Static Analysis Tools For Network-Device Stacks

by

Fabian Ruffy

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

New York University

May, 2025

Professor Anirudh Sivaraman

Dedication

To my dog, my loving partner, my wonderful family, and my great friends.

Acknowledgements

My academic adventure took time—five years (eight, if you count my research master's at UBC). It was inadvertently set in motion by my cousin, who invited me to visit Boston in September 2014. That trip was my first time in the U.S., and by chance, I wandered into CSAIL uninvited and found myself in a seminar by Hakim Weatherspoon ("Can you see the IDLES?"). I thought the whole environment was fascinating—and so, I decided to go on this little excursion into the academic world. Now I can say I have successfully concluded it. Of course, I didn't go on this adventure alone. Many mentors and friends helped me along the way. Life is a series of dice rolls, and each one can be an opportunity. I'm thankful to everyone who helped me roll the dice again and again throughout my PhD.

First, I owe much thanks to Nate Foster—without him, I would not be where I am today. He gave me the opportunity to work at Intel, which not only supported me through the pandemic years but was also the place where I started my second dissertation project, P4Testgen. That work deepened my involvement with the P4 community, eventually leading me to become its Chief Technical Architect and a member of the Technical Steering Team. These roles opened doors for both academic collaborations and industry work, ultimately landing me a great job after my PhD. Nate also invited me to the Bellairs workshop—twice—hosted me at Cornell, and introduced me to colleagues who became friends. All of this taught me much about the value of academic collaboration and mentorship.

I'm also grateful to Aurojit Panda, with whom I had countless brainstorming sessions.

Panda always made time, had an open ear, gave me new ideas, and pointed me to obscure but valuable research. He was instrumental in starting my final project, Flay. I worked on Flay with his close and great collaborator, Gianni Antichi, who invited me to Milan to give a talk and to drink excellent espresso—an invitation I gladly accept again :).

A big thanks to my advisor, Anirudh Sivaraman, for taking me on despite my rather uneven test scores, for patiently tolerating my tilting at university windmills, and for giving me the freedom to explore my work with the P4 community, Intel, and Google. His meticulous focus on clarity, precision, and readability made me a better writer and taught me the value of a good editor.

My UBC Master's crew—Vaastav Anand, Amanda Levin (née Carbonari), Clement Fung, Stewart Grant, Mihir Nanavati—and of course, my UBC advisor, Ivan Beschastnikh played a major role in my academic journey, teaching me unwritten rules of academia that set me up for success in my PhD. Ivan showed me the value of taking chances, helped me understand the inner workings of American academia, and taught me how to read papers critically (and what's actually worth paying attention to). He also drilled into me the importance of structure and discipline for success.

A shoutout to my conference and internship friends—Jonathan Dilorenzo, Serhat Arslan, Steffen Smolka, Alin Tomescu, and Liangcheng Yu—for making work and conferences fun. A special thanks to Jonathan and Steffen for giving me the opportunity to work with them at Google. And to my NYU PhD colleagues—Haseeb Ashfaq, Jessica Berg, Xiangyu Gao, Jinkun Lin, Cheng Tan, Tao Wang, and Eric Yu—for commiserating through the PhD grind and keeping things light when needed.

P4 is rather central to my dissertation, and so I want to thank the P4 community, particularly Antonin Bas, Mihai Budiu, Glen Gibb, Vladimir Gurevich, Andy Fingerhut, Chris Dodd, and Han Wang for welcoming me as a contributor and humouring my endless bug reports and pull requests. This openness is not guaranteed. I also want to thank

all the people who, in one way or another, influenced or informed my work: Boris Beylin, Bili Dong, Ryan Goodfellow, Riff Jiang, Ali Kheradmand, Anton Korobeynikov, Prathima Kotikalapudi, Davide Scano, Rob Sherwood, Chris Sommers, Vladimir Still, Hari Thantry, and many more. Working with people who genuinely care about a shared goal—and often sacrifice their free time for it—has made this work deeply rewarding.

Lastly, I want to briefly acknowledge the most important part of my life, my family and friends, both in the Americas and back home in Germany. You bring much joy to my life, always keep me grounded and focused on what matters, and you give me the stability to make it through. This dissertation is dedicated to you, even though I know you will never read it. But this does not matter because each one of you already knows how deeply grateful I am. Thank you. I love you all.

Abstract

The administration of computer networks is increasingly automated, and network devices are becoming more programmable. The network-device stack—software layers dedicated to forwarding packets and interpreting instructions from the network control plane—now implements much more operational logic. The increase in complexity in logic can increase the frequency of faults, which can have an outsized impact on a computer network. Hence, network operators and device manufacturers are turning to static analysis to ensure that the device stack is both well-optimized and functionally correct. The software in network-device stacks is extensive and often written in general-purpose languages such as Python or C++. These languages contain loops, aliasing, or indirection, which can make developing effective static analysis techniques challenging.

In this dissertation, we explore an opportunity to build better static analysis tools for network-device stacks. We use P4, a domain-specific language for network programming, as our foundation. We develop an execution model for P4 that describes the behavior of a network device, and we implement this execution model using satisfiability modulo theories (SMT), expressed in quantifier-free bit vectors. We refine this execution model through three distinct projects and show its utility by adopting techniques from software engineering research that are theoretically powerful but were considered practically limited for general-purpose languages. Applying our specialized techniques, we were able to find approximately 60 bugs in network-device stacks that cause incorrect packet processing. Furthermore, we

reuse our model to optimize network programs based on their control-plane configuration, which can reduce resource usage and increase packet-processing performance.

Our SMT-based execution model for packet processing is protocol-independent, device-agnostic, and precise enough for bug-finding and program optimization. We attribute these successes to tailoring our model to a DSL specialized in packet processing while also appropriately exploiting the restrictions of this DSL. We have contributed the tools that use this model to open-source projects, and these tools are now in broader use by the community.

Contents

List of Figures

List of Tables

1 | Introduction

In a packet-switched network, fixed units of data (packets) are sent independently from end host to end host across a system of network devices. Network devices read the beginning of the packet (the header), match this information against entries in a forwarding table, and then forward the packet to the next device until it reaches its destination. Packet-switched networks were introduced in the 1960s [184] and, because this design has proven itself to be reliable and scalable, it is now the foundation of the modern Internet and data centers. However, commensurate with the scale of these networks, their operational complexity has increased substantially, and so network operators have started to reach for network verification to ensure that their networks remain reliable.

Traditionally, we verify that a packet-switched network is working correctly by treating the network as a connected graph of devices, then checking the graph for properties such as loop-freedom (are there any cycles in the forwarding paths?), reachability (can machine A communicate with machine B?), or stability (do the forwarding paths remain stable?). This type of verification is effective at testing the behavior of a network as a whole, but it assumes that individual network devices are executing forwarding rules correctly. This assumption does not always hold—and increasingly less so—because we have started to push more and more logic into the software stack controlling our network devices.

The need for network-device verification. Why is this happening? Simply put, there is now more open-source, extensible network-device software, and this software is used to build sophisticated network-device stacks. Further, these stacks are increasingly tailored to packet-switched networks with particular traffic or workload patterns. Two interrelated technological trends contribute to this development: merchant silicon [66] and programmable forwarding chips [216]. Merchant silicon refers to barebones networking chips that provide hardware acceleration for a suite of protocols (e.g., IP forwarding) but are otherwise not supplied with any software. Network operators develop their own network-device stack for merchant silicon to gain control over their networking equipment infrastructure and protocols. This desire for control was then pushed to its logical conclusion with programmable forwarding [218, 37] and the associated chips. A programmable-forwarding chip is designed to accelerate processing for arbitrary network protocols. This is done by defining network protocol operations in the form of a network program, which is compiled and then loaded into a programmable section of the networking device.

But there is a downside: network operators use merchant silicon and programmable forwarding to specialize networking infrastructure to their own requirements. With that, the accompanying software falls under their purview. They are now responsible for ensuring their software stack remains reliable. What makes this challenging is that the newly created software increases the vertical depth of the stack, i.e., it adds layers of abstraction between the intent of the operator and the actual outcome. The reason the operator is adding these layers of abstraction is to adapt and evolve their networking infrastructure quickly. However, to confidently do so given the increase in software complexity, they need rigorous testing. Traditional network verification can help here, but it is routinely insufficient because it focuses on checking distributed properties at network runtime. It is used to test packet forwarding and protocol operations coarsely, and only after the devices are deployed [9]. At that point, software errors can already be disruptive, possibly bringing down the entire

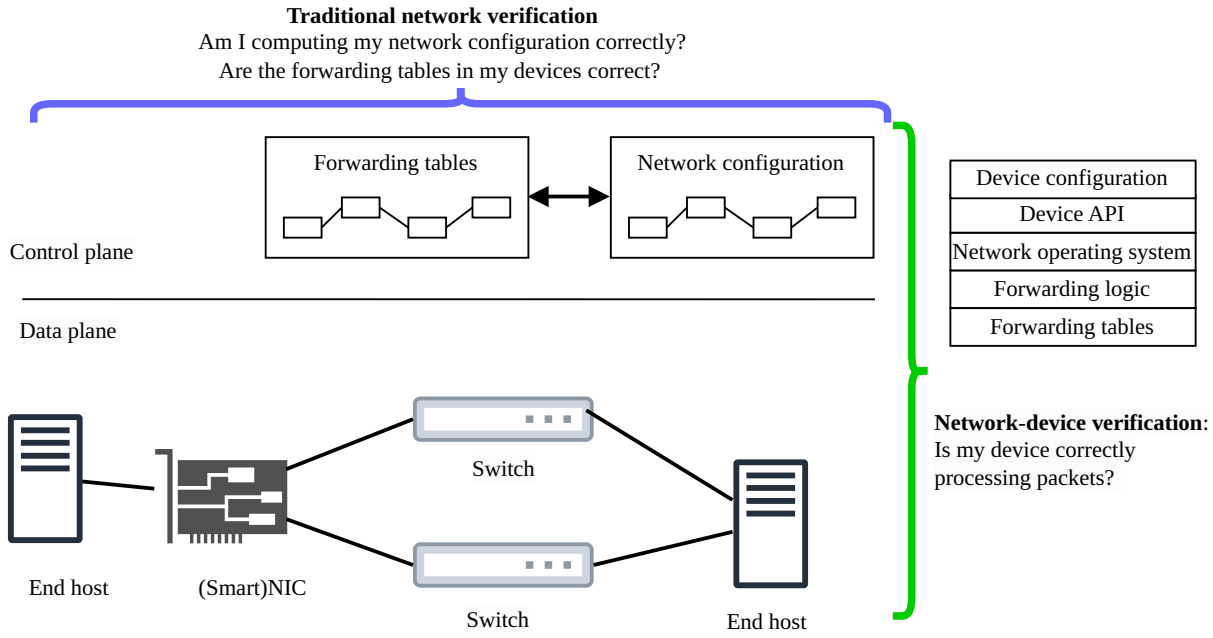


Figure 1.1: Traditional network verification and network-device verification.

network [211, 139]. Network operators such as Google, Meta, Microsoft, and Alibaba have recognized that they must trust their network-device stack before placing it in production and have begun to build specialized, automated testing tools for it [9, 211, 259, 139, 264].

The testing tools developed by these companies are a departure from traditional network verification; they are specialized toward *network-device verification*. Here, the focus shifts to the software stack of a single networking device. The emphasis is on functional correctness. For a given device configuration (this could be forwarding entries, options, or state) and a particular packet, we check whether the device behaves as expected. The goal is to capture differences between the intent of the operator and the actual implementation outcome of the network-device stack. Figure 1.1 illustrates the difference. Traditional network verification checks a network *horizontally*, whereas network-device verification does so *vertically*.

THE OPPORTUNITY OF DATA-PLANE PROGRAMMING LANGUAGES. The challenge with network-device verification, compared to traditional network verification, is that the soft-

ware we need to verify is written in general-purpose code, e.g., C++ or Python. Writing analysis and testing software for general-purpose languages is difficult because such software can be arbitrarily large and complex. On the other hand, we have a unique advantage when targeting software designed for packet processing. We can test this software by exploiting the data-plane programming languages intended to configure programmable-forwarding pipelines. These domain-specific languages (DSLs) are designed to capture the network data plane, the part of the network responsible for packet forwarding. Usually, these languages provide a set of primitives to write custom network protocols. A developer can describe how to extract information from a packet (e.g., the IP header) and how to use this information to modify, drop, or forward the packet. Further, the compiler for these data-plane DSLs enforces strict constraints to ensure that the program is executable on target devices. For example, in the P4 [26] language, it is generally not possible to allocate memory or write infinite loops, because the targeted device implements a run-to-completion model with strict time bounds. Coincidentally, these constraints, encoded in the DSL, also make developing formal methods easier, and we can reuse them for effective network-device verification.

Traditionally, the behavior of a network device was defined in standards or documentation by the device maker. The ambiguity in these documents made it difficult to develop accurate formal reasoning methods. In contrast, a data-plane DSL like P4 requires an execution model that defines packet-forwarding logic at the level of bits. The program is ultimately executed by the network device, which requires more rigorous thinking about device behavior and its interactions with higher-level software. We can take advantage of these primitives and constraints to develop analysis techniques that are both theoretically rigorous and practically feasible.

We show how this can be done in this dissertation. We use data-plane DSLs as a basis to develop methods that leverage the packet-processing computation model, adopt techniques that are theoretically powerful but considered impractical for general-purpose languages,

and use these techniques for effective analysis of network-device stacks. In our work, we use programmable-forwarding software as a basis to develop our tools and concentrate on the lower levels of the network-device stack. Specifically, we focus on three categories: how to test programmable-forwarding compilers, how to generate functional packet tests for network devices, and how to optimize packet forwarding programs under a particular network configuration.

1.1 Thesis and Contributions

Our thesis is as follows: *We can use a restricted data-plane programming language as foundation to develop effective static analysis tools for the entire network-device software stack.*

We root this thesis in the following two observations: 1) We could use static analysis tools designed for general-purpose programming languages to find bugs in network-device stacks, but these tools are usually not effective for specialized use cases such as packet processing. For example, the Klee [35] symbolic executor fails to find issues in data-plane programs because it naively explores all possible program paths, and these can easily number in the billions [222]. 2) Data-plane DSLs are explicitly designed to express the capabilities of programmable devices and implicitly encode an execution model that reflects the computational constraints of the hardware. Many network devices are pipelines designed to forward packets at wire speed in a single pass. To make this possible, operations must have a limited time budget, program memory must be preallocated, and floating-point operations are not supported. Aliasing, which is known to cause undecidability in program analysis [130], is also usually forbidden. These restrictions, particular to data-plane programming, allow us to create more robust automated reasoning tools. When we make these restrictions explicit during tool development, we can find bugs or analyze program correctness and resource usage at lower computational cost.

The challenge, of course, is that this implicit execution model needs to be made explicit within our static analysis tool. We need to develop a mapping from language constructs to a representation that is amenable to formal methods, e.g., satisfiable modulo theories (SMT) [54]. Further, while packet forwarding has a set of general behaviors, these behaviors can differ slightly from one network device to another. Whatever execution model we develop, if we want to generalize our analysis technique to many devices, we need to make this model extensible. Lastly, we want to show the utility of our approach. Hence, for any representation of packet processing we choose, we need to pick a testing and verification technique that we believe solves a particular software development problem, e.g., bug detection or performance optimization. We address these challenges in this dissertation and answer some of the questions on the choice of techniques.

1.1.1 Contributions

We describe the design and implementation of three different systems—Gauntlet, P4Testgen, and Flay. Each system has a different goal: Gauntlet finds bugs in compilers for the P4 data-plane programming language; P4Testgen produces high-fidelity input-output packet tests for network devices; and Flay automatically simplifies P4 programs to improve the resource usage of packet processing. For each system, we select a technique from general-purpose programming that we believe to be effective and adapt it to the restrictions of packet processing. Each system builds on the lessons of the previous one. From Gauntlet to P4Testgen to Flay, we reify a progressively more comprehensive execution model within an interpreter that can cover broader packet-processing semantics. Figure 1.2 provides an overview of the scope of each system's execution model. With Gauntlet, we start very small and focus only on the model of the data-plane language itself; with P4Testgen, we expand the model to capture device behavior. Finally, with Flay, we also model how control-plane configuration (e.g., forwarding entries) influences packet processing at large.

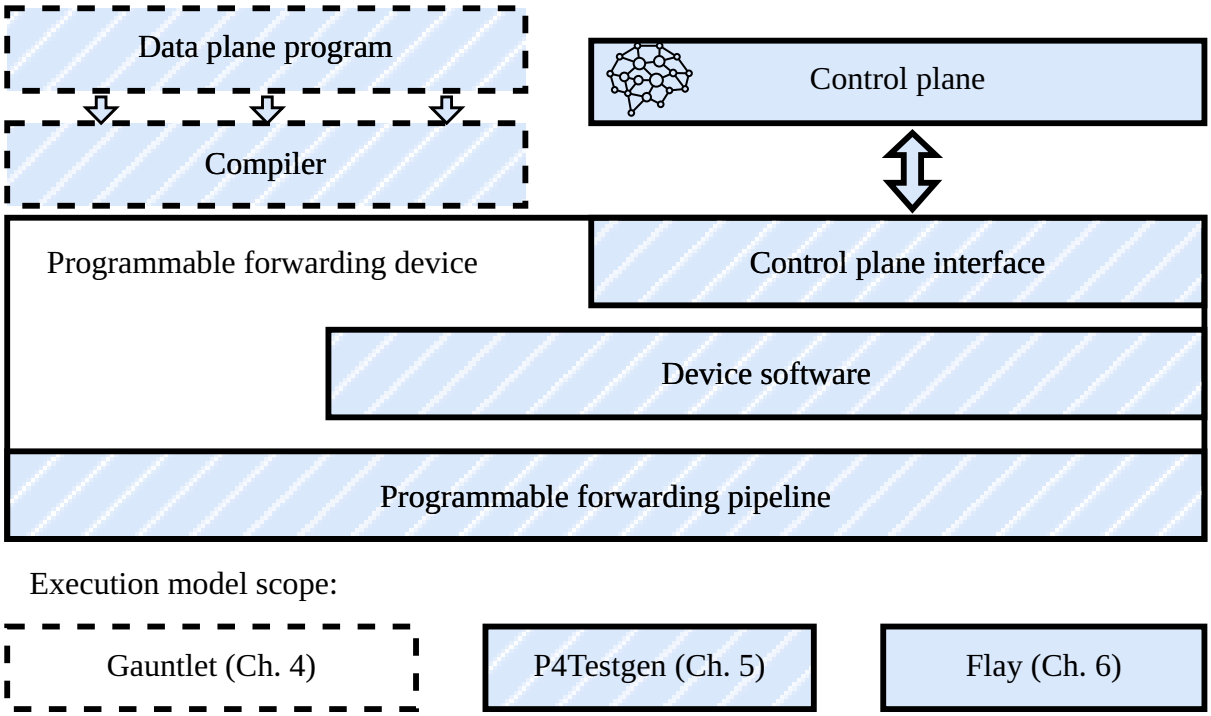


Figure 1.2: Scoping of the execution models used by the different static analysis tools.

We model the behavior of a single packet traversing a forwarding pipeline and represent the model using satisfiable modulo theories described in the quantifier-free theory of bit vectors (QF_BV) [159], a particularly efficient representation for hardware verification [19]. In Gauntlet, we use this representation for equality checking; in P4Testgen, for constraint solving; and in Flay, for fast expression simplification. We build our tools for the P4 language [26] because of its extensive open-source ecosystem. Nonetheless, the underlying ideas generalize to other data-plane DSLs, such as NPL [30], and even languages targeting the eBPF [62] virtual machine. We describe each tool in its own chapter.

1.1.1.1 Gauntlet (Chapter 4)

Gauntlet is a collection of tools specialized to test programmable-forwarding compilers. Our main achievement with Gauntlet is that we found over a hundred bugs in the front end of the official P4 reference compiler [34]. Many of these bugs were crashes, which we found

by generating specialized input programs, but almost 40 were miscompilations, where the compiler incorrectly transforms its input. This type of bug is usually difficult to find because no error is thrown. We found miscompilations with a two-pronged approach. First, we built an SMT model for the core of the P4 language that is precise enough to check whether two P4 programs are behaviorally equivalent. Second, we automatically generate valid test programs as inputs to the compiler that exercise "interesting" paths in the compiler code, something existing program fuzzers (afl [260] or p4fuzz [6]) could not do at the time. We generate these valid and interesting programs by building a grammar-guided random-program generator [261] that avoids undefined behavior and focuses on specific language constructs. For each generated program, we check whether it compiles correctly without crashing, then apply translation validation [172] to detect cases where the compiler miscompiles a program. Because of P4's language constraints, we scaled both techniques to thousands of generated programs and ultimately found 65 crash bugs and 38 miscompilations in the compiler's front end. Chapter 4 describes why finding these bugs was a non-trivial effort and how we set about developing the initial execution model.

1.1.1.2 P4Testgen (Chapter 5)

P4Testgen is a test-case oracle that produces input-output packet tests for P4-programmable network devices. What distinguishes P4Testgen is that the tool generalizes to many different programmable-forwarding devices and is simultaneously precise enough to find bugs in data-plane software. We designed P4Testgen this way to address the problem that the development of static analysis tools cannot keep up with the rate of emergence of new programmable network devices. The space of programmable forwarding is changing quickly, and for a static analysis tool to have lasting utility, it must be adaptable. With Gauntlet, we did not face this problem because we targeted the generic front end of the compiler. But for P4Testgen, to generate these tests reliably, we must also accurately model the behavior of

the network device for which we are generating tests. We accomplished this and built a tool that generated tests for more than eight different network devices, finding approximately 40 bugs across network-device stacks that are mature and heavily used. Our tool found these bugs because we generated tests with better coverage. To do so, we extend Gauntlet's execution model such that we can express the functionality of many different programmable-forwarding devices. Using this extensible execution model, we design a test oracle based on symbolic execution [13] with continuation-passing style [181] that can produce input-output tests for many different P4 programmable network devices. Notably, we can use P4Testgen to test the behavior of *any* network device as long as this behavior is expressible in P4. We describe in detail in Chapter 5 how we built this extensible execution model.

1.1.1.3 Flay (Chapter 6)

Flay specializes data-plane programs. Our insight with Flay is that a data-plane program is not complete without its control-plane configuration; once the program is configured, there are many opportunities to reoptimize the active data-plane program. Flay's goal is efficiency, and it aims to reduce the resource consumption of data-plane programs by aggressively specializing data-plane program code to the active control-plane configuration. Our preliminary experiments already show promise; for some configurations and data-plane programs, we reduced the use of stages on the Tofino switch by 20% while producing a program equivalent in behavior. Like P4Testgen, Flay builds on the previous system's execution model. In this case, we combine P4Testgen's semantics for data-plane programs with semantics for control-plane configuration. Flay can ingest a control-plane configuration specified in P4Runtime [231] format and convert this configuration into a semantic representation that is inserted into placeholders in our data-plane model. Using this combined representation, we can apply partial evaluation [114] to a data-plane program. Most importantly, because our semantics are decoupled using these placeholders, we can quickly react to changes in control-plane

configurations. This is how Flay distinguishes itself from previous tools. Because of a lack of publicly available control-plane configurations, we have also developed a fuzzing tool to generate control-plane configurations specific to different programmable-networking devices. Chapter 6 motivates why we believe a tool such as Flay is important and how we could potentially develop more specialization techniques.

1.1.1.4 Core Contribution

Ultimately, we demonstrate that it is possible to derive an SMT-based execution model for packet processing that is not tied to a particular network protocol, not restricted to a specific network device, and at the same time precise enough to find bugs and optimization opportunities in network-device stacks. We attribute these successes to using a DSL specialized for packet processing while also appropriately exploiting the restrictions of this DSL and extending it where necessary. We describe in detail in Section 3.3 how our approach distinguishes itself from other research projects dedicated to static analysis of network-device stacks.

1.1.2 Open-Source Community Contributions

We have made all the systems presented in this dissertation available as open-source projects. During the development of these tools, we have actively contributed to the P4 ecosystem[1]. Across all repositories, we have filed 217 issues (of which 142 have been closed) and 402 pull requests (of which 354 have been successfully merged).

Gauntlet is available at `https://p4gauntlet.github.io` and is running as part of the continuous integration pipeline of the official P4 compiler (P4C) [34][2]. The random-program generator, Bludgeon, has been contributed to P4C[3]. We have also contributed the tests we

[1] `https://github.com/p4lang/`

[2] `https://github.com/p4lang/p4c/pull/2458`

[3] `https://github.com/p4lang/p4c/tree/e814a334e45b78d21a2e15cd9c00e694592bb1bc/backends/p4tools/modules/smith`

generated to test Gauntlet to P4C[4].

We developed P4Testgen using P4Tools, a static analysis framework for P4 analysis tools. We initially developed P4Tools at Intel Corporation and open-sourced it as a back end of P4C[5]. P4Testgen is a P4Tools extension[6]. Members of the community have already contributed extensions to P4Testgen[7]. Flay is an independently available P4Tools extension[8]. The tool to generate random control-plane entries to test Flay is also a P4Tools extension[9].

1.2 Organization Of This Dissertation

This dissertation is split into four parts. In Part I, we briefly describe Software-Defined Networking (SDN) and how it influences the technologies for which we are building tools (Chapter 2). We also provide a brief introduction to programmable network devices and the P4 language. In parallel with SDN, research on testing networks evolved to use specifications and formal methods to test network behavior [21]. In Chapter 3, we describe some of this literature and how it relates to our work of using network programs to test network-device stacks. We then go into detail for each of the tools we have developed, in the order they were published. We split this description into two parts. Part II covers static analysis tools developed for network-device testing and describes Gauntlet, published at OSDI 2020 [196] (Chapter 4), and P4Testgen, published at SIGCOMM 2023 [195] (Chapter 5). In Part III, we switch from testing to optimization and cover Flay, published at HotNets 2024 [197] (Chapter 6). We conclude in Part IV by describing the general limitations of all these tools and possible future work using our particular style of SMT-based execution model.

[4] https://github.com/p4lang/p4c/pull/2661/files
[5] https://github.com/p4lang/p4c/tree/e814a334e45b78d21a2e15cd9c00e694592bb1bc/backends/p4tools
[6] https://github.com/p4lang/p4c/tree/e814a334e45b78d21a2e15cd9c00e694592bb1bc/backends/p4tools/modules/testgen
[7] https://github.com/p4lang/p4c/pull/5019
[8] https://github.com/nyu-systems/flay
[9] https://github.com/nyu-systems/rtsmith

Part I

Background

2 | Software-Defined Networking and Programmable Forwarding

Much of our work is influenced by Software-Defined Networking (SDN). The history, motivation, and development of SDN over the past two decades have been widely discussed. The interested reader may refer to Feamster et al., 2014 [70], Casado et al., 2019 [38], and Sivaraman, 2017 [216, §2]. In Section 2.1, we define SDN concepts we use frequently. In Section 2.2, we describe programmable packet forwarding, an academic progeny of SDN.

2.1 Software-Defined Networking: Concepts

The key driver behind SDN is the desire to control the packet-forwarding behavior of a computer network as if one were writing a program on their local machine (an idealized view of this is the one-big switch abstraction [118]). SDN is a broad term, and the research community has proposed several "tenets" intended to make network software development and automation easier [70]. We focus on two of these tenets: the logical split between data and control planes, and standardized APIs to configure network devices.

Data and control planes. The data plane is the part of the network responsible for forwarding packets as fast as possible, using rules provided by the control plane. Usually, these rules take the form of entries in device forwarding tables. The control plane computes

how a packet should travel across the network using the network topology, information provided by the data plane, and policies defined by the network operator. The control plane communicates with the data plane via an explicit API and a specific communication protocol (e.g., IPC, RPC, or REST). In this dissertation, we refer to any elements responsible for processing packets as belonging to the data plane. For example, P4 is a data-plane programming language because it allows developers to write programs to specify packet parsing and forwarding. Tables in the P4 language represent an interface to the control plane, accessible via a device-specific API. When we use the term data-plane target, we refer to any device designed to implement packet forwarding. This can also extend to software. For example, the eBPF virtual machine in the Linux kernel [62] can be considered a data-plane target because it allows developers to write and load custom packet-processing programs at various places in the kernel networking stack.

Standardized APIs. The second SDN tenet important to our work is making the decisions of the control plane independent of the underlying physical or virtual network or the device manufacturer. This implies that APIs exposed by the data plane to the control plane must be standardized and open. Defining an open API makes it possible to write control-plane programs that can generalize to many different network devices. The OpenFlow protocol [152] is an early instantiation of this idea. Its successor in spirit is the P4Runtime specification [231] designed for the P4 data-plane programming language. In our work, we use the P4Runtime specification to develop semantics for the control plane and expected device behavior, precisely because it can generalize to many different types of network devices.

2.2 Merchant Silicon and Programmable Network Devices

Traditionally, a network device makes decisions on how to forward a packet using standardized protocols defined in standards such as "Request for Comments" (RFCs). For instance, RFC 791 [173] specifies the IPv4 header format, while RFC 1716 [11] defines how devices should process these headers. Since we want to process packets at the highest possible speed, we typically implement these protocols in the data plane in fixed-forwarding pipelines—"fixed" because they support only predetermined protocol operations. For example, Figure 2.1.A shows a hypothetical fixed-forwarding pipeline. The pipeline supports three protocol headers (Ethernet, IPv4, and TCP) and two operations on packets with these headers (forwarding or dropping the packet). Packets that do not match this format are discarded. This fixed-forwarding pipeline is implemented either in hardware, within dedicated network devices such as network interface cards (NICs) and hardware switches, or in operating systems (OSs) such as Linux, with a specialized networking software stack that steers packets to the appropriate applications. Hardware network devices are produced by network device makers and sold to network operators.

Merchant silicon and programmable forwarding emerged because of tension between network device makers and network operators [82]. The tension boils down to a difference in objectives. A device maker strives to develop devices that reach a broad market segment and often tries to provide an "out-of-the-box" experience with the devices they sell. The devices they sell are tightly integrated with a proprietary software stack, which includes fixed implementations of data-plane protocols (e.g., IP [173], MPLS [243], VXLAN [143]), control-plane functionality (protocols such as BGP [180], VRF [247], LDP [240]), or certain command-line interfaces (SNMP [71], OpenConfig [199], NETCONF [64]). While a device

maker can capture a wide range of customers with this approach, and indeed many customers do not need anything else, it is not necessarily in the interest of network operators managing private networks with specific traffic patterns and network structure. The constraints of fixed-software and fixed-protocol devices can limit the ability to experiment and evolve a network [46, 15, 167, 55].

Merchant silicon. Merchant silicon [66], as sold by companies such as Broadcom and Marvell, consists of off-the-shelf silicon chips specialized for packet forwarding. These chips are effectively accelerators for packet-processing tasks such as MAC or IP forwarding, tunneling (VXLAN, GRE), or telemetry. Unlike an out-of-the-box network device, possibly sold by vendors such as Cisco, merchant silicon does not come with any proprietary software and instead is supplied with a software development kit that can be used to build a custom network-device stack. Network operators have started to build their own network-device stacks using merchant silicon. Importantly, these stacks are often open-source, which means they can be modified and extended. Examples of such networking-device stacks are SONiC [229], FBOSS [46], and PINS [228], frequently packaged as part of a software-development kit (SDK) for a network device. These kits make up the majority of what we refer to as the network-device stack.

2.2.1 Programmable Data-Plane Devices

The responsibility of the data plane is to forward packets based on user-defined rules. This usually involves two basic operations: 1) packet classification, i.e., parsing specific information from packets (e.g., the Ethernet header); and 2) packet control, i.e., forwarding, dropping, or modifying the packets. A programmable networking device contains at least one programmable block (e.g., the match-action pipeline in Tofino [25] or eBPF hooks in the Linux kernel). A programmable parser block might be a block in which a finite-state

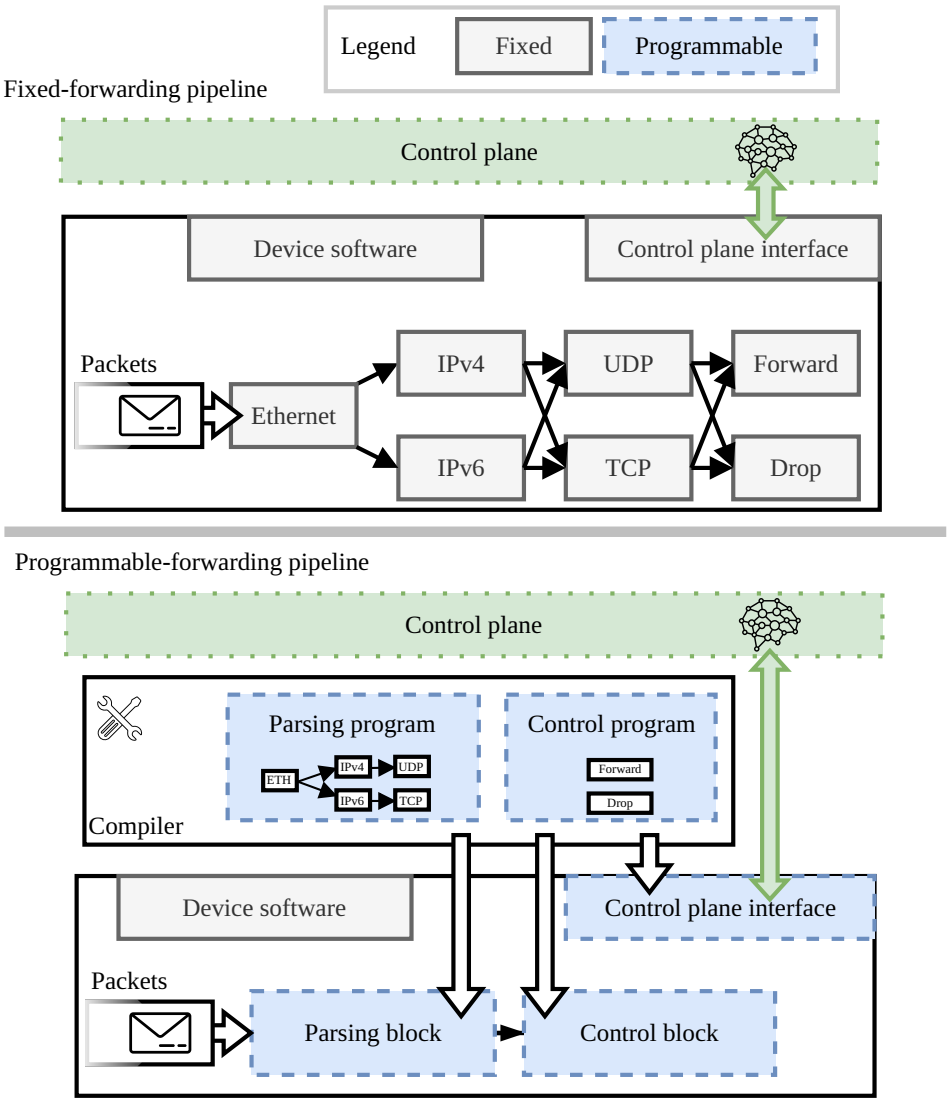


Figure 2.1: A fixed-forwarding pipeline compared to a programmable-forwarding pipeline.

machine repeatedly reads bytes from the incoming packet bit stream and writes them into header structures (e.g., IP header). A programmable control block, on the other hand, might contain operations for manipulating these header structures (e.g., setting the TTL in the IP header) or define lookup tables that can be manipulated by the control plane (e.g., an IP forwarding table matching on the destination address). Programmable blocks are configured by loading a binary produced by translating a network program written in a domain-specific

language such as $P4_{16}$ [34], eBPF written in restricted C [62], microcode [255], or NPL [30]. Network programs written in these languages also define a control-plane interface to influence the packet-processing behavior of the program during runtime. The control-plane interface is specific to each program. Prominent examples of this interface are tables in P4 or maps in eBPF. Other common instances of objects that can be changed at runtime are meters, qdiscs, or stateful registers. The intended behavior of a particular control-plane update is defined in specifications such as P4Runtime [231], OpenFlow [152], SAI [162], or NETCONF [64].

