



High-coverage Testing of Softwarized Networks

Santhosh Prabhu, Gohar Irfan Chaudhry, Brighten Godfrey,
Matthew Caesar

Presenter: KY Chou

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN



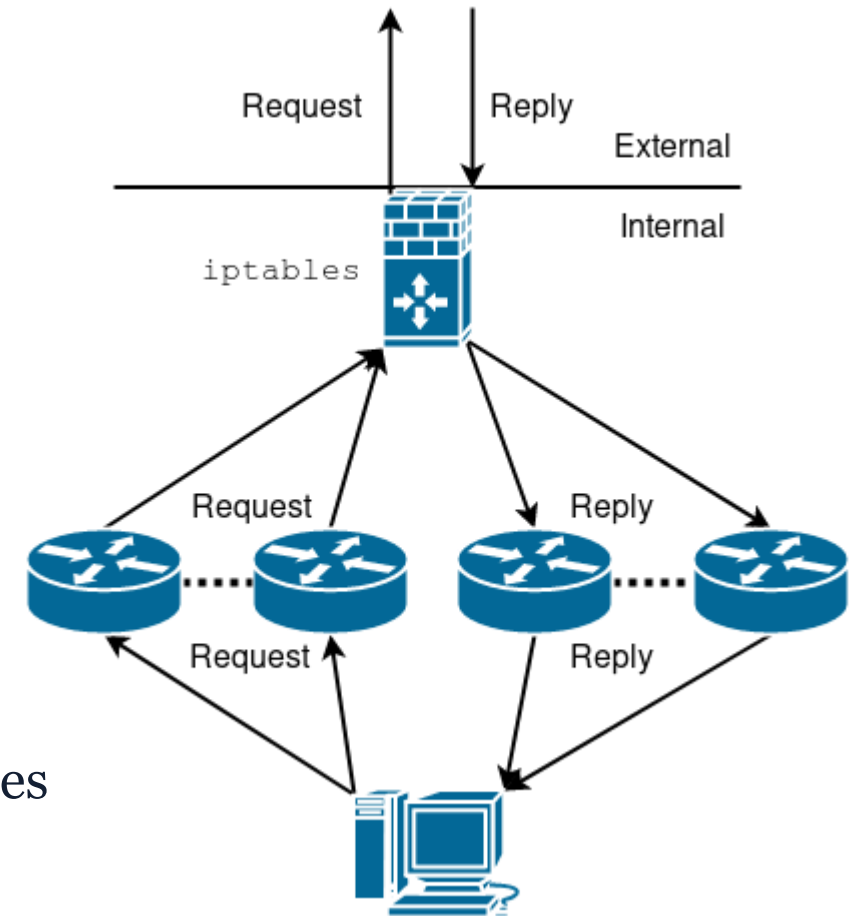
Softwarized Networks

- Networks are incorporating software components on commodity hardware
 - Easier to build sophisticated applications
 - Increases complexity
 - Prone to more bugs and misconfiguration
- How can we ensure correctness and security of such networks?



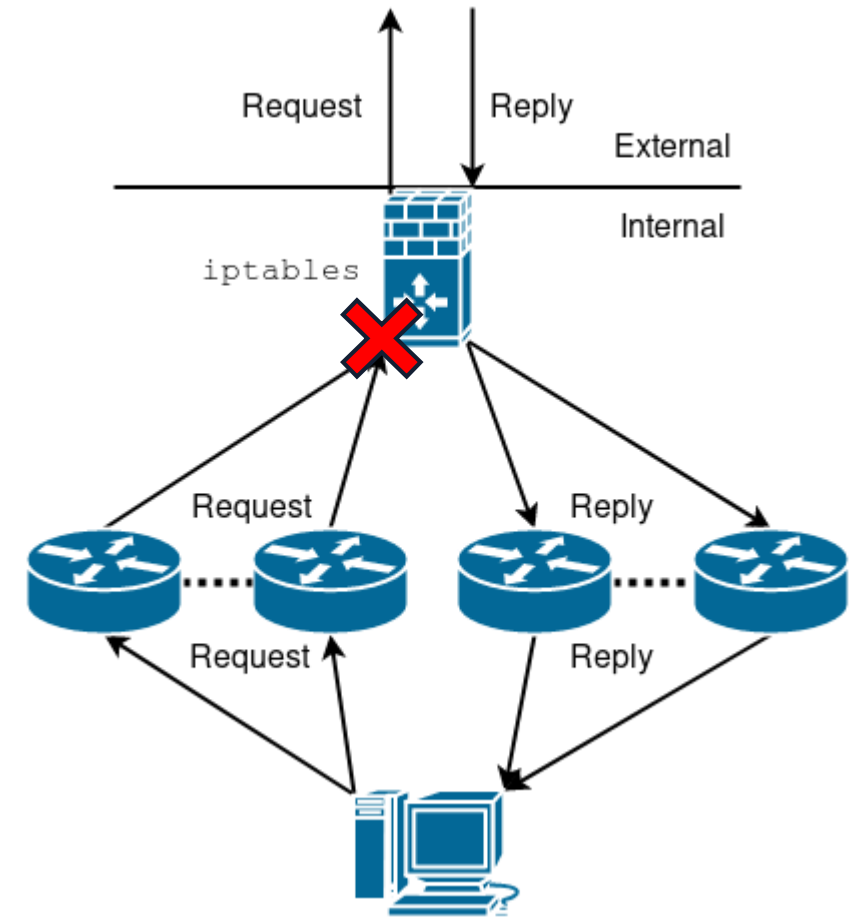
Example: Stateful Filtering

- Stateful filtering (with IPTables)
 - Only replies to past requests are permitted
 - Requests and replies are forwarded through different switches
 - Packet forwarding through switches would be nondeterministic (e.g. ECMP, load-balancing)
- Invariants:
 - Outgoing packets are always allowed
 - Incoming packets are always blocked, unless they are replies



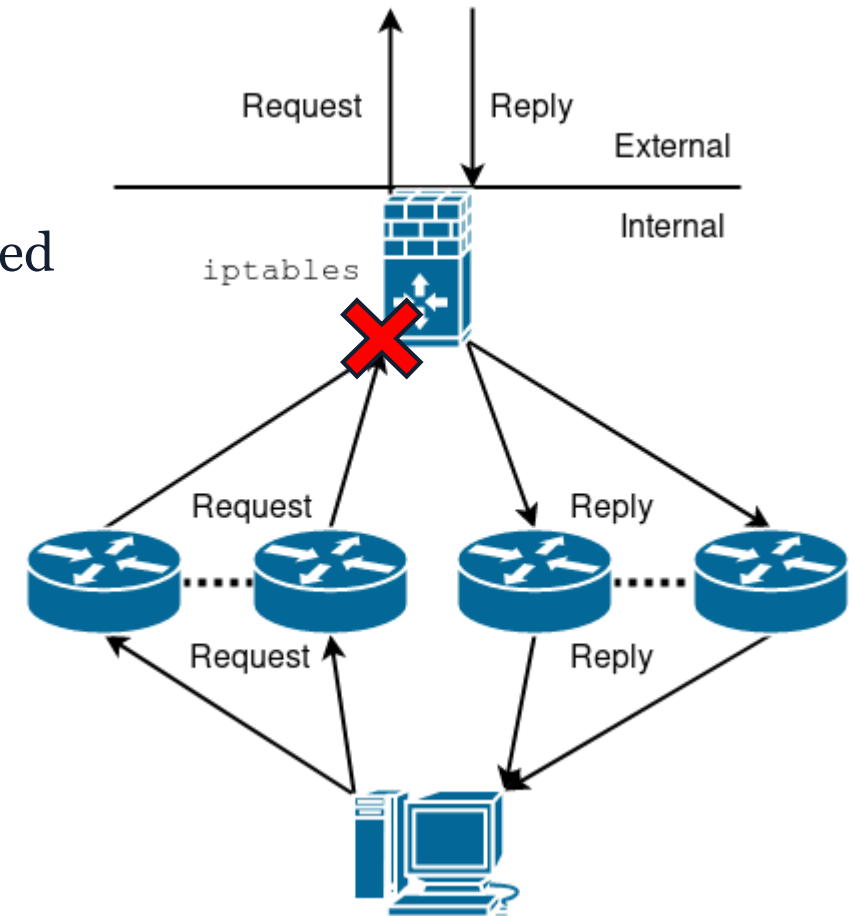
Why this is tricky?

