p4pktgen: Automated Test Case Generation for P4 Programs

Andres Nötzli
Stanford University

Jehandad Khan
Virginia Tech

Andy Fingerhut
Cisco Systems

Clark Barrett
Stanford University

Peter Athanas
Virginia Tech

ABSTRACT

With the rise of programmable network switches, network infrastructure is becoming more flexible and more capable than ever before. Programming languages such as P4 lower the barrier for changing the inner workings of network switches and offer a uniform experience across different devices. However, this programmability also brings the risk of introducing hard-to-catch bugs at a level that was previously covered by well-tested devices with a fixed set of capabilities. Subtle discrepancies between different implementations pose a risk of introducing bugs at a layer that is opaque to the user.

To reap the benefit of programmable hardware and keep—or improve upon—the reliability of traditional approaches, new tools are needed. In this work, we present `p4pktgen`, a tool for automatically generating test cases for P4 programs using symbolic execution. These test cases can be used to validate that P4 programs act as intended on a device.

ACM Reference Format:
Andres Nötzli, Jehandad Khan, Andy Fingerhut, Clark Barrett, and Peter Athanas. 2018. p4pktgen: Automated Test Case Generation for P4 Programs. In *SOSR '18: ACM SIGCOMM Symposium on SDN Research, March 28–29, 2018, Los Angeles, CA, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3185467.3185497

1 INTRODUCTION

Historically, a lot of the functionality of network devices was directly implemented in custom hardware for performance reasons. Faster hardware and research on software defined networking have recently made it practical to make network hardware programmable all the way down to the data plane.

This advance makes network infrastructure more flexible and capable. Devices can be reprogrammed to better suit the needs of users and programming languages offer a uniform way of programming hardware from different vendors. However, this ultimately results in replacing critical pieces of the network infrastructure that have a well-tested, fixed set of capabilities. While the design and testing was tightly coupled for this kind of hardware in the past, those new programmable programmable architectures introduce new layers and components that can fail. In addition to the danger of bugs in the software written for these devices, there is also the danger of subtle, hard-to-catch bugs in toolchains. Toolchains for traditional programming languages, such as C compilers, and hardware, such as x86, have been around for years or even decades but there are still hundreds of bugs that have been discovered in the past years [25, 27]. Unfortunately, toolchains for network hardware are unlikely to fare better, and subtle differences between toolchains—either due to bugs or different interpretations of the language specification—pose a risk of introducing bugs at a layer that is opaque to the user.

To reap the benefits of programmable hardware and keep—or improve upon—the reliability of traditional approaches, new tools are needed. Testing is one way to increase confidence in a system. For devices with a fixed set of capabilities, it is possible to generate a large amount of test cases once. Programmable devices, however, require not only hardware testing but also the associated toolchains to make sure that programs behave as intended by the user.

P4 [4] is a popular programming language for the data plane of network devices. It is backed by multiple commercial vendors—there are currently at least seven P4 compilers under development [10], each exposed to the aforementioned risks. The fact that we found several issues in the open source reference compiler, `p4c` [8], during the course of this work illustrates that those risks are real, as we discuss in Section 5.

In this work, we present `p4pktgen`,[1] an open source tool for automatically generating test cases for P4 programs. The generated test cases consist of test packets, table entries,

[1] https://github.com/p4pktgen/p4pktgen

SOSR '18, March 28–29, 2018, Los Angeles, CA, USA

ACM ISBN 978-1-4503-5664-0/18/03…$15.00
https://doi.org/10.1145/3185467.3185497

and expected paths. p4pktgen validates the test cases with BMv2 [7], a software reference implementation of a P4 switch. To generate test cases, it uses symbolic execution along concrete paths. Symbolic execution is a well-established analysis technique that translates programs into logical formulas to examine their behavior across all possible inputs. This technique can reason about programs with bit-level precision and has been used successfully to find bugs and vulnerabilities in software [5, 14, 15, 22]. Often, satisfiability modulo theories (SMT) solvers such as Z3 [11] or CVC4 [2] are used to perform the actual reasoning. Instead of re-purposing an existing tool, e.g. by translating P4 programs into a language supported by another tool, we created a new symbolic execution platform for P4 so that we could take advantage of the structure of the language for domain-specific optimizations.

