

Towards Runtime Verification of Programmable Switches

Apoorv Shukla¹ Kevin Hudemann^{2,*} Zsolt Vági^{3,*} Lily Hügerich¹ Georgios Smaragdakis¹
Stefan Schmid⁴ Artur Hecker⁵ Anja Feldmann^{6,7}

¹TU Berlin ²SAP ³Swisscom ⁴Faculty of Computer Science, University of Vienna ⁵Huawei ⁶MPI-Informatics ⁷Saarland University

*

Abstract

Is it possible to patch software bugs in P4 programs without human involvement? We show that this is partially possible in many cases due to advances in software testing and the structure of P4 programs. Our insight is that runtime verification can detect bugs, even those that are not detected at compile-time, with machine learning-guided fuzzing. This enables a more automated and real-time localization of bugs in P4 programs using software testing techniques like Taramula. Once the bug in a P4 program is localized, the faulty code can be patched due to the programmable nature of P4. In addition, platform-dependent bugs can be detected. From P4₁₄ to P4₁₆ (latest version), our observation is that as the programmable blocks increase, the patchability of P4 programs increases accordingly. To this end, we design, develop, and evaluate P6 that (a) detects, (b) localizes, and (c) patches bugs in P4 programs with minimal human interaction. P6 tests P4 switch non-intrusively, i.e., requires no modification to the P4 program for detecting and localizing bugs. We used a P6 prototype to detect and patch seven existing bugs in eight publicly available P4 application programs deployed on two different switch platforms: behavioral model (bmv2) and Tofino. Our evaluation shows that P6 significantly outperforms bug detection baselines while generating fewer packets and patches bugs in P4 programs such as `switch.p4` without triggering any regressions.

1 Introduction

Programmable networks herald a paradigm shift in the design and operation of networks. The network devices on the data plane, e.g., switches, that traditionally have fixed and vendor-specific network functionality and rely on proprietary hardware and software, can now be programmed and customized by network operators. The P4 language [1, 2] was introduced to enable the programmability and customization of data plane functionalities in network devices. P4 is an open-source domain-specific language designed to allow programming of packet forwarding planes, and is now supported by a number of network vendors.

While programmable networks enable to break the tie between vendor-specific hardware and proprietary software, they

facilitate an independent evolution of software and hardware. With the P4 language, one can define in a P4 program, the instructions for processing the packets, e.g., how the received packet should be read, manipulated, and forwarded by a network device, e.g., a switch. Nevertheless, with the new capabilities, new challenges in P4 software verification, i.e., ensuring that the software fully satisfies all the expected requirements, have been unleashed. The P4 switch behavior depends on the *correctness* of the P4 programs running on them. We realize that a bug in a P4 program, i.e., a small fault such as a missing line of code or a fat finger error, or a vendor-specific implementation error, can trigger unexpected and abnormal switch behavior. In the worst case, it can result in a network outage, or even a security compromise [3].

Problem Statement. In this paper, we pose the following question: “*Is it possible to detect, localize, and patch software bugs in a P4 program without human involvement?*”. We believe that being able to answer this question, even partially, unlocks the full potential of programmable networks, improves their security, as well as increases their penetration in operational and mission-critical networks.

Recently, a panoply of P4 program verification tools [4–9] has been proposed. These verification systems, however, fail to repair the P4 program containing bugs. Most of them [4–7] aim to statically verify user-defined P4 programs which are later, compiled to run on a target switch. They mostly find bugs that violate the memory safety properties, e.g., invalid memory access, buffer overflow, etc. Furthermore, they are prone to false positives and are unable to verify the *runtime* behavior on real packets. In addition, classes of bugs, e.g., checksum-related, ECMP (Equal-Cost Multi-path) hash calculations-related or platform-dependent bugs, cannot be detected by static analysis approaches. Since, runtime verification aims to verify the *actual* behavior against the *expected* behavior of a switch by passing specially-crafted input packets to the switch and observing the behavior, such verification is complementary to static analysis. Note, the detection of bugs causing the abnormal runtime behavior is a complex and challenging task. In particular, the P4 switch does not throw any runtime exceptions. Furthermore, the detection of bugs can be a nightmare if there is no output, i.e., packets are silently dropped instead of being forwarded. Thus, the runtime verification of the switch behavior is crucial.

A useful approach to verify the runtime behavior is fuzz testing or fuzzing [10–20], a well-known dynamic program

*Kevin Hudemann and Zsolt Vági worked on this paper while they were affiliated with TU Berlin.

testing technique that generates semi-valid, random inputs which may trigger abnormal program behavior. However, for fuzzing to be efficient, intelligence needs to be added to the input generation, so that the inputs are not rejected by the parser and it maximizes the chances of triggering bugs. This becomes crucial especially in networking, where the input space is huge, e.g., a 32-bit destination IPv4 address field in a packet header has 2^{32} possibilities. With the 5-tuple flows, the input space gets even more complex and large. To make fuzzing more effective, we consider the use of machine learning, to guide the fuzzing process to generate smart inputs that trigger abnormal target behavior. Recently, Shukla et al. [20] have shown that Reinforcement Learning (RL) [21, 22] can be used to train a verification system. We build upon [20] by adding (a) static analysis to the fuzzing process to significantly reduce the input search space, and thus, adding input structure awareness, and (b) support for platform-dependent bug detection.

Even if a bug in a P4 program is detected, the localization of code statements in the P4 code that are responsible for the bug, is non-trivial. The difficulty stems from the fact that practical P4 programs can be large with a dense conditional structure. In addition, the same faulty statements in a P4 program may be executed for both passed as well as failed test cases and thus, it gets hard to pinpoint the actual faulty line/s of code. Tarantula [23–25] is a dynamic program analysis technique that helps in fault localization by pinpointing the potential faulty lines of code. To localize the software bugs, we tailor Tarantula for generic software to P4 programs by building a localizer called P4Tarantula and integrating it with the bug detection of machine learning-guided fuzzing. In this paper, we combine these two approaches to detect and localize bugs in P4 programs in real-time.

P6. We, however, realize that automated program repair [26] is an uncharted territory and becomes increasingly important as the software development lifecycle in programmable networks is short [27] with insufficient testing. In this paper, we show that due to the structure of P4 programs, it is possible to automate the patching of platform-independent bugs in P4 programs, if the patch is available. To this end, we present P6, P4 with runtime *Program Patching*, a novel runtime P4-switch verification system that (a) detects, (b) localizes, and (c) patches software bugs in a P4 program with minimal human effort. P6 improves the existing work based on machine learning-guided fuzzing [20] in P4 by extending it and augmenting it with: (a) automated localization, and (b) runtime patching. P6 relies on the combination of static analysis of the P4 program and Reinforcement Learning (RL) technique to guide the fuzzing process to verify the P4-switch behavior at runtime.

