NetInfra

P4-BAR

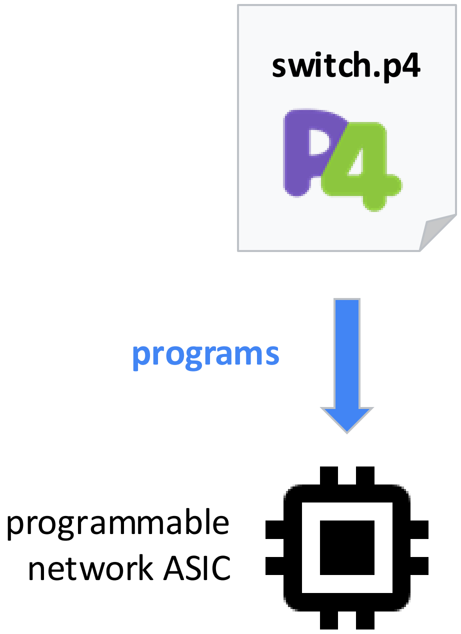
P4-Based **A**utomated **R**easoning (**P4-BAR**) for the (Networking) Masses!

P4 Workshop 2024 - Sunnyvale

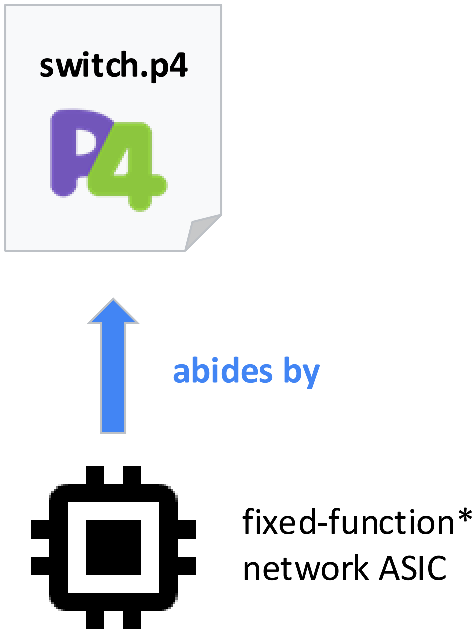
Steffen Smolka, Jonathan DiLorenzo, Ali Kheradmand

Google's Surprising Use of P4

P4 as intended



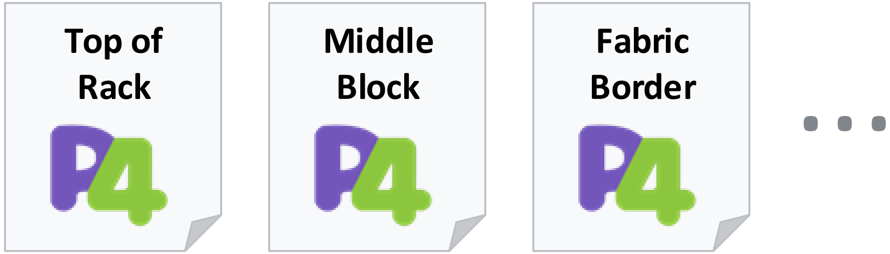
P4 at Google



* Oversimplified for ease of exposition.
All our ASICs are programmable to varying degrees, but few are fully P4-programmable.

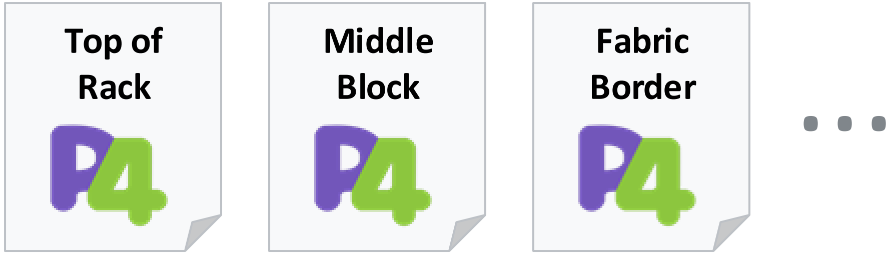
Google's View: P4 as a Specification Language

We view P4 programs as **machine-readable specifications** capturing all requirements for a switch
in a specific deployment role:



Google's View: P4 as a Specification Language

We view P4 programs as **machine-readable specifications** capturing all **requirements** for a switch
in a specific deployment role:

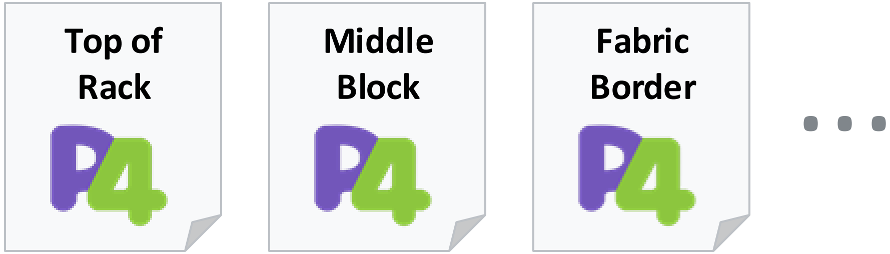


middleblock.p4

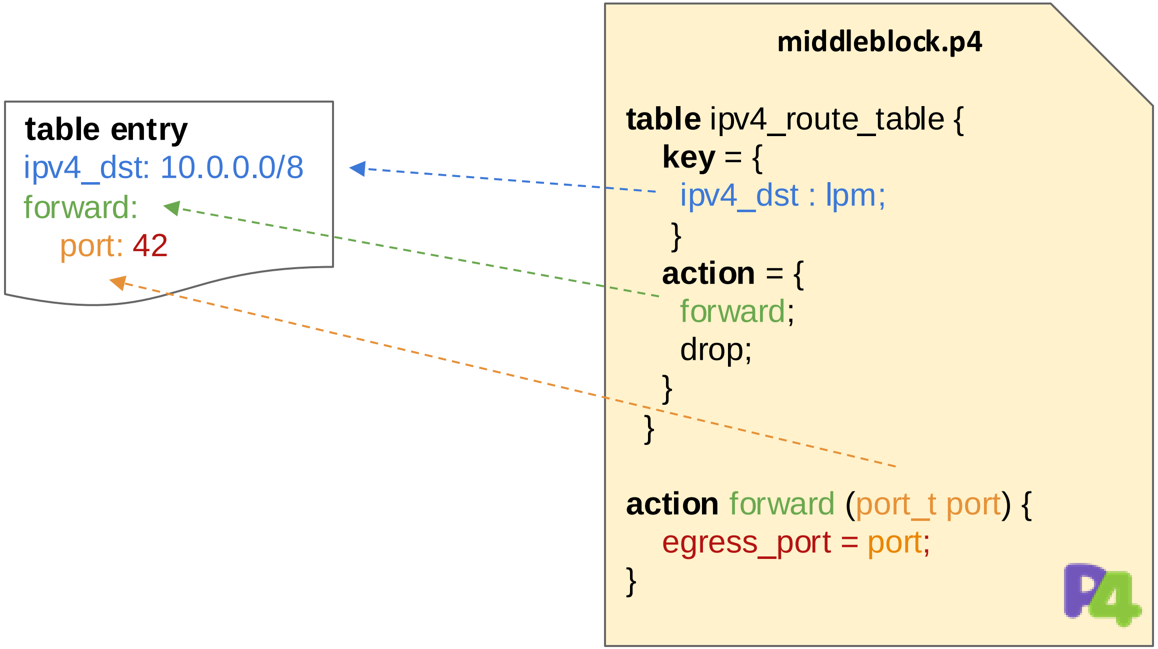
```
table ipv4_route_table {
    key = {
      ipv4_dst : lpm;
     }
    action = {
      forward;
      drop;
    }
  }

action forward (port_t port) {
    egress_port = port;
}
```

Google's View: P4 as a Specification Language

We view P4 programs as **machine-readable specifications** capturing all **requirements** for a switch
in a specific deployment role:



Schema of switch API



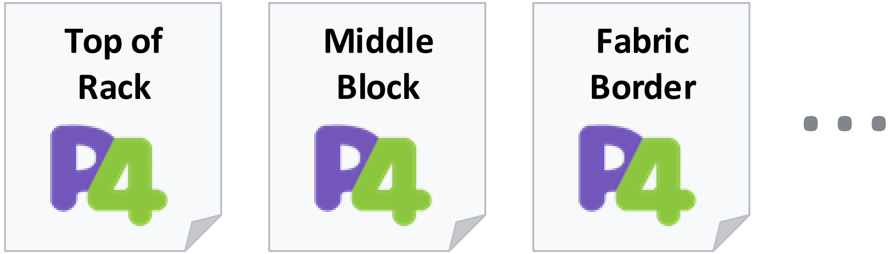
```
table entry
ipv4_dst: 10.0.0.0/8
forward:
   port: 42
```

middleblock.p4

```
table ipv4_route_table {
   key = {
     ipv4_dst : lpm;
   }
   action = {
     forward;
     drop;
   }
 }

action forward (port_t port) {
   egress_port = port;
}
```

Google's View: P4 as a Specification Language

We view P4 programs as **machine-readable specifications** capturing all **requirements** for a switch
in a specific deployment role:



