FP4: Line-rate Greybox Fuzz Testing for P4 Switches

Nofel Yaseen  
University of Pennsylvania

Liangcheng Yu  
University of Pennsylvania

Caleb Stanford  
University of Pennsylvania

Ryan Beckett  
Microsoft

Vincent Liu  
University of Pennsylvania

Abstract

Compared to fixed-function switches, the flexibility of programmable switches comes at a cost, as programmer mistakes frequently result in subtle bugs in the network data plane.

In this paper, we present the design and implementation of FP4, a fuzz-testing framework for P4 switches that achieves high expressiveness, coverage, and scalability. FP4 directly tests running switches by generating semi-random input packets and observing their resulting execution in the data plane. To achieve high coverage and scalability, at runtime, FP4 leverages P4 itself with another “tester” switch that generates and mutates billions of test packets per second entirely in the dataplane. Because testing some program branches requires navigating complex semantic input requirements, FP4 additionally leverages the programmability of P4 by instrumenting the tested program to pass coverage information back to the tester through the packet header.

We present case studies showing that FP4 can validate both safety and stateful properties, improves efficiency over existing random packet generation baselines, and reaches 100% coverage in under a minute on a wide range of examples.

1 Introduction

Computer networks have evolved to include more flexible platforms in which data plane functionality can be defined by programmers using domain-specific languages like P4 that describe devices’ data plane processing. These devices are opening doors to better support applications [23, 38] and to improve network management [21]. Although the increased programmability offers great benefits, it also brings the increased risk of introducing new bugs that are difficult to catch, either in the data plane, control plane, language compiler, ASIC implementation, or any combination of the above.

Bugs may arise as a result of anything from simple programmer typos to divergent interpretations of the P4 specification [11]. They can even include behavior that spans multiple packets because switches can store and recall state in registers, trigger control plane transitions, and reconfigure match-action rules as a result of incoming packets. Combined, all of these factors increase the complexity of bugs in switches [12, 25].

To reduce bugs, recent work has suggested static verification to prove the correctness of P4 programs [12, 14, 25]. For pure, stateless data plane programs, static verification is often effective since there are no complex pointer-based data structures or loops, making analysis both more accurate and tractable. However, real forwarding behavior depends on many other components: the control plane; the exact past, present and future match rules; the compiler translation; the switch state including registers; and specifics of the hardware implementation. For example, in the process of developing FP4, we discovered a subtle compiler/runtime bug in our installed SDE in which a multicast primitive in the default action of a table with no entries does not properly multicast the packet. All other commands in the default action execute correctly as does the same setup in the provided simulator. Static verification tools that only consider the P4 program itself cannot catch this class of bug.

We note that, in traditional programs, developers often rely on fuzz testing to catch this wider class of bugs.

A fuzzer generates semi-random inputs to discover assertion failures, memory leaks, and crashes. Fuzz testing is able to evaluate applications in their natural environment (ignoring issues that are impossible to reach and catching issues that only arise in the presence of the application’s surrounding components). Fuzzing, and particularly blackbox and greybox fuzzing, also tends to scale well with the complexity of control flow and state. As others have noted [8], these approaches often find more bugs than whitebox approaches like static verification and symbolic-execution-based test case generation as the latter either (a) spend significant time doing program analysis and constraint solving or (b) further sacrifice precision, e.g., by approximating functions that are hard to reason about analytically, such as hash functions.

In this work, we observe that, when applied to programmable switches, not only does fuzzing allow a developer to check the entire device in vivo—incorporating effects of the data plane, control plane, ASIC implementation, and compiler—it also allows the fuzzer to leverage the intrinsic hardware parallelization, pipelining, and acceleration of packet processing in today’s network devices. Explicitly optimized for fast packet processing, switch-based fuzzing potentially enables input testing that is orders of magnitude faster than is possible in a CPU.

To that end, we present FP4, a greybox fuzz testing framework for P4-programmable network devices that is both (a) full-stack and (b) line-rate. FP4 feeds semi-random packets to real programmable switches to attempt to trigger violations of programmer-specified assertions. The switches are purposely kept as faithful as possible to their production deployments and run instrumented versions of their original P4 programs and control planes.

As a preview of the performance benefits of FP4’s ap-
Figure 1: Maximum possible throughput of a single instance of modern fuzzing frameworks, both those for traditional programs (AFL, honggfuzz, and libfuzzer) and for P4 programs (p4pktgen and FP4). In each case, the fuzz target is an empty function or data plane program.

<table>
<thead>
<tr>
<th>Type</th>
<th>p4v</th>
<th>vera</th>
<th>p4wn</th>
<th>p4pktgen</th>
<th>P6</th>
<th>FP4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Line rate</td>
<td>static</td>
<td>static</td>
<td>static</td>
<td>runtime</td>
<td>runtime</td>
<td>runtime</td>
</tr>
<tr>
<td>DP logic</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>DP state</td>
<td>❌</td>
<td>✓</td>
<td>✓</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
</tr>
<tr>
<td>Compiler</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Control plane</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Hardware</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 1: Comparison of the features of a selection of P4 verification and testing frameworks, including whether they can catch bugs in data-plane logic, stateful behavior, the compiler, the control-plane, and hardware.

To address the first two challenges, FP4 splits switch state into a few categories. For data-plane-only state, it leverages the fact that switches are meant to run continuously and, thus, most network tasks allow for intrinsic state resets (e.g., when a counter rolls over or a flow entry times out) [46]. Thus, continued fuzzing will eventually allow the switch to re-explore previous states. For everything else (i.e., control plane, table entry, or configuration state), FP4 borrows another idea from traditional fuzz testing—context-sensitive branch coverage [9]—that seamlessly integrates state changes into greybox fuzzing approaches.

