNet* computes the final counting result.

Figure 2c illustrates this process. We use P_1, P_2, P_3, P_4 to represent the packet spaces $\{dstIP = 10.0.0.0/23\}$, $\{dstIP = 10.0.0.0/24\}$, $\{dstIP = 10.0.1.0/24, dstPort = 80\}$, and $\{dstIP= 10.0.1.0/24, dstPort \neq 80\}$, respectively. P_2, P_3 and P_4 are disjoint and $P_1 = P_2 \cup P_3 \cup P_4$. Each u in *DVNet* 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). We traverse all the nodes in *DVNet* 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, 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, 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 device W only forwards P_1 to D, not B.

Given a node u in *DVNet*, if the action of forwarding is of ANY-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 $A1$ 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 $A1$ is $(P_3, 1)$ because $W3$'s updated mapping is $(P_1, 1)$. Therefore, the updated mapping of P_3 at $A1$ 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 *DVNet*, 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 the number of copies that can be delivered from u to the destination along *DVNet*, by the network data plane in each universe; and (3) sends the computed result to devices where its upstream neighbors in *DVNet* 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 $u.dev$.

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 *DVNet* 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 plannner 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 $[(P_1, 0)]$, and sends the updates to A along $(B1, A1)$ and W along $(B2, W2)$, respectively. 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 $A1$ does not change; (2) no matter whether A forwards P_3 to B or W, 1 copy of each packet will be sent to D, and (3)

$$
\begin{array}{rcl}
\mathit{invs} & ::= & \mathit{inv}^* \\
\mathit{inv} & ::= & (\mathit{packet\_space}, \mathit{ingress\_set}, \mathit{behavior}, \\
 & & [\mathit{fault\_scenes}]) \\
\mathit{behavior} & ::= & (\mathit{match\_op},\ \ \mathit{path\_exp}) \mid \textbf{not}\ \mathit{behavior} \\
 & \mid & \mathit{behavior}\ \textbf{or}\ \mathit{behavior} \\
 & \mid & \mathit{behavior}\ \textbf{and}\ \mathit{behavior} \\
\mathit{path\_exp} & ::= & (\text{regular expression over the set of devices,} \\
 & & [\mathit{length\_filters}]) \\
\mathit{match\_op} & ::= & \textbf{exist}\ \ \mathit{count\_exp} \mid \textbf{equal} \\
\mathit{exist\_exp} & ::= & == N \mid >= N \mid > N \mid <= N \mid < N
\end{array}
$$

Figure 3: The basic abstract syntax of the Tulkun invariant specification language.

$P_2 \cup P_3 \cup P_4 = P_1$. 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 ($packet\_space$, $ingress\_set$, $behavior$) tuple, with semantics as explained in §2.2.1. Operators can also include an optional field $fault\_scenes$ in the tuple to specify fault tolerance of invariants (see §6 for details). To specify behaviors, we use the building block of ($match\_op$, $path\_exp$) entries. The basic syntax provides two $match\_op$ operators. One is **exist** $count\_exp$, which requires that in each universe, the number of traces matching $path\_exp$ satisfies $count\_exp$. 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 $path\_exp$ [21]. 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, $(S.^*D, (<= \textbf{shortest} + 1))$ 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 ($match\_op$, $path\_exp$) 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 [4, 9, 12, 18, 20], and more advanced invariants (*e.g.*, multicast, anycast and no-redundant-delivery routing). Another example is an invariant given in Azure RCDC [21], 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).

Invariants	Tulkun specifications
Reachability [10, 37, 42]	$(P, [S], (\textbf{exist} >= 1, S.^*D))$
Isolation [10, 37, 42]	$(P, [S], (\textbf{exist} == 0, S.^*D))$
Loop-freeness [42]	$(P, [S], (\textbf{exist} == 0, .^*\ \textbf{and}\ \textbf{not}((\textbf{not}\ X)^*\ \textbf{or}\ ((\textbf{not}\ X)^*X(\textbf{not}\ X)^*))\ \textbf{and}\ ((\textbf{not}\ Y)^*\ \textbf{or}\ ((\textbf{not}\ Y)^*Y(\textbf{not}\ Y)^*))\ \ldots,))$
Black hole freeness[42]	$(P, [S], (\textbf{exist} == 0, .^*\ \textbf{and}\ \ \textbf{not}\ S.^*D))$
Waypoint reachability [9]	$(P, [S], (\textbf{exist} >= 1, S.^*W.^*D))$
Reachability with limited path length [9]	$(P, [S], (\textbf{exist} >= 1, SD \mid S.D \mid S..D))$
Different-ingress same reachability [18, 42]	$(P, [X, Y], (\textbf{exist} >= 1, X.^*D \mid Y.^*D))$
All-shortest-path reachability [21]	$(P, [S], (\textbf{equal}, (S.^*D\ , (== \textbf{shortest}))))$
Non-redundant reachability [Tulkun]	$(P, [S], (\textbf{exist} == 1, S.^*D))$
Mulicast [Tulkun]	$(P, [S], ((\textbf{exist} >= 1, S.^*D)\ \textbf{and}\ (exist >= 1, S.^*E)))$
Anycast [Tulkun]	$(P, [S], ((\textbf{exist} >= 1, S.^*D)\ \textbf{and}\ (\textbf{exist} == 0, S.^*E))\ \ \textbf{or}\ \ ((\textbf{exist} == 0, S.^*D)\ \textbf{and}\ (\textbf{exist} == 1, S.^*E)))$

Table 1: Tulkun specifications for selected invariants.

Note that once an invariant is specified, Tulkun checks whether it is consistently satisfied across all universes. As such, the multipath consistency [10, 37] 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 (**match** >= 1 $path\_exp$) **and** (**match** == 0 .* **and** (**not** $path\_exp$)). We omit their details for the sake of simplicity.

Expressiveness and limitation. This language can express all "single-path" invariants that require the packet traces of one packet space to satisfy a certain regular expression pattern. It covers all invariants studied in DPV literature, except for middlebox traversal symmetry [10] (*i.e.*, S-D and D-S must pass the same middlebox). We discuss how to extend Tulkun to specify and verify such "multi-path" invariants that compare the packet traces of two packet spaces (*e.g.*, route symmetry and path node-/ link-disjointness) in §7.

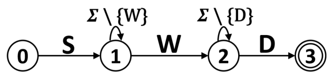


Figure 4: The finite automaton of $S.^*W.^*D$ with an alphabet $\Sigma = \{S, W, A, B, D\}$.

4 VERIFICATION PLANNER

We introduce *DVNet* 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 *DVNet*

Given a *path_exp* and a network, *DVNet* is a DAG representing all paths in the network that matches *path_exp*. *DVNet* can be constructed in different ways (*e.g.*, graph dual variables). We are inspired by network synthesis [39–41] and leverage the automata theory [38] for *DVNet* construction.

Specifically, given a *path_exp*, we first convert its regular expression into a finite automaton $(\Sigma, Q, F, q_0, \delta)$. Σ 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, and assigns each remaining node u a unique identifier to get the *DVNet*. An example of *DVNet* was given in Figure 2c.

4.2 Verification Decomposition

Our key insight is to transform DPV to a counting problem on *DVNet* and decompose it into on-device counting tasks. Specifically, an invariant on p in the form of (**exist** *count_exp*, *path_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 *DVNet* in each universe. It can be solved by a reverse topological traversal of *DVNet*, 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 *DVNet*, its count is 1. Denote the downstream neighbors of u_i in *DVNet* as $N_d(u_i) = \{v_j\}_j$, and their counting results as sets $\{\mathbf{c}_{v_j}\}_j$. 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.*, $\mathbf{c}_1 \otimes \mathbf{c}_2 = (a + b | a \in \mathbf{c}_1, b \in \mathbf{c}_2)$. If $u_i.dev$'s forwarding action for p is of type *ALL*, the count of p at u_i is,

$$\mathbf{c}_{u_i} = \otimes_{j:b_{ij}=1}(\mathbf{c}_{v_j}). \tag{1}$$

For example, in Figure 2c, for packets in P_1, the count at $W1$ is $[1]$, the result of $D1$, because W forwards P_1 to only D.

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 ANY, the count of p at u_i is,

$$\mathbf{c}_{u_i} = \begin{cases} \oplus_{j:b_{ij}=1}(\mathbf{c}_{v_j}), & \text{if } \delta = 0, \\ (\oplus_{j:b_{ij}=1}(\mathbf{c}_{v_j})) \oplus \mathbf{0}, & \text{if } \delta = 1. \end{cases} \tag{2}$$

Still in Figure 2c, for packets in P_3, the count at $A1$ is $[0, 1]$, the union of $[0]$ from $B1$ and $[1]$ from $W2$ because $A1$'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 B.1.

Distributed counting. This algorithm can be naturally decomposed into lightweight tasks, one for each node u in *DVNet*, 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 $\mathbf{c}_u$ using Equations (1)(2), and sends $\mathbf{c}_u$ to the corresponding devices of all u's upstream neighbors in *DVNet*. In the end, the counts at the source node of *DVNet* (*e.g.*, $\mathbf{c}_{S1}$ at $S1$ in Figure 2c) are the numbers of copies of p delivered to the destination of *DVNet* in all p's universes. The device of the source node can then easily verify the invariant.

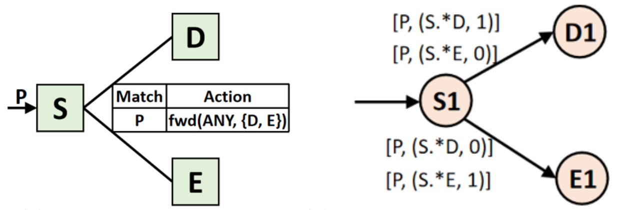
Optimizing counting result propagation. If there are a huge number of paths in *DVNet*, $\mathbf{c}_u$ can be large due to ANY-type actions at devices (*e.g.*, a chained diamond topology). Letting $u.dev$ send the complete $\mathbf{c}_u$ to the devices of u's upstream neighbors may result in large communication and computation overhead. Given an invariant, we define the *minimal counting information* of u as the minimal set of elements in $\mathbf{c}_u$ that needs sending to its upstream nodes so that the source node in *DVNet* can correctly verify the invariant, assuming arbitrary data planes at devices.

