p4v | Practical Verification for Programmable Data Planes

Jed Liu
Bill Hallahan
Cole Schlesinger
Milad Sharif
Jeongkeun Lee

Robert Soulé
Han Wang
Călin Caşcaval
Nick McKeown
Nate Foster





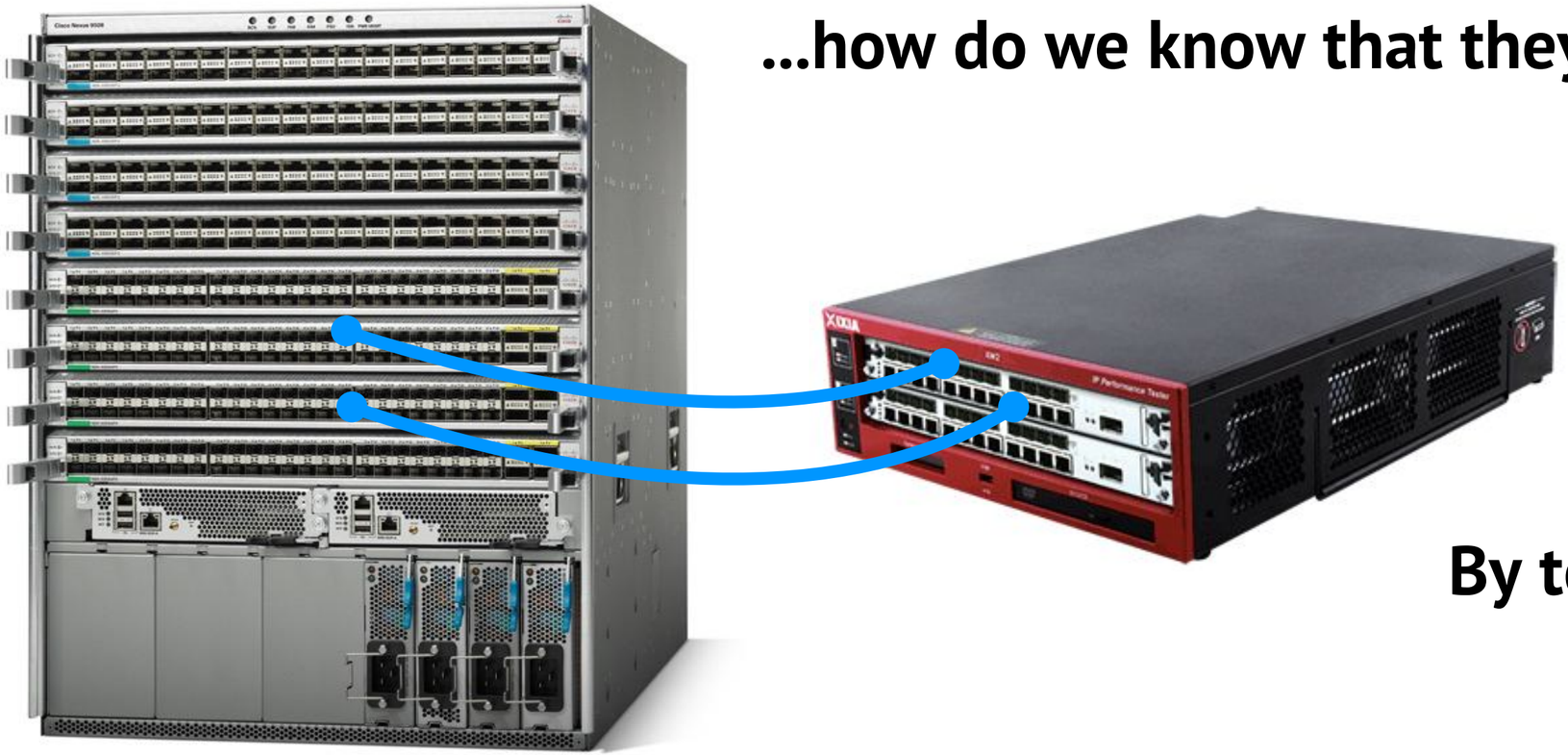
Fixed-function routers…

Fixed-function routers...

...how do we know that they work?

(with apologies to Insane Clown Posse)

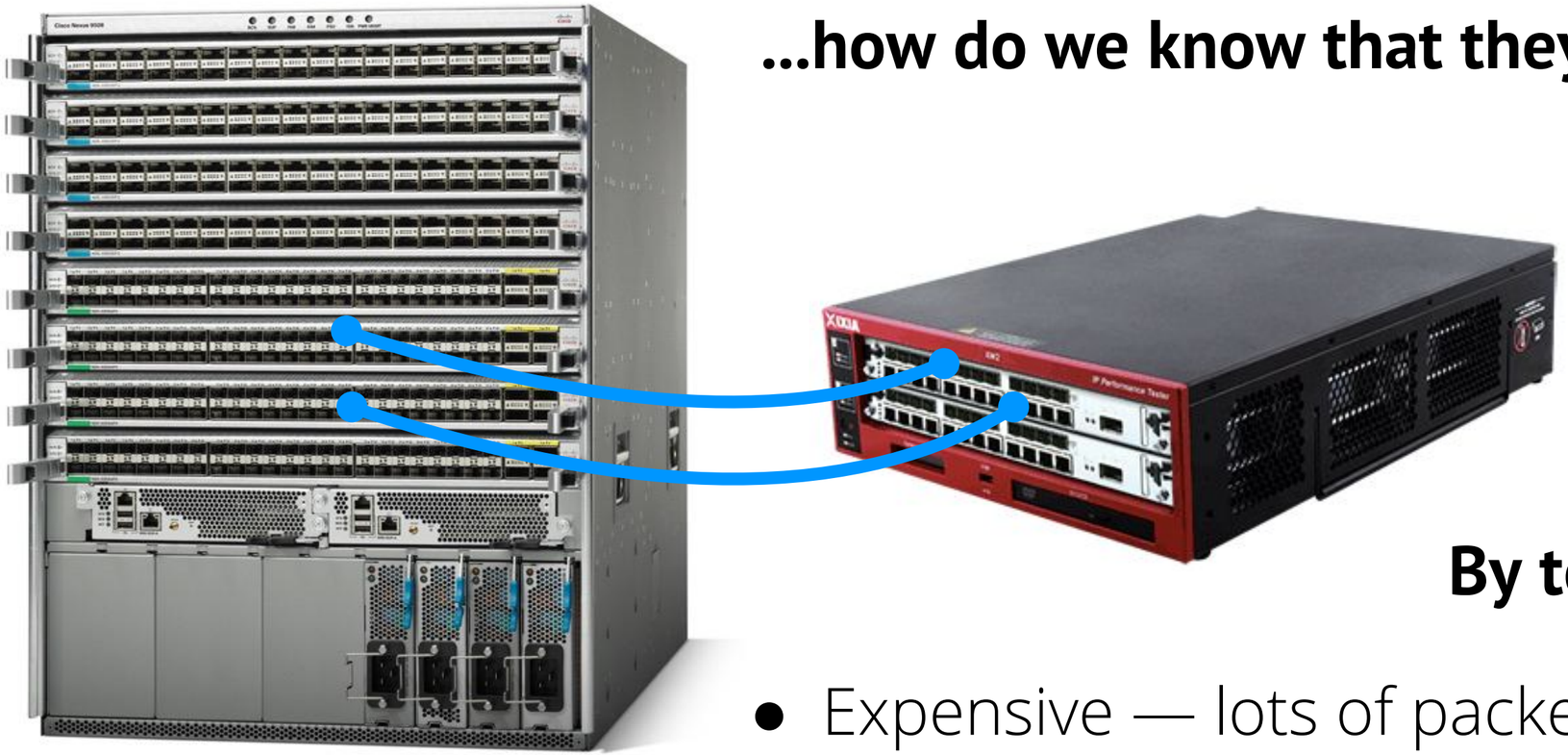
Fixed-function routers...



...how do we know that they work?

By testing!

Fixed-function routers…



…how do we know that they work?

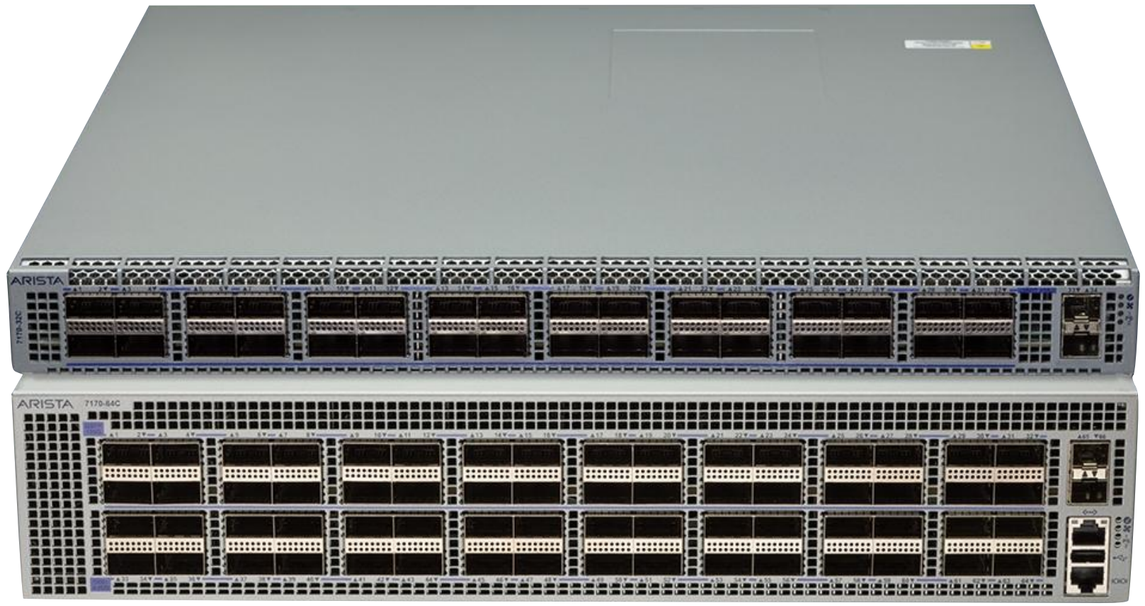
By testing!

- Expensive — lots of packet formats & protocols
- Pay cost once, during manufacturing

Programmable routers...

(specifically, programmable data planes)

...how do they work?

