Georgia Institute of Technology

Dr. Nick Feamster
Associate Professor

Software Defined Networking

In this course, you will learn about software defined networking and how it is changing the way communications networks are managed, maintained, and secured.

School of Computer Science

This Module: Verification

- **Motivation:** How do you know the network is doing the right thing?
- Verification techniques
  - Configuration Verification: rcc (pre-SDN)
  - Control Plane Verification: Kinetic
  - Data Plane Verification
    - Header Space Analysis
    - Veriflow

Simple Questions are Hard

- What are all the packet headers from A that can reach B?
- What will happen if I remove an entry from a firewall?
- Is Group X provably isolated from Group Y?
- Are there any loops in the network?
- Why is my network slow?

Configuration Defines Behavior

Provides flexibility for realizing operational goals

- How traffic enters and leaves the network
  - Load balance
  - Traffic engineering
  - Primary/backup paths
- Which neighboring networks can send traffic
  - Defines business relationships and contracts
- How routers within the network learn routes
  - Scaling and performance

Flexibility ⟶ **Complexity**

Most Important Goal: Correctness

Unfortunately…

Mistakes happen!

Why?

- Configuration is difficult. Operators make mistakes.
  - Complex policies
  - Configuration is distributed across routers
- Each network independently configured
  - Unintended policy interactions

Problem

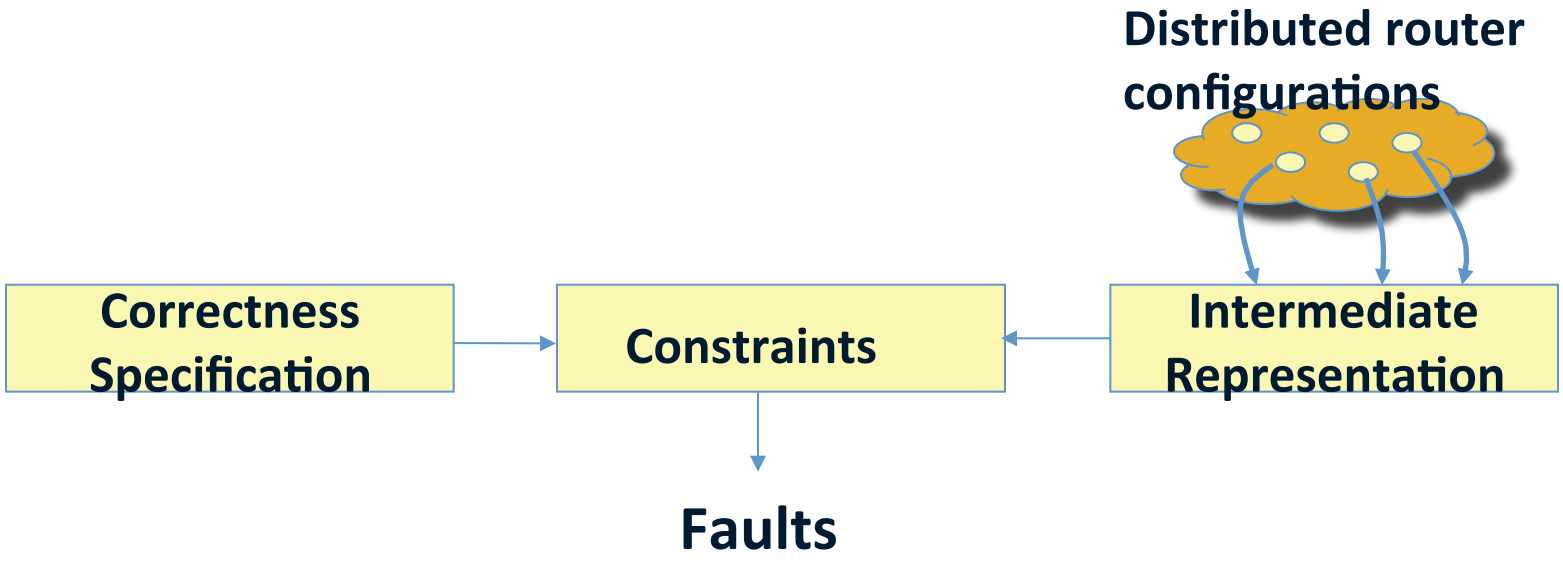
Guarantee correctness of the global routing system.

Examine only local configurations.

Checking Configuration

- Correctness specification and constraints for global Internet routing
- rcc ("router configuration checker")
  - Static configuration analysis tool for fault detection
  - Used by network operators (including large backbone networks)
- Analysis of real-world network configurations from 17 autonomous systems

rcc Design


Distributed router configurations
Correctness Specification
Constraints
Intermediate Representation
Faults

Challenges

- Defining a correctness specification
- Deriving verifiable constraints from specification
- Analyzing complex, distributed configuration
- Verifying correctness with local (per-AS) information

Correctness Specification

Path Visibility

For each usable path, a corresponding route advertisement must be available

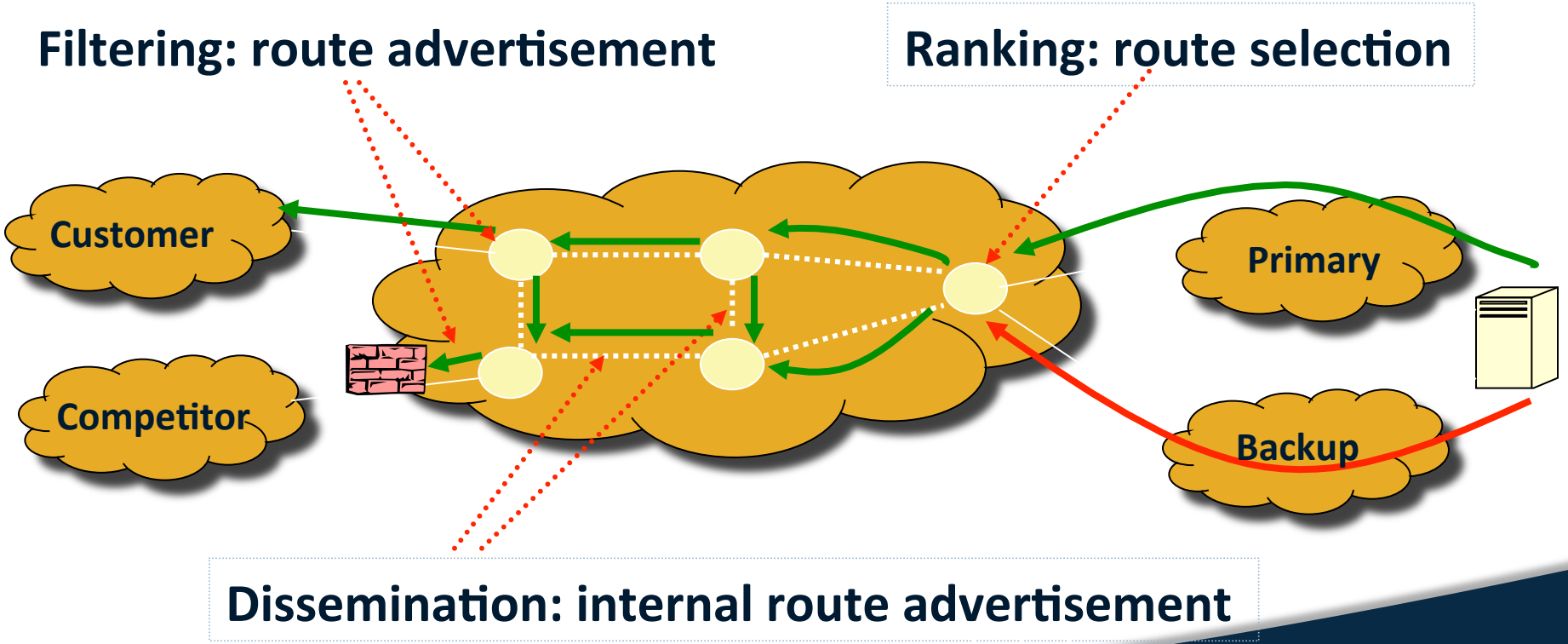
Route Validity

For each available route, there must exist a corresponding usable path

Safety

For any given set of configurations distributed across routers in different ASes, a stable path must exist, and the protocol must converge to it

Factoring Routing Configuration



Path Visibility

If every router learns a route for every usable path, then path visibility is satisfied.