- 1st invariant (requests are always allowed) breaks when reverse-path filtering (RPF) is enabled (`rp_filter = 1`)
 - RPF: Any packet to the source address of the incoming packet must be routed through the same interface



Problem

- How do we detect the violation?
 - Network behavior depends on exactly which software is used
 - Packet forwarding is nondeterministic
 - E.g. some violations may only occur when route updates are interleaved with packet delivery



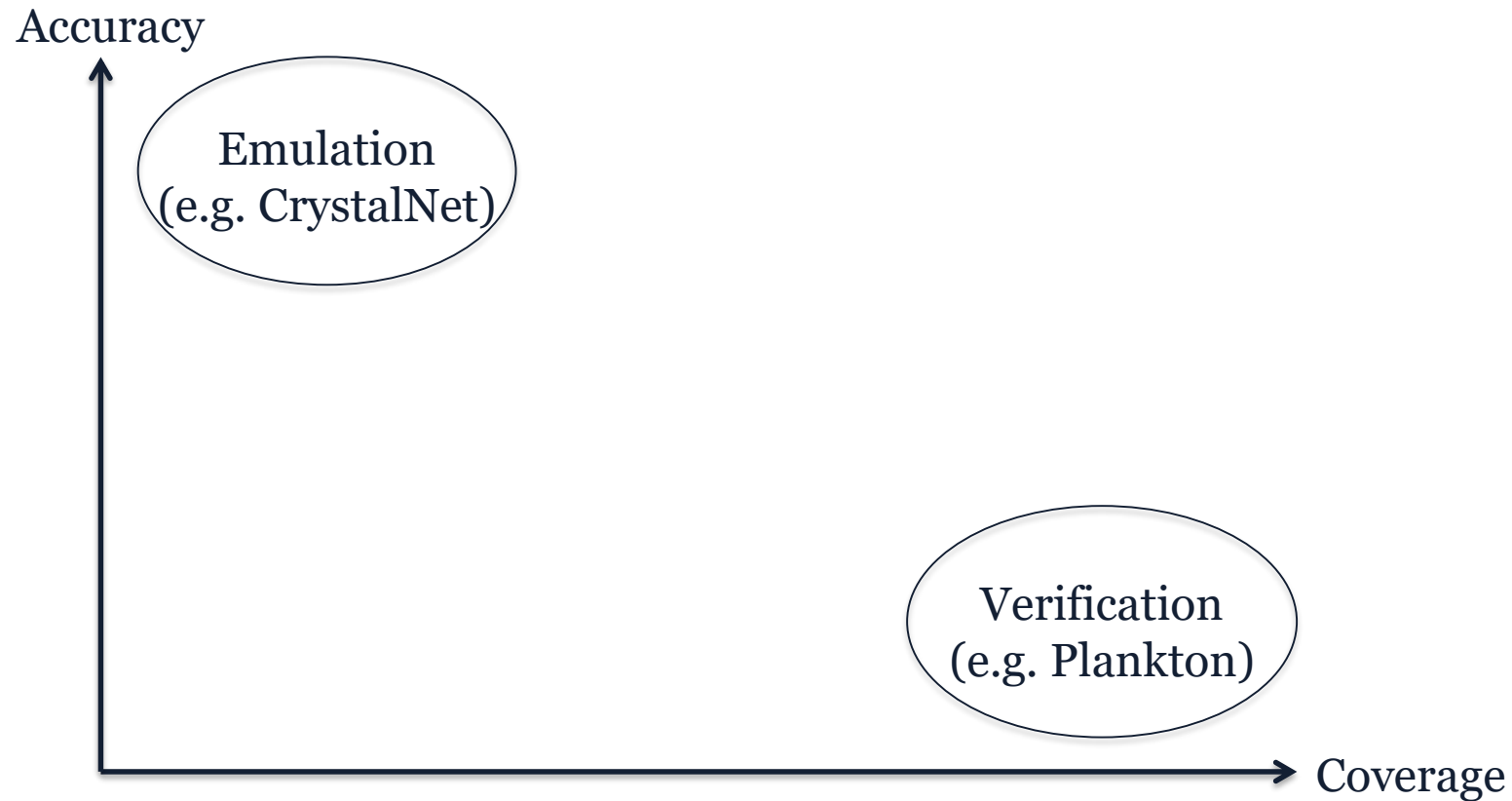
Emulation-based Testing

- Emulation-based Testing (e.g. CrystalNet)
 - Emulate the network with real software
- Problem
 - One execution path at a time, no guarantee of exploration
- Limitation: low coverage
 - May miss issues that stem only from a certain specific condition when there is nondeterminism in the network
 - E.g. Updating routes while packets being delivered

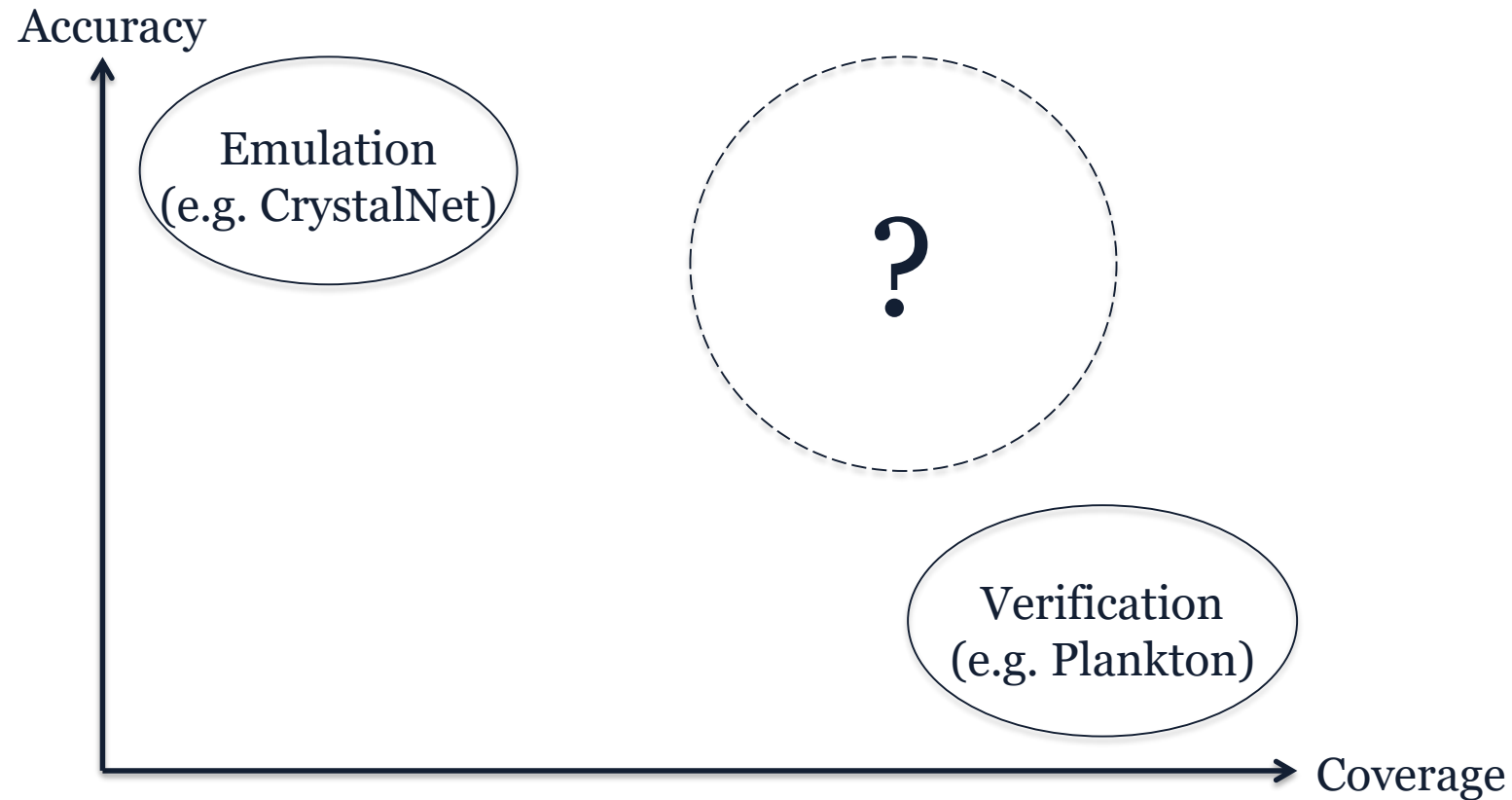
Model-based Verification

- Model-based Verification (e.g. Plankton)
 - Uses an abstraction or models of the network to formally verify some attributes (e.g. “Requests are never dropped”)
- Good: ensures exploration of all possible executions
- Bad: model may be unfaithful to reality
 - Particularly a problem for software components: behavior dependent on custom designs, versions, bugs, and environmental details
 - In our example:
 - Default value of “rp_filter” varies for different Linux distributions and different kernel versions.
 - Even within the same distro, “rp_filter = 1” may still have different semantic meaning in different releases (e.g. RedHat 5 & 6)
 - Full accuracy would require separate model for each variant → practically impossible

Both methods have their own limitations

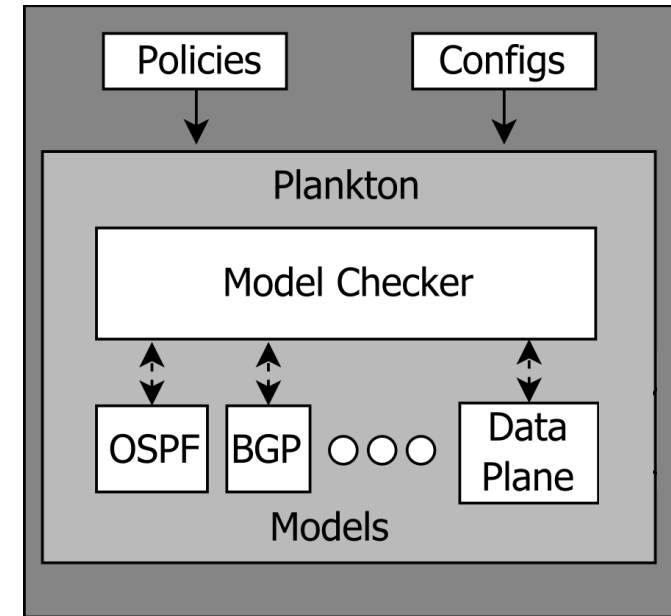


Can we get the best from both worlds?