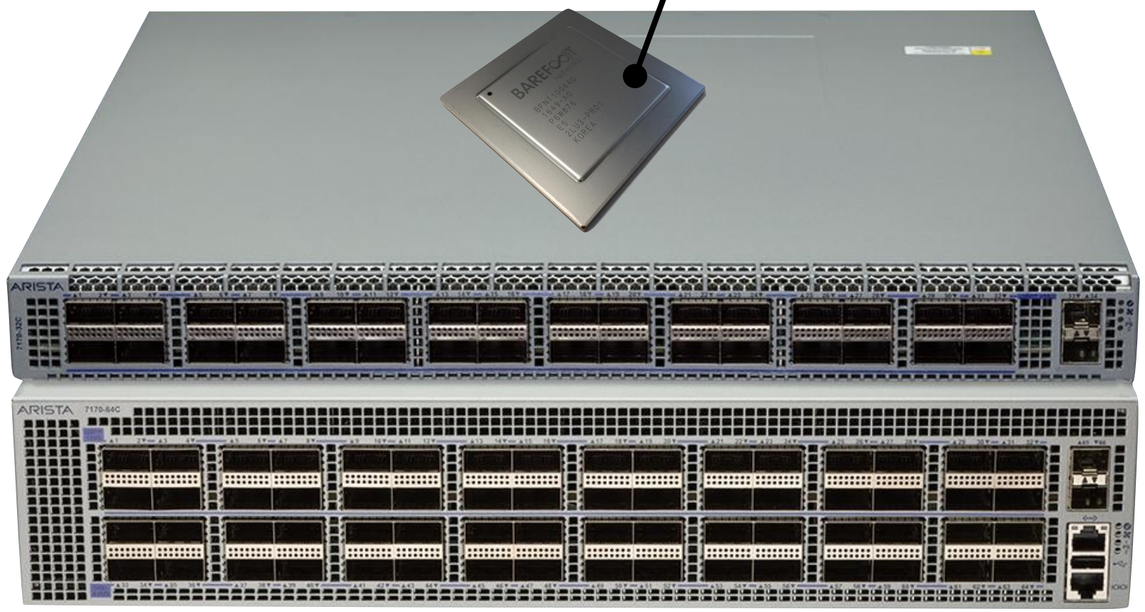


Arista 7170 series switches

Programmable routers…

(specifically, programmable data planes)

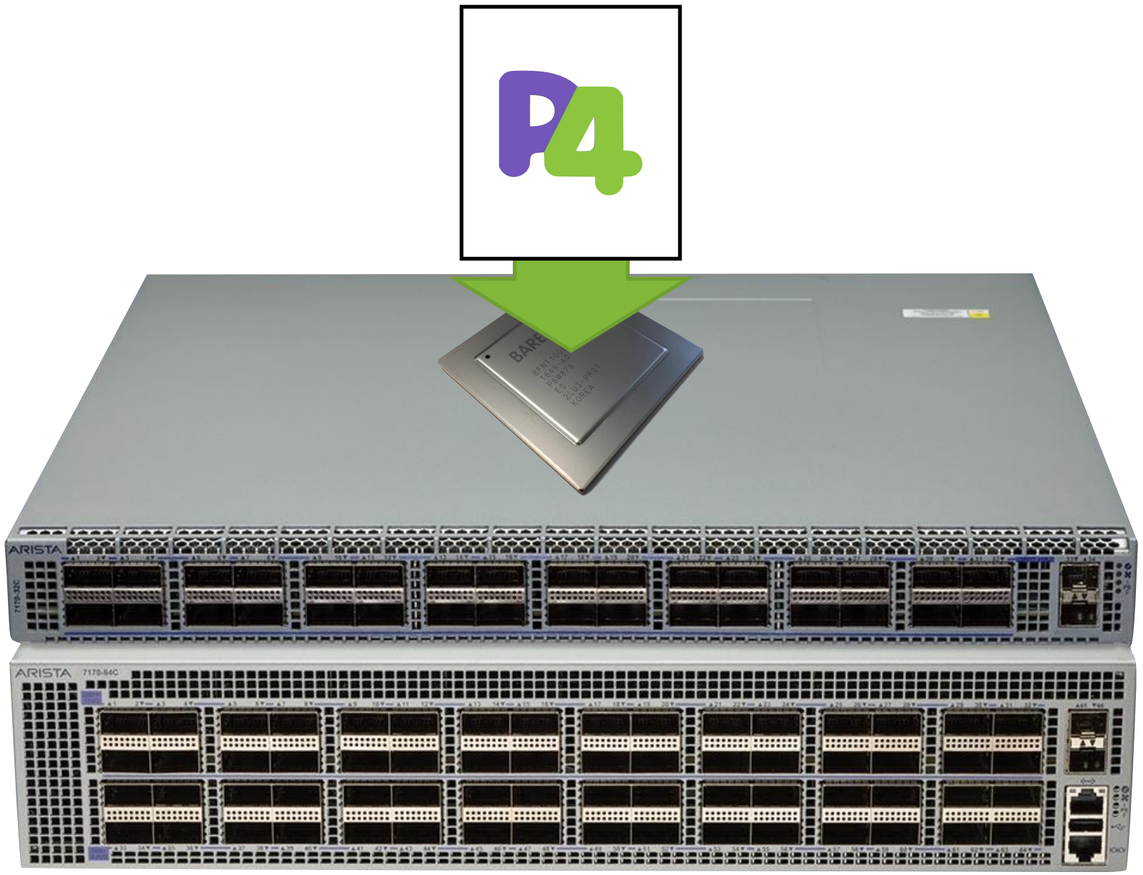
Barefoot Tofino chip



Arista 7170 series switches

…how do they work?

Programmable routers…



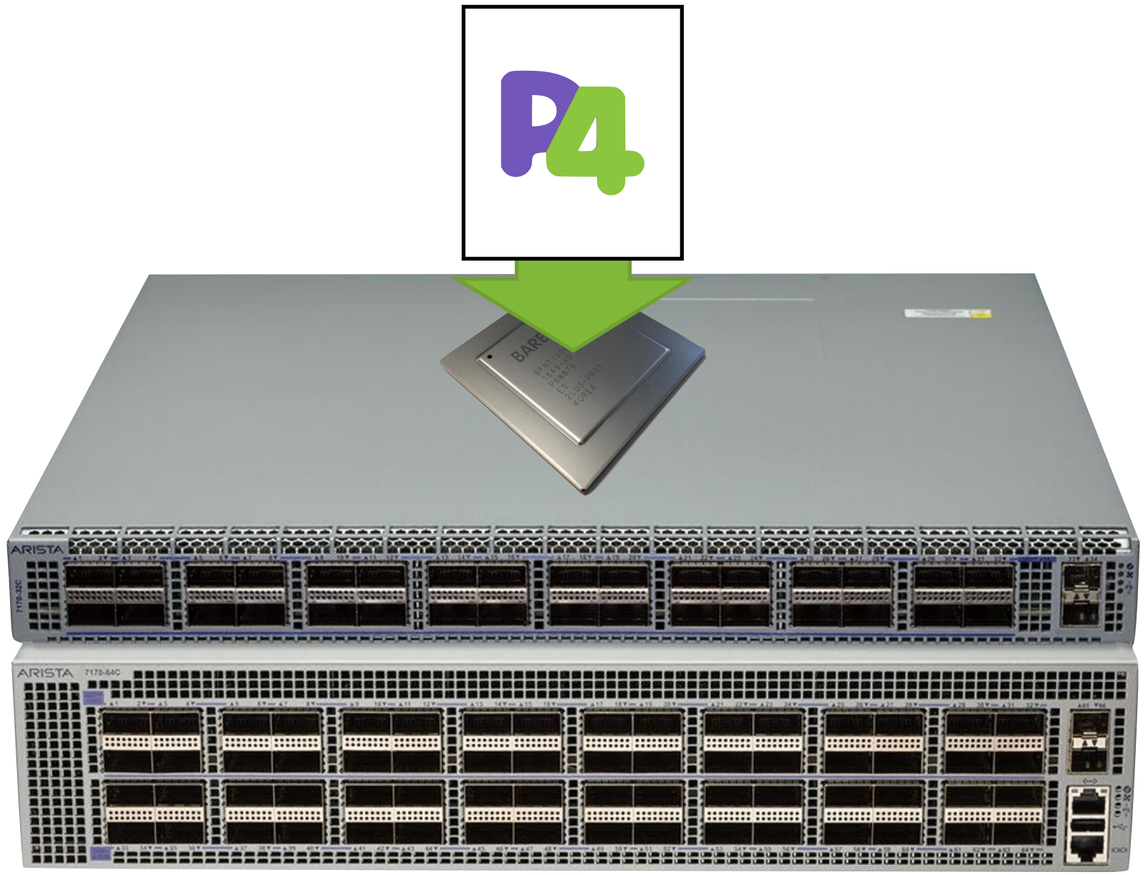
Arista 7170 series switches

(specifically, programmable data planes)

…how do they work?

Programmable routers…

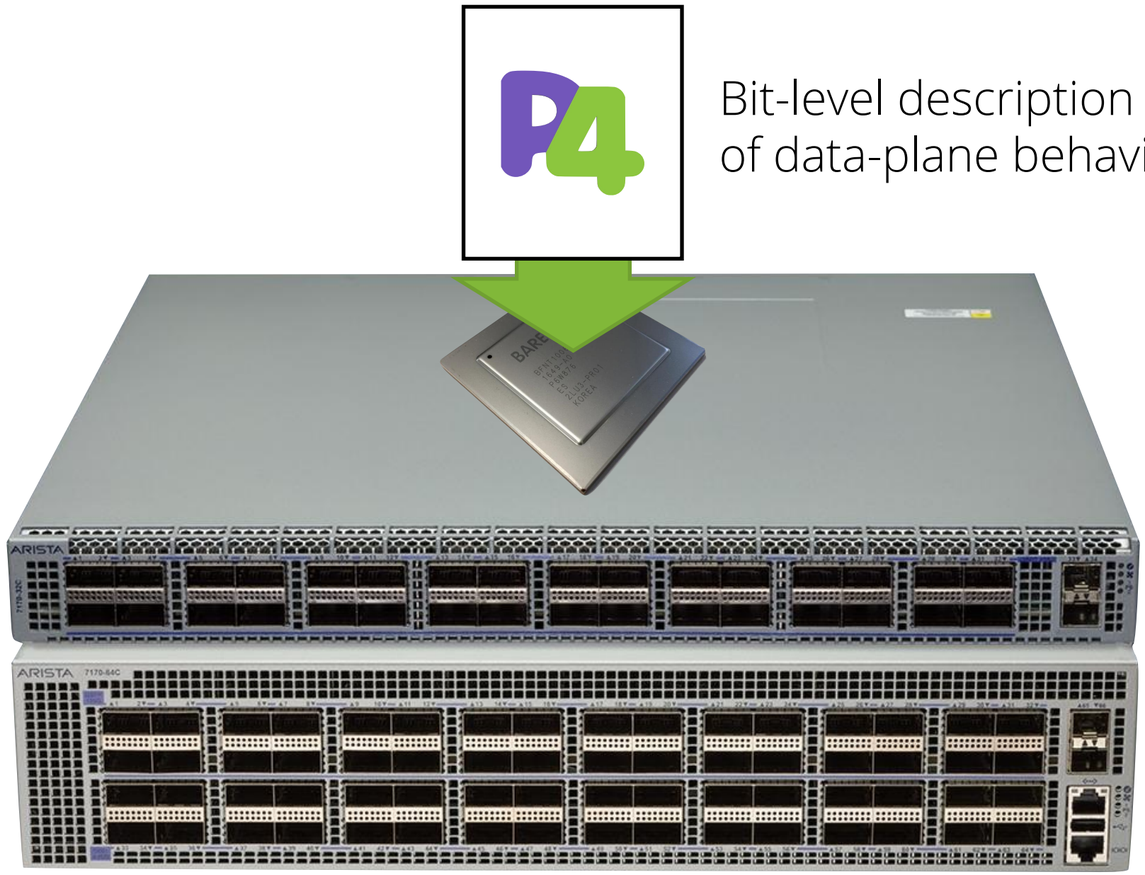
(specifically, programmable data planes)



Arista 7170 series switches

- New hotness
  - Rapid innovation
  - Novel uses of network
    - ✦ In-band network telemetry
    - ✦ In-network caching
- No longer have economy of scale for traditional testing

Let’s verify!



