P4Fuzz: A Compiler Fuzzer for Securing P4 Programmable Dataplanes

Master's Thesis

Andrei Alexandru Agape
Mădălin Claudiu Dănceanu

Aalborg University
Department of Computer Science
Selma Lagerløfs Vej 300
DK-9220 Aalborg Øst

Department of Computer Science
Selma Lagerløfs Vej 300
DK-9220 Aalborg Øst
http://www.cs.aau.dk

AALBORG UNIVERSITY
STUDENT REPORT

Title:

P4Fuzz: A Compiler Fuzzer for Securing P4 Programmable Dataplanes

Theme:
Master's Thesis

Project Period:
Spring Semester 2018

Project Group:
DEIS 1016f18

Participant(s):
Andrei Alexandru Agape
Mădălin Claudiu Dănceanu

Supervisor(s):
René Rydhof Hansen
Stefan Schmid

Copies: 1

Page Numbers: 101

Date of Completion:
June 8, 2018

Abstract:

The evolution of networking from a traditional approach towards a more flexible one shown an improvement in the quality of services offered. However, the changes required in this sense have to consider as well the security risks that are implied. Motivated by previous research and lack of security tools for newly developed technologies, we chose to cover an unexplored part of the attack surface and pursue a different approach. Our objective is to secure the programmable dataplanes by uncovering bugs in P4 compilers. We implement P4Fuzz - a smart, blackbox and generation-based fuzzer - inspired by Csmith, that incorporates taming techniques and complements related work. Our tool is able to generate up to 80 P4 programs per minute, and test the validity for up to 21 programs per minute. P4Fuzz is designed such that it can support multiple architectures: i.e: BMv2, eBPF, while others can be added in the future. We discovered and reported four bugs, out of which two of them have been fixed on the official repository of P4C, the standard compiler for P4. A case study which shows how compiler bugs can introduce security issues was also conducted, and we consider that P4Fuzz manages to fill a gap in the literature.

Contents

Summary

The evolution of networking from a traditional approach towards a more flexible one showed an improvement in the quality of services offered. The recent trends show that the networking environment is moving towards technologies such as OpenFlow, SDN and P4 programmable dataplanes.

However, the changes required in this sense have to consider as well the security risks that are implied. Our previous research that identified a large attack surface within SDN and P4, correlated with the lack of security tools for newly developed technologies made us consider this project.

Motivated by these aspects we chose to cover an unexplored part of the attack surface and pursue a different approach compared to the related work. Our objective is to secure the programmable dataplanes by uncovering bugs in P4 compilers.

We implement P4Fuzz: a smart, blackbox and generation-based fuzzer, inspired by Csmith. The tool incorporates taming techniques upon the found bugs using a token-based distance function and k-medoids clustering. The packet generator module of P4Fuzz complements a state of the art tool - P4pktgen - recently developed by experts from Stanford University, Cisco Systems and Virginia Tech.

Our tool is able to generate up to 80 P4 programs per minute, and test validity for up to 21 programs per minute. P4Fuzz is designed such that it can support multiple architectures such as BMv2 and eBPF, while others can be added in the future.

During the span of a few weeks we discovered and reported four bugs, out of which two of them have already been fixed on the official repository of P4C, the standard compiler for P4. A case study which shows how compiler bugs can introduce security issues was also conducted, and we consider that P4Fuzz can address this kind of problems, fulfilling its scope.

The report is structured as following: Chapter 1 highlights the motivation of our work, together with the main contributions achieved during the project. Chapter 2 presents the related work done within the field, such as papers and tools, used as sources of inspiration or to complement P4Fuzz. Chapter 3 presents the background about the P4 language and compiler required to understand the scope, design and implementation of our fuzzer.

Chapter 4 explains the main challenges encountered while designing and developing P4Fuzz, while Chapter 5 addresses these problems and describes the decisions taken during the project. Chapter 6 presents the technical details related to P4Fuzz

including the programming language, libraries, tools and specifications of the architectures used.

Chapter 7 evaluates the tool, focusing on aspects such as the bugs identified, overall performance and interpretation of the results. Chapter 8 explains how bitwise negation bug introduced by the compiler can represent a threat, proving that bugs identified in the P4 compiler help increasing the security of programmable dataplanes.

Finally, Chapters 9 and 10 show the limitations of the fuzzer, provide future development ideas and draw the conclusion of the report.

Preface

Acknowledgements

We would like to thank our supervisors, Assoc. Prof. René Rydhof Hansen and Assoc. Prof. Stefan Schmid, for their advice and guidance with which they have contributed throughout our meetings, the information and resources provided, and for the feedback regarding our thoughts and working process.

Aalborg University, June 8, 2018

Andrei Alexandru Agape
<aagape16@student.aau.dk>

Mădălin Claudiu Dănceanu
<mdance16@student.aau.dk>

Chapter 1

Introduction

1.1 Motivation

Evolution of networking in the last years from a traditional approach to a more flexible one has been made with the help of technologies such as OpenFlow, Software Defined Networking and most recently programmable dataplanes.

While many resources and studies exist for OpenFlow and SDN, this may not be the case of programmable dataplanes, more precisely the language that makes this programmability possible: P4, as well as its compilers. This is due to the novelty of technology and the fact that the development of language, compiler and existing tools has not fully stabilized yet.

One main aspect that must be considered in the transition from traditional dataplanes to programmable ones is security. This awareness is not something new, and has been raised before by other publications and papers. Our previous studies "Exposing Security Issues in P4 Programmable Software Defined Networks" [1] and "Charting the Security Landscape of Programmable Dataplanes" [2] present a security analysis of the environment, highlighting the possible attacks and countermeasures.

"P4pktgen: Automated Test Case Generation for P4 Programs" [3] written by experts from Stanford University, Cisco Systems and others, acknowledge as well the security problem that can occur and provide a tool meant to identify the issues. "Uncovering Bugs in P4 Programs with Assertion-based Verification" [4] argues that bugs can be prevented in P4 programs with the implementation of asserts that verify if the intended behaviour is satisfied.

The previously mentioned publications confirm the need for more research to be done on the security aspect of P4 programmable dataplanes. While P4pktgen focuses on generating test cases to "validate if P4 programs act as intended on a given device " [3], and P4 assert [4] tries to prevent bugs with the use of assertions meant to verify if the intended behaviour is satisfied, we cover another part of the attack surface and pursue a different approach.

Inspired by our previous work "Exposing Security Issues in P4 Programmable Software Defined Networks" [1] and "Charting the Security Landscape of Programmable

Dataplanes" [2] we focus our efforts on finding bugs in the P4 compiler, by developing a fuzzer for it.

The compilation process is for most developers unknown. While usually it is the programs that contain security flaws, sometimes the compilers can be the cause of the problem if the generated code is not following the developer's intent. Implementing a compiler fuzzer is motivated by the following reasons:

- an uncovered aspect of P4 is the lack of security tools that focus on compiler; the existing software focuses on either the packets (P4pktgen) or the language and programs (P4assert)
- fuzzing is a popular technique used in the security field to explore corner cases and find bugs that may prove to be exploitable.
- compiler fuzzers have been proven successful at finding bugs in other languages, such as it is the case of Csmith for C

Our previous research, the related work and the reasons mentioned above, lead to the hypothesis that a fuzzer can be used to discover bugs in the P4 compilers. By finding out and fixing compiler bugs, the overall security level of P4 programmable dataplanes can be increased. However, devising a compiler fuzzer for P4 is not a trivial process; it implies tailoring the fuzzer for different architectures, dealing with nested types and recursive statements as well as many other challenges fully described in Chapter 4. Among these challenges we can mention generating semantically valid programs, testing programs behaviour, and sorting faulty test cases based on error relevance.

1.2 Contribution

During the project we contributed to the robustness and security aspects of the recently developed compilers for P4; our hope is that P4Fuzz can be used on existing and future P4 compilers to identify bugs in their implementation, and mitigate the consequences. Among our contributions to the P4 community we can mention:

- Raise awareness of how P4 compilers that contain bugs in their implementation can affect the security of dataplanes.
- Developed P4Fuzz - a tool that generates syntactically and semantically valid P4 programs that stress the compilers in order to find bugs in their implementation.
- Complement the work of other existing projects within security and P4 field: 1) use P4pktgen to generate packets for the random generated P4 programs and explore their paths using symbolic execution. 2) use taming to sort interesting and different bugs by ranking them more highly.

- Using P4Fuzz, we identified four bugs in the implementation of the P4C compiler, out of which three of them were new discoveries. Issues #1291 and #1296 on P4C repository [5] were acknowledged and fixed, issue #1325 was marked as bug but not fixed yet, while issue #562 is a known problem that has not been solved.
- Evaluation of P4Fuzz shown the overall performance, bugs distribution across compiler stages, taming comparison between theoretical best case, random and token-based function. The tool can generate up to 80 P4 programs per minute, and test up to 21 programs per minute.
- Provide a token-based distance function for the taming algorithm as alternative to the Levenshtein distance and show how the time complexity is reduced by an exponential factor.

1.3 Report Structure

The report is structured as following: Chapter 2 presents the related work done within the field, such as papers and tools, used as sources of inspiration or to complement P4Fuzz. Chapter 3 presents the background about the P4 language and compiler required to understand the scope, design and implementation of our fuzzer.

Chapter 4 explains the main challenges encountered while designing and developing P4Fuzz, while Chapter 5 addresses these problems and describes the decisions taken during the project. Chapter 6 presents the technical details related to P4Fuzz including the programming language, libraries, tools and specifications of the architectures used.

Chapter 7 evaluates the tool, focusing on aspects such as the bugs identified, overall performance and interpretation of the results. Chapter 8 explains how bitwise negation bug introduced by the compiler can represent a threat, proving that bugs identified in the P4 compiler help increasing the security of programmable dataplanes.

Finally, Chapters 9 and 10 show the limitations of the fuzzer, provide future development ideas and draw the conclusion of the report.

Chapter 2

Related Work

This chapter presents the related work done within the field, such as papers and tools, used as sources of inspiration or to complement P4Fuzz. Csmith is one of the most popular fuzzers, developed in order to test C compilers. P4pktgen is a tool recently released and designed to craft packages for given P4 programs. "Taming Compiler Fuzzers" provides the theoretical background needed to order the bugs found such that the diverse, interesting cases are highly ranked. ASSERT-P4 presents how assertion checking and symbolic execution can be used to verify the dataplane programs and prevent bugs.

2.1 Csmith

Csmith [6] is a tool for generating randomized C programs as test case scenarios for finding bugs in C compilers. The approach taken by the tool is based on differential testing, a technique in which the same randomly generated C program (input) is used by multiple similar compilers (programs) with the purpose of observing differences in execution or output.

Even though Csmith has a different target language, the purpose is similar to ours, and thus the tool, together with the paper, provides us great inspiration regarding how a compiler fuzzer in general can be implemented. While our test case generator uses similar ideas such as keeping track of the current scope of the program, implementing probabilities for productions or filtering productions based on the current context, there are some major differences between Csmith and P4Fuzz.

One difference is that they are trying to find errors in a well-known and established general purpose language such as C, while we are trying to find bugs in a newer, domain specific language used in networking, namely P4 with the purpose of exposing security issues. The implications that arise from this difference are both positive and negative. One can argue there might be more bugs to find in new compilers for a new language. On the other hand, because it is a new language, there are also aspects that make the process of designing and implementing P4Fuzz more difficult. The lack of a P4 program reducer and the low number of back-ends available

represent limitations for the scope of P4Fuzz.

Another difference is that Csmith is specifically avoiding undefined behaviours since it is hard to argue that implementing those differently is indeed a bug, while we include undefined behaviour in the randomly generated programs as differences in implementation between compilers can lead to security threats. This difference arises from the fact that Csmith is trying to find aspects of the compilers which are not complying with the C standards, while the purpose of P4Fuzz is to help improve the P4 compiler but also to find potential security threats.

Both the similarities and the differences between Csmith and P4Fuzz are more deeply described in Section 5.4.2 and Section 6.4.

2.2 P4pktgen

P4pktgen is a tool specially developed in order to automatically generate test cases (packets) for P4 programs. The packets are generated using symbolic execution and can be used to check if a given P4 program acts as intended on a device [3].

While P4Fuzz generates P4 programs that are used to test the compiler implementation, such programs can only check for compile-time errors. In order to test if different compiler implementations produce target code that has the same behaviour, an input for the randomly generated program is needed. Using the p4pktgen tool, we are able to automate the fuzzing process and provide the randomly generated P4 programs with valid inputs that test its paths. The p4pktgen tool and P4Fuzz hence nicely complement each other.

The p4pktgen comes with three great advantages: it provides a valid input for the randomly generated programs; the packet generation process is entirely automated, thus no external intervention is needed; it finds and tests P4 program paths in an automated way, by using symbolic execution. The role of p4pktgen within the overall architecture of the P4Fuzz can be observed in Chapter 5.4.

2.3 Taming Compiler Fuzzers

While fuzzers are great at finding bugs within compiler implementations, the paper "Taming Compiler Fuzzers" [7] argues that the number of problems found can be overwhelming to process manually by developers. At the same time some bugs occur more often than others, and it is also hard to differentiate between their severity.

The paper provides a theoretical background by formulating and addressing what they call "the fuzzer taming problem: given a potentially large number of random test cases that trigger failures, order them such that diverse, interesting test cases are highly ranked" [7]. The approach proposed is to define a distance function between test case and then sort the list of test cases in furthest point first (FPF) order, thus finding an approximate solution to the taming problem [7]. The FPF sorting aims to identify the test cases that differ the most one from each other, finding this way more unique problems.

The taming problem and solution provides us with a way to rank diverse and interesting test cases. However a distance function for a generic compiler bug cannot be easily defined. It is highly related to the types of errors and the compiler fuzzer developed. Paper [7] defines a variety of functions that can be used including Levenshtein distance on test cases, compiler output and Valgrid output; Euclidean distance.

We firstly used the FPF algorithm on the Levenstein distance as described in [7], but it proved to be ineffective on the errors generated by P4 compiler. However we had better results by clustering the errors and calculating the distance using a custom function based on tokenization. The distance function chosen in our case is defined in chapter Chapter 6.8. The role of taming within the overall architecture of the P4Fuzz can be observed in Chapter 5.4.

2.4 ASSERT-P4

"Uncovering Bugs in P4 Programs with Assertion-based Verification" [4] is a paper that presents how assertion checking and symbolic execution can be used to verify the dataplane programs. Figure 2.1 describes the flow of the annotated P4 program through ASSERT-P4.

In order to use the developed tool - ASSERT-P4 - the developer has to annotate the program with assertions just as specified by the paper. The translator then creates a C-based model of the given program, together with the optionally provided forwarding rules. Finally the C-based model is verified using a symbolic execution engine by testing the possible paths and checking the assertions.

If any of the assertions is violated, the developer is informed and the program is considered invalid. ASSERT-P4 proves the need of more tools that improve the P4 programmable dataplanes, either by securing them, checking the execution paths or validating the intended behaviour.

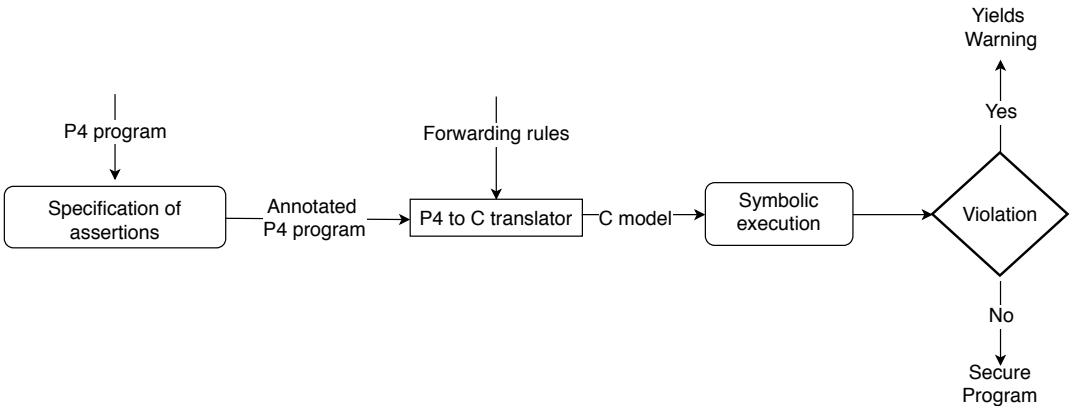

Figure 2.1: ASSERT-P4 flow [4]

Chapter 3

Background

This chapter presents the background on the P4 language and compiler required in order to understand the scope, design and implementation of P4Fuzz. The first part, Section 3.1 shows the details about the language, structure and the main components that can be found within a P4 program. The second part, Chapter 3.2 focuses on the P4 compiler P4C, the way it handles both version 14 and 16 of P4, and describes its components. The third part describes the fuzzing process in general and the types of fuzzers that can be used.

3.1 P4 Language

P4 is a domain-specific language that provides a number of constructs optimized around network data forwarding. P4 stands for Programming Protocol-Independent Packet Processors and it has three main design goals: Target independence, Protocol independence and Reconfigurability [1]. The previous release of the language is P4 version 14 [8]. The version used within this project is P4 16 [8], which is the current version as of June 2018.

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 back-end. The P4 compiler maps the P4 source code into the target switch model [1].

The **protocol independent** goal says 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 [1].

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 [1].

3.1.1 Program Structure

A P4 program can 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. 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. This type of structure matches the Protocol Independent Switch Architecture (PISA) shown in Figure 3.1, making it easier to program the dataplane behaviour [1].

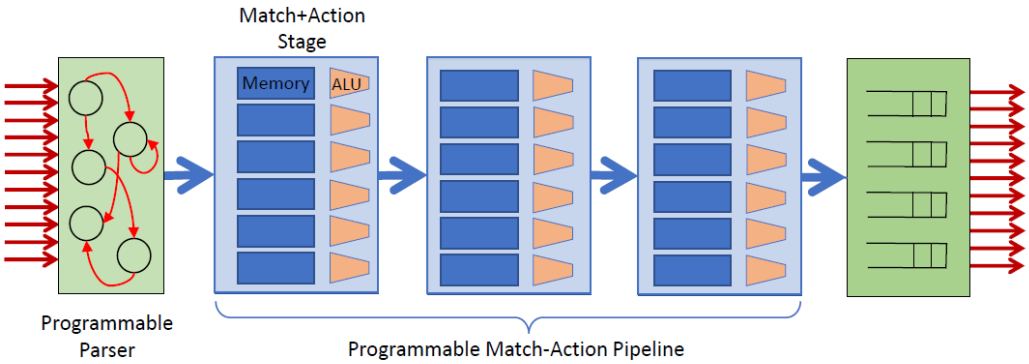

Figure 3.1: PISA: Protocol Independent Switch Architecture [9]

3.1.2 Data Declaration

The **data declaration** part provides a suitable place to declare the data types used later in the program execution. "P4 provides a number of type constructors that can be used to derive additional types including: enum, header, header stacks, struct, header_union, tuple, type specialization, extern, parser, control, package. The types header, header_union, enum, struct, extern, parser, control, and package can only be used in type declarations, where they introduce a new name for the type. The type can subsequently be referred to using this identifier." [10]

Figure 3.2 presents an overview of the most used derived and base types in the data declaration section, together with the relationship between them.

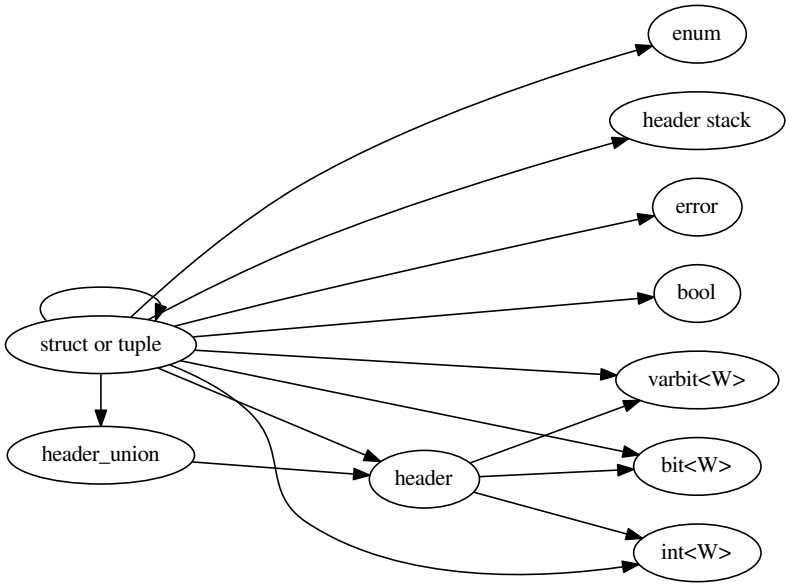

Figure 3.2: Data declaration [11]

Listing 3.1 is an example of P4 program data declaration section. In this case four headers are declared: ethernet, ipv4, packet_in and packet_out. Lastly, they are put together in the headers_t struct.

```
1
2  header ethernet_t {
3      bit<48> dst_addr;
4      bit<48> src_addr;
5      bit<16> ether_type;
6  }
7
8  header ipv4_t {
9      bit<4>  version;
10     bit<4>  ihl;
11     bit<8>  diffserv;
12     bit<16> len;
13     bit<16> identification;
14     bit<3>  flags;
15     bit<13> frag_offset;
16     bit<8>  ttl;
17     bit<8>  protocol;
18     bit<16> hdr_checksum;
19     bit<32> src_addr;
20     bit<32> dst_addr;
21 }
22
23 /* Packet-in header. Prepended to packets sent to the controller and used
```

```
    to carry the original ingress port where the packet was received. */
24
25 @controller_header("packet_in")
26 header packet_in_header_t {
27     bit<9> ingress_port;
28 }
29
30 /* Packet-out header. Prepended to packets received by the controller and
       used to tell the switch on which physical port this packet should be
       forwarded. */
31 @controller_header("packet_out")
32 header packet_out_header_t {
33     bit<9> egress_port;
34 }
35
36 /* For convenience we collect all headers under the same struct. */
37 struct headers_t {
38     ethernet_t ethernet;
39     ipv4_t ipv4;
40     packet_out_header_t packet_out;
41     packet_in_header_t packet_in;
42 }
```

Listing 3.1: P4 program source code - Data declaration [12]

3.1.3 Parsers

The **parsers** part provides a suitable place to declare the parsers and specify how the data is extracted from the incoming byte-stream. The parsers are defined as finite state machines: "a P4 parser describes a state machine with one start state and two final states. The start state is always named start. The two final states are named accept (indicating successful parsing) and reject (indicating a parsing failure). The start state is part of the parser, while the accept and reject states are distinct from the states provided by the programmer and are logically outside of the parser" [10].