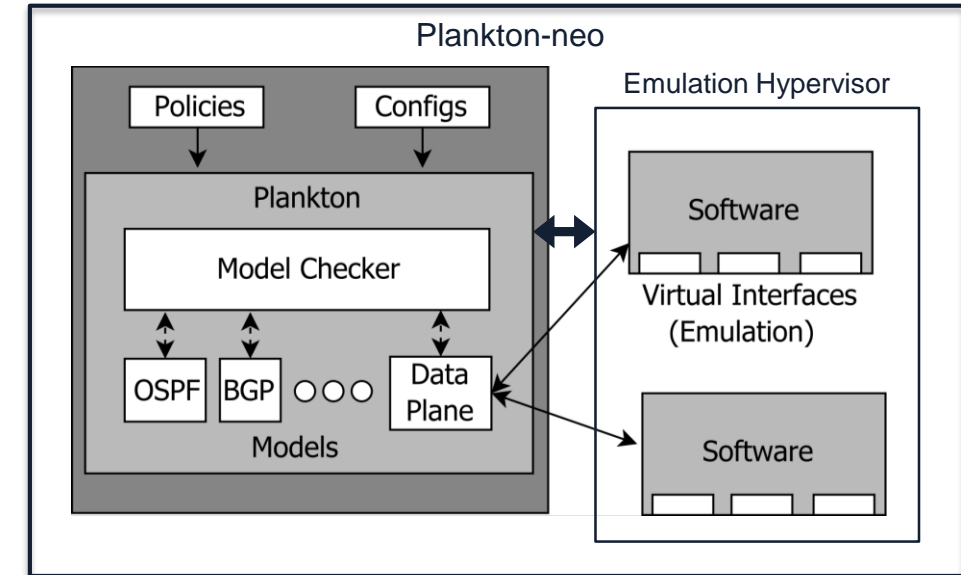
Plankton Overview

- A model-based verification framework
 - Plankton uses manually written models and SPIN model checker to formally verify the network
 - Overall system state is the combination of the states of all models
 - Designed to operate on *equivalence classes (ECs)*, rather than individual packets
 - Equivalence Class (EC): a logical entity representing a class of packets (e.g. request class, reply class, etc.)



Our Approach: Plankton-neo

- Incorporating real software in model-checking
 - An emulated device with virtual interfaces is created for each software component
- Representative packets for emulation
 - Instantiate a concrete representative packet for each EC for emulations
 - Representative packet is injected into the emulation instance
 - Result of the injected representative packet is interpreted by the data-plane model
 - If no packet reaches any virtual interfaces after a timeout (50 ms), it's considered dropped



Our Approach: Plankton-neo

- Emulation instance needs to be in the intended state before packet injection
 - E.g. Request packet has arrived, or routes have been updated, ...
- How to keep track of the emulation states?
 - Use a model inside Plankton to track emulation states
 - Emulation state := initial state + update1 + update2 + ...
 - History updates are replayed to bring back the emulation state

Our Approach: Plankton-neo

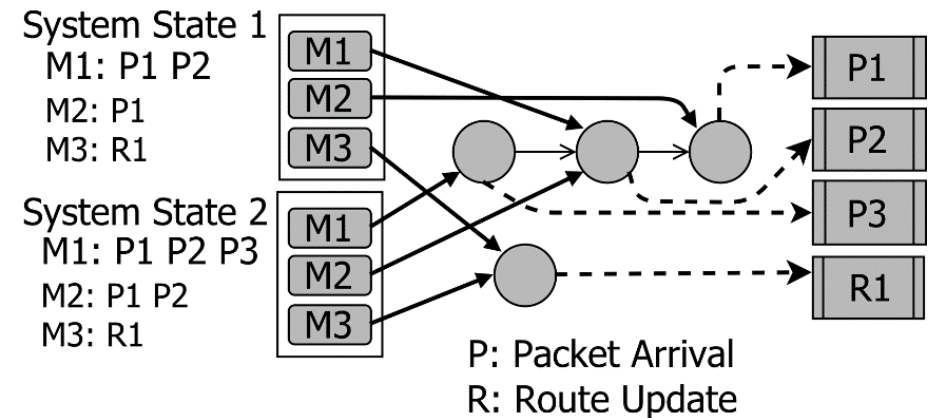
- Slower than pure verification (e.g. Plankton)
 - Instantiate emulations
 - Restore emulation states

Experiment	Plankton (model)	Plankton-neo (real software)
96-nodes network, full exploration	5.41 seconds	413.84 seconds (~6.90 min)
192-nodes network, full exploration	36.66 seconds	4732.13 seconds (~78.87 min)

- We have done some optimizations to mitigate the overhead

Optimization to Improve Performance

- Hashing history updates
 - Observations
 - The same update may be part of multiple update histories
 - Many update histories may differ from each other only in terms of a few updates
 - History updates and any sequence of updates are hashed to reduce the memory overhead
 - Each update is created only once



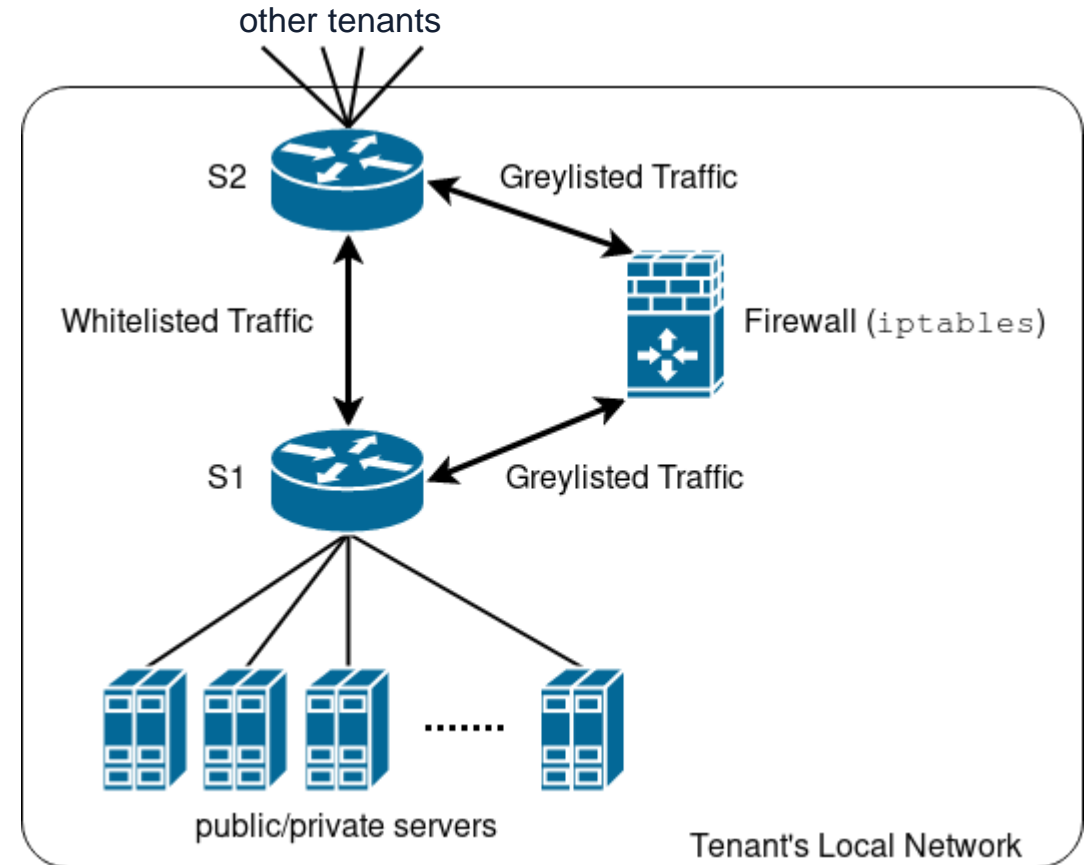
Another Optimization to Improve Performance

- Increase the number of emulation instances
 - It's possible to use multiple number of emulation instances
 - More emulation instances means it is more likely for the emulation to be in the intended state
 - Reduce the time used for state restoration
 - Using separate emulation for each component reduces execution time by ~39%

Experiment	Single Emulation	Multiple Emulation
Experiment setting I	5835.52 seconds	4732.13 seconds
Experiment setting II	680.28 seconds	413.84 seconds
Experiment setting III	29.22 seconds	27.73 seconds

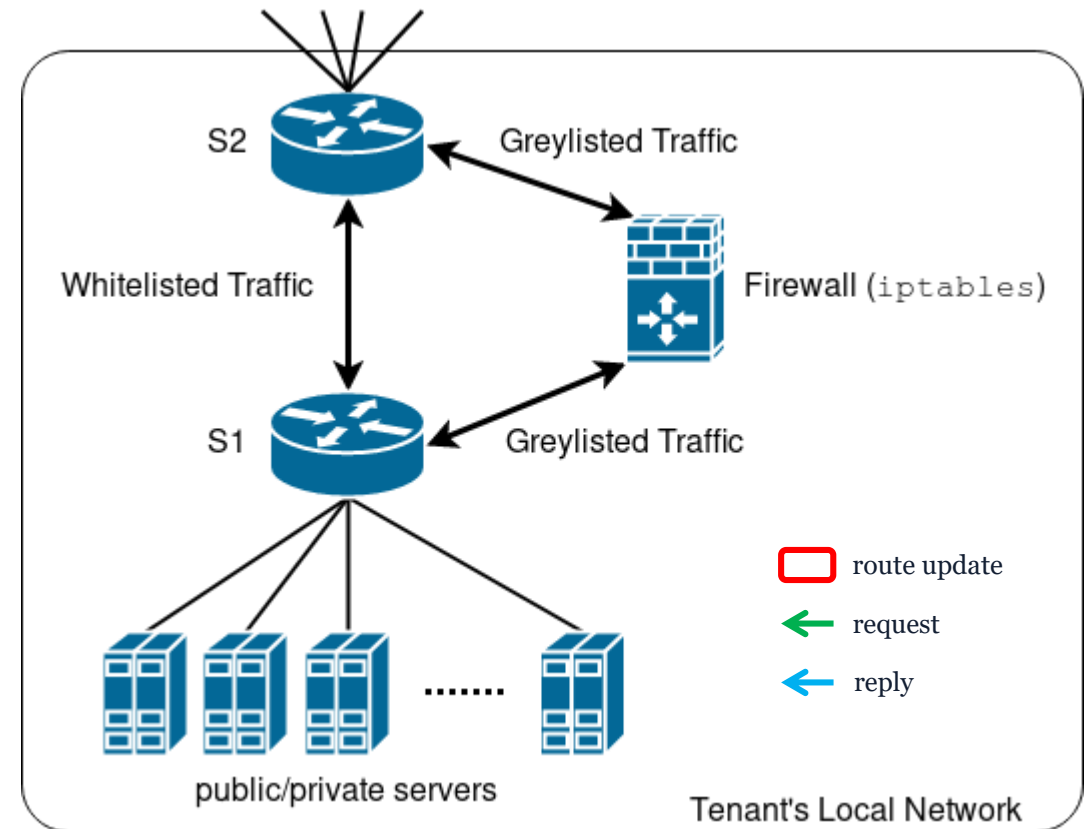
Experimental Results

- Topology: Multi-tenant datacenter
 - Inspired by COCONUT, Plankton
 - The figure shows the logical topology that represents the network of one tenant
 - Traffic: whitelisted/greylisted
 - Servers: public/private
 - Each tenant can see all the other tenants in layer 3
 - Invariants
 - All traffic is allowed for public servers
 - For private servers, whitelisted traffic is always allowed, but greylisted traffic is statefully filtered (only replies to past requests are allowed)



