P4Fuzz: Compiler Fuzzer for Dependable Programmable Dataplanes

Andrei Alexandru Agape
Mădălin Claudiu Dănceanu
René Rydhof Hansen
Aalborg University
Aalborg, Denmark

Stefan Schmid
University of Vienna
Vienna, Austria

ABSTRACT

Emerging software-defined networks and programmable dataplanes promise to render communication networks more dependable, overcoming today's manual and error-prone approach to operate networks. Indeed, programmable dataplanes such as P4 provide great opportunities for improving network performance and developing innovative security features, by allowing programmers to reconfigure and tailor switches towards their needs. However, extending programmability to the dataplane also introduces new threat models. In this paper, using a systematic security analysis, we identify a particularly worrisome vulnerability: the automated program compilers which lie at the core of programmable dataplanes. The dataplane compilers introduce a risk of persistent threats which are covert and hard to detect, and may be exploited for large-scale attacks, affecting many devices. Our main contribution is P4Fuzz, a compiler fuzzer to find bugs and vulnerabilities in P4 compilers, in an efficient and automated manner. We discuss the challenges involved in designing such a compiler fuzzer for P4, present our fuzzing and taming algorithms, and report on experiments with our prototype implementation, considering the standard compilers of BMv2, eBPF, and NetFPGA. Our experiments confirm that P4Fuzz is able to generate and test the validity of dozens of P4 programs per minute. Using P4Fuzz, we also successfully found several bugs which have been acknowledged and fixed by the community.

CCS CONCEPTS

• **Networks** → *Programming interfaces*; *Protocol correctness*; **Bridges and switches**; • **Security and privacy** → *Software security engineering*.

KEYWORDS

software defined networking, p4 compiler, fuzzing

Research supported by the Vienna Science and Technology Fund (WWTF) project ICT19-045.

ICDCN '21, January 5–8, 2021, Nara, Japan

ACM ISBN 978-1-4503-8933-4/21/01…$15.00
https://doi.org/10.1145/3427796.3427798

1 INTRODUCTION

Communication networks have become a critical infrastructure of our digital society. The resulting stringent dependability requirements however stand in stark contrast to today's manual and error-prone approach to manage and operate networks. Software-Defined Networks (SDNs), and its de facto standard protocol, OpenFlow, have recently emerged as an attractive alternative, enabling programmability and automation.

SDNs separate the two core functions of the network-processing elements, the control plane and the data plane, by outsourcing and consolidating the control over switches and routers to a logically centralized software. The latter can manipulate the network configuration via an API (e.g., using OpenFlow). This decoupling allows a variety of control plane implementations for each dataplane, and supports fast innovations in the control plane, at the speed of software development, and independent of the life cycles and restrictions of the underlying hardware.

However, despite the additional flexibility brought by separating these functions, SDN still assumes that the behavior of the network data plane is fixed. This is a significant impediment to innovation. For example, the deployment of new network security features and protocols, such as the tunneling protocol VxLAN which is now widely used for data center network virtualization, can take several years: in the case of VxLAN, there were 4 years between the initial proposal and its commercial availability in high-speed devices [3].

The advent of programmable dataplanes and P4 [2] overcomes this problem by unlocking the next stage of flexibility: after OpenFlow introduced great flexibilities in how switches from different vendors can be controlled in a unified manner, P4 supports *reconfigurability in the dataplane*. P4 makes the deployment of new protocols simpler, e.g., a VxLAN implementation in P4 requires 175 lines of code and can be deployed via a software upgrade.

More specifically, P4 allows to change the way switches process packets at runtime. At the heart of these emerging programmable networks lies an open, flexible and silicon-independent API, which unties switches from the specific network protocol ("*protocol independence*"), and which enables packet processing functionality to be programmed independently of the specifics of the underlying hardware ("*target independence*"). In particular, P4 not only provides a high-level programming language which can be compiled against many different types of execution machines (called "P4 targets", which have a P4 compiler back-end), but also offers a common so-called *P4 Runtime API* that allows an operator to change and immediately start using new forwarding tables, without restarting the API or the control plane.

This paper is motivated by the observation that programmable dataplanes do not only enable more flexible communication networks, interesting new use cases, and an unprecedented performance, but also new dependability challenges. In particular, we observe that the introduction of a domain-specific language, P4, and the concomitant development tools, especially the *compiler* which is a key asset on the P4 platform, also opens the doors to new bugs [8] and security vulnerabilities well-known from the software development community. This includes some of the most pernicious bugs, namely those found in compilers, leading to fundamental questions about how much trust can/should be put into the development tools and infrastructure [26].

Despite being a fundamental and constituent part of programmable dataplanes, the compilation process is for most developers unknown. Indeed, existing tools usually focus either on the packet generation (e.g., p4pktgen [13]) or the language and programs (e.g., ASSERT-P4 [8]), by checking the execution paths and validating the intended program behavior. However, the compiler does not only increase the attack surface, it may also be a particularly attractive target for adversaries, due to the large impact an attack can have: an (often hard-to-detect) vulnerability can effect many devices (i.e., all devices based on this code).

Contributions. This paper studies the novel security challenges introduced by programmable dataplanes. Charting a systematic taxonomy of the attack surface, we identify the compiler as a novel threat to the network's dependability.

Our main contribution is an automated *compiler fuzzer* framework for P4, called P4Fuzz which can enhance the dependability of emerging programmable networks. P4Fuzz is able to stress the P4 compiler and find bugs hidden deep in the implementation, by generating syntactically and semantically valid P4 programs, using novel algorithms. P4Fuzz follows a blackbox and generation-based approach and incorporates taming techniques which, after finding bugs, use a clustering algorithm to identify different categories of vulnerabilities. Generally, P4Fuzz is designed such that it can support multiple architectures such as BMv2, eBPF, or NetFPGA, and is easily extendable. Our experiments, using our prototype implementation, show that P4Fuzz generates up to 100 valid programs per minute and is able to test validity for approximately 20 programs per minute. Furthermore, during a feasibility study, in relatively short time, P4Fuzz discovered and reported four bugs, out of which two have already been fixed on the official repository of P4C, the standard compiler for P4. We also report on our experiments and experience with a NetFPGA board and compiler, for which further bugs were found.

To facilitate further research in this area and explore even more effective fuzzing strategies, we will make our tool, P4Fuzz, publicly available to the research community, together with this paper, as open source.

Novelty and Related Work. While fuzzing is a popular technique used in security to explore corner cases and find bugs that may prove to be exploitable (see e.g., [17, 25] for case studies in the context of SDN), apart from some successful examples (e.g., Csmith [27] for the C language), less is known about *fuzzers for compilers*: potentially an effective tool to find weaknesses and bugs *in the compiler*. We in this paper hence initiate the study of compiler fuzzers for P4.