Test cases generated by p4pktgen can be employed in a wide range of scenarios. One of the primary uses that we envision is to validate that P4 programs act as intended on their target devices, by running the same test packets through a software reference implementation and the target hardware and comparing the output of the two implementations. Alternatively, P4 can be used to express a specification of existing, non-programmable hardware, which can then be tested against a software reference implementation. Additionally, p4pktgen can be used for debugging purposes to quickly generate a packet for a given path.

To summarize, our contributions are as follows:

- We have formalized a large portion of the intermediate representation that the P4 compiler produces for BMv2, which closely resembles P4 and thus we indirectly provide a formalization of P4.
- We present p4pktgen, a tool that implements this formalization and is able to generate test cases fully automatically.
- We show that p4pktgen can generate test cases for large programs and that the threat of flawed toolchains is real.

2 RELATED WORK

There are multiple ongoing projects that involve formalizing components of software-defined networking. Foster et al. [13] have presented work on the verification of P4 programs. At the time of this writing, no publication is available, but the presentation indicates that they translate P4 programs to a small imperative language and then verify that user-annotated properties hold using an SMT solver. Like p4pktgen, their approach requires a formalization of P4, but their focus is on verifying the P4 programs themselves, which is desirable but complementary to our work. Kheradmand et al. [17] have presented work on formalizing P4 programs in the K framework [21]. Their focus is on P4_{14} whereas our approach is version-agnostic used and a direct formalization in SMT. Test case generation is one of their side-goals. Lopes et al. [20] developed a tool to check that a P4 program only delivers well-formed packets, modeling a small network of P4 switches in Datalog. While their focus is on verification, they also recognize that compilers may introduce bugs and suggest that their approach could be useful for verifying compiler optimizations by proving the equivalence of a P4 program before and after an optimization. NICE [6] is a tool that performs model checking to find issues in the controller programs of OpenFlow networks. p4pktgen is complementary to that approach since it focuses on testing the switches themselves but not the control plane. Dobrescu et al. [12] have proposed an approach to verify that binaries of a software data plane created with Click [18] satisfy a set of properties. This is desirable, but developers have to specify the properties and still have to trust that the switch executes what was analyzed. SOFT [19] is an approach for testing the interoperability between OpenFlow switches. Their approach performs symbolic execution of OpenFlow implementations to generate test cases and then uses these inputs to find differences between the implementations. The focus is on the OpenFlow interface and not the behavior of the data plane.

The problem of automatically generating test cases has been researched extensively. KLEE [5] is a well-established tool that performs symbolic execution of the LLVM IR along concrete execution paths to generate interesting test cases. Driller [23] combines fuzzing within code that is guarded with specific checks with symbolic execution for passing those checks. Some of these techniques may be applicable when generating test cases for P4 programs. There has also been work on fuzzing of network devices without symbolic execution. For example, Classbench [24] automatically generates packets for testing longest-prefix match algorithms. Our work does not systematically test matching algorithms but focuses on the packet processing pipeline as a whole. Zeng et al. [26] describe an approach for automatically generating test packets for testing and debugging networks. Their approach uses the header space framework [16] to model packet processing, requiring the implementation of protocol and vendor-specific translations into their model and assuming a fixed table configuration. We focus on a single switch and generate table configurations automatically.

