P4Testgen: An Extensible Test Oracle For P4

Fabian Ruffy[†] Jed Liu[¶] Prathima Kotikalapudi[‡] Vojtěch Havel[‡] Rob Sherwood[‡]

Vlad Dubina[**] Volodymyr Peschanenko[**] Nate Foster[*‡] Anirudh Sivaraman[†]

[¶]*Akita Software* [⋆]*Cornell University* [‡]*Intel* [**]*Litsoft* [†]*New York University*

Abstract

We present P4Testgen, a test oracle for the P4$_{16}$ language that supports automatic generation of packet tests for any P4-programmable device. Given a P4 program and sufficient time, P4Testgen generates tests that cover every reachable statement in the input program. Each generated test consists of an input packet, control-plane configuration, and output packet(s), and can be executed in software or on hardware.

Unlike prior work, P4Testgen is open source and extensible, making it a general resource for the community. P4Testgen not only covers the full P4$_{16}$ language specification, it also supports modeling the semantics of an entire packet-processing pipeline, including target-specific behaviors—i.e., *whole-program semantics*. Handling aspects of packet processing that lie outside of the official specification is critical for supporting real-world targets (e.g., switches, NICs, end-host stacks). In addition, P4Testgen uses taint tracking and concolic execution to model complex externs (e.g., checksums and hash functions) that have been omitted by other tools, and ensures the generated tests are correct and deterministic.

We have instantiated P4Testgen to build test oracles for the V1model, eBPF, and the Tofino (TNA and T2NA) architectures; each of these extensions only required effort commensurate with the complexity of the target. We validated the tests generated by P4Testgen by running them across the entire P4C program test suite as well as the Tofino programs supplied with Intel's P4 Studio. In just a few months using the tool, we discovered and confirmed 25 bugs in the mature, production toolchains for BMv2 and Tofino, and are conducting ongoing investigations into further faults uncovered by P4Testgen.

1 Introduction

We present P4Testgen, a *test oracle* for the P4$_{16}$ [9] language. Given a P4 program and sufficient time, P4Testgen automatically generates tests that cover every reachable statement (or path) in the input program. Each generated test consists of an input packet, control-plane configuration, and output packet(s), and can be executed on software or hardware targets.

P4Testgen generates tests to validate the *implementation* of a P4 program. It does not test whether a P4 program is written correctly (e.g., if it implements a given protocol correctly) but whether the target and its toolchain (i.e., the compiler [7], control plane [8, 19], and various API layers [18, 22, 36]) implement the behaviors specified by the program.

Tests generated by P4Testgen can be used in a multitude of ways (§ 2.2). Chip developers can use generated tests to validate their target's associated toolchain; compiler developers can use the tests for debugging code transformations and optimizations; and equipment vendors and network owners can use the tests to check that both fixed-function and programmable targets implement behaviors specified in P4, including custom and standard protocols.

P4Testgen is an open-source effort, similar to the P4 reference compiler (P4C) [7] and behavioral model (BMv2) [3]. Because the P4$_{16}$ language allows the structure and capabilities of the target pipeline to be specified as an *architecture* in the program, P4Testgen does not bake in assumptions about the underlying target. This distinguishes P4Testgen from prior work, which focuses on building tools for specific targets—e.g., Meissa and p4v for the Tofino switch [29, 48], SwitchV for legacy fixed-function switches [1], and `p4pktgen` for BMv2 [33]. In contrast, the main goal of P4Testgen is to be *extensible* to any P4 target. Other projects tackle complementary goals such as scalability [47, 48] and conformance of P4 programs to external specifications [14, 15, 24, 29, 42, 44, 45]. These techniques can be incorporated into P4Testgen in the future.

P4Testgen is designed to meet needs of the P4 community. The set of P4-enabled targets is growing rapidly [3, 10, 11, 21, 34, 35], and each new target requires validation. Unfortunately, current testing tools are not general, which fragments tool developer effort across the ecosystem and makes building new P4 targets needlessly hard. Developing validation tools requires having expertise in formal methods as well as a detailed understanding of the P4 language and the specific behaviors and quirks of the underlying target. Finding developers that are able to satisfy this trifecta is costly and difficult. The consequence is that many targets do not come with adequate test tooling, which creates friction for adoption of P4 targets.

Our position is that this fragmentation is undesirable and entirely avoidable. While there may be use-cases that warrant the development of specialized tooling, the common case—i.e., the generation of input–output pairs for a given P4 program—can be derived from the semantics of the P4 language, in a manner that is largely decoupled from the target. Developing a common platform for validation tools has several benefits. First, common software infrastructure (lexer, parser, type checker, etc.) and an interpreter that realizes the P4 semantics can be implemented just once and shared across many tools. Second, since it is open-source, better testing or verification techniques (e.g., for path selection or coverage) can be contributed to P4Testgen and benefit the whole community.

To provide a common tool platform, P4Testgen decomposes the *whole-program semantics* of a P4 program into (i) the core P4 language semantics mandated by the P4 language specification, and (ii) the target-specific interpretation of the P4 program. Concretely, a P4 program consists of multiple P4 programmable blocks (whose semantics are provided by the core language semantics) separated by interstitial target-specific elements (whose semantics are provided by target-specific extensions). Whole-program semantics are the main reason why P4Testgen can generate tests for a variety of P4 targets without sacrificing generality and accuracy.

P4Testgen's key technical contribution is developing whole-program semantics and addressing the following challenges:

1. **Pipeline templates**: Most targets perform additional processing that is not described by the P4 program itself. We use *pipeline templates* to succinctly describe target-specific differences in the pipeline-processing behavior.
2. **Target extensions and packet sizing**: Some targets do not implement the $P4_{16}$ specification to the letter and diverge from the default semantics of some language constructs. P4Testgen supports target-specific *extensions* to override default P4 behavior, including an intricate model of *packet-sizing*, which allows targets to flexibly change the size of packets during processing.
3. **Taint analysis**: Some targets exhibit non-deterministic behavior, making it hard to predict output packets. We use *taint analysis* to track the determined bits in P4 programs to ensure that generated tests are deterministic.
4. **Concolic execution**: Some targets have features that are too complex to be expressed in first-order logic. We use *concolic execution* [25,41] to model these features.

To validate our design for P4Testgen, we have built extensions for four different targets: the `v1model` [16] architecture of BMv2, the `ebpf_model` [20] for the Linux kernel, `tna` architecture for the Tofino 1 chip [10] and the `t2na` architecture for the Tofino 2 chip [10]. All four extensions implement their own target semantics and P4 code interpretation without requiring modification to the core parts of P4Testgen. We have tested the correctness of the P4Testgen oracle by generating input–output tests for the example P4 programs of the `v1model` and `tna`/`t2na` architectures. Executing the generated tests on the appropriate target toolchains, we have found 16 bugs in the toolchain of the Tofino compiler and 9 in the toolchain of BMv2. P4Testgen is available on GitHub under an Apache2 License at `https://github.com/p4lang/p4c/tree/main/backends/p4tools/modules/testgen`.

2 Motivation

P4 offers extraordinary flexibility for specifying data plane behavior, but the toolchains and targets used to implement P4 programs require extensive testing. With a P4-based system, the number of components that the network owner must administer is much larger than with traditional protocols and fixed-function devices, and functionality also evolves at software timescales. So as the P4 ecosystem matures, increased focus is being placed on tools for validating P4 implementations [1,5,13,28,33,39,47,48].

2.1 Why Automate Test-Case Generation?

A simple approach to validating P4 programs is to write input–output tests for specific features by hand. Each test comprises an input packet and control-plane configuration required to exercise the feature, and the expected output packet. The input packet is fed into a preconfigured target and the output is recorded. If the actual and expected outputs match, then the test passes and the target is deemed to implement the feature.

While this methodology is straightforward, it has two practical difficulties. First, the return on investment for writing tests is unclear. As a form of black-box testing, it can be hard to determine whether the test suite fully exercises the features being validated. Second, writing input–output tests is tedious. Even with high-level test frameworks [2,6], developers still often write long, detailed binary sequences to specify input and output packets as well as device configurations.

In practice, many network programmers do not write many tests (e.g., only around 300 tests are distributed with the Tofino SDE). Instead, they fall back to techniques like fuzz testing with arbitrary inputs [4,32]—an approach that is certainly useful, but does not provide a precise notion of coverage. The move to programmable devices only exacerbates the problems associated with validation as the set of features is not fixed in advance and can instead evolve over time.

At the same time, the move from fixed-function to programmable devices also presents an opportunity. With programmable devices, device functionality is precisely specified as a *program* in a domain-specific language (DSL) such as

P4 [9], which itself has clear semantics. Thus, we can draw from software engineering and testing research, to develop automated tools for testing network devices.

