P4Testgen – An Extensible Test Oracle for P4

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

NEW YORK UNIVERSITY

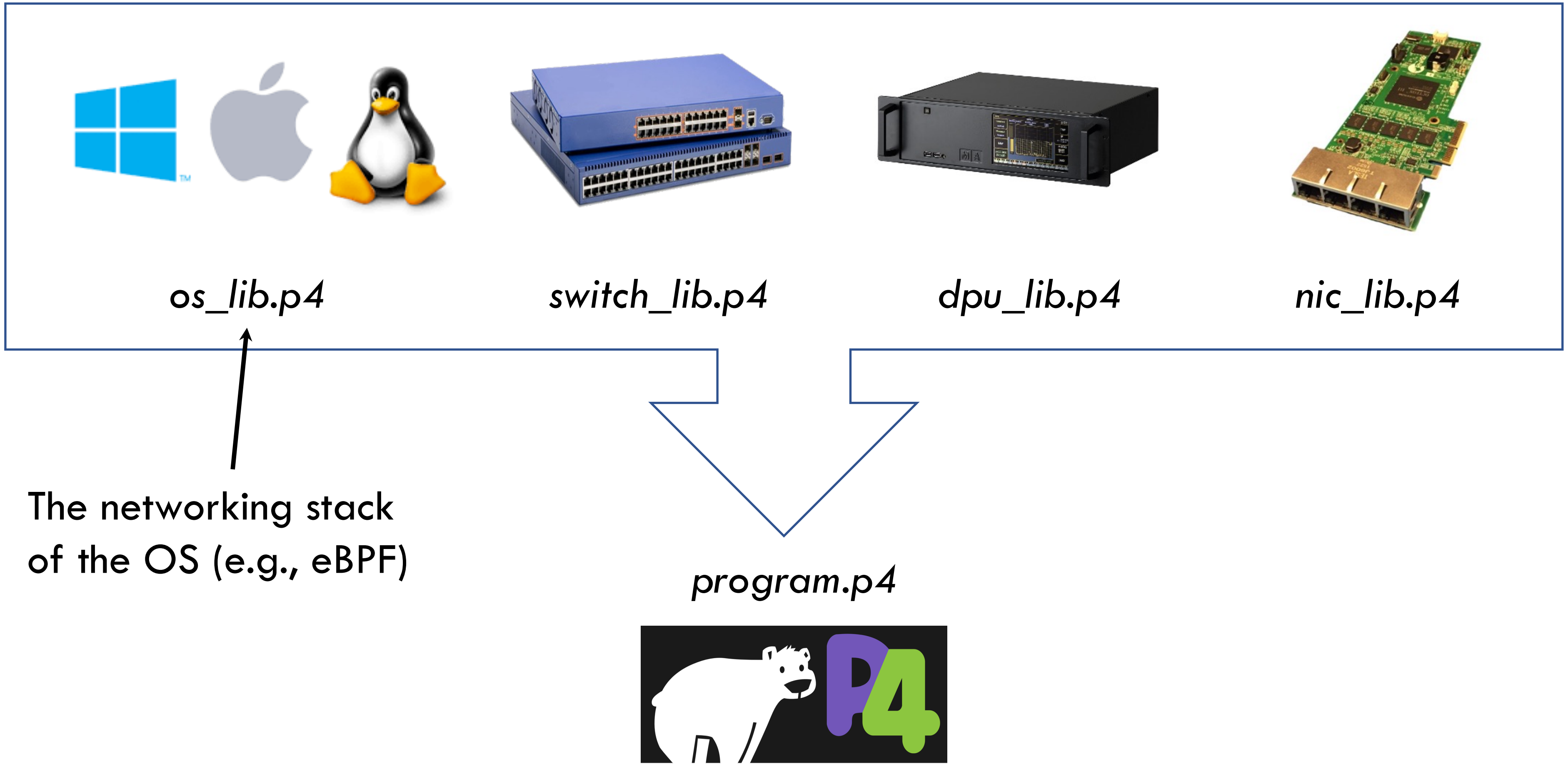
LitSoft Private Enterprise

intel

Brief Outline

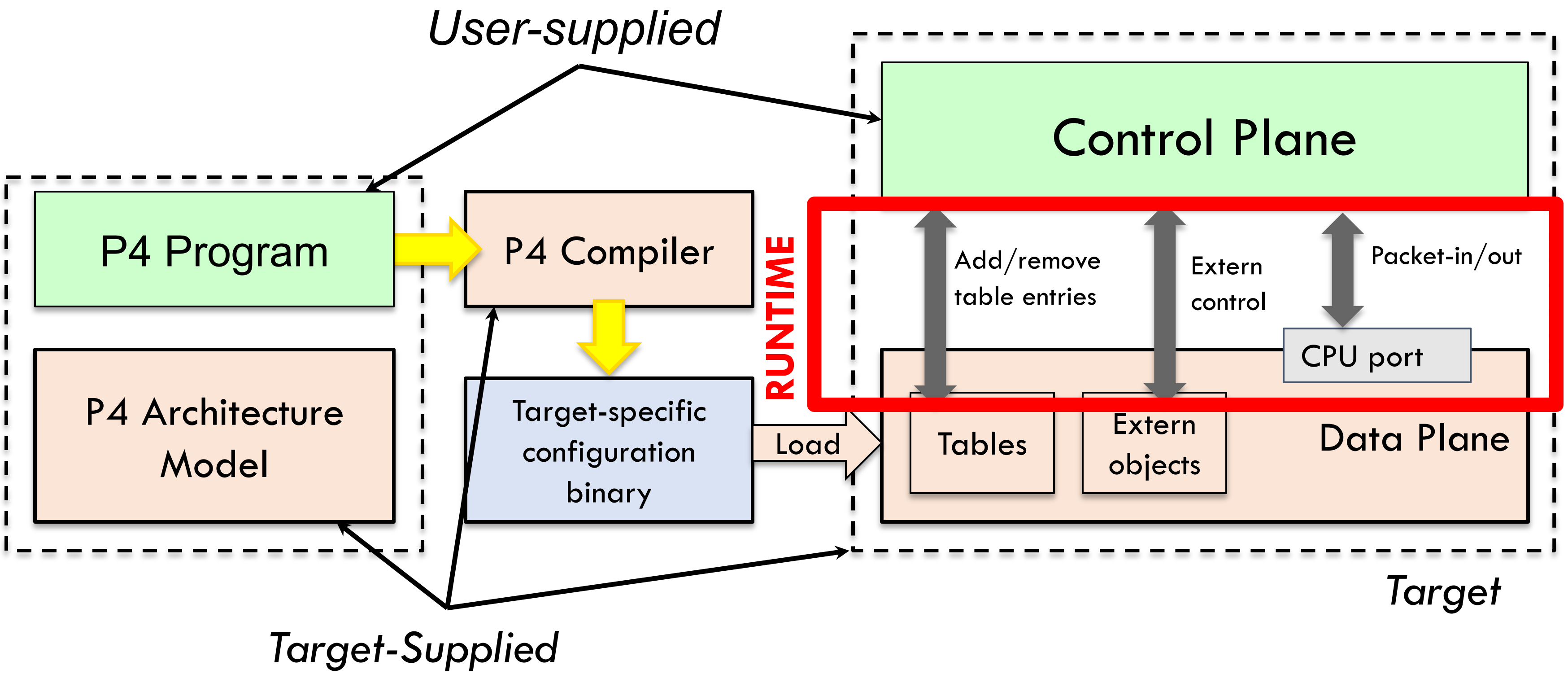
- Refresher on P4 Targets