More generally, there is much prior work on the design, performance and use cases for programmable dataplanes and P4 [2, 10, 14, 20], as well as the security [7, 9, 11, 12, 16, 18, 19, 24] and verification [1, 4] of software-defined networks. While some of the latter studies also apply to programmable dataplanes, programmable dataplanes introduce additional challenges, as we also show in this paper.

Dargahi et al. [6] presented a first analysis of the attack surface of programmable dataplanes and P4. Freire et al. proposed a verification approach to prevent bugs, through the use of assertions and symbolic execution, and presented *ASSERT-P4* [8] accordingly. Nötzli et al. proposed *p4pktgen* [13], a tool for automated test case generation for P4 programs, allowing to validate if P4 programs act as intended on a given device.

Our work is the first to consider issues related to the P4 compiler. However, very recently, Ruffy et al. [15] followed up on our work (started 2017 with a student thesis) and presented Gauntlet, which provides several domain-specific techniques to find bugs in programmable packet processing compilers.

Organization. The reminder of this paper is organized as follows. We provide a short background on programmable dataplanes and P4 in §2, conduct a STRIDE analysis of programmable dataplanes in §3, and introduce a threat model in §4. We describe our tool in §5, and evaluate it in §6–§8. We conclude in §9.

2 BACKGROUND

This section presents the background on the P4 language and compiler required in order to understand the scope, design and implementation of P4Fuzz.

P4 Language. P4 is a domain-specific language that provides a number of constructs optimized around network data forwarding. The version used within this project is P4-16 [23]. The *target independent* goal specifies that the language can be compiled against many different types of execution machines. These machines are known as "P4 targets", and each P4 target has a P4 compiler backend. The P4 compiler maps the P4 source code into the target switch model. The *protocol independent* goal states that the P4 programs are the ones that specify how a switch processes packets. The P4 language has no support for protocols; it is the P4 programmer that describes the header formats and field names. Once the programmer does so, they are interpreted and processed by the compiled program on the target device. The *reconfigurability* allows network administrators to change how switches process packets even after they are deployed. This design is made possible by the protocol independence and the abstract language model.

A P4 program can generally be structured into three main parts: data declaration, parse program and control flow program. The data declaration defines the data types for header and metadata bus. The parse program section defines the packet parser and de-parser as a finite state automaton using states and transitions. The control flow program section defines how packets are forwarded and processed using actions such as noop, drop, modify, push, pop etc.

P4 Compiler. P4C is the compiler for the P4 programming language that supports both the P4-14 and P4-16 versions[1]. P4C comes with a standard front-end and mid-end, which can be combined with

[1] https://github.com/p4lang/p4c

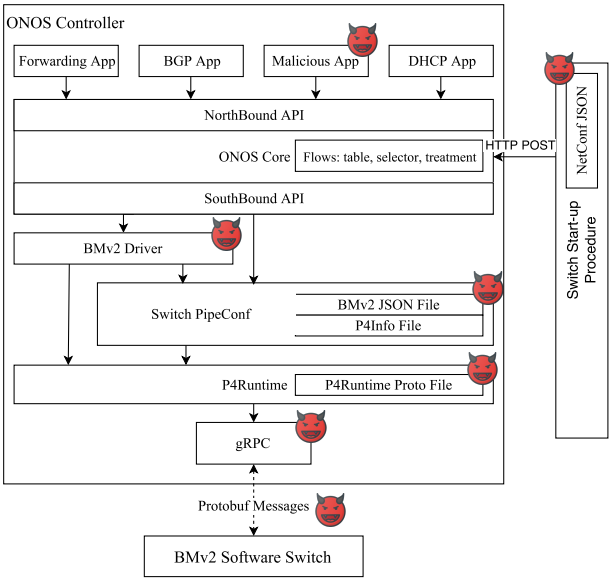

Figure 1: Overview of P4 components, assets and attack points in ONOS Controller

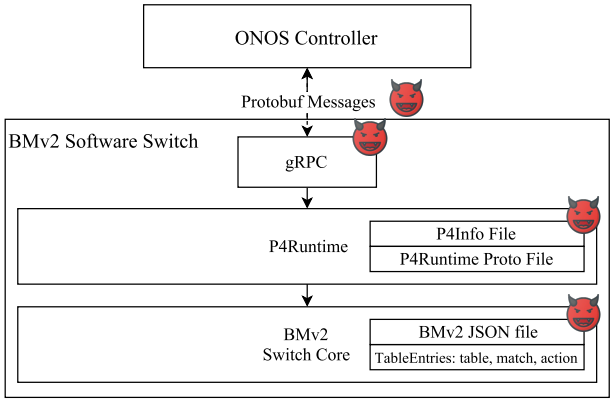

Figure 2: Overview of P4 components, assets and attack points in BMv2 Switch

a target-specific back-end in order to obtain a complete compiler. While the front-end deals with the semantic checks, the mid-end performs optimization, and the back-end outputs target-specific code. This separation makes it possible to easily integrate new back-ends. Multiple back-ends have already been developed, generating code for ASICs, NICs, FPGAs, software switches and other targets.

The main components of the compiler are the parser (either P4-14 or P4-16), an Intermediate Representation (IR) converter that enables backwards compatibility with P4-14 language, a fixed front-end component, customizable mid-ends, and back-ends provided by the vendor for specific targets.

The front-end is fixed, the back-end target specific. The P4C provides passes which can be used to implement various mid-ends, but new passes (or even a whole mid-end) can also be easily added.

3 ANALYSIS OF SECURITY CHALLENGES

We start by conducting a systematic study of the security challenges introduced by programmable dataplanes, from which we will derive our threat model. We first provide a brief overview of the main *assets* of a (typical) P4 SDN environment and then perform a STRIDE analysis of the *attack surface* presented by such an environment.

3.1 P4 Assets

In order to perform our security and vulnerability analysis of P4, we first have to identify and prioritise the potential targets, i.e., the *assets* of the P4 platform. An asset in this context is any data, device, or other component that supports information related P4 activities. For convenience we have grouped the P4 assets of interest into four general categories: control plane assets, channel plane assets, data plane assets, and the P4 compiler, see Fig. 1 and Fig. 2 for an overview. In the following we briefly describe the categories and their concomitant assets.

Control Plane. The control plane, as a whole, is concerned with the routing process, including ongoing management and setup of the process.

