Relational NetKAT

Han Xu

Princeton University

May, 2025

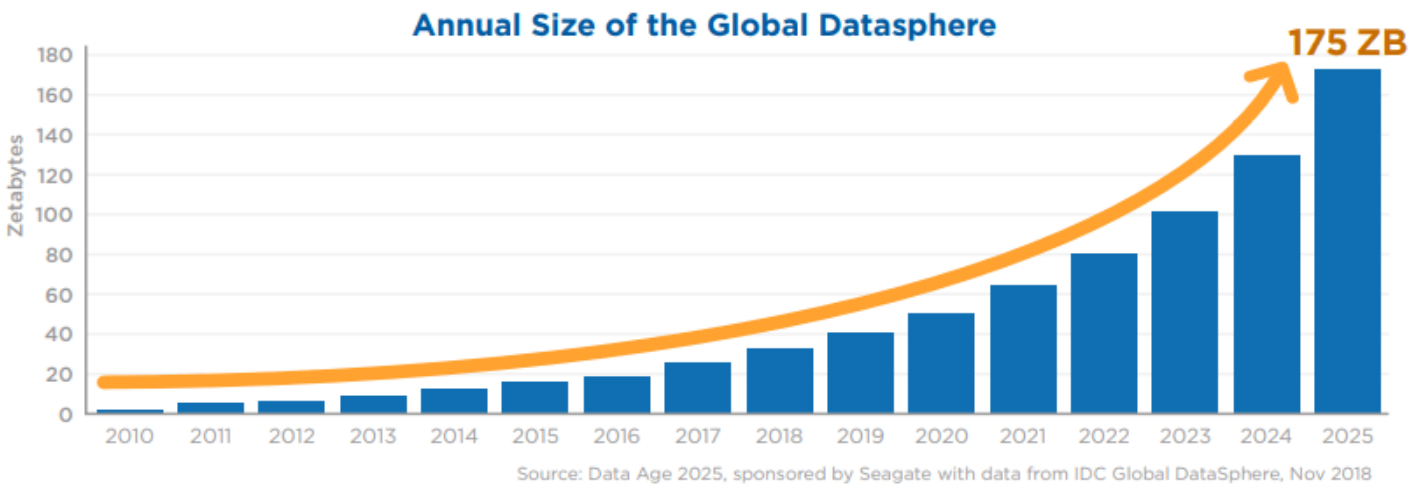
Background

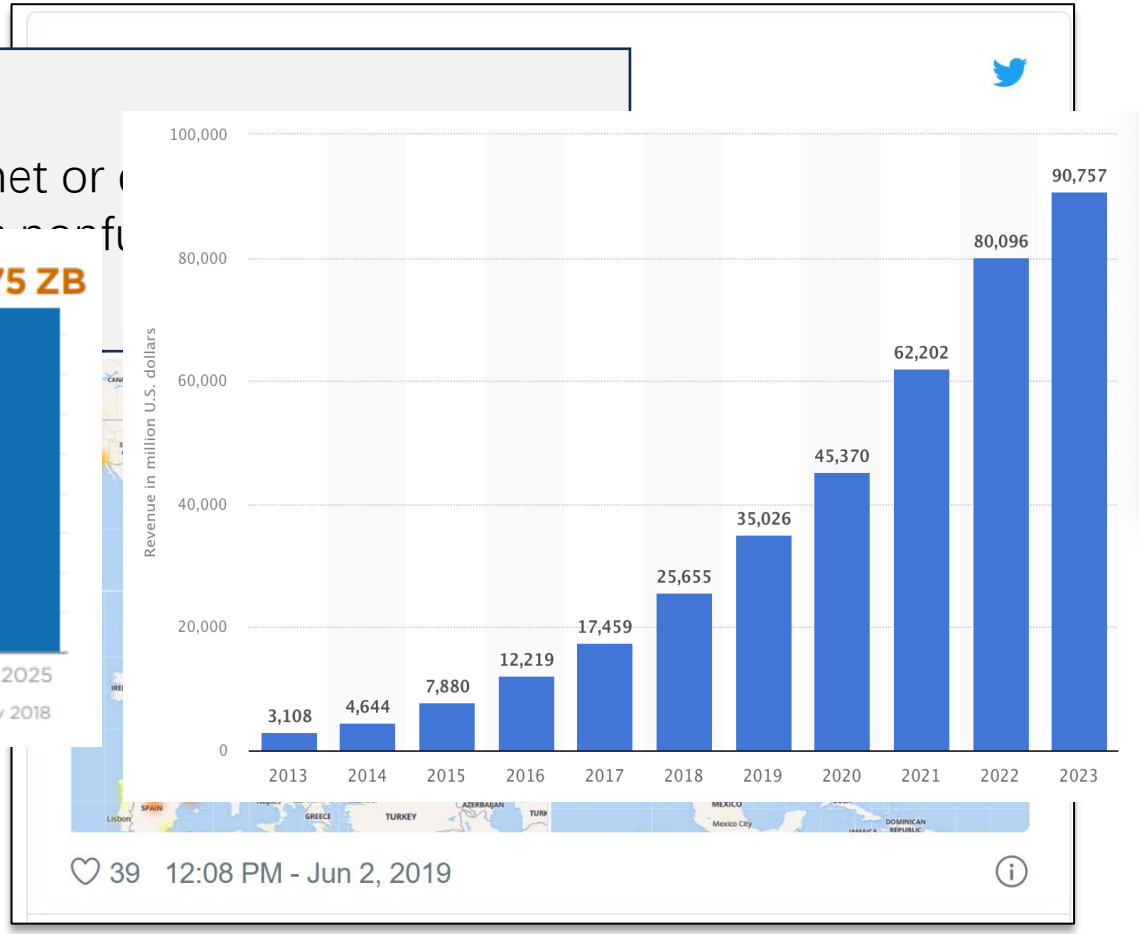
- Modern networks are very large.

Rodgers Canada Outage (2022)

- 13 million customers (⅓ of Canada) without Internet or
- Nationwide loss of debit services, and many ATMs nonfu

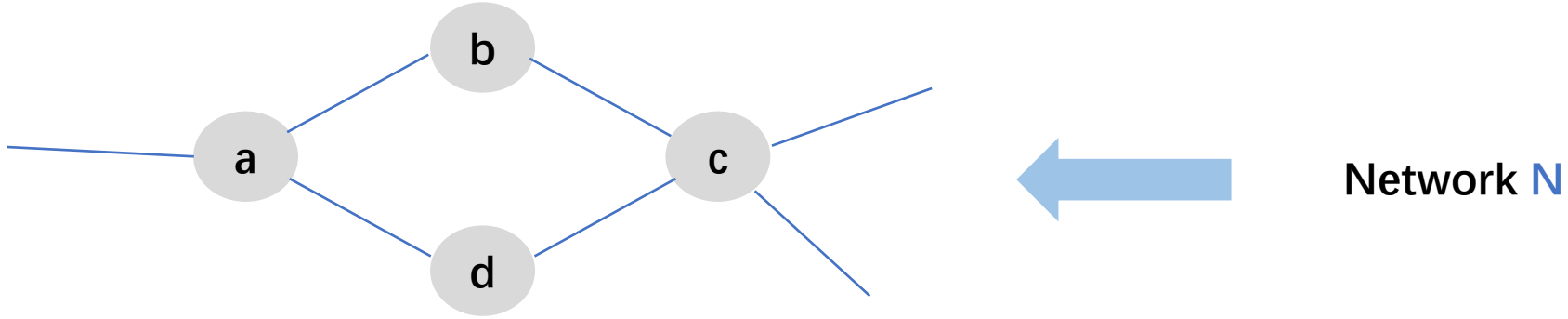
Compute service unavailable to cstomers in some parts of Europe for more than two hours.