Schema of switch API

Dataplane behavior

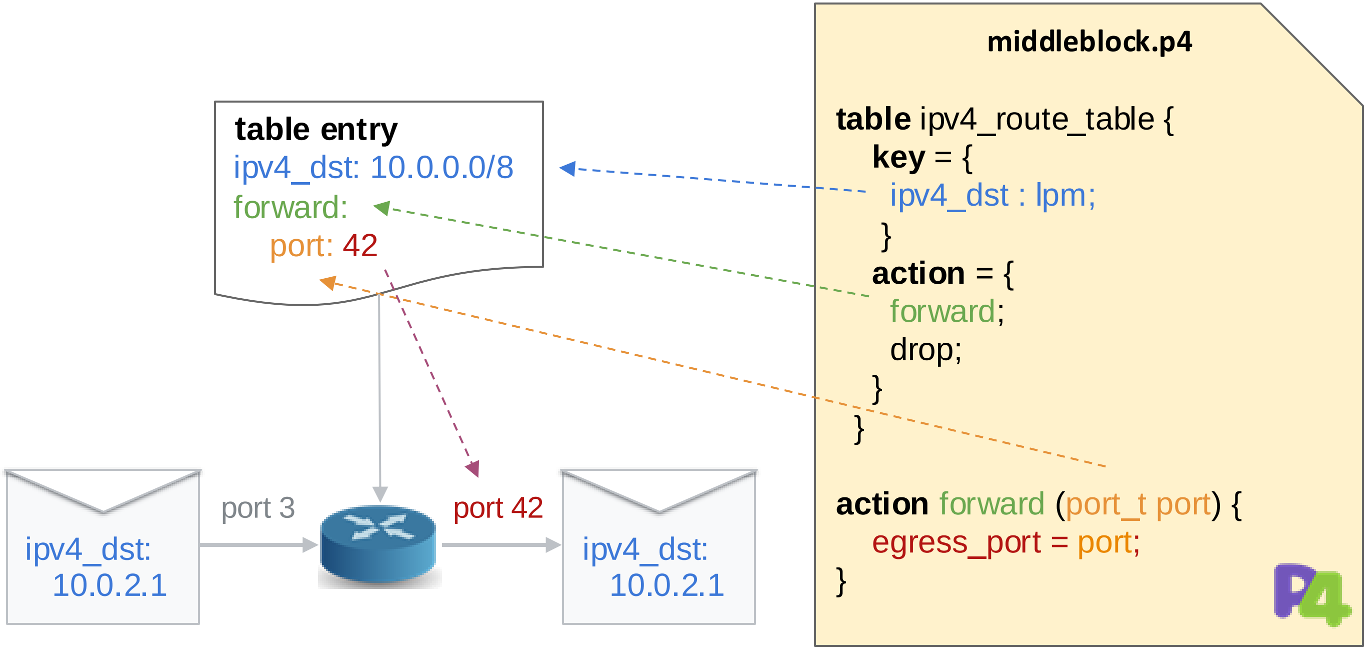
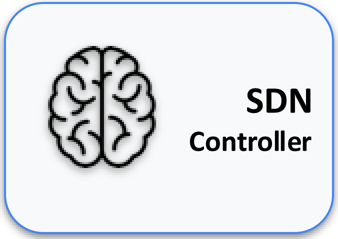


table entry
ipv4_dst: 10.0.0.0/8
forward:
  port: 42

middleblock.p4

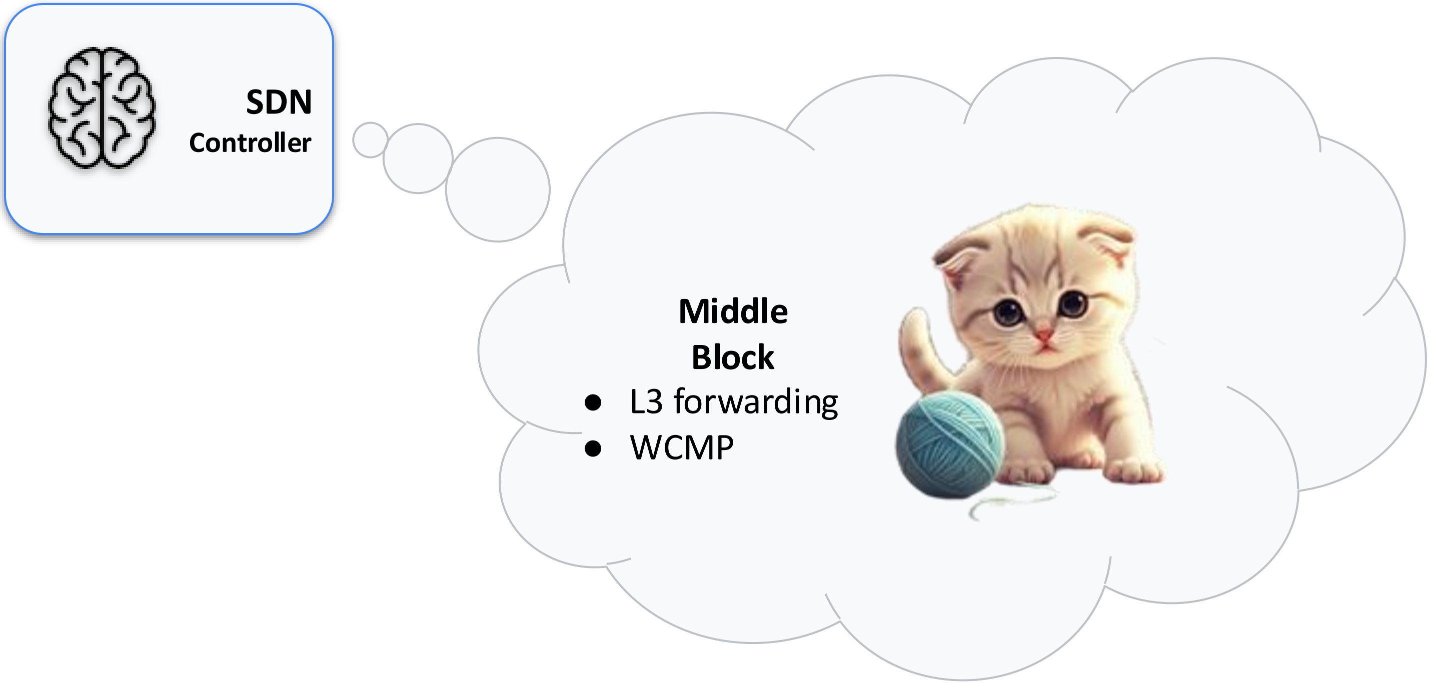
```
table ipv4_route_table {
    key = {
      ipv4_dst : lpm;
    }
    action = {
      forward;
      drop;
    }
  }

action forward (port_t port) {
    egress_port = port;
}
```

