SwitchV: Automated SDN Switch Validation with P4 Models

Kinan Dak Albab*
Brown University

Jonathan DiLorenzo
Google

Stefan Heule*
Financial Choice

Ali Kheradmand
Google

Steffen Smolka
Google

Konstantin Weitz*
Financial Choice

Muhammad Timarzi†

Jiaqi Gao
Harvard University

Minlan Yu
Harvard University, Google

ABSTRACT

Increasing demand on computer networks continuously pushes manufacturers to incorporate novel features and capabilities into their switches at an ever-accelerating pace. However, the traditional approach to switch development relies on informal specifications and handcrafted tests to ensure reliability, which are tedious and slow to maintain and update, effectively putting feature velocity at odds with reliability.

This work describes our experiences following a new approach during the development of switch software stacks that extend fixed-function ASICs with SDN capabilities. Specifically, we focus on SwitchV, our system for automated end-to-end switch validation using fuzzing and symbolic analysis, that evolves effortlessly with the switch specification. Our approach is centered around using the P4 language to model the data plane behavior of the switch as well as its control plane API. Such P4 models are then used as a *formal specification* by SwitchV, as well as a *switch-agnostic contract* by SDN controllers, and a *living documentation* by engineers.

SwitchV found a total of 154 bugs spanning all switch layers. The majority of bugs were highly relevant and fixed within 14 days.

CCS CONCEPTS

• **Networks** → *Network reliability*; Programming interfaces; • **Software and its engineering** → *System description languages*; *Software verification and validation*;

KEYWORDS

P4, P4 modeling, SDN switch validation, PINS, SAI, fuzzing, symbolic execution, automated test generation

ACM Reference Format:

Kinan Dak Albab, Jonathan DiLorenzo, Stefan Heule, Ali Kheradmand, Steffen Smolka, Konstantin Weitz, Muhammad Timarzi, Jiaqi Gao, and Minlan Yu. 2022. SwitchV: Automated SDN Switch Validation with P4 Models. In *ACM SIGCOMM 2022 Conference (SIGCOMM '22), August 22–26, 2022, Amsterdam, Netherlands.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3544216.3544220

*Work fully or partially performed while at Google.

†Work performed while at Google and Harvard University.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

SIGCOMM '22, August 22–26, 2022, Amsterdam, Netherlands

© 2022 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-9420-8/22/08.
https://doi.org/10.1145/3544216.3544220

1 INTRODUCTION

Motivation Demands on computer networks are ever-increasing. Networks are constantly challenged to become more reliable, more flexible, and more efficient. This drives manufacturers and operators to design, implement, and deploy new features and capabilities at an accelerating pace. In this paper, we focus on the following conundrum: How can we increase the reliability of our network infrastructure while simultaneously increasing feature velocity?

This question has led hyper-scalars to adopt novel approaches. Google uses *Software Defined Networking (SDN)* [19, 28], a network architecture that separates the control and data planes, to increase feature velocity and improve debugging. Microsoft [4] and Amazon [2] use *network verification (§8)* to detect network *configuration bugs*. Meanwhile, the way we build switches, and especially their software comprising network operating systems and control APIs, has largely remained the same: we write informal, often incomplete and ambiguous specifications (if any) in English, and check specification compliance using hand-crafted test suites. Verification techniques have been applied to switch hardware (§8), but not switch software or end-to-end correctness.

We are reaching an inflection point at which this approach no longer scales, due to these trends: First, with SDN, the controller, the switch software stack, and the switch hardware and drivers are typically developed by different teams across several companies. Timely and correct integration of these components thus hinges on precise encoding and understanding of the specification and API of each component, which must be agreed upon by various teams across organizational boundaries. Second, data from Microsoft Azure [55] and Facebook [40] suggest that the success of network verification at catching configurations bugs may have reached a point of diminishing returns, with many network failures now occurring due to *switch hardware and software bugs*. Finally, the accelerated evolution of switches is increasing the complexity of their software, which now often include an abundance of features and capabilities unlikely to be fully used by any single operator, which makes their validation and maintenance more challenging [13].

At its core, the traditional approach to switch design—relying on informal English specifications and hand-crafted tests—inherently puts feature velocity and reliability at odds, forcing switch designers to carry out a balancing act between these two properties. Increasing reliability entails writing more tests, which saddles the development of new features with having to update these tests as

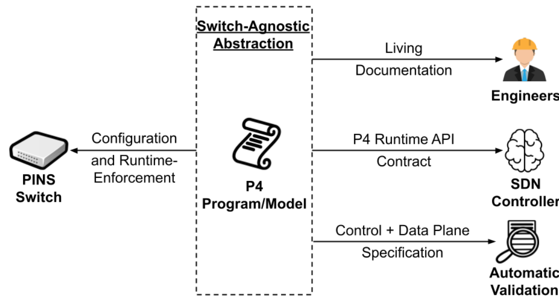

Figure 1: We model fixed-function PINS switches using P4 programs, which provide a implementation-agnostic abstraction for the components in the ecosystem.

the specification evolves. Developers are driven to quickly introduce new features to meet increased demand and new requirements, which often results in deprioritizing the maintenance of existing tests or the addition of new ones. Thus, designers constantly run the risk of their switches, various levels of specifications, and tests going out of sync, or of abandoning the specs altogether. Similarly, network operators are faced with a trade-off between quickly deploying new switches with new desired capabilities into their networks, and ensuring the overall reliability of their networks.

We attempt to reconcile these seemingly contradictory goals using a novel approach to switch design and development. This paper reports on our experience applying this approach during the co-development of (i) the P4 Integrated Network Stack (PINS) [20], (ii) a fixed-function [5] switch running PINS, and (iii) a data center fabric based on that switch (Figure 4).

Our Approach Our methodology is centered around automated validation that evolves automatically (*i.e.* without any additional effort) along with the switch specification. To that end, we use the P4 [16] programming language to model the end-to-end behavior of switches. While P4 is traditionally used to program P4-enabled switches, we view P4 programs as a *machine-readable* and *implementation-agnostic* formal specification of the control plane API (*i.e.* the tables that can be programmed via P4Runtime [15]), and the data plane behavior of switches (*i.e.* how the switch forwards packets), including switches that are not P4-enabled. Modeling via P4 programs is the linchpin that connects the various components of our approach as shown in Figure 1.

We design each of our P4 programs to model the behavior of a switch in a specific deployment role (*e.g.* ToR, WAN), modeling only the capabilities needed in that role. This makes our models simpler and more portable across switch implementations.

We introduce SwitchV, our framework for validating switches automatically. SwitchV validates that a given switch conforms to our P4 modeling with high confidence. At a high level, SwitchV automatically analyzes the P4 model and generates two types of *differential tests* for validating the *control plane API* and the *data plane* forwarding behavior of the switch. SwitchV monitors the behavior of the switch as the tests are run against it, and checks that the behavior matches the expected behavior of the P4 program. If a mismatch is detected, SwitchV generates an incident report that human testers can inspect to identify the root cause, which may be a bug in the switch or the P4 model.

Automated validation using SwitchV helps reconcile the tension between reliability and feature velocity, as it alleviates the burden of updating and extending hand-crafted tests. Instead, SwitchV automatically generates new tests as the specification evolves. We have structured our approach to provide safeguards and incentives to keep the P4 model in sync with the implementation: We run SwitchV periodically (*e.g.* daily), catching any divergence between the P4 model and the switch behavior almost immediately. Additionally, in contrast to informal English specifications, updating our P4 models as the implementation evolves provides immediate value by yielding test coverage for new or changed features "for free"; it is also a technical necessity for exposing features in PINS to the controller, as the control plane API is defined by the P4 program. Effectively, this makes our P4 models a *living documentation* that engineers can consult for a precise, yet abstract and implementation-agnostic view of the current end-to-end behavior of the switch, mitigating the problem of out-of-date specifications.

We used SwitchV to validate two switch stacks under development called PINS and Cerberus (§6). SwitchV found 122 and 32 bugs in the two stacks, including bugs in the hardware, various software layers, the P4 toolchain, and the P4 models themselves.

Ethics Statement This work does not raise any ethical issues.

2 OVERVIEW

The centerpiece of our approach is using P4 to specify the API and behavior of the switch in its intended role (§3). The choice of P4 as a modeling language is integral to SwitchV, as it enables the automated validation of the control plane API of the switch and its data plane forwarding behavior.

The P4 language semantics ensure that P4 programs are unambiguous, making them suitable for use as formal specifications. P4 has relatively few and simple constructs (*e.g.* compared to general-purpose languages such as C++), which makes it easier to automatically analyze and reason about P4 programs, while also being more mature and familiar to network engineers than a custom-made modeling language. We show a simplified portion of a P4 program that expresses parts of a typical IPv4 routing flow in Figure 2.

Fixed-Function Switches Our focus in this paper is on the use of SwitchV to validate fixed-function switches [5] running the PINS software stack (Figure 4). A fixed-function switch consists of a rigid ASIC with limited flexibility. Concretely, the forwarding pipeline in such a switch is mostly fixed and encoded in the hardware: Operators cannot arbitrarily change the routing logic, control flow, and supported protocol headers. However, such a switch can still be programmed by a controller by installing table entries that the fixed logic matches against. For example, the controller can install entries that forward packets with a certain destination IP on a certain port, or drop packets from a specific source address. The controller performs this programming by issuing requests via the switch's control plane API. There are also limited ways in which the so-called ACL tables, which are invoked at pre-determined places in the rigid packet-processing pipeline, can be configured prior to programming, allowing to trade off expressivity (# tables, # bits that a table matches on) with scalability (# of table entries supported).

```
1  control routing(in headers_t headers,
2                  inout metadata_t metadata) {
3   /* ... */
4   @entry_restriction("vrf_id != 0")
5   table vrf_tbl {
6    key = { metadata.vrf_id : exact; }
7    actions = { no_action; }
8    const default_action = no_action;
9    size = ROUTING_VRF_TABLE_MINIMUM_GUARANTEED_SIZE;
10  } // end of vrf_tbl
11  table ipv4_tbl {
12   key = {
13    metadata.vrf_id : exact @refers_to(vrf_tbl, vrf_id);
14    headers.ipv4.dst_addr : lpm;
15   }
16   actions = { drop; set_nexthop_id; /* ... */ }
17   const default_action = drop;
18   size = ROUTING_IPV4_TABLE_MINIMUM_GUARANTEED_SIZE;
19  } // end of ipv4_tbl
20  apply {
21   vrf_tbl.apply();
22   if (headers.ipv4.isValid()) {
23    ipv4_tbl.apply();
24   }
25   /* ... */
26  } // end of apply
27 }
```

Figure 2: Simplified portion of a fixed-function routing pipeline modeled as a P4 program.

```
// id table     match keys        => action        action args
   v1 vrf_tbl   1                 => no_action       void
   v2 vrf_tbl   0                 => no_action       void
   v3 vrf_tbl   3                 => set_nexthop_id   1
   i1 ipv4_tbl 1 10.*.*.*         => set_nexthop_id 3
   i2 ipv4_tbl 5 10.*.*.*         => drop            void
   i3 ipv4_tbl 1 10.*.*.*         => set_nexthop_id void
   i4 ipv4_tbl 1 0DB8:*:*:*:*:*:*  => set_nexthop_id 1
   i5 ipv4_tbl 1 10.0.*.*         => set_nexthop_id 10
```

Figure 3: Table entries for Figure 2 in a human-readable form. Entries v2, v3, i2, i3, and i4 are invalid.