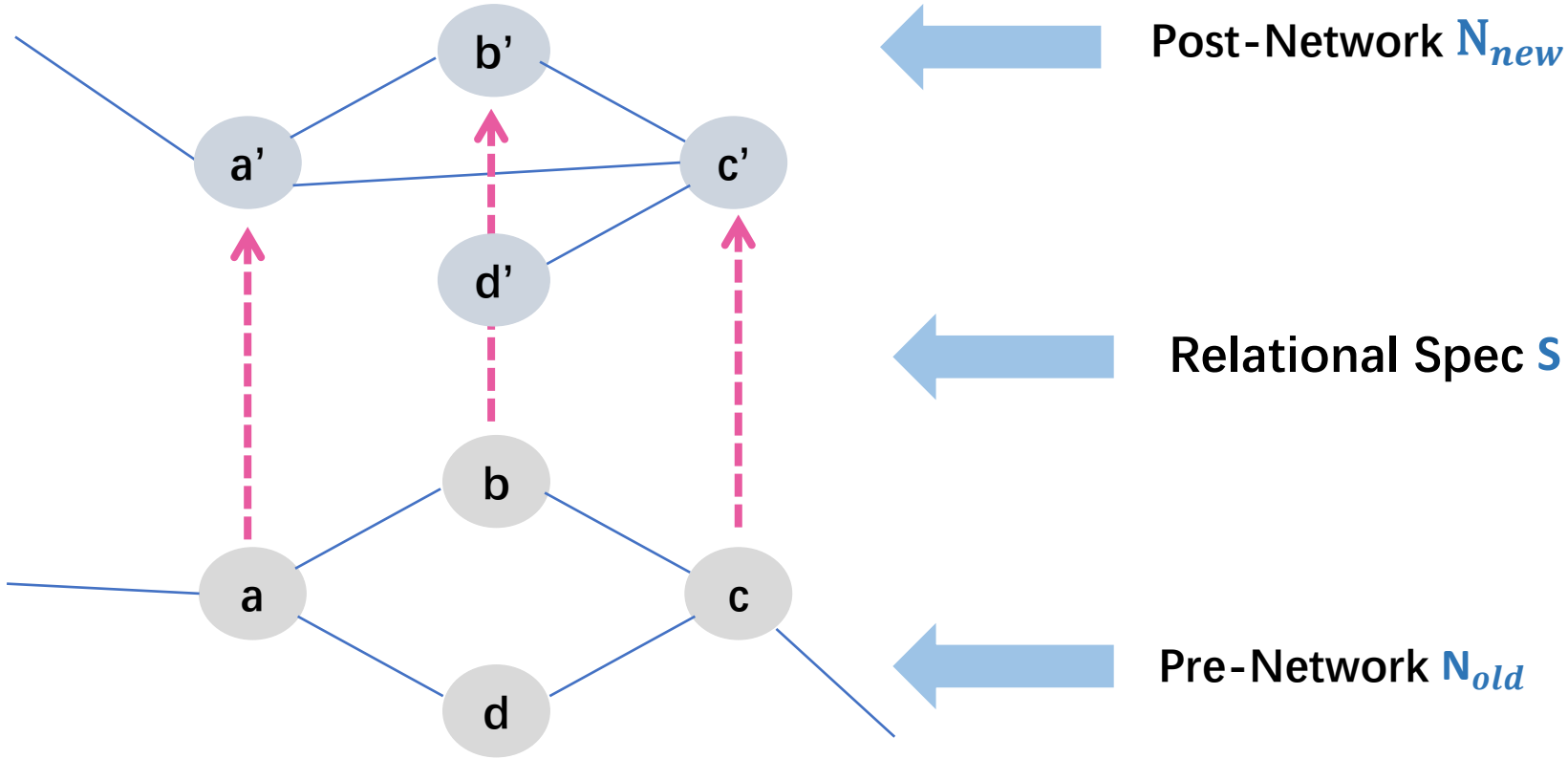
Traditional Network Verification

- Given a static snapshot of Network N, we want to test Property P that:
  - Reachability: a can reaches c
  - Loop Detection: packets from a will not reach a
  - …



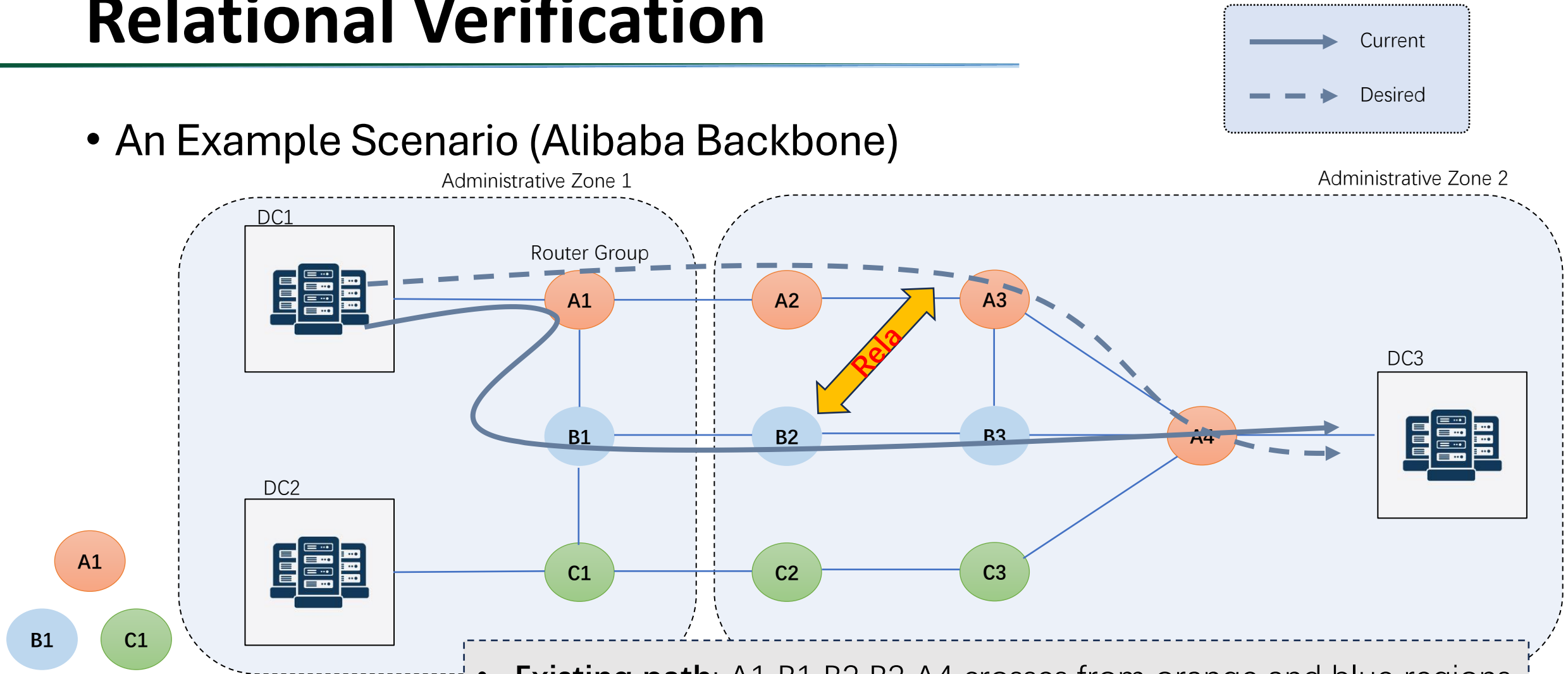
Relational Network Verification

- Given a Pre-Network $\mathbf{N}_{old}$, Post-Network $\mathbf{N}_{new}$ and Relational Spec **S**
  - Is $\mathbf{N}_{new}$ exactly updated from $\mathbf{N}_{old}$ under **S** ?



Relational Verification

- An Example Scenario (Alibaba Backbone)

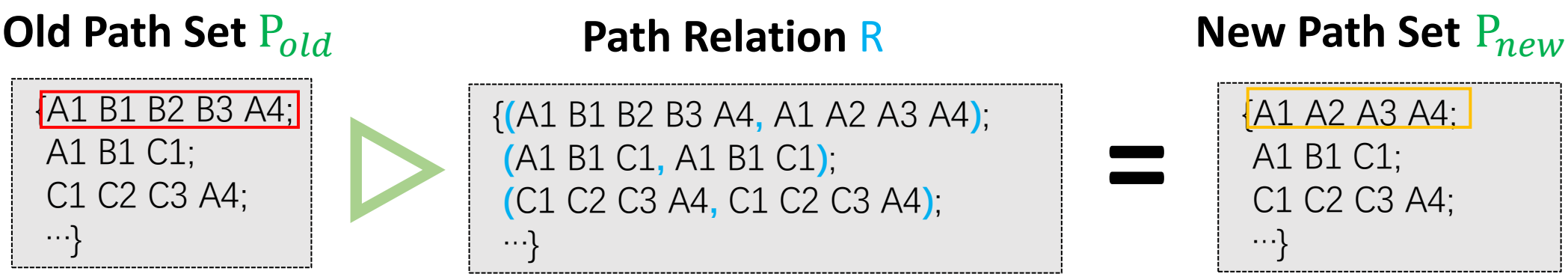


Device group (different color is different geographic region)

- **Existing path**: A1 B1 B2 B3 A4 crosses from orange and blue regions
- **Desired path**: A1 A2 A3 A4 stays in orange region

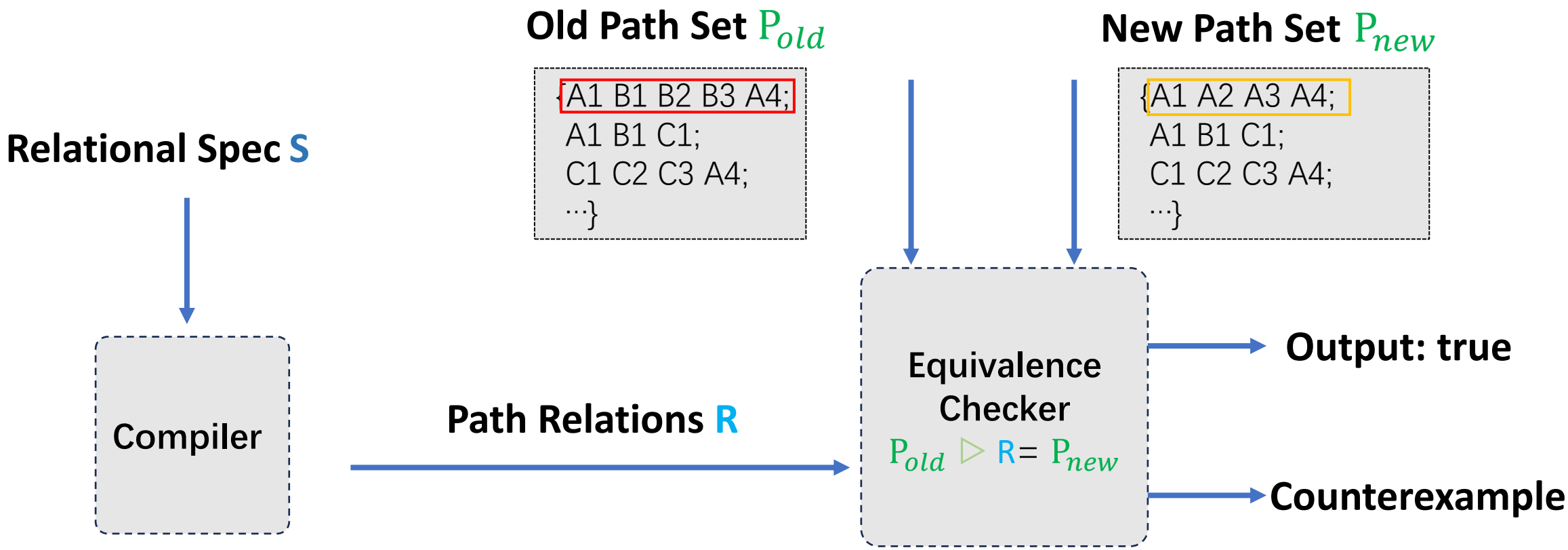
Rela Framework

- Rela models Networks N as set of paths P.
  - Path relation R is modeled as set of path pairs.
  - We then verify $\mathrm{P}_{old} \triangleright \mathrm{R} = \mathrm{P}_{new}$.