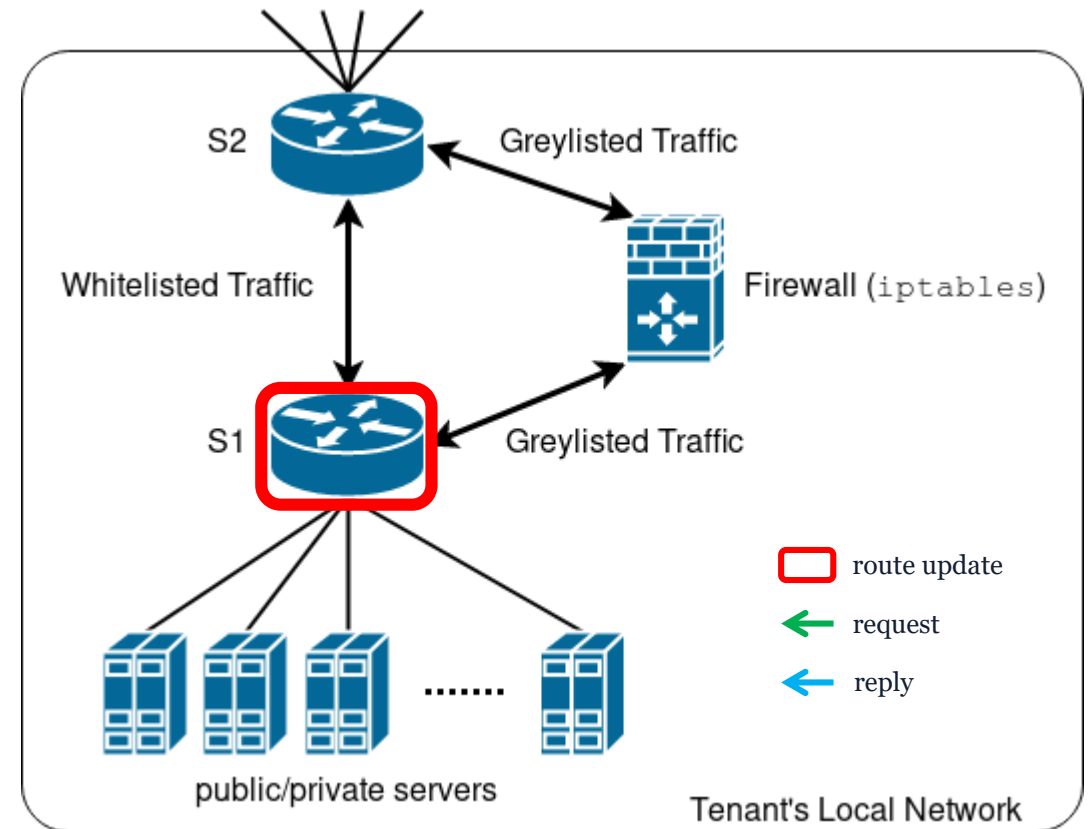
Experimental Results

- Reclassification of tenants' traffic
 - HTTP: greylisted → whitelisted
 - Update routes in S1 and S2
- Policy violation
 - Reply to past request is blocked
 - Event ordering
 - S1 update
 - Sending HTTP request
 - Receiving HTTP reply → dropped
 - S2 update



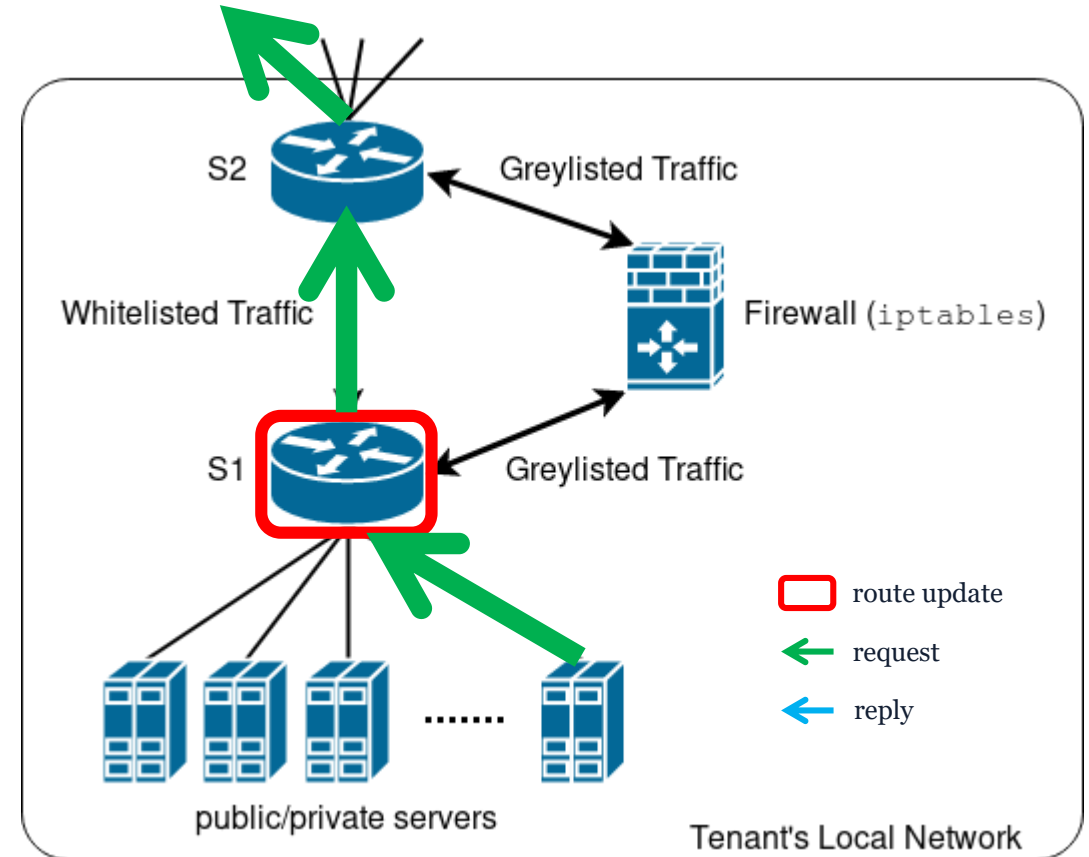
Experimental Results

- Reclassification of tenants' traffic
 - HTTP: greylisted → whitelisted
 - Update routes in S1 and S2
- Policy violation
 - Reply to past request is blocked
 - Event ordering
 - S1 update
 - Sending HTTP request
 - Receiving HTTP reply → dropped
 - S2 update



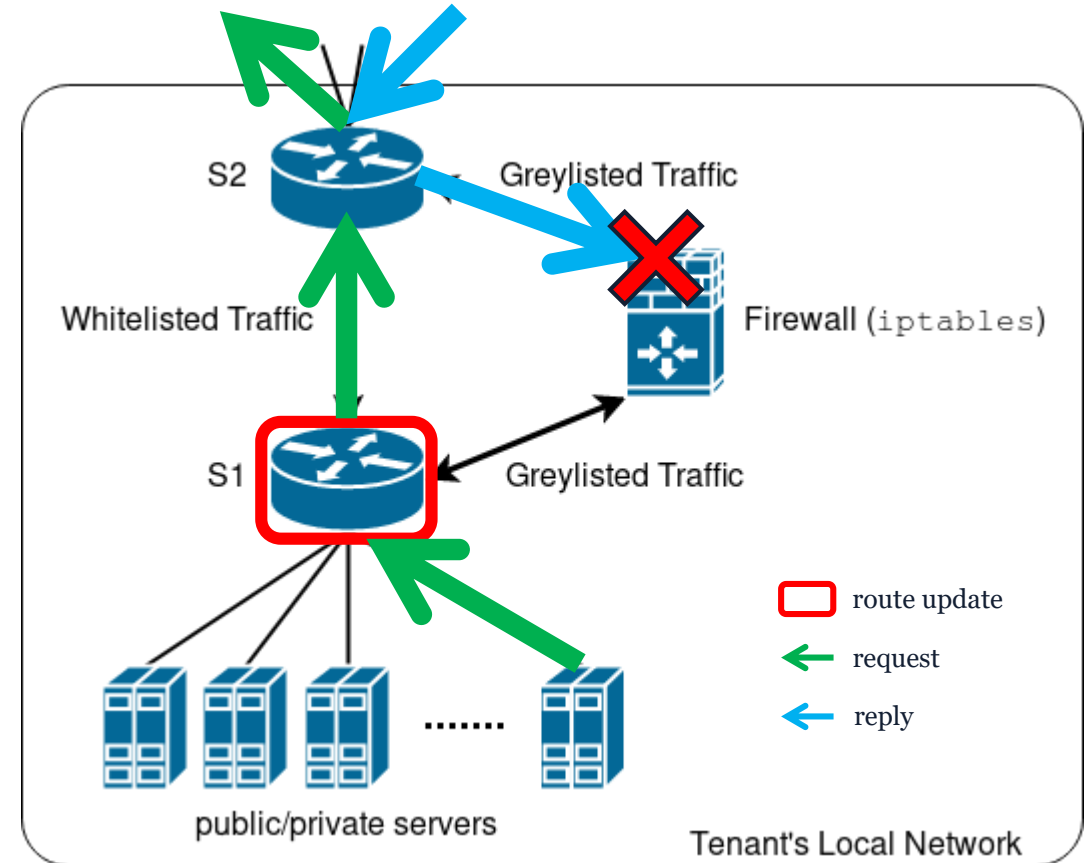
Experimental Results

- Reclassification of tenants' traffic
 - HTTP: greylisted → whitelisted
 - Update routes in S1 and S2
- Policy violation
 - Reply to past request is blocked
 - Event ordering
 - S1 update
 - Sending HTTP request
 - Receiving HTTP reply → dropped
 - S2 update



