

SIGCOMM 2026
DENVER

Toward Efficient Verification of Distributed In-Network Computing Programs

Mingyuan Song, Huan Shen, Jinghui Jiang, Qiang Su, Ziheng Zhang, Qingyu Song, Yuchao Zhang, Wanjian Feng, Fei Yuan, Yitao Xing, Wenjia Wei, Qiao Xiang, Jiwu Shu



厦門大學
XIAMEN UNIVERSITY



北京郵電大學
Beijing University of Posts and Telecommunications

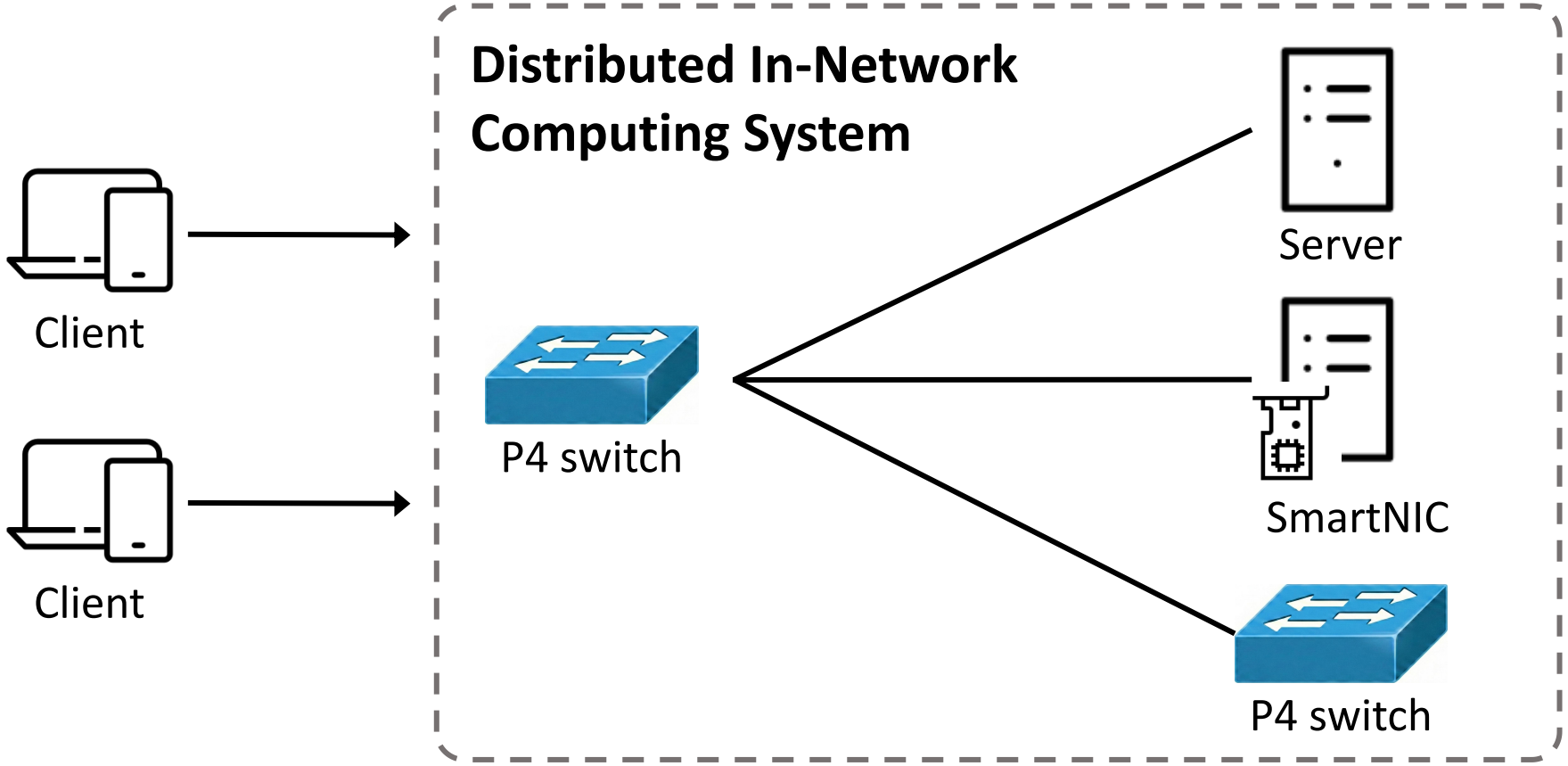
Yealink



HUAWEI

In-Network Computing Is Going Distributed

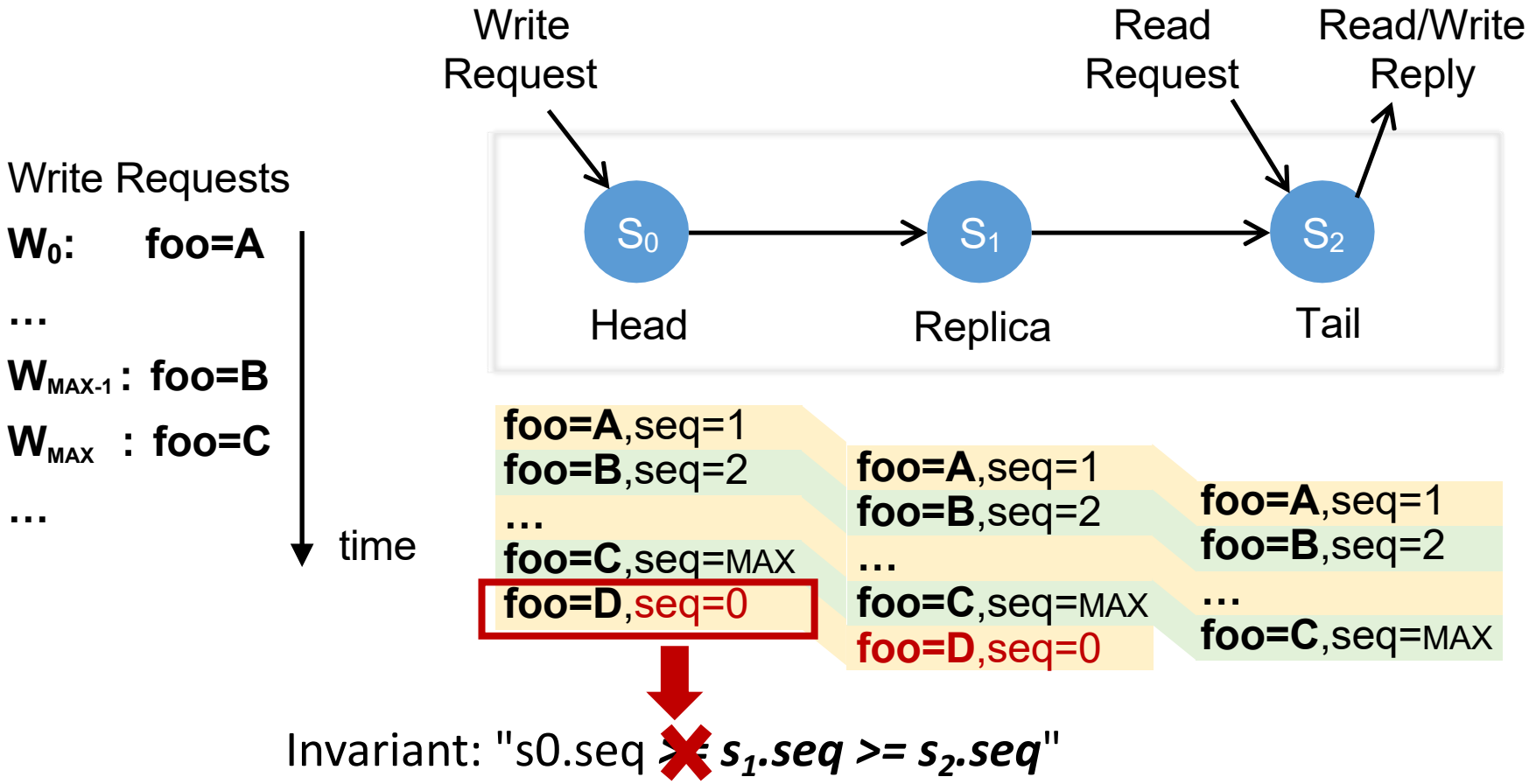
Many systems[1] already use multiple P4 programs to coordinate at line rate



[1] NetChain, Jin et al., NSDI'18 · DistCache, Liu et al., FAST'19 · P4xos, Dang et al., IEEE/ACM ToN 2020 · NetLock, Yu et al., SIGCOMM'20 · ATP, Lao et al., NSDI'21 · P4DB, Jasny et al., SIGMOD'22

Distributed Interactions Introduce Bugs

Wraparound silently breaks the ordering invariant, causing inconsistency



We need a verifier that understands
global state and interactions

Existing P4 Verifiers Miss Global State and Interaction

Cross-switch state and interaction semantics are missed.