For **exist** *count_exp* operation, suppose two sets $\mathbf{c}_1$, $\mathbf{c}_2$ with all non-negative elements. For any $x \in \mathbf{c}_1$ and $y \in \mathbf{c}_2$, $a = x + y \in \mathbf{c}_1 \otimes \mathbf{c}_2$ satisfies $a \geq x$ and $a \geq y$. We then have:

Proposition 1. *Given an invariant with* **exist** *count_exp operation, the minimal counting information of node u is $min(\mathbf{c}_u)$ ($max(\mathbf{c}_u)$) if count_exp is $\geq N$ or $> N$ ($\leq N$ or $< N$), and the first $min(|\mathbf{c}_u|, 2)$ smallest elements in $\mathbf{c}_u$ if count_exp is $== N$. The proof is in Appendix B.2.*

For an invariant with an **equal** operator, we prove that the minimal counting information of any u is $\emptyset$. Specifically, no node u even needs to compute $\mathbf{c}_u$. 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 *DVNet*. 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 [21] a special case.

Computing consistent counting results. Counting tasks are event-driven. Given an event (*e.g.*, a rule update or a count



(a) A network for anycast. (b) The correct *DVNet* and counting.

Figure 5: Verifying an anycast, an invariant with multiple *path_exp* with different destinations.

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 *DVNet* will eventually update its count result to be consistent with the network data plane.

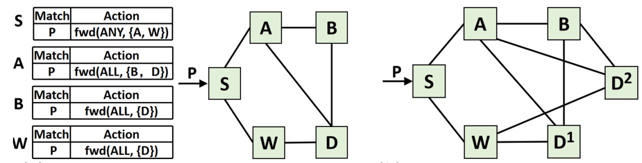
4.3 Compound Invariants

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

Regular expressions with different destinations. A natural strawman is to build a *DVNet* for each *path_exp*, let devices count along all *DVNet*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 *DVNet*s, $S1 \rightarrow D1$ and $S2 \rightarrow E1$, one for each destination. After counting on both *DVNet*s, $S1$ and $S2$ each get $[0, 1]$ for $D1$ and $E1$, respectively. The cross-product is $[(0,0), (0,1), (0,1), (1,1)]$, raising a false-positive network error.

To address this issue, for such an invariant, we first construct a single *DVNet* representing all paths in the network that match at least one regular expression in *path_exp*s 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 *DVNet*, including all destination nodes. Consider the anycast example. The planner computes one *DVNet* in Figure 5b. 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 $[(S.^*D, 0), (S.^*E, 1)]$. 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 also construct a single *DVNet* for the union of such *path_exp*s. However, because they have the same destination, the counting along *DVNet* cannot differentiate the counts for different *path_exp*s, unless the information of paths is collected and



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

Figure 6: Verifying an invariant with multiple *path_exp*s with the same destination.

sent along with the counting results. That would lead to large communication and computation overhead at devices.

Another strawman is to construct one *DVNet* for one *path_exp*, count separately and aggregate the result at the source in cross-product. But false positives can arise again. Consider Figure 6a and an invariant $(P, [S], (\textbf{exist} >= 2, (S.^*D$ **and loop_free**) **or** (**exist** $>= 1, S.^*W.^*D$ **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 *DVNet* for each *path_exp* 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 (**exist** $count\_exp_i, path\_exp_i$) pairs where $path_exp_i$s have 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, in the network topology. We then rewrite the destination of $path\_exp_i$ to D^i ($i = 1, \ldots, m$). Figure 6b gives the updated topology of Figure 6a to handle the invariant above.

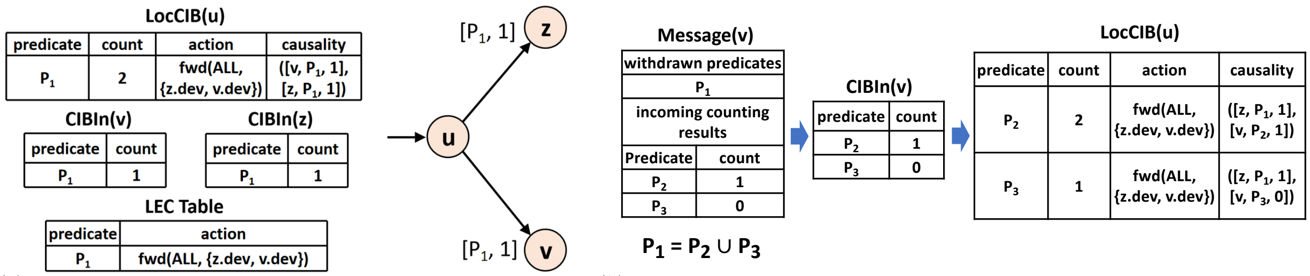
Afterward, we take the union of all *path_exp*s, and intersect it with an auxiliary *path_exp* specifying any two D^i, D^j should not co-exist in a path. We then multiply the resulting regular expression with the new topology to generate one single *DVNet*. Counting can then proceed as the case for regular expressions with different destinations, by letting each device treat all its actions forwarding to D as forwarding to *all* D^is, and adjust Equations (1)(2) accordingly.

5 DVM PROTOCOL

Given link (u, v) in *DVNet*, DVM defines the format and order of messages $v.dev$ sends to $u.dev$, and the actions $u.dev$ takes when receiving the messages. DVM is inspired by vector-based routing protocols [33, 34]. One distinction is that it *needs no loop-prevention mechanism*. It is because the messages are sent along the reverse direction in the DAG *DVNet*. 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



(a) A *DVNet* with LEC table of $u.dev$, *CIBIn* and *LocCIB* of u. **(b)** $u.dev$ handles an UPDATE from $v.dev$ to update $CIBIn(v)$ and $LocCIB(u)$.

Figure 7: An illustration example to demonstrate the key data structure and process of the DVM protocol.

device X, a LEC is a set of packets 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 [43]), and use BDD-based DPV tools [12, 20] to maintain a table of minimal number of LECs at devices. It is because in DVM, devices perform packet set operations (*e.g.*, $\cup$ and $\cap$), which can be realized efficiently using logical operations on BDD.

Given a device X, CIB stores for each $X.node$ in *DVNet* (*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 *DVNet*. Specifically, for each $X.node$, X stores three distinct types of CIB:

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

Figure 7a gives an example *DVNet*, 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 $([v, P_1, 1], [z, P_1, 1])$ because the *count* 2 of predicate P_1 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. 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 *DVNet* the result is propagated oppositely ((*e.g.*, $(W1, D1)$ or $(W2, D1)$ 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 *DVNet* and a stable data plane.

UPDATE message handling. Consider link (u, v) in *DVNet*. Suppose $u.dev$ receives from $v.dev$ an UPDATE message whose intended link is (u, v). $u.dev$ handles it in three steps.

Step 1: updating $CIBIn(v)$. $u.dev$ updates $CIBIn(v)$ by removing entries whose predicates belong to withdrawn predicates and inserting all entries in incoming counting results.

Step 2: updating $LocCIB(u)$. To update $LocCIB(u)$, $u.dev$ 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 *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, $u.dev$ computes the intersection of their predicates. If the intersection is not empty, a new entry r^{new} is created in $LocCIB(u)$ for predicate $r.pred \cap r'.pred$. The *count* of r^{new} is computed in two steps: (1) perform an inverse operation of $\otimes$ or $\oplus$ between $r.count$ and v's previous counting result in $r.causality$, to remove the impact of the latter; and (2) perform $\otimes$ or $\oplus$ between the result from the last step and $r'.count$ to get the latest counting result. The *action* field is the same as r. The *causality* of this entry inherits from that of r, with a tuple (v, 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 $v.dev$ to update its $CIBIn(v)$ and $LocCIB(u)$.

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

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

Rule update handling. If $u.dev$ has a rule update, 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 count 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 $X.node$ in *DVNet*, X sends a SUBSCRIBE message $sub(predicate_1, predicate_2)$ to all $v.dev$s, where v is a downstream node of $X.node$, to specify that v should send the counting result of $predicate_2$, not $predicate_1$, to $X.node$. $v.dev$ then follows this message to send the counting result of $predicate_2$ in UPDATE messages. X uses this received result to update the counting result of $predicate_1$, and sends it to the upstream neighbors of $X.node$. 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 *DVNet* and online recounting. Given an invariant with specified fault tolerance, (*e.g.*, shortest-path reachability under 2-link-failure), the planner computes a *DVNet* 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 [44, 45]. After synchronization, the destinations recount along paths in the *DVNet* corresponding to this scene. If an unspecified fault scene or one with no valid path in *DVNet* happens, any device finding this during flooding reports it to the planner.

Specifying fault-tolerance. Operators use the $fault\_scenes$ 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], (**exist** >= 1, ($S.^*D$), ({(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.*, **any_two** for all 2-link-failures).

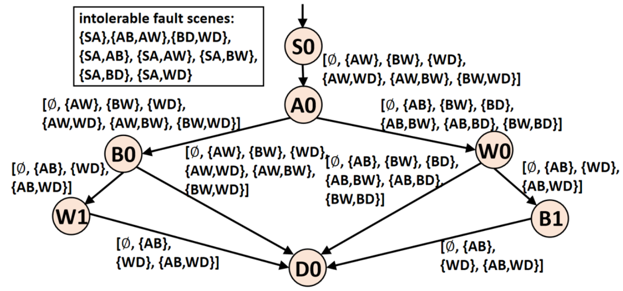


Figure 8: Fault-tolerant *DVNet* of ($\leq$ shortest + 1) reachability from S to D in Figure 2a with 2-link-failure.