3 THE P4 LANGUAGE

Below, we illustrate P4_{16} using an implementation of Equal Cost Multi Path (ECMP) routing, shown in Figures 1 and 2. A common function of routers is to calculate shortest paths through the network and install routes in the form of IP address prefixes mapped to output ports. Redundant paths in networks can be used to improve performance by balancing traffic to the same destination over multiple paths of the

```
1 header ethernet_t { bit<48> dst; bit<48> src; bit<16> etherType; }
2 struct headers { ethernet_t ethernet; ipv4_t ipv4; }
3 parser ParserImpl(packet_in packet, out headers hdr,
4   inout metadata meta, inout standard_metadata_t std_metadata) {
5  state start { transition parse_ethernet; }
6  state parse_ethernet {
7   packet.extract(hdr.ethernet);
8   transition select(hdr.ethernet.etherType) {
9    0x0800: parse_ipv4; default: accept;
10  }}
11  state parse_ipv4 {
12   packet.extract(hdr.ipv4);
13   verify(hdr.ipv4.version == 4 && ... && hdr.ipv4.ttl != 0,
14    error.BadIPv4Header);
15   transition accept;
16 }}
```

Figure 1: An Ethernet and IPv4 parser in $\texttt{P4}_{16}$.

```
1 control compute_ipv4_hashes(out bit<16> hash1, in headers hdr) {
2  apply {   // Cheap, but low quality, hash function
3   hash1 = (hdr.ipv4.srcAddr[31:16] + hdr.ipv4.srcAddr[15:0]
4            + ... + (bit<16>) hdr.ipv4.protocol);
5 }}
6 control ingress(inout headers hdr, inout metadata meta,
7   inout standard_metadata_t std_metadata) {
8  // ... actions set_l2ptr, set_ecmp_group_idx ...
9  table ipv4_da_lpm { key = { hdr.ipv4.dstAddr: lpm; }
10                      actions = { set_l2ptr; set_ecmp_group_idx; } }
11  // ... action set_ecmp_path_idx ...
12  table ecmp_group { key = { meta.ecmp_group_idx: exact; }
13                     actions = { set_l2ptr; set_ecmp_path_idx; } }
14  table ecmp_path { key = { meta.ecmp_group_idx    : exact;
15                            meta.ecmp_path_selector: exact; }
16                    actions = { set_l2ptr; } }
17  action set_bd_dmac_intf(bit<48> dmac, bit<9> intf) {
18   hdr.ethernet.dstAddr = dmac;
19   std_metadata.egress_spec = intf;
20   hdr.ipv4.ttl = hdr.ipv4.ttl - 1;
21  }
22  table mac_da { key = { meta.l2ptr: exact; }
23                 actions = { set_bd_dmac_intf; } }
24  apply {
25   if (hdr.ipv4.isValid()) {
26    compute_ipv4_hashes.apply(meta.hash1, hdr);
27    ipv4_da_lpm.apply();
28    if (meta.nexthop_type != L2PTR) {
29     ecmp_group.apply();
30     if (meta.nexthop_type != L2PTR) ecmp_path.apply();
31    }
32    mac_da.apply();
33 }}}
```

Figure 2: A match-action pipeline for ECMP routing.

same length. We now explain the three parts of P4 programs: the parser, the match-action pipeline, and the deparser.

Parser A P4 parser is a state machine that contains *parser states*, starting with the state `start`. The `select` statements define the transitions between the parser states based on the values of a list of fields. In Figure 1, the two parser states, `parse_ethernet` and `parse_ipv4`, parse raw Ethernet packets and IPv4 packets encapsulated in Ethernet packets. The parser state `parse_ethernet` has a transition to `parse_ipv4` that is executed if the Ethernet type matches `0x0800`; otherwise, the packet is accepted as a raw Ethernet packet by transitioning to `accept`. The `extract` statements extract information from a packet. They take a destination, and consume enough bits from the packet to fill it. In the example, the `extract` on line 7 fills the `ethernet` header, which is of type `ethernet_t`, defined on line 1. Using the `verify` statement, parser states can ensure that a condition is fulfilled. If the condition is not met, the parser exits and sets an error flag but does *not* drop the packet. Lines 13–14 use `verify` to perform sanity checks on the IPv4 packet.