A *usable path*:

- Reaches the destination
- Corresponds to the path that packets take when using that route
- Conforms to the policies of the routers on that path

Possible path visibility faults

Dissemination

- **Partition in session-level graph that disseminates routes**

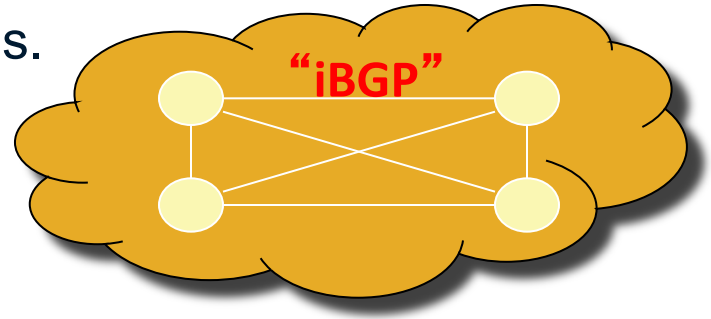
Filtering

- **Filtering routes for prefixes for usable paths**

Path Visibility: Internal BGP (iBGP)

- **Default:** dont re-advertise iBGP-learned routes.

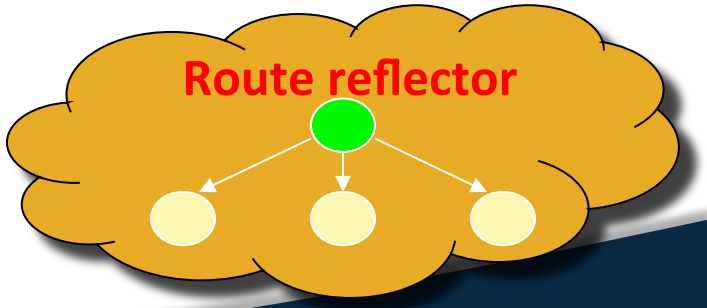
  Complete propagation requires "full mesh" iBGP. *Doesn't scale.*



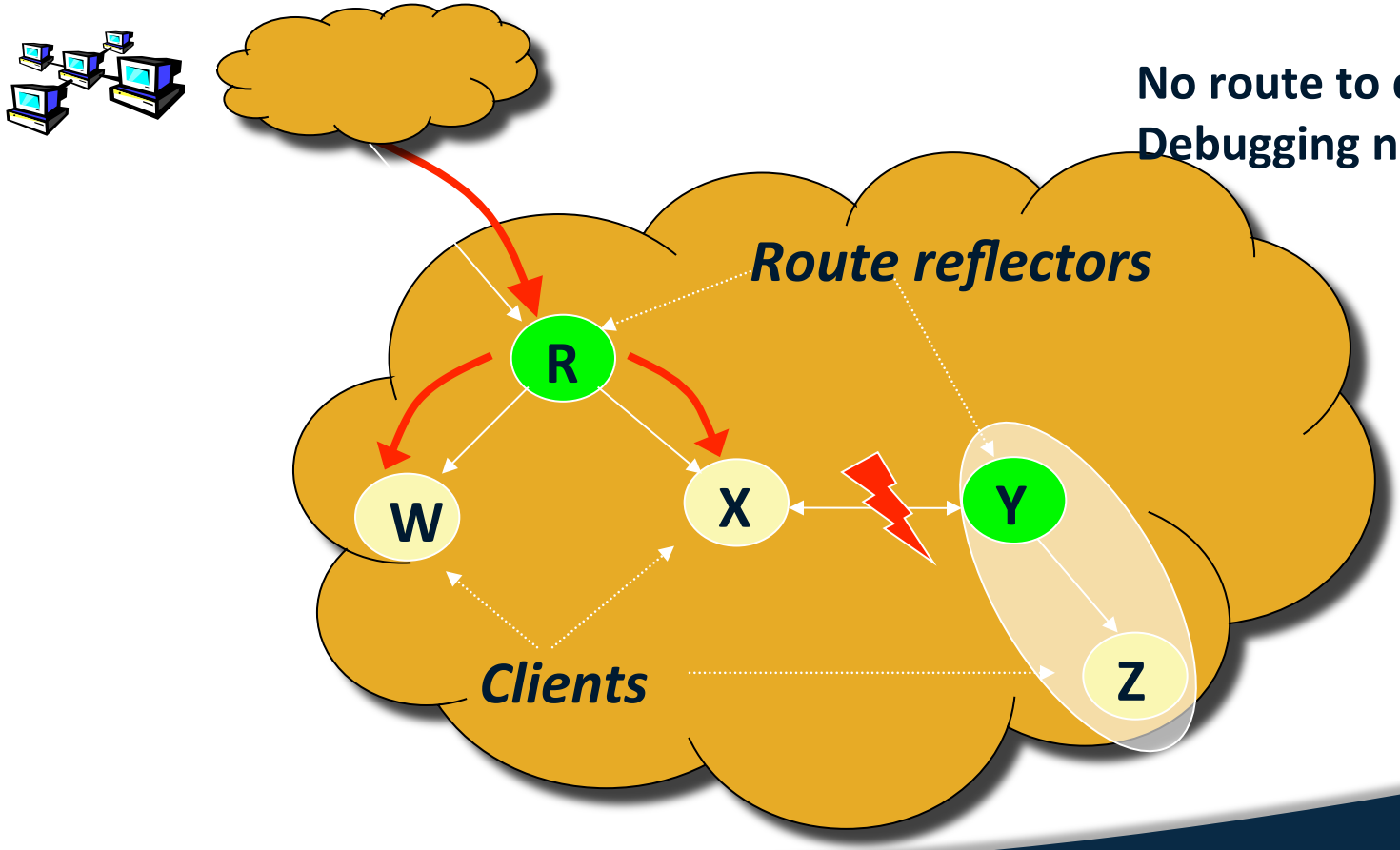
- "Route reflection" improves scaling.

  Client: re-advertise as usual.

  Route reflector: reflect non-client routes to all clients, client routes to non-clients and other clients.

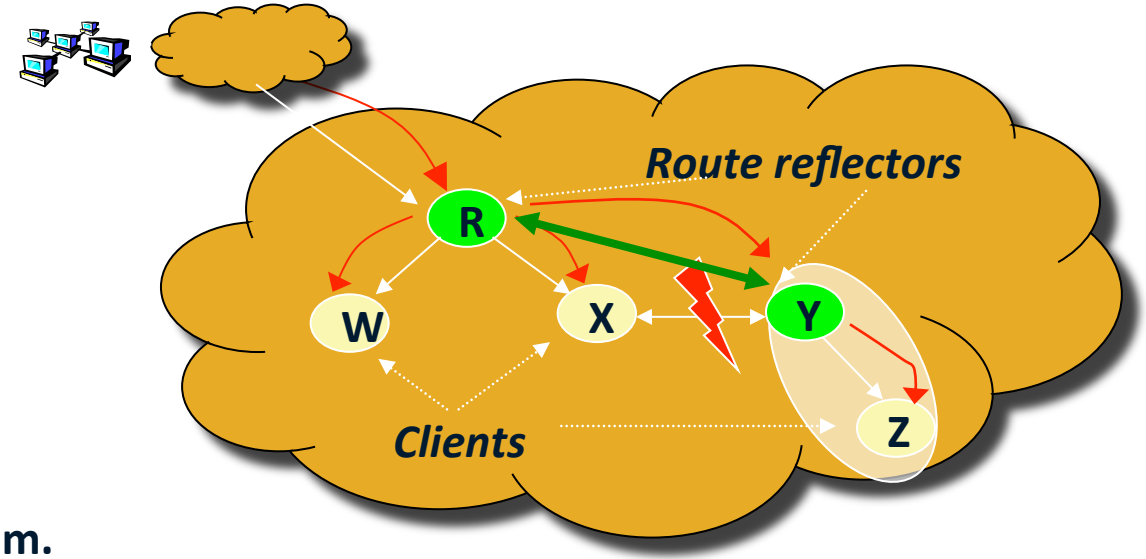


Path Visibility: iBGP Signaling



No route to destination.
Debugging nightmare!

Path Visibility: iBGP Signaling



Theorem.
Suppose the iBGP reflector-client relationship graph contains no cycles. Then, path visibility is safisfied if, and only if, *the set of routers that are not route reflector clients forms a full mesh*.

Condition is easy to check with static analysis.

Route Validity

If every route that a router learns corresponds to a usable path, then route validity is satisfied.

