P4Testgen: An Extensible Test Oracle For P4

Fabian Ruffy Jed Liu Prathima Kotikalapudi Vojtěch Havel Hanneli Tavante Rob Sherwood[#]
Vladyslav Dubina Volodymyr Peschanenko Anirudh Sivaraman Nate Foster

ABSTRACT

We present P4Testgen, a test oracle for the $P4_{16}$ language. P4Testgen supports automatic test generation for any P4 target and is designed to be extensible to many P4 targets. It models the complete semantics of the target's packet-processing pipeline including the P4 language, architectures and externs, and target-specific extensions. To handle non-deterministic behaviors and complex externs (e.g., checksums and hash functions), P4Testgen uses taint tracking and concolic execution. It also provides path selection strategies that reduce the number of tests required to achieve full coverage.

We have instantiated P4Testgen for the V1model, eBPF, PNA, and Tofino P4 architectures. Each extension required effort commensurate with the complexity of the target. We validated the tests generated by P4Testgen by running them across the entire P4C test suite as well as the programs supplied with the Tofino P4 Studio. Using the tool, we have also confirmed 25 bugs in mature, production toolchains for BMv2 and Tofino.

ACM Reference Format:
Fabian Ruffy, Jed Liu, Prathima Kotikalapudi, Vojtěch Havel, Hanneli Tavante, Rob Sherwood, Vladyslav Dubina, Volodymyr Peschanenko, Anirudh Sivaraman, and Nate Foster. 2023. P4Testgen: An Extensible Test Oracle For P4. In *ACM SIGCOMM 2023 Conference (ACM SIGCOMM '23), September 10–14, 2023, New York, NY, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3603269.3604834

1 INTRODUCTION

We present P4Testgen, an extensible *test oracle* for the $P4_{16}$ [14] language. Given a P4 program and sufficient time, it generates an exhaustive set of tests that cover every reachable statement in the program. Each test consists of an input packet, control-plane configuration, and the expected output packet.

P4Testgen generates tests to validate the *implementation* of a P4 program. Such tests ensure that the device executing the P4 code (commonly referred to as "target") and its toolchain (i.e., the compiler [8], control-plane [13, 25], and various API layers [26, 29, 57]) implement the behaviors specified by the P4 program.

Tests generated by P4Testgen can be used by manufacturers of P4-programmable equipment to validate the toolchains associated with their equipment [12, 15, 17, 43, 50, 51, 55], by P4 compiler writers for debugging optimizations and code transformations [8, 54], and by network owners to check that both fixed-function and programmable targets implement behaviors as specified in P4, including standard and custom protocols [1, 77].

The idea of generating an exhaustive set of tests for a given P4 program is not new. But prior work has largely focused on a specific P4 architecture [14, §4]. For example, `p4pktgen` [52] targets BMv2 [3], Meissa [77] and `p4v` [46] target Tofino [15], and SwitchV [1] targets fixed-function switches. The primary reason why these tools are so specialized is development effort. Building P4 validation tools requires simultaneously understanding (i) the P4 language, (ii) formal methods, and (iii) target-specific behaviors and quirks. Finding developers that satisfy this trifecta even for a single target is already challenging. Finding developers that can design a general tool for all targets is even harder. The unfortunate result is that developer effort has been fragmented across the P4 ecosystem. Most P4 targets today lack adequate test tooling, and advances made with one tool are difficult to port over to other tools.

Our position is that this fragmentation is undesirable and entirely avoidable. While there may be scenarios that warrant the development of target-specific tools, in the common case—i.e., generating input–output pairs for a given program—the desired tests can be derived from the semantics of the P4 language, in a manner that is largely decoupled from the details of the target. Developing a common, open-source platform for validation tools has several benefits. First, common software infrastructure (lexer, parser, type checker, etc.) and an interpreter that realizes the core P4 language semantics can be implemented just once and shared across many tools. Second, because it is open-source, improvements can be contributed back to P4Testgen and benefit the whole community.

P4Testgen combines several techniques in an open-source tool suitable for production use. First, P4Testgen provides an extensible framework for defining the semantics of the whole program ("whole-program semantics"), combining the semantics of the P4 code along with the semantics of the target on which it is executed. A P4 program generally consists of several P4 blocks (with semantics provided by the language specification) that are separated by interstitial architecture-specific elements (with semantics provided by the target). P4Testgen is the first tool that provides an extensible framework for such whole-program semantics, using a carefully designed interpreter based on the open-source P4 compiler (P4C) [8]. Second, while P4Testgen ultimately uses an SMT solver to generate tests, it also handles the "awkward squad" of complex functions that are difficult to model using an SMT solver—e.g., checksums, undefined values, randomness, and so on. To achieve this, P4Testgen uses taint tracking, concolic execution, and a precise model of packet sizing to model the semantics of the program accurately and at bit-level granularity. Third, P4Testgen offers advanced path selection strategies that can efficiently generate tests that achieve full statement coverage, even for large P4 programs that suffer from path explosion. In contrast to prior work, these strategies are fully automated and do not require annotations to use effectively.

To recap, P4Testgen's key technical innovations are as follows:

(1) **Whole-program semantics**: Most P4 targets perform processing that is not defined by the P4 program itself and is target-specific. P4Testgen uses *pipeline templates* to succinctly describe the behavior of an entire pipeline as a composition of P4-programmable blocks and interstitial target-specific elements.

New York University Postman Intel [#]NetDebug.com Litsoft Cornell University

(2) **Target-specific extensions**: Many real-world P4 targets deviate from the $P4_{16}$ specification in ways small and large. To accommodate these deviations, P4Testgen's extensible interpreter supports target-specific *extensions* to override default P4 behavior, including initialization semantics and an intricate model of *packet-sizing*, which accommodates targets that modify packet sizes during processing.
(3) **Taint analysis**: Targets can exhibit non-deterministic behavior, making it impossible to predict test outcomes. To ensure that generated tests are reliable, P4Testgen uses *taint analysis* to track non-deterministic portions of test outputs.
(4) **Concolic execution**: Some targets have features that can not easily be modelled using an SMT solver. P4Testgen uses *concolic execution* [31, 63] to model features such as hash functions and checksums.
(5) **Path selection strategies:** Real-world P4 programs often have a huge number of paths, making full path coverage infeasible. P4Testgen provides heuristic *path selection strategies* that can achieve full statement coverage, usually with orders of magnitude fewer tests than other approaches.

To validate our design for P4Testgen, we instantiated it for 5 different real-world targets and their corresponding P4 architecture: the `v1model` [23] architecture for BMv2, the `ebpf_model` [72] architecture for the Linux kernel [28], the `pna` [33] architecture for the DPDK SoftNIC [24], the `tna` [37] architecture for the Tofino 1 chip [15], and the `t2na` architecture for the Tofino 2 chip [16]. All 5 instantiations implement whole-program semantics without requiring modification to the core parts of P4Testgen. We have tested the correctness of the P4Testgen oracle itself by generating input–output tests for example P4 programs of all listed architectures. Executing P4Testgen's tests using the appropriate target toolchains, we have found 17 bugs in the toolchain of the Tofino compiler and 8 in the toolchain of BMv2. P4Testgen is available at the following URL: https://p4.org/projects/p4testgen.

2 MOTIVATION AND CHALLENGES

P4 offers new capabilities for specifying network behavior, but this flexibility comes at a cost: network owners must now navigate toolchains that are larger and more complex than with fixed-function devices. So, as the P4 ecosystem matures, increased focus is being placed on tools for validating P4 implementations [1, 6, 19, 45, 52, 61, 74, 77], often by exercising input–output tests.

At first glance, the task of generating tests for a given P4 program may seem relatively straightforward. Prior work such as `p4pktgen` [52], `p4v` [46], P4wn [39], Meissa [77], and SwitchV [1] has shown that it is possible to automatically generate tests using techniques from the programming languages and software engineering literature [31, 42, 63]. The precise details vary from tool to tool, but the basic idea is to first use symbolic execution to traverse a path in the program, collecting up a symbolic environment and a path constraint, and then use a first-order theorem prover (i.e., SAT/SMT solver) to compute an executable test. The theorem prover fills in the input–output packet(s) from the symbolic environment to satisfy the path constraint and also computes control-plane configurations required to execute the selected path—e.g., forwarding entries in match-action tables.

Technical Challenges. While prior work has shown the feasibility of automatic test generation using symbolic execution, existing tools have focused on specific targets (e.g., Tofino) and abstracted away important details (e.g., non-standard packets and other "corner cases" in the language), which limits their applicability in practice. In contrast, our goal for P4Testgen is to develop a general and extensible test oracle for P4 that can be readily applied to real-world P4 programs on arbitrary targets. Achieving this goal requires overcoming several technical challenges, described below.

(1) Missing inter-block semantics. A P4 program only specifies the target behavior *within* the P4 programmable blocks in the architecture. It does not specify the execution order of those blocks, or how the output of one block feeds into the input of the next, i.e., target-specific semantics in the interstices between blocks. 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. As another example, the P4 specification states that, if extracting a header fails because the packet is too short, the parser should step into `reject` and exit [14, §12.8.1]. However, the semantics after exiting the `reject` state is left up to the target: some drop the packet, others consider the header uninitialized, while others silently add padding to initialize the header. None of these behaviors are captured by the P4 program itself. P4Testgen offers features for describing such *inter-block semantics* (§4).