The Beauty And The Beast



SDN
Controller

The Beauty And The Beast

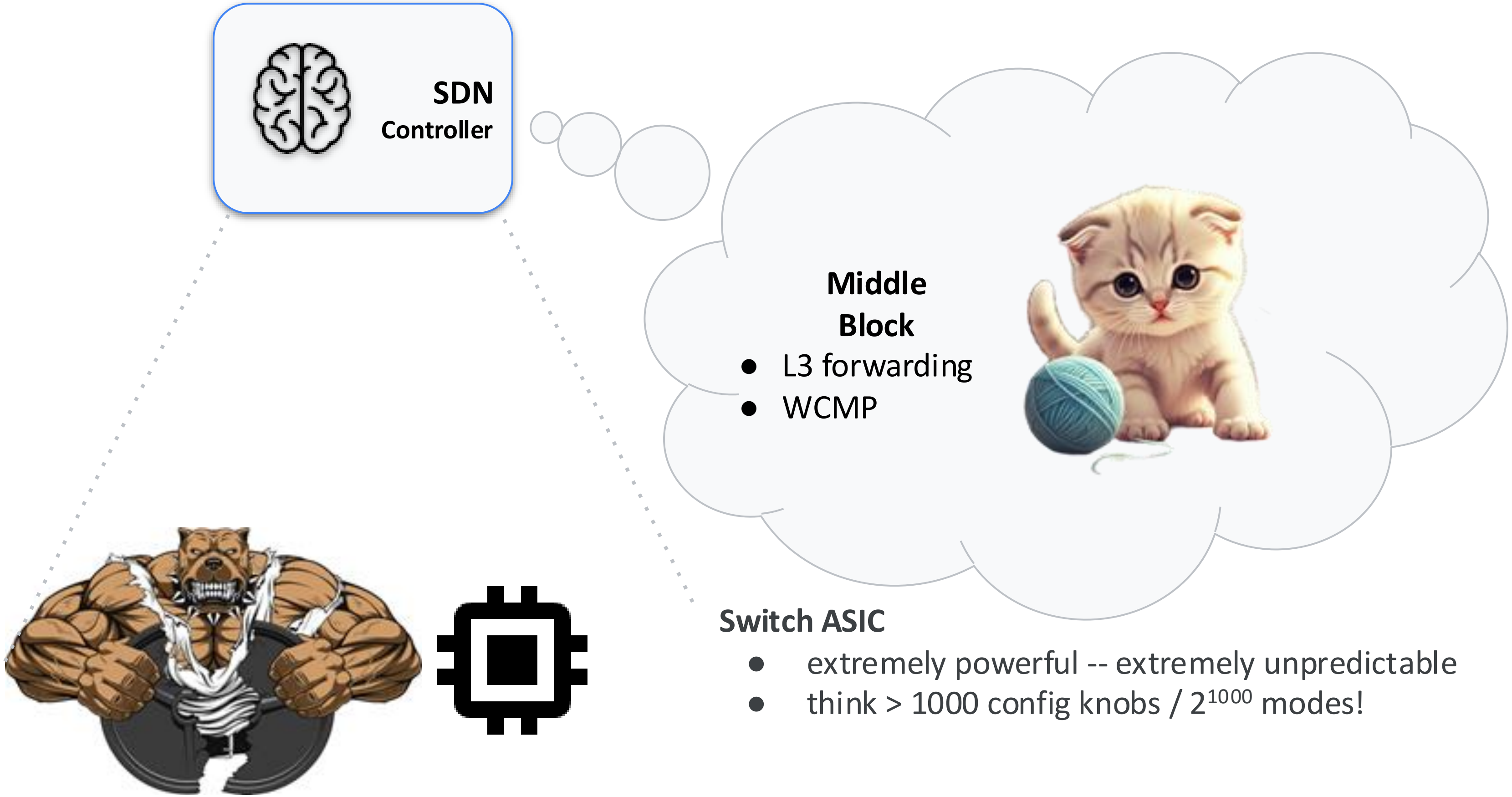


SDN
Controller

Middle Block

- L3 forwarding
- WCMP

The Beauty And The Beast



SDN Controller

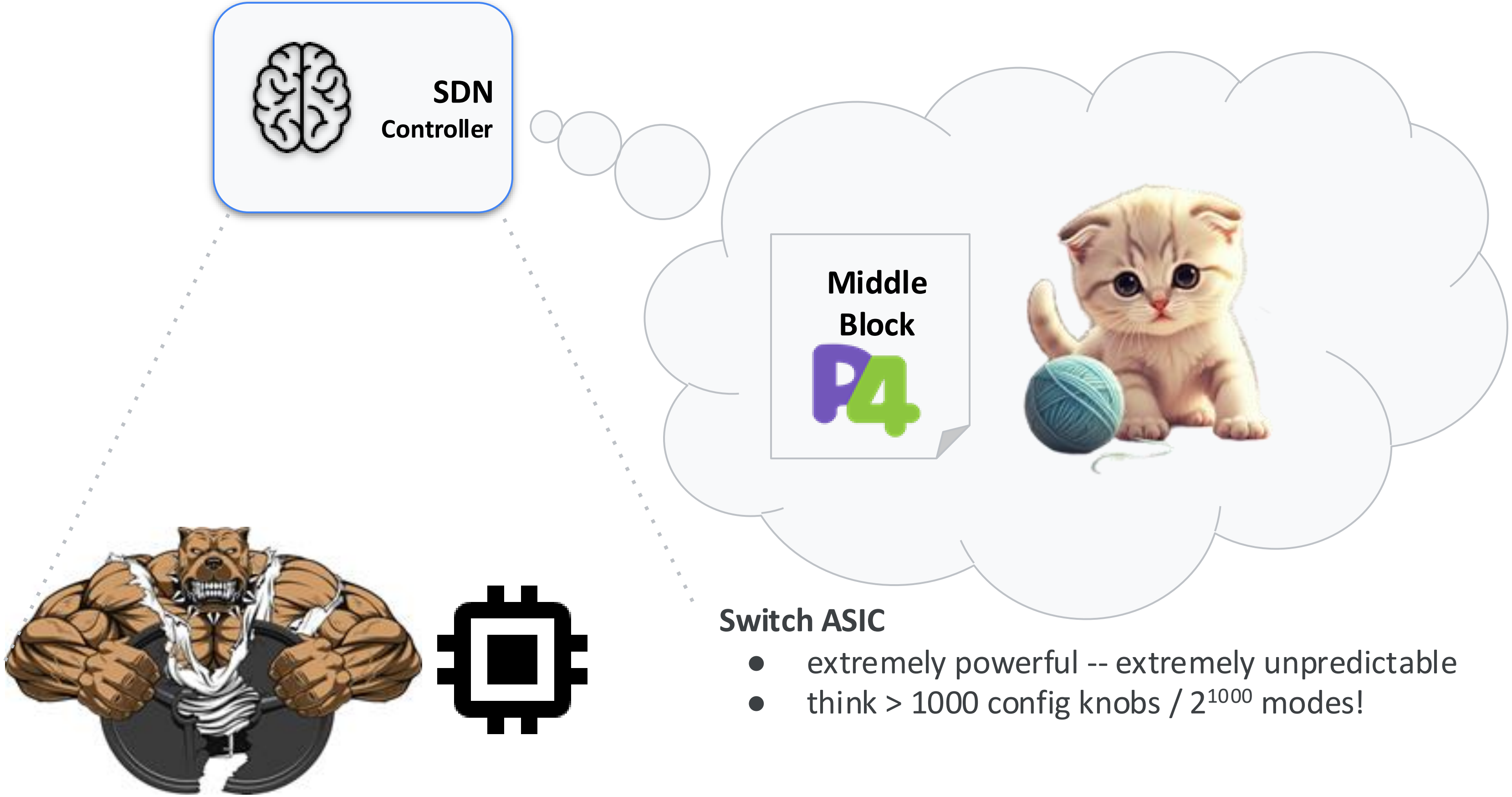
Middle Block

- L3 forwarding
- WCMP

Switch ASIC

- extremely powerful -- extremely unpredictable
- think > 1000 config knobs / 2^{1000} modes!

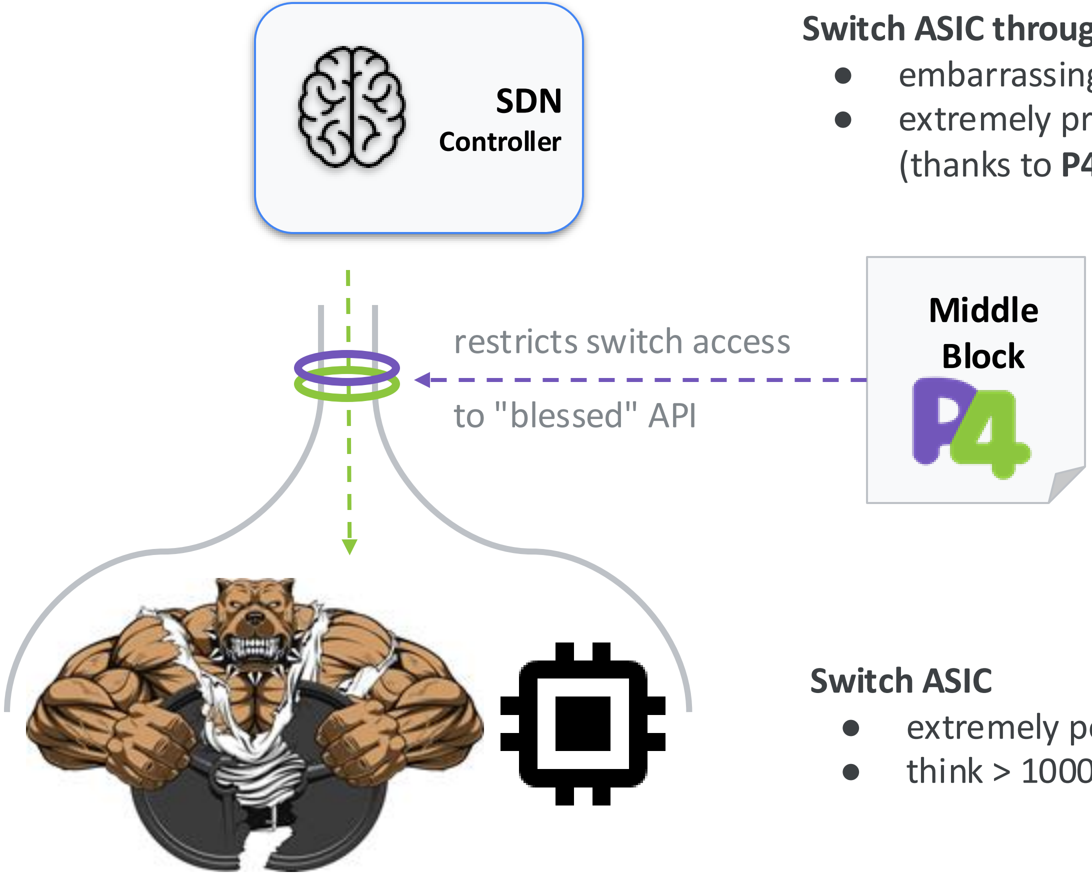
The Beauty And The Beast



Switch ASIC

- extremely powerful -- extremely unpredictable
- think > 1000 config knobs / 2^{1000} modes!

