2022 P4 Workshop
May 24-26th

p4testgen: Automated Test Generation for Real-World P4 Data Planes

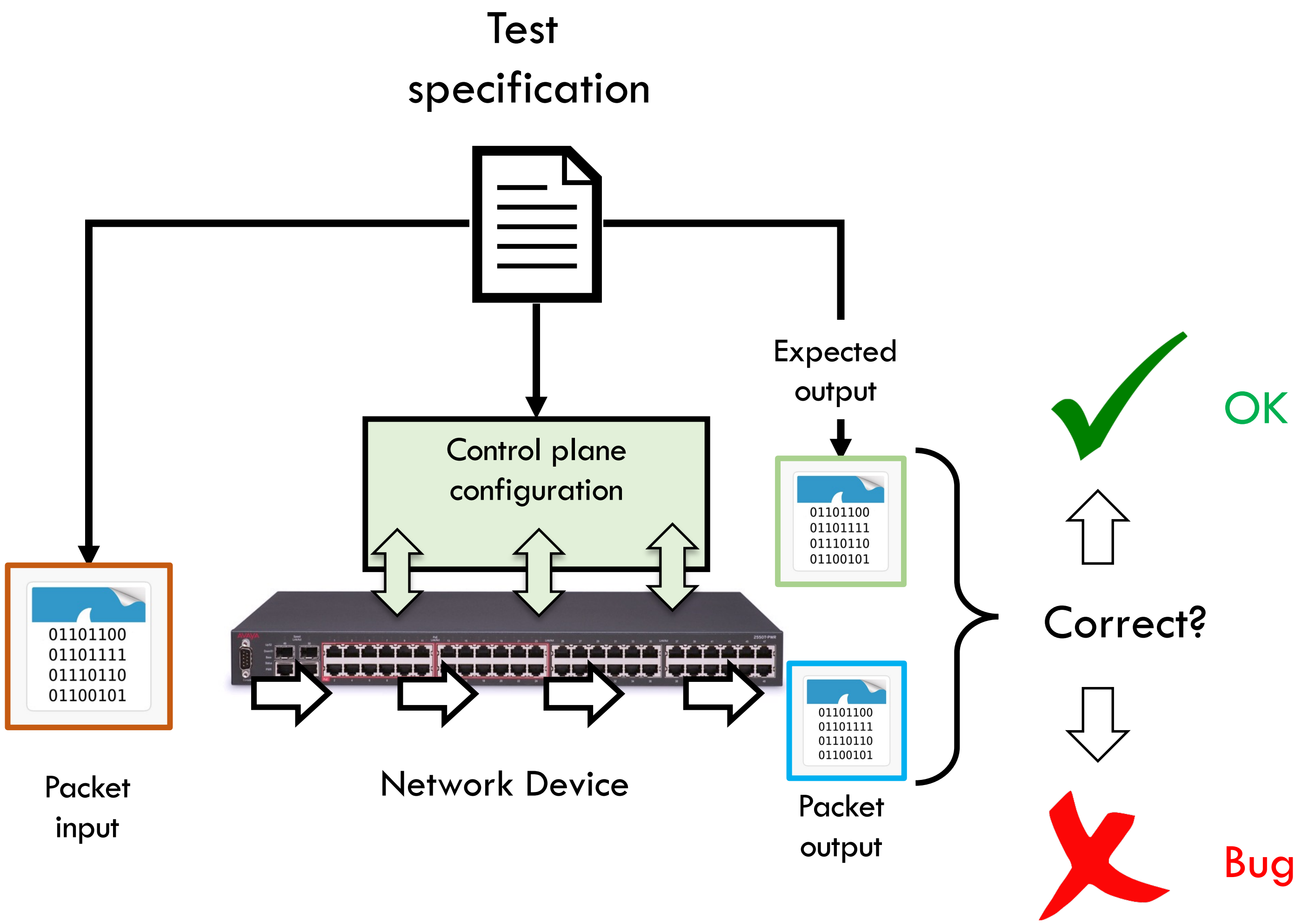
Fabian Ruffy (Intel/NYU), Jed Liu (Akita Software),
Volodymyr Peschanenko (LitSoft), Vladyslav Dubina (LitSoft),
Havel Vojtěch (Intel), Prathima Kotikalapudi (Intel),
Anirudh Sivaraman (NYU), Nate Foster (Intel/Cornell University)

NEW YORK UNIVERSITY

LitSoft Private Enterprise

intel

How Do We Test Network Hardware?

