Finding Hard-to-Find Data Plane Bugs with a PTA

Pietro Bressana
Università della Svizzera italiana
pietro.bressana@usi.ch

Noa Zilberman
University of Oxford
noa.zilberman@eng.ox.ac.uk

Robert Soulé
Yale University
robert.soule@yale.edu

ABSTRACT

Bugs in network hardware can cause tremendous problems. However, programmable network devices have the potential to provide greater visibility into the internal behavior of devices, allowing us to more quickly find and identify problems. In this paper, we provide a taxonomy of data plane bugs, and use the taxonomy to derive a Portable Test Architecture (PTA) which offers essential abstractions for testing on a variety of network hardware devices. PTA is implemented with a novel data plane design that (i) separates target-specific from target-independent components, allowing for portability, and (ii) allows users to write a test program once at compile time, but dynamically alter the behavior via runtime configuration. We report 12 diverse bugs on different hardware targets, and their associated software, exposed using PTA.

1 INTRODUCTION

Bugs in network hardware can result in financial loss, security breaches, or significant downtime for essential services. Unfortunately, despite extensive testing, these bugs can be very difficult to find [34].

Example. As an example, imagine that a device drops packets when the input traffic exceeds a certain rate. How would we find and diagnose this bug? This sounds like it would be a simple bug to catch. After all, we would certainly notice packet drops. In reality, the bug—which we found in the NetFPGA [27, 50] reference projects—was not discovered for more than a decade after the platform had been introduced.

The root cause of the bug was that the input arbiter in the design was not work-conserving, i.e., packets were held in an input queue even when the output was idle. However, the bug did not reveal itself on the NetFPGA 1G board with 4×1Gbps interfaces, or on the SUME board with 4×10Gbps interfaces. It was only revealed when the design was ported to a 2×100Gbps Alveo board. The bug in the design was passed from one generation to the next, and the capacity of the network interface masked the defect in the internal design.

So, how could we have found and fixed this bug sooner? There are a few immediate observations that we can make.

First, the bug only appears when the traffic rate exceeded a threshold, in this case, ~40Gbps aggregate throughput on SUME. So, software-based approaches like simulation or emulation, which can slow the execution of a program by a factor of 10^6 [10], would not help. Instead, we need a test framework that can generate and receive traffic at line rate.

Second, finding this bug requires *internal* access to the data plane. Even if we could externally generate and send traffic to the device under test at the target rate (e.g., using an Ixia [21] or Spirent [41] platform), we need a way to distinguish a limitation of the network interface from the inefficient implementation of the input arbiter.

Third, we see that the same reference design was used on several hardware targets, including NetFPGA 1G, 10G, SUME, and Alveo. Writing tests is time intensive, and having to repeat test-writing efforts for each target would be onerous. Just as the reference design can be ported across targets, we want the tests to be portable across hardware targets.

Although we have focused on an FPGA in this example, similar bugs and observations hold for programmable ASICs, although at greater scale. Consider trying to replicate the same test scenario on a Tofino 2 ASIC, which has 128×100G ports.

Prior Work. Verifying, debugging, testing, and validating network hardware is a well studied area. A range of software-based approaches have been proposed, including simulation or emulation and formal verification [16, 26, 42]. However, none of these offer a comprehensive solution. Simulators or emulators may not faithfully model actual deployments, and, as already mentioned, cannot test scale-related bugs. And, verifiers cannot catch several types of bugs, such as bugs in the compiler, performance bugs, or bugs to due to under-specification in the language.

Therefore, network operators often augment software-based techniques with hardware-based testing. For this purpose, equipment vendors such Ixia [21] and Spirent [41] sell highly-specialized platforms, which can generate and receive traffic at line rate. Unfortunately, these devices provide limited visibility because they function as external black-box testers. Moreover, the cost of such platforms is considerable. Prior research efforts [4, 37, 49] offer lower cost solutions with similar intents, but they are limited, in terms of features, scale, and performance, e.g., OSNT [4] does not scale beyond 4 ports, and none of them work with non-Ethernet packets.

Problem and Approach. This paper addresses the problem of how to develop a high-performance, comprehensive, portable test framework for network devices. The key idea is to leverage a portion of the resources in programmable network hardware—including SmartNICs and programmable ASICs—for testing. Programmable network hardware is an attractive option for use with testing for two reasons. First, it can send and receive traffic at high rates by design. Second, it can be adapted for use-cases beyond traditional forwarding, such as has been done with in-network computing [11, 12, 22, 23, 29].

Challenges. Using programmable network hardware as testing devices, rather than forwarding devices, presents a significant challenge, because testing and forwarding are fundamentally different. In particular, we identify three, high-level challenges: (i) active vs. reactive logic, (ii) dynamic processing behavior, and (iii) portability.

First, at the most basic level, forwarding devices are reactive, meaning that they execute logic only on the arrival of an incoming packet. In contrast, testing is an active process. A tester generates test stimuli in the form of test packets, and then checks a post-condition.

Second, testing devices require much more flexibility than forwarding devices. When used for forwarding, the data plane functionality of programmable NICs and switches only changes in limited ways, e.g., it might forward packets out a different port, depending on control plane configurations. But, the forwarding pipeline is not altered during operation.

In contrast, exhaustive testing often requires significant adaptation and permutation, dynamically changing the behavior depending on the needs of the test. As an example, imagine that we want to generate a variety of packets with different header sizes, similar to how Dumitru et al. [14] check for security exploits. Changing the data plane implementation of the test program for every permutation would result in significant overhead, in terms of compilation and installation, which can take hours on some platforms.

Third, the test architecture must be portable across a range of heterogeneous target devices. To provide portability, we need to identify a set of abstractions that are flexible and powerful enough to test for a variety of possible data plane bugs, but can be generally implemented on a range of devices.

Contributions. To address these challenges, we propose a new *data plane architecture* for data plane testing. We use the term data plane architecture in the same way that it is used in the P4 programming language [6]. It identifies the programmable blocks and their data plane interfaces. Essentially, it is the contract between the data plane program and the hardware target.

The P4 open-source community has begun to standardize a few data plane architectures, including the Portable Switch Architecture (PSA) [35] for network switches, and the Portable NIC Architecture (PNA) [7] which models NICs. This paper introduces the Portable Test Architecture (PTA).

Overall, this paper makes the following contributions:

- The requirements for PTA are derived from a taxonomy of bug types in programmable network devices and we detail bugs that we have found in commercial and open-source software and hardware using the tool.
- Driven by the requirements of the bug taxonomy, PTA offers a small but powerful set of abstractions to support debugging.
- PTA has a novel data plane design that: (i) separates target-specific from target-independent components, allowing for portability, and (ii) allows users to write a test program once at compile time, but dynamically alter the behavior via dynamic re-configuration.
- PTA complements prior work on automatic test packet generation [31], fuzz testing [3, 39, 44], and software validation [26], by providing a framework for running workloads generated by those tools on actual hardware. To demonstrate how PTA can be used in conjunction with existing tools, we have developed a proof-of-concept integration with P4v [26]. Users can extract assumptions and assertions from an annotated P4 program, and map them to a hardware test configuration.
- PTA uses programmable network hardware for testing, which differs from traditional forwarding in key ways. We present a set of lessons we've learned and assumptions that were challenged in the design of the framework.
- PTA is publicly available under an open-source license [36].

Key Results. We have implemented PTA for two different hardware targets: the NetFPGA SUME platform [50] and the Barefoot Tofino ASIC. We have used the framework to evaluate several P4 programs and two P4 compilers. Using PTA, we were able to identify 12 diverse bugs. These bugs are drawn from a broad spectrum of classes of bugs, demonstrating that PTA provides a comprehensive testing solution. Moreover, these bugs were in heavily-used, heavily-tested commercial and open-source systems.

2 REQUIREMENTS AND CONSTRAINTS

The design of PTA navigates the tension between developing an expressive framework that can test for a wide range of bugs, but can be implemented on a diverse set of hardware targets. Below, we discuss these requirements in more detail by first developing a taxonomy of error types and then discussing the constraints imposed by different hardware.

2.1 Data Plane Bug Taxonomy

A wide range of bugs can occur in network devices. These bugs can be due to incorrect program logic (i.e., functional bugs); or due to problems in compiler, target hardware architecture or others. Below, we provide a taxonomy of the types of bugs that a test framework must be able to detect. These error types provide requirements that motivate the design of PTA. Note that although PTA can be used to test both fixed-function and programmable hardware, our taxonomy highlights bugs that may be unique to programmable network hardware (e.g., compiler bugs), and may not be comprehensive.

Functional Bugs. A functional bug is one in which the functionality provided by the network device is not the same as the functionality intended by the programmer. Functional bugs can occur in both the data plane and in the control plane. An example data plane bug would be not supporting IPv6 headers where such functionality was supposed to be supported. An example control plane bug would be not filling all the required entries in a given size table.