- **Applications.** These are the primary assets in the control plane. An application is (potentially third-party) software designed to manage and perform specific actions within an SDN. Applications are one of the great attractions of SDN, since they offer deep and wide-ranging flexibility and can be installed/removed as and when needed for specific tasks. They are also one of the obvious targets of attack.
- **P4Info.** This is the result of compiling the P4 program. This asset contains critical information such as tables, meters, counter, etc. as well as assigned IDs, enabling communication between controller and switch. This information is also used by both the P4 controller to setup the forwarding configuration and the P4 runtime (denoted *P4Runtime* in Fig. 1) for translating IDs into objects.
- **P4DeviceConfig.** The result of compiling the P4 control program to the target switch using the appropriate back-end compiler, e.g., *bmv2JSON* is the output of the *bmv2* back-end compiler. This asset is used by the controller, together with the *P4Info* asset, to set up the forwarding plane configuration.
- **SwitchPipeConf.** This is a controller application that defines the switch pipeline by using the *P4Info* and *P4DeviceConfig* assets to set up a mapping between P4 and platform specific objects.
- **Switch driver.** The switch driver is a switch-specific application running on the controller and typically developed by the switch vendor. It provides an interface for adding and removing target specific table entries using the mapping set up by *SwitchPipeConf*. As an example, in ONOS, it maps ONOS flows to P4 table entries.
- **P4Runtime agent.** This is an application on the controller that serializes the P4 objects and the forwarding configuration to *Protobuf* and calls the intended RPC methods.
- **P4Runtime.proto.** The *P4Runtime* protocol specification. It defines the RPC methods and messages that can be used

between controller and switch. The protocol provides different RPC methods such as *SetForwardingPipelingConfig* and *StreamChanel*. On top of these, the protocol provides a multitude of message types. The *P4Runtime.proto* is present in both controller and switch.

- **gRPC.** The remote procedure call (RPC) system developed at Google. It uses HTTP/2 for transport, *Protocol Buffers* as the interface description language, and provides features such as authentication, bidirectional streaming and flow control, cancellation and timeouts etc.

From the above, it should be clear that the control plane contains a wide variety of assets, ranging from applications to simple files, resulting in a wide attack surface.

Channel Plane. The channel plane is concerned with inter-component communication (through channels), mainly between controller and switch. For our use, this plane comprises only a single asset:

- **Protobuf messages.** These are the messages exchanged between the controller and switch, serialized using Protocol Buffers.

Data Plane. The data plane is concerned with the actual forwarding of data (packets) and shares (some) asset types with the control plane.

- **gRPC.** This is similar to the *gRPC* asset on the controller, with the difference that if the switch *gRPC* is not available it is only the switch that cannot be controlled anymore, while if the controller gRPC is not available, the whole network becomes uncontrollable.
- **P4Runtime.proto.** The P4Runtime protocol specification that defines the RPC methods and messages that can be used between controller and switch. Similar to the file present on the controller.
- **Parser/de-parser.** These are defined in the P4 program as a deterministic finite automaton (DFA) using states and transitions.
- **Flow tables.** These are the tables used to define exactly how the packets are forwarded and processed.

Compiler. The compiler is a key asset on the P4 platform: this is what enables the rapid development of applications that can change major aspects of a network. However, as always, programmability also comes with many potential risks, requiring good (security aware) programming practices. For our purposes, we consider different parts of the compiler as separate assets.

- **Front-end compiler.** This is a target-independent and standard part of the compiler that deals with the semantics checks, and that can be combined with a target-specific back-end to create a complete P4 compiler. We here take the front-end compiler to include various optimization passes, performed before the generated *Intermediate Representation* (IR) is sent to the back-end compiler.
- **Converter (P4-14 to P4-16).** This part of the compiler enables backward compatibility with the P4-14 version of the P4 language. It parses the P4-14 into version 1 of the Intermediate Language before it is converted to the *Intermediate Representation* accepted by the front-end.

Table 1: STRIDE analysis

Threat	Property violated	Definition	Example
Spoofing	Authentication	Impersonating something or someone else	Pretends to be another switch in the network
			Pretends to be the controller
			Pretends to be the network controller administrator
Tampering	Integrity	Modifying data or code	Intercept and modify protobuf messages
			Take control of gRPC server and modify the protobuf messages
Repudiation	Non-repudiation	Claiming to have not performed an action	A switch that does not follow the controller instructions
			A controller claiming that a switch has not connected to it
Information Disclosure	Confidentiality	Exposing information to someone not authorized to see it	Read device tables: controller flows, switch tables
			Read protobuf messages
Denial of Service	Availability	Deny or degrade service to users	Crashing the P4Runtime gRPC service
			Flooding the switch-controller channel
			Modify and invalidate protobuf messages
			Intercept and deny arrival of the packets to the intended device
Elevation of privilege	Authorization	Gain capabilities without proper authorization	A switch changing information in the controller
			Configure a switch and decide how the traffic is handled

- **Back-end compiler.** This is the main target-specific component of the compiler, usually developed by the vendor of the network components.

While it is possible to go deeper and identify more specific assets, we conclude our survey of the P4 assets here: for an initial mapping of major (potential) security vulnerabilities, we have found that the above lists provide a good starting point.

3.2 STRIDE Analysis and Attack Surface

In the following we present a STRIDE analysis of the P4 platform, based on the assets identified in the previous section. STRIDE is a well-known model for categorising (potential) IT-security threats and a useful tool for structuring threat-analysis of IT systems. The name is a mnemonic derived from the threat categories comprising the model: *Spoofing*, *Tampering*, *Repudiation*, *Information disclosure*, *Denial-of-service*, and *Elevation of privilege*. The threat categories cover most, if not all, the "classic" threats/attacks that have been oberseved and reported in the literature.

Since STRIDE analysis is fairly standard and well-known, we will not discuss it in further detail here. We illustrate the general methodology by briefly discussing an excerpt of the STRIDE analysis for the *P4Runtime* component (see Table 1 for an overview).

Spoofing. A potential spoofing attack would be an attacker (successfully) masquerading as a (different) switch in the network. This would allow the attacker to elicit information from the controller, such as de-/parser configuration, pipeline configuration, forwarding table entries, and how table-miss flows are handled.

Tampering. If an attacker can modify, i.e., tamper with, protobuf messages, it can violate confidentiality, integrity, and availability properties of the network. An attacker that can take control of the switch *gRPC* or the controller *gRPC*, can modify the sent and

received protobuf messages, thus controlling the switch or the entire network.