Rela Framework

- Networks N as input packets with their set of paths P.



Problem 1: Packet Changes

- Path Changes fail to capture many useful changes.
  - Tunneling: No path changes reflected, but packet changes along the path.

Old Path Set P_{old}

{A1 B1 B2 B3 A4;
A1 B1 C1;
C1 C2 C3 A4;
…}

Path Relation R

{(A1 B1 B2 B3 A4, A1 A2 A3 A4);
(A1 B1 C1, A1 B1 C1);
(C1 C2 C3 A4, C1 C2 C3 A4);
…}

New Path Set P_{new}

{A1 A2 A3 A4;
A1 B1 C1;
C1 C2 C3 A4;
…}

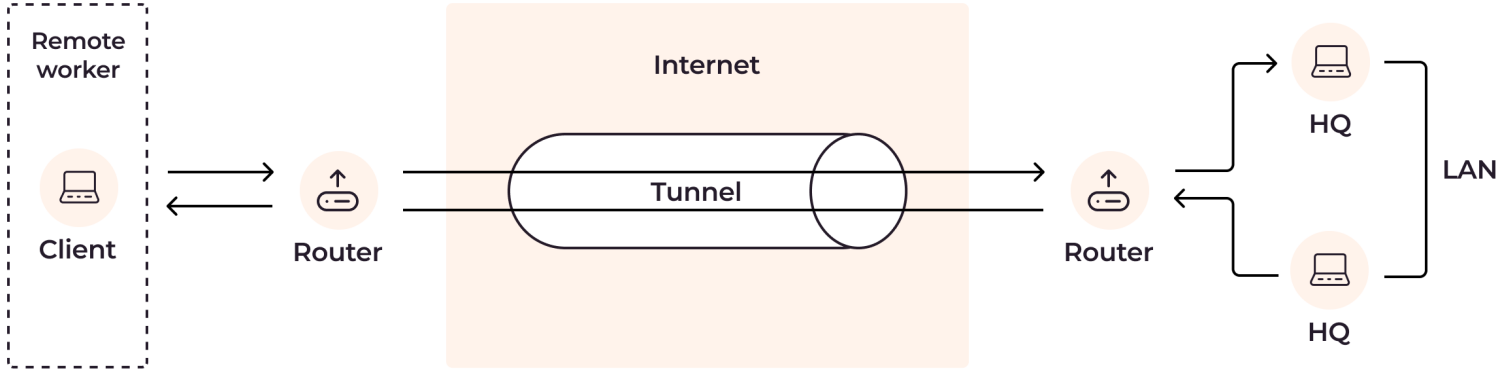
Problem 1: Packet Changes

- Sequence of location vs Sequence of packets.
  - We need to model traces and trace changes!

Paths:

Traces:

C1	R1	T1 T2 ⋯ Tn	R2	C2
pk1’	pk1’’	pk1 pk2 ⋯ pkn	pk2’’	pk2’



Example packets:

{loc = C1;
typ = SSH;
src = 10.0.0.8;
dst = 10.0.0.9;
content = ⋯}

Paths:

Traces:

C1	R1	T1 T2 ⋯ Tn	R2	C2
pk1’	pk1’’	pk1* pk2* ⋯ pkn*	pk2’’	pk2’

Problem 2: Sample-Based Checking

- Networks P is collected through sampling.
  - Risk of missing
  - We need symbolic checking over all paths!

Old Path Set P_{old}

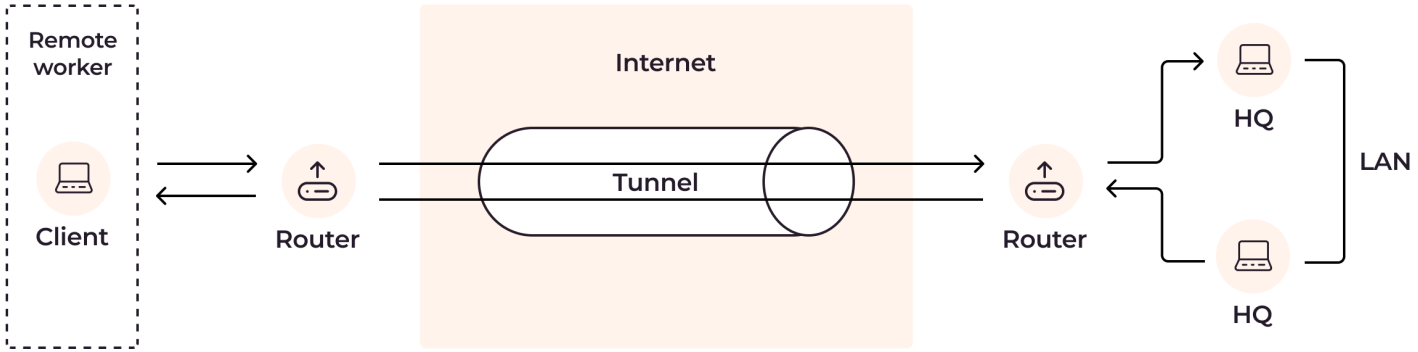
{A1 B1 B2 B3 A4;
A1 B1 C1;
C1 C2 C3 A4;
…}

New Path Set P_{new}

{A1 A2 A3 A4;
A1 B1 C1;
C1 C2 C3 A4;
…}

Observation

- Problems are mainly related with P.
  - We need a language to describe traces and trace changes!



Old Path Set P_{old}

{A1 B1 B2 B3 A4;
A1 B1 C1;
C1 C2 C3 A4;
…}

New Path Set P_{new}

{A1 A2 A3 A4;
A1 B1 C1;
C1 C2 C3 A4;
…}

Our Solution in 3 Steps

- Relational NetKAT language:
  - Step 1: Lift Path Sets P to Trace Sets (NetKAT) K.

Paths P_{old}, P_{new}

{A1 B1 B2 B3 A4;
A1 B1 C1;
C1 C2 C3 A4;
…}

Sequence of location

Traces K_{old}, K_{new}

{pk1 pk1' pk2' pk3' pk4;
pk1 pk1' pk1'';
pk1'' pk2'' pk3'' pk4;
…}

Sequence of packets

Our Solution in 3 Steps

- Relational NetKAT language:
  - Step 1: Lift Path Sets P to Trace Sets (NetKAT) K.
  - Step 2: Lift Path Relation R to Trace (NetKAT) Relation R.

Rela

Path Relations R {(A1 B1 B2 B3 A4, A1 A2 A3 A4); ⋯;} **Set of path pairs**

Our

NetKAT Relations R {(pk1 pk1' pk2' pk3' pk4, pk1 pk2 pk3 pk4); ⋯;} **Set of trace pairs**

Our Solution in 3 Steps

- Relational NetKAT language:
  - Step 1: Lift Path Sets P to Trace Sets (NetKAT) K.
  - Step 2: Lift Path Relation R to Trace (NetKAT) Relation R.
  - Step 3: Compile and Check $K_{old} \triangleright R = K_{new}$!

Our Solution in 3 Steps

- Relational NetKAT language:
  - Step 1: Lift Path Sets P to Trace Sets (NetKAT) K.

Paths P_{old}, P_{new}

{A1 B1 B2 B3 A4;
A1 B1 C1;
C1 C2 C3 A4;
…}

Sequence of location

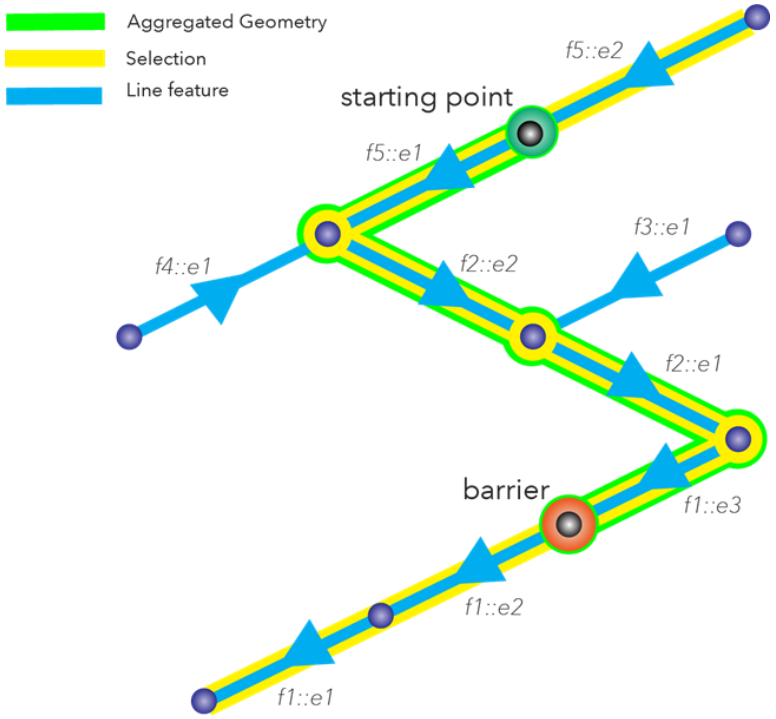
Traces K_{old}, K_{new}

{pk1 pk1' pk2' pk3' pk4;
pk1 pk1' pk1'';
pk1'' pk2'' pk3'' pk4;
…}

Sequence of packets

NetKAT Introduction

- NetKAT: a formal system and language for network verification.
  - Traces pk1 pk2 pk3 … pkn reflects history of packets.
  - NetKAT: A simple regular language representing set of traces.