The Beauty And The Beast



SDN Controller

restricts switch access

to "blessed" API

Middle Block

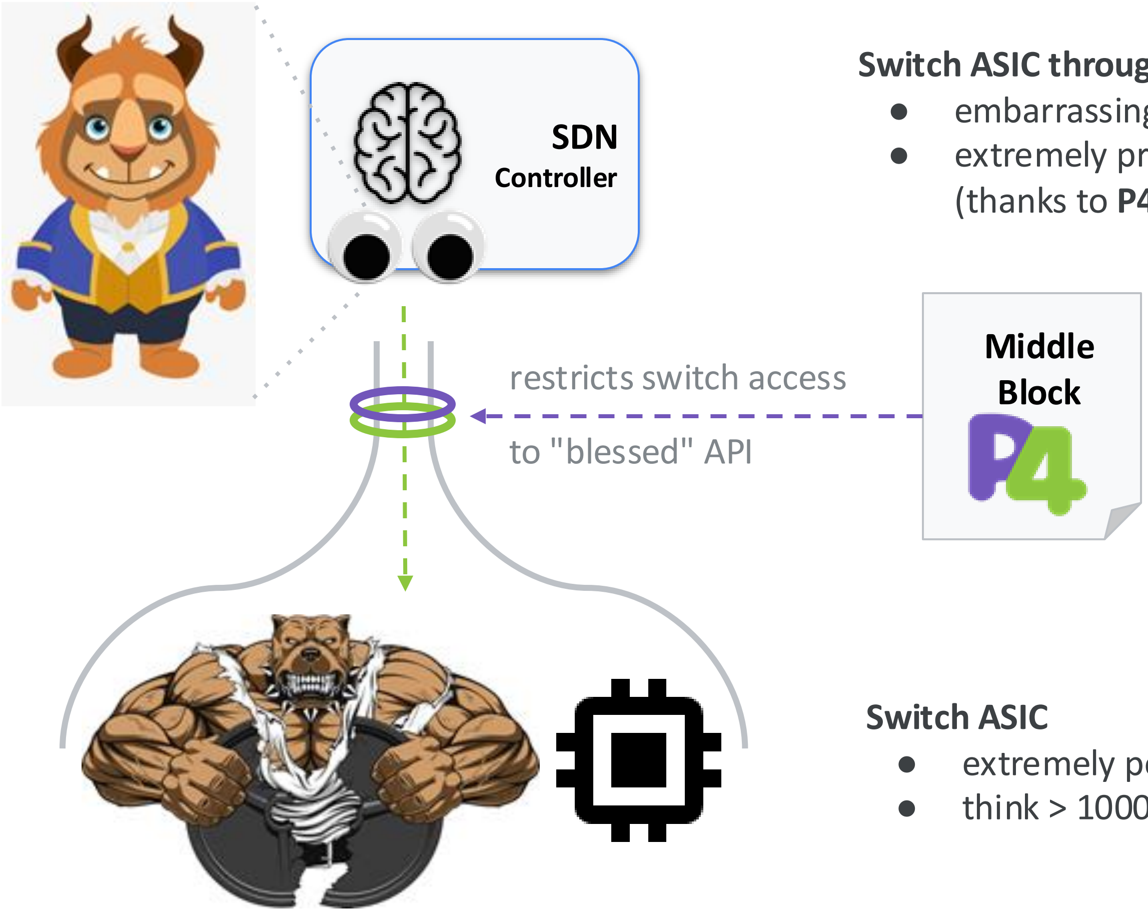
Switch ASIC through lens of P4 spec

- embarrassingly simple API
- extremely predictable
  (thanks to **P4**-**B**ased **A**utomated **R**easoning)

Switch ASIC

- extremely powerful -- extremely unpredictable
- think > 1000 config knobs / 2^{1000} modes!

The Beauty And The Beast



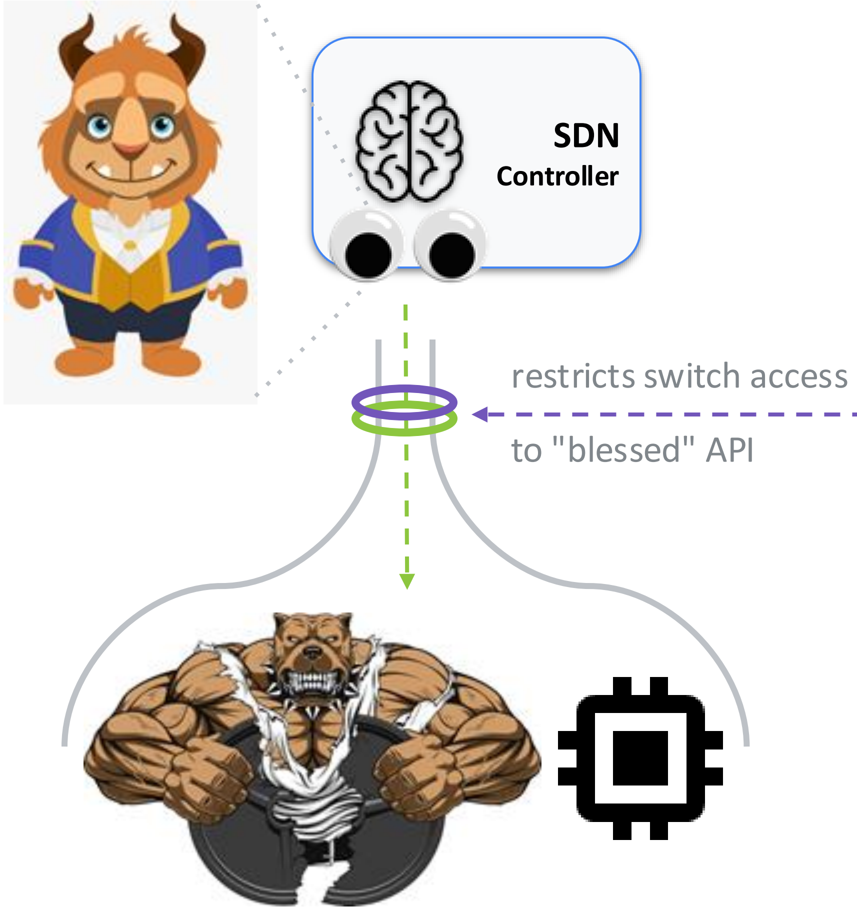
Switch ASIC through lens of P4 spec

- embarrassingly simple API
- extremely predictable
  (thanks to **P4**-**B**ased **A**utomated **R**easoning)

Switch ASIC

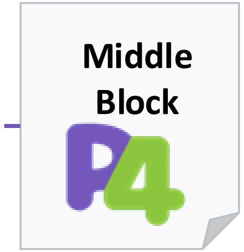
- extremely powerful -- extremely unpredictable
- think > 1000 config knobs / 2^{1000} modes!

The Beauty And The Beast



Switch ASIC through lens of P4 spec

- embarrassingly simple API
- extremely predictable (thanks to **P4**-**B**ased **A**utomated **R**easoning)



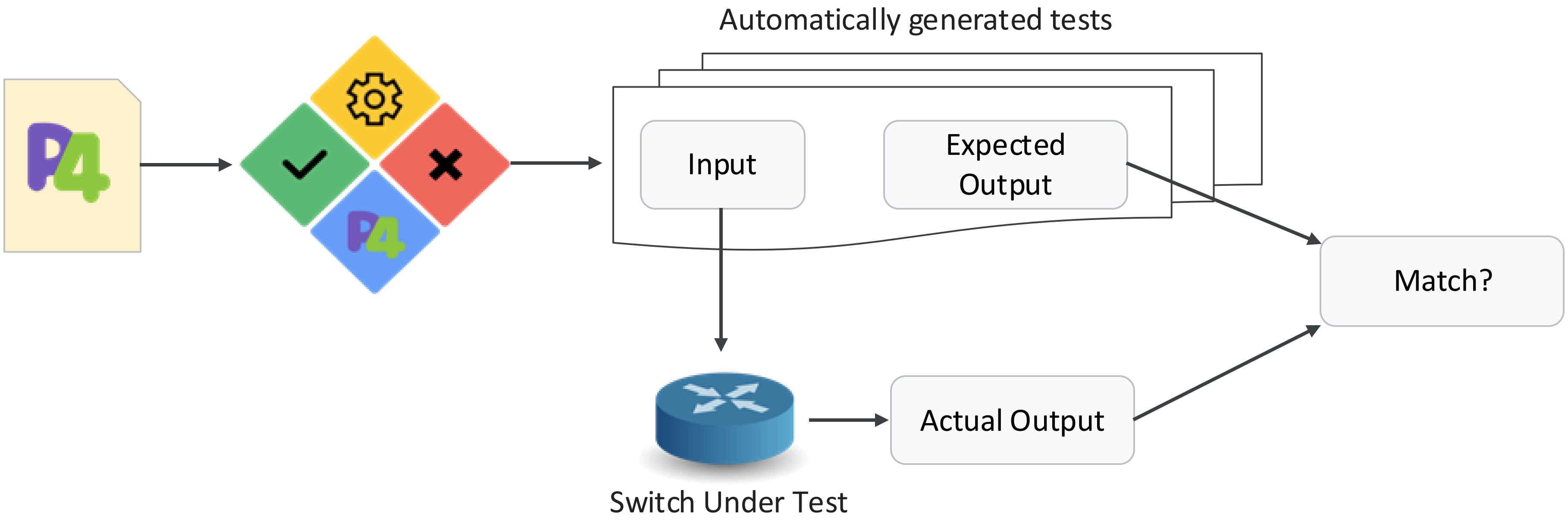
Additional Benefits:

- **Velocity:** can ship new/modified APIs quickly and confidently.
- **Optionality:** can confidently swap in any ASIC that meets the spec.