Repudiation. In a *repudiation* attack, an attacker can make the switch refuse configurations from a controller and claim they were not received, thus making the switch uncontrollable. An attacker can make the controller refuse connections from switches that try to connect to it and claim that no connections were instantiated, rendering the switch unable to handle traffic. The availability principle is therefore violated.

Information disclosure. An attacker with a presence on the network may be able to pick up information that is either sent in the clear, such as unencrypted protobuf messages, messages picked up directly from the control plane, or even exploiting specific timing properties for an advanced timing-attack on the controller or the switches.

Denial-of-service. A *denial-of-service attack* may crash the *gRPC* service, making the communication between the switch and the controller unavailable, potentially wrecking havoc in the network.

Elevation of privilege. As part of an *elevation of privilege* attack, an attacker can write malicious applications, and based on the controller configuration, allow the application to read, modify or deny data and services. It may also modify the forwarding tables.

Here, the attack surface is composed of the following resources: (1) the data in the system and messages exchanged, (2) the methods for processing applications, e.g., request/response methods, (3) the communication channels, e.g., HTTP, TCP.

For example, an attacker that has access to the switch configuration can craft protobuf messages to decide upon how the parser, deparser, pipeline and flow table entries look like. In terms of resources, the entry point is *gRPC* while the data is represented by the Protobuf messages sent. The methods that can be used are the ones defined in the '.proto' file and the communication channel is the gRPC. An attacker that has access to the SDN controller machine can try to access the administration panel. As an example, ONOS controller provides a graphical interface for administration on port 8181 with default credentials onos/rocks. In this case, the entry point is the ONOS login interface while the data is represented by the username/password combination. The method used is a POST request and the communication channel is HTTP.

4 THREAT MODEL

Motivated by our security analysis of the attack surface of programmable dataplanes, we consider a novel threat model and study an adversary which exploits the compiler of a programmable dataplane. The threat model is inspired by similar models arising in other contexts requiring "trust in trust" [26].

Our attacker can come in different flavors. A first kind of attacker may aim to introduce malicious code to the compiler directly, e.g., a backdoor. In large development teams, depending on the policy how to accept push requests (by a company insider or member of the open-source community), a "malicious" push to the back end may be easily overlooked.

A second kind of attacker however may be more problematic: an attacker may simply try to find bugs and subsequently take advantage of them. For example, a simple calculation bug may be exploited to produce wrong MAC addresses and hence to compromise ACLs or to forward traffic wrongly and, e.g., exfiltrate information. Being able to crash or subvert a compiler may introduce further vulnerabilities, especially in virtualized environments, e.g., the recent attack to compromise clouds via the virtualized dataplane [25]. This attack may require only very limited amounts of resources and skills, e.g., a simple fuzzing approach may be sufficient. The attack does generally also not require any privileged access to software, hardware, or distribution channels.

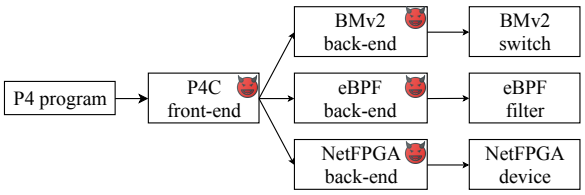

Figure 3: Attack surface of P4 compiler.

In particular, the attacker can consider attacks both on the front end compiler (for which finding bugs is harder but the impact of the attack is larger as it affects all devices) and on the back end compiler (where it may be easier to find mistakes as these compilers are more specific and may also contain non-implemented parts). See Fig. 3 for an illustration of the attack surface of the P4 compiler.

5 THE P4FUZZ COMPILER FUZZER

This section presents our compiler fuzzer, P4Fuzz, and discusses some of the design and implementation challenges faced. Designing a compiler fuzzer for P4 is a non-trivial task as the generated P4 programs should be as general and as specific as possible: the programs should be valid, and allow to explore a large spectrum of the possible programs; but at the same time, also identify and focus on the most critical and interesting programs where vulnerabilities are likely. We discuss different challenges related to, e.g., generating semantically valid programs, tailoring the fuzzer for different architectures, dealing with nested types and recursive statements, testing programs behavior, or sorting faulty test cases based on error relevance. We then present design choices and algorithms accordingly.

5.1 Fuzzer Design

The P4Fuzz fuzzer is designed as a framework comprising separate components, including the actual fuzzer, a test case generator, a packet handler, as well as a *taming engine* (which is not specific to the fuzzer). Our fuzzer is a so-called smart, black-box, generation-based fuzzer [21]. In other words, our fuzzer knows and takes advantage of the structure of input to the compiler, i.e., P4 programs; it does not have any knowledge of the P4 compiler's inner workings, e.g., through static and/or symbolic analysis; and the fuzzer generates input (P4 programs) from scratch. Fig. 4 shows an overview of the tool and the associated activity flow.

The "smart" fuzzer approach was taken, since the alternative, generating input strings randomly and feeding them to the P4 compiler, would rarely yield a valid program and would thus mainly be fuzzing the P4 compiler's parser. Even though the black-box

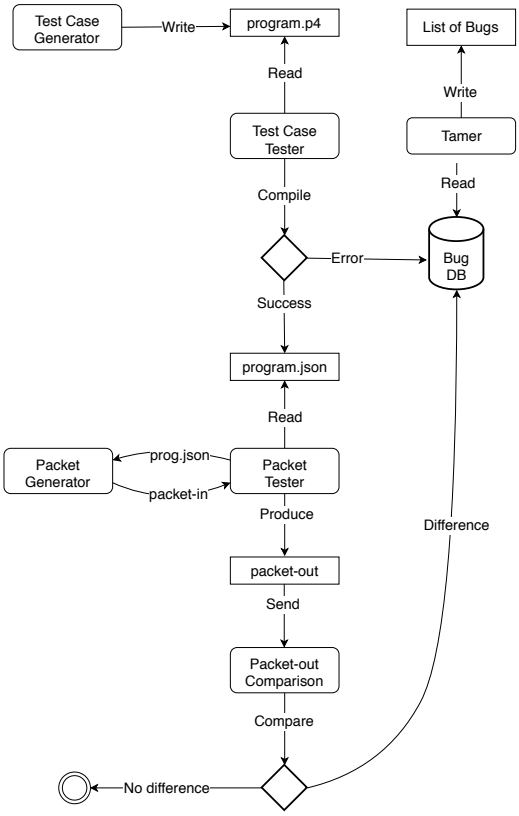

Figure 4: Flow of P4Fuzz tool.