Figure 3.3 presents an overview of the parser and its possible components, together with the relationship between them.

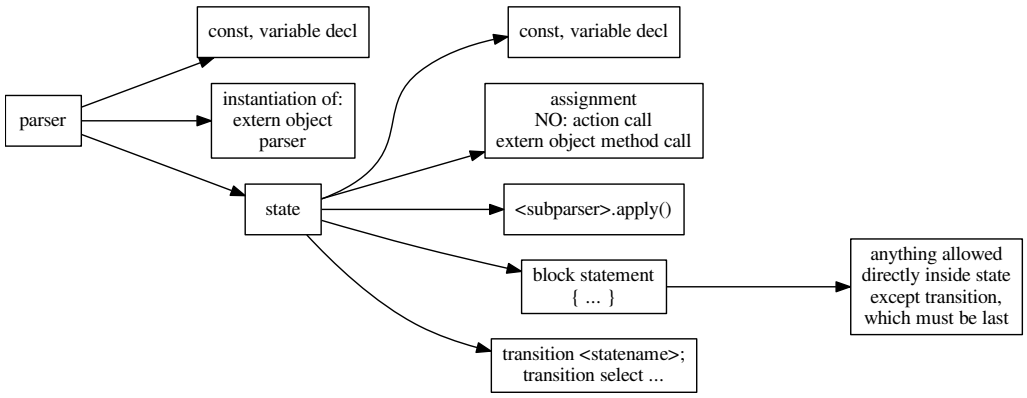

Figure 3.3: Parser [11]

Listing 3.2 is an example of P4 program parser section. In this case the *ParserImpl* parser takes as input four parameters: packet, hdr, meta and standard_metadata and implements four states: start, parser_packet_out, parser_ethernet and parser_ipv4. The states have either transition to the accept state or to other states i.e: **transition accept** and **transition parse_ethernet**.

```
1
2  parser ParserImpl(packet_in packet,
3                    out headers_t hdr,
4                    inout metadata_t meta,
5                    inout standard_metadata_t standard_metadata) {
6
7      state start {
8          transition select(standard_metadata.ingress_port) {
9              CPU_PORT: parse_packet_out;
10             default: parse_ethernet;
11         }
12     }
13
14     state parse_packet_out {
15         packet.extract(hdr.packet_out);
16         transition parse_ethernet;
17     }
18
19     state parse_ethernet {
20         packet.extract(hdr.ethernet);
21         transition select(hdr.ethernet.ether_type) {
22             ETH_TYPE_IPV4: parse_ipv4;
23             default: accept;
24         }
25     }
26
27     state parse_ipv4 {
28         packet.extract(hdr.ipv4);
29         transition accept;
```

```
30      }
31  }
```

Listing 3.2: P4 program source code - parser section [12]

3.1.4 Control Flow

The **control flow** part provides a suitable place to declare the control blocks containing actions, tables and data manipulation: "P4 parsers are responsible for extracting bits from a packet into headers. These headers (and other metadata) can be manipulated and transformed within control blocks. The body of a control block resembles a traditional imperative program. Within the body of a control block, match-action units can be invoked to perform data transformations. Match-action units are represented in P4 by constructs called tables." [10].

Figure 3.4 shown the main components that can be present within the control blocks.

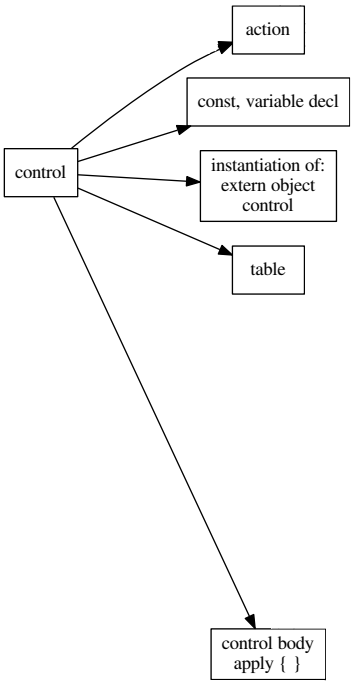

Figure 3.4: Control [11]

3.1.5 Actions

Actions are code fragments that can read and write the data being processed. Actions may contain data values that can be written by the control plane and read

by the data plane. Actions are the main construct by which the control-plane can influence dynamically the behavior of the data plane [10].

Figure 3.5 presents an overview of the **action** and its possible components.

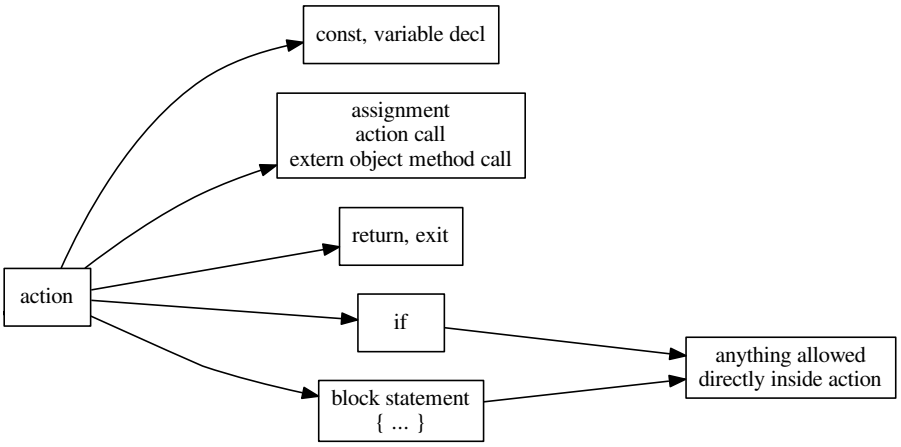

Figure 3.5: Action [11]

```
1
2  control IngressImpl(inout headers_t hdr,
3                      inout metadata_t meta,
4                      inout standard_metadata_t standard_metadata) {
5
6      action send_to_cpu() {
7          standard_metadata.egress_spec = CPU_PORT;
8
9          /*Packets sent to the controller needs to be prepended with the
       packet-in header. By setting it valid we make sure it will be deparsed
        before the ethernet header (see DeparserImpl). */
10
11         hdr.packet_in.setValid();
12         hdr.packet_in.ingress_port = standard_metadata.ingress_port;
13     }
14
15     action set_egress_port(port_t port) {
16         standard_metadata.egress_spec = port;
17     }
18
19     action _drop() {
20         standard_metadata.egress_spec = DROP_PORT;
21     }
22
23     /*Tables*/
24     /*Control body*/
25     }
```

Listing 3.3: P4 program source code - Action section [12]

3.1.6 Tables

A **table** describes a match-action unit. Processing a packet using a match-action table executes the following steps:

- Key construction [10]
- Key lookup in a lookup table (the "match" step). The result of key lookup is an "action" [10]
- Action execution (the "action step") over the input data, resulting in mutations of the data [10]

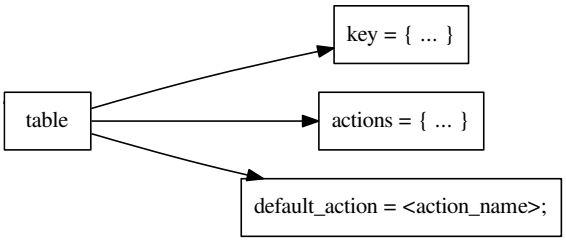

Figure 3.6: Table [11]

```
1
2  control IngressImpl(inout headers_t hdr,
3                      inout metadata_t meta,
4                      inout standard_metadata_t standard_metadata) {
5
6      /*Actions*/
7
8      table table0 {
9          key = {
10             standard_metadata.ingress_port  : ternary;
11             hdr.ethernet.dst_addr           : ternary;
12             hdr.ethernet.src_addr           : ternary;
13             hdr.ethernet.ether_type         : ternary;
14         }
15         actions = {
16             set_egress_port();
17             send_to_cpu();
18             _drop();
19         }
20         default_action = _drop();
21     }
22
23     table ip_proto_filter_table {
24         key = {
25             hdr.ipv4.src_addr : ternary;
26             hdr.ipv4.protocol : exact;
```

```
27          }
28          actions = {
29              _drop();
30          }
31      }
32
33      /*Control body*/
34      }
```

Listing 3.4: P4 program source code - Table section [12]

3.1.7 Control Body

According to the language specifications [10], "the body of the control block is executed, similarly to the execution of a traditional imperative program":

- At runtime, statements within a block are executed in the order they appear in the control block [10]
- Execution of the return statement causes immediate termination of the execution of the current control block, and a return to the caller. [10]
- Execution of the exit statement causes the immediate termination of the execution of the current control block and of all the enclosing caller control blocks. [10]
- Applying a table executes the corresponding match-action unit, as described above. [10]

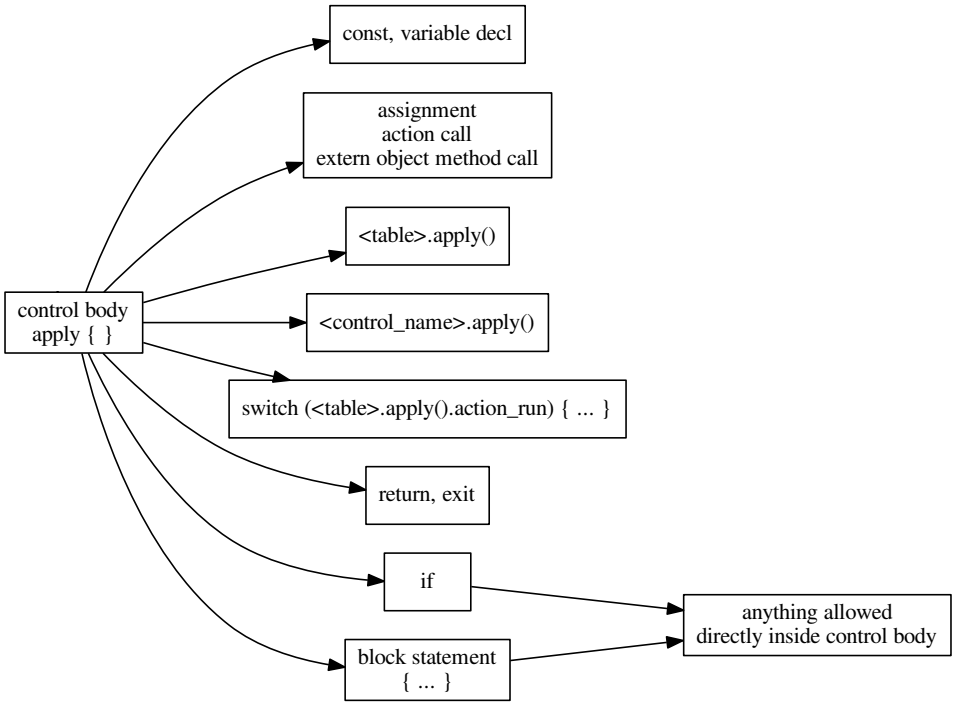

Figure 3.7: Control body [11]

```
1
2  control IngressImpl(inout headers_t hdr,
3                      inout metadata_t meta,
4                      inout standard_metadata_t standard_metadata) {
5
6      /*Actions*/
7      /*Tables*/
8
9     apply {
10         if (standard_metadata.ingress_port == CPU_PORT) {
11
12             /*Packet received from CPU_PORT, this is a packet-out sent by
       the controller. Skip pipeline processing, set the egress port as
       requested by the controller (packet_out header) and remove the
       packet_out header.*/
13
14             standard_metadata.egress_spec = hdr.packet_out.egress_port;
15             hdr.packet_out.setInvalid();
16         } else {
17             /* Packet received from switch port. Apply table0, if action
       is set_egress_port and packet is IPv4, then apply
       ip_proto_filter_table.*/
18
19             switch(table0.apply().action_run) {
20                 set_egress_port: {
```

```
21                      if (hdr.ipv4.isValid()) {
22                          ip_proto_filter_table.apply();
23                      }
24                  }
25              }
26          }
27
28          /*For each port counter, we update the cell at index = ingress/
    egress port. We avoid counting packets sent/received on CPU_PORT or
    dropped (DROP_PORT).*/
29
30          if (standard_metadata.egress_spec < MAX_PORTS) {
31              egr_port_counter.count((bit<32>) standard_metadata.egress_spec
    );
32          }
33          if (standard_metadata.ingress_port < MAX_PORTS) {
34              igr_port_counter.count((bit<32>) standard_metadata.
    ingress_port);
35          }
36       }
37     }
```

Listing 3.5: P4 program source code - Control body section [12]

3.2 P4 Compiler

P4C is the compiler for the P4 programming language that supports both the P4-14 and P4-16 versions. P4C comes with a standard front-end and mid-end, which can be combined with 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 [13]. Multiple back-ends have already been developed, generating code for ASICs, NICs, FPGAs, software switches and other targets [14][1].

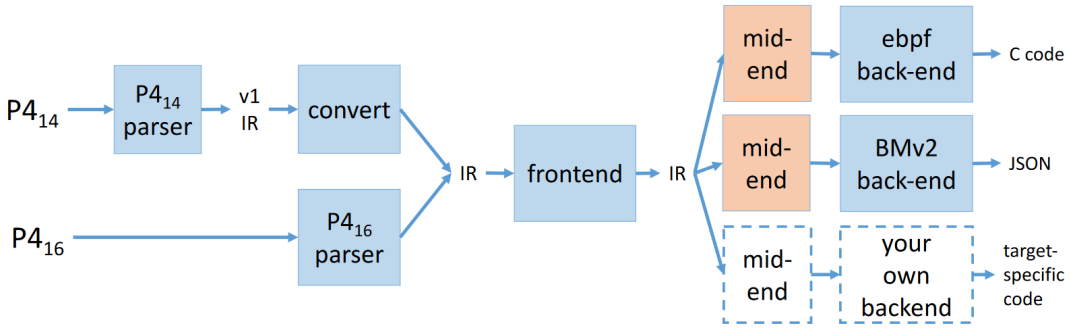

Figure 3.8: P4C compiler data flow [14]

3.2.1 Components

As shown in Figure 3.8 the main components of the compiler are the parser (either P4-14 or P4-16), an 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 [1].

The flow starts with the P4 program source code, either P4-14 or P4-16. In case of P4-14 an extra step is required, as the source code has to be parsed by the P4-14 parser into a version 1 of the Intermediate Representation (IR) before it is converted to the IR accepted by the front-end. The front-end then outputs the IR that can be optimized by the mid-end and sent to the back-end compilers [1].

The front-end is fixed, and the back-end is 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 [1].

3.3 Fuzzing

Fuzzing is the process of automatically generating data and provide it as input for specific applications. The generated inputs are randomized, and their purpose is to find unexpected behaviour of the programs. Ideally, the inputs are "semi-valid" such that they are not directly rejected by the application, and are able to explore corner cases that may not have been handled correctly. [15]

While compilers should be error free, it is not feasible to test all possible inputs. New bugs are introduced as features and optimizations are added to them. Having a set of test-cases may not be enough. An alternative is represented by random testing, also called fuzzing, to generate test-cases. Such tools exists for other programming languages such as C, where Csmith reported over 325 bugs to compiler [1]. According to book [16] and article [17], fuzzers can be split into multiple categories, such as:

- generation-based or mutation-based
- dumb or smart
- white-, grey-, or black-box

3.3.1 Dumb or Smart

The difference between dumb and smart fuzzers consists in their awareness towards input structure. While a dumb fuzzer can be used to test a wider variety of programs, it may generate more invalid inputs and stress their parser rather than specific components. A smart fuzzer is based on an input model to generate more valid inputs compared to a dumb fuzzer. For a compiler fuzzer for example, the input model can be the language specification and the language grammar [15].

3.3.2 White-, Grey-, or Black-box

The difference between white, grey and black-box fuzzers consists in their awareness towards program structure. A higher code coverage makes it possible to discover bugs hidden in more structural elements. A black-box fuzzer is unaware of the structural elements of the tested program, thus regardless of the number of cases generated it may only cover a small part of code[15].

A white-box fuzzer is based on program analysis to increase the code coverage. For example, using symbolic analysis it is possible to discover and explore more execution paths. This can be useful to find problems hidden deeper in the program, but at the same time it requires more resources and time as the analysis is a complex process [15].

The gray-box fuzzer uses instrumentation to get information about the program, rather than program analysis. The instrumentation makes it possible to get rid of the analysis complexity, yet still provide the fuzzer information regarding code coverage which makes the gray-box fuzzing a great alternative to the white-box one [15].

3.3.3 Generation-based or Mutation-based

Generations-based fuzzers are the type of fuzzers able to create inputs from scratch without requiring an already existing valid input. As opposed to generation-based fuzzers, the mutation-based ones make use of the provided seed, and systematically modifies it [15].

Both types can be used in different scenarios, for example: fuzzing an image library can use a mutation-based approach as making changes to the initial provided picture would still generate semi-valid cases. On the other hand, fuzzing a compiler by starting with a given program source code and mutating it would mostly likely always throw an invalid syntax error, as the input is much more sensitive [15].

Chapter 4

Challenges

This chapter presents the main challenges encountered while developing P4Fuzz. Sections 4.1 and 4.2 explain the challenges of generating both syntactically and semantically valid programs by using the P4 language specification. Section 4.3 focuses on testing randomly generated programs at run-time as well, not only at compile time. Once the faulty test cases are found by the fuzzer, they have to be sorted and presented such that the developer can easily identify a larger variety of cases as presented in Section 4.4.

While most of the P4 code is valid on any target, there are specific limits and constrains based on the architecture of the target. The challenge of making a compiler fuzzer that is also target independent is analyzed in Section 4.5. Because in P4 data types and expressions can be nested upon declaration or usage they represent an additional challenge for the code generator due to the maximum recursion depth limit imposed by the programming language in which the fuzzer is written - Section 4.6.

4.1 Generate Syntactically Valid P4 Programs

An important aspect for the randomly generated test cases is that they need to be syntactically valid, such that the compiler can understand them.

The challenge is to randomly generate P4 code in a structured manner such that it can be parsed by the P4 compiler based on its specification of the grammar.

4.2 Generate Semantically Valid P4 Programs

A syntactically valid program is not necessarily completely valid as the program also needs to be semantically valid. Because of this aspect, a P4 compiler fuzzer needs to implement and enforce the semantic rules of P4 when generating the test cases.

While many of the rules are written in the P4 Specifications [10], finding and implementing all the semantic rules that exist for the P4 language is not trivial

with some of the rules being very complicated while others are missing from the documentation. Another challenge is represented by the dependency between the semantic rules and the context in which the program finds itself for each of the generated instruction.

The code generator component of a P4 compiler fuzzer needs to 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.

4.3 Test P4 Programs Behaviour

A compiler fuzzer for finding bugs in P4 compilers should be able to test the randomly generated programs at runtime, not only at compile time. This allows the tool to find errors in the behaviour of the program, errors which could have been introduced by the compiler for example during the optimization phase.

Verifying the correctness of the randomly generated programs behaviour is a challenging task because the tool has no means of deciding which behaviour is intended and which one is a mistake.

4.4 Sort Faulty Test Cases Based on Error Relevance

During the process of generation and verification, the fuzzer finds many errors. These are considered potential bugs and are saved in a database. After careful inspection, some of the errors turn out to be newly discovered bugs while most of them represent already known issues or mistakes in the way the fuzzer generates the programs.