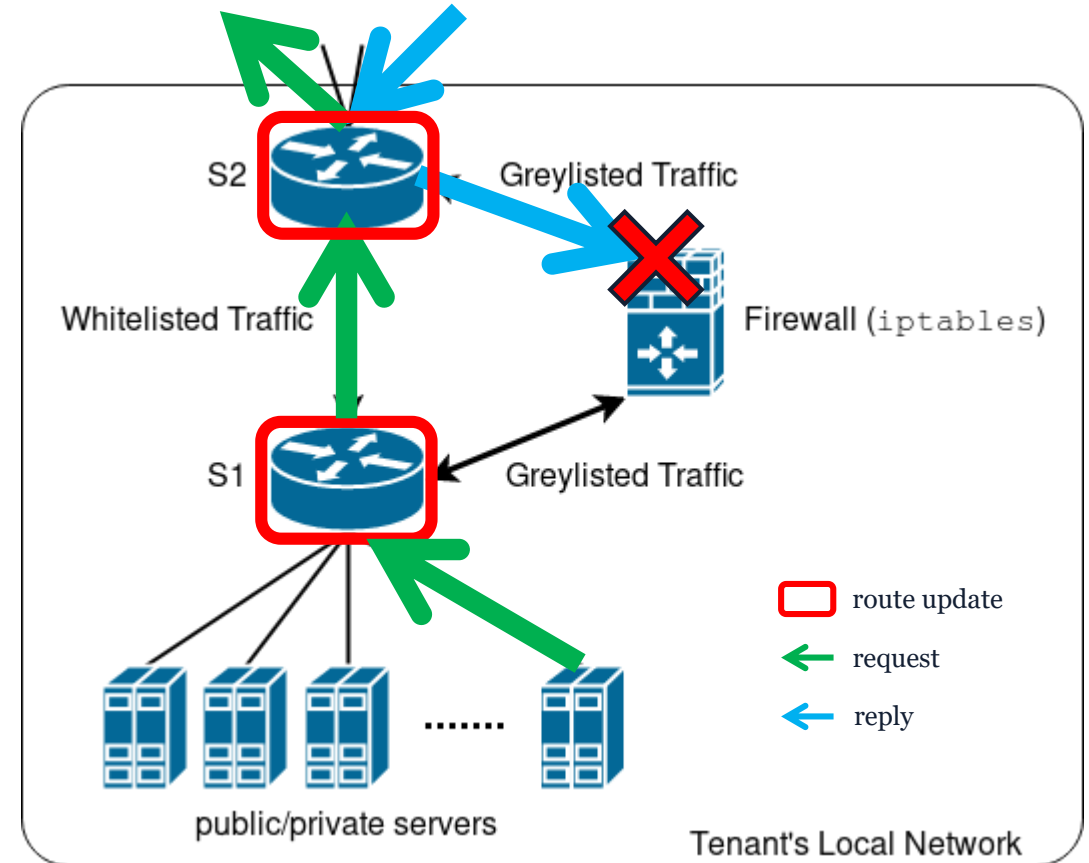
Experimental Results

- Reclassification of tenants' traffic
 - HTTP: greylisted → whitelisted
 - Update routes in S1 and S2
- Policy violation
 - Reply to past request is blocked
 - Event ordering
 - S1 update
 - Sending HTTP request
 - Receiving HTTP reply → dropped
 - S2 update

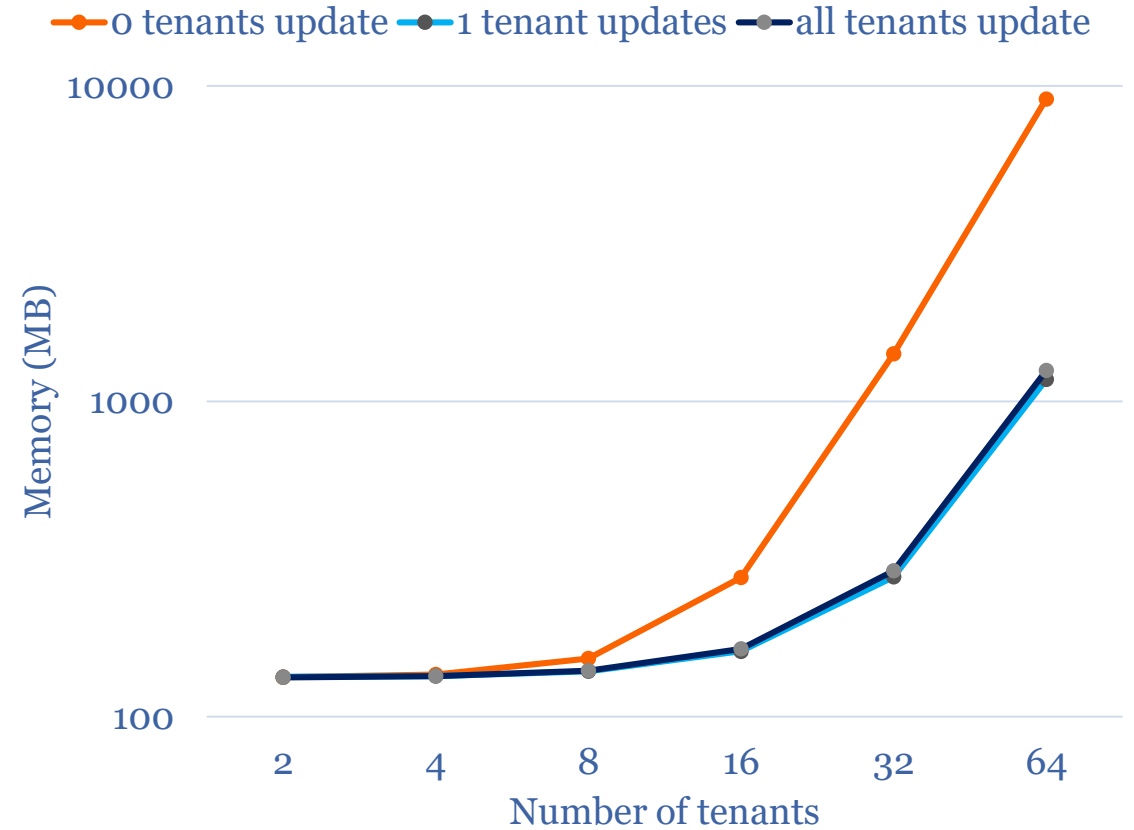
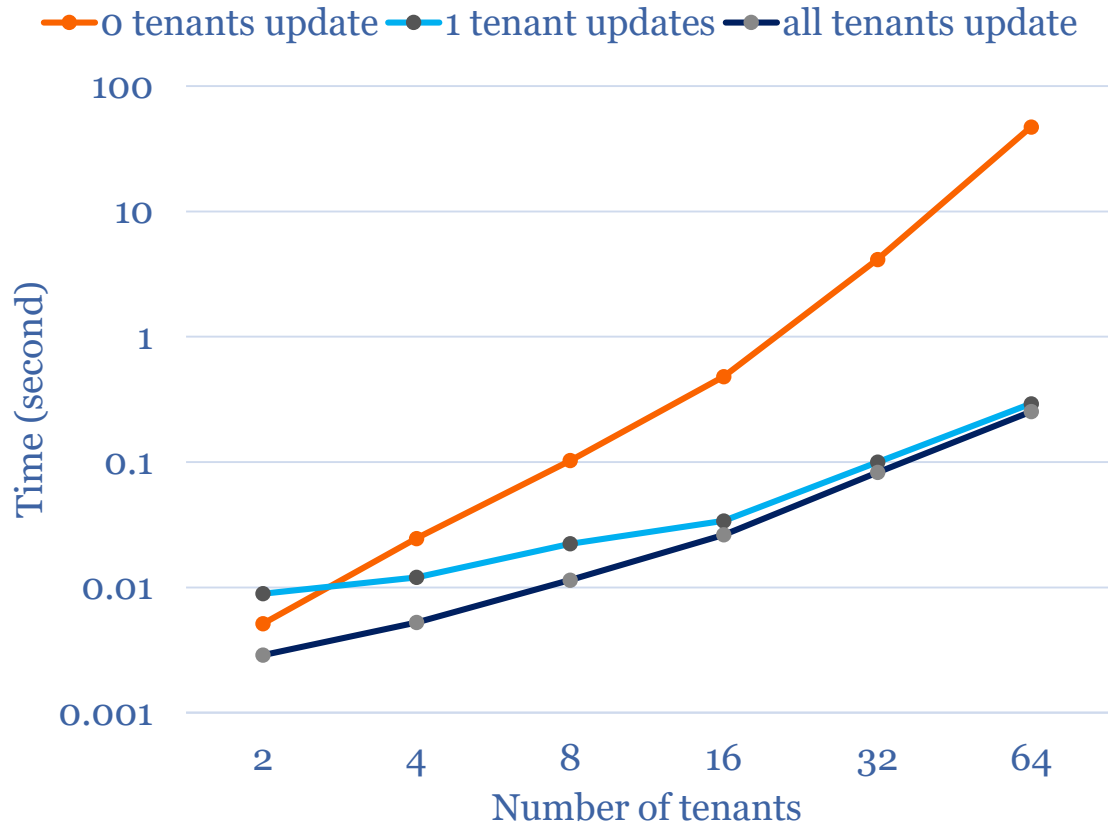