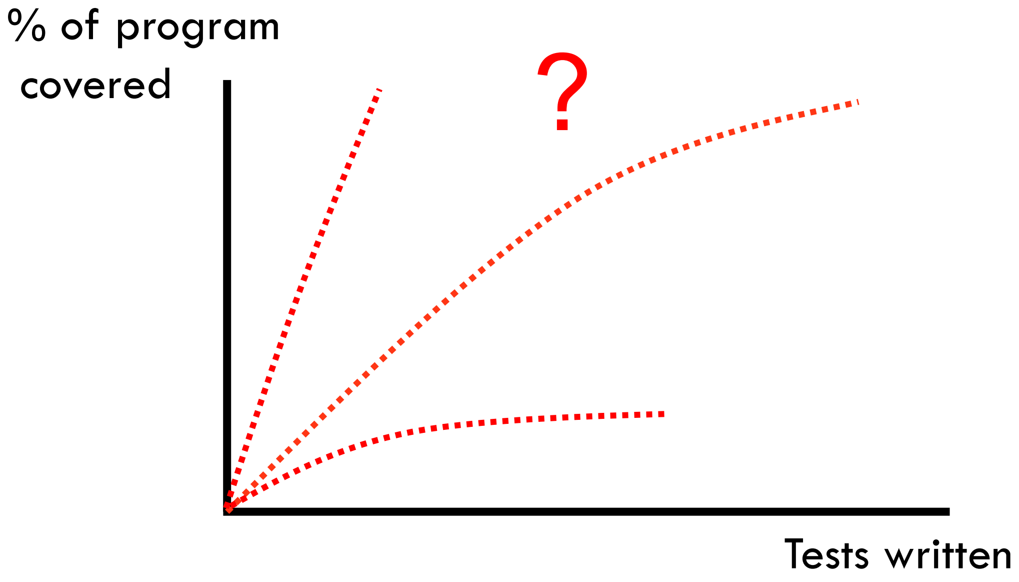


Reality

```
sap                     = 0xc # Arbitrary value
vpn                     = 0x0 # Arbitrary value
spi                     = 0x4 # Arbitrary value
si                      = 0x5 # Arbitrary value (ttl)
dsap                    = 7 # Arbitrary value
sf_bitmask              = 7 # Bit 0 = ingress, bit 1 = multicast, bit 2 = egress
nexthop_ptr             = 0x65 # Arbitrary value
bd                      = 1 # Arbitrary value
ig_lag_ptr              = 2 # Arbitrary value
eg_lag_ptr              = 0x10 # Arbitrary value

npb_nsh_chain_start_end_add(self, self.target,
    #ingress
    [ig_port], ig_lag_ptr, 0, sap, vpn, spi, si, sf_bitmask, rmac, nexthop_ptr, bd, eg_lag_ptr, 0, 0, [eg_port], 0,
dsap)

src_pkt =           Ether(b'\x00\x00\x5e\x00\x01\x01\x34\x41\x5d\x65\xd9\xe8\x08\x00')
src_pkt = src_pkt / IP    (b'\x45\x00\x00\x43\x00\x05\x00\x00\x80\x11\xcf\x13\x86\x8d\xbc\x62\x86\x8d\xa2\x14')
src_pkt = src_pkt / TCP   (b'\xee\xd7\x00\x35\x00\x2f\x67\xc8\xf9\xf7\x01\x00\x00\x01\x00\x00\x00\x00'
                        '\x00\x00\x07\x6f\x75\x74\x6c\x6f\x6f\x6b\x09\x6f\x07\x6f\x75\x74\x6c\x6f\x6f\x6b\x09\x6f')

exp_pkt = src_pkt
# -----------------------------------------------------------
logger.info("Sending packet on port %d", ig_port)
testutils.send_packet(self, ig_port, src_pkt)
# -----------------------------------------------------------
logger.info("Verify packet on port %d", eg_port)
testutils.verify_packets(self, exp_pkt, [eg_port])
logger.info("Verify no other packets")
testutils.verify_no_other_packets(self, 0, 1)
```