We implement and deploy FP4 to a hardware testbed in order to instrument and debug real P4 programs. FP4 works by modifying the input P4 program in a way such that it has no impact on normal packet processing. It also adds an extra header that stores information to track the actions visited and assertions failed by the packet. FP4 uses this tracking information to generate new seed packets. Our work makes the following contributions:

- We leverage the observation that programmable switches can generate semi-random packets at line-rate to design, implement and evaluate fuzz testing framework for P4 programs that generates test packets 6 orders of magnitude faster than similar work for P4.
- We introduce a novel technique to instrument P4 programs to track their coverage and check for assertion failures at line rate.
- We implement an FP4 prototype and evaluate it on a diverse set of P4 programs. Our results show FP4 achieves 100% coverage quickly – in <1 min in most cases.
- To ensure reproducibility and facilitate future work, we will release FP4 as an open source tool on publication.

2 Background and Motivation

In this section, we cover the challenges of discovering bugs in P4 switches and the possible role of fuzz testing.
2.1 Potential Bugs in P4 Programs

Bugs can occur in any point of the deployment and execution of a P4 program. They can include but are not limited to:

**Bugs in the application logic.** The most straightforward class of bugs exists in the P4 code itself. In some cases, these issues are a result of ambiguities or subtleties in the language specification [11]. More generally, however, programmers are fundamentally fallible and just as capable of introducing bugs to P4 programs as they are to traditional code, especially when trying to reason about edge cases or complex interactions between features. For example, a P4 reference program previously contained a bug where ACL rules were incorrectly applied to control-plane traffic [32].

**Issues in the compiler or hardware implementation.** The P4 program must be compiled to run on the hardware and optimized to adhere to resource limitations on pipeline stages, SRAM, TCAM, etc. As above, the programmers of these components are also fallible, creating instances where an otherwise correct P4 program produces unexpected behavior. While this class of bugs is typically rarer due to longer development cycles, lower-level specifications, and heavier testing, the above multicast issue and a glance at the errata of any processor or compiler documentation validates their presence. These are among the most difficult type of error to diagnose.

**Bugs in the control plane.** Switch operation depends on the combination of the data and control planes. While the data plane is responsible for handling per-packet processing, the control plane—operating in parallel on a general-purpose CPU—is responsible for managing the data plane and handling all of the tasks that are too complex for line-rate processing. These include installing, updating, and removing data-plane table rules as well as executing routing protocols. All of these can evolve based on the sequence of incoming packets—real control planes are both dynamic and stateful.

**Switch misconfigurations.** Finally, network switch behavior is also affected by switch configuration options like knobs in the traffic manager, buffer slicing, and port speeds. These configuration options can be independent and set separate from either the traditional data plane or control plane programs. Errors can arise either from operator misconfiguration or through interactions with other issues, e.g., in the hardware implementation.

Bugs can occur within any of the above components, and some only manifest when issues in multiple layers combine.

2.2 Fuzz Testing

For traditional applications, programmers often augment their software engineering workflows with fuzz testing [17]. Fuzz testing feeds the program a set of random inputs and observes whether the program behaves correctly on each such input. This process is able to automatically discover bugs, even when those bugs result from complex runtime behavior and interactions between heterogeneous systems.

As prior work has noted, however, the naïve approach of random inputs (i.e., pure blackbox fuzzing) can often lead to poor coverage as many inputs are simply invalid or fail to explore program paths with complex or hard-to-hit branch conditions [16]. On the other hand, approaches that try to reason precisely about the program’s structure (i.e., pure whitebox fuzzers) come with their own set of issues ranging from being unable to model complex functions (e.g., hash functions) to exhibiting poor scaling that makes them not worth the extra overhead [8]. In the end, many of the most prolific fuzzers take a greybox approach that attempts to strike a balance. FP4’s input generation takes inspiration from three methods from the literature on greybox fuzzing of traditional programs:

(1) **Coverage-guided fuzzing** is exemplified by the widely used AFL fuzzer [18]. AFL begins with a set of seed inputs that it subsequently mutates to create random inputs that are then fed to the program. Based on feedback from the program (detailing paths and branches covered), AFL learns the quality of the inputs and selectively updates the set of seed inputs to explore new execution paths. This leads to significant improvement in the rate of coverage compared to completely random inputs.

(2) **Generator-based fuzz testing** allows users to write generator programs for producing inputs (see [33]). As an example, consider the structure of an Ethernet/IP protocol stack, which might accept IP-related EtherTypes and discard all other packets as corrupted or otherwise invalid. A generator-based fuzzer will generate only valid IP packet inputs. This ensures that the fuzzer does not waste time on inputs that are immediately discarded by input sanitation.

(3) **Context-sensitive branch coverage** is introduced in the Angora fuzzer (see [9], Section 3.2). Angora observes that not every execution of the same code block (containing a conditional branch) is equal. Instead, the current state of the program and its call stack can make an execution of the same code block materially different from prior executions. Including this context in coverage tracking therefore improves feedback and responsiveness.