Performance Bugs. Performance bugs are related to aspects such as the maximum throughput or packet rate of a certain design, how certain packet sizes affect the throughput, whether congestion control is handled properly, and more. For performance testing, for example, the user must be able to continuously fill the pipeline with packets of a certain size and check that no packets are dropped or lost at the output. Another performance aspect is the ability to mix packet sizes in explicit ways, which exercise different parts of a design (e.g., programmable data plane, schedulers, memory access).

Compiler Bugs. Although compilers are tested with scrutiny, there may be bugs. There are at least two classes of compiler bugs. The first class of errors regards functionality bugs, e.g., where a language feature is supported but the implementation is missing, or the functionality is implemented incorrectly. A second class of errors covers the compliance with the programming language specification.

Under-Specification Bugs. The extent of a programming language definition, and the diversity between target platforms, leads to cases where the language specification is not detailed, either intentionally [1] or not. This can lead to unexpected or unintended behaviors, for example, if the specification does not detail whether the initialization of a header should be to zero, or if can remain unpopulated and random.

Architecture Bugs. Similar programs may target different data plane architectures, and even perfect programs may be susceptible to bugs in the underlying device architectures. One immediate class of such device architecture limitations is access hazards to tables, such as read-after-write. A second class of bugs uncovers limitations of the data plane architecture, such as a proprietary module (e.g., an extern) that is not responding within the expected time. A sub-class of bugs has to do with the integration of different modules in the architecture, such as caused by a mismatch in the connection of interfaces.

Security Vulnerabilities. Network devices can suffer from security vulnerabilities just like any other device, and programmable network devices introduce new threat vectors. Security vulnerabilities are commonly the result of a different class of bugs, and are highlighted due to their importance and the need for targeted tests. A test framework should allow users to quickly and efficiently test a large number of such security threats. One such example would be looking for the "Meltdown" [28] equivalent of a programmable data plane: can you craft a packet that would allow you to read the contents of previous packets, various tables, or memories? Another example is backdoors in the program, whether in the original users code or introduced as a by-product of the compilation process. The hardware test can reduce the security risk by testing the deployed program as it runs on the platform.

2.2 Heterogeneous Targets

The diversity of data plane bugs implies that a testing architecture should be flexible and expressive. However, the design of the architecture is necessarily constrained by the capabilities of the target hardware. We briefly summarize these below. We focus our discussion on two devices that are on opposite, extreme ends of the spectrum: FPGAs and ASICs. Other types of network devices, such as those based on System on Chip (SoC), fall between these two extremes [45].

FPGAs. Field Programmable Gate Arrays (FPGA) have a given set of resources, but provide users with extreme flexibility and full programmability. As long as a design does not exhaust resources, and users can compile the design while maintaining the constraints they set (e.g., on timing), FPGAs can implement almost any logical operation, with different levels of complexity.

ASICs. Like FPGA, ASICs also have a finite amount of resources. But, in contrast to FPGAs, they have a set device architecture. While ASICs have become programmable—significantly more so than in the past—their programmability is constrained to the architecture. Note that CPU architectures impose similar constraints, e.g., programming on an x86 CPU is different from programming an ARM core or RISC-V. The main advantages of switch ASICs over FPGA-based switches is that they achieve much higher clock rate (and therefore higher throughput), offer increased scale (e.g., number of ports), and use resources more efficiently.

Constraints. These differences between hardware targets hinder portability. It is often said that P4 allows users to write target independent programs. But, this is not true. A program written in P4, like programs written in other languages, is tied to the target architecture. Examples of architecture specific properties include externs, initialized values (of registers, memories and other stateful elements), and timestamp taking, among others. Portability issues are not always a property of complex hardware design. They can result from mundane aspects, such as the number of bits assigned on the metadata bus to indicate the egress port number (which may differ between an 8-port switch and a 256-port one, in order to minimize resource usage).

Abstraction	Description
Load_Image	Load the image file to the target
Init_Counters	Initialise the counters in the target
Init_Registers	Initialise the registers in the target
Generate_Packets	Generate test packets
Collect_Results	Collect raw results from target's registers

Table 1: PTA's user-facing abstractions.

3 DEBUG ABSTRACTIONS

One of the main challenges in designing PTA is identifying the core set of abstractions to support debugging. We adopt a requirement driven design process. Based on the taxonomy in the previous section, we systematically explored the necessary abstractions for each of those classes of bugs. The set of abstractions is intended to be minimal, so that it can be readily supported by diverse hardware. At the same time, it is intended to encompass the set of functions needed for testing.

To illustrate the process, we first walk through the running example—i.e., the input arbiter bug from the NetFPGA reference project from Section 1—before summarizing the complete set of PTA debug abstractions.

3.1 Requirement Driven Design

So, how might a developer find and isolate the performance bug in the input arbiter? Because the module is (incorrectly) not work conserving, we clearly need to be able to generate packets at data-path rate, creating controlled back-to-back arrival events to the arbiter.

Many bugs (e.g., functional, compiler) would depend on a particular data plane program, suggesting that the debug framework needs a method to load a data plane image. However, in this case, the bug is in the architecture of our target device, and therefore independent from the data plane program that we would load. To test the architecture, we need access to the low-level abstractions offered by the hardware, including the metadata bus, stateful ALUs, and any externs provided by the architecture of additional hardware modules.

We need modules to initialize and check the values of stateful elements, e.g., counters and registers. This allows us to confirm the number of packets sent and processed by the pipeline, and more generally, application specific logic.

Finally, to detect the presence of dropped packets (again at line rate), we need a way to collect and inspect output packets.

3.2 Core Abstractions

By following this requirements driven design process, we identify two classes of abstractions: user-facing abstractions (Table 1) and

Abstraction	Description
Metadata_Bus	Layout of the metadata bus
Stateful_ALU	Architecture of the Stateful ALUs
Extern	Architecture of the Extern modules
Register_Read	Interface for reading hardware registers
Register_Write	Interface for writing hardware registers

Table 2: PTA's back-end abstractions.

back-end abstractions (Table 2). User facing abstractions are used to specify the functionality of a test. Back-end abstractions represent the architecture of the target device.

User-Facing Abstractions. Users writing tests will be using user facing abstractions, similar to functions. As these abstractions are not target-specific, a test will be written only once. The abstractions are used to load the program image (`Load_Image`), initialize registers (`Init_Registers`) and counters (`Init_Counters`) and to generate and collect packets (`Generate_Packets` and `Collect_Results`).

Note that although packet generation is exposed via user-facing abstractions, it is adapted to the target (e.g. to vary transmission rate) using back-end abstractions that are transparent to the user.

Back-End Abstractions. Back-end abstractions are used to specify a network-device target, and are called by any tests using this target device. The back-end abstractions library includes both the layout of the metadata bus (`Metadata_Bus`), that is used by PTA as a configuration channel, and the architecture specification of both stateful ALUs (`Stateful_ALU`) and extern modules (`Extern`). Since the hardware components of the framework are usually accessed through a register interface, PTA provides two additional abstractions for reading and writing registers (`Register_Read` and `Register_Write`).

4 PORTABLE TEST ARCHITECTURE

Building on the core abstractions, PTA provides a comprehensive hardware data plane testing solution. PTA is *programmable*, meaning that the tool can be customized to the particular testing needs of the user for a diverse set of bugs. It is also *re-configurable*, meaning that new tests can be run via dynamic re-configuration (e.g., using register access), rather than re-programming (e.g., requiring a new image file). PTA allows for *integration with existing tools*, providing prior work on automatic test packet generation [31], fuzz testing [3], and software validation [26] with a path to run on hardware. When used to test devices with programmable data planes, PTA allows *access to internal state*, providing detailed fault localization. PTA allows users to test network devices in *real time* at full line rate, and test results are *reproducible*. We expect PTA to be deployed out of band, i.e., in parallel to live traffic. It does not incur additional latency or otherwise alter the traffic.

4.1 Overview

Imagine that a user wants to verify a certain data plane. We assume that the user has some information about the data plane functionality, e.g., the P4 program, but not all information of the dataplane, e.g. hardware-target's micro-architecture. The user will need to devise a test plan that covers the range of potential bugs, covered in Section 2.1. Most of these tests are generic (e.g., performance tests) while some are use-case specific (e.g., P4-program specific functional tests). Next, following this plan, imagine that the user wants to verify that this data plane runs at line rate for different packet sizes. To run this test, we need three components: a packet generator, to inject packets into the data plane; an output checker, to assert that post-conditions hold at the end of the test; and a management component, to run the test. All these components are illustrated in Figure 1, which shows the high-level design of PTA. Both the generation and checker modules are implemented in hardware, while the **management component** is a set of software programs.

Even for this simple example, there are a range of parameters and scenarios to be tested: How many packets should be sent? What sizes should the packets be? At what rate should be packets sent? And, at what rate do we expect the output packets to arrive? What protocols are being used in the packet headers? Hand-writing tests for each of these scenarios would be tedious, and possibly error-prone.

To help reduce the burden, PTA separates tests into two parts: the programmable part and the re-configurable part. The programmable part can be thought-of as data plane specific. Users can, for example, write a program to generate packets with different protocol headers, and write a checker to validate the emitted headers. The **re-configurable part** is test-specific. It is a control plane configuration of the programmable part, that allows users to change parameters such as packets sizes, sending rates, etc.