The task of analyzing and labeling the potential faulty test cases is done manually by inspecting the program and the error message for each test case. This is a very tedious work as the generated code is hardly readable and the person analyzing the program has to understand what the code does, what is the reason of the error and determine if that is indeed a bug in the compiler or in the fuzzer.

With a large number of errors given by the compiler fuzzer it is important for the tool to be able to rank the faulty test cases based on their novelty and importance. Ideally we need to sort them in a way that would maximize the number of newly discovered bugs in relation to the number of test cases manually analyzed.

Because the tool is not able to understand the error messages, the challenge is to make it able to present to the user the most relevant test cases first.

4.5 Working with Different Architectures

The P4 language is target independent which means that P4 programs are capable of being compiled and run on many different appliances, each of them having different architectures.

The P4 compiler (P4C) is able to take the same P4 program and compile it for different targets because it is modular, with a front-end that generates an intermediate representation of the program and multiple back-ends which generate the code for the specified target.

In order for the tool to be able to largely test P4 compilers, it also needs to be target independent. As each target comes with specific limits and constraints, the challenge is to be able to generate P4 programs that can be compiled for the chosen architecture. While some of the particularities are specified in the package declaration of the architecture model, most of them are only implemented in the back-end compiler.

4.6 Recursion and Cycles in P4

In P4, data types and expressions can be nested upon declaration or usage according to the rules described in the specification of the language[10].

One example is the specialized type (e.g. extern objects) variable declaration for which the specialization is also another specialized type and so on. The generated code for this example can be seen in Listing 4.1. Lines 1-3 describe the declaration of an extern object type that is specialized on T. Line 6 shows how variable declarations for the declared extern object type can be nested indefinitely.

```
1 extern test_extern<T> {
2     test_extern();
3 }
4 test_extern<test_extern<test_extern<test_extern<bit<32>>>>> test;
```

Listing 4.1: Nested types in variable declarations

There are two types of recursion that can occur while generating P4 programs and they can be seen in the grammar of the language:

- Immediate recursion: Occurs when a production rule contains itself
- Cycle: Occurs when there is a chain of production rules that contains a cycle

The immediate recursion can be seen in the grammar of P4 for example in Listing 4.2 and as a derivation tree in Figure 4.1. Bolded code shows the recursion.

```
1 lvalue
2     : prefixedNonTypeName
3     | lvalue '.' member
4     | lvalue '[' expression ']'
5     | lvalue '[' expression ':' expression ']'
6     ;
```

Listing 4.2: lvalue grammar rule showing immediate recursion

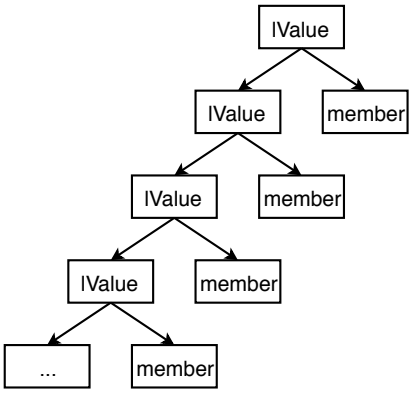

Figure 4.1: Immediate recursion shown as derivation tree of lValue

The cycle type of recursion can be seen in P4 grammar for example in the usage of type references as shown in Listing 4.3 and as derivation tree in Figure 4.2. Bolded code shows the recursion cycle.

```
typeRef
    : baseType
    | typeName
    | specializedType
    | headerStackType
    | tupleType
    ;
specializedType
    : prefixedType '<' typeArgumentList '>'
    ;
typeArgumentList
    : typeArg
    | typeArgumentList ',' typeArg
    ;
typeArg
    : DONTCARE
    | typeRef
    ;
```

Listing 4.3: P4 grammar showing cycles

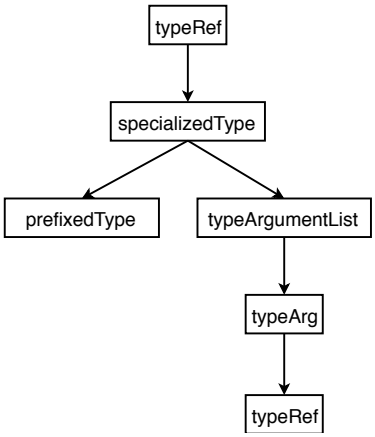

Figure 4.2: Cycles shown as derivation tree of typeRef

Nested types and expressions represent a challenge for the code generator component of a P4 compiler fuzzer because of the maximum recursion depth limit imposed by the programming language in which the generator is written.

In order to allow nested types and expressions in the randomly generated programs, the code generator has to limit the recursion depth.

Chapter 5

Design of the P4Fuzz Tool

This chapter describes and argues about the design decisions taken during the project. Section 5.1 analyses the challenges identified in Chapter 4 and presents the solutions that made it possible to address them. Section 5.2 reformulates the main types of fuzzing and defines which ones fit the best for the case of P4Fuzz. Sections 5.3.1 and 5.3.2 highlight the architectures considered, how they can be used to achieve our scope, as well as the implications. Section 5.4 explains how P4Fuzz is structured, focusing on each of the component's role: test case generator and tester; packet generator and tester; packet-out comparison and taming techniques.

5.1 Addressing the Challenges

The challenges identified in Chapter 4 are addressed within the next paragraphs, alongside short non-technical description of the solutions.

Generate Syntactically Valid P4 Programs

The challenge regarding generation of syntactically valid P4 programs described in Section 4.1 is addressed in the P4Fuzz design by reading through the grammar available in the specifications of the P4 language [10] and extracting the rules as 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.

Generate Semantically Valid P4 Programs

The solution for the challenge presented in Section 4.2 is to design the fuzzer tool in such a way that it tracks the available user defined types as well as the available variables, their types and if they have been initialized or not. This is done by saving information about data types and variables while the program gets generated.

Test P4 Programs Behaviour

Testing the P4 programs behaviour in P4Fuzz is made with the help of differential testing, a technique in which the same test case (P4 program) is verified using multiple compilers or back-ends. If the output of the different compilers on the same source code is different, the program gets marked as a potential bug.

Sort Faulty Test Cases Based on Error Relevance

The challenge of ranking the faulty test cases based on the relevance of the error is described in Section 4.4. Solving it can greatly improve the efficiency of the bug discovery process as it can help the users select the most interesting cases first.

Our solution to this problem is to group the errors that are very similar (trigger the same bug), such that we create 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.

Working with Different Architectures

As P4 is a target independent language, different architectures come with their own particularities such as limitations and mandatory structure. The fuzzer needs to be able to generate valid code based on the chosen architecture. The solution in P4Fuzz is to have different code generator wrappers based on what target is 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.

Recursion and Cycles in P4

As described in Section 4.6, nesting and recursion represent important challenges for any P4 compiler fuzzer. In P4Fuzz this challenge is addressed 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.

5.2 Fuzzing Strategies

Using the types of fuzzers described in Section 3.3 we designed P4Fuzz: a smart, black-box and generation-based fuzzer.

The fuzzer uses as input model the P4 language specifications to generate both syntactically and semantically valid programs. From this point of view, P4Fuzz is a smart compiler that is aware of the type of input that is producing (P4 programs) and follows the specifications. Taking the "dumb" approach would only stress the parser, covering a small portion of compiler's code.

Even though the black-box technique covers a smaller part of code, we chose this approach due to it being the faster one to implement and requiring less effort. In the earlier stages of development, the focus goes into implementing the base functionality such as generating test cases, compiling the test cases, generating packets for the programs, and interpreting the compiler's output. On top of this, we consider the decision of either black-box or white/grey box is not a permanent one; static analysis and instrumentation can be added later to the fuzzer as features, improving its quality.

The debate between generation and mutation based fuzzing favored the last one, producing the best results in the case of compilers. 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 would not eliminate all the errors, and it would require more time and effort. Moreover mutating given valid programs requires implementing a parser as well, which is beyond the scope of our project.

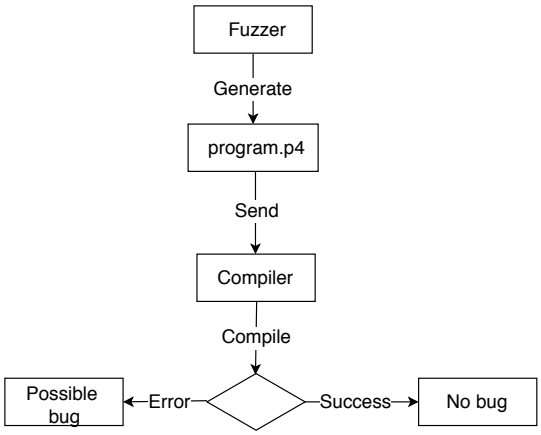

Figure 5.1: Basic fuzzer flow

Figure 5.1 presents a naïve activity flow of the fuzzer. The process starts with the generation of the P4 programs, that are sent to the compiler. Once the compilation is done, the output is interpreted and it is decided if the generated program found a possible bug in the compiler or not. A more concise and elaborate description of how the P4Fuzz approaches the fuzzing process can be found in Section 5.4.

5.3 Tailoring the Fuzzer to Specific Targets

As described in Section 4.5, a good fuzzer tool is capable of testing P4 programs on multiple architectures using different back-end compilers.

Because of the particularities each architecture has, the programs need to follow a specific structure included in a P4 architecture model description file. The model

file is an interface that specifies the controls that need to be implemented and the external objects and functions that are available on the device.

As the structure of the program is different based on the architecture used, the fuzzer needs to be tailored for each target.

5.3.1 BMv2

The Behaviour Model version 2 (BMv2) model requires that the program includes the v1model.p4 architecture model description file. This file contains P4 code that specifies the minimum requirements for a P4 program to be able to run on BMv2.

Parser

The program needs to implement a parser with the following parameters: packet in, parsed header, metadata and the standard metadata as shown in Listing 5.1.

```
1 parser Parser<H, M>(packet_in b,
2                     out H parsedHdr,
3                     inout M meta,
4                     inout standard_metadata_t standard_metadata);
```

Listing 5.1: Parser declaration in the v1model interface

Deparser

Together with the parser, a deparser is also needed in order to construct the packet out. The deparser only needs to have the packet out and the header parameters as seen in Listing 5.2.

```
1 control Deparser<H>(packet_out b, in H hdr);
```

Listing 5.2: Deparser declaration in the v1model interface

Pipelines

As the BMv2 has two pipelines, one ingress and one egress, they have to be implemented as controls. Both pipelines have the same parameters as input: the header, the metadata and the standard metadata as seen in Listing 5.3.

```
1 control Ingress<H, M>(inout H hdr,
2                       inout M meta,
3                       inout standard_metadata_t standard_metadata);
4
5 control Egress<H, M>(inout H hdr,
6                       inout M meta,
7                       inout standard_metadata_t standard_metadata);
```

Listing 5.3: Pipelines declaration in the v1model interface

Checksums

Two additional controls are required by the BMv2 target: one that computes the checksum and another one that verifies that the checksum on the packet is correct. They both take as paramets the header and the metadata as seen in Listing 5.4.

```
control VerifyChecksum<H, M>(inout H hdr, inout M meta);
control ComputeChecksum<H, M>(inout H hdr, inout M meta);
```

Listing 5.4: Checksum controls declaration in the v1model interface

Externals

The most important external objects and functions available in the BMv2 are: the counter object type, the meter object type, the register object type, the verify checksum function and the recirculate function. The purpose of the counter type is to be able to count different aspects, while the register is used to store and access any type of data quickly. The verify checksum extern function can be called to verify the checksum of the data based on the chosen algorithm. If the checksum is invalid it will set a checksum_error bit on the standard metadata. The declarations for the external object types and functions can be seen in Listing 5.5.

```
extern counter {
    counter(bit<32> size, CounterType type);
    void count(in bit<32> index);
}
extern register<T> {
    register(bit<32> size);
    void read(out T result, in bit<32> index);
    void write(in bit<32> index, in T value);
}
extern void verify_checksum<T, O>(in bool condition, in T data, inout O
    checksum, HashAlgorithm algo);
extern void recirculate<T>(in T data);
```

Listing 5.5: External object types and function available in BMv2

In order for the test cases to compile on the BMv2 target, all the above described parser, deparser and controls need to be implemented. This is why the fuzzer needs to make sure, when generating the random code, that these aspects are implemented. This is achieved by having a wrapper for each of the targets that enforce these requirements. In the case of BMv2, the wrapper for the test case generator outputs the randomly generated code in the appropriate positions in relation to the general structure of a valid BMv2 program.

5.3.2 eBPF

Similar to BMv2 model, eBPF requires that the program includes the architecture model description file, in this case ebpf_model.p4. This file contains P4 code that specifies the minimum requirements for a P4 program to be able to run on eBPF.

Parser

The program needs to implement a parser with the following parameters: packet in and parsed header as shown in Listing 5.6.

```
parser parse<H>(packet_in packet,
                out H headers);
```

Listing 5.6: Parser declaration in the eBPF interface

Pipelines

The eBPF model has one mandatory pipeline (the filter) that takes as parameters: the inout headers and the out bool accept as shown in Listing 5.7

```
control filter<H>(inout H headers,
                  out bool accept);
```

Listing 5.7: Filter declaration in the eBPF interface

Externals

The most important external objects available in the eBPF are: the counter array object type, the array table object type and the hash table object type. The declarations for the external object types can be seen in Listing 5.8.

```
extern CounterArray {
    /** Allocate an array of counters.
     * @param max_index  Maximum counter index supported.
     * @param sparse     The counter array is supposed to be sparse. */
    CounterArray(bit<32> max_index, bool sparse);
    /** Increment counter with specified index. */
    void increment(in bit<32> index);
}

extern array_table {
    /// @param size: maximum number of entries in table
    array_table(bit<32> size);
}

extern hash_table {
    /// @param size: maximum number of entries in table
    hash_table(bit<32> size);
}
```

Listing 5.8: External object types available in eBPF

In order for the test cases to compile on the eBPF target, the above described parser and control need to be implemented. This is why the fuzzer needs to make sure, when generating the random code, that these aspects are implemented, similar to the BMv2 architecture.

5.4 P4Fuzz Components

Figure 5.2 presents an overview of the P4Fuzz structure, how the components connect with each others, and what are the inputs, outputs and decisions made at specific points.

5.4.1 Overview of the P4Fuzz Components

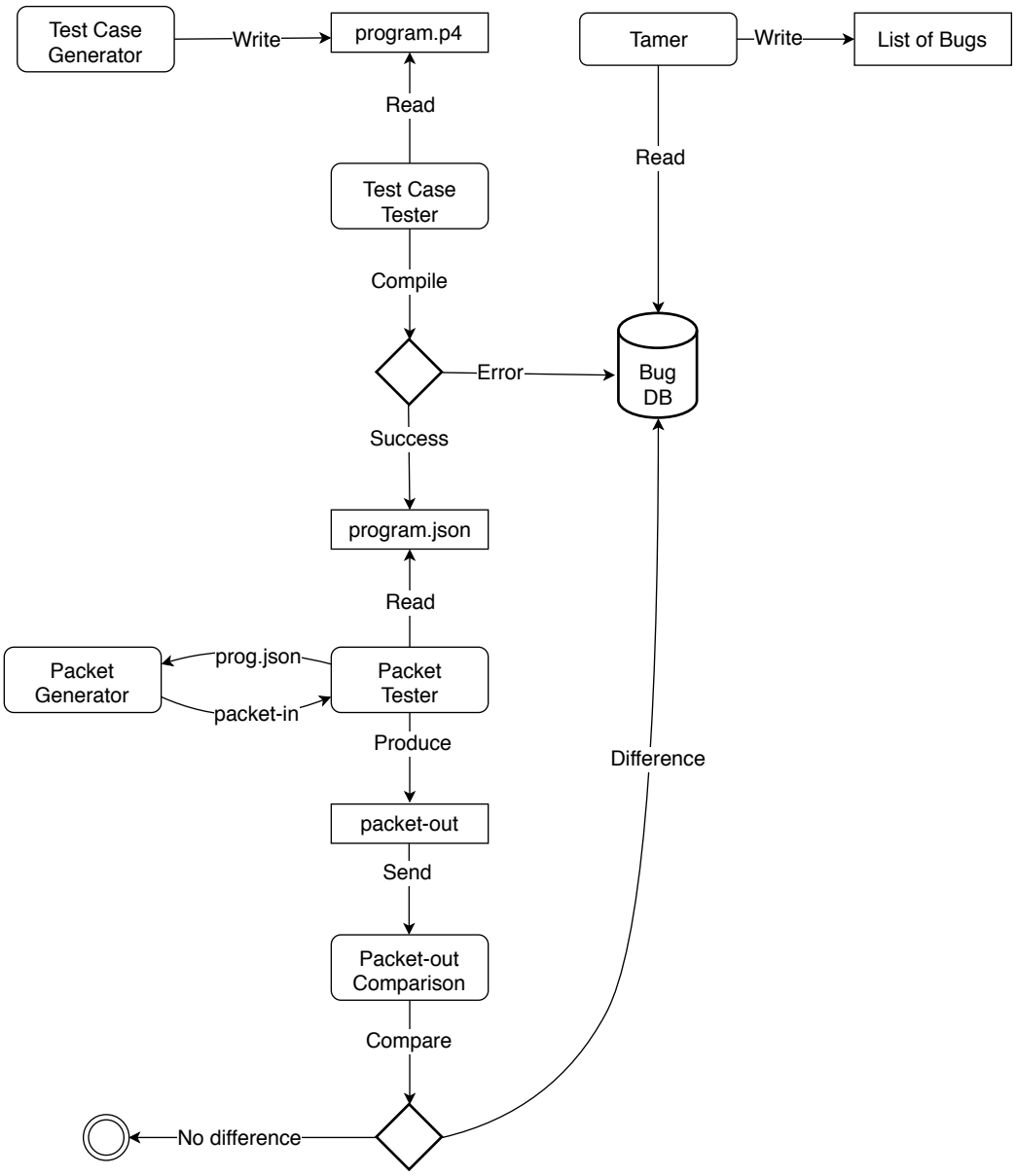

Figure 5.2: P4Fuzz flow

First step of the fuzzing process starts with the Test Case Generator. This module writes random P4 programs, each one in its own file, and saves them into a folder. After a few P4 programs have been written, the Test Case Tester can begin to select and process these programs. Because the generation process is much faster than the process one, it is safe to execute the Test Case Generator and Test Case Tester in parallel, since it will never run out of test cases.

Once the Test Case Tester reads a P4 program from the ones generated earlier, the test case is compiled and the output is checked for errors. Details about the programs that throw errors at compilation are saved into a Database of Bugs. Successfully compiled programs generate JSON files (program.json) which are saved in to a folder similar to the test cases one.

P4 programs that did not throw an error at compilation are 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 (p4pktgen tool) crafts the packets based on the program.json deployed on the switch. The Packet Tester module then reads each of these packets as packet-in and outputs the packet-out.

On the final stage of the process, the Packet-out Comparison checks if there are any differences between the packets-outs of distinct switches (i.e: Simple Switch, eBPF). If there are any differences between distinct architectures for the same P4 program and the same packet in, the test case is market as 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 is the one reading bugs saved into the database, sort them using a defined distance functions such that the most interesting and different ones are ranked higher, and prints them out to the user.

5.4.2 Test Case Generator

The Test Case Generator module of the P4Fuzz tool is the component that is in charge of randomly generating P4 programs (the test cases) for the rest of the components.

As explained in Sections 4.2 and 5.2, P4Fuzz is a smart compiler fuzzer because the generation of the programs is made in such a way that it follows the P4 language specifications [10], including the grammar and the semantic rules.

Target Independence

Because the P4Fuzz is also target independent, as described in Section 4.5, the generator is using different wrappers that enforce the target's limitations and required structure of the program, to generate valid code that follows the architecture specifications. The process is shown in Figure 5.3.

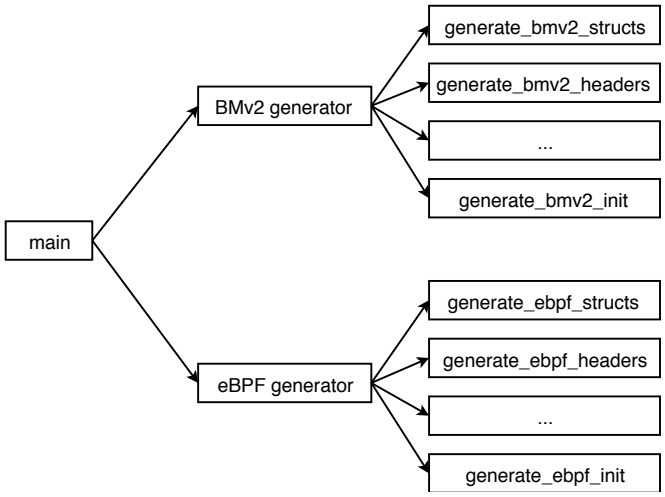

Figure 5.3: Code generation process is based on specified target

To make sure we generate syntactically valid code, our approach is to manually program production rule classes extracted from the grammar rules. Using these classes, the generator is able to randomly create a valid derivation tree used to generate valid P4 code.