P6 in a nutshell. In P6, the first step is to capture the expected behavior of a P4 switch, which is accomplished using information from three different sources: (i) the control plane

configuration, (ii) queries in p4q (§3.2.1), a query language which we leverage to describe expected behavior using conditional statements, and (iii) accepted header layouts, e.g., IPv4, IPv6, etc, learned via static analysis of the P4 program. If the *actual* runtime behavior to the test packets generated via machine-learning guided fuzzing differs from the *expected* behavior through the violation of the p4q queries, it signals a bug to P6 which then identifies a patch from a library of patches. If the patch is available, P6 modifies the original P4 program to no longer trigger the bug signaled by the p4q queries. Then, the patched P4 program is subjected to sanity and regression testing.

We develop a prototype of P6 and evaluate it by testing it on eight P4₁₆ application programs from switch.p4 [28], P4 tutorial solutions [29], and NetPaxos codebase [30] across two P4 switch platforms, namely, behavioral model version 2 (bmv2) [31] and Tofino [32]. Our results show that P6 successfully detects, localizes and patches diverse bugs in all P4₁₆ programs while significantly outperforming bug detection baselines without introducing any regressions.

Contributions. Our main contributions are:

- We design, implement, and evaluate P6, the first end-to-end runtime P4 verification system that detects, localizes, and patches bugs in P4 programs non-intrusively. (§3)
- We observe that the success of P6 relies on the increased patchability of P4 program from old (P4₁₄) to the new version (P4₁₆). We confirm this by studying the code of P4 applications in two different P4 versions (P4₁₄, P4₁₆). (§2)
- We present a P6 prototype and report on an evaluation study. We evaluate our P6 prototype on a P4 switch running eight P4₁₆ programs (including switch.p4 with 8715 LOC) from publicly available sources [28–30] across two platforms, namely, behavioral model and Tofino. Our results show that P6 non-intrusively detects both platform-dependent and platform-independent bugs, and significantly outperforms state-of-the-art bug detection baselines. (§5)
- We show that in case of platform-independent bugs, P6 can localize bugs and fix the P4 program, when the patch is available, without causing any regressions or introducing new bugs in an automated fashion. (§3, §5)
- To ensure reproducibility and facilitate follow-up work, we will release the P6 software and library of ready patches for all existing bugs in the P4 programs to the research community.

2 Challenges & Opportunities in P4 Verification

In this section, we describe the P4 packet processing pipeline. Then, we outline the challenges in discovering bugs in P4 programs or switches, and we motivate the need for runtime verification. We conclude this section by characterizing the evolution and structure of P4 programs, that, to our surprise provides opportunities for automated P4 program patching.

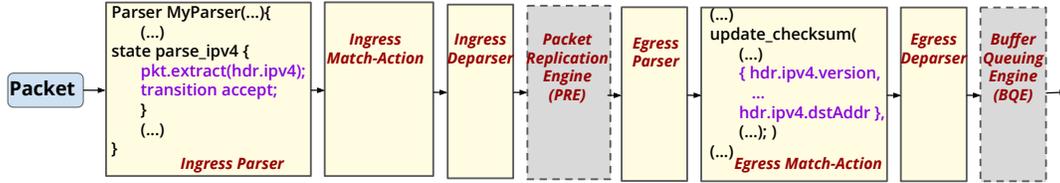


Figure 1: An example of a platform-independent bug in P4₁₆ packet processing pipeline.

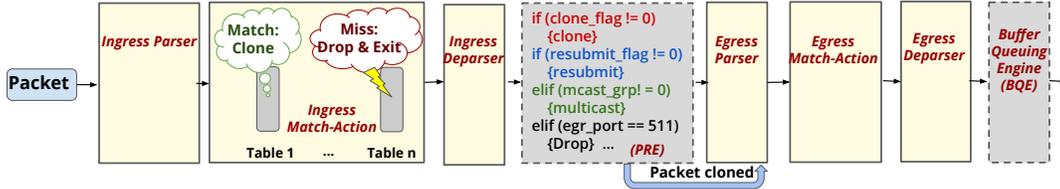


Figure 2: An example of a platform-dependent bug in P4₁₆ packet processing pipeline.

2.1 Packet Processing Pipeline of P4

P4 [1, 2] is a domain-specific language comprising of packet-processing abstractions, e.g., headers, parsers, tables, actions, and controls. The P4 packet processing pipeline evolved from [33] to its current form P4₁₆ [2], see Figure 1. In P4₁₆ packet processing pipeline, there are six programmable blocks that are platform-independent, namely, ingress parser, ingress match-action, ingress deparser, egress parser, egress match-action, and egress deparser. The programmable blocks are annotated with a solid line in Figures 1 and 2.

The ingress parser transforms the packet from bits into headers according to a parser specification provided by the programmer. After parsing, an ingress match-action (also called ingress control function) decides how the packet will be processed. Then, the packet is queued for egress processing in the ingress deparser. Upon dequeuing, the packet is processed by an egress match-action (also called egress control function). The egress deparser specification dictates how packets are deparsed from separate headers into a bit representation on output, and finally, the packet leaves the switch. Note that both ingress and egress match-actions (control functions) direct the packet through any number of match-action tables.

In the P4₁₆ packet processing pipeline, there are also two platform-dependent blocks (annotated with dashed lines in Figures 1 and 2), that rely on proprietary implementations of the hardware vendors and are non-programmable. These blocks are the packet replication engine (PRE) and the buffer queuing engine (BQE).

2.2 Challenges: Runtime Bugs in P4

Bugs or errors can occur at any stage in the P4 pipeline. If a bug occurs in any of the programmable blocks, then the bug is platform-independent and software patching can solve the problem. If the bug appears in the non-programmable or platform-dependent blocks, namely, the PRE or BQE, then the vendor has to be informed to fix the issue as the implementation is vendor-specific. P4 program verification systems [4–7]

are able to detect bugs using static analysis. Unfortunately, static analysis is (i) prone to false positives, (ii) cannot detect platform-dependent bugs, and (iii) cannot detect runtime bugs that require to actively send real packets.