(2) Target-specific intra-block semantics. Even though P4 describes the behavior of a programmable block, targets may also have different *intra-block semantics*, i.e., they interpret the P4 code within the programmable block differently. The P4 specification delegates numerous decisions to targets and targets may not implement all parts of the specification. For instance, hardware restrictions can make it difficult to implement parser exceptions faithfully [34]. Match-action table execution can also be customized using target-specific properties (e.g., action profiles) and annotations can influence the semantics of headers and other language constructs in subtle ways—see Tbl. 6 in the appendix for a (non-exhaustive) list of target-specific deviations. As part of its whole-program semantics model, P4Testgen offers a flexible abstract machine based on an extensible class hierarchy, which makes it easy to accommodate target-specific refinements of the P4 specification.

(3) Unpredictable program behavior. Not all parts of a P4 program are well-specified by the code. For instance, reading from an uninitialized variable may return 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. P4Testgen uses *taint-tracking* to keep track of unpredictable bits in the output (§4.4), ensuring that it never produces nondeterministic tests unless explicitly asked to do so.

(4) Complex primitives. Like other automated test generation tools, P4Testgen relies on a first-order theorem prover to compute input–output tests. However, not all primitives can easily be encoded into first-order logic—e.g., checksums and other hash functions, or

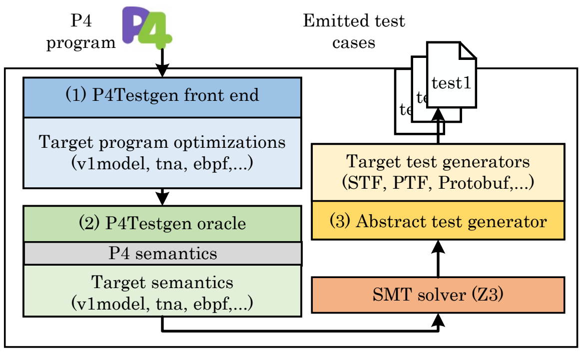


Figure 1: The P4Testgen test case generation process.

programs that modify the size of the packet using dynamic values. For instance, consider a program that uses the `advance` function to increment the parser cursor by an amount that depends on values within the symbolic input header. Modeling this behavior precisely either requires bit vectors of symbolic width, which is not well-supported in theorem provers, or branching on every possible value, which is impractical. P4Testgen uses *concolic execution* to accommodate computations which cannot be encoded into first-order logic (§4.5).

(5) Path explosion. By default, P4Testgen uses depth-first search (DFS) to select paths throughout the P4 program. It does not prioritize any path and it explores all valid paths to exhaustion. However, real-world P4 programs often have dense parse graphs and large match-action tables, so the number of possible paths grows exponentially [46, 68]. Achieving full path coverage would require generating an excessive number of tests. P4Testgen provides *strategies* for controlling the selection of paths, including random strategies and coverage-guided heuristics that seek to follow paths containing previously unexplored statements. These strategies enable achieving full statement coverage with orders of magnitude fewer tests compared to other approaches (§5).

Outlook. To our knowledge, P4Testgen is the first test generation tool for P4 that meets all of these challenges. Moreover, P4Testgen has been designed to be fully extensible, and it is freely available online under an open-source license, as a part of the P4C compiler framework. We are hopeful that P4Testgen will become a valuable resource for the P4 community, providing the necessary infrastructure to rapidly develop accurate test oracles for a wide range of P4 architectures and targets, and generally reducing the cost of designing, implementing, and validating data planes with P4.

3 P4TESTGEN OVERVIEW

As shown in Fig. 1, 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 the generated tests, P4Testgen reports which segments of the program (statements, externs, actions) are covered by each test. P4Testgen's workflow can be summarized as a three-step process.

Step 1: Translate the input program and target into a symbolically executable representation. P4Testgen takes as input a P4 program, the target architecture, and the desired test framework (e.g., STF [7] or PTF [4]). It parses the P4 program and converts it into the P4C intermediate representation language (P4C-IR). P4Testgen then transform the parsed P4C-IR into a simplified form that makes symbolic execution easier, e.g., P4Testgen unrolls parser loops and replaces run-time indices for header stacks with conditionals and constant indices. The correctness of P4Testgen's tests is predicated on the correctness of the P4C front-end and these transformations.

Step 2. Generate the test case specification. After the input program has been parsed and transformed, P4Testgen symbolically executes the program by stepping through individual AST nodes (parser states, tables, statements). By default, the P4Testgen interpreter provides a reference implementation for each P4 construct. However, each step can be customized to reflect target-specific semantics by overriding methods in the symbolic executor. Targets must also define whole-program semantics (§4) which describe 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 called intrinsic metadata) is initialized. Typically, this target-specific information can be inferred from the documentation for the P4 architecture or the target itself. Detailed knowledge of hardware microarchitecture is not necessary.

Step 3. Emit 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. This abstract test specification is then concretized for execution on different test frameworks (STF, PTF, etc.).

3.1 P4Testgen in Action

As an example to illustrate the use of P4Testgen, consider two P4 programs, as shown in Fig. 2, written for a fictitious, BMv2-like target with a single parser and control block.

Example 1. In the first program (Fig. 2a), Ethernet packets are forwarded based on a table that matches on the EtherType field. There are four different input–output pairs that could be generated. The first pair is a valid Ethernet packet, but no table entries are associated with the input. Since the default action is `noop`, the output port of the packet does not change. The second pair is a configuration with a table entry that executes `set_out` whenever `h.eth.type` matches a given value. Since the program previously set `h.eth.type` to `0xBEEF` the table entry must match on `0xBEEF`. The output port is defined by the control plane. The third pair is similar, except `noop` is chosen as action, which does not alter the output port. For the last input pair the packet is too short and the `extract` call fails. Hence, the target stops parsing and continues to the control. For this particular target the packet will be emitted, but `forward_table` will not execute because the match key is uninitialized. P4Testgen is able to generate four distinct tests for this program. For input–output

```
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();
}
```

```
        Size Port eth.dst      eth.src      eth.type
--- Test 1 ------------------------------------------------
Input:  112  0    000000000000 000000000000 0000
Output: 112  0    000000000000 000000000000 BEEF
--- Test 2 ------------------------------------------------
Input:  112  0    000000000000 000000000000 0000
Output: 112  2    000000000000 000000000000 BEEF
Table Config: match(type=0xBEEF),action(set_out(2))
--- Test 3 ------------------------------------------------
Input:  112  0    000000000000 000000000000 0000
Output: 112  0    000000000000 000000000000 BEEF
Table Config: match(type=0xBEEF),action(noop())
--- Test 4 ------------------------------------------------
Input:  96   0    000000000000 000000000000
Output: 96   0    000000000000 000000000000
```

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

```
1  parser Parser(...) {
2      pkt.extract(hdr.eth);
3      transition accept;
4  }
5  control Verify(...) {
6      meta.checksum_err = verify_checksum(
7      hdr.eth.isValid(),
8      {hdr.eth.dst, hdr.eth.src},
9      hdr.eth.type);
10 }
11 control Ingress(...) {
12     if (meta.checksum_err == 1) {
13         mark_to_drop(); // Drop packet.
14     }
15 }
```

```
        Size Port eth.dst      eth.src      eth.type
--- Test 1 ------------------------------------------------
Input:  112  0    BADC0FFEE0DD F00DDEADBEEF
Output: 112  0    BADC0FFEE0DD F00DDEADBEEF
--- Test 2 ------------------------------------------------
Input:  112  0    BADC0FFEE0DD F00DDEADBEEF FFFF
--- Test 3 ------------------------------------------------
Input:  112  0    BADC0FFEE0DD F00DDEADBEEF 7072
Output: 112  0    BADC0FFEE0DD F00DDEADBEEF 7072
```

(b) P4 program that validates the Ethernet checksum.

Figure 2: P4Testgen test case examples. "Port" denotes the input–output port. "Size" is the packet bit-width.

pairs 2 and 3, P4Testgen synthesizes control plane entries, which execute the appropriate action. For input–output pair 4, P4Testgen makes use of its *packet sizing* (§ 4.3.1) implementation to generate a packet that is too short. P4Testgen uses *taint tracking* (§ 4.4) to identify that `h.eth.type` is uninitialized. Since this target will not match on uninitialized keys, P4Testgen does not generate an entry for `forward_table`.

Example 2. The second program (Fig. 2b) parses an Ethernet header. If it is valid (line 7), the program tests whether the checksum computed on `hdr.eth.dst` and `hdr.eth.src` (lines 6–9) corresponds to the value in field `hdr.eth.type` (line 10).[1] If not, `meta.checksum_err` is set to `true` and the packet is dropped. This program produces three distinct input–output pairs. The first pair is an input packet that is too short, which causes the Ethernet header to be invalid. Hence, `verify_checksum` is not executed, the error is not set, and the packet is forwarded. The second and third input–output pair include a valid Ethernet header. In the second pair, `hdr.eth.type` matches the computed checksum value and the packet is forwarded. In the third pair, the value does not match and the packet is dropped. Note that for input–output pair 2 and 3, P4Testgen uses *concolic execution* (§ 4.5) to model the checksum computation. P4Testgen picks a random concrete assignment to `hdr.eth.dst` and `hdr.eth.src`, computes the checksum, and compares the result to `hdr.eth.type`. As there are no other restrictions on the value of `hdr.eth.dst` and `hdr.eth.src`, P4Testgen produces tests where the checksum either matches (test 3) or does not match (test 2).