A *usable path*:

- Reaches the destination
- Corresponds to the path that packets take when using that route
- Conforms to the policies of the routers on that

Possible route validity faults

Filtering

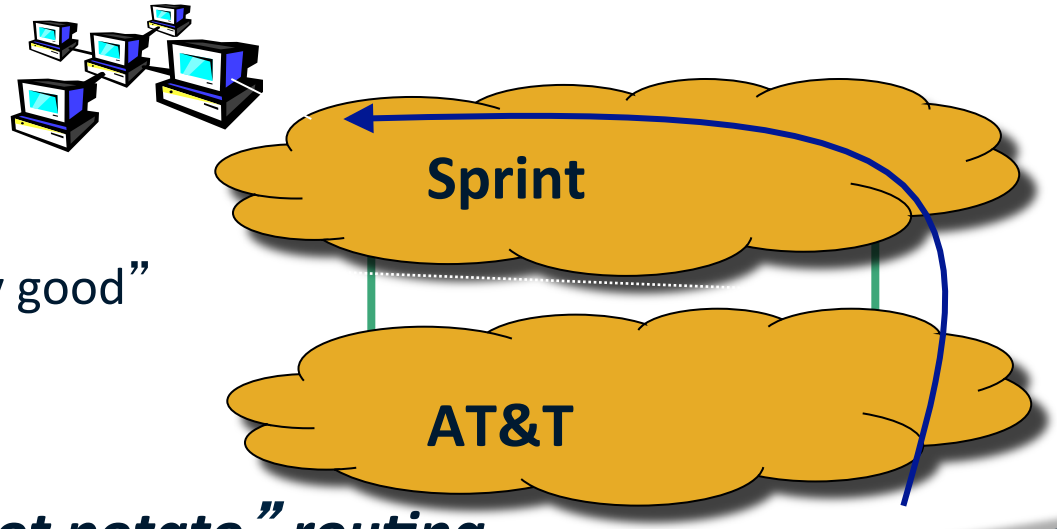
- Unintentionally providing transit service
- Advertising routes that violate higher-level policy
- Originating routes for private (or unowned) address space

Dissemination

- Loops and “deflections

Route Validity: Consistent Export

- Rules of *settlement-free peering*:
  - Advertise routes at all peering points
  - Advertised routes must have equal "AS path length"



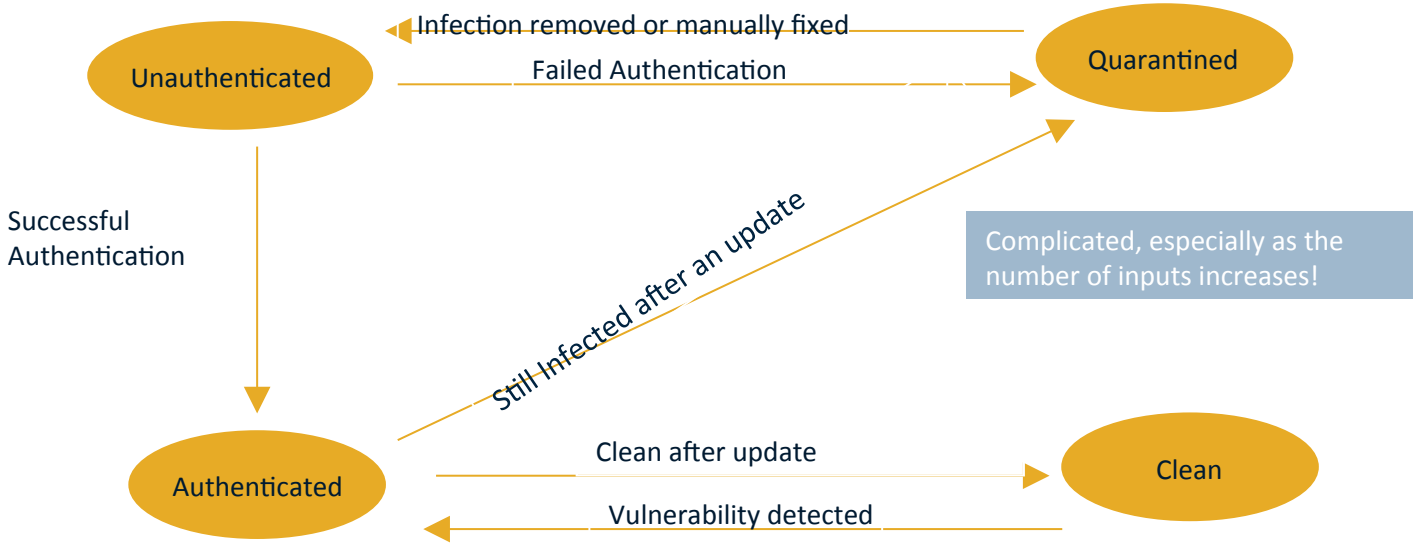
"equally good" routes

Enables "hot potato" routing.

This Module: Verification

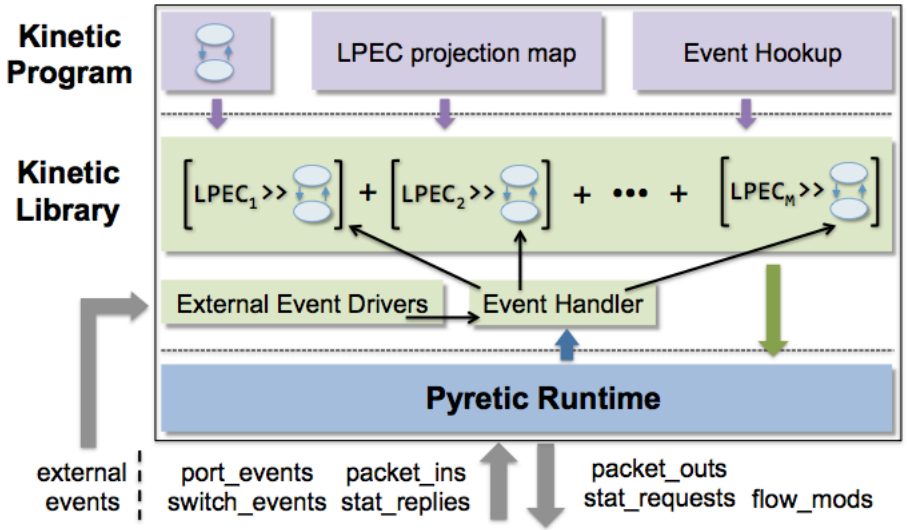
- **Motivation:** How do you know the network is doing the right thing?
- Verification techniques
  - Configuration Verification: rcc (pre-SDN)
  - Control Plane Verification: Kinetic
  - Data Plane Verification
    - Header Space Analysis
    - Veriflow

Kinetic: Verifiable Event-Based Network Control



- Network policies represented as FSMs
- FSMs are verifiable!

Kinetic System Architecture



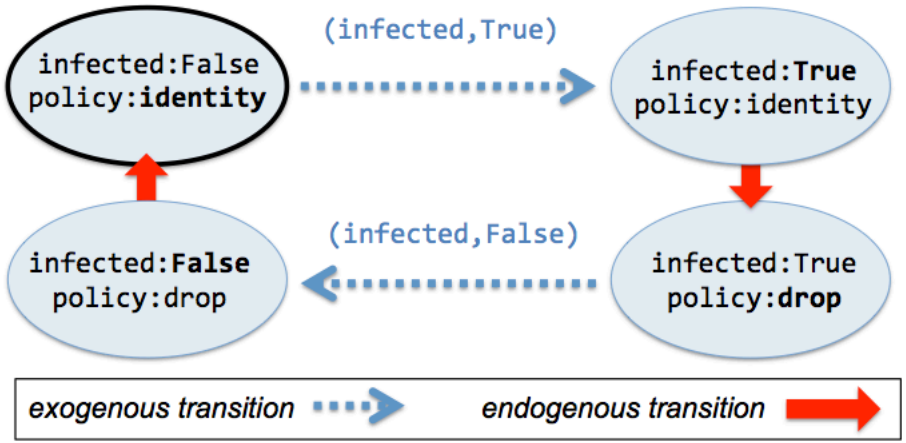
- LPEC projection map divides located packets into equivalence classes
- Event hookup for external events

Kinetic Language Architecture

Kinetic	K ::= P | FSMPolicy(L,M) | K + K | K >> K L ::= *f* : *packet* -> F M ::= FSMDef([*var_name*=V]) V ::= VarDef(*type*,*init_val*,T) T ::= [case(S,D)] S ::= D==D | S & S | (S | S) | ! S D ::= C(*value*) | V(*var_name*) | event
Pyretic	P ::= Dynamic() | N | P + P | P >> P
Static Pyretic	N ::= B | F | modify(h=v) | N + N | N >> N F ::= A | F & F | (F | F) | ~F A ::= identity | drop | match(h=v) | B ::= FwdBucket() | CountBucket()

- Extensions to Pyretic
- Special dynamic policy class FSMPolicy
- FSM descriptions and basic values

Example: Intrusion Detection System




```
1  @transition
2  def infected(self):
3    self.case(occured(self.event),self.event)
4
5  @transition
6  def policy(self):
7   self.case(is_true(V('infected')),C(drop))
8   self.default(C(identity))
9
10 self.fsm_def = FSMDef(
11  infected=FSMVar(type=BoolType(),
12                  init=False,
13                  trans=infected),
14  policy=FSMVar(type=PolType({drop,identity}),
15                init=identity,
16                trans=policy))
```