We model fixed-function switches as P4 programs. This is an unorthodox use of P4, which is designed and traditionally used to install custom forwarding pipelines onto P4-enabled switches. In contrast to a fixed-function switch, a P4-enabled switch re-arranges its pipeline whenever a P4 program is installed on it, effectively acting as an interpreter of that program. After installation, the controller issues control requests to the P4 switch to manage its table entries, whose signatures must match those defined in the installed P4 program, as shown in Figure 3. The interface exposed to the controller is governed by the P4Runtime Protocol [15], a standardized, RPC-based protocol specifying the exact binary format of these requests and how the switch is allowed to handle them.

PINS Fixed-function switches from different vendors or of different makes may have different capabilities and internals and may expose different APIs and protocols to the controller. This makes deploying and managing a heterogeneous network challenging. Recent work proposes various switch software layers [13, 39, 50] that provide common abstractions and APIs. The *P4 Integrated Network Stack* (PINS) [20] is a new software switch stack that extends fixed-function switches with limited programming capabilities and a unified control API. PINS is based on SONIC [56], an open-source network operating system build atop the vendor abstraction layer SAI [49], allowing it to run across hardware from different vendors. PINS extends SONIC with a P4Runtime interface—configured and governed by an accompanying P4 program—to the controller. Together, the P4Runtime Protocol and the P4 program constitute a contract between the PINS switch and the controller, precisely specifying program-independent concerns, *e.g.* the signature and abstract semantics of RPC calls, and program-dependent concerns, *e.g.* the tables exposed for programming and the packet-forwarding semantics of their entries, and the *constraints* on table entries that encode hardware limitations, respectively. The same P4 program is used to configure the ACLs on the PINS switch. PINS implements a P4Runtime server that receives requests from the controller, checks that they comply with the aforementioned constraints, and applies them to the underlying ASIC via a vendor-agnostic abstraction layer called the Switch Abstraction Interface (SAI) [49].

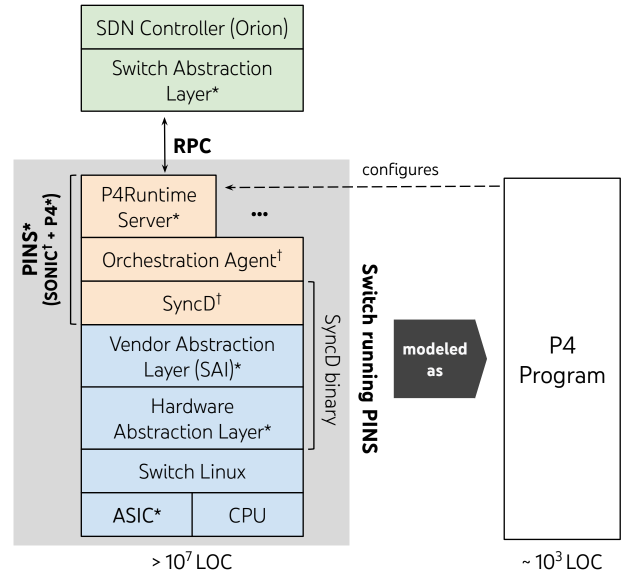

Figure 4: A fixed-function switch running PINS. New or extended components are marked with ∗ and † respectively.

In addition to PINS, we used SwitchV to validate Cerberus, another software stack for fixed-function switches whose details we discuss in §6. In principle, SwitchV is also directly applicable to P4-enabled switches. The control plane API validation component of SwitchV relies on the switch exposing a P4Runtime API to communicate with the switch under test and to judge—based on a given P4 program and the P4Runtime standard—whether the observed behavior is admissible. Our data plane validation component is largely independent of the P4Runtime Protocol, and can in principle be extended to validate switches that do not support it.

Scope We used SwitchV to validate the control plane API and the packet forwarding behavior of PINS-based and Cerberus-based switches end-to-end. This includes validating both the new layers added by these stacks (*e.g.* the P4Runtime interface) as well as the existing layers they are built on top of (*e.g.* the hardware ASIC, the underlying operating system), to the extend that these layers affect the control plane API and packet forwarding.

SwitchV does not validate QoS (which, to a first approximation, only affects forwarding during congestion) or "management" and "operational" aspects of the switch, *e.g.* port speed configuration or TLS certificate configuration, respectively.

Recent work categorizes bugs found in switches into several classes [7]. SwitchV has detected bugs from a majority of these classes in practice, including *functional bugs*, bugs in the *architecture* and associated *development tools*, as well as bugs caused by *under-specified behavior.* SwitchV is *not* designed to detect bugs from the two remaining classes, *security* and *performance*, which are usually an artifact of the switch's configurations.

Design SwitchV consists of `p4-fuzzer` (§4) and `p4-symbolic` (§5) responsible for generating tests for the control plane API and data plane behavior, respectively. `p4-fuzzer` generates a sequence of control plane requests for installing, modifying, or deleting various table entries, including valid requests as well as "useful" invalid ones. `p4-symbolic` generates test packets that satisfy the coverage assertions provided by test engineers, *e.g.* hitting every table entry.

Testers provide both components with an input P4 program that acts as a specification. Additionally, they provide `p4-symbolic` with a set of table entries that represent the switch's present forwarding state. These are usually a replay of production table entries. Testers also provide a coverage metric (*e.g.* branch or trace coverage) relative to the P4 program encoded using Boolean assertions.

For each type of generated test, SwitchV provides a mechanism for judging whether the switch's response was admissible or not. `p4-fuzzer` provides an Oracle that determines whether the response of a switch to a control plane request complies with the P4Runtime standard instantiated for the given P4 program, which determines the format of table entries and may contain additional constraints in the form of `@refers_to` and `@entry_restriction` (§3) annotations. We run test packets generated by `p4-symbolic` against the BMv2 P4 simulator configured with the input P4 program and table entries, and check that the behavior of the switch matches some observed behavior of BMv2. For both types of tests, we do not predict a single correct outcome, but rather check that the observed behavior is valid. For any test, there may be multiple valid behaviors due to under-specification in the P4Runtime Protocol, or non-determinism in the P4 program.

When SwitchV encounters switch behavior that it deems to be invalid, it produces a log of the incident. A human must inspect this log to investigate the root cause of the issue, and how it can be addressed. This may be a result of a bug in the switch, our P4Runtime Oracle, or in the P4 simulator. Additionally, when the switch is fixed-function, it may be that the switch's behavior is correct, and the P4 program incorrectly encoded the desired functionality.

3 MODELING A FIXED-FUNCTION SWITCH IN P4

We discuss our experience designing P4 programs that we use as models for fixed-function switches running PINS (Figure 4). At a high level, our P4 programs are an encoding of SAI with a similar structure to the SAI object model [50]. For the most part, we encode each SAI object as a P4 match-action table. Our P4 models are role-specific: they share a similar high-level structure and re-use many of the same components. However, they differ in components that depend on the deployment role of the switch.

We encode various resource limits and semantic constraints into these P4 programs, such that the P4 program includes all the necessary information to determine whether any control plane request (*e.g.* installation of table entries) would be accepted by the switch. Figure 2 shows simple examples of this where the P4 program specifies a minimum number of entries (i.e. size) for each table that the hardware is guaranteed to meet. This guarantees that the switch will accept any request that is valid from the perspective of the P4 program (and its embedded resource limits) per the P4Runtime Protocol semantics.

Furthermore, we design these P4 programs to exhibit our desired packet forwarding behavior. Given the same table entries and configuration, the switch must forward a packet the same way that the P4 program would, *e.g.* if it was run via a simulator. In realistic pipelines, such as SAI, the forwarding behavior may include non-determinism (*e.g.* for load balancing purposes), and thus this guarantee is defined over the set of possible behaviors per packet.

We found P4 to be suitable for modeling due to several important properties. P4 programs lend themselves well to *automated validation.* They specify both the control plane API and the data planes behavior of switch, and are *unambiguous* and *machine-readable.* This becomes apparent when contrasted with traditional approaches that write specifications informally in English. Furthermore, these programs are *implementation-agnostic*, they depend on the deployment role rather than the exact switch capabilities, and thus can be reused for different switches when deployed in the same role. Additionally, these P4 programs are *living documentation* that encode the specification, the contract, and the exposed functionality all at once, and thus ensure that the implementation and validation remain in sync. Finally, they enable *rapid innovation* as new switch hardware or software features can be quickly exposed by updating the P4 program, without having to wait for the lengthy process of exposing them via newer releases of NOS, SAI, or various standards.

P4 Language Features P4 is designed for programming P4-enabled switches, rather than modeling fixed-function ones. However, we found the core language features, specifically its tables and match action pipelines, expressive enough to allow us to model our target pipelines while also being amenable to symbolic execution. Conversely, several language features, including header stacks, unions, and registers were not needed to model our target pipelines, even though they may be important for P4 programming generally.