3 Overview

Like most other coverage-guided greybox fuzzers, FP4 is built around a single loop in which FP4 generates a packet from a selected set of seeds, mutates the packet semi-randomly, passes it through the target (programmable switch), and computes the state and path coverage to determine whether the packet is a good candidate for a new seed. However, unlike other fuzzers, FP4 is extremely fast. It achieves this speed with two domain-specific insights:
Modern switches can execute packet processing orders of magnitude faster than commodity CPUs, and that speed is independent of the complexity of the program as long as it fits within a single pass through the switch. As mentioned, running the target system in vivo provides benefits to the speed, completeness, and accuracy of fuzz testing. A key contribution of FP4 is to demonstrate that the fuzzer is also deployable to programmable switches, generating, modifying, and checking at line rate.

Switch programs are intended to be long-lived. This means that, in steady state, switch programs typically have intrinsic mechanisms that reset persistent state, e.g., when a counter overflows, when a ring buffer wraps, or when routes are torn down. This allows FP4 to test most state transitions without needing an explicit reset of the switch. An exception are bugs that occur during initialization, but those are typically straightforward for operators to catch during development and canarying.

An FP4 testbed consists of two switches: (1) the switch under test, which runs the target data plane, control plane, and switch configuration; and (2) the switch doing the test, which generates inputs, checks program coverage, and manages seed packet selection. See Figure 2 for a visual depiction.

**Switch Under Test (SUT).** The SUT executes the target switch system, including both its P4 data plane and control plane program. The system should operate identically to a real deployment with the exception of some additional annotations and instrumentation.

The annotations come in the form of a simple operator-specified error conditions on a given packet’s contents or the state in the switch (i.e., registers, counters, and meters).

In the automatic instrumentation step, FP4 then inserts code into both the data plane and (optionally) control plane programs to aid in checking for path coverage to trigger the above errors. This instrumentation takes the form of an additional packet header, an operation in every data-plane action, emulated output ports, and a couple of additional tables for bookkeeping and assertion checking. All of the above changes incur minimal overhead and, crucially, leave the original metadata, headers, and control path intact.

**Switch Doing Test (SDT).** Alongside the SUT, we run a second switch, the SDT. The SDT is responsible for all of the traditional tasks of a fuzzer: generating test packets, mutating them, sending them to the SUT, and checking for violations and coverage after packets return from the SUT.

It consists of a data-plane test generator, which is a synthesized companion P4 program that generates the test packets, mutates them, tracks coverage, and checks for assertion failures. The generator leverages a set of dynamically updated seed packets to generate billions of semi-random test packets per second. FP4 mutates the packets at line rate using the programmability of the P4 switch. The mutations are such that they retain the validity of the packet but attempt to steadily increase the coverage of the fuzz testing.

Supporting the data-plane test generator is a control-plane fuzzing manager that is used to consider seed packets candidates, modify the data-plane generator accordingly, notify users of assertion failures, and perform other tasks that are beyond the capabilities of today’s programmable data planes. Like in traditional networks, executing these tasks asynchronously in the control-plane CPU (while carefully maintaining correctness) allows the data plane to continue operating at line rate.

The remainder of this paper describes the design of FP4’s SUT and SDT in more detail.
We begin by outlining FP4’s modifications to the SUT.

4 Switch Under Test (SUT) Instrumentation

We begin by outlining FP4’s modifications to the SUT.

4.1 Programmer Assertions

One type of instrumentation in FP4 involves programmers adding assertions in their P4 programs that define error conditions in the processing of a packet. One example of violation might be where the time to live field of a packet is zero, but the program fails to actually drop the packet:

```c
assert(ipv4.ttl != 0 || std_metadata.drop == 1)
```

More generally, operators specify fields and their range of invalid values using basic comparison operators and boolean logic.

These assertions serve as syntactic sugar that FP4 uses to automatically generate a set of tables, actions, and table rules that will catch the assertion at runtime. Violations are marked in an FP4 packet header that is appended to the packet (see Figure 3 for the header’s format). Note that operators can use this syntax to detect issues that span multiple packets by manually tracking relevant information in stateful elements.

4.2 Coverage Instrumentation

The other type of instrumentation in FP4 enables its greybox, coverage-guided fuzzing within the data plane. Traditional fuzzers typically track coverage at the granularity of basic blocks, adding instrumentation to each branch to record ‘seed-worthy’ inputs that trigger additional program coverage (i.e., that are not redundant with existing seeds). Programmable switches, with their concomitant control planes and frequent rearrangements of control flow (via control plane intervention), impose additional restrictions on what it means for an input to be worthy of use as a seed. FP4 considers:

- **Actions**: In most P4 implementations, the most convenient single-entry, single-exit, straight-line (with the exception of ALU operations) block of code are the actions of the match-action pipeline. The goal of FP4 is to fuzz test all possible actions, so coverage of novel actions is cause for addition to the input corpus.
- **Table entries**: The actions that are triggered and the conditions under which they are triggered are determined by the table entries of the match-action pipeline. The overall path is defined by a sequence of table entry hits. Thus, table entries—and in particular, their union—have a massive influence on the reachability of bugs.
- **Control plane state**: Finally, while FP4, its design, and its assertions are primarily focused on bugs in the data plane, we note that packets sometimes pass through the control plane as part of their processing (e.g., routing updates that eventually add/remove table entries). FP4 is not concerned with the code coverage of the control-plane program but does care about how it might eventually affect the data plane, e.g., through table entry updates.

Purposefully missing from the above set is data-plane state. While data-plane state (like table entries and control plane state) may also impact control flow and lead to additional program coverage, we found that properly tracking the uniqueness of data-plane state in the presence of per-packet register access limitations and packet reordering imposed too much overhead and too many limitations on the scope of P4 programs that FP4 can test. We leave an exploration of more efficient methods of state tracking to future work.