LPEC Policy Description

Step (1)

```
match(srcip=IPAddr('10.0.0.1'))
```

Step (2)

```
def ids_lpec_pm(pkt):
    return match(srcip=IPAddr('10.0.0.1'))
```

Step (3)

```
17  def ids_lpec_pm(pkt):
18    return match(srcip=pkt['srcip'])
```

- Specify LPEC
- Define proejction MAP
- Parameterizes using input packet

Conversion to NuSMV

```
1  MODULE main
2    VAR
3      policy    : {identity,drop};
4      infected  : boolean;
5    ASSIGN
6      init(policy)   := identity;
7      init(infected) := FALSE;
8    next(policy) :=
9      case
10       infected  : drop;
11       TRUE      : identity;
12     esac;
13   next(infected) :=
14     case
15       TRUE      : {FALSE,TRUE};
16       TRUE      : infected;
17   esac;
```

- FSMs translate directly to NuSMV model checker
- Can check properties in CTL

CTL Examples for Kinetic IDS

NuSMV	Description
`AG infected →` `(policy=drop)`	If infection event arrives, the system should drop the packet.
`AG !infected →` `(policy=identity)`	If infection is cleared, the system should allow the packet.
`AG EF policy=identity`	From any state, it is possible to go to allowed state again.
`A [ policy=identity` `U infected ]`	For all paths, policy allows packet until an infection occurs.

- Rules expressed using CTL

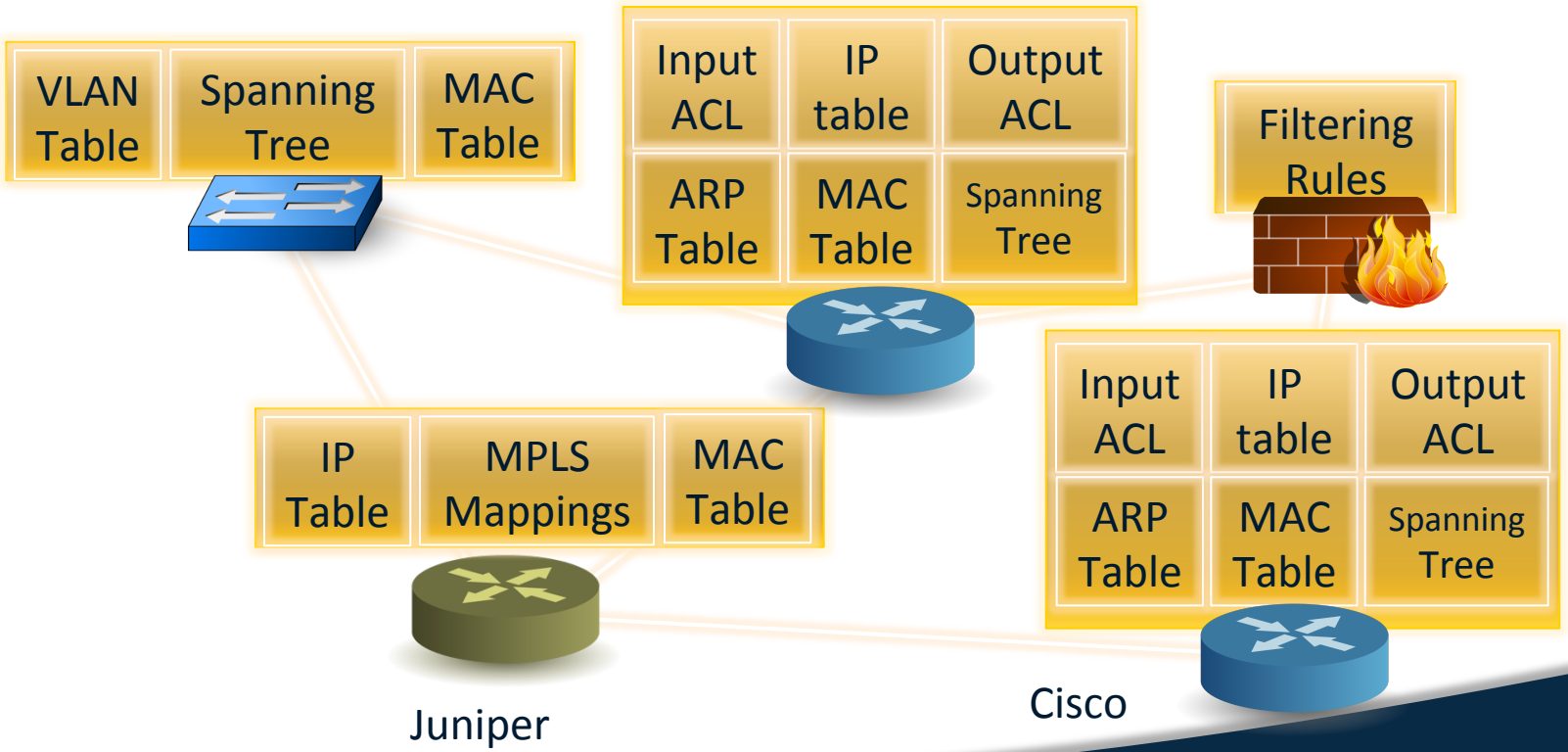
Summary

- Event-based control is a common idiom
- Need to verify dynamic properties of network control, not only data-plane properties
- Kinetic: Verifiable dynamic network control
  - Policies expressed as FSMs
  - FSMs map naturally to model checking
  - Properties can be checked in CTL

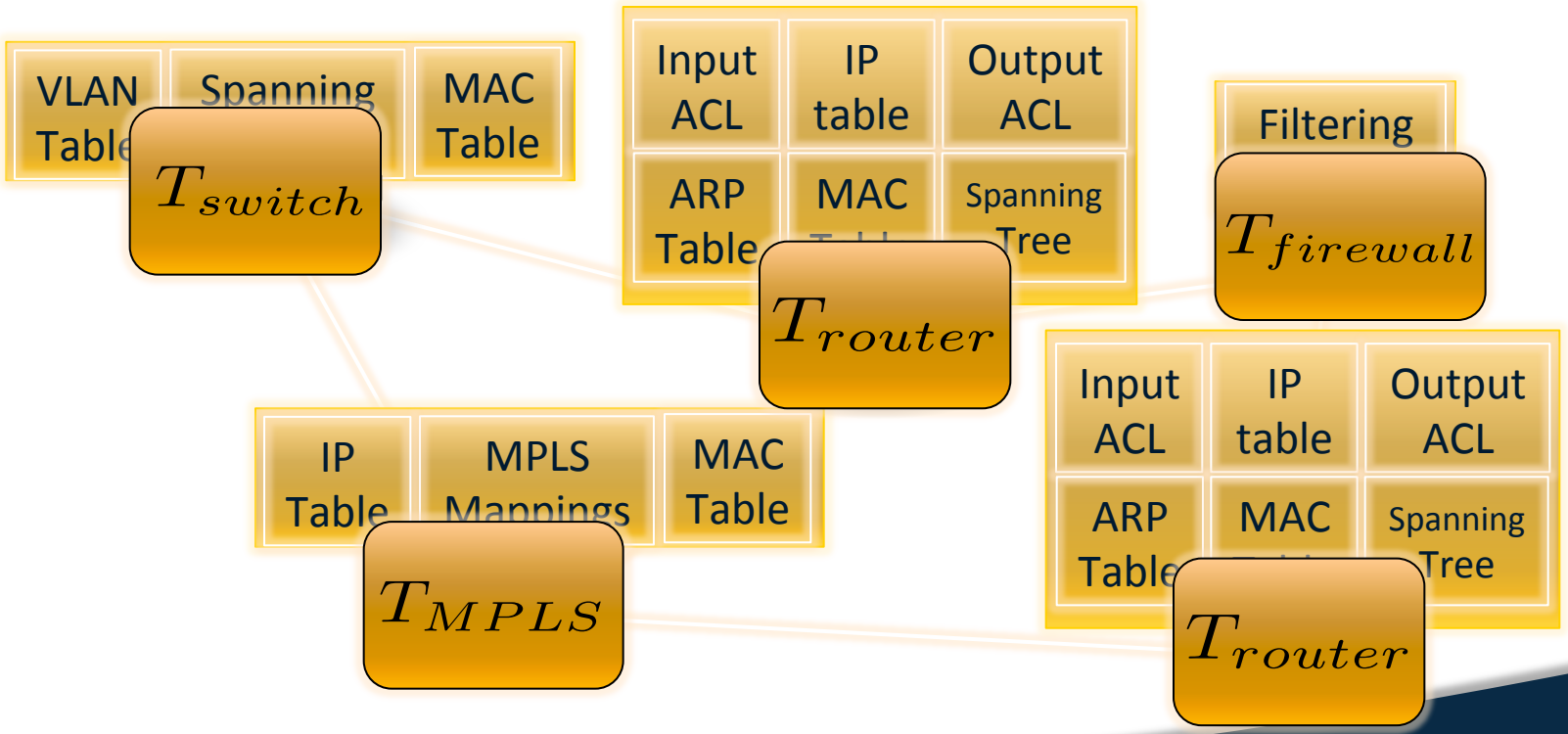
This Module: Verification

- **Motivation:** How do you know the network is doing the right thing?
- Verification techniques
  - Configuration Verification: rcc (pre-SDN)
  - Data Plane Verification
    - Header Space Analysis
    - Veriflow
  - Control Plane Verification: Kinetic