- P4-NoD [MSTR'16]
- ASSERT-P4 [CoNEXT'18]
- Vera [SIGCOMM'18]
- p4v [SIGCOMM'18]
- bf4 [SIGCOMM'20]
- Aquila [SIGCOMM'21]

→ Capture neither stateful nor interaction semantics

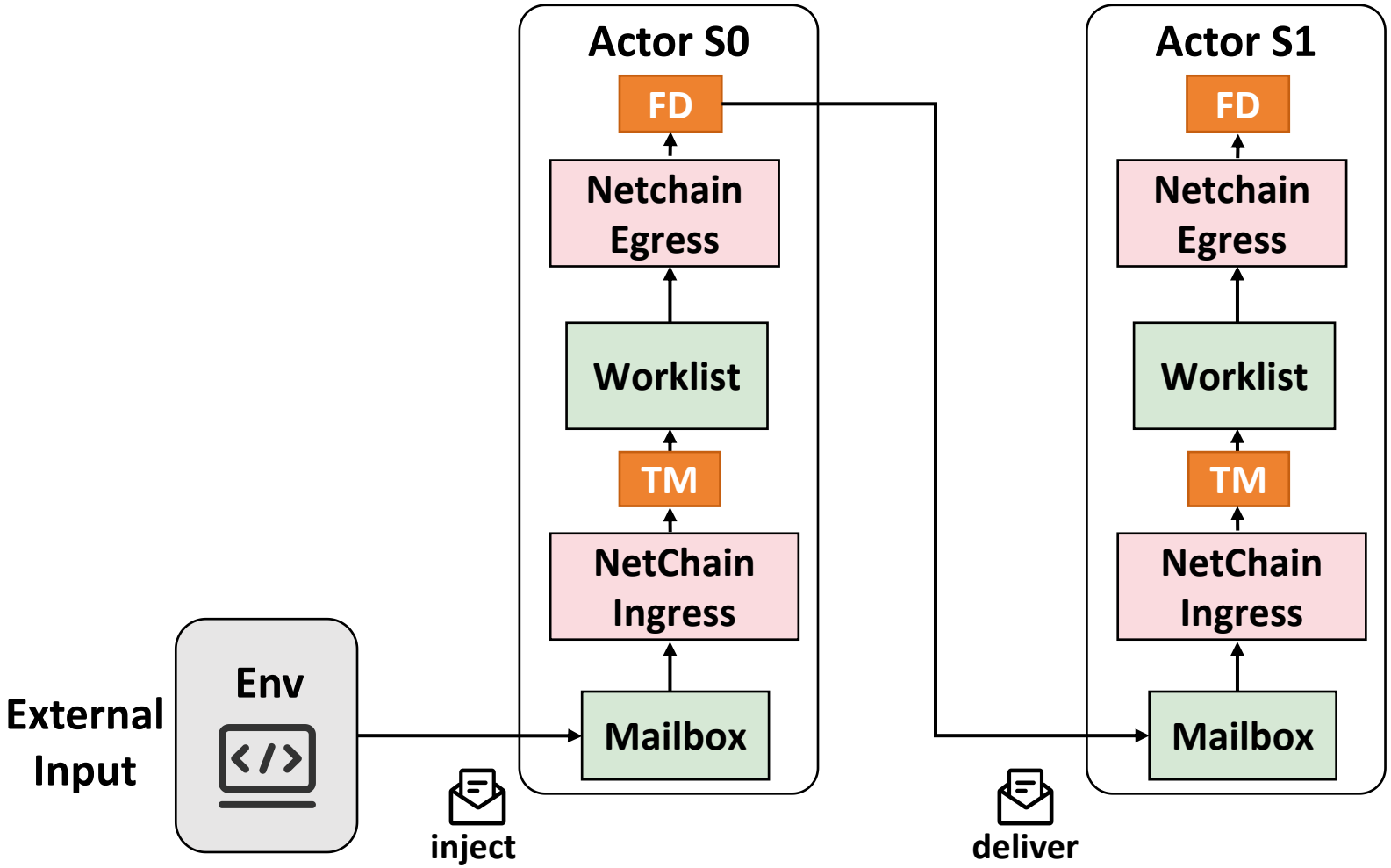
- P4Inv [INFOCOM'24]
- p4tv [NSDI'25]

→ Capture local state, but neither global state nor interaction semantics

Challenge #1: How to model
global state and interactions?