Figure 2.1 contrasts fixed-function and programmable pipeline devices. In a fixed-forwarding device, only the set of protocols and API functions defined by the device maker are available, whereas the programmable target does not constrain which protocols are supported or how the control plane should interact with these protocols. A network operator can use this to specialize packet forwarding to their own infrastructure. For example, they can accelerate network functions (e.g., firewalls, load balancers) by migrating them from software into programmable hardware, quickly roll out functional updates to the network by updating the active device program, or exploit workload-specific properties to improve power usage and performance network-wide [241, 167, 82]. Notable examples of programmable data-plane targets are the Intel Tofino [25] and XSight Labs X2 [252] switches or AMD Pensando [87], NVIDIA BlueField [251], Intel IPU [107], and Altera Agilex SmartNICs [108]. Programmable data-plane targets can also be found in pure software, for example in the form of the eBPF/XDP [100] virtual machine and TC flower [95] in the Linux kernel.

2.3 P4

One approach is to write packet-processing programs within a framework in a general-purpose language such as C, then compile and manage the programs using widely available toolchains such as GCC [220] or LLVM [132]. Examples of such frameworks include DPDK [226],

VPP [230], ClickNF [84], Snabb [165], or Bess [97]. This approach can be attractive because developers are already familiar with the programming language, the toolchains are mature, and many testing and analysis utilities are available. However, general-purpose languages such as C express a richer computational model than what modern packet-processing chips are designed to support. This can create a mismatch between the programmer's intent and the device capabilities, which in turn can lead to correctness or performance errors. To make programming network devices easier, data-plane programming languages have emerged.

The most prominent example is P4 [26]. P4 is a statically typed DSL designed to describe computations on network packet headers. It has gained traction as the primary language to both implement and specify network data-plane functionality. Proposed in 2014, the language is now widely used in industrial networks (Alibaba [241], Baidu [45]), by packet brokers (Extreme Networks [65], Cubro [50], Keysight [113]), and chip manufacturers (Intel [25], NVIDIA [124], AMD [3], Cisco [47]). We primarily use P4 in this dissertation to build our packet-processing model, and so it is helpful to provide terminology and concepts of the language we frequently refer to. We describe $P4_{16}$, the latest version of P4 [236]. Figure 2.2 summarizes the main P4 concepts, explained below.

Architectures and targets. A P4 program consists of a set of procedures; each procedure is loaded into a programmable block of the target (e.g., a switch [25], a NIC [87], or OS networking stack). These programmable blocks correspond to various subsystems such as the parser or the match-action pipeline. The *architecture* and its accompanying package describe the available programmable blocks in a target. One example of an architecture for a target is the `v1model`, which models the pipeline of a particular BMv2 [232] software switch target, referred to as "simple switch" [74]. For simplicity, we will refer to BMv2 as the target instead of simple switch in this dissertation. $P4_{16}$ can also be thought of as a family of languages, with each architecture describing a particular dialect of the language.

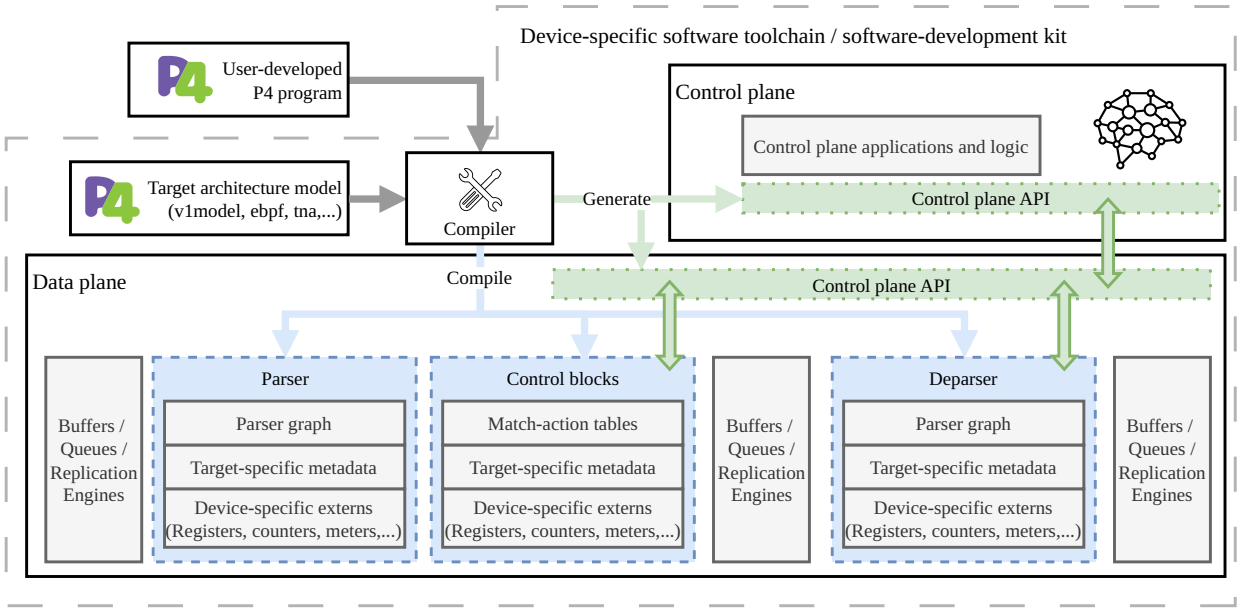


Figure 2.2: The P4 architecture model.

Each dialect will also need its own $P4_{16}$ compiler back end.

P4 COMPILERS. A P4 compiler translates a P4 program and the target architecture model into target-dependent instructions. These target instructions are combined with the non-programmable blocks (e.g., a fixed scheduler) to form the target's data plane. These instructions also specify how this data plane can be accessed and configured by the control plane (Figure 2.2). P4C [34] is the official open-source reference compiler infrastructure of the P4 language and implements the current state of the specification.

COMPILER BACK ENDS. To implement a P4 compiler for a specific target, developers write a P4C back end. This back end uses P4C's standard front-end and mid-end passes, adding its own specific transformations to translate the intermediate P4 code into instructions for the target device. Examples of production-grade P4C back ends are the back end for the Tofino [25] programmable switch chip and the BMv2 [232] data-plane software model.

Parsers and control blocks. A P4 parser is a finite state machine that transforms an incoming byte sequence received at the target into a structured representation of header definitions. For example, incoming bytes may be parsed as packets containing Ethernet, IP, and TCP/UDP headers. A deparser converts this representation back into a byte sequence. Control blocks describe the per-packet operations that are performed on the input header. These operations are expressed in the form of the core primitives of the language: tables, actions, metadata, and extern objects.

Tables. Tables are objects in the control block similar to a Python dictionary. Table entries are match-action pairs inserted by the control plane. When a table is applied to a packet traversing the control block, relevant fields from the packet header are compared against the match key of all match-action entries in the table. If any entry's key matches the extracted fields, the action associated with the match is executed. Actions are procedures that can modify state and/or input headers.

Calling conventions. P4 uses "copy-in/copy-out" [236, §6.7] semantics for method calls. For any callable object in P4, the parameter direction (also known as mode [105, §8.2]) explicitly specifies which parameters are read-only and which parameters can be modified, with the modifications persisting after function termination. Modifiable parameters are labeled with the direction `inout` or `out` in the definition of the procedure. Read-only parameters are marked `in`. At the start of a procedure call, the arguments are copied left-to-right into the associated parameter slots. Parameters labeled `out` remain uninitialized. Once the procedure has terminated, all procedure parameters labeled `inout` or `out` are copied back to the original input arguments.

Metadata. Metadata is programmer-defined or target-specific data that is associated with a packet header while it traverses the target. Examples of metadata include the packet input

port, packet length, queue depth, or priority; this information is interpreted by the target according to target-specific rules. Metadata can also be modified during the execution of the control block.

EXTERNS. Externs are an extensibility mechanism that allows targets to describe built-in functionality. Externs are object-like and have methods. Examples include calls to checksum units, hash units, counters, and meters. P4's "copy-in/copy-out" semantics allow reasoning about externs to some degree; we can discern which input arguments might be modified by the extern (those passed to `out` or `inout` parameters) and which are read-only (`in`).

3 | Network Testing Research

The main purpose of a computer network is to provide a communication service to its users, even when individual components might fail [48]. This purpose has largely remained the same over recent decades, but the operational complexity of computer networks has increased substantially. Many services people depend on are now online, increasing the demand on networks. A common aspirational goal for reliability is "five nines," with the goal to make a network available 99.999% of the time [92]. This number corresponds to only five minutes of downtime per year! The increase in demand and expectations placed on computer networks necessitates better testing methodologies to ensure that networks remain reliable. Today, network testing methodologies range from examining the outcome of sending a single packet to a machine to correctly processing billions of packets at a global scale. In this chapter, we provide an overview of various network testing techniques used over the decades. Much network testing focuses on benchmarking, not necessarily functional analysis. Since we use static analysis tools for verification and optimization, we provide only a brief overview of benchmarking and focus instead on functional verification.

3.1 Network benchmarking

Network benchmarking largely focuses on validating performance with respect to throughput, jitter, latency, and frame loss. RFC 1242 [28] represents an early attempt to establish a com-

mon terminology for evaluating the performance of networking components. Furthermore, ITU-T Y.1564 [109], RFC 2544 [27], and RFC 2889 [145] outline procedures for testing Ethernet and LAN devices. These standards discuss the correctness of packet processing and expected behavior when different protocols interact, but typically in a limited form. Correctness is usually checked by validating checksums, measuring packet loss, or checking individual protocol fields (e.g., ensuring the MAC destination address is preserved).

3.2 Network verification

In contrast to network benchmarking, network verification checks whether a single packet with specific values is forwarded correctly. Typically, this verification is performed globally, across all relevant network devices.

Traditionally, network verification is classified into two categories: data-plane and control-plane verification. Verifying the network data plane aims to ensure that packet forwarding is working correctly across the entire network. This is typically done by retrieving a snapshot of the forwarding tables of all network devices and checking whether these forwarding tables violate specific properties of interest. Anteater [144], Veriflow [122], and Header Space Analysis [120] are examples of this approach. Control-plane verification focuses on the software that instruments entries in the data plane; i.e., the goal is to ensure that the algorithms responsible for computing paths and policies in the network accurately capture the operator's intent, for example, by comparing network configurations against a specification. Batfish [31] and Minesweeper [20] are prominent tools in this space. Beckett and Mahajan [21] discuss this particular branch of network verification in depth.

3.3 Network-device verification

In contrast to network benchmarking, which sends many packets at once to measure performance, and traditional network verification, which tests the functional behavior of a network at large, network-device verification focuses on a single network device and its functional forwarding behavior. In this section, we describe past and concurrent research and how it differs from the type of verification we apply. Note, while this dissertation discusses the development of general static analysis tools, we do not cover work on automatically optimizing network programs in this section. We describe a selection of projects relevant to optimization in Section 6.5.

Validating network protocol implementations. Packet processing is largely driven by protocols, and so much early network-device verification focused on conformance testing with respect to protocol stacks [178]. This work generally used model checking to test whether a particular protocol implementation conformed to a manually written specification. An early approach involved unique input-output sequences [200, 39]. These sequences were produced from an abstract specification for a particular protocol; each sequence was then converted into input-output packet test sequences. If the observed output did not match the expected output, a flaw in the protocol implementation was likely present. Much model checking work also validated specific TCP/IP protocol stacks. For example, Bishop et al., 2005 [23] and Musuvathi and Engler, 2004 [156] validated the Linux kernel TCP implementation by writing a TCP specification and generating tests from it. The limitation of model-based checking approaches is that they can be difficult to scale or maintain for a particular protocol stack. Tools such as `packetdrill` [36] instead adopted a test-driven approach. `packetdrill` introduced a DSL of imperative commands that allows users to write scripts that inject packets and specify expected outcomes; it was intended to test an

entire protocol stack. For example, one could write a script that injects a SYN packet and expects a SYN/ACK back. This procedure already exercises much of the Linux networking stack. With the introduction of network programmability, this type of testing also evolved and moved away from being centered around a particular protocol stack (e.g., TCP/IP). A notable example is the Packet Test Framework (PTF) [233], which can be used to inject and receive packets at the bit level on arbitrary network devices (switches, NICs, OS networking stacks). Note that these test frameworks still require manual creation of packet tests. With P4Testgen, we show that we can use our model to automatically generate tests for frameworks such as PTF.

Generating single test packets. For the most part, early checking of networks was performed using tools such as `traceroute` and `ping`, which provided rough heuristics but not widespread coverage [262]. With more complex networks, interest in validating coverage and ensuring that forwarding rules across the network were working correctly grew. This type of validation typically involves generating individual test packets and sending them between hosts (e.g., from host A to host B). Importantly, these test packets were crafted to achieve specific coverage of rules or behaviors in the network. For example, Automatic Test Packet Generation (ATPG) [262] automates input packet generation to validate a configured switch network by computing packets intended to cover every switch link and table rule. If a packet traverses an unexpected path, some network element might be behaving incorrectly. Monocle [168] and Pronto [263] are similar systems. The key point is that these tools generate packets that exercise the device software stack, but they typically do so coarsely. They all use the control-plane configuration as ground truth, which allows them to check whether packets with the expected headers are forwarded out the correct ports. We target a richer data plane model than these prior approaches because we use a data-plane DSL as our model. We also focus more narrowly on a single device's software stack, not the entire network's

forwarding rules.

Verifying P4 intent. Many tools help verify P4 programs against a formal specification. Tools in this domain usually rely on assertions that model relational properties—e.g., the program does not read or write invalid headers [221, 212, 60, 140, 83, 59, 241, 213, 119]. Our goals are orthogonal to these tools. We produce tests for a P4 program but do not check the correctness of the program itself. Some of these tools [212, 140, 213, 119] can generate concrete test inputs in the form of input packets. The observed outputs for these inputs are then compared against developer-supplied assertions. In theory, with good assertions, this method can also detect bugs in a given network-device software stack. P6 [212] and Meissa [264] in particular describe "platform-dependent bugs" and "non-code" bugs, which are comparable to network-device stack bugs.

Testing programmable-network device software. There are a variety of tools that use data-plane programs as a source to generate tests for network infrastructure [9, 257, 29, 140, 212, 160, 129]. Since these tools are most closely related to P4Testgen, we describe in detail in Section 5.8 how these tools differ from our testing approach. In general, we differ by developing an execution model that can both serve as an oracle to generate tests (many packet-generation tools require developer-written assertions) and is also extensible, in contrast to tools such as SwitchV [9] or `p4v` [140], which are specialized to a particular network device.

3.4 Summary

In this chapter, we have described the historical context of network-device verification and how concurrent research compares to the tools we have developed. The model we present in this dissertation differentiates itself by combining three particular traits.

Protocol independence. We use primitives (e.g., arithmetic, key-value lookup in the form of tables, and registers for state) to describe network protocol operations, instead of modeling the behavior of a single protocol.

Device independence. We do not specialize toward testing a particular device; instead, we provide extensions that accurately model the forwarding behavior of different network devices.

The ability to predict output. Our approach produces a test oracle. We do not require a second compiler or device model to perform our testing. We also do not require developer-written assertions to generate tests. For both compiler validation and test-case generation, our execution model is sufficient to detect issues.

Our approach achieves these distinctions because we derived our SMT execution model from the P4 language. P4 is designed to describe arbitrary packet processing without being tied to particular network protocols or network devices. Furthermore, both the language specification [236] and the related control-plane specification, P4Runtime [231], are intentionally device-independent and also detailed enough to construct the semantics for an oracle.

Part II

Testing Network-Device Stacks

4 | Gauntlet: Testing The Compiler For A Network Device

An integral part of the programmable network device stack is the compiler, which translates the high-level intent described in the data-plane DSL to target-specific instructions. A data-plane DSL compiler applies domain-specific optimizations, rejects programs that are not suitable for the target, and performs appropriate resource allocation and placement.

Programmable-forwarding devices often process every packet going through the network, which makes them a critical part of the network infrastructure. However, like any piece of general software, a data-plane DSL compiler can have bugs that cause it to crash or even miscompile a program. While users may consider crashes a mere nuisance, they may not even notice a miscompiled program that persistently affects packet processing or exposes exploitable vulnerabilities. Further, it can be hard to track down miscompilations due to the lack of sophisticated debugging support on these targets. As programmable targets become increasingly common, the corresponding DSL compilers will need to be as dependable as general-purpose C compilers such as GCC [220] and LLVM [132].

In this chapter, we describe Gauntlet, a toolkit we have developed to detect bugs in compilers for data-plane targets. We introduce domain-specific techniques to detect both

abnormal termination of the compiler (crash bugs) and incorrect translation (miscompilations). We do so by generating targeted random programs, feeding these programs into the compiler-under-test, and then checking whether the translated program is semantically different from the input. We perform these checks using translation validation [172] with an SMT-based semantic model we have developed for the P4 language. We first describe approaches to testing compilers, then detail our method of random-program generation and the execution model we have developed for translation validation. The original work also used this execution model to apply a limited form of model-based testing. However, this work has been largely superseded by P4Testgen, which we describe in Chapter 5. We omit the description of Gauntlet's model-based testing in this dissertation.

4.1 Overview

Bug finding in compilers is a well-studied topic, especially in the context of C [256, 134, 224, 41, 135]. Past approaches (§4.2) to bug finding in C compilers include fuzz testing by using randomly generated C programs [256, 134], translation validation (i.e., proving that a compiler correctly translated a given input program to an output program) [158, 172], and verification of individual compiler passes [141]. These prior approaches have to contend with many difficulties inherent to a general-purpose language like C, e.g., generating random programs that avoid undefined and unspecified behavior [256, 134], providing semantics for pointers and memory aliasing [141], and inferring loop invariants and simulation relations to successfully perform translation validation [172].

Our key insight is that the restricted nature of a DSL such as P4 allows us to avoid much of the complexity associated with bug finding in general-purpose language compilers. In particular, the simpler nature of P4 (e.g., no loops or pointers) allowed us to more easily develop formal semantics, which can then be used as the basis for automated high-accuracy

translation validation. We apply a two-pronged approach with Gauntlet: random program generation and translation validation. We now describe these ideas and show how the restrictions of P4 allow them to be simpler than prior work.

First, we use random program generation (§4.4) to produce syntactically correct and well-typed P4 programs that still induce P4 compiler crashes. Because P4 has very little undefined behavior [236, §7.1.6], random program generation is considerably simpler for P4 than for C [256]. The generator does not have to painstakingly avoid generating programs with undefined and unspecified behavior, which can be interpreted differently across different compilers. The smaller and simpler grammar of P4 relative to C also simplifies the development of a random program generator.

Second, we use translation validation (§4.5) [172, 158] to find miscompilations in P4 compilers where we can access the transformed program after every compiler pass. Translation validation has been used in the context of C compilers before, but has suffered one of two shortcomings. It either needs considerable manual effort per compiler pass (e.g., Crellvm [117] requires several hundred lines of manual proof-generation code for each pass; Alive [141] requires manual translation of optimizations into the Alive DSL) or suffers from a small rate of false positives and false negatives (e.g., [98, 158]). Fundamentally, this is inevitable for unrestricted C: proving program equivalence in the presence of unbounded loops is undecidable. In our case, however, the finite nature of P4[1] makes P4 program equivalence decidable and addresses both shortcomings. Thus, our use of translation validation is both precise and fully automated, requiring manual effort only to develop semantics for the P4 language—not manual effort per compiler pass.

We applied Gauntlet to 3 platforms (§4.6): (1) the open-source P4 compiler infrastructure (P4C) [34], which serves as a common base for different P4 compiler implementations; (2)

[1]Finite in that input and output packets and state are finite bit vectors. Loops are bounded (parsing [236, §12]) or forbidden (control flow [236, §13]).

the P4 back end for the open-source P4 behavioral model (BMv2) [232], a reference software switch for P4; and (3) the P4 back end for Barefoot Tofino, a high-speed programmable switching chip [25]. Across these 3 platforms, as of June 2022, we found a total of 103 new and distinct bugs, all of which were confirmed and assigned to a compiler developer. Our efforts also led to 6 changes [236, §A.1] to the P4 specification. 98 of these bugs have already been fixed. We analyze these bugs in detail and describe where they were found, their root causes, and which commits introduced them. Gauntlet has been merged into the continuous integration pipeline of the official P4 reference compiler [185]. To our knowledge, Gauntlet is the first example of using translation validation for compiler bug finding on a production compiler as part of its continuous integration workflow. While Gauntlet has been very effective, it is still restricted in the kinds of bugs, compiler passes, and language constructs it can handle. We describe these restrictions to motivate future work (§7.1). Further, while we developed these bug-finding techniques in the context of P4, we believe the lessons we have learned (§4.6.4) apply beyond P4 to other DSLs with simpler semantics relative to general-purpose languages (e.g., the HLO IR for the TensorFlow [1] XLA compiler [239]).

4.2 Approaches to Testing Compilers

Levels of compiler testing. A compiler must reject incorrect programs with an appropriate error message and accurately translate correct programs. However, a program can be correct to varying levels. McKeeman [151] provides a taxonomy of these levels in the context of C (Table 4.1). Each level corresponds to the program getting deeper into the compiler before it is rejected (e.g., lexer, parser, type checker, optimizer, code generator). The difficulty of generating test programs also goes up with increasing input level. For instance, while general-purpose fuzzers such as AFL [260] are sufficient to stress test the lexer, more sophistication is required to generate syntactically correct and well-typed programs,

Level	Input Class	Example of incorrect input
1	Sequence of ASCII characters	Binary files
2	Sequence of words and spaces	Variable name beginning with $
3	Syntactically correct	Missing semicolon
4	Type correct	Adding int to string
5	Statically conforming	Undefined variables
6	Dynamically conforming	Program throwing exceptions
7	Model-conforming	Program producing wrong outputs

Table 4.1: McKeeman's [151] 7 levels of C compiler correctness.

which are required to test the optimizer. In the context of the P4 compiler, we observed very limited success in bug finding using a general-purpose fuzzer such as AFL. This is because testing at the first few levels of Table 4.1 is already handled adequately by P4C's test suite [34, §3.4].

Hence, we only consider programs at the higher levels: static, dynamic, and model-conforming. These are programs that pass the lexing, parsing, type checking, and semantic analysis phases of the compiler, but still trigger compiler bugs. Like Csmith [256], we categorize bugs into *crash bugs* and *miscompilations*. A crash bug occurs when the compiler abnormally terminates on an input program without producing either an output program or a useful error message. Crash bugs include segmentation faults, assertion violations, incomplete error messages, and out-of-memory errors. A miscompilation occurs when the compiler produces an output executable, but the executable's behavior is different from the input program, e.g., due to an incorrect program transformation in a compiler optimization pass. In P4, miscompilations manifest as any packet output that differs from the expected packet output given an input packet. Crash bugs we are interested in correspond to level 5 in Table 4.1; miscompilations correspond to levels 6 and 7.

Bug-finding strategies. We now look at how compiler bugs are found. A key challenge in compiler bug finding is the oracle problem. Given an input program to a compiler, the expected outcome (i.e., should it accept/reject the program and what should the output be?)

is unclear unless one consults an all-knowing oracle. Below, we outline the major techniques used to approximate this oracle knowledge.

In differential testing [151], given two compilers that both receive the same input program, if compiler A's output (after compiling and running the program) differs from compiler B's output, there is a bug in one of them. This works as long as there are at least two independent compiler implementations for the same language. Csmith [256] is one example of this approach; it feeds the same randomly generated C program to multiple C compilers and checks whether the outputs generated by executing the binary produced by each compiler differ. Another example is Different Optimization Levels (DOL) [41], which selectively omits compiler optimizations and compares compiler outputs with and without these optimization passes. If the end result differs after specific passes have been skipped or added, it points to a bug. This technique can be used in any compiler framework that supports selective omission of optimizations.

Metamorphic testing [42] can serve a similar role as differential testing, especially when multiple compilers are not readily available or optimization passes cannot be easily disabled. Instead of feeding the same input program to different compilers, different input programs that are expected to produce the same compiler output are fed to the same compiler. The run-time outputs after compiling these different input programs are compared to determine if there is a bug or not. EMI is an example of this approach [134]. Given a randomly generated C program P, and random input I to this program, EMI uses the path coverage tool `gcov` [88] to identify dead code in P when run on input I. EMI then prunes away this dead code to produce new programs P' whose output must agree with P's output when run on the input I. Then EMI compiles and runs both P and P' to check whether they indeed produce the same output when given I as input.

Translation validation is a bug-finding technique that converts the program before and after a compiler optimization pass into a logical formula and checks if both programs/for-

mulas are equivalent using a constraint solver [158, 172, 141, 265]. A failed check indicates a miscompilation. Program equivalence is an undecidable problem for Turing-complete languages such as C, requiring manual assistance to perform translation validation. Typical examples of manual assistance are (1) simulation relations, which encode correspondences between variables in two programs; and (2) loop invariants, required to prove the equivalence of programs with loops. While it is possible to just unroll loops a constant number of times [98] or learn these relations [209, 158], these techniques are not guaranteed to be precise and occasionally generate false alarms [117]. The occurrence of false alarms makes translation validation an unlikely choice for recurring use in compiler testing for general-purpose languages (e.g., for continuous integration). This is because the number of false alarms typically exceeds compiler developer tolerance [201].

4.3 Motivating Gauntlet's Design

Random program generation for crash bugs. From EMI and Csmith, we borrow the idea of generating random programs that are lexically, syntactically, and semantically correct. Unlike EMI and Csmith, however, our random program generation is simpler. It does not have to avoid undefined behavior, which, by design, is quite limited in $P4_{16}$. Further, generating programs with undefined behavior helps us flag compiler passes that might exploit undefined behavior in counter-intuitive ways [246]. We feed these randomly generated programs to the compiler to see if it generates a crash, typically a failure of an assertion written by the P4 compiler developers.

Translation validation for miscompilations. Differential and metamorphic testing allow us to compare different run-time outputs from compiled programs to detect miscompilations. However, we cannot directly apply either to P4 compilers. Differential testing

requires two or more independent compiler implementations that are comparable in their output. $P4_{16}$ compilers for different hardware and software targets are not comparable because program behavior is target-dependent [34, §2.1]. Presently there aren't multiple independent compilers for the same target. Developing an entirely new compiler exclusively for the sake of testing the existing compiler is not productive because it can only be reused for one target. Metamorphic testing [134], on the other hand, requires the use of code-coverage tools such as `gcov` to determine which parts of the program are touched by a given input. Concurrent research [127] has proposed such tools for P4, but these tools were not available when we commenced work on Gauntlet.

On the other hand, P4's domain-specific restrictions make translation validation easier relative to general-purpose languages such as C. P4 programs are finite-state and finite-time, which makes program equivalence decidable at a theoretical level. At the practical level, P4's lack of pointers, memory aliasing, and unstructured control flow (e.g., goto) allow for easier generation of language semantics. Furthermore, using an SMT solver together with translation validation is more precise than randomized testing approaches such as EMI and Csmith because the solver exhaustively searches over all packet inputs to a program to find miscompilations.

To perform translation validation, we convert P4 programs before and after a compiler pass into logic formulas and assert equivalence of these formulas. To do so, we could have converted P4 programs into C code and then asserted equality using Klee's equivalence-checking mode [35]. However, instead, we directly converted P4 programs into logic formulas in Z3 [53] for two reasons. First, the effort to convert P4 to semantically equivalent C is about the same as producing Z3 formulas directly. The difficulty lies in correctly formalizing all the language constructs of P4, not in the output format. Second, generating Z3 formulas directly gives us more control and allows us to leverage domain-specific techniques to optimize these formulas.

4.3.1 Goals and Non-Goals

Find many, but not all bugs. Our goal is to find many crash bugs and miscompilations in the P4 compiler, but our tool is not exhaustive. Specifically, we do not intend to build or replace a fully verified compiler like CompCert [136], given the large labor and time cost associated with such an undertaking with respect to the breadth of P4 back ends. We want to strengthen existing P4 compilers, not write a safe replacement.

Check the compiler, not the programmer. We are not verifying that a particular P4 program is devoid of certain kinds of bugs. This problem is addressed by orthogonal work on P4 program verification [140, 59, 221, 83, 63] and P4 testing [212]. Although Gauntlet can in principle be used for verifying a P4 program, we have not designed it for such use cases. The random programs we generate to find bugs in the P4 compiler are much smaller and more targeted than a typical P4 switch program. Our tool does not need to be able to generate and efficiently solve Z3 formulas for large P4 programs to tease out compiler bugs, although it achieves acceptable performance on large programs (Table 4.4).

Unlike p4v [140] and Vera [221], whose goal is to provide semantics to find bugs in large programs such as `switch.p4`, we have developed our semantics for efficient equality checks of diverse, but relatively small, P4 programs. Because of this difference in goals, we believe our semantics cover a broader set of P4 language constructs and corner cases than p4v and Vera—broad enough that we have found bugs in the P4 specification.

Develop target-independent techniques. With Gauntlet we primarily target the front end of the compiler. We designed our tools to be as target-independent as possible and specialize them to test the front and mid ends of the compiler. While we support restricted forms of back-end testing, we do so in a way that allows us to quickly integrate and adapt to new back ends without having to understand detailed target-specific behavior. In particular,

we do not cover target-specific semantics such as externs [236, §4.3]. We do this by generating programs that are defined in a target-neutral manner with respect to $P4_{16}$'s semantics, i.e., we avoid generating target-specific extern calls. We approach this problem with P4Testgen in Chapter 5.

Only test mature compilers. We only test mature compilers such as P4C and the corresponding behavioral model[2] as well as the commercial Tofino compiler. For example, P4C supports other back ends such as the eBPF [242], uBPF [164], TC [95], and PSA [163] targets, which are pre-alpha quality and preliminary compiler toolchains. Finding bugs is likely unhelpful for the respective compiler developers at this moment.

4.4 Random Program Generation

Gauntlet's random program generator produces valid $P4_{16}$ programs to directly trigger a crash bug. If these programs do not cause a compiler crash they serve as input for our translation validation technique.

4.4.1 Design

We require diverse input programs to exercise code paths within many compiler passes—and hence bugs in those passes. P4C already contains a sample of over 1000 programs as part of its test suite. During testing, the reference outputs of each of the test programs are textually compared to the actual outputs after the front- and mid-end passes to check for regressions [34, §3.4]. However, this comparison technique is inadequate for miscompilations. Further, these programs are typically used to test the lexer and parser, not deeper portions of the compiler.

[2]Both have entered "permanent beta-status" since November 2019: https://github.com/p4lang/p4c/issues/2080

P4Fuzz [6] is a tool that can generate random P4 programs. However, when we tried using P4Fuzz, we found that the programs generated by it are not complex enough to find a large number of new crash bugs or miscompilations. For example, P4Fuzz generates programs with complex declarations (e.g., `structs` within `structs`), but does not generate programs with sufficiently complicated control flow. Hence, it does not cause P4C to execute a diverse set of compiler passes. We developed our own generator for random P4 programs that works by generating random abstract syntax trees (ASTs). With this generator we can exercise the majority of language constructs in P4. This leads to diverse test programs covering many combinations of P4 expressions. We can use these test programs to find programs that lead to unexpected crashes.

Gauntlet's random program generator is influenced by Csmith [256] and follows its philosophy of generating only well-formed input programs that pass the lexer, parser, and type checker. The generator grows an AST corresponding to the random program by probabilistically determining what kind of AST node to add to the AST at each step. By adjusting the probabilities of generating each AST node, we can steer the generator towards the language constructs we want to focus on. We can also use these probabilities to keep the size of the average generated program small, in both the number of code lines and program paths. With this technique we can find an ample number of miscompilations while also avoiding programs with too many paths; such "branchy" programs pose challenges for translation validation.

Undefined behavior. We differ from Csmith in the treatment of undefined behavior. Whereas Csmith tries to avoid generating expressions that lead to undefined behavior, we accommodate such language constructs (e.g., reading from variables that are not initialized). We record the output affected by undefined behavior as part of the logic formulas that we generate from P4 programs during translation validation (§4.5.2). These formulas allow us to track changes in programs with undefined behavior across compiler passes, which we

use to inform compiler developers of suspicious—but not necessarily incorrect—compiler transformations [246].

4.4.2 Implementation

We implement our random P4 program generator as an extension to P4C. The generator uses the intermediate representation (IR) of P4C (P4C-IR) to automatically grow an abstract syntax tree (AST) by expanding branches of the tree at random. For example, a block statement may generate up to (say) 10 statements or declarations, which in turn may result in further sub-nodes. The generated P4C-IR AST is then converted into a P4 program using P4C's `ToP4` module. Our random program generator can be specialized towards different compiler back ends by providing a skeleton of the back-end-specific P4 architecture, back-end-specific restrictions, and which architecture blocks are to be filled in with randomly generated program snippets. We have currently implemented two back ends for our random program generator corresponding to the BMv2 [74] and Tofino [25] targets.

Programs generated by our random program generator are required to be syntactically sound and well-typed. Our aim is not to test if P4C can correctly catch syntax and type errors (levels 3 and 4 of Table 4.1). If P4C's parser and type checker (correctly) reject a generated program, we consider this to be a bug in our random program generator. For example, if an action parameter has an `inout` or `out` qualifier, only writable variables may be passed as arguments.

4.5 Translation Validation

To detect miscompilations, we employ translation validation [172], a classic technique from the compiler literature in which an external tool certifies that a particular compiler pass has correctly transformed a given input program.