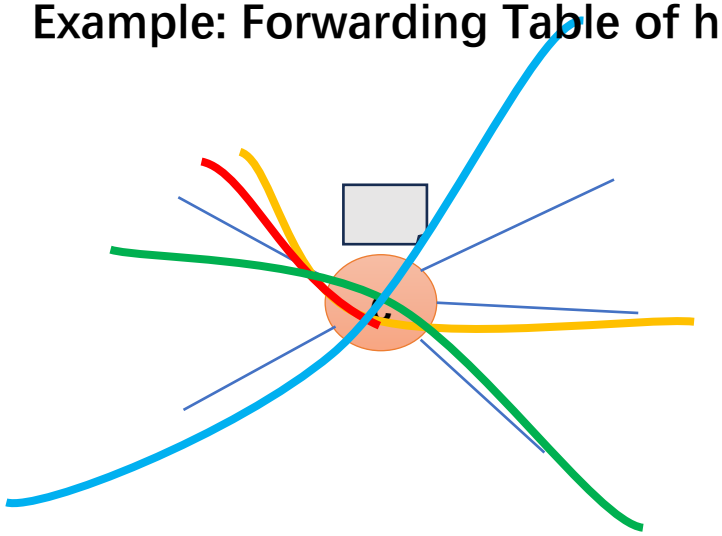


Policy p, q, r ::=
| 0 // drop all packet
| 1 // accept all packets
| f = v/ f != v // filter f
| f <- v // update field f to v
| p · q // do p then q
| p + q // do p and q in parallel
| p* // do p zero or more times
| Dup // Record the current packet to traces

NetKAT Introduction

- NetKAT: a formal system and language for network verification.

Example: Forwarding Table of hop C



Header	Action
src = 10.0.0.1	drop
src = 10.0.0.2	forward
src = 10.0.0.3	forward when dst = 10.0.0.5
src = 10.0.0.4	assign dst <- 10.0.0.5

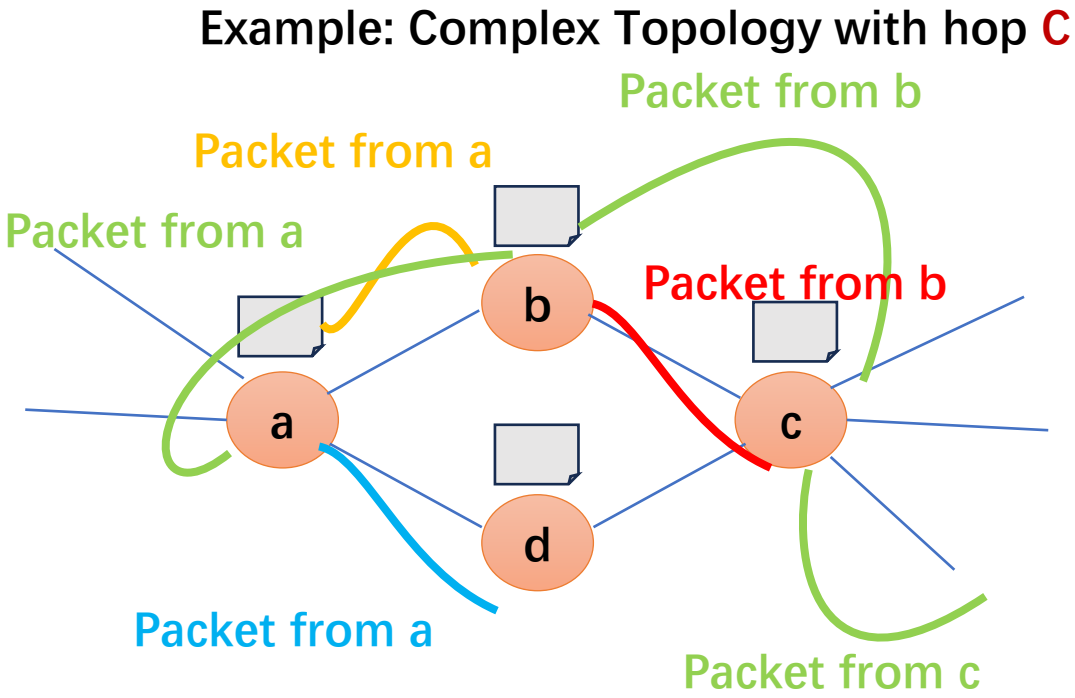
- Encoding Policy
- Encoding Topology

Policy p,q,r ::=

```
  0              // drop all packet        Traces: {}
| 1              // accept all packets     Traces: {(pk,pk)}
| f = v/ f != v // filter f                Traces: {(pk,pk)|pk.f=v}
| f <- v        // update field f to v     Traces: {(pk,pk[f:=v])}
| p · q         // sequential composition
| p + q         // union two forwarding table
```

NetKAT Introduction

- NetKAT: a formal system and language for network verification.

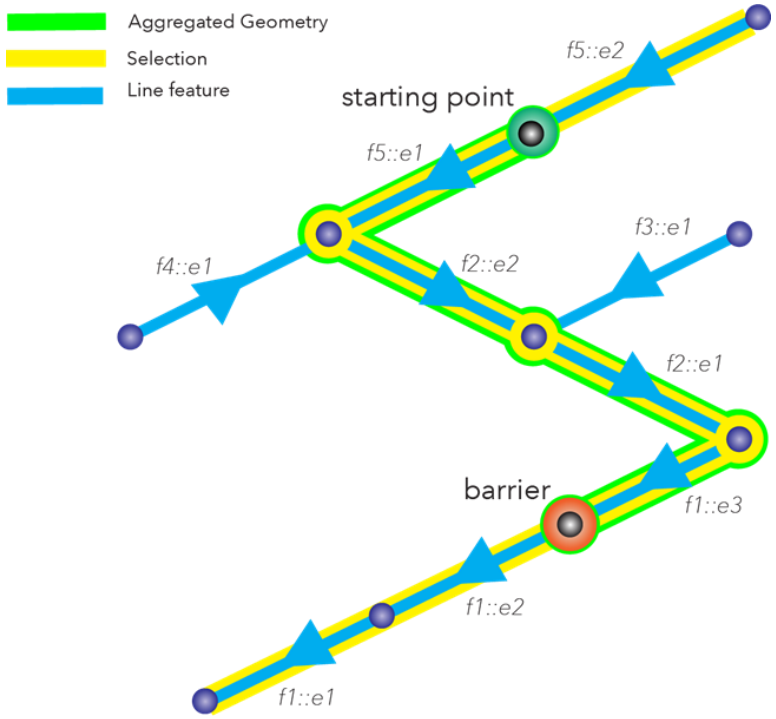



```
Policy p, q, r ::=
| 0              // drop all packet
| 1              // accept all packets
| f = v/ f != v // filter f
| f <- v        // update field f to v
| p · q         // do p then q
| p + q         // do p and q in parallel
| p*            // do p zero or more times
```

- Encoding Policies
- Encoding Topology

NetKAT Introduction

- NetKAT: a formal system and language for network verification.
  - Dup explicitly records the current packet on the fly.




```
Policy p, q, r ::=
 | 0              // drop all packet
 | 1              // accept all packets
 | f = v/ f != v // filter f
 | f <- v        // update field f to v
 | p · q         // do p then q
 | p + q         // do p and q in parallel
 | p*            // do p zero or more times
 | Dup           // Record the current packet to traces
```

Our Solution in 3 Steps

- Relational NetKAT language:
  - Step 1: Lift Path Sets P to Trace Sets (NetKAT) K.
  - Step 2: Lift Path Relation R to Trace (NetKAT) Relation R.