As an example, consider the scenario in Figure 1 (solid line blocks) that illustrates part of the implementation of Layer-3 (L3) switch, provided in the P4 tutorial solutions [29]. Here, the parser does not check if the IPv4 header contains IPv4 options or not, i.e., if the IPv4 `ihl` field is equal to 5 or not. When updating the IPv4 checksum of the packets during egress processing, IPv4 options are not taken into account, hence for those IPv4 packets with options, the resulting checksum is wrong causing such packets to be forwarded and *incorrectly* dropped at the next hop. This leads to anomalies in network behavior. Other bugs that fall in this category are those related to IPv4/6 checksum in the packet (see Figure 5 later). Such bugs are *platform-independent*, as they only result from programming errors.

Figure 4 illustrates a simple scenario where due to a bug or fault, the packet reaching a P4 switch has a time-to-live (TTL) field in the IP header with value as 0. The expected behavior is that the packet gets dropped, but currently, there is no such check in the P4 code. Thus, the switch forwards the packet and decreases the value of the TTL field, causing it to be increased to 255 incorrectly. This happens due to wraparound caused by underflow in an 8-bit field. Such an anomaly can be non-trivial to detect and localize. In addition, it can be responsible for the abnormal behavior of a P4 switch.

As illustrated in Figure 5, the problem lies in the fact that the P4 program fails to specify that the IPv4 checksum inside the packet needs to be verified before forwarding. An adversary can easily intercept the packets and modify them (e.g., as a Man-in-the-Middle attack (MitM)) and does not even need to recalculate the checksum. Therefore, additional information can be inserted even into encrypted packets. When such a malicious or malformed packet arrives at the P4 switch, it selects the corresponding action based on the match-action tables and forwards the packet without verifying the checksum. Such checksum-related bugs may inflict serious

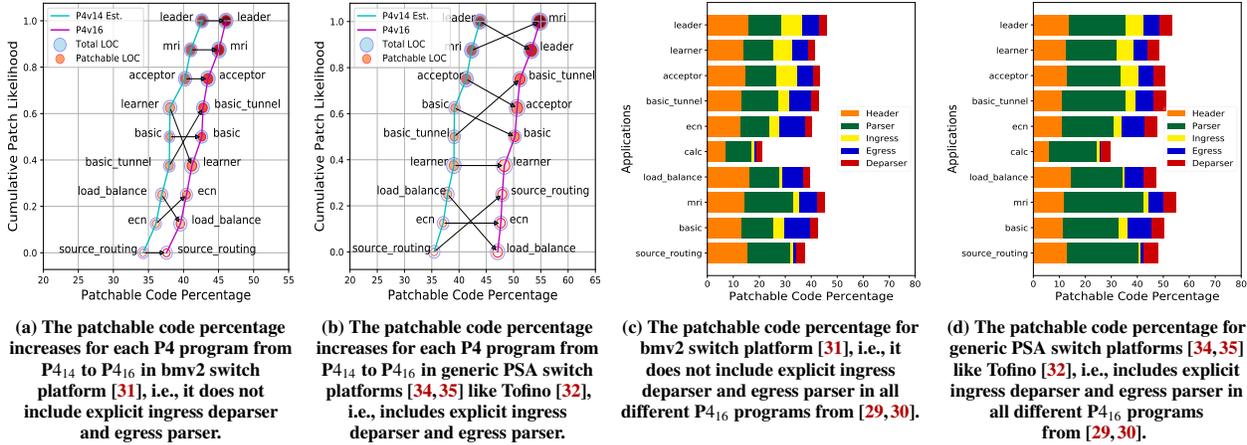


Figure 3: Evolution of the P4 program structure from P4₁₄ to P4₁₆ version.

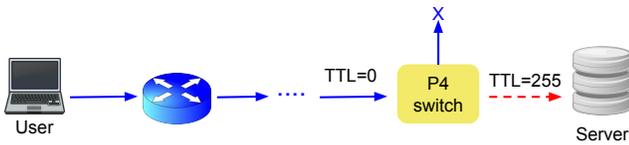


Figure 4: P4 switch running the P4 program does not check if the time-to-live (TTL) value in the packet is 0. Blue arrows show the expected, red dashed arrows show the actual path.

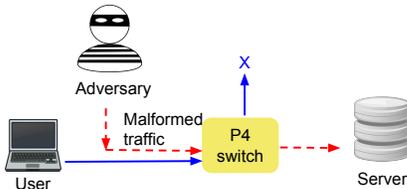


Figure 5: P4 switch running the P4 program does not check the faulty IPv4 checksum in the packet. Blue arrows show the expected path, red dashed arrows show the actual path.

damages to critical servers and can be a nightmare to debug.

For a *platform-dependent bug*, consider the scenario shown in Figure 2 (dashed line blocks). Here, we assume a P4 program implements at least two match-action tables. Any table except the last one could be a longest prefix match (LPM) table, offering unicast, clone and drop actions (ingress match-action block). The last match-action table implements an access control list (ACL). So, the packets can either be dropped or forwarded according to the chosen actions by the previous tables. In this case, it is possible that conflicting forwarding decisions are made. Consider packets are matched by the first table (Table 1) and a clone decision is made, later, those are dropped by the ACL table (Table n). In such a case, the forwarding behavior depends on the implementation of the PRE, which is platform-dependent. The implementation of PRE of the SimpleSwitch target in the behavioral model (bmv2) is illustrated in Figure 2. It would drop the original packet, however, forward the cloned copy of the packet. Similar bugs can occur, if instead of the clone action, the resubmit action is chosen (blue). Similarly, another bug can be found when implementing multicast (green).

The above motivates us to turn our attention to runtime

detection of bugs. Runtime verification is a useful and complementary tool in the P4 verification repertoire that detects both *platform-independent bugs* resulting from programming errors as well as *platform-dependent bugs*.

2.3 Opportunities for Patching: The Structure of a P4 Program

In the evolution of P4, there are two recent versions: P4₁₄ [36] and P4₁₆ [2]. P4₁₆ allows programmers to use definitions of a target switch-specific architecture, PSA (Portable Switch Architecture) [34,35]. P4₁₆ is an upgraded version of P4₁₄. In particular, a large number of language features have been eliminated from P4₁₄ and moved into libraries including counters, checksum units, meters, etc., in P4₁₆. P4₁₄ allowed the programmer to explicitly program three blocks: ingress parser (including header definitions of accepted header layouts), ingress control and egress control functions. Recall that P4₁₆ allows to explicitly program six programmable blocks (see Figure 1).