Insight: Each Pipeline Is an Event-Driven Actor

A P4 pipeline naturally behaves as a reactive actor



Mailbox: packets waiting for ingress

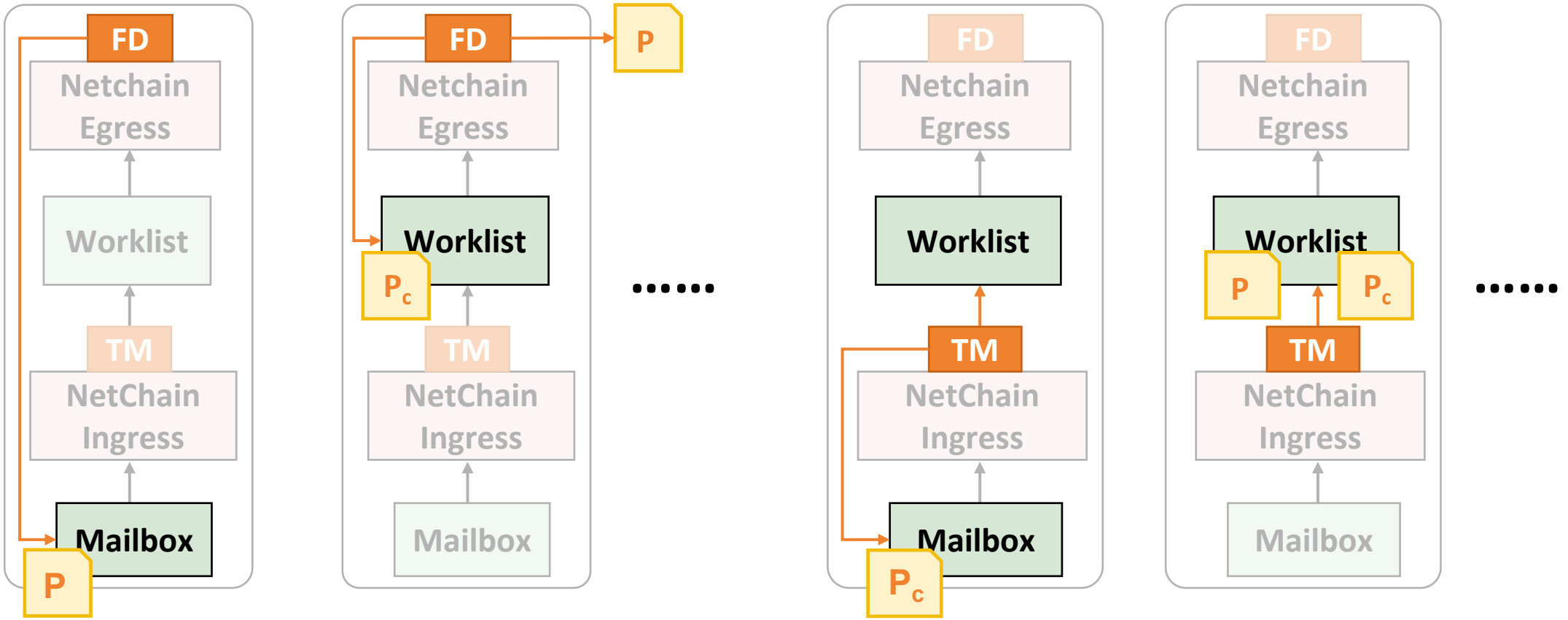
Worklist: packets waiting for egress

FD: Egress → next actor or self

TM: Ingress → next action

Unify P4 Interactions as Message Passing

Different P4 mechanisms are unified as packet movements between actor states



P: original packet Pc: cloned packet

Specify interaction context

import
Program instances

topology
Interaction topology

node
Node-local constraints

global
System-level property

```
import s0 from "netchain_16.p4"
   entries "table_0.txt";
import s1 from "netchain_16.p4"
   entries "table_1.txt";
import s2 from "netchain_16.p4"
   entries "table_2.txt";

topology {
   link s0 -> s1 ALL;
   link s1 -> s2 ALL;
}

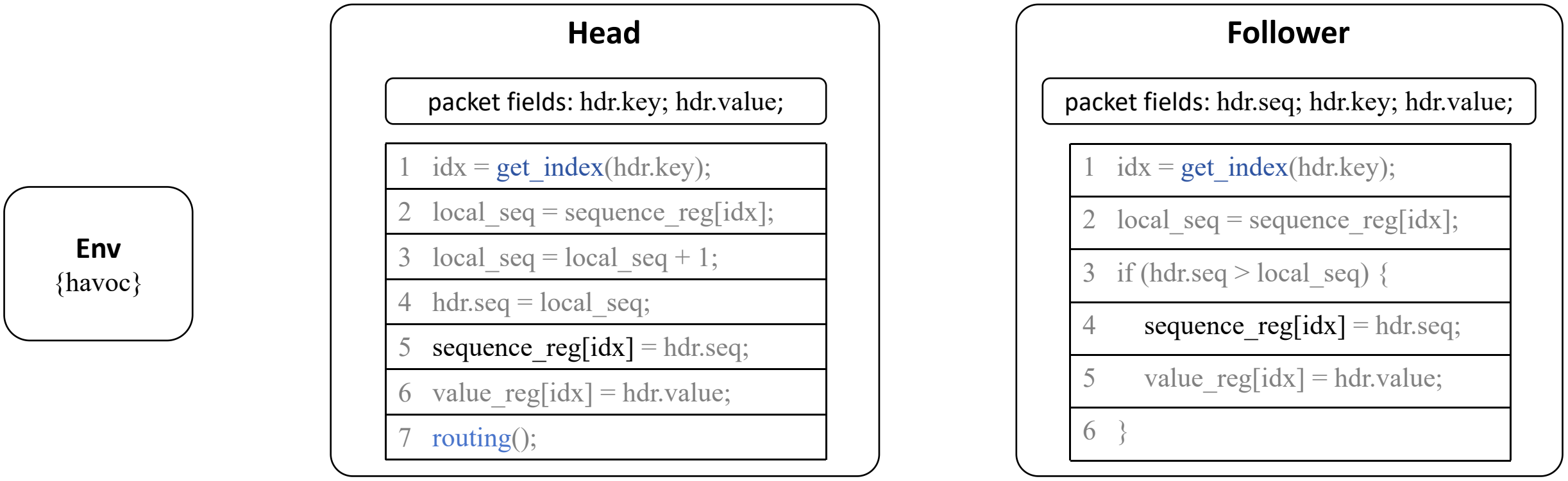
node s0 { external_input = true; }

global {
   queue_capacity = 1;
   assert {
      (s0.sequence_reg[i] >= s1.sequence_reg[i]);
      (s1.sequence_reg[i] >= s2.sequence_reg[i]);
   }
}
```

Challenge #2: How to prune **the state space**?

Insight: Keep Only What the Assertion Needs

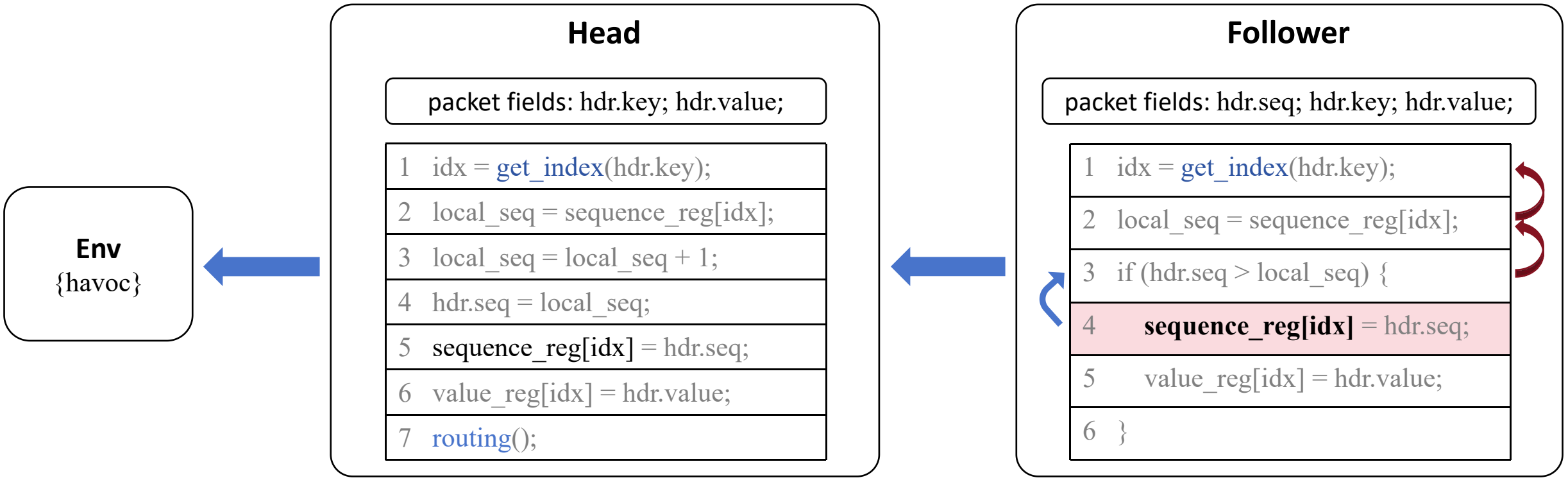
Assertion-irrelevant statements and inputs are pruned without losing bugs



assert (Head.sequence_reg[idx] >= **Follower.sequence_reg[idx]**)

Intent-Driven Pruning

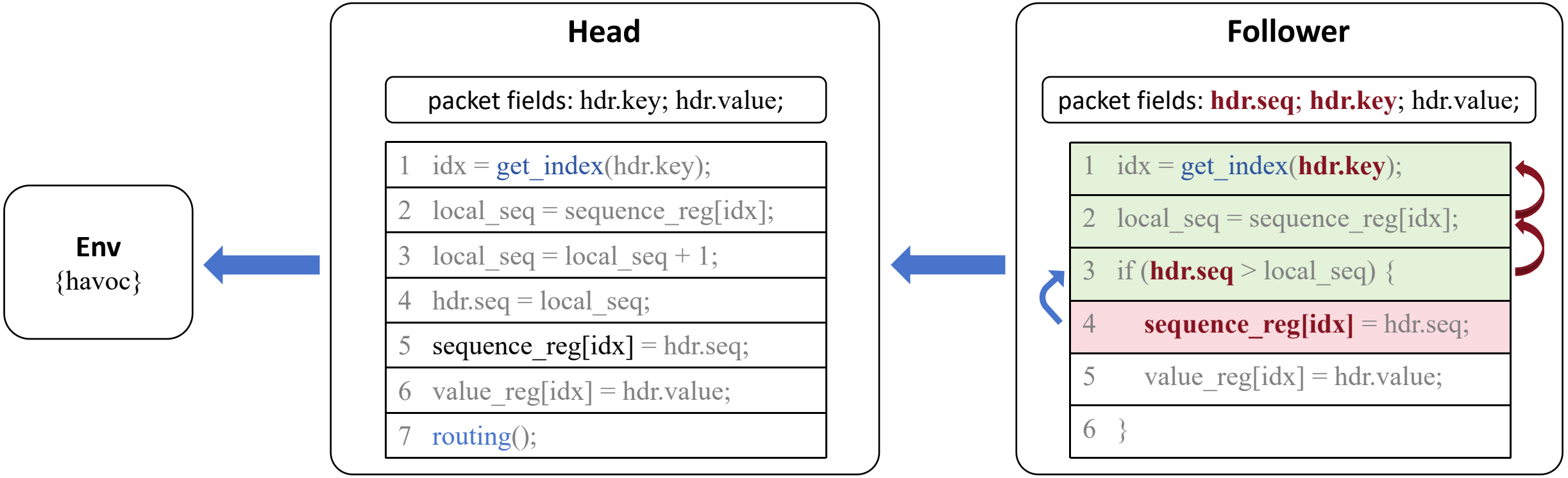
① Start from the assertion: the follower's sequence_reg update is the seed



assert (Head.sequence_reg[idx] >= **Follower.sequence_reg[idx]**)

Intent-Driven Pruning

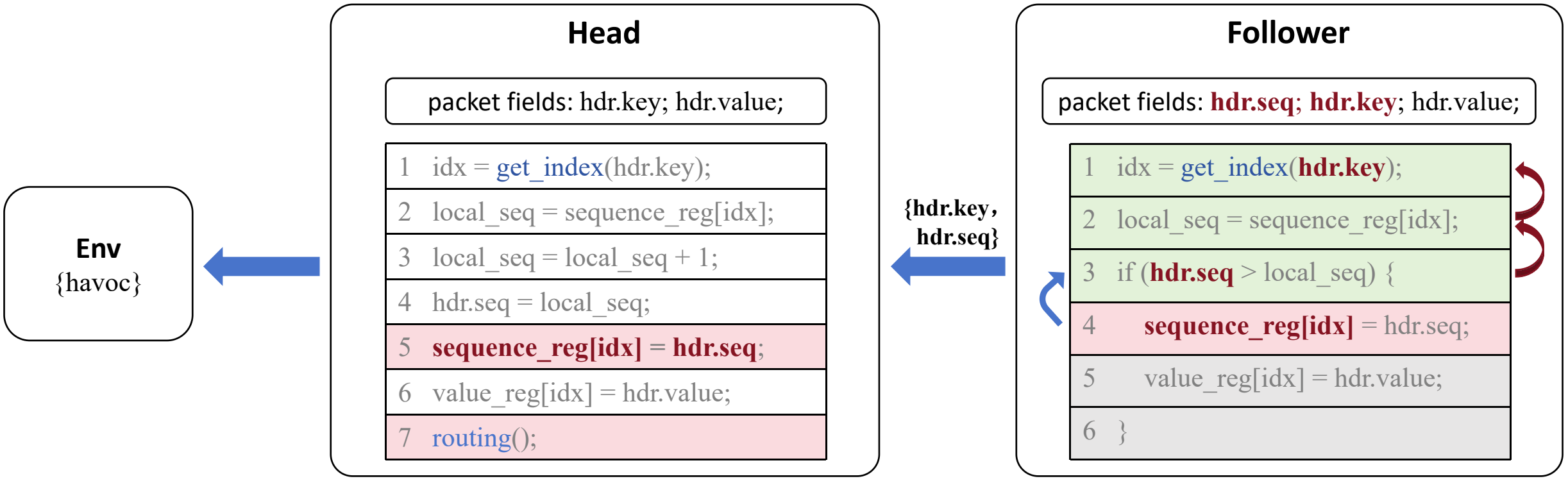
② Slice the follower locally: keep what the seed depends on → needs {hdr.key, hdr.seq}



assert (Head.sequence_reg[idx] >= **Follower.sequence_reg[idx]**)