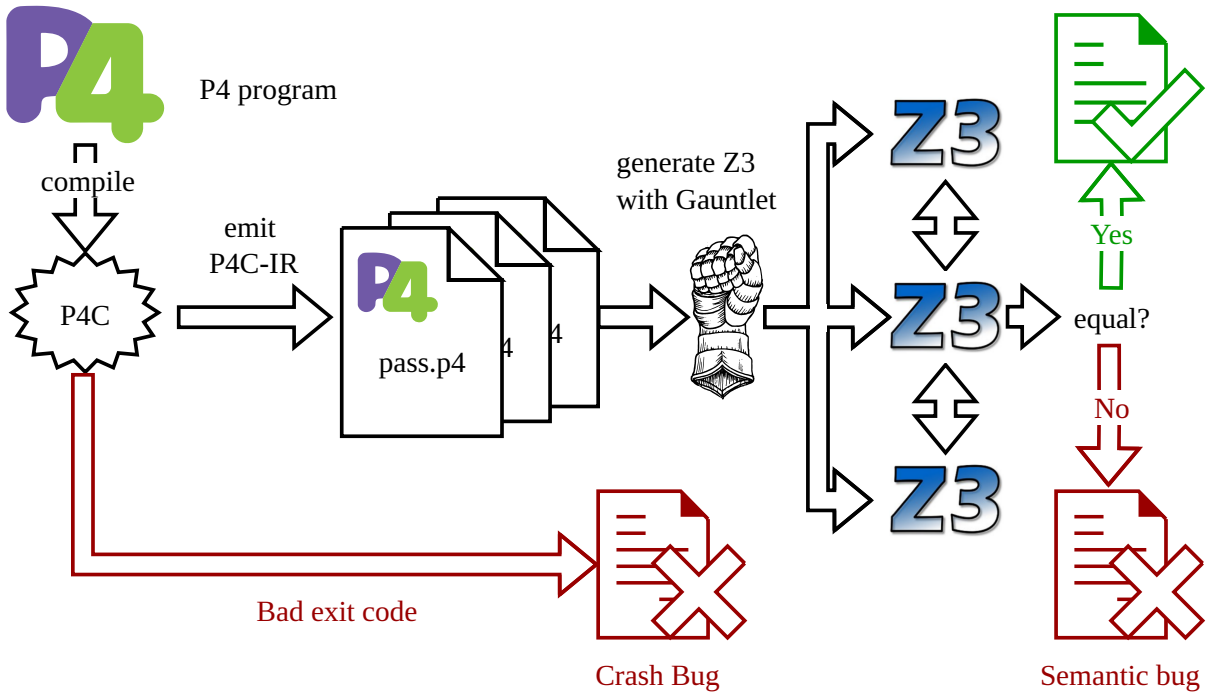


Figure 4.1: Translation validation in Gauntlet.

4.5.1 An Execution Model For P4

To perform translation validation, we developed an execution model for the $P4_{16}$ language in the form of an interpreter that translates P4 programs into Z3 formulas [53]. Figure 4.1 describes our workflow. To validate a P4 program, the symbolic interpreter converts the program into a Z3 formula capturing its input-output semantics. An equivalence checker then submits the Z3 formulas of a program before and after a compiler pass to the Z3 SMT solver. The solver tries to find an input that violates equivalence of these two formulas. If it finds such an input, this is a miscompilation. Translation validation has two advantages over random testing. First, it can accurately detect subtle differences in program semantics without any knowledge about expected input packets or table entries. Second, when we can access intermediate P4 programs after each compiler pass, we can pinpoint the erroneous pass.

4.5.2 Implementation

Like our random program generator, we wrote the interpreter as an extension to P4C. We use the IR generated by the P4C parser to determine the semantics of a P4 program. Each programmable block of a P4 architecture represents an independent Z3 formula. For example, the `v1model` architecture [74] of the BMv2 back end has 6 different independent programmable blocks: `Parser`, `VerifyChecksum`, `Ingress`, `Egress`, `ComputeChecksum`, and `Deparser`. For each block, we generate a separate Z3 formula.

Developing the execution model for the symbolic interpreter. Overall, it took us 5 months of implementation effort until our interpreter execution model was refined enough to find new miscompilations in P4 compilers, instead of encountering false alarms that were actually bugs in our interpreter. The fact that P4C contains a sizeable test suite [34, §3.4] was helpful in stress testing our interpreter during development. We started our development process by performing translation validation on programs in the P4C test suite. A miscompilation on one of these test programs is probably a false alarm and a bug in our interpreter. This is because it is unlikely that the compiler miscompiles test suite programs. The reference outputs of each test after the front- and mid-end passes are tracked as part of regression testing, and the reference outputs themselves are audited by the compiler developers. We also continuously consulted with the compiler developers to ensure our understanding of the language semantics was correct.

However, we quickly realized that we also needed to generate random programs to achieve coverage and truly stress test our symbolic interpreter. Subsequently, we co-evolved the interpreter with our generator. We attribute part of our success in finding bugs to this development technique, since it forced us to consider many edge cases—more than P4C does. The test suite for our interpreter now has over 1200 P4C tests plus over 100 of our own tests.

Eventually, our interpreter had become complete and trustworthy enough to perform

```
1 struct Hdr { bit<8> a; bit<8> b; }
2
3 control ingress(inout Hdr hdr) {
4   action assign() { hdr.a = 1; }
5   table t {
6     key = hdr.a : exact;
7     actions = {
8       assign();
9       NoAction();
10    }
11    default_action = NoAction();
12  }
13  apply {
14    t.apply();
15  }
16 }
```

(a) Simplified P4 program applying a table.

```
Input:  t_table_key, t_action, hdr
Output: hdr_out

hdr_out =
    if (hdr.a == t_table_key) :
        if (1 == t_action) : Hdr(1, hdr.b)
        otherwise : Hdr(hdr.a, hdr.b)
    otherwise : Hdr(hdr.a, hdr.b)
```

(b) Functional semantic representation in Z3.

Figure 4.2: A P4 table converted to Z3 semantics.

translation validation for randomly generated programs so as to trigger miscompilations in P4C. After we had detected the first miscompilation, we randomly generated around 10000 programs every week and added the resulting compiler bugs to our backlog. Adding support for new P4 language features as part of random program generation typically first led to a crash in our interpreter. After we fixed our own interpreter, we were frequently able to find new miscompilations in the P4 compiler that pertained to those language features. Because any of the compiler passes may have bugs, our symbolic interpreter does not rely on any compiler pass of P4C. It only relies on the P4C parser and the `ToP4` module to produce P4 code from the IR. Hence, we designed our interpreter to handle any P4 program that successfully passed the P4C parser, i.e., before the program is desugared into any normalized form. This allows us to detect miscompilations in the earliest front-end passes.

Converting P4 programs into Z3 formulas. We now describe our execution model to convert a P4 program into a Z3 logic formula. Figure 4.2 shows an example. Conceptually, our goal is to represent P4 programs in a functional form so that the input-output behavior of

the functional form is identical to the input-output behavior of the P4 program. To determine function inputs and outputs, we use the parameter directions of each P4 architecture package. Parameters with the direction `inout` and `out` make up the output Z3 data type of the function whereas parameters with the `in` and `inout` are free Z3 variables that represent the input of the function.

To determine the functional form, the symbolic interpreter traverses each path through the P4 program, maintaining expressions representing path conditions for branching. Once it reaches a portion of the program where execution ends, it stores an if-then-else Z3 expression with the condition set to the path condition and the return value set to a tuple consisting of the `inout` and `out` parameters at that point. Ultimately, the interpreter will return a single nested if-then-else Z3 expression, with each branch corresponding to a unique output from the program under a set of conditions. Using this expression we can perform operations such as equivalence checking between two Z3 formulas for translation validation or querying Z3 to provide an output for particular input for test case generation.

Handling tables. The contents of a table are unknown at compile-time. Since we want to make sure we cover any possible table content, we interpret match-action pairs in tables symbolically. Figure 4.2 describes a simplified example of how Gauntlet interprets tables within a control block. Per match-action table call, we generate one symbolic match (`t_table_key`) and one symbolic action variable (`t_action`), which represent a single match key and its choice of action, respectively. We compare the symbolic packet header with the symbolic match key (`hdr.a == t_table_key`). If the expression evaluates to true, it implies the execution of a specific action, which is chosen based on the value of the symbolic action index (`t_action`). We express this as a series of nested if-then-else statements per action available to the table. Finally, if the key does not match, the default action is selected. For instance, in Figure 4.2, we execute action `assign` (action id 1) iff the symbolic match vari-

able (`t_table_key`) equals the symbolic header (`hdr.a`) and the symbolic action variable (`t_action`) equals 1. With this encoding, we can avoid having to use a separate symbolic match-action pair for every entry in the match-action table, which is a prohibitively large number of symbolic variables.

Header validity. The $P4_{16}$ specification does not explicitly restrict the behavior of header validity. We model our semantics to align with the implementation in P4C. We clarified these assumptions with the compiler and specification maintainers [190]. If a previously invalid header is marked valid, all fields in that header are initially undefined. If an invalid header is returned in the final output, all fields in the header are set to invalid as well.

Interpreting function calls. Any `out` parameter in a function call is initially set to undefined. If the function returns, we also generate a new free Z3 variable. In our interpreter, externs are treated as a function call that returns an arbitrary value. In addition, each argument for a parameter that has the label `inout` and `out` is set to a new free Z3 variable because the behavior of externs is unknown. Copy-in/copy-out semantics, albeit necessary to control side effects in extern objects, have been a persistent source of bugs in the compiler. A significant portion of the miscompilations we identified were caused by erroneous passes that perform incorrect argument evaluation and side effect ordering in relation to copy-in/copy-out.

Checking equivalence between P4 programs. We use `p4test` to emit a P4 program after each compiler pass. `p4test` is a P4C back-end used to test P4C. It does not produce any executable output but exercises all the default front- and mid-end passes. We only examine passes that actually modify the input program and ignore any emitted intermediate program that has a hash identical to its predecessor. We explicitly reparse each emitted P4

file to also catch bugs in the parser and the `ToP4` module.

For an input program A and the transformed output program B after a compiler pass, we perform a pairwise equivalence check for each programmable block. We use our interpreter to retrieve the Z3 formulas for all programmable blocks of the program architecture package and compare each individual block of A to the corresponding block in B. The query for the Z3 solver is a simple inequality. It is satisfiable only if there is a Z3 assignment (e.g., a packet header input or table match-action entry) in which the Z3 formula of A produces a different output from B.

If the inequality query is satisfiable, it produces the assignment that would lead to different results and saves the failed passes for later analysis. With this technique, we can precisely pinpoint in which pass a miscompilation may have happened and we can also infer the packet values we need to trigger the bug. If the report turns out to be a false alarm and is not confirmed by compiler developers, this is a bug in our symbolic interpreter, which we fix. The generated Z3 formulas could in principle be very large and checking could take a long time. However, we use the quantifier-free theory of bit vectors (QF_BV) [159] for the equality check, which can be solved efficiently in Z3 [53]. Even very large expression trees can be compared in under a second.

Handling undefined behavior. We track changes in undefined behavior in which the undefined portion of a P4 program has more restricted (less undefined) behavior after a compiler pass. This means we can identify scenarios where the compiler transforms a program fragment based on undefined behavior. While not immediately harmful, such changes might still indicate problematic behavior in the compiler that may be surprising to a programmer [246].

To track undefined behavior, any time a variable is affected by undefined behavior (e.g., a header is set to invalid and then valid) we label that variable "undefined." This undefined

variable effectively acts as taint. Every read or write to this undefined variable is tainted. When comparing Z3 formulas before and after a pass, we can choose to replace tainted expressions with concrete values in the formula before a pass.[3] With this, we can determine if a translation validation failure was caused by undefined behavior. If we find a failure based on undefined behavior, we classify it as unstable code [246] to avoid confusion with real bugs.

Addressing parser loops. Parser loops can recurse infinitely many times in the core P4 language. To still terminate with our parser analysis, we assume that any well-formed P4 program must advance the parser counter with each loop (usually this is done by parsing the next header in a header stack). Hence, we compute our maximum bound for parser analysis by summing up the size of all header stacks. Each time we encounter a parser state we have already visited we increment a counter. The parser will have to advance the header stack index at least once per loop iteration. If not, we have encountered an infinite parser loop. If this count exceeds the maximum bound, we terminate the parser analysis and use the generated expression for equality comparison. With this approach, we generate a very large expression, but handle the majority, if not all, possible transformations of the parser graph. Nonetheless, our approach is a heuristic. We have not proven that it can capture all possible program transformation scenarios. We leave optimizations on this naive approach as future work.

4.6 Results

We now analyze the P4 compiler bugs found by Gauntlet. A detailed breakdown can be found in Section 4.8. The cut-off date for these results was June 14th, 2022. Our main findings are summarized below.

[3] We only replace tainted expressions in the "before" formula so that we can detect compiler bugs where a previously well-defined expression turns undefined, which is an actual compiler bug, not just an unsafe optimization.

Bug Type	Status	P4C	BMv2	Tofino
Crash	Filed	39	4	35
	Confirmed	36	4	25
	Fixed	36	4	20
Semantic	Filed	35	1	10
	Confirmed	30	1	7
	Fixed	30	1	7
Total	**103**	**66**	**5**	**32**

Table 4.2: Bug summary. Bugs that have not been fixed have been assigned.

Location	P4C	BMv2	Tofino	**Total**
Front End	42	-	-	42
Mid End	24	-	-	24
Back End	-	5	32	37
Total	66	5	32	**103**

Table 4.3: Distribution of bugs in the P4 compiler front end, mid end, and the BMv2 and Tofino back ends.

1. We confirmed a total of 103 new, distinct bugs across the P4C framework and the BMv2 and Tofino P4 compilers. Of these bugs, 65 are crash bugs and 38 are miscompilations.
2. Our efforts led to 6 P4 specification changes [236, §A.1].
3. We achieved this in the span of only 8 months of testing with Gauntlet, and despite only generating random programs from a subset of the $P4_{16}$ language.

4.6.1 Sources of Bugs

We distinguish the bugs we found into three primary sources: bugs we found in the common P4C framework and bugs we found in the compiler back ends for BMv2 and Tofino. Both the BMv2 and Tofino back ends use the P4C front- and mid-end passes. Hence, most bugs detected in P4C also likely apply to these back ends. Note that since the Tofino back end is closed-source, we don't know which P4C passes it uses.

All miscompilations in P4C were found by translation validation because we had full access to the compiler IR. Where applicable, we reproduced the miscompilations using model-based testing and attached the failing input-output packet pair with our bug report. All the miscompilations in the Tofino compiler were found with model-based testing. We do not describe model-based testing in this dissertation, but include the bug counts for completeness and to demonstrate that even a limited technique, such as model-based testing on the P4 DSL

only, is able to find bugs in device compilers. For a detailed description of our model-based testing approach, refer to the original Gauntlet paper [196].

Distribution of Bugs. Table 4.3 lists where we identified bugs. The overall majority of bugs were found in the P4C front- and mid-end framework, mainly because we concentrated on these areas. The majority of the back end bugs were found in the Tofino compiler. There are two reasons for this. First, the Tofino back end is more complex than BMv2 as it compiles for a high-speed hardware target. Second, we did not test the BMv2 back end as extensively as other parts of the compiler.

Bugs in the P4C infrastructure. As Table 4.2 shows, we were able to confirm 103 distinct bugs. 66 were uncovered in P4C, with a comparable distribution of crash bugs (36) and miscompilations (30). Initially, the majority of bugs that we found were crash bugs. However, after these crash bugs were fixed, and as our symbolic interpreter became reliable, the miscompilations began to exceed the crash bugs.

In addition, 6 of the bugs we found led to corresponding changes in the specification as they uncovered missing cases or ambiguous behavior because our interpretation of a specific language construct clashed with the interpretation of the compiler developers and language designers. We also continuously checked out the master branch to test the latest compiler improvements for bugs. Many bugs (16 out of 66) were introduced by recent merges of pull requests during the months in which we used Gauntlet for testing. Gauntlet was able to quickly detect these bugs. To catch such bugs as soon they are introduced, the P4C developers have now integrated Gauntlet into P4C's continuous integration (CI) pipeline.

Derivative bugs. 6 of the 103 bugs we found were crash bugs that were not directly caused by random programs generated by Gauntlet. Instead, they were caused by hand-crafting specific $P4_{16}$ programs containing specific language constructs. These handcrafted

Program	Arch	LoC	Time (seconds)
tna_simple_switch.p4	TNA	1555	~**0.38**
switch_tofino_x2.p4	TNA	6752	~**7.54**
switch_tofino2_y2.p4	TNA2	7039	~**9.06**
fabric.p4	V1Model	958	~**0.45**
switch.p4 (from $P4_{14}$)	V1Model	5885	~**12.13**

Table 4.4: Time needed to get semantics from a P4 program.

$P4_{16}$ programs were inspired by discussions related to either specification changes or compiler bugs originally found by Gauntlet. We included these bugs in our count because even though they were handcrafted, they were seeded by bug reports originating from Gauntlet. We also encountered 3 new crash bugs when manually reducing our randomly generated programs for miscompilations. For instance, we uncovered a crash bug caused by a P4 parser loop because we removed transition expressions [236, §12.5] as part of reducing one of our randomly generated programs. However, all our miscompilations were found directly by Gauntlet.

Fixing the bugs. Out of the 103 new bugs we filed, 98 have been fixed. The remaining bugs have been assigned to a developer, but are still open because we filed them very recently, they required a specification change to be resolved first, or they have been de-prioritized in favor of more pressing bug reports. We have received confirmation from the Tofino compiler developers that 8 bugs have already been resolved; the remainder are targeted to be resolved by the next release.

4.6.2 Performance on Large P4 Programs

We also measured the time Gauntlet currently requires to generate semantics for several large P4 programs (Table 4.4). Generating semantics is the slowest part of our validation check; comparing the equality of the generated formulas in Z3 is typically fast. We have observed that retrieving semantics for a single pass takes on the order of seconds for a large program.

Parser branching, in particular dense branching caused by parser loops, is a scaling challenge for our tool.

4.6.3 Deep Dive into Bugs

Ripple effects. A common crash we observed occurs because a compiler pass incorrectly transforms an expression or does not process it at all. Back-end compiler developers rely on the front end to correctly transform the IR of the P4 program. But, if a pass misses a language construct it is responsible for, the back end often cannot handle the resulting expression and generates an assertion failure. For example, in program 4.3b, the front end `SideEffectOrdering` [32] pass should have converted the conditional operator in line 3 into normal if-then-else control flow. However, because of the addition expression, the pass failed to transform the conditional operator, which ultimately caused an assertion to fire in the Tofino back end [189]. In another case, the `InlineFunctions` [32] pass did not fully inline all function calls, causing a crash in back ends that were not able to understand function calls and expected them to have been inlined by then [186].

Crashes in the type checker. Many of the crashes (21 out of 36) were in the type checker infrastructure. The code in 4.3a shows an expression that crashed type checking [188]. It is not possible to shift this value since its width is unknown at compile-time. This program was deemed illegal, but the specification did not explicitly forbid it. The type checker tried to infer a type regardless and crashed. This bug also triggered an update to the $P4_{16}$ specification [75]. In other cases, the type checker was incorrectly forbidding a valid expression. In example 4.3c, the program was legal, but because a safety check in the `StrengthReduction` [32] pass was incorrectly implemented, the resulting slice index was overflowing and turned negative, which prompted the type checker to terminate with an error message [188].

```
control ig(inout Hdr h, ...) {
  apply {
    h.mac_src = (1 << h.modifier)
    + 8w1;
  }
}
```

(a) A crash in the type checker.

```
control ig(inout Hdr h, ...) {
  apply {
    h.mac_src =
    (h.mac_src > 2 ? 48w1 : 48w2) + h.
    mac_src;
  }
}
```

(b) A bug caused by a defective pass.

```
control ig(inout Hdr h, ...) {
  apply {
    bool tmp = 1 != 8w2[7:0];
  }
}
```

(c) An incorrect type checking error.

```
control ig(inout Hdr h, ...) {
  action assign_eth_type(inout bit<8> val
    ) {
    h.eth_type[15:8] = 0xFF;
  }
  apply {
    assign_eth_type(h.eth_type[7:0]);
  }
}
```

(d) Incorrect deletion of an assignment.

```
control ig(inout Hdr h, ...) {
  apply {
    h.ipv4.setInvalid();
    h.ipv4.src_addr = 1;
    h.eth.src_addr = h.ipv4.
    src_addr;
    if (h.eth.src_addr != 1) {
      h.ipv4.setValid();
      h.ipv4.src_addr = 1;
    }
  }
}
```

(e) An unsafe compiler optimization.

```
control ig(inout Hdr h, ...) {
  action assign_and_exit(inout bit<16>
    val) {
      val = 0xFFFF;
      exit;
  }
  apply {
    assign_and_exit(h.eth_type);
  }
}
```

(f) Incorrect interpretation of exit statements.

Figure 4.3: Examples of bugs that were caught by Gauntlet.

Handling side effects. Side effects from a function operate on the concept of copy-in/copy-out semantics, described earlier. However, these semantics, while seemingly simple, turn out to be hard to implement correctly in the compiler. A particularly tricky case can be seen in 4.3d [192].

In the program, a slice of a variable is passed as an `inout` parameter. At the same time, a disjoint subset of the variable is assigned within the function. The correct behavior here is to leave the assignment unchanged, and copy back the sliced portion of the variable alone. However, the compiler assumed that the entire variable would be copied back and removed the assignment in line 3, an incorrect optimization.

A large subset of the miscompilations we found in P4C (at least 12 out of 30) can be traced to incorrect handling of side effects and copy-in/copy-out. Copy-in/copy-out is difficult to handle because for a compiler pass that reorders expressions or statements, side effects can be translated incorrectly.

Unstable code. Even though the $P4_{16}$ language has limited undefined behavior, we also found incidents of unstable code [246]. This unstable code conforms with the specification but may lead to instability in specific back end targets. Dumitru et al., 2020 also discuss the potential safety consequences of undefined variable access [61]. Program 4.3e is a concrete example. The compiler collapses the assignment of line 4 into line 5, setting `h.eth.src_addr`, which is still part of a valid header, to 1. All of this is legal behavior, since read and write operations on invalid header values are undefined as part of the P4 specification. The compiler is free to perform these optimizations. However, these changes may cause issues in specific back ends, e.g., back ends in which assignments to invalid headers are no-ops. In this case, the compiler has chosen a particular subset interpretation of undefined behavior, which may clash with the expectations of programmers for that back end. We raised this with the compiler developers, who agreed to print a warning [190].

Consequences of compiler changes. Once we started actively monitoring the master branch of P4C we observed that many (19 out 66) of the bugs we filed in P4C were introduced by recent merges into master. A notable example is a recent change to the `Predication` [32] pass, which caused at least 6 (1 crash and 5 semantic) new bugs. We caught and filed these bugs quickly during our weekly routine random code generation. The compiler pass has become so complicated that the compiler maintainers are now relying on Gauntlet to ensure correctness [14]. A P4 programmer also filed a bug on this issue [76]. The report was considered a duplicate because of our earlier reports, highlighting that the bugs we find do affect actual P4 programmers.

Specification changes. Some of our bug reports kicked off larger discussions and changes around the P4 language specification. Our bug reports and questions have led to at least 6 distinct specification changes. For example, a concern we had about the validity of uninitialized headers (at what point does a header variable become valid?) led to three clarification pull requests on the specification and a suggestion to propose more fundamental changes for the next language version [78].

Another prominent example was caused by ambiguity in the specification. In example 4.3f, the `RemoveActionParameters` [32] compiler pass moved the statement in line 3 after the exit statement, because the assumption was that exits called within functions ignore the copy-in/copy-out semantics. We instead interpreted exit statements to still respect copy-in/copy-out semantics and caught the discrepancy. This is a significant difference. A packet that traverses the control program could lose all the modifications that have been written to its header, a potential security risk. We filed this as a concern with the open-source community [187] and our interpretation was deemed reasonable, which required a specification update [79]. The corresponding compiler changes resulted in at least 3 new bugs, which we detected and filed.

Invalid transformations. Because P4C provides the option to emit transformed programs after each pass as a valid P4 program, the compiler developers maintain an invariant that each compiler pass in the front end and mid end needs to emit syntactically correct P4. We uncovered several bugs with how P4 code is emitted and transformed across compiler passes. We detected these bugs by reparsing each P4 program after it had been emitted by the `ToP4` compiler module. If the emitted program cannot be reparsed, it indicates a bug in one of three compiler components: the `ToP4` module, the P4C parser, or the compiler pass. While these bugs typically do not harm correctness, they affect compiler debugging. Overall, we identified 4 bugs of invalid intermediate P4, all of which were fixed; these 4 are not included in our count of 103. Additionally, because we reparse P4 after each compiler pass, we found a case where the emitted program being parsed incorrectly was a symptom of a larger bug in the P4C parser [191].

4.6.4 Lessons Learned

P4C debugging support. P4C has several facilities that were useful for bug finding. The ability to dump the intermediate representation, specify which passes to dump, and the `ToP4` tool, which converts P4C-IR to P4 programs accelerated our development process. In addition, the compiler has comprehensive assert instrumentation with distinct messages, which we used to identify unique crash bugs and to distinguish them from valid error messages. The AST visitor library in P4C allowed us to develop extensions like our random program generator and interpreter.

P4C's nanopass architecture, which factors the compiler into a large number of "thin" passes, helps with bug fixing, especially for miscompilations that were narrowed down to one pass by translation validation. A different architecture that has fewer "thick" passes would need more developer effort to fix miscompilations. We also observed that almost all crash bugs were assertion violations where an invariant was violated in a particular compiler pass

due to an incorrect or absent compiler transformation from a previous pass. In the absence of such assertions, these crash bugs could have easily manifested as miscompilations that are harder to detect.

Reporting bugs. This project would not have been possible without the responsiveness and receptiveness of the P4 community. Our questions, concerns, and bug reports were answered within a day and in great detail. The developers were able to even dissect our initial questions and confusions into bug reports, guiding us further in our development effort. We were encouraged to participate in the language design working group that discusses changes to the P4 specification.

Likewise, when we filed bugs for the closed-source and proprietary Tofino compiler, we found the developers to be receptive and responsive. Still, the pace of bug finding and fixing with the Tofino compiler was slower than the open-source compiler because of two unavoidable reasons. First, we naturally did not have access to the company bug tracker to assess the life cycle of our bug once it had been filed. Second, the official binary of the Tofino compiler updates less frequently than P4C, which can be rebuilt from source after every commit. Hence, we would trigger the same bugs repeatedly in our testing until a new Tofino compiler version with a bug fix was released. Neither of these two problems would manifest, if our tool was to be used internally as part of the compiler development process for Tofino.

4.7 Discussion

With Gauntlet we have built an execution model for the P4 language that is target-independent and precise enough to find bugs in the front end of a P4 compiler. To our knowledge, Gauntlet is the first instance of translation validation running as part of a compiler's CI infrastructure.

We believe this ability to exploit domain specificity for more effective compiler bug finding will increasingly be applicable to other DSLs beyond P4. That being said, there are several follow-ups to Gauntlet we would like to explore.

Extending translation validation to the compiler back end. So far we have applied translation validation only to compiler front and mid ends. This is because these passes allow us to dump the P4 program before and after the pass has run, allowing us to compare the before and after programs for equality. The back end is typically proprietary, inaccessible, and uses an opaque intermediate representation. To understand the constraints of these back ends we would like to work with industry compiler developers to integrate translation validation into their compilers. We will develop translation validation techniques that allow us to compare a P4 program's semantics with the semantics of a back end language that is not P4.

Automatic test case reduction. We have not developed an automatic test case reduction suite (e.g., C-Reduce [179]) and reduce buggy programs manually. After our testing pipeline has identified problematic programs in a randomly generated batch, we inspect each P4 program individually. We prune the random P4 program that caused the bug until we get a sufficiently small program that can be attached to a bug report. We are currently automating this process.

Long-term study on translation validation in CI. We would like to perform empirical, long-term studies on the utility of translation validation as a compiler testing technique. We want to identify which passes frequently cause semantic issues and understand why they do. We would also like to observe how developer-friendly our tool is. For example, to avoid confusing compiler developers, we already had to make sure that Gauntlet does not report changes in undefined behavior [77] or fails gracefully when Gauntlet does not support

a particular language construct [33].

Coverage metrics for random program generation. We also do not track how much of the compiler source code we actually cover with our program generator. For future work, we would like to measure the compiler code coverage of a generated P4 program with `gcov` to understand avenues for improvement. In concurrent work, Kodeswaran et al. [127] use the Ball-Larus encoding [17] to track the execution path of packets traversing the switch. By inspecting this path, a developer can verify that packets have actually taken the expected path through the P4 program. This technique is complementary to our symbolic execution approach. We are considering using it as a path coverage tool for metamorphic testing such as EMI [134].

4.7.1 Limitations of Gauntlet's Model-Based Testing

Gauntlet also used model-based testing to generate input-output packet tests to find bugs in P4C and the Tofino compiler. We list the bugs we found in Table 4.2 and Table 4.3. The reason we are not including a description of model-based testing is that the technique, in contrast to translation validation that runs entirely on a formal logic-based representation of the P4 program, model-based testing had several limitations caused by needing to run actual end-to-end tests on real targets. P4Testgen and Flay, which we describe in Chapter 5 and Chapter 6, address these limitations.

Dropped packets in the testing framework. A key assumption in the model-based testing approach is that the generated test cases can actually be fed to the testing framework of the back end. However, the semantics of the generated P4 program do not describe hardware-specific restrictions. For example, some devices impose minimum packet size requirements or drop packets with invalid MAC addresses. More generally, we have found

that test cases where the input packets have certain values in their headers can be dropped silently by the back end without generating an output packet. Effectively, there is a mismatch between the Z3 semantics, which says that a certain output packet must be produced, and the back end's semantics, which produces no output packet. In such cases, we have had to discard these test cases, reducing the breadth of coverage for testing the compiler.

Unknown interfaces between programmable blocks. P4 also does not provide semantics on the treatment of packets between the individual control or parser blocks. This is not an issue for translation validation since we compare each programmable block individually. For an end-to-end test, however, we need to know how data is modified between these blocks so that we know what output packet to expect.

Test case complexity. Paths with many branches can generate a large number of distinct path conditions. Thus, millions of input-output packet pairs might be generated. Since small programs have sufficed so far for bug finding, we have not run into these issues. In the future, we may need an efficient path selection technique to tease out more complex bugs on closed-source compilers.

4.8 Details on Bug Results

Table 4.5: Crashes found in open-source P4C.

Issue	Location	Compiler Stage	Status
Compiler Bug: visitor returned non-Statement type (2102)	2102.p4	Mid end	Fixed
Compiler Bug: no definitions (2104)	2104a.p4	Front end	Fixed
Compiler Bug: no definitions (2104)	2104b.p4	Front end	Fixed
Compiler Bug: no locations known for (2105)	2105.p4	Front end	Fixed
InlineFunctions pass sometime seems to generate invalid code. (2126)	2126.p4	Front end	Fixed
p4c-bm2-ss crashes on undefined header conditional in method (2148)	2148.p4	Mid end	Fixed
Follow-up to issue 2104, exit also kills all variables (2151)	2151.p4	Front end	Fixed
Compiler Bug: At this point in the compilation typechecking should not infer new types anymore, but it did. (2190)	2190.p4	Front end	Fixed
Compiler Bug: Null cst (2206)	2206.p4	Front end	Fixed
Compiler Bug: boost::too_few_args (2207)	2207.p4	Front end	Rejected
Predication pass leads to unsafe variable assignment (2248c)	2248c.p4	Mid end	Fixed
Compiler Bug: Null stat (2258a)	2258a.p4	Front end	Fixed
"error: Duplicates declaration" when initializing struct in function with integer values (2261)	2261.p4	Front end	Fixed
"Compiler Bug Null stat" also triggered in action properties (2266)	2266.p4	Front end	Fixed
Compiler Bug: visitor returned invalid type Vector for IndexedVector (2289)	2289.p4	Mid end	Fixed
Compiler Bug: Unexpected type for nested headers (2290)	2290.p4	Front end	Fixed
BMV2 Backend Compiler Bug unhandled case (2291)	2291.p4	Back end	Fixed
Compiler Bug: Could not find type of @name (2336)	2336.p4	Front end	Fixed

Continued on next page

Issue	Location	Compiler Stage	Status
Compile-time-known and slices (2342)	2342.p4	Front end	Fixed
Compiler Bug: Null cst for InfInt Parameters (2354)	2354.p4	Front end	Fixed
Silent crash: Struct Parameters (2355)	2355.p4	Front end	Fixed
Negative bit index in Slice after shifting function output (2356)	2356.p4	Front end	Fixed
Int as parameter and assignments (2357)	2357.p4	Front end	Fixed
Mixing exits and returns in actions (2359)	2359.p4	Mid end	Fixed
Another expression with side-effects in table keys and actions (2362)	2362.p4	Front end	Fixed
No Definitions in Parser Loop (2373)	2373.p4	Front end	Fixed
Crash when running end-to-end tests with simple switch (2375)	2375.p4	Back end	Fixed
Compiler Bug: no definitions (2393)	2393.p4	Front end	Fixed
p4test: x: declaration not found (2435a)	2435a.p4	Mid end	Fixed
Range starting from zero: Can not shift by a negative value (2485)	2485.p4	Mid end	Fixed
Conditional execution in actions with struct initializers (2486)	2486.p4	Mid end	Rejected
StructInitializer in Mux expressions (2487)	2487.p4	Mid end	Fixed
p4c-bm2-ss: Compiler Bug: Could not convert to Json (2495)	2495.p4	Back end	Fixed
Control Inlining: Key declaration not found (2542)	2542.p4	Front end	Rejected
Some problems with function calls in struct initialization (2543a)	2543a.p4	Mid end	Fixed
Some problems with function calls in struct initialization (2543b)	2543b.p4	Front end	Fixed
Compiler Bug: Could not find type of for declaration of same name (2544)	2544.p4	Front end	Fixed

Continued on next page

Issue	Location	Compiler Stage	Status
Calling an extern with a local variable: read-only error (2545)	2545.p4	Mid end	Fixed
Side-effect function call in table key (2546b)	2546b.p4	Front end	Fixed
Compiler Bug : Null firstCall (2597)	2597.p4	Front end	Fixed
Compiler Bug: Exiting with SIGSEGV (2648)	2648.p4	Mid end	Fixed
simple_switch died with return code -6 (887)	887.p4	Back end	Fixed

Table 4.6: Semantic Bugs found in open-source P4C.