Rela

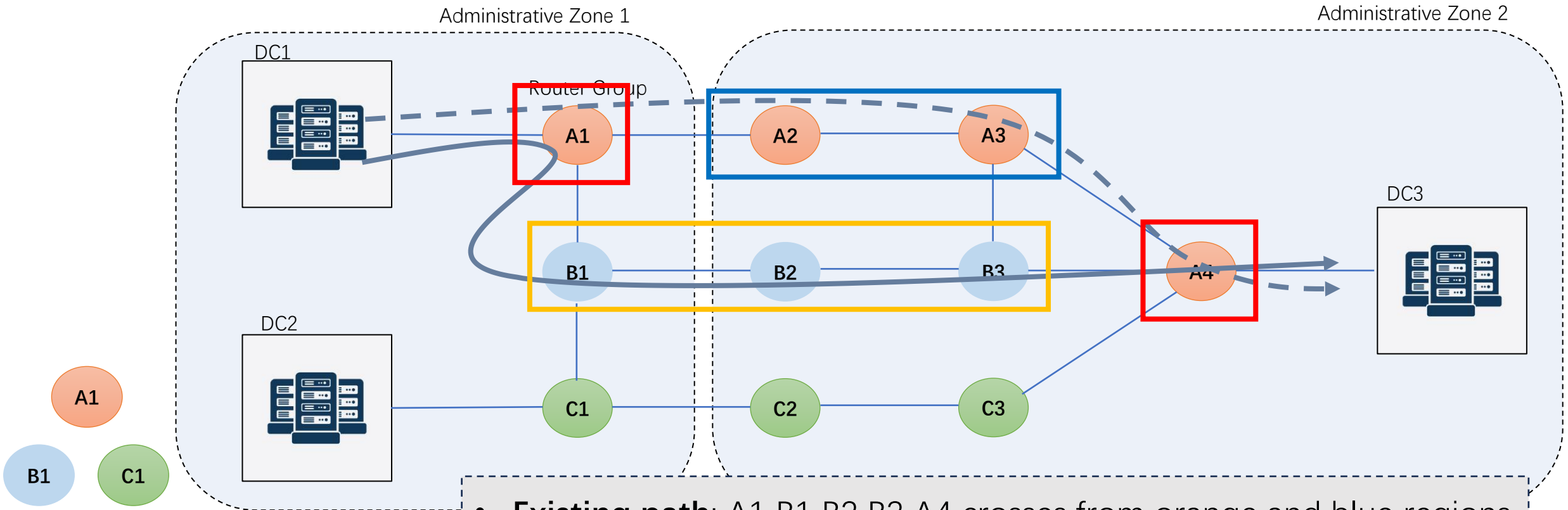
Path Relations R {(A1 B1 B2 B3 A4, A1 A2 A3 A4); ⋯;} **Set of path pairs**

Our

NetKAT Relations R {(pk1 pk1' pk2' pk3' pk4, pk1 pk2 pk3 pk4); ⋯;} **Set of trace pairs**

Path Relations

- Goal: Design R for P ▷ R



Device group (different color is different geographic region)

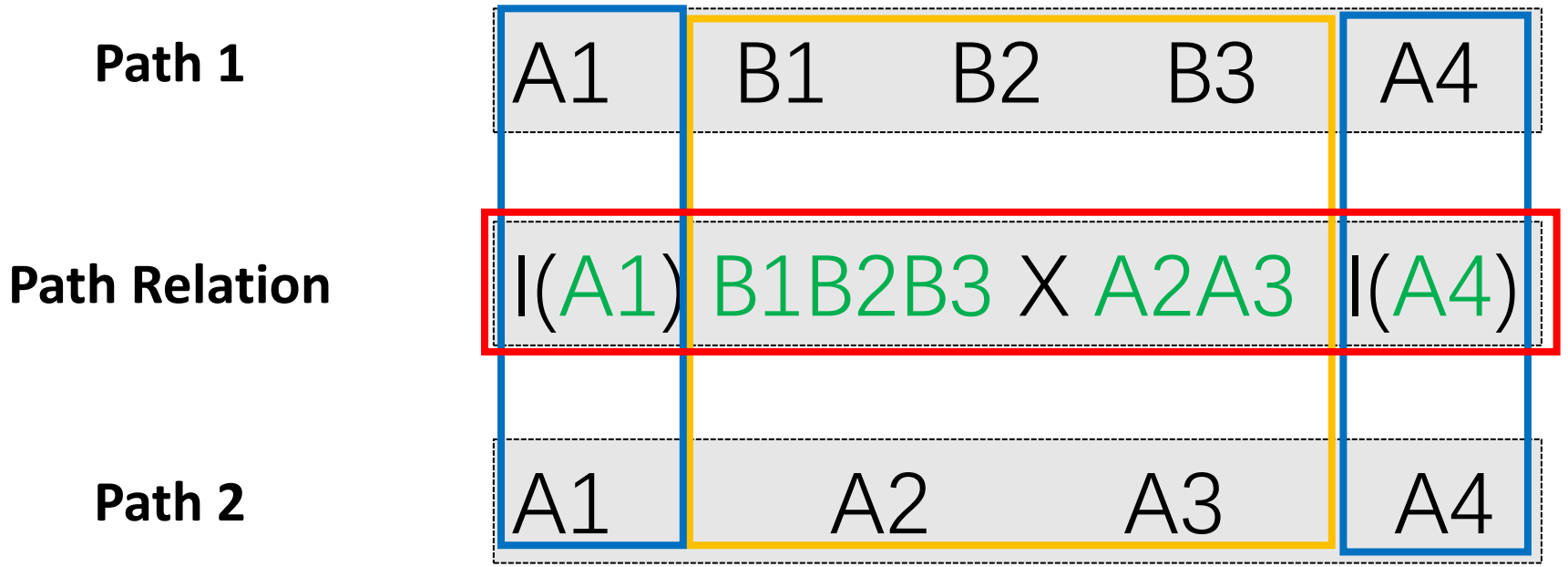
- **Existing path**: A1 B1 B2 B3 A4 crosses from orange and blue regions
- **Desired path**: A1 A2 A3 A4 stays in orange region

Rela

Rela Path Relations

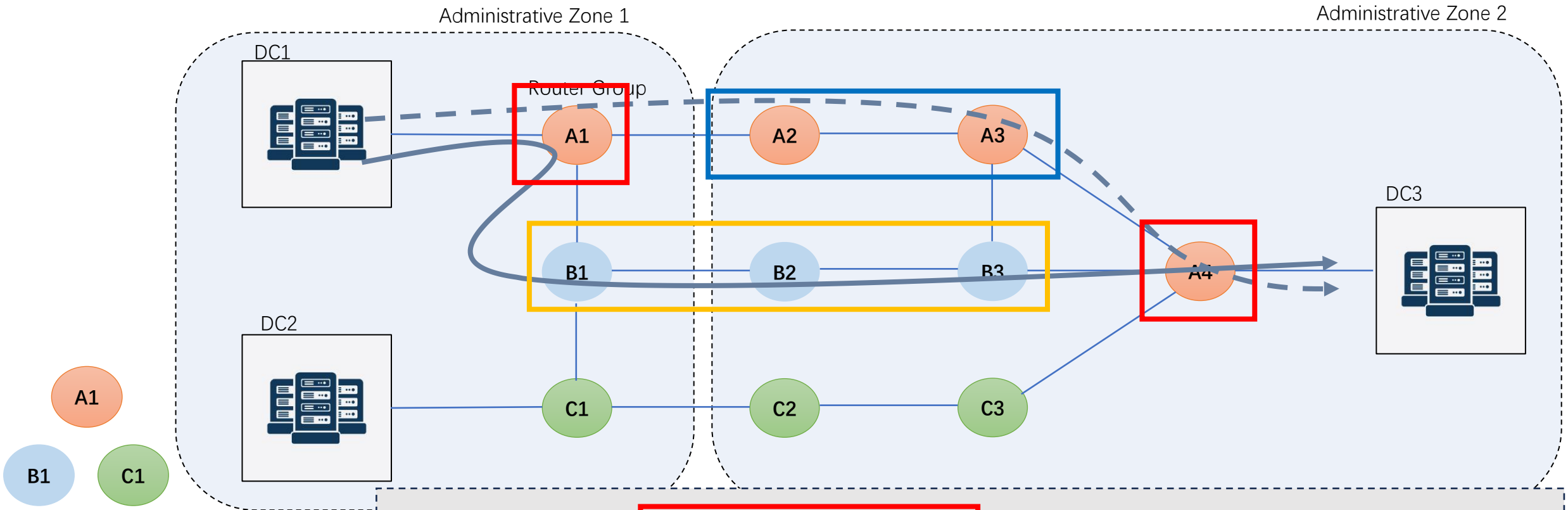
$R = P1 \times P2 \mid I(P) \mid 0 \mid 1 \mid (R1 \mid R2) \mid R1R2 \mid R*$

- Rela components:
  - P1 × P2 : Replace paths of P1 to paths of P2.
  - I(P) : Keeps paths of P unchanged.
  - R1R2 : Concatenate two relations.



Traces Relations

- Goal: Design R for K ▷ R



Device group (different col
different geographic regior

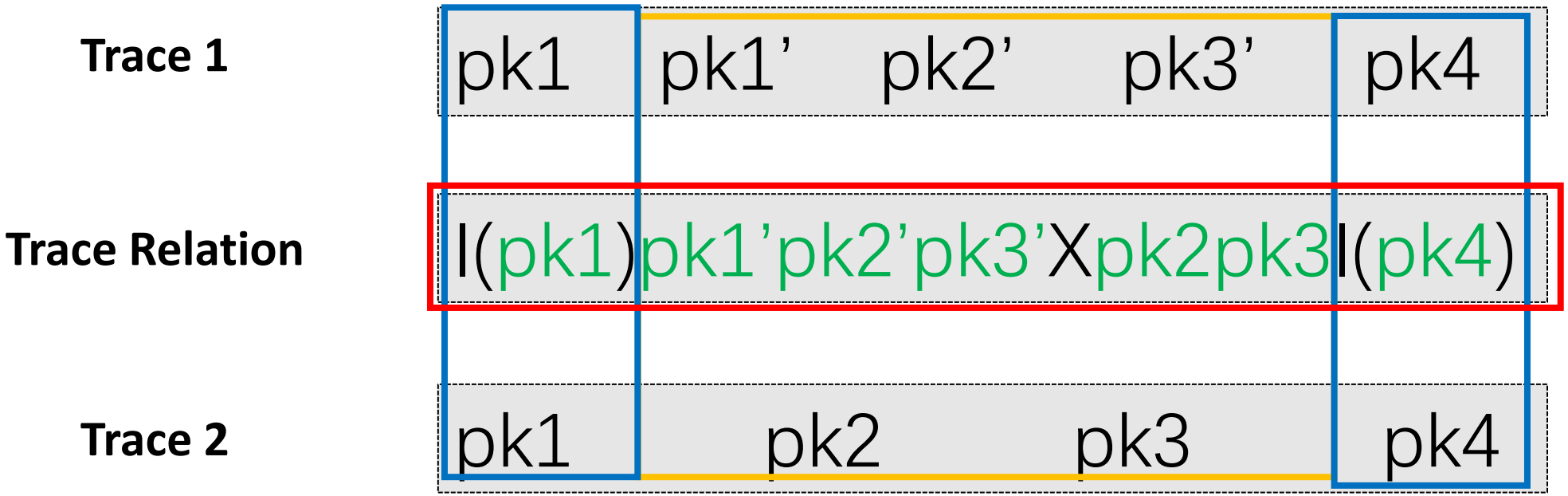
- **Existing trace**: pk1 pk1' pk2' pk3' pk4 crosses from orange and blue regions
- **Desired trace**: pk1 pk2 pk3 pk4 stays in orange region