In the remainder of this section, we describe the SUT instrumentation required to track changes to the above entities.

4.2.1 Actions Visited

To track the coverage of every action, FP4 assigns a bit in the `fp4_header` for each action, and marks the respective bit in each packet as it passes through the switch pipeline. For example, consider the target program of Figure 4, there are four total actions and, thus, four reserved bits in the header. Laying it out in this way ensures that every unique path through the pipeline results corresponds with a unique ‘visited’ bitstring. Note that if the same action is used in multiple tables, a naive application of the above may leave ambiguity in the packet’s path through the processing pipeline. FP4 addresses this by duplicating the action and renaming it, so each action is unique to a table. The renaming has no impact on the switch hardware resources.

4.2.2 Control-plane State Changes

To account for the impact of control-plane changes, FP4 augments the control plane to track its internal state. Note that this can include everything relevant to the processing of future packets, from object attributes that persist across packet events to the current state of the stack (which reflects function calls, parameter changes, and returns). This additional data can only improve coverage, but is not necessary for functionality.

Naïvely, one could consider every packet that causes a state change as a candidate for inclusion in the seed corpus. Unfor-
Unfortunately, this fails to distinguish new states from previously seen ones. Instead, FP4 leverages the CRC-32 algorithm to compute a 32-bit hash of the control-plane state (all global variables followed by the sequence of function calls on the stack) that is both efficient to update and can distinguish between unique states. CRC-32 values can be updated bi-directionally (i.e., they support both pushing and popping bytes), so the hash is maintained on both function calls and returns in constant time. Our FP4 prototype implements this approach with a semi-automatic annotation process. It currently assumes that the control plane is written in Python and all functionality is contained within a single class. Programmers annotate the class with a superclass and add a decorator to each of its methods and local variables, which wraps them to update the CRC value on each call, return, and modification (in principle, these can be automated). FP4 also automatically stores the last-seen packet/digest from the data plane and wraps all table entry modifications.

Whenever a table entry is added or the control-plane state changes, FP4 checks if the CRC value is novel and there is an active ‘input packet/digest.’ If both are true, the SUT control plane forwards the new state hash and the original headers of the input packet to the SDT for inclusion in the seed corpus.

4.2.3 Table Entry and Configuration Changes

Runtime updates to table entries and switch configurations can also affect the data-plane behavior, and FP4 tracks them using a similar technique as above. Specifically, FP4 automatically computes the CRC-32 of the string representation of all the configuration changes (the value is kept separate from the hash of internal control-plane state). For example, it imposes on the table write/update library calls to automatically track the content of every table. It represents each entry in its runtime-CLI-command format (which provides a simple, unique representation of the entry), and computes the hash of a sorted list of such commands. As above, FP4 automatically checks the uniqueness of the state and, if unique, forwards the input packet to the SDT.

5 Switch Doing Test (SDT) Design

To handle the fuzzing tasks, FP4 leverages a second programmable switch: the SDT. For the tasks that must execute for every fuzzing input, FP4 leverages the line-rate processing capabilities of the SDT’s data plane. For other tasks, FP4 leverages the SDT’s control-plane CPU to implement more complex behaviors that improves its choice of inputs. In total, the SDT generates test packets, mutates them, sends them to the SUT, and checks for violations/coverage after packets return from the SUT. Figure 5 illustrates this lifecycle.

5.1 Packet Generation

To generate the input packets for the target at line rate, FP4’s SDT uses the built-in hardware packet generation capabilities of the Tofino and similar switches. Generated packets contain an fp4_visited header (depicted in Figure 3) with all fields initialized to 0s. The generated packets are then passed through automatically synthesized tables that transform the zeroed packet into one of a limited set of seed packets. The tables first select a random seed number. Based on that seed, the SDT will add a set of headers to the packet and fill them in with an initial value corresponding to one of the configured seed packets. Figure 6 provides a snippet of the synthesized actions that FP4 uses to transform the generated packets.

When FP4 is first executed, it only contains few seed packets based on the program in SUT; more are added during runtime (see Section 5.3).

Deriving expected packet formats. The first step in any P4 program is the parser, which takes a sequence of bits from the MAC layer and parses it into its constituent headers. Packets at this stage must adhere to strict formats—all others are dropped before reaching the ingress pipeline of the switch. When generating packets, FP4 ensures that all seeds (whether from the initial set or added later) pass this stage using a generator-based fuzzing approach.
Figure 6: Actions that are used to create an Ethernet+IPv4 packet from an existing seed. These correspond to the ‘Add headers’ and ‘Set field values’ steps of Figure 5. For the example target of Figure 4, a separate set of actions would be synthesized for creating Ethernet-only packets.

More specifically, we note that P4 parsers are structured as state machines. The parser transitions between different states depending on the contents of the packet; a packet is only fed into the packet processing pipeline if it reaches an accept state in the parser. FP4 analyzes the state machine to find all paths from the start to any terminal state; it records the headers extracted in each path along with any field and header contents that triggered the path.

FP4 synthesizes implementations of the above seed-packet generation tables so that it is able to configure seed packet contents from the control plane. Adding a seed is as simple as inserting a table rule to the above tables.

Computing an initial set of seeds. FP4 also uses information from the parser to compute the initial set of seeds. Specifically, it pre-computes one seed packet for each unique parser path, setting the content of the seed packet header to specific values that are part of transitions in the state machine1. All other fields in the seed header that are not constrained by the parser state machine transitions are randomly populated.