technique most likely covers a smaller part of the targeted compilers code, we chose this approach due to it requiring less effort and no specialised setup or specialised configuration for the user. Furthermore, information about a particular P4 compiler or target platform may be added and exploited at a later stage. Using a generation based approach for the fuzzer, rather than a mutation based one, is also due to the fact that mutating a valid program using one or more letters at a time would most likely throw syntax errors as unexpected tokens and invalid keywords are produced. Using mutation on known-valid keywords, instead of individual characters, would likely still generate (too many) programs with syntax errors and it would require more time and effort. Thus, a generation based fuzzer is more likely to be able to generate tests for later stages of the compiler

5.2 P4Fuzz Components and Activity Flow

The first step of the fuzzing process starts with the *Test Case Generator* module, which randomly generates P4 programs. After a few P4 programs have been generated, the *Test Case Tester* can begin to select and process the generated test programs; to speed up execution, these two modules are running in parallel. The *Test Case Tester* reads a (previously generated) P4 program, compiles it, and checks the compiler output for errors and warnings. Details about programs that give rise to compilation errors are saved into a *Database of Bugs*. The result of a successfully compiled P4 program, in the form of a `program.json` file, is saved and tested for run-time errors. In order to do so, the JSON file is deployed to the switch together with specially crafted packets. The *Packet Generator* (here we use the p4pktgen tool) crafts packets based on the target `program.json` file deployed on the switch. The *Packet Tester* module then reads each of these packets as packet-in and outputs the packet-out. In the final stage of the fuzzer, the *Packet-out Comparison* module checks if there are any differences between the packet-outs of distinct switches, i.e., Simple Switch or eBPF. If there were any differences between distinct architectures for the same P4 program and the same packet-in, the test case is marked as a possible bug and added to the database. Otherwise the process ends for that test case, and another one is selected from the test case pool. The *Tamer* module reads bugs saved to the database, sorts them using a distance function such that the most interesting and 'unique' ones are ranked higher, and prints them out to the user.

5.3 Design Challenges and Solutions

In the following we discuss some of the specific design challenges and corresponding solutions in more detail.

Generating Syntactically Valid P4 Programs. Obviously, the test programs have to be syntactically correct in order for the compiler to accept them. This challenge was addressed in the P4Fuzz design by systematically translating the grammar available in the specifications of the P4 language [22] into rules for generating nodes that can be used in an abstract syntax tree. Each node is then responsible for emitting code according to the P4 grammar rules during the code generation process.

In order to enable users to adapt and specialise P4Fuzz to specific needs, it is possible to change the probability with which a particular production rule (from the grammar) is selected, i.e., not all the available productions have the same chance of being chosen. This allows for tweaking the fuzzer to generate specific types of programs more often and by changing the probabilities, the user can change the focus of the fuzzer on different components of the P4 language.

Finally, since the number of generated cases per unit of time is an important aspect of a fuzzer, multiple instances of the test case generator can run in parallel as the processes do not interfere, thus enabling the use of multiple cores available on the machine. Fig. 5 illustrates this aspect by showing how two generators save different test cases using different test case numbers.

Generating Semantically Valid P4 Programs. Test programs must also be semantically valid, which is much harder to enforce than syntactical validity: some (semantic) rules are rather complex and may not be well documented (if at all). A further challenge is the dependency between semantic rules and the context of each instruction generated by the fuzzer, e.g., a variable must be declared before it is referenced. Hence, a P4 compiler fuzzer must be able to track the context of the program while generating it and enforce the semantic rules accordingly in order to generate valid P4 programs.

For P4Fuzz, since there is no formalization of the semantics of P4 available, we first extracted the semantic rules for each of the supported grammar rules, by reading through the P4 language specification [22] and implementing these rules as *filters*. Filters in P4Fuzz are a way of specifying requirements that have to be met

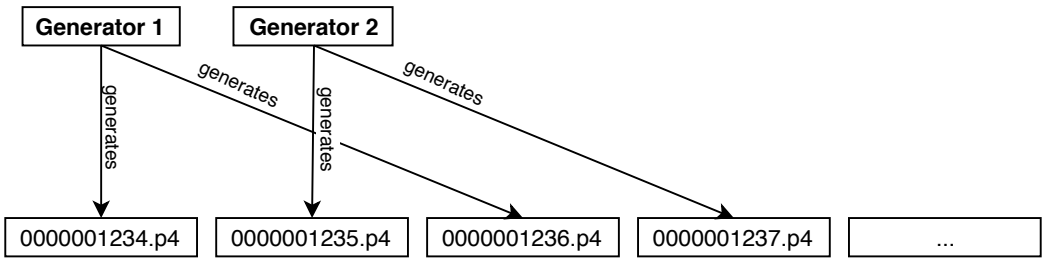

Figure 5: The Test Case Generator module can run as multiple processes each generating a different test case

in order for a production to be selected or not. As an example, the type nesting rules described in the language specifications, disallow other headers or structs to be used inside header declarations. Enforcing this semantic rule is achieved through a filter that does not allow a production for a header or struct to be selected when a header is declared.

Additionally, at all points during the program generation, P4Fuzz tracks the available user defined types as well as the available variables, their types and if they have been initialized or not. This tracking is implemented by recording the variable and data type declarations in a stack of scopes emulating the run-time stack. The information tracked during program generation can be extended to also include non-functional and/or high-level semantic properties, e.g., unreachable code and self-contradictory/meaningless conditionals, to verify that the compiler handles such cases correctly.

Testing Behavior of P4 Programs. In order to find errors in the compiled P4 programs, the fuzzer must be able to test the compiled programs (or compare them to the output of a different compiler). Verifying the correctness of randomly generated programs is challenging because a fuzzer has no means of deciding which behavior is intended and which is a mistake. Furthermore, the P4 language is relatively young and few advanced tools for analyzing/testing P4 programs have been developed yet. In P4Fuzz, testing the generated P4 programs behavior is done using *differential testing*, a technique in which the test case (a P4 program in this case) is compiled using multiple compilers and/or back-ends. If the output of the different compilers on the same source code is different, the program is marked as potentially buggy.

Sorting Faulty Test Cases based on Error Relevance. When testing a P4 compiler on randomly generated test programs, the fuzzer finds many potential errors, requiring further manual test and analysis which can be very time consuming. In order to facilitate this task, the fuzzer should *rank* the potential errors based on their novelty and importance (relative to the potential errors found); this is also called *taming* the fuzzer output [5].

Taming can greatly improve the efficiency of the bug discovery process as it can help users select the most interesting cases first. In P4Fuzz we tackle this problem by grouping the errors that are very similar (trigger the same bug) and thus creating clusters for each type of error. In this way, the person that manually investigates the interesting test cases can cover more error types with less effort. We discuss taming in more detail below.

