1. Hello, everyone, I am Qiao Xiang from Xiamen University, China. It's my pleasure to present our paper, Beyond a Centralized Verifier: Scaling Data Plane Checking via Distributed, On-Device Verification.
2. Network Verification is a Powerful Tool to Check Network Forwarding Behaviors. It examines the configurations, the forwarding tables or ACLs, or the runtime of routers and switches, to determine whether a network would forward or are actually forwarding packets along pre-determined sets of paths.

This talk focuses on data plane verification, or DPV for short, because they provide a high coverage of the actual forwarding behaviors of the network.

1. DPV tools usually adopt an SDN-like architecture. A cluster of servers play the role of a centralized verifier. It collects the data plane from routers and switches through a management network, and then run computation-intensive, memory-intensive algorithms to verify the network
2. Although people have designed very powerful data structures and algorithms such as atomic predicates to improve the efficiency of the verifier, centralized DPVs do not scale to large networks due to its architectural limitations. Specifically, it requires a highly-available management network, which is hard to build. The verifier itself also becomes a performance bottleneck and single PoF. More importantly, because the verification happens outside the network, reacting to the found errors is slow as the decisions need to go through the management network again.
3. To tackle these scaling limitations of DPV, we propose distributed, on-device verification. Our basic idea is to offload verification to distributed computations on network devices. By removing the centralized verifier, this distributed architecture minimizes the involvement on management network, avoids the verifier becomes a bottleneck or single PoF. It also could support flexible in-situ fast reroute such as convergence free routing by allowing vendors to provide verification-as-a-service

With all these potential benefits from the architecture point of view, a fundamental challenge remains: that is, how to achieve flexible, correct distributed DPV on commodity switches?

1. We illustrate how we tackle this fundament challenge with an example. Suppose the operator wants to verify in this topology, whether all packets entering the network from S with a destination IP in 10.0.0.0/23 will be delivered to D in a simple path waypointing W.
2. We first design a declarative language for operator to specify this invariant in a 4-tuple of the packet space, ingress, set of valid paths expressed in regular expression, and number of valid paths allowed in the network.
3. Given the topology and the specified invariant, the problem of DPV can be defined as to count how many valid paths exist in the data plane. To decompose this computation-intensive problem, our key insight is that by treating the topology as a deterministic finite automata and multiply it with the regular expression in the invariant, we can compute a DAG called the DPVNet, which compactly represents all valid paths in the topology. Afterwards, the problem of DPV can be solved by a reverse topological traversal along the DPVNet.
4. This reverse counting can be naturally decomposed into on-device tasks corresponding to nodes on the DPVNet, where each node counts the number of downstream paths provided by the data plane, and sends the result to all its upstream neighbors.

To illustrate the process, assume device D's action is to forward all packets in 10.0.0.0/23, denoted as P1, to the corresponding egress.

Therefore, D1's counting is (P1, 1). D sends D1's results to the devices of D1's upstream neighbors, W1, W2, B2.

When W receives the result, because W forward P1 to D, W1 and W2 each updates its counting to (P1, 1).

However, because B only forwards packets in P3 and P4 to D, but drops packets in P2, B2's counting is updated as (P2, 0) and (P3 and P4, 1).

W and B then each sends the results of W1, W2 and B2 to these nodes' upstream neighbors. The process continues as B1 updates its result to (P1, 0) because B1's downstream neighbor is W1, but B does not forward any packet to W.

When A receives the results of both B1 and W2, A1 will update its results for P2 and P4 as 1. This is because although A forwards P2 to both B and W, because B1's results is 0 and W2 is 1, the counting results of A1 for P2 is calculated as 1+0 equals to 1. For packets in P3, A forwards them either to B or W, but not both, depending on the blackbox hashing algorithm of device A. Therefore, the counting of P3 at A1 has 2 possibilities: 0 or 1.

After A sends the results to S, the ingress of the invariant, and S computes its own result, S will determine that the invariant is violated because not all packets in P1 can always be delivered to D along a simple path passing W.

1. This distributed counting is event-driven and incremental. Suppose B updates its action to forward P3 and P4 to W. nodes B1 and B2 will update their counting accordingly, and then send to their upstream neighbors in the DPVNet for further updates. Eventually, S will update its result to (P1, 1), indicating that the network data plane is correct. Our design guarantees the eventual consistency: assuming the network becomes stable at some point, the ingress devices will eventually update its count result to be consistent with the network data plane.
2. Putting things together, we develop a generic, distributed, on-device DPV framework called Tulkun, consisting of an invariant specification language for operators, a verification planner to compute the DPVNet and decompose it into on-device tasks, and a messaging protocol specifying how neighboring devices communicate counting results in an efficient, correct way.
3. Tulkun has a series of extensions, including a sub-pub mechanism to verify packet transformations, a message-free local verification optimization, making Azure RCDC a special case, a precomputing mechanism to verify invariants such as shortest path under k-link-failure, and an incremental deployment design.
4. We implement an open-source prototype of Tulkun and propose its design to the SONiC community
5. To evaluate Tulkun's capability and performance, we first build a 6-commodity-switch testbed to demonstrate how to verify flexible invariants, such as waypoint reachability, multicast, anycast, different-ingress consistent reachability and all-shortest-path availability. We also design an interactive frontend backed with a simulator for visitors to try out Tulkun with different topologies and invariants.
6. We then build a 9-switch testbed mimicking the early Internet2 by using its public dataset and injecting propagation latency based on the geo information of the topology. Tulkun verifies the all-pair loop-free, blackhole-free, limited-path reachability on the snapshot in less than 1 second, and 80% of incremental verification in 5.42 ms.
7. We also compare Tulkun with 5 centralized DPV tools over 13 datasets using simulation, spanning campus network, WAN and DCN.
8. In snapshot verification, Tulkun is substantially faster than all tools in comparison, and can verify a 6000-device DCN in less than 41 seconds.
9. For incremental verification, Tulkun verifies at least 73% of rule updates in less than 10 ms in all datasets and up to 2355x faster than SOTA in 80% quantile.
10. For all our devices in simulation, we run trace-based microbenchmark on 4 models of commodity switches, including one using ARM-based CPU. On all four switches, all devices in the datasets complete initialization in 1.75𝑠, a maximal memory of 19.6𝑀𝐵 and a small CPU load.
11. For 90% messages, all switches process it in 3.52ms, with low CPU and memory overhead
12. To summarize, in this talk, we propose to scale DPV by offloading verification computation to lightweight, distributed computing on commodity network devices, and demonstrate its feasibility and benefits. In the future, we are expanding its scope to distributed and interdomain configuration verification and diagnosis, with preliminary results appear at APNet23 and an ongoing work on DNS verification. We also find synergies between our work and Hydra, and are investigating to extend Tulkun to support distributed runtime verification. Thank you for listening. I'm happy to take any questions.