Intent-Driven Pruning

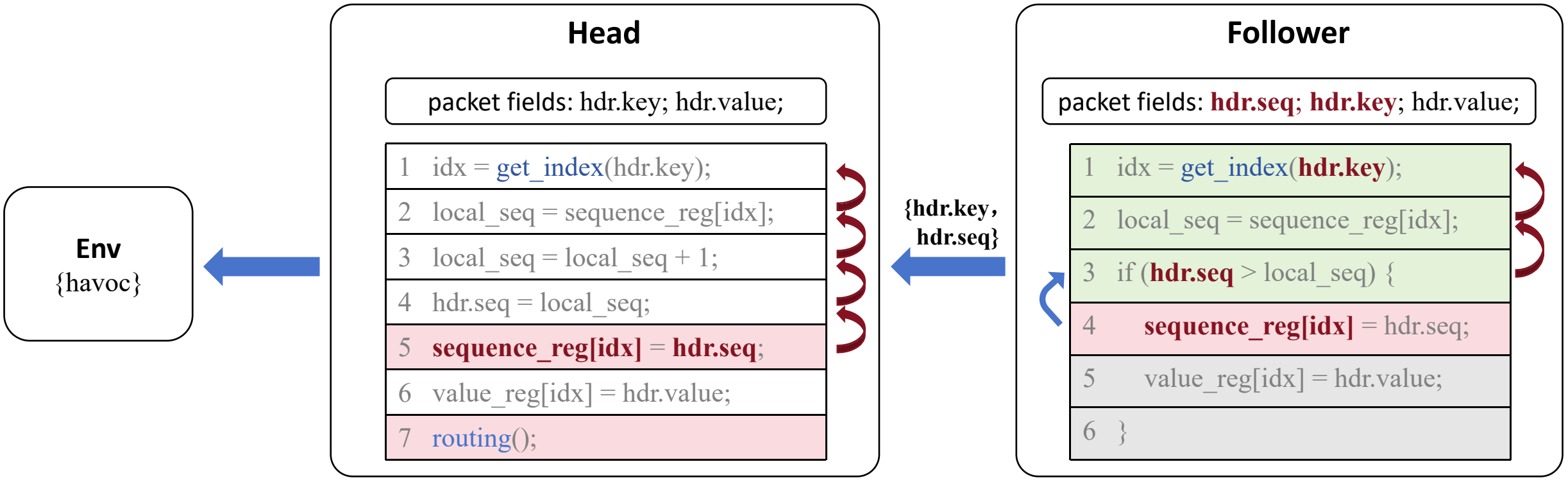
③ Propagate across nodes: the follower's need becomes the head's required output



assert (Head.sequence_reg[idx] >= **Follower.sequence_reg[idx]**)

Intent-Driven Pruning

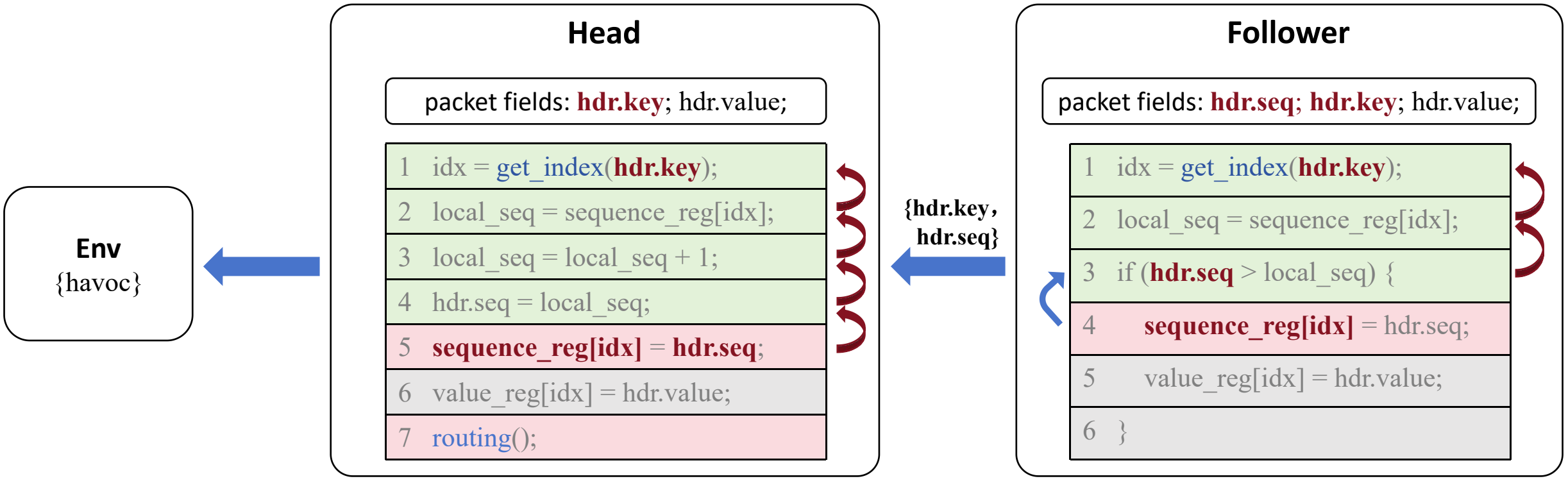
④ Slice the head the same way



assert (**Head.sequence_reg[idx] >= Follower.sequence_reg[idx]**)

Intent-Driven Pruning

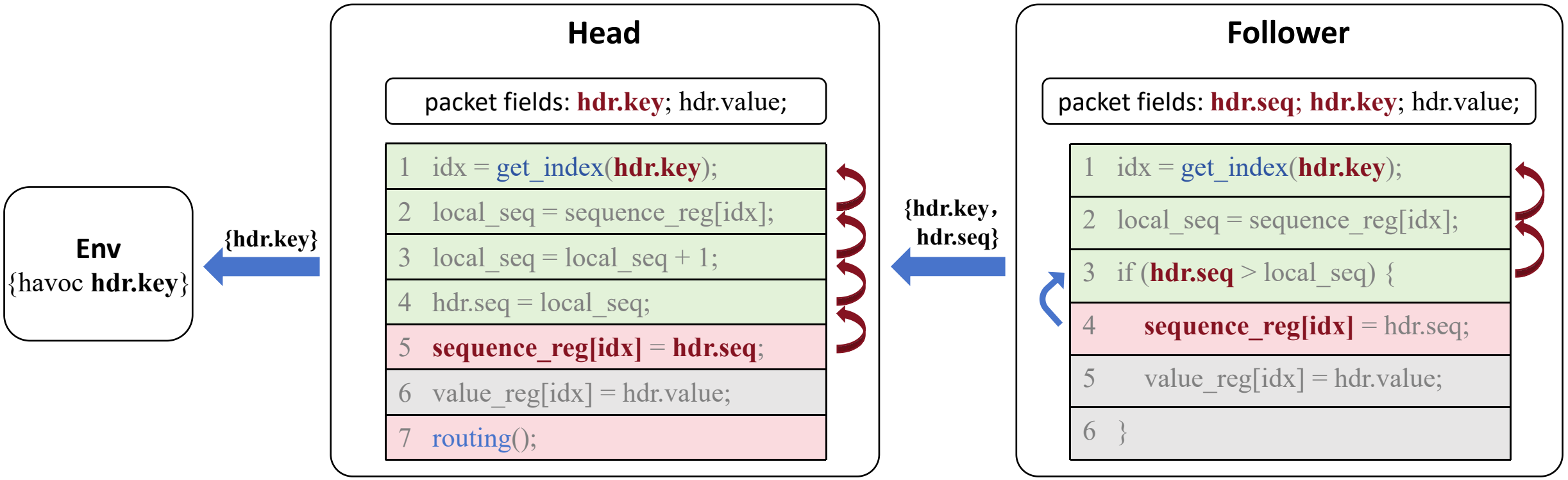
⑤ hdr.value and value_reg never reach the assertion — prune them



assert (**Head.sequence_reg[idx] >= Follower.sequence_reg[idx]**)