Match-Action Pipeline The match-action pipeline does the actual computation in a P4 program. The underlying control flow of the ECMP example is on lines 24–33 in Figure 2. If the packet has an IPv4 header, then `compute_ipv4_hashes` calculates a hash of several of its fields. On line 27, the program performs a lookup in the *table* `ipv4_da_lpm` using the packet's destination address as a key. The router's control plane is responsible for installing *table entries*, which map keys to actions: `set_l2ptr` if there is a single shortest path to the destination, or `set_ecmp_group_idx` otherwise.

The action `set_l2ptr` assigns the constant `L2PTR` to the variable `meta.nexthop_type`, causing the multi-path code to be skipped. The action `set_ecmp_group_idx` on the other hand, assigns a value to the variable `meta.ecmp_group_idx`, indicating the group of paths that the packet should be sent to. This value is used as a key in the table `ecmp_group` on line 29. This table has again two actions: (1) `set_l2ptr` mentioned above and (2) `set_ecmp_path_idx`, which assigns a value to the variable `meta.ecmp_path_selector` that uniquely identifies the path to take within the group, based upon the hash calculated earlier in the control block `compute_ipv4_hashes`. In the last case, the table `ecmp_path` is looked up, always resulting in the action `set_l2ptr`. By the time we apply the table `mac_da`, we have a value for `meta.l2ptr`, which is used as a key. The table has a single action, `set_bd_dmac_intf`, which performs standard IP routing header modifications and sets the output port for the packet through `egress_spec`.

Deparser The deparser assembles the final output packets from the values computed by the match-action pipeline. Typically it only contains a sequence of `emit` calls, which copy the contents of headers to the output packet. Currently, `p4pktgen` does not model the deparser because most of the program's control flow happens in the parser and pipeline. Instead, it generates a test input and relies on `BMv2` to determine the expected output packet.

4 IMPLEMENTATION

In the following, we discuss how `p4pktgen` generates test cases for P4 programs by symbolically executing concrete paths. First, we describe the process at a high level; then, we describe the formalization of P4; and finally, we explain the optimizations that we added to generate test cases efficiently.

At a high level, `p4pktgen` takes a P4 program and produces test cases in the form of packets and table configurations.

Instead of consuming P4 programs directly, `p4pktgen`

parses the JSON files that the p4c compiler produces for BMv2. The format [1] is a relatively straightforward translation of the original P4 source code. The JSON format makes parsing easier because we do not have to parse P4 directly and because the format is largely agnostic to the P4 version ($P4_{14}$ or $P4_{16}$) used. An additional benefit is that p4c performs type-checking and desugaring, such as inlining calls to control blocks (e.g. p4c inlines `compute_ipv4_hashes` in `ingress` when compiling the program in Figure 2).

From the JSON file, p4pktgen extracts the parser state graph and the control flow graph of the input program. P4 guarantees that any loops in these graphs have upper bounds on the number of times they can get executed. Thus, P4 programs execute only a bounded number of steps for each packet.[2] This allows us to unroll all loops a bounded number of times while preserving the semantics.

Given a path through the program, the goal of p4pktgen is to generate a packet that exercises that path. To do this, a packet has to be crafted that triggers the correct parser transitions, the correct conditional branches, and the correct table actions. These requirements can be translated into a set of SMT constraints by symbolic execution of a given path. Starting from symbolic inputs, we encode the operations on the path as SMT constraints. If the constraints can be satisfied, the SMT solver returns concrete values for the symbolic inputs, which can be used to construct both the packet and the table entries needed to ensure that the program takes the given path. If the solver determines that the constraints are unsatisfiable, no packet can possibly take the given path.

After p4pktgen generates the test packet and table configurations for a path, it configures the BMv2 tables and sends the test packet. It compares the path extracted from BMv2's log to the expected path to sanity check the test case. A failing check indicates either a bug in p4pktgen or BMv2. If the check passes, the test case can be used to test other P4 implementations by comparing their output packets for the test case against the output packets of BMv2.