The programmable part of a test can be divided into infrastructure that is target-independent and target-dependent. The **target-independent** infrastructure is written in P4, and controls, for example, the definition of protocol headers. The **target-dependent** infrastructure is the device-specific functionality (e.g., generation of blank packets).

Note that although PTA's programmable parts are implemented in P4, the data plane under test does not need to be written in P4. PTA can be used to test data planes designed using a variety of different workflows and languages, including high level synthesis, C/C#, and HDLs (e.g., Verilog).

It is important to stress that all of the components of PTA are implemented *inside* the target network platform. This provides PTA with several important advantages. First, it allows PTA to test the data plane while avoiding the surrounding hardware, including the network interfaces. A failure of a test can guarantee that the cause is not in the interfaces but in the tested data plane. Second, it enables testing the device at line rate and at real time. Testing a device at line rate is challenging due to the cost of external traffic generators (e.g., Ixia [21]), making it outside the reach of many users. Thus, the internal data path may have a certain speed-up over the external interfaces, making it very hard to create and detect hazard scenarios such as read-after-write in two consecutive clock cycles, or certain cross-traffic scenarios that lead to consistency issues. Finally, PTA allows users to test and debug their data plane in the field, without additional equipment, and without changing the physical settings.

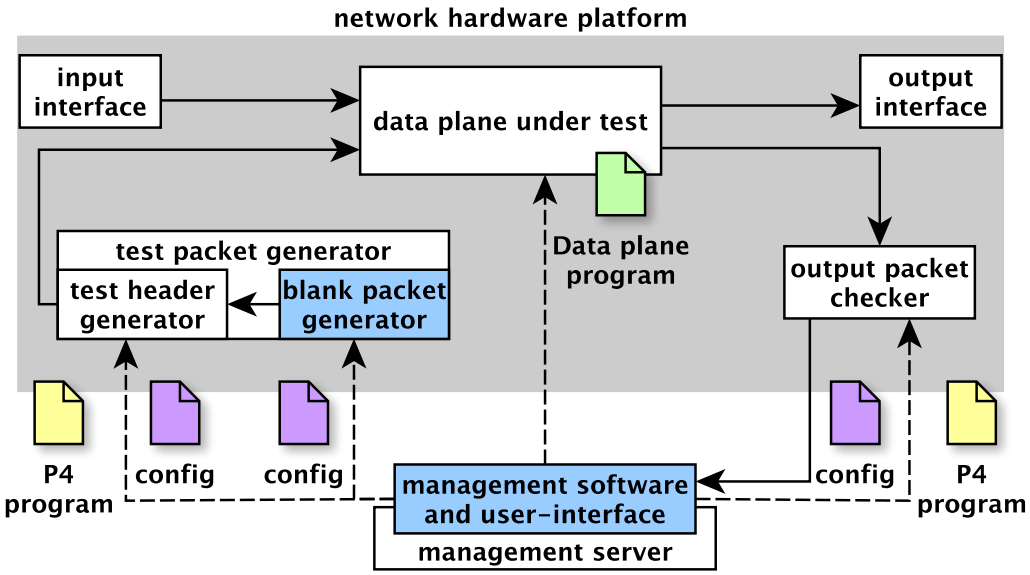

Figure 1: The proposed architecture: target specific infrastructure (light-blue), portable suite of P4 test programs (yellow) and test-specific configurations (purple).

4.2 Target-Independent Test Infrastructure

The target-independent infrastructure of PTA is a set of P4 programs used for packet header generation and output checking. Both components are implemented as a sequence of match-action tables and a set of registers that change control flow. The entries in these tables are re-configurable, and provide the flexibility to support different tests.

PTA includes a set of default programs that developers may use to generate packets with standard protocols (e.g. Ethernet, IP, TCP, UDP, etc.) and to check for common conditions. Thus, for many test scenarios, users need not write any P4 code themselves. To support custom protocols and to check for data plane specific test scenarios (e.g., to generate a packet with a Paxos protocol header [11] and test for a specific post-condition), users can expand on these default programs for custom-protocols using P4.

Packet Header Generator. The packet header generator takes blank input packets, and turns them into stimulus packets injected to the data plane under test. The P4 program defines the protocols that need to populate the header and properties of the contents. Because tests are written in P4, developers can use any protocol that is implementable in P4, and can easily add custom headers, different fields, change the ordering of headers, and more. For example, an empty packet entering the test header generator will be emitted as a standard TCP/IP packet, with a certain sequence number and valid checksum. Combined with a blank packet generator, the test header generator will control packet size and contents, traffic pattern (e.g., inter-packet gap), and may even intentionally craft illegal stimulus packets. The output of the test header generator connects to the input of the data plane under test.

Output Packet Checker. The output of the data plane under test is connected to the output packet checker. The output packet checker, implemented in P4 and shown in Figure 1, can be programmed to expect specific values or sequences of values within the returned packets. It compares these values against the input traffic stream (e.g., to detect packet drop, reordering, or other points of failure). The stages within the checker's stages support different types of functionality, such as matching specific header fields, or comparing metadata bus values. The outcome of each check is stored in a memory. Typical types of stages include an ALU, that performs both logic and arithmetic operations over headers and metadata, and CAM and TCAM blocks that compute simple matches against header and metadata fields. The number of received packets is an example of a common functionality implemented using counters.

4.3 Target-Dependent Test Infrastructure

P4-based data planes are packet driven, and do not generate packets without a stimulus. For this reason, PTA uses a blank packet generator that creates empty packets, feeding the test packet generator's P4 pipeline. By a blank packet, we mean a packet with no header fields and no payload. The blank packet generator is target specific. For example, some ASIC switches (e.g., Tofino) already have a built in-packet generator, while other devices (e.g., FPGA) require a dedicated implementation. Even if a packet generator already exists within the device, it varies in features and properties between devices, and is therefore target specific.

Additional target specific infrastructure is focused on the connectivity of PTA: connecting the output of the test packet generator to the data plane under test, and connecting the output of the data plane under test to the output packet checker. This connectivity depends on the hardware architecture of the device, where PTA is internal to the device. For example, NetFPGA has a 256-bit wide AXI-4 streaming

bus, and the output of the test packet generator is connected to NetFPGA's input arbiter. On other devices, the bus type and width will vary, as well as the connection points.

Finally, PTA includes 4 additional functions implemented in hardware that are useful for testing: random number generation, counters, time-stamping, and a method to swap fields. These are used as externs in the P4 programs.

4.4 Re-Configuration

To fully explore the parameter space for a test, PTA allows users to re-configure tests. To re-configure a test, a user writes a simple test script. The script allows users to change features such as the packet rate burst length, the gap length between packet transmissions, the packet sizes, the payload sizes, etc. It also allows users to set initial meta data flags and fields.

All configurations updates are control plane changes that happen dynamically at runtime. This allows users of PTA to explore a wide variety of test scenarios without having to recompile the test programs and install a new image—which can take a long time for some targets.

4.5 Interactions with the Control Plane

PTA monitors the data plane, and can identify and verify packets going to the control plane. It does not have direct visibility into the control plane. However, the functionality of a networked-program is a combination of the data plane program and the control plane configuration. During tests, PTA treats both as a single unit. A correctly programmed data plane with a misconfigured control plane may lead to a test failure, for example if a missing entry in a table leads to packet drop. In this context, PTA can indicate why a test has failed (e.g., a packet was dropped), but not what was the source of the failure (i.e., data or control plane). As we describe in Section 8.2 (test#05), PTA successfully detects control plane related bugs. Another advantage of PTA's approach is that it further enables testing different control plane configurations.

4.6 Management and User-Interface

While P4 programs define the type of packets that can be generated by PTA, the management software defines the properties of the test. It is responsible for configuring the control and data planes, triggering tests, collecting and processing results, and declaring Pass/Fail. For this purpose, PTA includes a set of Python libraries to help manage the tests.

Test packet generation is determined by configuring the blank packet generator. This includes both information about the generated packet stream (e.g., number of packets, packet size, burst properties) as well as the input metadata accompanying the packet (e.g., path through the generation data plane, which headers to add).

Test results are specified using assertions. The management software reads from the output packet generator the results of the test (e.g. number of packets received), and compares them to the expected values set as Pass criteria.

An example test appears in Appendix A.

4.7 Test Generation

The definition of validation tests is always a challenging task, which we address in Section 9. In this section, we focus on the generation of pre-defined tests.

First, PTA reuses verification tests as validation tests through an automated integration with P4v, as discussed in Section 5. This means that tests previous written and executed for P4v are compiled and run on the hardware target using PTA.

Aspects that can not be tested using verification tools, such as P4v, must be written by the user. The user has a high degree of freedom in test generation. PTA includes a test library that allows users to specify test pre-conditions and post-conditions in a Python scripts. These are automatically translated to a configuration of PTA. An example test script is presented in Appendix A. A lot of these tests, though not all, can be defined once and then be reused as data plane programs change or for regression tests (as is the case with the NetFPGA).