- How are network devices tested? What are the problems?
- P4Testgen (Overview)
- P4Testgen (Details)
- P4Testgen (Status)

P4$_{16}$ Compiles to Many Targets



The networking stack of the OS (e.g., eBPF)

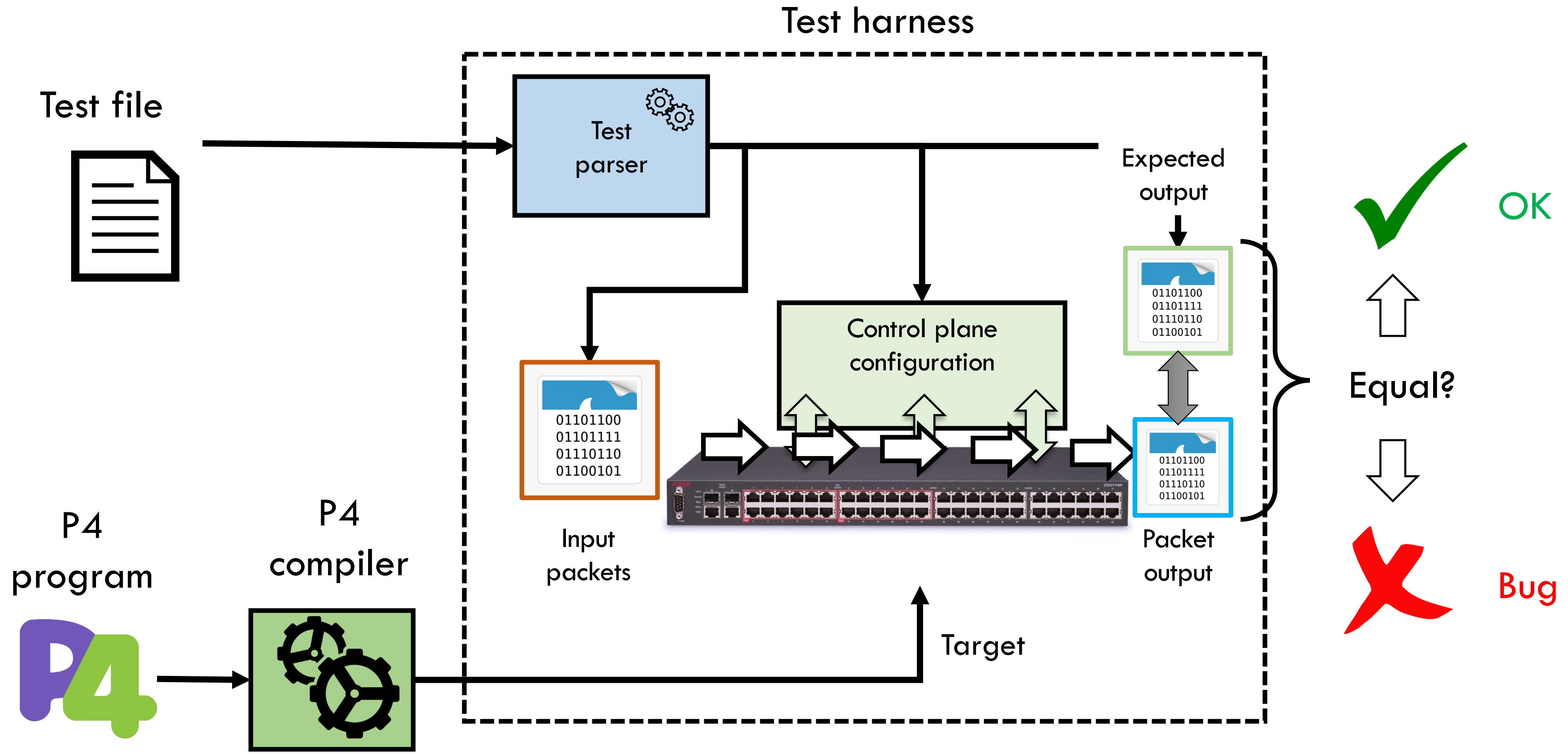
P4$_{16}$ Target-Independent Software Workflow