Experimental Results

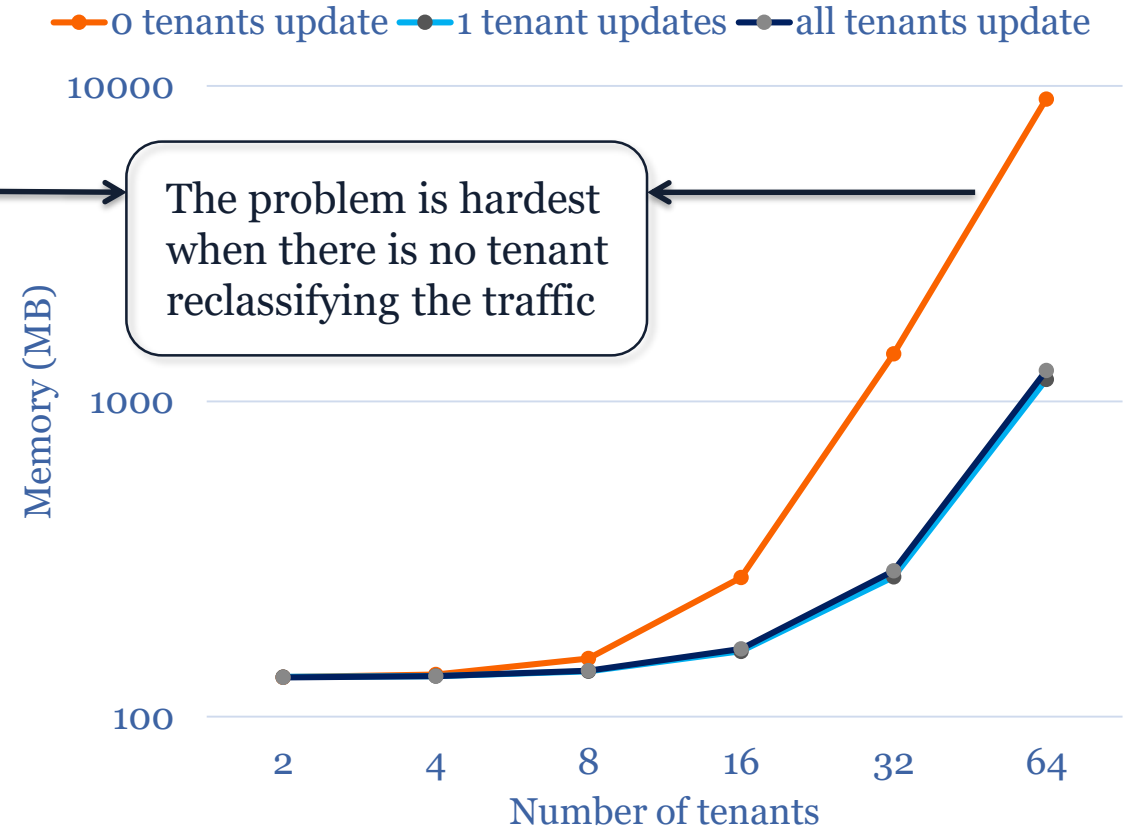
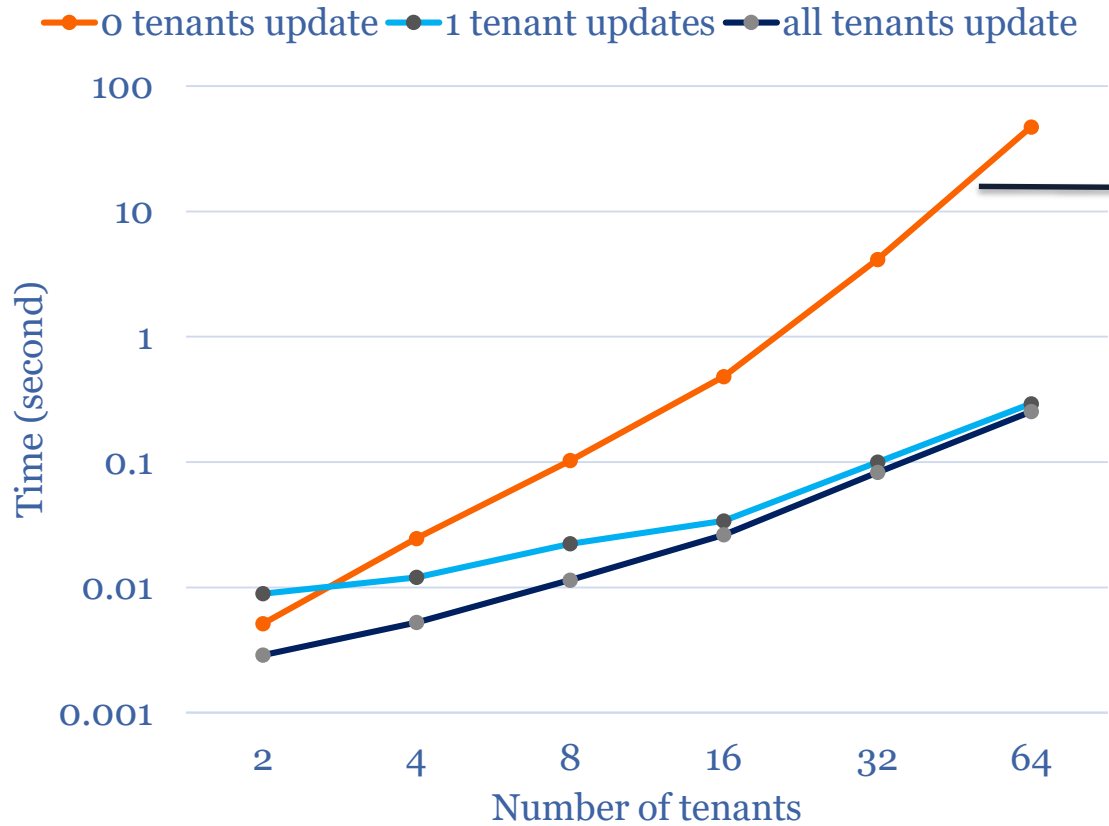
- Reclassification of tenants' traffic
 - HTTP: greylisted → whitelisted
 - Update routes in S1 and S2
- Policy violation
 - Reply to past request is blocked
 - Event ordering
 - S1 update
 - Sending HTTP request
 - Receiving HTTP reply → dropped
 - S2 update



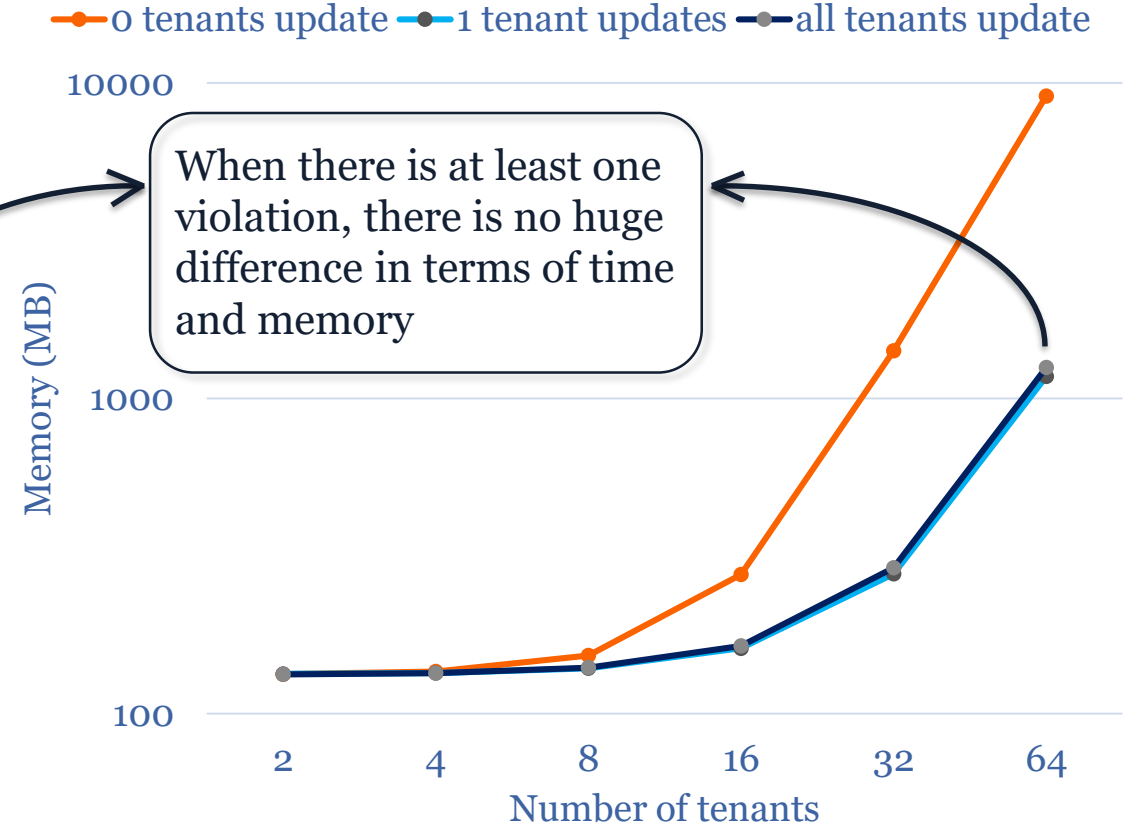
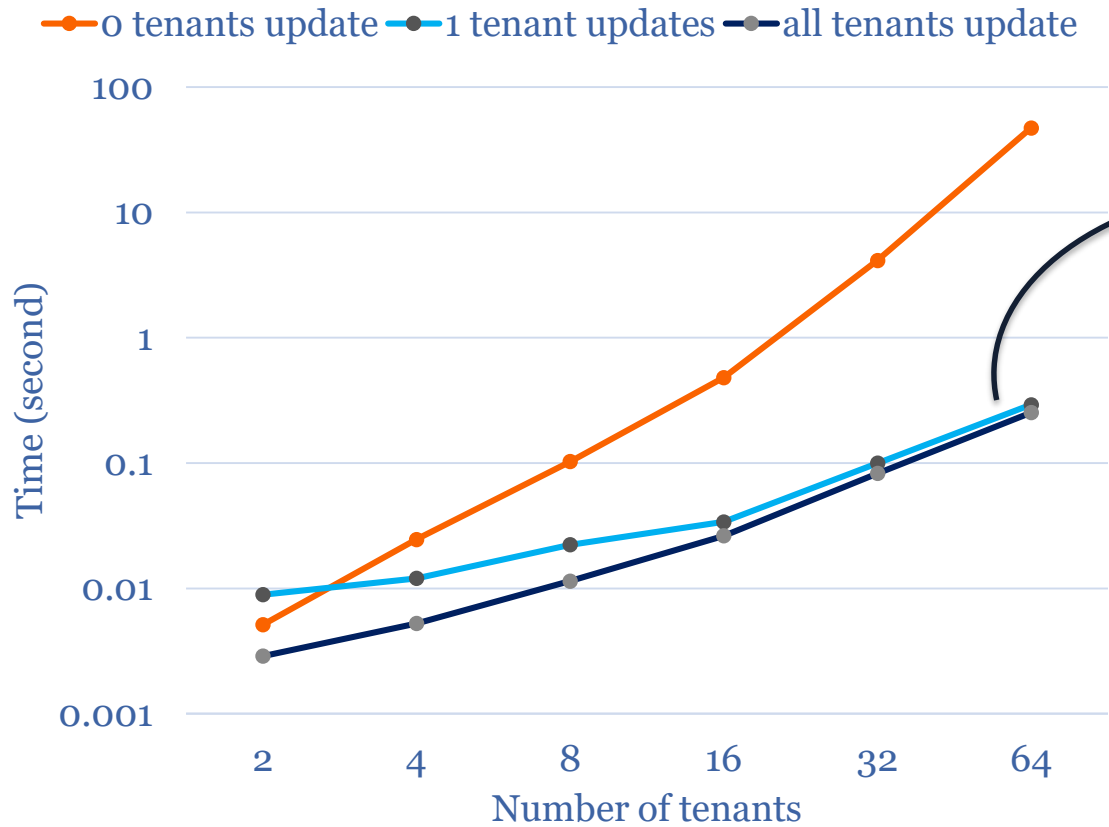
Performance (Time/Memory)