2.2 Who Can Use P4Testgen?

With a growing P4 ecosystem, a general-purpose test oracle like P4Testgen can serve many different kinds of users.

Network operators need assurance that their programmable devices and associated toolchains work as intended. With P4Testgen, they can automatically generate a unique set of tests for each version of their deployed programs.

Compiler developers need assurance that code transformations and optimizations are correct. With P4Testgen, they can automatically generate tests that exercise specific optimizations. If a generated test passes when the optimizations are disabled but fails when they are enabled, there is a bug in the compiler. We have used P4Testgen to uncover a variety of compiler bugs in P4 compiler back ends (§ 7).

Equipment vendors need assurance that their target adheres to published standards. Chip manufacturers can instantiate P4Testgen for their chip and associated toolchain, and generate tests to check that representative P4 programs are implemented correctly. Other OEMs and ODMs can ensure that their products correctly implement standard protocols as well as custom P4 programs, according to customer requirements.

Users of legacy toolchains are sometimes reluctant to upgrade due to concerns about bugs—new versions of a toolchain can break workarounds to known issues. With P4Testgen, these users can generate tests to increase the confidence because P4Testgen is not tied to any particular version of the toolchain. When they are ready to upgrade, they can execute tests generated by P4Testgen to ensure their P4 programs behave as intended.

Users of fixed-function devices can use P4 to model the functionality of their equipment [1]. P4Testgen can derive appropriate validation tests from the P4 model without the overhead of differential testing and can also provide coverage guarantees.

2.3 What Are the Concrete Challenges?

(1) P4 does not specify the behavior of the full pipeline. A P4 program only specifies the behavior of certain programmable blocks. It does not specify the execution order of those blocks, or how the output of one block feeds into the input of the next. For instance, Tofino's `tna` and `t2na` architectures contain independent ingress and egress pipelines, with a traffic manager between them. The traffic manager can forward, drop, multicast, clone, or recirculate packets, depending on their size, content, and associated metadata. None of these behaviors are captured in the P4 program itself.

(2) Many programs behave differently on different targets. The P4 specification delegates numerous decisions to targets and many targets deviate from the specification. For instance, match-action table execution can be customized using target-specific properties. Annotations can influence the semantics of headers and other language constructs. As another example, the P4 specification states that if extracting a header fails because the packet is too short, the parser should transition to the `reject` state and signal an error. However, the way this error is handled is left up to the target: some drop the packet, others consider the header uninitialized, while others silently add padding to initialize the header. App. A.1 contains a (non-exhaustive) list of such target-specific behaviors.

(3) P4 programs can be non-deterministic or even random. Not all parts of a P4 program are well-specified. For instance, reading from an uninitialized variable returns an undefined value. P4 programs may also invoke arbitrary extern functions, such as pseudo-random number generators, which produce unpredictable output. To ensure that generated tests are deterministic, P4Testgen needs facilities to track program segments that may cause unpredictable output. These outputs can then be mitigated by providing hints to P4Testgen's oracle or test back end to appropriately address flaky output bits.

(4) P4 programs rely on computations that can not be easily encoded into first-order logic. Like many other automated verification tools, P4Testgen relies on a first-order theorem prover (i.e., SAT/SMT solver) under the hood. However, not all data plane computations can easily be encoded into first-order logic—e.g., checksums and other hash functions, or programs that modify the size of the packet using dynamic values. Consider a program that invokes the `advance` function, which increments the parser cursor, on a previously-parsed value. Modeling this behavior precisely either requires bit vectors of symbolic width, which is not well-supported in solvers, or branching on every value, which is impractical.

2.4 What Are P4Testgen's Coverage Goals?

Existing tools such as Meissa [48] and FP4 [47] purport to provide full coverage, but their guarantees are subject to some important qualifications. Meissa tracks program coverage only for the limited set of paths that satisfy programmer-specified preconditions (written in the LPI specification language [45]). It also assumes that tables are pre-populated, which substantially reduces the set of feasible paths. Similarly, FP4 only tracks coverage of the table actions executed in the program. In general, ensuring full path coverage is a challenge due to the "branchy" nature of real-world P4 programs, which causes path explosion. For P4Testgen, we focus on statement coverage as our main metric. Our goal is for P4Testgen's test case generation to scale in proportion to the size of the program, and to ensure a good distribution of tests across statements.

```
parser Parser(...) {
    pkt.extract(hdr.eth);
    transition accept;
}
control Ingress(...) {
    action set_out(bit<9> port) {
        meta.output_port = port;
    }
    table forward_table {
        key = { h.eth.type: exact; @name("type")
          }
        actions = { noop; // Default action.
                    set_out; }
    }
    h.eth.type = 0xBEEF;
    forward_table.apply();
}
```

(a) P4 program that forwards using the source MAC.

```
parser Parser(...) {
    pkt.extract(hdr.eth);
    transition accept;
}
control Verify(...) {
    meta.checksum_err = verify_checksum(
    hdr.eth.isValid(),
    {hdr.eth.dst, hdr.eth.src},
    hdr.eth.type);
}
control Ingress(...) {
    if (meta.checksum_err == 1) {
        mark_to_drop(); // Drop packet.
    }
}
```

(b) P4 program that validates the Ethernet checksum.

```
 1 Input Packet                                      |Output packet                                     | Table configuration
 2 Size In eth.dst      eth.src      eth.type |Size Out eth.dst      eth.src      eth.type |
 3 --- Example 1                                   |                                                   |
 4 112  0   000000000000 000000000000 0000     |112  0   000000000000 000000000000 BEEF     | N/A
 5 112  0   000000000000 000000000000 0000     |112  2   000000000000 000000000000 BEEF     | match(type=0xBEEF),action(set_out(2))
 6 112  0   000000000000 000000000000 0000     |112  0   000000000000 000000000000 BEEF     | match(type=0xBEEF),action(noop())
 7 96   0   000000000000 000000000000          |96   0   000000000000 000000000000          | N/A
 8 --- Example 2                                   |                                                   |
 9 96   0   BADC0FFEE0DD F00DDEADBEEF          |96   0   BADC0FFEE0DD F00DDEADBEEF          | N/A
10 112  0   BADC0FFEE0DD F00DDEADBEEF 7072     |112  0   BADC0FFEE0DD F00DDEADBEEF 7072     | N/A
11 112  0   BADC0FFEE0DD F00DDEADBEEF FFFF     |x    x                                        | N/A
```

(c) P4Testgen tests for program 1a and 1b. "In" and "out" denote the in- and output port. "Size" is the bit-width of the packet.

Figure 1: P4Testgen test case generation examples.

3 P4Testgen in Action

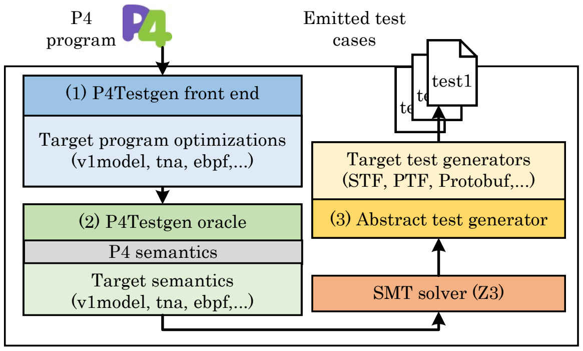
Consider two P4 programs written for a simple BMv2-like target with a single parser and control block in Fig. 1.

Example 1. In the first program (Fig. 1a), Ethernet packets are forwarded based on a table that matches on the EtherType. P4Testgen generates four distinct tests for this program (lines 4–7 in Fig. 1c). In the first, the Ethernet packet is valid but there are no table entries. Since the default action is noop, the output port of the packet does not change. In the second, the configuration has a table entry that executes `set_out` whenever `h.eth.type` matches a given value. P4Testgen generates the corresponding table entry using symbolic execution. Since the program previously set `h.eth.type` to `0xBEEF` the match entry is `0xBEEF` and the output port is chosen at random. In the third, the test is similar, except that P4Testgen chooses the second valid action, noop, which does not alter the output port. In the fourth, the P4Testgen makes use of its *packet sizing* (§ 5.2.1) implementation to generate a packet that is too short and fails the `extract` call. Hence, the target stops parsing and continues to the control, which matches `forward_table` on an uninitialized key. Using *taint tracking*, (§ 5.3) P4Testgen identifies that it can not generate a table entry that is guaranteed to match. So it executes the default action rather than inserting an entry, which would lead to a flaky test.