By analyzing programs in the P4₁₄ and P4₁₆ versions, we realize that as more blocks of the P4 program get programmable, there is more onus on the programmer to write a program that behaves as expected (when it gets compiled and deployed on the P4 switch). Missing checks or fat finger errors can cause havoc in the network. However, this is a blessing in disguise as the more programmable the code is, the more patchable it is. Thus, programming errors can be fixed. Figures 3a and 3b illustrate that the potentially patchable code percentage increases from P4₁₄ to P4₁₆ in all applications (excluding calculator) from P4 tutorial solutions [29] and NetPaxos codebase [30] in behavioral model (bmv2) switch platform [31] and other generic PSA switch platforms [34,35], e.g., Tofino [32] respectively. Figures 3c and 3d illustrate the patchable code percentage in the latest P4₁₆ version. The patchable code percentage comes from the six programmable blocks in P4₁₆. Roughly, whatever is programmable, is patchable. In principle, around 40-45% of a P4 program is patchable in P4₁₆ programs for behavioral model (bmv2) switch

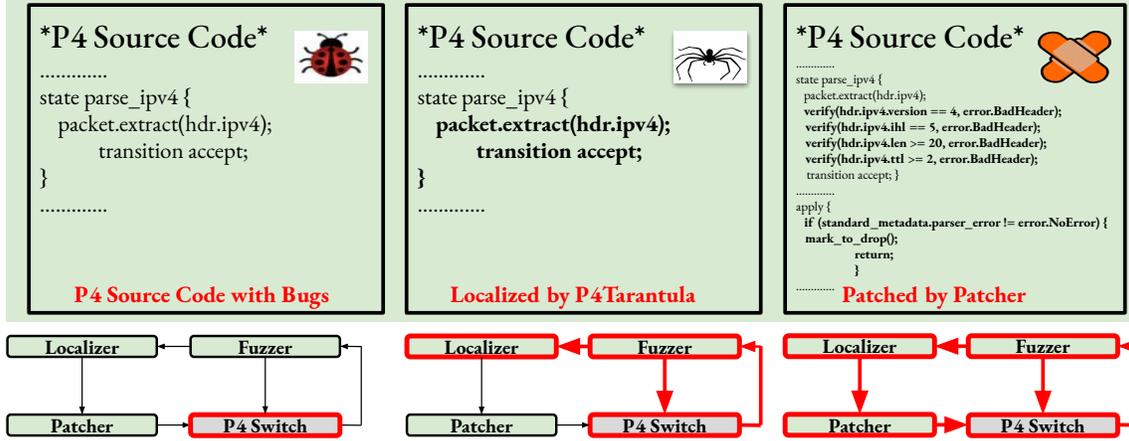


Figure 6: P6 in Action: depicting the automated detection, localization and patching of a bug in a L3 switch P4 program [29].

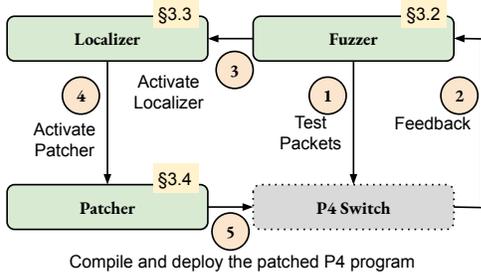


Figure 7: P6 Workflow. Modules of P6 (in solid green boxes).

platform [31] (Figure 3c). This increases to 50-55% if the ingress deparser and egress parser are programmable for other target switch platforms, e.g., Tofino [32] (Figure 3d). More importantly, the parser and header definitions account for 20-40% of the total patchable code.

Observation 1: From P4₁₄ to P4₁₆, the P4 program possesses twice as many programmable blocks doubling the opportunities for patching.

If there is no bug in the parser/header code, the incoming packets with invalid header values will be dropped as expected and the packets with valid header values will be transmitted to the upcoming blocks else the invalid or semi-valid packets will incorrectly pass through the parser and may trigger abnormal runtime behavior. Assuming there was no bug in the parser/header code, those valid packets that are transmitted to the next blocks may still exhibit abnormal runtime behavior, if the programmable blocks containing, e.g., the application code logic, deparser have bug/s or due to bug/s in the platform dependent part which is vendor implementation-specific.

Observation 2: Once, a bug is detected and localized in the platform-independent part of a P4 program, it is patchable; a platform-dependent bug is not patchable, however, the vendor can be informed if and when detected.

3 P6: System Design

3.1 P6: Overview

The goal of P6 (see Figure 7) is to detect, localize and patch the software bugs in a P4 program at runtime with minimal human effort. This is achieved by verifying the *actual* runtime behavior against the *expected* behavior of a P4 switch running a pre-compiled P4 program to the incoming packets.

The P6 system contains three main modules:

- (1) **Fuzzer:** Generates test packets using RL-guided fuzzing, static analysis, and p4q queries (§3.2.1) to the P4 switch running the pre-compiled P4 program. (§3.2)
- (2) **Localizer:** P4Tarantula is the Localizer which pinpoints faulty lines of code causing bugs in the P4 program. (§3.3)
- (3) **Patcher:** Automates patching of the bugs localized by P4Tarantula Localizer, if patchable. Then, Patcher compiles and loads the patched P4 program on the P4 switch. (§3.4)

P6 Workflow. P6 is a closed-loop control system. Through a pre-generated dictionary from control plane configuration, p4q queries, and static analysis of a P4 program, the expected runtime behavior of the P4 switch is captured and sent as an input to the Fuzzer containing the RL Agent and the Reward System (§3.2). As shown in Figure 7, the Fuzzer selects appropriate mutation actions such as add/delete/modify bytes in a packet to generate test packets towards the P4 switch running the pre-compiled P4 program (1). If the actual runtime behavior towards the packets defies the expected behavior through the violation of the p4q queries, it signals a bug in the form of a reward as a feedback to the Reward System which is then, exploited by the RL Agent to improve the training process by selecting better mutation actions on the packet (2). After the bug detection, the Fuzzer automatically triggers Localizer (§3.3), P4Tarantula (only for platform-independent bugs; for platform-dependent bugs, the vendor is informed) which pinpoints the faulty line of code (3) to trigger the Patcher (§3.4) which searches for the appropriate patch from a library of patches for the corresponding

P4 program (4). If the patch is available, Patcher modifies the original P4 program, compiles and loads it on the P4 switch and checks if the bug is no longer triggered by p4q queries by repeating the whole-cycle and executing sanity and regression testing (5). Note, P6 is non-intrusive and thus, requires no modification to the P4 program for testing before patching.