Intent-Driven Pruning

⑥ Infer inputs: Env only havocs hdr.key — hdr.seq is generated

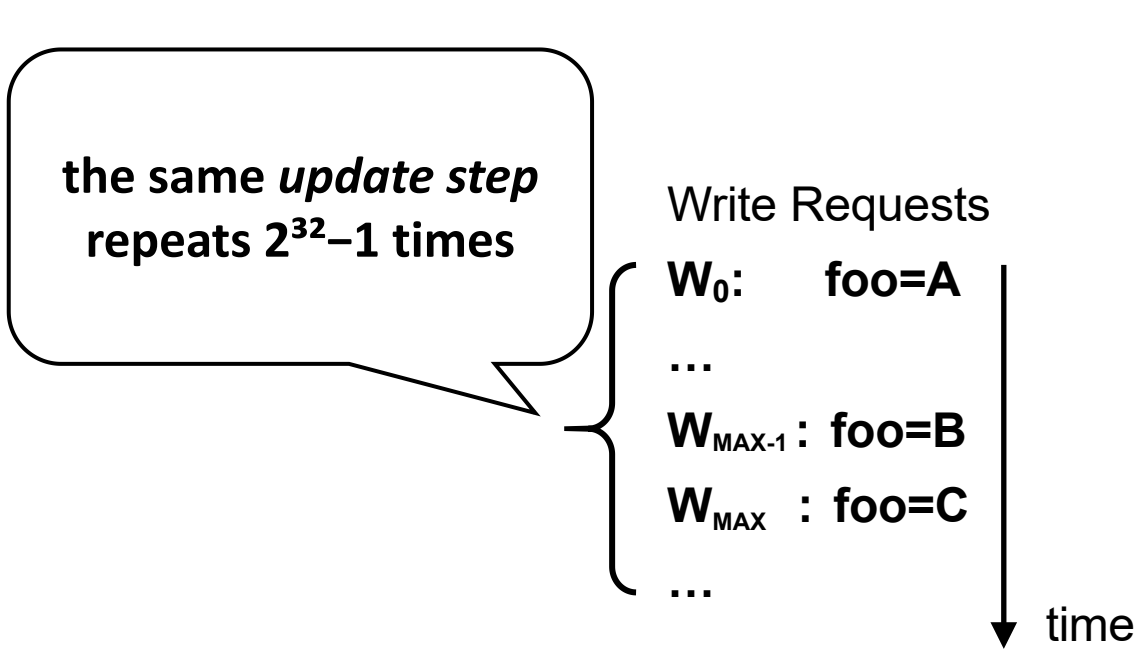


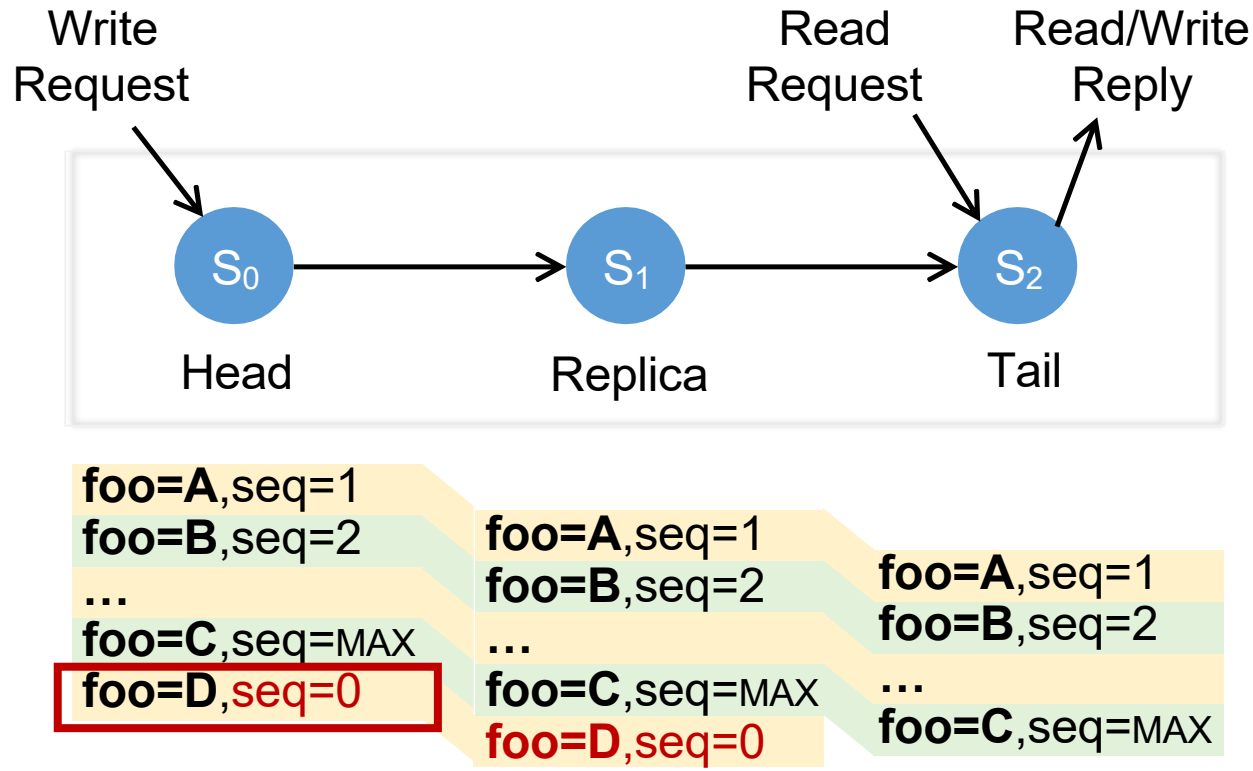
assert (**Head.sequence_reg[idx] >= Follower.sequence_reg[idx]**)

Challenge #3: How to find **bugs too deep to explore**

Insight: Certify One Round, Skip the Long Trace

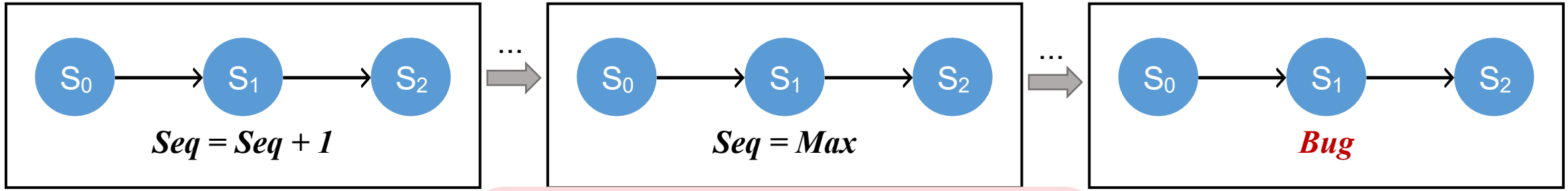
The same update round repeats until the register wraps





Three-Stage Checking

Break one long-trace exploration into three small checks



Step 1:
Check whether one round increments Seq by 1

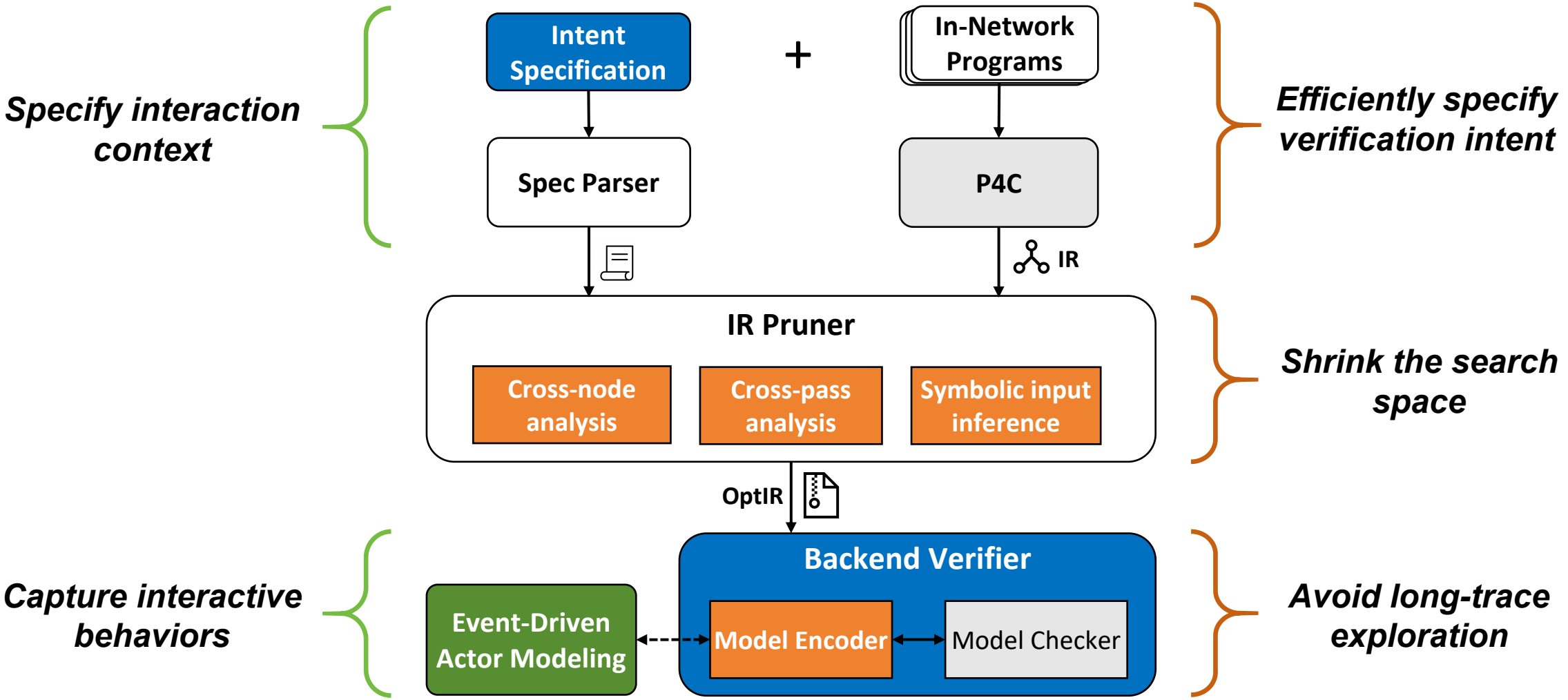
Formal queries and the replayability proof are detailed in the paper

Step 2:
Check whether the bug state is reachable from $Seq = MAX$

Step 3:
Check whether the same round can repeat until $Seq = MAX$

Put It All Together: Procurator

Efficient verification framework for distributed in-network programs



Evaluation

Can it find bugs? At what cost?