Arista 7170 series switches

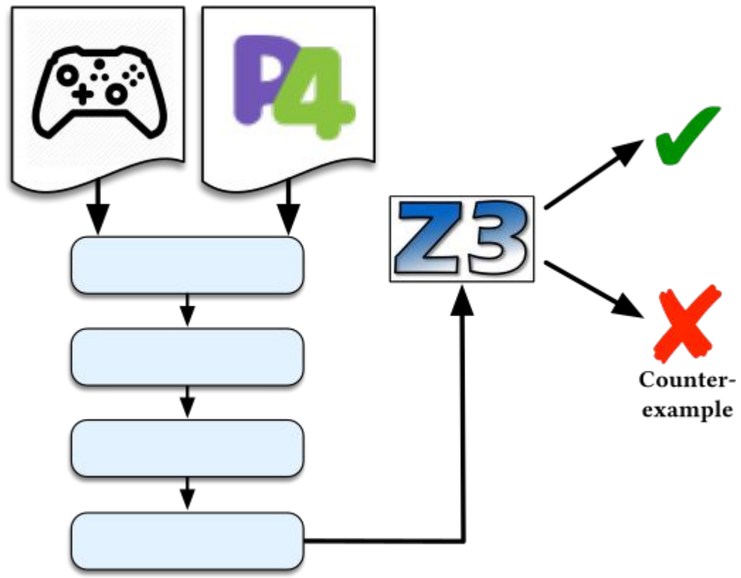
Bit-level description
of data-plane behaviour

Give programmers language-based verification tools

P4 also used as HDL for fixed-function devices

p4v overview

- **Automated** tool for verifying P4 programs
- Considers **all paths**
  - But also practical for **large programs**
- Includes basic safety properties for any program
- **Extensible** framework
  - Verify custom, program-specific properties
  - Assert-style debugging



Anatomy of a P4 program

```
/* Headers and Instances */
header_type ethernet_t {
 fields {
  dst_addr:48;
  src_addr:48;
  ether_type:16;
 }
}
header_type ipv4_t {
 fields {
  pre_ttl:64;
  ttl:8;
  protocol:8;
  checksum:16;
  src_addr:32;
  dst_addr:32;
 }
}
header ethernet_t ethernet;
header ipv4_t ipv4;
```

Headers

```
/* Parsers */
parser start {
  extract(ethernet);
  return select(ethernet.ether_type) {
    0x800: parse_ipv4;
    default: ingress;
  }
}
parser parse_ipv4 {
  extract(ipv4);
  return ingress;
}
```

Parsers

Convert bitstreams into headers

```
/* Actions */
action allow() {
 modify_field(standard_metadata.egress_spec,1);
}
action deny() { drop(); }
action nop() { }
action rewrite(dst_addr) {
  modify_field(ipv4.dst_addr,dst_addr);
}
```

Actions

Modify headers, specify forwarding

```
/* Tables */
table acl {
 reads {
  ipv4.src_addr:lpm;
  ipv4.dst_addr:lpm;
 }
 actions { allow; deny; }
}

table nat {
  reads { ipv4.dst_addr:lpm; }
  actions { rewrite; nop; }
  default_action: nop();
}
```

Tables

Apply actions based on header data

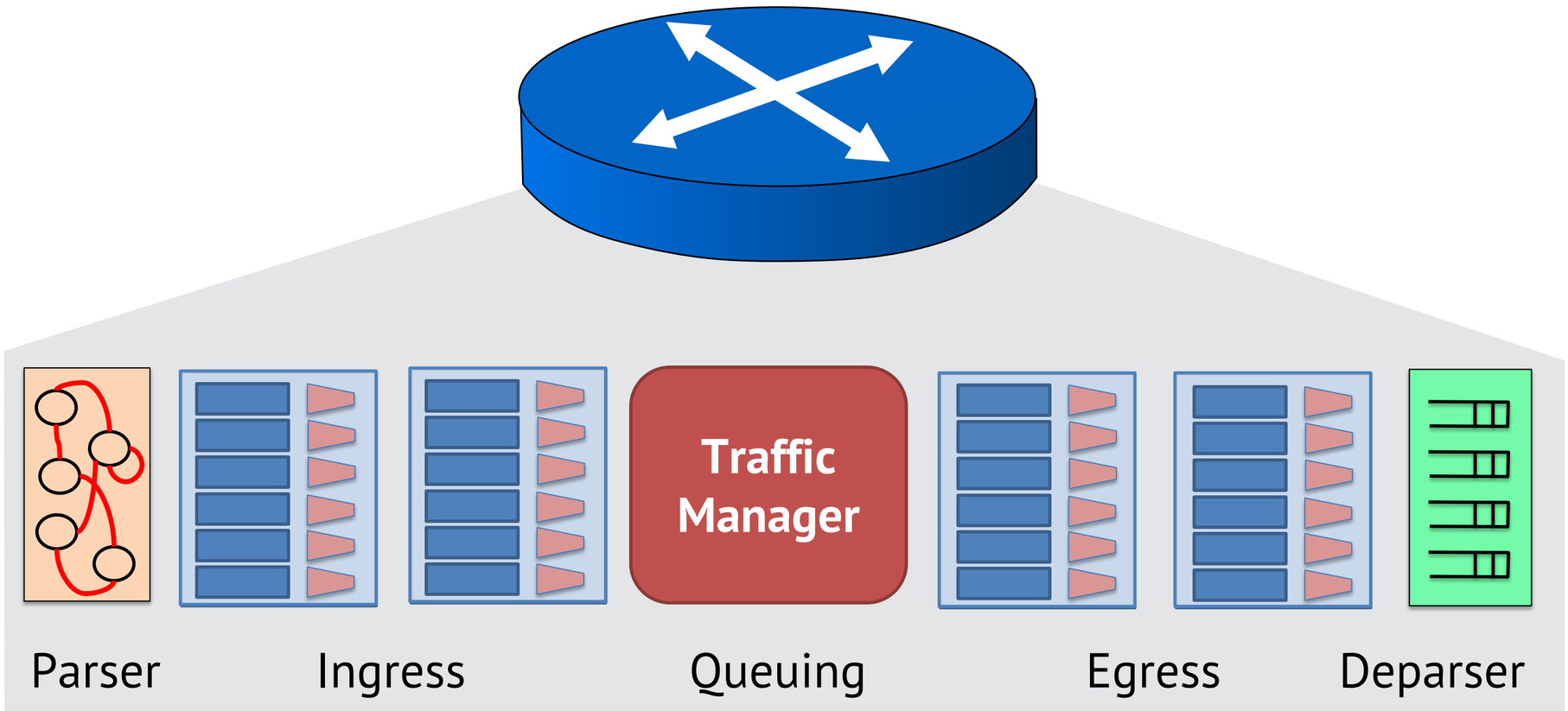
```
/* Controls */
control ingress {
 apply(nat);
 apply(acl);
}

control egress { }
```

Controls

Sequences of tables

P4 hardware model

PISA [SIGCOMM 2013]
Protocol-Independent Switch Architecture



P4 by example

- P4 is a low-level language → many gotchas
- Let’s explore by example!
  - IPv6 router w/ access control list (ACL)

P4 by example

- P4 is a low-level language → many gotchas
- Let's explore by example!
  - IPv6 router w/ access control list (ACL)

```
control ingress { apply(acl); }

table acl {


}
```

P4 by example

- P4 is a low-level language → many gotchas
- Let's explore by example!
  - IPv6 router w/ access control list (ACL)

```
control ingress { apply(acl); }

table acl {
  reads { ipv6.dstAddr: lpm; }
  actions { allow; deny; }
}
```

P4 by example

- P4 is a low-level language → many gotchas
- Let's explore by example!
  - IPv6 router w/ access control list (ACL)

```
control ingress { apply(acl); }

table acl {
  reads { ipv6.dstAddr: lpm; }
  actions { allow; deny; }
}

action allow() {
  modify_field(std_meta.egress_spec, 1);
}
```

P4 by example

- P4 is a low-level language → many gotchas
- Let's explore by example!
  - IPv6 router w/ access control list (ACL)

```
control ingress { apply(acl); }

table acl {
  reads { ipv6.dstAddr: lpm; }
  actions { allow; deny; }
}

action allow() {
  modify_field(std_meta.egress_spec, 1);
}

action deny() { drop(); }
```

P4 by example

- P4 is a low-level language → many gotchas
- Let's explore by example!
  - IPv6 router w/ access control list (ACL)

```
control ingress { apply(acl); }

table acl {
  reads { ipv6.dstAddr: lpm; }
  actions { allow; deny; }
}

action allow() {
  modify_field(std_meta.egress_spec, 1);
}

action deny() { drop(); }
```

What could *possibly* go wrong?

What if we didn't receive an IPv6 packet?

`ipv6` header will be **invalid**

What goes wrong

Table reads arbitrary values

→ Intended ACL policy violated

```
control ingress { apply(acl); }

table acl {
  reads { ipv6.dstAddr: lpm; }
  actions { allow; deny; }
}

action allow() {
  modify_field(std_meta.egress_spec, 1);
}

action deny() { drop(); }
```

What if we didn't receive an IPv6 packet?

`ipv6` header will be **invalid**

What goes wrong

Table reads arbitrary values

→ Intended ACL policy violated

Can read values from a previous packet

→ Side channel vulnerability!

```
control ingress { apply(acl); }

table acl {
  reads { ipv6.dstAddr: lpm; }
  actions { allow; deny; }
}

action allow() {
  modify_field(std_meta.egress_spec, 1);
}

action deny() { drop(); }
```

What if we didn't receive an IPv6 packet?

`ipv6` header will be **invalid**

What goes wrong

Table reads arbitrary values

→ Intended ACL policy violated

Can read values from a previous packet

→ Side channel vulnerability!

Real programs are complicated:
hard to keep validity in your head

```
control ingress { apply(acl); }

table acl {
  reads { ipv6.dstAddr: lpm; }
  actions { allow; deny; }
}

action allow() {
  modify_field(std_meta.egress_spec, 1);
}

action deny() { drop(); }
```

Property #1: header validity

What if `acl` table misses (no rule matches)?

Forwarding decision is unspecified

What goes wrong

Forwarding behaviour depends on hardware

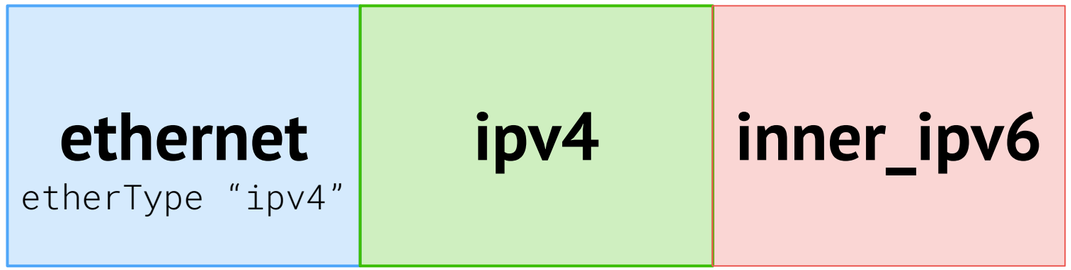
- May not do what you expect!
- Code not portable

```
control ingress { apply(acl); }

table acl {
  reads { ipv6.dstAddr: lpm; }
  actions { allow; deny; }
}

action allow() {
  modify_field(std_meta.egress_spec, 1);
}

action deny() { drop(); }
```