Several P4 targets, including the BMv2 P4 simulator [45], do not allow revisiting tables in multiple locations in a pipeline. This restriction stems from practical limitations in the targeted programmable switches (*e.g.* Intel Tofino switches [27]). However, it poses a challenge when modeling certain components, such as SAI's router interfaces (RIFs), which interface with the underlying switch ports at both ingress and egress. Such components cannot be modeled as a single P4 table, since such a table could be matched on more than once (*e.g.* at both ingress and egress). Instead, they need to be modeled using workarounds, such as replicating them in several tables, which are then used in different locations. Such workarounds are merely modeling artifacts and must be accompanied by explicit constraints in the model to ensure their consistency,

e.g. the replicas in our example must have the same table entries since they correspond to the same actual component.

P4-Constraints A critical feature missing from P4 is the ability to encode semantic constraints on table entries to match the semantics of the underlying pipeline being modeled. Since it is primarily designed for P4-enabled switches, P4Runtime is a relatively permissive API that disallows syntactically invalid control plane requests, but is oblivious to semantic validity which differs between scenarios. This flexibility causes challenges in PINS, which uses this permissive protocol for programming the restrictive underlying fixed-function hardware. We mitigated this by providing mechanisms for specifying API constraints in the P4 program and enforcing these constraints at run time in PINS's P4Runtime layer.

Consider a simple ACL implementation that looks up the IPv4 or IPv6 destination addresses in a ban-list. This can be modeled in P4 as a table that matches on IPv4 and IPv6 destination addresses as well as the packet type. From the perspective of P4Runtime, this is a table with three match keys, each with no particular semantic significance. This means that P4Runtime may accept nonsensical entries, such as entries that match the IPv6 destination address of IPv4 packets and vice versa. Additionally, P4Runtime may accept entries that cannot be mapped to hardware, entries not in canonical form, and entries that would interfere with the internals of the switch being modeled. For example, in PINS, the default VRF `0` is reserved by the hardware and cannot be programmed by the controller with table entries (Figure 2 line 4).

To capture such semantic restrictions, we built `P4-constraints` [14], a P4 extension that enables us to specify custom constraints on table entries using annotations in the P4 program. These constraints are part of the contract with the controller, and we use the constraints while validating the control plane API of switches to determine the semantic validity of the generated test requests. In our experience, we needed to model two kinds of constraints: (1) (Isolated) requirements imposed by the underlying switch, such as excluding special built-in values. (2) Integrity constraints relating entries in different tables that correspond to inter-related switch components, or to the same component that is captured by multiple tables for modularity or due to other modeling artifacts.

We express the first kind of constraints via `@entry_restriction`, which can be attached to tables to restrict their entries using Boolean constraints that may refer to the keys of the table along with Boolean and relational operators. The `@refers_to` annotation allows us to encode the second kind of constraints and provides referential integrity, which essentially disallows dangling references between two tables. For example, Figure 2 encodes a common pattern where VRF IDs, which are modeled by an earlier table (`vrf_tbl`), are matched against in a later table (`ipv4_tbl`). By using `@refers_to` in line 13, we disallow `ipv4_tbl` entries that use non-existing VRF IDs, such as entry *i*2 from Figure 3. This captures a restriction of SAI, which requires that VRFs must be allocated (modeled in P4 by programming the VRF table) before they can be used. We open-sourced the P4-constraints extension, which is now a part of the P4 toolchain.

Role Specific Instantiations Developing a general model of PINS switches is undesirable. Such a hypothetical model must capture all the capabilities of the switch, even the ones that are not used in its deployed role. This would make the model overly permissive and unnecessarily complex. For example, on many ASICs, ACL can be configured to match against various combinations of packet headers and metadata. However, due to hardware (TCAM) limitations, only a few of the available headers fields can be matched on at a time. A natural way of modeling this in P4 is via a table that matches on all header fields. Table entries can then choose to match against any desired subset, and leave the remaining keys unset. However, such a model accepts table entries that match on keys beyond the capacity of the switch hardware, and is thus too permissive to use as a specification for SwitchV, or as a contract between a PINS switch and the controller.

Instead, we construct a different P4 model for each role the switch might be deployed in (*e.g.* ToR, WAN). For each of these roles, the combination of keys used for ACL is fixed and fits within hardware limits. Thus, the role-specific ACL can be directly expressed as a P4 table that matches only on that specific combination. We view these role-specific models as *"instantiations"* of the same blueprint. They have the same high-level structure as SAI. They re-use a lot of the same components and pipelines and only differ in the role-specific functionality, which currently only includes ACL. We simplify the effort required to design and maintain these instantiations by grouping all common components into a common P4 library, and instantiating from it using macros and preprocessors.

When a PINS switch is deployed in a role, we push the corresponding instantiation onto it to configure its ACL and establish its control plane API. Using the same instantiation for validation and configuration is an added benefit, but is incidental to SwitchV, which discovers deviations between a P4 model and a switch, regardless of how the model is organized, or whether the deviations stem from programmable or fixed-function components in the switch.

Bounded Internal Resources Fixed-function switches have a variety of internal mechanisms and resources that are handled by low-level components of the switch. Sometimes it makes sense to model them even if they do not affect packet-forwarding directly, since our P4 programs also aim to capture resource limits and availability. For example, in SAI, VRFs are limited resources that have to be allocated before they can be used. Therefore, we modeled VRFs in our P4 programs (Figure 2) using a table whose P4 semantics is a no-op, but whose semantics in PINS is to allocate and deallocate VRFs when entries are inserted or removed.

Hashing Switches often use hashing for load balancing purposes (*e.g.* to implement WCMP [59]). However, the exact hashing algorithm used is an internal detail that may differ across different switches and may be kept private by vendors. This makes precise modeling of the hashing algorithm challenging. Furthermore, hashing algorithms are often complex, and modeling them would result in complex models that are harder to analyze automatically. We chose to model hashing as an unspecified black box in our P4 programs, which we view as a "free" operation from a validation perspective (§5).

Recently, P4wn [29] proposed a more sophisticated treatment of hashing and other stateful constructs in forwarding pipelines for adversarial testing. Our modeling of hashing is far more primitive, and we did not need to use stateful constructs, such as registers, in our P4 models. It may be interesting to adapt some of P4wn's gray

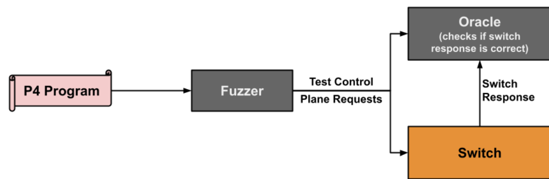

Figure 5: The design of `p4-fuzzer`.

box analysis of probabilistic and stateful structures to SwitchV in the future to support more complex stateful pipelines.

Mirror Sessions SAI provides a mirroring API that clones packets to a particular port. However, P4 provides a different clone API that expects a session ID managed by a packet replication engine. We reconcile these differences by introducing an additional logical table in our P4 programs. The P4 table that models SAI mirroring sets the target port, which is then translated to a session ID using the logical table, which gets passed to the P4 clone API. This logical table is merely a modeling artifact. It correctly models the effects of cloning on the switch but does not express how it is actually done, and does not need to be programmed by the controller.

Batching Table Entries The P4Runtime specification does not enforce any particular order when executing a batch table entry update. This allows the P4Runtime to non-deterministically accept or reject certain batches, particularly those that contain table entries that refer to each other. We rely on the aforementioned `@refers_to` annotation to detect entries that may cause ordering issues. We also use this annotation to group (§4) entries into separate batches with no ordering issues during control plane testing, and when installing the entries for data plane testing, as well as when the controller programs the switch.

Fidelity of P4 Models Our P4 programs can become moderately complex as they evolve with the switch. Our P4 models of PINS consist of 14 tables, 700 lines of role-specific and nearly 1000 lines of common P4 code. Various inaccuracies may be introduced due to human errors or misunderstanding of the desired behavior. SwitchV checks that the behavior of the switch and P4 program are identical. While we mostly use this to uncover bugs in the switch, it has also revealed 18 bugs in the models in the two validated switch stacks (§6). Complimentary P4 program verification techniques [21, 32, 36, 44] can be used in conjunction to further increase confidence in the models if needed.

4 CONTROL PLANE API VALIDATION

Control plane API validation checks that the switch correctly accepts valid and rejects invalid control plane requests from the controller given the switch's current state, and that the switch does not crash or otherwise get into an unresponsive state.

We built a fuzzer (Figure 5) for P4 to detect inconsistencies between the expected and observed control plane API of the switch. Given an input P4 program, `p4-fuzzer` generates *sequences* containing both valid and invalid table updates via fuzzing (*i.e.* semi-randomly generating entries in a directed fashion). It then uses an oracle to check that the switch handled the updates correctly.

Under-specified Behaviors The P4Runtime specification generally allows for more than one possible behavior given a request. Here we discuss two such examples.

Example 1. Consider a P4 program that specifies a match-action table T of size 10. Assume T already has 10 entries and received a control plane request to add an 11th entry. The P4Runtime specification allows the switch to accept the request and install the entry (provided sufficient resources), or reject the request (regardless of available resources).

Example 2. P4Runtime supports batch table updates. A single Write RPC may contain n updates, and the switch is free to execute these updates in any order. In theory, this gives rise to $n!$ different executions, though many may lead to the same outcome in practice.

To deal with this, our oracle does not predict a single expected outcome. Instead, the oracle observes the switch's response to ensure it belongs to the set of valid ones. Note that this requires the oracle to keep track of the switch's current state.

Valid and Invalid Requests We define a request to be *syntactically valid* if it conforms to the format specified by the P4 program and the P4Runtime specification [15]. A request is *constraint compliant* if it does not violate any user-defined constraints annotated in the P4 program using the P4-constraints extension [14]. A request is *valid* if the P4Runtime specification dictates that it *may* be accepted by the switch in some state. A request is valid if and only if it is syntactically valid and constraint compliant. Valid requests may still be rejected in some states, for example, a valid request may be rejected due to insufficient resources, or because it tries to delete a non-existent table entry. Conversely, a request is *invalid* if it *must* be rejected, *i.e.* by being syntactically invalid or not constraint compliant.

The syntactic validity of a request can be assessed by analyzing the input P4 program, which determines the appropriate request format (*e.g.* the allowed headers or actions). The P4 program also includes the constraints as annotations.