The Problem With Manual Testing

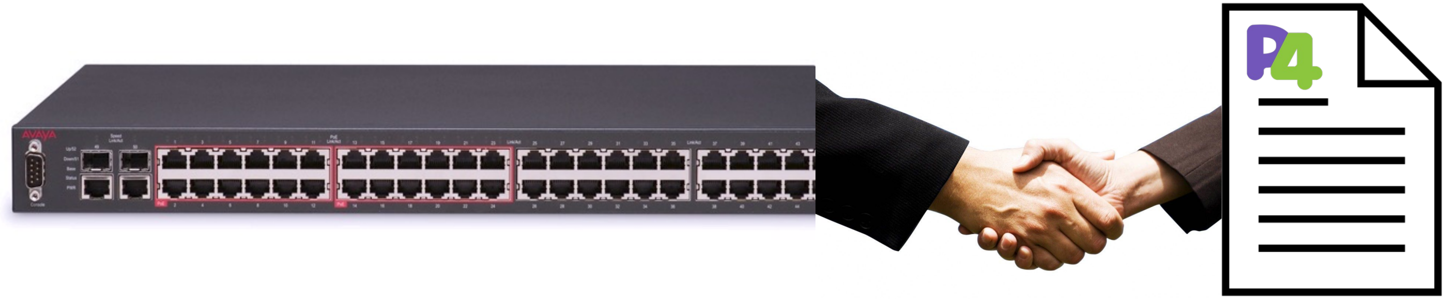
- Return of investment for a test is **unclear**
  - What does this test actually cover?
  - Have we covered enough?
- Writing packet tests is **hard**
  - Inputs are sequences of bits
  - Tedious boilerplate required to test a single feature



⇨ We do not write that many end-to-end tests for network equipment

This is also true for programmable networks!

We Can Do Better



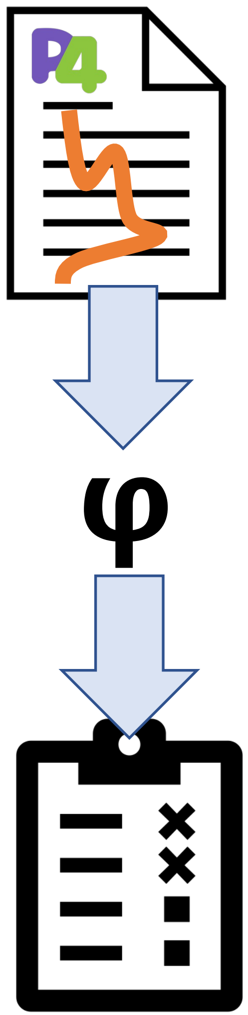
- P4 gives a machine-readable contract on how the network device will behave

- We have **full** access to the P4 source code and its semantics
  - We also **know** how the target device interprets P4 code
  - Rich body of software engineering research and formal methods exists

Let’s automate testing!

Idea: Generate Tests With Symbolic Execution

- Walk a random path through the P4 program
- Collect up a symbolic path constraint
- Encode the constraint as a first-order logic formula
- Use an SMT solver to find a model (if it exists)
- Convert the model into an input **and** output test
- Emit the test and the associated program trace



Two Conflicting Requirements

Do **not** tailor to a target device

(Tofino, eBPF/XDP, BMv2, IPU…)

Model **whole program semantics**

(How does the HW **actually** interpret the P4 code?)