Relating fault-tolerant *DVNet* with $length_filter$s. Given an invariant, we compute its fault-tolerant *DVNet* based on the $length\_filters$ in its $path\_exp$. 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.*, == *shortest*). Given a network G and an invariant Ψ, denote the set of valid paths as $R(G, \Psi)$. Given a fault scene f, its topology G_f is a subgraph of G. We have:

Proposition 2. *If Ψ has no symbolic length filter, for any fault scene f, $R(G_f, \Psi) \subset R(G, \Psi)$. Otherwise, for any two fault scenes f, f' such that $f' \subset f$, $R(G_f, \Psi) \subset R(G_{f'}, \Psi)$.*

Computing fault-tolerant *DVNet*. Given an invariant Ψ with fault tolerance but no symbolic length filter, its fault-tolerant *DVNet* is the same as that without any failure, a direct result of Proposition 2. For such an invariant, when a link fails, the verifiers on the devices of this link do not flood the fault scene, but update the counts of predicates forwarded along this link as 0 and propagate these updated counts to other devices along the *DVNet*.

Given an Ψ with symbolic filters and a network G, the planner traverses all fault scenes, including the original topology, in ascending order of the number of failed links, to iteratively compute valid paths for each scene and label them. For each fault scene f, if $R(G, \Psi)$ does not use any link in f, the algorithm skips f because $R(G, \Psi) = R(G_f, \Psi)$. Otherwise, it first computes the concrete values of symbolic length filters for paths that match reg_exp in G_f. For each filter, it looks for a maximal subset fault scene f' that is previously traversed and has the same filter values as f. If f' is found, it checks all the valid paths of f' and labels the ones that still exist when all links in $f - f'$ fail as the valid paths of f. If no f' is found, the algorithm performs a breadth-first-search to find the set of valid paths matching reg_exp and the filters in G_f. If no valid path is found for f, Tulkun records it as an intolerable fault scene. Intermediate results (*e.g.*, paths matching reg_exp but not the filters in G_f or the other way around) are stored for incremental search in the next iteration.

The algorithm's correctness also lies in Proposition 2. Figure 8 shows the fault-tolerant *DVNet* of invariant ($dstIP$ = 10.0.0.0/23, [S], (**exist** >= 1, ($S.^*D$, (<= **shortest** + 1)), (**any_two**)) in the topology in Figure 2a.

7 DISCUSSION

Why not forward propagation? Although forward propagation along *DVNet* can also get the correct result, we choose backpropagation 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 [24, 25] and fast switching among data planes [27, 29]) 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 *DVNet* 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 *DVNet*. 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.

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 *DVNet* 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.

8 PERFORMANCE EVALUATION

We implement a prototype of Tulkun in Java with ~9K LoC (Appendix C) and conduct extensive evaluations. We study four questions: (1) What is the capability of Tulkun in verifying generic invariants? (§8.1) (2) What is the performance of Tulkun in a testbed with different types of network devices, mimicking a real-world WAN? (§8.2) (3) What is the performance of Tulkun in various real-world, large networks under various DPV scenarios? (§8.3) (4) What is the overhead of running Tulkun on commodity network devices? (§8.4)

8.1 Functionality Demonstrations

We build a network of 6 switches in Figure 2a: 4 Mellanox [46], 1 Edgecore [47] and 1 UfiSpace [48], equipped with SONiC [49] or ONL [50]. 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 [21]. We run each demo with correct and erroneous data planes. The network always computes the right results. Details and an interactive demo can be found at [35].

Network	#Device	#Links	#Rules	Type	Network	#Device	#Links	#Rules	Type
INet2 [51]	9	28	7.74×10^4	WAN	NTT	47	63	1.98×10^5	WAN
B4-13	12	18	7.92×10^4	WAN	AT2-1 [19]	68	158	3.81×10^4	WAN
STFD [4]	16	74	3.84×10^3	LAN	AT2-2	68	158	4.56×10^5	WAN
AT1-1 [19]	16	26	2.83×10^4	WAN	OTEG	93	103	7.22×10^5	WAN
AT1-2	16	26	9.60×10^4	WAN	FT-48	2,880	55,296	3.31×10^6	DC
B4-18	33	56	2.11×10^5	WAN	NGDC	6,016	43,008	3.23×10^7	DC
BTNA	36	76	2.52×10^5	WAN					

Figure 9: Datasets statistics.

8.2 Testbed Experiments

We add 3 UfiSpace switches to mimic the 9-device INet2 WAN [51]. We install public dataset rules [51] on switches and inject propagation latencies between switches based on INet2 topology [52]. We verify the loop-free, blackhole-free, all-pair reachability along paths with ($\leq$ **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 10a).

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 $\leq 5.42ms$, outperforming the best centralized DPV in comparison by 4.90× (Figure 10c). 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 (Appendix F).

8.3 Large-Scale Simulations

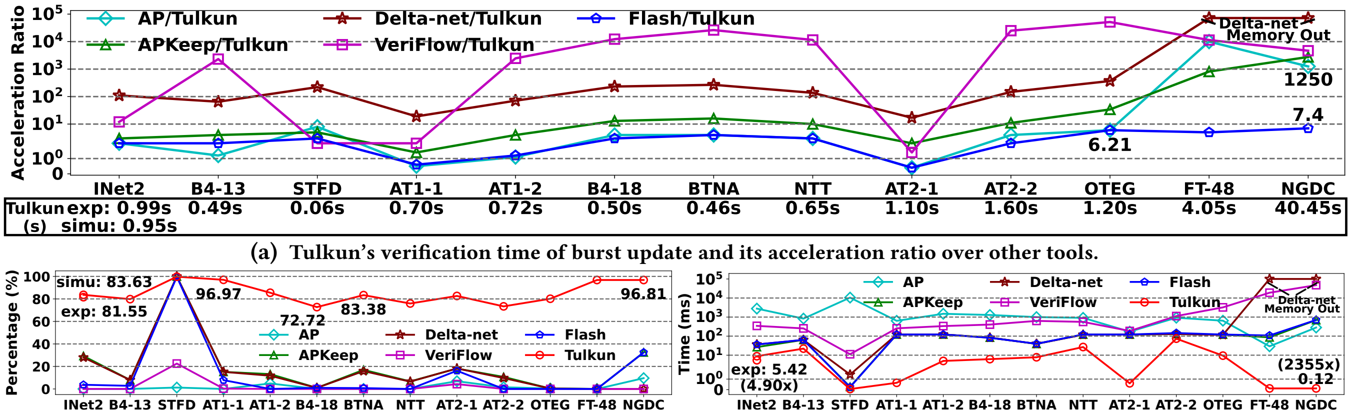
We implement an event-driven simulator to evaluate Tulkun in various networks on a server with 2 Xeon 4210R CPUs.

8.3.1 Simulation Setup. We first introduce the settings.

Datasets. We use 13 datasets in Figure 9. Four are public ones and the others are synthesized with public topologies [53–56]. FT-48 is a 48-ary fattree [57]. NGDC is a real, Clos-based DC. For WAN, we assign link latencies based on topologies [52]. 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 [12], APKeep [20], Delta-net [19], Veriflow [18] and Flash [6]. We also compare Tulkun with APT [58] and Katra [7], two DPV tools designed to support packet transformation, in Appendix G. We reproduce Katra, and use the open-sourced version of AP, Veriflow, Flash, Delta-net, APKeep and APT.

Invariants. We verify the all-pair loop-free, blackhole-free, ($\leq$ **shortest** + 2)-hop reachability in §8.2 with 3-link-failure



(a) Tulkun's verification time of burst update and its acceleration ratio over other tools.

(b) Percentage of incremental verification time < 10ms.

(c) 80% quantile of incremental verification time.

Figure 10: The verification time of Tulkun and other tools in LAN/WAN/DC datasets.

for WAN/LAN and the 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 Appendix I.

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 (Appendix F) and the latency of Tulkun planner to compute *DVNet* with different k-link-failures. Figure 12 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 *DVNet* in <95s (<1440s).

8.3.2 *Results: Burst Update.* Figure 10a gives the verification time of Tulkun, and its acceleration ratio over other tools. For WAN/LAN, Tulkun completes the verification in $\leq 1.60s$ 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 (Delta-net reports memory-out error after 5 hours). Even compared with Flash (297.26s), a recent tool designed specifically to verify such large-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 Appendix D.

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 (3.39× and 11.97×). 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 (Appendix E). As such, with more rules, Tulkun becomes faster than AP and Flash.

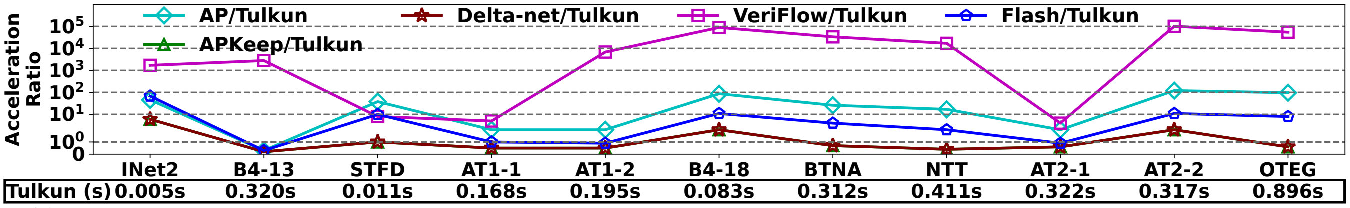
8.3.3 *Results: Incremental Update.* We evaluate Tulkun for incremental verification using the same methodology as in §8.2. The 80% quantile verification time of Tulkun is up to 2355× faster than the fastest centralized DPV (Figure 10c). Among all datasets, Tulkun finishes verifying at least 72.72% rule updates in less than 10ms, while this lower bound of other tools is less than 1% (Figure 10b). It is for the same reason as in experiments (§8.2), and proves that Tulkun enables scalable DPV under various networks and DPV scenarios.

8.3.4 *Results: Fault-Tolerance.* For each LAN/WAN, we generate 50 fault scenes of $\leq$ 3 link failures based on the statistic of Microsoft's WAN [59]. For each scene, we measure the verification time of recounting along *DVNet* with failure flooding (Figure 11a); and generate 1K random rule updates after that to measure the incremental verification time (Figure 11b, 11c). Tulkun consistently outperforms others as in §8.3.3 and §8.3.2. It shows that by computing a fault-tolerant *DVNet* and online recounting, Tulkun efficiently verifies fault-tolerant invariants without involving the planner.