Switch ASIC

- extremely powerful -- extremely unpredictable
- think > 1000 config knobs / 2^{1000} modes!

P4-Based Automated Reasoning (P4-BAR)



Automatically generated tests

Input

Expected Output

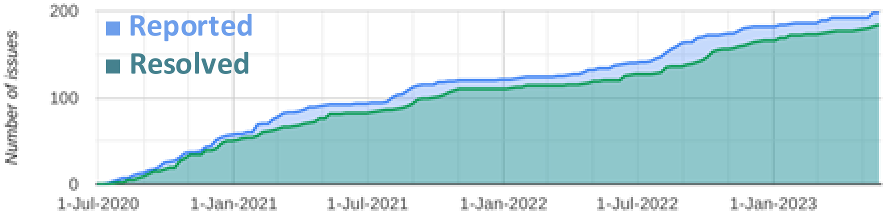
Match?

Actual Output

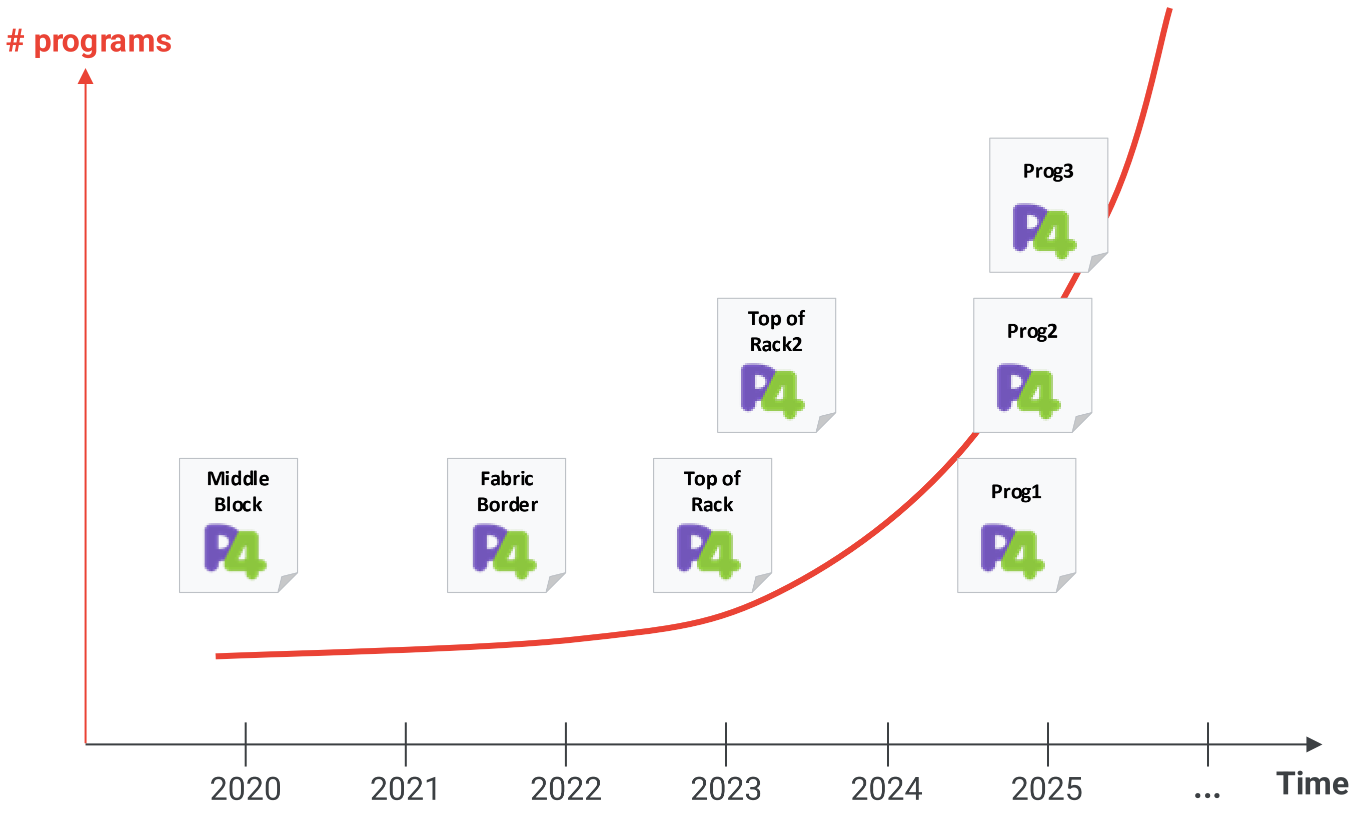
Switch Under Test

A success story

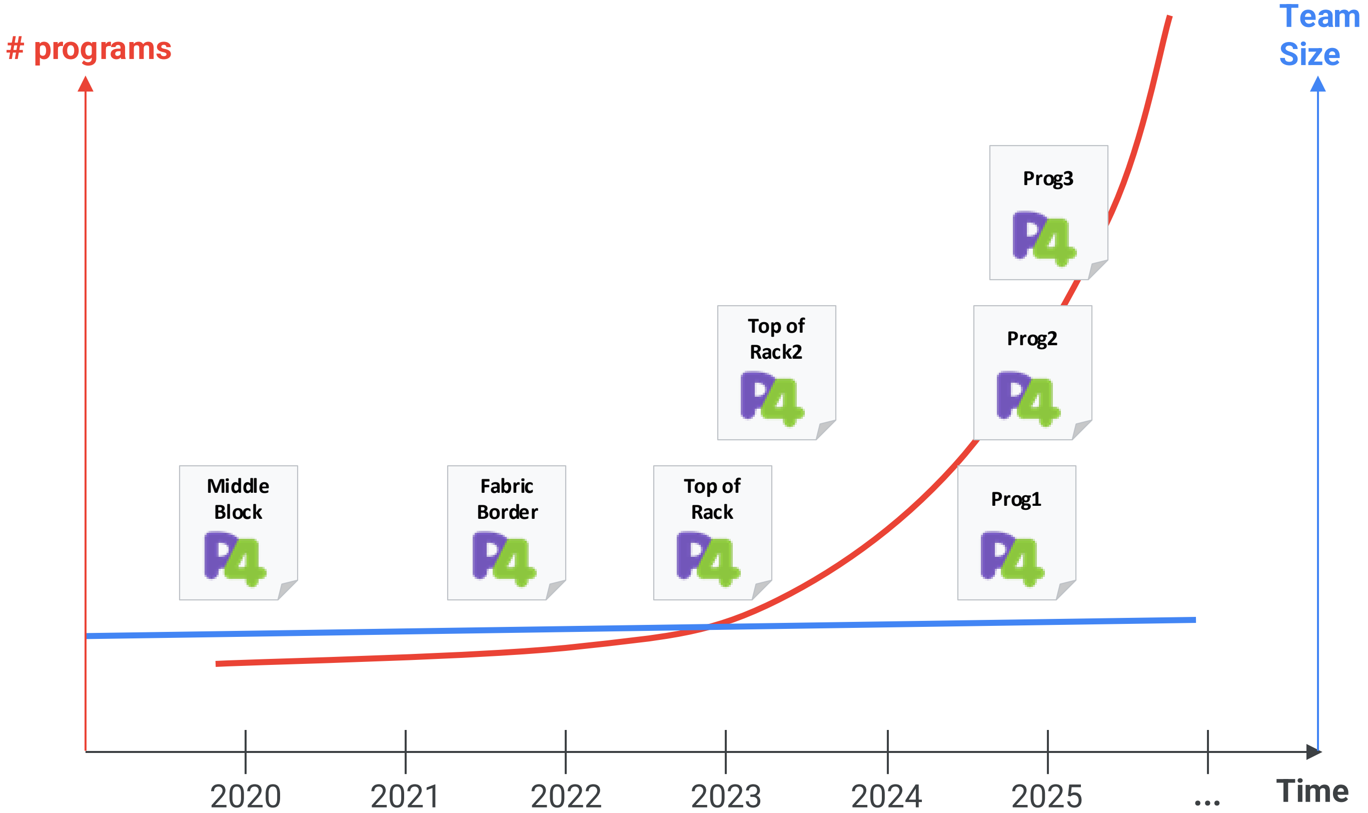
- used for every DC deployment role since 2020
- > 200 bugs unique bugs found, < 5 escaped
- published at SIGCOMM 22 ("SwitchV")