No existing tool bridges this gap!

p4testgen

- **Target-independent**
  - Designed to support test case generation for **any** P4 target
  - **Anyone** can add their own target as an extension (we can reuse code!)

- **Whole program semantics**
  - Covers the semantics of the P4 program **and** the device that executes the program
  - Implicitly models the device **specification** for single packet tests

- **Generates inputs and outputs**
  - p4testgen not only checks crashes, but also semantically incorrect behavior

p4testgen: Example

```
parser parser(...) {
    pkt.extract(hdr.eth);
}
control ingress(...) {
    action set_output_port(bit<9> out) {
        meta.output_port = out;
    }
    table forward_table {
        key = { h.eth.src: exact; }
        actions = { noop; // default action
                    set_output_port; }
    }        ?
    h.eth.src = 48w1;
    forward_table.apply();
}
control deparser(...) {
    pkt.emit(hdr.eth);
}
```

Generated test

Required input

Input port	$input_port
Input packet	$eth.dst ++ $eth.src ++ $eth.type ++ $payload

Required control plane configuration

Table key	48w1
Chosen action	"set_output_port"
Action argument	$out

Expected output

Output packet	$eth.dst ++ 48w1 ++ $eth.type ++ $payload
Output port	$out

p4testgen: Example - Solved

```
parser parser(...) {
    pkt.extract(hdr.eth);
}
control ingress(...) {
    action set_output_port(bit<9> out) {
        meta.output_port = hash(h.eth.dst, out);
    }
    table forward_table {
        key = { h.eth.src: exact; }
        actions = { noop; // defau
                    set_output_port; }

    h.eth.src = 48w1;
    forward_table.apply();
}
control deparser(...) {
    pkt.emit(hdr.eth);
}
```

What if the packet is too short?

What if this is a hash function?

What if this table does not match?