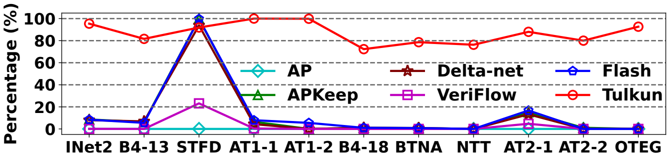
8.4 On-Device Microbenchmarks

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.

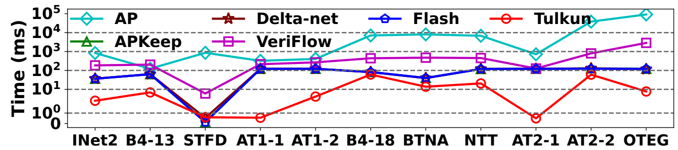
Initialization overhead. For each of 414 devices from WAN / LAN and 6 devices from NGDC/Fattree (one edge, aggregation and core switch, respectively), we measure the overhead of its initialization phase in burst update (*i.e.*, computing the initial LEC and CIB), in terms of total time, maximal memory and CPU load, on all four switch models. The CPU load is computed as $CPU\ time\ /(total\ time \times number\ of\ cores)$. Figure 13 plots their CDFs. On all four switches, all devices in the datasets complete initialization in $\leq 1.75s$, with a CPU load ≤ 0.48, and a maximal memory $\leq 19.6MB$.



(a) The average time of Tulkun to verify the complete network with the updated topology of 50 fault scenes and its acceleration ratio.



(b) Percentage of incremental verification time < 10ms in fault scenes.



(c) 80% quantile of incremental verification time in fault scenes.

Figure 11: The verification time of Tulkun under fault scenes.

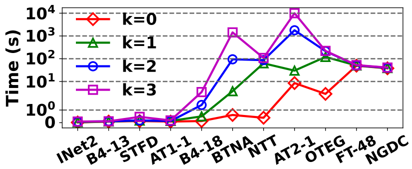


Figure 12: *DVNet* computation latency.

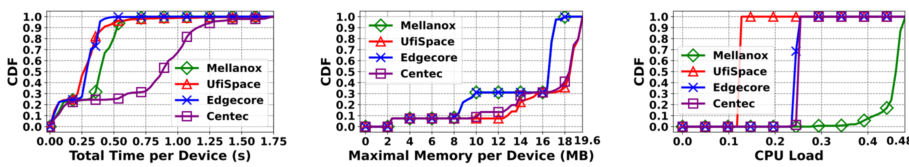


Figure 13: Initialization overhead.

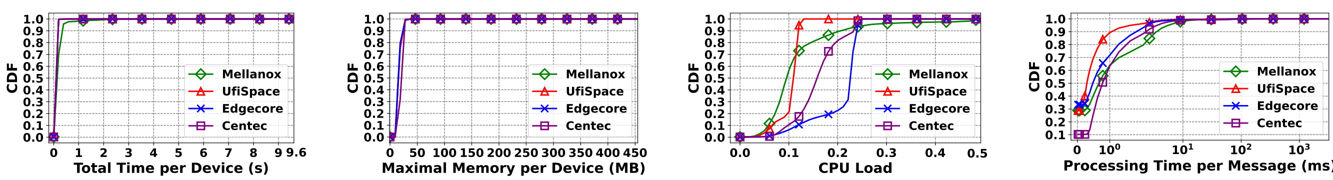


Figure 14: DVM UPDATE message processing overhead.

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 14). 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.

9 RELATED WORK

Network verification includes CPV that checks errors in configurations [37, 60–76]; and DPV that checks errors in the data plane. Tulkun is a DPV tool, and can help simulation-based CPV [37, 77, 78] verify the simulated DP.

Centralized DPV. Existing DPV tools [1–6, 8–16, 18–21] use a centralized verifier to collect and analyze the data planes. Despite substantial 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. Libra [5], RCDC [21] and Flash [6] focus on scale up DPV using parallelization and batch processing. However, they are still centralized designs with the limitations above.

A recent paper [32] proposed the idea of distribuetd 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 [79–83] and programmable DP (*e.g.*, P4 [84]) [85, 86]. Extending Tulkun to stateful and programmable DP is an interesting future work.

Network synthesis. Synthesis [39–41, 87, 88] is complementary to verification. Tulkun is inspired by some of them [39–41] to use automata theory to generate *DVNet*.

Predicate representation. Tulkun chooses BDD [43] to represent packets for its efficiency. Recent data structures (*e.g.*, ddNF [89] and #PEC [90]) may benefit Tulkun.

10 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.

Ethics: This work does not raise any ethical issues.

REFERENCES

[1] Geoffrey G Xie, Jibin Zhan, David A Maltz, Hui Zhang, Albert Greenberg, Gisli Hjalmtysson, and Jennifer 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.

[2] Ehab Al-Shaer and Saeed 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.

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

[4] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *NSDI*, volume 12, pages 113–126, 2012.

[5] Hongyi Zeng, Shidong Zhang, Fei Ye, Vimalkumar Jeyakumar, Mickey Ju, Junda Liu, Nick McKeown, and Amin Vahdat. Libra: Divide and conquer to verify forwarding tables in huge networks. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 87–99, 2014.

[6] Dong Guo, Shenshen Chen, Kai Gao, Qiao Xiang, Ying Zhang, and Y. Richard Yang. Flash: Fast, consistent data plane verification for large-scale network settings. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 314–335, New York, NY, USA, 2022. Association for Computing Machinery.

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

[8] Peng Zhang, Aaron Gember-Jacobson, Yueshang Zuo, Yuhao Huang, Xu Liu, and Hao Li. Differential network analysis. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 601–615, 2022.

[9] Peyman Kazemian, Michael Chan, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. Real time network policy checking using header space analysis. In *NSDI*, pages 99–111, 2013.

[10] Nuno P Lopes, Nikolaj Bjørner, Patrice Godefroid, Karthick Jayaraman, and George Varghese. Checking beliefs in dynamic networks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 499–512, 2015.

[11] Gordon D Plotkin, Nikolaj Bjørner, Nuno P Lopes, Andrey Rybalchenko, and George Varghese. Scaling network verification using symmetry and surgery. *ACM SIGPLAN Notices*, 51(1):69–83, 2016.

[12] Hongkun Yang and Simon 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.

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

[14] Huazhe Wang, Chen Qian, Ye Yu, Hongkun Yang, and Simon 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.

[15] Huazhe Wang, Chen Qian, Ye Yu, Hongkun Yang, and Simon S Lam. Practical network-wide packet behavior identification by ap classifier. *IEEE/ACM Transactions on Networking*, 25(5):2886–2899, 2017.

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

[17] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. Netkat: Semantic foundations for networks. *Acm sigplan notices*, 49(1):113–126, 2014.

[18] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P Brighten 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.

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

[20] Peng Zhang, Xu Liu, Hongkun Yang, Ning Kang, Zhengchang Gu, and Hao Li. Apkeep: Realtime verification for real networks. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 241–255, 2020.

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

[22] 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.

[23] Dong Guo. Flash Artifact for SIGCOMM22. https://github.com/snlab/flash, 2022.

[24] Karthik Lakshminarayanan, Matthew Caesar, Murali Rangan, Tom Anderson, Scott Shenker, and Ion Stoica. Achieving convergence-free routing using failure-carrying packets. *ACM SIGCOMM computer communication review*, 37(4):241–252, 2007.

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

[26] Aaron Gember-Jacobson, Costin Raiciu, and Laurent 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.

[27] Stefano Vissicchio, Luca Cittadini, Olivier Bonaventure, Geoffrey G Xie, and Laurent 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.

[28] Sean Choi, Boris Burkov, Alex Eckert, Tian Fang, Saman Kazemkhani, Rob Sherwood, Ying Zhang, and Hongyi Zeng. Fboss: Building switch software at scale. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 342–356, 2018.

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

[30] Hongkun Yang and Simon 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.

[31] Amogh Dhamdhere, David D Clark, Alexander Gamero-Garrido, Matthew Luckie, Ricky KP Mok, Gautam Akiwate, Kabir Gogia, Vaibhav Bajpai, Alex C Snoeren, and Kc Claffy. Inferring persistent interdomain congestion. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 1–15, 2018.

[32] Qiao Xiang, Ridi Wen, Chenyang Huang, Yuxin Wang, and Franck Le. Network can check itself: scaling data plane checking via distributed,