P6 in Action. Before we dive into the details of Fuzzer, Localizer and Patcher, we demonstrate the operation of P6. Figure 6 illustrates how P6 detects, localizes, and patches an existing bug in a layer-3 (L3) switch P4 source code (program) from [29] in an automated fashion. The left part of Figure 6 shows the P4 program containing a platform-independent bug in the parser code, i.e., no header field validation is implemented, hence all IPv4 packets are *incorrectly* accepted by the parser. After the P4 program is deployed on the P4 switch, P6 is triggered. Initially, the Fuzzer detects the bug violating the corresponding p4q query based on the feedback (reward) received from the P4 switch. Then, it triggers the P4Tarantula for localization (shown in the center of Figure 6) where it pinpoints the problematic part of the code (highlighted). Afterwards, the Patcher is triggered automatically, patching the necessary problematic parts of the code, i.e., adding header field verification statements (highlighted in right), after checking if the patch was indeed missing from the P4 program. Finally, Patcher automatically compiles [37] and deploys the patched P4 program on the P4 switch, and triggers P6 to ensure that the patches caused no regressions and fixed the detected bug.

3.2 Fuzzer: RL-guided Fuzzing

The goal of Fuzzer is to detect the runtime bugs discussed in §2.2. We improve [20] by augmenting Fuzzer with the static analysis of a P4 program which makes the Fuzzer aware of the input structure or accepted header layouts, e.g., IPv4, IPv6, etc. and thus, it significantly reduces the input search space. Indeed, techniques to further reduce the input search space within the accepted headers are discussed in [38], which can be augmented to static analysis. We guide the mutation-based white-box fuzzing [12] via RL [21, 22]. The feedback in the form of *rewards* is received from the switch based on the evaluation of *actual* against *expected* runtime behavior. Note, the expected behavior is determined using the static analysis, the control plane configuration, i.e., forwarding rules and p4q queries (§3.2.1). p4q queries are conditional queries (if-then-else) where each query has multiple conditions and each condition acts as a test case. A violation of a test case represents a bug detection.

Reinforcement Learning (RL). Reinforcement learning [21, 22] is a machine learning technique that aims at enabling an *Agent* to learn how to interact with an environment, based on a series of reinforcements, i.e., rewards or punishments received from the target environment, in our case, a switch.

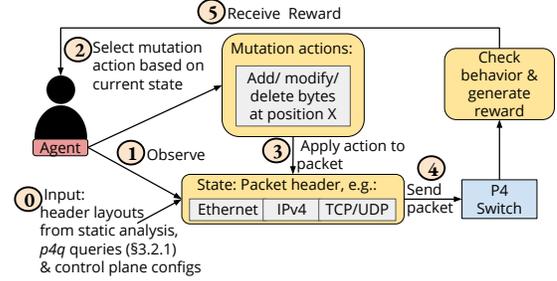


Figure 8: Fuzzer. *Reward System* (in yellow) and *Agent* (in pink).

The *Agent* observes the switch and chooses an action to be executed. After the action is executed, the *Agent* receives a reward or punishment from the switch. While the goal of learning is to maximize the rewards, we argue it is equally crucial to design a machine learning model which is general enough for any kind of target environment. To detect the bugs triggered by fuzzing, one can observe the output of the target switch in response to the input packets. Thus, reinforcement learning allows developing a *Reward System* where feedback in the form of rewards from the switch trains the *Agent* and thus, guides the fuzzing process.

In our RL-based model, we define states, actions, and rewards as follows:

States: The sequence of bytes forming the packet header.

Actions: The set of mutation actions for each individual packet header field, e.g., add, modify or delete bytes at a given position in the packet header. Note, the add and modify actions either use random bytes or bytes from a pre-generated dictionary (explained below).

Rewards: The *Agent* can immediately receive the reward, after a mutated packet was sent to the target switch and the results of the execution are evaluated. It is likely to experience sparse rewards when most of the sent packets do not trigger any bug. Thus, the reward is defined as 0, if the packet did not trigger a bug and 1, if the packet successfully triggered a bug.

The input to the Fuzzer is a dictionary (hereafter, referred to as *dict*) that comprises information extracted from static analysis, the control plane configuration, and the queries defined with p4q (§3.2.1). The static analysis is used to derive the input structure awareness such as accepted header layouts and available header fields in the P4 program. The control plane configuration comprises the forwarding table contents and the platform-dependent configuration. Boundary values for the header fields may be extracted from the p4q queries, i.e., when queries explicitly compare packet header fields with values, e.g., TTL > 0.

Figure 8 depicts the Fuzzer workflow. In step 0 (initialization), the *Reward System* receives the *dict* as an input. Then, the *Agent* observes the current state or the current packet header (see the initialization in §3.2.2). The observed state is the input for the neural networks of the *Agent* (§3.2.2), which outputs the appropriate mutation action. The selected action is applied for the given packet, and the packet is sent to the

```

1 (ing.hdr.ipv4 &                                     Query 1
2   ing.hdr.ipv4.chksum != calcChksum() ,
3   egr.egress_port == False, )
4 (ing.hdr.ipv4 & ing.hdr.ipv4.ver != 4,             Query 2
5   egr.egress_port == False, )
6 (ing.hdr.ipv4 & ing.hdr.ipv4.ihl < 5,             Query 3
7   egr.egress_port == False, )
8 (ing.hdr.ipv4 &                                     Query 4
9   [ing.hdr.ipv4.len < ing.hdr.ipv4.ihl * 4 |
10  ing.hdr.ipv4.len < 20],
11  egr.egress_port == False, )
12 (ing.hdr.ipv4 & ing.hdr.ipv4.ttl < 2,             Query 5
13  egr.egress_port == False, )
14 # IPv4 Unicast                                     Query 6
15 (ing.hdr.ipv4,
16  egr.hdr.eth.srcAddr == ing.hdr.eth.dstAddr &
17  egr.hdr.eth.dstAddr == table_val() &
18  egr.hdr.ipv4.ttl == ing.hdr.ipv4.ttl-1 &
19  egr.hdr.ipv4.chksum == calcChksum() &
20  egr.egress_port == table_val(), )
21 # IPv4 Clone                                       Query 7
22 (ing.hdr.ipv4,
23  egr.hdr.eth.srcAddr == ing.hdr.eth.dstAddr &
24  egr.hdr.eth.dstAddr == table_val() &
25  egr.hdr.ipv4.ttl == ing.hdr.ipv4.ttl-1 &
26  egr.hdr.ipv4.chksum == calcChksum() &
27  egr.egress_port IN {clone_sess}), )
28 # IPv4 Multicast                                   Query 8
29 (ing.hdr.ipv4,
30  egr.hdr.eth.srcAddr == ing.hdr.eth.dstAddr &
31  egr.hdr.eth.dstAddr == table_val() &
32  egr.hdr.ipv4.ttl == ing.hdr.ipv4.ttl-1 &
33  egr.hdr.ipv4.chksum == calcChksum() &
34  egr.egress_port IN {mcast_grp}), )

```

Figure 9: p4q Queries. Queries 1-6 represent platform-independent, and Query 7-8 represent platform-dependent queries respectively.