Issue	Location	Compiler Stage	Status
SimplifyDefUse incorrectly removes assignment in actions with slices as arguments (2147)	2147.p4	Front end	Fixed
Switchstatement: Assignment in switch-case removed (2153)	2153.p4	Mid end	Fixed
Question about parser behavior with right shifts (2156)	2156.p4	Front end	Fixed
Associative order of shift operators (2161)	2161.p4	Back end	Fixed
RemoveReturns deletes return in SwitchStatements (2170)	2170.p4	Mid end	Fixed
Assignment removed despite conditional return (2175)	2175.p4	Front end	Fixed
Question on precise behavior of copy-out (2176)	2176.p4	Front end	Fixed
Compiler Bug: At this point in the compilation typechecking should not infer new types anymore, but it did. (2190)	2190a.p4	Front end	Fixed
Question on side-effect ordering (2205)	2205.p4	Front end	Fixed
Clarification question on setInvalid (2212)	2212.p4	Front end	Rejected
StrengthReduction ignores side-effects in function calls (2221)	2221.p4	Front end	Fixed
Calling exit in actions after an assignment (2225)	2225.p4	Mid end	Fixed
Predication pass leads to unsafe variable assignment (2248a)	2248a.p4	Mid end	Fixed
Predication pass leads to unsafe variable assignment (2248b)	2248b.p4	Mid end	Fixed
Missed side-effect case StrengthReduction (2287)	2287.p4	Front end	Fixed
SideEffectOrdering: Regression? (2288a)	2288a.p4	Front end	Fixed

Continued on next page

Issue	Location	Compiler Stage	Status
SideEffectOrdering: Regression? (2288b)	2288b.p4	Front end	Fixed
MoveInitializers and parser loops (2314)	2314.p4	Front end	Fixed
More questions on setInvalid (2323)	2323.p4	Front end	Rejected
Another issue with Predication (2330)	2330.p4	Mid end	Fixed
Another missed case of StrengthReduction (2343)	2343.p4	Front end	Fixed
Inlining functions and duplicate table calls (2344)	2344.p4	Mid end	Fixed
Incorrect transformation in Predication pass (2345b)	2345b.p4	Mid end	Fixed
Follow-up issue on exit statements (2358a)	2358a.p4	Front end	Fixed
Follow-up issue on exit statements (2358b)	2358b.p4	Front end	Fixed
Def-Use and exit statements (2374)	2374.p4	Front end	Rejected
InlineActions also seems to handle exit statements incorrectly (2382)	2382.p4	Front end	Rejected
Clarification question on uninitialized local headers (2383)	2383.p4	Mid end	Fixed
Question on comparison to negative constants (2392)	2392.p4	Front end	Fixed
Inlining controls with out parameters (2470)	2470.p4	Front end	Rejected
Side effects in StructInitializers (2488)	2488.p4	Front end	Fixed
Follow-up on slice arguments (2498)	2498.p4	Front end	Fixed
Side-effect function call in table key (2546b)	2546b.p4	Mid end	Fixed
Fix: Predication issue (2564)	2564.p4	Mid end	Fixed
Fix: Issue #2004 parser duplicated matches not optimized out (2591)	2591.p4	Mid end	Fixed
Predication: Another problem (2613)	2564.p4	Mid end	Fixed
StrengthReduction: Incorrect slice optimization (2614)	2614.p4	Front end	Fixed
Some more predication issues (2647)	2647a.p4	Mid end	Fixed

Table 4.7: Crash Bugs found in BF-P4C (P4Studio 9.9.0).

Issue	Name	Compiler Stage	Status
1	bug1.p4	Front end	Rejected
2	bug2.p4	Front end	Rejected
3	bug3.p4	Back end	Rejected (Front-end issue)
4	bug4.p4	Front end	Rejected
5	bug5.p4	Back end	Fixed
6	bug6.p4	Back end	Fixed
7	bug7.p4	Back end	Fixed
8	bug8.p4	Back end	Fixed
9	bug9.p4	Back end	Fixed
10	bug10.p4	Back end	Fixed
11	bug11.p4	Back end	Fixed
12	bug12.p4	Back end	Confirmed
13	bug13.p4	Back end	Rejected (Front-end issue)
14	bug14.p4	Back end	Fixed
15	bug15.p4	Back end	Fixed
16	bug16.p4	Back end	Fixed
17	bug17.p4	Back end	Confirmed
18	bug18.p4	Back end	Fixed
19	bug19.p4	Back end	Fixed
20	bug20.p4	Back end	Fixed
21	bug21.p4	Back end	Fixed
22	bug22.p4	Back end	Confirmed
23	bug23.p4	Back end	Confirmed
24	bug24.p4	Back end	Fixed
25	bug25.p4	Back end	Fixed
26	bug26.p4	Back end	Confirmed
27	bug27.p4	Back end	Rejected (Front-end issue)

Continued on next page

Issue	Name	Compiler Stage	Status
28	bug28.p4	Back end	Rejected
29	bug29.p4	Back end	Rejected (Front-end issue)
30	bug30.p4	Back end	Fixed
31	bug31.p4	Back end	Fixed
32	bug32.p4	Back end	Rejected
33	bug33.p4	Back end	Fixed
34	bug34.p4	Back end	Rejected (Front-end issue)
35	bug35.p4	Back end	Fixed

Table 4.8: Semantic bugs found in BF-P4C (P4Studio 9.9.0).

Issue	Name	Compiler Stage	Status
1	semantic_bug1.p4	Back end	Fixed
2	semantic_bug2.p4	Back end	Fixed
3	semantic_bug3.p4	Back end	Fixed
4	semantic_bug4.p4	Back end	Fixed
5	semantic_bug5.p4	Back end	Fixed
6	semantic_bug6.p4	Back end	Rejected
7	semantic_bug7.p4	Back end	Fixed
8	semantic_bug8.p4	Back end	Fixed
9	semantic_bug9.p4	Back end	Rejected
10	semantic_bug10.p4	Back end	Rejected

5 | P4Testgen: Generating Test Packets For Network-Device Stacks

The model we built for Gauntlet's translation validation is effective at validating the correctness of a P4 compiler. However, it has three major limitations: 1) It only describes the core of the P4 language, i.e., it does not cover back-end-specific behaviors and program constructs (most importantly externs). 2) It models the programmable blocks of a device in isolation. There are no semantics describing how these blocks are linked together, what happens between them, and how a device may execute them. 3) The components involved in forwarding a packet in a network device extend beyond the compiler. The control plane, the runtime API, and the device hardware all play a role.

One pragmatic approach to ensure these components are working as intended is to apply end-to-end tests which exercise all necessary software layers and APIs. Typically, these tests are input-output tests which describe the expected behavior for a particular set of inputs and configuration. We can use the P4 program as a model to generate these tests and test our networking device. At first glance, the task of generating tests for a given P4 program seems straightforward. Prior work on Vera [221], `p4pktgen` [160], P4wn [119], Meissa [264], SwitchV [9] and others has shown that it is possible to automatically generate tests using

techniques from the programming languages literature [123, 206, 89]. The precise details vary from tool to tool, but the basic idea is to first use symbolic execution [13] to traverse a path in the program, collecting a symbolic environment and a path constraint, and then use an SMT solver[1] to compute an executable test. The SMT solver fills in the input and output packets from the symbolic environment and path constraint, and also computes the control-plane configuration that is needed to execute the selected path—e.g., forwarding entries for match-action tables.

This prior work, however, does not address the first two limitations mentioned above. Existing tools have focused on specific targets (e.g., Tofino) and abstracted away important details (e.g., non-standard packets and other "corner cases" in the language), which limits their applicability in practice. In contrast, our goal is to develop a general and extensible test oracle for P4 that can be readily applied to real-world P4 programs on arbitrary targets. This means we must extend our model to cover these cases.

5.1 Introduction

P4Testgen is an extensible *test oracle* for the $P4_{16}$ [236] language. Given a P4 program and sufficient time, it generates an exhaustive set of tests that cover every reachable statement in the program. Each test consists of an input packet, control-plane configuration, and the expected output packet.

P4Testgen generates tests to validate the *implementation* of a P4 program. Such tests ensure that the device executing the P4 code (which we refer to as the "target" in this chapter) and its toolchain (i.e., the compiler [34], control plane [231, 8], and various API layers [162, 229, 228]) implement the behaviors specified by the P4 program.

Tests generated by P4Testgen can be used by manufacturers of P4-programmable equip-

[1] Just as we did with Gauntlet.

ment to validate the toolchains associated with their equipment [25, 107, 124, 87, 47, 147], by P4 compiler writers for debugging optimizations and code transformations [34, 90], and by network owners to check that both fixed-function and programmable targets implement behaviors as specified in P4, including standard and custom protocols [264, 9].

The idea of generating an exhaustive set of tests for a given P4 program is not new. However, prior work has largely focused on a specific P4 architecture [236, §4]. For example, `p4pktgen` [160] targets BMv2 [232], Meissa [264] and p4v [140] target Tofino [25], and SwitchV [9] targets fixed-function switches. The primary reason why these tools are so specialized is development effort. Building P4 validation tools requires simultaneously understanding (i) the P4 language, (ii) formal methods, and (iii) target-specific behaviors and quirks. Finding developers that satisfy this trifecta even for a single target is already challenging. Finding developers that can design a general tool for all targets is even harder. The unfortunate result is that developer effort has been fragmented across the P4 ecosystem. Most P4 targets today lack adequate test tooling, and advances made with one tool are difficult to port over to other tools.

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

P4Testgen combines several techniques in an open-source tool suitable for production use. First, P4Testgen provides an extensible execution model for the whole program ("whole-

program semantics"), combining the semantics of the P4 code along with the semantics of the target on which it is executed. A P4 program generally consists of several P4 blocks (with semantics provided by the language specification) that are separated by interstitial architecture-specific elements (with semantics provided by the target). P4Testgen is the first tool that provides an execution model for such whole-program semantics, using a carefully designed interpreter based on the open-source P4 compiler (P4C) [34]. Second, while P4Testgen ultimately uses an SMT solver to generate tests, it also handles the "awkward squad" of complex functions that are difficult to model using SMT—e.g., checksums, undefined values, randomness, and so on. To achieve this, P4Testgen uses taint tracking, concolic execution, and a precise model of packet sizing to model the semantics of the program accurately and at bit-level granularity. Third, P4Testgen offers advanced path selection strategies that can efficiently generate tests that achieve full statement coverage, even for large P4 programs that suffer from path explosion. In contrast to prior work, these strategies are fully automated and do not require annotations to use effectively.

We implement P4Testgen's execution model using the following key technical innovations:

1. **Whole-program semantics**: Most P4 targets perform processing that is not defined by the P4 program itself and is target-specific. P4Testgen uses *pipeline templates* to succinctly describe the behavior of an entire pipeline as a composition of P4-programmable blocks and interstitial target-specific elements.
2. **Target-specific extensions**: Many real-world P4 targets deviate from the $P4_{16}$ specification in ways small and large. To accommodate these deviations, P4Testgen's extensible interpreter supports target-specific *extensions* to override default P4 behavior, including initialization semantics and an intricate model of *packet-sizing*, which accommodates targets that modify packet sizes during processing.
3. **Taint analysis**: Targets can exhibit non-deterministic behavior, making it impossible to predict test outcomes. To ensure that generated tests are reliable, P4Testgen uses

taint analysis to track non-deterministic portions of test outputs.

4. **Concolic execution**: Some targets have features that cannot easily be modeled using an SMT solver. P4Testgen uses *concolic execution* [89, 206] to model features such as hash functions and checksums.
5. **Path selection strategies:** Real-world P4 programs often have a huge number of paths, making full path coverage infeasible. P4Testgen provides heuristic *path selection strategies* that can achieve full statement coverage, usually with orders of magnitude fewer tests than other approaches.

To validate our design for P4Testgen, we instantiated it for 5 different real-world targets and their corresponding P4 architecture: the `v1model` [74] architecture for BMv2, the `ebpf_model` [242] architecture for the Linux kernel [62], the `pna` [235] architecture for the DPDK SoftNIC [58], the `tna` [106] architecture for the Tofino 1 chip [25], and the `t2na` architecture for the Tofino 2 chip [7]. All 5 instantiations implement whole-program semantics without requiring modification to the core parts of P4Testgen. We have tested the correctness of the P4Testgen oracle itself by generating input–output tests for example P4 programs of all listed architectures. Executing P4Testgen's tests using the appropriate target toolchains, we have found 19 bugs in the toolchain of the Tofino compiler and 8 bugs in the toolchain of BMv2. P4Testgen is available at the following URL: https://p4.org/projects/p4testgen.

5.2 Motivation and Challenges

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

tna/t2a target detail
☞ `tna` has ~48 extern functions and 6 programmable blocks [106]. `t2na` has over 100 externs and 7 programmable blocks.
☞ Tofino 2 adds a programmable block, the ghost thread. This block can update information related to queue depth in parallel with the packet traversing the program.
☞ In the Tofino parser, if a packet is too short to be read by an extern (extract/advance/lookahead), the packet is dropped, unless Tofino's ingress control reads the parser error variable. Then the packet header causing the exception is in an unspecified state [106, §5.2.1].
☞ The packet that enters the Tofino parser is augmented with additional information, which needs to be modeled. Tofino 1 and 2 prepend metadata to the packet [106, §5.1]. A 4-byte Ethernet frame check sequence (FCS) is also appended. The parser can parse these values into P4 data structures.
☞ If the egress port variable is not set in the P4 program, the packet is practically dropped (no unicast copy is made) [106, §5.1].
☞ The value of the output port in Tofino matters. Some values are associated with the CPU port or recirculation, some are not valid, some forward to an output port. The semantics and validity of the ports can be configured [106, §5.7].
☞ Tofino follows the Ethernet standard. Packets must have a minimum size of 64 bytes. Otherwise, the packet will be dropped [106, §7.2]. The exception to this rule is packets injected from the Tofino CPU PCIe port.
☞ The Tofino compiler provides annotations which can affect program semantics. Some annotations can alter the size of the P4 metadata structure. If not handled correctly, this can affect the size of the output packet [106, §11]. Another convenience annotation will initialize all otherwise random metadata to 0.
☞ The Tofino compiler removes all fields that are not read in the P4 program from the egress metadata structure. This influences the size of the packet parsed by the egress parser.
☞ Invalid access to header stacks in a parse loop will not cause a `StackOutOfBounds` error. Instead, execution transitions to the control block with `PARSER_ERROR_CTR_RANGE` set [106, §5.2.1].
☞ Control plane keys in the Barefoot Runtime (Bfrt) may contain dollar signs ($). When generating PTF/STF tests, these must be replaced using a compiler pass.
☞ Tofino has a metadata variable, which tells the traffic manager to skip egress processing entirely [106, §5.6].
☞ Tofino 2 has a metadata variable, which instructs the deparser to truncate the emitted packet to the specified size.

Table 5.1: A collection of `tna`/`t2a` target details that require whole-program semantics.

P4 programs on arbitrary targets. Achieving this goal requires overcoming several technical challenges, described below.

(1) Missing inter-block semantics. A P4 program only specifies the target behavior *within* the P4 programmable blocks in the architecture. It does not specify the execution order of those blocks, or how the output of one block feeds into the input of the next, i.e., target-specific semantics in the interstices between blocks. For instance, Tofino's `tna` and `t2na` architectures contain independent ingress and egress pipelines, with a traffic manager

`v1model` target detail
☞ `v1model` has ~26 extern functions and 6 programmable blocks [73].
☞ BMv2's default output port is 0 [73]. BMv2 drops packets when the egress port is 511.
☞ When using Linux virtual Ethernet interfaces with BMv2, packets that are smaller than 14 bytes produce a curious sequence of hex output (02000000) [193].
☞ BMv2 supports a special technique to preserve metadata when recirculating a packet. Only the metadata that is annotated with field_list and the correct index is preserved [73].
☞ BMv2 supports the assume/assert externs which can cause BMv2 to terminate abnormally [96].
☞ BMv2's clone extern behaves differently depending on the location where it was called in the pipeline. If recirculated in ingress, the cloned packet will have the values after leaving the parser and will be directly sent to egress. If cloned in egress, the recirculated packet will have the values after it was emitted by the deparser [73].
☞ BMv2 has an extern that takes the payload into account for checksum calculation. This means a payload must always be synthesized for this extern [73].
☞ A parser error in BMv2 does not drop the packet. The header that caused the error will be invalid and execution skips to ingress [73].
☞ All uninitialized variables are implicitly initialized to 0 or false in BMv2.
☞ Some `v1model` programs include P4Constraints, which limits the types of control plane entries that are allowed for a particular table.
☞ The table implementation in BMv2 supports the priority annotation, which changes the order of evaluation of constant table entries.

Table 5.2: A collection of `v1model` target details that require whole-program semantics.

`ebpf_model` target detail
☞ `ebpf_model` has 2 extern functions and 2 programmable blocks.
☞ The eBPF target does not have a deparser that uses emit calls. It can only filter.
☞ `extract` or `advance` have no effect on the size of the outgoing packet.
☞ A failing `extract` or `advance` in the eBPF kernel automatically drops the packet.

Table 5.3: A collection of `ebpf_model` target details that require whole-program semantics.

between them. The traffic manager can forward, drop, multicast, clone, or recirculate packets, depending on their size, content, and associated metadata. As another example, the P4 specification states that, if extracting a header fails because the packet is too short, the parser should step into `reject` and exit [236, §12.8.1]. However, the semantics after exiting the `reject` state is left up to the target: some drop the packet, others consider the header uninitialized, while others silently add padding to initialize the header. None of these behaviors are captured by the P4 program itself. P4Testgen offers features for describing such *inter-block semantics* (§5.4).

(2) Target-specific intra-block semantics. Even though P4 describes the behavior of a programmable block, targets may also have different *intra-block semantics*, i.e., they interpret the P4 code within the programmable block differently. The P4 specification delegates numerous decisions to targets, and they may not implement all parts of the specification. For instance, hardware restrictions can make it difficult to implement parser exceptions faithfully [94]. Match-action table execution can also be customized using target-specific properties (e.g., action profiles) and annotations can influence the semantics of headers and other language constructs in subtle ways. See Table 5.1, 5.2, and 5.3 for a (non-exhaustive) list of target-specific deviations. Where possible, we cited a source. Some details are not explicitly documented. As part of its whole-program semantics model, P4Testgen offers a flexible abstract machine based on an extensible class hierarchy, which makes it easy to accommodate target-specific refinements of the P4 specification.

(3) Unpredictable program behavior. Not all parts of a P4 program are well-specified by the code. For instance, reading from an uninitialized variable may return an undefined value. P4 programs may also invoke arbitrary extern functions, such as pseudo-random number generators, which produce unpredictable output. To ensure that generated tests are deterministic, P4Testgen needs facilities to track program segments that may cause unpredictable output. P4Testgen uses *taint-tracking* to keep track of unpredictable bits in the output (§5.4.4), ensuring that it never produces nondeterministic tests unless explicitly asked to do so.

(4) Complex primitives. Like other automated test generation tools, P4Testgen relies on a first-order theorem prover to compute input–output tests. However, not all primitives can easily be encoded into first-order logic—e.g., checksums and other hash functions, or programs that modify the size of the packet using dynamic values. For instance, consider a program that uses the `advance` function to increment the parser cursor by an amount that

depends on values within the symbolic input header. Modeling this behavior precisely either requires bit vectors of symbolic width, which is not well-supported in theorem provers, or branching on every possible value, which is impractical. P4Testgen uses *concolic execution* to accommodate computations which cannot be encoded into first-order logic (§5.4.5).

(5) Path explosion. By default, P4Testgen uses depth-first search (DFS) to select paths throughout the P4 program. It does not prioritize any path and explores all valid paths to exhaustion. However, real-world P4 programs often have dense parse graphs and large match-action tables, so the number of possible paths grows exponentially [140, 221]. Achieving full path coverage would require generating an excessive number of tests. P4Testgen provides *strategies* for controlling the selection of paths, including random strategies and coverage-guided heuristics that seek to follow paths containing previously unexplored statements. These strategies enable achieving full statement coverage with orders of magnitude fewer tests compared to other approaches (§5.5).

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

5.3 P4Testgen Overview

As shown in Fig. 5.1, we implement P4Testgen's execution model using symbolic execution. The tool selects a path in the program, encodes the associated path constraint in SMT logic, and then solves the constraint using an SMT solver. If it finds a solution to the

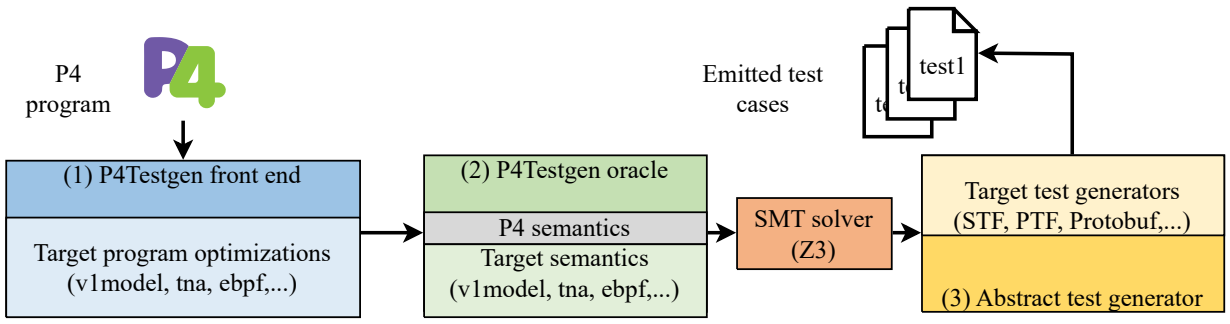


Figure 5.1: The P4Testgen test case generation process.

constraint, then it emits a test comprising an input packet, output packets, and any control-plane configuration required to execute the path. If it finds no solution, then the path is infeasible. Along with the generated tests, P4Testgen reports which segments of the program (statements, externs, actions) are covered by each test. P4Testgen's workflow can be summarized as a three-step process.

Step 1: Translate the input program and target into a symbolically executable representation. P4Testgen takes as input a P4 program, the target architecture, and the desired test framework (e.g., STF [32] or PTF [233]). It parses the P4 program and converts it into the P4C intermediate representation language (P4C-IR). P4Testgen then transforms the parsed P4C-IR into a simplified form that makes symbolic execution easier, e.g., P4Testgen unrolls parser loops and replaces runtime indices for header stacks with conditionals and constant indices. The correctness of P4Testgen's tests is predicated on the correctness of the P4C front end and these transformations.

Step 2. Execute the program and generate the test case specification. After the input program has been parsed and transformed, P4Testgen symbolically executes the program by stepping through individual AST nodes (parser states, tables, statements). By default, the P4Testgen interpreter provides a reference implementation for each P4 construct.

However, each step can be customized to reflect target-specific semantics by overriding methods in the symbolic executor. Targets must also define whole-program semantics (§5.4) that describe how individual P4 blocks are chained together (i.e., the order in which a packet traverses the P4 blocks), what kind of parsable data can be appended or prepended to packets (e.g., frame check sequences), and how target system data (also called intrinsic metadata) is initialized. Importantly, this target-specific information can be inferred from the documentation for the P4 architecture or the target itself. Detailed knowledge of hardware microarchitecture is not necessary.

Step 3. Emit the test case. Once P4Testgen has executed a path, it emits an abstract test specification, which describes the expected system state (e.g., registers and counters) and output packets for the given packet input and control-plane configuration. This abstract test specification is then concretized for execution on different test frameworks (STF, PTF, etc.).

5.3.1 P4Testgen in Action

As an example to illustrate the use of P4Testgen, consider two P4 programs, as shown in Fig. 5.2, written for a fictitious, BMv2-like target with a single parser and control block.

Example 1. In the first program (Fig. 5.2a), Ethernet packets are forwarded based on a table that matches on the EtherType field. There are four different input–output pairs that could be generated. The first pair is a valid Ethernet packet, but no table entries are associated with the input. Since the default action is `noop`, the output port of the packet does not change. The second pair is a configuration with a table entry that executes `set_out` whenever `h.eth.type` matches a given value. Since the program previously set `h.eth.type` to `0xBEEF`, the table entry must match on `0xBEEF`. The output port is defined by the control plane. The third pair is similar, except `noop` is chosen as the action, which does not alter

```
parser Parser(...) {
    pkt.extract(hdr.eth);
    transition accept;
}
control Ingress(...) {
    action set_out(bit<9> port) {
        meta.output_port = port;
    }
    table forward_table {
        key = { h.eth.type: exact; @name("type") }
        actions = { noop; // Default action.
                    set_out; }
    }
    h.eth.type = 0xBEEF;
    forward_table.apply();
}
```

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

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

```
parser Parser(...) {
    pkt.extract(hdr.eth);
    transition accept;
}
control Verify(...) {
    meta.checksum_err = verify_checksum(
    hdr.eth.isValid(),
    {hdr.eth.dst, hdr.eth.src},
    hdr.eth.type);
}
control Ingress(...) {
    if (meta.checksum_err == 1) {
        mark_to_drop(); // Drop packet.
    }
}
```

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

(b) P4 program that validates the Ethernet checksum.

Figure 5.2: P4Testgen test examples. "Port" denotes the input–output port. "Size" is the packet bit-width.

the output port. For the last input pair, the packet is too short and the `extract` call fails. Hence, the target stops parsing and continues to the control block. For this particular target, the packet will be emitted, but `forward_table` will not execute because the match key is uninitialized. P4Testgen is able to generate four distinct tests for this program. For input–output pairs 2 and 3, P4Testgen synthesizes control plane entries, which execute the appropriate action. For input–output pair 4, P4Testgen makes use of its *packet sizing* (§ 5.4.3.1) implementation to generate a packet that is too short. P4Testgen uses *taint tracking* (§ 5.4.4) to identify that `h.eth.type` is uninitialized. Since this target will not match on uninitialized keys, P4Testgen does not generate an entry for `forward_table`.

Example 2. The second program (Fig. 5.2b) parses an Ethernet header. If it is valid (line 7), the program tests whether the checksum computed on `hdr.eth.dst` and `hdr.eth.src` (lines 6–9) corresponds to the value in field `hdr.eth.type` (line 10).[2] If not, `meta.checksum_err` is set to `true` and the packet is dropped. This program produces three distinct input–output pairs. The first pair is an input packet that is too short, which causes the Ethernet header to be invalid. Hence, `verify_checksum` is not executed, the error is not set, and the packet is forwarded. The second and third input–output pairs include a valid Ethernet header. In the second pair, `hdr.eth.type` matches the computed checksum value and the packet is forwarded. In the third pair, the value does not match and the packet is dropped. Note that for input–output pairs 2 and 3, P4Testgen uses *concolic execution* (§ 5.4.5) to model the checksum computation. P4Testgen picks a random concrete assignment to `hdr.eth.dst` and `hdr.eth.src`, computes the checksum, and compares the result to `hdr.eth.type`. As there are no other restrictions on the value of `hdr.eth.dst` and `hdr.eth.src`, P4Testgen produces tests where the checksum either matches (test 3) or does not match (test 2).

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

Summary. As shown, P4Testgen prefers to maximize program coverage even though it may lead to path explosion. The behaviors exhibited by the tests in Fig. 5.2 are possible on the underlying targets and testing them is important. Indeed, we have used P4Testgen to uncover a variety of bugs in compilers, drivers, and software models—see §5.7 for details. Moreover, these bugs were not for toy programs or early versions of systems under development. Rather, they were found in production code for mature systems that had already undergone extensive validation with traditional testing.

5.4 An Extensible Execution Model for P4

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

5.4.1 P4Testgen's Abstract Machine

Fig. 5.3 summarizes the design of the abstract machine that powers P4Testgen's symbolic executor. It has standard elements, such as a stack frame, symbolic environment, and so on, as well as a continuation, which encodes the rest of the computation. A full treatment of continuations [181] is beyond the scope of our work. In a nutshell, continuations make it easy to encode non-linear control flow such as packet recirculation, which many P4 architectures support, and they also preserve execution contexts across paths, which is helpful for implementing different path selection heuristics.

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

Figure 5.3: Execution state for P4Testgen's abstract machine.

5.4.2 The Pipeline Template

Pipeline templates are a succinct mechanism for describing the *pipeline state* and *control flow* for an architecture—and with those two, its inter-block semantics. By default, they capture the common case where the state associated with the packet simply flows between P4-programmable blocks in a straightforward manner—e.g., by copying output variables of one block to the input variables of the next. P4Testgen also handles more complicated forms of packet flow in the architecture, such as recirculation, but this requires writing explicit code against the abstract machine.

5.4.2.1 Pipeline State

Pipeline state describes the per-packet data that is transferred between P4-programmable blocks. Fig. 5.4 shows the pipeline state description for the `v1model` in a simple C++ DSL. The objects listed in the data structure are mapped onto the programmable blocks in the top-level declaration of a P4 program (shown in comments). The declaration order of these objects determines the order in which the blocks are executed by default, but this

```
ArchitectureSpec("V1Switch", {
  // parser Parser<H, M>(packet_in b,
  //                     out H parsedHdr,
  //                     inout M meta,
  //                     inout standard_metadata_t sm);
  {"Parser", {none, "*hdr", "*meta", "*sm"}},
  // control VerifyChecksum<H, M>(inout H hdr,
  //                              inout M meta);
  {"VerifyChecksum", {"*hdr", "*meta"}},
  // control Ingress<H, M>(inout H hdr,
  //                       inout M meta,
  //                       inout standard_metadata_t sm);
  {"Ingress", {"*hdr", "*meta", "*sm"}},
  // control Egress<H, M>(inout H hdr,
  //                      inout M meta,
  //                      inout standard_metadata_t sm);
  {"Egress", {"*hdr", "*meta", "*sm"}},
  // control ComputeChecksum<H, M>(inout H hdr,
  //                               inout M meta);
  {"ComputeChecksum", {"*hdr", "*meta"}},
  // control Deparser<H>(packet_out b, in H hdr);
  {"Deparser", {none, "*hdr"}}});
```

Figure 5.4: The pipeline state for the `v1model` architecture. Comments describe the associated P4 block. The word `none` indicates parameters irrelevant to the state.

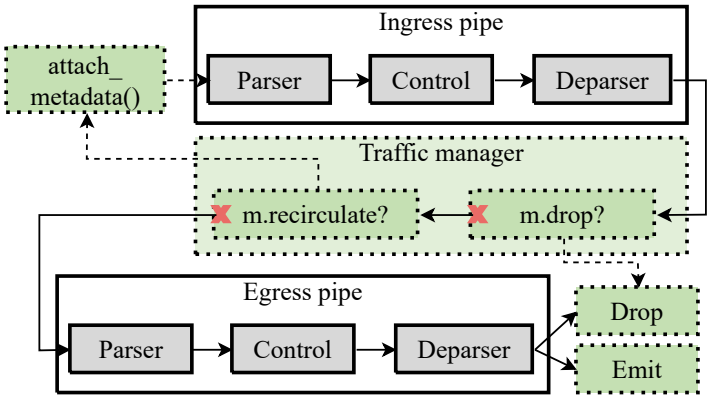
can be overridden by the pipeline control flow based on a packet's per-packet data values. Arguments with the same name are threaded through the programmable blocks in execution order. For example, the `*hdr` parameter in the parser is first set to undefined, as it is used in an `out` position as shown in the comments in Fig. 5.4. After executing the parser, it is copied into the checksum unit, then to the ingress control, etc.

5.4.2.2 Pipeline Control Flow

P4Testgen allows extension developers to provide code to model arbitrary interpretation of the pipeline state. Fig. 5.5 shows an example of a P4 program snippet being interpreted in the context of P4Testgen's pipeline control flow. The target is a fictitious target with an implicit traffic manager between ingress and egress pipelines. The green dashed segments in the figure are target-defined and interpret the variables set in the `Ingress` control block. If

```
control Ingress(...) {
  if (hdr.ip.ttl == 0) {
    // Drop packet
    m.drop = 1;
  }
  if (hdr.ip.ttl == 1) {
    // Resubmit packet
    resubmit.emit(m);
  }
}
Pipeline(
I_Parser(), Ingress(),
    I_Deparser(),
E_Parser(), Egress(),
    E_Deparser()
) pipe;
```

(a) P4 program snippet that sets metadata state.



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

Figure 5.5: P4Testgen's pipeline control flow.

`m.drop` is set, the packet will be dropped by the traffic manager, skipping execution of the entire egress pipeline. If the `resubmit.emit()` is called, `m.recirculate` will implicitly be set, causing P4Testgen to reset all metadata and reroute the execution back to the ingress parser. We have modeled this control flow for targets such as `v1model`, `tna`, and `t2na`.

5.4.3 Handling Target-Specific Behavior

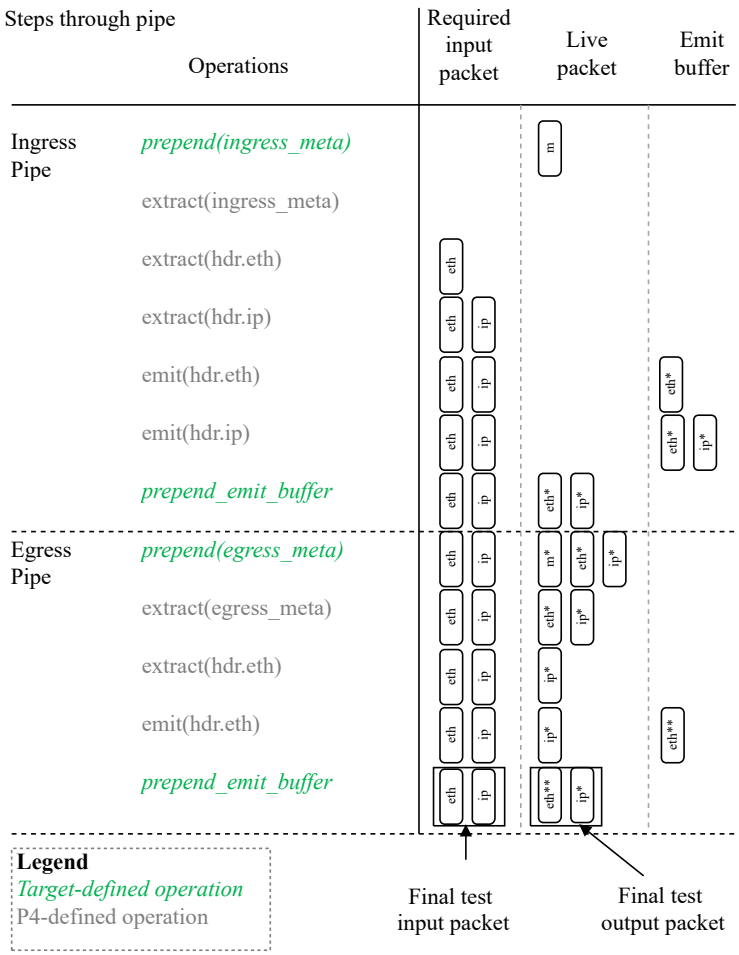
Targets have different intra-block semantics and diverge in their interpretation of core P4 language constructs. P4Testgen is structured such that every function in the abstract machine can be overridden by target extensions. For example, the `v1model` P4Testgen extension overrides the canonical P4Testgen table continuation to implement its own annotation semantics (e.g., the "priority" annotation, which reorders the execution of constant table entries based on the value of the annotation). Targets may also reinterpret core parsing functions (e.g., `extract`, `advance`, `lookahead`).