Slide Credit: Mihai Budiu

Testing Your Target

How is a $P4_{16}$ Target Tested?

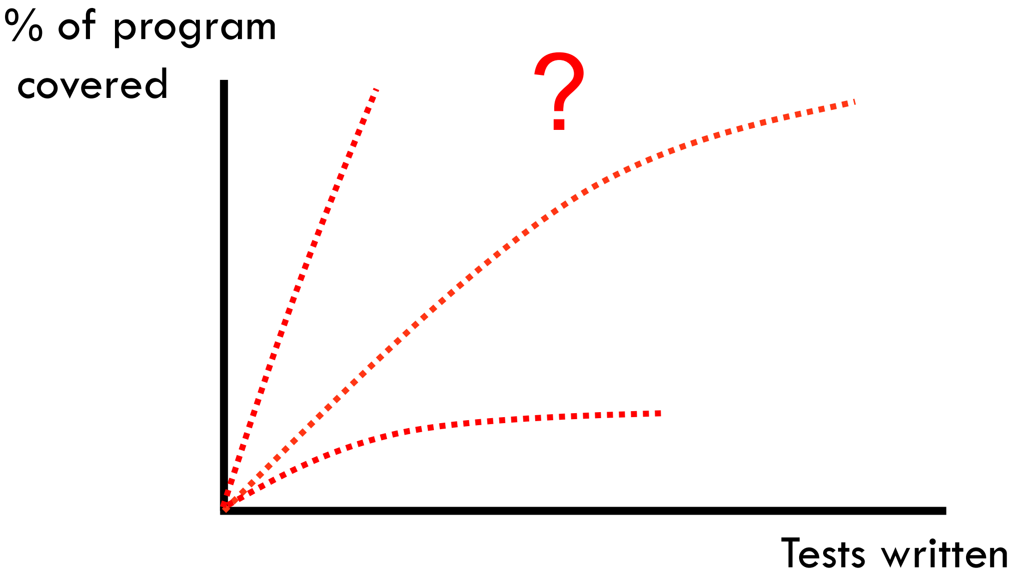


Reality

```
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)
```

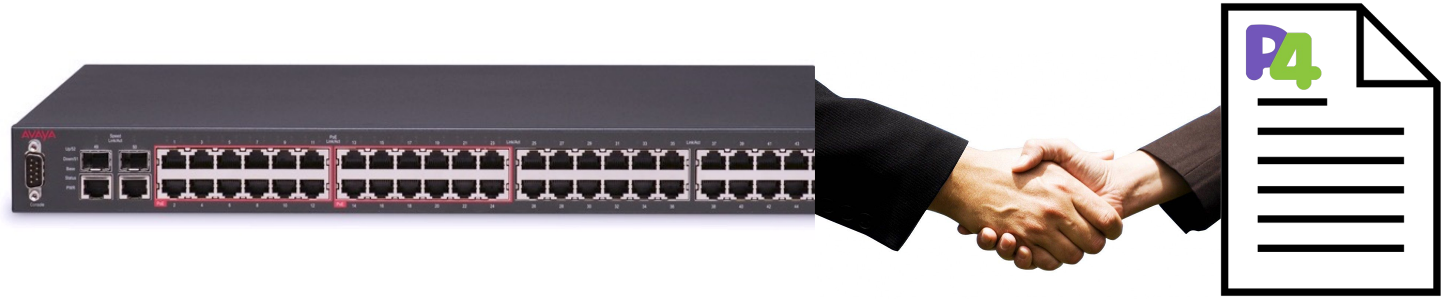
The Problem With Manual Testing

- 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 switch programs.