As an example, consider Figure 4 and its synthesized actions in Figure 6. The parser for this program always extracts an Ethernet header, and then only extracts an IPv4 header if the contents of the EtherType are “0x0800.” Its state machine, therefore, consists of three states: the start state (not shown), a state to parse the Ethernet header, and a state to parse the IPv4 header. Further, there are only two unique paths through this state machine that lead to accepting states: (1) an L2 frame with only an Ethernet header and (2) an L3 packet with both Ethernet and IPv4 headers. During initialization of the SDT, FP4 discovers these two paths and randomly generates two initial seed packets that will trigger these parser paths.

1Note that, like other implementations of the P4 compiler, FP4 limits parser recursion to a specified depth, ensuring finite seed packets.

5.2 Mutating the Generated Packets

With the seed packet in hand, FP4’s goal is then to use the packet to expose new switch behavior. A simple straw man approach to mutating packets would be to randomly select a subset of the header fields and set them to random values. While such an approach will eventually catch any bug, blindly mutating packets may only rarely result in inputs that traverse new control flow or trigger new table actions. Instead, FP4 takes a more targeted approach using one of three techniques per packet (with configurable probability of each decision).

(1) Targeting a specific table entry or conditional statement. Fundamentally, code coverage is determined by the actions triggered in the SUT. For a packet to trigger a given action, its headers and metadata must match the set of ‘keys’ in a table entry that is mapped to the target action. FP4 takes advantage of the fact that the current set of table entries are known precisely at runtime to implement a ‘magic value’ approach to fuzzing [24, 36].

More specifically, the SDT contains a ‘mutation’ table with an action corresponding to each match-action table and conditional statement in the SUT. In the case of a SUT table, the action takes its match fields as parameters; thus, whenever an entry is added to a SUT table, FP4 can attempt to add a corresponding entry to the mutation table with the match-key constants passed as parameters to the table-specific action. LPM keys are converted to their base value; ‘do not care’ bits of ternary matches are converted to zeros. In the case of a conditional, the action takes any referenced fields as parameters, and FP4 tries to add a corresponding entry based on static analysis of the program.

When target values are sparse and directly dependent on the input packet header, this technique can greatly speed coverage. For example, consider an ingress MAC filter that only matches the interface and broadcast MAC addresses, two values out of $2^{48}$. Even at ∼2 billion packets per second, triggering the action would take an average of ∼20 hours.
(2) Targeting a table or if statement. Note that not all matched fields are directly configurable. Examples include tables that match on metadata or on fields that are changed in previous stages of the pipeline. For these cases, FP4 takes advantage of the P4 program’s structure to preferentially mutate a set of fields together if it knows these fields are more likely to result in “hitting” a new table entry or conditional branch, thus dramatically shrinking the search space of mutations.

But how does FP4 decide what groups of fields should be mutated together? To determine this, FP4 makes use of a lightweight, stage-sensitive static analysis over the program control flow graph (CFG). The analysis extends traditional flow-insensitive static analysis techniques [28] to be sensitive to packet modifications occurring at different stages. In particular, FP4 statically analyzes the input program to create a dependency graph that captures whether each packet field “could be” relevant for a given table lookup. The graph for Figure 4 is shown in Figure 7. It contains nodes for each of the packet’s header and metadata fields as well as for each table appearing in the program.

In P4, each table is associated with a stage number. FP4 labels the table nodes with that stage number and adds an arrow between two nodes when there is a dependency between these components. For example, if on_if($a) modifies the egress_port of the packet, the egress_port field would depend on the content of both the ipv4.dstAddr and a vrf metadata field. The vrf field may, in turn, depend on ethernet.dstAddr if it is modified in an action of the ethernet_forward table. FP4 adds arrows for fields that are used as keys in tables as well as between tables whose actions may influence the future lookup of other tables. It also adds dependency edges when fields are used in conditionals or updates of stateful ALUs. In most cases, there is only one edge between tables; however, when an if statement immediately follows a table lookup, there can be more than one dependency edge as there are multiple possible next tables.

Each dependency edge is labelled with the earliest stage in which the dependency occurs. The mutation procedure is, thus, as follows: For each table in the graph, FP4 precomputes the set of all packet fields that can impact that table. These fields are all those that can reach the table node in the dependency graph using only edges with stage labels less than or equal to the table stage. At runtime, for each seed packet, FP4 stores the list of tables that this seed might be able to mutate given the fields present in the seed (which might be a subset of all possible fields). During mutation, after selecting seed packet, it picks one table from the list and preferentially mutate together fields that are keys of that table.

(3) Targeting stateful counters. Finally, while FP4 does not explicitly track data-plane state (for the reasons in Section 4.2), FP4 can still discover bugs that depend on stateful behavior. The table-targeted mutation technique, for instance, will properly track the dependencies of stateful registers and modify them if the register outputs are useful for increasing coverage or testing assertions (it will not attempt to mutate state that is write-only). There are, however, edge cases where FP4’s lack of visibility into data-plane state changes may impede its efficiency. For example, consider a target program that counts the number of packets for each 5-tuple and triggers an assertion violation if any counter exceeds a threshold. FP4 will quickly cover the action with the stateful counter, and it will eventually trigger the violation (after sufficient random collisions), but may do so slowly. Noticing this tendency toward counters in network programs, FP4 will occasionally repeat packets to ensure that the most common classes of stateful behaviors are captured quickly.