5.4.3.1 Our Approach to Packet-Sizing

One area where there is significant diversity among targets is in the semantics of operations that change the size of the packet. Some paths in a P4 program are only executable with a specific packet size. P4 externs such as `extract` can throw exceptions when the packet is too short or malformed. These packet paths are often sparsely tested when developing a new P4 target and toolchain. Particularly on hardware targets, packets with an unexpected size may not be parsed as expected. Correspondingly, P4Testgen must be able to control the size of the input packet (Challenge 2). And, since some of these inputs may trigger parser exceptions, it also needs to model the impact these exceptions have on the content and length of the packet. P4Testgen implements packet-sizing by making the packet size a symbolic variable in the set of path constraints. This encoding turns out to be non-trivial. Since the required packet size to traverse a given path is now a symbolic variable, it is only known after the SMT solver is invoked. However, at the same time, externs in P4 manipulate the size of the packets (e.g., `extract` calls shorten while `emit` calls lengthen the packet), which requires careful bookkeeping in first-order logic. Targets also react differently to specific packet sizes (e.g., BMv2 produces garbage values for 0-length packets [193], whereas Tofino drops packets smaller than 64 bytes [106, §7.2]). Lastly, some targets add and remove content from the packet (e.g., Tofino adds internal metadata to the packet [106, §5.1]). Any packet-sizing mechanism needs to handle these challenges, while remaining target independent.

Our approach is to model packet-sizing as described in the P4 specification. For each program path, we calculate the *minimum* header size required to successfully exercise the path without triggering a parser exception. The packet-sizing model defines and manipulates three symbolic bit vector variables: the required input packet (I), the live packet (L), and the emit buffer (E). The input packet I represents the *minimum* header content required to reach a particular program point without triggering an exception. The live packet L

```
parser IngressParser(...) {
  state start {
    pkt.extract(ingress_meta);
    pkt.extract(hdr.eth);
    pkt.extract(hdr.ipv4);
  }
}

control IngressControl(...) {
  apply {}
}

control IngressDeparser(...) {
  apply {
    pkt.emit(ingress_meta);
    pkt.emit(hdr.eth);
    pkt.emit(hdr.ipv4);
  }
}

parser EgressParser(...) {
  state start {
    pkt.extract(egress_meta);
    pkt.extract(hdr.eth);
  }
}

control EgressControl(...) {
  apply {}
}

control EgressDeparser(...) {
  apply {
   pkt.emit(hdr.eth);
  }
}

Pipeline(
  IngressParser(), IngressControl(),
  IngressDeparser(),
  EgressParser(), EgressControl(),
  EgressDeparser()
) pipe;
```



(a) Extern sequence manipulating Ethernet and IPv4 headers.

(b) Change in the packet sizing variables as P4Testgen steps through the program. Each block corresponds to a P4 header.

Figure 5.6: Packet-sizing for a Tofino program.

represents the packet header content available to the interpreter stepping through the P4 program, e.g., `extract` will consume content from L. The emit buffer E is a helper variable that accumulates the headers produced by `emit`. This is necessary to preserve the correct order of headers, as prepending headers to L each time `emit` is executed would cause it to be inverted.

Initially, all variables are zero-width bit vectors. While traversing the program, parser externs (e.g., `extract` or `advance`) in the P4 program slice data from the live packet L. If L is empty (meaning we have run out of packet header data), P4Testgen allocates a new symbolic packet header and adds it to I. Targets may augment the input packet with custom parsable data (e.g., metadata) that reduces the input packet needed to avoid triggering a parser exception. Correspondingly, this content is added to the live packet variable L. Once P4Testgen has finished executing a path, I will denote the content of the final input packet in the generated test. L, on the other hand, will correspond to the content of the expected packet output. Fig. 5.6 illustrates the variables used for an example pipeline.

This design also handles multi-parser, multi-pipe targets, such as Tofino. Each Tofino pipeline has two parsers: ingress and egress. The egress parser receives the packet (L) after the ingress pipeline and traffic manager. If the egress parser runs out of content in L, P4Testgen must again append symbolic content to I, increasing the size of the minimum packet required to parse successfully.

5.4.4 Controlling Unpredictable Behavior

Many P4 programs are non-deterministic, which can lead to unpredictable outputs (Challenge 3). To avoid generating "flaky" tests, we use taint analysis [204]. As P4Testgen steps through the program, we keep track of which bits have a known value (i.e., "untainted"), and which bits have an unknown value (i.e., "tainted"). For example, a declaration of a variable that is not initialized and reads from random memory will be designated as tainted.

The result of any operation that references a tainted variable will also be tainted. Later, when generating tests, we use the taint to avoid generating tests that might fail—i.e., due to testing tainted values. For example, if the output packet contains taint, we know that certain bits are unreliable. We use test-framework-specific facilities (e.g., "don't care" masks) to ignore tainted output bits. On the other hand, if the output port is tainted and the test framework does not support wildcards for the output port, P4Testgen cannot reliably predict the output, so we drop the test and issue a warning.

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

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

Prototyping extensions using taint. Another useful byproduct of taint analysis is the ability to easily prototype a P4Testgen extension and its externs. Rather than implementing the entire P4Testgen extension at once, a developer can substitute taint variables for the parts that may need time-intensive development (a form of *angelic programming* [24]). By constraining the non-determinism of the unimplemented parts of the extension, it is possible to generate deterministic tests early. We used this approach to generate initial stubs for many externs (e.g., checksums, meters, registers) before implementing them precisely.

5.4.5 Supporting Complex Functions

To handle complex functions that cannot be easily encoded into first-order logic (Challenge 4), P4Testgen uses *concolic execution* [89, 206]. Concolic execution is an advanced technique that combines symbolic and concrete execution. In a nutshell, it leaves hard-to-model functions unconstrained initially, and adds constraints later using the concrete implementation of the function. The `verify_checksum` function described in § 5.3.1 is an example where concolic execution is necessary. The checksum computation is too complex to be expressed in first-order logic. Instead, we model the return value of the checksum as an uninterpreted function dependent on the input arguments of the extern. While P4Testgen's interpreter steps through the program, this uninterpreted function acts as a placeholder. If the function becomes part of a path constraint, the SMT solver is free to fill it in with any value that satisfies the constraint.

Once we have generated a full path, we need to assign a concrete value to the result of the uninterpreted function. First, we invoke the SMT solver to provide us with concrete values for the input arguments of the uninterpreted function that satisfy the path constraints we have collected on the rest of the path. Second, we use these input arguments as inputs to the actual extern implementation (e.g., the hash function executed by the target). Third, we add equations to the path constraints that bind all the values we have calculated to the

appropriate input arguments and output of the function. We then invoke the solver a second time to assess whether the result computed by the concrete function satisfies all of the other constraints in the path. If so, we are done and can generate a test with all the values we calculated.

Handling unsatisfiable concolic assignments. In some cases, the newly generated constraints cannot be satisfied using the inputs chosen by the SMT solver. In practice, retrying by generating new inputs may not lead to a satisfiable outcome. Before discarding this path entirely, we try to apply function-specific optimizations to produce better constraints for the concolic calculation. For example, the `verify_checksum` function (see also §5.3.1) tries to match the computed checksum of input data with an input reference value. If the computed checksum does not match with the reference value, `verify_checksum` reports a checksum mismatch. Instead of retrying to find a potential match, we add a new path that forces the reference value to be equal to the computed checksum. This path is satisfiable if the reference value is derived from symbolic inputs, which is often the case. Note that in situations where the reference value is a constant, we are unable to apply this optimization.

5.5 Path Selection Strategies

Methodologies that assess the program coverage of tests have become standard software engineering practice. While path coverage is often infeasible (as the number of paths grows exponentially), statement coverage, also known as line coverage, has been proposed as a good metric for evaluating a test suite [35]. P4Testgen allows users to pick from several different path selection strategies to produce more diverse tests, including Random Backtracking and Coverage-Optimized Search. As the name suggests, Random Backtracking simply jumps back to a random known branch point in the program once P4Testgen has generated a test.

Coverage-Optimized Search is similar to the concept with the same name in Klee [35]. After a new test has been generated, it selects the first path from all unexplored paths it has seen so far that will execute P4 statements that have not yet been covered. If no path with new statements can be found, Coverage-Optimized Search falls back to random backtracking until a path with new statements is discovered. This greedy search covers new statements quickly, but at the cost of higher memory usage (because it accumulates unexplored paths with low potential) and slower per-test case performance. We measure how these strategies perform on large P4 programs in §5.7.3. Our path selection framework is extensible, allowing us to integrate many different selection strategies. We can easily add other success metrics, such as table, action, or parser state coverage.

Targeted test generation with preconditions. Path selection strategies guide test case generation towards a goal, but they do not select for a specific type of test. P4Testgen also gives users the ability to instrument their P4 program with a custom extern (`testgen_assume`). This P4Testgen-intrinsic extern adds a path constraint on variables accessible within the P4 program (e.g., `h.eth_hdr.eth_type == 0x0800`), which forces P4Testgen to only produce tests that satisfy the provided constraint. Assume statements are similar to p4v's assumptions [140], Vera's NetCTL constraints [221], or Aquila's LPI preconditions [241]. We study the effect of these constraints in §5.7.3.

Instrumenting fixed control-plane configurations. Network operators generally have restricted environments in which only a limited set of packets and control plane configurations are actually valid. Similar to Meissa [264] and SwitchV [9], we are developing techniques to instrument a particular fixed control plane configuration before generating tests. We are looking into a specification method that allows users to only generate tests that comply with their environment assumptions. As an initial step in this direction, P4Testgen implements SwitchV's P4Constraints framework (§5.6.1.1).

5.6 Implementation

P4Testgen is written as an extension to P4C using about 28k lines of C++ code, including both P4Testgen core and its extensions. To resolve path constraints, P4Testgen uses the Z3 [53] SMT solver.

Different levels of precision can lead to a highly varying number of branches being generated. A rather simple `max` extern, which returns the larger of two values, can be implemented by simply picking one of the values and adding a constraint that it is larger. The function can also be modeled to branch into three different paths instead: one where the input values are equal, one where the first value is larger, and one where the second value is larger.

Interacting with the control plane. P4Testgen uses the control plane to trigger some paths in a P4 program (e.g., paths dependent on parser value sets [236, §12.11], tables, or register values). Since P4Testgen does not perform load or timing tests, the interaction with the control plane is mostly straightforward. For each test that requires control-plane configuration, P4Testgen creates an abstract test object, which becomes part of the final test specification. For tables, P4Testgen creates forwarding entries, and if the test framework provides support, it can also initialize externs such as registers, meters, counters, and check their final state after execution. In general, richer test framework APIs give P4Testgen more control over the target—e.g., STF lacks support for range-based match types, which means some paths cannot be executed.

5.6.1 P4Testgen Extensions

Table 5.4 lists the targets we have instantiated with P4Testgen. We also list the LoC required by each extension, noting that `tna` and `t2na` share a significant amount of code. Further, `v1model` LoC are inflated because of the P4Constraints parser and lexer implementation

Architecture	Target	Test back end	C/C++ LoC
`v1model`	BMv2	STF, PTF, Protobuf, Meta	7651
`tna`	Tofino 1	STF, PTF	499 (3525 shared)
`t2na`	Tofino 2	STF, PTF	502 (3525 shared)
`ebpf_model`	Linux Kernel	STF	981
`pna`	DPDK SoftNIC	PTF, Meta	2065

Table 5.4: P4Testgen extensions. The core of P4Testgen is 12284 LoC.

specific to the `v1model` extension. We modeled the majority of the Tofino externs based on the P4 Tofino Native Architecture (TNA) available in the Open-Tofino repository [106]. Each extension also contains support for several test frameworks. The `v1model` instance supports PTF, STF, Protobuf [91] messages, and the serialization of metadata state. The Tofino instance supports PTF and STF. The eBPF instance supports STF. The Portable NIC Architecture (PNA) [235] instance only has metadata serialization.

5.6.1.1 `v1model`

P4Testgen supports the `v1model` architecture, including externs such as `recirculate`, `verify_checksum`, and `clone`. The `clone` extern requires P4Testgen's entire toolbox to model its behavior, so we explain it in detail below.

Implementing clone. The `clone` extern duplicates the current packet and submits the cloned packet into the egress block of the `v1model` target. It alters subsequent control flow based on the place of execution (ingress vs. egress control block). Depending on whether `clone` was called in the ingress vs. egress control block, the content of the recirculated packet will differ. Further, which user metadata is preserved in the target depends on input arguments to the `clone` extern.

We modeled this behavior entirely within the BMv2 extension to P4Testgen without having to modify the core code of P4Testgen's symbolic executor. We use the pipeline control flow and continuations to describe `clone`'s semantics, concolic execution to compute

the appropriate clone session IDs, and taint tracking to guard against unpredictable inputs.

P4Constraints. P4Testgen's BMv2 extension also implements the P4Constraints framework [9] for `v1model`. P4Constraints annotates tables to describe which control plane entries are valid for this table. P4Constraints are needed for programs such as `middleblock.p4` [227], which models an aggregation switch in Google's Jupiter network [214] that only handles specific entries. To generate valid tests for such programs, P4Testgen must accommodate constraints on entries. It does so by converting P4Constraints annotations into its own internal predicates, which are applied as preconditions, restricting the possible entries, and hence, the number of generated tests (§5.7).

5.6.1.2 tna/t2na

We have implemented the majority of externs for `tna` and `t2na`, including meters, checksums, and hashes. For others, such as registers, we make use of rapid prototyping using taint. Our `t2na` extension leverages much of the `tna` extension, but `t2na` is richer, so it took more effort to model its capabilities. Not only does `t2na` use different metadata, it also adds a new programmable block ("ghost") and doubles the number of externs. Also, both `tna` and `t2na` support parsing packets at line rate, which is significantly more complex than BMv2 [106, §5].

Parsing packets with Tofino. The Tofino targets prepend multiple bytes of metadata to the packet [106, §5.1]. As an Ethernet device, they also append a 32-bit frame check sequence (FCS) for each packet. Both the metadata and FCS can be extracted by the parser but are not part of the egress packet in the emit stage. If the packet is too short and externs in the parser trigger an exception, Tofino drops the packet in the ingress parser, but not in the egress parser [106, §5.2.1]. However, if the ingress control block reads from the `parser_error` metadata variable, the packet is not dropped and instead skips the remaining

parser execution and advances to the ingress control block. The content of the header that triggered the exception is unspecified in this case. We model this behavior entirely in the Tofino instantiations of P4Testgen. We treat the metadata, padded content, and FCS as taint variables that are prepended to the live packet L. Since Tofino's parsing behaves differently from the description in the P4 specification, we extend the implementations of `advance`, `extract`, and `lookahead` in the Tofino extensions to model the target-specific behavior.

5.6.1.3 EBPF_MODEL

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

5.6.1.4 PNA

PNA [235] is a P4 architecture describing the functionality of end-host networking devices such as SmartNICs. A variety of targets using the `pna` architecture have been put forward by Xilinx [3], Keysight [113], NVIDIA [124], AMD [87], and Intel [107]. We have instantiated a P4Testgen extension for a publicly available `pna` instance, the DPDK SoftNIC [58]. Since there are no functional testing frameworks (e.g., PTF or STF) yet available for this target,

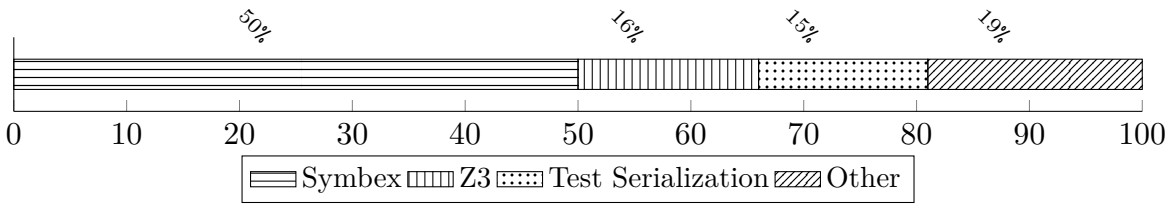


Figure 5.7: Average CPU time spent in P4Testgen.

we generate abstract test templates, which describe the input–output behavior and expected metadata after each test. By generating these abstract tests, we can already perform preliminary analysis on existing `pna` programs (§5.7.3).

5.7 Evaluation

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

5.7.1 Performance

To evaluate P4Testgen's performance when generating tests, we measured the percentage of cumulative time spent in three major segments: 1) stepping through the symbolic executor, 2) solving Z3 queries, 3) serializing an abstract test into a concrete test. Fig. 5.7 shows P4Testgen's CPU time distribution for generating 10000 tests for the larger programs listed in Table 5.5. In general, solving path constraints in Z3 accounts for around 16% of the overall CPU time. P4Testgen spends the majority of time in the symbolic executor. This is expected, as we prioritized extensibility and debuggability for P4Testgen's symbolic execution engine, not performance. We expect performance to improve as the tool matures. From informal conversations, we are aware that P4Testgen generates tests with efficiency on the same order as SwitchV's `p4-symbolic` tool does.

5.7.2 Correctness

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

Producing valid tests for diverse P4 programs. To ensure that P4Testgen's interpretations of P4 and target semantics are correct, we generated tests for a suite of programs and executed them on the target. For `v1model`, `pna`, and `ebpf_model`, we selected all the P4 programs available in the P4C test suite. For Tofino, we used the programs available in the P4Studio SDE and a selected set of compiler tests given to us by the Tofino compiler team. The majority of these programs are small and easy to debug, as they are intended to test the Tofino compiler. In total, we tested on 458 Tofino 1, 191 Tofino 2, 507 BMv2, 62 PNA, and 30 eBPF programs.

We used P4Testgen to generate 10 input–output tests with a fixed random seed for each of the above programs. We then executed these tests using the appropriate software model and test back ends. In fact, on every repository commit of P4Testgen, we execute P4Testgen on all 5 extensions and their test back ends (Table 5.4), totaling more than 2800 P4 programs and 10 tests per program. We used this technique to progressively sharpen our semantics over the course of a year, running P4Testgen millions of times. If the execution of a test did not lead to the output expected by P4Testgen, we investigated. Sometimes, it was a bug in P4Testgen, which we fixed. Sometimes, the target was at fault and we filed a bug (see §5.7.4).

Producing valid tests for large P4 programs. For the `v1model`, we chose two actively maintained P4 models of real-world data planes: `middleblock.p4` (§ 5.6.1.1) and

`up4.p4` [142]. `up4.p4` is a P4 program developed by the Open Networking Foundation (ONF) that models the data plane of 5G networks. We have considered other programs but they were either written in $P4_{14}$ [215] or not sufficiently complex to provide a useful evaluation [43]. For `tna/t2na`, we generate tests for the appropriate version of `switch.p4`, the most commonly used P4 program for the Tofino programmable switch ASIC. We execute the generated tests on either BMv2 or the Tofino model (a semantically accurate software model of the Tofino chip). For each target, we generate 100 PTF tests. The eBPF kernel target does not have a suite of representative programs. Instead, we generated tests for P4C's sample programs. The tests we have generated pass, showing that we can correctly generate tests for large programs. `pna` on the DPDK SoftNIC does not have an end-to-end testing pipeline available yet, but we still generate tests for its programs. As a representative program, we picked `dash_pipeline.p4`, which models the end-to-end behavior of a programmable data plane in P4 [223]. `dash_pipeline.p4` is still under development, but is already complex enough to generate well over a million unique tests.

5.7.3 Coverage

When generating tests, P4Testgen tracks the statements (after dead-code elimination) covered by each test. Once P4Testgen has finished generating tests, it emits a report that details the total percentage of statements covered. We use this data to identify any P4 program features that were not exercised. For example, some program paths may only be executable if the packet is recirculated.

How well does P4Testgen cover large programs? We tried to exhaustively generate tests for the programs chosen in the previous section. Table 5.5 provides an overview of the number of tests generated for each program (this number correlates with the number of possible branches as modeled by P4Testgen) and the best statement coverage we have

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

Table 5.5: Coverage statistics for large P4 programs using DFS (measured 2023-09-01).

achieved using DFS. As expected, for the `switch.p4` programs of `tna` and `t2na`, we generate too many paths to terminate in a reasonable amount of time. For the `switch.p4` programs, we list the coverage we achieved before halting generation after the millionth test.

How does path selection help with statement coverage? Table 5.5 shows that the number of tests generated for larger P4 programs can be overwhelming. In practice, users want tests with specific properties, which necessitates the use of path selection strategies. We measure the effect of P4Testgen's path selection strategies (§5.5). We select `middleblock.p4` and `up4.p4` as representative sample programs for `v1model`. For `tna` and `t2na`, we select `simple_switch.p4`, which we patched such that all statements in the program are reachable. We have chosen `simple_switch.p4` for two reasons: (i) we have not implemented all features to fully cover `switch.p4` (specific register/meter configurations, recirculation) to achieve full statement coverage,[3] and (ii) `simple_switch.p4` is an open-source program available at the OpenTofino repository [106]. `simple_switch.p4` is still a complex Tofino program: it produces over 30 million unique, valid tests. We generate tests with each strategy until we hit 100% statement coverage. We compare Random Backtracking and our Coverage-Optimized Search to standard DFS. We measure the total number of tests needed to achieve coverage across a sample of 10 different seeds.

Fig. 5.8 shows the mean coverage across 10 seeds over 1000 timesteps for `simple_-`

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

Program	Metric (Median)	Strategy		
		DFS	Random Backtracking	Coverage-Optimized Search
middleblock.p4	Tests	25105	956	86
	Time per Test	~0.05s	~.08s	~.17s
	Total time	~1321s	~81s	~11s
up4.p4	Tests	12932	2463	3581
	Time per Test	~0.06s	~0.07s	~0.06s
	Total time	~726s	~169s	~243s
simple_switch.p4	Tests	*	*	4612
	Time per Test	~0.07s	~0.09s	~0.12s
	Total time	*	*	~555s
dash_pipeline.p4	Tests	*	*	63
	Time per Test	~0.06s	~0.13s	~0.41s
	Total time	*	*	~22s

Table 5.6: Path selection results for 100% statement coverage on representative P4 programs for 10 different seeds. "*" indicates that the strategy did not achieve 100% coverage within 60 minutes.

`switch.p4`. We stopped a heuristic if it did not achieve 100% coverage within an hour of generating tests. Only Coverage-Optimized Search reliably accomplishes full coverage in this time frame and outperforms Random Backtracking and DFS by a wide margin. Coverage-Optimized Search always outperforms DFS and generally outperforms Random Backtracking. In some cases, however, (e.g., `up4.p4`) Coverage-Optimized Search is not sophisticated enough to find the path that covers a sequence of statements. In those cases, it will perform similarly to Random Backtracking. Table 5.6 shows detailed results for all selected programs.

How do preconditions affect the number of generated tests? We conducted a small experiment to measure the impact of applying preconditions and simplified extern semantics on `middleblock.p4`. We measured the number of generated tests when fixing the input packet size (thus avoiding parser rejects in externs) and applying SwitchV's P4Constraints. Fig. 5.9 shows the results. The number of generated tests can vary widely, based on these input parameters. Applying the input packet size and the P4Constraints table entry restrictions can reduce the number of generated tests by as much as 71%. Adding

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

Table 5.7: Effect of preconditions on the number of tests generated for `middleblock.p4`. Fixed packet size is 1500B.

`testgen_assume` (§5.5) statements, which mandates that we only produce packets with TCP/IP headers, reduces the generated tests by 95%. Table 5.7 has detailed statistics.

What are the limits of P4Testgen's statement coverage? There are P4 programs where P4Testgen cannot achieve full statement coverage. An example is `blink.p4` [101], a P4 program where statement execution depends on the timestamp metadata field that is set by the target when a packet is received. Since P4Testgen cannot control the initialization of the timestamp for BMv2 (yet), we are unable to cover any statement depending on it. Other tools such as FP4 [257] and P4wn [119] are able to cover these statements as they generate packet sequences that may eventually cause the right timestamp to be generated. This limitation is not insurmountable. In the future, we plan to mock timestamps using a match-action table, or add an API for controlling timestamps directly.

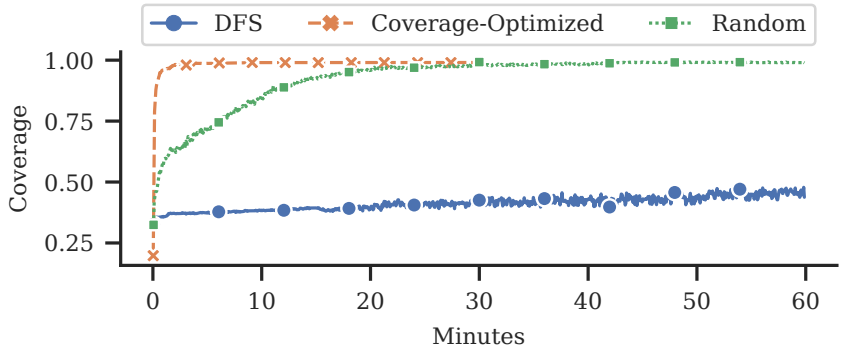


Figure 5.8: Path selection strategy performance on `simple_switch.p4`.

5.7.4 P4Testgen in Practice

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

Generating tests for abstract network-device models. An increasingly popular use case of P4 is to use it as a modeling language to describe network data planes [9, 223]. Often, these data plane models lack tests. P4Testgen can exhaustively generate tests for the P4 data plane model, where the tests also satisfy particular coverage criteria. Further, because P4Testgen is extensible, a developer modeling their device can use arbitrary P4 architectures. For example, the DASH [223] and SwitchV [9] developer teams are interested in applying P4Testgen to their data plane models written for the `pna` and `v1model` architectures.

5.7.4.1 Bugs

For any validation tool, the bottom line is whether it effectively finds bugs, particularly in mature, well-tested systems. To evaluate P4Testgen's effectiveness, we used the workflow described in §5.7.2, by running P4Testgen on each program in the appropriate test suite. Table 5.9 summarizes the bugs we found. Table 5.8 provides details on the bugs we have

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

Table 5.8: BMv2 bugs found by P4Testgen.

filed for BMv2. For confidentiality reasons, we are unable to provide details on Tofino bugs.

What are the bugs we are interested in? We report only *target stack bugs*—i.e., a bug in the software or hardware stack. We consider a target stack bug any failing test that was generated by P4Testgen but was not an issue with P4Testgen itself. This includes

Bug Type	Bug Cause	BMv2	Tofino	**Total**
Exception	Untested packet path	2	7	9
	Control plane	5	1	6
	Untested packet size	0	2	2
	Untested device function	0	1	1
	Total	7	10	17
Wrong Code	Untested packet path	0	5	5
	Control plane	1	2	3
	Untested packet size	0	0	0
	Untested device function	0	1	1
	Total	1	9	10
Total		**8**	**19**	**27**

Table 5.9: Bugs in targets discovered by P4Testgen.

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

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

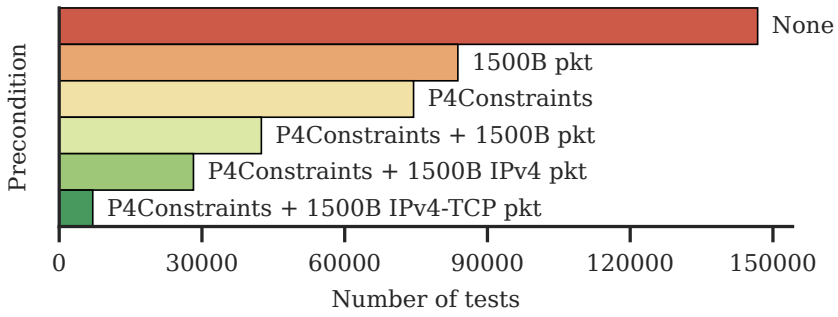


Figure 5.9: Effects of preconditions on the total number of tests generated for `middleblock.p4`.

What features of P4Testgen were important for finding a bug? Eight of the total 27 bugs we have found were triggered by P4Testgen synthesizing table and extern configurations. Two were triggered by P4Testgen implementing a detailed model of extern functions. Two were triggered by P4Testgen generating tests with unexpected packet sizes. The remaining bugs were caused because P4Testgen's generated tests exercised untested

Tool	Input generation method	Synthesizes control-plane?	Multi-target?	Models target semantics?	Data plane coverage metric
Meissa [264]	Symbex.	×	×	✓	Symbolic model
SwitchV [9]	Symbex.	×	×	✓	Symbolic model, Assertions
`p4pktgen` [160]	Symbex.	✓	×	×	Symbolic model
Gauntlet [196]	Symbex.	×	✓	×	Symbolic model
PTA [29]	Fuzzing	×	✓	×	Symbolic model (`p4v`)
DBVal [129]	Fuzzing	×	✓	×	Tables, Actions
FP4 [257]	Fuzzing	×	✓	×	Actions
P6 [212]	Fuzzing	×	✓	×	Symbolic model
P4Testgen	Symbex.	✓	✓	✓	Symbolic model, source code

Table 5.10: P4 tools generating input–output tests. Data plane coverage describes how the tool measures coverage of the generated inputs. Symbex. abbreviates symbolic execution.

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

Reachability bugs in P4 programs. A side-effect of P4Testgen's support for explicit coverage heuristics is its ability to detect reachability bugs in P4 programs. In some cases, Coverage-Optimized Search is unable to cover a particular program statement. This may be because of failures in the heuristic, but often the code is simply non-executable—i.e., dead. We encountered several instances of such dead code for proprietary and public production-grade programs [194]. The developers were usually appreciative of our bug reports, which occurred in complex programs that are difficult to debug, especially early in the development process.

5.8 Related Work

Testing P4 toolchains. Other tools focus on validating P4 implementations by generating test inputs. Table 5.10 provides a summary. Compared to P4Testgen, these tools are typically tailored to a single target or use case. P4Testgen relies on formal semantics to compute inputs and outputs, avoiding running a second system to produce the output [9, 257]. In particular, developers using P4Testgen do not need to understand the semantics of the P4 program to generate tests; P4Testgen provides these semantics as part of its tool.

`p4pktgen` [160] is a symbolic executor that automatically generates tests. It focuses on the `v1model`, STF tests, and BMv2. In spirit, `p4pktgen` is close in functionality to P4Testgen. However, the tool does not implement all aspects of the P4 language and `v1model` architecture—its capabilities as a test oracle are limited. We tried to reproduce the bugs listed in Table 5.8 using `p4pktgen` but were unable to. `p4pktgen` either was unable to produce tests for the program or did not achieve the necessary coverage. While `p4pktgen` does support a form of packet-sizing to trigger parser exceptions, its model only considers a simple parser-control setup, not multiple subsequent parsers such as Tofino's.

SwitchV [9] uses differential testing to find bugs in switch software. It automatically derives inputs from a switch specification in P4, feeds the derived inputs into both the switch and a software reference model, and compares the outputs. SwitchV uses fuzzing and symbolic execution to generate inputs that cover a wide range of execution paths. To limit the range of possible inputs, the tool relies on pre-defined table rules and the P4Constraints framework. It also does not generate control-plane entries. Like `p4pktgen`, SwitchV is specialized to `v1model` and BMv2.

Meissa [264] is a symbolic executor specialized to the Tofino target. Meissa builds on the LPI language's pre- and post-conditions [241] to generate input–output tests. The tool is designed for scalability and uses techniques such as fixed match-action table rules, code sum-

maries for multi-pipeline programs, and path pruning to eliminate invalid paths according to the input specification. P4Testgen's preconditions and path selection strategies combat the same scaling issues as Meissa. Meissa's source code is proprietary, which precludes a direct comparison.

PTA [29] and DBVal [129] both implement a target-independent test framework designed to uncover bugs in the P4 toolchain. Both PTA and DBVal augment the P4 program under test with extra assertions to validate the correct execution of the pipeline at runtime. Both projects provide only limited support for test case generation.

FP4 [257] is a target-independent fuzzing tool that uses a second switch as a fuzzer to test the implementation of a P4 program. FP4 automatically generates the necessary table rules and input packets to cover program paths. To validate whether outputs are correct, FP4 requires custom annotations instrumented by the user.

Coverage. There are important differences in how testing tools assess coverage—see Table 5.10 for a summary. P4Testgen marks a node in the source P4 program as covered when the symbolic executor steps through that node and generates a test. FP4 measures action coverage by marking bits in the test packet header to track which actions were executed. As FP4 generates packets at line rate, it achieves coverage for actions faster than P4Testgen. `p4pktgen` discusses branch coverage, which can be estimated by parsing generated tests to see which control-plane constructs (tables, actions) were executed. Meissa reports coverage based on the branches of its own formal model of the P4 program. SwitchV also measures branch coverage based on developer-provided goals derived from its symbolic model. Another important consideration is whether programmers can annotate the program with constraints or preconditions—see Fig 5.9. In many scenarios, these constraints are necessary to model assumptions made by the overall system, but they also affect coverage since they reduce the number of legal paths.

Extensibility. Petr4 [57] and Gauntlet [196] are designed to support multiple P4 targets. Petr4 provides a "plugin" model that allows the addition of target-specific semantics. However, it does not support automatic test case generation and does not aim to provide path coverage. Gauntlet can generate input–output tests for multiple P4 targets but it does not model externs, nor does it implement whole-program semantics to model the tested target.

5.9 Summary

With P4Testgen, we have built a P4 test oracle that automatically generates input–output tests for arbitrary P4 targets. P4Testgen's success recipe is its extensible execution model. This execution model is implemented using whole-program semantics to model the behavior of the P4 program and combined with taint-tracking, concolic execution, and path selection strategies to generate tests that achieve coverage. P4Testgen is intended to be a resource for the entire P4 community. It supports input–output test generation for various open-source P4 targets and several extensions for closed-source targets exist. By designing it as a target-independent, extensible platform, we hope that P4Testgen will be well-positioned for long-term success. Moreover, since P4Testgen is a back end of P4C, it should be easy for developers to build on our tool, lowering the barrier of adoption.