We Can Do Better



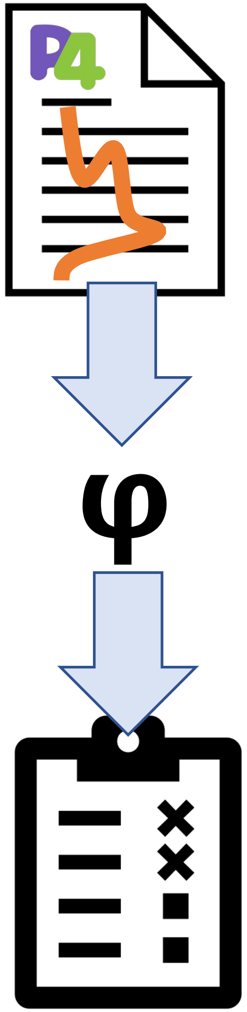
- P4 gives a machine-readable contract on how the network device will behave.

- 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 target **actually** interpret the P4 code?)

No existing tool bridges this gap!

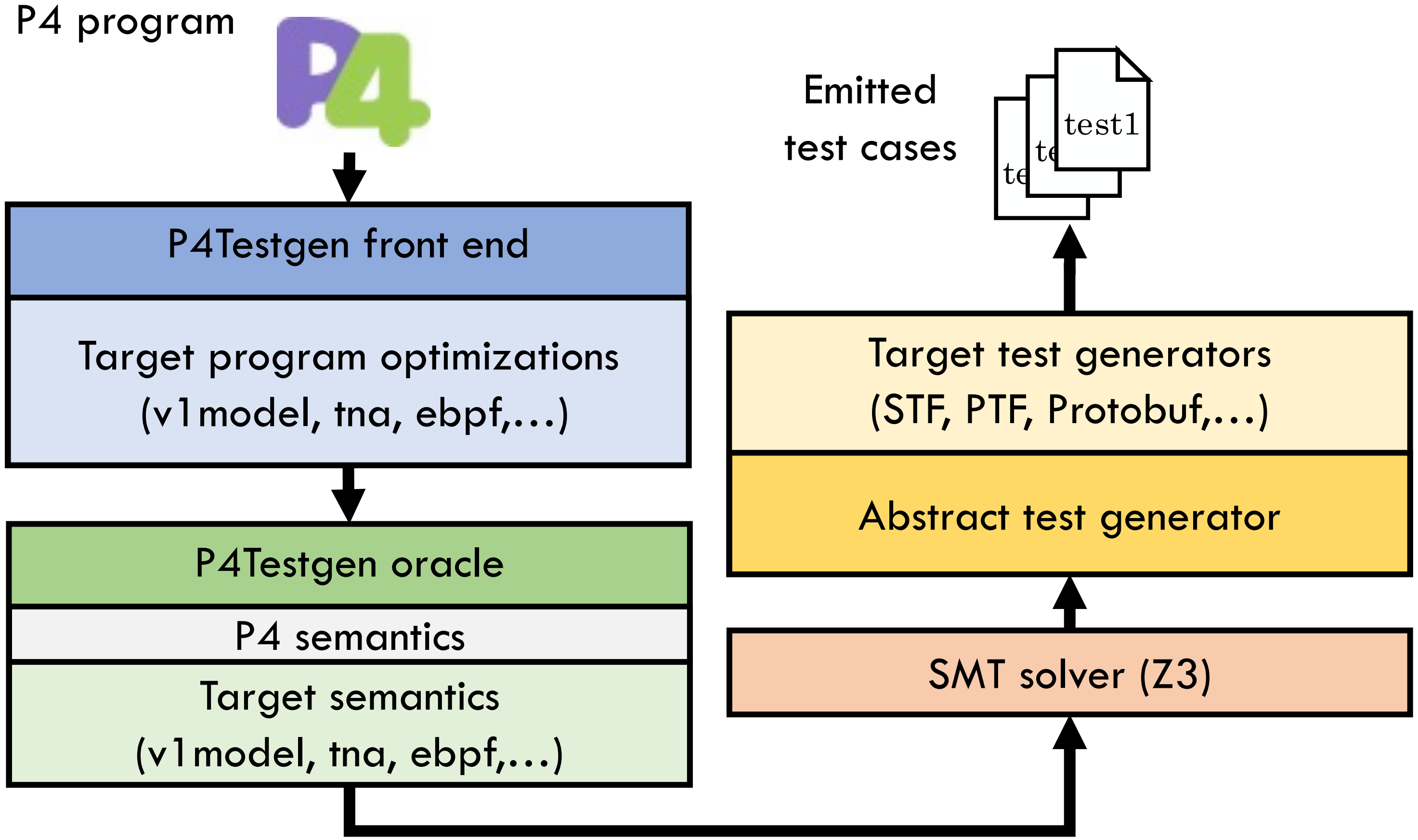
P4Testgen

- **Generates inputs and outputs.**
  - P4Testgen not only checks crashes, but also semantically incorrect behavior.

- **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.