Problem: Scaling it to the Masses



Problem: Scaling it to the Masses



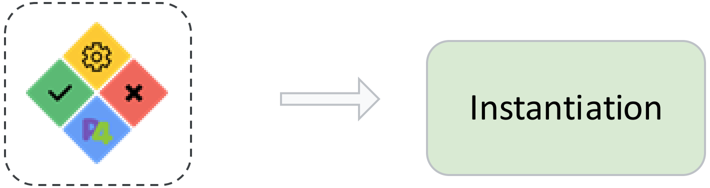
Problem: Cost of P4-BAR Validation

cost(P4-BAR validation) =

cost(P4-BAR dev) +

Problem: Cost of P4-BAR Validation

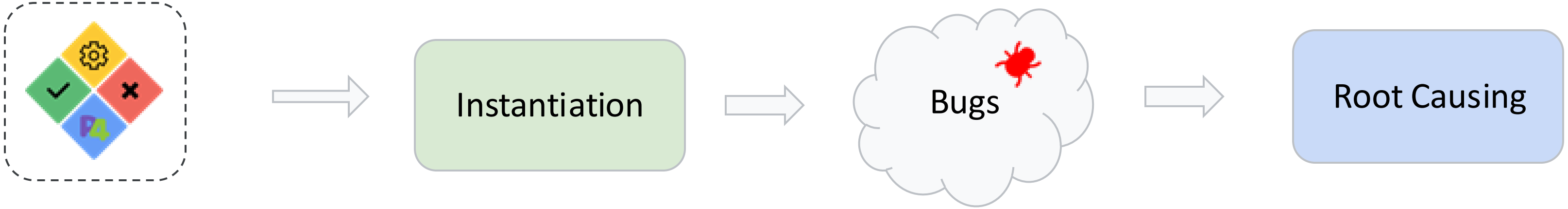
$$\text{cost(P4-BAR validation)} = \text{cost(P4-BAR dev)} + \sum_{\text{prog} = 1}^{n} \left[\text{cost(P4-BAR instantiation)} + \right]$$



Instantiation

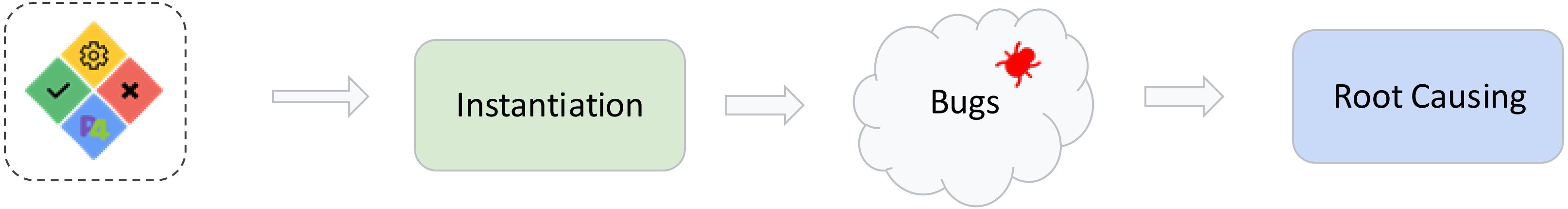
Problem: Cost of P4-BAR Validation

$$\text{cost(P4-BAR validation)} = \text{cost(P4-BAR dev)} + \sum_{\text{prog} = 1}^{n} \left[\text{cost(P4-BAR instantiation)} + \#\,\text{bugs(prog)} \cdot \text{cost(P4-BAR root causing)} \right]$$



Problem: Cost of P4-BAR Validation

$$\text{cost(P4-BAR validation)} = \text{cost(P4-BAR dev)} + \sum_{\text{prog} = 1}^{n} \left[\text{cost(P4-BAR instantiation)} + \#\,\text{bugs(prog)} \cdot \text{cost(P4-BAR root causing)} \right]$$



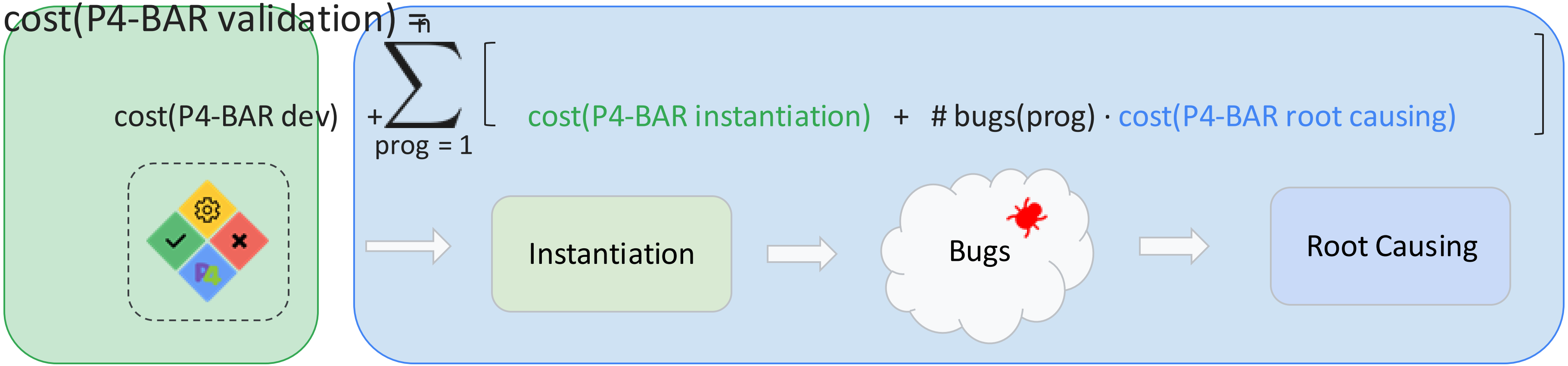
Idea 1: Reduce root cause cost

- How: Automation

Idea 2: Reduce instantiation cost

- How: Modular APIs

Problem: Cost of P4-BAR Validation



$$\text{cost(P4-BAR validation)} = \text{cost(P4-BAR dev)} + \sum_{\text{prog}=1}^{n} \left[\text{cost(P4-BAR instantiation)} + \#\,\text{bugs(prog)} \cdot \text{cost(P4-BAR root causing)} \right]$$

Owned by us

Owned by our clients

Idea 1: Reduce root cause cost
- How: Automation

Idea 2: Reduce instantiation cost
- How: Modular APIs

Idea 3: Delegate per-program work
- How: Powerful yet easy-to-use APIs

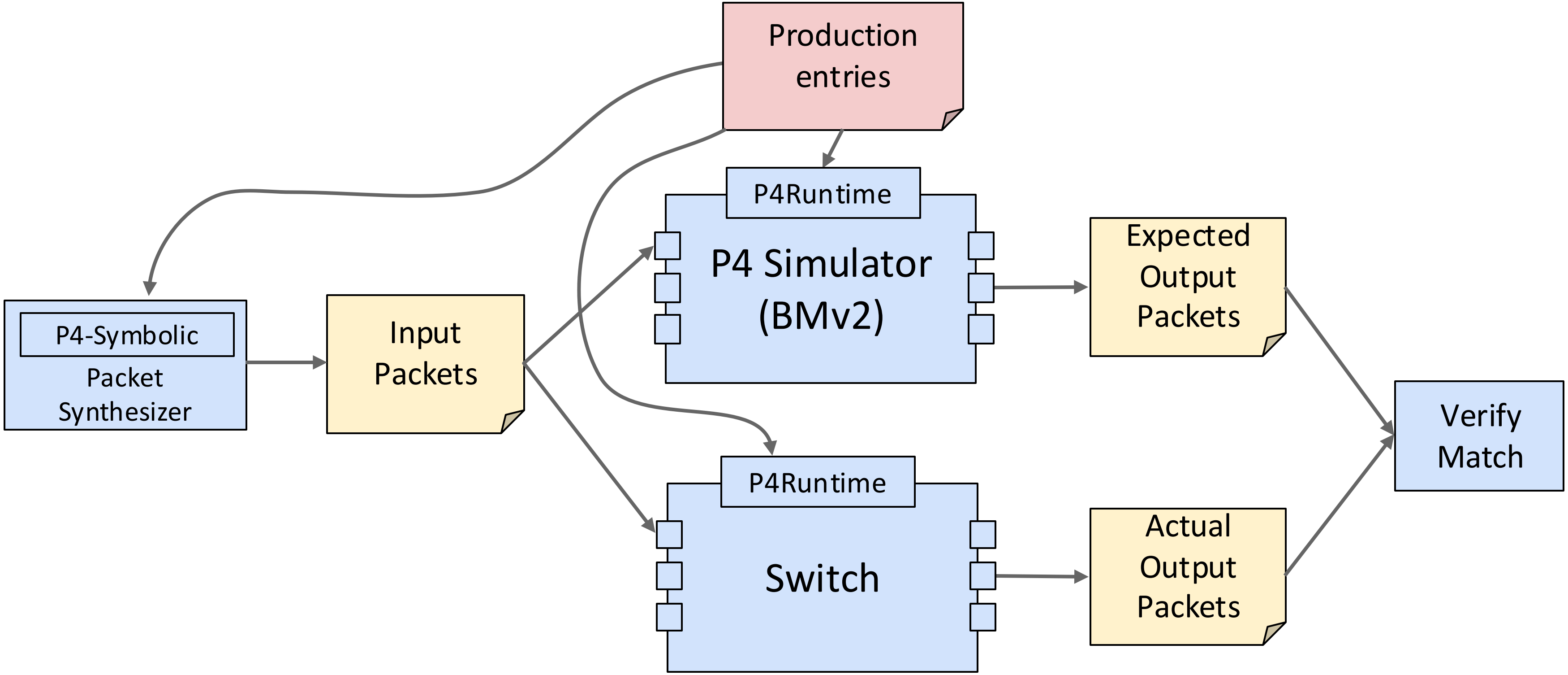
Mission: Build tools so user-friendly & powerful that no one wants to write manual tests.