Property #2: unambiguous forwarding

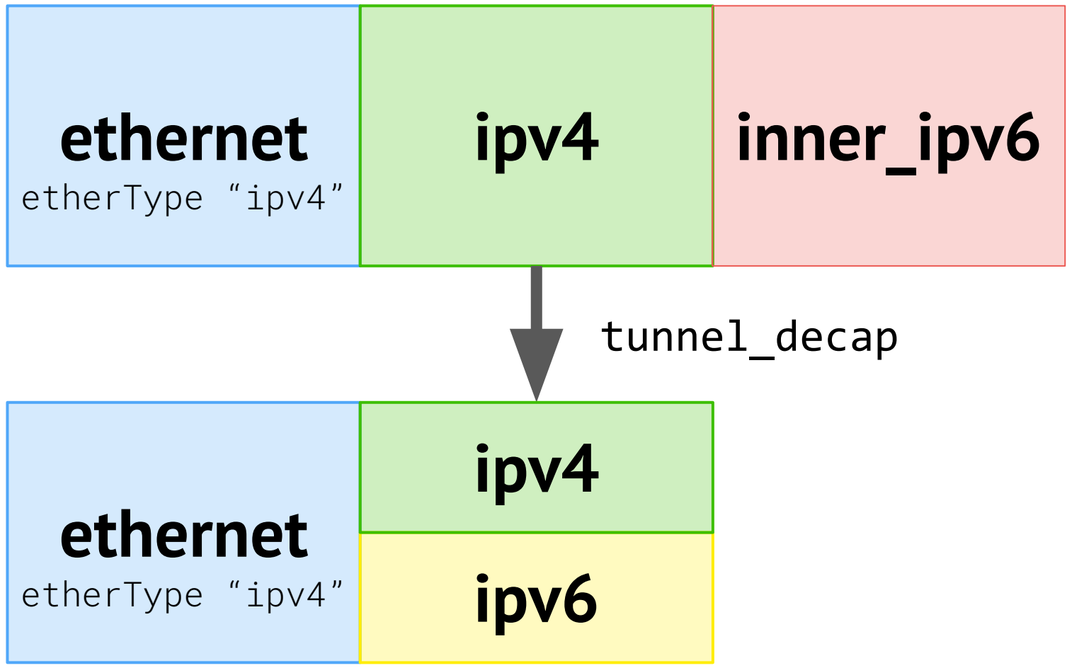
Let’s add 6in4 tunnelling!

```
table tunnel_decap {
  ...
  actions { decap_6in4; }
}

action decap_6in4() {
  copy_header(ipv6, inner_ipv6);
  remove_header(inner_ipv6);
}

table tunnel_term {
  ...
  actions { term_6in4; }
}

action term_6in4() {
  remove_header(ipv4);
  modify_field(ethernet.etherType, 0x86dd);
}
```



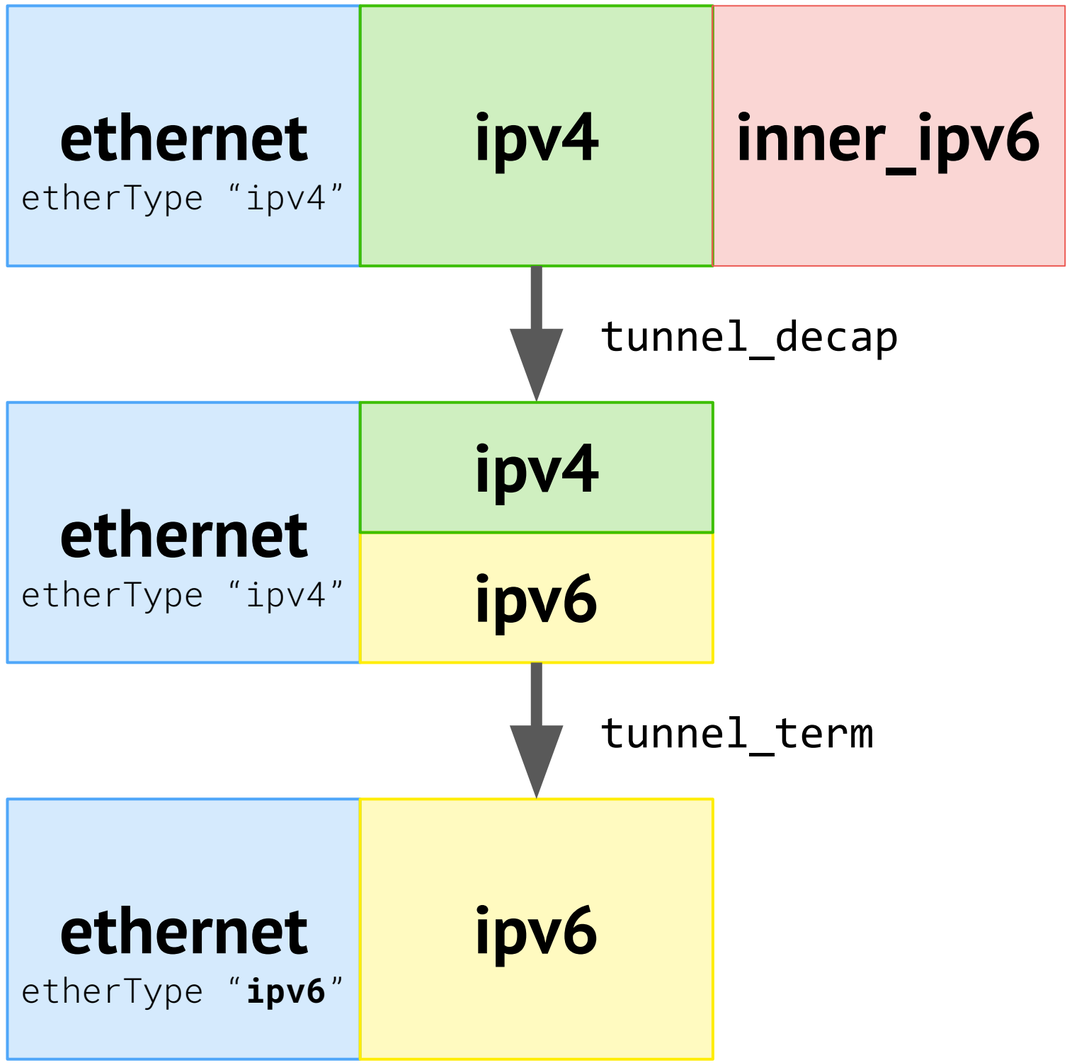
Let’s add 6in4 tunnelling!

```
table tunnel_decap {
  ...
  actions { decap_6in4; }
}

action decap_6in4() {
  copy_header(ipv6, inner_ipv6);
  remove_header(inner_ipv6);
}

table tunnel_term {
  ...
  actions { term_6in4; }
}

action term_6in4() {
  remove_header(ipv4);
  modify_field(ethernet.etherType, 0x86dd);
}
```

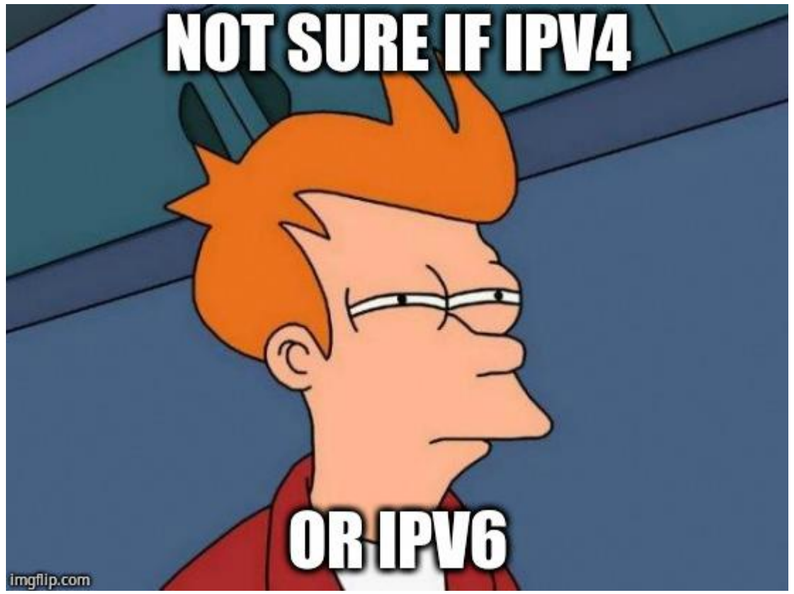


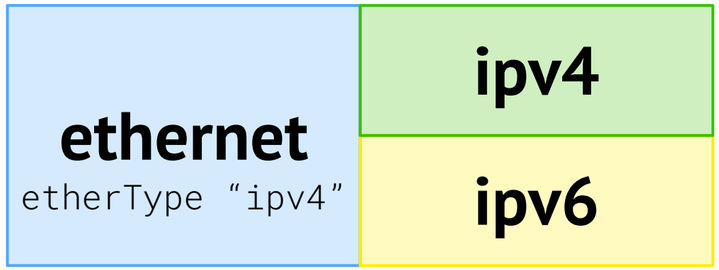
Let’s add 6in4 tunnelling!

```
table tunnel_decap {
  ...
  actions { decap_6in4; }
}

action decap_6in4() {
  copy_header(ipv6, inner_ipv6);
  remove_header(inner_ipv6);
}

table tunnel_term {
  ...
  actions { term_6in4; }
}

action term_6in4() {
  remove_header(ipv4);
  modify_field(ethernet.etherType, 0x86dd);
}
```



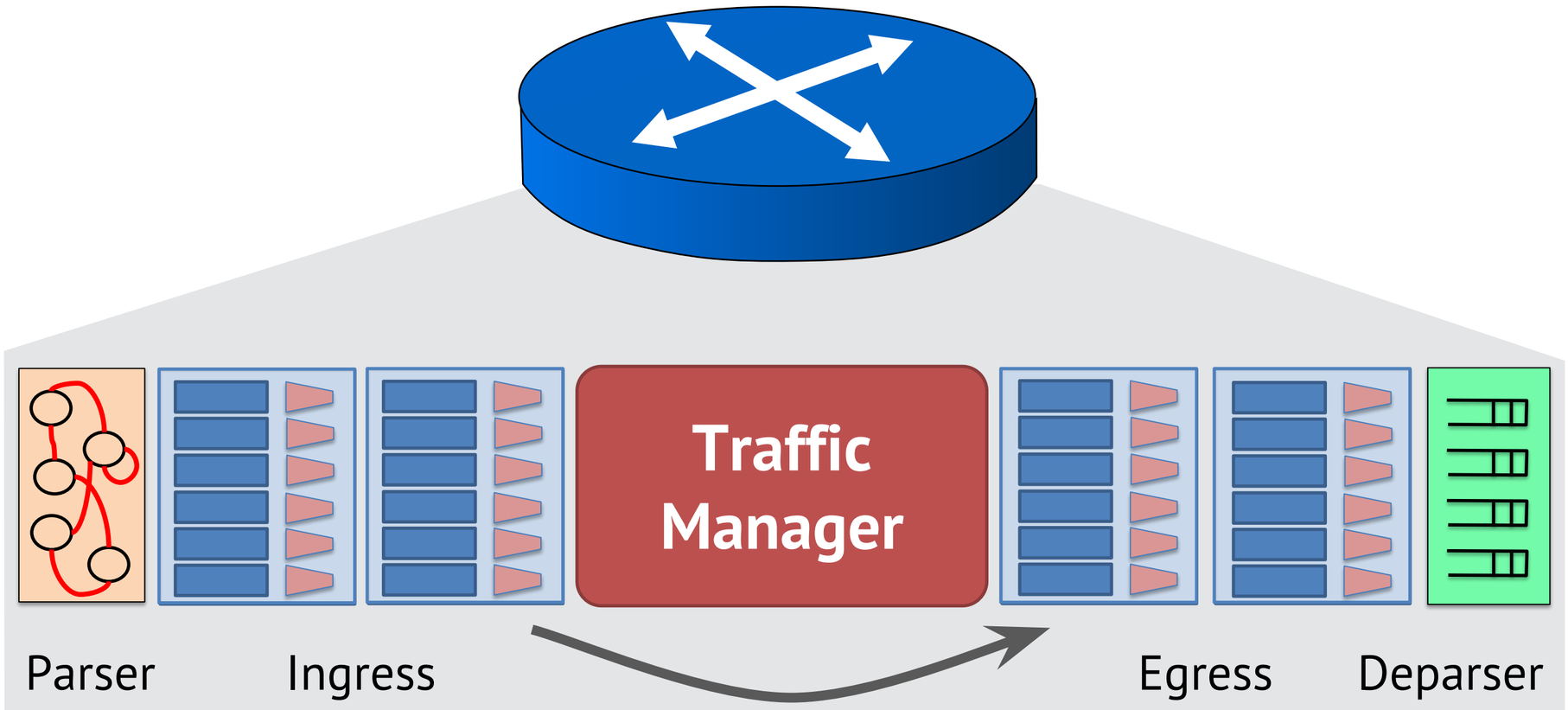
Let’s add 6in4 tunnelling!