P4Testgen: Workflow



P4 program

P4Testgen front end

Target program optimizations
(v1model, tna, ebpf,…)

P4Testgen oracle

P4 semantics

Target semantics
(v1model, tna, ebpf,…)

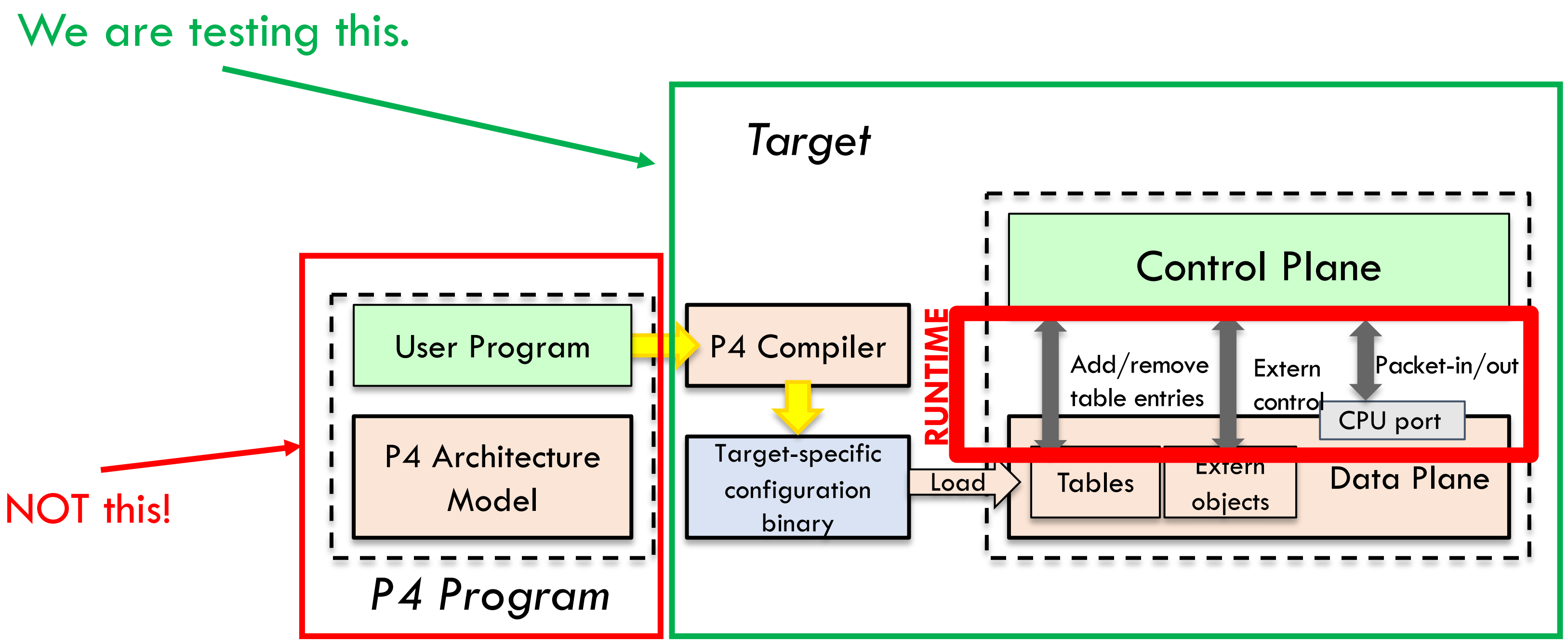
SMT solver (Z3)

Abstract test generator

Target test generators
(STF, PTF, Protobuf,…)