Summary. As shown, P4Testgen prefers to maximize program coverage even though it may lead to path explosion. The behaviors exhibited by the tests in Fig. 2 are possible on the underlying targets and testing them is important. Indeed, we have used P4Testgen to uncover a variety of bugs in compilers, drivers, and software models—see §7 for details. Moreover, these bugs were not for toy programs or early versions of systems under development. Rather, they were found in production code for mature systems that had already undergone extensive validation with traditional testing.

4 WHOLE-PROGRAM SEMANTICS

The symbolic execution of P4 programs requires a model of not only the P4 code blocks (parsers, controls, etc.), but also the transformations performed by the rest of the target. However, the P4 language does not specify the behavior of the target architecture (e.g., the order of execution of P4 programmable blocks). P4Testgen addresses this limitation through a *flexible abstract machine* and *pipeline templates*.

4.1 P4Testgen's Abstract Machine

Fig. 3 summarizes the design of the abstract machine that powers P4Testgen's symbolic executor. It has standard elements, such as a stack frame, symbolic environment, and so on, as well as a continuation, which encodes the rest of the computation. A full treatment of continuations [58] is beyond the scope of this paper. In a nutshell, continuations make it easy to encode non-linear control flow such as packet recirculation, which many P4 architectures support, and they also preserve execution contexts across paths, which is helpful for implementing different path selection heuristics.

4.2 The Pipeline Template

Pipeline templates are a succinct mechanism for describing the *pipeline state* and *control flow* for an architecture—and with those two, its inter-block semantics. By default, they capture the common

[1] Note this is a non-standard use of EtherType for the sake of the example.

```
class ExecutionState {
   // Small-step Evaluator: can be overriden by targets
   friend class SmallStepEvaluator;
   // Symbolic Environment: maps values to variables
   SymbolicEnv env;
   // Visited: previously-visited nodes for coverage
    P4::Coverage::CoverageSet visitedNodes;
   // Path Constraint: must be satified to execute this path
   std::vector<const IR::Expression *> pathConstraint;
   // Stack: tracks namespaces, declarations, and scope
   std::stack<const StackFrame &>> stack;
   // Continuation: remainder of the computation
   Continuation::Body body;
   ...
}
```

Figure 3: Execution state for P4Testgen's abstract machine.

```
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"}}});
```

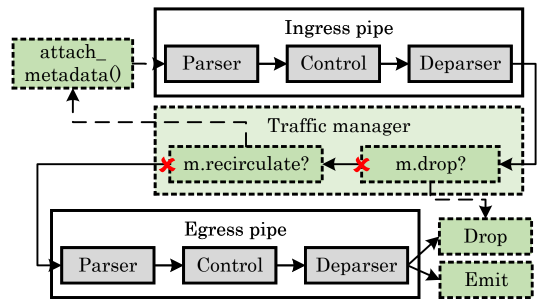
Figure 4: The pipeline state for the `v1model` architecture. Comments describe the associated P4 block. The word `none` indicates parameters irrelevant to the state.

case where the state associated with the packet simply flows between P4-programmable blocks in a straightforward manner—e.g., by copying output variables of one block to the input variables of the next. P4Testgen also handles more complicated forms of packet flow in the architecture, such as recirculation, but this requires writing explicit code against the abstract machine.

4.2.1 Pipeline State. Pipeline state describes the per-packet data that is transferred between P4-programmable blocks. Fig. 4 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 (shown in comments). 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 programmable blocks in execution order. For example, the `*hdr` parameter in the parser is first set undefined, as it is used in an `out` position as seen by the comments in Fig. 4. After executing the parser, it is copied into the checksum unit, then to the ingress control, etc.

```
control Ingress(...) {
    if (hdr.ip.ttl == 0) {
        m.drop = 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;
```

(a) P4 program snippet that sets metadata state.



(b) P4Testgen control-flow. Dashed segments are target-defined. ✗ is `false`

Figure 5: P4Testgen's pipeline control flow.

4.2.2 Pipeline Control Flow. P4Testgen allows extension developers to provide code to model arbitrary interpretation of the pipeline state. Fig. 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` 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`.

4.3 Handling Target-Specific Behavior

Targets have different intra-block semantics and diverge in their interpretation of core P4 language constructs. P4Testgen is structured such that every function in the abstract machine 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 core parsing functions (e.g., `extract`, `advance`, `lookahead`).

4.3.1 P4Testgen's Approach to Packet-Sizing. One area where there is significant diversity among targets is in the semantics of operations that change the size of the packet. Some paths in a P4 program are only executable with a specific packet size. P4 externs such as `extract` can throw exceptions when the packet is too short or malformed. These packet paths are often sparsely tested when developing a new P4 target and toolchain. Particularly on hardware targets, packets with an unexpected size may not be parsed as expected. Correspondingly, P4Testgen must be able to control the size of the input packet (Challenge 2). And, since some of these inputs may trigger parser exceptions, it also needs to model the impact these exceptions have on the content and length of the packet.

P4Testgen implements packet-sizing by making the packet size a symbolic variable in the set of path constraints. This encoding turns out to be non-trivial. Since the required packet size to traverse a given path is now a symbolic variable, it is only known after the SMT solver is invoked. However, at the same time, externs in P4 manipulate the size of the packets (e.g., `extract` calls shorten while `emit` calls lengthen the packet), which requires careful bookkeeping in first-order logic. Targets also react differently to specific packet sizes (e.g., BMv2 produces garbage values for 0-length packets [59], whereas Tofino drops packets smaller than 64 bytes [37, §7.2]). Lastly, some targets add and remove content from the packet (e.g., Tofino adds internal metadata to the packet [37, §5.1]). Any packet-sizing mechanism needs to handle these challenges, while remaining target independent.

Our approach is to model packet-sizing as described in the P4 specification. For each program path, we calculate the *minimum* header size required to successfully exercise the path without triggering a parser exception. The packet-sizing model defines and manipulates three symbolic bit vector variables: the required input packet (I), the live packet (L), and the emit buffer (E). The input packet I represents the *minimum* header content required to reach a particular program point without triggering an exception. The live packet L represents the packet header content available to the interpreter stepping through the P4 program, e.g., `extract` will consume content from L. The emit buffer E is a helper variable which accumulates the headers produced by `emit`. This is necessary to preserve the correct order of headers, as prepending headers to L each time `emit` is executed would cause it to be inverted.

Initially, all variables are zero-width bit vectors. While traversing the program, parser externs (e.g., `extract` or `advance`) in the P4 program slice data from the live packet L. If L is empty (meaning we have run out of packet header data), P4Testgen allocates a new symbolic packet header and adds it to I. Targets may augment the input packet with custom parsable data (e.g., metadata) which reduces the input packet needed to avoid triggering a parser exception. Correspondingly, this content is added to the live packet variable L. Once P4Testgen has finished executing a path, I will denote the content of the final input packet in the generated test. L on the other hand will correspond to the content of the expected packet output. Fig. 9 in App. A.3 illustrates the variables used for an example pipeline.

This design also handles 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 must again append symbolic content to I, increasing the size of the minimum packet required to parse successfully.

4.4 Controlling Unpredictable Behavior

Many P4 programs are non-deterministic, which can lead to unpredictable outputs (Challenge 3). To avoid generating "flaky" tests, we use taint analysis [62]. As P4Testgen steps through the program, we keep track of which bits have a known value (i.e., "untainted"), and which bits have an unknown value (i.e., "tainted"). For example, a declaration of a variable that is not initialized and reads from random memory will be designated as tainted. The result of any operation that references a tainted variable will also be tainted. Later, when generating tests, we use the taint to avoid generating tests that might fail—i.e., due to testing tainted values. For example, if the output packet contains taint, we know that certain bits are unreliable. We use test-framework-specific facilities (e.g., "don't care" masks) to ignore tainted output bits. On the other hand, if the output port is tainted and the test framework does not support wildcards for the output port, P4Testgen can not reliably predict the output, so we drop the test and issue a warning.

Mitigating taint spread. A common issue with taint analysis is *taint spread*, the proliferation of taint throughout the program, quickly tainting most of the state. In extreme situations, taint spread can make test generation almost useless, as the generated tests have many "don't care" wild cards. To mitigate taint spread we use a few heuristics. First, we apply optimizations to eliminate unnecessary tainting (for example, multiplying a tainted value with `0` results in `0`). Second, we exploit freedom in the P4 specification to avoid taint. For example, when a ternary table key is tainted, we insert a wildcard entry that always matches. Third, we model target-specific determinism. For example, the Tofino compiler provides an annotation which initializes all target metadata with 0. Applying these heuristics significantly reduces taint in practice.

Applying taint analysis. In our experience, taint analysis is essential for ensuring that P4Testgen can generate predictable tests. It substantially reduces the signal-to-noise ratio for validation engineers, enabling them focus on analyzing genuine bugs rather than debugging flaky tests. And, although it was not intended for this purpose, P4Testgen's taint analysis can be used to track down undefined behavior in a P4 program. P4Testgen does this by offering a "restricted mode," which triggers an assertion when the interpreter reads from an undefined variable on a particular path. The more "correct" a P4 program is written (i.e., by carefully validating headers) the less taint (and fewer assertions) it produces.

Prototyping extensions using taint. Another useful byproduct of taint analysis is the ability to easily prototype a P4Testgen extension and its externs. Rather than implementing the entire P4Testgen extension at once, a developer can substitute taint variables for the parts that may need time-intensive development (a form of *angelic programming* [5]). By constraining the non-determinism of the unimplemented parts of the extension it is possible to generate deterministic tests early. We used this approach to generate initial stubs for many externs (e.g., checksums, meters, registers) before implementing them precisely.