Making it Work Across Different Architectures. The P4 language is target independent which means that P4 programs can be compiled and run on many different appliances, each of them having different architectures, in a modular way. Thus, the P4 compiler fuzzer must also be *target independent* and generate only P4 programs that can be compiled for specific back ends, i.e., architectures.

The solution chosen in P4Fuzz is to have different code generator wrappers based on the target specified (e.g. BMv2 or eBPF). These wrappers include the different code skeletons, mandatory includes, headers and controls, together with the limitations of the target for which it generates the code.

Handling Recursion in the P4 Syntax. The P4 language allows data types and expressions to be nested, giving rise to recursion in the grammar underlying the P4 syntax. Consequently, such recursion must be controlled and correctly handled when generating random P4 programs. In P4Fuzz we address this challenge by keeping track of the current depth of the recursion for each specific rule implementation and increase it when the same production rule is used. In case the new depth level exceeds a specific maximum, the production rule responsible for this is no longer selected.

5.4 Taming

Once the fuzzing phase is over and all the potential bugs have been saved to the bug database, it is left to inspect all the individual (potential) bug reports and check whether there is a genuine bug, or if it is a false positive. Inspecting hundreds or thousands of reports produced by randomly generated test cases can be tedious work and may produce unreliable results, especially if the same bug is reported several times [5]. To alleviate this problem, P4Fuzz uses the concept of *taming*: taming consists of ordering the (potential) bugs reported in the earlier stages, such that the (relatively most) diverse and interesting test cases are ranked higher [5]. In this way the compiler developer can easily identify a larger number of bugs without spending time on duplicates. Inspired by [5] we try to find an approximate solution to the taming problem using two approaches: *(1)* Furthest point first (FPF) sorting using Levenshtein distance between each pair of errors; *(2)* Using k-medoids clustering based on a distance function which measures the number of different tokens/words between the two error messages.

Below we describe these approaches in more detail, including the challenges and outcomes for each approach.

Another recommendation seen in previous work, is to *reduce* the programs that produce errors, thus minimizing the source code by deleting unused statements. Such tools exist for C and JavaScript

but not for P4 and we leave it for future work to develop such a tool.

5.4.1 FPF on Levenshtein distance. The Levenshtein distance[2] is commonly used to describe how different two given strings are by comparing their characters. This is one of the distance functions suggested by prior work and the one that produced successful results in previous evaluations, although it is still not fully understood [5].

Based on our results, we were able to identify two main problems with this approach. Firstly, the Levenshtein distance is not suitable for calculating the distance between compiler outputs, especially when lines of code or variable names are included in the error text, since the Levenshtein distance is based on characters and is unable to differentiate between variable names and error description. We conjecture that it may be possible to overcome this problem by adapting the Levenshtein distance to work on tokens (formed by the compiler output) instead of characters. Secondly, the time complexity of the algorithm, $O(len(string1) \cdot len(string2))$, is too high, making it impossible to run the algorithm on all the test cases discovered. To compare only two test cases that have over 1000 characters, the algorithm has to execute over one million operations.

5.4.2 Token-Based Distance for k-medoids. Inspired by [5] and the FPF on Levenshtein distance, we defined a distance function based on tokenized error messages; the pseudo-code is shown in Listing 1.

```
int tokens_distance(s1, s2)
{
    int init_dist, dist;

    s1_tokens[] = s1.tokenize(' ')
    s2_tokens[] = s2.tokenize(' ')

    init_dist = length(s1_tokens + s2_tokens)
    dist = length(s1_tokens + s2_tokens)

    for iterator i in s1_tokens {
        for iterator j in s2_tokens {
            if j == i {
                dist = dist - 2
                j = s2_tokens.remove(j)
                break
            } else { j++ }
        }
    }
    return (dist / init_dist) * 100
}
```

Listing 1: Token-based distance function

Firstly, the test cases were split into tokens based on the space character (each word became a token). The *initial_distance* and *distance* are set to the sum of the two strings. Then, for each token in the first string, we checked if it exists in the second string. If it does, we delete the token from the second string and decrease the *dist* by 2. Finally, the distance between two strings is stored in the *dist* variable.

While this approach has a similar complexity as the Levenshtein distance (the number of tokens is, in the worst case, the same as the number of characters in a string, i.e., when all words in the string contain only one character), in practice it is faster. Another advantage is that it handles better the cases where the compiler output contains large parts of randomly generated code, since long variable names for example are considered as only one token regardless of the length (in comparison with Levenshtein that considers each character separately).

[2]https://en.wikipedia.org/wiki/Levenshtein_distance (accessed 5 December 2019).

6 PERFORMANCE EVALUATION

We evaluate P4Fuzz in three stages. First, in this section, we evaluate the performance. In the next two sections, we will report on the new compiler bugs we have found using the tool, and report on our experiments with a NetFPGA board. The following experiments were conducted using the latest version of the tool on a data set of 10k's test cases. The environment used for running P4Fuzz was a virtual machine installed on one of the physical rack servers at our university (Intel Xeon E5420 @ 2.50GHz).

In the following we analyze and discuss performance of the fuzzers' components.

Generating Test Cases. Test case generation is one of the key aspects of P4Fuzz. The performance of the fuzzer is directly affected by the speed at which it can produce programs. In Fig. 6, the average time required by the test case generator module to produce a single program is represented on the vertical axis while the size in terms of tokens of the generated case is shown on the horizontal axis. It can be seen that while the generation time increases with the number of tokens used, the growth rate is linear.

Testing the Programs. Fig. 7 shows how the tester module scales in relation to the number of tokens the program uses. While the time required for testing a case greatly depends on the tasks achieved by the program, it can be seen that the total number of tokens used, plays an important role: the more tokens used, the more time is required for the tester to check the program. The figure indicates that testing programs larger than 14k–15k tokens is not advisable as the time required for the process increases drastically.

Crashing Test Cases Rate. In Fig. 9 the results of an experiment for determining the distribution of the number of generated cases based on the number of tokens are shown (represented by the red bars). The figure also shows the rate at which the programs crash the compiler (blue bars). Analyzing them in relation to the generated test cases, a crash rate of nearly 50% can be determined. This happens as P4Fuzz has a high chance of randomly generating declaration of nested data types and variables which seems to not be implemented as intended in the P4C compiler.

Taming. While taming is an expensive process, it can be seen from Fig. 8 that it gives better results when compared to analyzing the potential bugs in a random order. Because of the clustering of errors based on their similarity in terms of word occurrences, for the first few number of test cases analyzed, we get the same number of distinct errors as the theoretical best. Overall it seems that the clustering helps improving the process of analyzing potential bugs by increasing the number of distinct errors analyzed using fewer test cases.