Enforcing Semantic Rules

As there is no formalization of the semantics of P4 available, the semantic rules were extracted for each of the supported grammar rules, by reading through the Language Specifications [10] and implementing them as filters.

Filters in P4Fuzz are the way of specifying requirements that have to be met 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.

Specializing the Fuzzer Using Probabilities

Another strategic decision made was to include the concept of probability for a specific production to be selected. In this way, 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.

The probabilities have some initial values by default but they can be changed by the users to better reflect their wishes. The probabilities are given as a number ranging from 0 to 100, representing the chance of occurrence as a percentage. The random generation process follows the probabilities when choosing the next production.

Keeping Track of the Context

As explained in Section 4.2, the fuzzer has to keep track of the context while it generates the program, a requirement needed for enforcing the semantic rules. P4Fuzz keeps track of the content by recording the variable and data type declarations in a stack of scopes. The way the process works is the following:

1. a new scope is created and pushed in the stack when a new block is generated
2. the declaration's name, type and other relevant information is inserted into the current scope when a new variable or type is declared
3. the scope is removed when the randomization of that block is finished

Optimizing the Randomization Process

We decided to make the randomization process as flexible as possible and provide support for other libraries as well. This gives the opportunity to futures developers to change this aspect of the fuzzer and configure it as they please.

Another decision we made is to use a seed based randomizer and save the randomly generated seed as a comment in the beginning of each test case file. This way the test case generation can be replicated, which helps recovering lost test cases or debugging the bugs in the fuzzing process.

Dealing with Nesting and Recursivity

To deal with the recursivity challenge described in Section 4.6, the production rules that lead to immediate recursion or cycles, are protected by limiting the recursion depth level to a maximum. In this way we make sure the generation process does not enter an infinite loop and also does not exceed the limit of the maximum recursion depth imposed by the language of implementation.

Overview of the Test Case Generation Process

The test case generation process is done in multiple consecutive steps as follows:

1. The generator is given a target (e.g. BMv2) and it calls the specific wrapper for that architecture in order to randomly generate the program
2. The wrapper generates different parts of the program (e.g. headers, structs, externs, parsers, controls) by creating the objects of the specified types and applying a randomization method on them.
3. Each object representing a production rule randomly selects a production based on the probabilities specified for each of its available productions. It then creates an object of the selected production rule which also gets to be randomized.

4. A filter checks if the selected production is valid or not
   4.1. If the selected production is not valid, it retries to select and randomize another production.
   4.2. In the case of the rules that can lead to recursivity, the maximum recursion level allowed is also checked.
5. When the rule is introducing a new block, a new scope is created and inserted in a stack of active scopes.
   5.1. If the current rule is declaring types or variables, they are inserted into the current active scope.
6. When the randomization process ends, the derivation tree is complete and the wrapper starts generating the code.
   6.1. Each node in the derivation tree is generating code for itself.
7. The wrapper concatenates all of the generated code pieces into a P4 program representing the test case.

Automation of the Generation Process

The P4Fuzz includes a script for automation of the test case generation process. The script generates test cases using the Test Case Generator module and saves each of them with a filename composed of 10 characters representing the test case number (e.g. 0000001234.p4).

Because the number of generated cases per unit of time is an important aspect of a fuzzer, multiple instances of the script can run in parallel as the processes do not interfere. This allows spawning multiple test case generation processes and make use of the multiple cores available on the machine. Figure 5.4 illustrates this aspect by showing how two generators save different test cases using different test case numbers.

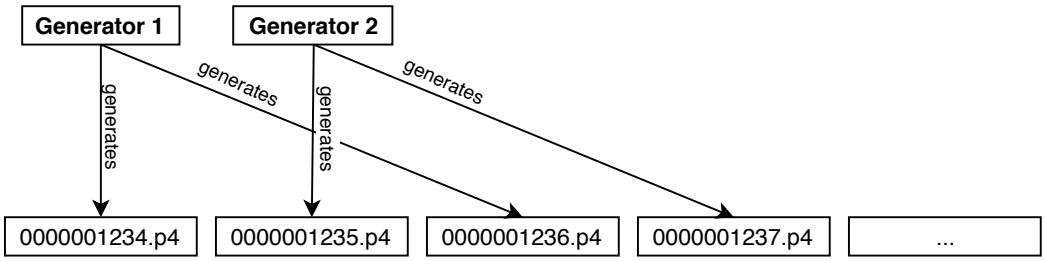

Figure 5.4: The Test Case Generator module can run as multiple processes each generating a different test case

5.4.3 Test Case Tester

The Test Case Tester module's role in the P4Fuzz context is to verify if the randomly generated P4 program (test case) can be successfully compiled for a specified target. Assuming the packet is valid, we mark as potential bug any program that rejects the packet or throws an error.

As P4Fuzz runs it finds many of these interesting test cases, which represent potential bugs, and need to be stored in order to be manually investigated later. The Test Case Tester saves the interesting test cases, for which the compilation has failed, in two locations. The P4 program source code is saved into an errors folder as testcasenumber.p4 (e.g. 0000001051.p4), while the error message from the compiler is saved as a testcasenumber.err (e.g. 0000001051.err) file in the same errors folder. Additionally, the tester saves the test case number, error message, filepath of the source file and the randomly generated seed of the test case in a database for easier data management.

Because the P4Fuzz tool is designed to be target independent, the tester module is capable of checking P4 programs on multiple targets. According to the target selected, different compiler back-ends are used.

As the power of a fuzzer lies in the large number of test cases that it can generate and check in a short amount of time, it is important that the tester module can verify many P4 programs as quickly as possible. For this reason, the tester is designed such that it can be spawned multiple times without the processes interfering with each other, as seen in Figure 5.5, thus enabling work to be done in parallel on multiple CPUs.

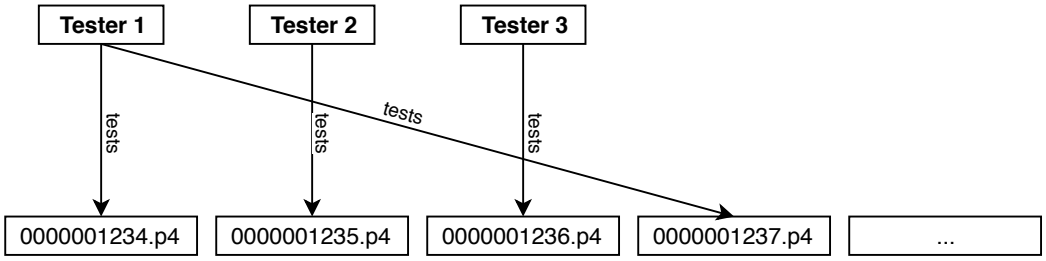

Figure 5.5: The Test Case Tester module can run as multiple processes each testing a different test case

5.4.4 Packet Generator

The Packet Generator module is using P4pktgen to craft packets for a given P4 program. The packets are generated using symbolic execution and can be used to check if a given P4 program acts as intended on a device [3].

Within the P4Fuzz, the Packet Generator takes as input the JSON files resulting from the compilation of P4 programs randomly generated, and outputs packets for the given test case. P4pktgen complements our work by providing a way to test the

generated programs with valid inputs, ensuring that most of the paths are explored by using symbolic execution.

P4pktgen is the state of the start, but also a novelty within P4 domain, without which we would had to either limit to only compilation of P4 programs, or write our own packet generator. Writing a packet generator is however a project of its own, thus making use of its features makes more sense for the given scenario.

5.4.5 Packet Tester

The Packet Tester module is inspired by the P4pktgen in the sense that is able to test packets for a given program. However, compared with the P4pktgen that is able to run the given program only on simple switch (using –run-simple-switch command), the Packet Tester can support multiple architectures i.e: simple switch, eBPF.

This module takes as input the compiled P4 program in JSON format, the table entries, the packet, and the target. While the first three parameters are provided by the Packet Generator, classes for each of the architectures have to be written in order to be supported. The project structure however makes it easy to add such class, since the Packet Tester is designed as a wrapper around them.

The base concept behind the Packet Tester is that we first start the switch, then we populate the tables with the entries provided by the generator, and finally we send the crafted packet. The packet-out is then returned and used later in the fuzzing process to compare it with other packets.

5.4.6 Packet-out Comparison

The Packet-out Comparison module of the P4Fuzz is the component responsible with comparing the packets emitted by different platforms (e.g. BMv2 and eBPF). Considering the P4 programs which run on the different tested devices are equivalent, using the same input packet for all the platforms should emit identical packets, unless timestamps or other non-deterministic field values are present.

The comparison module checks if the output packets are identical or not between the tested targets. Assuming that the programs are equivalent, if the packets are different, it means a new potential bug has been discovered. The test case assets such as the P4 programs involved, the packet-in and the output packets are saved on disk and a new entry in the potential bugs table in the database is created documenting the issue.

5.4.7 Taming

Once the programs have been checked for syntax and run-time errors, and they are saved in the database we can start interpreting these errors. However interpreting hundreds or thousands of errors produced by random generated cases can be a tedious work, as described by [7] especially if the same errors occur more often.

Taming the compiler errors is one of the last actions within our fuzzing process, alongside the Packet-out Comparison. The taming module's role is to order the bugs found in the earlier stages, such that the diverse and interesting test cases are ranked higher [7]. In this way the compiler developer can easily identify a larger number of bugs without spending time on duplicates.

Inspired by [7] we try to find an approximate solution to the taming problem ("given a potentially large number of random test cases that trigger failures, order them such that diverse, interesting test cases are highly ranked" [7]) using two approaches:

- furthest point first (FPF) sorting using Levenshtein distance between each pair of errors
- k-medoids clustering using a distance function which measures the number of different tokens/words between the two error messages

Following subsections describe each distance function used, their challenges and outcome. The Levenshtein distance is a very common function used to describe how different two given strings are by comparing their characters. This is one of the distance functions suggested by [7], and the one that produced successful results in their evaluation even though a strong conclusion about it being the best distance function for taming could not be formulated by the authors.

Another recommendation is that the programs that produce errors should be "reduced", thus minimizing the source code by deleting unused statements. Such tools exist for C and JavaScript (which is the case for [7]) but not for P4, therefore we can only consider working with the "raw" source code as produced by the Test Case Generator.

However we consider the reduction would not provide great improvement, as we calculate the Levenshtein function only on the compiler output rather than program source code. The only possible advantage given by the reduction in our case would be related to the errors that include variable/functions names; by reducing the random generated names to a minimum, the distance between errors could be more accurate since the focus is on the error itself rather than long variable names.

FPF on Levenshtein distance

We implemented the taming using Levenshtein distance function based on description provided by [7] and shown in Figure 5.6. The formula calculates the distance between two strings a and b. The length of the strings are defined as $|a|$ and $|b|$. The indicator function $1_{(a_i \neq b_j)}$ returns 0 if character i of string a is the same as character j of string b, or 1 otherwise. On the *min* case of the function, each branch corresponds to a specific action: first branch is deletion, second branch is insertion and third branch is match/mismatch [18].

$$\mathrm{lev}_{a,b}(i,j) = \begin{cases} \max(i,j) & \text{if } \min(i,j) = 0, \\ \min \begin{cases} \mathrm{lev}_{a,b}(i-1,j) + 1 \\ \mathrm{lev}_{a,b}(i,j-1) + 1 \\ \mathrm{lev}_{a,b}(i-1,j-1) + 1_{(a_i \neq b_j)} \end{cases} & \text{otherwise.} \end{cases}$$

Figure 5.6: Levenshtein distance [18]

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. As stated in the paper [7], the most effective distance function differs from compiler to compiler, and a general purpose one does not exist. For our case this happens due to the fact that the Levenshtein function checks the strings based on characters, and is unable to differentiate between variable names and error description.

A second problem identified for FTF on Levenshtein distance algorithm is based on the complexity of the Levenshtein distance. In fact, the time complexity

$$O(len(string1) * len(string2))$$

made 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 1 million operations. For example, the built-in Levenshtein function in PHP limits the string length to only 255 characters.

An alternative to calculate the Levenshtein distance between each cases every time the algorithm runs is to calculate the distances between existing errors and the one currently being inserted in the database. Thus, the algorithm would be able to run FPF on the pre-calculated distances. This solution has a space complexity of n^2, where n is the number of test cases.

K-medoids on token-based distance

Inspired by [7] and the FPF on Levenshtein distance, we defined a distance function based on tokenized error messages. Listing 5.9 show the pseudo-code for the token-based distance function defined by us.

```
1  int tokens_distance(s1, s2)
2  {
3      int init_dist, dist;
4
5    s1_tokens[] = s1.tokenize(' ')
6    s2_tokens[] = s2.tokenize(' ')
7
8    init_dist = length(s1_tokens + s2_tokens)
9    dist = length(s1_tokens + s2_tokens)
10
11   for iterator i in s1_tokens {
```

```
12          for iterator j in s2_tokens {
13              if j == i {
14                  dist = dist - 2
15                  j = s2_tokens.remove(j)
16                  break
17              } else {
18                  j++
19              }
20          }
21      }
22
23      return (dist / init_dist) * 100
24  }
```

Listing 5.9: 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

Even though this approach has a similar complexity, the number of tokens is in worst case is the same as the number of characters in strings (i.e: all words in the string contain only one character) which makes it faster to use in practice.

Another advantage is that it handles better the cases where the compiler output contains large parts of random 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).

Chapter 6

Implementation

This chapter presents the technical details related to P4Fuzz including the programming language, libraries, tools and specifications of the architectures used. The second part of the chapter discusses each main component described in Section 5.4: the Test Case Generator, Test Case Tester, Packet Generator, Packet Tester and Compiler Fuzzer Taming. The scope of the chapter is to highlight how these components are implemented, as well as arguing for the technical decisions made.

6.1 Overview of Technologies Used

This section presents the technologies used for the implementation of P4Fuzz including the programming languages, tools and libraries. While Python is the most used language within the modules of the fuzzer, other components are written in C++ as it is the case of more complex algorithms, or PHP for data display. Tools such as PyClustering, SWIG and Slim provide the resources for sorting algorithms, communication between Python and C++, or the user interface.

6.1.1 Programming Languages

This section describes and explains the decisions made regarding the programming languages used for the implementation of different components of P4Fuzz. Table 6.1 shows statistics about the languages used in terms of number of files as well as number of blank lines, comments and lines of code. These numbers were obtained using CLOC (Count Lines of Code) [19], a tool for counting lines of code which supports many of the most popular programming languages.

Table 6.1: Statistics on programming languages used

Language	Files	Blank	Comment	Code
Python	167	1691	794	6694
PHP	7	63	68	238
C	1	20	15	139
Bourne Shell	2	26	3	121
C++	1	5	0	79
HTML	1	6	0	51
SQL	1	2	0	30
C/C++ Header	1	1	0	7
CSS	1	0	10	1
TOTAL	182	1814	890	7360

Python

As shown in Table 6.1, Python is by far the most used programming language in P4Fuzz. Most of the components of our tool (the test case generator, test case tester, packet generator and the packet tester) are written in Python.

The reasons behind choosing Python are the following:

- **Productivity:** being an easy language to learn and use, Python provides great speed of development by achieving more while writing less code.
- **Third party libraries:** The community around Python is large and the support for well known but not standard algorithms and techniques is provided by the large number of third party libraries available.
- **Support:** Python is supported on multiple architectures and platforms meaning we can develop and use the tool on both Windows and UNIX.

Python also has disadvantages out of which the most relevant for our project are: being an interpreted language it is slower than compiled languages and being a dynamically typed language the errors only show up at runtime.

Regarding the slow speed of execution, we got over this disadvantage by implementing the most expensive algorithms in a small Python library written in C++.

As for the errors at runtime, because each component is programmed separately, the fuzzing process does not stop at the first error it encounters but it moves over that specific testcase and continues the process. It is not a big issue for the scope of our project since P4Fuzz components run a large number of times and debugging can be easily done even while the tool runs.

C++

Parts of the P4Fuzz tool implementation, such as expensive algorithms (e.g. calculating the distances between each error message) need to be written in a language

oferring a high speed of execution.

C++ was chosen because it offers great performance. This is possible both because it is compiled directly into binaries and it allows the programmer to make decisions on each aspect, such as using pointers instead of copying objects around.

The main disadvantages for C++ are: allowing programmers to deal with memory management can lead to memory leaks and because of its complexity, writing programs in C++ is not as productive as in other languages. As we are only using it for implementing the most expensive algorithms, neither of these disadvantages are affecting us.

In the scope of P4Fuzz tool, C++ is used for calculating the distance matrix for the errors (since the complexity of that is $O(n^2)$ with n being the number of errors). It is also used for calculating the token-based distance between two errors (as the complexity of that is $O(n * m)$ with n and m being the number of tokens in the first and second string that are compared).

PHP

The tasks of P4Fuzz are: to find potential bugs in P4 compilers and to display them in a way such that the users can easily investigate as many distinct test cases as possible. For ease of access and increased readability, P4Fuzz displays the errors on a web page.

While Python can also be used in relation to web development, the process is complex and the support is not very good. Thus the presentation of the potential errors found is made using a web page that uses PHP as the back-end programming language.

PHP was chosen because it is well known and used in the context of web development and numerous frameworks exist for increased productivity. As shown in Table 6.1, with only 238 lines of code we have managed to create a web interface that retrieves the errors from the database and displays them according to the sorting and grouping managed by the taming process.

6.1.2 Libraries and Tools

This section presents the most important libraries and tools used in P4Fuzz together with explanations on how are they used and why they were chosen.

PyClustering

Since P4Fuzz is applying taming as a technique for grouping and sorting of failed test cases based on the error messages, the tool requires the use of clustering algorithms.

PyClustering [20] is a data mining Python library written in C++. It implements lots of the well known clustering and neural network algorithms such as K-Means, K-Medoids (PAM) or CNN (Chaotic Neural Network).

This library was chosen because it implements the algorithm required by the P4Fuzz (K-Medoids) as well as it being written in C++, achieving good performance in terms of speed. Listing 6.1 shows how the K-Medoids algorithm can be applied using the PyClustering library in order to cluster any data.

```
#import the kmedoids algorithm from the pyclustering library
from pyclustering.cluster.kmedoids import kmedoids
# initialize the kmedoids object using pre-computed distance matrix and
    some randomly chosen initial medoids
kmedoids_instance = kmedoids(distance_matrix, initial_medoids data_type='
    distance_matrix')
# process the data
kmedoids_instance.process()
# retrieve the clusters
clusters = kmedoids_instance.get_clusters()
# retrieve the medoids for each cluster
medoids = kmedoids_instance.get_medoids()
```

Listing 6.1: Usage of PyClustering library for clustering data using the K-Medoids algorithm

SWIG

As described in Section 6.1.1, some algorithms were implemented in C++ rather than Python for better performance. While written in C++ the algorithms are still required to be used from the Python code. To achieve this, a wrapper library which facilitates the communication between the main program and the C++ library is required.

SWIG [21] is a tool that automates the process of enabling the usage of C or C++ written libraries in most popular programming languages among which Python is also supported.

The way SWIG works is by defining an interface between C/C++ and the targeted programming language. This interface defines the headers of the available functions as well as mappings for the more complex data types. An example of such an interface can be seen in Listing 6.2.

```
%module p4fuzzclib

%{
#define SWIG_FILE_WITH_INIT
#include "p4fuzzclib.h"
%}

%include "std_string.i"
%include "std_vector.i"

%template(vector_string) std::vector<std::string>;
%template(vector_integer) std::vector<int>;
%template(vector_vector_integer) std::vector< std::vector<int> >;

int common_tokens_distance(std::string s1, std::string s2);
```

```
16 std::vector< std::vector<int> > calc_distance_matrix(std::vector<std::
       string> data);
17 std::vector<int> calc_max_distance_cluster(std::vector< std::vector<int> >
        dist, std::vector<int> cluster);
```

Listing 6.2: SWIG interface for enabling usage of C++ library in Python

SWIG was chosen as a tool for pairing the C++ implemented library for P4Fuzz (P4FuzzCLib) with the rest of the P4Fuzz components, because it is one of the easiest ways of achieving such a task while not reducing the performance. The main disadvantage for SWIG is that the support for more complex C++ types and structures is not great, but for the purpose of P4Fuzz, this does not constitute an issue, as P4FuzzCLib is only using types defined in the standard library.

Slim Framework

P4Fuzz is displaying the potential bugs found, in a clean and accessible way, by implementing a web interface. As described in Section 6.1.1, the web pages are written in PHP for which numerous frameworks exist.

Slim Framework [22] is a minimalist PHP framework for fast development of web applications. It offers well tested means for implementing web interfaces while also being lightweight compared to other more powerful and complex frameworks such as the Zend Framework or Symfony.

An example on how a route can be defined and implemented is given in Listing 6.3.

```
1  // Defining the cases route from the main app object
2  $app->get('/cases', function ($request, $response, $args) {
3      // Creating the database mapper for the test cases
4      $mapper = new db\TestCaseMapper($this->db);
5      // retrieving the clustered test cases from the database
6      $clusteredCases = $mapper->getClusteredTestCases();
7      // Calling the view component of the framework to render a response
       using the test-cases.html template and clusteredTestCases data
8      return $this->view->render($response, 'test-cases.html', [
9          'clusteredTestCases' => $clusteredCases
10     ]);
11 })->setName('cases');
```

Listing 6.3: Test cases route implementation using Slim Framework

Slim Framework was chosen in the context of P4Fuzz because it is a very accessible, lightweight PHP framework which is easy to learn and use. It is required in order to develop a web application needed for showing the potential bugs found by the P4Fuzz tool.