4.1 Generating Valid Requests

`p4-fuzzer` analyzes information regarding the existing tables in the input P4 program. This includes the table types and the headers and actions that they match. `p4-fuzzer` uses this information to generate control plane requests that violate no obvious rules in the P4Runtime specification. For example, `p4-fuzzer` abides by the defined bit-size of each field in the generated table entry, and selects actions from the set of permitted actions in the corresponding table definition. We currently do not enforce constraint compliance, and thus frequently generate invalid requests for tables with constraints (*e.g.* $v2$ in Figure 3). We discuss ongoing work to support this in §7.

4.2 Generating Invalid Requests

Naïvely generating invalid requests by randomly choosing values (*e.g.* random table or action IDs) produces "uninteresting" requests. Such naïve random requests are syntactically invalid with a high probability and end up exercising only the first few checks in the switch. This would leave most of the deeper and more complex control space untested.

Instead of sampling the space of requests uniformly, we use a mutation-based approach. After generating a valid control plane

request (as described above), `p4-fuzzer` applies a *mutation* randomly chosen from a specified list to produce a new request. Our mutations are based on historical analysis of control and data plane bugs and the expertise of engineers who manually debug them. Many also derive from the P4Runtime specification. These mutations produce entries that are usually "interestingly" invalid. Each invalid request is generated by applying a single mutation to a valid request, and the same valid request can be used many times to produce different invalid requests via different mutations. We give a few example mutations below.

Single Action Tables As shown in Figure 3, each table entry must consist of a valid table ID, an action ID permitted by the table, an appropriate number of arguments of appropriate sizes for the action, and at most one match field entry per each key in the table. We can generate "interesting" invalid entries by intentionally modifying a valid entry to violate any one of the above properties. For example, the *Invalid ID* mutation takes a valid entry and replaces its table, match field, or action ID with an ID that does not exist in the P4 program. *Invalid Table Action* replaces a valid action ID with an out-of-scope action. Other similar mutations include *Invalid Match Type*, *Duplicate Match Field* and *Missing Mandatory Match Field*

One-shot Action Selector Programming [15] Tables of this type allow an entry to map to a set of actions, each accompanied by a strictly positive probability weight. The *Invalid Action Selector Weight* mutation assigns a non-positive weight to an action in an action selector set. The *Invalid Table Implementation* mutation attempts to send a table entry with an action set to a single-action table and vice versa.

Other Mutations The *Invalid Reference* mutation picks a non-existing value for a field that refers to another field as specified by the `refers_to` annotation. Other mutations generate entries that refer to invalid resources (*e.g.* port or QoS queue), duplicate existing entries, or delete non-existing ones.

4.3 Oracle

We built an oracle that encodes the P4Runtime specification to determine if the switch behaves correctly. Due to under-specified behavior, there may be multiple correct output switch states and behaviors for a given request and input state. Attempting to keep track of all valid states throughout a *sequence* of requests can quickly lead to a state-explosion. Instead, our oracle issues a read to the switch to observe its actual state after a batch of requests, and then determines whether that state is valid or not (given the observed state of the switch prior to the batch). If the new state is indeed valid, our oracle can forget the prior state, and repeat the same process for the next batch of requests. This lets us process the next batch from a single state, reducing the non-determinism to the current batch of requests and a single starting state.

The oracle significantly simplifies the valid and invalid request generation process, since it allows the process to be unsound. While the P4Runtime specification is complex, the oracle's implementation is significantly smaller than the P4Runtime layer on the switch. Furthermore, SwitchV can detect bugs in the oracle implementation since such bugs likely lead to a divergence between the expected and observed switch behavior.

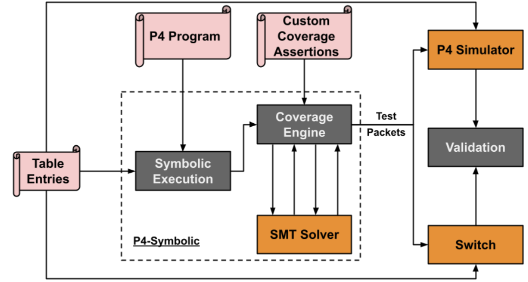

Figure 6: The design of `p4-symbolic`.

4.4 Running Test Requests

We generate many (*e.g.* few thousands) valid and invalid requests using the above procedure. We then group these requests into several batches. We send the batches to the switch sequentially. The switch may process the requests inside a single batch in parallel and in any order. Therefore, the batches must only include independent requests whose validity does not depend on the order of execution.

We ensure this by automatically analyzing the `@refers_to` annotations in our P4 model, which encode any dependencies between table entries. We use this information to sequence any dependent requests to different batches. Note that for valid requests, we only generate ones that depend on previously installed entries or have no dependencies. Our *Invalid Reference* mutation generates invalid requests that refer to and thus depend on non-existent entries, to test whether the switch correctly rejects such requests.

5 DATA PLANE VALIDATION: P4-SYMBOLIC

Overview `p4-symbolic` validates the packet-forwarding behavior of a switch by generating a set of test packets. SwitchV runs the test packets against the switch and the BMv2 P4 simulator, and monitors their behavior. Mismatches indicate a potential issue in either the switch, model, or simulator. Testers provide `p4-symbolic` with three inputs: (1) the P4 program that models the switch's expected behavior, (2) the table entries used to configure both the switch and simulator, and (3) coverage assertions describing the minimum coverage guarantees the generated test packets must meet.

Symbolic Execution `p4-symbolic` maintains a symbolic state $\mathcal{S}$ that maps the header and metadata fields from the P4 program to their corresponding symbolic values at the current state of execution. Additionally, `p4-symbolic` builds a symbolic trace that maps each control flow construct in the program to a symbolic expression which evaluates to true iff the construct is executed.

Initially, the symbolic trace is empty as the program is not executed yet, and the symbolic state consists of a mapping of each packet header and metadata field to a unique unconstrained symbolic variable. We denote the set of such variables by $\mathcal{X}$. A concrete input test packet is an assignment of concrete values to the variables in $\mathcal{X}$. As the program is symbolically analyzed, the symbolic state and trace are mutated with the effects of the analyzed expressions.

At the end of the symbolic execution, the symbolic state $\mathcal{S}$ maps each field of the output packet header to a symbolic expression over variables from $\mathcal{X}$. We denote these output symbolic expressions by $\mathcal{Y}$. The symbolic trace now similarly maps every control flow

construct, such as a specific branch or table entry, to a Boolean symbolic expression over $\mathcal{X}$. We denote this complete trace by $\mathcal{T}$.

Coverage Constraints After symbolic execution is completed, p4-symbolic iteratively poses several coverage constraints over $\mathcal{X}, \mathcal{Y}$, and $\mathcal{T}$. Each constraint asserts a certain desired property about the input packet, output packet, or the execution trace. This ensures that generated packets collectively meet the desired coverage requirements. Each constraint is passed to our backend SMT solver, Z3 [17], which produces a satisfying test packet if the constraint is satisfiable.

If p4-symbolic is configured to maximize branch coverage, this step produces a sequence of $|\mathcal{T}|$ constraints, each asserting that a unique control flow construct (*e.g.* a branch or table entry) is executed or matched. Alternatively, maximizing trace coverage requires an assertion per each possible combination of program branches and entries. As the number of control constructs in the program increases, the number of such combinations (and thus assertions) increases combinatorically. This makes trace coverage impractical for complex P4 programs. To alleviate this, p4-symbolic exposes $\mathcal{X}, \mathcal{Y}$, and $\mathcal{T}$ to test engineers so they can ensure coverage of a selected subset of important traces, allowing for a practical middle ground between branch and trace coverage.

Decidability Control blocks in P4 encode single-pass forwarding pipelines. Tables cannot be reused, and there are no mechanisms for iteration and recursion. Furthermore, p4-symbolic generates SMT formulas that are quantifier-free. The SMT constraints generated by p4-symbolic for typical programs only use the theories of bitvectors and equality, which are decidable. Replacing bitvectors with unbounded integers, it may be possible to build *contrived* examples of P4 programs and coverage assertions that require undecidable theories (*e.g.* Peano arithmetic). Even then, Z3 is mature and capable of solving many instances of such theories in a reasonable time using its built-in heuristics.

Limitations We do not support certain P4 constructs that we did not use in our P4 programs (header stacks, unions, and named calculations). To reduce the implementation effort, we deprioritized the support for generic P4 parsers. Instead, we rely on semi-hardcoded support for parser patterns of interest. Supporting a generic parser is mainly an engineering task that we leave as future work.

Hashing P4 programs may compute hashes over packet header fields and other metadata for purposes such as load-balancing. The exact hashing algorithm used by the switch is often unknown and may differ across switches. p4-symbolic interprets the hash as a free operation: The output of the hash is allowed to be any value, even in cases where these values are outside the range of the concrete hash. We rely on constraints set further down the symbolic execution to restrict the values the hash can take, for example when the value of the hash is used in a table match. To judge the correctness of the switch, SwitchV configures the P4 simulator to use round-robin hashing, and runs the test packet through it several times (*i.e.* until the same behavior occurs twice) to build the set of all possible behaviors, and then checks that it includes the observed switch behavior.

Trace Isolation In regular execution, only one branch of a conditional expression is executed, and the remaining ones are ignored. However, this is not the case during symbolic execution where all branches are analyzed. This poses a challenge: We must ensure that the symbolic side effects from all such sibling branches are isolated.

A common approach (*e.g.* in KLEE [8]) is to symbolically execute each trace in the program in isolation, with a completely separate state. This guarantees isolation of side effects. However, the number of traces in P4 programs is prohibitive: while a P4 program typically includes a handful of *explicit* conditionals (*e.g.* if statements), the table entries (the number of which in our experience can be in the order of several hundreds) constitute an *implicit* form of branches that can quickly blow-up the number of traces. For example, a simple flow with three consecutive tables, with 100 entries each, would result in $100^3 = 1,000,000$ traces.