Environment

A server with two Intel Xeon Silver 4210R processors and 128 GB of memory

Dataset	Contents	*Purpose*
Dataset 1	12 distributed in-network computing systems	*Bug-detection capability and system overhead*
Dataset 2	5 single-device in-network applications	*Comparison with p4tv*

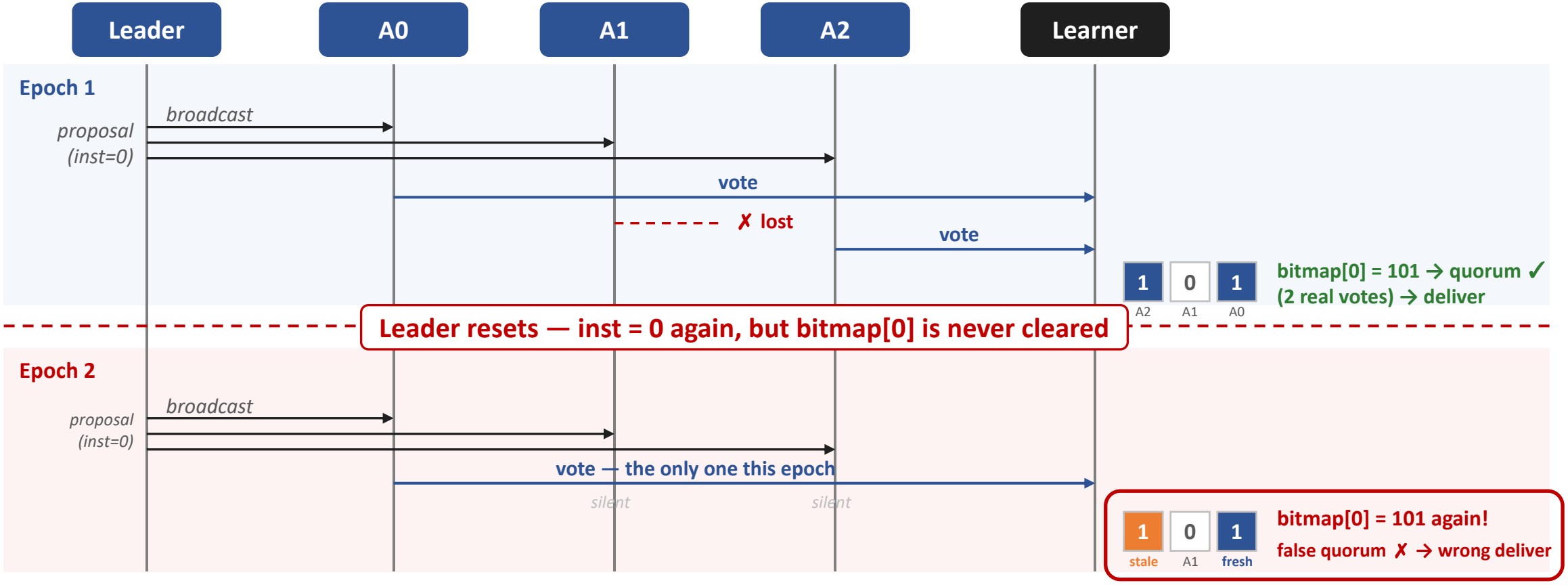
Bug Finding

Procurator Found 28 Bugs Across 12 Distributed Systems

Type	Count	Description	Example
Implementation	15	In-node processing logic inconsistent with the specification	***P4xos*** *misses validation on the instance index, enabling out-of-policy access*
Functional	5	End-to-end behavior deviates from the specification	***P4NIS*** *encapsulation uses inner fields, violating tunnel isolation*
Interleaving	4	Incorrect state under specific event orderings	***P4xos*** *reuses stale quorum history, enabling a false quorum conclusion*
Wraparound	4	Fixed-width register or counter crosses a boundary	***NetChain*** *sequence-number wraparound breaks monotonicity across chained replicas, violating consistency*

Case Study: A False Quorum in P4xos

The learner keeps one vote bitmap per Paxos instance — and never clears it across epochs



The bug hides in one specific interleaving of reset and votes. Procurator finds it automatically and returns the exact violating trace.

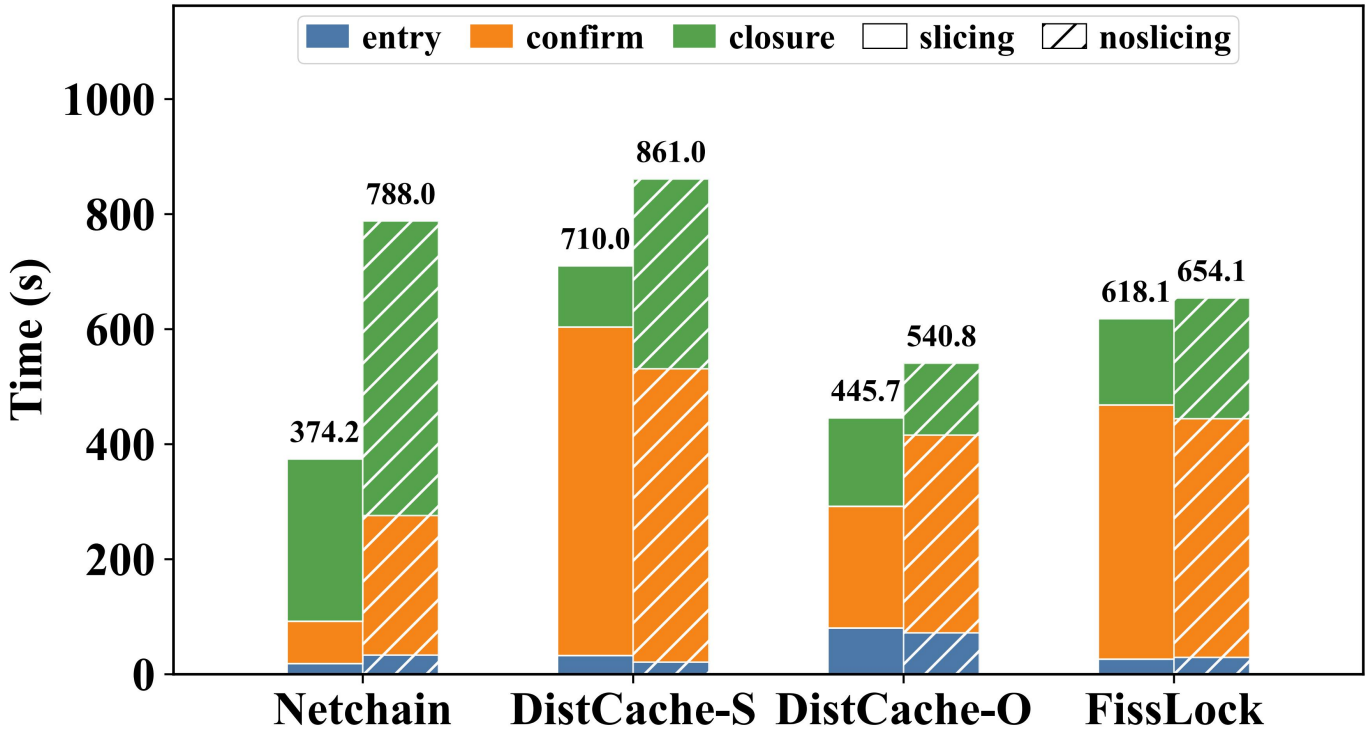
Compare with p4tv

Up to 9.1× faster and correct where p4tv fails

Program	Property	Procurator		p4tv	
		Res	Time	Res	Time
P4NIS.p4	Safe: Tun. Egress	√	*15.3s*	√	*33.4s*
	Safe: No Drop	√	*26.3s*	√	*72.3s*
	Bug: Tun. Leak	√	*26.2s*	*T.O.*	-
acceptor.p4	Safe: Round Num	√	*55.8s*	*Err*	-
	Bug: Round Num	√	*26.0s*	√	*237.7s*
learner.p4	Bug: Quorum	√	*29.0s*	√	*173.7s*
spec_ops.p4	Safe: Recirc.	√	*<1s*	×	*29.2s*
partition.p4	Bug: Wraparound	√	*57.7s*	*T.O.*	-

Three-Stage Checking Terminates All Wraparound Tasks

With schedule-replay acceleration, all four tasks terminate; slicing further cuts the time.