Example 2. The second program (Fig. 1b) parses an Ethernet header. If it is valid, the program then checks whether the computed checksum (i.e., on `hdr.eth.dst` and `hdr.eth.src`) corresponds to EtherType in the packet (`hdr.eth.type`).[1] If not, `meta.checksum_err` is set to `true` and the packet is dropped. P4Testgen generates three distinct tests for this program (lines 9–11 in Fig. 1c). In the first, the input packet is too short and the Ethernet header is invalid. Hence, `verify_checksum` is not executed and the error is not set, and the packet is forwarded. In the second, P4Testgen generates a packet with a valid Ethernet header and `verify_checksum` computes a checksum using the values `hdr.eth.dst` and `hdr.eth.src`. Here, P4Testgen uses *concolic execution* (§ 5.4) to model the checksum computation and assigns `hdr.eth.type` to the resulting value. Hence, the Ethernet checksum of the packet is correct, and the packet is forwarded. For the final test, P4Testgen generates a test input where `hdr.eth.type` does not match with the computed checksum. Here, `verify_checksum` signals an error, which causes the target to drop the packet in the ingress.

Summary. To the best of our knowledge, neither of these examples would be handled correctly by existing validation frameworks [29, 33, 44, 48], due to the presence of packets with non-standard sizes and complex externs. Nevertheless,

[1]We note that this is a custom, non-standard use of the EtherType header.



P4 program
Emitted test cases
test1
(1) P4Testgen front end
Target program optimizations (v1model, tna, ebpf,...)
(2) P4Testgen oracle
P4 semantics
Target semantics (v1model, tna, ebpf,...)
Target test generators (STF, PTF, Protobuf,...)
(3) Abstract test generator
SMT solver (Z3)

Figure 2: P4Testgen workflow. Numbers are referenced in § 4.

the behaviors exhibited by these tests are possible on the underlying targets, so testing them is important.

4 P4Testgen Overview

P4Testgen generates tests using symbolic execution. It selects a path in the program, encodes the associated path constraint as a first-order formula, and then solves the constraint using an SMT solver. If it finds a solution to the constraint, then it emits a test comprising an input packet, output packet(s), and any control-plane configuration required to execute the path. If it finds no solution, then the path is infeasible. Along with generated tests, P4Testgen records detailed statement coverage information for the P4 program, using heuristics to try to maximize coverage with the fewest number of paths.

Test generation. Fig. 2 shows P4Testgen's three-phase workflow:

1. Translate the input program and target into a symbolically executable representation. P4Testgen takes as input a P4 program, the identifier of the target, and the desired test framework. It parses the P4 program and converts it into the P4C intermediate representation (IR) language. P4Testgen then applies a series of compiler optimizations to transform the P4 IR into a simplified form that streamlines symbolic execution. For instance, P4Testgen unrolls parser loops up to a bound and replaces run-time indices for header stacks with conditionals and constant indices. We assume these optimizations are free of bugs that lead to semantically incorrect output; tools like Gauntlet [39] or Petr4 [13] can be used to verify this assumption.

2. Generate the test case specification. After the input has been parsed and transformed, P4Testgen symbolically executes the program by stepping through the individual nodes (parser states, tables, statements). By default, P4Testgen provides a reference implementation for each P4 construct, but each step can be customized to reflect target-specific semantics by overriding methods in the symbolic executor's visitor. Targets must also define how individual P4 blocks are chained together (i.e., the order in which a packet traverses the P4 blocks), what kind of parsable data can be appended or prepended to packets (e.g., frame check sequences), and how target system data (also known as intrinsic metadata) is initialized.

```
ArchitectureSpec("V1Switch", {
  // parser Parser<H, M>(packet_in b,
  //                     out H parsedHdr,
  //                     inout M meta,
  //                     inout standard_metadata_t sm);
  {"Parser", {none, "*hdr", "*meta", "*sm"}},
  // control VerifyChecksum<H, M>(inout H hdr,
  //                              inout M meta);
  {"VerifyChecksum", {"*hdr", "*meta"}},
  // control Ingress<H, M>(inout H hdr,
  //                       inout M meta,
  //                       inout standard_metadata_t sm);
  {"Ingress", {"*hdr", "*meta", "*sm"}},
  // control Egress<H, M>(inout H hdr,
  //            inout M meta,
  //            inout standard_metadata_t sm);
  {"Egress", {"*hdr", "*meta", "*sm"}},
  // control ComputeChecksum<H, M>(inout H hdr,
  //                       inout M meta);
  {"ComputeChecksum", {"*hdr", "*meta"}},
  // control Deparser<H>(packet_out b, in H hdr);
  {"Deparser", {none, "*hdr"}}});
```

Figure 3: The pipeline state for the `v1model` architecture. Comments describe the associated P4 block. The word none indicates parameters irrelevant to the state.

3. Emitting the test case. Once P4Testgen has executed a path, it emits an abstract test specification, which describes the expected system state (e.g., registers and counters) and output packet(s) for the given packet input and control-plane configuration. Different frameworks can concretize this abstract test specification for execution (e.g., STF [6] or PTF [2]).

5 Whole-Program Semantics

P4 symbolic execution (§ 4) requires a semantic representation of the entire program. Whole-program semantics is P4Testgen's solution to this problem and addresses § 2.3's challenges. It uses 4 techniques: pipeline templates, target extensions and packet sizing, taint analysis, and concolic execution.

5.1 The Pipeline Template

The P4 language does not provide any information about the behavior of the target architecture (e.g., the order of execution of P4 programmable blocks) (Challenge 1). Hence, P4Testgen must provide a mechanism to describe the target-specific data and control flow. In P4Testgen, each target extension must define a pipeline template. A pipeline template has two components: its *state* and its *control flow*. The pipeline state maps variables in the P4 program to per-packet data maintained

```
control Ingress(...) {
    if (hdr.ip.ttl == 0) {
        m.drop_ctl = 1; // Drop packet
    }
    if (hdr.ip.ttl == 1) {
        resubmit.emit(m); // Resubmit packet
    }
}
Pipeline(I_Parser(), Ingress(), I_Deparser(),
     E_Parser(), Egress(), E_Deparser()) pipe;
```

Figure 4: P4 program snippet that sets metadata state.

by the target—e.g., which parameters of a parser correspond to the metadata on the target. The pipeline control flow describes how this per-packet data is manipulated throughout the program and how it influences the execution of the program.

5.1.1 Pipeline State

Pipeline state describes the per-packet data that is transferred between P4-programmable blocks. Fig. 3 shows the pipeline state description for the `v1model` in a simple C++ DSL. The objects listed in the data structure are mapped onto the programmable blocks in the top-level declaration of a P4 program. The declaration order of these objects determines the order in which the blocks are executed by default, but this can be overridden by the pipeline control flow based on a packet's per-packet data values. Arguments with the same name are threaded through the control blocks in execution order. For example, the `*hdr` parameter in the parser is first reset, as it is used in an `out` position. After executing the parser, it is copied into the checksum unit, then to the ingress control, etc.

5.1.2 Pipeline Control Flow

Modelling the pipeline control flow. P4Testgen allows extension developers to provide code to models arbitrary interpretation of the pipeline state. Figs. 4-5 shows an example of a P4 program snippet being interpreted in the context of P4Testgen's pipeline control flow. The target is a fictitious target with an implicit traffic manager between ingress and egress pipelines. The green dashed segments in the figure are target-defined and interpret the variables set in the `Ingress` control. If `m.drop_ctl` is set, the packet will be dropped by the traffic manager, skipping execution of the entire egress. If the `resubmit.emit()` is called, `m.recirculate` will implicitly be set, causing P4Testgen to reset all metadata and reroute the execution back to the ingress parser. We have modeled this control flow for targets such as `v1model`, `tna`, and `t2na`; `ebpf_model` does not support recirculation.

Continuations. To model how per-packet-data (e.g., headers, metadata) flows between (and is transformed by) the programmable blocks of a target, P4Testgen uses *continuations*, which are a general way to represent control flow in a program [37]. An advantage of using continuations is that they can represent arbitrary control flow, including recirculation, in a simple way. In particular, a continuation can expand, shorten, or alter the subsequent execution stack. In P4, they model the traversal of a packet through an arbitrary series of pipelines. Another benefit of continuations is to preserve execution contexts across paths in the program. Preserving contexts enables trying different heuristics for path exploration, which optimize for different goals.

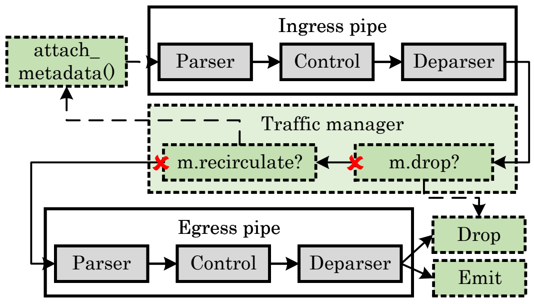


Figure 5: Target control flow as modeled by P4Testgen. Green, dashed segments are target-defined. ✗ denotes `false`.