Relational NetKAT

Trace Relations

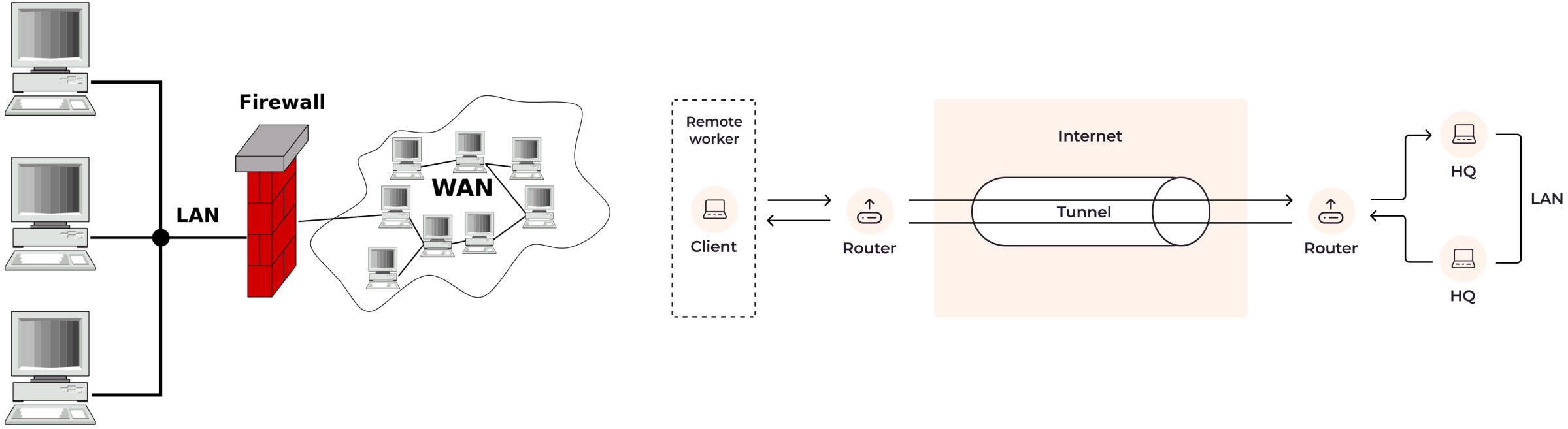
$R = K1 \times K2 \mid Id(K) \mid 0 \mid 1 \mid (R1 \mid R2) \mid R1R2 \mid R* \mid \cdots$

- Relational NetKAT components:
  - $K1 \times K2$: Replace traces of $K1$ to traces of $K2$.
  - $I(K)$: Keeps traces of K unchanged.
  - $R1R2$: Concatenate two relations.



Additional Features

- Filter(pkr) and Apply(pkr, K) for the:
  - Fire Wall: Packets are filtered with no path changes.
  - Tunneling: Packets are changed with possible path changes.

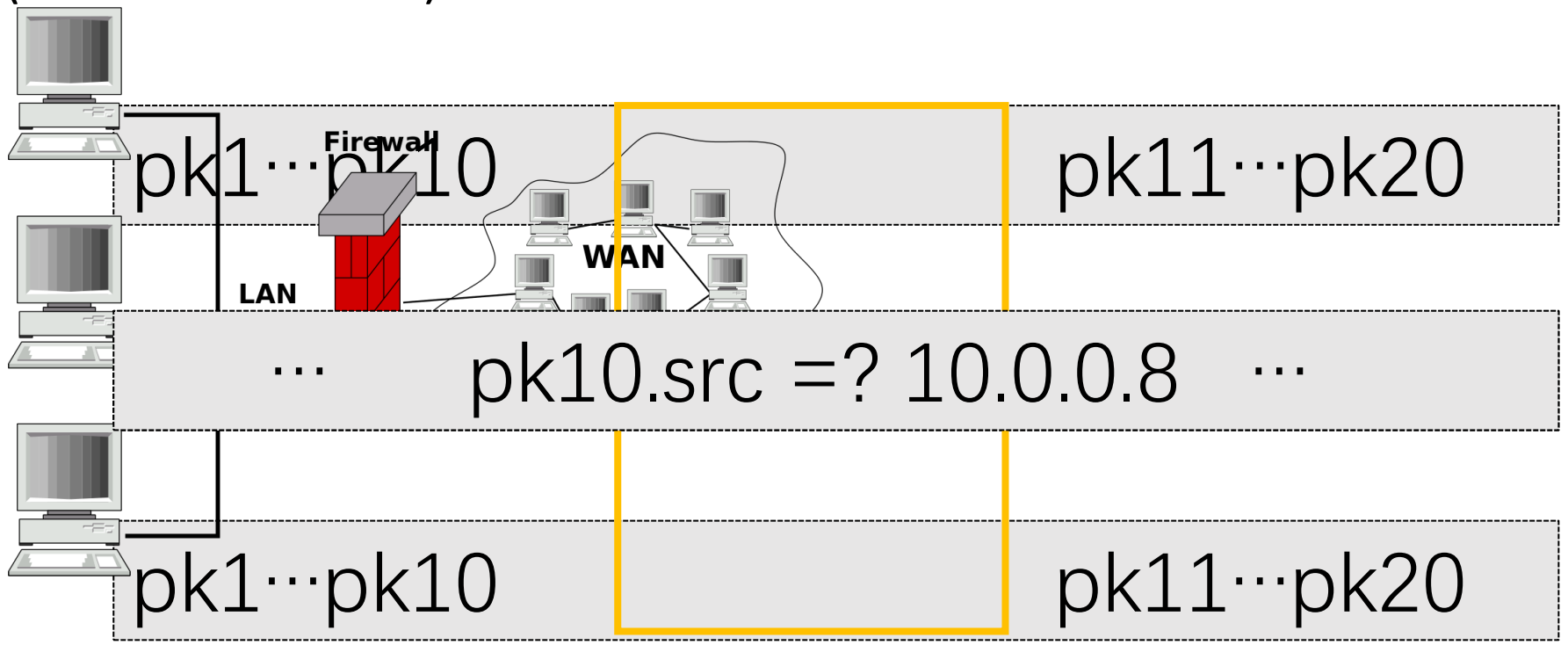


Fire Wall

Additional Relations

R = ⋯| **Filter**(pkr) | **Apply** (pkr, K) | ⋯
pkr = f <- v | f -> v | Id | pk1 × pk2
| pkr1 ∪ pkr2 | pkr1·pkr2 |
| pkr1 ∩ pkr2 | ⋯

- Additional Components:
  - **pkr** : Packet relations.
  - **Filter**(**pkr**) : Filter out packets don't satisfy **pkr**.
  - Let **pkr'** = Id ∩ (1 × src != 10.0.0.8)

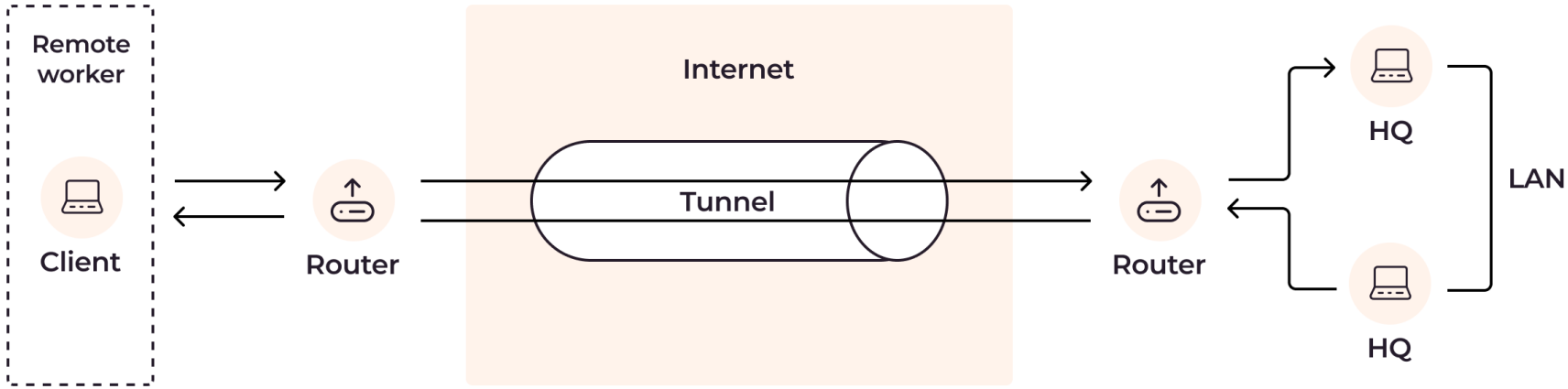


Relational NetKAT

Additional Relations

R = ⋯| **Filter**(**pkr**) | **Apply** (**pkr**, K) | ⋯
pkr = f <- v | f -> v | Id | pk1 × pk2
| **pkr1** ∪ **pkr2** | **pkr1**·**pkr2** |
| **pkr1** ∩ **pkr2** | ⋯

- Additional Components:
  - **pkr** : Packet relations.
  - **Apply** (**pkr**, K): select traces of K, apply **pkr** to each packet.
  - Let **pkr'** = typ <- SSH ∩ …, K= pk11…pk20.