Conclusion

- Models each P4 pipeline as a reactive actor, capturing both intra- and inter-device interactions missed by existing P4 verifiers.

- Prunes assertion-irrelevant code and decomposes long counter-examples; with staged checking, all four wraparound tasks terminate.

- Found 28 bugs across 12 distributed in-network computing systems.



SIGCOMM 2026
DENVER

Q&A

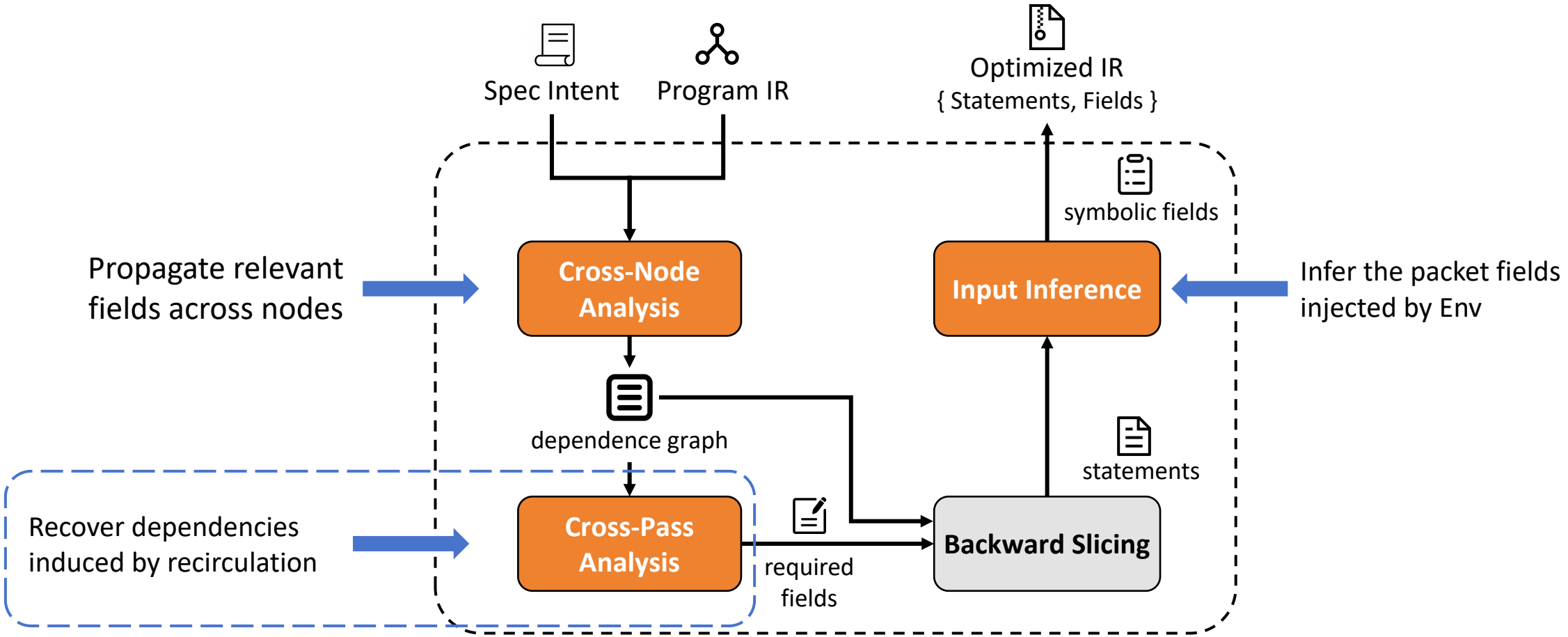
Distributed in-network programs, verified with explicit interaction semantics.

Mingyuan Song | mingyuan.song.xmu@gmail.com
System and Networking Group | https://sngroup.org.cn/

QR Code

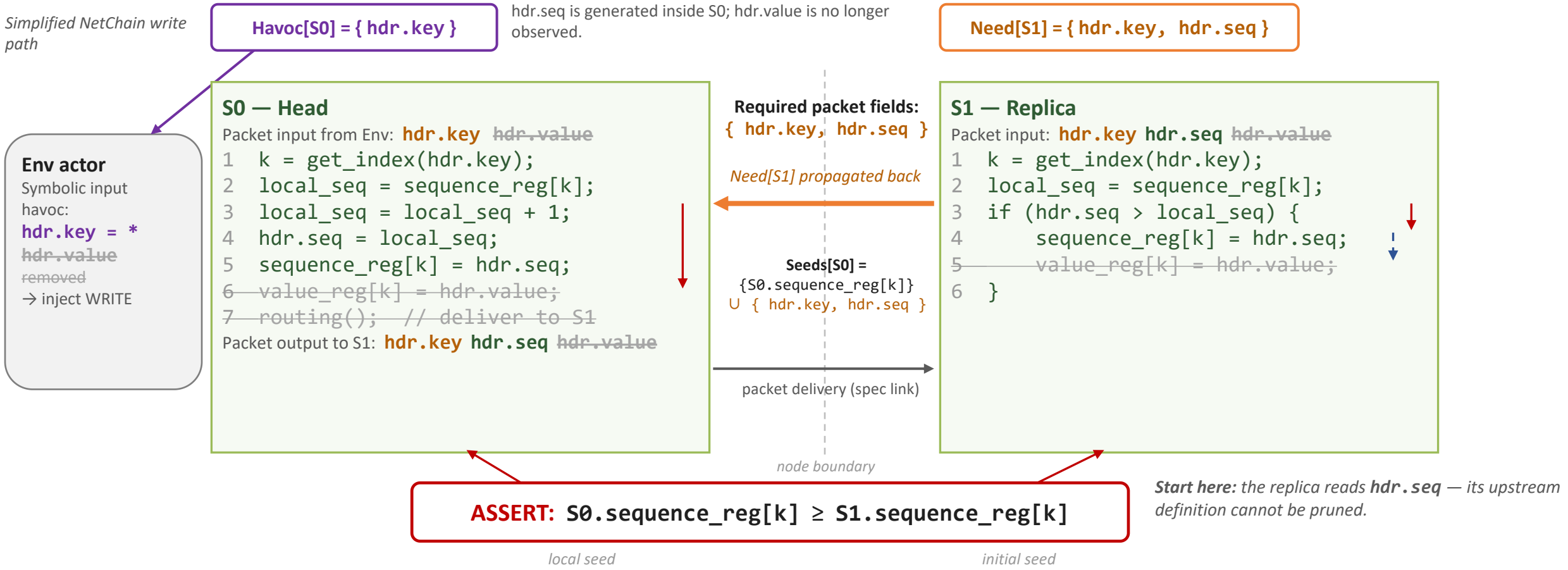
IR Pruner

Key Insight: prune assertion-irrelevant statements and inputs — without losing bugs



IR Pruner

Assertion seed → local dependencies → cross-node need → Env havoc



- solid red — data dependency
- dashed blue — control dependency
- thick orange — cross-node Need propagation
- struck-through gray — pruned
- purple capsule — inferred Havoc input
- green code — kept after slicing

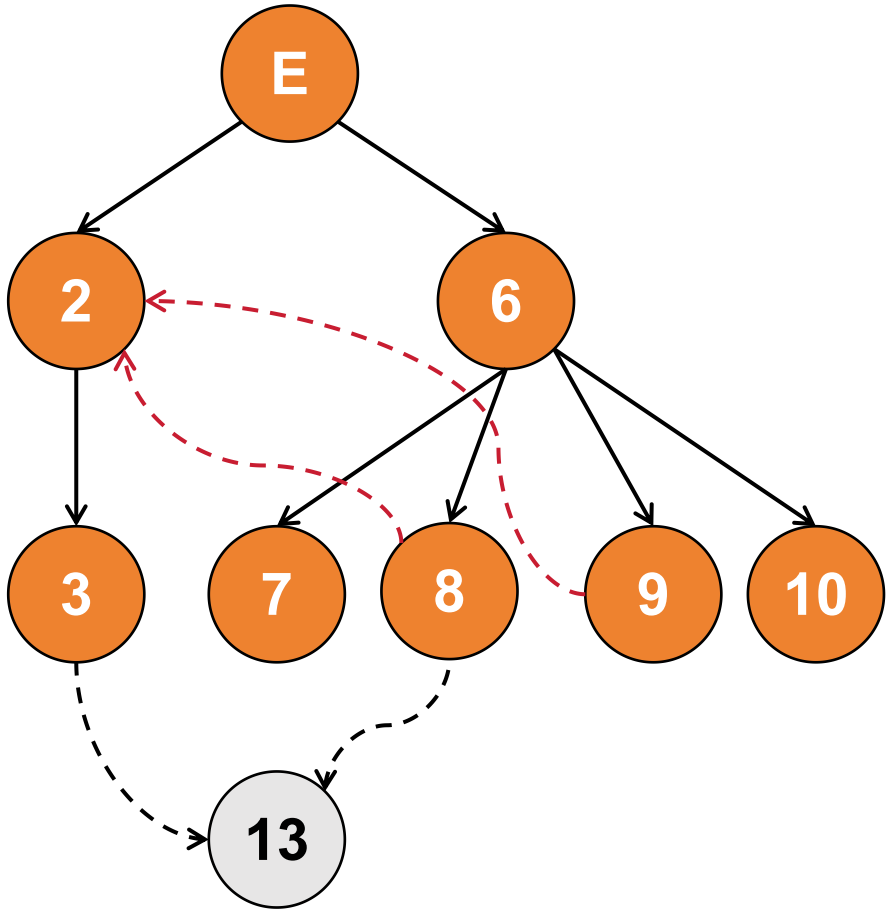
Downstream need for `hdr.seq` preserves its upstream definition; only `hdr.key` stays symbolic at the boundary.