5.2 Accommodating Target-Specific Behavior

Since some targets diverge in their interpretation of core P4 language constructs, P4Testgen is structured such that every continuation function can be overridden by target extensions. For example, the `v1model` P4Testgen extension overrides the canonical P4Testgen table continuation to implement its own annotation semantics (e.g., the "priority" annotation, which reorders the execution of constant table entries based on the value of the annotation). Targets may also reinterpret the core P4 packet parsing functions (`extract`, `advance`, `lookahead`).

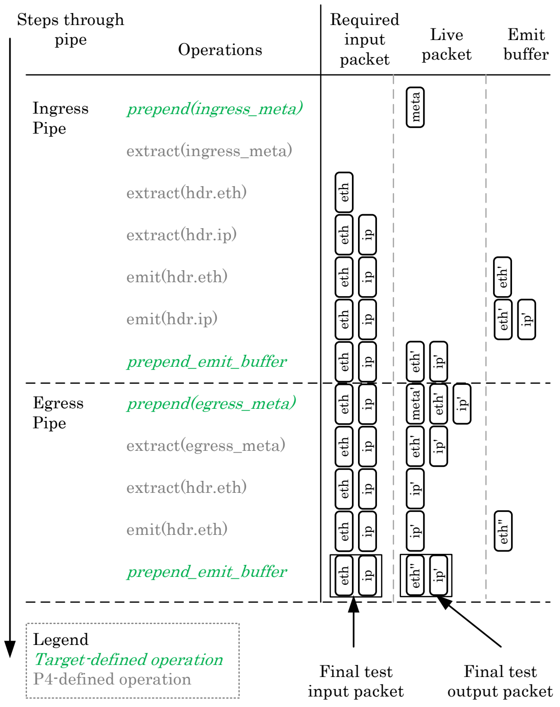
5.2.1 P4Testgen's Approach to Packet-Sizing

To address the variety of interpretations of parsing functions and the traffic manager (Challenge 2) we have also developed a custom model for packets that supports dynamic packet resizing. This turns out to be non-trivial due to the need to encode our model as a first-order logic formula for input to an SMT solver. Recall that P4 externs such as `extract` can throw exceptions when the packet is too short or malformed. While these events are currently sparsely tested when developing a new P4 target and toolchain, P4Testgen generates tests that trigger such exceptions. Particularly on hardware targets, short packets may not be parsed as expected.

P4Testgen makes the packet size a symbolic variable in the path constraint. However, making the packet size part of the

```
parser IngressParser(...)
    pkt.extract(ingress_meta);
    pkt.extract(hdr.eth);
    pkt.extract(hdr.ipv4);
control IngressDeparser(...)
    pkt.emit(hdr.eth);
    pkt.emit(hdr.ipv4);
parser EgressParser(...)
    pkt.extract(egress_meta);
    pkt.extract(hdr.eth);
control EgressDeparser(...)
    pkt.emit(hdr.eth);
```

(a) Extern sequence manipulating Ethernet and IPv4 headers.



(b) Change in the packet sizing variables as P4Testgen steps through the program. Each block corresponds to a P4 header.

Figure 6: Depiction of packet sizing for a Tofino program.

path constraints of P4Testgen is not straightforward. First, as the packet size variable is symbolic, the required packet size to traverse a particular path is only known after the SMT solver is invoked. Second, externs in P4 manipulate the size of the packets (e.g., `extract` calls shorten while `emit` calls lengthen the packet), which requires careful bookkeeping. Third, various targets both react to specific packet sizes (e.g., BMv2 produces garbage values for 0-length packets [38], whereas Tofino drops packets smaller than 64 bytes). Fourth, some targets add and remove content from the packet (e.g., Tofino adds internal metadata to the packet). Any packet-sizing mechanism needs to handle these challenges, while remaining target independent.

Solution to packet sizing. Our approach to model packet sizes is based on a custom model for parsers. For each program path, we calculate the *minimum* header size required to successfully exercise the path without triggering an error in the parser. Our packet-sizing model can be defined in terms of three variables: the required input packet (I), the live packet (L), and the emit buffer (E). The input packet I represents the *minimum* header size required to traverse a particular program path. It also denotes the length of the final input packet in the generated test. The live packet L represents the current packet header that can be manipulated by the P4 program. L also corresponds to the length of the expected packet output in the test. The emit buffer E accumulates the headers generated by calls to `emit`, preserving their order.

Fig. 6 demonstrates how the variables are built while traversing an example pipeline. Initially, all variables are zero-width bit vectors. Targets can add and remove content from the variables by concatenating or slicing the variable bit vectors. For example, targets may prepend parseable metadata to the input packet. In P4Testgen, this metadata will be added to L. While traversing the program, content is sliced from the live packet by extern calls in the P4 program. If L is empty (meaning we have run out of packet content), P4Testgen allocates a new packet variable and appends it to I. This implicitly records a requirement that a larger packet is needed to pass a particular parser extern call.

In various P4 targets, headers have to be explicitly emitted in the deparser stage using an `emit`. With every successful `emit` call, P4Testgen adds the header to E. Once the P4 program traverses a trigger point (usually after exiting a deparser), P4Testgen prepends E to L.

This design becomes important in multi-parser, multi-pipe targets, such as Tofino. Each Tofino pipeline has two parsers: ingress and egress. The egress parser receives the packet (L) after the ingress and traffic manager. If the egress parser runs out of content in L, P4Testgen will again need to append symbolic content to I, increasing the size of the minimum required input packet to successfully traverse the egress parser.

5.3 Controlling Unpredictable Behavior

To avoid unpredictable behavior in the P4 program (Challenge 3), we use taint analysis [40]. Taint is defined as bits which can either be 0 or 1 during program execution. Taint analysis defines how this nondeterminism propagates between sources that produce taint and sinks that consume this taint. In our case, we track how taint propagates when P4Testgen steps through the P4 program. For example, a declaration of a variable that is not initialized will be designated as a source of taint, and any operation that references a tainted variable is in turn tainted.

We use taint to guide several choices in P4Testgen. For example, if the output port is tainted in the test-generation phase, P4Testgen cannot predict its value and therefore drops the test.[2] On the other hand, when the output packet contains taint, we know that certain bits are unreliable. We can make use of test-framework-specific facilities (e.g., "don't care" masks) to ignore tainted output bits in the generated test.

Mitigating taint spread. A common problem with taint analysis is *taint spread*, the proliferation of taint throughout the entire program, quickly tainting all operations. Taint spread makes test case generation almost useless, as the generated tests will come with many "don't care" wild cards. To mitigate taint spread we make use of a few heuristics. (1) We apply optimizations to eliminate any unnecessary tainting (for example, multiplying a tainted value with `0` results in `0`). (2) We exercise freedom in the P4 specification to avoid taint. For example, when a ternary table key is tainted, we can insert a wildcard entry that always matches and remove the non-determinism from the match. (3) We exploit target-specific determinism. For example, Tofino has an `auto_init_metadata` annotation that initializes all target metadata with 0. Applying these heuristics significantly reduces the amount of taint spread in the program.

Rapid extern prototyping. A positive outcome of using taint is the ability to quickly prototype target externs. In general, implementing a new extern is complex. However, a developer instantiating P4Testgen for a given target can use taint variables to quickly prototype parts that may need time-intensive development but are less critical for test-case generation. We used this approach to generate initial stubs for many externs (e.g., checksums, meters) before implementing them precisely.

5.4 Supporting Complex Functions

To handle complex functions that cannot be easily encoded into first-order logic (Challenge 4), P4Testgen uses *concolic execution* [25, 41]. Concolic execution is an advanced technique that combines symbolic and concrete execution. At a high level, concolic execution leaves difficult-to-model functions initially unconstrained, and adds constraints later using a concrete implementation of the function.

The `verify_checksum` function described in § 3 is an example where concolic execution is necessary. The checksum computation is too complex to be expressed in first-order logic. Instead, we model the return value of the extern function as an uninterpreted function dependent on the input arguments of the extern. This uninterpreted function is propagated throughout the program as a placeholder variable while P4Testgen continues executing. If this function becomes part of a path constraint, the SMT solver is free to fill it in with any function that satisfies the rest of the path constraint.

Once we have generated a full path, we need to assign a concrete value to the result of the uninterpreted function.

[2] No available test framework is able to check whether the output port is one of many possible values. Addressing this is future work.

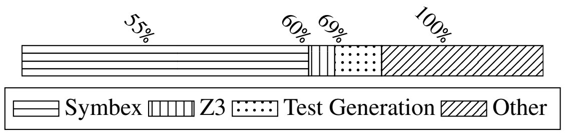


Figure 7: Average CPU time spent in P4Testgen.