A look behind the curtain

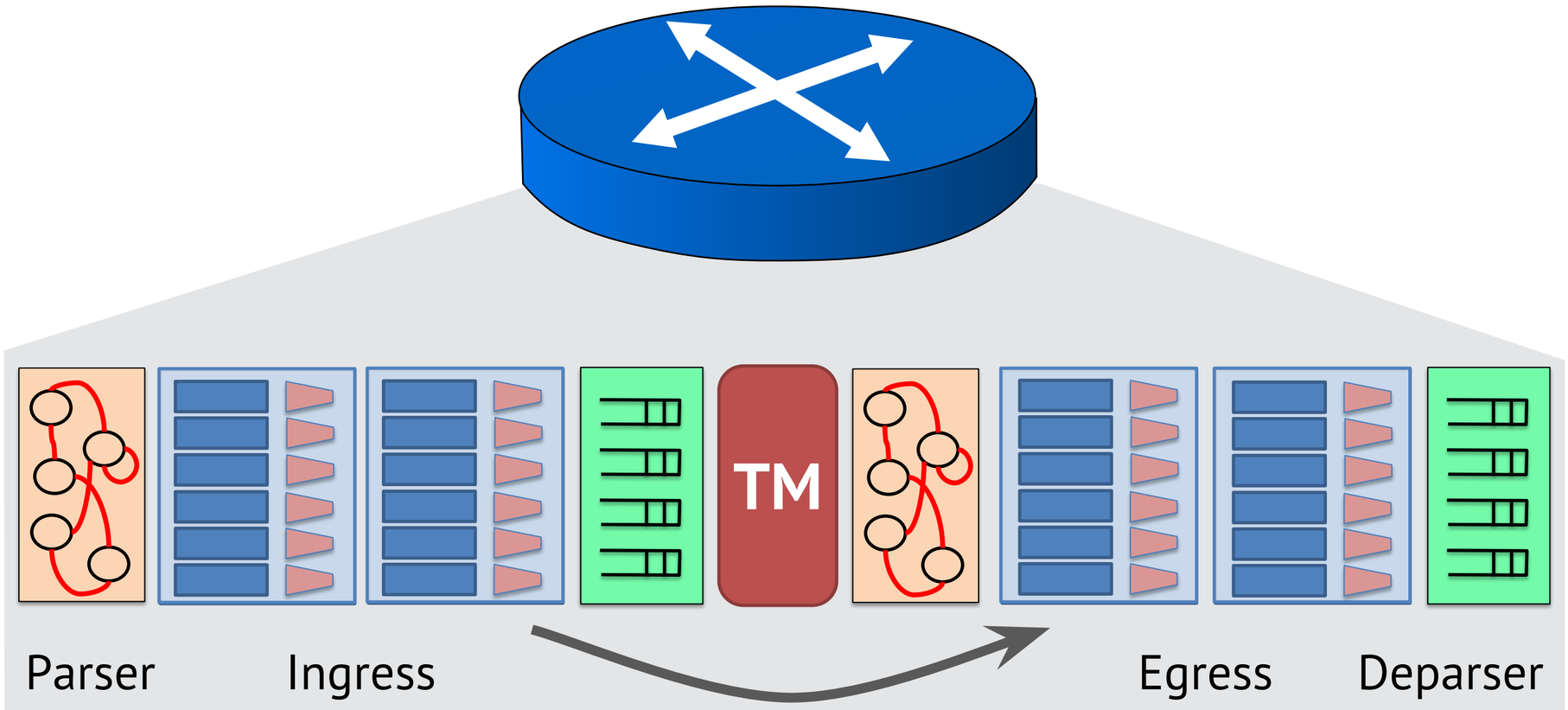
In PISA, state is copied verbatim from ingress to egress…



A look behind the curtain

In PISA, state is copied verbatim from ingress to egress…

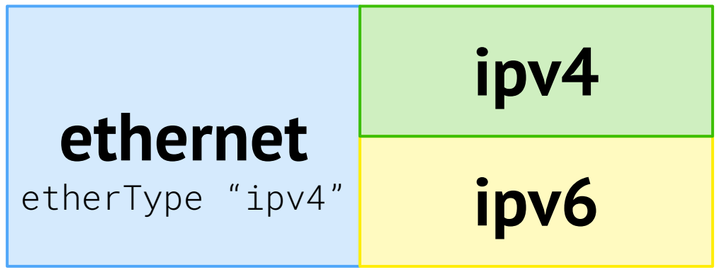
Some architectures use parser and deparser to bridge state!



What if the architecture reparses the packet?

- IPv4 and IPv6 are mutually exclusive protocols

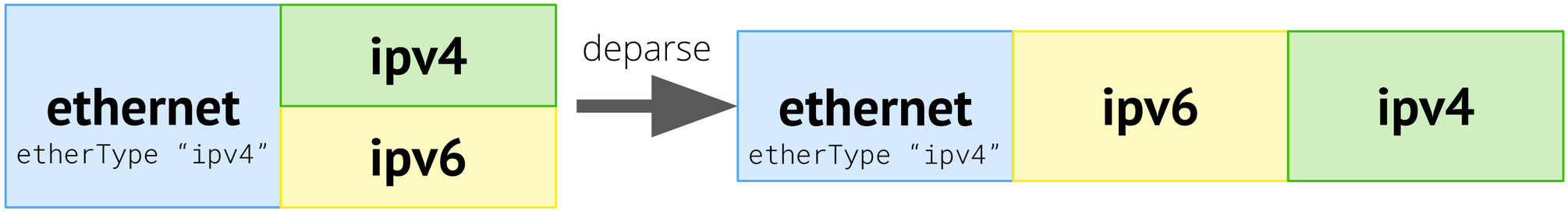
What goes wrong



What if the architecture reparses the packet?

- IPv4 and IPv6 are mutually exclusive protocols

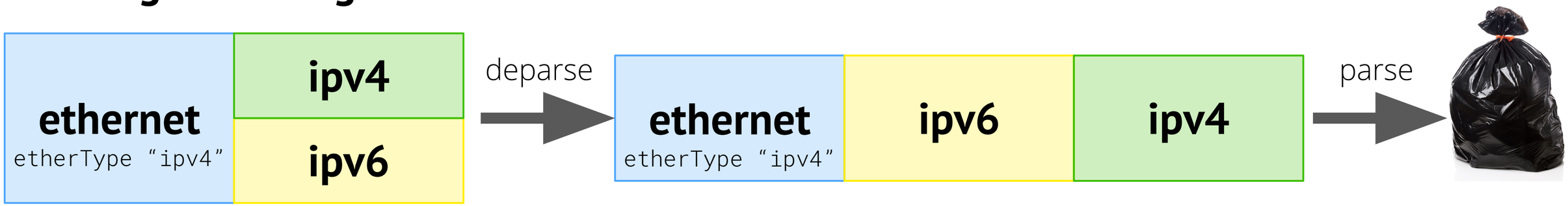
What goes wrong



What if the architecture reparses the packet?

- IPv4 and IPv6 are mutually exclusive protocols

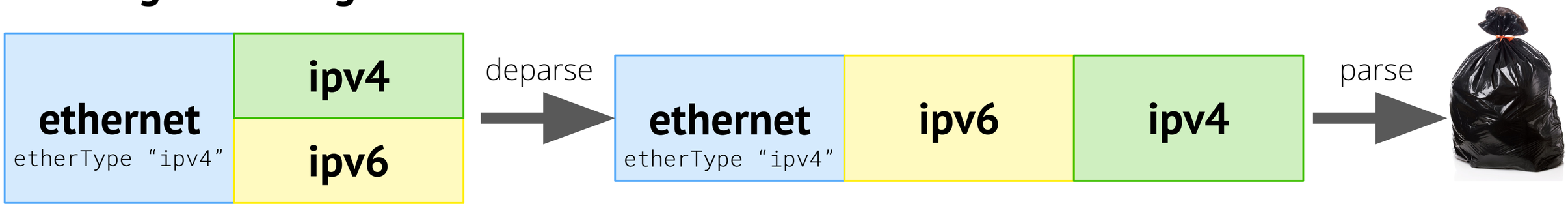
What goes wrong



What if the architecture reparses the packet?

- IPv4 and IPv6 are mutually exclusive protocols

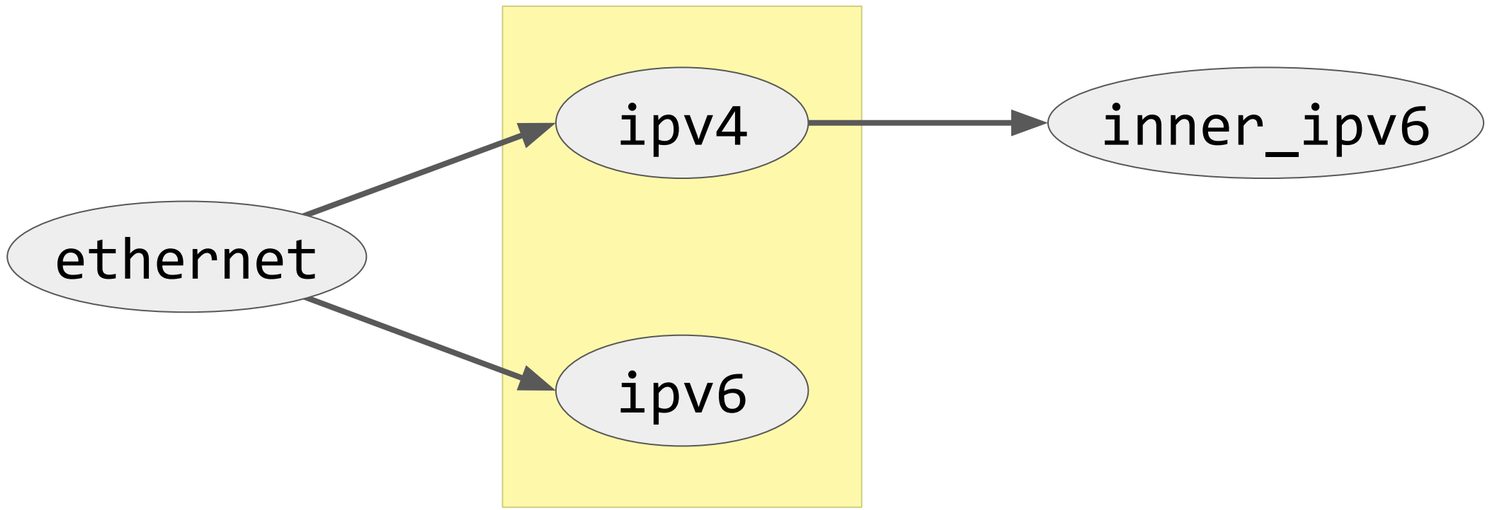
What goes wrong



Property #3: reparseability

Another look behind the curtain

- Hardware devices have limited resources
- Compilers have options to improve resource usage
  - e.g., if headers are mutually exclusive in parser, assume they stay mutually exclusive in rest of program
  - Mutually exclusive headers can be overlaid in memory!