Existing work that symbolically executes P4 programs (*e.g.* [43]) avoids this by reducing the number of table entries, which negatively impacts the coverage. Instead, p4-symbolic performs only a single pass over the program, executing branches against the same symbolic state. We ensure isolation by encoding the *context* of the branch as a logical guard that is applied to all consequent side effects. Guarded assignments, and more generally guarded commands, are a well-known technique dating back to the seminal work of Dijkstra [18]. We adapt this technique to P4, where guards are deduced from branches and control flow statements, but also from table entries and action matches.

Example Consider the P4 program from Figure 2. Assume the table entries $v1$, $i1$, and $i5$ from Figure 3 are passed as inputs to p4-symbolic. We focus on p4-symbolic as it reaches the application of the ipv4_tbl table (line 23). At this point, the symbolic state $\mathcal{S}$ maps headers and metadata fields to their current symbolic values, e.g. $\mathcal{S} := \{ipv4.isValid \rightarrow x_{ipv4},\ ipv4.dst_addr \rightarrow x_{dst_addr},\ vrf_id \rightarrow x_{vrf_id}\}$. The current context C captures all the constraints for the program to reach this point of execution, *e.g.* because the execution is inside the body of a conditional (line 22), its corresponding condition ($x_{ipv4} = true$) is reflected in C.

p4-symbolic iterates over the entries in ipv4_tbl in descending order of priority (longest prefix match in this case). In each step, it maps the corresponding entry e to an expression $\mathcal{T}[e]$ capturing the condition in which the entry gets matched. The expression is the conjunction of the current context, the constraints on the current values of the header and metadata fields for the entry to match, and the negation of match conditions of higher priority entries. For instance, $\mathcal{T}[i5] := C \wedge (x_{vrf_id} = 1) \wedge match(x_{dst_addr}, 10.0.*.*)$ and $\mathcal{T}[i1] := C \wedge (x_{vrf_id} = 1) \wedge match(x_{dst_addr}, 10.*.*.*) \wedge \neg((x_{vrf_id} = 1) \wedge match(x_{dst_addr}, 10.0.*.*))$. The last conjunct in $\mathcal{T}[i1]$ (the negation) ensures the higher priority (longer prefix) entry $i5$ does not match.

After handling matching on the table entries, p4-symbolic handles the actions they invoke. Assume that the set_nexthop_id action sets the metadata field *nexthop_id* to the argument passed to that action. Thus, p4-symbolic must set *nexthop_id* to 10 or 3 if either $i5$ or $i1$ is matched, respectively. For trace isolation, p4-symbolic *guards* this assignment by the condition that expresses when each entry is matched, which gives $\mathcal{S}[nexthop_id] := if\ \mathcal{T}[i5]\ then\ 10\ else\ (if\ \mathcal{T}[i1]\ then\ 3\ else\ v)$, where v is the old value prior to the match. At the end of symbolic execution, the current symbolic state $\mathcal{S}$ becomes $\mathcal{Y}$, *e.g.* $\mathcal{Y}$ maps *nexthop_id* to

the value above. For unmodified fields, $\mathcal{Y}$ maintains their initial values, *e.g.* $\mathcal{Y}[ipv4.isValid] = x_{ipv4}$.

After the symbolic execution, we pose our desired coverage constraints. For example, to produce a packet that matches on $i1$, we assert that $\mathcal{T}[i1]$ must evaluate to true. We then ask the SMT solver to produce a concrete assignment for the variables and expressions in $\mathcal{X}$ and $\mathcal{Y}$ that satisfy our assertions (if such an assignment exists), from which we extract the test packet. In our example, $x_{ipv4} := true, x_{dst_addr} := 10.1.0.0$ satisfies the assertion.

6 SWITCHV IN PRACTICE

We share our findings using SwitchV to validate two switch stacks, Cerberus and PINS, during their development. Both projects involved extending existing open-source switch software stacks with limited programmability on top of the underlying ASIC, and exposing these programmable features using a P4Runtime-based API.

We developed SwitchV in parallel to these projects. The projects were carried out by the same team at Google. Cerberus began first and was not completed; PINS has been in development for two years and remains ongoing at the time of writing. Both projects used different versions of SwitchV, with Cerberus using `p4-fuzzer` and `p4-symbolic` for 10 and 12 months, and PINS using them for 21 and 26 months. SwitchV detected a total of 122 and 32 bugs in these projects respectively.

The PINS project involves many engineers inside and outside of Google and comprises more than 2.5 million lines of code, which amounts to more than 10 million lines of code when combined with the pre-existing layers on a switch. In contrast, `p4-symbolic` and `p4-fuzzer` are about 5k and 3.5k lines of code, respectively, with no more than three engineers and a few interns working on SwitchV at any point in time.

SONIC, the open-source stack on which PINS is based, is mature and used in production. However, the P4Runtime server added by PINS is new and under development, and required changes in all SONIC layers (Figure 4). The underlying stack in Cerberus was being developed by a vendor. The switch vendor was internally performing traditional testing for most features, although we did use SwitchV to test a few features in parallel or prior to the vendor. Compared to PINS, the P4 programs used in Cerberus were more complex, with more involved forwarding pipelines and additional features such as encapsulation and decapsulation.

During PINS's development, we used the following testing procedure: (1) Unit tests; (2) Component tests; and (3) Switch end-to-end validation using SwitchV (nightly test). Recently, PINS successfully underwent the so-called *Design Verification Testing* (DVT) phase, in which a data center fabric containing PINS-based switches was tested end-to-end using more traditional means. DVT did not uncover any bugs in the switches' control plane API or forwarding behavior, which may be explained by SwitchV discovering 122 bugs pre-DVT—we believe this is a strong testament to the effectiveness and impact of P4-based automated validation. With that said, we have not yet completed our in-production testing for this switch, and thus we cannot be certain that SwitchV found all of the testable bugs.

PINS Component	Bugs	`p4-fuzzer`	`p4-symbolic`
P4Runtime Server	47	11	36
gNMI	2	0	2
Orchestration Agent	24	12	11
SyncD Binary	23	10	13
Switch Linux	9	0	9
Hardware	1	1	0
P4 Toolchain	2	1	1
Input P4 Program	15	2	13
Total	122	37	85

Cerberus Component	Bugs	`p4-fuzzer`	`p4-symbolic`
Switch software	24	14	10
Hardware	1	0	1
Input P4 Program	3	0	3
BMv2 P4 Simulator	4	4	0
Total	32	18	14

Table 1: Bugs found by SwitchV by component.

6.1 Detected Bugs

Our PINS-based switch stack is shown in Figure 4. On top is the application layer, which includes the P4Runtime server that implements the P4Runtime protocol for communication with external SDN controllers. It is possible to have other top-level applications running concurrently at this layer. Below is the Orchestration Agent that synchronizes the state of the top-level applications and applies it to the hardware via the interface provided by SyncD. Below, SyncD builds on top of SAI to provide a vendor and hardware agnostic database interface to the ASIC. These layers constitute PINS, and run on top of the switch's hardware and our version of Linux, which includes various switch-related daemons. Due to the limited visibility we had into the stack in Cerberus, which was being developed by an external vendor, its bugs are categorized more coarsely.

As shown in Table 1, SwitchV found bugs across the whole stack. While a plurality of these bugs were found in the new parts of the stacks under development (the P4Runtime and Orchestration Agent), some were found in pre-existing code and components, primarily because they were used in new ways. Additionally, SwitchV found bugs in the P4 programs. These bugs manifested as differences between the observed behavior of the switch (specifically the ASIC behavior) and the P4 program. Upon inspection, we determined that the switch was actually correct, and the P4 programs did not encode our desired specifications correctly. Finally, SwitchV detected bugs in the P4 toolchain, including in the BMv2 P4 simulator and the P4-PDPI [24] framework.

We discuss a few of the more interesting examples to give a flavor of the kinds of bugs SwitchV can find.

In PINS, an application in the *P4Runtime server* was accidentally sending certain out packets back to the controller via the packet-in [15] mechanism. This was identified via `p4-symbolic`. The *P4Runtime server* would get into an inconsistent state after receiving certain sequences of L3 table entry deletions. This was identified by `p4-fuzzer`. The *SyncD binary* did not support default

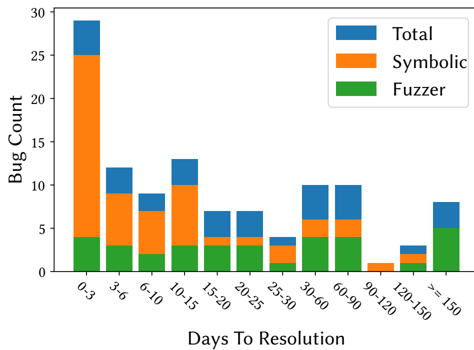

Figure 7: Number of days required to resolve bugs in PINS by SwitchV component. 9 bugs have not been resolved.

route deletion while other routes were present in the same VRF. This was identified using production table entries to setup `p4-symbolic` tests. The *Switch Linux* was running a traditional LLDP networking application, which was interfering with our SDN controller's LLDP application. This was identified by `p4-symbolic`, which detects when unexpected packets get punted to the controller.

In Cerberus, `p4-symbolic` identified that the *switch software* reversed the destination IP address used for packet encapsulation, because of an issue with endianness. The *hardware* dropped packets on a port with a certain port speed due to electric interference. This bug was detected by `p4-symbolic` in a pre-production unit, and was independently detected and fixed by the switch manufacturer.

We changed the underlying chip used in PINS past the mid-point during development. After the change, we noticed a resurgence of bugs in components that had been successfully validated before, including software components and even our P4 program. These bugs were the result of variations in the exact contract provided by the new chip compared to the old one. For example, the new chip has a built-in fixed-function trap that did not exist in the old switch which immediately punts packets with TTL 0 or 1, even when our controller/P4Runtime extension says otherwise.

Some of the bugs do not fit exactly into a single component. Rather, they are symptoms of a larger design issue throughout many components. For example, the new chip imposes restrictions on entries in certain tables that are incompatible with desired ACL behaviors, and with the design decisions that higher software layers made to support these behaviors. We found that 33% of the bugs identified in PINS were integration bugs resulting from a misunderstanding of the contract between two components.