PTA supports some types of fuzz testing, such as random payload or random header contents, and these were used in a few functional tests described in Section 8.2.

The open-source nature of PTA means that users can also change the hardware infrastructure to support new or different tests that were not considered by the authors of this paper.

5 INTEGRATION WITH A VERIFIER

There has been significant prior work on workload and test case generation. This work covers a broad range of techniques, including automatic test packet generation [31], fuzz testing [3], and software validation [26]. PTA provides a path for these tools to run the tests that they generate on hardware. As a proof-of-concept about how such tools could integrate with PTA, we developed a prototype P4v-to-PTA translator.

Many software verification tools, including P4v [26] and Assert-P4 [16], are based on Hoare logic, which provides a formal system for reasoning about the correctness of computer programs. The central feature of Hoare logic is the Hoare triple. A Hoare Triple is of the form $\{P\}\ c\ \{Q\}$, where P is the precondition, Q is the postcondition, and c is the command, i.e., a piece of code that changes the state of the computation. A verifier can translate these assumptions and assertions into logical formulas, and use an automated theorem-prover to check if there is an initial state that leads to a violation.

The P4v-to-PTA translator parses an annotated P4 program and automatically extracts the assumptions (i.e., the Ps) and assertions (i.e., the Qs). For each assumption, the tool collects a list of test headers to be generated and identifies suitable values for each field. It then generates the P4 code and configuration for the test packet generator. For each assertion, the tool generates the output packet checker's data plane and the accompanying configuration, as illustrated in Figure 2.

This process is fully automated. To test an annotated P4 program, a user simply needs to place the P4 code in a specified folder. The PTA tool not only generates the test generator and checker P4 programs for PTA along with the associated configuration, but also compiles and runs the user's application on Tofino. Furthermore, the management scripts collect test results and present them to the user. The entire process is similar to the using the P4v command line tool,

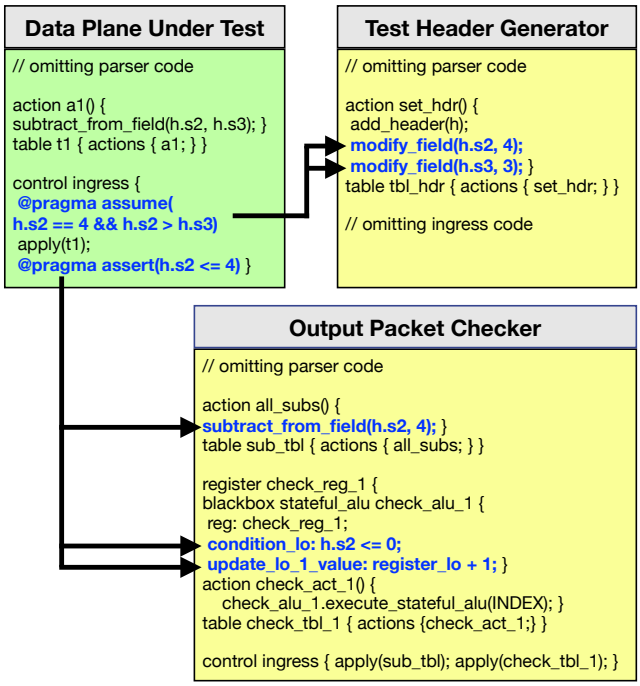

Figure 2: Integration with P4v.

but runs real, extensive hardware validation. Appendix B describes an example P4v-to-PTA test.

Using PTA as a runtime tester, we were able to identify two bugs in the Barefoot Networks SDE compiler, related to saturating integers (Section 8.1). These bugs would not be caught by formal verification. The programs that suffered from these bugs were logically correct, but still exhibited unexpected behavior.

6 IMPLEMENTATION

We have implemented PTA on two target devices: NetFPGA SUME platform and Barefoot Networks' Tofino ASIC. The code base, which includes both the implementations, is open source, and available on GitHub [36].

FPGA Implementation. An FPGA-based prototype is implemented on the NetFPGA SUME platform [50] using the P4→NetFPGA workflow [19]. It uses Xilinx's SDNet[47] 2018.2 and supports P4_16.

PTA builds upon the NetFPGA SUME reference architecture, which is composed of a data plane that processes traffic arriving from four independent network interfaces and a host (over PCIe). PTA taps to the architecture through a sixth input interface and a sixth output interface. Our implementation of PTA on NetFPGA follows the design outlined in Section 4. We provide a detailed description of our FPGA implementation in Appendix C.

ASIC Implementation. We also implemented PTA on Barefoot Networks' Tofino, a 6.5Tbps programmable Ethernet switch ASIC [5]. The Tofino ASIC provides either two or four hardware packet processing pipelines, depending on the ASIC model. The four pipelines of Tofino allow us to implement PTA using an architecture similar to the FPGA architecture: the packet generator and checker are implemented within separate pipelines, and the data plane under test is loaded in a dedicated pipeline. We note that the current Barefoot Networks SDE control plane software does not allow users to manage different programs loaded on to different data planes. This limitation is not inherent, and we expect the functionality will be supported in future releases. However, as a temporary work-around, we performed our debug experiments using three switches instead of one. Our implementation of PTA on Tofino is based on P4_14, as P4v supports only this version of the language, and as Barefoot's development tools supported P4_14 more extensively at time of development.

7 EVALUATION

Validation. We validate PTA independently from any data plane under test, both on Tofino and on NetFPGA. The validation uses external traffic generation (e.g., OSNT [4]) and capture tools (e.g., Endace DAG) to confirm assumptions such as traffic rate and contents. Barefoot further confirmed to us that the packet generator built inside the Tofino chip runs at line-rate. We conduct a functional validation of PTA, testing using both external and internal tools (counters, logic analyzer) to examine each feature. Testing of programmable data planes began only once the PTA infrastructure was tested.

Performance. We have evaluated and confirmed that both NetFPGA SUME and Tofino-based programs run at line rate, using the setup previously described and ranging packet sizes from 64B to 1514B on NetFPGA[1]. PTA implementation on NetFPGA does not allow for congestion propagation into PTA's pipelines, meaning that any flow control indication leads to packet drop outside the modules. For both NetFPGA and Tofino, support for congestion control within the pipeline is the same as for any other programmable dataplane [47]. We showed these properties of PTA in Section 8.4, by testing networked programs for line-rate and identifying bugs.

Resource Consumption. PTA introduces two new modules to a device. On the generator side, NetFPGA programs required between 2 and 4 pipeline stages, using one table and 1-2 externs, and Tofino implementations required 2 tables. On the checker size, NetFPGA programs required between 5 and 7 pipeline stages, using two tables and 3-5 externs. On Tofino, 7 tables and 5 stateful ALUs were required in the checker. A breakdown of these results is provided in [36].

We report the resources overhead introduced by PTA, but caution that it is difficult to quantify resources in a meaningful way, since the amount used depends on the program under test and the compiler. For example, on NetFPGA the compiler requires that all tables have least 64 entries, even if 16 entries would be sufficient. A newer version of SDNet (2019.1), not currently supported by NetFPGA, is more resource efficient.

On NetFPGA, representing an FPGA-based use case, the resource overhead of PTA (i.e, average of logic and memory use) never exceeded 15%, which was for the experiments with NDP [18], compiled with SDNet 2018.2. In many cases (e.g., INT, Learning Switch) this number drops to 9%. The blank packet generator required just

[1] Details of Tofino evaluation are under NDA.

Metric	Device Property	Example
Max # of headers in a single test	PHV size	4Kb
Max # of checks in a single test	# of Stateful ALUs	40
Max # of packets in a single test	Counter width	4 billion
Max test speed	Pipeline Bandwidth	1.6 Tbps
Max packet size	Max Transmission Unit	1514 B

Table 3: Additional Evaluation Metrics. Example values are indicative of the proof-of-concept implementations.

0.13% logic overhead, and no memory. Detailed resource consumption is provided in [36].

On Tofino, PTA tested a data-plane program on one pipeline using other pipelines. Since resources are not shared between pipelines, PTA does not "take away" resources from the data plane under test. ASIC resources are given, and PTA easily fits, using the resources noted above.

Test Completion Time. The PTA run time includes four components: platform setup (i.e., downloading an FPGA bit file), configuration, test execution time, and results collection and report. The test execution time is test-dependent, i.e., it depends on how long a user wants to send traffic, the number of parameters to explore, etc. For the tests that we ran on NetFPGA, the average overall time was ~110s, including all four components, though for some tests this number was reduced to ~70s. Out of that, the platform setup time, which is a one time process, is ~20s, and test re-configuration, including populating tables, is in the order of seconds. An exhaustive performance test on NetFPGA SUME, which tests throughput under each and every supported packet size, with a billion packets per packet size, was ~3000s.

Additional Metrics. Many of PTA's performance metrics, summarized in Table 3, are a property of the hardware target, not PTA. For example, the number of headers depends on the size of the packet header vector (PHV), and the number of verified aspects in a test (e.g., dropped packets, correct headers checks) depends on the number of stateful ALUs in the device.