4.5 Supporting Complex Functions

To handle complex functions that cannot be easily encoded into first-order logic (Challenge 4), P4Testgen uses *concolic execution* [31, 63].

Concolic execution is an advanced technique that combines symbolic and concrete execution. In a nutshell, it leaves hard-to-model functions unconstrained initially, and adds constraints later using the concrete implementation of the function. The `verify_checksum` function described in § 3.1 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 checksum as an uninterpreted function dependent on the input arguments of the extern. While P4Testgen's interpreter steps through the program, this uninterpreted function acts as a placeholder. If the function becomes part of a path constraint, the SMT solver is free to fill it in with any value that satisfies the constraint.

Once we have generated a full path, we need to assign a concrete value to the result of the uninterpreted function. 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 function-specific optimizations to produce better constraints for the concolic calculation. For example, the `verify_checksum` function (see also §3.1) 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.

5 PATH SELECTION STRATEGIES

Methodologies that assess the program coverage of tests have become standard software engineering practice. While path coverage is often infeasible (as the number of paths grows exponentially), statement coverage, also known as line coverage, has been proposed as a good metric for evaluating a test suite [9]. P4Testgen allows users to pick from several different path selection strategies to produce more diverse tests, including Random Backtracking and Coverage-Optimized Search. As the name suggests, Random Backtracking simply jumps back to a random known branch point in the program once P4Testgen has generated a test. Coverage-Optimized Search is similar to the concept with the same name in Klee [9]. After a new test has been generated, it selects the first path from all unexplored paths it has seen so far which will execute P4 statements that have not been covered. If no path with new statements can be found, Coverage-Optimized Search falls back to random backtracking until a path with new statements is discovered. This greedy search covers new statements quickly, but at the cost of higher memory usage (because it accumulates unexplored paths with low potential) and slower per-test-case performance. We measure in §7.3 how these strategies perform on large P4 programs. Our path selection framework is extensible, allowing us to integrate many different selection strategies. We can easily add other success metrics, such as table, action, or parser state coverage.

Targeted test generation with preconditions. Path selection strategies guide test case generation towards a goal, but they do not select for a specific type of test. P4Testgen also gives users the ability to instrument their P4 program with a custom extern (`testgen_assume`). This P4Testgen-intrinsic extern adds a path constraint on variables accessible within the P4 program (e.g., `h.eth_hdr.eth_type == 0x0800`), which forces P4Testgen to only produce tests that satisfy the provided constraint. Assume statements are similar to p4v's assumptions [46], Vera's NetCTL constraint [68], or Aquila's LPI preconditions [71]. We study the effect of these constraints in §7.3.

Instrumenting fixed control-plane configurations. Network operators in general have restricted environments in which only a limited set of packets and control plane configuration is actually valid. Similar to Meissa [77] and SwitchV [1], we are developing techniques to instrument a particular fixed control plane configuration before generating tests. We are looking into a specification method to allows users to only generate tests which comply with their environment assumptions. As an initial step in this direction, P4Testgen implements SwitchV's P4Constraints framework (§6.1.1).

6 IMPLEMENTATION

P4Testgen is written as an extension to P4C using about 15k lines of C++ code, including both P4Testgen core and its extensions. To resolve path constraints, P4Testgen uses the Z3 [18] SMT solver.

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 [14, §12.11], tables, or register values). Since P4Testgen does not perform load or timing tests, the interaction with the control plane is mostly straightforward. For each test that requires control-plane configuration, P4Testgen creates an abstract test object, which becomes part of the final test specification. For tables, P4Testgen creates forwarding entries, and if the test framework provides support, it can also initialize externs such as registers, meters, counters and check their final state after execution. In general, richer test framework APIs give P4Testgen more control over the target—e.g., STF lacks support for range-based match types, which means some paths cannot be executed.

6.1 P4Testgen Extensions

Tbl. 1 lists the targets we have instantiated with P4Testgen. We also list the LoC every extension required, noting that `tna` and `t2na` share a lot of code. Further, `v1model` LoC are inflated because of the P4Constraints parser and lexer implementation specific to the `v1model` extension. We modeled the majority of the Tofino externs based on the P4 Tofino Native Architecture (TNA) available in the Open-Tofino repository [37]. Each extension also contains support

Architecture	Target	Test back end	C/C++ LoC
`v1model`	BMv2	STF, PTF, Protobuf, Meta	5289
`tna`	Tofino 1	Internal, PTF	475 (3314 shared)
`t2na`	Tofino 2	Internal, PTF	478 (3314 shared)
`ebpf_model`	Linux Kernel	STF	1011
`pna`	DPDK SoftNIC	PTF, Meta	1694

Table 1: P4Testgen extensions. The core of P4Testgen is 6679 LoC.

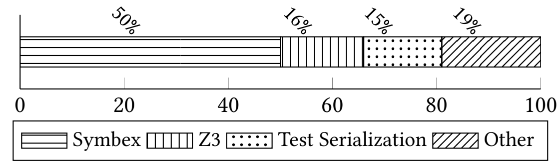


Figure 6: Average CPU time spent in P4Testgen.

for several test frameworks. The `v1model` instance supports PTF, STF, Protobuf [47] messages, and the serialization of metadata state. The Tofino instance supports PTF and an internal compiler testing framework. The eBPF instance supports STF. The Portable NIC Architecture (PNA) [33] instance only has metadata serialization.

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 current 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 vs. egress control). Depending on whether `clone` was called in the ingress vs. egress control, the content of the recirculated packet will differ. Further, which user metadata is preserved in the target depends on input arguments to the `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 inputs.

P4Constraints. P4Testgen's BMv2 extension also implements the P4Constraints framework [1] for `v1model`. P4Constraints annotates tables to describe which control plane entries are valid for this table. P4Constraints are needed for programs such as `middleblock.p4` [27], which models an aggregation switch in Google's Jupiter network [66] that only handles specific entries. To generate valid tests for such programs, P4Testgen must accommodate constraints on entries. It does so by converting P4Constraints annotations into its own internal predicates, which are applied as preconditions, restricting the possible entries, and hence, the number of generated tests (§7).

6.1.2 `tna`/`t2na`*.* We have implemented the majority of externs for `tna` and `t2na`, including meters, checksums, and hashes. For others, such as registers, we make use of rapid prototyping using taint. Our `t2na` extension leverages much of the `tna` extension, but `t2na` is richer, so it took more effort to model its capabilities. Not not only does `t2na` use different metadata, it also adds a new programmable block ("ghost") and doubles the number of externs. Also, both `tna` and `t2na` support parsing packets at line-rate, which is significantly more complex than BMv2 [37, §5].

Parsing packets with Tofino. The Tofino targets prepend multiple bytes of metadata to the packet [37, §5.1]. As an Ethernet device, they also append a 32-bit frame check sequence (FCS) for each packet. Both the metadata and FCS can be extracted by the parser but are not part of the egress packet in the emit stage. If the packet is too short and externs in the parser trigger an exception, Tofino drops the packet in the ingress parser, but not in the egress parser [37, §5.2.1]. However, if the ingress control does read from the `parser_error` metadata variable, the packet is not dropped and 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 FCS as taint variables which are prepended to the live packet L. Since Tofino's parsing behaves differently to the description in the P4 specification, we extend the implementations of `advance`, `extract`, and `lookahead` in the Tofino extensions to model the target-specific behavior.

6.1.3 `ebpf_model`*.* As a proof of concept for P4Testgen's extensibility we also implemented an extension for an end-host target. `ebpf_model` is a fairly simple target, but it differs from `tna` and `t2na`, which are switch-based. The pipeline has a single parser and control. The control is applied as a filter following the parser. 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. As there is no deparser, we model implicit deparsing logic by implementing a helper function that iterates over all headers in the packet header structure and emits headers based on their validity. We were able to implement the eBPF target in a few hours and generate input–output tests for all the available programs (30) in the P4C repository. Because of the lack of maturity of the target, we did not track any bugs in the toolchain.

6.1.4 `pna`*.* PNA [33] is a P4 architecture describing the functionality of end-host networking devices such as (Smart-)NICs. A variety of targets using the `pna` architecture have been put forward by Xilinx [73], Keysight [38], NVIDIA [43], AMD [55], and Intel [17]. We have instantiated a P4Testgen extension for a publicly available `pna` instance, the DPDK SoftNIC [24]. Since there are no functional testing frameworks (e.g., PTF or STF) yet available for this target, we generate abstract test templates, which describe the input–output behavior and expected metadata after each test. By generating these abstract tests we can already perform preliminary analysis on existing `pna` programs (§7.3).

7 EVALUATION

Our evaluation of P4Testgen considers several factors: performance, correctness, coverage, and effectiveness at finding bugs.

7.1 Performance

To evaluate P4Testgen's performance when generating tests, we measured the percentage of cumulative time spent in three major segments: 1) stepping through the symbolic executor, 2) solving Z3 queries, 3) serializing an abstract test into a concrete test. Fig. 6

shows P4Testgen's CPU time distribution for generating 10000 tests for the larger programs listed in Tbl. 2. In general, solving path constraints in Z3 accounts for around 16% of the overall CPU time. P4Testgen spends the majority of time in the symbolic executor. This is expected, as we prioritized extensibility and debuggability for P4Testgen's symbolic execution engine, not performance. We expect performance to improve as the tool matures. From informal conversations we are aware that P4Testgen generates tests on the same order of efficiency as SwitchV's `p4-symbolic` tool does.