on-device verification. In *Proceedings of the 21st ACM Workshop on Hot Topics in Networks*, pages 85–92, 2022.
[33] Gary S. Malkin. RIP Version 2. RFC 2453, November 1998.
[34] Yakov Rekhter, Susan Hares, and Dr. Tony Li. A Border Gateway Protocol 4 (BGP-4). RFC 4271, January 2006.
[35] Anonymous Authors. Tulkun system functionality demonstration. http://distributeddpvdemo.tech/, 2022.
[36] Open Networking Foundation. Openflow switch specification 1.5.1. Open Networking Foundation (on-line), March 2015.
[37] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. A general approach to network configuration analysis. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 469–483, 2015.
[38] Harry R Lewis and Christos H Papadimitriou. Elements of the theory of computation. *ACM SIGACT News*, 29(3):62–78, 1998.
[39] 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.
[40] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. Don't mind the gap: Bridging network-wide objectives and device-level configurations. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 328–341, 2016.
[41] Kuo-Feng Hsu, Ryan Beckett, Ang Chen, Jennifer Rexford, Praveen Tammana, and David Walker. Contra: A programmable system for performance-aware routing. *to apper at NSDI'20*, 2020.
[42] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P Brighten Godfrey, and Samuel Talmadge King. Debugging the data plane with anteater. *ACM SIGCOMM Computer Communication Review*, 41(4):290–301, 2011.
[43] Randal E Bryant. Graph-based algorithms for boolean function manipulation. *Computers, IEEE Transactions on*, 100(8):677–691, 1986.
[44] Network Working Group. OSPF version 2. https://www.rfc-editor.org/rfc/rfc2328.html, 1998.
[45] Facebook Group. Facebook Open/R. https://github.com/facebook/openr, 2016.
[46] Mellanox SN2700 Switch. https://www.mellanox.com/related-docs/prod_eth_switches/PB_SN2700.pdf, 2015.
[47] Edgecore Wedge32-100X Switch. https://www.edge-core.com/productsInfo.php?cls=1&cls2=5&cls3=181&id=335, 2021.
[48] UfiSpace S9180-32X S9180-32X Switch. https://www.ufispace.com/uploads/able/files/productfilemanager/000045467d1fc648d792c404372956a0.pdf, 2019.
[49] Microsoft. Sonic: The networking switch software that powers the microsoft global cloud. https://azure.github.io/SONiC/, 2021.
[50] Open Compute Project. Open network linux. http://opennetlinux.org/, 2021.
[51] The Internet2 Observatory. The internet2 dataset. http://www.internet2.edu/research-solutions/research-support/observatory, 2021.
[52] WonderNetwork. Global ping statistics. https://wondernetwork.com/pings, 2021.
[53] Simon Knight, Hung X Nguyen, Nickolas Falkner, Rhys Bowden, and Matthew Roughan. The internet topology zoo. *IEEE Journal on Selected Areas in Communications*, 29(9):1765–1775, 2011.
[54] Neil Spring, Ratul Mahajan, and David Wetherall. Measuring isp topologies with rocketfuel. *ACM SIGCOMM Computer Communication Review*, 32(4):133–145, 2002.
[55] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, et al. B4: Experience with a globally-deployed software defined wan. *ACM SIGCOMM Computer Communication Review*, 43(4):3–14, 2013.
[56] Chi-Yao Hong, Subhasree Mandal, Mohammad Al-Fares, Min Zhu, Richard Alimi, Chandan Bhagat, Sourabh Jain, Jay Kaimal, Shiyu Liang, Kirill Mendelev, et al. B4 and after: Managing hierarchy, partitioning, and asymmetry for availability and scale in google's software-defined wan. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 74–87, 2018.
[57] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. *ACM SIGCOMM computer communication review*, 38(4):63–74, 2008.
[58] Hongkun Yang and Simon S Lam. Scalable verification of networks with packet transformers using atomic predicates. *IEEE/ACM Transactions on Networking*, 25(5):2900–2915, 2017.
[59] Jiaqi Zheng, Hong Xu, Xiaojun Zhu, Guihai Chen, and Yanhui Geng. We've got you covered: Failure recovery with backup tunnels in traffic engineering. In *2016 IEEE 24th International Conference on Network Protocols (ICNP)*, pages 1–10. IEEE, 2016.
[60] Konstantin Weitz, Doug Woos, Emina Torlak, Michael D Ernst, Arvind Krishnamurthy, and Zachary Tatlock. Scalable verification of border gateway protocol configurations with an smt solver. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 765–780, 2016.
[61] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. A general approach to network configuration verification. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 155–168. ACM, 2017.
[62] Fangdan Ye, Da Yu, Ennan Zhai, Hongqiang Harry Liu, Bingchuan Tian, Qiaobo Ye, Chunsheng Wang, Xin Wu, Tianchen Guo, Cheng Jin, et al. Accuracy, scalability, coverage–a practical configuration verifier on a global wan. In *to appear in SIGCOMM'20*, 2020.
[63] Seyed K Fayaz, Tushar Sharma, Ari Fogel, Ratul Mahajan, Todd Millstein, Vyas Sekar, and George Varghese. Efficient network reachability analysis using a succinct control plane representation. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 217–232, 2016.
[64] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. Control plane compression. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 476–489, 2018.
[65] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. Abstract interpretation of distributed network control planes. pages 1–27. ACM New York, NY, USA, 2019.
[66] Aaron Gember-Jacobson, Raajay Viswanathan, Aditya Akella, and Ratul Mahajan. Fast control plane analysis using an abstract representation. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 300–313, 2016.
[67] Aaron Gember-Jacobson, Aditya Akella, Ratul Mahajan, and Hongqiang Harry Liu. Automatically repairing network control planes using an abstract representation. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 359–373, 2017.
[68] Anubhavnidhi Abhashkumar, Aaron Gember-Jacobson, and Aditya Akella. Tiramisu: Fast multilayer network verification. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 201–219, 2020.
[69] Santhosh Prabhu, Kuan Yen Chou, Ali Kheradmand, Brighten Godfrey, and Matthew Caesar. Plankton: Scalable network configuration verification through model checking. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 953–967, 2020.
[70] Samuel Steffen, Timon Gehr, Petar Tsankov, Laurent Vanbever, and Martin Vechev. Probabilistic verification of network configurations. In *to appear in SIGCOMM'20*.

[71] Nick Giannarakis, Devon Loehr, Ryan Beckett, and David Walker. Nv: An intermediate language for verification of network control planes. In *PLDI*, pages 958–973, 2020.

[72] Mingchen Zhao, Wenchao Zhou, Alexander JT Gurney, Andreas Haeberlen, Micah Sherr, and Boon Thau Loo. Private and verifiable interdomain routing decisions. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 383–394, 2012.

[73] Siva Kesava Reddy Kakarla, Ryan Beckett, Behnaz Arzani, Todd Millstein, and George Varghese. Groot: Proactive verification of dns configurations. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 310–328, 2020.

[74] Nick Feamster and Hari Balakrishnan. Detecting bgp configuration faults with static analysis. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pages 43–56, 2005.

[75] Anduo Wang, Limin Jia, Wenchao Zhou, Yiqing Ren, Boon Thau Loo, Jennifer Rexford, Vivek Nigam, Andre Scedrov, and Carolyn Talcott. Fsr: Formal analysis and implementation toolkit for safe interdomain routing. *IEEE/ACM Transactions on Networking*, 20(6):1814–1827, 2012.

[76] Thomas Ball, Nikolaj Bjørner, Aaron Gember, Shachar Itzhaky, Aleksandr Karbyshev, Mooly Sagiv, Michael Schapira, and Asaf Valadarsky. Vericon: Towards verifying controller programs in software-defined networks. In *Proceedings of the 35th ACM SIGPLAN conference on programming language design and implementation*, pages 282–293, 2014.

[77] Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Jiaxin Cao, Sri Tallapragada, Nuno P Lopes, Andrey Rybalchenko, Guohan Lu, and Lihua Yuan. Crystalnet: Faithfully emulating large production networks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 599–613, 2017.

[78] Nuno P Lopes and Andrey Rybalchenko. Fast bgp simulation of large datacenters. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 386–408. Springer, 2019.

[79] Aurojit Panda, Ori Lahav, Katerina Argyraki, Mooly Sagiv, and Scott Shenker. Verifying reachability in networks with mutable datapaths. In *14th USENIX symposium on networked systems design and implementation (NSDI)*, pages 699–718, 2017.

[80] Kaiyuan Zhang, Danyang Zhuo, Aditya Akella, Arvind Krishnamurthy, and Xi Wang. Automated verification of customizable middlebox properties with gravel. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 221–239, 2020.

[81] Arseniy Zaostrovnykh, Solal Pirelli, Luis Pedrosa, Katerina Argyraki, and George Candea. A formally verified nat. In *Proceedings of the conference of the acm special interest group on data communication*, pages 141–154, 2017.

[82] Yifei Yuan, Soo-Jin Moon, Sahil Uppal, Limin Jia, and Vyas Sekar. Netsmc: A custom symbolic model checker for stateful network verification. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 181–200, 2020.

[83] Shenghui Chen, Zhiming Fan, Haiying Shen, and Lu Feng. Performance modeling and verification of load balancing in cloud systems using formal methods. In *2019 IEEE 16th International Conference on Mobile Ad Hoc and Sensor Systems Workshops (MASSW)*, pages 146–151. IEEE, 2019.

[84] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[85] Jed Liu, William Hallahan, Cole Schlesinger, Milad Sharif, Jeongkeun Lee, Robert Soulé, Han Wang, Călin Caşcaval, Nick McKeown, and Nate Foster. P4v: Practical verification for programmable data planes. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on data communication*, pages 490–503, 2018.

[86] Dragos Dumitrescu, Radu Stoenescu, Lorina Negreanu, and Costin Raiciu. Bf4: Towards bug-free p4 programs. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 571–585, 2020.

[87] Bingchuan Tian, Xinyi Zhang, Ennan Zhai, Hongqiang Harry Liu, Qiaobo Ye, Chunsheng Wang, Xin Wu, Zhiming Ji, Yihong Sang, Ming Zhang, et al. Safely and automatically updating in-network acl configurations with intent language. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 214–226. 2019.

[88] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin Vechev. Netcomplete: Practical network-wide configuration synthesis with autocompletion. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 579–594, 2018.

[89] Nikolaj Bjørner, Garvit Juniwal, Ratul Mahajan, Sanjit A Seshia, and George Varghese. Ddnf: An efficient data structure for header spaces. In *Haifa Verification Conference*, pages 49–64. Springer, 2016.

[90] Alex Horn, Ali Kheradmand, and Mukul R Prasad. A precise and expressive lattice-theoretical framework for efficient network verification. In *2019 IEEE 27th International Conference on Network Protocols (ICNP)*, pages 1–12. IEEE, 2019.

[91] A. Vahidi. A bdd and z-bdd library written in java. https://bitbucket.org/vahidi/jdd, 2020.

[92] Google. A language-neutral, platform-neutral extensible mechanism for serializing structured data. https://developers.google.com/protocol-buffers, 2020.

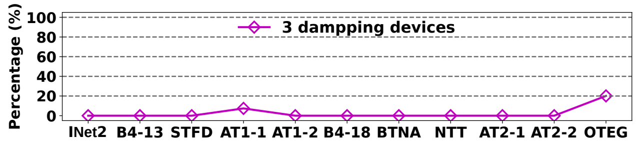


Figure 15: The 80% quantile of the percentage of errors detected by Flash when only three switches cannot reach the verifier.