7 BUGS FOUND

Using P4Fuzz, we successfully found four new and distinct bugs within the P4C compiler. Out of these four bugs, two (Bugs #1291 and #1296) were fixed on the official repository, one of them (Bug #562) is known but not fixed, and one of them (Bug #1325) is

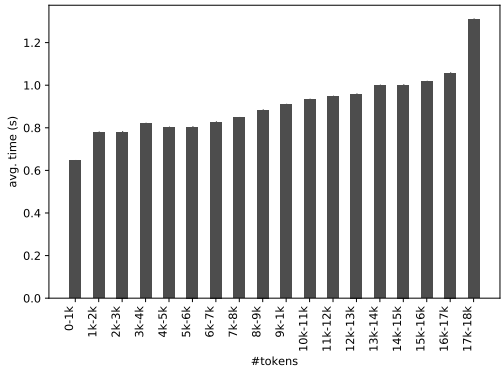

Figure 6: Average generation time for test cases based on number of tokens used

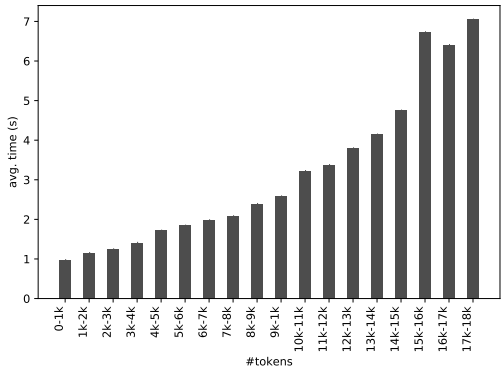

Figure 7: Average compilation time for a test cases based on number of tokens used

pending. The following sections present descriptions of the bugs alongside ways of reproducing them.

Bug #1296: Specializing Extern Objects. Issue #1296[3] shows that declaring specialized `extern` objects by specializing them on the same `extern` object type triggers a compiler bug as shown in the following example:

```
register<register<bit<32>>>(1) test;
```

We have tried compiling the above snippet for the BMv2 target which leads to a compilation error. This is a BMv2 limitation. The following shows the output of the compiler when a P4 program triggering the explained bug is used:

```
In file: p4c/ir/visitor.cpp:54
Compiler Bug: IR loop detected
```

The bug occurred 770 times within 4920 possible bugs saved in the database. Issue #1296 was tagged as a bug and solved in the official P4C repository.

Bug #1291: Varbit Declaration in Structs. The P4 specification states that varbits are allowed in a struct container type [22, §7.2.7]:

```
struct metadata_t { varbit<8> test; }
```

[3]https://github.com/p4lang/p4c/issues/1296

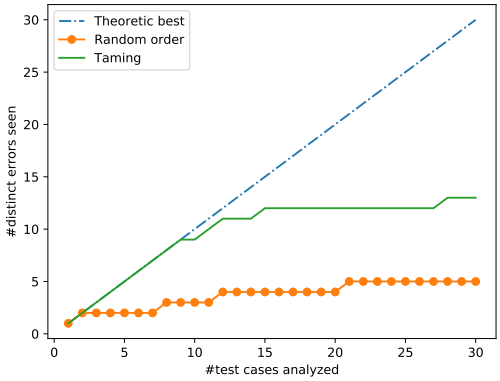

Figure 8: Tamed vs. random ordering distinct errors found per number of cases analyzed

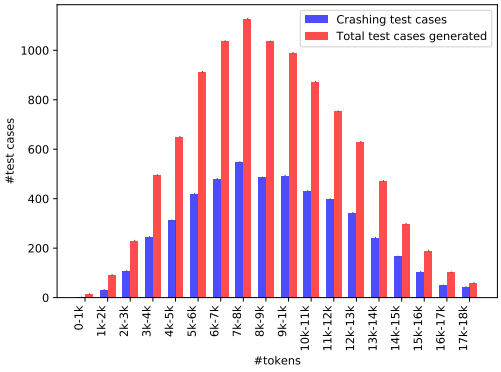

Figure 9: Distribution of test cases in relation to the number of tokens used

Issue #1291[4] shows that declaring varbit types inside structs and compiling for the BMv2 target leads to compilation error:

```
Compiler Bug: issue.p4(8): varbit<8>: Unhandled type
  ↪ for @name("userMetadata.test") varbit<8> test
    varbit<8> test;
              ^
issue.p4(8)
    varbit<8> test;
              ^^^^
```

The bug occurred 195 times within 4920 possible bugs saved in the database. Issue #1291 was tagged as bug and solved in the official P4C repository.

Bug #1325: Error Type in Nested Struct The P4 specification states that both structs and errors are allowed in struct container types [22, §7.2.7]:

```
struct test_struct { error test_error; }
struct local_metadata_t { test_struct test; };
```

However, compiling it for the BMv2 target leads to a compilation error, registered as Issue #1325[5]. The bug occurred 1019 times within 4920 possible bugs saved in the database. Issue #1325 was tagged as bug in the official P4C repository.

[4]https://github.com/p4lang/p4c/issues/1291
[5]https://github.com/p4lang/p4c/issues/1325

Bug #562: Nested Structs According to the P4 language specification struct types are allowed in struct container types [22, §7.2.7]:

```
struct alt_t { bit<1> valid; bit<7> port; };
struct row_t { alt_t alt0; alt_t alt1; };
struct parsed_packet_t {};
struct local_metadata_t { row_t row; };
```

However, this leads to a compilation error when compiling for the BMv2 target . The error was registered as Issue #562[6]. The bug occurred 617 times within 4920 possible bugs saved in the database. Issue #562 was tagged as bug in the official P4C repository.

8 NETFPGA EXPERIMENTS

In addition to the BMv2 target, we have also experimented with a NetFPGA SUME Board. Using P4Fuzz on the NetFPGA-SUME compiler (the P4 to SDNet translator) we have discovered some limitations of the compiler: tables are not allowed to have empty key lists, empty headers and structs are not allowed, and the error and the varbit types cannot be used in structs. While all these limitations are explicitly verified by the compiler, there are still cases in which the *p4c-sdnet* tool reports errors as compiler bugs. One such case is when the *select* transition for a parser state is empty:

```
state start{
    transition select(user_metadata.test){  }
}
```

Which yields the following compiler output:

```
terminate called after throwing an instance of 'Util::
 ↪ CompilerBug'
 what():  In file: /scratch/p4c_sdnet/build/p4c/
 ↪ extensions/sdnet/analysis/typeMap.cpp:7
 Compiler Bug: /scratch/p4c_sdnet/build/p4c/
 ↪ extensions/sdnet/analysis/typeMap.cpp:7: Null type
```

Another case is represented by the external objects specialization on the same external object type:

```
extern test_extern<T> {
    test_extern();
    void read(out T result);
    void write(in T value);
}
test_extern<test_extern<bit<32>>>() test;
```

9 CONCLUSION

We initiated the study of security issues related to the P4 compiler, arguably a fundamental aspect: the P4 Runtime API is likely a common component of both present and future SDN solutions. Our paper hence complements related work focusing on the program's expected behavior or the program's execution paths.