Chaining mutations. Note that, particularly for (1) and (2), it may be advantageous to chain mutations to trigger matches that are impossible with only one mutation of existing seeds. FP4 can pack a few such mutations within a single pipeline. It can also optionally recirculate the packet to apply even more rounds of mutations, albeit at the cost of throughput.

5.3 Evaluating Assertions and Coverage

At the end of pipeline, FP4 makes a copy of all headers to reserved _original headers. If the packet is later determined to be seed-worthy, the cloned headers serve as a record of the original input packet, prior to any SUT modifications.

Two types of packets will return from the SUT: test packets that have traversed the SUT data plane and state-change notifications from the SUT control plane. It may also receive packets generated at the SUT and destined for remote devices (e.g., a periodic control plane routing keepalive message), but these never result in an addition of seed packet (as they are not the result of an SDT input).

For packets from the SUT, the header will contain the visited action bitstring and assertion failure flags along with the original header. The packet has attained additional coverage iff its visited bitstring is novel, i.e., it visited a unique sequence of actions or path. Because the string can potentially be large, FP4 tracks uniqueness with the help of a bloom filter. On a filter miss or a set assertion flag, FP4 sends the packet and its original header to the SDT control plane for further processing. Otherwise, FP4 recycles the packet by removing all the headers; the recycled packets are treated as a freshly generated packet. Packets from the SUT control plane are always sent to the SDT control plane for addition to the seed packet set. The SUT control plane should have already verified its uniqueness.

6 Implementation

We implemented a prototype of FP4, including the SUT instrumentation and SDT data plane and control plane agent. Our hardware testbed consists of two Barefoot Wedge100BF-
32X programmable switches with all ports on both switches connected by an array of 100 GbE DAC cables. Our implementation currently uses a single line card on each switch.

**SUT Instrumentation.** The FP4 instrumentation adds the required changes to the input P4 program stated in Section 4 to generate an instrumented P4 program. In total, instrumentation implementation comprises around 5500 lines of C++ code, with 4300 lines for a frontend to parse input P4 code using Flex/Bison and build an AST of the input program and 1200 lines to instrument the target SUT program.

The SUT control plane scaffolding currently requires that the programmer annotate their code as described in Section 4.2.2 and adhere to a general entrypoint signature.

**SDT implementation.** Our prototype SDT data plane implements the pipeline and functionality detailed in Section 5. The code to synthesize the program for SDT incorporates an additional 4200 lines of code. The program synthesis code also outputs a json file to be used by the SDT control plane. This json file contains the structure of packets so the control-plane can parse the incoming packets.

Hardware limitations in the per-stage random number generator restrict the size of mutations to 32-bits, but we find that table-entry-targeted mutations are sufficient to fill this gap. Our prototype SDT control plane takes in the json file generated during instrumentation, adds initial seed packets, parses packets coming from the data plane, track coverage and installs new seed packets in the SDT data plane. The Python control plane is more than 1200 lines of code.

### 7 Evaluation

To evaluate the performance of FP4, we conducted experiments on a diverse collection of programs that vary in size and complexity. Rather than merely reaching a particular behavior (such as an assertion violation or invalid header access on a particular line), we focus on the more holistic problem of achieving full coverage of all actions and paths in P4 programs, which can be combined with assertions to catch specific bugs. As such, our evaluation aims to address the following questions: (1) How quickly does FP4 achieve 100% code coverage, compared to existing tools for software-based fuzz testing? (2) Which factors of FP4s design have the biggest impact on coverage, and how does its performance compare to more naïve baselines? (3) What is the performance overhead of the added FP4 instrumentation on the switch under test? (4) Finally, can FP4 be used successfully to find bugs in existing P4 programs—with a particular eye to bugs that could not be caught with static verification techniques?

**Programs tested.** Table 2 lists tested programs. Load Balancer, Basic Routing, and Rate Limiter represent programs designed primarily for packet forwarding. The Load Balancer makes use of hashing, while Rate Limiter tests Stateful ALUs.

---

![Figure 8: Action coverage over time for FP4 and p4pktgen.](image)

Firewall represents a more complex application using Bloom filtering. Netchain also uses concurrent control.

In order to avoid false positives due to invalid table rules inserted by the control plane [12], for these examples, we employ a static controller which inserts a fixed number of table rules.

We also evaluate an additional two programs, Mirroring and DV Router. The first program contains the bug mentioned in the introduction: our goal is to determine whether we can catch the bug automatically with fuzzing. The second program is included to test the behavior of switch together with a dynamic, stateful control plane. This example involves additional instrumentation in the control plane, as overviewed in Section 4.2.2. Both examples include components that cannot be handled by static verifiers: a hardware-only bug in the first case, and a dynamic stateful control plane in the second case.

#### 7.1 FP4 Covers Code Quickly

We begin by evaluating the speed at which FP4 can provide code coverage and test programmable switches. We ran FP4 over all of the programs described in Table 2 on the setup described in Section 6, and we log every time a packet arrives at the SDT control plane with a newly covered path through the P4 program or state.

Figure 8a shows the speed at which FP4 triggers all actions in the target programs. The y-axis is the fraction of unique actions triggered divided by the total number of unique table-action pairs in the programs. We note that, in all but one of our test programs, FP4 provides complete coverage for all the
can eventually trigger this sequence of events, step (2) is FP4 with only a single entry in a table with a wide keyspace. The (we modified p4pktgen so that it would stop considering im-

 programs within around 1 min. This is true even for programs with only a single entry in a table with a wide keyspace. The only exception was DV Router, where an action hit depends on (1) a properly formatted incoming routing update followed by (2) a matching ARP response for the next-hop router and (3) a packet that hits the target routing table entry. While FP4 can eventually trigger this sequence of events, step (2) is currently improbable as it relies on a metadata field (nextHop) whose dependencies cross the DP/CP boundary. Figure 9a also shows results from the same run for paths through the program, defined as either a unique sequence of triggered actions or a change in the control-plane state.