7.2 Correctness

As a general test-oracle, P4Testgen is designed to support multiple targets. We consider our design successful if a target extension is both able to generate correct test files for a wide variety of P4 programs and also produce tests that pass for complex, representative programs on each target.

Producing valid tests for diverse P4 programs. To ensure that P4-Testgen's interpretations of P4 and target semantics are correct, we generated tests for a suite of programs and executed them on the target. For `v1model`, `pna`, 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 458 Tofino, 191 Tofino 2, 507 BMv2, 62 PNA, and 30 eBPF programs.

We used P4Testgen to generate 10 input–output tests with a fixed random seed for each of the above programs. We then executed these tests using the appropriate software model and test back ends. In fact, on every repository commit of P4Testgen, we execute P4-Testgen on all 5 extensions and their test back ends (Tbl. 1), totaling more than 2800 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 did not lead to the output expected by P4Testgen, we investigated. Sometimes, it was a bug in P4Testgen, which we fixed. Sometimes, the target was at fault and we filed a bug (see §7.4).

Producing valid tests for large P4 programs. For the `v1model`, we chose two actively maintained P4 models of real-world data planes. `middleblock.p4` (§ 6.1.1) and `up4.p4` [48]. `up4.p4` is a P4 program developed by the Open Networking Foundation (ONF) which models the data plane of 5G networks. We have considered other programs but they were either written in $P4_{14}$ [67] or not sufficiently complex to provide a useful evaluation [11]. For `tna`/`t2na`, we generate tests for the appropriate version of `switch.p4`, the most commonly used P4 program for the Tofino programmable switch ASIC. 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 PTF tests. The eBPF kernel target does not have a suite of representative programs. Instead, we generated tests for P4C's sample programs. The tests we have generated pass, showing that we can correctly generate tests for large programs. `pna` on the DPDK SoftNIC does not have an end-to-end testing pipeline available yet, but we still generate tests for its programs. As a representative program we picked `dash_pipeline.p4`, which models the end-to-end behavior of a programmable data plane in P4 [70]. `dash_pipeline.p4` is still under development, but is already complex enough to generate well over a million unique tests.

P4 program	Valid tests	Time	Stmts.	Stmts. covered
`middleblock.p4 (v1model)`	74472	~40m	150	100%
`up4.p4 (v1model)`	57853	~55m	185	100%
`dash_pipeline.p4 (pna)`	>1M	~668m	256	~90%
`simple_switch.p4 (tna)`	>1M	~628m	300	~43%
`switch.p4 (tna)`	>1M	~2653m	921	~36%
`switch.p4 (t2na)`	>1M	~2719m	1024	~31%

Table 2: Coverage statistics for large P4 programs using DFS.

7.3 Coverage

When generating tests, P4Testgen tracks the statements (after dead-code elimination) covered by each test. Once P4Testgen has finished generating tests, it emits a report that details the total percentage of statements 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.

How well does P4Testgen cover large programs? We tried to exhaustively generate tests for the programs chosen in the previous section. Tbl. 2 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) and the best statement coverage we have achieved using DFS. As expected, for the `switch.p4` programs of `tna` and `t2na`, we generate too many paths to terminate in a reasonable amount of time. For the `switch.p4` programs we list the coverage we achieved before halting generation after the millionth test.

How does path selection help with statement coverage? Tbl. 2 shows that the number of tests generated for larger P4 programs can be overwhelming. In practice, users want tests with specific properties, which necessitates the use of path selection strategies. We measure the effect of the P4Testgen's path selection strategies (§5). We select `middleblock.p4` and `up4.p4` as representative sample programs for `v1model`. For `tna` and `t2na`, we select `simple_switch.p4`, which we patched up such that all statements in the program are reachable. We have chosen `simple_switch.p4` for two reasons: (i) we have not implemented all features to fully cover `switch.p4` (specific register/meter configurations, recirculation) to achieve full statement coverage,[2] and (ii) `simple_switch.p4` is an open-source program available at the OpenTofino repository [37]. `simple_switch.p4` is still a complex Tofino program: it produces over 30 million unique, valid tests. We generate tests with each strategy until we hit 100% statement coverage. We compare Random Backtracking and our Coverage-Optimized Search to standard DFS. We measure the total number of tests needed to achieve coverage across a sample of 10 different seeds.

Fig. 7 shows the mean coverage across 10 seeds over 1000 timesteps for `simple_switch.p4`. We stopped a heuristic if it did not achieve 100% within an hour of generating tests. Only Coverage-Optimized Search reliably accomplishes full coverage in this time frame and outperforms Random Backtracking and DFS by a wide margin. Coverage-Optimized Search always outperforms DFS and generally outperforms Random Backtracking. In some cases, however, (e.g.,

[2] We currently achieve around 90% coverage using Coverage-Optimized Search.

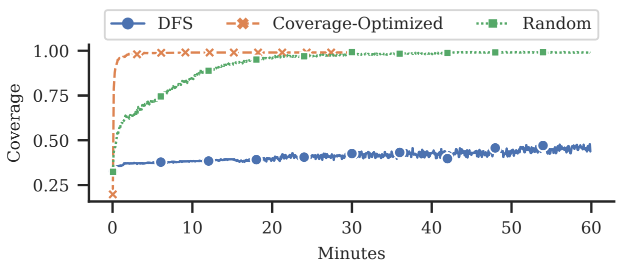


Figure 7: Path selection strategy performance on `simple_switch.p4`.

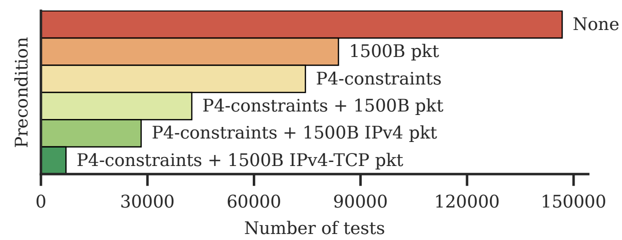


Figure 8: Effects of preconditions on the total number of tests generated for `middleblock.p4`.

up4.p4) Coverage-Optimized Search is not sophisticated enough to find the path which covers a sequence of statements. In those cases, it will perform similarly to Random Backtracking. Tbl. 7 of the appendix shows a results breakdown for all selected programs.

How do preconditions affect the number of generated tests? 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 P4Constraints. Fig. 8 shows the results. The number of generated tests can vary widely, based on these input parameters. Applying the input packet size and the P4Constraints table entry restrictions can reduce the number of generated tests by as much as 71%. Adding `testgen_assume` (§5) statements, which mandates that we only produce packets with TCP/IP headers, reduces the generated tests by 95%. Tbl. 8 in the appendix has detailed statistics.

What are the limits of P4Testgen's statement coverage? There are P4 programs where P4Testgen can not achieve full statement coverage. An example is `blink.p4` [36], a P4 program, where statement execution depends on the timestamp metadata field which is set by the target when a packet is received. Since P4Testgen can not control the initialization of the timestamp for BMv2 (yet), we are unable to cover any statement depending on it. Other tools such as FP4 [74] and P4wn [39] are able to cover these statements as they generate packet sequences which may eventually cause the right timestamp to be generated. This limitation is not insurmountable. In the future, we plan to mock timestamps using a match-action table, or add an API for controlling timestamps directly.

7.4 P4Testgen in Practice

We have used P4Testgen to successfully generate tests for nearly a year. Compiler developers rely on P4Testgen to gain confidence in the implementation of new compiler features. For instance, they can generate tests for an existing program, enable the new compiler feature, and check that the tests still pass. This approach identified several flaws in new compiler targets and features during development. We have also used P4Testgen to give users of Tofino confidence to upgrade their targets or their toolchains. In one of our use cases, a switch vendor had reservations on migrating their P4 programs from Tofino 1 to Tofino 2. The vendor could not ensure that the behavior of the program remained semantically equivalent in this new environment. Using P4Testgen we generated a high-coverage test suite, which reassured the team that they could safely migrate to the Tofino 2 chip.

Generating tests for abstract network device models. An increasingly popular use-case of P4 is to use it as a modeling language to describe network data planes [1, 70]. Often, these data plane models lack tests. P4Testgen can exhaustively generate tests for the P4 data plane model, where the tests also satisfy particular coverage criteria. Further, because P4Testgen is extensible, a developer modelling their device can use arbitrary P4 architectures. We are now working with the DASH [70] and SwitchV [1] developer teams, who are interested in applying P4Testgen to their data plane models written for the `pna` and `v1model` architectures.

7.4.1 *Bugs.*

For any validation tool, the bottom line is whether it effectively finds bugs, particularly in mature, well-tested systems. To evaluate P4Testgen's effectiveness, we used the workflow described in §7.2, by running P4Testgen on each program in the appropriate test suite. Tbl. 4 summarizes the bugs we found. Tbl. 3 provides details on the bugs we have filed for BMv2. For confidentiality reasons, we are unable to provide details on Tofino bugs.

What are the bugs we are interested in? We report only *target stack bugs*—i.e., a bug in the software or hardware stack. We consider a target stack 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. We only count bugs that are both new, distinct (i.e., cause a new entry in the issue tracker), and non-trivial (bugs which require either a particular packet size, control-plane configuration, or extern to be exercised). If a bug is considered a duplicate by the developers we only count it once. P4Testgen revealed two types of bugs: (1) exceptions, where the combination of inputs caused an exception or other fault; and (2) wrong code bugs, where the test inputs did not produce the expected output.