This Talk

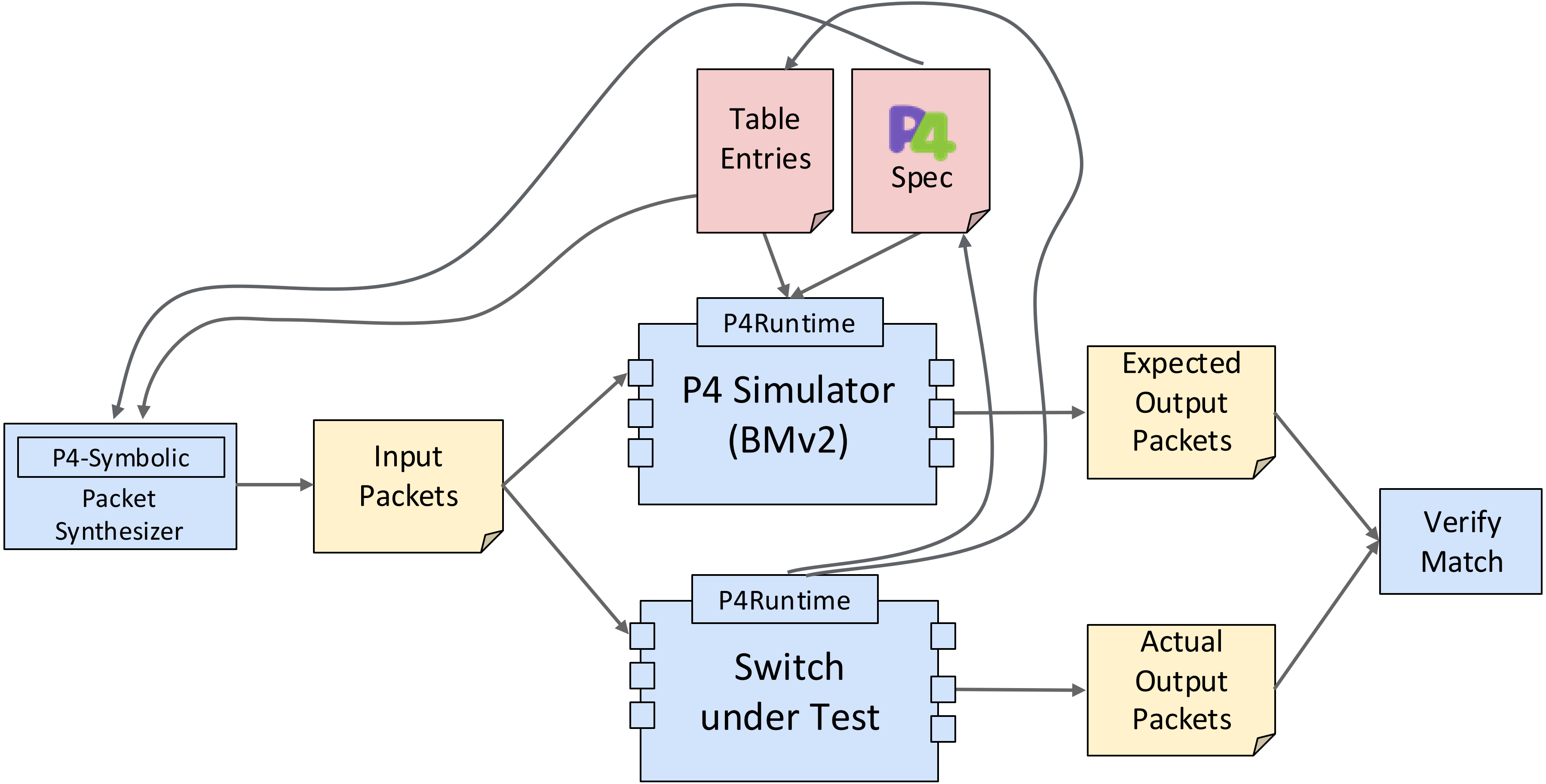
Proprietary + Confidential

1. P4 as a Specification Language ✓
2. Problem: Scaling P4-BAR to the masses! ✓
3. **Approach 1: High-Level APIs**
4. Approach 2: Automating Root Causing

Dataplane Testing - Historically

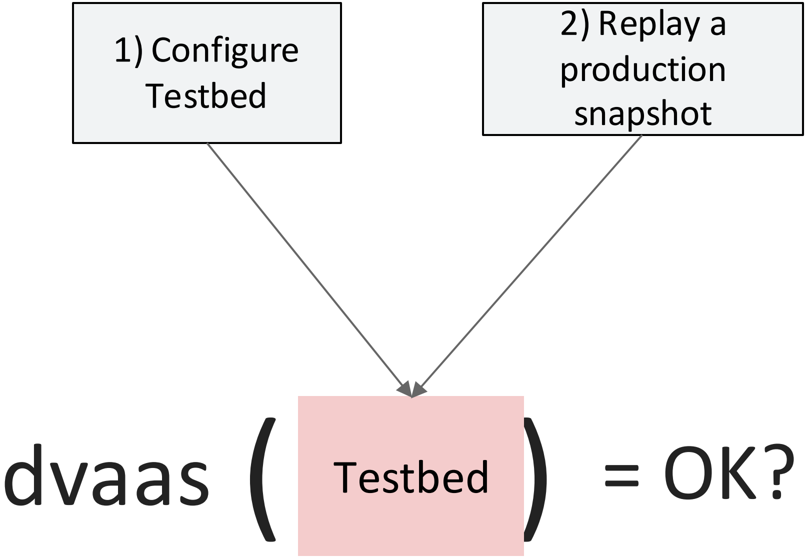


DVaaS: Dataplane Validation as a Service



DVaaS: Ease of use

Example: Replay Testing

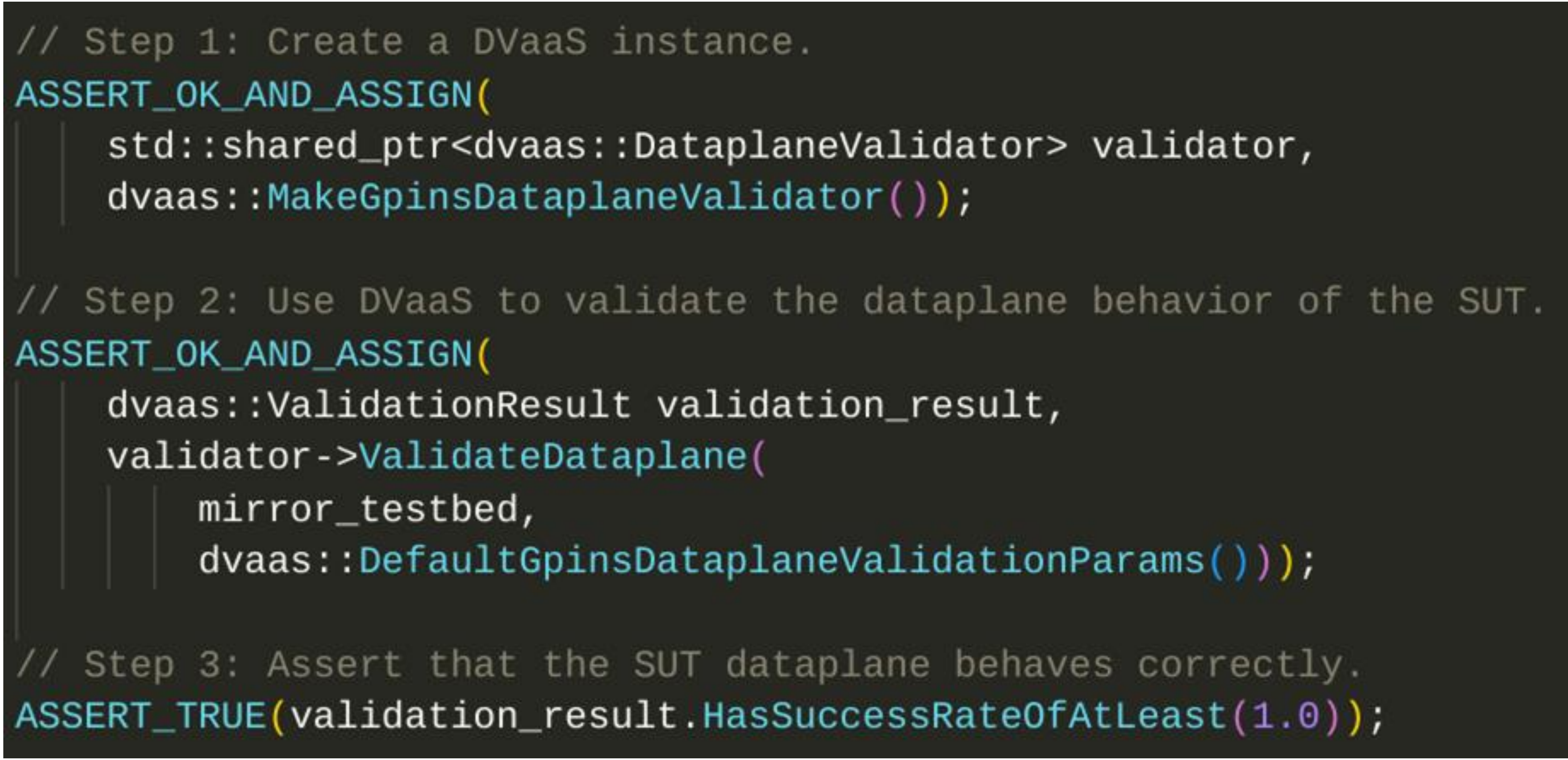


1) Configure Testbed

2) Replay a production snapshot

dvaas (Testbed) = OK?