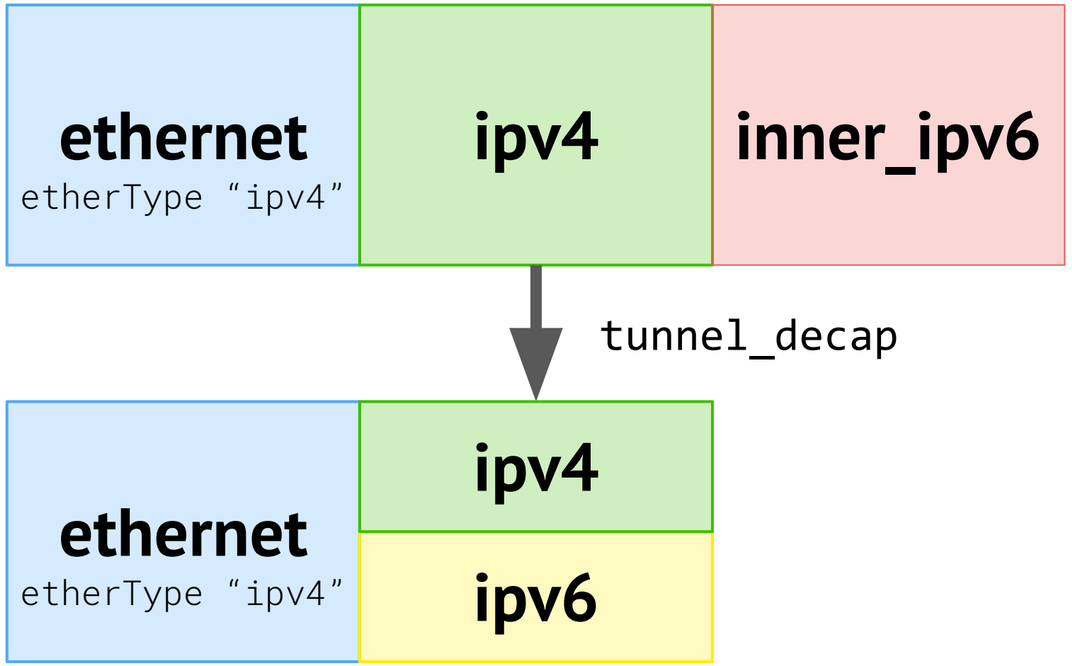
What if headers share memory?

- IPv4 and IPv6 might be overlaid

What goes wrong

Data corruption

- e.g., `tunnel_decap` clobbers `ipv4`



Parsers are complicated in practice

Hard to keep track of mutually exclusive states

Property #4: mutual exclusion of headers

Types of properties

General safety

- **Header validity**
- Arithmetic-overflow checking
- Index bounds checking (header stacks, registers, meters, ...)

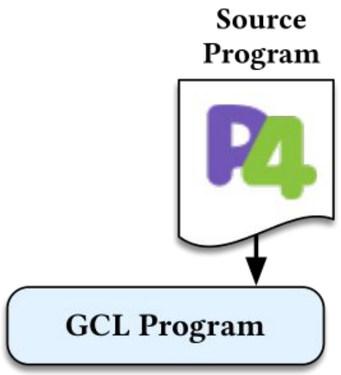
Architectural

- **Unambiguous forwarding**
- **Reparseability**
- **Mutual exclusion of headers**
- Correct metadata usage (e.g., read-only metadata)

Program-specific

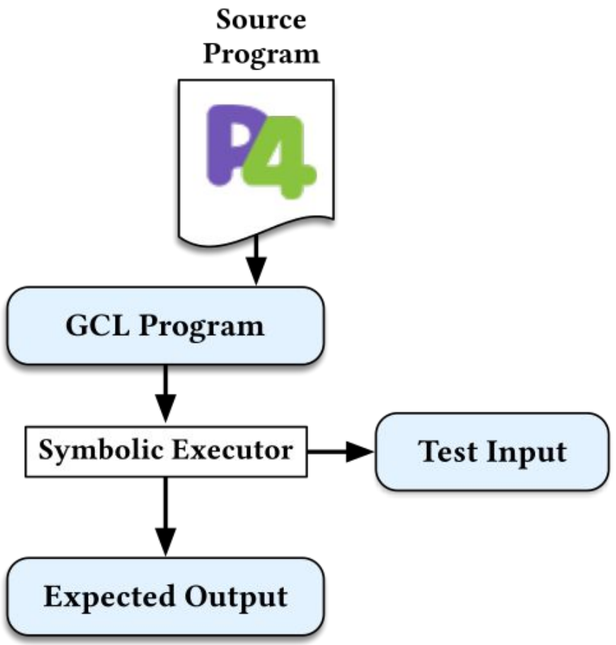
- Custom assertions in P4 program — e.g., IPv4 `ttl` correctly decremented

Challenge #1: imprecise semantics



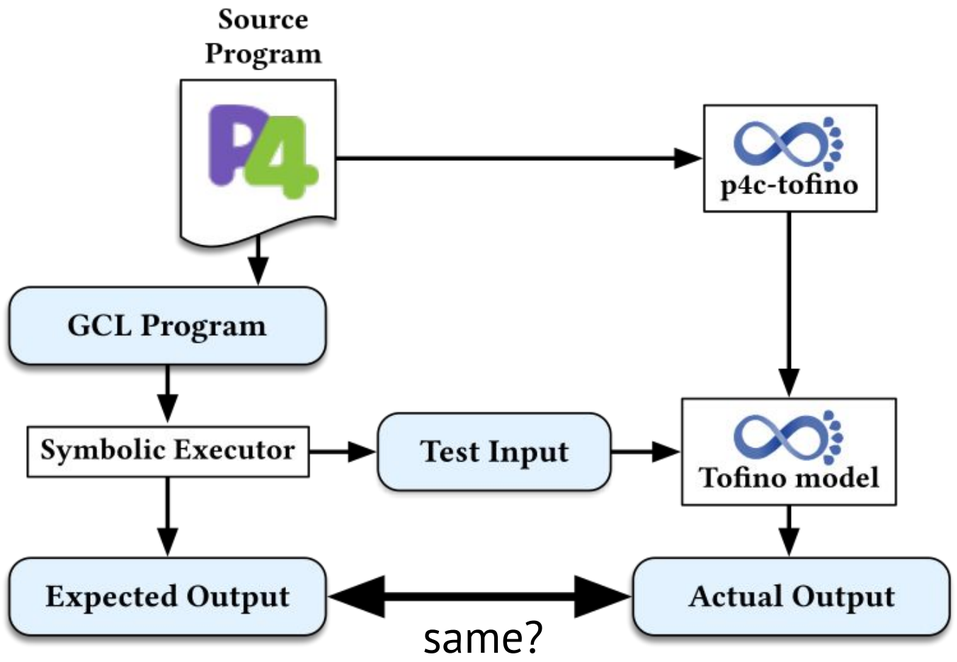
- P4 language spec doesn't give precise semantics
- Defined semantics by translation to GCL (a simple imperative language)

Challenge #1: imprecise semantics



- P4 language spec doesn't give precise semantics
- Defined semantics by translation to GCL (a simple imperative language)
- Tested semantics
  - Symbolically executed GCL to generate input–output tests for several programs

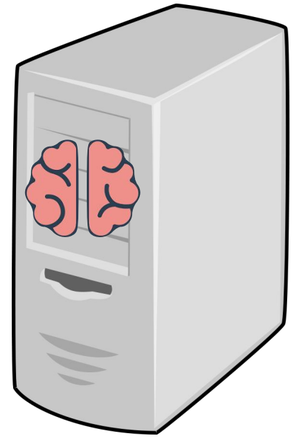
Challenge #1: imprecise semantics

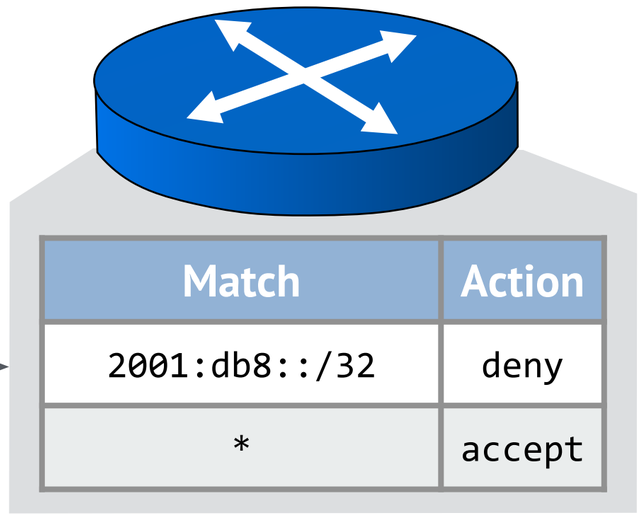


- P4 language spec doesn't give precise semantics
- Defined semantics by translation to GCL (a simple imperative language)
- Tested semantics
  - Symbolically executed GCL to generate input–output tests for several programs
  - Ran w/ Barefoot P4 compiler & Tofino simulator

Challenge #2: modelling the control plane

- A P4 program is just half the program
  - Table rules are not statically known
  - Populated by the control plane at run time



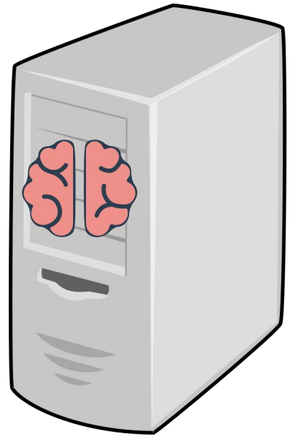


```
table acl {
  reads {
    ipv6.dstAddr: lpm;
  }
  actions { allow; deny; }
}
```

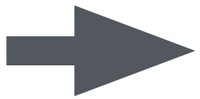
Match	Action
2001:db8::/32	deny
*	accept

Challenge #2: modelling the control plane

- A P4 program is just half the program
  - Table rules are not statically known
  - Populated by the control plane at run time



```
table acl {
  reads {
    ipv6.dstAddr: lpm;
  }
  actions { allow; deny; }
}
```

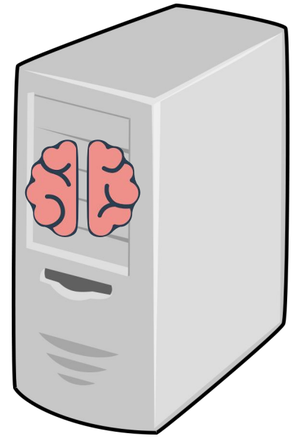


```
   ( @[ Action ] acl <hit> (allow);
     std_meta.egress_spec := 1)

[] ( @[ Action ] acl <hit> (deny);
     std_meta.egress_spec := 511)

[]   @[ Action ] acl <miss>
```

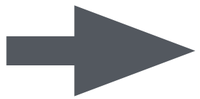
Tables translated into *unconstrained* nondeterministic choice