Network Verification Vision

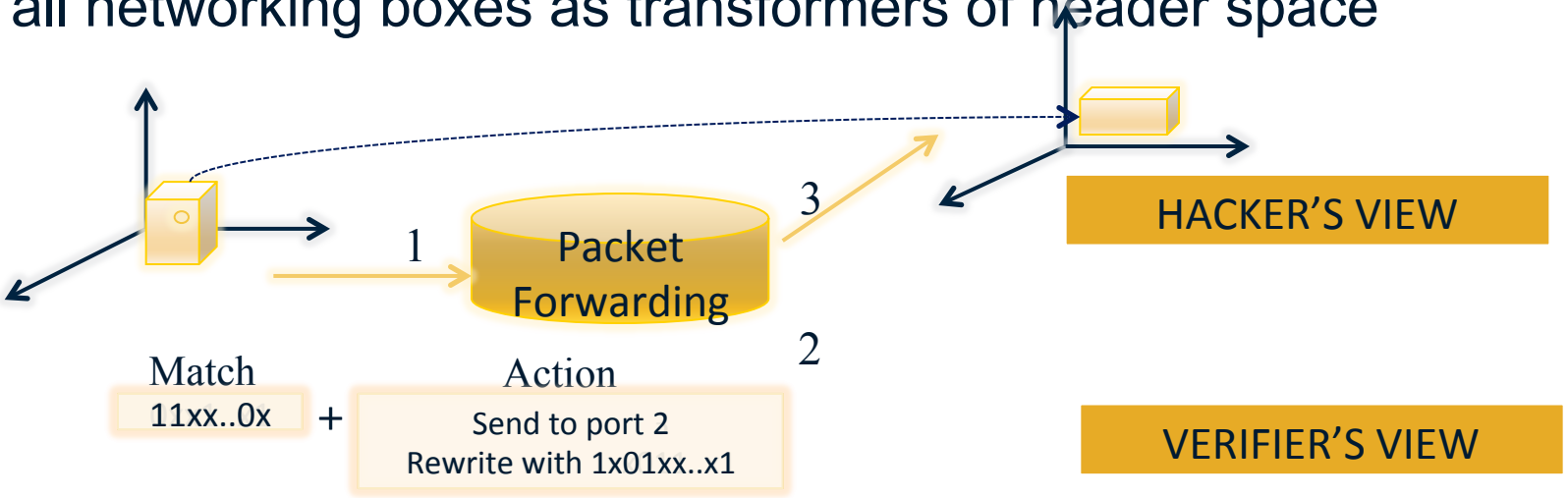


Network Verification Vision



Insight: Treat Network as a Program

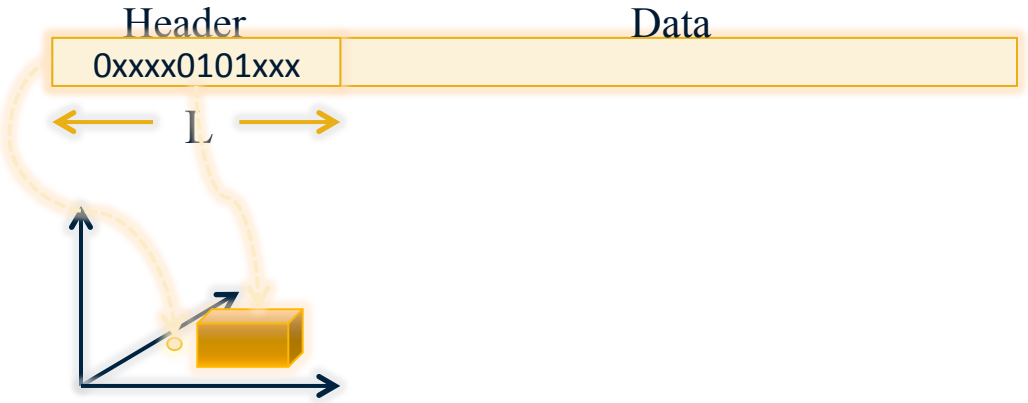
- Model header as point in high dimensional space and all networking boxes as transformers of header space



ROUTER ABSTRACTED AS SET OF GUARDED COMMANDS . .
NETWORK BECOMES A PROGRAM →**CAN USE PL TOOLS**

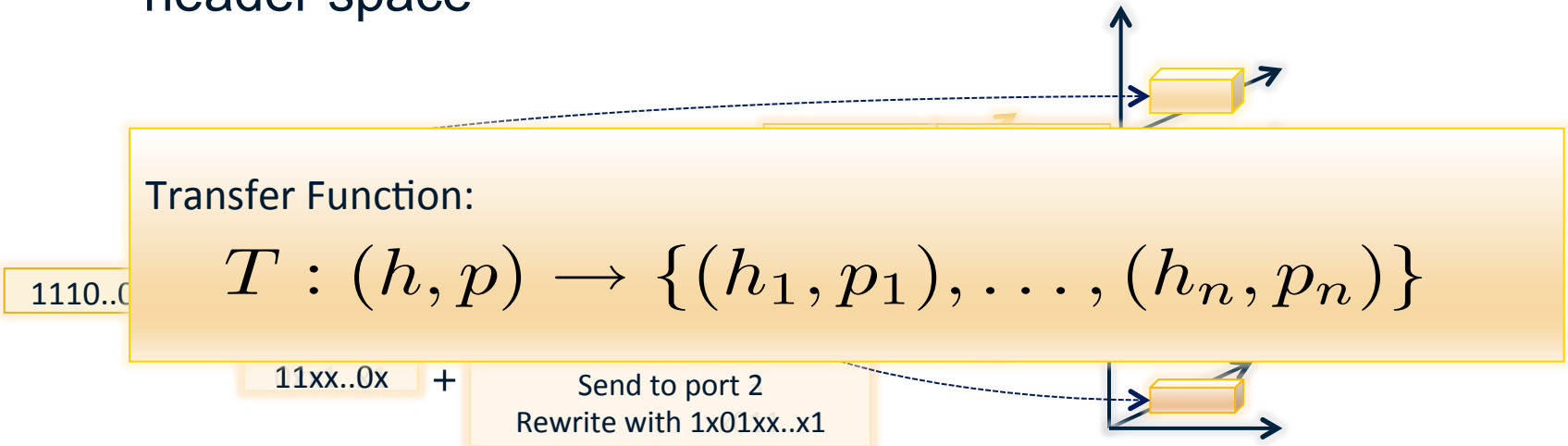
Header Space Framework

- Step 1 - Model a packet, based on its header bits, as a point in $\{0,1\}^L$ space – The Header Space



Header Space Framework

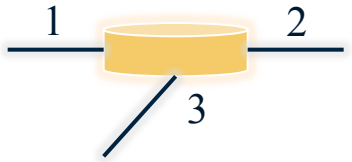
- Step 2 – Model all networking boxes as transformers of header space



Transfer Function Example

IPv4 Router – Forwarding Behavior

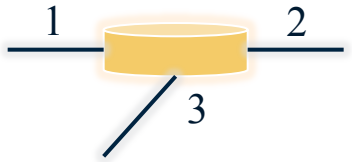
- 172.24.74.x Port1
- 172.24.128.x Port2
- 171.67.x.x Port3



$$T(h, p) = \begin{cases} (h,1) & \text{if dst\_ip}(h) = 172.24.74.x \\ (h,2) & \text{if dst\_ip}(h) = 172.24.128.x \\ (h,3) & \text{if dst\_ip}(h) = 171.67.x.x \end{cases}$$

Transfer Function Example

- IPv4 Router – forwarding + TTL + MAC rewrite
  - 172.24.74.x Port1
  - 172.24.128.x Port2
  - 171.67.x.x Port3