Dependencies

Because of its complexity, P4Fuzz has lots of dependencies on other software. Table 6.2 shows the most important requirements for running the compiler fuzzer.

Table 6.2: P4Fuzz dependencies

Dependency	Note
mysql-server	Used for storing the potential errors generated by the fuzzer
Python 2.7	Needed for running P4Fuzz
PHP7	Required for displaying the potential errors on a web platform
Apache2	Used as the web server for the web platform
Composer	Used for installation of Slim Framework and it's dependencies
BMv2	Needed for testing packets using the simple_switch target
Protobuf	Required by P4C compiler
P4C	The compiler that we are fuzzing
PyClustering	Used for taming the fuzzer
mysql_connector for Python	Used as a client for accessing the database server
pyroute2	Required by eBPF packet tester
p4pktgen	Needed for generating packets from P4 programs
SWIG	Required for accessing C++ written modules from Python progams

6.2 BMv2 Specifications

BMv2, which is an abbreviation for Behavioural Model version 2, is a framework for implementing P4 programmable software switches with the purpose of modelling the behaviour of other networking appliances [1]. The framework allows vendors to run P4 programs on software switches that model the behaviour of their own physical devices [1]. It also serves the purpose of letting the P4 language developers demonstrate the capabilities of the language in a more practical way [1].

Even though the BMv2 and the simple_switch are implemented in C++, the P4 programs are compiled into JSON files, which are then interpreted by the switch [1]. This is opposed to the first version of the framework in which the P4 programs were compiled into C++. Compilation into JSON structures enables fast development of programs because it allows changes to be deployed more easily without the need to re-compile and restart the entire switch [1]. The compilation command for the BMv2 simple_switch can be seen in Listing 6.4.

```
1 p4c-bm2-ss source_file.p4 -o target_file.json
```

Listing 6.4: Compilation command for the BMv2 simple_switch

The simple_switch is an implementation of the abstract model presented in the P4 Language Specifications document[10]. Figure 6.1 shows an overview of the abstract model as it is described in the specifications.

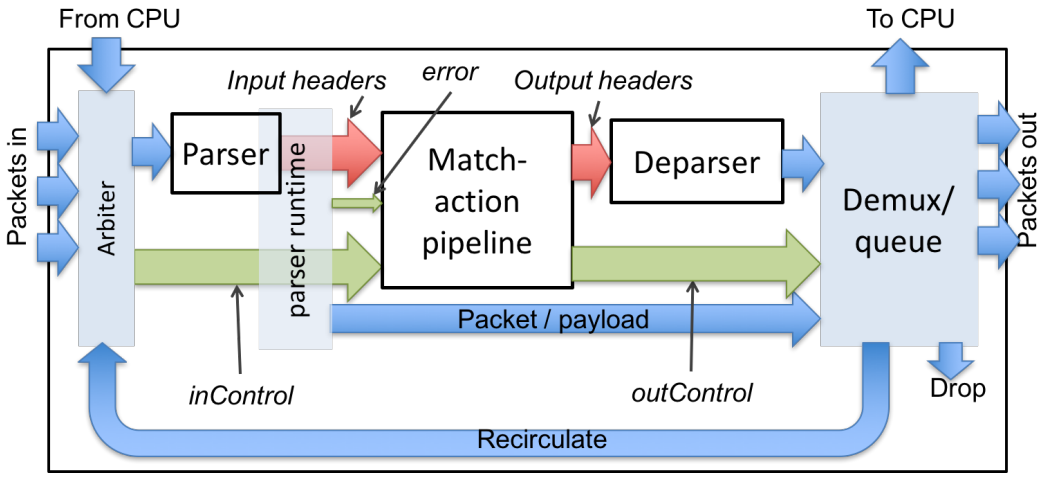

Figure 6.1: The simple_switch architecture [10]

It can be seen in Figure 6.1 that the simple_switch has multiple input ports, some standard and two special ones: the recirculate and the CPU port [1]. Packets coming from these ports are then parsed by the parser, processed by the match-action pipeline and then deparsed to form out-packets [1]. The outgoing packets are then emitted through either the standard packet-out ports, the CPU port, the drop port or the recirculate port [1].

The switch can be controlled (e.g. adding or removing entries in the tables) using CLI commands, using a software defined networking controller such as ONOS or programatically using Apache Thrift, an interface definition language and communication protocol. Because the packet tester module, which facilitates differential testing in P4Fuzz, needs to be able to control the switch directly, the Apache Thrift solution was chosen. Listing 6.5 shows how the packet tester module connects to and controls the simple_switch using Thrift.

```
# Connecting to the switch using thrift
standard_client, mc_client = thrift_connect(
                    'localhost',
                    str(self.thrift_port_num),
                    RuntimeAPI.get_thrift_services(pre))
# Adding a new entry in a table using the runtime_CLI provided by the
    p4pktgen
standard_client.bm_mt_add_entry(
            0, table_name, match_key, action_name, runtime_data,
            BmAddEntryOptions(priority = priority))
```

Listing 6.5: Connecting and controlling the the simple_switch using the Thrift protocol

For the purpose of this project, the P4Fuzz tool is using the simple_switch as one of the available targets, both for compilation of the randomly generated P4 programs, as well as for performing differential testing between multiple targets.

The reasons behind choosing the BMv2 simple_switch, include the fact that it is

developed by the same organization behind the P4 language, namely the P4 Language Consortium, as well as being one of the few switches for which a compiler back-end is available.

6.3 eBPF Specifications

Compared to the BMv2, the eBPF architecture requires one extra step in order for the P4 program to be executed. While the JSON file generated by p4c-bm2-ss can be sent directly to the switch, a direct compiler from P4 to eBPF does not exist. As shown in Figure 6.2, the p4c compiler outputs a C program that can than be used by the BPF Compiler Collection (BCC) to produce the intended eBPF.

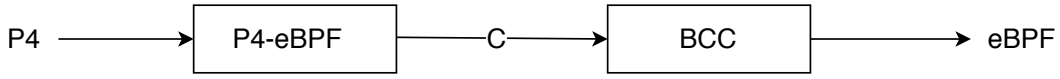

Figure 6.2: P4 to eBPF

Listing 6.6 shows the command ran on the P4C compiler to generate the C representation of the input P4 program.

```
p4c-ebpf -o program.c program.p4
```

Listing 6.6: p4c-ebpf compile command

Once the C file is created, we use the BCC python library to create a BPF object that hooks the program to a kernel function, and then add a filter to the specified interface as shown in Listing 6.7:

```

#!/usr/bin/env python3
from bcc import BPF
from pyroute2 import IPRoute
import time

b = BPF(src_file="valid_ebpf.c", debug=0)
fn = b.load_func("ebpf_filter", BPF.SCHED_CLS)
counter = b.get_table("pipe_counters")


ipr = IPRoute()
interface_name="enp0s3"
if_index = ipr.link_lookup(ifname=interface_name)[0]

ipr.tc("add-filter", "bpf", if_index, ":1", fd=fn.fd, name=fn.name, parent
    ="ffff:", action="ok", classid=1)


while True:
        try:
                print(counter)
```

```
22                 for k in counter.keys():
23                         print(k)
24                         print('counter k de value')
25                         print(counter[k].value)
26                 time.sleep(1)
27         except KeyboardInterrupt:
28                 print("Removing filter from device")
29                 break
```

Listing 6.7: BCC load C program

Finally, we send the packets generated by the Packet Generator module to test the eBPF filter, and we interpret the output.

6.4 Test Case Generator

One of the most important modules of the P4Fuzz tool is represented by the test case generator, the component responsible for randomly generating valid P4 programs. The following sections describe the important aspects of the generator.

Target Independence

P4Fuzz is target independent in the same way the P4 compiler is. This means that it is capable of testing different back-ends for multiple architectures. As described in Section 5.4.2, the implication regarding the implementation of the test case generator is that each supported target has its own generator wrapper in charge of producing valid test cases for the specified architecture.

```
1  def generate(self):
2    scope.start_local()
3    code = ""
4    code += self.output_seed_comment()
5    code += self.generate_bmv2_includes()
6    code += self.generate_bmv2_structs()
7    code += self.generate_random_structs()
8    code += self.generate_random_headers()
9    code += self.generate_random_externs()
10   code += self.generate_extern_variables()
11   code += self.generate_bmv2_parser()
12   code += self.generate_random_parsers()
13   code += self.generate_bmv2_controls()
14   code += self.generate_random_controls()
15   code += self.generate_bmv2_init()
16   scope.stop_local()
17   return code
```

Listing 6.8: BMv2 wrapper for the test case generator

Listing 6.8 shows how the generate method of the BMv2 wrapper is able to enforce the structure and limitations required for the BMv2 simple_switch. Line 2 starts

the global scope while lines 3-15 generate each part of the program. In the end on line 16 the scope is ended and the code is returned in line 17.

Each of the methods in lines 4-15 from Listing 6.8 are responsible for randomly generating parts of the abstract syntax tree as well as generate the code from the tree.

Listing 6.9 stands as an example of such a method and shows how the external objects are generated and the code is emitted. It can be observed in line 2 that options such as the minimum and the maximum number of generated external objects are taken into consideration when randomly deciding the cardinality of the external declarations list. Line 5 creates a node of the type extern_declaration while in line 6 the randomization method is called on the node, which randomly generates a partial syntax tree starting from the extern_declaration. Lines 9 and 10 are responsible with producing the code using the extern_declaration node.

```
1  def generate_random_externs(self):
2    rnd = randomizer.randint(self.min_random_externs, self.
       max_random_externs)
3    extern_declarations = []
4    for x in range(0, rnd):
5      _extern_declaration = extern_declaration()
6      _extern_declaration.randomize()
7      extern_declarations.append(_extern_declaration)
8    code = ""
9    for _extern_declaration in extern_declarations:
10     code += _extern_declaration.generate_code()
11   return code
```

Listing 6.9: Generation of external objects

Enforcing Semantic Rules

One of the challenges linked to the test case generator is being able to produce semantically valid programs. As explained in Section 5.4.2, the solution chosen in P4Fuzz is to implement the semantic rules described in the specification of the language as filters on each of the affected AST nodes.

```
1  def filter(self):
2    if self.name.generate_code() in scope.get_available_types() or self.name
       .generate_code() in scope.get_available_variables():
3      return True
4    if isinstance(self.fromObj, header_type_declaration):
5      if self.type_ref.get_ref_type() in self.__class__.header_banned_types:
6        return True
7    return False
```

Listing 6.10: Filter method applied in the generation of header fields

Listing 6.10 shows how a filter implements semantic rules such as avoiding redeclaration of identifiers (lines 2-3) or avoiding nesting invalid types in a header declaration (lines 4-6).

Specializing the Fuzzer Using Probabilities

P4Fuzz can be used to target specific areas of the P4 language by changing the probabilities of selection for each production. The probabilities are specified as numbers from 0 to 100 representing the chances a production has for being chosen.

```
1 @staticmethod
2 def getRandom(probabilities):
3   rnd = random.random() * sum(probabilities)
4   for i, w in enumerate(probabilities):
5     rnd -= w
6     if rnd < 0:
7       return i
```

Listing 6.11: The getRandom method which selects a production based on a list of probabilities

Listing 6.11 shows the implementation of the method responsible for selecting productions based on their probabilities. It works by generating a random floating point number between 0 and 1 in line 3 and multiplying that with the sum of the probabilities. It then subtracts each of values from the generated number until it goes negative which signals that the current production rule index is to be used.

```
1 class base_type(object):
2   type = None
3   types = ["BOOL", "ERROR", "BIT", "BITOFINTEGER", "INTOFINTEGER", "
      VARBITOFINTEGER"]
4   probabilities = [20,20,20,20,10,10]
5   value = None
6
7   def randomize(self):
8       self.type = randomizer.getRandom(self.probabilities)
9 ................................................
```

Listing 6.12: Using the probabilities for randomly selecting a base_type production

Listing 6.12 demonstrates how the default values for the probabilities are defined and how they are used in order to select a base_type production.

Keeping Track of the Context

One important aspect of the test case generator module, in order to be capable of producing semantically valid programs, is the ability to keep track of the context. This includes remembering which data types (headers, structs, external objects) and which variables have been previously declared, as well as being able to tell which of them are available at any specific point in the generation process.

As described in Section 5.4.2, P4Fuzz implements a stack of scopes such that when a new block of code gets generated, a new scope is created and pushed into the stack. If data types or variables are declared during the randomization process for the current block, the variable or data type will be added to this scope. When a variable or data type is needed, it will be selected from the list of available types or variables based on the context.

```
@staticmethod
def get_available_types(only_type=None):
  available_types = {}
  for local_scope in scope.scope_stack:
    for local_type in local_scope["types"].items():
      if (only_type is not None and local_type[1]["type"] == only_type) or
     only_type is None:
        available_types.update({local_type[0]: local_type[1]})
  return available_types
```

Listing 6.13: Computing the list of available data types based on the current scope

Listing 6.13 shows how the test case generator computes the list of previously declared, available data types. It is also possible to filter the list based on a specific type.

```
def randomize(self):
  scope.start_local()
  # randomization code start
  ........................................
  # randomization code end
  scope.stop_local()
  scope.insert_type(self.name.generate_code(), "struct")
```

Listing 6.14: Starting and stopping a scope as well as inserting a data type in the struct declaration class

Listing 6.14 shows how the scope is used in the declaration of structs. Line 2 starts a new scope before the randomization process begins, while line 6 stops this scope afterwards. The declared struct is then added to the context with name and type using the insert_type method.

Optimizing the Randomization Process

P4Fuzz is designed to avoid being locked on a specific randomization library. This is achieved by the generator implementing a randomization interface that is always called when randomization is required. In the current version of P4Fuzz, the interface implements the standard random library, but this can easily be changed by implementing another interface using another library. Listing 6.15 shows an example of the randint method as it is implemented in the interface using the standard random library from Python.

```
@staticmethod
def randint(min_list_size, max_list_size):
  return random.randint(min_list_size, max_list_size)
```

Listing 6.15: The randint method of the randomization interface implementation using the standard random library

Another design decision explained in Section 5.4.2 describes the usage of seeds for reproducing test cases. The fact that the generator has an interface that deals

with all the randomization process also helps in the case of the seed, as there is only one class where this needs to be implemented in. Listing 6.16 shows how the seed is randomly generated and assigned to the random library.

```
@staticmethod
def generateRandomSeed():
  return random.randint(0, 10000000000)

@staticmethod
def setSeed(seed):
  randomizer.seed = seed
  random.seed(seed)

seed = randomizer.generateRandomSeed()
randomizer.setSeed(seed)
```

Listing 6.16: Generating a random seed and assigning it to the random library

Dealing with Nesting and Recursivity

As described in Sections 4.6 and 5.4.2, nesting and recursivity are important aspects of the P4 language which constitute a challenge for the test case generator. This is because Python enforces a limit on the maximum recursion depth a program can handle. We have experienced this issue many times during the initial iterations of the fuzzer with the interpreter complaining about the generator exceeding this limitation.

The solution for this challenge is to keep track of the current recursion depth per node and prevent the randomizer selecting a production which would further increase the depth level. Listing 6.17 shows how this solution is implemented for the type_ref class. The static variable *curDepth* keeps track of the current depth and productions are only selected if this counter is less than the *maxDepth*.

```
class type_ref(object):
    maxDepth = 5
    curDepth = 0

    def randomize(self):
        self.__class__.curDepth += 1
        while True:
            # randomly select production start
            ..........................
            # randomly select production end
            if self.__class__.curDepth < self.__class__.maxDepth:
                self.value.randomize()
                if not self.filter():
                    break
```

Listing 6.17: Limiting the recursion depth using a current depth level counter

Automation of the Generation Process

The test case generator as a stand alone module is capable of only generating one case each time it is ran. For the purpose of fuzzing, this process needs to be automated.

P4Fuzz includes a Python script which makes this automation process possible. The test generator script reads the files from an input folder and identifies the last generated test case. It then continues generating cases until a specified maximum test case number is reached. Because it always continues from the last test case number for each generation, multiple processes can run in parallel without affecting each other. Listing 6.18 shows how the automation of the test case generation is implemented in P4Fuzz.

```
while curNo < maxNo:
    curNo += 1
    filename = str(curNo).zfill(10)
    filepath = os.path.abspath(os.path.join(curDir, "../../input/" +
    filename + ".p4"))
    subprocess.call([sys.executable, os.path.abspath(curDir + "/../p4-
    testcase-generator/main.py"), "-f " + filepath, "-s false"])
    print "Test " + filename + " generated"
```

Listing 6.18: Implementation of the code generation process automation

6.5 Test Case Tester

The test case tester's task is to verify if a P4 program can be compiled using a specific target. By design the test cases are supposed to be syntactically valid. If the tester receives any errors from the compiler it marks the test case as a potential bug. Information about the test case is saved on the disk and in the database. Listing 6.19 shows how this processes is implemented in P4Fuzz.

```
try:
    # try to compile the test case using the BMv2 simple_switch back-end
  subprocess.check_output(["/usr/local/bin/p4c-bm2-ss --Wdisable " + input
     + " -o " + output + " > /dev/null"],
                  stderr=subprocess.STDOUT, shell=True)
    # if no exceptions were thrown it means the test was a success
  print "Test " + currentTest + " passed"
except subprocess.CalledProcessError as e:
    # an error has occured, create the path where the test case should be
    saved
  errorFile = os.path.abspath(os.path.join(errorsPath, currentTest + ".p4"
    ))
  # do the same for the error message file
  errorLogFile = os.path.abspath(os.path.join(errorsPath, currentTest + ".
    err"))
  # read the file's first line as it contains the seed
  seed_line = open(input).readline().rstrip()
  # search for the seed
  seed_match = re.search('//seed: ([0-9]+)', seed_line)
```

```
16    # extract the seed
17    seed = seed_match.group(1) if seed_match is not None else 0
18    # save the test case as an error file
19    shutil.copyfile(input, errorFile)
20    # save the error message in the error file
21    f = open(errorLogFile, "w")
22    f.write(e.output)
23    f.close()
24
25      # data for inserting the possible bug into the database
26    data_bug = (currentTest, e.output, errorFile, seed, 0)
27    # insert it
28    cursor.execute(add_bug, data_bug)
29    # commit the changes
30    cnx.commit()
31
32    print "Test " + currentTest + " failed"
```

Listing 6.19: Implementation of the test case tester's main action

The tester as a stand-alone module is capable of verifying only one test case per run, but P4Fuzz offers a script for automation of the process.

The script starts by scanning the input folder in order to find available test cases. It then selects the lowest test case number and renames it to a temporary name to prevent other testers from also selecting it. This enables the possibility for multiple testers to run in parallel without interfering with each other.