8 BUGS FOUND

Our implementations of PTA enabled us to uncover bugs within different programs and architectures, while covering use cases discussed in Section 2. Table 4 provides a partial list of tests run and bugs found using PTA. The table indicates the name of the program we used for the data plane under test, a brief description of the category of bug, the hardware platform, and whether or not the test passed. Note that when the program name is *Any*, it indicates that the bug was not tied to a particular program. We discuss these particular bugs as they highlight the diversity of test cases that PTA enables.

8.1 Compiler Checks

PTA was able to find or confirm three bugs in version 8.9.1 of the Barefoot SDE compiler. These bugs were found when integrating with P4v, discussed in Section 5. In the first bug, (test #01), the compiler generated incorrect byte swapping code (e.g., between big and little endian). This bug has been fixed in the 9.0.0 release of the SDE. The second bug is related to "saturating" an attribute in header fields (test #02); header fields marked as "saturating" always collapse to their minimum value after a "subtract" operation is computed on them. The third bug prevents the implemented data plane from correctly processing "signed" header fields (test #03). Although represented in two's complement form, "signed" fields are treated as unsigned numbers in the hardware, thus generating incorrect results. We reported these bugs to the developer, and they have since been fixed.

8.2 Functional Tests

We discovered several functional bugs in multiple designs implemented on NetFPGA SUME, including the Verilog and P4 Learning Switch designs, and NDP [18]. First, packets with invalid source MAC addresses pass through the data plane and reach the output network interfaces, even though they should have been dropped before traversing the pipeline (test #04). This bug differs from the Parse Reject bug (test #08), as the value within the header should be banned, not the header itself. Furthermore, the issue is Ethernet compatibility, not compatibility with the P4 specification. Second, we find that, when the number of entries written to the MAC lookup table exceeds the size of the table, the write pointer will wrap around, and the first entry will be over-written (test #05).

When testing a P4 implementation of In-band Network Telemetry (INT) [20] on the NetFPGA platform, we detect some missing functionality (test #06). This includes missing measurement of switch hop latency, egress port utilization, or queue congestion status. The design also does not report which rules matched while traversing the data plane or provides information about other flows traversing the same network queues.

A more serious bug in the implementation of INT is the handling of packets with a large instructions count (test #07). The INT specification [20] states that "a device would cease processing an INT packet with an Instruction Count higher than the number of instructions that it is able to support". In our test, we find that if more than five instructions are requested, the program fails to set the Bottom-of Stack (BOS) flag to the last (fifth) INT header.

8.3 Under-Specification Tests

Because different hardware targets have different capabilities and features, they may exhibit different behavior. And, in some cases, it would be unreasonable to force all targets to have a uniform behavior, because doing so would add unnecessary complexity to a design, or add additional performance overhead. For such situations, the language specification often leaves the implementation details as up to the compiler.

One example of such behavior is illustrated in the snippet of code in Figure 3, which shows the implementation of a parser in P4 (test #08). It includes logic to extract the Ethernet header and examine the type field of the Ethernet header. If the type field indicates IPv4, the parser will transition to the parser state for extracting IPv4 headers. Otherwise, the program will drop the packet (i.e., `reject`).

The intention of this program is that any non-IPv4 packets should be dropped. However, the behavior of the program when compiled

```
1  // Parse packet headers by specifying state
2  // machine transitions.
3  parser Parser(packet_in b,
4             out Parsed_packet p,
5             inout sume_metadata_t sume_metadata) {
6
7     state start {
8        b.extract(p.ethernet);
9        transition select(p.ethernet.etherType) {
10          IPV4_TYPE: parse_ipv4;
11          default: reject;
12       }
13    } // Eliding IPv4 parser
14 }
```

Figure 3: Subset of a P4 program that reject non-IPv4 packets. The behavior of the bold line is unspecified.

using P4→NetFPGA [19] might run counter to user expectations—the packet is forwarded through the programmable pipeline and out of the device.

The reason is because the P4 language spec leaves the choice of how to implement a parser reject state up to the architecture. The SDNet compiler [47] does not implement the reject state as drop and P4→NetFPGA [19] does not use the reject indicator provided by the Simple SUME Switch architecture used by SDNet.

Technically, forwarding the rejected packet is not a bug, since the implementation is not contrary to the specification. However, it does result in unintuitive behavior that might surprise a developer. And, this behavior would not necessarily be caught by verification tools like P4v [26] or Vera [42], depending on how they model `reject`.

8.4 Performance Tests

We evaluate the performance of several P4 and HDL based programmable designs built upon the NetFPGA infrastructure. We first discuss bugs that are specific to a given program. We then discuss bugs that are a property of the NetFPGA infrastructure.

In the NetFPGA Reference Switch design, we discover that the lookup table is not able to sustain subsequent entry updates with packet sizes of less than 385B, due to the write access latency (test #09). When the packet size is 385B or bigger, meaning 13 clock cycles or more between two updates, the design functions as expected.

Running a similar test on the P4-based learning switch resulted in a failure to support consecutive table updates at line rate, regardless of packet size. The root cause to this limitation is the separation of control and data planes, which means that updates to the lookup table must go through the host by design.

An evaluation of the P4-based INT design on NetFPGA yielded interesting performance results (test #10). We find that the data plane can sustain the full internal throughput (50Gbps) only with unaligned packet sizes (e.g., 65B, 97B), but not for data path aligned packet sizes (e.g., 64B, 96B). We expect that this issue is caused by the expansion of the packet within the encrypted data plane module, beyond the INT header added to the packet. This is also the explanation proposed to us by the P4→NetFPGA designers.

8.5 Architecture Tests

A few of the bugs uncovered by PTA had to do with the architecture of specific designs or with the underlying hardware infrastructure (test #11). For example, initial throughput testing of both HDL-based and P4-based learning switches resulted in a large number of packet drops. The cause was found in the arbiter at the input to the data plane, that turned out not to be work conserving. An additional architecture limitation was discovered at the output of the data plane, at the output queues. Full rate traffic through the data plane under test led to packet drops at the queues, which turned out to be an intentional design choice by the NetFPGA team. They designed the overall supported outputs queues throughput to be circa 40Gbps. We note that PTA found this bug after the platform had already been in use for more than 10 years. The fix has supported 2 more recent NetFPGA-based projects.

8.6 Security Checks

In the course of working on this paper, we have conducted several experiments, both on P4→NetFPGA and on Tofino, trying to uncover security vulnerabilities. In our exploration, we focused on one aspect of the P4 language, which is *Undefined Behaviors* (Section G.2 of P4 Specification v1.1.0 [32]). This includes aspects such as uninitialized variables, accessing header fields of invalid headers, and accessing header stacks with an out of bounds index.

First, we tried to identify the networking-equivalent of a "Meltdown" [28] bug by attempting to infer the contents of previous packets using malformed packets (test #12). In principle, we try to infer the contents of the memory by reading a value of a non-existing header in the packet, in an attempt to use previously stored header contents. This is one form of accessing header fields of invalid headers. Positively, we find that the SDNet compiler returns a zero value for such attempts, providing stateless operation between packets.

In another test, we attempted to read headers beyond the end of the packet, with a similar motivation (test #13). In this case the result was positive as well, with the Tofino switch dropping the "aggressor" packet. SDNet does not allow such operations either, invalidating all parsed bits of the offending packet. This case is interesting, as it touches on the delicate interface between compiler and architecture. Although SDNet guards against such operations, P4→NetFPGA did not handle the error indication from SDNet. Therefore, the unprocessed and partially corrupted packet may still be emitted.

8.7 Comparing Designs

By comparing seemingly identical designs, we do not identify bugs, as these are covered by previous scenarios. However, we do identify gaps in specification, behavior, or performance. For example, we compare two implementations of a learning switch: one written in Verilog, and one written in P4. Both designs share the same NetFPGA infrastructure, and differ only in the data plane module. Despite the similarity, we find two differences in functionality. First (test #14), in the P4-based design, two ports cannot be assigned to the same MAC; once a Port-MAC binding has been learned, it cannot be overwritten by other packets. In contrast, in the Verilog-based design, after a Port-MAC binding has been learned, it can be overwritten by other packets. Second (test #15), in the Verilog-based design, overflow happens when exceeding the table size and the first entry is overwritten without any notice (as noted before). In the P4-based design, on the other hand, no overflow happens when

Test#	Program	Category	Description	HW Plat.	Pass/Fail
01	*Any*	Compiler	Byte swapping	Tofino	Fail
02	*Any*	Compiler	Saturating	Tofino	Fail
03	*Any*	Compiler	Signed fields	Tofino	Fail
04	NDP, Switch (P4, Verilog)	Functional	Invalid MAC	NetFPGA	Fail
05	NDP, Switch (P4, Verilog)	Functional	Table wrap	NetFPGA	Fail
06	INT	Functional	INT features	NetFPGA	Fail
07	INT	Functional	Instructions count	NetFPGA	Fail
08	INT	Underspecification	Parser reject	NetFPGA	Fail
09	NDP, Switch (P4, Verilog)	Performance	Write latency	NetFPGA	Fail
10	INT	Performance	Aligned packet sizes	NetFPGA	Fail
11	*Any*	Architecture	Input arbiter	NetFPGA	Fail
12	*Any*	Security	Meltdown	NetFPGA/Tofino	Pass/Pass
13	*Any*	Security	Read headers beyond	NetFPGA/Tofino	Fail/Pass
14	Switch (P4)	Designs	Port MAC	NetFPGA	Fail
15	Switch (Verilog)	Designs	Table wrap	NetFPGA	Fail

Table 4: A subset of the tests we ran and bugs found using PTA.

exceeding the table size. In principle, such updates are expected to be silently dropped by the control plane. This is a property of the closed-source compiler, which we don't have visibility to test.