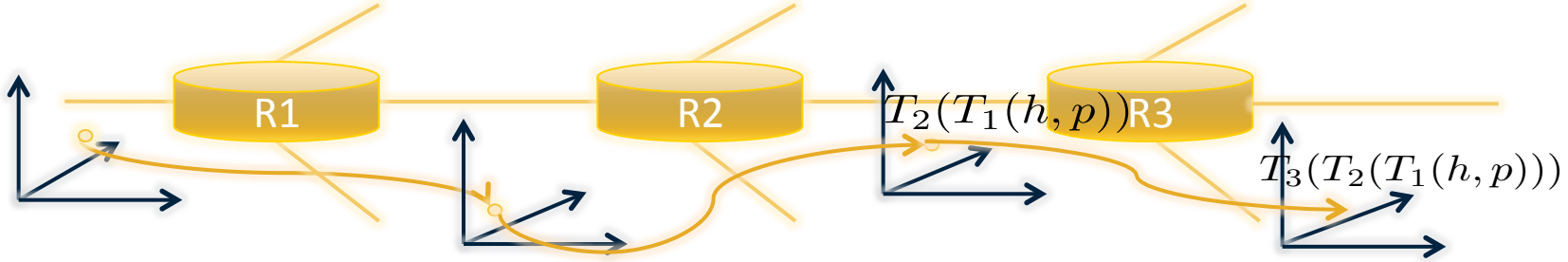
$$T(h, p) = \begin{cases} (\text{rw\_mac}(\text{dec\_ttl}(h), \text{next\_mac}), 1) & \text{if dst\_ip}(h) = 172.24.74.x \\ (\text{rw\_mac}(\text{dec\_ttl}(h), \text{next\_mac}), 2) & \text{if dst\_ip}(h) = 172.24.128.x \\ (\text{rw\_mac}(\text{dec\_ttl}(h), \text{next\_mac}), 3) & \text{if dst\_ip}(h) = 171.67.x.x \end{cases}$$

Example Actions

- Rewrite: rewrite bits 0-2 with value 101
  - (h & 000111…) | 101000…
- Encapsulation: encap packet in a 1010 header.
  - (h >> 4) | 1010….
- Decapsulation: decap 1010xxx… packets
  - (h << 4) | 000…xxxx
- TTL Decrement:
  - if ttl(h) == 0: Drop
  - if ttl(h) > 0: h – 0…00000010…0
- Load Balancing:
  - LB(h,p) = {(h,P_1),…(h,P_n)}

Composing Transfer Functions

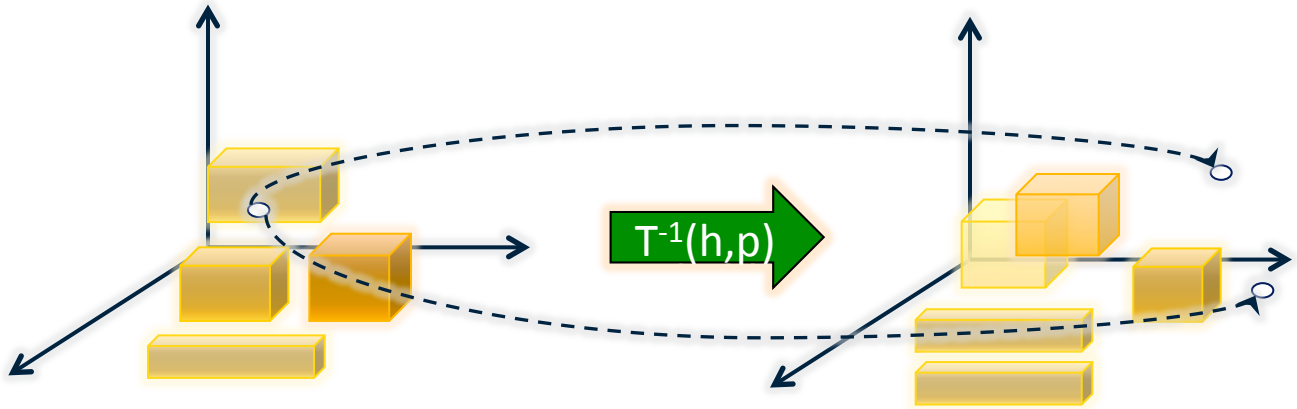
- We can determine end to end behavior by composing transfer functions,



$$T_3(T_2(T_1(h,p)))$$

Inverting Transfer Functions

- Tell us all possible input packets that can generate an output packet.

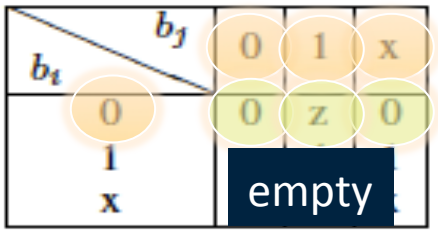


Header Space Framework

- Step 3: Header Space Set Algebra
  - Intersection
  - Complementation
  - Difference
  - Check subset and equality condition.
- Every region of Header Space, can be described by union of Wildcard Expressions. (example: 10xx U 011x)
- **Goal**: do set operation on wildcard expressions.

HS Set Algebra: Intersection

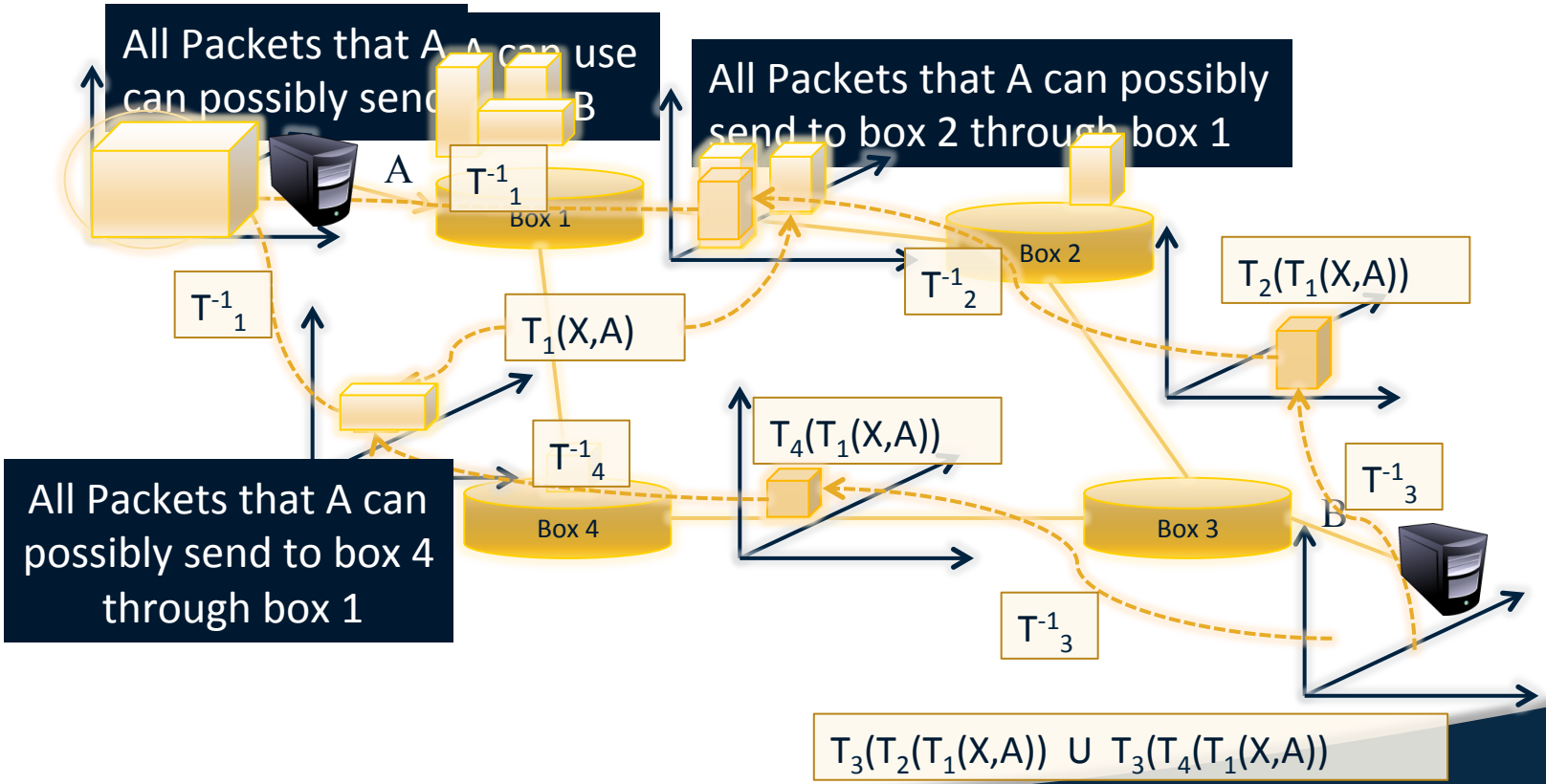
- Bit by bit intersect using intersection table:
  - Example: $10xx \cap 1xx0 = 10x0$
  - If result has any 'z', then intersection is empty:
  - Example: $10xx \cap 0xx0 = z0x0 = \phi$



Header Space Framework

- Simple abstraction that gives us:
  - Common model for all packets
    - Header Space.
  - Common model for forwarding functionality of all networking boxes.
    - Transfer Function.
  - Mathematical foundation to check end-to-end properties about networks.
    - $T(h,p)$ and $T^{-1}(h,p)$.
    - Set operations on Header Space.