Challenge #2: modelling the control plane

- A P4 program is just half the program
  - Table rules are not statically known
  - Populated by the control plane at run time
- Control planes are carefully programmed
  - Tables rarely take arbitrary actions
- To rule out false positives, need to model behaviour of control plane



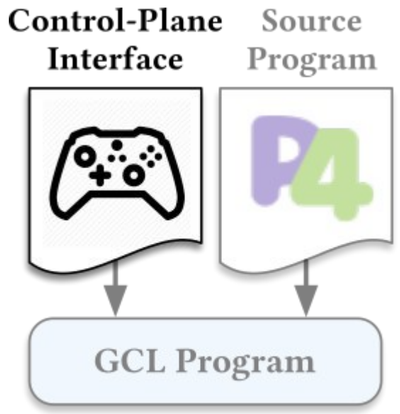
```
table acl {
  reads {
    ipv6.dstAddr: lpm;
  }
  actions { allow; deny; }
}
```



```
    ( @[ Action ] acl <hit> (allow);
      std_meta.egress_spec := 1)

[] ( @[ Action ] acl <hit> (deny);
      std_meta.egress_spec := 511)

[]   @[ Action ] acl <miss>
```

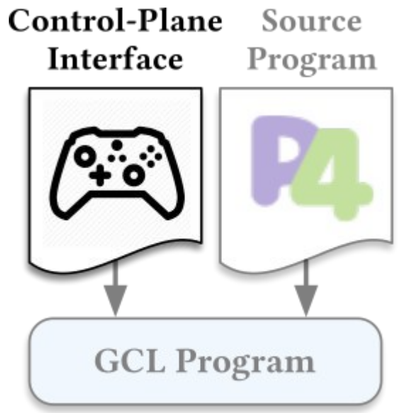
Tables translated into *unconstrained* nondeterministic choice

Control-plane interface



- Given as second input to p4v
- Constrains choices made by tables
- Written in domain-specific syntax

Control-plane interface

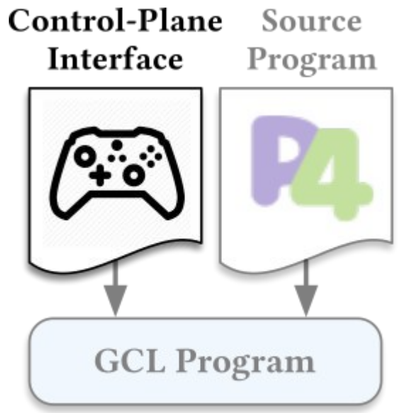


- Given as second input to p4v
- Constrains choices made by tables
- Written in domain-specific syntax

```
table acl {
  reads {
    ipv6.dstAddr: lpm;
  }
  actions { allow; deny; }
}
```

```
assume
  reads(acl, ipv6.dstAddr) == 2001:db8::/32
implies
  action(acl) == deny
```

Control-plane interface



- Given as second input to p4v
- Constrains choices made by tables
- Written in domain-specific syntax

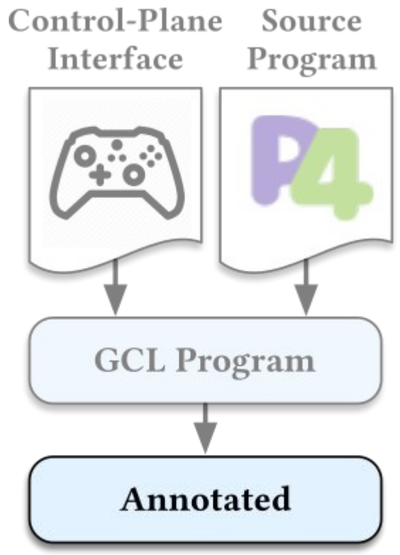
```
table acl {
  reads {
    ipv6.dstAddr: lpm;
  }
  actions { allow; deny; }
}
```

```
assume
  reads(acl, ipv6.dstAddr) == 2001:db8::/32
implies
  action(acl) == deny
```

```
table tunnel_decap {
  ...
  actions { decap_6in4; }
}

table tunnel_term {
  ...
  actions { term_6in4; }
}
```

```
assume
  action(tunnel_decap) == decap_6in4
iff
  action(tunnel_term) == term_6in4
```

Challenge #3: annotation burden



Many verification tools require users to annotate both assumptions and assertions.

p4v can automatically generate assertions for many properties

Currently supported:

- Header validity
- Unambiguous forwarding
- Reparseability
- All valid headers deparsed
- Expression definedness
- Index bounds

Challenge #4: handling large programs

- Not using compositional verification
  - High burden: needs annotations at component boundaries
- Not using symbolic execution
  - Exponential path explosion → explicitly exploring paths is not tractable

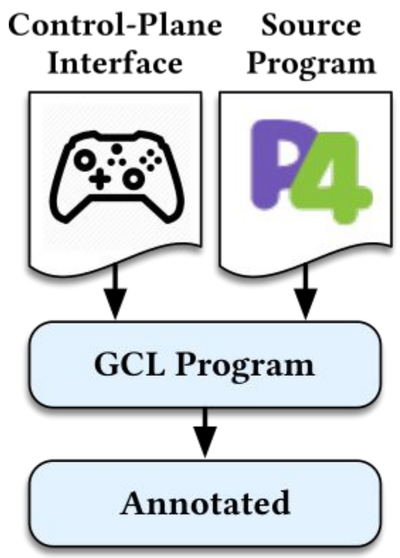
Challenge #4: handling large programs

- Not using compositional verification
  - High burden: needs annotations at component boundaries
- Not using symbolic execution
  - Exponential path explosion → explicitly exploring paths is not tractable
- Instead, generate **single logical formula** (a verification condition)
  - Formula valid ⇔ program satisfies assertions on **all execution paths**
  - Hand formula to solver → verification success or **counterexample**

Challenge #4: handling large programs

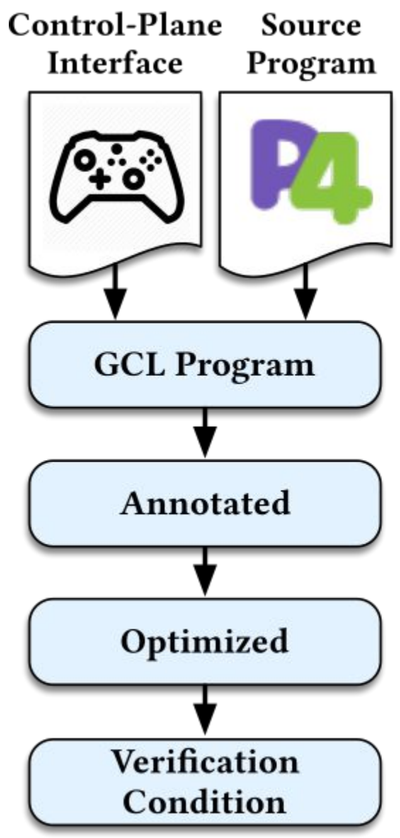
- Not using compositional verification
  - High burden: needs annotations at component boundaries
- Not using symbolic execution
  - Exponential path explosion → explicitly exploring paths is not tractable
- Instead, generate **single logical formula** (a verification condition)
  - Formula valid ⇔ program satisfies assertions on **all execution paths**
  - Hand formula to solver → verification success or **counterexample**
- Also do standard optimizations
  - Constant folding / propagation
  - Dead-code elimination

p4v architecture



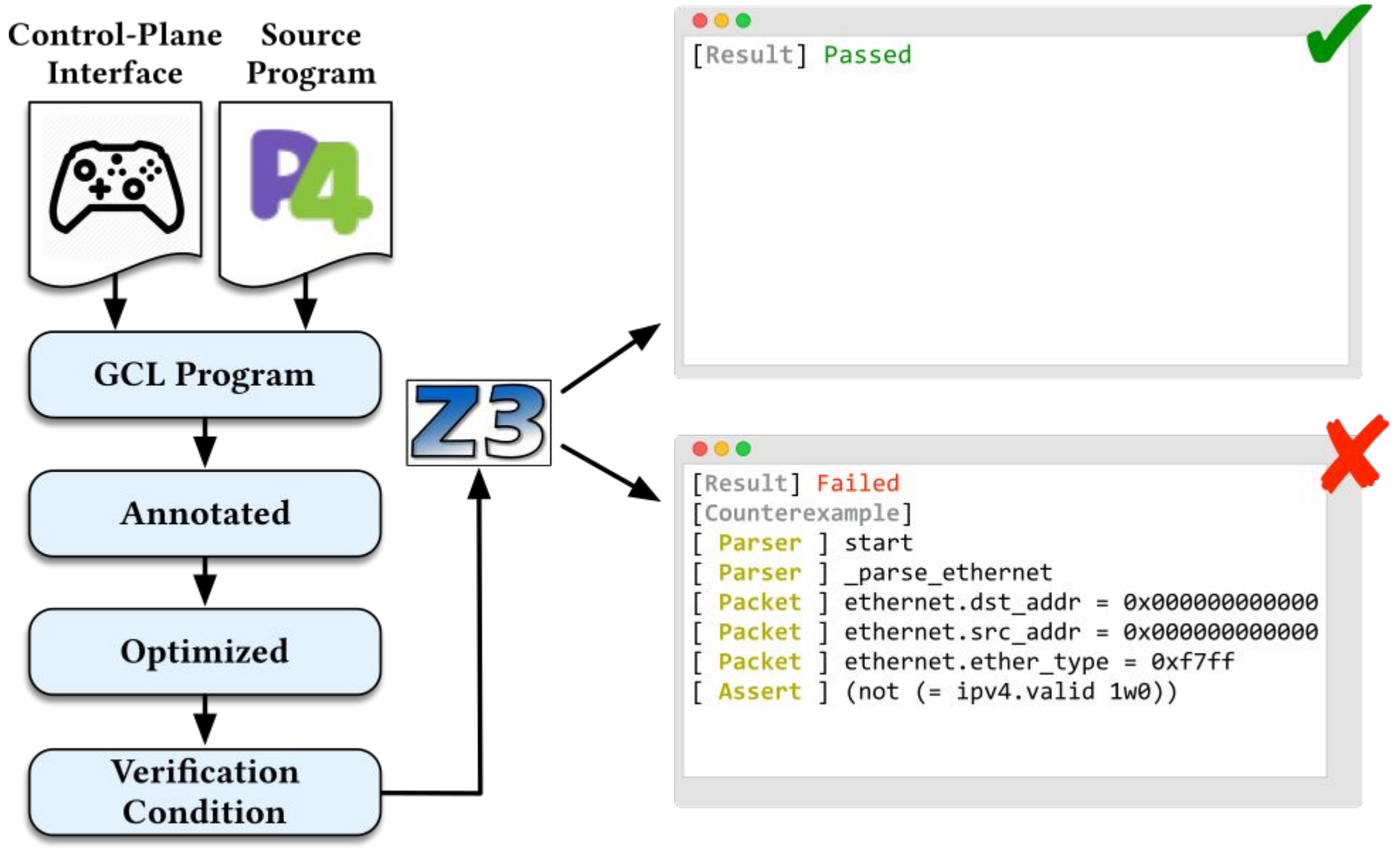
1. Start w/ CPI & P4 program
2. Translate to GCL
3. Auto-annotate w/ assertions

p4v architecture



1. Start w/ CPI & P4 program
2. Translate to GCL
3. Auto-annotate w/ assertions
4. Standard optimizations
5. Generate formula

p4v architecture



1. Start w/ CPI & P4 program
2. Translate to GCL
3. Auto-annotate w/ assertions
4. Standard optimizations
5. Generate formula
6. Send to Z3
7. Success or counterexample
   - Input packet
   - Program trace
   - Violated assertion