8.8 Ethics and Corrective Actions

Ethical issues have been considered as part of this research. We have focused on the handling of vulnerabilities and weaknesses discovered in the different designs. Vulnerabilities have been disclosed and discussed with code and platform originators, which also helped us clarify what is considered a bug, a known design limitation, or an unsupported feature. We have further taken a positive approach and contributed code fixes to open-source projects (e.g., NetFPGA), as a means to improve their quality based on our findings.

The reject limitation found in P4→NetFPGA was reported to the NetFPGA project and Xilinx Labs. Xilinx Labs have proposed a work-around that enables users to support functionality similar to reject in the pipeline, even though actions as a result of reject is not implemented in the compiler.

We discussed the architecture and performance issues in the NetFPGA Reference designs with the NetFPGA team. The NetFPGA team indicated to us that they were aware of a minimum-access latency limitation for table updates, but not to the discovered extent. The authors of this paper have also contributed a fix to the NetFPGA input arbiter module as part of this work, as well as the packet generation module of PTA.

Bugs in the Barefoot Networks SDE were reported by entering a ticket on the FASTER portal and via personal email communication. All bugs reported in this paper have since been fixed by the developers.

9 LESSONS LEARNED

In this section, we summarize our some lessons learned through our experiences with testing network data planes.

9.1 P4 as a Language for Writing Tests

PTA allows users to write tests in the P4 language. It is natural to question if this is a good choice. After all, P4, by design, trades-off expressiveness for performance.

One reason for using P4 is opportunistic. There are P4 compilers targeting ASICs [6], NPUs [30], FPGAs [46, 47], GPUs [25], and CPUs [24, 33, 38]. Therefore, tests written in P4 should be portable to a wide variety of devices.

On the other hand, P4 is not completely target-independent. For PTA, we needed target-specific implementations of packet-generator code. Overall, though, we found that the target-specific code could be modularized, and that having a portable test written in a common language was attractive.

A second reason for using P4 is that we found that the match-action abstractions offered by the language were well-suited to writing test code. Of course, such a statement is a matter of taste. But, our experiences were that creating tests in P4 was relatively simple, and that they provided a nice high-level abstraction over hardware.

However, we are also using P4 for a task for which it was not intended. And, as a result, there are some language extensions which could be added to P4 to make it more amenable for use with testing. An obvious first step is to extend P4 with language features that allow users to generate specific types of packets. In other words, to provide a programmatic interface for packet generation.

PTA is designed to provide a programmable test framework with an internal view of a network device. However, this internal view is hampered by the closed-source nature of hardware solutions, e.g.,

modules generated by SDNet are encrypted. Testing would be improved if users could set hooks within the code or access the state or values of certain language constructs. A hook "breaks" the data plane structure, since it allows users to inspect status at a certain point within the design. Such extensions to the P4 language would allow testing in case of a failure, or when the pipeline is stuck.

Supporting watch-points and stepping through code are required future contributions in the field of programmable network devices. Some of the bugs introduced in this work, such as the performance limitation identified in the INT design, can not be easily tracked and tested today even by compiler vendors, with full access to the code.

9.2 Under-Specification as a Source of Bugs

Because enforcing a uniform behavior on all hardware targets is impractical, some functionality is compiler specific (e.g., uninitialized values). Under-specification in the language may lead to bugs (Section 8) or security vulnerabilities [14].

Another form of under-specification, i.e., in the interface and division of responsibilities between the data plane and the rest of the device, can also cause errors. Integration bugs are not uncommon in hardware design, but the problem is exacerbated where different technologies interface. This ranges from the integration of a programmable pipeline within an otherwise fixed-function switch, as well as with the integration of externs within P4 programs.

Attending to under-specification requires a closer hardware/software co-design. This is a shift from the common paradigm focusing on the end-host (e.g., operating system and NIC) to a more holistic view that also includes network switches. While such an integrated design is likely to reduce bugs, the disadvantage is that portability may be restricted.

9.3 Writing a Test Suite

One of the challenges in testing a network device is creating a comprehensive corpus of tests. Considering the bugs detected by PTA, we identify three classes of tests.

The first class of tests is the "expected" list of tests. This includes tests that were run during the design stage, and need to be validated on a newly produced target, e.g., tests written for P4v and later translated by PTA. It also includes traditional network tests, such as those using external test equipment, e.g., checking throughput under different scenarios (test #10).

A second class of tests is tests generated in a response to a bug discovered by a user, e.g., the byte swapping bug (test #01). The goal of such a test would be to (i) validate that the bug is fixed in a newer version of the program. (ii) be used as part of regression tests in future releases (iii) test deployment of bug fixes in the field. The last use case is a good example of the usefulness of PTA, as it enables testing devices in the field without physically connecting them to test equipment.

The third class of tests targets known potential points of failure that are typically hard to test. An example is testing saturation (test #02), which one would typically verify in a block-level simulation, but would be hard to trigger as part of traditional hardware validation without additional built in self text (BIST) resources. PTA enables users to craft such tests without per-test resource overhead and while specifically targeting sensitive elements in the design.

As tests are target independent in PTA, we envision building such an open-source corpus of tests for community benefit, starting with the tests included in the PTA repository [36].

9.4 Coverage and Test Case Generation

One of the advantages of network tests is that they can be run at line rate. On ASIC, that means exhaustive testing is feasible, since billions of packets with billions of header values can be tested every second. On the NetFPGA SUME platform, the supported packet rate is about sixty million packets per second. While these numbers are high, they are insufficient to fully cover all potential cases. For example, to test all combinations of 48bit source Ethernet MAC header, it would take about eight hours on a switch capable of processing ten billion packets per second. Testing the combination of both source and destination MAC header would take $\times 2^{48}$ longer. As switches often need to drop packets where the source and destination address are the same, or some forbidden MAC addresses, this is not a crazy scenario. Note that with PTA, users can write such exhaustive tests, or write tests using random field values (e.g., source and destination MAC address), as well as specific scenarios (identical source and destination MAC addresses). The advantage of PTA is that there is no need to write new P4 code for these different scenarios. One can simply re-configure the test via register access.

10 LIMITATIONS OF PTA

Implementation. Our two implementations present different facets of PTA. The implementation on NetFPGA uses P4_16, demonstrates full integration with a device's data plane, enables in-field testing (e.g., for smartNIC applications), but lacks the performance and realism of commercial ASICs. The implementation on Tofino demonstrates the feasibility of using PTA to detect bugs on commercial ASICs, and supports full line rate, but does not support in-device integration (as the ASIC's architecture is fixed), nor in-field testing. As we show in Section 7, it does allow detecting data plane bugs.

Test Generation. With PTA, users must still define tests themselves. PTA also helps map logical tests to physical hardware tests, as in the case of P4v-to-PTA. The problem of defining tests is a long standing research problem [3, 26, 39] which is beyond the scope of this work.

Bugs in PTA. Despite best efforts, there is no guarantee that PTA is bug free. To reduce the likelihood of bugs, we separated PTA's infrastructure from users' data plane and architecture, and validated it independently using external tools (Section 7). We note that a bug in a user program will not affect PTA's operation. Similarly, a bug in PTA will not affect a user program. In this case, PTA's result may be incorrect.

Portability. Porting PTA between targets is not effortless. Different targets require changes to the hardware infrastructure, and require validating again PTA's infrastructure (e.g., traffic generation performance and correctness). It may also require changes to the control plane, as different targets use different interfaces.

11 RELATED WORK

PTA is related to programmable network programming languages, network testing, and data plane verification.

Network Programming Languages. Developers specify the packet-processing behavior of re-configurable ASICs using a variety of domain specific programming languages. Notable examples include Huawei's POF [40], Xilinx Labs' PX [8], Broadcom's NPL [2], and the P4 Consortium's P4 [6]. PTA is implemented in P4, but there is nothing that inherently depends on P4. It could be ported to any of the above languages.

Network Testing. There has been significant prior work in both industry and academia on testing fixed-function switches. The most similar to PTA is the service activation test (SAT) [15]. SAT, used by carrier Ethernet service providers, is intended to ensure that network services are configured as specified and meet the predefined Service Acceptance Criteria (SAC). SAT uses packet injection as a means to test the service, but it is closely defined and not programmable, covering only a limited set of the aspects enabled by PTA. More low-level approaches, such as ATPG [13], focus on manufacturing faults rather than bugs. FPGA debug tools, such as ILA [43], enable limited functionality testing, far less than the tests described in this paper, and are timing-affecting.