The test case tester module is called with the filepath as input parameter and the process continues with the next test case until all the cases are depleted. Listing 6.20 shows the automation of the test case testing process as it is implemented in P4Fuzz.

```
1  # loop while there is still a current case to process
2  while currentTest:
3      # build the path to the test case file
4      currentFile = os.path.abspath(os.path.join(inputPath, currentTest + ".
       p4"))
5      # build the random path for the file to be renamed to
6      tmpFile = os.path.abspath(os.path.join(inputPath, currentTest + ".tmp"
       ))
7      # rename it
8      os.rename(currentFile, tmpFile)
9      # build the path for the compiled target file
10     outFile = os.path.abspath(os.path.join(outputPath, currentTest + ".
       json"))
11     # call the test case tester module using the temporary filepath as
       input
12     output = subprocess.call([sys.executable, os.path.abspath(curDir + "
       /../p4-testcase-tester/main.py"), "-n " + currentTest, "-i " + tmpFile
       , "-o" + outFile, "-e " + errorsPath])
13     #remove the temporary file
14     os.remove(tmpFile)
15     # extract a list of sorted still available test case files
```

```
16      files = [os.path.splitext(f)[0] for f in os.listdir(inputPath) if re.
        match(r'[0-9]{10}\.p4', f)]
17      # select the lowest available test case numer
18      currentTest = None if len(files) < 1 else sorted(files)[0]
```

Listing 6.20: Implementation of the test case tester automation process

6.6 Packet Generator

The Packet Generator module is implemented as a python script that uses the p4pktgen functionality, captures the produced table entries and packets, and saves them to a .pkt file.

```
1 p4pktgen json_file -au -rss -d
```

Listing 6.21: p4pktgen call

Listing 6.21 presents the bash command used to call the p4pktgen and generate the table entries and packets. The json_file parameter is the path to the compiled P4 program. The '-au' parameter allows uninitialized reads (reads of uninitialized fields return 0). The '-rss' (run simple switch) parameter is needed in order to display the Thrift command used by the p4pktgen to send to table entries, alongside with the specific values. The '-d' (debug) parameter is optional.

```
1 import commands
2 import sys
3
4 file = sys.argv[1]
5 bash_command = 'p4pktgen ' + file + ' -au -rss -d'
6 error, output = commands.getstatusoutput(bash_command)
```

Listing 6.22: Packet Generator use of p4pktgen

In Listing 6.22 the module reads the json program filename sent as first argument to the script (line 4), creates the bash command for that specific file (line 5), and finally saves the error code number and output to variables (line 6).

```
1  f = open('packets/' + filename + '.pkt','w')
2
3  for line in output.splitlines():
4          if 'INFO: table_add' in line:
5                  l = line.replace('INFO: table_add ','')
6                  # print 'writing: '+ l
7                  f.write(l)
8                  f.write('\n')
9          if 'INFO: packet' in line:
10                 l = line.split(") ",1)[1]
11                 f.write(l)
12                 f.write('\n')
13 f.close()
```

Listing 6.23: Packet Generator save table entries to file

Second part of the script (Listing 6.23) creates a .pkt file for the given json program, checks each line of the output for specific substrings known to preceed the table entries and packets (i.e: 'INFO: table_add', 'INFO: packet"), and saves the specific line to the file. The Packet Generator module can be called using the command in Listing 6.24.

```
python packet-generator.py program.json
```

Listing 6.24: Call Packet Generator module

An example of the created file that contains table entries and packets is shown in Listing 6.25. The lines contain information such as: table name, action, value, parameters, priority and packet.

```
IngressImpl.table0 IngressImpl.set_egress_port 0&&&511 0&&&281474976710655
    0&&&281474976710655 0&&&65535 => 0 1
00000000000000000000000000000000
IngressImpl.table0 IngressImpl.set_egress_port 0&&&511 0&&&281474976710655
    0&&&281474976710655 0&&&65535 => 0 1
00000000000000000000000000000000
IngressImpl.table0 IngressImpl.set_egress_port 0&&&511 0&&&281474976710655
    0&&&281474976710655 0&&&65535 => 511 1
00000000000000000000000000000000
IngressImpl.table0 IngressImpl.set_egress_port 0&&&511 0&&&281474976710655
    0&&&281474976710655 0&&&65535 => 511 1
00000000000000000000000000000000
```

Listing 6.25: Example of file produced by Packet Generator

Each P4 program has its table entries and packets saved into a .pkt file with the same name. Once a program's entries and packets have been generated, P4Fuzz can parse the .pkt file and run the next module in the fuzzing flow - the Packet Tester.

6.7 Packet Tester

The Packet Tester module is designed as a wrapper surrounding the classes for each architecture. The module decides based on the "target" parameter which architecture class to instantiate. Other parameters provided to the Packet Tester module are: jsonfile, table, action, values, parameters, priority, packet. Values for each of these parameters are provided by the other modules of P4Fuzz.

The json_file parameter is the path to the compiled P4 program. It is used by the architecture class constructor for the initial set up. In BMv2 with Simple Switch for example, this implies creating named pipes for each input pcap file, starting the simple switch, open all input pcap named pipes for writing, write a pcap file header to it, and finally open the Thrift TCP connection to simple switch's port where it is listening for control plane commands [23].

Listing 6.26 shows how the SimpleSwitch class is instantiated; the tmpdir parameter is the directory where the simple switch starts.

```
switch = SimpleSwitch(json_file, tmpdir)
```

Listing 6.26: Instantiate Simple Switch

Once the switch is started and the Thrift connection is established, we can start adding entries to the match tables. This is done by using the table_add() function which parses the parameters into a "line" used by Thrift as shown in Listing 6.27.

```
def table_add(self, table, action, values, params, priority):
    self.modified_tables.append(table)
    priority_str = ""
    if priority:
        priority_str = " %d" % (priority)
    self.api.do_table_add('{} {} {} => {}{}'.format(table, action, ' '.
    join(values), ' '.join([str(x) for x in params]), priority_str))
```

Listing 6.27: Add entry to table

The self.api object is an instance of Runtime_CLI class, meant to hide the low level implementation of Thrift actions. Both the Runtime_CLI class and table_add() function are provided by the p4pktgen [3].

After the table entries are added, the packets can be sent to the switch as shown in Listing 6.28.

```
switch.send_packet(packet)
```

Listing 6.28: Send packet

The send_packet() function shown in Listing 6.29 is a modification of the original send_ and_ check_ only_ 1_ packet() provided by p4pktgen, and it only prints the output of the simple switch which can be later used by the Packet-out Comparison module. Currently, the packets are always send into port 0 of simple_switch as this is the case for p4pktgen as well.

```
def send_packet(self, packet):
    # TBD: Right now we always send packets into port 0 of
    # simple_switch.  Later should generalize to enable sending
    # packets into any of several ports.
    intf_num = 0
    logging.info('Sending packet to port {}'.format(intf_num))
    self.intf_info[intf_num]['pcap_in_fp']._write_packet(packet)
    for i in sorted(self.intf_info):
        self.intf_info[i]['pcap_in_fp'].flush()
        self.intf_info[i]['pcap_in_fp'].close()
    logging.debug('Finished _write_packet() to %s',
                  self.intf_info[intf_num]['pcap_in_fname'])

    # Print the simple_switch output
    for b_line in iter(self.proc.stdout.readline, b''):
        line = str(b_line)
        print("Line from simple_switch log: " + line.strip())
        if 'Pipeline \'egress\': end' in line or 'Dropping packet at the
    end of ingress' in line:
```

```
19            break
```

Listing 6.29: Send packet function

Finally the tables are cleared and the switch is shutdown. This is needed as testing multiple packets and programs can throw errors as "Cannot start switch; an instance is already running" or "Broken pipe".

```
1 switch.clear_tables()
2 switch.shutdown()
```

Listing 6.30: Clear tables and shutdown switch

6.8 Compiler Fuzzer Taming

The tamer component in P4Fuzz has the role of grouping together errors of the same type based on similarities found in the error messages descriptions, in order to improve the process of manually investigation of the potential bugs. The challenges and the proposed solution regarding taming are well described in Sections 4.4 and 5.4.7.

P4Fuzz is using a clustering algorithm for grouping the similar potential bugs discovered. The algorithm is called k-medoids and it is included in the third party library PyClustering which is described in Section 6.1.2. The implementation of the algorithm requires either a distance function to be used for computing distances between error messages while the clustering is made or a pre-computed distance matrix which records the distances between each test case error message.

Our tool computes a distance matrix using the distance function described in Listing 5.9 from Section 5.4.7 using P4FuzzCLib, a P4Fuzz library available for Python but written in C++ for better performance.

Listing 6.31 shows how the distance matrix is computed. The function receives as parameter a vector of strings representing the error messages. Lines 2-5 deal with allocating memory for the distance matrix on the heap since the stack is too small for allowing large matrices. Lines 7-15 deal with looping through the messages while computing and storing the distances in the matrix. It can be seen that the inner loop does not start from 0, but from the index of the outer loop since distances are symmetric (dist(a, b) = dist(b, a)). Lines 16-22 are used for converting the result from arrays to standard library vectors. Lines 23-26 deal with memory management, deallocating the heap space used for the matrix to avoid memory leaks.

```
1 std::vector< std::vector<int> > calc_distance_matrix(std::vector<std::
    string> data){
2     int **dist = new int*[data.size()];
3     for (size_t i = 0; i < data.size(); i++){
4         dist[i] = new int[data.size()];
5     }
6     std::vector< std::vector<int> > result;
```

```
7       for(std::vector<std::string>::iterator i = data.begin(); i != data.end
        (); i++){
8           int i_index = i - data.begin();
9           for(std::vector<std::string>::iterator j = data.begin() + i_index;
         j != data.end(); j++){
10              int j_index = j - data.begin();
11              int distance = _common_tokens_distance(&*i, &*j);
12              dist[i_index][j_index] = distance;
13              dist[j_index][i_index] = distance;
14          }
15      }
16      for(unsigned int i=0; i<data.size(); i++){
17          std::vector<int> partial;
18          for(unsigned int j=0; j<data.size(); j++){
19              partial.push_back(dist[i][j]);
20          }
21          result.push_back(partial);
22      }
23      for (size_t i = data.size(); i > 0; ){
24          delete[] dist[--i];
25      }
26      delete[] dist;
27      return result;
28  }
```

Listing 6.31: Computing the distance matrix using the token-based distance function

The clustering algorithm groups the items into k clusters, with k being a parameter that the algorithm needs to know beforehand. Because in the case of P4Fuzz tamer, k is unknown, it is required for the tool to implement an incremental algorithm which starts with $k = 2$ clusters. After each clustering process is finished, it tests if the highest difference between two error messages in the same cluster is greater than a threshold currently set at 60%. If this is the case then a new cluster is created and the k-medoids algorithm runs again using a new k incremented by 1. This happens until all the clusters include error messages that differ by less than 60%. Listing 6.32 shows how this is achieved in the implementation of P4Fuzz tamer.

```
1  # flag for knowing if a large distance still exists in any of the clusters
2  has_large = True
3  # counter for tracking the number of clusterings
4  cnt = 1
5  # while there is a larger distance than the threshold in any cluster
6  while has_large:
7      # increment the counter
8    cnt += 1
9    # re-initialize the flag to false
10   has_large = False
11   # loop through the existing clusters
12   for i, cluster in enumerate(clusters):
13       # identify the medoid of this cluster
14     medoid = medoids[i]
15     # extracting the distances between the medoid and each item
```

```
16      medoid_distances = dist[medoid]
17      # calculates the maximum distance between all the points in the
        cluster
18      max_points = p4fuzzclib.calc_max_distance_cluster(dist_tuple, cluster)
19      # extract the maximum distance
20      max_dist = dist[max_points[0]][max_points[1]]
21      # compare it to the threshold
22      if max_dist > 60:
23          # set the flag to true
24        has_large = True
25        # compute a new medoid
26        new_medoid = max_points[0] if medoid_distances[max_points[0]] >
        medoid_distances[max_points[1]] else max_points[1]
27        #re-initialize the k-medoids algorithm
28        initial_medoids = medoids
29        initial_medoids.append(new_medoid)
30        kmedoids_instance = kmedoids(dist, initial_medoids, 100, data_type='
        distance_matrix')
31        # process the new clustering
32        kmedoids_instance.process()
33        # extract the clusters
34        clusters = kmedoids_instance.get_clusters()
35        # extract the medoids
36        medoids = kmedoids_instance.get_medoids()
37    print "Clustered #" + str(cnt) + " ..."
```

Listing 6.32: Implementation of the clustering algorithm with unknown number of clusters expected

Chapter 7

Evaluation

This chapter presents the evaluation of the developed P4 compiler fuzzer. The first part of the chapter introduces the environment used in the development process, including the servers, programming languages, tools, and hardware specifications. The second part focuses on the test cases produced by the fuzzer, the actual bugs that have been reported, as well as the overall performance of the tool.

7.1 Experimental Setup

The evaluation of P4Fuzz is based on the experiments conducted using the latest version of the tool on a data set of 10.000 test cases. The environment used for running P4Fuzz was a virtual machine installed on one of the physical rack servers at the university. Because of its large number of dependencies shown in Table 6.2, Vagrant was also used to ease the process of virtual machine re-installation when needed. Table 7.1 shows the hardware and software technical specifications of the environment.

Table 7.1: Experiment setup technical specifications

Physical server CPUs	8 x Intel Xeon E5420 @ 2.50GHz
Physical server memory	32 GB
Physical server OS	Ubuntu 16.04.1 LTS
Vagrant version	Vagrant 1.8.1
VirtualBox version	5.0.40r115130
VM CPUs	4
VM memory	8 GB
VM OS	Ubuntu 16.04.4 LTS
P4C Git commit	48a57a6ae4f96961b74bd13f6bdeac5add7bb815

7.2 Bugs Found

This section provides details about the discovered bugs within P4C compiler. During our evaluation of P4Fuzz we managed to identify four distinct bugs, out of which two of them (Bugs #1291 and #1296) got fixed on the official repository, one of them (Bug #562) is known but not fixed, and one of them (Bug #1325) was recently submitted thus it is not fixed yet.

The following sections present descriptions of the bugs, alongside ways of reproducing them, and their occurrence. Figure 7.1 highlights the occurrences of the mentioned bugs within the test cases generated by P4Fuzz. As both bugs #562 and #1325 occur when nested structs are generated, many of the test cases contain both of them as shown in column "#562 + #1325". The "Others" column refers to the errors thrown by the compiler due to invalid test cases, bugs in the fuzzer generator or possible P4 compiler bugs that have not been reviewed.

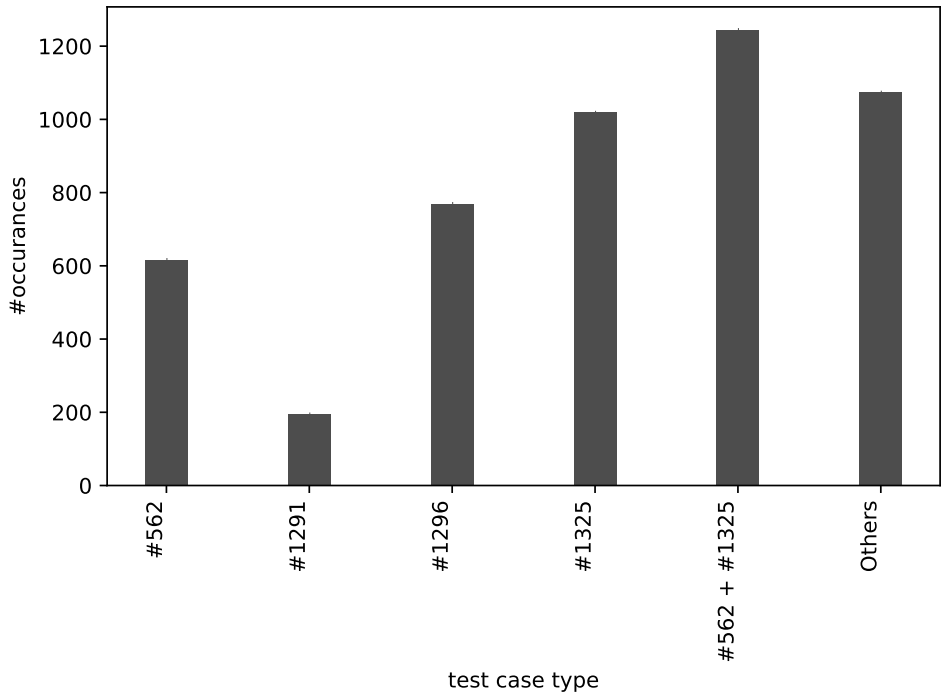

Figure 7.1: Bugs occurrences

7.2.1 Bug #1296: Specializing Extern Objects on the Same Extern Object Type

Issue #1296 shows that declaring specialized extern objects by specializing them on the same extern object type triggers a compiler bug [24]. Listing 7.1 presents an example in this sense, while Listing A.1.1 presents a P4 program that includes the

bug.

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

Listing 7.1: Specializing Extern Objects on the Same Extern Object Type

"We have tried compiling for the BMv2 target which leads to compilation error. As we thought it is a BMv2 limitation, we have also tested with a simple extern object defined inside the program and compiling using the p4test and the result is the same. This happens when the extern object type declaration contains a method which uses the specialization argument in any of the method parameters." [24]

Listing 7.2 presents the output of the compiler when a P4 program triggering the explained bug is used. The bug occurred 770 times within 4920 possible bugs saved in the database. Issue #1296 was tagged as bug and solved on the official p4c repository.

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

Listing 7.2: Issue #1296 compiler output

7.2.2 Bug #1291: Varbit Declaration in Structs

Issue #1291 shows that declaring varbit types inside structs and compiling for the BMv2 target leads to compilation error. In the language specification [10] it is written under Section 7.2.7 (Type nesting rules) that varbits are allowed in a struct container type. Listing 7.3 presents an example in this sense, while Listing A.1.2 presents a P4 program that includes the bug [25].

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

Listing 7.3: Varbit Declaration in Struct

Listing 7.4 presents the output of the compiler upon the run of a P4 program containing the explained bug. The bug occurred 195 times within 4920 possible bugs saved in the database. Issue #1291 was tagged as bug and solved on the official p4c repository.

```
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;
             ^^^^
```

Listing 7.4: Issue #1291 Compiler Output

7.2.3 Bug #1325: Error Type in Nested Struct

Issue #1325 shows that declaring error types inside nested structs and compiling for the BMv2 target leads to compilation error. In the language specification [10] it is written under Section 7.2.7 (Type nesting rules) that both structs and errors are allowed in struct container types.

Since p4test is able to compile it, we assume this might be a BMv2 back-end issue. Listing 7.5 presents an example in this sense, while Listing A.1.3 presents a P4 program that includes the bug [26].

```
1 struct test_struct {
2   error test_error;
3 }
4
5 struct local_metadata_t {
6   test_struct test;
7 };
```

Listing 7.5: Error Type in Nested Struct

Listing 7.6 presents the output of the compiler when a P4 program triggering the explained bug is used. The bug occurred 1019 times within 4920 possible bugs saved in the database. Issue #1325 was tagged as bug on the official p4c repository, but has not been solved as of June 2018.

```
1  In file: p4c/backends/bmv2/header.cpp:204
2  Compiler Bug: /usr/local/share/p4c/p4include/core.p4(23): error { NoError,
       PacketTooShort, NoMatch, StackOutOfBounds, HeaderTooShort,
     ParserTimeout }: unexpected type for struct test_struct {
3    error { NoError, PacketTooShort, NoMatch, StackOutOfBounds,
       HeaderTooShort, ParserTimeout } test_error; }.test_error
4  error {
5  ^
6  issue.p4(6)
7  struct test_struct {
8         ^^^^^^^^^^^
9  issue.p4(7)
10   error test_error;
11         ^^^^^^^^^^
```

Listing 7.6: Issue #1325 Compiler Output

7.2.4 Bug #562: Nested Structs

Issue #562 shows that declaring nested structs and compiling for the BMv2 target leads to compilation error. In the language specification [10] it is written under Section 7.2.7 (Type nesting rules) that struct types are allowed in struct container types.

Listing 7.7 presents an example in this sense, while Listing A.1.4 presents a P4 program that includes the bug [27].

```
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;
};
```

Listing 7.7: Error Type in Nested Struct

Listing 7.8 presents the output of the compiler when a P4 program triggering the explained bug is used. The bug occurred 617 times within 4920 possible bugs saved in the database. Issue #562 was tagged as bug on the official P4C repository, but has not been solved as of June 2018.

```
In file: /home/vagrant/p4c/backends/bmv2/header.cpp:180
Compiler Bug: issue3.p4(8): struct row_t {
  alt_t alt0;
  alt_t alt1; }: nested structure
struct row_t {
       ^^^^^
```

Listing 7.8: Issue #562 Compiler Output

7.3 Test Cases Size Growth

Each test case program is generated by the fuzzer using representations of code structures called tokens. A test case token is a node in the randomly generated abstract syntax tree. Given the code in Listing 7.9, in which an empty struct is declared, five tokens were used as shown in Figure 7.2.

```
struct my_struct {
}
```

Listing 7.9: A simple struct declaration in P4

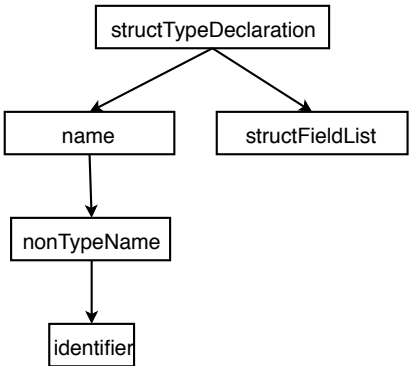

Figure 7.2: AST showing tokens used for declaring an empty struct

Figure 7.3 shows how the size of the generated programs grows linearly with the number of tokens used by the generator.

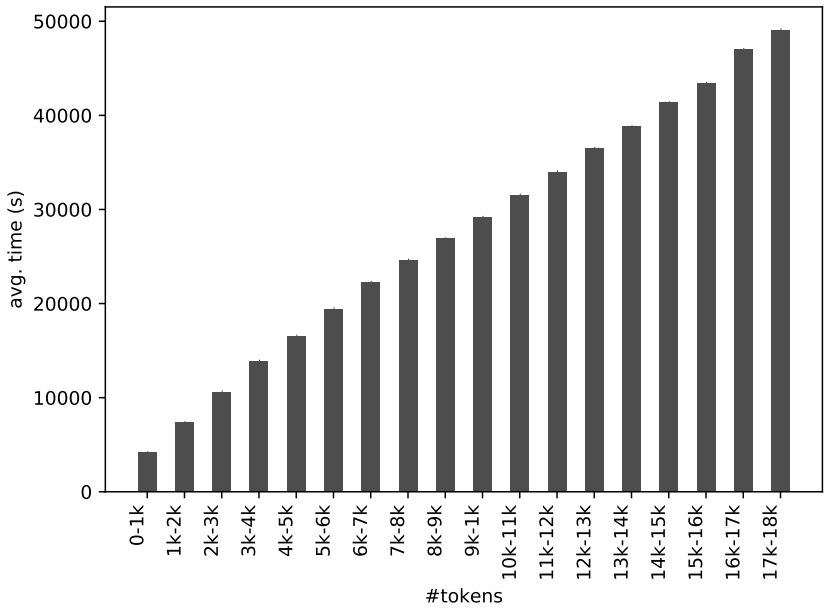

Figure 7.3: Test cases size growth in relation to number of tokens used

7.4 Bugs Distribution Accross Compiler Stages

Once a potential error is manually analyzed and is marked as a real problem, the developers responsible with fixing it are notified. Because P4C is a modular compiler consisting of one front-end and multiple back-ends, it is important to know which of the components fails.