Evaluation: header validity in switch.p4

Statistics

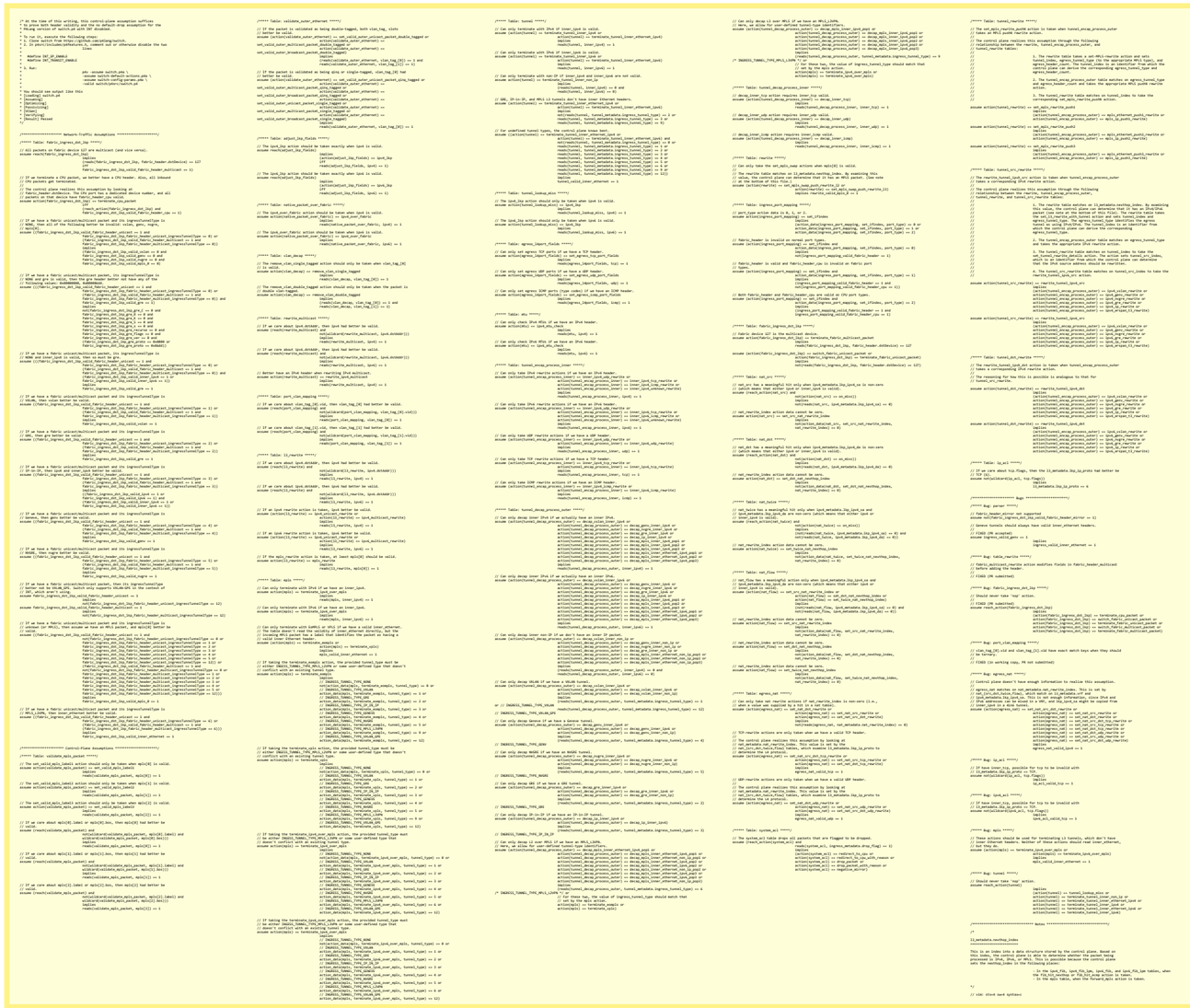
- 5,599 LoC
- 58 parser states
- 120 match–action tables

Control-plane interface

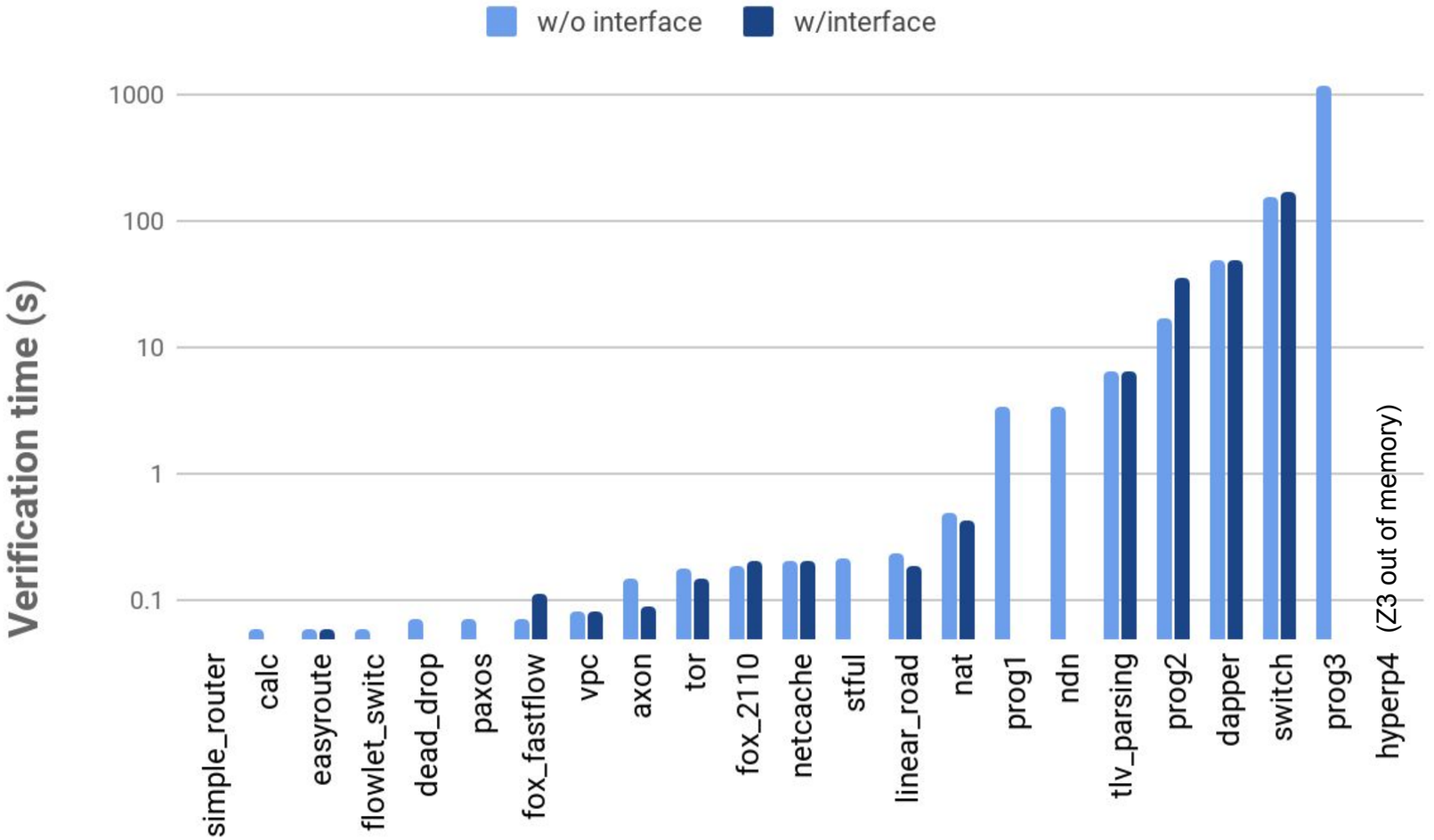
- 758 LoC
- ~2 days' programmer effort
- Default actions (31)
- Fabric wellformedness (14)
- Table actions (66)
- Guarded reads (10)
- Action data (14)

Found 10 bugs

- Parser bugs (2)
- Action flaws (4)
- Infeasible control-plane (3)
- Invalid table read (1)



Evaluation: performance



- Diverse set of 23 programs
  - Open & closed source
  - Conventional forwarding
  - Data centre routing
  - Content-based networking
  - Performance monitoring
  - In-network processing
- Header validity for all but two
  - Cross-cutting property
  - Reasoning about almost all control-flow paths
- All but three programs checked in under a minute
  - < 1 s for most
- One ran out of memory
  - hyperp4: virtual data planes

Related work

- Transfer functions & reachability analysis
  - Xie et al. (2005), Anteater (2011), Header space analysis (2012)
- Incremental verification & optimizations
  - VeriFlow (2013), Atomic Predicates (2013), ddNF (2016), network symmetry (2016)
- Control-plane verification
  - RCC (2015), Batfish (2015), ARC (2016), Bagpipe (2016), Minesweeper (2017)
- Middlebox verification
  - Dobrescu & Argyraki (2015), SymNet (2016), Panda et al. (2017), VigNAT (2017)
- P4 verification
  - McKeown et al. (2016), P4K (2018), p4pktgen (2018), p4-assert (2018), Vera (2018)

p4v: a practical tool for *all-paths* verification of P4 programs

Future work

- More front-ends & architectures
  - P4$_{16}$ support
  - Other architectures (e.g., Xilinx FPGAs)

- Control-plane interfaces
  - Integrate into P4 language?
  - Manually written — can we synthesize from traces?
  - Trusted — can we validate?

- Verify network-wide properties?
  - Problem becomes undecidable [Panda et al. 2017]
  - Likely need to abstract data plane behaviour to get scalability

p4v | Practical Verification for Programmable Data Planes

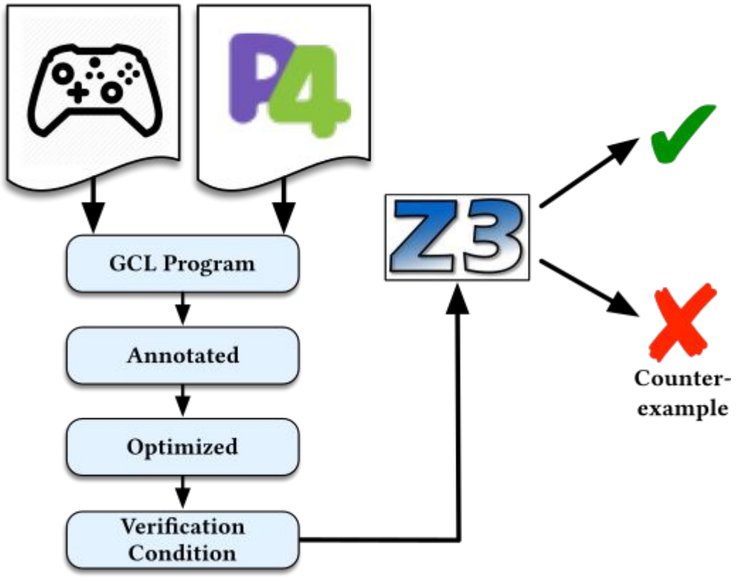
Jed Liu
Bill Hallahan

Cole Schlesinger
Milad Sharif

Jeongkeun Lee
Robert Soulé

Han Wang
Călin Caşcaval

Nick McKeown
Nate Foster



Automated all-paths verification

Scales to large programs (`switch.p4`)

Clean control–data plane interface