Network Testing Using P4. In-band network telemetry (INT) [20] and postcards [17] use programmable network hardware for monitoring and network-level diagnostics. In contrast, PTA focuses on detecting bugs on a device, in a compiler, or in the logic of a data plane program. Moreover, PTA is active, rather than passive, meaning that it will generate specific packets to facilitate ad-hoc and exploratory analysis.

Data Plane Verification. There are several recent projects on P4 program verification, including Vera [42], P4v [26], and Assert-P4 [16]. The details of the approaches differ, but essentially, all of these systems translate P4 programs and some control plane state into a logical formula, and use techniques such as symbolic execution [9] to check that correctness properties are not violated (e.g., a header field in a packet is not accessed if it has not been parsed). PTA complements these efforts by providing runtime testing. PTA provides grey box testing, as often only partial information exists on the data plane under test. Grey box testing has been further motivated by prior works on router and network level, such as NetSonar [48].

12 CONCLUSION

We have presented PTA, a portable test architecture for testing data planes. PTA leverages both the P4 language and hardware design to provide flexibility and visibility into programmable network devices. We have built a prototype of PTA, and used it to detect numerous hard-to-find bugs. PTA addresses an urgent need for improved tools and techniques for data plane testing and verification.

Acknowledgments. We thank the NetFPGA core development team who helped us develop and debug PTA. We thank the anonymous shepherd and reviewers, who helped us improve this paper. We acknowledge the support from the Swiss National Science Foundation (SNF) (project 407540_167173).

REFERENCES

[1] 2017. Undefined behaviors - $P4_{16}$ Language Specification. https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html#sec-undefined-behaviors.

[2] 2019. Introduction to NPLSpec. https://github.com/nplang/NPL-Spec.

[3] Andrei-Alexandru Agape and Madalin Claudiu Danceanu. 2018. *P4Fuzz: A Compiler Fuzzer for Securing P4 Programmable Dataplanes*. Technical Report. Aalborg University. https://projekter.aau.dk/projekter/files/281195651/Master_Thesis_Project___P4_Fuzzer.pdf

[4] Gianni Antichi, Muhammad Shahbaz, Yilong Geng, Noa Zilberman, Adam Covington, Marc Bruyere, Nick McKeown, Nick Feamster, Bob Felderman, Michaela Blott, et al. 2014. OSNT: Open source network tester. *IEEE Network Magazine* 28, 5 (2014), 6–12.

[5] Barefoot-Networks. 2018. Barefoot Tofino. https://barefootnetworks.com/products/brief-tofino/

[6] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: Programming Protocol-Independent Packet Processors. *SIGCOMM Computer Communication Review (CCR)* 44, 3 (July 2014), 87–95.

[7] Gordon Brebner. 2018. Extending the range of P4 programmability. Keynote.

[8] Gordon Brebner and Weirong Jiang. 2014. High-speed packet processing using reconfigurable computing. *IEEE Micro* 34, 1 (2014), 8–18.

[9] 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)*. 209–224. http://dl.acm.org/citation.cfm?id=1855741.1855756

[10] Kevin Camera, Hayden Kwok-Hay So, and Robert W. Brodersen. 2005. An Integrated Debugging Environment for Reprogrammble Hardware Systems. In *Proceedings of the Sixth International Symposium on Automated Analysis-driven Debugging (AADEBUG'05)*. 111–116. http://doi.acm.org/10.1145/1085130.1085145

[11] Huynh Tu Dang, Marco Canini, Fernando Pedone, and Robert Soulé. 2016. Paxos Made Switch-y. *SIGCOMM Computer Communication Review (CCR)* 44 (April 2016), 87–95.

[12] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. 2015. NetPaxos: Consensus at Network Speed. In *ACM SIGCOMM Symposium on SDN Research*.

[13] Pierre Duhamel and J Rault. 1979. Automatic test generation techniques for analog circuits and systems: A review. *IEEE transactions on Circuits and Systems* 26, 7 (1979), 411–440.

[14] Mihai Dumitru, Dragos Dumitrescu, and Costin Raiciu. 2020. Can we exploit buggy P4 programs?. In *ACM SIGCOMM Symposium on SDN Research*.

[15] Metro Ethernet Forum. 2014. Carrier Ethernet Service Activation Testing (SAT), Technical Specification MEF48. https://www.mef.net/Assets/Technical_Specifications/PDF/MEF_48.pdf.

[16] 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)*. Article 4, 7 pages. http://doi.acm.org/10.1145/3185467.3185499

[17] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Mazières, and Nick McKeown. 2014. I know what your packet did last hop: Using packet histories to troubleshoot networks. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 71–85.

[18] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W Moore, Gianni Antichi, and Marcin Wójcik. 2017. Re-architecting datacenter networks and stacks for low latency and high performance. In *SIGCOMM*. ACM, 29–42.

[19] Stephen Ibanez, Gordon Brebner, Nick McKeown, and Noa Zilberman. 2019. The P4-> NetFPGA Workflow for Line-Rate Packet Processing. In *FPGA*. ACM, 1–9.

[20] Inband Network Telemetry 2017. Inband Network Telemetry (INT). https://github.com/p4lang/p4factory/tree/master/apps/int.

[21] Ixia. 2019. IxNetwork. https://www.ixiacom.com/products/ixnetwork.

[22] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. 2018. NetChain: Scale-Free Sub-RTT Coordination.. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[23] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *ACM Symposium on Operating Systems Principles (SOSP)*.

[24] Sándor Laki. 2016. High-Speed Forwarding: A P4 Compiler with a Hardware Abstraction Library for Intel DPDK. In *3rd P4 Workshop*.

[25] Peilong Li and Yan Luo. 2016. P4GPU: Accelerate Packet Processing of a P4 Program with a CPU-GPU Heterogeneous Architecture. In *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*.

[26] 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)*. 490–503. http://doi.acm.org/10.1145/3230543.3230582

[27] John W Lockwood, Nick McKeown, Greg Watson, Glen Gibb, Paul Hartke, Jad Naous, Ramanan Raghuraman, and Jianying Luo. 2007. NetFPGA–an open platform for gigabit-rate network switching and routing. In *MSE*. IEEE, 160–161.

[28] Meltdown and Spectre 2020. Meltdown and Spectre. https://meltdownattack.com.

[29] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. 2017. SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM))*. 15–28. http://doi.acm.org/10.1145/3098822.3098824

[30] Netronome. 2016. Intelligent Server Adapters. http://open-nfp.org/media/Netronome_NFP-6480_2x40GigE_Product_Brief_11-15_laQID9Y.pdf.

[31] 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 2018, Los Angeles, CA, USA, March 28-29, 2018*. 5:1–5:7. https://doi.org/10.1145/3185467.3185497

[32] P4 16 Language Specification 2018. $P4_{16}$ Language Specification Version 1.1.0. https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html.

[33] p4bm 2015. The P4 Reference Switch. https://github.com/p4lang/behavioral-model.

[34] Peter Hoose 2011. Monitoring and Troubleshooting: One Engineer's rant. https://archive.nanog.org/meetings/nanog53/presentations/Monday/Hoose.pdf.

[35] Portable Switch Architecture (PSA) 2019. $P4_{16}$ Portable Switch Architecture (PSA). https://p4.org/p4-spec/docs/PSA.html.

[36] PTA 2020. PTA, Blinded Reporsitory. https://github.com/pta-project-repo/pta-artifacts.

[37] Murali Ramanujam and Noa Zilberman. 2018. Towards a highly scalable network tester. In *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. 154–155.

[38] Muhammad Shahbaz, Sean Choi, Ben Pfaff, Changhoon Kim, Nick Feamster, Nick McKeown, and Jennifer Rexford. 2016. PISCES: A Programmable, Protocol-Independent Software Switch. In *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*.

[39] Bhargava Shastry, Markus Leutner, Tobias Fiebig, Kashyap Thimmaraju, Fabian Yamaguchi, Konrad Rieck, Stefan Schmid, Jean-Pierre Seifert, and Anja Feldmann. 2017. Static program analysis as a fuzzing aid. In *RAID'17*. 26–47.

[40] Haoyu Song. 2013. Protocol-oblivious Forwarding: Unleash the Power of SDN Through a Future-proof Forwarding Plane. In *Workshop on Hot Topics in Software Defined Networking*. 127–132.

[41] Spirent 2020. Spirent. https://www.spirent.com/.

[42] Radu Stoenescu, Dragos Dumitrescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. 2018. Debugging P4 Programs with Vera. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '18)*. 518–532. http://doi.acm.org/10.1145/3230543.3230548

[43] Charles Stroud, Eric Lee, Srinivasa Konala, and Miron Abramovici. 1996. Using ILA testing for BIST in FPGAs. In *Test Conference*. IEEE, 68–75.