Finding Reachability

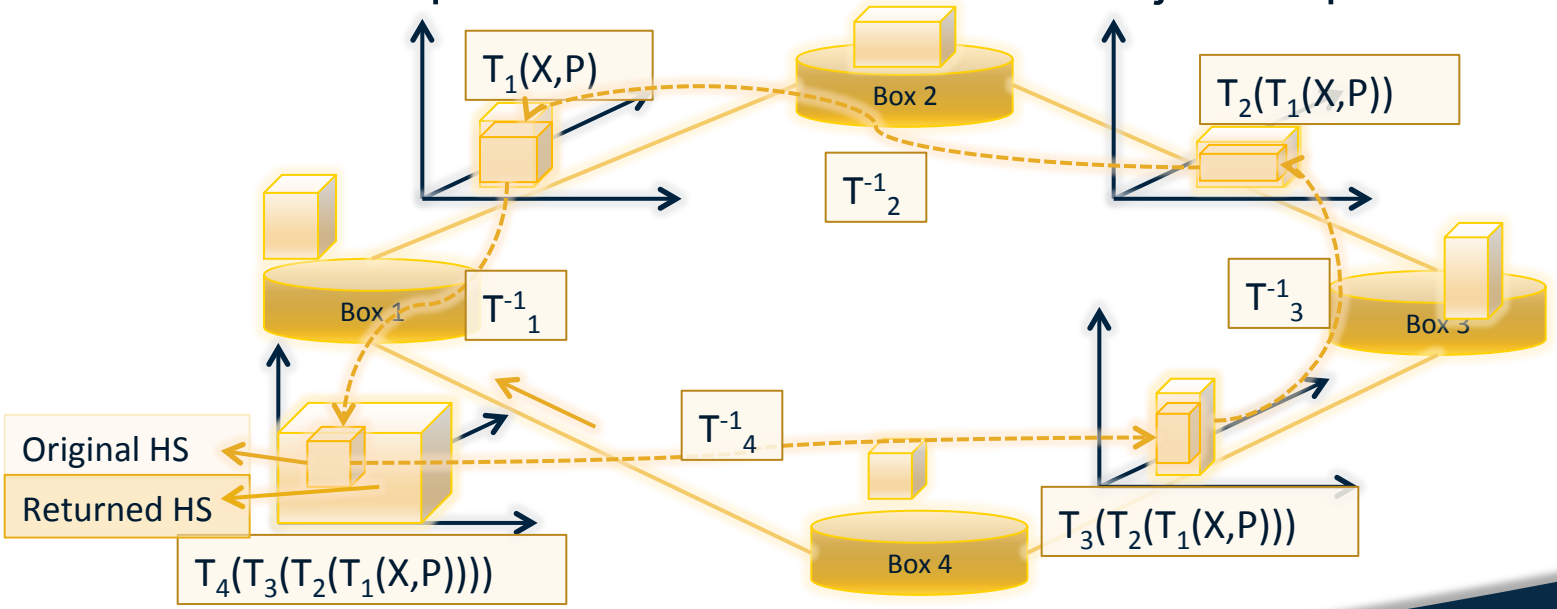


Predicates on Paths: Policies

- Can generalize to check *path predicates:*
  - Blackhole freedom (A → B and notice unexpected drop)
  - Communication via middle box. (A→B packets must pass through C)
  - Maximum hop count (length of path from A → B never exceeds L)
  - Isolation of paths (http and https traffic from A→B don't share the same path)

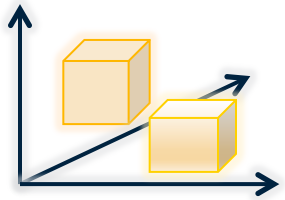
Finding Loops

- Is there a loop in the network?
  - Inject an all-x test packet from every switch-port
  - Follow the packet until it comes back to injection port

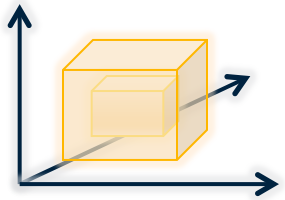


Finding Loops

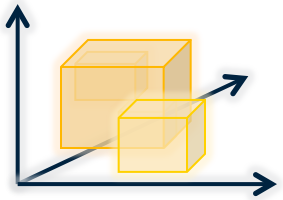
- Is the loop infinite?



Finite Loop



Infinite Loop



?

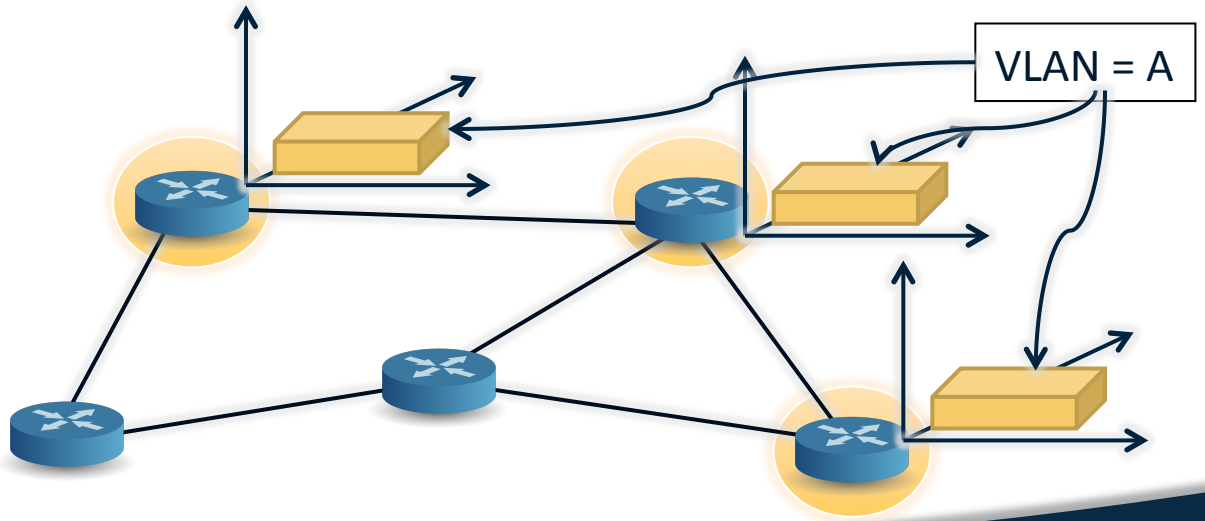
Network Slices

- By slicing network we can share network resources. (e.g. Bank of America and Citi share the same infrastructure in a financial center).
- Like VM, we need to ensure no interaction between slices. (security, independence of slices).

→ We need to check isolation of slices.

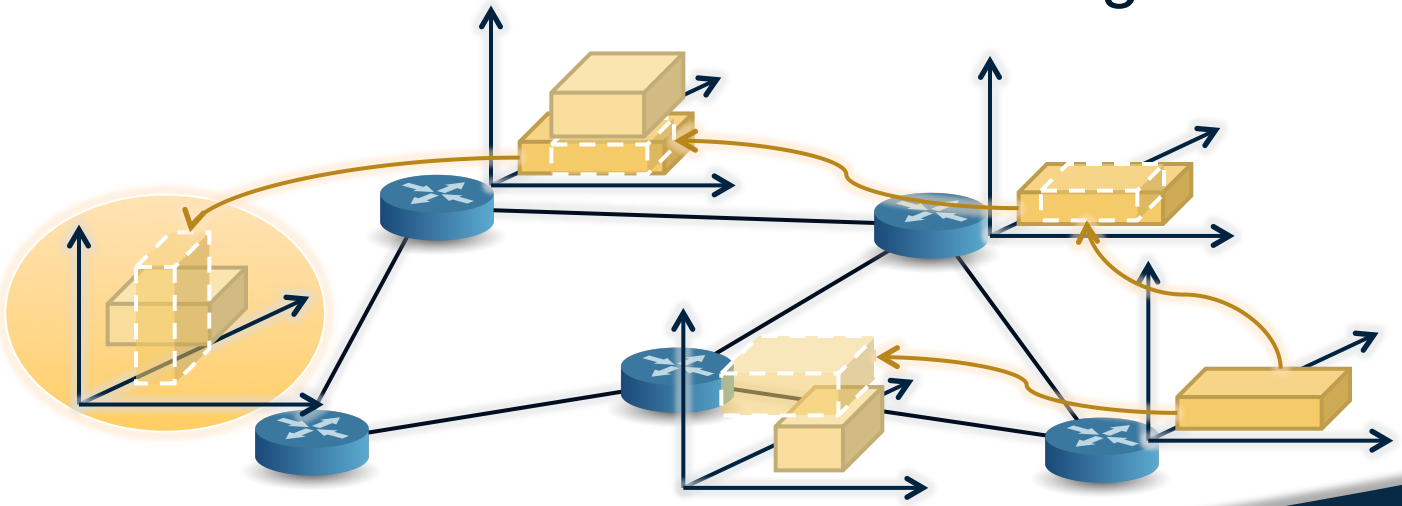
Definition of Slice in HSA

- Network slice is a piece of network resources defined by
  - A topology consisting of switches and ports.
  - A set of predicates on packet headers.