The majority of bugs encountered in PINS were fixed within 14 days, with 33% of bugs fixed within 5 days. This indicates that most of the bugs found by SwitchV were important enough for developers to fix quickly. This is in contrast to other automated techniques we used in the past, where tools would find bugs that developers would not deem important enough to act on. Anecdotally, we filed issues for 3 bugs independent from SwitchV related to bad error messages, and their mean resolution time was much worse, at 66 days. We reported 9 bugs that remain unresolved as of this writing.

Test	PINS	Cerberus
Set P4Info	22 (18%)	0 (0%)
Table entry programming	15 (12%)	0 (0%)
Read all tables	10 (8%)	2 (6%)
Packet-in	12 (10%)	4 (13%)
Packet-out	4 (3%)	1 (3%)
Packet forwarding	0 (0%)	0 (0%)
Not found by any test above	60 (49%)	25 (78%)

Table 2: Which bugs could be found using the trivial test suite

SwitchV reports inconsistencies between the switch and the P4 model but leaves it up to the testers to identify the source of the inconsistency. The number of days until bug resolution, shown in Figure 7, includes the time to identify that source, as well as to apply any relevant fixes. In our experience, most of the reported time was spent either on fixing the bug or on the issue waiting in the backlog pending developers' availability.

6.2 Bug Complexity

While a vast majority of bugs found by SwitchV were deemed important by developers, many of them are simple bugs that might have been detected by simpler alternative forms of testing. To evaluate how many bugs detected by SwitchV are harder to detect via traditional means, we devised the following trivial suite of traditional integration tests:

1. **Set P4Info:** Push the P4Info configuration to the switch.
2. **Table entry programming:** Install a rule in every table, including an ACL entry that punts packets to the controller and an IPv4 route.
3. **Read all tables:** Read back all tables and compare with the set of entries installed earlier.
4. **Packet-in:** Send a packet that matches the previously installed ACL punt rule and check that the correct packet gets received on the packet-io channel.
5. **Packet-out:** Send a packet via packet-out for each port, and ensure the switch sends it out through those ports in the dataplane.
6. **Packet forwarding:** Send an IPv4 packet to the switch and check that it gets forwarded correctly according to the IPv4 route installed earlier.

The tests are meant to be executed in sequence. Table 2 shows the percentage of bugs that would be found by each sub-sequence of tests, excluding bugs that would already be found by earlier tests in that sequence. We observe that about 49% of the bugs from PINS would have been found by the trivial test suite. Some of the bugs not found by our trivial suite may still be found using other reasonable test suites. Additionally, 67% of the identified bugs are restricted to a single component. Many such bugs can be detected using better unit and component tests. We have not used any other kind of integration testing in PINS, and instead relied on SwitchV to find many of the trivial bugs, since it was easier to deploy and use than manually designing test cases.

In Cerberus, we estimate that 78% of the bugs cannot be found by our trivial test suite. This is expected since most of the simpler bugs in Cerberus would have been detected by the vendor during

P4 Prog.	Entries	Generation (w/c)	Testing
Inst1	798	413s (14s)	58s
Inst2	1314	1099s (6s)	64s

P4 Prog.	Fuzzed Entries	Entries/s
Inst1	50384	97
Inst2	48521	96

Table 3: Time required to run `p4-symbolic` (top table) and `p4-fuzzer` (bottom table) for two production P4 programs. Time with caching enabled is reported in parentheses.

their testing. Some simple bugs slipped through, partly because we sometimes performed testing with SwitchV in parallel to the vendor. Additionally, this project exhibited more complex bugs because its forwarding pipeline is more complex and feature-rich.

6.3 Performance of SwitchV

Table 3 shows the performance of `p4-symbolic` and `p4-fuzzer` on two different production P4 programs. We ran each experiment on a single virtual CPU core in a containerized environment. We use `p4-symbolic` to hit every reachable input table entry at least once (*i.e.* branch coverage). The generation column measures the time taken for this step (with and without a cache), while the testing column shows the time needed to run the generated packets through the switch and BMv2, and compare their behavior. We configure `p4-fuzzer` to generate 1000 write requests with approximately 50 table entry updates each. We have found this to be sufficient to catch a wide variety of bugs when run daily. We have not focused on optimizing performance, beyond caching for `p4-symbolic`, as the current performance is acceptable for our purposes.

Caching The slowest stage in SwitchV is generating the test packets by repeatedly invoking Z3 to solve the SMT constraints produced by `p4-symbolic`. If the input P4 program and table entries are unchanged from previous runs, or their changes do not affect the SMT constraints, we simply lookup the test packets from a cache. Thus, we only need to run the expensive generation stage when the specifications have changes that affect the SMT formula, which is less frequent than updating the switch stack under validation.

7 DISCUSSION

P4 as a Specification Language Our experience demonstrates that P4 can successfully model the behavior of switches, including fixed-function ones. Many of the challenges we encountered while modeling SAI in P4 stem from the flexibility (and thus permissiveness) of P4 and the P4Runtime Protocol compared to the restrictive and fixed semantics of the underlying switches and SAI. These challenges offer important insights that can help design the next generation of modeling and specification tools for networking. Indeed, we developed several P4 extensions [14, 24] to overcome some of these challenges, and up-streamed them as standalone open-source modules integrated into the P4 toolchain.

P4 programs provide a balance between the low-level detail required to capture the correctness of individual switches and the simplicity and formalism required for effective automated analysis. Using P4 programs as specifications allows SwitchV to validate the *control plane API* of a switch and its *data plane* behavior. This allows SwitchV to detect bugs that would not be observed using only data plane validation. Furthermore, bugs caused by errors in the control plane API (*e.g.* during installation of table entries) are detected earlier, rather than during packet forwarding where the root cause is more removed from where the bug occured (*e.g.* packets being forwarded via the wrong port because a table entry was installed incorrectly). Currently, table entries generated using fuzzing are only used to validate the control plane API. A potential future extension is to also pass these entries to `p4-symbolic` and use them with the generated test packets, which can exercise additional control paths during data plane validation.

In other domains, testing tools often rely on "throw-away" specifications whose sole purpose is validation. These specifications can be complex and often use domain-specific modeling languages. Thus, updating and maintaining these specifications as the system they describe evolves can require significant effort, and may lead to them going out of sync, even when validation is automated. This is a common problem that extends beyond switch validation to other domains ranging from maintenance of software test cases [41] to formal proofs [51]. By contrast, our P4 specifications are multi-purpose and "organic" to the networking ecosystem, and thus are more likely to be always up-to-date. They are the primary interface that expose new features in PINS, they define the contract with the controller, and they provide a living documentation that engineers can consult, and have additional incentives to continuously maintain. The effectiveness of our P4 models as documentation depends on the familiarity of the readers with P4, and we found that many developers prefer them to having to read thousands of lines of low-level C implementation.

Fuzzing `p4-fuzzer` relies on mutations to generate valid and invalid control plane requests. This is a common technique that has been used extensively for fuzzing [6, 10, 46]. Unlike state-of-the-art general purpose smart fuzzing tools [52, 57], our fuzzer is intentionally specialized to our target domain with manually curated mutations based on the expertise of network engineers. This allows us to find *interesting* invalid requests, whose space is much smaller than the entire space of invalid requests, while also minimizing the engineering effort to build and maintain `p4-fuzzer`.

While our experience demonstrates that our fuzzer is capable of detecting interesting bugs throughout the switch stack, our approach does not provide provable formal coverage guarantees. This is a direction for future research, that may benefit from using smart fuzzing mechanisms, including coverage-based fuzzers, provided that they can be succinctly tuned to the idiosyncrasies of the P4Runtime protocol (*e.g.* using domain-specific testing goals [47]).

`p4-fuzzer` may benefit from adding more mutations that increase the complexity of the generated invalid requests. One possibility is to use techniques that deduce mutations algorithmically [35] or via learning [23, 54]. Furthermore, we are currently implementing an explicit mutation for reasoning about P4-constraints based on binary decision diagrams (BDD). The mechanism transforms every constraint in the P4 program into a BDD over the bits of the header and metadata fields referred to in that constraint. We can efficiently sample solutions to this BDD to ensure that our valid

tests are constraint-compliant, and randomly mutate one of the nodes of the BDD to generate (otherwise valid) table entries that violate the corresponding constraint.

We considered alternative designs for control plane API validation that do not rely on fuzzing. While we can easily generate syntactically valid table entries with an SMT solver, it is challenging to generate "*interesting*" invalid ones for black box implementations. Instead, we would need to use program analysis techniques to analyze the implementation (specifically, PINS's P4Runtime layer), and find invalid entries that exercise different "deep" control paths within that implementation. This may offer some advantages as it can provide more precise coverage guarantees, similar to `p4-symbolic`. However, it requires dealing with implementations far more complex than the P4 programs that `p4-symbolic` analyzes, both in terms of size and friendliness to automated analysis (the switch's implementation is in C rather than P4). Furthermore, the implementation may frequently change, which may require updating the automated analysis, *e.g.* in case unsupported features were used. While existing tools have had some success with program analysis of complex system implementations (*e.g.* KLEE [8]), we opted to rely on mutation-based fuzzing as it provides a simpler mechanism for generating useful invalid entries with black box implementations.

Development Processes Using SwitchV We believe SwitchV is best suited to be run periodically and frequently (*e.g.* nightly) during the development of switch stacks. This allows SwitchV to detects issues, including complex integration bugs, earlier in the development cycle. Additionally, developers get quick feedback after their changes and can correct mistakes quickly. Furthermore, SwitchV can be used earlier and more frequently than more expensive forms of testing, such as DVT or fabric testing, which are difficult to run on incomplete stacks. We do not recommend replacing fabric testing with SwitchV, rather we recommend running SwitchV frequently prior to fabric testing to shorten the development cycle, and running fabric and in-production testing normally after specific milestones are achieved.

We track the progress of larger projects at Google by defining milestones in terms of objectives and key results (OKRs). SwitchV provides a methodical way of tracking the state of components of the switch stack, and we found that it provides a natural set of metrics to measure the progress towards completing an OKR for some feature F. For example, the percentage of fuzzed table entries related to F that are correctly handled by the switch, or the percentage of table entries related to F that produce correct output packets when hit by tests packets using `p4-symbolic`.

8 RELATED WORK

Automatic test generation is an established technique shown to be an effective alternative to manual testing in various domains, including system programs [8], circuits [11], enterprise applications [53], and GUI applications [42]. P4pktgen [43] and ATPG [58] adapt this technique for validating the *data plane* of a switch, but **not** its *control plane API*.