[44] Kashyap Thimmaraju, Bhargava Shastry, Tobias Fiebig, Felicitas Hetzelt, Jean-Pierre Seifert, Anja Feldmann, and Stefan Schmid. 2018. Taking control of sdn-based cloud systems via the data plane. In *ACM SIGCOMM Symposium on SDN Research*. 1–15.

[45] Yuta Tokusashi, Huynh Tu Dang, Fernando Pedone, Robert Soulé, and Noa Zilberman. 2019. The Case For In-Network Computing On Demand. In *Proceedings of the Fourteenth EuroSys Conference 2019 (EuroSys '19)*. Article 21, 16 pages. http://doi.acm.org/10.1145/3302424.3303979

[46] Han Wang, Robert Soulé, Huynh Tu Dang, Ki Suh Lee, Vishal Shrivastav, Nate Foster, and Hakim Weatherspoon. 2017. P4fpga: A rapid prototyping framework for p4. In *Proceedings of the Symposium on SDN Research*. ACM, 122–135.

[47] Xilinx. 2014. SDNet. http://www.xilinx.com/products/design-tools/software-zone/sdnet.html.

[48] Hongyi Zeng, Ratul Mahajan, Nick McKeown, George Varghese, Lihua Yuan, and Ming Zhang. 2015. Measuring and troubleshooting large operational multipath networks with gray box testing. *Mountain Safety Res., MSR-TR-2015-55* (2015).

[49] Yu Zhou, Zhaowei Xi, Dai Zhang, Yangyang Wang, Jinqiu Wang, Mingwei Xu, and Jianping Wu. 2019. HyperTester: high-performance network testing driven by programmable switches. In *ACM International Conference on Emerging Networking Experiments and Technologies (CoNEXT)*. 30–43.

[50] Noa Zilberman, Yury Audzevich, G. Adam Covington, and Andrew W. Moore. September 2014. NetFPGA SUME: Toward 100 Gbps as Research Commodity. *IEEE Micro* (September 2014).

A EXAMPLE PTA TEST

To further illustrate the use of PTA, this section provides an in depth description of how a user would use the framework to detect the bug described in Section 1 (test #11 in Table 4).

The bug reveals itself when the traffic rate exceeds a threshold. So, to check for this bug, a user of PTA will write a test that injects traffic at increasing rates until either the data plane reaches peak performance (i.e., PASS) or packet drops are detected (i.e., FAIL). We note that this is an *internal* test, so traffic injection is independent of the external interfaces.

To perform the test, the user of PTA only needs to write a short Python script, using a set of libraries provided by our framework. The PTA libraries include a Python module, named `test_mod`, which provides functions for interfacing with the underlying hardware platform, and a set of Bash shell scripts, which are called by the library modules and execute register access to the device.

The user-supplied Python script appears in Figure 4. The main logic of the test is repeated for increasing traffic rates (line 7). Each iteration first initializes the counters (line 9) and registers (line 10). Then, it sends a fixed number (i.e., 1000000000, line 4) of packets of a certain size (i.e., 128B, line 5) at a given rate (i.e, `rate`, line 7). Finally, the test checks that the number of packets received is equal to the number sent (lines 13-17).

For this particular bug, the assertion in line 14 is the important one. It states that the number of packets exiting the Input Arbiter module (see Appendix C) must be equal to the number of generated packets. In other words, it checks if any packet drops occur in the Input Arbiter module. Similarly, the assertions at line 15 and 16 check that the Output Queues module is functioning correctly.

The example test script is included in PTA's open-source code repository [36].

```
1  import test_mod
2
3  rate_list = [ ..., 5, 10, 20, 30, 40, 50 ]
4  num = "1000000000"
5  size = "128"
6
7  for rate in rate_list:
8   test_mod.load_image()
9   test_mod.init_counters()
10  test_mod.init_registers()
11  test_mod.generate_packets(num, size, rate)
12  test_mod.collect_results()
13 assertions_list = [
14 ("input_arbiter_packets_out", "EQ", num),
15 ("output_queues_packets_in", "EQ", num),
16 ("output_queues_packets_out", "EQ", num)
17 ]
18 test_mod.chk_assertions(results,assertions_list)
```

Figure 4: Example test script.

B EXAMPLE P4V-TO-PTA TEST

The P4v-to-PTA integration tool, targeting the Barefoot Tofino ASIC, automatically generates a complete test configuration from P4 source code annotated with P4v assumptions and assertions [26]. In this section, we disuss the example shown in Section 5, Figure 2 in detail.

We first describe the data plane under test, colored green in Figure 2. There is a table, `t1`, that performs an action, `a1`. The action subtracts the value in one header field, `h.s3`, from another, `h.s2`. The resulting value is stored in `h.s2`. The ingress control block simply applies table `t1`.

We assume that a user has annotated this program with assumptions and assertions, highlighted in blue text. In the example, the user specifies an assumption that states that field `s2` must have an initial value of 4 and that field `s3` must have an initial value which is less than the one of `s2`. The assertion specifies that the result of the subtract operation, stored in `s2`, is expected to be less than or equal to the initial value of 4.

With this annotated source code as input, the P4v-to-PTA tool automatically generates the entire test configuration. The generated test initializes the hardware, generates the test packets, collects raw results, and reports the results to the user. In Figure 2, we highlight the test header generator and the output packet checker, which are colored yellow.

The test header generator must inject packets that match the P4v assumption, namely that the packets have a header `h` with fields `s2` and `s3` set to values 4 and 3, respectively. We note that in some cases, as in this one, the assumption allows our tool to select a range of values. We currently use basic heuristics (i.e., the closed value to the maximum in a range) to select a concrete value. As discussed in Section 4.2, the packet header generator module is target-independent, and programmed in P4. The code block in the top of Figure 2, shows the generated P4 code.

The output packet checker must test the P4v assertions, i.e., that `h.s2 <= 4`. Again, the packet checker code is programmed using P4, as shown in the yellow code block in the bottom of Figure 2. The code includes two actions: `all_subs( )` and `check_act_1( )`. Since the stateful ALUs in the Tofino switch can only compare the value of a header field with zero, the tool splits the comparison in the assertion in two pieces: a subtraction and a comparison with zero. The first action subtracts from `s2` its initial value (`subtract_from_field(h.s2, 4)`). The second action implements the comparison through a stateful ALU (`condition_lo: h.s2 <= 0`). The outcome of the comparison is stored into a register, that can be read through the control plane of the switch (`update_lo_1_value: register_lo + 1`). The value stored in the register is incremented if the comparison is true.

C NETFPGA SUME IMPLEMENTATION

Figure 5 illustrates our open source, FPGA-based, prototype implementation, which builds upon NetFPGA SUME's reference architecture. NetFPGA's reference architecture is composed of a data plane that processes traffic arriving from four independent network interfaces and a host (over PCIe). Incoming data from the different interfaces is admitted to the data plane through an *input arbiter*. PTA taps to the architecture through a dedicated input interface and a hardware module, called "packet mirror", that provides a copy of the traffic processed by the data plane under test.

The test packet generator module is composed of two sub-modules. First, because a P4 pipeline is not capable of generating packets without a stimulus, a blank packet generator module generates blank packets, i.e., packets with no header fields and no payload. Next, a

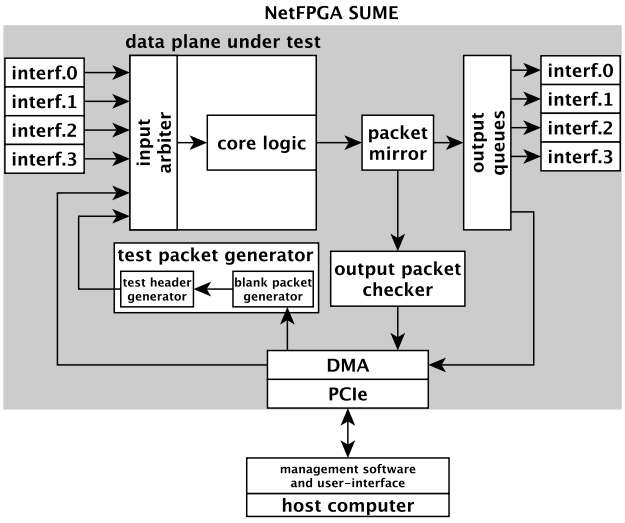

Figure 5: NetFPGA SUME implementation.

test header generator module, which is P4-programmable, modifies the blank packet to add the appropriate headers and data for the test case. Finally, the packets are injected into the data plane under test.

At the output of the data plane under test, traffic is transparently mirrored; one copy is forwarded to the output queues and interfaces of the card, while the other enters the output packet checker module. The checker module checks packet header fields, based on the P4 specification provided by the user, and updates stateful elements, e.g., packet and byte counters. Finally, test results are sent to the host computer over the PCIe interface.

PTA's tables are configured using NetFPGA's control plane infrastructure, i.e. using NetFPGA's table load and register access primitives, configured over PCIe. However, PTA does not share the control plane program with the data plane under test. All control plane accesses are done either before or after the configuration of the test subject have taken place, and not during test run time. In this manner, PTA prevents contention on PCIe or the control plane.