Generated test

Required input

Input port	9w0
Input packet	48w0 ++ 48w0 ++ 16w0 ++ 1500w0

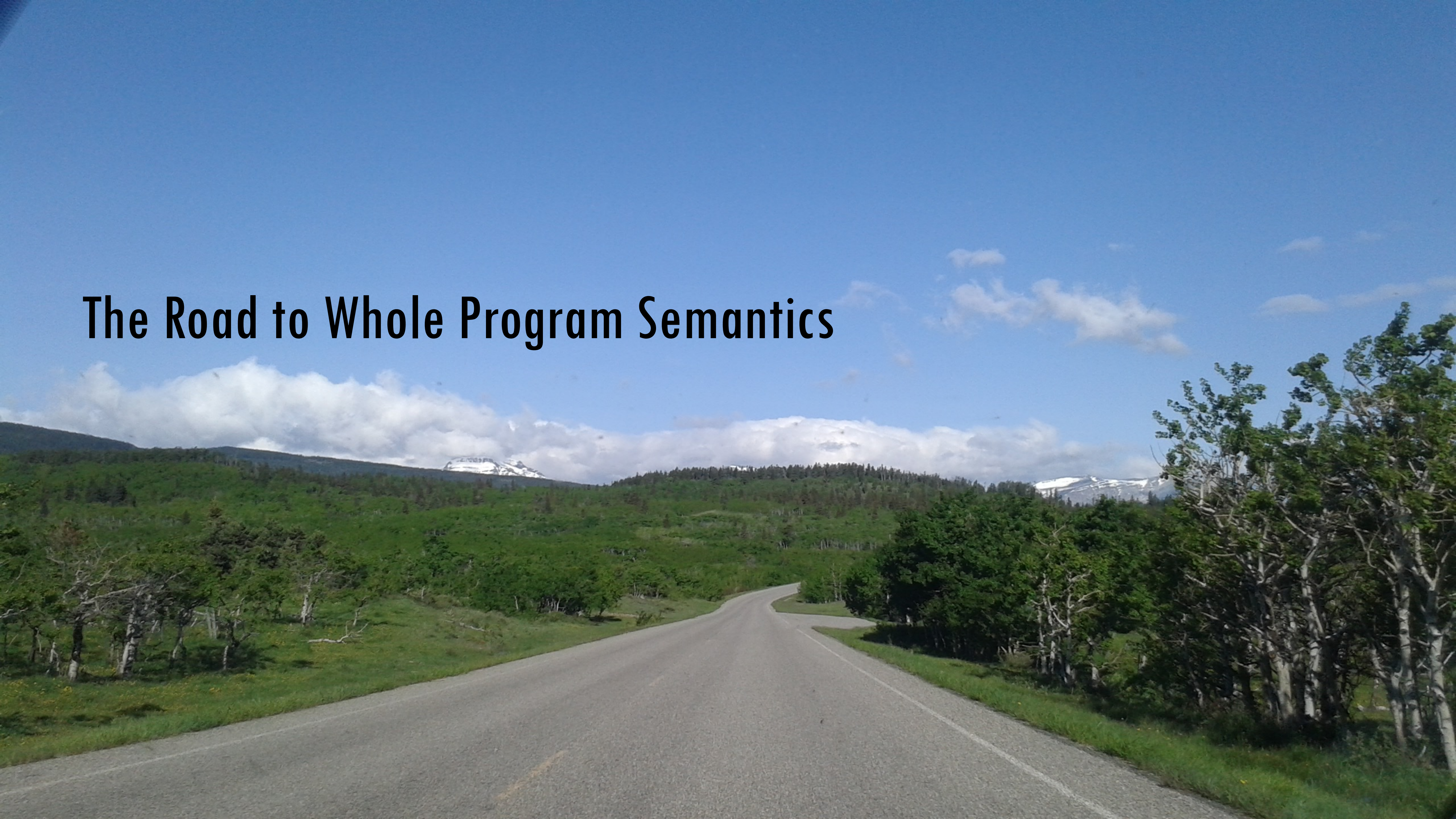
Required control plane configuration

Table key	48w1
Chosen action	"set_output_port"
Action argument	9w2

Expected output

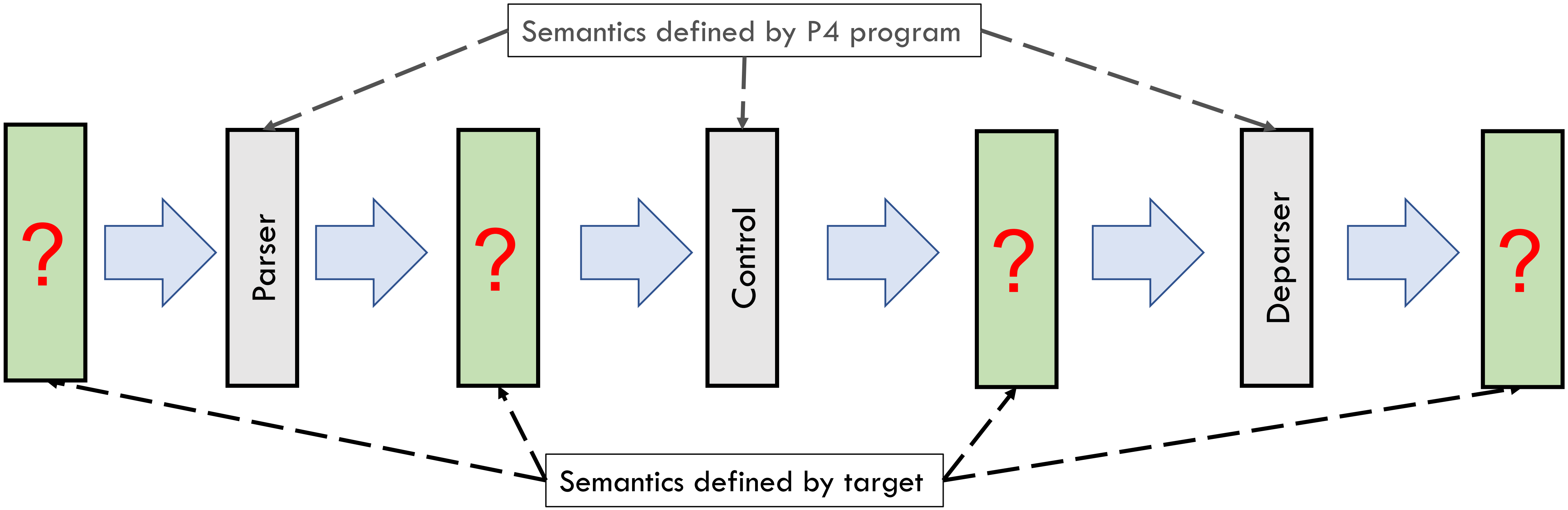
Output packet	48w0 ++ 48w1 ++ 48w0 ++ 1500w0
Output port	9w2