·pk30

1⋯pk30)

Trac

Trace 2

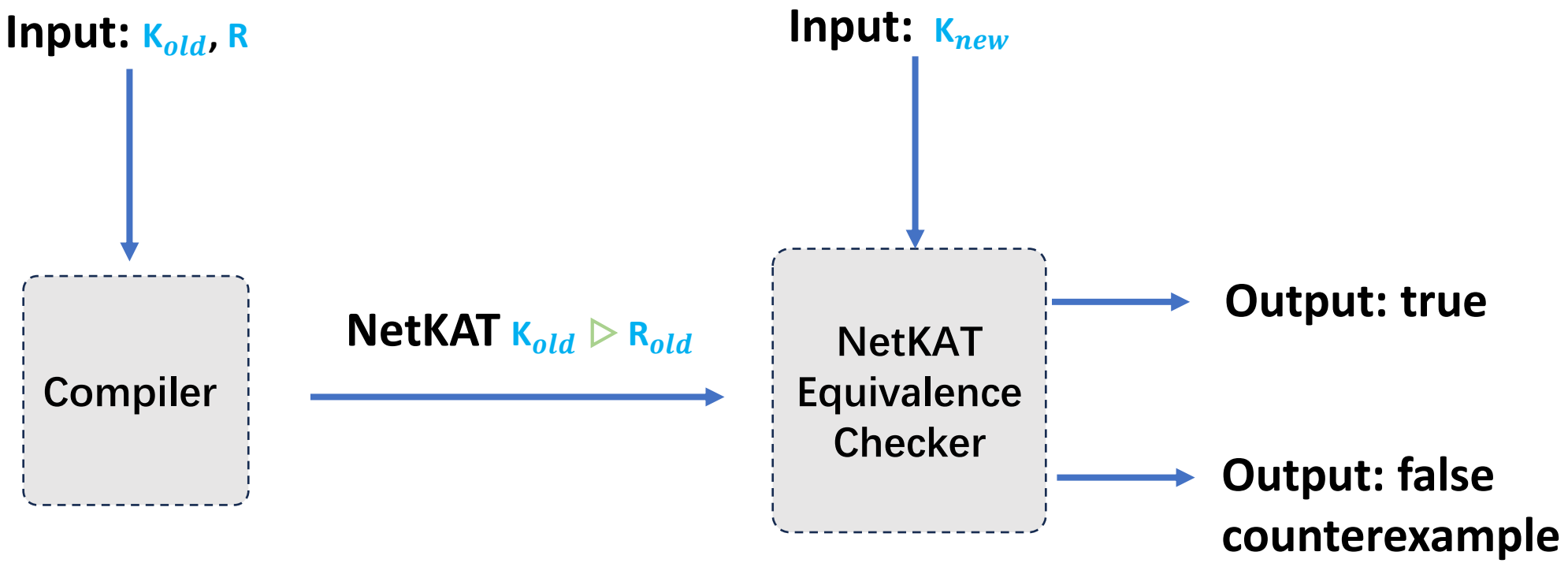
pk1⋯pk10 pk11⋯pk20 pk21⋯pk30

Our Solution in 3 Steps

- Relational NetKAT language:
  - Step 1: Lift Path Sets P to Trace Sets (NetKAT) K.
  - Step 2: Lift Path Relation R to Trace (NetKAT) Relation R.
  - Step 3: Compile and Check $K_{old} \triangleright R = K_{new}$!

Framework

- Components: Compiler for K ▷ R, Checker for K_{old} ▷ R = K_{new}



Compiler K ▷ R

Compiler

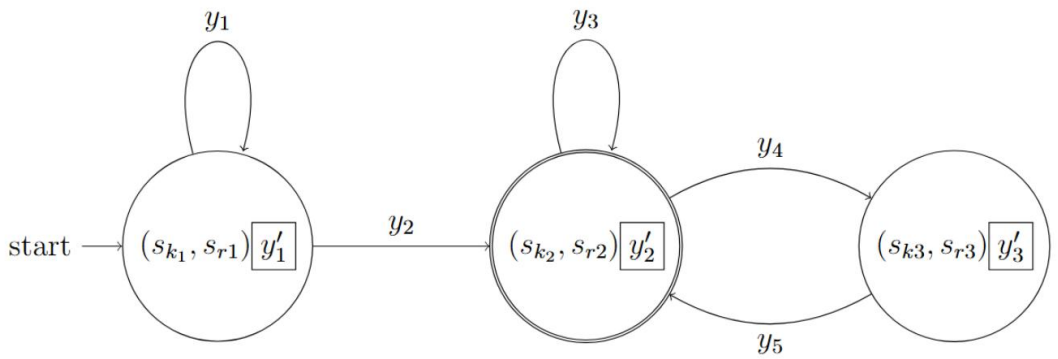
- (Symbolic) Automata Compilation.

Pre-Network K → Compile → Symbolic Automata A(K) Size: O(K)

Changes R → Compile → Symbolic Automata A(R) Size: O(R)

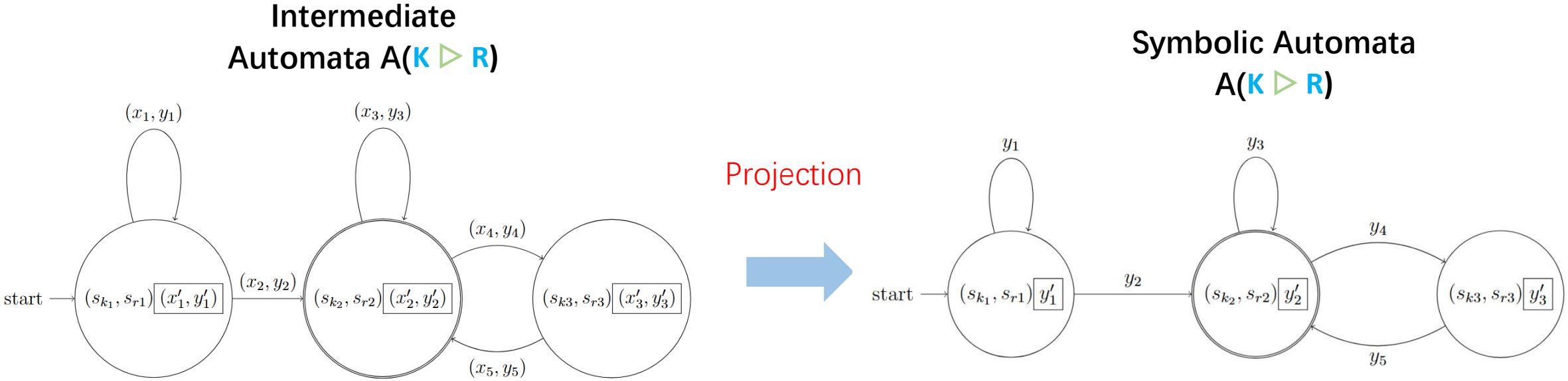
Cross-product And ϵ Elimination → Intermediate Automata A(K ▷ R) Size: O(K*R)

Projection → Symbolic Automata A(K ▷ R) Worst Case Size: O(K*R*|Pk|)



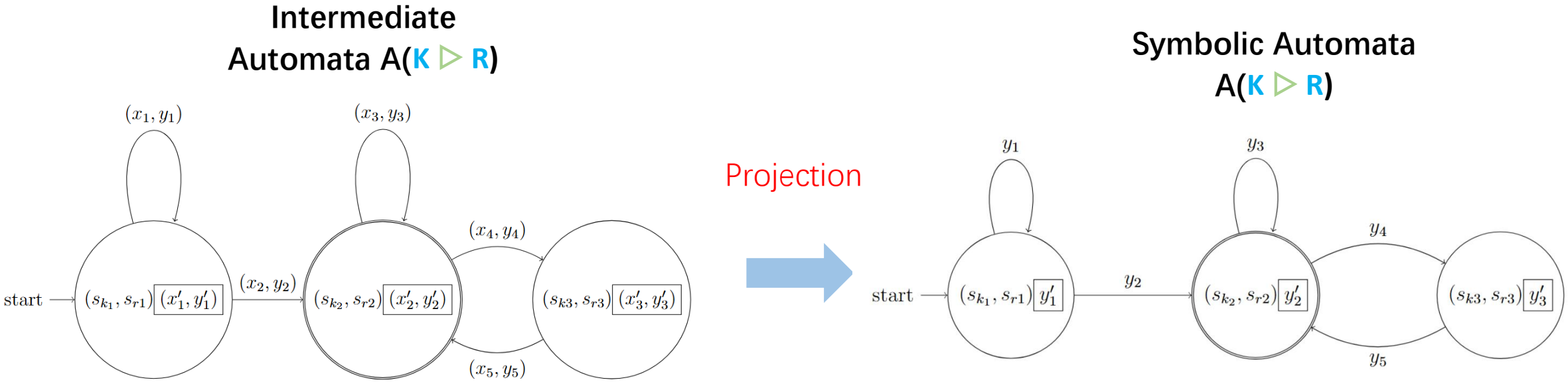
Projection Overview

- NetKAT K generates trace X, Relation R relates traces X and Y .
- Tape elimination usually scale up to O(|**Pk**|) ≈ O(**2^32**)!