4.1 Modeling P4

Given a path, p4pktgen executes each step in the path symbolically by modeling the impact of each step on the state of the program as SMT constraints. SMT solvers support a rich constraint language based on *theories*, which define theory-specific functions and predicates. p4pktgen uses the theory of bitvectors, which defines functions and predicates on integers consisting of a fixed number of bits. Because p4pktgen only uses quantifier-free bitvectors, all the problems that it generates are *decidable*—given enough time, the solver can either find a satisfying assignment or decide that no such assignment exists. In the following, we discuss how we model various aspects of P4. In Section 6, we discuss our future plans for covering the remaining features.

Input Representation When modeling a language, one of the primary design decisions is how to represent inputs. The parser in P4 programs contains valuable information about the relationships between header fields, so p4pktgen uses a symbolic network packet as the main input and models the parser. We model the input packet as a large bitvector of a fixed size and keep track of the symbolic packet length in a separate bitvector variable *packet_length*. For every P4 program, there is a (statically computable) upper limit to the number of bits it can read from a packet. Thus, we can always choose a bitvector of that size to ensure it is large enough for all executions. We explain the motivation for this encoding below. If the solver decides that a particular path is feasible, p4pktgen retrieves a concrete value n for the packet length from the solver; it then obtains the packet by taking the first n bits of the large bitvector that models the packet.

Extracting Fields While parsing, P4 extracts fields from a packet. Because of variable-length fields, p4pktgen keeps track of the current position in the packet as a symbolic value. Thus, p4pktgen has to be able to extract fields with symbolic length from a packet at a symbolic position. The SMT input language for bitvectors does not support extracting bits from symbolic positions but the same effect can be obtained by using a variable-length shift followed by an extract:

$$field\_val(pos, sz) := (packet \gg pos)[sz - 1 : 0]$$

where $\gg$ is the right-shift operator. Notice that this would be difficult to model if the packet were split into different bitvectors, because each bitvector has a fixed size and so there would be no easy way to determine which bits to read, given a symbolic offset into the packet. Because we model the packet as a single, large bitvector, lookahead can be treated in the exact same way as extracting a header field.

For variable-length fields, we extract the maximum number of bits and then zero out the number of bits that are outside of its actual (symbolic) length:

$$field\_val\_var(pos, sz, max\_sz) := \\ field\_val(pos, max\_sz) \;\&\; ((-1)^{[max\_sz]} \gg (max\_sz - sz))$$

where & represents the bitwise-and operator, and $(-1)^{[n]}$ represents the bitvector of all ones of length n. We also add the constraint that $sz \leq_u max\_sz$ where $\leq_u$ is an unsigned comparison and a constraint that $sz \;\&\; \texttt{0x7} = 0$ to make sure that sz is divisible by eight, as required by $P4_{14}$.

Context As p4pktgen steps through a path in a program, it keeps track of the current symbolic values in a *context*. The context maps header fields and other variables to symbolic values and keeps track of metadata such as the validity of headers. For reads, p4pktgen looks up the symbolic value in the context, and for writes, it updates the corresponding value in the context. Thus, the context always contains

[2] Except in cases of recirculation, which we currently do not model.

the current symbolic expression of all variables. When reading values from the context that have not been initialized, p4pktgen supports treating them as zero-initialized ($P4_{14}$ semantics) or as undefined values where each read may result in a different value ($P4_{16}$ semantics). Similarly, p4pktgen can warn about writing to fields in headers that are not valid, which some implementations may treat as a memory corruption even though it is defined as a no-op in $P4_{14}$. When changing a header to invalid, all values in the header are removed from the context.

Parser States In parser states, we translate all the parser operations, e.g. header extractions, one-by-one. Header extractions are modeled as a series of field extractions and marking the header as valid. The path determines which case of a `select` statement the input has to hit. This entails matching the required case and ensuring that none of the cases before could apply, resulting in the constraint $\neg(c_1) \wedge \cdots \wedge \neg(c_{n-1}) \wedge c_n$ where c_i are the conditions for the cases and n is the index of the case that corresponds to the path. For `select` statements that match tuples instead of single fields, p4pktgen concatenates all the elements in the tuple and performs checks on the concatenated bitvector.