Algorithm 1: COUNT(*DVNet*, p).

foreach u_i, $i = 1, \ldots, n$ *in reverse topological order* **do**
  if u_i *is a destination* **then**
    $\mathbf{c}_i \leftarrow 1$
  else
    foreach $v_j \in N_d(u_i)$ **do**
      if $v_j.dev \in u_i.dev.fwd(p)$ **then**
        $b_{ij} \leftarrow 1$;
    if $u_i.dev.fwd(p).type == ALL$ **then**
      Update $\mathbf{c}_u$ with Equation (1);
    else
      Update $\mathbf{c}_u$ with Equation (2);
return $\mathbf{c}_n$;

A QUANTIFYING THE NEED OF CENTRALIZED DPV ON A RELIABLE MANAGEMENT NETWORK

We test Flash's early detection mechanism using its open-source prototype [23]. We use the datasets of 1 LAN and 10 WAN. We randomly modify the rules of 5 switches in each network to create errors. We simulate the scenario of an unreliable management network by randomly selecting only 3 switches in each network as the ones the verifier cannot reach. We repeat the simulation 50 times. With Flash's early detection mechanism, in 9 out of the 11 datasets, Flash can detect zero errors in 80% quantile of all experiments. This test quantitatively measures the importance of a reliable management network for centralized DPV.

B PROOFS OF *DVNET* BACKWARD COUNTING

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

For presentation purposes, we first summarize the backward counting algorithm in *DVNet* in Algorithm 1. Given a packet p and a *DVNet*, 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 *DVNet* along paths in the *DVNet* in each universe. Suppose Algorithm 1 is incorrect. There could be three cases: (1) there exists a path in *DVNet* that is provided by the network data plane, but is not counted by Algorithm 1; (2) There exists a path in *DVNet* that is not provided by the network data plane, but is counted by Algorithm 1; (3) Algorithm 1 counts a path out of *DVNet*. None of these cases could happen because at each node u, Equations (1) (2) only counts $\mathbf{c}_{v_j}$ of v_j with $b_{ij} = 1$, *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.

B.2 Proof of Proposition 1

Consider $\mathbf{c}_u$ 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 $\otimes$, 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 *DVNet* is greater than or equal to the number of copies of p that can be sent from u to the destination in *DVNet*. As such,

- When $count\_exp$ is $\geq N$ or $> N$, each u only sends $min(\mathbf{c}_u)$ to its upstream neighbors. With such information, in the end, the source node of *DVNet* can compute the lower bound of the number of copies of p delivered in all universes. If this lower bound satisfies $count\_exp$, then all universes satisfy it. If this lower bound does not satisfy $count\_exp$, a network error is found.
- When $count\_exp$ is $\leq N$ or $< N$, each u only sends $max(\mathbf{c}_u)$ to its upstream neighbors. The analysis is similar, with the source node computing the upper bound.
- When $count\_exp$ is $== N$, if $\mathbf{c}_u$ 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 $\mathbf{c}_u$ to let them know that. If $\mathbf{c}_u$ 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 $min(|\mathbf{c}_u|, 2)$ smallest elements in $\mathbf{c}_u$ to its upstream neighbors.

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

C IMPLEMENTATION

Our Tulkun prototype has ~9K lines of Java code, including a verification planner and on-device verifiers. Figure 16 shows the implementation structure. The planner computes the *DVNet* based on the invariant and the topology, and decides the counting task of on-device verifiers.

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

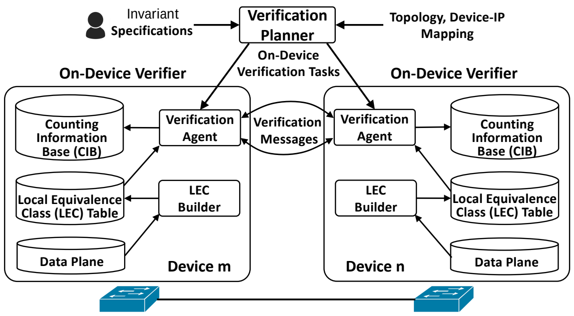


Figure 16: The implementation of Tulkun.

thread is assigned for a node in a *DVNet.* 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 [91] library to support the serialization and deserialization between BDD and the Protobuf data encoding [92], so that BDDs can be efficiently transmitted between devices in UPDATE messages.

D TABLE OF BURST UPDATE VERIFICATION TIME

As a supplementary to the statistics in Figure 10a , INet2 [51], STFD [4], AT1-1 [19], AT2-1 [19]. Figure 17 shows the complete results of the burst update verification time of Tulkun and all centralized DPV tools in comparison for each dataset, as well as the acceleration ratio of Tulkun. Results show that Tulkun consistently achieves fast data plane verification in networks with various types and scales.

E VERIFICATION TIME BREAKDOWN OF BURST UPDATE

This section studies why Tulkun is slower than AP in burst update verification for AT1-1 and AT2-1, but faster for AT1-2 and AT2-2. Figure 19 provides a verification time breakdown of burst update of different DVP tools for these four datasets. AT1-1 and AT1-2 have the same topology but different numbers of rules. So are AT2-1 and AT2-2. In particular, centralized DPV tools work in three phases: (1) FIB collection that collects data planes of all devices to the server, (2) model computation that takes as input the data planes of all devices and computes the equivalence classes, and (3) verification computation that takes as input the computed ECs to verify invariants. We observe that model computation is the bottleneck of all four centralized DPV tools (except for VeriFlow, whose bottleneck is both model computation and verification computation). The model computation time is proportional to the total number of rules in the data plane. As such, with the number of rules increasing from AT1-1/AT2-1 to AT1-2/AT2-2 (3.39× and 11.97×), the performance of centralized DPV tools degrades approximately with the same ratio.

In contrast, to verify burst update, Tulkun operates in two phases: (1) LEC initialization at devices and (2) counting and propagation among devices. Because LEC initialization is performed by each device in parallel, its time is only proportional to the number of rules at each device. As such, it is not the performance bottleneck of Tulkun, and has only a small impact on the total verification time of Tulkun when the total number of rules increases (*e.g.*, from AT1-1/AT2-1 to AT1-2/AT2-2). As such, with more rules, Tulkun becomes faster than AP. As a result, we conclude that Tulkun achieves better scalability than centralized DVP tools such as AP when the total number of rules increases.

F MESSAGING OVERHEAD OF TULKUN IN INCREMENTAL VERIFICATION

For all datasets in our experiments and simulations, we first plot CDFs of the number of DVM UPDATE messages sent in the network per rule update (Figure 20a) and the number of devices whose counting results change per rule update (Figure 20b).

Figure 20a shows that for each dataset, at least 70% of rule updates do not incur any DVM UPDATE message in Tulkun. Figure 20b further shows that for at least 75% of rule updates, the number of devices whose counting results change is no more than two. This shows that by decomposing verification to on-device counting tasks, a large portion of incremental verifications become local verification on a single network device, or only require sharing counting results among a small number of network devices. As such, the Tulkun achieves substantial scaling up on incremental verification.

We next plot the size of DVM UPDATE message incurred across 10,000 rule updates (Figure 20c). We observe that all UPDATE messages are smaller than 150KB, in particular, for NGDC and FT-48, their UPDATE messages are all smaller than 396 bytes. This indicates that the bandwidth overhead of Tulkun is very low.

In the end, we plot the number of CIB entries of each device after 10,000 rule updates (Figure 20d), in supplementary to the maximal memory microbenchmark results in Figure 14, which shows that the Tulkun on-device verifiers only consume a small amount of memory on commodity network switches.

Network	AP	APKeep	Delta-net	VeriFlow	Flash	Tulkun
INet2	2.07 (2.2×)	2.88 (3.0×)	104.71 (148.1×)	11.20 (11.8×)	2.19 (2.3×)	**exp: 0.99** **simu: 0.95**
B4-13	0.58 (1.2×)	1.84 (3.8×)	31.86 (65.0×)	1,134 (2314×)	0.80 (1.6×)	**0.49**
STFD	0.49 (8.2×)	0.29 (4.8×)	13.28 (221.3×)	0.12 (2.0×)	0.18 (3.0×)	**0.06**
AT1-1	**0.35** (0.5×)	1.00 (1.4×)	13.57 (19.3×)	1.14 (1.6×)	0.41 (0.6×)	0.70
AT1-2	0.82 (1.1×)	2.56 (3.6×)	50.96 (70.8×)	1,771 (2460×)	0.84 (1.2×)	**0.72**
B4-18	1.76 (3.5×)	6.30 (12.6×)	115.75 (231.5×)	6,139 (12278×)	1.63 (3.3×)	**0.50**
BTNA	2.05 (4.5×)	7.25 (15.8×)	123.17 (267.8×)	12,086 (26274×)	1.94 (4.2×)	**0.46**
NTT	1.65 (2.5×)	6.53 (10.1×)	89.05 (137.0×)	7,472 (11496×)	1.74 (2.7×)	**0.65**
AT2-1	0.49 (0.5×)	1.95 (1.8×)	19.08 (17.4×)	1.55 (1.41×)	**0.45** (0.4×)	1.10
AT2-2	5.93 (3.7×)	18.23 (11.4×)	237.16 (148.2×)	39,980 (24988×)	3.88 (2.4×)	**1.60**
OTEG	7.62 (6.4×)	41.18 (34.3×)	442.48 (368.7×)	62,028 (51690×)	7.45 (6.2×)	**1.20**
FT-48	40,609 (10027×)	3,293 (813×)	Memory Out	46,103 (11384×)	20.78 (5.1×)	**4.05**
NGDC	50,574 (1250×)	111,558 (2758×)	Memory Out	188,692 (4665×)	297.26 (7.4×)	**40.45**

Figure 17: Verification time of burst update (seconds).