P4pktgen P4pktgen uses symbolic execution to analyze P4 programs and generate test packets. P4pktgen does not take table entries as inputs. Instead, it generates at most a single entry per table along side the test packet. Thus, the generated packets have lower coverage of the control paths in the switch stack. For example, P4pktgen cannot detect bugs in the switch's implementation of priority or longest prefix matching, since such bugs cannot be observed without at least two *correlated* entries in the same table.

ATPG ATPG operates on the switch's configuration files and FIBs, and uses them to build a model of the entire network. This model essentially views a switch as a single match table (*i.e.* sequence of rules) that directly match and produce packets. ATPG analyzes the model and generates test packets that exercise various links or rules in the network, and test its performance under different loads. ATPG's abstract view of a switch cannot represent lower-level switch functionalities, such as behavior that depends on the switch state (*e.g.* NAT), non-determinism (*e.g.* WCMP), and punting. However, this level of abstractions allows ATPG to effectively reason about network-wide properties (*e.g.* reachability, liveness) and performance (*e.g.* congestion).

Header Space Analysis HSA [30] is a well-established technique for analyzing packet forwarding behavior, and is used by many tools (including ATPG) as a foundation for their data plane analysis. Similarly, HSA could be used for data plane validation in SwitchV rather than symbolic execution with an SMT solver. We chose our particular `p4-symbolic` design because of the availability, flexibility, and ease of use of off-the-shelf SMT solvers, and our familiarity with them. This worked well for our use cases at Google as shown by our empirical results, Our main insights and contributions surround the use of P4 for modeling, which can also be compatible with HSA and other data plane testing approaches.

Validation vs Verification While the last decade has seen exciting progress on network and P4 data plane verification [1, 9, 21, 22, 25, 26, 31, 33, 34, 36, 38, 44, 48], these tools solve a different and orthogonal problem to SwitchV. The crucial difference is that such tools never analyze the actual switch or network, but only its configuration. Thus, they can find bugs in the configuration, but not in the switch. In contrast, SwitchV finds bugs in the switch, but not in its configuration. Formal techniques have been used to verify hardware designs [3, 12], including switch hardware [37]. However, such techniques cannot be used to verify black box switches, require significant device-specific modeling, as well as verification overhead and expertise, and cannot reason about non-hardware layers in the switch stack. Verification and validation can be combined to reap the benefits of both. Specifically, P4 verification tools (*e.g.* p4v [36] or ASSERT-P4 [21]) can be used to verify that our P4 models indeed encode our desired correctness properties, to increase our faith in the fidelity of these models and the validation process as a whole.

ACKNOWLEDGMENTS

We thank the SwitchInfra team at Google for their patience and support in pursuing this novel approach to switch validation, and for root causing numerous bugs identified by SwitchV, at times including false positives. We thank Waqar Mohsin for supporting us in publishing this work, and Jeff Mogul and Colin Scott for suggesting many improvements to the paper. We are also grateful to Malte Schwarzkopf, our shepherd Ang Chen, and five anonymous SIGCOMM reviewers for their help in improving the paper. This work was supported in part by NSF grant CNS-2107078.

REFERENCES

[1] Saswat Anand, Edmund K. Burke, Tsong Yueh Chen, John Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, and Phil Mcminn. 2013. An Orchestrated Survey of Methodologies for Automated Software Test Case Generation. *J. Syst. Softw.* 86, 8 (Aug. 2013), 1978–2001. https://doi.org/10.1016/j.jss.2013.02.061

[2] John Backes, Pauline Bolignano, Byron Cook, Andrew Gacek, Kasper Soe Luckow, Neha Rungta, Martin Schaef, Cole Schlesinger, Rima Tanash, Carsten Varming, et al. 2019. One-click formal methods. *IEEE Software* 36, 6 (2019), 61–65.

[3] A. Biere, T. van Dijk, and K. Heljanko. 2017. Hardware model checking competition 2017. In *2017 Formal Methods in Computer Aided Design (FMCAD)*. 9–9. https://doi.org/10.23919/FMCAD.2017.8102233

[4] Nikolaj Bjørner and Karthick Jayaraman. 2015. Checking cloud contracts in Microsoft Azure. In *International Conference on Distributed Computing and Internet Technology*. Springer, 21–32.

[5] 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. *ACM SIGCOMM Computer Communication Review* 44, 3 (2014), 87–95.

[6] Thomas Braibant, Jonathan Protzenko, and Gabriel Scherer. 2014. ArtiCheck: well-typed generic fuzzing for module interfaces. http://gallium.inria.fr/ scherer/doc/articheck-long.pdf. (2014). Accessed September 5, 2021.

[7] Pietro Bressana, Noa Zilberman, and Robert Soulé. 2020. Finding Hard-to-Find Data Plane Bugs with a PTA. In *Proceedings of the 16th International Conference on Emerging Networking EXperiments and Technologies (CoNEXT '20)*. Association for Computing Machinery, New York, NY, USA, 218–231. https://doi.org/10.1145/3386367.3431313

[8] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. USENIX Association, USA, 209–224.

[9] Marco Canini, Daniele Venzano, Peter Perešíni, Dejan Kostić, and Jennifer Rexford. 2012. A NICE Way to Test Openflow Applications. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12)*. USENIX Association, USA, 10.

[10] Sang Kil Cha, Maverick Woo, and David Brumley. 2015. Program-Adaptive Mutational Fuzzing. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (SP '15)*. IEEE Computer Society, USA, 725–741. https://doi.org/10.1109/SP.2015.50

[11] W.-T. Cheng and T.J. Chakraborty. 1989. Gentest: an automatic test-generation system for sequential circuits. *Computer* 22, 4 (1989), 43–49. https://doi.org/10.1109/2.25381

[12] Joonwon Choi, Muralidaran Vijayaraghavan, Benjamin Sherman, Adam Chlipala, and Arvind. 2017. Kami: A Platform for High-Level Parametric Hardware Specification and Its Modular Verification. *Proc. ACM Program. Lang.* 1, ICFP, Article 24 (Aug. 2017), 30 pages. https://doi.org/10.1145/3110268

[13] Sean Choi, Boris Burkov, Alex Eckert, Tian Fang, Saman Kazemkhani, Rob Sherwood, Ying Zhang, and Hongyi Zeng. 2018. FBOSS: Building Switch Software at Scale. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '18)*. Association for Computing Machinery, New York, NY, USA, 342–356. https://doi.org/10.1145/3230543.3230546

[14] P4 community. 2020. P4 Constraints. https://github.com/p4lang/p4-constraints. (2020). Accessed December 20, 2020.

[15] P4 community. 2020. P4 Runtime specification. https://p4.org/p4runtime/spec/v1.2.0/P4Runtime-Spec.html. (2020). Accessed January 13, 2021.

[16] The P4 Language Consortium. 2017. $P4_{16}$ Language Specification. https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html. (2017). Accessed January 19, 2022.

[17] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340.

[18] Edsger W Dijkstra. 1975. Guarded commands, non-determinacy and a calculus for the derivation of programs. *ACM SIGPLAN Notices* 10, 6 (1975), 2–2.

[19] Andrew D Ferguson, Steve Gribble, Chi-Yao Hong, Charles Edwin Killian, Waqar Mohsin, Henrik Muehe, Joon Ong, Leon Poutievski, Arjun Singh, Lorenzo Vicisano, et al. 2021. Orion: Google's Software-Defined Networking Control Plane.. In *NSDI*. 83–98.

[20] Open Networking Foundation. 2021. P4 Integrated Network Stack (PINS). https://opennetworking.org/pins/. (2021). Accessed September 5, 2021.

[21] Lucas Freire, Miguel Neves, Lucas Leal, Kirill Levchenko, Alberto Schaeffer-Filho, and Marinho Barcellos. 2018. Uncovering Bugs in P4 Programs with Assertion-Based Verification. In *Proceedings of the Symposium on SDN Research (SOSR '18)*. Association for Computing Machinery, New York, NY, USA, Article 4, 7 pages. https://doi.org/10.1145/3185467.3185499

[22] Timon Gehr, Sasa Misailovic, Petar Tsankov, Laurent Vanbever, Pascal Wiesmann, and Martin Vechev. 2018. Bayonet: Probabilistic Inference for Networks. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 586–602. https://doi.org/10.1145/3192366.3192400

[23] Patrice Godefroid, Hila Peleg, and Rishabh Singh. 2017. Learn amp;Fuzz: Machine learning for input fuzzing. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 50–59. https://doi.org/10.1109/ASE.2017.8115618

[24] Google. 2020. P4 PDPI: Program Dependent Intermediate Representation. https://github.com/google/p4-pdpi. (2020). Accessed January 27, 2021.

[25] Alex Horn, Ali Kheradmand, and Mukul R. Prasad. 2017. Delta-net: Real-time Network Verification Using Atoms. In *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017*. USENIX Association, 735–749. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/horn-alex

[26] Alex Horn, Ali Kheradmand, and Mukul R. Prasad. 2019. A Precise and Expressive Lattice-theoretical Framework for Efficient Network Verification. In *27th IEEE International Conference on Network Protocols, ICNP 2019, Chicago, IL, USA, October 8-10, 2019*. IEEE, 1–12. https://doi.org/10.1109/ICNP.2019.8888144

[27] Intel. 2022. Intel® Tofino™ Series Programmable Ethernet Switch ASIC. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html. (2022). accessed Jun, 19 2022.

[28] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, et al. 2013. B4: Experience with a globally-deployed software defined WAN. *ACM SIGCOMM Computer Communication Review* 43, 4 (2013), 3–14.

[29] Qiao Kang, Jiarong Xing, Yiming Qiu, and Ang Chen. 2021. Probabilistic Profiling of Stateful Data Planes for Adversarial Testing. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2021)*. Association for Computing Machinery, New York, NY, USA, 286–301. https://doi.org/10.1145/3445814.3446764

[30] Peyman Kazemian, George Varghese, and Nick McKeown. 2012. Header Space Analysis: Static Checking for Networks. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12)*. USENIX Association, USA, 9.