The Road to Whole Program Semantics



Challenge 1 - Semantics for the Entire Pipeline

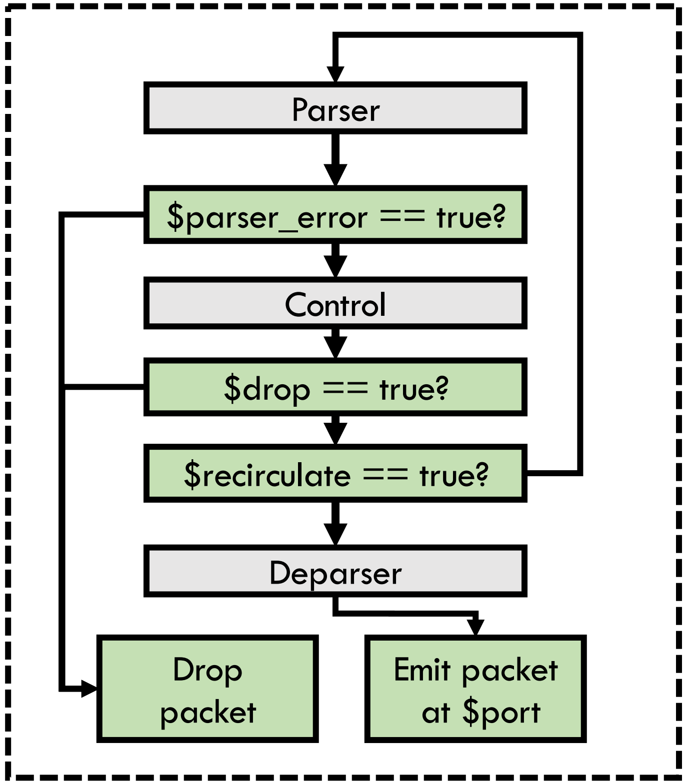
- P4 programs only describe the programmable blocks of the target
- How can we know what happens in-between these blocks?



Solution 1 - Architecture Model

- Each target **must** describe an architecture model
    - Packets can be dropped, recirculated, or modified
- Current architecture model is a C++ DSL
    - Converted into custom control flow (right)
    - We are planning to model this in P4 going forward
- Reusable
    - Common code can be reused across targets

Architecture description



Technical detail: We use continuations to implement this model

Challenge 2 - Avoiding Unreliable Tests

1. Some program state is undefined or random
    - We have **no** control over this state, and we can **not** know the generated output
    - What to do when a table reads on an uninitialized key field? How can we know we match?