Emitted
test cases

P4Testgen Checks The Target Stack - Not P4 Code



DEMO

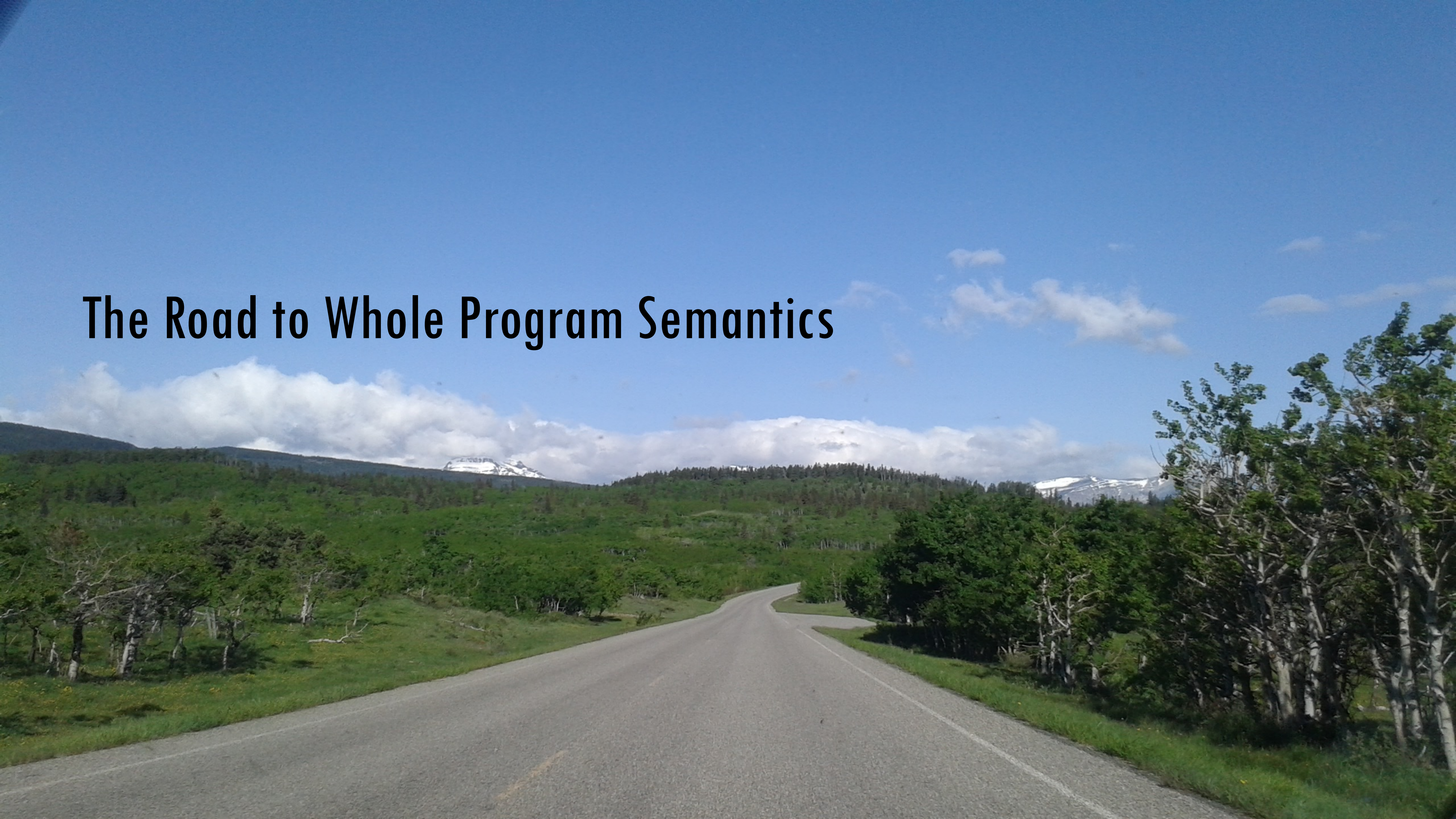
Who Benefits From P4Testgen?

- **Compiler developers can use P4Testgen to…**
    - …validate their back end optimization passes.

- **Network operators can use P4Testgen to…**
    - …generate tests for their programmable devices and deployed programs.

- **Equipment vendors can use P4Testgen to…**
    - …certify they are compliant with the manufacturer and P4 specification.

- **Users of fixed-function devices can use P4Testgen to…**
    - …derive validation tests from the P4 model of the device under test.



The Road to Whole Program Semantics

Whole-Program Semantics

Three Requirements

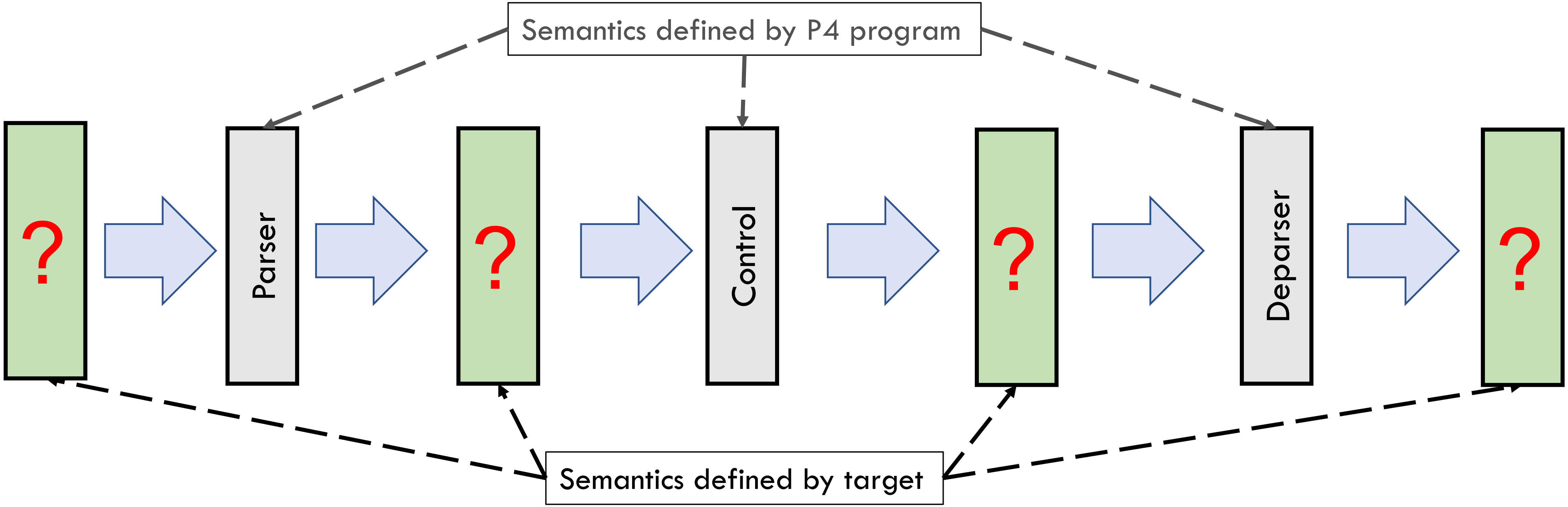
1. P4Testgen must be an **oracle** for the P4 language.
    - Should not worry about P4 semantics when writing a P4Testgen extension.