P4Testgen already receives contributions from the broader community to improve and extend its functionality. For example, two common community requests are to extend P4Testgen with the ability (i) to generate arbitrarily many entries per table and (ii) produce tests with a state-preserving sequence of input–output packets. In the future, to further validate P4Testgen's generality, we would like to complete P4Testgen extensions for the P4-DPDK SoftNIC target and the open-source PSA [234] target for NIKSS [164], as well as proprietary SmartNICs [107, 87, 124]. We also intend to develop additional P4 validation tools based on P4Testgen's framework that apply ideas from software testing in the network-

ing domain—e.g., random program generation, mutation testing, and incremental testing. We are also interested in network-specific coverage notions—e.g., for parsers, tables, actions, etc.

Software testing is always important, but testing the packet processing programs that power our network infrastructure, processing billions of packets per second, is especially important. In time, there will inevitably be better approaches than our P4-based, extensible execution model for generating high-quality tests for packet processing systems. The P4Testgen framework can serve as a vehicle for prototyping these approaches, and for integrating them into the P4 ecosystem. Inspired by efforts from other communities [18, 125], we envision having an open benchmark suite of standard test programs, control plane configurations, and various notions of coverage to standardize comparisons between different testing approaches—enabling more rapid progress for the whole networking community.

Part III

Optimizing Network Device Stacks

6 | Flay: Incremental Specialization of Data-Plane Programs

So far, we have focused on *testing* network-device stacks; however, we can also *optimize* these stacks using our SMT-based model. More specifically, we can introduce better compiler optimizations. We demonstrate this with our next tool, Flay. Flay embodies a concept more than a specific tool: an approach to specialize a data-plane program using additional information, such as the current control-plane or environmental configuration. What distinguishes our type of specialization from previous approaches is its focus on handling updates to the configuration used for specialization. This is particularly important in a networking environment where a network device can receive control-plane updates on the order of seconds. Our approach to handling configuration changes is *incrementality*. Incrementality involves a three-step process for each update: (1) identify the components influenced by this update, (2) check whether the update changes the semantics of the component, and, if the semantics changed, (3) recompile only this component. Using this approach, we can minimize the impact of each configuration update and avoid recompilation and changes to the underlying data-plane program as long as possible.

We have not only designed this approach but also developed a prototype implementation,

Flay, which leverages our execution model. By combining P4Testgen's model for the control-plane interface with specific control-plane semantics, we can use an active control-plane configuration to perform advanced program optimizations, including program specialization. We could remove unused data-plane code (e.g., table lookups or parser paths), inline code, or substitute better matching algorithms for a particular control-plane configuration. However, P4Testgen's execution model, as is, cannot be used directly for these types of optimizations. P4Testgen was initially designed to generate only a single control-plane entry for each potential branch point. Hence, it has incomplete semantics for a control-plane configuration, i.e., it does not model how a device behaves when many control-plane entries are installed. It also lacks a mechanism to relate a given control-plane configuration to its data-plane execution model. Furthermore, control-plane configurations may change, and our current semantics have no means to handle these changes gracefully.

In this chapter, we first introduce the motivation for incremental specialization, show several use cases, and then demonstrate how we can extend our execution model to support this type of specialization and implement a form of incrementality.

6.1 Introduction

Packet-processing programs on network devices (SmartNICs, switches, networking stacks) must rapidly process packets with limited resources (e.g., tables, ALUs, cores, CPU cycles), while simultaneously supporting many different features (e.g., ACLs, routing, NATs). Compilers for packet-processing languages [100, 34, 137, 217, 86, 253, 146] play an important role in determining the final resource requirements and performance of such programs. Typically, compilers translate the packet-processing program into an implementation when it is first authored, leaving the implementation unchanged over the program's lifetime.

This "one-and-done" approach misses many opportunities to improve the implementation

over a program's lifetime. In addition to the program's source code, the program's resource usage is also determined by control-plane configurations (e.g., ACL or forwarding rules). For instance, if an ACL table is empty, it can be removed, making room for additional features. Such *specializations* are especially beneficial for "kitchen-sink" programs that capture the union of all possible features [215, 227], where only a subset of features is active at any time. Prior projects have leveraged this observation: they treat control-plane configurations as constant inputs to a packet-processing program and introduce a specializing compiler to further optimize the implementation of the program before it is run [2, 154, 56].

In reality, however, control-plane configurations are *pseudo-constant*: many parts of the control-plane only change in response to policy changes, maintenance, or failures (Fig. 6.1) and are thus infrequent. Other parts, however, change frequently (e.g., IP routes, NATs). Control-plane updates can also occur in bursts, with changes happening at once quickly followed by a long quiescence [102]. Given this pattern, our core claim is that any specializing compiler must be able to respecialize a program quickly when control-plane constants change. Moreover, because recompiling network programs is expensive and many control-plane updates do not affect the program implementation, the runtime must decide when respecialization is actually needed. Hence, to be effective, such compilers must be (1) *control-plane-triggered* so that they continuously respecialize program implementations in response to control-plane changes and (2) *incremental*, to perform as little processing as possible on program sources and control-plane configurations for each update.

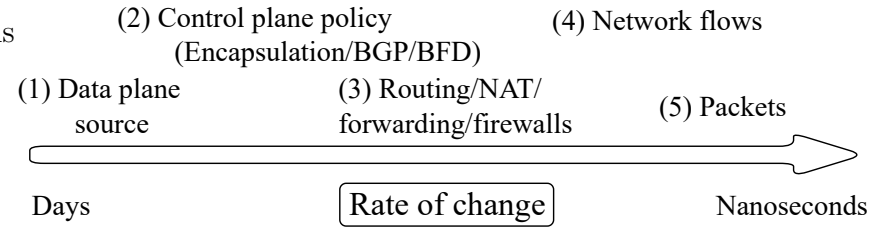


Figure 6.1: Varying rate of change of network program input.

We describe a design sketch for such an incremental compiler, operating as a shim layer

between the network controller and the data plane (§6.2). We also describe several tangible benefits enabled by this approach, such as saving hardware resources, and optimizing the memory footprint and performance of packet classifiers. To demonstrate that our call for an incremental specializing compiler is feasible, we build a prototype, Flay. Flay is a partial evaluator [114] that combines several techniques (dead-code elimination, constant propagation, table inlining) to specialize P4 programs. Flay leverages the fact that P4 is a restricted domain-specific language (DSL) with a few core primitives (e.g., tables, control programs) to construct SMT formulae that can quickly identify when recompilation is necessary. This allows Flay to process a control-plane update within ~100 milliseconds and avoid recompilation for all control-plane updates that do not require it. By treating control variables as pseudo-constants, Flay can also save pipeline resources for Tofino programs.

In contrast to Gauntlet and P4Testgen, we see Flay less as a concrete tool and instead as a vehicle to explore a broader research agenda. In this chapter, we outline several concrete avenues for future work. First, for the control-plane updates that do trigger respecialization, we plan to use Flay as a vehicle to explore the tradeoff between recompilation time and specialization quality. Second, during respecialization, we are still bottlenecked by existing device compilers that monolithically compile the entire program, causing much longer compile times than necessary. We can push incremental specialization much further by (1) rearchitecting device compilers to also operate incrementally, e.g., by only recompiling the tables in the program that actually changed, and (2) through hardware support for partial configuration.

6.2 Control-Plane-Driven Specialization

Our goal. We want to develop a compiler that can specialize network programs given a control-plane configuration. This compiler must be *incremental* with respect to the con-

trol plane: The compiler must support automatic respecialization whenever control-plane configurations change, without incurring substantial time on every update—given that most updates do not affect program implementation. We do not consider traffic profiles when specializing because traffic may change more rapidly than control-plane configurations [137].

Program	switch [215]	scion [55]	Beaucoup [44]	ACCTurbo [10]	DTA [131]
Time	106 s	38 s	22 s	28 s	25 s

Figure 6.2: bf-p4c [25] compile times for Tofino $P4_{16}$ programs.

WHY AN INCREMENTAL, SPECIALIZING COMPILER? Even though control-plane updates occur less frequently than packet arrivals in the data plane, they do change from time to time, often in response to external events like routing changes, and often in bursts. At the same time, most control-plane updates do not require recompilation of the specialized program because they do not change program semantics. Existing specializing tools such as Morpheus [154], Pipeleon [251], or ESwitch [155] approach this problem by either introducing resource-consuming fall-back datapaths or recompiling the data-plane program every time the control plane issues an update. When control-plane updates arrive in bursts of hundreds of rules in a few seconds [110, 112, 104, 93], recompiling a network program from scratch, which can take several tens of seconds (Tbl. 6.2), is too slow for a specializing compiler to keep up. Even more recent incremental recompilation approaches require on the order of seconds to complete recompilation [72, 51, 174, 250]. A specializing compiler that is unable to quickly distinguish between a trivial update that does not need recompilation (e.g., adding a new NAT entry) and a major data-plane change (e.g., enabling an IPv6 ACL table) will be stuck constantly respecializing.

OUR INSIGHT. In any network program, we can distinguish runtime-dependent variables into two types: the *data-plane variable*, which depends on data-plane input (e.g., variables

parsed from a packet header), and the *control-plane variable*, which depends on control-plane input (e.g., an ACL entry which decides whether a packet is forwarded or dropped). For example, in P4, data-plane variables are sourced from the packet through parser extraction calls, whereas control-plane variables are stored in tables and stateful registers. In eBPF, on the other hand, data-plane variables are sourced from reads of the packet metadata structure (e.g., `sk_buff`), and control-plane variables are stored in maps (e.g., BPF maps). An incoming packet results in a concrete assignment to the data-plane variables in the program. Similarly, a control-plane update results in an assignment to a subset of control-plane variables.

Any control-plane update can be directly mapped to a component in the data plane (e.g., a table, register, or map). We can use this mapping to implement an incremental compiler. Not every control-plane update introduces a semantic change. Many control-plane entries just increase the likelihood for an already existing data-plane program path to be taken. This allows us to implement a form of taint tracking, which lets us quickly identify the affected components. Restrictions in networking DSLs, such as a lack of pointer-based indirection, unbounded loops, or jumps, make taint tracking tractable. With a taint-tracking system in place, we only need to check whether a particular component's behavior has changed given an update. The way we compute these behavioral semantics, identify affected components, and quickly check whether a change is necessary depends on the particular incremental specialization technique we use. We show a concrete example in §6.4.

A Control-Plane-Triggered Compiler. Fig. 6.3 shows a sketch of our proposed approach. (1) The control-plane-triggered compiler is intended to be invoked on every control-plane update and provides feedback on whether a control-plane update requires recompilation. (2) Once a new update is sent to the compiler, it identifies the affected program components based on the control-plane variables "tainted" by the control-plane update. (3)

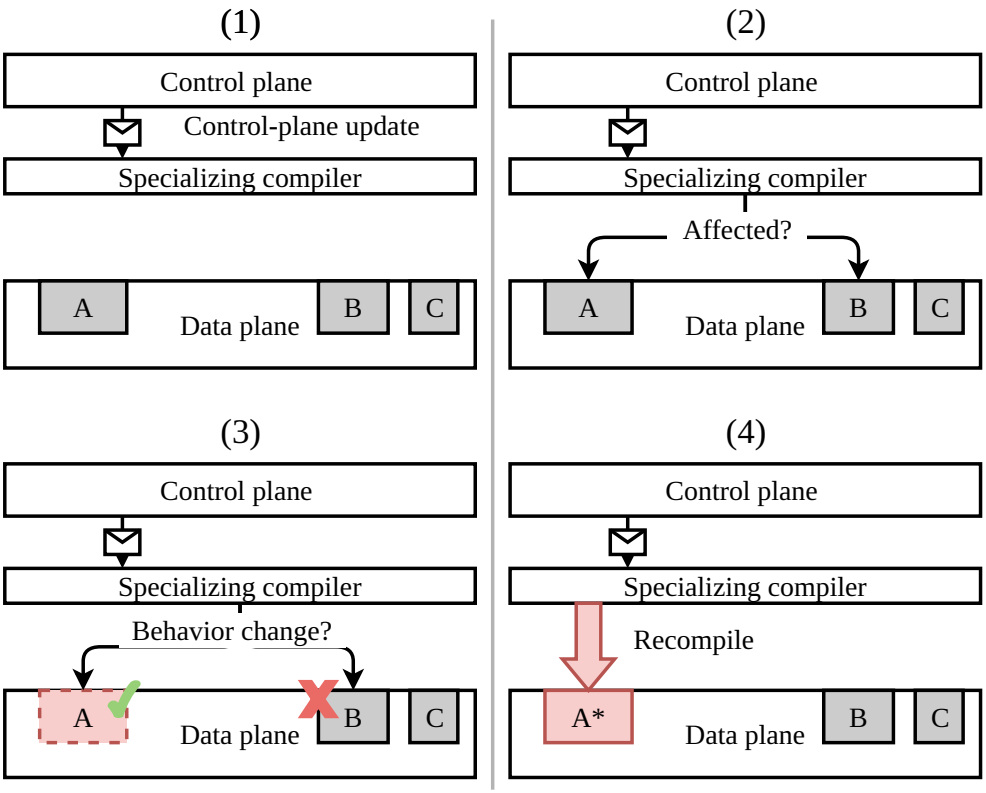


Figure 6.3: Control-plane-triggered, incremental specialization. Letters describe objects configurable by the control plane.

After identifying the affected components, the compiler checks whether the semantics of those data-plane components change. (4) For components that do not need changes, the compiler will forward the update to the device. If the compiler's query indicates that the behavior of a component in the data-plane program will change, the compiler needs to recompile that component before the control-plane update can be installed onto the device. This recompilation (if needed) is done by the device-specific compiler.

6.3 Specialization Use Cases

Control-plane-triggered specialization as described in §6.2 can improve resource usage across different network devices. We outline several kinds of specialization use cases.

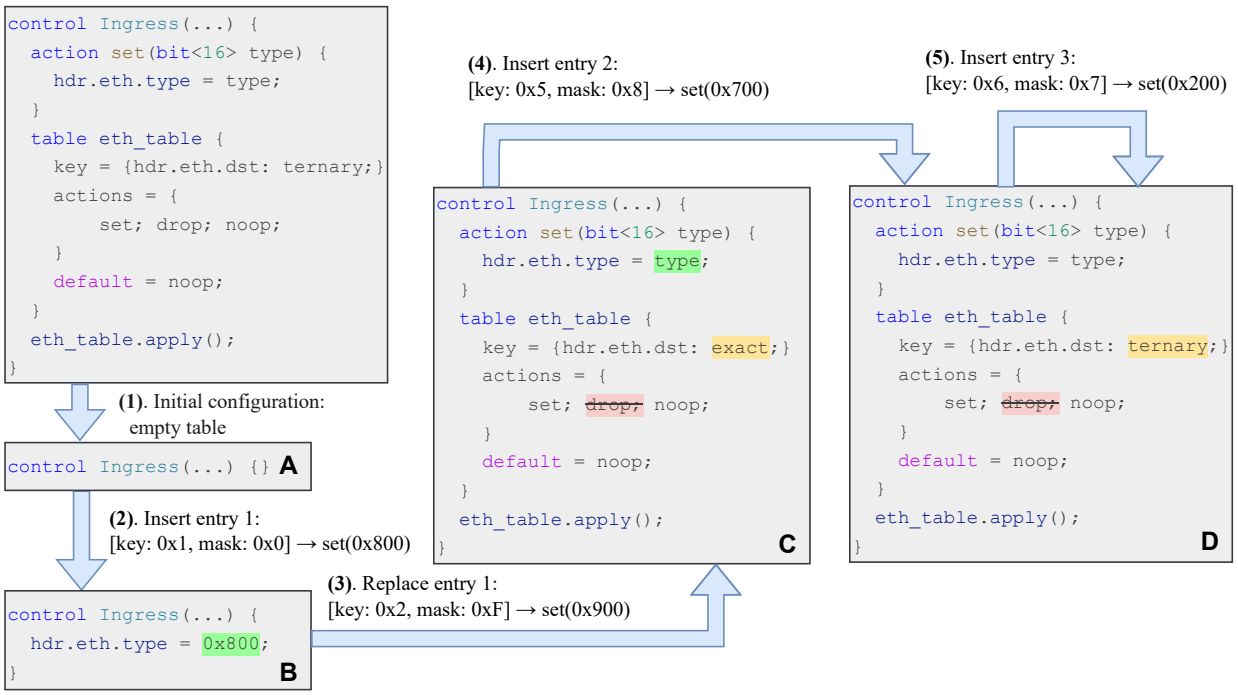


Figure 6.4: For the program on the left, we show control-plane updates 1–5 and their effect on data path implementation.

Resource savings over a program's lifetime. On RMT-style pipelines with hard constraints on the number of computation units, tables, and stateful memories, we can substantially save on hardware resources by specializing to control-plane configurations. As an example, Fig. 6.4 describes how the implementation of a single P4 table can change in response to different control-plane updates. Initially, the table is empty and can be removed entirely (impl. A). We then insert a single entry, a ternary match with a 0 mask that executes `set(0x800)` as its action. Here, we can inline the table action and save the cost of a table lookup. We then replace the existing entry with a ternary match that uses the full mask (impl. B). This is effectively an exact match entry. Because there is no other entry in the table, we can change the match type of the key, saving Ternary Content Addressable Memory (TCAM) resources. Once we insert entry 2, the table must be implemented as a ternary table (impl. C). The last entry (3) does not change the behavior of the table, and hence no recompilation is needed. Note that, in both implementations C and D, the unused

`drop` action is removed from the table, freeing up computation units.

Parser specializations. Several specializations are also possible for parsers in network programs. The `parse_break` command in NPL [30] temporarily suspends the parser to perform table lookups. If the accessed table is empty, we can remove entire parse branches that depend on this particular lookup. P4 Parser value sets (PVS) [236, §12.11] serve a similar function. We can free the TCAMs and SRAMs associated with a PVS that is not configured. Another network-program-specific specialization is parser-tail pruning. Once we have specialized the program, we can check whether the parser itself is doing unnecessary work. Any header at the tail of the parser that is not accessed in the program could be classified as payload. Reducing the number of parse calls can reduce PHV usage in Tofino or improve packet-processing latency in OvS [166].

Savings in other hardware resources. One, the Tofino programmable switch supports the use of action profiles to support actions (e.g., setting a packet's output port metadata) that are shared among tables. If an action profile is empty, an incremental compiler can specialize the implementation of all tables associated with this action profile. Two, we can specialize device-specific functions. Consider a hardware unit that computes a checksum on a set of headers. Further, let us assume that one of these headers H is set as part of some table action A in table T—as opposed to being parsed out of an incoming packet. If there is no control-plane entry for T that uses A as its action, we know that H is invalid, and hence the checksum will also be invalid, allowing us to directly compute the checksum result, saving a checksum unit. Third, if a header is only written by one action and this particular action does not exist in the control-plane configuration, we can simply remove the header in the RMT pipeline to free up packet-header-vector (PHV) resources.

Specializing packet classification. We can specialize data structures used in the data plane to classify packets based on the actual patterns present in the active control-plane configuration. Often, these techniques involve choosing a less expensive data structure for the given network device. For example, a common, but expensive data structure to classify packets is the TCAM. TCAMs allow matching on header fields based on bitmasks. If we can tell from the current control-plane configuration that only few or no masks at all are necessary, we can replace the TCAM with a simpler matching data structure, e.g., a Semi-TCAM (STCAM) in AMD devices [5]. ESwitch [155] and Morpheus [154] have shown how we can apply similar specializations to software packet-processing devices, such as Open vSwitch (OVS) [170] and eBPF, respectively. NeuroCuts [138] and NuevoMatch [177] train neural networks for more efficient packet classification by mapping a control-plane configuration to an efficient lookup data structure.

How incremental compilation could help. In all of the use cases above, knowledge of the currently active control-plane configurations can help a compiler specialize the underlying implementation of the data-plane program. Further, if we had an incremental compiler [205], it could localize the compiler's effort to specific aspects of the data-plane program. For instance, in Fig. 6.4, all of the control-plane updates (and hence all of the specializations) pertain to the implementation of a single table, allowing the incremental compiler to ignore the rest of the data-plane program. Similarly, if parser compilation were treated independently of the rest of the program, we could specialize the parser separately in response to which headers are accessed by control-plane entries. Finally, in the context of packet classification, the control-plane update tells us which specific table's implementation to focus on, permitting us to specialize that alone.

6.4 A Model For Efficient And Incremental Data-Plane Specialization

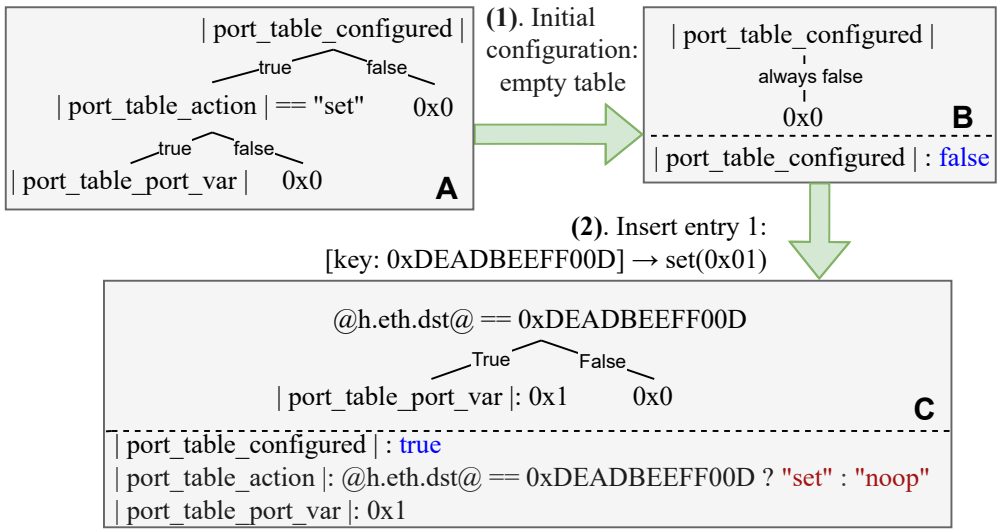
As a preliminary feasibility study, we built Flay. Flay implements incremental partial evaluation for P4 programs. Partial evaluation [114] is a program optimization technique which specializes a program by treating some inputs as constants. Flay specializes P4 programs subject to their control-plane configuration by continuously reoptimizing the running P4 program based on incoming control-plane updates. We picked partial evaluation because, simply by eliminating newly dead code and inlining constants based on the current control-plane configurations, we can already implement many of the resource-saving specializations discussed in §6.3. Flay supports P4 program specialization for various targets (BMv2 [232], Tofino [25], or Xilinx Versal [126]). We also believe that Flay can generalize to packet-processing environments such as restricted C for eBPF [62], NPL [30], or microcode [255]. Flay is available at `https://github.com/nyu-systems/flay`.

6.4.1 The Execution Model For The Control-Plane Interface

We extend P4Testgen's whole-program semantics to the interface between the data and control planes, model the semantics of a control-plane configuration, and determine how this configuration influences program behavior. Flay implements incremental specialization by representing a network program as a combination of data-plane expressions and control-plane assignments. We can ask specialization queries by substituting the control-plane assignments into placeholders within the data-plane expressions. Fig. 6.5 shows the high-level workflow of Flay.

```
1 control Ingress(...) {
2   action set(bit<9> port_var) {
3     egress_port = port_var;
4   }
5   table port_table {
6     key = {h.eth.dst: exact;}
7     actions = {set; noop;}
8   }
9   apply {
10    egress_port = 0;
11    port_table.apply();
12    h.eth.dst = egress_port == 0 ? 0xAAAAAAAAAAAA : 0xBBBBBBBBBBBB
13  }
14 }
15 # Symbolic value of egress_port variable after executing a line:
16 # Line 9: @egress_port@
17 # Line 10: 0
18 # Line 11: |port_table_configured| && |port_table_action| == "set" ?
19 #          |port_table_var| : 0
20 # Line 12: *unchanged*
```

(a) P4 program setting a port variable based on a table entry.



(b) Value of `egress_port` at line 12 after each entry update.

Figure 6.6: Flay's representation of `egress_port`. |x| denotes a control-plane symbol; @x@ a data-plane symbol. Entries below the dotted line are the active control-plane assignments.

Data-plane expressions. Flay differs in its approach from P4Testgen, which traverses each program path independently to generate tests. For Flay, we use a simple data-flow analysis coupled with state-merging [16, §5.6] to generate data-plane expressions. For any input program, Flay first computes the data-plane semantics of the program and annotates program points of interest (e.g., if-statements, match-action table execution, map lookups, or variable assignments) with a data-plane expression. The control-plane variables within the expressions act as placeholders and are later substituted with control-plane assignments. Data-plane variables can assume any value since we do not specialize based on traffic profiles. Our state-merging approach makes any program point annotation *hermetic*, i.e., we can evaluate queries on each annotated program point independently. Lines 15–20 in Fig. 6.6a demonstrate how we use state-merging to annotate each program line with a snapshot of the value of `egress_port`. If the table does not have a control-plane entry, `port_table_configured` is `false` and `egress_port` will evaluate to `0`. Hence, we can simplify the assignment on line 12 to `h.eth.dst = 0xAAAAAAAAAAAA`. The type of specialization we use influences the number of program points (and hence the work required on each control-plane update). For dead-code elimination we may just want to annotate if-statements, but for constant substitution we may need to annotate any variable read.

Control-plane assignments. We represent control-plane entries as a set of control-plane variable assignments. This representation implements the semantics of the control plane as described by the appropriate specification (e.g., P4Runtime). For example, entries that are duplicates or eclipsed by higher-priority entries (and thus have no effect) are omitted in the set of control-plane assignments. To infer the initial assignment set for any configurable data-plane element we consult the device specification. Flay maintains a map, which associates a control-plane variable with the set of program points it can influence. On each control-plane update, Flay retrieves all the affected program points from this map-

ping. For any affected program point, Flay substitutes the control-plane assignments into the expressions associated with the point.

Specialization queries. Once Flay has combined the gathered data-plane expressions with the initial control-plane assignments, it specializes the program by asking queries on the joint representation. Typically, the query indicates that the value of the expression has not changed, and Flay will forward the update directly to the network device without triggering recompilation. If any program point indicates a change in behavior, Flay must trigger the reoptimization process for the affected data-plane components. We currently ask two types of queries using Flay: 1) Is this particular piece of code executable? and 2) Can we replace this program variable with a constant? Concretely, we remove unnecessary table dependencies by deleting unused actions, inline P4 tables which always execute the same action, simplify extern calls, and replace variables and conditions with constants. Flay then passes the specialized program to the device-specific compiler, which optimizes it further. We evaluate some of the benefits of these incremental specializations in §6.4.2.

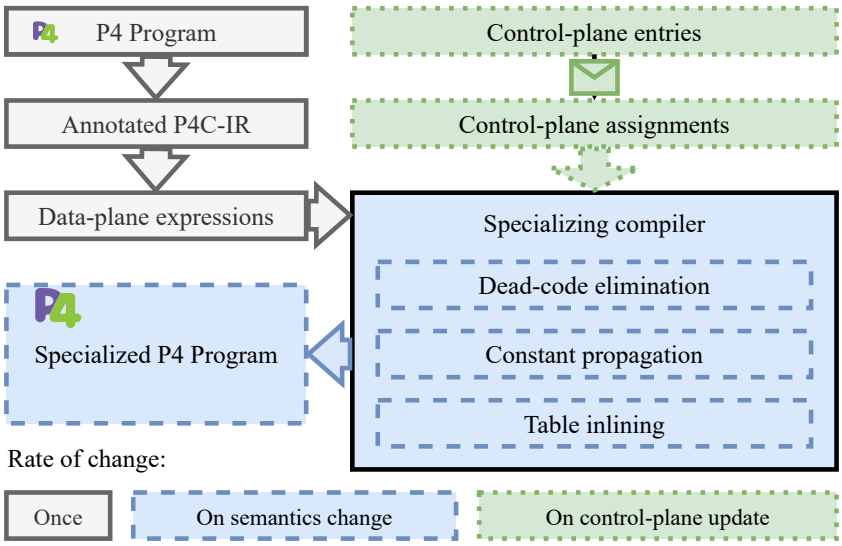


Figure 6.5: Flay's design.

An example. Fig. 6.6b shows how Flay uses a constant propagation query to compute the value of `egress_port` at line 12. The data-plane model in Block `A` represents all the possible values `egress_port` can assume at this line. After obtaining this general representation, we

specialize it using the initial control-plane assignment (Block `B`). The control-plane specification for this device prescribes that an empty table executes the default action, which does nothing here. Hence, the assignment sets `port_table_configured` to `false`, which causes `egress_port` to be 0. After receiving an update, we can match on a key field and execute the `set(0x01)` action, which sets `egress_port` to 1 (Block `C`). There are now two possible outcomes for the value of `egress_port`: 0 or 1.

Processing updates quickly. Since, once computed, data-plane expressions do not change, Flay performs extensive preprocessing on expressions to quickly compute queries after control-plane updates. Preprocessing increases initial analysis time but greatly reduces query time. (1) Flay reduces the expression complexity by applying constant folding, common subexpression elimination, and strength reduction. (2) Flay converts each data-plane expression into a representation that supports fast incremental checking specialized towards the particular query. For example, for efficient expression substitution we use Z3's [53] e-matching [52] implementation. Instead of e-matching, we could also use an incremental Datalog evaluation API such as Souffle [202].

Currently, Flay does not support incrementality well in scenarios where tables with complex match keys have many control-plane entries. We show an example of Flay's performance degradation in such scenarios in §6.4.2. The cause is an inefficient control-plane representation. Since we model the potential matches of an incoming key against all table entries as a single and deeply nested expression, complex keys coupled with large tables can produce a very large expression. Substituting such a complex key expression into a data-plane annotation and checking whether the annotation resolves to a constant can be slow. To make reasonably fast decisions we compromise on Flay's sensitivity. Once a certain threshold of entries (e.g., 100) has been reached, we overapproximate: we assume the entries in the table cover all its possible actions and action parameters. For example, in Fig 6.6b, overap-

P4 Program	Program statements	Compile time	Data-plane analysis time	Update analysis time
scion[55]	582	38s	2s	90ms
switch [215]	786	106s	9s	90ms
middleblock [9]	346	2s	0.6s	5ms
dash [223]	509	2s	1.5s	12ms

Table 6.1: Flay evaluation times for P4 programs. Compilation is from scratch. Flay's data-plane analysis step runs once and skips the parser. At runtime, Flay only runs update analysis.

proximation would assign `*any*` to `port_table_action` and `port_table_port_var`, which would cause the computed value of `egress_port` to revert to the model shown in Block `A`. In practice, crossing the threshold rarely requires respecialization because tables with many entries likely cover most of their possible paths already. We discuss ideas to improve the performance of our control-plane representation in Sec. 6.6.

6.4.2 Evaluating Flay

We evaluate how Flay specializes the SCION [55] border router P4 programs written for the Tofino 2 [7] switch. We chose the SCION programs for evaluation because, besides being moderately complex (~1700 LoC), they are supplied with representative control-plane configurations. We use this program to answer questions on specialization, incrementality support, and analysis time.

Can specialization save resources? First, we compile the SCION program without applying Flay's specialization. The program requires the maximum number of Tofino 2 stages. We then specialize the SCION program using the supplied configuration. This configuration does not use IPv6, and all the IPv6 program paths are unused. After removing these unused paths, the program requires 20% fewer stages.

What is the cost of initial data-plane analysis? Our state-merging data-plane analysis is sensitive to programs with many control-flow statements [150]. The initial pass

through the program is cheap, but the generated data-plane expressions can become deeply nested. Preprocessing expressions for incrementality support can quickly become slow. To accelerate processing for large programs (e.g., `switch.p4`) we added an option to skip parser analysis. Since Flay's specializations focus on constructs in the control block, skipping the parser has little impact on their effectiveness. We evaluate Flay's complexity on a suite of sample programs. The increase is exponential in terms of the number of control paths. Nevertheless, even for large, complex programs, we can complete our initial analysis within a few minutes (Tbl. 6.1).

What influences Flay's update processing speed? We use a fuzzer [198] to generate 1000 unique IPv4 entries and insert the entries into the SCION IPv4 forwarding table to test how Flay handles a burst of semantics-preserving updates. Flay can determine within a second that the batch of updates does not require program recompilation. We then send a batch of updates that enables the previously unused IPv6 paths in the SCION program. Flay determines respecialization is necessary and triggers the recompilation process. After recompilation, the SCION program requires the maximum number of stages again because all program paths are used. Tbl. 6.1 shows that Flay is not very sensitive to program complexity. While the time required to process updates increases with program complexity, it generally stays below 100 ms.

Conversely, as discussed in §6.4, Flay slows down when a complex table has many entries. An example of such a table is the Pre-Ingress ACL table of Google's Middleblock P4 switch model [9]. To characterize the slowdown, we initialize this ACL table with varying numbers of entries, then send a single update and measure how much time Flay requires to make a decision. Tbl. 6.2 shows the results. The precise update implementation, which evaluates all entries, already takes 100 ms at only 100 installed entries. Once we overapproximate the entries, update processing time becomes low again.

Total updates installed	Analysis time for 1 incoming update	
	Precise	Overapproximated (>100 entries)
1	~1ms	–
10	~5ms	–
100	~100ms	~1ms
1000	~4000ms	~1ms
10000	~265319ms	~1ms

Table 6.2: Influence of installed updates on Flay's update processing times for middleblock.p4 [9].

6.5 Related Work

Incremental computation [175] is a mature field with a wide range of applications [114, 115, 49, 207, 210]. Recently, work on JIT compilers [249] and feedback-directed optimization [40] articulated the need for incremental specialization.

For network programs, many specialization frameworks use either packet traces or control-plane configurations as input. We classify these into frameworks which specialize network programs once before deployment (offline) and frameworks that continuously specialize the program (online).

Offline-specialization tools. In the context of P4, P5 [2] proposes control-plane-based optimization to simplify dependencies between P4 tables. P2GO [248] is a profile-guided specialization tool where a profile is the combination of a packet trace and the expected control-plane configuration. Parasol [99] uses traffic profiles to generate data structures optimized to that profile. NFReducer [56] and PacketMill [67] specialize network function chains by applying a series of framework (e.g., ClickNF [128]) optimizations based on initial control-plane configurations. mSwitch [103] inlines switching rules within the VALE [183] software switch. Relative to these tools, our approach is to specialize *continuously*, in response to every control-plane update.

Online-specialization tools. Bhatia et al. [22] specialize the Linux network stack by inlining installed IPv4 routes and bridging route changes using a NAT until respecialization has completed. ESwitch [155] and Hoda [166] continuously specialize OvS. ESwitch optimizes OvS by changing packet-matching templates based on user-supplied traffic and flow entry patterns. Hoda instead produces a new, specialized parser and megaflow cache from existing cache rules. Pipeleon is a profile-guided specialization framework targeting P4 SmartNICs [251]. Morpheus [154] performs profile-guided optimization for eBPF. To deal with input profile changes, these tools respecialize on each control-plane update or periodically trigger recompilation. Our approach can defer recompilation until program semantics change.