2. Not all target functions (externs) can be modelled using first-order logic
    - Expressing **hash functions** is difficult and solving them can be very slow
    - But we still **need** a concrete mapping to avoid producing unreliable tests

Solution 2.1 - Taint Tracking

1. Mark state affected by unreliable program segments tainted
   - E.g.: An assignment that reads from an uninitialized variable will taint the destination

2. Resolve tainted reads as needed:
   - Either further propagate taint or resolve taint directly at the program node
   - E.g.: An if statement with a tainted condition could execute either branch

3. When generating a test…
   - Use "don't care" settings for unreliable outputs (e.g., tainted segments of the output packet)
   - Discard the test wherever we have no choice (e.g., tainted output ports)

Solution 2.2 - Concolic Execution

- We could mark complicated externs tainted, but this will cause taint explosion
- Use **concolic execution** instead

Approach

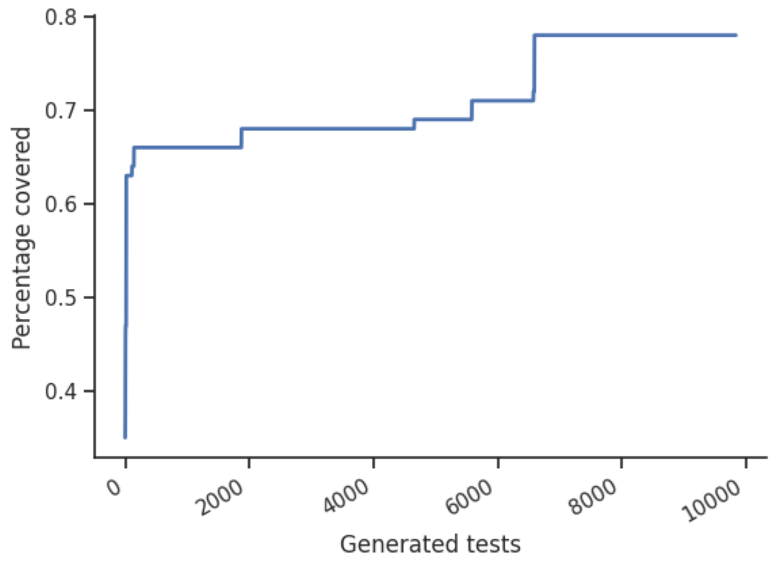
1. Pick a set of random inputs for the function
2. Calculate the function output using these inputs
3. Encode these inputs and outputs as constraints for the SMT solver
4. Check whether the solver can find a model
5. Yes? Done.
6. No? Try again or abandon this particular branch.

Current Status

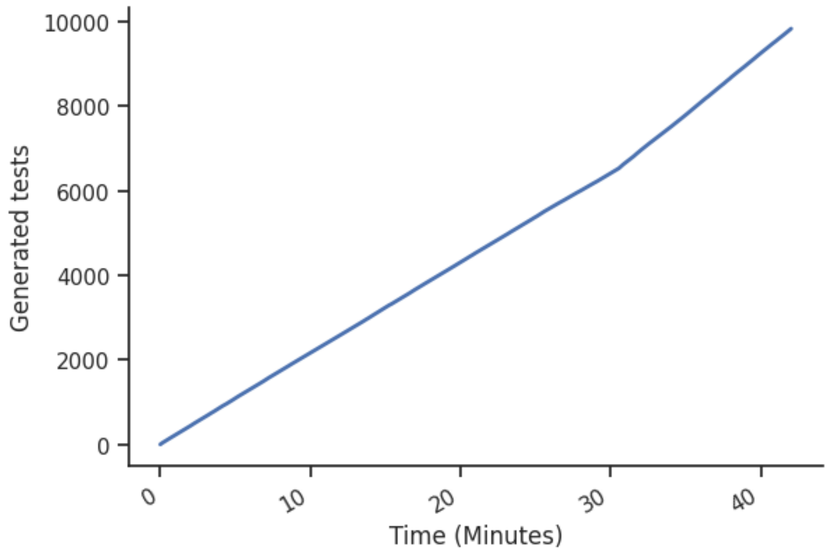
p4testgen: PINS Case Study

- Ran p4testgen on a P4 model of a fixed-function switch
  - Part of the [P4 Integrated Networking Stack (PINS)]
- We cover all **reachable** statements in the program
  - Many branches are hidden within extern execution (e.g., extract)