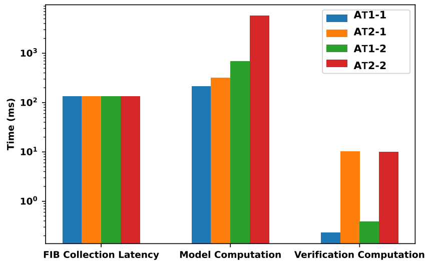
Network	Percentage < 10 ms						80% quantile (ms)					
	AP	APKeep	Delta-net	VeriFlow	Flash	Tulkun	AP	APKeep	Delta-net	VeriFlow	Flash	Tulkun
INet2	0.00%	29.29%	27.97%	0.03%	3.72%	**exp: 81.55% simu: 83.63%**	2782.10	26.55	36.55	345.62	36.55	**5.42 (4.90×)** **8.83 (3.01×)**
B4-13	0.04%	7.96%	7.96%	0.00%	2.77%	**79.80%**	831.54	63.65	63.23	254.35	63.97	**22.17 (2.85×)**
STFD	1.23%	99.93%	98.73%	22.55%	99.74%	**99.76%**	10522.66	0.27	1.43	11.22	0.24	**0.06 (4.0×)**
AT1-1	0.00%	15.19%	15.19%	0.30%	7.83%	**96.97%**	635.30	121.74	121.77	256.93	121.48	**0.65 (186.89×**
AT1-2	4.89%	13.20%	11.76%	0.00%	0.00%	**85.50%**	1488.84	122.25	122.44	336.88	122.09	**4.99 (24.47×)**
B4-18	0.00%	0.96%	0.96%	0.00%	0.96%	**72.72%**	1287.57	83.77	83.13	404.34	81.81	**6.21 (13.39×)**
BTNA	0.00%	16.98%	15.87%	0.00%	0.75%	**83.38%**	1000.06	39.17	39.28	631.19	39.78	**7.78 (5.03×)**
NTT	0.00%	6.73%	6.49%	0.00%	0.00%	**75.94%**	911.03	120.79	118.67	559.44	122.08	**26.88 (4.41×)**
AT2-1	6.88%	18.12%	18.23%	4.29%	16.19%	**82.64%**	171.85	121.55	121.56	186.38	121.57	**0.60 (202.58×**
AT2-2	1.49%	10.73%	9.69%	0.00%	0.00%	**73.38%**	924.10	127.78	129.81	1,127.78	145.67	**73.64 (1.74×)**
OTEG	0.39%	0.38%	0.39%	0.00%	0.00%	**80.16%**	645.92	120.92	116.18	3,228.21	121.47	**9.48 (12.26×)**
FT-48	0.00%	0.00%	Memory Out	0.00%	0.00%	**96.81%**	29.21	86.17	Memory Out	18,962.00	108.29	**0.12 (243.42×)**
NGDC	9.60%	32.43%	Memory Out	0.00%	32.43%	**96.81%**	282.56	693.63	Memory Out	47,385.64	708.22	**0.12 (2354.67×)**

Figure 18: Verification time of incremental update (seconds).

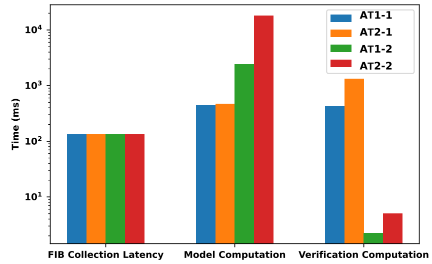
G VERIFYING NETWORKS WITH PACKET TRANSFORMATION RULES

We next compare Tulkun's performance with APT [58] and Katra [7], two state-of-the-art centralized DPV tools designed to verify data plane with packet transformation rules. Specifically, we use three datasets: INet2, BTNA, and OTEG because they represent LAN and WAN of different scales, and follow the methodology in [7, 58] to conduct the following experiments.

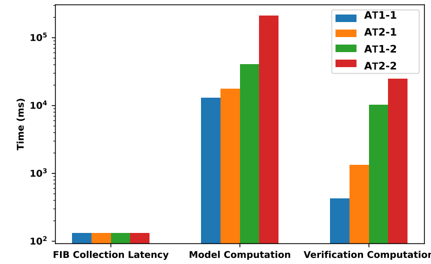
Experiment 1: Burst update. For each dataset, we randomly set up IP-in-IP tunnels between switches. The number of tunnels ranges from 50 to 250. Figure 21 gives the verification time of burst update. We observe that in all experiments, Tulkun substantially outperforms the APT and Katra, *i.e.*, up to 5.34× than APT and up to 16.58× than Katra. In addition, we observe that the verification time of Tulkun is insensitive to the number of tunnels, while the other tools' verification time increases as the number of tunnels does. The reason is the same as why Tulkun outperforms other DPV tools in experiments without packet transformation rules: Tulkun decomposes verification into lightweight on-device tasks to achieve high-level parallelization in computation.



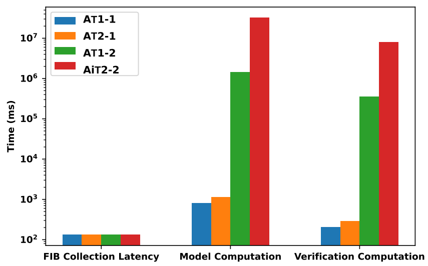
(a) AP.



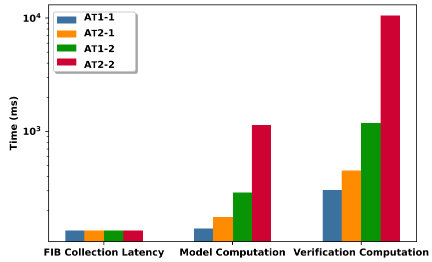
(b) APKeep.



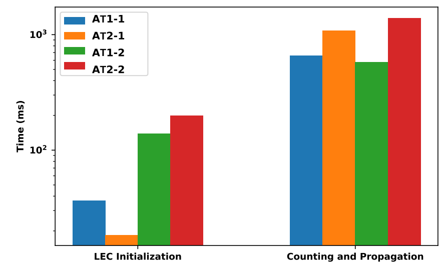
(c) Delta-net.



(d) VeriFlow.

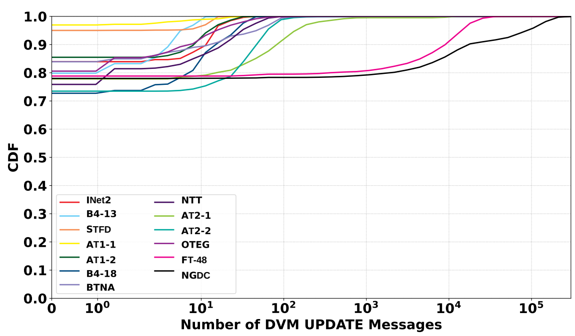


(e) Flash.

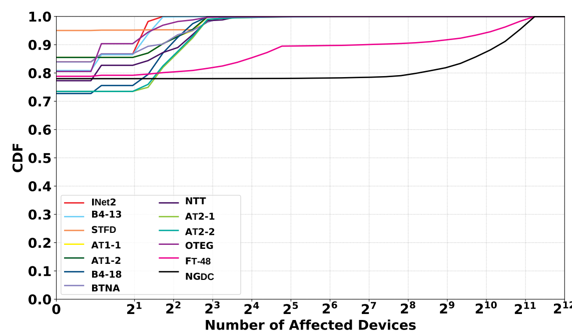


(f) Tulkun.

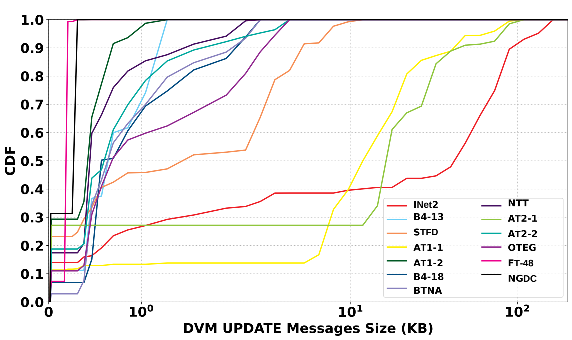
Figure 19: Verification time breakdown of different tools in burst update.



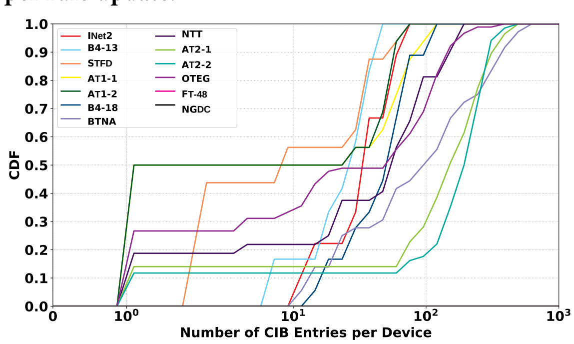
(a) Number of DVM UPDATE messages per rule update.



(b) Number of devices with changed counting results per rule update.



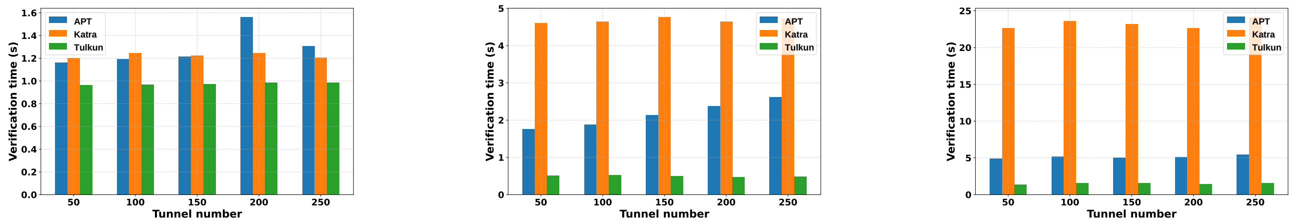
(c) Size of DVM UPDATE message.



(d) Number of CIB entries per device after all rule updates.

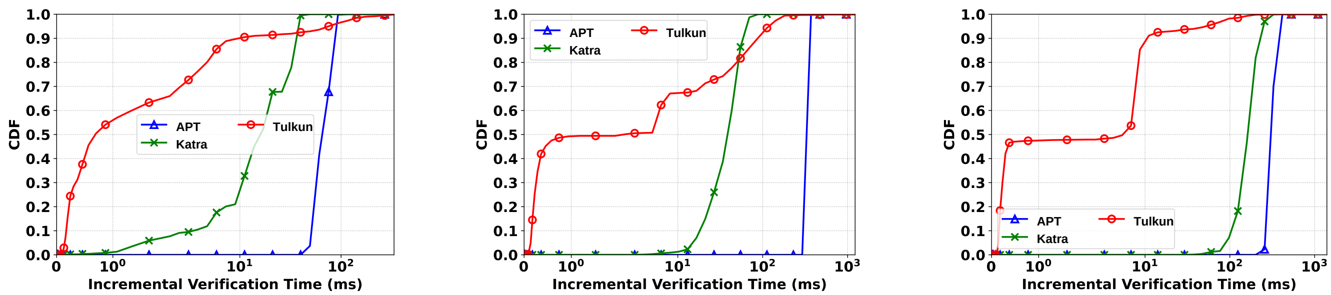
Figure 20: The messaging overhead of Tulkun in incremental verification.