P4C comes with a method of running only the front-end component for testing the

validity of P4 programs without compiling them to target code. Using this method we were able to observe, for each bug discovered, which part of the compiler is failing and in this way better document the issue notifications.

Table 7.2 shows which component was failing in the case of the four bugs that were found by the P4Fuzz. It can be seen that most of them (75%) are issues in the back-end. Even though the number of discovered bugs is not large enough to be able to draw a strong conclusion, this findings confirm our hypothesis that back-ends are the most vulnerable.

Table 7.2: Bugs Distribution Accross Compiler Stages

Bug	Component
Bug #562	BMv2 back-end
Bug #1291	BMv2 back-end
Bug #1296	P4C front-end
Bug #1325	BMv2 back-end

7.5 Performance

This section focuses on the performance aspect of the fuzzer, more specifically the test case generator, the test case tester and the taming module. Each of these components are analyzed from the point of view of time required to perform their operations.

7.5.1 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 Figure 7.4, 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.

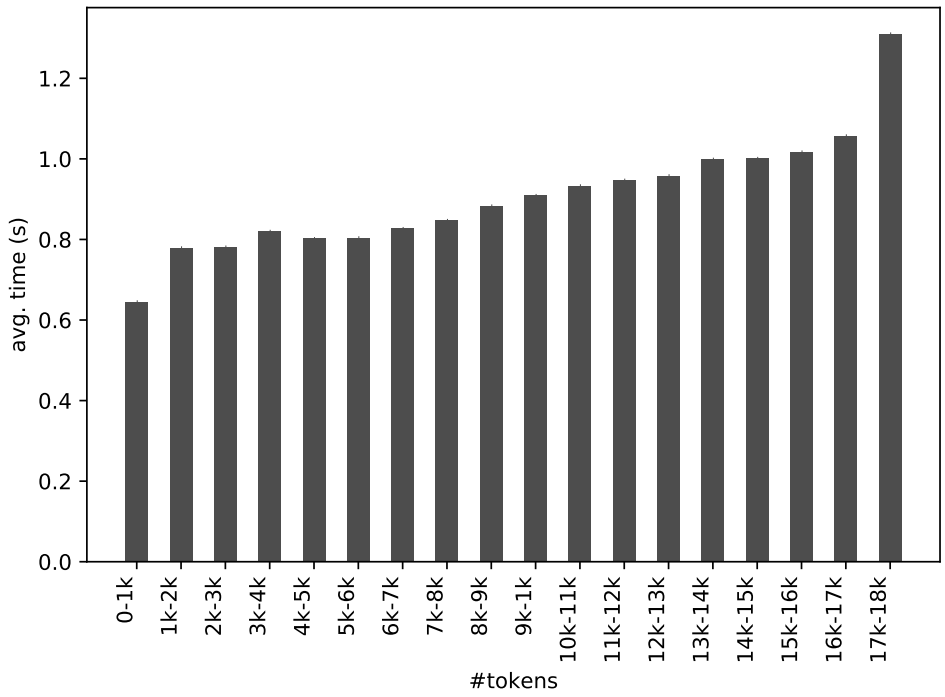

Figure 7.4: Average generation time for test cases based on number of tokens used

7.5.2 Testing the Programs

Similar to the program generator, the test case tester module of the P4Fuzz is also in close relationship with the overall performance of the fuzzer. Being able to verify a large number of programs in a short amount of time makes the compiler fuzzer more efficient at finding interesting bugs.

Figure 7.5 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 play as well 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 14.000-15.000 tokens is not advisable as the time required for the process increases drastically.

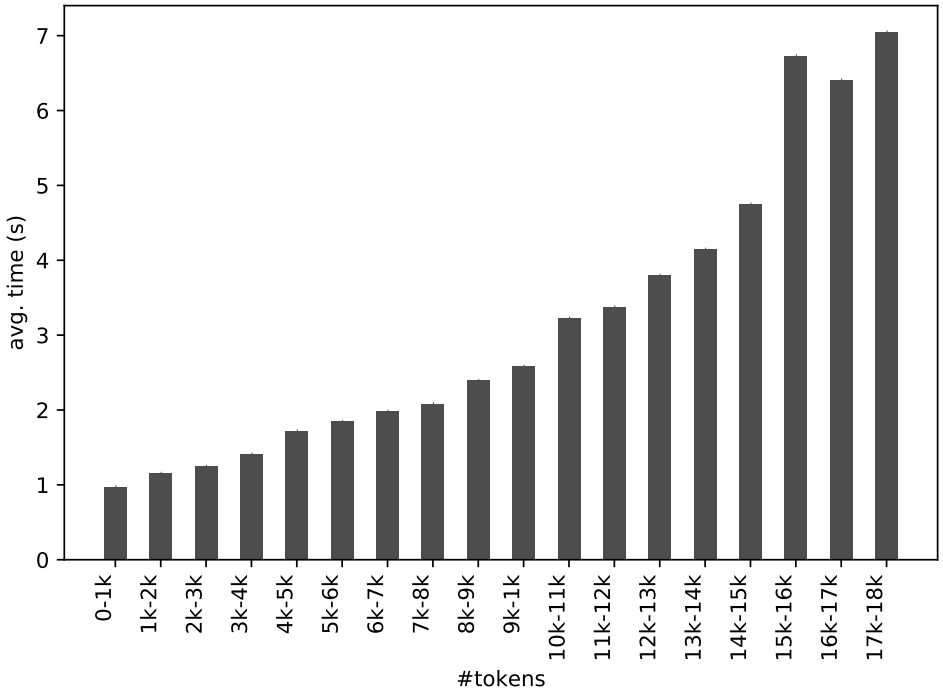

Figure 7.5: Average compilation time for a test cases based on number of tokens used

7.5.3 Crashing Test Cases Rate

Using the latest version of P4Fuzz an experiment was conducted for determining the distribution of the number of generated cases based on the number of tokens. The results also show the rate at which the programs crash the compiler.

In Figure 7.6, the red bars represent the distribution of the number of generated test cases based on the number of tokens used. As expected from the fact that the programs are randomly generated, the chart seems to follow a Gaussian distribution. The blue bars represent test cases that are crashing the compiler. 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.

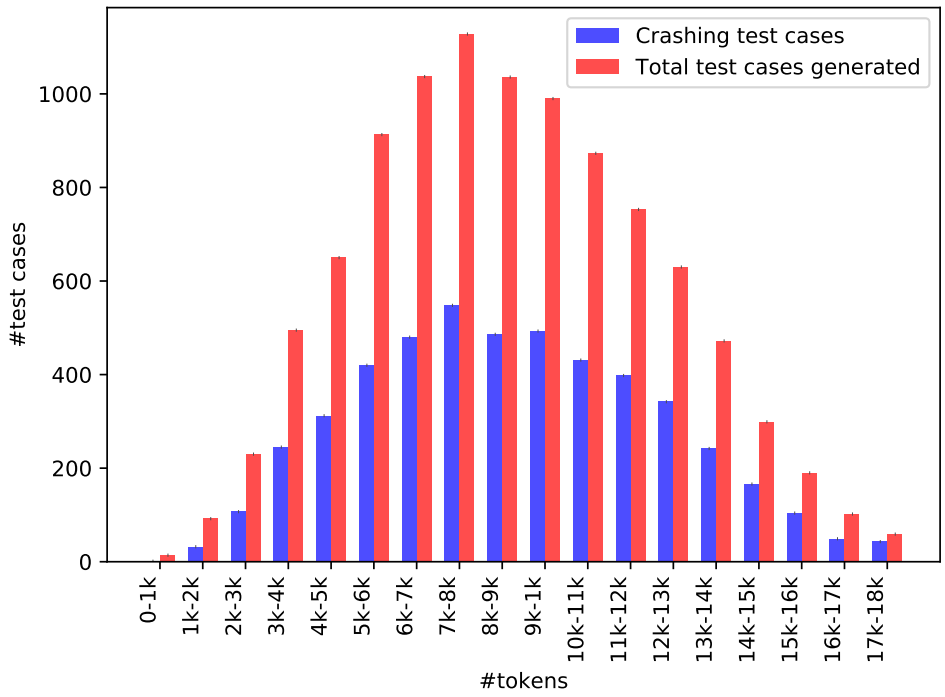

Figure 7.6: Distribution of test cases in relation to the number of tokens used

7.5.4 Taming

Figure 7.7 shows the difference between the runtime complexity of the two distance functions used in the taming process: Levenshtein function and Token-based function. The improvement from an exponential time complexity to a linear one allowed us to tame the possible bugs found in a feasible way.

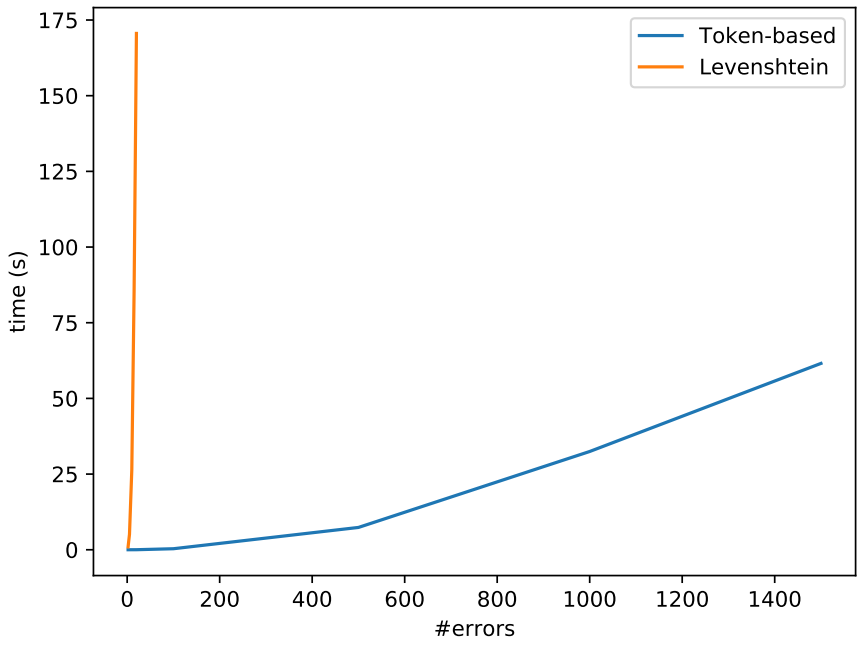

Figure 7.7: Token-based and Levenshtein distance functions growth rates

Figure 7.8 shows the time required for the taming algorithm (distance function and clustering) to run on a specific number of potential bugs. Since an important part of the taming algorithm is influenced by the distance function calculation, it can be seen that the Token-based function made it possible to process more than 1000 cases in less than 5 minutes. According to our estimations, a taming algorithm based on Levenshtein distance function would require several hours to complete for the same amount of cases.

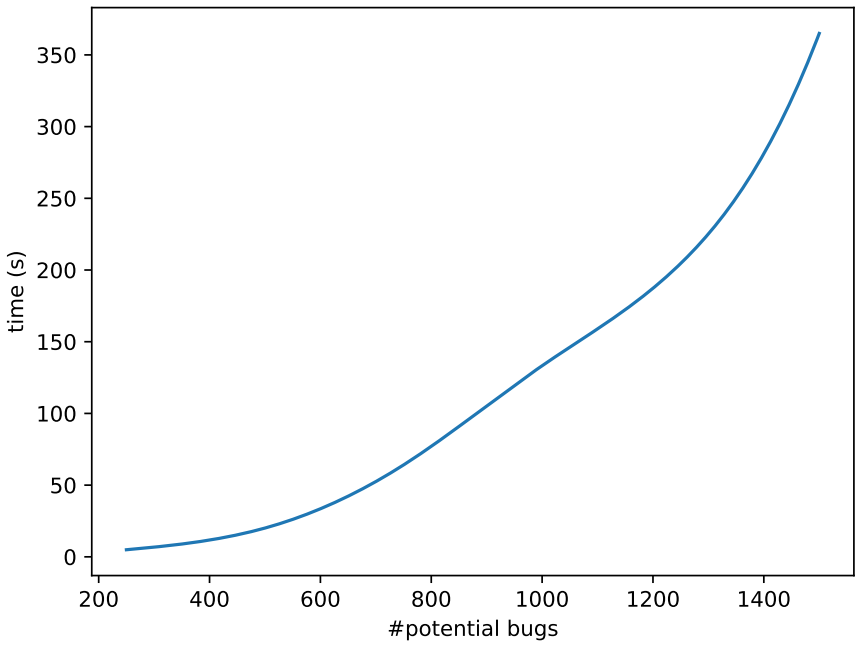

Figure 7.8: Taming Time Based on Bugs Analyzed

While taming is an expensive process, it can be seen from Figure 7.9 that it gives better results when compared to analyzing the potential bugs in a random order. Because clustering groups of errors based on their similarity in terms of word occurrences, it can be seen that for the first few number of test cases analyzed it gives 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.

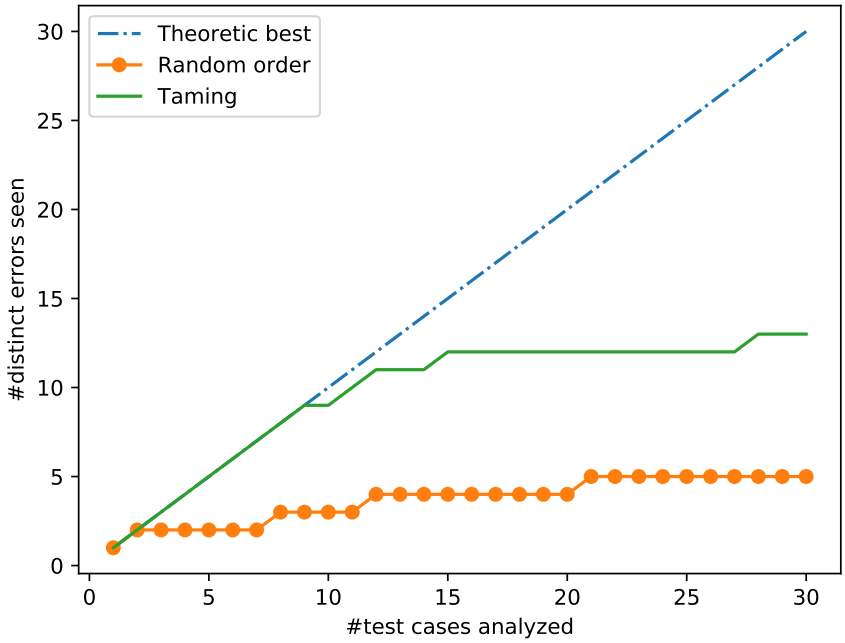

Figure 7.9: Tamed vs. random ordering distinct error found per number of cases analyzed

Chapter 8

Case Study - How Can Compiler Bugs Introduce Security Threats

Security threats are not always caused by the source code of the vulnerable program but can also be introduced by compiler bugs. This chapter presents a case study on how the bit-wise negation bug discovered in the P4C compiler for simple_switch can represent a vulnerability for the dataplane.

The purpose of this case study is to show an example of how a bug in the compiler can affect security aspects of dataplanes. Even though the P4 program used for demonstrating this case is purely fictive, the possibility of something similar happening is a serious matter.

The context described in this case study is the following: a simple partial MAC address filter is implemented in P4. While the implementation is correct, the filter is not behaving properly. The following sections show the implementation of the filter, explain the bug involved and discuss the implications of issue.

8.1 The Simple Partial MAC Address Filter

MAC address filters are widely used but most of them only support exact matching. A partial match filter on the other hand could also be used for dropping packets received from devices developed by specific vendors.

Listing 8.1 shows partial implementation of the Simple Partial MAC Address Filter as a P4 program. The complete source code is available in Appendix A.2. The program works by using a metadata field in which it extracts the OUI (Organizationally Unique Identifier) part of the source MAC address. The extraction operation using bit manipulation is shown in lines 39-40. A simpler solution would use the bit-slice operator available in P4. After the extraction of the OUI value, the match-action table is applied which drops the packet if a match is found.

```
1 typedef bit<48> ethernet_address;
2 typedef bit<9> port_t;
```

```
3  const port_t ACCEPT_PORT = 1;
4  const port_t DROP_PORT = 0;
5
6  header ethernet_t {
7      ethernet_address dstAddr;
8      ethernet_address srcAddr;
9      bit<16> etherType;
10 }
11
12 struct metadata {
13     bit<24> vendorOui;
14 }
15
16 struct headers {
17     ethernet_t ethernet;
18 }
19
20 // Omitted code for the parser which parses the ethernet header
21
22 control ingress(inout headers hdr,
23                 inout metadata user_meta,
24                 inout standard_metadata_t standard_metadata){
25
26     action _drop() {
27         standard_metadata.egress_spec = DROP_PORT;
28     }
29
30     table vendor_ouis {
31         key = {
32             user_meta.vendorOui : exact;
33         }
34         actions = {
35             _drop();
36         }
37     }
38     apply {
39         bit<48> extractor = ~((bit<48>) 24w0xffffff);
40         user_meta.vendorOui = (bit<24>) ((hdr.ethernet.srcAddr & extractor
       ) >> 6);
41         standard_metadata.egress_spec = ACCEPT_PORT;
42         vendor_ouis.apply();
43     }
44 }
45 // Omitted code for blank implementation of other controls and
       initialization
```

Listing 8.1: Implementation of the Simple Partial MAC Address Filter in P4

We exemplify the extraction process with a MAC address $(00-B5-D0-AB-05-ED)$ in which the first 3 bytes $(00-B5-D0)$ represent an OUI owned by Samsung. The extractor variable from line 39 is initially declared as a 24-bit value $(0xffffff)$ which is then widened to 48 bits, and then negated to $0xffffff000000$. Listing 8.2 shows how starting with the MAC address, the OUI part of the address

can be extracted.

```
0000 0000 1011 0101 1101 0000 1010 1011 0000 0101 1110 1101 &
1111 1111 1111 1111 1111 1111 0000 0000 0000 0000 0000 0000 =
0000 0000 1011 0101 1101 0000 0000 0000 0000 0000 0000 0000

0000 0000 1011 0101 1101 0000 0000 0000 0000 0000 0000 0000 >> 6 =
0000 0000 0000 0000 0000 0000 0000 0000 1011 0101 1101 0000

0000 0000 0000 0000 0000 0000 0000 0000 1011 0101 1101 0000 (narrow to 24
    bit) =
0000 0000 1011 0101 1101 0000 (0x00B5D0)
```

Listing 8.2: Example of how the extraction process should work

8.2 The Bit-wise Negation and Widening Bug

Issue #983[28] describes a bug in the BMv2 back-end compiler for the simple_switch in which a value that gets widened and then negated produces a wrong computation of the result. An example of this problem can be seen in Listing 8.3. The bug was acknowledged and fixed by the developers after it was submitted.

```
bit<48> extractor = ~((bit<48>) 24w0xffffff);
// extractor now holds the value 48w0x000000000000 instead of 48
    w0xffffff000000
```

Listing 8.3: Example of the Bit-wise Negation and Widening Bug

8.3 How is the Bug Affecting the Filter?

Because of the Bit-wise Negation and Widening Bug, the extractor variable is equal to $0x000000000000$ instead of $0xffffff000000$. As the variable is used to extract the first 3 bytes into the metadata field for the OUI, this bug is changing the behaviour of the program without the developer being aware of it.

Listing 8.4 shows how the extraction process runs as a result of the bug. It can be seen that in the end, the extracted 3 bytes are $00 - 00 - 00$ instead of the OUI value of $00 - B5 - D0$.

```
0000 0000 1011 0101 1101 0000 1010 1011 0000 0101 1110 1101 &
0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 =
0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000

0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 >> 6 =
0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000

0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 (narrow to 24
    bit) =
0000 0000 0000 0000 0000 0000 (0x000000)
```

Listing 8.4: Example of how the extraction process works because of the bug

8.4 Implications

Because of the faulty implementation in the P4 compiler, the filter running the P4 program is no longer able to do its job of dropping packets based on the OUI part of the MAC address. Failure in filtering packets represents a serious problem which affects the entire network behind the filter.

A future version of P4Fuzz that implements the Packet-out Comparsion module for at least two different architectures (i.e: BMv2, eBPF) may have been able to discover this issue. Assuming that one compiler implementation contains the presented bug and the others do not, the fuzzer would report the program as a potential issue based on the differences in the output.