Coverage over time



Test case generation per minute



p4testgen: Technical Details

- p4testgen is written as a back end for P4C – the P4 reference compiler
  - Uses P4C's visitor framework to implement the interpreter
  - p4testgen benefits from improvements to P4C
- p4testgen currently supports test generation for the Tofino and BMv2 targets
  - Goal is to implement the full device specification for single packet tests
  - Also includes end-to-end testing scripts
- p4testgen supports the packet/simple test framework (PTF/STF)
  - Other test frameworks can easily be added

p4testgen: Limitations

- We only generate **single packet tests**
  - No load testing
  - Can not check for race conditions or timing issues
  - Can not check for runtime issues with shared register state or memory writes
- **No control** over metadata or state
  - This is purely an implementation problem. HW target may not support this
- No silver bullet
  - Target developers still need to spend time to implement the **device specification**

Who Can (Or Should) Use p4testgen?

- **Compiler developers can use p4testgen to…**
  - …prototype new back ends
- **Device manufacturers can use p4testgen to…**
  - …specify the desired behavior of the target device and validate execution
- **Resellers/vendors can use p4testgen to…**
  - …certify they are compliant with the manufacturer and P4 specification
- **Device users can use p4testgen to…**
  - …automatically generate validation tests for their P4 programs

Disclaimer

- Intel technologies may require enabled hardware, software or service activation.
- No product or component can be absolutely secure.
- Your costs and results may vary.
-

Thank You

Contact:

Fabian Ruffy – fruffy@nyu.edu
Nate Foster – nate.foster@intel.com

Challenge 1 - Flexible Semantics for P4

Three requirements

1. p4testgen must be an **oracle** for the P4 language
   - Should not worry about P4 semantics when writing a p4testgen extension

2. p4testgen must be as **broad** as the P4 language specification
   - Leave room for target-specific behavior (e.g., drop packet when certain metadata is set)

3. We must be able to **stop/resume** execution
   - We want to continue generating tests after we have completed one branch
   - We also want to use different test generation strategies and easily switch branches

Solution 1 - The P4 Abstract Machine

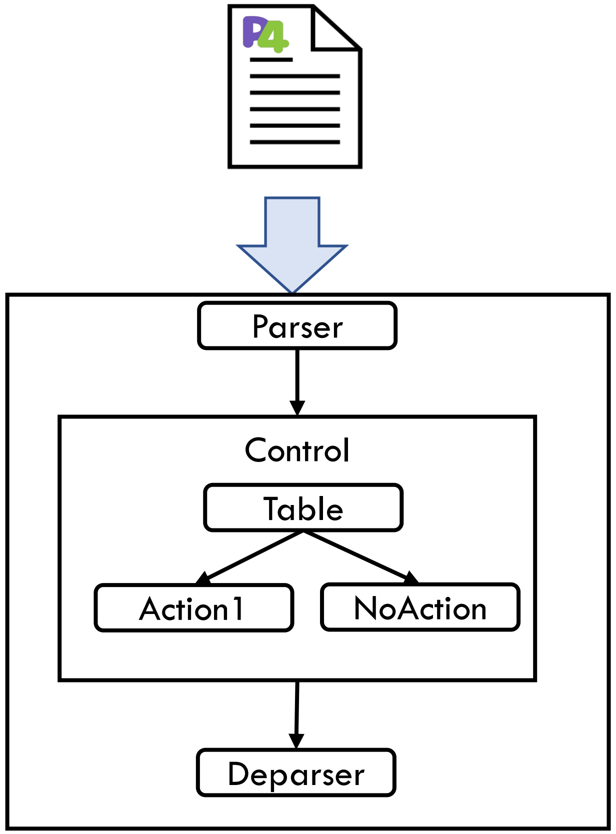
- Convert P4 code into tree of program nodes
- Walk each branch and builds program state
- Emit test at each leaf node, then backtrack (**Depth first**)

State is fully independent

- Can easily switch between program branches

Every node can change subsequent program execution

- Target extensions can implement their own control flow
- Target can change the semantics of every program node (Tables!)



Program nodes

Technical detail: We use continuations to implement this model