6.6 Discussion

We have argued for control-plane-triggered and incremental compilation as a new way of thinking about compilers for packet processing. As a proof of concept for our research agenda, we have implemented Flay, a partial evaluation framework for the P4 language. We extend the execution model implemented for P4Testgen with explicit semantics of control-plane rules. Our initial results are encouraging. We are able to show that we can reduce resource usage in data-plane programs and quickly react to configuration changes within milliseconds. We conclude this chapter with possible future directions specific to Flay as a partial evaluator.

Incremental compilation for data-plane programming. While we attempt to avoid recompilation for as many control-plane updates as possible, we must eventually recompile when an update triggers a semantic change. In such cases, recompilation times should ideally be low. Currently, however, we rely on device-specific compilers that treat

the entire program as a monolithic unit to be compiled from scratch. Recent work on modularity in network programming languages [219, 245, 68] and hardware support for partial reconfiguration [250, 245, 258] points toward recompiling only the modules (such as specific tables) that have changed.

A control-plane representation for faster checks. Our current representation can significantly slow update processing when the targeted table matches on a complex key and already contains many entries. Constructing the key expression, substituting it into the appropriate placeholders, and checking whether the expression simplifies to a constant does not scale well. Ideally, our representation should be compact enough to enable efficient substitution and expression rewriting. This representation differs from the packet-classification problem [176], where the goal is to quickly determine whether a given set of values matches another set. Instead, we focus on developing a symbolic representation that can be evaluated quickly. Prior techniques for efficiently representing control-plane configurations, such as Atomic Predicates [254], ERA [69], or Flash [93], could inform the development of a compact control-plane representation to speed up update processing with complex configurations.

Developing a similarity metric for control-plane configurations. Flay specializes P4 programs based on control-plane configurations and handles updates to these configurations. This requires checking every update for potential respecialization, which introduces overhead—particularly with frequent updates—and risks thrashing (i.e., cycling between different versions of a specialized program). Defining classes of control-plane configurations based on a similarity metric could mitigate this issue. A program specialized for a given class would remain valid for all configurations within that class, eliminating unnecessary checks or spurious respecialization. An operator could then deliberately switch between these predefined classes (e.g., representing specific routing update patterns, BGP policies, or

device setups) to minimize disruptions to packet processing. The Tofino switch already provides program profiles specialized for different configurations. Concepts from configuration generation [225, 149] and device profiles could help define the structure of these configuration classes.

Extending Flay to more targets. We have primarily validated Flay on the Tofino switch. Tofino uses an all-or-nothing compilation approach, guaranteeing line-rate packet processing for any program that compiles. With Flay, we can mostly reduce program size but cannot significantly affect packet-processing performance. This may differ for other targets. For example, the programmable X2 switch [252] or SmartNICs [126, 107, 87] do not use this all-or-nothing compilation approach. Further, packet-processing software frameworks such as eBPF/XDP [100] or DPDK-based systems [230, 97, 170, 84] are heavily influenced by compiler optimizations. We plan to extend our ideas to these programmable network ecosystems.

Exploring the tradeoff between specialization time and specialization quality. We have implemented three specialization rewrite rules for Flay: dead-code elimination, table inlining, and constant propagation. As we add more rewrite rules, recompilation may take longer, which could become problematic when rapid responses to control-plane updates are required. Specialization rewrite rules are beneficial, but some may be comprehensive yet slow. We plan to use Flay to explore tradeoffs between specialization time and specialization quality, with the goal of making incremental specialization a practical technique for packet-processing compilers.

Part IV

Conclusion

7 | Limitations and Future Directions

We describe the limitations of our particular SMT-based execution model, possible future work, and conclude with some lessons we have learned.

7.1 Limitations

We have addressed the limitations specific to each of the tools in the appropriate chapter. There are also limitations common to all tools.

Lack of formal semantics for the execution model. We motivated our work by using the execution model of the P4 packet-processing language to develop analysis tools. All three projects use SMT to encode their execution model. However, we do not provide denotational semantics for this model. Developing denotational semantics would allow us to describe packet processing independently of the framework we are using. Instead, our interpretation of the P4 language is currently only available in the form of C++ code written as part of the P4 compiler. From a practical viewpoint, our tools can be considered simulators of device behavior, using P4 code and control-plane configurations as their input. Several other projects have formulated denotational semantics for the P4 language specifically [121, 57, 12, 169, 244], and so we consider formulating our own semantics out of scope.

Our execution model is not designed to be stateful. All our work focuses on a single packet traversing a network device in a specific state. We do not model statefulness, i.e., what happens when a particular pipeline receives multiple packets in succession. This can make it difficult to test some properties; for example, we cannot check whether a counter tracks packets correctly or whether a rate-limiting algorithm actually throttles traffic. In P4Testgen, we have worked around this limitation by assuming that any device we test can be set to a testable state. This is not guaranteed. In contrast, tools such as FP4 [257] are able to test more device states by generating packets at line rate, which ultimately cover many more possible counter or rate-limiter configurations.

Our execution model does not capture concurrent access. We are not modeling concurrent accesses to particular memory resources. For example, it is possible that when memory resources are shared between packets, a compiler bug can cause write conflicts, which in turn leads to incorrect packet-processing outcomes. Our model is unable to catch such behavior because we assume all memories are accessed in isolation. This problem becomes important as more SmartNICs and switching chips are introduced which may use concurrent processing primitives such as asynchronous execution and barriers. The P4 specification already describes a thread model and several primitives for parallel execution [236, §17.4.1.]. We would need to extend our execution model to be able to detect bugs in this kind of behavior.

We lack validation on complex network-device stacks. Our execution model can be used to test network-device stacks, but we only corroborate this claim with P4Testgen. Specifically for P4Testgen, we only tested software for the Tofino hardware switch [25], P4-for-eBPF [242], and BMv2 [232]. To ensure that our thesis holds true, we should also apply our testing techniques to complex real-world network device stacks, such as PINS [228] or SONiC [229], and validate the effectiveness of our model by finding bugs there. Emulation

platforms such as AMD HACCS [4] or NVIDIA AIR [203] could make testing at scale possible without requiring local access to dedicated hardware.

Extension to other data-plane programming languages. All the analysis tools presented in this dissertation have been built for the P4 language. The language has an active ecosystem, extensive emulation software, industrial users, and traits that are amenable to static analysis. We presume that our techniques can also apply to other languages designed for packet processing, but we have not proven that they can. Ideally, we should be able to build our tooling for other available packet-processing DSLs, such as eBPF/XDP [100] or DPDK programming frameworks [97, 84, 230], or NPL [30] to substantiate our claims.

7.2 Future Directions

In this dissertation, we have developed a representation of packet-processing programs in SMT and adapted a variety of general-purpose techniques from programming language and software testing research to our particular execution model. For Gauntlet, we adopted translation validation [172] and grammar-aided random-program generation [261]. In P4Testgen, we took inspiration from symbolic execution and test generation tools such as Klee [35], Dart [89], or CUTE [206]. With Flay, we implemented a combination of incremental computation [175] and partial evaluation [114]. This, of course, is only a small selection of possible techniques, and there are many other techniques we can explore. We can use this representation for a variety of follow-up work, including more verification and testing projects, resource modeling, the development of general interpreters, or network-protocol validation. We describe some concrete examples in this section.

7.2.1 Better Software Testing

By construction, Gauntlet's translation validation is sound. When it finds a mismatch, it is a bug in a compiler transformation. It can also be considered complete, as any semantically meaningful transformation the compiler undertakes is checked by Gauntlet. However, Gauntlet is not complete with respect to the programs we generate to exercise the compiler, because our fuzzer only produces a subset of all possible input programs. P4Testgen is neither sound nor complete. Because P4Testgen uses concolic execution, it may not capture niche cases (e.g., a particular hash is calculated incorrectly). While its goal is to be complete with respect to the P4 program, because it generates tests for any possible program path, it is not with respect to the entire network-device stack. There are several techniques we can use to capture bugs that Gauntlet or P4Testgen may not be able to detect.

7.2.1.1 Further Testing Techniques for Packet-Processing Compilers

Gauntlet uses random-program generation and translation validation to test packet-processing compilers. There are more techniques we can adopt to specifically test compilers for packet processing.

Metamorphic testing. Metamorphic testing [42] is a technique to find bugs in a tool, without requiring knowledge about the expected output for a given input. It does so by checking the relation between two inputs and ensuring that this relation is preserved in the outputs. In the case of the compiler, we can modify a source program in a semantics-preserving way (for example, reordering commutative instructions or variable assignments) and then compare whether this modification leads to a semantic difference in compiler output. This way, we can capture types of programs our current random-program generator is unable to generate. For hardware compilers such as the Tofino compiler, we can use metamorphic testing to identify problems in hardware resource usage. If there are two pieces of code that

are semantically equivalent but cause different resource usage, this might indicate a bug in the compiler's transformations or resource allocation algorithm.

Equivalence-modulo inputs. EMI checks whether code that is functionally considered "dead" influences the execution semantics of the program. Under EMI, one picks an input, exercises the input on the system-under-test, then removes any code pieces that should not be exercised by this particular test. Then, the same test is run again. If the outcome of the test changes, unexpected code has influenced the test behavior, and there might be a bug. We initially dismissed equivalence-modulo-inputs (EMI) [134] for compiler testing in Section 4.2 because Gauntlet can identify miscompilations with higher precision. However, EMI in combination with a test-case oracle such as P4Testgen can uncover bugs that Gauntlet may be unable to detect. The bugs EMI can find are bugs that occur when a compiler chooses to apply different passes based on the structure of the program. We can generate a set of tests with P4Testgen, use the trace produced by P4Testgen, and remove all program elements not exercised by this trace. The test generated by P4Testgen must pass on this reduced program, as we have only removed segments that are, by definition, unused. If the compiler applies a different set of passes for the reduced program, and one of the passes has a bug, the test can fail. While Gauntlet technically could find this failure, Gauntlet's random-program generator is unlikely to produce this kind of reduced program.

Finding performance bugs. Gauntlet cannot find compiler bugs that affect performance or resource usage of generated code. For a switching ASIC that guarantees line-rate performance, the compiler must produce code that consumes a small number of computational and memory units [85]. For software targets where line-rate performance is not guaranteed, the generated code must have good performance. For example, the P4-eBPF compiler, which converts P4 to eBPF/XDP [100] bytecode, occasionally produces code with poor performance [242]. Methods are needed to identify when a compiler pass negatively af-

fects performance and resource usage. We anticipate that handling such bugs would require techniques that are conceptually different from our methods, which deal with correctness bugs.

7.2.1.2 More Coverage of The Network-Device Toolchain

P4Testgen can generate tests comprehensively, but the number of generated tests is often overwhelming. P4Testgen may be complete with respect to coverage for a P4 program, but if it takes years to exercise all tests, the utility for users is low. Hence, we have introduced path-selection strategies and preconditions to generate targeted tests. These two techniques are not the only methods we could use to make test-case generation more targeted.

Property-based testing. We could use P4Testgen for property-based testing [81]. Property-based testing is a test-case generation technique that checks whether a particular function or program satisfies specific abstract properties. In the context of network-device verification, we can use property-based testing to check whether particular protocols are implemented correctly. For example, we could check whether the device stack implements a particular tunneling protocol correctly. This would involve using the P4 program as a base to generate the appropriate input packets and control-plane entries. This requires developing a framework that translates the abstract properties into constraints for P4Testgen's test-case generation and maps them correctly to variables in the P4 program. If all generated tests pass without violating any properties, the property-based test is considered successful. Systems such as Aquila [241], FP4 [257], or Lumina [259] already implement property-based network-device testing. We would need to compare their implementation with our tool.

Mutation testing. The goal of mutation testing [161] is to evaluate test suite quality by checking its ability to detect artificially introduced faults (mutations). A mutation-testing tool inserts faults (mutants) into the program based on predefined mutation operators. If a

test suite fails to distinguish the mutated program from the original (i.e., fails to 'kill' the mutant), it indicates a potential weakness in the test suite. In the context of our modeling work, we could parse an existing packet test, infer its coverage using a tool such as P4Testgen, and then introduce mutations into the code covered by this test. If the test still passes (i.e., does not kill the mutant), it suggests the test may not adequately exercise the mutated code region.

7.2.2 Improving Compiler Optimizations

Our SMT-based model provides a fast, comprehensive way to check program equality. We can use this model to introduce better and safer compiler optimizations.

Superoptimization. We could apply superoptimizations [148] to find the canonical, optimal sequence of instructions for a particular program segment. Closely related to superoptimization are correct-by-construction techniques such as program synthesis [171] or calculating compilers [153]. For all these techniques, we would impose constraints on a sequence of operations (e.g., this particular function should cost no more than 8 instructions) and then try to find an operation sequence that satisfies this constraint. This is usually done via brute-force search, but recent synthesis-based advances also make it possible to use SMT-based techniques [116, 111]. Usually, these SMT-based techniques require translating general-purpose code into logic suitable for posing queries. This can be a difficult process for general-purpose languages, but not for data-plane DSLs. We already have an approach to represent a program in the form of SMT expressions. We can use this representation to ask superoptimization queries. Using our techniques, we are able to describe, with high fidelity, the interfaces and arithmetic operations of a particular function. We can use this compressed information to generate code that is optimal according to criteria that we determine, e.g., latency, throughput, power draw, or memory usage.

Supporting aggressive compiler optimizations. Similar to credible compilation [182], we could repurpose a tool such as Gauntlet as an attachable compiler plugin to facilitate development of experimental compiler optimizations. During compilation, if a newly added optimization produces semantically incorrect code, Gauntlet will notify the compiler to discard the optimization. With this technique, a developer can integrate potentially buggy code into the compiler while still guaranteeing a safe compilation process. However, for the plugin to be useful, Gauntlet's translation validation needs to be fast enough so that compilation time remains acceptable.

7.2.2.1 Program Specialization Up The Network-Device Stacks

With Flay, we specialize data-plane programs by identifying parts of the program that can be changed to better utilize resources. We can also use this model to inform upper layers about which protocols and features are actually available in the data plane. This way, we can automatically disable certain API calls or remove unneeded protocol software implementations. The challenge here is to identify which particular segment of program code correlates to a particular protocol in the upper layers. This mapping is not explicit, and we would need to infer it.

7.2.2.2 Modeling Resource Usage of P4 Programs

Network devices usually have limited resources or operate under tight constraints. A particular problem a developer writing programs for these devices faces is the "fitting problem" [137], i.e., the compiler places memory and resources of the program onto the device in ways the developer does not understand. The fitting problem is common for the Tofino switches, which have an all-or-nothing guarantee of compilation. If the program consumes too many resources, it will be rejected. However, because the Tofino compiler uses certain heuristics, it is poorly understood why it makes the decisions it does. One potential approach to this

problem is to provide high-level feedback early using an abstract resource allocation model. If, for a given device resource map and a program, no placement is possible, feedback can be provided early instead of waiting for a compiler decision. This model could also be baked into the compiler to provide better feedback to users.

7.2.3 A Common Analysis Toolchain for Packet-Processing Programs

We put considerable effort into ensuring that our execution model can generalize to different network devices. We did this by developing whole-program semantics. Whole-program semantics give us a solid foundation to expand our tooling to more areas in network-device programming.

7.2.3.1 An IR to Represent Packet-Processing Behavior

We intend to expand our testing techniques to other ecosystems, primarily eBPF [62]. eBPF also has an active community, existing production applications, and extensive available tooling. What stands in the way of our efforts is that all our tools have been built using P4C's C++ framework, which we use to parse P4 code into P4C-IR. Unfortunately, the compiler front end does not support parsing eBPF code and P4C-IR may not capture all eBPF behavior because it is specialized to P4 programs. We likely need a more general IR, built to support multiple network programming languages and focused on functional analysis of network programs. This IR does not need to capture memory or resource semantics accurately. A viable starting point can be LLVM's MLIR platform [133] that provides common infrastructure and tooling for many languages. There are already projects such as the P4 MLIR project [238], that implements an MLIR dialect for the P4 language.

7.2.3.2 Extending the P4 Language to Model Network-Device Behavior

Both P4Testgen and Flay model a network device's forwarding behavior, and they interpret the P4 language to do so. However, while the P4 language has traits that are amenable to formulating rigorous semantics, it also has gaps. Among others we had to develop whole-program semantics in P4Testgen using a C++ DSL. Anyone who writes a P4Testgen or Flay extension needs to describe the semantics using this C++ DSL, a practice that is cumbersome and confusing. Ideally, we should be able to model an entire device's behavior in the P4 language without having to drop into a separate, general-purpose language. What we believe P4 is currently missing is a way to describe control and data flow between the programmable blocks and also higher-fidelity description of the behavior of various externs, which execute device functionality. Work is already underway to specify P4 architectures and their control flow in P4 [80]. We could extend P4Testgen and Flay to support these new language constructs and check whether they capture the same behavior as our current whole-program semantics.

A P4-based interpreter for packet processing. P4Testgen models how a packet is processed by a network device using a P4 program as its source of truth. It does so accurately enough that it could also be used as a general P4 program interpreter. Combined with architectural P4 descriptions, P4Testgen could be extended with an interface allowing users to query the output generated for a particular input and control-plane configuration. A general interpreter can aid in different types of program analysis. For example, regarding coverage, one could ask questions about which program elements are exercised by a particular packet input. Another use case is linting, where P4 programs could be annotated with diagnostics (e.g., a warning that a particular code path is non-executable on a device). Advanced debugging support is also possible by integrating P4Testgen's small-step approach with an IDE. Developers could then step through a P4 program as if they were working with

general-purpose code. Similar ideas have been proposed with Petr4 [57], but Petr4's main focus remains type-checking P4.

7.2.3.3 Formalizing RFC Standards And Protocol Interactions

Our overarching goal is to develop static analysis tools for network-device software stacks. Often, these software stacks implement network protocols based on RFCs. However, the problem is that these RFCs use human prose and rarely include a formal specification. These RFCs can be ambiguous, which leads to misunderstandings and conflicts between network-device makers and their customers—or even among device makers themselves. How RFCs interact with each other is also poorly understood. Since data-plane languages can describe packet-forwarding behavior using arithmetic and stateful primitives (externs or registers), it is possible to formally specify a network protocol as a data-plane program. We can describe the RFC and even generate tests for it using tools such as P4Testgen to ensure compliance. Work already exists that attempts to generate specifications using NLP or LLM-based techniques [208]. We could produce similar specifications in the form of P4 language programs for a given RFC.

7.3 Concluding Thoughts

Peter Naur once described the act of programming as a form of theory building [157]. When we design a data-plane DSL, we inadvertently create a theory for packet forwarding at the device level—and this theory can help us in developing tools that are simpler and more effective. We believe our execution model would have been less effective had we built it for general-purpose programming languages. For Gauntlet, we were able to find bugs effectively because we exploited the constraints of the P4 language, which was deliberately designed to be unambiguous. In P4Testgen, we were able to develop an effective test-case oracle because

the language had facilities that made it easier to generate input-output packet tests. For Flay, we were able to develop a very efficient partial evaluation algorithm because we focused on the simple language core made available by the DSL. These lessons indicate that focusing development resources on a particular DSL can be beneficial.

Of course, many of these lessons were learned in hindsight. When we started the Gauntlet project, we were not certain whether we would be able to capture P4's execution model entirely in SMT formulas. SMT encoding proved to be a great match for packet processing. We were able to expand this encoding beyond P4 to the entire networking device, and even to the control plane. Now we have an extensible model that can capture a wide range of device behaviors, and we can use it for a variety of static analysis tooling.

Another factor that made the development of this model significantly easier is the availability of an open-source programming language and software stack. Historically, much of packet-processing technology was closed off and compartmentalized, restricted to proprietary interfaces of expensive hardware [38, 82]. This made it difficult to collaborate and innovate. On the other hand, recent openness has created a great environment for experimentation. When we pursued our bug-finding with Gauntlet and P4Testgen, we were encouraged by the responsiveness of the community. And, we now have open and well-specified data-plane programming languages in the form of P4 and eBPF with active communities, open networking-device stacks (SONiC [229], PINS [228], FBOSS [46]), open-hardware SDKs such as OpenTofino and its emulator [237], and even open programmable networking hardware [252]. These open-source frameworks are lowering the barrier to entry into networking research and are also democratizing access to leading-edge technology. We built much of our analysis tools using open-source tooling such as emulators and compilers. The better the fidelity of this tooling, the better analysis tools we can develop, and the more we will be able to improve network packet processing at large.

Bibliography

[1] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. Tensorflow: A system for large-scale machine learning. In *USENIX OSDI* (2016).

[2] Abhashkumar, A., Lee, J., Tourrilhes, J., Banerjee, S., Wu, W., Kang, J.-M., and Akella, A. P5: Policy-driven optimization of P4 pipeline. In *ACM SOSR* (2017).

[3] Advanced Micro Devices, Inc. AMD Alveo SN1000 SmartNIC Accelerator Card. https://web.archive.org/web/20250226015516/https://www.amd.com/en/products/accelerators/alveo/sn1000/a-sn1022-p4.html, 2023. Accessed: 2025-05-01.

[4] Advanced Micro Devices, Inc. Heterogeneous accelerated compute clusters. https://web.archive.org/web/20250227000117/https://www.amd-haccs.io/, 2023. Accessed: 2025-05-01.

[5] Advanced Micro Devices, Inc. Content addressable memory (CAM). https://web.archive.org/web/20241206222954/https://www.xilinx.com/products/intellectual-property/ef-di-cam.html, 2024. Accessed: 2025-05-01.

[6] Agape, A. A., Dănceanu, M. C., Hansen, R. R., and Stefan, S. P4Fuzz: Compiler fuzzer for dependable programmable dataplanes. In *ACM ICDCN* (2021).

[7] Agrawal, A., and Kim, C. Intel Tofino2–a 12.9 tbps P4-programmable ethernet switch. In *2020 IEEE Hot Chips 32 Symposium (HCS)* (2020).

[8] Ahluwalia, S. Table driven interface (TDI): Usages and advantages. https://web.archive.org/web/20250112221624/https://opennetworking.org/news-and-events/blog/table-driven-interface-api-opens-p4-programmable-data-plane-features/, 2021. Accessed: 2025-05-01.

[9] Albab, K. D., Dilorenzo, J., Heule, S., Kheradmand, A., Smolka, S., Weitz, K., Tirmazi, M., Gao, J., and Yu, M. SwitchV: Automated SDN switch validation with P4 models. In *ACM SIGCOMM* (2022).

[10] Alcoz, A. G., Strohmeier, M., Lenders, V., and Vanbever, L. Aggregate-based congestion control for pulse-wave DDoS defense. In *ACM SIGCOMM* (2022).

[11] Almquist, P., and Kastenholz, F. Towards Requirements for IP Routers. RFC 1716, 1994.

[12] Alshnakat, A., Lundberg, D., Guanciale, R., and Dam, M. HOL4P4: Mechanized small-step semantics for P4. *Proceedings of the ACM on Programming Languages* (2024).

[13] Amadini, R., Gange, G., Schachte, P., Sø ndergaard, H., and Stuckey, P. J. Abstract interpretation, symbolic execution and constraints. In *Recent Developments in the Design and Implementation of Programming Languages* (2020).

[14] anasyrmia. Fix: Predication issue #2345. https://github.com/p4lang/p4c/pull/2564, 2020. Accessed: 2025-05-01.

[15] Arumugam, M., Bansal, D., Bhatia, N., Boerner, J., Capper, S., Kim, C., McClure, S., Motwani, N., Narasimhan, R., Panchal, U., et al. Bluebird: High-performance sdn for bare-metal cloud services. In *USENIX NSDI* (2022).

[16] Baldoni, R., Coppa, E., D'Elia, D. C., Demetrescu, C., and Finocchi, I. A survey of symbolic execution techniques. *ACM Computing Surveys* (2018).

[17] Ball, T., and Larus, J. R. Efficient path profiling. In *IEEE MICRO* (1996).

[18] Barrett, C., de Moura, L., and Stump, A. SMT-COMP: Satisfiability modulo theories competition. In *Computer Aided Verification (CAV)* (2005).

[19] Barrett, C. W., Dill, D. L., and Levitt, J. R. A decision procedure for bit-vector arithmetic. In *Proceedings of the 35th Annual Design Automation Conference* (1998).

[20] Beckett, R., Gupta, A., Mahajan, R., and Walker, D. A general approach to network configuration verification. In *ACM SIGCOMM* (2017).

[21] Beckett, R., and Mahajan, R. Capturing the state of research on network verification. https://web.archive.org/web/20240625135948/https://netverify.fun/2-current-state-of-research/. Accessed: 2025-05-01.

[22] Bhatia, S., Consel, C., Le Meur, A.-F., and Pu, C. Automatic specialization of protocol stacks in operating system kernels. In *29th Annual IEEE International Conference on Local Computer Networks* (2004).

[23] Bishop, S., Fairbairn, M., Norrish, M., Sewell, P., Smith, M., and Wansbrough, K. Rigorous specification and conformance testing techniques for network protocols, as applied to TCP, UDP, and Sockets. In *ACM SIGCOMM* (2005).

[24] Bodik, R., Chandra, S., Galenson, J., Kimelman, D., Tung, N., Barman, S., and Rodarmor, C. Programming with angelic nondeterminism. In *ACM POPL* (2010).

[25] Bosshart, P. Programmable forwarding planes at terabit/s speeds. In *2018 IEEE Hot Chips 30 Symposium (HCS)* (2018).

[26] Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G., et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review* (2014).

[27] Bradner, S., and McQuaid, J. Benchmarking methodology for network interconnect devices. RFC 2544, 1999.

[28] Bradner, S. O. Benchmarking Terminology for Network Interconnection Devices. RFC 1242, 1991.

[29] Bressana, P., Zilberman, N., and Soulé, R. Finding hard-to-find data plane bugs with a PTA. In *ACM CoNEXT* (2020).

[30] Broadcom, Inc. NPL: Open, high-level language for developing feature-rich solutions for programmable networking platforms. https://web.archive.org/web/20250118083052/https://nplang.org/, 2019. Accessed: 2025-05-01.

[31] Brown, M., Fogel, A., Halperin, D., Heorhiadi, V., Mahajan, R., and Millstein, T. Lessons from the evolution of the Batfish configuration analysis tool. In *ACM SIGCOMM* (2023).

[32] Budiu, M. The $P4_{16}$ reference compiler implementation architecture. https://github.com/p4lang/p4c/blob/03c6717fd3e6bd2db969eab8ff0e4553dd8aa26e/docs/compiler-design.pptx, 2018. Accessed: 2025-05-01.

[33] Budiu, M. Tuple elim. https://github.com/p4lang/p4c/pull/2451, 2020. Accessed: 2025-05-01.

[34] Budiu, M., and Dodd, C. The $P4_{16}$ programming language. *ACM SIGOPS Operating Systems Review* (2017).

[35] Cadar, C., Dunbar, D., Engler, D. R., et al. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *USENIX OSDI* (2008).

[36] Cardwell, N., Cheng, Y., Brakmo, L., Mathis, M., Raghavan, B., Dukkipati, N., Chu, H.-k. J., Terzis, A., and Herbert, T. packetdrill: Scriptable network stack testing, from sockets to packets. In *USENIX ATC* (2013).

[37] Casado, M., Koponen, T., Moon, D., and Shenker, S. Rethinking packet forwarding hardware. In *Proceedings of the 7th ACM Workshop on Hot Topics in Networks* (2008).

[38] Casado, M., McKeown, N., and Shenker, S. From ethane to SDN and beyond. *ACM SIGCOMM Computer Communication Review* (2019).

[39] Chan, W. Y., Vuong, C., and Otp, M. An improved protocol test generation procedure based on UIOs. *ACM SIGCOMM Computer Communication Review* (1989).

[40] Chen, D., Li, D. X., and Moseley, T. AutoFDO: Automatic feedback-directed optimization for warehouse-scale applications. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization* (2016).

[41] Chen, J., Hu, W., Hao, D., Xiong, Y., Zhang, H., Zhang, L., and Xie, B. An empirical comparison of compiler testing techniques. In *ACM/IEEE ICSE* (2016).

[42] Chen, T. Y., Cheung, S. C., and Yiu, S. M. Metamorphic testing: A new approach for generating next test cases. *arXiv preprint arXiv:2002.12543* (1998).

[43] Chen, X. Open source P4 implementations. https://github.com/Princeton-Cabernet/p4-projects, 2018. Accessed: 2025-05-01.

[44] Chen, X., Feibish, S. L., Braverman, M., and Rexford, J. Beaucoup: Answering many network traffic queries, one memory update at a time. In *ACM SIGCOMM* (2020).

[45] Cheng, G. P4 practice at Baidu. https://web.archive.org/web/20241203002715/https://opennetworking.org/wp-content/uploads/2021/05/2021-P4-WS-Gang-Cheng-Slides.pdf, 2021. Accessed: 2025-05-01.

[46] Choi, S., Burkov, B., Eckert, A., Fang, T., Kazemkhani, S., Sherwood, R., Zhang, Y., and Zeng, H. FBOSS: building switch software at scale. In *ACM SIGCOMM* (2018).

[47] Chopra, R. Cisco Silicon One breaks the 51.2 Tbps barrier. https://web.archive.org/web/20241212115412/https://blogs.cisco.com/sp/cisco-silicon-one-breaks-the-51-2-tbps-barrier, 2023. Accessed: 2025-05-01.

[48] Clark, D. The design philosophy of the DARPA internet protocols. In *Symposium proceedings on Communications architectures and protocols* (1988).

[49] Consel, C., Lawall, J. L., and Le Meur, A.-F. o. A tour of Tempo: a program specializer for the c language. *Science of computer programming* (2004).

[50] Cubro. Next generation network packet broker based on latest p4 programmable chips. https://web.archive.org/web/20240723105341/https://www.cubro.com/en/blog/next-generation-network-packet-broker-based-on-latest-p4-programmable-chips/, 2022. Accessed: 2025-05-01.

[51] Das, R., and Snoeren, A. C. Memory management in ActiveRMT: Towards runtime-programmable switches. In *ACM SIGCOMM* (2023).

[52] De Moura, L., and Bjørner, N. Efficient e-matching for SMT solvers. In *CADE-21: 21st International Conference on Automated Deduction* (2007).

[53] De Moura, L., and Bjørner, N. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems* (2008).

[54] De Moura, L., and Bjørner, N. Satisfiability modulo theories: An appetizer. In *Brazilian Symposium on Formal Methods* (2009), Springer.

[55] de Ruiter, J., and Schutijser, C. Next-generation internet at terabit speed: SCION in P4. In *ACM CoNEXT* (2021).

[56] Deng, B., Wu, W., and Song, L. Redundant logic elimination in network functions. In *ACM SOSR* (2020).

[57] Doenges, R., Arashloo, M. T., Bautista, S., Chang, A., Ni, N., Parkinson, S., Peterson, R., Solko-Breslin, A., Xu, A., and Foster, N. Petr4: Formal foundations for P4 data planes. In *ACM POPL* (2021).

[58] Dumitrescu, C. Develop your CPU network stack in P4. https://web.archive.org/web/20240723052907/https://opennetworking.org/wp-content/uploads/2022/05/Cristian-Dumitrescu-Final-Slide-Deck.pdf, 2022. Accessed: 2025-05-01.

[59] Dumitrescu, D., Stoenescu, R., Negreanu, L., and Raiciu, C. bf4: Towards bug-free P4 programs. In *ACM SIGCOMM* (2020).

[60] Dumitrescu, D., Stoenescu, R., Popovici, M., Negreanu, L., and Raiciu, C. Dataplane equivalence and its applications. In *USENIX NSDI* (2019).

[61] Dumitru, M. V., Dumitrescu, D., and Raiciu, C. Can we exploit buggy P4 programs? In *ACM SOSR* (2020).

[62] eBPF.io. eBPF: Introduction, tutorials & community resources. https://web.archive.org/web/20250321050340/https://ebpf.io/, 2022. Accessed: 2025-05-01.

[63] Eichholtz, M., Campbell, E., Foster, N., Salvaneschi, G., and Mezini, M. How to avoid making a billion-dollar mistake: Type-safe data plane programming with SafeP4. *arXiv preprint arXiv:1906.07223* (2019).

[64] Enns, R. NETCONF configuration protocol. RFC 4741, 2006.

[65] Extreme Networks. Extreme 9920: Cloud-native network visibility platform. https://web.archive.org/web/20250125232230/https://www.extremenetworks.com/products/network-packet-broker/9920-hardware-platform/extreme-9920-cloud-native-network-visibility-platform. Accessed: 2025-05-01.

[66] Farrington, N., Rubow, E., and Vahdat, A. Data center switch architecture in the age of merchant silicon. In *2009 17th ieee symposium on high performance interconnects* (2009), IEEE.

[67] Farshin, A., Barbette, T., Roozbeh, A., Maguire Jr, G. Q., and Kostić, D. Packetmill: toward per-core 100-gbps networking. In *ACM ASPLOS* (2021).

[68] Fattaholmanan, A., Baldi, M., Carzaniga, A., and Soulé, R. P4 weaver: Supporting modular and incremental programming in P4. In *ACM SOSR* (2021).

[69] Fayaz, S. K., Sharma, T., Fogel, A., Mahajan, R., Millstein, T., Sekar, V., and Varghese, G. Efficient network reachability analysis using a succinct control plane representation. In *USENIX OSDI* (2016).

[70] Feamster, N., Rexford, J., and Zegura, E. The road to SDN: an intellectual history of programmable networks. *ACM SIGCOMM Computer Communication Review* (2014).

[71] Fedor, M., Schoffstall, M. L., Davin, J. R., and Case, D. J. D. Simple Network Management Protocol (SNMP). RFC 1157, 1990.

[72] Feng, Y., Chen, Z., Song, H., Xu, W., Li, J., Zhang, Z., Yun, T., Wan, Y., and Liu, B. Enabling in-situ programmability in network data plane: From architecture to language. In *USENIX NSDI* (2022).

[73] Fingerhut, A. The BMv2 simple switch target. https://github.com/p4lang/behavioral-model/blob/d12eefc7bc19fb4da615b1b45c1235899f2e4fb1/docs/simple_switch.md, 2016. Accessed: 2025-05-01.

[74] Fingerhut, A. Behavioral model targets. https://github.com/p4lang/behavioral-model/blob/d12eefc7bc19fb4da615b1b45c1235899f2e4fb1/targets/README.md, 2018. Accessed: 2025-05-01.

[75] Fingerhut, A. Forbid shifts with unknown widths. https://github.com/p4lang/p4-spec/pull/814, 2020. Accessed: 2025-05-01.

[76] Fingerhut, A. Incorrect transformation in predication pass. https://github.com/p4lang/p4c/issues/2345, 2020. Accessed: 2025-05-01.