First, we invoke the SMT solver to provide us with concrete values to the input arguments of the uninterpreted function that satisfy the path constraints we have collected on the rest of the path. Second, we use these input arguments as inputs to the actual extern implementation (e.g., the hash function executed by the target). Third, we add equations to the path constraints that bind all the values we have calculated to the appropriate input arguments and output of the function. We then invoke the solver a second time to assess whether the result computed by the concrete function satisfies all of the other constraints in the path. If so, we are done and can generate a test with all the values we calculated.

Handling unsatisfiable concolic assignments. In some cases, the newly generated constraints cannot be satisfied using the inputs chosen by the SMT solver. In practice, retrying by generating new inputs may not lead to a satisfiable outcome. Before discarding this path entirely, we try to apply domain-specific optimizations to produce better constraints for the concolic calculation. For example, the `verify_checksum` function (see also §3) tries to match the computed checksum of input data with an input reference value. If the computed checksum does not match with the reference value, `verify_checksum` reports a checksum mismatch. Instead of retrying to find a potential match, we add a new path that forces the reference value to be equal to the computed checksum. This path is satisfiable if the reference value is derived from symbolic inputs, which is often the case. Note that in situations where the reference value is a constant, we are unable to apply this optimization.

6 Implementation

P4Testgen is written as an extension to P4C in about 20k lines of C++ code. For every program path, P4Testgen maintains an independent execution state object that tracks the state of this particular path. This includes the symbolic environment and variable values, collected path constraints, execution traces, test properties, and target state necessary to traverse that path. To compute path constraints, P4Testgen uses Z3 [12] configured with incremental solving as its SMT solver. Z3 has not been a performance bottleneck. Fig. 7 shows P4Testgen's CPU time distribution for generating 10000 tests for the larger programs listed in Tbl. 4a. Solving path constraints in Z3 accounts for less than 10% of the overall CPU time spent.

Path traversal. By default, P4Testgen uses depth-first search

Architecture	Target	Test back end
`v1model`	BMv2	STF, PTF, Protobuf
`tna`	Tofino 1	Internal, PTF
`t2na`	Tofino 2	Internal, PTF
`ebpf_model`	Linux Kernel	STF

Table 1: P4Testgen extensions.

(DFS) to explore paths. It does not prioritize any path and it explores all valid paths to exhaustion. To reduce the number of paths, P4Testgen prunes unsatisfiable paths and exploits fixed, target-specific preconditions, which restrict the available initial packets. Such target-specific preconditions include a minimum packet size or initializing metadata to zero.

Interacting with the control plane. P4Testgen uses the control plane to trigger some paths in a P4 program (e.g., paths dependent on parser value sets [9, §12.11], tables, or register values). Since P4Testgen does not perform load or timing tests, the interaction with the control plane is straightforward. For each path that requires control-plane configuration, P4Testgen creates an abstract test object, which becomes part of the final test case specification. For tables, P4Testgen creates a single control-plane entry for each table of the P4 program to trigger a single match-action pair. If the test framework provides an API, P4Testgen can also initialize externs such as registers, meters, counters with the appropriate value and validate their state after test execution. In general, the richer the API of the test framework, the more P4Testgen can exercise the control plane of the target—e.g., STF has significantly fewer configuration options than PTF. Among others, BMv2 STF does not yet support adding range entries to tables. This restriction means that in some cases P4Testgen will cover fewer paths than is otherwise possible.

6.1 P4Testgen Extensions

Tbl. 1 lists the targets we have instantiated with P4Testgen including the `v1model` architecture for BMv2, the `ebpf_model` for the eBPF kernel module, and the `tna` and `t2na` architecture for the Tofino 1 and 2 chips. We modeled the majority of the Tofino externs based on the P4 Tofino Native Architecture (TNA) available in the Open-Tofino repository. [27]. Each extension also contains support for several test frameworks. The `v1model` extension supports PTF, STF, and custom Protobuf [30] messages. For the Tofino extensions we were given access to an internal compiler testing framework and a variant of PTF. The eBPF extension uses STF.

6.1.1 v1model

P4Testgen supports the `v1model` architecture, including externs such as `recirculate`, `verify_checksum`, and `clone`. The `clone` extern requires P4Testgen's entire toolbox to model its behavior, so we explain it in detail below.

Implementing clone. The `clone` extern duplicates the input packet and submits the cloned packet into the egress block of the `v1model` target. It alters subsequent control flow based on the place of execution (ingress or egress control block). Depending on whether `clone` was called in the ingress or egress control block, the content of the recirculated packet will differ. Moreover, which user metadata is preserved in the target depends on input arguments to `clone` extern.

We modeled this behavior entirely within the BMv2 extension to P4Testgen without having to modify the core code of P4Testgen's symbolic executor. We use the pipeline control flow and continuations to describe `clone`'s semantics, concolic execution to compute the appropriate clone session IDs, and taint tracking to guard against unpredictable input arguments.

`P4-constraints`. P4Testgen's BMv2 extension also implements the `P4-constraints` framework [1] for `v1model`. `P4-constraints` annotates tables to describe which control plane entries are valid for this table. `P4-constraints` are needed for programs such as `middleblock.p4` [23], which model fixed-function data center spine switches. To be able to generate valid tests for such programs, P4Testgen must implement `P4-constraints`. P4Testgen does so by converting `P4-constraints` annotations into its own internal predicates. These predicates are then applied as preconditions at the beginning of program execution, which restricts the entries that can be generated by P4Testgen. This also reduces the overall number of tests generated, as discussed in §7.

6.1.2 tna/t2na

We have implemented the majority of externs for `tna` and `t2na`, including registers, checksums, and hashes. For other externs, such as meters, we make use of rapid prototyping using taints to support test-case generation. Our `t2na` extension leverages much of the `tna` extension, but `t2na` is much richer, so it took additional effort to model its extra capabilities. In particular, not only does `t2na` attach different metadata, it also adds a new programmable block ("ghost") and doubles the number of available extern functions. Both `tna` and `t2na` are architectures for the Tofino chip family that process packets at line-rate, which makes their packet-processing pipeline more complex and nuanced than BMv2. Packet parsing in particular has significantly different semantics [27, §5].

Parsing packets with Tofino Tofino prepends multiple bytes of metadata to the packet [27, §5.1]. As an Ethernet device, it also computes and appends a 32-bit frame check sequence for each packet. Both the metadata and frame check sequence can be extracted by the parser but are not part of the egress packet in the emit stage. If the packet is too short, Tofino drops the packet in the ingress parser, but not in the egress

Bug Type	BMv2	Tofino	Total
Exception	8	9	**17**
Wrong Code	1	7	**8**
Total	**9**	**16**	**25**

Table 2: Bugs in targets discovered by P4Testgen.

parser [27, §5.2.1]. However, if the ingress control reads from the `parser_error` metadata variable, the packet is not dropped, but instead skips the remaining parser execution and advances to the ingress control. The content of the header that triggered the exception is unspecified in this case. We model this behavior entirely in the Tofino instantiations of P4Testgen. We treat the metadata, padded content, and frame check sequence as tainted variables which are prepended to the live packet *L*. Since Tofino's parsing behaves slightly different as defined in the P4 language specification, we extend the implementations of `advance`, `extract`, and `lookahead` in the Tofino extensions to model its observed behavior.

6.1.3 ebpf_model

Both `v1model` and `tna/t2na` are switch-based. To understand P4Testgen's extensibility capabilities we also implemented a proof-of-concept extension for an end-host P4 target, the `ebpf_model.p4`. `ebpf_model.p4` is a fairly simple target. It only has a parser and a filter control. The filter control is applied after the parser and there is no deparser. The eBPF kernel target rejects a packet based on the value of the `accept` parameter in the filter block. If `false`, the packet is dropped. Since there is no deparser, we have to model implicit deparsing logic. We do so by implementing a helper function that iterates over all headers in the packet header structure and emits headers based on their validity. We implemented the eBPF target in a few hours as a proof of concept and executed all the available tests (30) in the P4C repository. Because of the lack of maturity of the target, we did not track any bugs in the eBPF toolchain. The experience of implementing the eBPF extension was encouraging and we are looking into extending our implementation to the XDP [46] back end.

7 Evaluation

Does P4Testgen produce correct tests? We want to ensure that P4Testgen's interpretation of the P4 and target semantics are correct. As an oracle, P4Testgen produces input–output tests. If these tests fail because of nondeterminism or mistakes in P4Testgen's semantics, they are not useful.

Hence we relied on several sources of truth to guide our development of P4Testgen's semantics. For the P4 semantics, we developed our tool according to the P4 specification, also enforced by P4C itself. For the target semantics we relied on the software models (BMv2, Tofino model, eBPF kernel).