We also show results for another open-source P4 fuzzer, p4pktgen [31]. We note an important difference between the experiments: p4pktgen uses symbolic execution to solve for a small number of input packets and assumes it can freely configure the control plane rules when doing so. As such, some of the inputs/configurations are not actually achievable in real networks with real control planes. In fact, we needed to remove all default_action statements from the test programs (we modified p4pktgen so that it would stop considering im-

Figure 9: Path coverage over time for FP4 and p4pktgen over up to a 5 min trace. When comparing (a) and (b), we caution readers to consider the differences laid out in Section 7.1.

<table>
<thead>
<tr>
<th>Program</th>
<th>Features</th>
<th>Resource Overhead</th>
<th>Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LoC</td>
<td>Actions</td>
<td>Stateful ALUs</td>
</tr>
<tr>
<td>Load Balancer</td>
<td>159</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>Basic Routing</td>
<td>165</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>Rate Limiter</td>
<td>197</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>Firewall</td>
<td>313</td>
<td>12</td>
<td>4</td>
</tr>
<tr>
<td>Netchain [22]</td>
<td>264</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>Mirroring</td>
<td>213</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>DV Router</td>
<td>284</td>
<td>11</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 2: P4 programs on which we evaluate FP4, resource overhead of the instrumentation, and time to achieve full coverage.

Figure 10: Packet efficiency (i.e., actions covered, relative to total, per test packet) with and without FP4's optimizations.

In contrast, FP4 takes the deployed program and its existing table entries and configurations. Despite this advantage, Figures 8b and 9b demonstrate that p4pktgen still struggles to achieve full action coverage in real programs. While they can achieve some coverage very quickly, limitations in its current language support (e.g., statefulness and egress logic) mean that some paths are never solvable.

Note that the higher path coverage of p4pktgen in Figure 9b is due it taking impossible paths. For example, Firewall contains a IPv4 lookup followed by per-port map lookup. p4pktgen finds paths that include an IPv4 miss (where the packet is dropped) followed by a map hit on a non-existent port. The static table rules do not allow FP4 to take this path.

7.2 Factor Analysis

To evaluate the benefits of different features of FP4, we evaluate the packet efficiency of FP4 in covering the P4 program over a 5 min period. Figure 10 shows these results for (1) the full FP4 implementation, (2) FP4 without magic values (the mutations of Section 5.2 that are targeted for specific table entries or conditional statements), and (3) FP4 without coverage-guided fuzzing (and only the parser-based seed generation). We show values for two programs that illustrate different effects. Basic Routing benefits from magic values to quickly cover most actions, but hitting the remaining actions relies solely on the long tail of randomness. DV Router, on the other hand, is aided by FP4's optimizations in both its initial (first 100 packets) and final coverage.
We also use FP4 to test for issues in a device in which the P4 data plane and its control plane interact to implement a distance-vector powered IPv4 router. The router provides the basic functionality required for it to be placed in an arbitrary network and learn its surroundings. Thus, the router’s functionality includes the ability to handle ARP requests and responses, to understand Ethernet forwarding, and to execute a distance vector protocol to determine the correct set of LPM routing table entries. Like a real router, the data plane is designed to be general to the topology, port counts, and address assignments of the network; instead, the control plane will configure all tables based on a provided interface configuration and data learned from neighbors or passing packets.

The original purpose of this router is as a teaching tool: students are given some skeleton code and are expected to fill in the P4 and control-plane logic for the above protocols. Testing and debugging students’ implementations is a critical task. While testing the end-to-end correctness of the implementation is straightforward (e.g., by deploying a set of their routers and connecting two Linux hosts to either side), it is often also useful to test for violations of basic invariants, which can serve as both sanity checks and a method to localize errors. Because of the tight integration of the control and data plane, static verification tools are not sufficient and prone to false positives. In particular, there are many invariants where one set of table rules may result in correct behavior and another that result in errors. For example:

- Testing that the router only responds to L2 frames destined for the local interface’s MAC address or the broadcast address.
- Testing that outgoing packets are all filled with a sender MAC address corresponding to the egress port.
- Testing that outgoing ARP responses match the incoming requests (e.g., the correct operation code and sender/target hardware and protocol addresses).

Further, the target includes routing packets that are generated by the onboard control plane, and while the data plane could produce errors depending on the contents of the generated packets, they will not because of the correctness of the control plane. Example of such properties include:

- Testing that outgoing distance-vector updates include the correct source IP.
- Testing that the outgoing distance-vector updates are properly formatted, e.g., the advertised cost is less than some preconfigured ‘small infinity’ so as to mitigate the count-to-infinity problem of distance vector protocols.

In all cases, FP4 enables expressive testing of the complete programmable switch that only finds bugs that are feasible.

### 7.3 Overhead of FP4

FP4’s primary overheads stem from the instrumentation it adds to the SUT in order to gain enough visibility to implement its greybox fuzzing approach. We note that the resource consumption of the SDT is less important as the program is not co-resident with any other programs. Rather, Table 2 shows the key resources required on the SUT, which are generally low across all tested programs. The overhead does mean that not all programs are amenable to FP4's greybox approach. An exploration of which of FP4’s features can be relaxed to address programs that are already close to exhausting all resources is out of the scope of this work.