What caused these bugs? The causes of the bugs found were diverse. Some were due to errors in the compiler back end, others due to mistakes in the software model, while still others due to errors in the control plane software and test framework. For each bug, we filed an issue in the respective tracker system. Several issues either anticipated a customer bug that was filed later or reproduced an existing issue that was still open. In several instances, P4Testgen was able to discover bugs where hand-written tests lacked coverage.

What features of P4Testgen were important for finding a bug? 8 of the total 25 we have found were triggered by P4Testgen synthesizing table and extern configurations. 2 were triggered by P4Testgen implementing a detailed model of extern functions. 2 were triggered by P4Testgen generating tests with unexpected packet sizes. The

Bug label	Type	Bug description
p4lang/PI/issues/585	Exception	The open-source P4Runtime server has incomplete support for the `p4runtime_translation` annotation.
p4lang/behavioral-model/issues/1179	Exception	BMv2 crashes when trying to add entries for a shared action selector.
p4lang/p4c/issues/3423	Exception	BMv2 crashes when accessing a header stack with an index that is out of bounds.
p4lang/p4c/issues/3514	Exception	The STF test back end is unable to process keys with expressions in their name.
p4lang/p4c/issues/3429	Exception	The output by the compiler was using an incorrect operation to dereference a header stack.
p4lang/p4c/issues/3435	Exception	Actions, which are missing their "name" annotation, cause the STF test back end to crash.
p4lang/p4c/issues/3620	Exception	BMv2 can not process structure members with the same name.
p4lang/p4c/issues/3490	Wrong code	The compiler swallowed the `table.apply()` of a switch case, which led to incorrect output.

Table 3: BMv2 bugs found by P4Testgen.

Bug Type	Feature	BMv2	Tofino	Total
Exception	Unusual path	2	5	7
	Synthesized control plane	5	1	6
	Packet-sizing	0	2	2
	Extern model	0	1	1
	Total	7	9	17
Wrong Code	Unusual path	0	5	5
	Synthesized control plane	1	1	2
	Packet-sizing	0	0	0
	Extern model	0	1	1
	Total	1	7	8
Total		**8**	**17**	**25**

Table 4: Bugs in targets discovered by P4Testgen.

remaining bugs were caused because P4Testgen's generated tests exercised untested program paths or esoteric language constructs (e.g., a stack-out-of-bounds error or header union access). Overall, we found more incorrect behavior bugs with Tofino because of (i) its complexity and (ii) the fact that we focused our bug-tracking efforts on Tofino and gave BMv2 issues lower priority.

Reachability bugs in P4 programs A side-effect of P4Testgen's support of explicit coverage heuristics is its ability to detect reachability bugs in P4 programs. In some cases, Greedy-Lookahead is unable to cover a particular program statement. This may be because of failures in the heuristic, but often the code is simply non-executable—i.e., dead. We encountered several instances of such dead code for proprietary and public production-grade programs [60]. The developers were usually appreciative of our bug reports, which occurred in complex programs that are difficult to debug, especially early in the development process.

8 RELATED WORK

Automatic test generation for Software-Defined Networks (SDNs). The SDN literature has considerable research dedicated to automated network testing, frequently using symbolic execution to verify the correctness of network invariants [10, 40, 41, 49, 69]. Some of these projects verify network data-plane configurations by generating test input packets, for example Automatic Test Packet Generation (ATPG) [75]. ATPG automates input packet generation to validate a configured switch network by computing all possible packets that cover every switch link and table rule. Monocle [56] and Pronto [76] are similar systems. All use the control-plane configuration as ground truth, which allows them to check whether the right packet headers have been forwarded out on the correct port. P4Testgen targets a richer data plane model than these prior approaches because the data plane is effectively specified in a DSL. But, P4Testgen focuses more narrowly on a single device's data plane implementation, not the entire network's forwarding rules.

Verifying P4 programs. Many tools help verify P4 programs against a formal specification. Tools in this domain usually rely on assertions that model relational properties—e.g., the program does not read or write invalid headers [20, 21, 30, 39, 46, 64, 65, 68, 71]. P4Testgen is orthogonal to these tools. It produces tests for a P4 program but does not check the correctness of the program itself.

Some of these tools [39, 46, 64, 65] are able to generate concrete test inputs in the form of input packets. The outputs of these inputs are then compared against developer-supplied assertions. In theory, with good assertions, this method can also detect bugs in a given P4 toolchain. P6 [64] in particular considers "platform-dependent bugs", which are comparable to toolchain bugs.

Testing P4 toolchains. Other tools focus on validating P4 implementations by generating test inputs. Tbl. 5 provides a summary. Compared to P4Testgen, these tools are typically tailored to a single target or use case. P4Testgen relies on formal semantics to compute inputs and outputs, avoiding running a second system to produce the output [1, 74]. In particular, developers using P4Testgen do not need to understand the semantics of the P4 program to generate tests; P4Testgen provides these semantics as part of its tool.

`p4pktgen` [52] is a symbolic executor that automatically generates tests. It focuses on the `v1model`, STF tests, and BMv2. In spirit, `p4pktgen` is close in functionality to P4Testgen. However, the tool does not implement all aspects of the P4 language and `v1model` architecture—its capabilities as a test oracle are limited. We tried to reproduce the bugs listed in Tbl. 3 using `p4pktgen` but were not able to. `p4pktgen` either was not able to produce tests for the program or did not achieve the necessary coverage. While `p4pktgen` does support a form of packet-sizing to trigger parser exceptions, its model only considers a simple parser-control setup, not multiple subsequent parsers such as Tofino's.

SwitchV [1] uses differential testing to find bugs in switch software. It automatically derives inputs from a switch specification in P4, feeds the derived inputs into both the switch and a software reference model, and compares the outputs. 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 pre-defined table rules and the P4Constraints framework. It also does not generate control-plane entries. Like `p4pktgen`, SwitchV is specialized to `v1model` and BMv2.

Meissa [77] is a symbolic executor specialized to the Tofino target. Meissa builds on the LPI language's pre- and post-conditions [71] to generate input–output tests. The tool is designed for scalability and uses techniques such as fixed match-action table rules, code summaries for multi-pipeline programs, and path pruning to eliminate invalid paths according to the input specification. P4Testgen's

Tool	Input generation method	Synthesizes control-plane?	Multi-target?	Models target semantics?	Data plane coverage metric
Meissa [77]	Symbolic Execution	×	×	✓	Symbolic model
SwitchV (via p4-symbolic) [1]	Symbolic Execution	×	×	✓	Symbolic model, Assertions
p4pktgen [52]	Symbolic Execution	✓	×	×	Symbolic model
Gauntlet (model-based testing) [61]	Symbolic Execution	×	✓	×	Symbolic model
PTA (uses p4v) [6]	Fuzzing	×	✓	×	Symbolic model (p4v)
DBVal [45]	Fuzzing	×	✓	×	Tables, Actions
FP4 [74]	Fuzzing	×	✓	×	Actions
P6 [74]	Fuzzing	×	✓	×	Symbolic model
P4Testgen	Symbolic Execution	✓	✓	✓	Symbolic model, source code

Table 5: P4 tools generating input–output tests. Data plane coverage describes how the tool measures coverage of the generated inputs.

preconditions and path selection strategies combat the same scaling issues as Meissa. Meissa's source code is proprietary, which precludes a direct comparison.

PTA [6] and DBVal [45] 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.

FP4 [74] 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 packets to cover program paths. To validate whether outputs are correct, FP4 requires custom annotations instrumented by the user.

Coverage. There are important differences in testing tools assess coverage—see Tbl. 5 for a summary. P4Testgen marks a node in the source P4 program as covered when the symbolic executor steps through that node and generates a test. FP4 measures action coverage by marking bits in the test packet header to track which actions were executed. As FP4 generates packets at line rate it achieves coverage for actions faster than P4Testgen. p4pktgen discusses branch coverage, which can be estimated by parsing generated tests to see which control-plane constructs (tables, actions) were executed. Meissa reports coverage based on the branches of its own formal model of the P4 program. SwitchV also measures branch coverage based on developer-provided goals derived from its symbolic model. Another important consideration is whether programmers can annotate the program with constraints or preconditions—see Fig 8. In many scenarios, these constraints are necessary to model assumptions made by the overall system, but they also affect coverage since they reduce the number of legal paths.

Extensibility. Petr4 [19] and Gauntlet [61] are designed to support multiple P4 targets. Petr4 provides an "plugin" 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. Gauntlet can generate input–output tests for multiple P4 targets but it does not model externs, nor does it implement whole-program semantics to model the tested target.

9 CONCLUSION

P4Testgen is a new P4 test oracle that automatically generates input–output tests for arbitrary P4 targets. It uses whole-program semantics, taint-tracking, concolic execution, and path selection strategies to model the behavior of the P4 program and generate tests that achieve coverage. P4Testgen is intended to be a resource for the entire P4 community. It already supports input–output test generation for three open-source P4 targets and several extensions for closed-source targets are in development. By designing it as a target-independent, extensible platform, we hope that P4Testgen will be well-positioned for long-term success. Moreover, since P4Testgen is a back end of P4C, it should be easy for developers to build on our tool, lowering the barrier of adoption.