Experiment 2: Incremental update. For each dataset, when the burst update with 250 IP-in-IP tunnels finishes, we randomly generate 1K rule updates and apply them one by one. After each update, we incrementally verify the network. Figure 22 shows the CDF of incremental verification time of three tools in all three datasets. We observe that the



(a) INet2. (b) BTNA. (c) OTEG.

Figure 21: Verification time of burst update with packet transformation



(a) INet2. (b) BTNA. (c) OTEG.

Figure 22: The CDF of incremental verification time with packet transformation

80% quantile of Tulkun is 4.74 / 48.53 / 7.98 milliseconds, while APT and Katra are 84.95 / 340.17 / 333.71 and 36.82 / 52.05 / 197.48 milliseconds, respectively. We also observe the performance of Tulkun is similar to its performance in experiments on the same dataset without packet transformation rules. This is because Tulkun handles packet transformation rules via a Sub-Pub mechanism between neighboring devices while keeping the counting tasks at on-device verifiers lightweight. When a rule update happens, only devices whose counting results may change need to incrementally update their results, and send them to needed neighbors incrementally based on the DVM protocol.

H CDF OF INCREMENTAL UPDATE VERIFICATION TIME

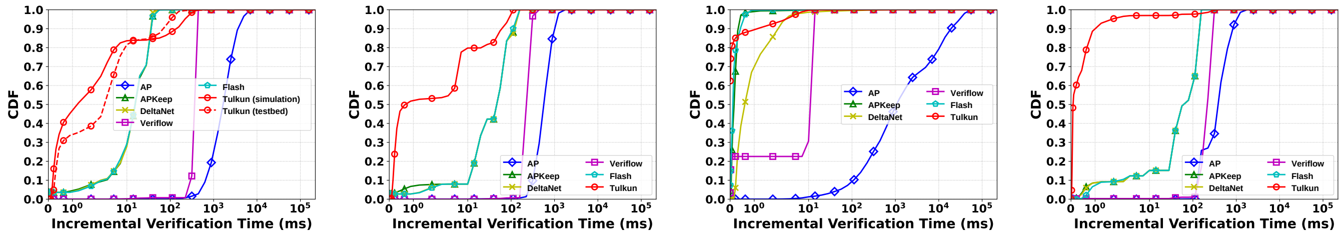
As a supplementary to the statistics in Figure 18, Figure 23 and Figure 24 plot the CDF of the incremental verification time of Tulkun and centralized DPV tools in comparison for each dataset, to show that Tulkun consistently outperforms state-of-the-art centralized DPV tools for incremental update verification in all datasets.

I VERIFYING RCDC LOCAL CONTRACTS USING TULKUN

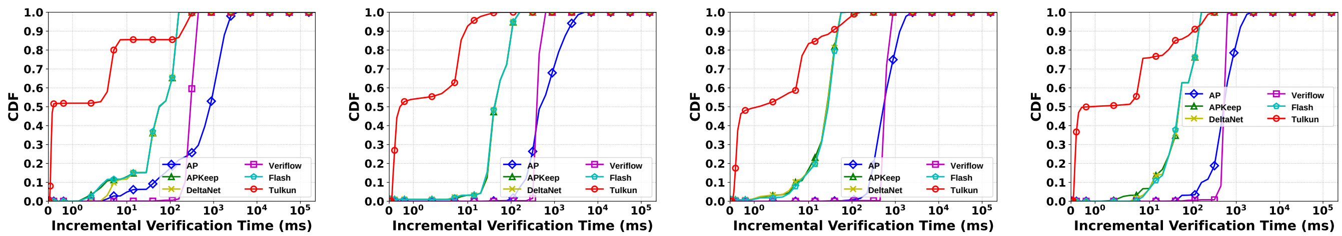
In §4.2, we have proved that the local contracts to verify all-shortest-path availability invariant in Azure RCDC [21] is a special case of the counting tasks in Tulkun. One distinction, however, is that RCDC verifies those local contracts in centralized computation instances. In this experiment, we study the feasibility of letting Tulkun on-device planners verify these local contracts on commodity network devices. Specifically, we pick three devices (one edge, one aggregation and one core) in the FT-48 and the NGDC datasets, respectively, and verify their local contracts on three commodity switches. We plot the results in Figure 25. Results show that all local contracts are verified on commodity switches in less than $320ms$, with a CPU load ≤ 0.47 and a maximal memory $\leq 15.2MB$. The latency is consistent with the result of RCDC running in off-device computation instances (*e.g.*, $O(100)ms$ in Section 2.6.1 of RCDC [21]).

We further go beyond verifying these local contracts from a green start to verifying them incrementally. To this end, for each DC network, we randomly generate 1,000 rule updates across the three devices, and evaluate how fast the Tulkun on-device verifiers on commodity network devices can incrementally verify their counting tasks. Results in Figure 26 show that the 90% quantile of incremental verification time on each switch model is $0.08ms$ in FT-48, and $0.15ms$ in NGDC.

From these results, we demonstrate that Tulkun can efficiently verify the local contracts of RCDC on commodity network switches, with low overhead.

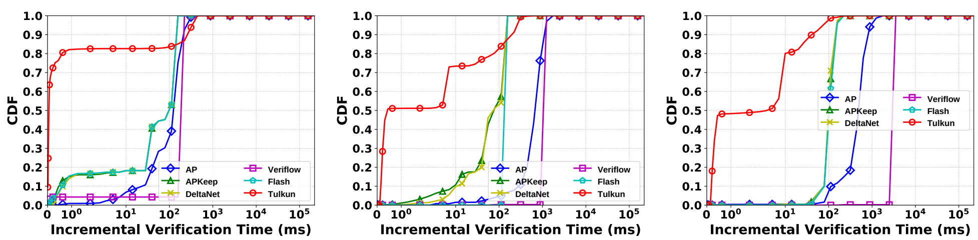


(a) INet2. (b) B4-13. (c) STFD. (d) AT1-1.

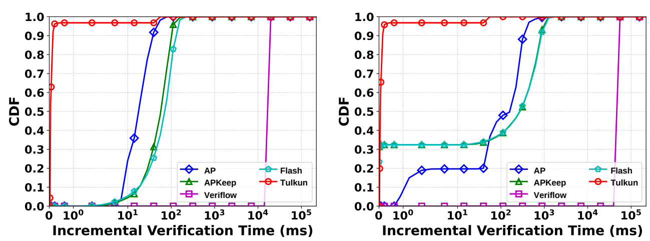


(e) AT1-2. (f) B4-18. (g) BTNA. (h) NTT.

Figure 23: The CDF of incremental verification time of all datasets - part 1.

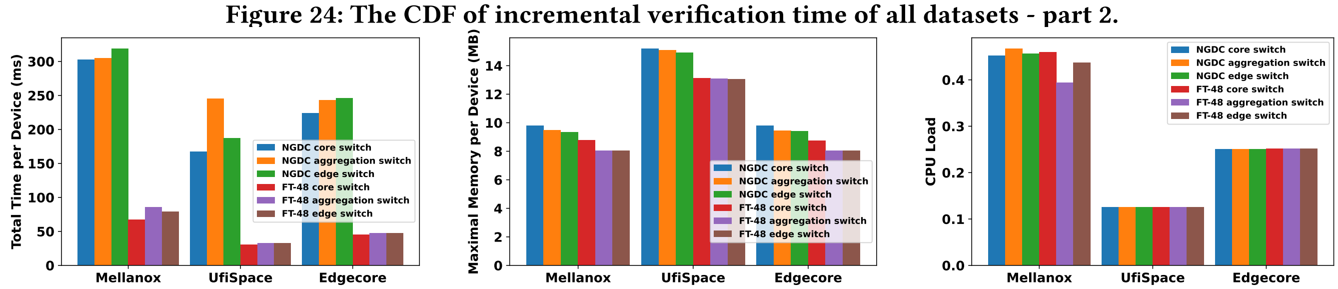


(a) AT2-1. (b) AT2-2. (c) OTEG.



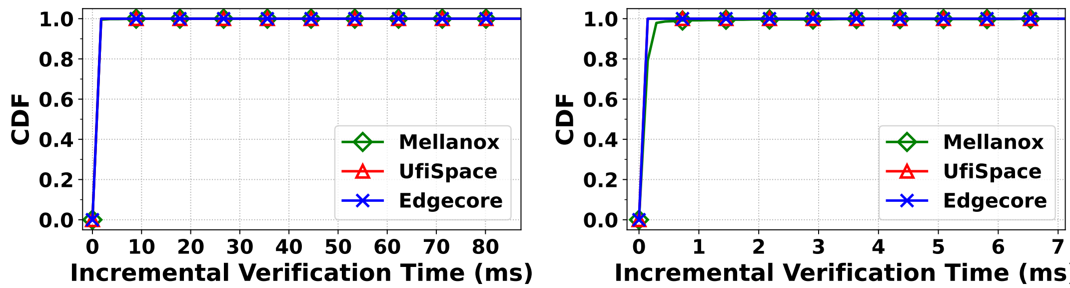
(d) FT-48. (e) NGDC.

Figure 24: The CDF of incremental verification time of all datasets - part 2.



(a) Total time. (b) Maximal memory. (c) CPU load.

Figure 25: Time and overhead of verifying all-shortest-path availability in DC networks from green start on commodity network devices.



(a) FT-48. **(b) NGDC.**

Figure 26: Time of verifying all-shortest-path availability in DC networks incrementally on commodity network devices.