Conditionals The path determines the outcome of a conditional, e.g. whether `hdr.ipv4.isValid()` should be `true` or `false` in Figure 2. We simply translate the condition and add a constraint that sets it equal to the desired outcome.

Actions, Tables and Table Entries Table entries are set at runtime, thus p4pktgen has to generate the table entries for a given path. For each table, the path determines the action that needs to be executed. Our current implementation assumes that each table can only appear once in a single path, mirroring a limitation of the JSON format. Thus, p4pktgen has to generate at most a single table entry for each table. To get the values for the key, p4pktgen records the symbolic values of the match keys each time a table is used. p4pktgen retrieves the concrete values from the SMT solver's assignment and uses them to generate the keys of table entries. Additional constraints are not necessary. All match types (exact, longest-prefix match, ternary, and range) support the special case of exact match on all bits, and p4pktgen always creates such table entries. Table entries include values for the action parameters. Each time an action appears in a path, p4pktgen generates fresh variables for the action parameters and records them in the context indexed by a table-action pair for later retrieval. The actions themselves consist of primitive operations that are translated sequentially.

Assuming that a table only appears once on a given path is not a fundamental limitation of our approach. If we wanted to support a table appearing multiple times, we could add the following constraint for each reuse of a table:

$$keys_1 = keys_2 \Rightarrow action_1 = action_2 \wedge params_1 = params_2$$

Note that we know the value of $action_1 = action_2$ statically from the path. If the values of the keys are the same for both uses of the table, then we must pick the same action and the action parameters remain the same because the table will execute the same action for the same input. In this case a single table entry is required. Otherwise, the actions and action parameters can be different and a separate table entry must be generated.

Error Paths Packet processing in a P4 program can result in errors. It is important to exercise those corner cases because they are not used as often during normal execution. Recall that parser states use `verify` to check whether a packet fulfills a requirement and reject it otherwise. To generate a test case that fails a `verify` statement, p4pktgen simply adds the negation of the condition to the constraints.

If an extract or a lookahead should fail in a given path because a packet is too short, p4pktgen does not perform the operation and instead restricts the packet length to be shorter than the length required by the operation but long enough that no preceding operation fails. Variable-length fields can exceed their maximum length when doing an extract. In that case, we add a constraint that the value of the expression that computes the length of the variable-length field has to be greater than the maximum length.

4.2 Optimizations

In p4pktgen, we generate paths by visiting nodes in a depth-first order. Instead of attempting to find packets for complete paths, p4pktgen first tries to solve prefixes of paths and backtracking as soon as it determines that a prefix is impossible to satisfy. This allows p4pktgen to discard infeasible pieces of the search space early. For the parser, we do not attempt to solve prefixes because unsatisfiable parser paths are rare.

State-of-the-art SMT solvers support *incremental solving* as defined by SMT-LIB [3]: The user can push (save) and pop (restore) the set of assertions and check for satisfiability in between. This mechanism works well with our backtracking optimization. Whenever p4pktgen tries to solve a prefix, it pushes the current context and pops the context when it backtracks. This allows the solver to reuse work from shorter prefixes when solving longer ones.

5 EVALUATION

Our evaluation of p4pktgen focuses on two questions: (1) Can p4pktgen effectively generate test cases for large P4 programs? (2) Can p4pktgen's test cases reveal bugs in a P4 toolchain? For the former, we show that p4pktgen is able to produce a large number of test cases for P4 programs. For the latter, we describe four bugs that we discovered in p4c.