REFERENCES

[1] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. 2014. NetKAT: Semantic foundations for networks. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 113–126.

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

[3] Mihai Budiu. 2019. Programming networks with P4. In *VMware Research Blog*.

[6] https://github.com/p4lang/p4c/issues/562

[4] Marco Canini, Daniele Venzano, Peter Peresini, Dejan Kostic, and Jennifer Rexford. 2012. A NICE way to test OpenFlow applications. In *Proc. 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[5] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Z. Fern, Eric Eide, and John Regehr. 2013. Taming compiler fuzzers. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2013)*. 197–208. https://doi.org/10.1145/2462156.2462173

[6] Tooska Dargahi, Alberto Caponi, Moreno Ambrosin, Giuseppe Bianchi, and Mauro Conti. 2017. A survey on the security of stateful SDN data planes. *IEEE Communications Surveys & Tutorials* 19, 3 (2017), 1701–1725.

[7] Mohan Dhawan, Rishabh Poddar, Kshiteej Mahajan, and Vijay Mann. 2015. SPHINX: Detecting Security Attacks in Software-Defined Networks. In *Proc. Annual Network & Distributed System Security Symposium (NDSS)*, Vol. 15. 8–11.

[8] Lucas Freire, Miguel C. Neves, Lucas Leal, Kirill Levchenko, Alberto E. Schaeffer Filho, and Marinho P. Barcellos. 2018. Uncovering Bugs in P4 Programs with Assertion-based Verification. In *Proceedings of the Symposium on SDN Research (SOSR 2018)*. 4:1–4:7. https://doi.org/10.1145/3185467.3185499

[9] Sungmin Hong, Lei Xu, Haopei Wang, and Guofei Gu. 2015. Poisoning Network Visibility in Software-Defined Networks: New Attacks and Countermeasures.. In *Proc. Annual Network & Distributed System Security Symposium (NDSS)*, Vol. 15. 8–11.

[10] Changhoon Kim, Anirudh Sivaraman, Naga Katta, Antonin Bas, Advait Dixit, and Lawrence J Wobker. 2015. In-band network telemetry via programmable dataplanes. In *Proc. ACM SIGCOMM*.

[11] Rowan Kloti, Vasileios Kotronis, and Paul Smith. 2013. Openflow: A security analysis. In *Network Protocols (ICNP), 2013 21st IEEE International Conference on*. IEEE, 1–6.

[12] Diego Kreutz, Fernando Ramos, and Paulo Verissimo. 2013. Towards secure and dependable software-defined networks. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*. ACM, 55–60.

[13] Andres Nötzli, Jehandad Khan, Andy Fingerhut, Clark Barrett, and Peter Athanas. 2018. p4pktgen: Automated Test Case Generation for P4 Programs. In *Proceedings of the Symposium on SDN Research (SOSR 2018)*. 5:1–5:7. https://doi.org/10.1145/3185467.3185497

[14] Diana Andreea Popescu, Gianni Antichi, and Andrew W Moore. 2017. Enabling fast hierarchical heavy hitter detection using programmable data planes. In *Proc. Symposium on SDN Research (SOSR)*. ACM, 191–192.

[15] Fabian Ruffy, Tao Wang, and Anirudh Sivaraman. 2020. Gauntlet: Finding Bugs in Compilers for Programmable Packet Processing. In *USENIX OSDI*.

[16] Sandra Scott-Hayward, Gemma O'Callaghan, and Sakir Sezer. 2013. SDN security: A survey. In *Future Networks and Services (SDN4FNS), 2013 IEEE SDN For.* IEEE, 1–7.

[17] Bhargava Shastry, Markus Leutner, Tobias Fiebig, Kashyap Thimmaraju, Fabian Yamaguchi, Konrad Rieck, Jean-Pierre Seifert Stefan Schmid, and Anja Feldmann. 2017. Static Program Analysis as a Fuzzing Aid. In *Proc. 20th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*.

[18] Seungwon Shin, Vinod Yegneswaran, Phillip Porras, and Guofei Gu. 2013. Avant-guard: Scalable and vigilant switch flow management in software-defined networks. In *Proc. ACM SIGSAC Conference on Computer & Communications Security (CCS)*. 413–424.

[19] Seung Won Shin, Phillip Porras, Vinod Yegneswara, Martin Fong, Guofei Gu, and Mabry Tyson. 2013. Fresco: Modular composable security services for software-defined networks. In *Proc. 20th Annual Network & Distributed System Security Symposium (NDSS)*.

[20] Anirudh Sivaraman, Changhoon Kim, Ramkumar Krishnamoorthy, Advait Dixit, and Mihai Budiu. 2015. Dc. p4: Programming the forwarding plane of a data-center switch. In *Proc. ACM SIGCOMM Symposium on Software Defined Networking Research (SOSR)*. ACM.

[21] Michael Sutton, Adam Greene, and Pedram Amini. 2007. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley.

[22] The P4 Language Consortium. 2017. $P4_{16}$ Language Specification. Published online (Version 1.0.0). https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html Last accessed: 3 August 2018.

[23] The P4 Language Consortium. 2018. P4 Language and Related Specifications. https://p4.org/p4-spec/. Accessed: 2018-05-29.

[24] K. Thimmaraju, L. Schiff, and S. Schmid. 2017. Outsmarting Network Security with SDN Teleportation. In *Proc. IEEE European Symposium on Security and Privacy (EuroSP)*. 563–578.

[25] Kashyap Thimmaraju, Bhargava Shastry, Tobias Fiebig, Felicitas Hetzelt, Jean-Pierre Seifert, Anja Feldmann, and Stefan Schmid. 2018. Taking Control of SDN-based Cloud Systems via the Data Plane. In *Proc. ACM Symposium on SDN Research (SOSR)*.

[26] Ken Thompson. 1984. Reflections on trusting trust. *Commun. ACM* 27, 8 (1984), 761–763.

[27] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2011)*. 283–294. https://doi.org/10.1145/1993498.1993532