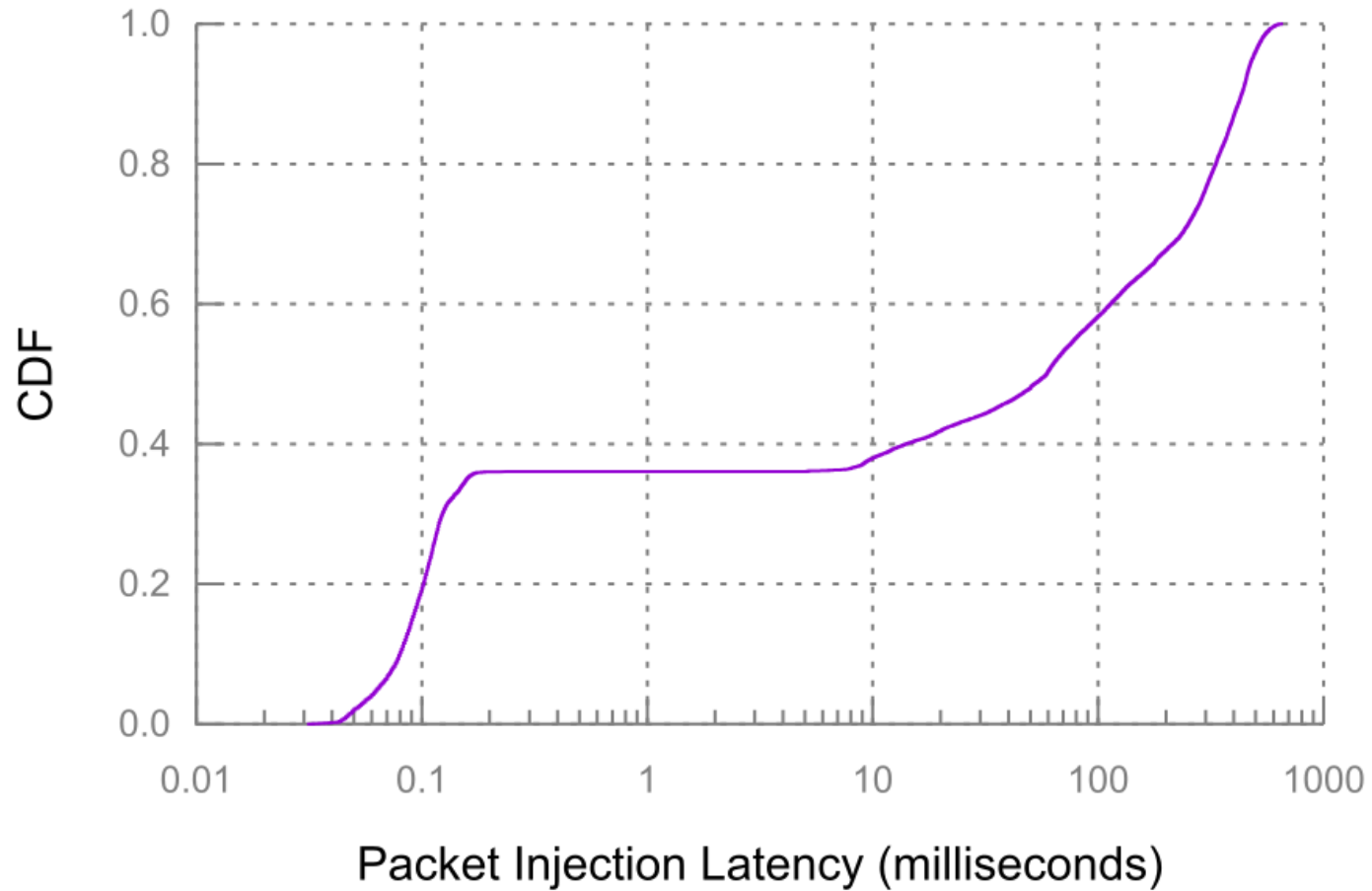
Performance (Time/Memory)



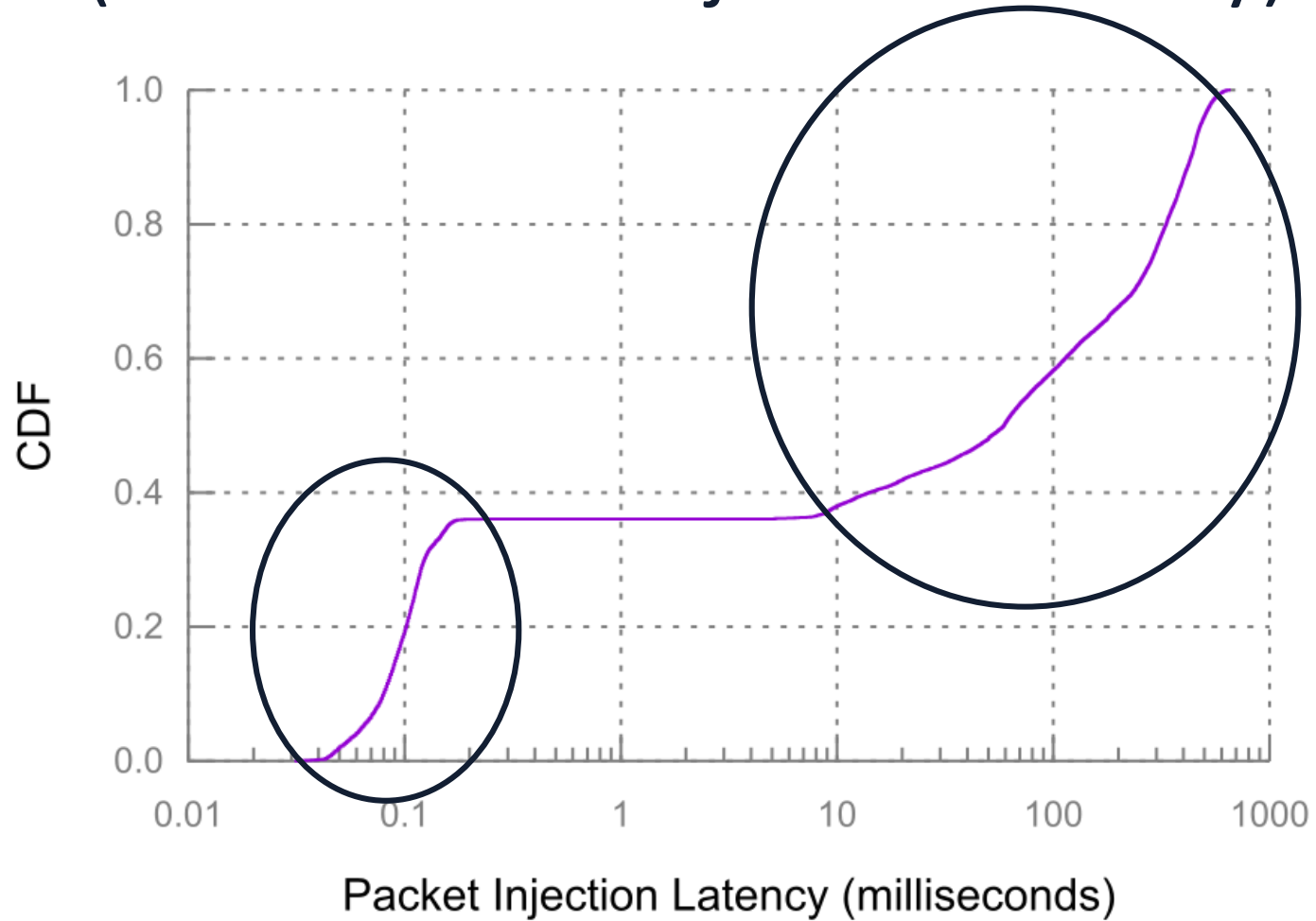
Performance (Time/Memory)