P4 switch. After the packet is processed by the switch, the behavior is evaluated, the reward of 1 is generated when the p4q query specifying the expected behavior is violated and returned to the *Agent*. In particular, the packet which was sent to the P4 switch is saved together with a final *verdict* (pass or fail). A packet’s *verdict* is considered *either* passed: if the generated reward is equal to 0, i.e., *actual* runtime behavior matches *expected* behavior when the p4q query is not violated *or* failed: if the generated reward is equal to 1, i.e., *actual* runtime behavior does not match *expected* behavior when the p4q query is violated. Then, the *Agent* (§3.2.2) uses the received reward to improve the action selection in subsequent executions (exploitation).

3.2.1 p4q: Query Language

Before diving into the details of the *Agent* training, we explain the query language, p4q [20], used for specifying the *expected* switch behavior. To achieve the goal of an automated runtime verification system, P6 system must query the *actual* runtime behavior of a P4 switch against a specification defining the *expected* behavior. To extend the query repertoire of p4q from [20], we augment it with platform-dependent queries. In a nutshell, p4q queries are used to compare *expected* against *actual* switch behavior.

p4q queries. In a p4q query, the behavior is expressed using `if-then-else` statements in the form of tuples. The programmer can specify conditions for packets to fulfill at ingress of the switch (`if`), with corresponding conditions to fulfill at egress (`then`). In addition, the programmer can describe alternative conditions (`else`), e.g., if the condition of the `then`

expr:		assign:		var:
expr & expr	Conjunction	var == var	Equality	header_field
expr expr	Disjunction	var < var	Less than	header_field_value
expr ^ expr	Exclusive disjunction	var > var	Greater than	int
!expr	Negation	var <= var	Less or Equal	table_val()
assign		var >= var	Greater or Equal	isCorrect()
int:		var != var	Not Equal	
int + int	Addition	var IN {...}	Is element of	
int - int	Subtraction			
int * int	Multiplication			
int				

Figure 10: p4q Grammar.

branch is not fulfilled at egress. To automate the usage of P6, an option to execute all the queries of p4q with a single command is provided (see §4). To define these conditions, the p4q syntax and grammar are used.

p4q Grammar. Figure 10 depicts the grammar and constructs defined in p4q. The p4q grammar allows common boolean expressions and relational operators as they can be found in many programming languages like C, Java or Python, to ease the work for the programmer. The boolean expressions and relational operators have the same semantics as common logical operators and expressions. Variables can either be integers, header fields, header field values, or the evaluation result of the primitive methods, e.g., `calcChksum()` and `table_val()`. Each header has a prefix (`ing.` or `egr.`) indicating if it is the packet arriving at ingress or exiting the switch at egress.

Figure 9 illustrates an example of how the packet processing behavior of an IPv4 layer 3 (L3) switch, written in P4, can be queried easily using p4q. Query 1 (lines 1-3), defines that incoming packets with a wrong IPv4 checksum are expected to be dropped. Similarly, the following four queries (lines 4-13) express the validation of the IPv4 version field, the IPv4 header length, the packet length and the IPv4 time-to-live (TTL) field for packets at ingress of the switch respectively. However, there are also conditions for packets at the egress of the switch. These conditions are described by Query 6 (lines 15-20). Namely, changing source and destination MAC addresses to the correct values, decrementing the TTL value by 1, recalculating the IPv4 checksum and emitting the packet on the correct port as instructed by the control plane configuration (forwarding rules). Query 7 (lines 22-27) corresponds to the platform-dependent part of the switch (PRE) and defines conditions for packets that are cloned by the switch. Such packets need to fulfill the same conditions as per Query 6, but the egress port should correspond to the clone session configuration of the target switch. Similarly, Query 8 (lines 29-34) expresses the conditions for multicast packets that need to fulfill the same conditions as per Query 7 but the egress ports should correspond to the configured multicast group configuration of the target switch.

Note, queries written in p4q can be extended, reused and provided in the form of libraries. More importantly, the p4q queries help in the availability of a library of pre-defined patches for the corresponding violations. Note, an easily extensible interface is provided to augment p4q further with

Algorithm 1: Agent Training

Input: Empty prioritized experience replay memory M , uninitialized online and target network
Output: Trained online and target network models

```
1 Initialize online network with random weights
2 Initialize target network with copy of online network parameters
3 for  $i = 1$  to  $num\_episodes$  do
4   Initialize byte sequence  $b_1$ 
5   Preprocess  $b_1$  to get the initial state  $s_1 = preprocess(b_1)$ 
6   for  $step = 1$  to  $max\_ep\_len$  do
7     Select action  $a_{step}$  randomly with probability  $\epsilon$  (exploration) or use
      online network to predict  $a_{step}$  (exploitation)
8     Execute action  $a_{step}$ , observe reward  $r_{step}$  and byte sequence  $b_{step+1}$ 
9     Set  $s_{step+1} = preprocess(b_{step+1})$ 
10    Save the transition  $(s_{step}, a_{step}, r_{step}, s_{step+1}, terminal)$  in  $M$ 
11    Sample batch of transitions  $(s_j, a_j, r_j, s_{j+1})$  from  $M$ 
12    
$$y_j = \begin{cases} r_j & \text{if terminal} \\ r_j + \gamma * Q(s_{j+1}, \max_a Q(s_{j+1}, a; \Theta), \Theta') & \text{otherwise} \end{cases}$$

13    Perform stochastic gradient descent using categorical cross entropy
      loss function
```

user queries as per the deployment scenario to allow detection of more bugs.