For each extension we selected a suite of tests and executed them on the corresponding software target. For `v1model` and `ebpf_model`, we selected all the P4 programs available in the P4C test suite. For Tofino, we used the programs available in the P4Studio SDE and a selected set of compiler tests given to us by the Tofino compiler team. The majority of these programs are small and easy to debug, as they are intended to test the Tofino compiler. In total, we tested on 393 Tofino programs, 486 BMv2 programs, and 30 eBPF programs.

We use P4Testgen to generate 10 input–output tests with a fixed seed for each of the above programs. We then execute these tests using the appropriate software model and test back ends. Also, on every commit to the P4Testgen repository, we execute P4Testgen on all 4 extensions and their test back ends (Table 1), totaling over more than ~2000 P4 programs and 10 tests per program. We used this technique to progressively sharpen our semantics over the course of a year, running P4Testgen millions of times. If the execution of a test does not lead to the output expected by P4Testgen, we investigate. Sometimes, the compiler or the software model was at fault and we filed a bug.

Can P4Testgen model large programs? P4Testgen is meant to be extensible, which means it must be able to support multiple targets. We consider our design useful if we are able to generate tests that pass end-to-end execution for a complex, representative program of the particular architecture.

For the `v1model`, we chose as representative programs `middleblock.p4` (§ 6.1.1) and `up4.p4` [31], a P4 program developed by the Open Networking Foundation (ONF) which models the data plane of 5G networks. For `tna/t2na`, we generate tests for the appropriate version of `switch.p4` for either architecture. We execute the generated tests on either BMv2 or the Tofino model (a semantically accurate software model of the Tofino chip). For each target, we generate 100 tests for a selected test framework (STF for `middleblock.p4`/`up4.p4`, PTF for `switch.p4`). The tests we have generated pass, showing that we can generate valid tests for large programs.

What exactly do P4Testgen's tests cover? As discussed in § 2.3, realizing a good system of target coverage is difficult. We developed our own methods to track the coverage of P4Testgen, focusing on statement coverage.

When generating a successful test for a P4 program, P4Testgen tracks the statements (after dead-code elimination) it has covered with that test. Once P4Testgen has finished generating tests, it emits a report that details the total percentage of statements covered and lists the statements not covered. We use this data to identify any P4 program features that were not exercised. For example, some program paths may only be executable if the packet is recirculated, which requires implementation of the corresponding extern in P4Testgen. Tbl. 4a lists coverage we have calculated for

Bug label	Status	Type	Bug description
P4C-1	Open	Exception	The STF test back end is unable to process keys with expressions in their name.
P4C-2	Open	Exception	The compiler did not correctly transform a varbit `extract` call with an expression as second argument.
P4C-3	Open	Exception	The output by the compiler was using an incorrect operation to dereference a header stack.
BMV2-1	Open	Exception	BMv2 crashes when accessing a header stack with an index that is out of bounds.
P4C-4	Open	Exception	Actions, which are missing their "name" annotation, cause the STF test back end to crash.
P4C-5	Fixed	Exception	A second instance where the compiler was using the wrong operation to manipulate header stacks.
P4C-6	Open	Exception	The compiler should have flattened a header union input for `emit` calls.
P4C-7	Fixed	Wrong code	The compiler swallowed the `table.apply()` of a switch case, which led to incorrect output.
P4C-8	Open	Exception	BMv2 can not process structure members with the same name.

Table 3: Details on BMv2 bugs found by P4Testgen. References to public issues have been anonymized.

P4 program	Arch.	Valid tests	Time	Stmt. cov.
`middleblock.p4`	`v1model`	~238k	13h	100%
`up4.p4`	`v1model`	~34k	2h	95%
`switch.p4`	`tna`	>1,000k	N/A	41%
`switch.p4`	`t2na`	>1,000k	N/A	30%

(a) P4Testgen statistics for large P4 programs.

Applied precondition	Valid test paths	Reduction
None	237846	0%
Fixed-size pkt.	178384	25%
`P4-constraints`	135719	43%
`P4-constraints` & fixed-size pkt.	101789	57%

(b) Effects of preconditions on the number of tests generated for `middleblock.p4`. All approaches cover 100% of the program statements. Fixed packet size is 1500 byte.

Table 4: Statistics on target programs.

larger P4 programs. We do not fully cover `up4.p4` because we have not yet added meter configuration to either STF or PTF. We do not cover the case in the P4 program where the meter extern returns a `RED` value, which drops the packet. For the `switch.p4` programs we list the coverage we achieved before ceasing generation at the millionth test.

How many tests does P4Testgen generate for large programs? We also tried to exhaustively generate tests for the chosen programs. Tbl. 4a provides an overview of the number of tests generated for each program (this number correlates with the number of possible branches as modelled by P4Testgen). Unfortunately, for the `switch.p4` programs of `tna` and `t2na`, we generate too many paths to terminate in a reasonable amount of time. We anticipated this because the number of unique paths increases exponentially with the number of parser and table branches. Liu et al. [29] and Stoenescu et al. [44] have observed similar issues.

The large number of paths is also partly caused by P4Testgen's extensibility goal. P4Testgen allows us to create a detailed model of target behavior, which prompted us to model Tofino and BMv2 externs with many potential input–output conditions. This in turn increases the number of generated branches dramatically. Another issue causing the large number of paths is that P4Testgen only requires an input program. By default the tool does not reduce the input space a priori by imposing preconditions as Meissa does.

We conducted a small experiment to measure the impact of applying preconditions and simplified extern semantics on `middleblock.p4`. We measured the number of generated tests when fixing the input packet size (thus avoiding parser rejects in externs) and applying SwitchV's `P4-constraints`. Tbl. 4b shows the results. The number of generated tests can vary widely, based on these input parameters. Applying both fixed-input packets and the `P4-constraints` preconditions can reduce the number of generated tests by as much as 57%.

We have plans to make the number of generated tests tractable. In the future, we plan to add a query mechanism that allows users to only generate tests that cover selected attributes, implement new path exploration strategies, and adopt techniques that further restrict the number of possible inputs as applied in Meissa [48].

Is P4Testgen detailed enough to find bugs? A straightforward method to demonstrate the value of a test-case oracle such as P4Testgen is to track the number of bugs discovered using the tool. If we can find bugs in the targets for which P4Testgen is generating tests, this implies that the tool can model targets with enough detail to identify discrepancies. To find bugs, we used the workflow described in §7 and ran P4Testgen on available sample programs. We did not execute the tests we have generated on a hardware target and we did not track bugs in the eBPF toolchain.

We focus only on bugs in the toolchain that executes the P4 program. We consider a toolchain bug any failing test that was generated by P4Testgen but was not an issue with P4Testgen itself. This includes compiler bugs as well as crashes of the control-plane software, driver, or software simulator. In general, we observe toolchain bugs when executing the target's test framework. Our tool caused two types of bugs: (1) exceptions, where the combination of inputs caused a crash in the software model, test framework, or control plane software; and (2) "wrong code," where the test inputs did not generate

Tool	Generation method	No extra input?	Target agnostic	Target-specific semantics
Gauntlet [39]	Symbex	✓	✓	×
Meissa [48]	Symbex	×	×	✓
SwitchV [1]	Hybrid	×	×	✓
Petr4 [13]	Symbex	×	✓	✓
p4pktgen [33]	Symbex	✓	×	×
PTA [5]	Fuzzing	×	✓	×
DBVal [28]	Fuzzing	×	✓	×
FP4 [47]	Fuzzing	×	✓	×
P4Testgen	Symbex	✓	✓	✓

Table 5: Tools that test the P4 toolchain.

the expected output. Tbl. 2 summarizes the bugs we have found in the two targets. Tbl. 3 provides details on the bugs we have filed for BMv2. For confidentiality reasons, we are unable to discuss the Tofino 1 and 2 bugs in detail.

The causes of these toolchain bugs are diverse. Bugs may either be mistranslations in the compiler back end, an incorrect implementation of the software model, or errors in the control plane software and test framework. Any type of bug was considered significant. Issues we filed were quickly assigned to a responsible engineer.

Overall, we found more issues with Tofino than with BMv2. The explanation for this is two-fold. First, Tofino is a real hardware target with significantly more features (both externs and pipeline capabilities) and a complex tool chain. Hence, issues are more likely to emerge than on BMv2, which is smaller, simpler, and a software target. Second, we focused our bug-tracking efforts on the Tofino targets and treated BMv2 issues with lower priority. We have invested less effort into generating tests for the BMv2 test suite compared to the examined Tofino programs.

For Tofino, several of our issues either anticipated a bug that was later filed by a customer or reproduced an existing bug that was still open (we only include novel bugs in our count). Some programs already had packet tests associated with them, but these tests were not able to catch the bugs we filed. P4Testgen found these bugs because prior tests did not achieve the same coverage.

8 Related Work

Verifying P4 programs.