Considering the example of the simple partial MAC address filter program and how the behaviour of the filter is altered, it can be concluded that at least some compiler bugs have the potential of introducing security issues.

As programmable dataplanes and the P4 language become more and more popular, it is important to also examine the P4 compilers from a security point of view in order to better secure the P4 programmable dataplanes and prevent introduction of threats.

Chapter 9

Limitations

9.1 Packet-out Comparsion on P4 and eBPF

Because P4 and eBPF have in common a small part of their scopes (Figure 9.1), the number of test cases generated that are able to run on both architectures is small; this makes it hard for the packet-out comparison module to produce conclusive results by comparing their behaviours and outputs. Since the common scope of the P4 and eBPF is packet filtering and simple forwarding, only randomly generated programs that are designed in this sense represent valid test cases for comparison.

We consider this more of an eBPF limitation, but that however propagates to our fuzzer as well. We expect to have better results when comparing different architectures once we:

- run the fuzzer for a longer period of time and generate more P4 programs that have as main purpose packet filtering and simple forwarding
- add more architectures - the P4 programs are then not limited by restrictions as it is the case of eBPF
- add more semantic support - the P4 programs become more complex within the packet filtering and simple forwarding categories

"(...) P4 and EBPF programming languages have different expressive powers. However, there is a significant overlap in their capabilities, in particular, in the domain of network packet processing. We expect that the overlapping region will grow in size as both P4 and EBPF continue to mature. The following image illustrates the situation." [29]

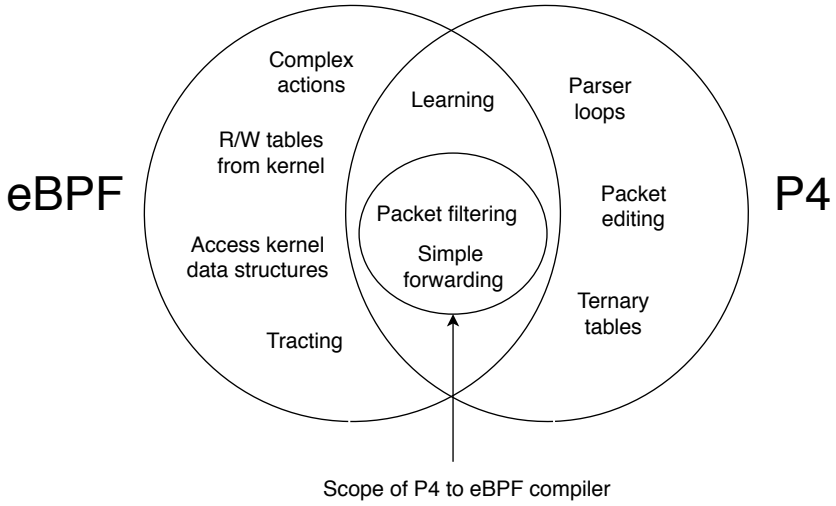

Figure 9.1: Scope of P4 and eBPF languages [29]

9.2 Compiler Code Coverage

One limiting factor is represented by the lack of tools for analyzing the code coverage of the P4C source code. Observing how different fuzzing strategies would affect this coverage could help better design the tool.

Being able to improve the code coverage of the tested compilers could potentially prevent lots of security threats introduced by bugs as well as help increase the robustness of the compilers. Unfortunately it is not guaranteed that a compiler fuzzer can drastically improve the already existing coverage. In the case of Csmith, their evaluation shows that even while augmenting the set of test cases with 10.000 randomly generated programs, the code coverage only improved by less than 1%.

9.3 Missing Packet-out Comparison

As the intersections between the scopes of the eBPF and P4 is small and because we are lacking other properly implemented back-ends for P4C, packet-out comparison was not conducted. This is limiting the power of P4Fuzz because compiler bugs that manifest at runtime and are specific to different architecture implementations cannot be discovered by the tool. These type of bugs may be found by analyzing the differences in the packets emitted by multiple different devices running the same P4 program using the same in-packet.

Chapter 10

Conclusion

The P4Fuzz project proposes to raise the security of programmable dataplanes by focusing on an un-explored attack surface and pursuing a different approach compared to related work. Our objective was to secure the dataplanes by uncovering bugs in P4 compilers.

Given the case study of a previously discovered problem where the Bit-Wise Negation bug introduced by the P4 compiler proved to be a real threat, we show that bugs at this level can actually introduce security risks for the dataplanes.

An automated tool that generates syntactically and semantically valid P4 programs allows us to stress the compiler and find bugs hidden deeper in the implementation. By identifying these problems we are able to mitigate the security risks that some of them may imply.

Over a relatively short period of time P4Fuzz discovered four distinct bugs in P4C, the standard compiler for P4, as presented in Section 7.2. While none of these bugs represent a security risk, we would like to point out that more bugs are likely to be discovered as:

- the tool keeps running
- the tool implements language specification rules that are currently not supported
- the tool adds support for more architectures
- changes are made in the compilers implementations

By providing an automated way that generates and executes P4 programs, the compiler developer is able to identify and fix the bugs in a much easier way. While not all of the bugs found represent security risks, some of them, as it is the case of Bit-Wise Negation, may be a threat for the dataplane.

Furthermore, P4Fuzz can also be used to improve the quality of services offered by the compilers, to raise the standards, and as a testing tool for the newly developed or modified P4 compilers.

Finally, we addressed a gap in the literature mentioned previously in Chapter 1 by implementing a security tool that focuses on the P4 compiler, rather than the program's expected behaviour or the program's execution paths, as it is the case of ASSERT-P4 or p4pktgen. However the synergy between P4Fuzz and related work can be observed at design and implementation levels, as the papers serve us with the technical background required, while the tools complement our project.

10.1 Future Work

Even though P4Fuzz can be used in its current version to generate P4 programs, test them with specially crafted packets and tame the list of errors, multiple features are still to be added or improved upon. Among the future work that can contribute to the quality of P4Fuzz we can remark:

- **Implement more semantic rules**: while most of the syntax rules are implemented in the P4Fuzz, some of the semantic ones are still missing. The effort and tedious work of extracting the semantics from the informal language specification may however prove very rewarding, as more diverse and complex P4 programs could get generated.
- **Implement Packet-out Comparison module**: as described in Chapter 5.4 the Packet-out Comparison can automate the fuzzing process even more, by comparing outputs of different architectures that execute the same P4 program (i.e: NetFPGA and BMv2), and inform the developer about any differences.
- **Add support for NetFPGA**: an open-source project designed for prototyping network devices. The new architecture would provide on top of the list of possible bugs present in NetFPGA compiler, an additional component to compare against in the Packet-out Comparison module.
- **Run P4Fuzz for a longer period of time**: due to long period of development, the latest version of P4Fuzz ran a relatively short amount of time, producing a not so large data set of results. One interesting case would be to have the fuzzer running on multiple machines for a longer period of time (i.e: 1-2 weeks) and interpret the results, similar to the Csmith approach.
- **P4C code coverage tool**: a tool that can measure the code coverage of the P4C compiler. Being able to analyze the coverage can help both the compilers and the compiler fuzzers. In the case of the compilers, the coverage can be used as an indicator on how well tested the code is. This metric can also help compiler fuzzers in the decision making process regarding fuzzing strategies. The choices can then be based on which of the strategies improve the coverage the most.

Bibliography

[1] Mădălin Claudiu Dănceanu Andrei Alexandru Agape. *Exposing Security Issues in P4 Programmable Software Defined Networks*. 2018.

[2] René Rydhof Hansen Stefan Schmid Andrei Alexandru Agape, Mădălin Claudiu Dănceanu. *Charting the Security Landscape of Programmable Dataplanes*. 2018.

[3] Andy Fingerhut Clark Barrett Peter Athanas Andres Nötzli, Jehandad Khan. p4pktgen: Automated test case generation for p4 programs. https://conferences.sigcomm.org/sosr/2018/sosr18-finals/sosr18-final72.pdf, 2018. Accessed: 2018-05-18.

[4] Lucas Leal Alberto E. Schaeffer-Filho Marinho P. Barcellos Kirill Levchenko Lucas M. Freire, Miguel C. Neves. *Uncovering Bugs in P4 Programs with Assertion-based Verification*. UFRGS, UC San Diego, 2017.

[5] p4lang. p4c github. https://github.com/p4lang/p4c, 2018. Accessed: 2018-06-01.

[6] Eric Eide John Regehr Xuejun Yang, Yang Chen. Finding and understanding bugs in c compilers. http://www.cs.utah.edu/~regehr/papers/pldi11-preprint.pdf, 2011. Accessed: 2018-01-11.

[7] Chaoqiang Zhang Weng-Keen Wong Xiaoli Fern-Eric Eide John Regehr Yang Chen, Alex Groce. Taming compiler fuzzers. https://www.cs.utah.edu/~regehr/papers/pldi13.pdf, 2013. Accessed: 2018-05-18.

[8] P4 Language Consortium. P4 language and related specifications. https://p4.org/p4-spec/, 2018. Accessed: 2018-05-29.

[9] lightreading. P4 runtime: Putting the control plane in charge of the forwarding plane. https://www.youtube.com/watch?v=reqJ3cqpfwo, 2017. Accessed: 2017-12-13.

[10] The P4 Language Consortium. P416 language specification. https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html, 2017. Accessed: 2018-05-24.

[11] Andy Fingerhut (jafingerhut). p4-16-allowed-constructs.png file | github. https://github.com/jafingerhut/p4-guide/tree/1a1362496ab12af9d6839a94b8240878075c30c1, 2018. Accessed: 2018-04-25.

[12] Open Networking Foundation. main.p4 file | github. https://github.com/opennetworkinglab/onos/blob/master/apps/p4-tutorial/pipeconf/src/main/resources/main.p4, 2017. Accessed: 2018-01-11.

[13] p4lang. p4c description. https://github.com/p4lang/p4c, 2017. Accessed: 2017-11-15.

[14] Chris Doss Mihai Budiu. The architecture of the p4-16 compiler. http://p4.org/wp-content/uploads/2017/06/p4-ws-2017-p4-compiler.pdf, 2017. Accessed: 2017-11-15.

[15] Wikipedia. Fuzzing. https://en.wikipedia.org/wiki/Fuzzing, 2018. Accessed: 2018-04-30.

[16] Pedram Amini Michael Sutton, Adam Greene. *Fuzzing: Brute Force Vulnerability Discovery*. 2007.

[17] John Neystadt. *Automated Penetration Testing with White-Box Fuzzing*. 2008.

[18] Wikipedia. Levenshtein distance. https://en.wikipedia.org/wiki/Levenshtein_distance, 2018. Accessed: 2018-05-29.

[19] AlDanial. Cloc (count lines of code). https://github.com/AlDanial/cloc, 2018. Accessed: 2018-06-02.

[20] Andrei Novikov. annoviko/pyclustering: pyclustering 0.8.1 release, May 2018.

[21] SWIG. Swig. http://www.swig.org/, 2018. Accessed: 2018-06-02.

[22] Slim Framework. Slim framework. https://www.slimframework.com/, 2018. Accessed: 2018-06-02.

[23] p4pktgen. p4pktgen. https://github.com/p4pktgen/p4pktgen/blob/master/src/p4pktgen/switch/simple_switch.py, 2018. Accessed: 2018-05-22.

[24] Mastertrap21. Specializing extern objects on the same extern object type, leads to compiler bug: Ir loop detected. https://github.com/p4lang/p4c/issues/1296, 2018.

[25] Mastertrap21. Varbit declaration in structs leading to compilation error. https://github.com/p4lang/p4c/issues/1291, 2018.

[26] Mastertrap21. Error type in nested struct gives compiler bug error. https://github.com/p4lang/p4c/issues/1325, 2018.

[27] antoninbas. Support nested structs in bmv2 compiler backend. https://github.com/p4lang/p4c/issues/562, 2017.

[28] jafingerhut. Wrong result for bit-wise negating then widening a calculation. https://github.com/p4lang/p4c/issues/983, 2017.

[29] Mihai Budiu. Compiling p4 to ebpf | bcc | github. https://github.com/iovisor/bcc/tree/master/src/cc/frontends/p4, 2015. Accessed: 2018-04-25.

Appendix A

Appendix A

A.1 P4 Programs that Contain Bugs

A.1.1 Specializing Extern Objects on the Same Extern Object Type: Compiler Bug IR Loop Detected - Issue #1296

```
1
2  #include <core.p4>
3  #include <v1model.p4>
4
5  struct headers_t {
6  }
7
8  struct metadata_t {
9  }
10
11 extern test_extern<T> {
12     test_extern();
13     void read(out T result);
14     void write(in T value);
15 }
16
17 parser ParserImpl(packet_in packet,
18                   out headers_t hdr,
19                   inout metadata_t meta,
20                   inout standard_metadata_t standard_metadata) {
21     state start {
22         transition select() {
23             default: accept;
24         }
25     }
26 }
27
28 control IngressImpl(inout headers_t hdr,
29                     inout metadata_t meta,
30                     inout standard_metadata_t standard_metadata) {
31     test_extern<test_extern<bit<32>>>() test;
```

```
32      apply {
33      }
34  }
35
36  control EgressImpl(inout headers_t hdr,
37                     inout metadata_t meta,
38                     inout standard_metadata_t standard_metadata) {
39      apply{
40      }
41  }
42
43  control VerifyChecksumImpl(inout headers_t hdr, inout metadata_t meta) {
44      apply {
45      }
46  }
47
48  control ComputeChecksumImpl(inout headers_t hdr, inout metadata_t meta) {
49      apply {
50      }
51  }
52
53  control DeparserImpl(packet_out packet, in headers_t hdr) {
54      apply {
55      }
56  }
57
58  V1Switch(ParserImpl(),
59           VerifyChecksumImpl(),
60           IngressImpl(),
61           EgressImpl(),
62           ComputeChecksumImpl(),
63           DeparserImpl()) main;
```

Listing A.1: Specializing extern objects on the same extern object type: IR loop detected - Issue #1296

A.1.2 Varbit Declaration in Structs: Compiler Bug - Issue #1291

```
1
2   #include <core.p4>
3   #include <v1model.p4>
4
5   struct headers_t {
6   }
7
8   struct metadata_t {
9      varbit<8> test;
10  }
11
12  parser ParserImpl(packet_in packet,
13                    out headers_t hdr,
14                    inout metadata_t meta,
```

```
                  inout standard_metadata_t standard_metadata) {
    state start {
        transition select() {
            default: accept;
        }
    }
}

control IngressImpl(inout headers_t hdr,
                    inout metadata_t meta,
                    inout standard_metadata_t standard_metadata) {
    apply {
    }
}

control EgressImpl(inout headers_t hdr,
                   inout metadata_t meta,
                   inout standard_metadata_t standard_metadata) {
    apply {
    }
}

control VerifyChecksumImpl(inout headers_t hdr, inout metadata_t meta) {
    apply {
    }
}

control ComputeChecksumImpl(inout headers_t hdr, inout metadata_t meta) {
    apply {
    }
}

control DeparserImpl(packet_out packet, in headers_t hdr) {
    apply {
    }
}

V1Switch(ParserImpl(),
         VerifyChecksumImpl(),
         IngressImpl(),
         EgressImpl(),
         ComputeChecksumImpl(),
         DeparserImpl()) main;
```

Listing A.2: Varbit declaration in structs: Compilation error - Issue #1291

A.1.3 Error Type in Nested Struct: Compiler bug - Issue #1325

```
#include <core.p4>
#include <v1model.p4>

struct parsed_packet_t {};
```

```
5
6  struct test_struct {
7    error test_error;
8  }
9
10 struct local_metadata_t {
11   test_struct test;
12 };
13
14 parser parse(packet_in pk, out parsed_packet_t hdr,
15              inout local_metadata_t local_metadata,
16              inout standard_metadata_t standard_metadata) {
17   state start {
18     transition accept;
19   }
20 }
21
22 control ingress(inout parsed_packet_t hdr,
23                 inout local_metadata_t local_metadata,
24           inout standard_metadata_t standard_metadata) {
25   apply { }
26 }
27
28 control egress(inout parsed_packet_t hdr,
29                inout local_metadata_t local_metadata,
30          inout standard_metadata_t standard_metadata) {
31   apply { }
32 }
33
34 control deparser(packet_out b, in parsed_packet_t hdr) {
35   apply { }
36 }
37
38 control verify_checks(inout parsed_packet_t hdr,
39                       inout local_metadata_t local_metadata) {
40   apply { }
41 }
42
43 control compute_checksum(inout parsed_packet_t hdr,
44                          inout local_metadata_t local_metadata) {
45   apply { }
46 }
47
48 V1Switch(parse(), verify_checks(), ingress(), egress(),
49          compute_checksum(), deparser()) main;
```

Listing A.3: Error type in nested struct: Compiler bug - Issue #1325

A.1.4 Error type in nested struct: Compiler bug - Issue #1325

```
1
2  #include <v1model.p4>
```

```
3
4  struct alt_t {
5    bit<1> valid;
6    bit<7> port;
7  };
8
9  struct row_t {
10   alt_t alt0;
11   alt_t alt1;
12 };
13
14 struct parsed_packet_t {};
15
16 struct local_metadata_t {
17   row_t row;
18 };
19
20 parser parse(packet_in pk, out parsed_packet_t hdr,
21              inout local_metadata_t local_metadata,
22              inout standard_metadata_t standard_metadata) {
23   state start {
24     transition accept;
25   }
26 }
27
28 control ingress(inout parsed_packet_t hdr,
29                 inout local_metadata_t local_metadata,
30           inout standard_metadata_t standard_metadata) {
31   apply { }
32 }
33
34 control egress(inout parsed_packet_t hdr,
35                inout local_metadata_t local_metadata,
36          inout standard_metadata_t standard_metadata) {
37   apply { }
38 }
39
40 control deparser(packet_out b, in parsed_packet_t hdr) {
41   apply { }
42 }
43
44 control verify_checksum(in parsed_packet_t hdr,
45                         inout local_metadata_t local_metadata) {
46   apply { }
47 }
48
49 control compute_checksum(inout parsed_packet_t hdr,
50                          inout local_metadata_t local_metadata) {
51   apply { }
52 }
53
54 V1Switch(parse(), verify_checksum(), ingress(), egress(),
55          compute_checksum(), deparser()) main;
```

Listing A.4: Support nested structs in bmv2 compiler backend - Issue #562

A.2 The Simple Partial MAC Address Filter P4 Program

```
#include <core.p4>
#include <v1model.p4>

typedef bit<48> ethernet_address;

typedef bit<9> port_t;
const port_t ACCEPT_PORT = 1;
const port_t DROP_PORT = 0;

header ethernet_t {
    ethernet_address dstAddr;
    ethernet_address srcAddr;
    bit<16> etherType;
}

struct metadata {
    bit<24> vendorOui;
}

struct headers {
    ethernet_t ethernet;
}

parser IngressParserImpl(packet_in buffer,
                         out headers hdr,
                         inout metadata user_meta,
                         inout standard_metadata_t standard_metadata){
    state start {
        buffer.extract(hdr.ethernet);
        transition accept;
    }
}

control ingress(inout headers hdr,
                inout metadata user_meta,
                inout standard_metadata_t standard_metadata){

    action _drop() {
        standard_metadata.egress_spec = DROP_PORT;
    }

    table vendor_ouis {
        key = {
            user_meta.vendorOui : exact;
```

```
45          }
46          actions = {
47              _drop();
48          }
49      }
50      apply {
51          bit<48> extractor = ~((bit<48>) 24w0xffffff);
52          user_meta.vendorOui = (bit<24>) ((hdr.ethernet.srcAddr & extractor
        ) >> 6);
53          standard_metadata.egress_spec = ACCEPT_PORT;
54          vendor_ouis.apply();
55      }
56  }
57
58  control egress(inout headers hdr,
59                 inout metadata user_meta,
60                 inout standard_metadata_t standard_metadata)
61  {
62      apply { }
63  }
64
65  control DeparserImpl(packet_out packet, in headers hdr) {
66      apply {
67          packet.emit(hdr.ethernet);
68      }
69  }
70
71  control verifyChecksum(inout headers hdr, inout metadata meta) {
72      apply { }
73  }
74
75  control computeChecksum(inout headers hdr, inout metadata meta) {
76      apply { }
77  }
78
79  V1Switch(IngressParserImpl(),
80           verifyChecksum(),
81           ingress(),
82           egress(),
83           computeChecksum(),
84           DeparserImpl()) main;
```

Listing A.5: The Simple Partial MAC Address Filter complete implementation

Appendix B

Appendix B

B.1 Using P4Fuzz

In order for other developers to continue our work, we provide the source code of the P4Fuzz on the GitHub repository: https://github.com/andrei8055/p4-compiler-fuzzer

Currently we consider providing two solutions:

- Create a snapshot of the virtual machine that we used for the developing of P4Fuzz. This may have the disadvantage of generating a very big file, but the process is out of box and requires not further configuration or installations
- Using the vagrant file and installation instructions on the GitHub repository. This process may be more tedious, but it does not require the use of any snapshot file, or being limited by other restrictions of the initial development process

Details about either of the solutions shall be updated in the installation instructions file on the repository.