Performance We ran p4pktgen on four P4 programs using an Intel Core i5-4258U CPU: the `ECMP` example, two `mTag`

Program	Runtime	Total paths	Test cases	Imp. Prefixes
ECMP	4.8*s*	210	28	14
mTag-aggregation	83.5*s*	1,728	1,345	3
mTag-edge	176.6*s*	3,456	1,974	465
switch.p4	1,800*s* (t/o)	2.7×10^{34}	3,425	3,664

Table 1: Results of running p4pktgen on P4 programs. "Imp. Prefixes" refers to path prefixes that p4pktgen found to be impossible to generate a packet for.

programs, and a slightly simplified version of switch.p4 [9]. The mTag example originates from the original P4 paper [4] and implements simple tag-based packet forwarding. It consists of mTag-edge, which inserts and removes tags in switches attached to hosts and mTag-aggregation, which forwards packets based on the tags. We modified switch.p4 to remove features not yet supported by p4pktgen (such as header stacks, hashing, and action profiles) and we replaced a call to random by an action parameter to make the program deterministic. We summarize the results in Table 1. The runtime includes the time of checking the test cases with BMv2.

We were able to generate test cases for all paths in the smaller three programs in less than 3 minutes each. Our largest test case switch.p4, is too large to run to completion due to the explosion in the number of paths, a common issue when doing symbolic execution of paths. For this case, we limited the runtime to 30 minutes. p4pktgen generated 16.1 test cases per second in the best case (mTag-aggregation) and 1.9 test cases per second in the worst case (switch.p4). In the case of mTag-aggregation, the control block consists only of three table applications, so the control block does not introduce any constraints, which makes finding test cases a matter of picking the right values for the parser and adding the correct table entries. For switch.p4, the process gets significantly more complicated due to the size of the program: the average length of a successful path is 41 nodes. Thus, this shows that our method is able to create test cases for programs that are significantly larger than toy examples.

For ECMP, the sum of test cases generated and impossible prefixes (i.e. the paths explored) is significantly smaller than the total number of paths, indicating that pruning is effective. This is due to conditions such as the check at the beginning of the control block that requires an IPv4 header. Thanks to the pruning described in Section 4.2, p4pktgen does not explore any paths beyond that condition if the path does not parse an IPv4 header. In mTag-edge, pruning is also effective, cutting roughly 30% of the total paths. Solving took 8–27% of the runtime and running the test cases through the BMv2 took 21–39%. The rest of the time was used to initialize the translator, start BMv2, generate paths, and generate constraints.

Case Studies While developing p4pktgen, we wrote simple P4 programs that ultimately uncovered bugs in p4c. When the test cases that p4pktgen generated did not produce the results we expected, we investigated the cause and found, in all cases reported below, that the issue was the p4c JSON backend. With two compiler implementations, it is possible to find these bugs fully automatically by checking for differences in the output of the two compiled programs. We reported the following issues that have since been fixed:[3]

- Issue #914: Incorrect JSON for select statements with multiple key fields.
- Issue #983: Incorrect JSON when bit-wise negating a bitvector, then casting it to a wider bitvector.
- Issue #995: Incorrect JSON for select statements that use masks for ternary matching.
- Issue #1025: incorrect JSON specifying maximum length of headers with variable-length fields.

This list of issues shows that the danger of bugs in P4 compilers is real and that the test cases can help reveal them.

We also found p4pktgen useful for finding the root cause of an issue reported by a third party (issue #950), where p4c generated incorrect JSON for Boolean variables in conditions.

6 LIMITATIONS AND FUTURE WORK

We plan to add support for more P4 features in p4pktgen such as header stacks, header unions, hashes, and action profiles. Those features do not require fundamental changes to our approach. P4 also allows state to be maintained between packets, e.g. using registers and meters. For P4 implementations that allow setting the state using the control plane, p4pktgen can be modified to simply generate commands to set the desired state before each test case. However, some P4 programs may assume that the state is never modified externally after initialization. To support such behavior, we would have to find an efficient way to model sequences of packets. Currently, p4pktgen does not model the egress pipeline, but it is similar to the ingress pipeline, so modeling it does not require fundamental changes except for the recirculation of packets, which can result in arbitrary length loops.

Exploring all paths on large P4 programs is impossible, thus we are working on a mode that prioritizes branch coverage instead of path coverage. The challenge is to efficiently find a set of paths that exercise all branches. We are considering adding support for user-supplied constraints to express assertions and assumptions on table entries and action parameters to better capture the intention of the programmer.