A number of tools have been recently proposed to help programmers verify that a P4 program satisfies a formal specification. Tools in this domain typically rely on assertions inserted into the program that capture relational properties—e.g., the program does not read or write invalid headers [14, 15, 24, 29, 42–45]. P4Testgen is orthogonal to this line of work. It produces tests for a P4 program but does not check the correctness of the program itself.

Petr4 [13] provides formal semantics for P4 and a reference interpreter. Like P4Testgen, Petr4 also supports an extension model that allows the addition of target-specific semantics. However, it does not support automatic test case generation and does not aim to provide path coverage.

Testing P4 toolchains. Several tools focus on validating P4 implementations. Many of these rely on differential testing, comparing the system under test's output to a second system's output. Tbl. 5 provides a summary. Compared to P4Testgen, these tools are typically tailored to a single target device or use case. Also, because P4Testgen relies on semantics to produce both inputs and outputs, we avoid the overhead of running a second, independent, system to produce the reference output [1].

`p4pktgen` [33] is a symbolic executor that automatically generates tests. It focuses on the `v1model`, STF tests, and BMv2. However, `p4pktgen` is incomplete: it does not implement all aspects of the P4 language and `v1model` architecture—its capabilities as a test oracle are limited.

SwitchV [1] is a project that uses differential testing to find bugs in switch software. It automatically derives input packets from a fixed-function switch specification written in P4, feeds the derived inputs into both a fixed-function switch and a reference software model, then compares the outputs to check for bugs. SwitchV uses fuzzing and symbolic execution to generate inputs that cover a wide range of execution paths. To limit the range of possible inputs, the tool relies on predefined table rules and the `P4-constraints` framework. Like `p4pktgen`, SwitchV is specialized towards the `v1model` and only tests against BMv2.

Meissa [48] is a symbolic executor specialized to the Tofino target, which can generate input–output tests using pre- and post-conditions specified in the LPI [45] language. The tool is designed for scalability and uses techniques such as fixed input table rules, code summary for multi-pipe Tofino programs, and path pruning to eliminate invalid paths according to the input specification. Meissa is not publicly available, precluding a direct comparison with P4Testgen. As mentioned before, P4Testgen could incorporate Meissa's techniques to reduce the number of tests it generates for a large P4 program.

Gauntlet [39] uses a form of model-based testing to generate input–output tests. Gauntlet is target-independent but can only generate tests according to the P4 specification; it does not implement whole-program semantics to describe the semantic model of the target under test.

PTA [5] and DBVal [28] both implement a target-independent test framework designed to uncover bugs in the P4 toolchain. Both PTA and DBVal augment the P4 program under test with extra assertions to validate the correct execution of the pipeline at runtime. Both projects provide only limited support for test-case generation. In general, developers are expected to write their own unit tests.

FP4 [47] is a target-independent fuzzing tool that uses a second switch as a fuzzer to test the implementation of a P4 program. FP4 automatically generates the necessary table rules and input packet headers to hit unique action paths in the program. It tracks the actions that were executed for an input packet by adding an extra header to the program under test. However, to validate whether an output packet is correct, FP4 requires custom annotations in the P4 program. Hence, it does not act as a general test oracle.

9 Conclusion

P4Testgen is a new P4 test-case generation oracle that allows users to produce input–output tests for their own target. P4Testgen provides whole-program semantics for P4, and handles the full behavior of real-world targets including `v1model`, `tna`, `t2na`, and `ebpf_model`. Going forward, using P4Testgen as a base, we would like to develop additional tools for the P4 language. These tools can adapt several ideas from software testing such as random program generation, mutation testing, program verification, and compiler validation and apply these techniques to P4 using domain-specific optimizations. We also believe there is more work to be done on domain-specific notions of test coverage. In particular, statement coverage only tracks if all P4 program statements at the source level have been covered. It doesn't track whether all possible behaviors of the underlying target have been exercised. To track target behavior, it would be helpful to define a notion like extern coverage, which tracks whether all possible functionality of an extern has been exercised. Since P4Testgen is a community-driven effort, we also welcome contributions from the broader community to improve and extend its functionality.

References

[1] Kinan Dak Albab, Jonathan Dilorenzo, Stefan Heule, Ali Kheradmand, Steffen Smolka, Konstantin Weitz, Muhammad Tirmazi, Jiaqi Gao, and Minlan Yu. SwitchV: Automated SDN switch validation with P4 models. In *ACM SIGCOMM*, 2022. 1, 2, 3, 9, 12

[2] Antonin Bas. PTF: Packet testing framework. `https://github.com/p4lang/ptf`. Accessed: 2022-09-20. 2, 5

[3] Antonin Bas. The reference P4 software switch. `https://github.com/p4lang/behavioral-model`. Accessed: 2022-09-20. 1

[4] Scott Bradner and Jim McQuaid. Benchmarking methodology for network interconnect devices (RFC 2544). IETF Request For Comments, 1999. 2

[5] Pietro Bressana, Noa Zilberman, and Robert Soulé. Finding hard-to-find data plane bugs with a PTA. In *ACM CoNEXT*, 2020. 2, 12

[6] Mihai Budiu. The $P4_{16}$ reference compiler implementation architecture. `https://github.com/p4lang/p4c/blob/master/docs/compiler-design.pptx`, 2018. Accessed: 2022-09-20. 2, 5

[7] Mihai Budiu and Chris Dodd. The $P4_{16}$ programming language. *ACM SIGOPS Operating Systems Review*, 2017. 1

[8] The P4.org consortium. The p4runtime specification, version 1.3.0. `https://p4.org/p4-spec/p4runtime/v1.3.0/P4Runtime-Spec.html`, December 2020. 1

[9] The P4.org consortium. The $P4_{16}$ language specification, version 1.2.3. `https://p4.org/p4-spec/docs/P4-16-v-1.2.3.html`, July 2022. 1, 3, 9

[10] Intel Corporation. Industry-first co-packaged optics Ethernet switch. `https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch.html`. Accessed: 2022-09-20. 1, 2

[11] Intel Corporation. The infrastructure processing unit (IPU). `https://www.intel.de/content/www/de/de/products/network-io/smartnic.html`. Accessed: 2022-09-20. 1

[12] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008. 8

[13] Ryan Doenges, Mina Tahmasbi Arashloo, Santiago Bautista, Alexandar Chang, Newton Ni, Samwise Parkinson, Rudy Peterson, Alaia Solko-Breslin, Amanda Xu, and Nate Foster. Petr4: Formal foundations for P4 data planes. In *ACM POPL*, 2021. 2, 5, 12

[14] Dragos Dumitrescu, Radu Stoenescu, Lorina Negreanu, and Costin Raiciu. bf4: Towards bug-free P4 programs. In *ACM SIGCOMM*, 2020. 1, 12

[15] Dragos Dumitrescu, Radu Stoenescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. Dataplane equivalence and its applications. In *USENIX NSDI*, 2019. 1, 12

[16] Andy Fingerhut. Behavioral model targets. `https://github.com/p4lang/behavioral-model/blob/master/targets/README.md`, 2018. Accessed: 2022-09-20. 2

[17] Andy Fingerhut. The bmv2 simple switch target. `https://github.com/p4lang/behavioral-model/blob/main/docs/simple_switch.md`, 2022. Accessed: 2022-09-20. 16

[18] Open Networking Foundation. PINS: P4 integrated network stack. https://opennetworking.org/pins/. Accessed: 2022-09-20. 1

[19] Open Networking Foundation. TDI: Table driven interface. https://github.com/p4lang/tdi. Accessed: 2022-09-20. 1

[20] The Linux Foundation. eBPF: Introduction, tutorials & community resources. https://ebpf.io/. Accessed: 2022-09-20. 2

[21] The Linux Foundation. IPDK: The infrastructure development kit. https://ipdk.io/. Accessed: 2022-09-20. 1

[22] The Linux Foundation. SONIC: Software for open networking in the cloud. https://sonic-net.github.io/SONiC/. Accessed: 2022-09-20. 1

[23] The Linux Foundation. middleblock.p4. https://github.com/sonic-net/sonic-pins/blob/main/sai_p4/instantiations/google/middleblock.p4, 2021. Accessed: 2022-09-20. 9

[24] Lucas Freire, Miguel Neves, Lucas Leal, Kirill Levchenko, Alberto Schaeffer-Filho, and Marinho Barcellos. Uncovering bugs in P4 programs with assertion-based verification. In *ACM SOSR*, 2018. 1, 12

[25] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. In *ACM POPL*, 2005. 2, 8

[26] Enisa Hadzic. Added support for assert and assume primitives in bm_sim. https://github.com/p4lang/behavioral-model/pull/762, 2022. Accessed: 2022-09-20. 16

[27] Intel Corporation. *P4-16 Intel Tofino Native Architecture - Public Version*, 2022. https://github.com/barefootnetworks/open-tofino. Accessed: 2022-09-20. 9, 10, 16