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. P4Testgen must be **resilient** against target quirks.
    - Detect or mitigate non-determinism.
    - Model target-environment constraints (e.g., influence of packet size on processing semantics).
    - Allow for non-standard interpretation of the P4 specification.

Challenge 1 - How to Model a Target's Data and Control Flow?

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



Semantics defined by P4 program

Semantics defined by target

Solution 1.1 - The P4Testgen Abstract Machine

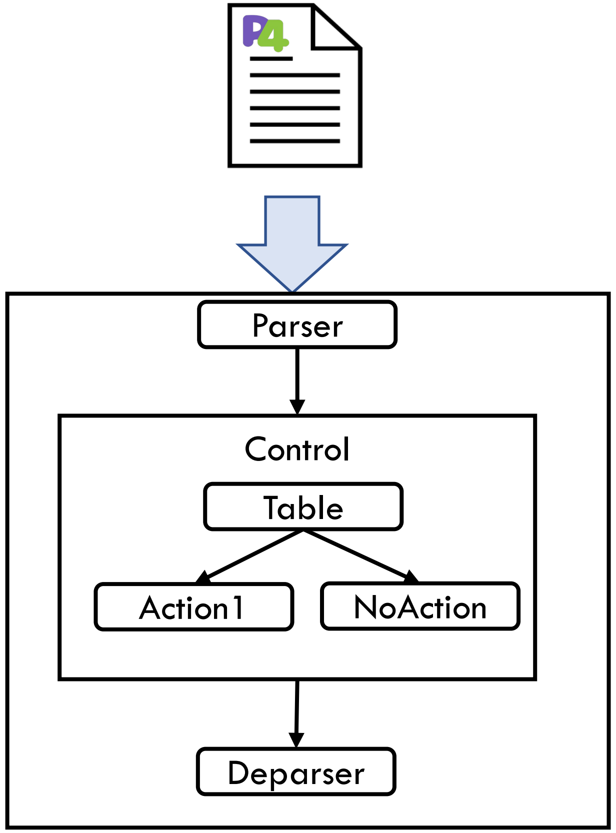
1. Convert P4 code into tree of program nodes (P4C IR).
2. Walk each branch and build program state.
3. Emit test at each leaf node, then backtrack (**depth first**).

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 (P4 Tables!).



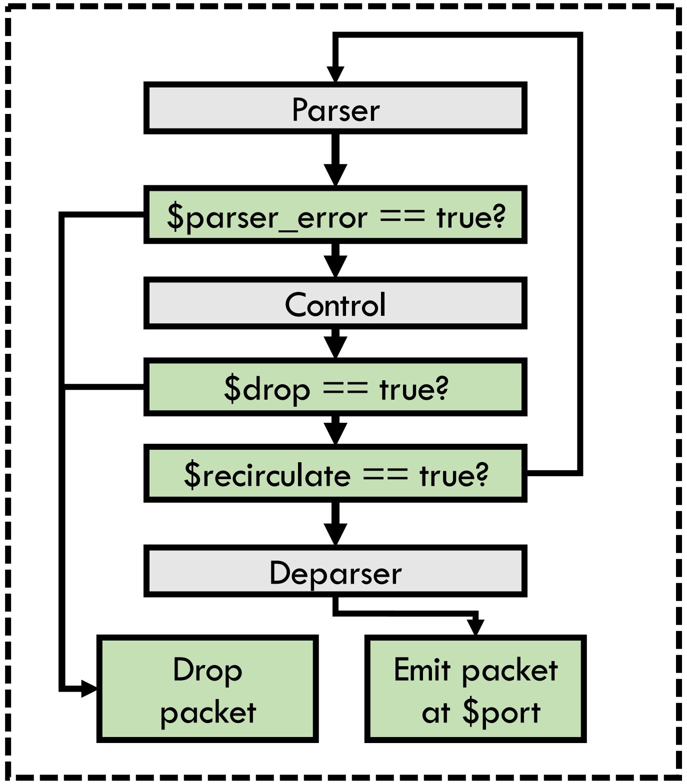
Program nodes

Technical detail: We use continuations to implement this model.

Solution 1.2 - Pipeline Templates

- 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.
  - Ideally, we would want to express this in P4 only.
- Useful side-effect: Reusability.
  - There is significant overlap in network processing logic.
  - Common code can be reused across targets.

Architecture description



Challenge 2 - Dealing with Nondeterminism and Complexity

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.
    - Example: 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.
    - Example: 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: Extensions

- `v1model` (BMv2)
  - Supports the p4-constraints framework, which limits eligible table entries.
- `tna` (Tofino 1) and `t2na` (Tofino 2)
  - Tofino has two parsers and deparsers.
  - Tofino pre- and appends metadata to each packet.
- `ebpf_model` (linux kernel eBPF)
  - ebpf can not modify packets, the model has no deparser.