As P4Testgen is an open-source tool, we welcome contributions from the broader community to improve and extend its functionality. For example, two common community requests are to extend P4Testgen with the ability (i) to generate arbitrarily many entries per table and (ii) produce tests with a state-preserving sequence of input–output packets. In the future, to further validate P4Testgen's generality, we would like to complete P4Testgen extensions for the P4-DPDK SoftNIC target and the open-source PSA [32] target for NIKSS [53], as well as proprietary SmartNICs [17, 43, 55]. We also intend to develop additional P4 validation tools based on P4Testgen's framework which apply ideas from software testing in the networking domain—e.g., random program generation, mutation testing, and incremental testing. We are also interested in network-specific coverage notions—e.g., for parsers, tables, actions, etc.

Software testing is always important, but testing the packet processing programs that power our network infrastructure, processing billions of packets per second, is especially important. In time, there will inevitably be better approaches than P4Testgen for generating high-quality tests for packet processing systems. The P4Testgen framework can serve as a vehicle for prototyping these approaches, and for integrating them into the P4 ecosystem. In the future, inspired by efforts from other communities [2, 44], we envision having an open benchmark suite of standard test programs, control plane configurations, and various notions of coverage to standardize comparisons between different testing approaches—enabling more rapid progress for the whole community.

Ethics. Our work on P4Testgen does not raise any ethical issues.

ACKNOWLEDGEMENTS

We wish to thank Boris Beylin, Brad Burres, Călin Caşcaval, Chris Dodd, Glen Gibb, Vladimir Gurevich, Changhoon Kim, Nick McKeown, Chris Sommers, Vladimir Still, and Edwin Verplanke for their support, detailed explanations of the semantics of different P4 targets, and help with analyzing and classifying bugs. We are also grateful to Mihai Budiu, Andy Fingerhut, Dan Lenoski, Jonathan DiLorenzo, Ali Kheradmand, Steffen Smolka, and Hari Thantry for insightful conversations on P4 validation and the broader P4 ecosystem. Finally, we would also like to thank Aurojit Panda, Tao Wang, our shepherd Sanjay Rao, and the participants of the Bellairs Workshop on Network Verification for valuable feedback on this work. This work was support in part by NSF grant CNS-2008048.

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.

[2] Clark Barrett, Leonardo de Moura, and Aaron Stump. SMT-COMP: Satisfiability modulo theories competition. In *Computer Aided Verification (CAV)*, 2005.

[3] Antonin Bas. The reference P4 software switch. https://github.com/p4lang/behavioral-model, 2014. Accessed: 2023-07-25.

[4] Antonin Bas. PTF: Packet testing framework. https://github.com/p4lang/ptf, 2015. Accessed: 2023-07-25.

[5] Rastislav Bodik, Satish Chandra, Joel Galenson, Doug Kimelman, Nicholas Tung, Shaon Barman, and Casey Rodarmor. Programming with angelic nondeterminism. In *ACM POPL*, 2010.

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

[7] Mihai Budiu. The $P4_{16}$ reference compiler implementation architecture. https://github.com/p4lang/p4c/blob/master/docs/compiler-design.pptx, 2018. Accessed: 2023-07-25.

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

[9] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *USENIX OSDI*, 2008.

[10] Marco Canini, Daniele Venzano, Peter Perešíni, Dejan Kostić, and Jennifer Rexford. A NICE way to test openflow applications. In *USENIX NSDI*, 2012.

[11] Xiaoqi Chen. Open source P4 implementations. https://github.com/Princeton-Cabernet/p4-projects, 2018. Accessed: 2023-07-25.

[12] Cisco. Cisco Silicon One. https://www.cisco.com/c/en/us/solutions/service-provider/innovation/silicon-one.html. Accessed: 2023-07-25.

[13] 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.

[14] The P4.org consortium. The $P4_{16}$ language specification, version 1.2.4. https://p4.org/p4-spec/docs/P4-16-v1.2.4.html, May 2023.

[15] 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: 2023-07-25.

[16] Intel Corporation. Second-generation P4-programmable Ethernet switch ASIC that continues to deliver programmability without compromise. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-2-series.html. Accessed: 2023-07-25.

[17] Intel Corporation. The infrastructure processing unit (IPU). https://www.intel.de/content/www/de/de/products/network-io/smartnic.html, 2022. Accessed: 2023-07-25.

[18] 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.

[19] 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.

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

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

[22] Andy Fingerhut. The BMv2 simple switch target. https://github.com/p4lang/behavioral-model/blob/main/docs/simple_switch.md, 2016. Accessed: 2023-07-25.

[23] Andy Fingerhut. Behavioral model targets. https://github.com/p4lang/behavioral-model/blob/master/targets/README.md, 2018. Accessed: 2023-07-25.

[24] Open Networking Foundation. P4 DPDK target components. https://github.com/p4lang/p4-dpdk-target. Accessed: 2023-07-25.

[25] Open Networking Foundation. TDI: Table driven interface. https://github.com/p4lang/tdi. Accessed: 2023-07-25.

[26] Open Networking Foundation. PINS: P4 integrated network stack. https://opennetworking.org/pins/, 2022. Accessed: 2023-07-25.

[27] The Linux Foundation. middleblock.p4. https://github.com/sonic-net/sonic-pins/blob/main/sai_p4/instantiations/google/middleblock.p4, 2021. Accessed: 2023-07-25.

[28] The Linux Foundation. eBPF: Introduction, tutorials & community resources. https://ebpf.io/, 2022. Accessed: 2023-07-25.

[29] The Linux Foundation. SONiC: Software for open networking in the cloud. https://sonic-net.github.io/SONiC/, 2022. Accessed: 2023-07-25.

[30] 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.

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

[32] The P4.org Architecture Working Group. $P4_{16}$ portable switch architecture (psa), version 1.1. https://p4.org/p4-spec/docs/PSA-v1.1.0.html, November 2018.

[33] The P4.org Architecture Working Group. $P4_{16}$ portable nic architecture (pna), version 0.7. https://p4.org/p4-spec/docs/PNA-v0.7.html, December 2022.

[34] Vladimir Gurevich. Change parser exception model and provide better controls for exceptional situation handling. https://github.com/p4lang/p4-spec/issues/880, 2020. Accessed: 2023-07-25.

[35] Enisa Hadzic. Added support for assert and assume primitives in bm_sim. https://github.com/p4lang/behavioral-model/pull/762, 2019. Accessed: 2023-07-25.

[36] Thomas Holterbach, E Costa Molero, Maria Apostolaki, Alberto Dainotti, Stefano Vissicchio, and Laurent Vanbever. Blink: Fast connectivity recovery entirely in the data plane. In *USENIX NSDI*, 2019.

[37] Intel Corporation. *P4-16 Intel Tofino Native Architecture - Public Version*, 2021. https://github.com/barefootnetworks/Open-Tofino/blob/master/PUBLIC_Tofino-Native-Arch.pdf. Accessed: 2023-07-25.

[38] Stefan Johansson. Packet deduplication in p4. https://opennetworking.org/2021-p4-workshop-content/, 2021. Accessed: 2023-07-25.

[39] Qiao Kang, Jiarong Xing, Yiming Qiu, and Ang Chen. Probabilistic profiling of stateful data planes for adversarial testing. In *ACM ASPLOS*, 2021.

[40] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *USENIX NSDI*, 2012.

[41] Ahmed Khurshid, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. Veriflow: Verifying network-wide invariants in real time. In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, 2012.

[42] James C. King. Symbolic execution and program testing. *Commununications of the ACM (CACM)*, 1976.

[43] Ariel Kit. Programming the entire data center infrastructure with the NVIDIA DOCA SDK. https://developer.nvidia.com/blog/programming-the-entire-data-center-infrastructure-with-the-nvidia-doca-sdk/. Accessed: 2023-07-25.

[44] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Conference on Computer and Communications Security (CCS)*, 2018.

[45] K Shiv Kumar, PS Prashanth, Mina Tahmasbi Arashloo, Venkanna U, and Praveen Tammana. DBVal: Validating P4 data plane runtime behavior. In *ACM SOSR*, 2021.

[46] 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.

[47] Google LLC. Protocol buffers. https://protobuf.dev, 2008. Accessed: 2023-07-25.

[48] 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.

[49] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P. Brighten Godfrey, and Samuel Talmadge King. Debugging the data plane with Anteater. In *ACM SIGCOMM*, 2011.

[50] Marvell. Marvell Teralynx 10 data center Ethernet switch. https://www.marvell.com/content/dam/marvell/en/public-collateral/switching/marvell-teralynx-10-data-center-ethernet-switch-product-brief.pdf. Accessed: 2023-07-25.

[51] Extreme Networks. Extreme 9920: Cloud-native network visibility platform. https://www.extremenetworks.com/product/extreme-9920/. Accessed: 2023-07-25.

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

[53] Tomasz Osiński, Jan Palimąka, Mateusz Kossakowski, Frédéric Dang Tran, El-Fadel Bonfoh, and Halina Tarasiuk. A novel programmable software datapath for software-defined networking. In *ACM CoNEXT*, 2022.

[54] Oxide. p4: A P4 compiler. https://github.com/oxidecomputer/p4. Accessed: 2023-07-25.

[55] Pensando. A new way of thinking about next-gen cloud architectures. https://p4.org/p4/pensando-joins-p4.html, 2020. Accessed: 2023-07-25.

[56] Peter Perešíni, Maciej Kuźniar, and Dejan Kostić. Monocle: Dynamic, fine-grained data plane monitoring. In *ACM CoNEXT*, 2015.