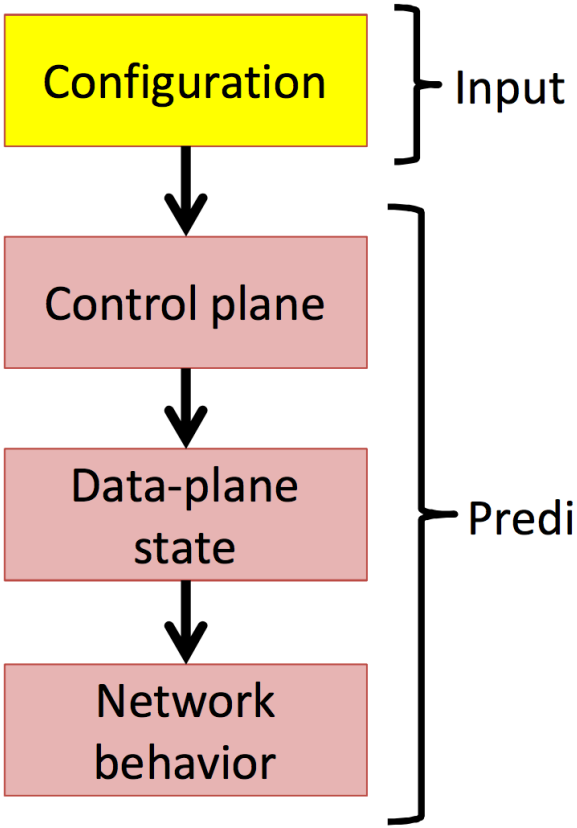


Checking Isolation of Slices

- How to check if two slices are isolated?
  - Slice definitions don't intersect.
  - Packets don't leak after forwarding.

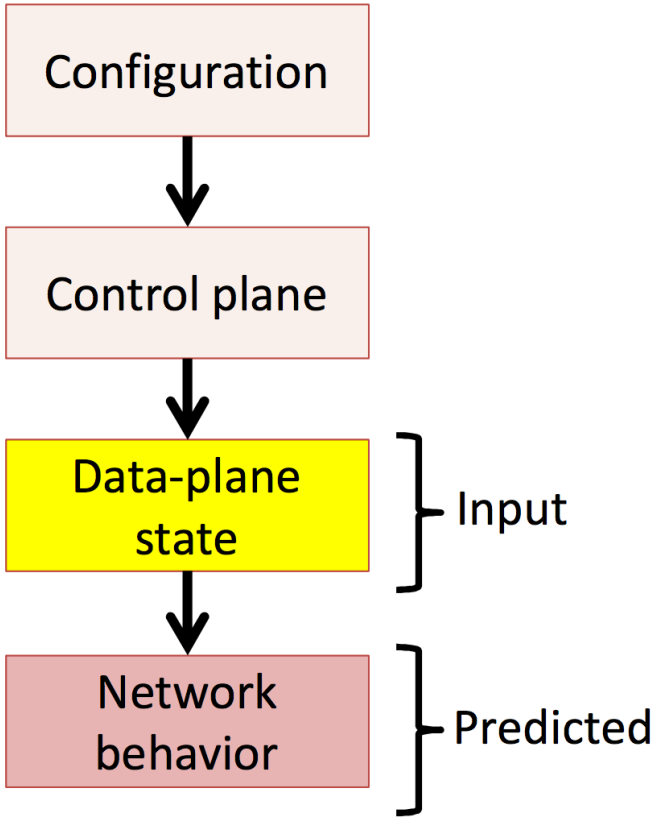


Limitations of Configuration Verification



- Prediction is difficult
  - Various configuration languages
  - Dynamic distributed protocols
- Prediction misses implementation bugs in control plane

Veriflow: Data-Plane Verification



- Less prediction
- Closer to actual network behavior
- Unified analysis for multiple control-plane protocols
- Can catch control-plane implementation bugs

Challenges with Real-Time Verification

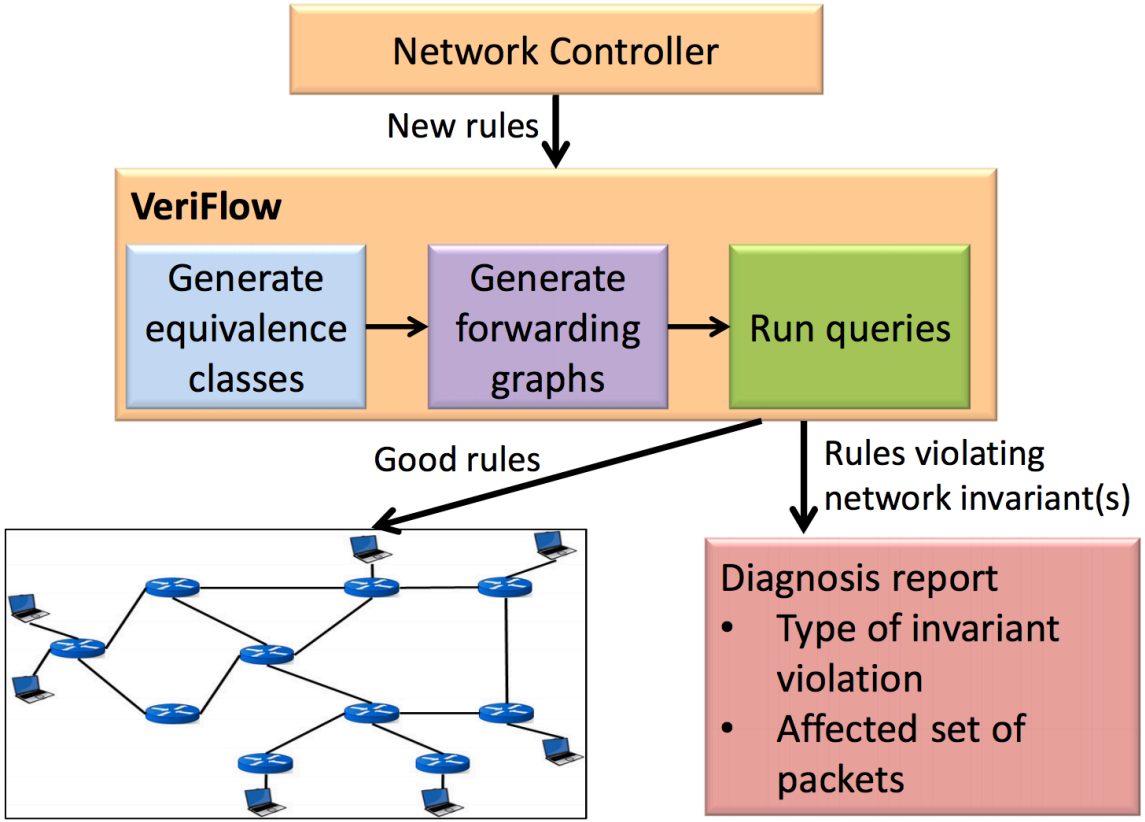
- Challenge 1: Obtaining real-time view of network
  - Solution: Utilize the **centralized** data-plane view available in an **SDN (Software-Defined Network)**
- Challenge 2: Verification speed
  - Solution: Off-the-shelf techniques?

No, too slow!

Veriflow: Check Data-Plane State in Real-time

- VeriFlow checks network-wide invariants in **real time** using data-plane state
  - Absence of routing loops and black holes, access control violations, etc.
- VeriFlow functions by
  - Monitoring **dynamic changes** in the network
  - Constructing a **model** of the **network behavior**
  - Using **custom algorithms** to automatically derive whether the network contains errors

VeriFlow Operation



Three Steps

- Limit search space
  - Packets experiencing same forwarding actions throughout the network are an equivalence class
- Represent forwarding behavior
  - Reresented as forwarding graphs
- Run query to check invariants
  - Produce types of invariant actions