Cross-Pass Analysis

Recover interaction-induced dependencies across pipeline passes

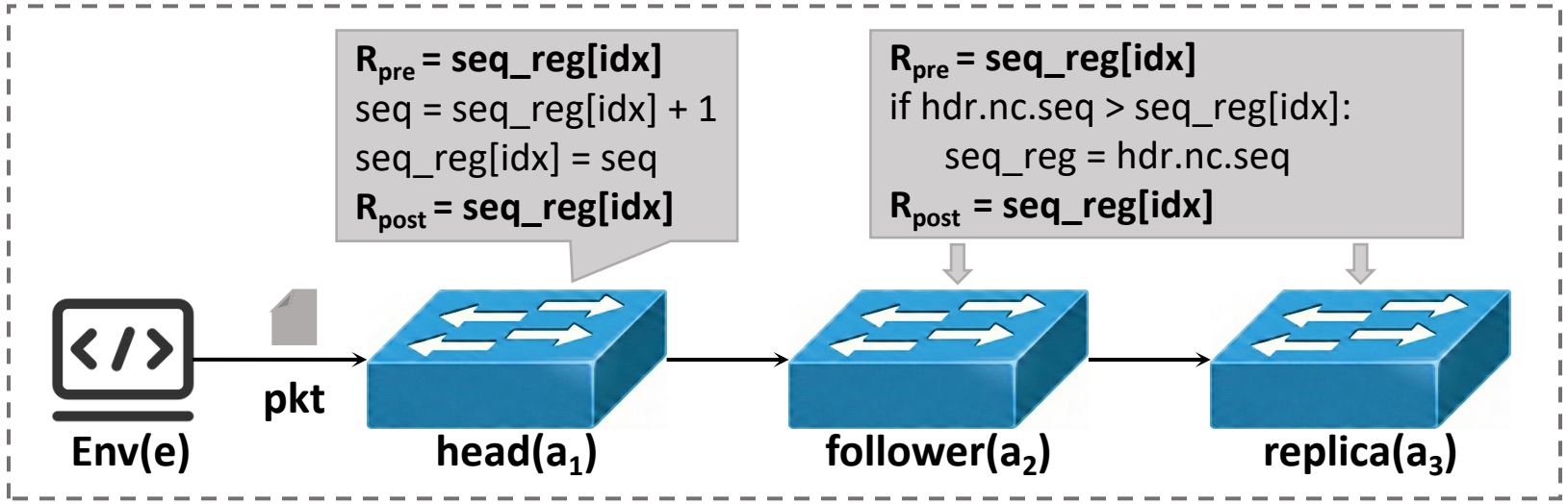
```
1  //  fallback phase: forward to the backup replica
2  if (meta.failover == 1 && hdr.dst_ip == BACKUP_IP) {
3      meta.egress_port = BACKUP_UPLINK_PORT;
4  }
5  //  service lookup miss: switch the packet to the backup replica
6  if (!service_table.apply().hit) {
7    fail_count_reg[meta.idx] += 1;
8    meta.failover = 1;
9    hdr.dst_ip = BACKUP_IP;
10   recirculate();
11 }
12 //  once in fallback mode, the packet must finally go to the backup replica
13 assert(meta.failover == 1 ->
14     meta.egress_port == BACKUP_UPLINK_PORT);
```

Service lookup with recirculation



Stage 1: Entry Check

Extract an update schedule σ (the order nodes execute in one round) and its step



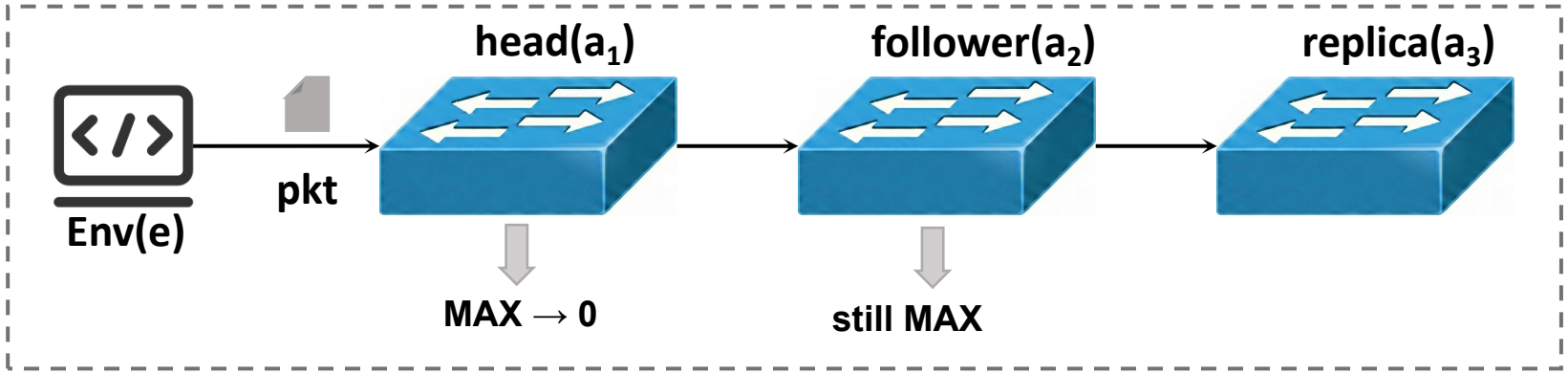
Query: $\bigwedge_{s \in \{a1, a2, a3\}} s.Rpre == s.R_{post}$

CounterExample Trace

schedule σ: | e | a_1 | a_2 | a_3 | step: step = R_{post} - R_{pre}

Stage 2: Near-Wrap Check

From a near-wrap state, search for a short suffix that triggers the violation



Query: $a_1.seq_reg[i] \geq a_2.seq_reg[i] \geq a_3.seq_reg[i]$

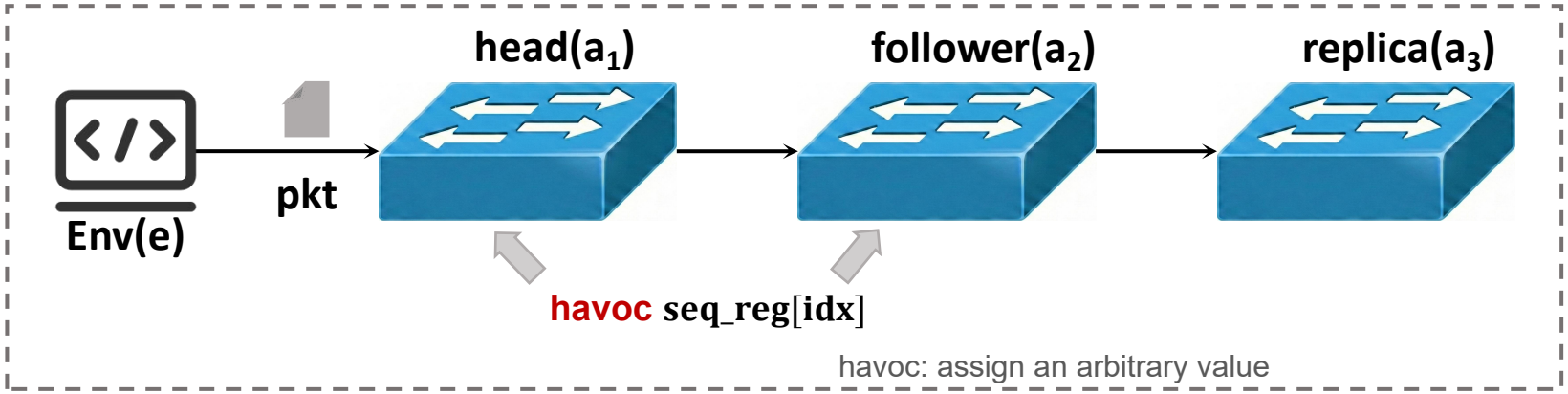
i.e., does the wraparound at the head break the ordering across replicas?

CounterExample Trace

suffix: τ_{sf}

Stage 3: Closure Check

Certify that the schedule can replay N times to reach the near-wrap state



Query: $\bigwedge_{s \in \{a1, a2\}} s.Rpre + step == s.Rpost$

Safe

Construct counter-example: $\tau_{pre} \circ \tau_{sf}$

counter-example = prefix replayed N times + suffix