[28] K Shiv Kumar, PS Prashanth, Mina Tahmasbi Arashloo, Venkanna U, and Praveen Tammana. DBVal: Validating p4 data plane runtime behavior. In *ACM SOSR*, 2021. 2, 12

[29] Jed Liu, William Hallahan, Cole Schlesinger, Milad Sharif, Jeongkeun Lee, Robert Soulé, Han Wang, Călin Caşcaval, Nick McKeown, and Nate Foster. p4v: Practical verification for programmable data planes. In *ACM SIGCOMM*, 2018. 1, 4, 11, 12

[30] Google LLC. Protocol buffers. https://developers.google.com/protocol-buffers, 2022. Accessed: 2022-09-20. 9

[31] Robert MacDavid, Carmelo Cascone, Pingping Lin, Badhrinath Padmanabhan, Ajay Thakur, Larry Peterson, Jennifer Rexford, and Oguz Sunay. A P4-based 5G user plane function. In *ACM SOSR*, 2021. 10

[32] Extreme Networks. Extreme 9920: Cloud-native network visibility platform. https://www.extremenetworks.com/product/extreme-9920/. Accessed: 2022-09-20. 2

[33] Andres Nötzli, Jehandad Khan, Andy Fingerhut, Clark Barrett, and Peter Athanas. p4pktgen: Automated test case generation for P4 programs. In *ACM SOSR*, 2018. 1, 2, 4, 12

[34] Orange. psabpf - in-kernel p4 software switch. https://github.com/P4-Research/psabpf. Accessed: 2022-09-20. 1

[35] Ben Pfaff, Debnil Sur, Leonid Ryzhyk, and Mihai Budiu. P4 in open vswitch with ofp4. https://opennetworking.org/wp-content/uploads/2022/05/Ben-Pfaff-Final-Slide-Deck-Tech-Brief-1.pdf, 2021. Accessed: 2022-09-20. 1

[36] Open Compute Project. SAI: Switch abstraction interface. https://www.opencompute.org/projects/sai. Accessed: 2022-09-20. 1

[37] John C. Reynolds. The discoveries of continuations. *LISP and Symbolic Computation*, 1993. 6

[38] Fabian Ruffy. Question about expected output when all headers are invalid. https://github.com/p4lang/behavioral-model/issues/977, 2021. Accessed: 2022-09-20. 7, 16

[39] Fabian Ruffy, Tao Wang, and Anirudh Sivaraman. Gauntlet: Finding bugs in compilers for programmable packet processing. In *USENIX OSDI*, 2020. 2, 5, 12

[40] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *IEEE S&P*, 2010. 7

[41] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A concolic unit testing engine for C. In *ACM ESEC/FSE*, 2005. 2, 8

[42] Apoorv Shukla, Kevin Hudemann, Zsolt Vági, Lily Hügerich, Georgios Smaragdakis, Stefan Schmid, Artur Hecker, and Anja Feldmann. Towards runtime verification of programmable switches. *arXiv preprint arXiv:2004.10887*, 2020. 1, 12

[43] Apoorv Shukla, Kevin Nico Hudemann, Artur Hecker, and Stefan Schmid. Runtime verification of p4 switches

with reinforcement learning. In *Proceedings of the 2019 Workshop on Network Meets AI & ML*, 2019. 12

[44] Radu Stoenescu, Dragos Dumitrescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. Debugging P4 programs with Vera. In *ACM SIGCOMM*, 2018. 1, 4, 11, 12

[45] Bingchuan Tian, Jiaqi Gao, Mengqi Liu, Ennan Zhai, Yanqing Chen, Yu Zhou, Li Dai, Feng Yan, Mengjing Ma, Ming Tang, et al. Aquila: a practically usable verification system for production-scale programmable data planes. In *ACM SIGCOMM*, 2021. 1, 3, 12

[46] William Tu, Fabian Ruffy, and Mihai Budiu. P4C-XDP: Programming the linux kernel forwarding plane using P4. In *Linux Plumbers Conference*, 2018. 10

[47] Nofel Yaseen, Liangcheng Yu, Caleb Stanford, Ryan Beckett, and Vincent Liu. FP4: Line-rate grey-box fuzz testing for p4 switches. *arXiv preprint arXiv:2207.13147*, 2022. 1, 2, 3, 12

[48] Naiqian Zheng, Mengqi Liu, Ennan Zhai, Hongqiang Harry Liu, Yifan Li, Kaicheng Yang, Xuanzhe Liu, and Xin Jin. Meissa: Scalable network testing for programmable data planes. In *ACM SIGCOMM*, 2022. 1, 2, 3, 4, 11, 12

A Appendix

A.1 Target Implementation Details

`tna`/`t2a` target detail
☞ `tna` has ~48 extern functions and 6 programmable blocks [27]. `t2na` has over a 100 externs and 7 programmable blocks.
☞ Tofino 2 adds a programmable block, the ghost thread. This programmable block can insert control plane entries and manipulate program state in parallel to the packet traversing the program.
☞ Packets that are too short are dropped in the Tofino parser, unless Tofino's ingress control reads the parser error variable. Then the packet header causing an exception is in an unspecified state [27, §5.2.1]. Tofino 2 will not execute the `extract` call.
☞ The packet defined in the test is not the packet that the parser will receive. Tofino 1 and 2 prepend 128-256 bits of metadata to the packet [27, §5.1]. The software model also appends an Ethernet frame check sequence that is 32 bits wide. The parser can parse these values into P4 data structures.
☞ If the egress port variable is not written in the P4 program, the packet is automatically considered dropped [27, §5.1].
☞ Output ports in Tofino have different semantics. Some forward the packet to the CPU, some drop the packet, some recirculate the packet, some forward to an output port. The mapping of the ports can be configured [27, §5.7].
☞ Packets must have a minimum size of 64 bytes. Otherwise, the packet will be dropped [27, §7.2]. The exception to this rule are packets injected from the Tofino CPU.
☞ Tofino has many annotations that can affect the semantics of the program. For example, the flexible and padding annotations can reduce/expand the size of the P4 metadata structure, which can affect the size of the output packet [27, §11]. `auto_init_metadata` will initialize all otherwise random metadata to 0 at the beginning of the program.
☞ Tofino has a notation of direct and indirect externs. Direct externs are attached to tables and can update register values in place. Indirect externs are only able to update register values from packet to packet. Reading and writing these externs on the same packet has no effect [27, §7.1].
☞ The Tofino compiler removes all fields that are not read in the P4 program from the egress metadata structure.
☞ Control plane keys in Tofino may contain dollar signs ($). These have to be rewritten because not every control plane framework considers these valid.
☞ Tofino has a metadata variable `bypass_egress` that tells the traffic manager to skip egress processing entirely [27, §5.6].
☞ Tofino has a metadata variable `mtu_truncate`, which truncates the emitted packet to the size as specified by the metadata variable.
`v1model` target detail
☞ `v1model` has ~26 extern functions and 6 programmable blocks [17].
☞ BMv2's default output port is 0 [17]. BMv2 drops packets when the egress port is 511.
☞ Packets that are smaller than 14 bytes in the behavioral model produce a curious sequence of hex output (02000000) [38].
☞ BMv2 supports a special technique to preserve metadata when recirculating a packet. Only the metadata that is annotated with field_list and the correct index is preserved [17].
☞ BMv2 supports the assume/assert externs which can cause BMv2 to terminate abnormally [26].
☞ BMv2's clone extern behaves differently depending on the location it was called in the pipeline. If recirculated in ingress, the cloned packet will have the values after leaving the parser and is directly sent to egress. If cloned in egress, the recirculated packet will have the values after it was emitted by the deparser [17].
☞ BMv2 has an extern that takes the payload into account for checksum calculation. This means you always have to synthesize a payload for this extern [17].
☞ A parser error in BMv2 does not drop the packet. The header that caused the error will be invalid and execution skips to ingress [17].
☞ All uninitialized variables are implicitly initialized to 0 or false in BMv2.
☞ Some `v1model` programs include `P4-constraints`, which limits the types of control plane entries that are allowed for a particular table.
☞ The table implementation in BMv2 supports the priority annotation, which changes the order of evaluation of constant table entries.
`ebpf_model` target detail
☞ `ebpf_model` has 2 extern functions and 2 programmable blocks.
☞ The eBPF target does not have a deparser that uses emit calls. It can only filter.
☞ `extract` or `advance` have no effect on the size of the outgoing packet.
☞ A failing `extract` or `advance` in the eBPF kernel automatically drops the packet.

Table 6: A nonexhaustive collection of target implementation details that require P4Testgen's use of whole-program semantics. Where possible, we cited a source. Some details are not explicitly documented.