P4Testgen: Evaluation

- Correctness is checked by running packet tests on respective model.
  - In total, ~2000 program tests per commit.
- We execute on the P4C and Tofino program suites.
  - Filed **25 bugs** (9 in BMv2, 16 in Tofino).
  - Most of the bugs are compiler bugs (some are incorrect transformations).
- Produces too many tests for Tofino switch.p4 flavours
  - Stopped generating at >1,000,000 tests.
  - P4Testgen produces too many branches because it handles many edge cases.
    - We are working on making this practical.

P4Testgen: Future Work

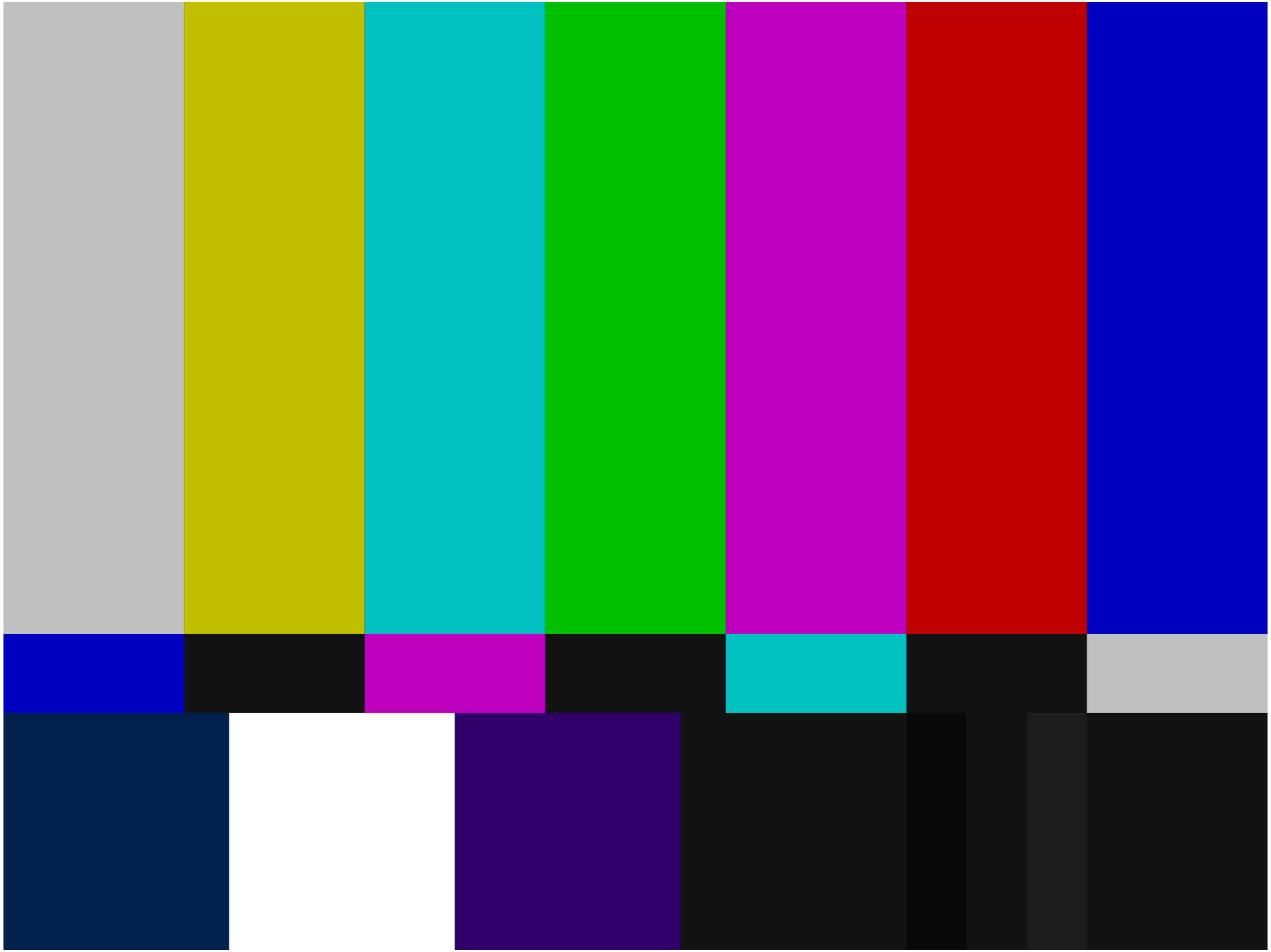
- Path queries to produce targeted tests.
  - Example: “**Only** produce tests that hit table ipv4_acl with a valid ipv4 packet.”
- Exploration strategies to maximize coverage.
  - Example: “Pick the branch that contains unexplored program nodes.”
- Implement more extensions.
  - Test P4Testgen’s limits in expressiveness.
  - Explore targets with non-trivial control flow.
  - Example: P4DPDK or general P4 FPGA targets.

P4Testgen: Conclusion

- Test-case oracle that produces input-output packet tests for P4 targets.

- Implements **whole-programs semantics** to model P4 target pipelines.
  - Requires pipeline templates, taint analysis, concolic execution.

- Supported extensions: v1model (BMv2), tna/t2na (Tofino), and eBPF
  - Initial results: Found ~**25** bugs in the BMv2 and Tofino toolchains.



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`

`48w1 = 48 bit wide number with value 1`

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

`48w1 = 48 bit wide number with value 1`