### 7.4 Case Studies

We now present our experiences testing and finding issues in real programs using FP4.

#### 7.4.1 Mirroring Bug

As previously mentioned, during the development of FP4, we stumbled on a bug in the interaction between mirroring and default actions. Specifically, given a table such as Figure 11, if the action aiMirror() configures the packet to be mirrored, the P4 specification would suggest that the packet should be mirrored even if the user does not add any entries beyond the default entry. In reality, while other primitive actions in aiMirror() execute properly, we find that packet is not multicasted. Only after adding a non-default entry does the multicast properly apply. After adding an assertion to the input program (Mirroring), FP4 finds the path that violates the assertion in under 1 seconds.

While this bug was identified and addressed in more recent versions of the switch SDE, static verification tools that only consider the P4 program itself would not have found any faults in this program. Further, we note that the simulators provided as part of the behavioral model and the switch SDE also do not catch this bug. Only when this program is deployed to a hardware switch will this issue manifest.

#### 7.4.2 Testing a Distance Vector Router

We also use FP4 to test for issues in a device in which the P4 data plane and its control plane interact to implement a distance-vector powered IPv4 router. The router provides the basic functionality required for it to be placed in an arbitrary network and learn its surroundings. Thus, the router’s functionality includes the ability to handle ARP requests and responses, to understand Ethernet forwarding, and to execute a distance vector protocol to determine the correct set of LPM routing table entries. Like a real router, the data plane is designed to be general to the topology, port counts, and address assignments of the network; instead, the control plane will configure all tables based on a provided interface configuration and data learned from neighbors or passing packets.

The original purpose of this router is as a teaching tool: students are given some skeleton code and are expected to fill in the P4 and control-plane logic for the above protocols. Testing and debugging students’ implementations is a critical task. While testing the end-to-end correctness of the implementation is straightforward (e.g., by deploying a set of their routers and connecting two Linux hosts to either side), it is often also useful to test for violations of basic invariants, which can serve as both sanity checks and a method to localize errors. Because of the tight integration of the control and data plane, static verification tools are not sufficient and prone to false positives. In particular, there are many invariants where one set of table rules may result in correct behavior and another that result in errors. For example:

- Testing that the router only responds to L2 frames destined for the local interface’s MAC address or the broadcast address.
- Testing that outgoing packets are all filled with a sender MAC address corresponding to the egress port.
- Testing that outgoing ARP responses match the incoming requests (e.g., the correct operation code and sender/target hardware and protocol addresses).

Further, the target includes routing packets that are generated by the onboard control plane, and while the data plane could produce errors depending on the contents of the generated packets, they will not because of the correctness of the control plane. Example of such properties include:

- Testing that outgoing distance-vector updates include the correct source IP.
- Testing that the outgoing distance-vector updates are properly formatted, e.g., the advertised cost is less than some preconfigured ‘small infinity’ so as to mitigate the count-to-infinity problem of distance vector protocols.

In all cases, FP4 enables expressive testing of the complete programmable switch that only finds bugs that are feasible.

### 8 Related Work

Fuzzing and testing for programmable dataplanes. Prior to FP4, p4pktgen [31], P4RL [40], and P6 [39] were the first to propose fuzzing for P4 dataplanes. These tools demonstrated that P4 fuzz testing could be effective at identifying a wide variety of bugs. Going off the past observation from software fuzzing that more tests equals more bugs [8], FP4 leverages programmable switches to generate test packets up to 6 orders
of network protocols [1, 6, 15, 29, 34, 35, 44]). Compared to these works, FP4 can not efficiently restart the switch, and thus opts not to generate sequences of test inputs. Instead it leverages the observation that most P4 programs are meant to be run in a “continuous” mode and naturally reset switch state. FP4 also collects control plane state change information and incorporates this information to derive new seed inputs.

9 Limitations and Future Work

In principle, by virtue of randomness, FP4 is eventually able to trigger any possible data plane bug except those that involve unused fields or that only manifest when incoming packets are rare. One could also cover those types of bugs, e.g., if FP4 had support for dropping a random number of seed packets, but doing so would greatly sacrifice throughput.

Looking forward, we note that there are significant opportunities for extending FP4 to incorporate more recent advancements in the field of fuzzing, and/or to more efficiently cover several classes of bugs that are currently inefficient for FP4 to catch. The space of possible optimizations is infinite, but promising directions include: coordinated time stamp emulation in the data plane and control plane to better handle timing-triggered bugs, pruning of old or useless seeds to improve the efficiency of exploration, prioritization of existing seeds to target known gaps in the program coverage, gating seed packets based on the current system state to guarantee uniqueness of seed coverage, and tracking control plane code coverage to more efficiently explore bugs that originate there.

10 Conclusion

In this paper, we present FP4, the first line-rate greybox fuzzing framework for P4 programmable switches. FP4 adapts several carefully selected, time-tested ideas from the realm of traditional application fuzz testing and demonstrates how to adapt the ideas to programmable switches—with the switches serving as both the target and the fuzzer. Our evaluation demonstrates that FP4 can quickly find bugs in programs, even if the bugs are not in the P4 program itself (e.g., in the case of compiler and control-plane bugs).

References


p4lang. ipv6 nd fix and prevent cpu bound packets hitting egress acl (issue #82), Dec 2016.