[31] Ali Kheradmand. 2020. Automatic Inference of High-Level Network Intents by Mining Forwarding Patterns. In *SOSR '20: Symposium on SDN Research, San Jose, CA, USA, March 3, 2020*. ACM, 27–33. https://doi.org/10.1145/3373360.3380831

[32] Ali Kheradmand and Grigore Rosu. 2018. P4K: A Formal Semantics of P4 and Applications. *CoRR* abs/1804.01468 (2018). arXiv:1804.01468 http://arxiv.org/abs/1804.01468

[33] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and Philip Brighten Godfrey. 2013. VeriFlow: Verifying Network-Wide Invariants in Real Time. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, Lombard, IL, USA, April 2-5, 2013*. USENIX Association, 15–27. https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/khurshid

[34] Dexter Kozen. 2014. NetKAT — A Formal System for the Verification of Networks. In *Programming Languages and Systems*, Jacques Garrigue (Ed.). Springer International Publishing, Cham, 1–18.

[35] Caroline Lemieux and Koushik Sen. 2018. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 475–485.

[36] Jed Liu, William Hallahan, Cole Schlesinger, Milad Sharif, Jeongkeun Lee, Robert Soulé, Han Wang, Călin Caşcaval, Nick McKeown, and Nate Foster. 2018. P4v: Practical Verification for Programmable Data Planes. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '18)*. Association for Computing Machinery, New York, NY, USA, 490–503. https://doi.org/10.1145/3230543.3230582

[37] Yuan Lu and Mike Jorda. 2004. Verifying a Gigabit Ethernet Switch Using SMV. In *Proceedings of the 41st Annual Design Automation Conference (DAC '04)*. Association for Computing Machinery, New York, NY, USA, 230–233. https://doi.org/10.1145/996566.996631

[38] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P. Brighten Godfrey, and Samuel Talmadge King. 2011. Debugging the Data Plane with Anteater. *SIGCOMM Comput. Commun. Rev.* 41, 4 (Aug. 2011), 290–301. https://doi.org/10.1145/2043164.2018470

[39] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. 2008. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Comput. Commun. Rev.* 38, 2 (March 2008), 69–74. https://doi.org/10.1145/1355734.1355746

[40] Justin Meza, Tianyin Xu, Kaushik Veeraraghavan, and Onur Mutlu. 2018. A large scale study of data center network reliability. In *Proceedings of the Internet Measurement Conference 2018*. 393–407.

[41] Mehdi Mirzaaghaei, Fabrizio Pastore, and Mauro Pezzè. 2010. Automatically repairing test cases for evolving method declarations. In *2010 IEEE International Conference on Software Maintenance*. 1–5. https://doi.org/10.1109/ICSM.2010.

5609549
[42] Bao N. Nguyen, Bryan Robbins, Ishan Banerjee, and Atif Memon. 2014. GUITAR: An Innovative Tool for Automated Testing of GUI-Driven Software. *Automated Software Engg.* 21, 1 (March 2014), 65–105. https://doi.org/10.1007/s10515-013-0128-9
[43] Andres Nötzli, Jehandad Khan, Andy Fingerhut, Clark Barrett, and Peter Athanas. 2018. P4pktgen: Automated Test Case Generation for P4 Programs. In *Proceedings of the Symposium on SDN Research (SOSR '18)*. Association for Computing Machinery, New York, NY, USA, Article 5, 7 pages. https://doi.org/10.1145/3185467.3185497
[44] Mohammad A. Noureddine, Amanda Hsu, Matthew Caesar, Fadi A. Zaraket, and William H. Sanders. 2019. P4AIG: Circuit-Level Verification of P4 Programs. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks – Supplemental Volume (DSN-S)*. 21–22. https://doi.org/10.1109/DSN-S.2019.00016
[45] P4lang. 2022. The BMv2 Simple Switch target. https://github.com/p4lang/behavioral-model/blob/d52ac6257bb3a58606383d03b31ed89671504791/docs/simple_switch.md. (2022). accessed Jun, 19 2022.
[46] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019. Semantic Fuzzing with Zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 329–340. https://doi.org/10.1145/3293882.3330576
[47] Rohan Padhye, Caroline Lemieux, Koushik Sen, Laurent Simon, and Hayawardh Vijayakumar. 2019. Fuzzfactory: domain-specific fuzzing with waypoints. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–29.
[48] Santhosh Prabhu, Kuan-Yen Chou, Ali Kheradmand, Brighten Godfrey, and Matthew Caesar. 2020. Plankton: Scalable network configuration verification through model checking. In *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*. USENIX Association, 953–967. https://www.usenix.org/conference/nsdi20/presentation/prabhu
[49] Open Compute Project. 2015. *Switch Abstraction Interface (SAI): A Reference Switch Abstraction Interface for OCP.* Technical Report.
[50] Open Compute Project. 2017. SAI Object Model. https://github.com/opencomputeproject/SAI/blob/master/doc/object-model/pipeline_object_model.pdf. (2017). Accessed January 12, 2022.
[51] Talia Ringer. 2021. *Proof Repair.* Ph.D. Dissertation. University of Washington.
[52] Kostya Serebryany. 2016. Libfuzzer: A library for coverage-guided fuzz testing (within llvm). *URL https://releases.llvm.org/7.0* (2016).
[53] Haruto Tanno, Xiaojing Zhang, Takashi Hoshino, and Koushik Sen. 2015. TesMa and CATG: Automated Test Generation Tools for Models of Enterprise Applications. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 2. 717–720. https://doi.org/10.1109/ICSE.2015.231
[54] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2017. Skyfire: Data-Driven Seed Generation for Fuzzing. In *2017 IEEE Symposium on Security and Privacy (SP)*. 579–594. https://doi.org/10.1109/SP.2017.23
[55] Xin Wu, Daniel Turner, Chao-Chih Chen, David A Maltz, Xiaowei Yang, Lihua Yuan, and Ming Zhang. 2012. NetPilot: Automating datacenter network failure mitigation. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*. 419–430.
[56] Lihua Yuan and Microsoft Azure Network Team. 2018. SONiC: Software for Open Networking in the Cloud. (Aug. 2018). Retrieved July 1, 2022 from https://conferences.sigcomm.org/events/apnet2018/slides/lihua.pdf
[57] Michał Zalewski. 2019. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl. (2019). Accessed September 5, 2021.
[58] Hongyi Zeng, Peyman Kazemian, George Varghese, and Nick McKeown. 2012. Automatic Test Packet Generation. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies (CoNEXT '12)*. Association for Computing Machinery, New York, NY, USA, 241–252. https://doi.org/10.1145/2413176.2413205
[59] Junlan Zhou, Malveeka Tewari, Min Zhu, Abdul Kabbani, Leon Poutievski, Arjun Singh, and Amin Vahdat. 2014. WCMP: Weighted Cost Multipathing for Improved Fairness in Data Centers. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys '14)*. Association for Computing Machinery, New York, NY, USA, Article 5, 14 pages. https://doi.org/10.1145/2592798.2592803

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

A LISTING OF SELECTED BUGS FOUND IN PINS

Bug Description	Discovery	Component	Days to Resolution	Integration Issue?	Trivial test that would find bug
Deleting non-existing entry causes entire batch to fail	`p4-fuzzer`	P4Runtime server	14	no	
Does not handle MODIFY requests correctly, leaving old action parameters unchanged in table entries	`p4-fuzzer`	P4Runtime server	4	no	
P4Info push failures are not propagated up to the controller.	`p4-symbolic`	P4Runtime server	0	yes	Table entry programming
Does not support reading ternary fields	`p4-symbolic`	P4Runtime server	0	no	Read all tables
Does not capitalize ACL table names	`p4-symbolic`	P4Runtime server	16	yes	Table entry programming
Incorrect error message for duplicate entries	`p4-symbolic`	P4Runtime server	2	no	
PacketOut packets incorrectly get punted back to controller	`p4-symbolic`	P4Runtime server	26	no	Packet-out
Uses an orchestration agent API that does not support the space character in keys. This leads to the rejection of all ACL table entries.	`p4-symbolic`	P4Runtime server	34	no	Table entry programming
Incorrect handling of zero bytes in IDs	`p4-fuzzer`	P4 Toolchain	22	no	Set P4Info
Does not clean up all WCMP group members when creation of one fails.	`p4-fuzzer`	Orch. Agent	6	no	
Switch rejects WCMP groups with buckets with the same action, violating the P4RT specifications	`p4-fuzzer`	Orch. Agent	157	yes	Table entry programming
A bug in WCMP group updating logic causes unchanged group members to get removed	`p4-symbolic`	Orch. Agent	3	no	
VRF deletion fails due to incorrect ALPM flag usage & VRF response path is broken	`p4-fuzzer`	Orch. Agent and SyncD Binary	15	no	
Does not clean up invalid entries in ACL tables, causing RESOURCE_EXHAUSTED error after 30 entries	`p4-fuzzer`	SyncD Binary	120	no	
L3 forwarding not enabled for submit-to-ingress packets, causing them to be dropped with the new chip	`p4-symbolic`	SyncD Binary	19	yes	
Switch occasionally re-marks DSCP to 0 in forwarded packets	`p4-symbolic`	SyncD Binary	53	yes	
A port sync daemon restarts unexpectedly, breaking all packet IO	`p4-symbolic`	Switch Linux	3	yes	Packet-In
Daemons crash when network interface goes down	`p4-symbolic`	Switch Linux	164	yes	
Runs a daemon that creates conflicting VRF configurations with other new services	`p4-symbolic`	Switch Linux	5	yes	Set P4Info
Switch sends IPv6 router solicitation packets unexpectedly	`p4-symbolic`	Switch Linux	unresolved	yes	
Runs LLDP causing packets to be punted to controller	`p4-symbolic`	Switch Linux	9	yes	Packet-In
Resource guarantees for router_interface_table are unrealistically high for the new chip	`p4-fuzzer`	P4 Program	47	yes	
Header fields get rewritten before ACL is applied	`p4-symbolic`	P4 Program	14	no	
P4 program does not reflect that switch drops IPv4 packets with destination IP 255.255.255.255	`p4-symbolic`	P4 Program	36	no	
Program matches on the wrong ICMP field	`p4-symbolic`	P4 Program	13	no	Packet-In