3.2.2 Agent

The *Agent* houses the RL algorithm (Algorithm 1), which is inspired by Double Deep Q Network (Double DQN) [39], an improved version of Deep Q Networks (DQN) [40].

Double Deep Q Network (DDQN). DDQN algorithm [39] is a recently-developed algorithm based on Q-learning [22], hence a *model-free* reinforcement learning algorithm. *Model-free* means, the *Agent* does not need to learn a model of the dynamics of an environment and how different actions affect it. This is beneficial, as it can be difficult to retrieve accurate models of the environment. At the same time, the goal is to provide sample efficient learning, i.e. reduce the number of packets sent to the target switch, makes the DDQN a suitable choice. The basic concept of the algorithm is to use the current state (packet header) as an input to a neural network, which predicts the action the *Agent* shall select to maximize future rewards. In addition, Double DQN algorithm splits action selection in a certain state from the evaluation of that action. To achieve, it uses two neural networks: (i) the online network responsible for action selection, and (ii) the target network evaluating the selected action. This improves the learning process of the *Agent*, as overoptimism of the future reward when selecting a certain action, is reduced and thus, helps to avoid overfitting.

Prioritized Experience Replay. Experience replay [41] was introduced to eliminate problems of oscillation or divergence of parameters, resulting from correlated data. To overcome this problem, the experiences of the *Agent*, i.e., a tuple comprising the current state, predicted action, reward received, and resulting state are saved in the memory of *Agent*. To enable learning by experience replay, the neural network model is updated using random samples from past experiences. To counter the scenario of sparse rewards, a simple form of pri-

Algorithm 2: P4Tarantula (Localizer)

Input: P4 source code (SC), sent packets (Ps) and corresponding *verdicts* (V)
Output: $S[j]$ - suspiciousness score for the corresponding line j
// $V[p]$ represents the *verdict* about packet p (pass or fail)
// $SC[j]$ represents line j of the source code
// **Initialization**

```
1 totalFailed = 0, totalPassed = 0
2 foreach  $p$  in  $Ps$  do
3   if  $V[p] == pass$  then
4     totalPassed += 1
5   else
6     totalFailed += 1
7   end
8   follow  $p$  through  $SC$ :
9   foreach executed line  $j$  in  $SC$  do
10    if  $V[p] == pass$  then
11       $SC[j].pass += 1$ 
12    else
13       $SC[j].fail += 1$ 
14    end
15    
$$S[j] = \frac{SC[j].fail / totalFailed}{SC[j].pass / totalPassed + SC[j].fail / totalFailed}$$

16  end
17 end
18 call Patcher
```

oritized experience replay, inspired by [42], is applied. The memory is sorted by absolute reward and each experience is prioritized by a configurable factor and the index.

Agent Training Algorithm. Algorithm 1 presents the training algorithm of the *Agent* in the P6 system. For our algorithm, we rely on the use of Multi-Layer Perceptron (MLP) [43]. In the initialization phase, the weights of online and target neural networks are initialized (Lines 1-2). For each execution, the current state is reinitialized by randomly choosing a packet header in byte representation from a pre-generated set of packet headers. The corresponding bytes are then converted to a sequence of float representations (Lines 4-5). An ϵ -greedy policy is applied to determine the action to be executed (Line 7). Applying an ϵ -greedy policy means that during training of the *Agent*, an action is selected randomly by the *Agent* with probability ϵ to ensure sufficient exploration. As the training progresses, probability ϵ is decreased linearly until a lower bound is reached. This helps in reducing overfitting as well, since the *Agent* never stops exploring the effects of other actions on the environment during training. The determined action will be executed, the result is observed and saved in the experience memory (Lines 8-10). As a last step, a sample out of the experience memory is selected to calculate y_j which is used to calculate the categorical cross-entropy loss and perform the stochastic gradient descent step to update the network weights (Lines 11-13).

3.3 Localizer: P4Tarantula

P4Tarantula is the Localizer or the bug localization module of P6. P4Tarantula is based on a dynamic program analysis technique for generic software, Tarantula [23, 25]. In case a bug is discovered by Fuzzer, it automatically notifies P4Tarantula. *Note, P4Tarantula will not be notified in case of platform-dependent bugs, as they are neither localizable nor patchable.*

As an input, P4Tarantula uses the P4 program or source code, the packets that were sent by Fuzzer as per the `p4q` query (test cases) to trigger the bug and the pass or fail *verdict* corresponding to those sent packets. Recall, a *verdict* corresponds to a condition of the `p4q` query which acts as a test case.

Algorithm 2 presents the localization algorithm used by P4Tarantula. First, P4Tarantula initializes two counters, measuring the number of passed or failed *verdicts* corresponding to the sent packets (Line 1). In the next step, P4Tarantula increments the counters according to the *verdicts* made for the given packet (Lines 3-7). Now, the P4 source code needs to be traversed line-by-line (similar to symbolic execution but with actual packet header values to avoid all possible header values), to find the code execution path for the given packet (Line 8). For each line in the P4 source code that is executed for the given packet, counters for the corresponding *verdicts* are incremented (Lines 10-14). For the executed lines of the P4 source code, a *suspiciousness score* [25] is calculated (Line 15). The suspiciousness score is between 0 and 1 as the same line/s can be executed for passed and failed *verdicts* corresponding to packets. This score corresponds to the likelihood that a line of code is causing a potential bug. The closer it is to 1, the more likely it is that the corresponding line of code is problematic. Finally, the P4 source code lines are ordered as per their suspiciousness score to localize the bug. Then, Patcher is notified.

More details on code traversal by P4Tarantula. While implementing P4Tarantula, we accurately traverse the P4 program execution path for any given packet. To overcome, we have implemented our code-traversal solution as a part of a script responsible for calculating the suspiciousness scores for the lines of the source code of the P4 program. We follow the execution in the source code of the P4 program from the “start” state of the parser until the end of the execution, i.e., the deparser stage of the P4 packet processing pipeline. By following the execution of the program as soon as it receives a given packet containing header values, we can determine how the different conditions in a P4 program are evaluated for the packet and follow the correct branch of the P4 program at different branching points.

3.4 Patcher

Patcher is the novel automated patching module of the P6 system. If a bug is localized by P4Tarantula, it notifies Patcher. The input for Patcher is the P4 source code, the results of static analysis of the P4 source code, the localization results of P4Tarantula, and the violated `p4q` query. Patcher compares the localized problematic parts of the code with appropriate available patches. Note, Patcher comes with a library of patches for P4 programs, i.e., those which violate `p4q` queries. Nevertheless, it can be easily extended when, previously unseen bugs, e.g., bugs in application code logic, are detected.