Projection Overview

- Most relations are variants of Id and Havoc.
- Y uniquely determine X up to next transition.
- Id and Havoc only splits out 1 state instead of 2^32!



Correctness

- All of the compilation steps are followed with correctness proof.

Theorem 12. *For Δ transitions, there exists $x_0, x_1, \ldots, x_n$ such that:*

$$(s_0, (x_0, y_0)) \xrightarrow{(x_1,y_1)(x_2,y_2)\ldots(x_n,y_n)}_{kr} (s_n, (x_n, y_n)),$$

if and only if for $|\Delta|$ transitions, we have:

$$(s_0, y_0) \xrightarrow{y_1 y_2 \ldots y_n}_{kr} (s_n, y_n).$$

Theorem 8 (Composition). *Let $\mathcal{A}(K)$ be $(S_k, A, s_{k0}, S_{kf}, \Delta_k)$, $\mathcal{A}(R)$ be $(S_r, A', s_{r0}, S_{rf}, \Delta_r, \Delta_{r1}, \Delta_{r2})$, and $\mathcal{A}(K \triangleright R)$ be $(S_{kr}, A', s_{kr0}, S_{krf}, \Delta_{kr})$, which is the composition of $\mathcal{A}(K)$ and $\mathcal{A}(R)$. If $n \geq 1$, then:*

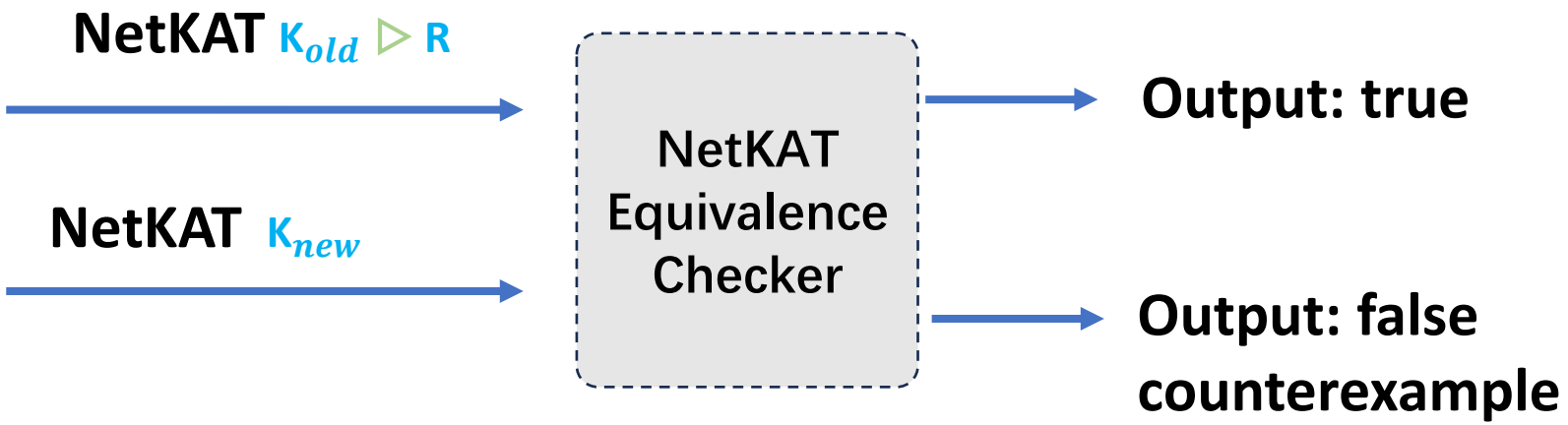
$$\exists x_1, x_2, \ldots, x_n, (s_{kr0}, (x_0, y_0)) \xrightarrow{(x_1,y_1)\cdots(x_n,y_n)}_{kr} ((s_{kn}, s_{rn}), (x_n, y_n))$$

if and only if

$$\exists w, \quad (s_{k0}, x_0) \xrightarrow{w}_k (s_{kn}, x_n) \quad \text{and} \quad (s_{r0}, (x_0, y_0)) \xrightarrow{(w, y_1 \cdots y_n)}_r (s_{rn}, (x_n, y_n)).$$

Checker $K_1 = K_2$

- Algorithm adapted from KATch paper [Moeller et al. 24].



Implementation

- 1000 Loc of code and 600 Loc of test cases.

```
(* Correspond to delta_re1^* transistion in the paper *)
let delta_krx (man:man) (pk1:pk) (pk2:pk) (pk3:pk) (pk4:pk) (nkro:(NK.t option*Rel.t option)):(MLBDD.t)NKROMap.t =
  let worklist = Queue.create() in
  let nk1 = (RN.NK.Seq (Pred True, RN.NK.Star(RN.NK.Union (RN.SNK.add (Asgn (1,true)) (RN.SNK.add RN.NK.Dup RN.SNK.empty))))) in
  let nk2 = (RN.NK.Seq (Asgn (2,true), RN.NK.Seq (RN.NK.Dup,(RN.NK.Union (RN.SNK.add (RN.NK.Seq (Asgn (3,true),RN.NK.Dup)) (RN.SNK.add
  (* Inter (nk1,nk2) |> Id *)
  (* nil (Id) = (pk,pk) *)
  (* App (Id,Id) = (pk1pk2,pk1pk2) *)
  (* nil(Id) App (Id,Id) = {(pk,pk)|pk\in PK} . {(pk1pk2,pk1pk2)|pk1,pk2 \in PK} = {(pk.pk1pk2,pk.pk1pk2)|pk,pk1,pk2 \in PK} *)
  (* = {(pk.pk1pk2,pk.pk1pk2)|pk,pk1,pk2 \in PK /\ pk = pk1} = {(pk1pk2,pk1pk2)|pk1,pk2 \in PK} = App (Id,Id)*)
  let nkro7 = (Some (RN.NK.Inter (nk1,nk2)), Some (RN.Rel.StarR (RN.Rel.App (Id,Id)))) in
  let nkrosmap7 = RN.generate_all_transition man pk1 pk2 pk3 pk4 nkro7  in
  let nkrobmap7 = RN.simplify_all_transition man pk1 pk2 pk3 pk4 nkrosmap7 in
  let (nkrobsmap7,start11) = RN.determinization nkro7 nkrobmap7 in
  (* nk1 |> Id (nk2) *)
  let nkro8 = (Some nk1, Some (RN.Rel.Id nk2)) in
  let nkrosmap8 = RN.generate_all_transition man pk1 pk2 pk3 pk4 nkro8  in
  let nkrobmap8 = RN.simplify_all_transition man pk1 pk2 pk3 pk4 nkrosmap8 in
  let (nkrobsmap8,start12) = RN.determinization nkro8 nkrobmap8 in
  assert_equal true (RN.bisim man pk3 pk4 start11 start12 nkrobsmap7 nkrobsmap8);
      delta_krx_aux (ROMap.fold (fun ro bdd acc -> Queue.add ((nko,ro),bdd) worklist;
                                  add_nkro_mapping (nko,ro) bdd acc) epsilon_closure_ro_map_left NKROMap.empty)
```

Conclusions and Ongoing Works

- Relational NetKAT: An all-path verifier for traces and trace changes.
- Compact and expressive syntax.
- Fast and efficient checking and compilation.
- Implementation is ready.
  - Correctness: 😊
  - Performance: 🤔

Appendix

A quick look of projection algorithm

1. **Consistency:** For all $s, s', x_1, x_1', x_2, x_2', y, y_1', y_2'$:

$$((x_1, y), (x_1', y_1')) \in \Delta\ s\ s' \wedge ((x_2, y), (x_2', y_2')) \in \Delta\ s\ s'$$

implies:

$$((x_1, y), (x_2', y_2')) \in \Delta\ s\ s' \wedge ((x_2, y), (x_1', y_1')) \in \Delta\ s\ s'.$$

2. **Summary:** For all s, s', y, one of the following holds:
   - For all x, it holds that

$$\exists x', y', ((x, y), (x', y')) \in \Delta\ s\ s' \iff (x, y) \in R(s).$$

   - For all x, x', y', it holds that:

$$((x, y), (x', y')) \notin \Delta\ s\ s'.$$

3. **Preservation of R:** For all s, s', x, y, x', y', if:

$$((x, y), (x', y')) \in \Delta\ s\ s',$$

then:

$$(x', y') \in R(s').$$

Appendix

Algorithm 1: NetKAT Automata Equivalence Check

Input: Two automata $\mathcal{A}_1 = (S_1, A, s_{10}, S_{1f}, \Delta_1)$, $\mathcal{A}_2 = (S_2, A, s_{20}, S_{2f}, \Delta_2)$.

Output: A Boolean indicating whether the automata are equivalent.

1. Initialize $W \leftarrow \{(s_{10}, s_{20}, \alpha) \mid \alpha \in A\}$.
2. **While** W changes:
   (a) **For each** $(s_1, s_2, \alpha) \in W$:
      i. **If** $s_1 \in S_{1f}$ and $s_2 \notin S_{2f}$ or vice versa, **then return false**.
      ii. **Else** Update $W \leftarrow W \cup \{(s_1', s_2', \alpha') \mid (\alpha, \alpha') \in (\Delta_1(s_1, s_1') \cap \Delta_2(s_2, s_2'))\}$.
3. **Return true**.