DVaaS: Actual usage code



```
// Step 1: Create a DVaaS instance.
ASSERT_OK_AND_ASSIGN(
    std::shared_ptr<dvaas::DataplaneValidator> validator,
    dvaas::MakeGpinsDataplaneValidator());

// Step 2: Use DVaaS to validate the dataplane behavior of the SUT.
ASSERT_OK_AND_ASSIGN(
    dvaas::ValidationResult validation_result,
    validator->ValidateDataplane(
        mirror_testbed,
        dvaas::DefaultGpinsDataplaneValidationParams()));

// Step 3: Assert that the SUT dataplane behaves correctly.
ASSERT_TRUE(validation_result.HasSuccessRateOfAtLeast(1.0));
```

This Talk

1. P4 as a Specification Language ✓
2. Problem: Scaling P4-BAR to the masses! ✓
3. Approach 1: High-Level APIs ✓
4. **Approach 2: Automating Root Causing**

Root Causing: Historic Output

```
Expected: DATAPLANE packet gets forwarded (1 copies)
  Actual: DATAPLANE packet got dropped

Showing the first failure only.
See test artifacts for full list of errors.

== INPUT ===================================================
type: DATAPLANE
packet {
  port: "1"
  headers {
    ethernet_header {
      ethernet_destination: "ff:ee:dd:cc:bb:aa"
      ethernet_source: "55:44:33:22:11:00"
      ethertype: "0x86dd"
    }
  }
...
== EXPECTED OUTPUT =========================================
packets {
  port: "8"
  headers {
    ethernet_header {
      ethernet_destination: "06:05:04:03:02:01"
      ethernet_source: "01:02:03:04:05:06"
      ethertype: "0x86dd"
    }
  }
...
```

Root Causing: Common Questions

```
Expected: DATAPLANE packet gets forwarded (1 copies)
  Actual: DATAPLANE packet got dropped

Showing the first failure only.
See test artifacts for full list of errors.

== INPUT ============================================================
type: DATAPLANE
packet {
  port: "1"
  headers {
    ethernet_header {
      ethernet_destination: "ff:ee:dd:cc:bb:aa"
      ethernet_source: "55:44:33:22:11:00"
      ethertype: "0x86dd"
    }
  }
...
== EXPECTED OUTPUT ==========================================
packets {
  port: "8"
  headers {
    ethernet_header {
      ethernet_destination: "06:05:04:03:02:01"
      ethernet_source: "01:02:03:04:05:06"
      ethertype: "0x86dd"
    }
  }
...
```

Regression? Or new test?

Is this an outlier or the norm?

Perhaps this is a flake?
Can it be reproduced?

Is this even a valid input packet?
How do ***I*** reproduce this?

Why do you expect this?
Maybe you shouldn't / the test is broken?

Problem: Answering these questions currently requires humans.

Root Causing: Common Questions

```
Expected: DATAPLANE packet gets forwarded (1 copies)
  Actual: DATAPLANE packet got dropped

Showing the first failure only.
See test artifacts for full list of errors.

== INPUT ============================================================
type: DATAPLANE
packet {
  port: "1"
  headers {
    ethernet_header {
      ethernet_destination: "ff:ee:dd:cc:bb:aa"
      ethernet_source: "55:44:33:22:11:00"
      ethertype: "0x86dd"
    }
  }
...
== EXPECTED OUTPUT ==========================================
packets {
  port: "8"
  headers {
    ethernet_header {
      ethernet_destination: "06:05:04:03:02:01"
      ethernet_source: "01:02:03:04:05:06"
      ethertype: "0x86dd"
    }
  }
...
```

Is this reproducible or a flake?

Simple Solution: Retry packet 100x

```
Sending the same input packet reproduces this error
100.00% of the time
```

Root Causing: Common Questions

```
Expected: DATAPLANE packet gets forwarded (1 copies)
  Actual: DATAPLANE packet got dropped

Showing the first failure only.
See test artifacts for full list of errors.

== INPUT ============================================================
type: DATAPLANE
packet {
  port: "1"
  headers {
    ethernet_header {
      ethernet_destination: "ff:ee:dd:cc:bb:aa"
      ethernet_source: "55:44:33:22:11:00"
      ethertype: "0x86dd"
    }
  }
...
== EXPECTED OUTPUT ==========================================
packets {
  port: "8"
  headers {
    ethernet_header {
      ethernet_destination: "06:05:04:03:02:01"
      ethernet_source: "01:02:03:04:05:06"
      ethertype: "0x86dd"
    }
  }
...
```

Is this an outlier, or the norm?

Simple Solution: Report Statistics.

```
88.27% of 3027 test vectors passed
88.27% of 3027 test vectors produced the correct number
and type of output packets
987 test vectors forwarded, producing 996 forwarded
output packets
1712 test vectors punted, producing 1712 punted output
packets
774 test vectors produced no output packets
All of 1 test vectors attempted had deterministically
reproducible failures
```

Root Causing: Common Questions

```
Expected: DATAPLANE packet gets forwarded (1 copies)
  Actual: DATAPLANE packet got dropped

Showing the first failure only.
See test artifacts for full list of errors.

== INPUT ============================================================
type: DATAPLANE
packet {
  port: "1"
  headers {
    ethernet_header {
      ethernet_destination: "ff:ee:dd:cc:bb:aa"
      ethernet_source: "55:44:33:22:11:00"
      ethertype: "0x86dd"
    }
  }
...
== EXPECTED OUTPUT ==================================================
packets {
  port: "8"
  headers {
    ethernet_header {
      ethernet_destination: "06:05:04:03:02:01"
      ethernet_source: "01:02:03:04:05:06"
      ethertype: "0x86dd"
    }
  }
...
```

Why do you expect this?

Solution: Report packet traces.

```
== EXPECTED INPUT-OUTPUT TRACE (P4 SIMULATION) ==

Table 'some_table': miss

Table 'ipv4_route_table': hit
  Match: ipv4_dst: 10.0.0.0/8
  Action: forward(port: 42)

Primitive: 'mark_to_drop' (routing.p4(275))

Table 'multicast_table': hit
   ...

Packet replication: 4 replicas
```

Root Causing: Common Questions

```
Expected: DATAPLANE packet gets forwarded (1 copies)
  Actual: DATAPLANE packet got dropped

Showing the first failure only.
See test artifacts for full list of errors.

== INPUT ===========================
type: DATAPLANE
packet {
  port: "1"
  headers {
    ethernet_header {
      ethernet_destination: "ff:ee:dd:cc:bb:aa"
      ethernet_source: "55:44:33:22:11:00"
      ethertype: "0x86dd"
    }
  }
...
== EXPECTED OUTPUT =========================================
packets {
  port: "8"
  headers {
    ethernet_header {
      ethernet_destination: "06:05:04:03:02:01"
      ethernet_source: "01:02:03:04:05:06"
      ethertype: "0x86dd"
    }
  }
...
```

Can be minimized to further simplify debugging

How do I reproduce this?

Solution: Output an executable* proto.

```
PacketTestVector test_vector; # Packet + Expected Output
repeated p4::v1::Entity entities; # Entities causing bug
p4.config.v1.P4Info p4info; # API causing bug.
Any additional_metadata_for_reproduction; # Just in case
```

* In reality, there's a test fixture that executes the proto.

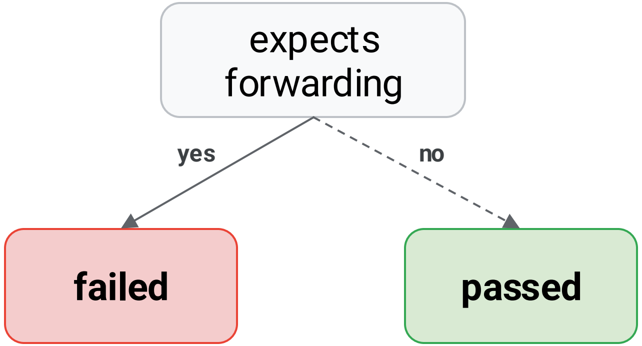
Root Causing: Common Questions

```
Expected: DATAPLANE packet gets forwarded (1 copies)
  Actual: DATAPLANE packet got dropped

Showing the first failure only.
See test artifacts for full list of errors.

== INPUT ==============================================================
type: DATAPLANE
packet {
  port: "1"
  headers {
    ethernet_header {
      ethernet_destination: "ff:ee:dd:cc:bb:aa"
      ethernet_source: "55:44:33:22:11:00"
      ethertype: "0x86dd"
    }
  }
...
== EXPECTED OUTPUT ===========================================
packets {
  port: "8"
  headers {
    ethernet_header {
      ethernet_destination: "06:05:04:03:02:01"
      ethernet_source: "01:02:03:04:05:06"
      ethertype: "0x86dd"
    }
  }
...
```

What's the pattern?

Aspirational Solution:

- Interpretable Machine Learning
- Fit a binary classifier to the data, e.g. decision trees



Wrapping Up

Summary

The P4-Based Automated Reasoning (P4-BAR) paradigm:

- Views P4 programs as **machine-readable specifications**.
- Automatically establishes that a given switch meets a given specification (with high probability).