Finally, we would like to use the generated test cases to test different P4 implementations to realize the full potential of p4pktgen. We believe that our approach is one of several tools required for reliable P4 implementations.

Acknowledgments

This work was partially supported by a grant from Cisco. We thank Fraser Brown and Christopher Aberger for their valuable feedback. We thank Colin Burgin for his help with the implementation.

[3]The programs that we used to find the issues are attached to the reports.

REFERENCES

[1] 2017. *BMv2 JSON input format.* Technical Report. https://github.com/p4lang/behavioral-model/blob/master/docs/JSON_format.md.

[2] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *CAV*.

[3] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2016. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`. (2016).

[4] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. 2014. P4: Programming protocol-independent packet processors. *SIGCOMM* (2014).

[5] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs.. In *OSDI*.

[6] Marco Canini, Daniele Venzano, Peter Peresini, Dejan Kostic, and Jennifer Rexford. 2012. A NICE way to test OpenFlow applications. In *NSDI*.

[7] P4 community. 2017. BMv2. https://github.com/p4lang/behavioral-model. (2017).

[8] P4 community. 2017. p4c. https://github.com/p4lang/p4c. (2017).

[9] P4 community. 2017. Switch. https://github.com/p4lang/switch. (2017).

[10] Huynh Tu Dang, Han Wang, Theo Jepsen, Gordon Brebner, Changhoon Kim, Jennifer Rexford, Robert Soulé, and Hakim Weatherspoon. 2017. Whippersnapper: A P4 Language Benchmark Suite. In *SOSR*.

[11] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. *TACAS* (2008).

[12] Mihai Dobrescu and Katerina Argyraki. 2015. Software dataplane verification. *Commun. ACM* (2015).

[13] Nate Foster, Cole Schlesinger, Robert Soulé, and Han Wang. 2017. A Program Logic for Automated P4 Verification. (2017). http://p4.org/wp-content/uploads/2017/06/p4-ws-2017-hoare.pdf.

[14] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: directed automated random testing. In *ACM Sigplan Notices*.

[15] Patrice Godefroid, Michael Y Levin, and David Molnar. 2012. SAGE: whitebox fuzzing for security testing. *Commun. ACM* (2012).

[16] Peyman Kazemian, George Varghese, and Nick McKeown. 2012. Header Space Analysis: Static Checking for Networks.. In *NSDI*.

[17] Ali Kheradmand and Grigore Rosu. 2017. Executable Formal Semantic of P4 and Applications. (2017). http://p4.org/wp-content/uploads/2017/06/p4-ws-2017-p4k-executable-formal-semantic.pdf.

[18] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M Frans Kaashoek. 2000. The Click modular router. *TOCS* (2000).

[19] Maciej Kuzniar, Peter Peresini, Marco Canini, Daniele Venzano, and Dejan Kostic. 2012. A SOFT way for openflow switch interoperability testing. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*.

[20] Nuno P. Lopes, Nick Mckeown, Dan Talayco, and George Varghese. 2016. *Automatically verifying reachability and well-formedness in P4 Networks.* Technical Report. https://www.microsoft.com/en-us/research/publication/automatically-verifying-reachability-well-formedness-p4-networks/

[21] Grigore Rosu and Traian Florin Şerbănuţă. 2010. An overview of the K semantic framework. *The Journal of Logic and Algebraic Programming* (2010).

[22] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: a concolic unit testing engine for C. In *ACM SIGSOFT Software Engineering Notes*.

[23] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *NDSS*.

[24] David E Taylor and Jonathan S Turner. 2007. Classbench: A packet classification benchmark. *TON* (2007).

[25] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *ACM SIGPLAN Notices*.

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

[27] Qirun Zhang, Chengnian Sun, and Zhendong Su. 2017. Skeletal program enumeration for rigorous compiler testing. In *PLDI*.