Performance (CDF of Packet Injection Latency)

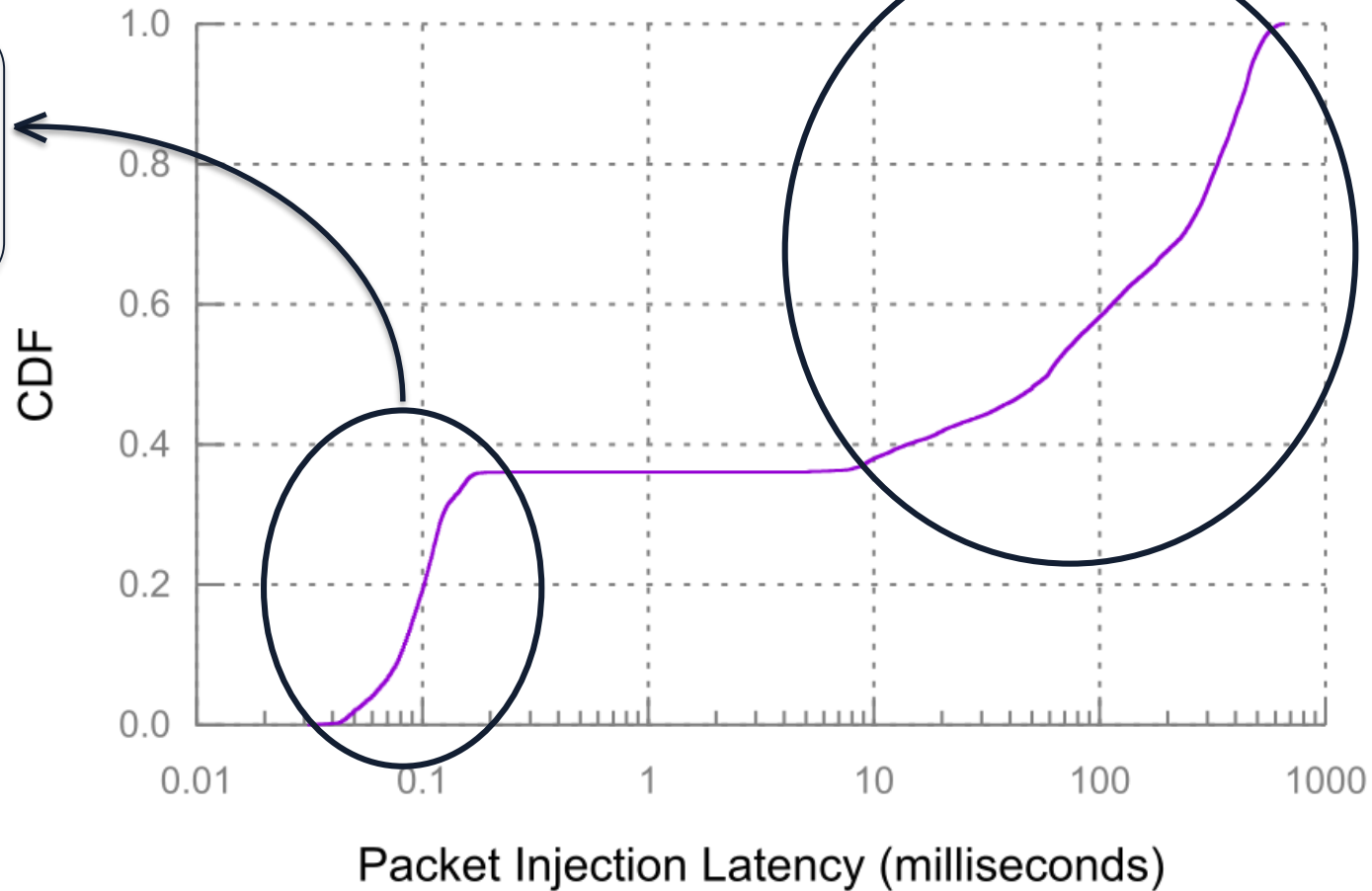


Performance (CDF of Packet Injection Latency)

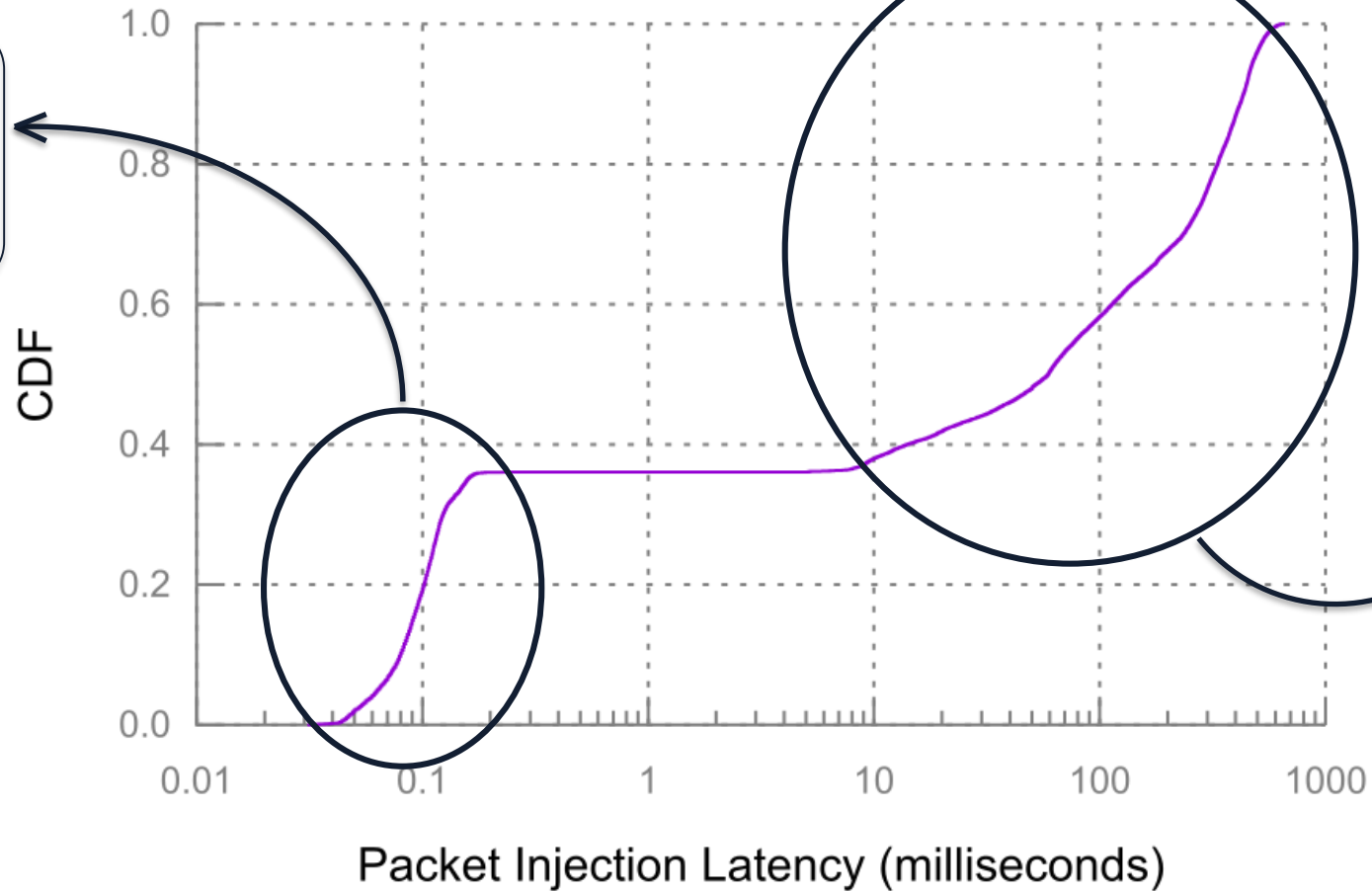


Performance (CDF of Packet Injection Latency)

The faster measurements indicate that the packet is processed immediately without restoring the emulation states



Performance (CDF of Packet Injection Latency)



The faster measurements indicate that the packet is processed immediately without restoring the emulation states

The slower measurements may be due to state restoration or waiting for timeout

Conclusion

- Plankton-neo, a high-coverage testing technique for softwarized networks
- Combining emulation and model-based verification
- The technique itself is not limited to open-source software
- Future work
 - applying the hybrid approach to various software and hardware components



Q&A



UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

Factors of non full-coverage

- Factor 1: nondeterminism inside the software
 - Each representative packet is only injected once
 - Even if the software component is deterministic, the behavior may still be different if we chose a different representative packet from the same EC (this is possible when there are bugs inside the software)
- Factor 2: state transition of emulated component may produce a finer set of packet equivalence classes
 - E.g. stateful filtering
 - The emulated component is treated as a blackbox
 - That finer set of ECs is invisible outside the emulated component
 - Plankton-neo uses an educated guess: considering the reverse packet as a new EC