[77] Fingerhut, A. Make stricter PSA tests that verify packet_path and instance fields. https://github.com/p4lang/p4c/pull/2509, 2020. Accessed: 2025-05-01.

[78] Fingerhut, A. Reducing requirements for initializing headers. https://github.com/p4lang/p4-spec/issues/849, 2020. Accessed: 2025-05-01.

[79] Fingerhut, A. Specify that copy-out behavior still occurs after return/exit statements. https://github.com/p4lang/p4-spec/pull/823, 2020. Accessed: 2025-05-01.

[80] Fingerhut, A. Specifying p4 architectures. https://github.com/jafingerhut/p4-guide/tree/ee492b89e3e58cfe219ab843049d557ece5ed6b1/specifying-p4-architectures, 2024. Accessed: 2025-05-01.

[81] Fink, G., and Bishop, M. Property-based testing: a new approach to testing for assurance. *ACM SIGSOFT Software Engineering Notes* (1997).

[82] Foster, N., McKeown, N., Rexford, J., Parulkar, G., Peterson, L., and Sunay, O. Using deep programmability to put network owners in control. *ACM SIGCOMM Computer Communication Review* (2020).

[83] Freire, L., Neves, M., Leal, L., Levchenko, K., Schaeffer-Filho, A., and Barcellos, M. Uncovering bugs in P4 programs with assertion-based verification. In *ACM SOSR* (2018).

[84] Gallo, M., and Laufer, R. ClickNF: a modular stack for custom network functions. In *USENIX ATC* (2018).

[85] Gao, X., Kim, T., Wong, M. D., Raghunathan, D., Varma, A. K., Kannan, P. G., Sivaraman, A., Narayana, S., and Gupta, A. Switch code generation using program synthesis. In *ACM SIGCOMM* (2020).

[86] Gao, X., Raghunathan, D., Fang, R., Wang, T., Zhu, X., Sivaraman, A., Narayana, S., and Gupta, A. Cat: A solver-aided compiler for packet-processing pipelines. In *ACM ASPLOS* (2023).

[87] Gmitter, J. Transforming AI networks with AMD Pensando Pollara 400. https://web.archive.org/web/20241217185814/https://community.amd.com/t5/corporate/transforming-ai-networks-with-amd-pensando-pollara-400/ba-p/716566, 2024. Accessed: 2025-05-01.

[88] GNU Project. gcov–a test coverage program. https://web.archive.org/web/20250321012000/https://gcc.gnu.org/onlinedocs/gcc/Gcov.html, 1987. Accessed: 2025-05-01.

[89] Godefroid, P., Klarlund, N., and Sen, K. DART: Directed automated random testing. In *ACM POPL* (2005).

[90] Goodfellow, R. Building a rack-scale computer with P4 at the core. https://web.archive.org/web/20241225042831/https://opennetworking.org/news-and-events/blog/building-a-rack-scale-computer-with-p4-at-the-core/, 2023. Accessed: 2025-05-01.

[91] Google LLC. Protocol buffers. https://web.archive.org/web/20250321035939/https://protobuf.dev/, 2008. Accessed: 2025-05-01.

[92] Greene, W., and Lancaster, B. Carrier-grade: Five nines, the myth and the reality. *Pipeline magazine* (2007).

[93] Guo, D., Chen, S., Gao, K., Xiang, Q., Zhang, Y., and Yang, Y. R. Flash: fast, consistent data plane verification for large-scale network settings. In *ACM SIGCOMM* (2022).

[94] Gurevich, V. Change parser exception model and provide better controls for exceptional situation handling. https://github.com/p4lang/p4-spec/issues/880, 2020. Accessed: 2025-05-01.

[95] Hadi Salim, J., Chatterjee, D., Nogueira, V., Tammela, P., Osinski, T., Haleplidis, E., Sambasivam, B., Gupta, U., Jain, K., and Sethuramapandian, S. Introducing P4TC-a P4 implementation on linux kernel using traffic control. In *Proceedings of the 6th on European P4 Workshop* (2023).

[96] Hadzic, E. Added support for assert and assume primitives in bm_sim. https://github.com/p4lang/behavioral-model/pull/762, 2019. Accessed: 2025-05-01.

[97] Han, S., Jang, K., Panda, A., Palkar, S., Han, D., and Ratnasamy, S. SoftNIC: A software NIC to augment hardware. Tech. rep., University of California at Berkeley, 2015.

[98] Hawblitzel, C., Lahiri, S. K., Pawar, K., Hashmi, H., Gokbulut, S., Fernando, L., Detlefs, D., and Wadsworth, S. Will you still compile me tomorrow? static cross-version compiler validation. In *ACM ESEC/FSE* (2013).

[99] Hogan, M., Loehr, D., Sonchack, J., Feibish, S. L., Rexford, J., and Walker, D. Automated optimization of parameterized data-plane programs with Parasol. *IEEE/ACM Transactions on Networking* (2024).

[100] Høiland-Jørgensen, T., Brouer, J. D., Borkmann, D., Fastabend, J., Herbert, T., Ahern, D., and Miller, D. The EXpress Data Path: Fast programmable packet processing in the operating system kernel. In *ACM CoNEXT* (2018).

[101] Holterbach, T., Costa Molero, E., Apostolaki, M., Dainotti, A., Vissicchio, S., and Vanbever, L. Blink: Fast connectivity recovery entirely in the data plane. In *USENIX NSDI* (2019).

[102] Holterbach, T., Vissicchio, S., Dainotti, A., and Vanbever, L. SWIFT: Predictive fast reroute. In *ACM SIGCOMM* (2017).

[103] Honda, M., Huici, F., Lettieri, G., and Rizzo, L. mswitch: a highly-scalable, modular software switch. In *ACM SOSR* (2015).

[104] Huang, D. Y., Yocum, K., and Snoeren, A. C. High-fidelity switch models for software-defined network emulation. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking* (2013).

[105] Ichbiah, J. D., Krieg-Brueckner, B., Wichmann, B. A., Barnes, J. G., Roubine, O., and Heliard, J.-C. Rationale for the design of the Ada programming language. *ACM SIGPLAN notices* (1979).

[106] Intel Corporation. P4-16 Intel Tofino native architecture - public version. https://github.com/barefootnetworks/Open-Tofino/blob/5b6ef19873698fc7d9c49cb33fd54c5fca2ecadd/PUBLIC_Tofino-Native-Arch.pdf, 2021. Accessed: 2025-05-01.

[107] Intel Corporation. The infrastructure processing unit (IPU). https://web.archive.org/web/20241127021021/https://www.intel.de/content/www/de/de/products/details/network-io/ipu.html, 2022. Accessed: 2025-05-01.

[108] Intel Corporation. Agilex 9 FPGA and soc FPGA. https://web.archive.org/web/20250207120602/https://www.intel.com/content/www/us/en/products/details/fpga/agilex/9.html, 2024. Accessed: 2025-05-01.

[109] International Telecommunication Union. Ethernet service activation test methodology. Y.1564, 2016.

[110] Jain, S., Kumar, A., Mandal, S., Ong, J., Poutievski, L., Singh, A., Venkata, S., Wanderer, J., Zhou, J., Zhu, M., et al. B4: Experience with a globally-deployed software defined WAN. In *ACM SIGCOMM* (2015).

[111] Jangda, A., and Yorsh, G. Unbounded superoptimization. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (2017).

[112] Jin, X., Liu, H. H., Gandhi, R., Kandula, S., Mahajan, R., Zhang, M., Rexford, J., and Wattenhofer, R. Dynamic scheduling of network updates. *ACM SIGCOMM Computer Communication Review* (2014).

[113] Johansson, S. Packet deduplication in p4. https://web.archive.org/web/20241008234238/https://opennetworking.org/wp-content/uploads/2021/05/2021-P4-WS-Stefan-Johansson-Slides.pdf, 2021. Accessed: 2025-05-01.

[114] Jones, N. D. An introduction to partial evaluation. *ACM Computing Surveys* (1996).

[115] Jones, N. D., and Glenstrup, A. J. Program generation, termination, and binding-time analysis. In *Generative Programming and Component Engineering: ACM SIGPLAN/SIGSOFT Conference* (2002).

[116] Joshi, R., Nelson, G., and Randall, K. Denali: A goal-directed superoptimizer. *ACM SIGPLAN Notices* (2002).

[117] Kang, J., Kim, Y., Song, Y., Lee, J., Park, S., Shin, M. D., Kim, Y., Cho, S., Choi, J., Hur, C.-K., et al. Crellvm: Verified credible compilation for LLVM. In *ACM PLDI* (2018).

[118] Kang, N., Liu, Z., Rexford, J., and Walker, D. Optimizing the" one big switch" abstraction in software-defined networks. In *ACM CoNEXT* (2013).

[119] Kang, Q., Xing, J., Qiu, Y., and Chen, A. Probabilistic profiling of stateful data planes for adversarial testing. In *ACM ASPLOS* (2021).

[120] Kazemian, P., Varghese, G., and McKeown, N. Header space analysis: Static checking for networks. In *USENIX NSDI* (2012).

[121] Kheradmand, A., and Rosu, G. P4K: A formal semantics of P4 and applications. *arXiv preprint arXiv:1804.01468* (2018).

[122] Khurshid, A., Zhou, W., Caesar, M., and Godfrey, P. B. Veriflow: Verifying network-wide invariants in real time. In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks* (2012).

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

[124] Kit, A. Programming the entire data center infrastructure with the NVIDIA DOCA SDK. https://web.archive.org/web/20250117050617/https://developer.nvidia.com/blog/programming-the-entire-data-center-infrastructure-with-the-nvidia-doca-sdk/, 2020. Accessed: 2025-05-01.

[125] Klees, G., Ruef, A., Cooper, B., Wei, S., and Hicks, M. Evaluating fuzz testing. In *Conference on Computer and Communications Security (CCS)* (2018).

[126] Knopp, T., Chu, J., and Ahmad, S. AMD Versal AI edge series gen 2 for vision and automotive. In *2024 IEEE Hot Chips 36 Symposium (HCS)* (2024).

[127] Kodeswaran, S., Arashloo, M. T., Tammana, P., and Rexford, J. Tracking P4 program execution in the data plane. In *ACM SOSR* (2020).

[128] Kohler, E., Morris, R., Chen, B., Jannotti, J., and Kaashoek, M. F. The Click modular router. *ACM Transactions on Computer Systems (TOCS)* (2000).

[129] Kumar, K. S., Prashanth, P., Arashloo, M. T., U, V., and Tammana, P. DBVal: Validating P4 data plane runtime behavior. In *ACM SOSR* (2021).

[130] Landi, W. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems* (1992).

[131] Langlet, J., Ben Basat, R., Oliaro, G., Mitzenmacher, M., Yu, M., and Antichi, G. Direct telemetry access. In *ACM SIGCOMM* (2023).

[132] Lattner, C., and Adve, V. Llvm: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.* (2004), IEEE.

[133] Lattner, C., Amini, M., Bondhugula, U., Cohen, A., Davis, A., Pienaar, J., Riddle, R., Shpeisman, T., Vasilache, N., and Zinenko, O. Mlir: A compiler infrastructure for the end of moore's law. *arXiv preprint arXiv:2002.11054* (2020).

[134] Le, V., Afshari, M., and Su, Z. Compiler validation via equivalence modulo inputs. *ACM SIGPLAN Notices* (2014).

[135] Le, V., Sun, C., and Su, Z. Finding deep compiler bugs via guided stochastic program mutation. In *ACM OOPSLA* (2015).

[136] Leroy, X. Formal certification of a compiler back-end or: Programming a compiler with a proof assistant. In *ACM POPL* (2006).

[137] Li, Y., Gao, J., Zhai, E., Liu, M., Liu, K., and Liu, H. H. Cetus: Releasing p4 programmers from the chore of trial and error compiling. In *USENIX NSDI* (2022).

[138] Liang, E., Zhu, H., Jin, X., and Stoica, I. Neural packet classification. In *ACM SIGCOMM* (2019).

[139] Liu, H. H., Zhu, Y., Padhye, J., Cao, J., Tallapragada, S., Lopes, N. P., Rybalchenko, A., Lu, G., and Yuan, L. Crystalnet: Faithfully emulating large production networks. In *ACM SOSP* (2017).

[140] Liu, J., Hallahan, W., Schlesinger, C., Sharif, M., Lee, J., Soulé, R., Wang, H., Ca şcaval, C., McKeown, N., and Foster, N. p4v: Practical verification for programmable data planes. In *ACM SIGCOMM* (2018).

[141] Lopes, N. P., Menendez, D., Nagarakatte, S., and Regehr, J. Provably correct peephole optimizations with Alive. In *ACM PLDI* (2015).

[142] MacDavid, R., Cascone, C., Lin, P., Padmanabhan, B., Thakur, A., Peterson, L., Rexford, J., and Sunay, O. A P4-based 5G user plane function. In *ACM SOSR* (2021).

[143] Mahalingam, M., Dutt, D., Duda, K., Agarwal, P., Kreeger, L., Sridhar, T., Bursell, M., and Wright, C. Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks. RFC 7348, 2014.

[144] Mai, H., Khurshid, A., Agarwal, R., Caesar, M., Godfrey, P. B., and King, S. T. Debugging the data plane with Anteater. In *ACM SIGCOMM* (2011).

[145] Mandeville, R., and Perser, J. Benchmarking methodology for LAN switching devices. RFC 2889, 2000.

[146] Mao, J., Ding, H., Zhai, J., and Ma, S. Merlin: Multi-tier optimization of eBPF code for performance and compactness. In *ACM ASPLOS* (2024).

[147] Marvell. Marvell Teralynx 10 data center Ethernet switch. https://web.archive.org/web/20250308094312/https://www.marvell.com/content/dam/marvell/en/p

ublic-collateral/switching/marvell-teralynx-10-data-center-ethernet-switch-product-brief.pdf. Accessed: 2025-05-01.

[148] Massalin, H. Superoptimizer: a look at the smallest program. *ACM SIGARCH Computer Architecture News* (1987).

[149] Matoušek, J., Antichi, G., Luč anskỳ, A., Moore, A. W., and Kořenek, J. Classbench-ng: Recasting classbench after a decade of network evolution. In *2017 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)* (2017), IEEE.

[150] McCabe, T. J. A complexity measure. *IEEE Transactions on software Engineering* (1976).

[151] McKeeman, W. M. Differential testing for software. *Digital Technical Journal* (1998).

[152] McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., and Turner, J. OpenFlow: Enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review* (2008).

[153] Meijer, H. J. M. *Calculating Compilers*. PhD thesis, Radboud University Nijmegen, 1992.

[154] Miano, S., Sanaee, A., Risso, F., Ré tvári, G., and Antichi, G. Domain specific run time optimization for software data planes. In *ACM ASPLOS* (2022).

[155] Molnár, L., Pongrácz, G., Enyedi, G. á., Kis, Z. L., Csikor, L., Juhá sz, F., Kőröси, A., and Rétvári, G. b. Dataplane specialization for high-performance openflow software switching. In *ACM SIGCOMM* (2016).

[156] Musuvathi, M., Engler, D. R., et al. Model checking large network protocol implementations. In *USENIX NSDI* (2004).

[157] Naur, P. Programming as theory building. *Microprocessing and microprogramming* (1985).

[158] Necula, G. C. Translation validation for an optimizing compiler. In *ACM PLDI* (2000).

[159] Niemetz, A., and Preiner, M. Bitwuzla. In *International Conference on Computer Aided Verification* (2023).

[160] Nötzli, A., Khan, J., Fingerhut, A., Barrett, C., and Athanas, P. p4pktgen: Automated test case generation for P4 programs. In *ACM SOSR* (2018).

[161] Offutt, A. J., and Untch, R. H. Mutation 2000: Uniting the orthogonal. *Mutation testing for the new century* (2001).

[162] Open Compute Project. SAI: Switch abstraction interface. https://web.archive.org/web/20241210200738/https://www.opencompute.org/projects/sai, 2015. Accessed: 2025-05-01.

[163] Orange. psabpf - in-kernel p4 software switch. https://github.com/NIKSS-vSwitch/nikss. Accessed: 2025-05-01.

[164] Osiński, T., Palimąka, J., Kossakowski, M., Tran, F. D., Bonfoh, E.-F., and Tarasiuk, H. A novel programmable software datapath for software-defined networking. In *ACM CoNEXT* (2022).

[165] Pall, M., Gorrie, L., Wingo, A., Rottenkolber, M., Garcia, D. P., Takikawa, A., Tallon, J., Pino, D., Gall, A., et al. *Snabb Reference Manual*, 2019. Accessed: 2025-01-06.

[166] Pan, H., He, P., Li, Z., Zhang, P., Wan, J., Zhou, Y., Duan, X., Zhang, Y., and Xie, G. Hoda: a high-performance Open vSwitch dataplane with multiple specialized data paths. In *Proceedings of the Nineteenth European Conference on Computer Systems* (2024).

[167] Pan, T., Liu, K., Wei, X., Qiao, Y., Hu, J., Li, Z., Liang, J., Cheng, T., Su, W., Lu, J., et al. LuoShen: A hyper-converged programmable gateway for multi-tenant multi-service edge clouds. In *USENIX NSDI* (2024).

[168] Perešíni, P., Kuźniar, M., and Kostić, D. Monocle: Dynamic, fine-grained data plane monitoring. In *ACM CoNEXT* (2015).

[169] Peterson, R., Campbell, E. H., Chen, J., Isak, N., Shyu, C., Doenges, R., Ataei, P., and Foster, N. P4Cub: A little language for big routers. In *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs* (2023).

[170] Pfaff, B., Pettit, J., Koponen, T., Jackson, E., Zhou, A., Rajahalme, J., Gross, J., Wang, A., Stringer, J., Shelar, P., et al. The design and implementation of open vswitch. In *USENIX NSDI* (2015).

[171] Pnueli, A., and Rosner, R. On the synthesis of a reactive module. In *ACM POPL* (1989).

[172] Pnueli, A., Siegel, M., and Singerman, E. Translation validation. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (1998), Springer.

[173] Postel, J. Internet Protocol. RFC 791, 1981.

[174] Qiu, Y., Beckett, R., and Chen, A. Synthesizing runtime programmable switch updates. In *USENIX NSDI* (2023).

[175] Ramalingam, G., and Reps, T. A categorized bibliography on incremental computation. In *ACM POPL* (1993).

[176] Rashelbach, A., Rottenstreich, O., and Silberstein, M. A computational approach to packet classification. In *ACM SIGCOMM* (2020).

[177] Rashelbach, A., Rottenstreich, O., and Silberstein, M. Scaling open vswitch with a computational cache. In *USENIX NSDI* (2022).

[178] Rayner, D. OSI conformance testing. *Computer Networks and ISDN Systems* (1987).

[179] Regehr, J., Chen, Y., Cuoq, P., Eide, E., Ellison, C., and Yang, X. Test-case reduction for C compiler bugs. In *ACM PLDI* (2012).

[180] Rekhter, Y., Hares, S., and Li, T. A Border Gateway Protocol 4 (BGP-4). RFC 4271, 2006.

[181] Reynolds, J. C. The discoveries of continuations. *LISP and Symbolic Computation* (1993).

[182] Rinard, M. C. Credible compilation. Tech. rep., Massachusetts Institute of Technology, 2003.

[183] Rizzo, L., and Lettieri, G. VALE, a switched ethernet for virtual machines. In *ACM CoNEXT* (2012).

[184] Roberts, L. G. The evolution of packet switching. *Proceedings of the IEEE* (1978).

[185] Ruffy, F. Add Travis validation tests for P4C. https://github.com/p4lang/p4c/pull/2458, 2020. Accessed: 2025-05-01.

[186] Ruffy, F. BMV2 backend compiler bug unhandled case. https://github.com/p4lang/p4c/issues/2291, 2020. Accessed: 2025-05-01.

[187] Ruffy, F. Calling exit in actions after an assignment. https://github.com/p4lang/p4c/issues/2225, 2020. Accessed: 2025-05-01.

[188] Ruffy, F. Compiler bug: Null cst. https://github.com/p4lang/p4c/issues/2206, 2020. Accessed: 2025-05-01.

[189] Ruffy, F. Missing StrengthReduction for complex expressions in actions. https://github.com/p4lang/p4c/issues/2279, 2020. Accessed: 2025-05-01.

[190] Ruffy, F. More questions on setInvalid. https://github.com/p4lang/p4c/issues/2323, 2020. Accessed: 2025-05-01.

[191] Ruffy, F. Question about parser behavior with right shifts. https://github.com/p4lang/p4c/issues/2156, 2020. Accessed: 2025-05-01.

[192] Ruffy, F. SimplifyDefUse incorrectly removes assignment in actions with slices as arguments. https://github.com/p4lang/p4c/issues/2147, 2020. Accessed: 2025-05-01.

[193] Ruffy, F. Question about expected output when all headers are invalid. https://github.com/p4lang/behavioral-model/issues/977, 2021. Accessed: 2025-05-01.

[194] Ruffy, F. Dead code in dash_pipeline.p4 pna version. https://github.com/sonic-net/DASH/issues/399, 2023. Accessed: 2025-05-01.

[195] Ruffy, F., Liu, J., Kotikalapudi, P., Havel, V., Tavante, H., Sherwood, R., Dubina, V., Peschanenko, V., Sivaraman, A., and Foster, N. P4Testgen: An extensible test oracle for P4. In *ACM SIGCOMM* (2023).

[196] Ruffy, F., Wang, T., and Sivaraman, A. Gauntlet: Finding bugs in compilers for programmable packet processing. In *USENIX OSDI* (2020).

[197] Ruffy, F., Wang, Z., Antichi, G., Panda, A., and Sivaraman, A. Incremental specialization of network programs. In *Proceedings of the 23rd ACM Workshop on Hot Topics in Networks* (2024).

[198] Ruffy, F., and Zhanghan, W. ControlPlaneSmith: Generate control-plane configurations from P4 programs. https://github.com/nyu-systems/rtsmith, 2024. Accessed: 2025-05-01.

[199] Ruffy, F., and Zhanghan, W. OpenConfig: Vendor-neutral, model-driven network management designed by users. https://web.archive.org/web/20250319152554/https://www.openconfig.net/, 2024. Accessed: 2025-05-01.

[200] Sabnani, K., and Dahbura, A. A protocol test generation procedure. *Computer Networks and ISDN systems* (1988).

[201] Sadowski, C., Aftandilian, E., Eagle, A., Miller-Cushon, L., and Jaspan, C. Lessons from building static analysis tools at google. *Communications of the ACM* (2018).

[202] Scholz, B., Jordan, H., Subotić, P., and Westmann, T. On fast large-scale program analysis in datalog. In *Proceedings of the 25th International Conference on Compiler Construction* (2016).

[203] Schuur, S. An introduction to NVIDIA Air. https://web.archive.org/web/20250129002635/https://developer.nvidia.com/blog/an-introduction-to-nvidia-air/, 2024. Accessed: 2025-05-01.

[204] Schwartz, E. J., Avgerinos, T., and Brumley, D. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *IEEE S&P* (2010).

[205] Schwartz, M. D., Delisle, N. M., and Begwani, V. S. Incremental compilation in magpie. *ACM SIGPlan Notices* (1984).

[206] Sen, K., Marinov, D., and Agha, G. CUTE: A concolic unit testing engine for C. In *ACM ESEC/FSE* (2005).

[207] Sharif, H., Abubakar, M., Gehani, A., and Zaffar, F. TRIMMER: application specialization for code debloating. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (2018).

[208] Sharma, P., and Yegneswaran, V. Prosper: Extracting protocol specifications using large language models. In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks* (2023).

[209] Sharma, R., Schkufza, E., Churchill, B., and Aiken, A. Data-driven equivalence checking. In *ACM OOPSLA* (2013).

[210] Shen, H., Pszeniczny, K., Lavaee, R., Kumar, S., Tallam, S., and Li, X. D. Propeller: A profile guided, relinking optimizer for warehouse-scale applications. In *ACM ASPLOS* (2023).

[211] Sherwood, R., Shi, J., Zhang, Y., Spring, N., Sundaresan, S., Bagga, J., Peddi, P., Kukkadapu, V., Shrivastava, R., Manikantan, K., et al. Netcastle: Network infrastructure testing at scale. In *USENIX NSDI* (2024).

[212] Shukla, A., Hudemann, K., Vági, Z., Hü gerich, L., Smaragdakis, G., Hecker, A., Schmid, S., and Feldmann, A. Fix with P6: Verifying programmable switches at runtime. In *IEEE INFOCOM* (2021).

[213] Shukla, A., Hudemann, K. N., Hecker, A., and Schmid, S. Runtime verification of P4 switches with reinforcement learning. In *Proceedings of the 2019 Workshop on Network Meets AI & ML* (2019).

[214] Singh, A., Ong, J., Agarwal, A., Anderson, G., Armistead, A., Bannon, R., Boving, S., Desai, G., Felderman, B., Germano, P., et al. Jupiter rising: A decade of clos topologies and centralized control in Google's datacenter network. In *ACM SIGCOMM* (2015).

[215] Sivaraman, A., Kim, C., Krishnamoorthy, R., Dixit, A., and Budiu, M. DC.p4: Programming the forwarding plane of a data-center switch. In *ACM SOSR* (2015).

[216] Sivaraman Kaushalram, A. *Designing fast and programmable routers*. PhD thesis, Massachusetts Institute of Technology, 2017.

[217] Sonchack, J., Loehr, D., Rexford, J., and Walker, D. Lucid: A language for control in the data plane. In *ACM SIGCOMM* (2021).

[218] Song, H. Protocol-oblivious forwarding: unleash the power of SDN through a future-proof forwarding plane. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking* (2013).

[219] Soni, H., Rifai, M., Kumar, P., Doenges, R., and Foster, N. Composing dataplane programs with μP4. In *ACM SIGCOMM* (2020).

[220] Stallman, R. M. *Using and porting the GNU compiler collection*. Free Software Foundation Boston, MA, USA, 1999.

[221] Stoenescu, R., Dumitrescu, D., Popovici, M., Negreanu, L., and Raiciu, C. Debugging P4 programs with Vera. In *ACM SIGCOMM* (2018).

[222] Stoenescu, R., Popovici, M., Negreanu, L., and Raiciu, C. Symnet: Scalable symbolic execution for modern networks. In *ACM SIGCOMM* (2016).

[223] Sudarshan, R., and Sommers, C. P4 as a single source of truth for SONiC DASH use cases on both softswitch and hardware. https://web.archive.org/web/20240919194035/https://opennetworking.org/wp-content/uploads/2022/05/Reshma-Sudarshan-Chris-Sommers-Final-Slide-Deck.pdf, 2022. Accessed: 2025-05-01.

[224] Tate, R., Stepp, M., Tatlock, Z., and Lerner, S. Equality saturation: A new approach to optimization. In *ACM POPL* (2009).

[225] Taylor, D. E., and Turner, J. S. Classbench: A packet classification benchmark. *IEEE/ACM transactions on networking* (2007).

[226] The Linux Foundation. DPDK: The data plane development kit. https://web.archive.org/web/20250319144003/https://www.dpdk.org/, 2010. Accessed: 2025-05-01.

[227] The Linux Foundation. middleblock.p4. https://github.com/sonic-net/sonic-pins/blob/3f52760f3bbaf2723bcfb2de5ca68a8a826273f1/sai_p4/instantiations/google/middleblock.p4, 2021. Accessed: 2025-05-01.

[228] The Linux Foundation. PINS: P4 integrated network stack. https://web.archive.org/web/20250304093609/https://opennetworking.org/pins/, 2022. Accessed: 2025-05-01.

[229] The Linux Foundation. SONiC: Software for open networking in the cloud. https://web.archive.org/web/20250320021526/https://sonicfoundation.dev/, 2022. Accessed: 2025-05-01.

[230] The Linux Foundation. Vector packet processing. https://github.com/FDio/vpp/, 2024. Accessed: 2025-05-01.

[231] The P4 Langage Consortium. The P4Runtime specification, version 1.3.0. https://web.archive.org/web/20231129181507/https://p4.org/p4-spec/p4runtime/v1.3.0/P4Runtime-Spec.html, 2020.

[232] The P4 Language Consortium. The reference P4 software switch. https://github.com/p4lang/behavioral-model, 2014. Accessed: 2025-05-01.

[233] The P4 Language Consortium. PTF: Packet testing framework. https://github.com/p4lang/ptf, 2015. Accessed: 2025-05-01.

[234] The P4 Language Consortium. $P4_{16}$ portable switch architecture (psa), version 1.1. https://web.archive.org/web/20240919101339/https://p4.org/p4-spec/docs/PSA-v1.1.0.html, 2018.

[235] The P4 Language Consortium. $P4_{16}$ portable nic architecture (pna), version 0.7. https://web.archive.org/web/20240914064108/https://p4.org/p4-spec/docs/PNA-v0.7.html, 2022.

[236] The P4 Language Consortium. The $P4_{16}$ language specification, version 1.2.4. https://web.archive.org/web/20240914072609/https://p4.org/p4-spec/docs/P4-16-v1.2.4.html, 2023.

[237] The P4 Language Consortium. The open p4 studio. https://github.com/p4lang/open-p4studio, 2025. Accessed: 2025-05-01.

[238] The P4 Language Consortium. P4MLIR - bringing MLIR to P4. https://github.com/p4lang/p4mlir, 2025. Accessed: 2025-05-01.

[239] The XLA Team. XLA – TensorFlow compiled. https://web.archive.org/web/20250303085610/https://developers.googleblog.com/en/xla-tensorflow-compiled/, 2017. Accessed: 2025-05-01.

[240] Thomas, B., Andersson, L., and Minei, I. LDP Specification. RFC 5036, 2007.

[241] Tian, B., Gao, J., Liu, M., Zhai, E., Chen, Y., Zhou, Y., Dai, L., Yan, F., Ma, M., Tang, M., et al. Aquila: a practically usable verification system for production-scale programmable data planes. In *ACM SIGCOMM* (2021).

[242] Tu, W., Ruffy, F., and Budiu, M. P4C-XDP: Programming the linux kernel forwarding plane using P4. In *Linux Plumbers Conference* (2018).

[243] Viswanathan, A., Rosen, E. C., and Callon, R. Multiprotocol Label Switching Architecture. RFC 3031, 2001.

[244] Wang, Q., Pan, M., Wang, S., Doenges, R., Beringer, L., and Appel, A. W. Foundational verification of stateful P4 packet processing. In *14th International Conference on Interactive Theorem Proving (ITP 2023)* (2023), Schloss-Dagstuhl-Leibniz Zentrum für Informatik.

[245] Wang, T., Yang, X., Antichi, G., Sivaraman, A., and Panda, A. Isolation mechanisms for high-speed packet-processing pipelines. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)* (2022).

[246] Wang, X., Zeldovich, N., Kaashoek, M. F., and Solar-Lezama, A. Towards optimization-safe systems: Analyzing the impact of undefined behavior. In *ACM SOSP* (2013).

[247] Wijnands, I., Hitchen, P., Leymann, N., Henderickx, W., Gulko, A., and Tantsura, J. Multipoint Label Distribution Protocol In-Band Signaling in a Virtual Routing and Forwarding (VRF) Table Context. RFC 7246, 2014.

[248] Wintermeyer, P., Apostolaki, M., Dietmüller, A., and Vanbever, L. P2GO: P4 profile-guided optimizations. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks* (2020).

[249] Würthinger, T., Wimmer, C., Humer, C., Wöss, A., Stadler, L., Seaton, C., Duboscq, G., Simon, D., and Grimmer, M. Practical partial evaluation for high-performance dynamic language runtimes. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2017).

[250] Xing, J., Hsu, K.-F., Kadosh, M., Lo, A., Piasetzky, Y., Krishnamurthy, A., and Chen, A. Runtime programmable switches. In *USENIX NSDI* (2022).

[251] Xing, J., Qiu, Y., Hsu, K.-F., Sui, S., Manaa, K., Shabtai, O., Piasetzky, Y., Kadosh, M., Krishnamurthy, A., Ng, T. E., et al. Unleashing SmartNIC packet processing performance in P4. In *ACM SIGCOMM* (2023).

[252] Xsight Labs. Xsight Labs announces X2 programmable SDN Ethernet switches. https://web.archive.org/web/20241102232425/https://ai-techpark.com/xsight-labs-announces-x2-programmable-sdn-ethernet-switches/, 2024. Accessed: 2025-05-01.

[253] Xu, Q., Wong, M. D., Wagle, T., Narayana, S., and Sivaraman, A. Synthesizing safe and efficient kernel extensions for packet processing. In *ACM SIGCOMM* (2021).

[254] Yang, H., and Lam, S. S. Real-time verification of network properties using atomic predicates. *IEEE/ACM Transactions on Networking* (2015).

[255] Yang, M., Baban, A., Kugel, V., Libby, J., Mackie, S., Kananda, S. S. R., Wu, C.-H., and Ghobadi, M. Using Trio – Juniper networks' programmable chipset – for emerging in-network applications. In *ACM SIGCOMM* (2022).

[256] Yang, X., Chen, Y., Eide, E., and Regehr, J. Finding and understanding bugs in C compilers. In *ACM PLDI* (2011).

[257] Yaseen, N., Yu, L., Stanford, C., Beckett, R., and Liu, V. FP4: Line-rate greybox fuzz testing for p4 switches. *arXiv preprint arXiv:2207.13147* (2022).

[258] Yu, L., Sonchack, J., and Liu, V. Mantis: Reactive programmable switches. In *ACM SIGCOMM* (2020).

[259] Yu, Z., Su, B., Bai, W., Raindel, S., Braverman, V., and Jin, X. Understanding the micro-behaviors of hardware offloaded network stacks with lumina. In *ACM SIGCOMM* (2023).

[260] Zalewski, M. american fuzzy lop. https://github.com/google/AFL, 2013. Accessed: 2025-05-01.

[261] Zeller, A., Gopinath, R., Böhme, M., Fraser, G., and Holler, C. Fuzzing with grammars. In *The Fuzzing Book*. CISPA Helmholtz Center for Information Security, 2024. Accessed: 2025-05-01.

[262] Zeng, H., Kazemian, P., Varghese, G., and McKeown, N. Automatic test packet generation. In *ACM CoNEXT* (2012).

[263] Zhao, Y., Wang, H., Lin, X., Yu, T., and Qian, C. Pronto: Efficient test packet generation for dynamic network data planes. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)* (2017).

[264] Zheng, N., Liu, M., Zhai, E., Liu, H. H., Li, Y., Yang, K., Liu, X., and Jin, X. Meissa: Scalable network testing for programmable data planes. In *ACM SIGCOMM* (2022).

[265] Zuck, L., Pnueli, A., Fang, Y., and Goldberg, B. VOC: A translation validator for optimizing compilers. *Electronic notes in theoretical computer science* (2002).