[57] Open Compute Project. SAI: Switch abstraction interface. https://www.opencompute.org/projects/sai. Accessed: 2023-07-25.

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

[59] Fabian Ruffy. Question about expected output when all headers are invalid. https://github.com/p4lang/behavioral-model/issues/977, 2021. Accessed: 2023-07-25.

[60] Fabian Ruffy. Dead code in dash_pipeline.p4 pna version. https://github.com/sonic-net/DASH/issues/399, 2023. Accessed: 2023-07-25.

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

[62] 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.

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

[64] Apoorv Shukla, Kevin Hudemann, Zsolt Vági, Lily Hügerich, Georgios Smaragdakis, Artur Hecker, Stefan Schmid, and Anja Feldmann. Fix with P6: Verifying programmable switches at runtime. In *IEEE INFOCOM*, 2021.

[65] 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.

[66] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, et al. Jupiter rising: A decade of clos topologies and centralized control in Google's datacenter network. In *ACM SIGCOMM*, 2015.

[67] Anirudh Sivaraman, Changhoon Kim, Ramkumar Krishnamoorthy, Advait Dixit, and Mihai Budiu. DC.p4: Programming the forwarding plane of a data-center switch. In *ACM SOSR*, 2015.

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

[69] Radu Stoenescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. Symnet: Scalable symbolic execution for modern networks. In *ACM SIGCOMM*, 2016.

[70] Reshma Sudarshan and Chris Sommers. P4 as a single source of truth for sonic dash use cases on both softswitch and hardware. https://opennetworking.org/2022-p4-workshop-gated/, 2022. Accessed: 2023-07-25.

[71] 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.

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

[73] Xilinx. Alveo: The composable SmartNIC. https://www.xilinx.com/applications/data-center/network-acceleration/alveo-sn1000.html, 2023. Accessed: 2023-07-25.

[74] Nofel Yaseen, Liangcheng Yu, Caleb Stanford, Ryan Beckett, and Vincent Liu. FP4: Line-rate greybox fuzz testing for p4 switches. *arXiv preprint arXiv:2207.13147*, 2022.

[75] Hongyi Zeng, Peyman Kazemian, George Varghese, and Nick McKeown. Automatic test packet generation. In *ACM CoNEXT*, 2012.

[76] Yu Zhao, Huazhe Wang, Xin Lin, Tingting Yu, and Chen Qian. Pronto: Efficient test packet generation for dynamic network data planes. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, 2017.

[77] 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.

A APPENDIX

Appendices are supporting material that has not been peer-reviewed.

A.1 Target Implementation Details

`tna`/`t2a` target detail
☞ `tna` has ~48 extern functions and 6 programmable blocks [37]. `t2na` has over a 100 externs and 7 programmable blocks. ☞ Tofino 2 adds a programmable block, the ghost thread. This block can update information related to queue depth in parallel to the packet traversing the program. ☞ In the Tofino parser, if a packet is too short to be read by an extern (extract/advance/lookahead) the packet is dropped, unless Tofino's ingress control reads the parser error variable. Then the packet header causing the exception is in an unspecified state [37, §5.2.1]. ☞ The packet that enters the Tofino parser is augmented with additional information, which needs to be modelled. Tofino 1 and 2 prepend metadata to the packet [37, §5.1]. A 4-bytes Ethernet frame check sequence (FCS) is also appended. The parser can parse these values into P4 data structures. ☞ If the egress port variable is not set in the P4 program, the packet is practically dropped (no unicast copy is made) [37, §5.1]. ☞ The value of the output port in Tofino matters. Some values are associated with the CPU port or recirculation, some are not valid, some forward to an output port. The semantics and validity of the ports can be configured [37, §5.7]. ☞ Tofino follows the Ethernet standard. Packets must have a minimum size of 64 bytes. Otherwise, the packet will be dropped [37, §7.2]. The exception to this rule are packets injected from the Tofino CPU PCIe port. ☞ The Tofino compiler provides annotations which can affect program semantics. Some annotations can alter the size of the P4 metadata structure. If not handled correctly, this can affect the size of the output packet [37, §11]. Another convenience annotation will initialize all otherwise random metadata to 0. ☞ The Tofino compiler removes all fields that are not read in the P4 program from the egress metadata structure. This influences the size of the packet parsed by the egress parser. ☞ Invalid access to header stacks in a parse loop will not cause a `StackOutOfBounds` error. Instead, execution transitions to the control with `PARSER_ERROR_CTR_RANGE` set [37, §5.2.1]. ☞ Control plane keys in the Barefoot Runtime (Bfrt) may contain dollar signs ($). When generating PTF/STF tests, these have to be replaced using a compiler pass. ☞ Tofino has a metadata variable, which tells the traffic manager to skip egress processing entirely [37, §5.6]. ☞ Tofino 2 has a metadata variable, which instructs the deparser to truncate the emitted packet to the specified size.
`v1model` target detail
☞ `v1model` has ~26 extern functions and 6 programmable blocks [22]. ☞ BMv2's default output port is 0 [22]. BMv2 drops packets when the egress port is 511. ☞ When using Linux virtual Ethernet interfaces with BMv2, packets that are smaller than 14 bytes produce a curious sequence of hex output (02000000) [59]. ☞ 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 [22]. ☞ BMv2 supports the assume/assert externs which can cause BMv2 to terminate abnormally [35]. ☞ 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 [22]. ☞ 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 [22]. ☞ A parser error in BMv2 does not drop the packet. The header that caused the error will be invalid and execution skips to ingress [22]. ☞ All uninitialized variables are implicitly initialized to 0 or false in BMv2. ☞ Some `v1model` programs include P4Constraints, 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 to provide an accurate model. Where possible, we cited a source. Some details are not explicitly documented.

A.2 Program Measurements

Program	`middleblock.p4`			`up4.p4`			`simple_switch.p4`			`dash_pipeline.p4`		
Strategy \ Metric (Median)	Tests	Time per test	Total time	Tests	Time per test	Total time	Tests	Time per test	Total time	Tests	Time per test	Total time
DFS	25105	~0.05	1321.4s	12932	~.06s	726.34s	*	~0.07	*	*	~0.06	*
Random Backtracking	956	~.08s	80.92s	2463	~.07s	169.3s	*	~0.09	*	*	~0.13	*
Coverage-Optimized Search	86	~.17s	11.41s	3581	~.06s	242.6s	4612	~0.12	555.24	63	~0.41	21.86s

Table 7: Path selection results for 100% statement coverage on representative P4 programs for 10 different seeds. "*" indicates that the strategy did not achieve 100% coverage within 60 minutes.

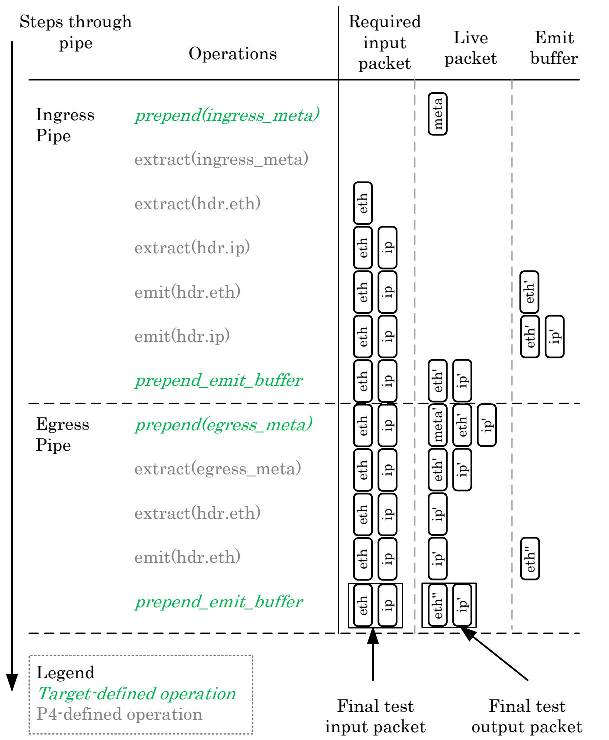
Applied precondition	None	Fixed-Size Packet	P4Constraints	P4Constraints Fixed-Size Packet	P4Constraints, Fixed-Size IPv4 Packet	P4Constraints, Fixed-Size IPv4-TCP Packet
Valid test paths	146784	83784	74472	42486	28216	7054
Reduction	**0%**	**~43%**	**~49%**	**~71%**	**~81%**	**~95%**

Table 8: Effect of preconditions on the number of tests generated for `middleblock.p4`. Fixed packet size is 1500B.

A.3 Packet-Sizing

```
parser IngressParser(...) {
    state start {
        pkt.extract(ingress_meta);
        pkt.extract(hdr.eth);
        pkt.extract(hdr.ipv4);
    }
}
control IngressControl(...) {
    apply {}
}
control IngressDeparser(...) {
    apply {
        pkt.extract(ingress_meta);
        pkt.extract(hdr.eth);
        pkt.extract(hdr.ipv4);
    }
}
parser EgressParser(...) {
    state start {
        pkt.extract(egress_meta);
        pkt.extract(hdr.eth);
    }
}
control EgressControl(...)
    apply {}
}
control EgressDeparser(...) {
    apply {
        pkt.extract(hdr.eth);
    }
}
Pipeline(
    IngressParser(), Ingress(), IngressDeparser(),
    EgressParser(), Egress(), EgressDeparser()
    ) pipe;
Switch(pipe) main;
```

(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 9: Packet-sizing for a Tofino program.