From the results of the static analysis, Patcher can extract

Algorithm 3: Patcher

Input: P4 source code (*SC*), static analysis results (*Sr*), localization results (*Lr*) and violated `p4q` query (*q*)
Output: A patched version of the source-code (*PSC*)
// The patcher offers a patch only for those lines where the suspiciousness score ≥ 0.5

```

1 Import & process user-defined parser state names, header and header field
  names, metadata and metadata field names from Sr required for patches in the
  patch-library
2 for lines in Lr do
3   if Suspiciousness score  $\geq 0.5$  then
4     check corresponding line/s of code pinpointed by P4Tarantula
5     if the patch is missing and violating q then
6       apply the preferred patch
7     else
8       inform the programmer
9     end
10    Goto next line
11  else
12    Goto next line
13  end
14 end
15 Compile & re-deploy the patched P4 program (PSC) and notify Fuzzer for
  testing the patches and regressions

```

the needed parser state names, header names, header field names, metadata names and metadata field names for the patches in the current version of the library of patches. In P4, metadata is used to pass information from one of the programmable or non-programmable blocks to another.

Note, in most P4 programs (including the publicly available programs from [28–30]) no variables apart from user-defined names for parser states or header/metadata fields are present. Thus, with the gathered knowledge about user-defined names Patcher can compare through, e.g., regex or string comparison, if the patch (correct code) is already present in the P4 source code or if missing, the patch needs to be applied. Note, if the patches in the patch library require the analysis of custom variables or stateful components, e.g., registers and meters, the comparison if the patch is present or not requires further analysis of the code.

In case no appropriate patch is available, the programmer is informed by the Patcher. After Patcher finishes the execution, it calls the P4 compiler (`p4c`) to re-compile the patched version of the P4 program and triggers the re-deployment of the code on the P4 switch. In addition, the Fuzzer is notified by Patcher to test the patched program again, to confirm the patches and ensure no regressions were caused by the patches by testing via the `p4q` queries and executing regression testing.

A patch has the following properties: (a) preferably, few lines of code, e.g., missing checks in parser, (b) makes the P4 program conform to the expected behavior, (c) passes the sanity testing or checks for basic functionality, and (d) does not cause regressions breaking existing functionality elsewhere.

Algorithm 3 shows the Patcher algorithm. First, Patcher imports the needed header or metadata field names, as well as parser state names for the currently available patches in the library of patches. Then, for each line in the localization results, Patcher checks if the suspiciousness score is greater than the

defined threshold of 0.5¹ (Line 2), as it is highly likely that the corresponding line of code is responsible for triggering the detected bug. In case the suspiciousness score is above the defined threshold, Patcher will check the corresponding line of code. The Patcher, then, checks if the patch is available, e.g., through string comparison with the appropriate patch to be applied for the violated p4q query. If the patch is indeed missing, then the problematic line of code is patched, else the programmer is informed as the appropriate patch is not available (Lines 3-8). Once, all the localization results are processed (Lines 9-12), the patched P4 program is compiled by triggering the compiler (p4c) to be re-deployed on the P4 switch and the Fuzzer is triggered to re-test the patched code (Line 14).

When is the Patcher automated? Currently, the library of patches exists for platform-independent bugs, i.e., those violating p4q queries 1-6 in Figure 9. If the bug exists in a deployment-specific application code logic, then the programmer can be informed by the Patcher to provide patches. Recall, for platform-dependent bugs, the P4Tarantula will not be invoked and the vendor will be informed accordingly.

4 P6 Prototype

We develop a P6 prototype using Python version 3.6 with $\approx 3,100$ lines of code (LOC); Fuzzer with $\approx 2,200$ LOC, P4Tarantula with ≈ 490 LOC and Patcher with ≈ 430 LOC. Fuzzer is implemented using Keras [44] library with TensorFlow [45] backend and Scapy [46] for packet generation and monitoring. Currently, P6 only supports programs written in P4₁₆ [2] as the P4 compiler (p4c [37]) supports the translation of programs written in P4₁₄ [36] to P4₁₆. The Agent was trained separately, for each condition of each query written in p4q. The training process as well as the later execution using the trained Agents, however, can be parallelized. For queries described in Figure 9, the trained model of the Agent can be reused for testing different P4 programs that implement IPv4 packet processing.

In addition to the modules described in §3, we implement a control plane module using P4Runtime [47] and Python. For the P4 switch, we rely on software switches supporting P4₁₆, namely behavioral model (bmv2) [31] with SimpleSwitch-Grpc target (Version 1.12.0), and Barefoot Tofino Model [32] (Version 8.3.0). To simplify and automate the usage of P6, a default option is provided where all queries of p4q are executed by: P6 'p4/source_code_location' -default

5 Evaluation

In this section, we evaluate the verification capabilities of P6.

¹Threshold is configurable as per deployment scenario.

Bug IDs	Bugs	Queries (Figure 9)
1	Accepted wrong checksum (PI)	Query 1
2	Generated wrong checksum (PI)	Query 6 (Line 19)
3	Incorrect IP version (PI)	Query 2
4	IP IHL value out of bounds (PI)	Query 3
5	IP TotalLen value is too small (PI)	Query 4
6	TTL 0 or 1 is accepted (PI)	Query 5
7	TTL not decremented (PI)	Query 6 (Line 18)
8	Clone not dropped (PD)	Query 7 (Line 27)
9	Resubmitted packet not dropped (PD)	Query 6 (Line 20)
10	Multicast packet not dropped (PD)	Query 8 (Line 34)

Table 1: Bugs (with Bug IDs) detected by the P6 prototype through the violation of the corresponding p4q queries (in Figure 9). Note, PI and PD refer to platform-independent and -dependent respectively.

5.1 Baselines

We compare P6 against three baseline fuzzing approaches:

(1) Advanced Agent. The first baseline is an Advanced Agent only relying on random fuzz action selection, i.e., without prioritized experience replay. Thus, Advanced Agent can execute the same mutation actions as P6, but cannot learn which actions lead to rewards. It represents the intelligent baseline.

(2) IPv4-based fuzzer. The second baseline is an IPv4-based fuzzer, which is aware of the IPv4 header layout and randomizes the different available header fields, except IP options fields and the destination IP as it prevents the packets from being dropped by the forwarding rules of the P4 switch. The actual behavior is evaluated using the queries of p4q.

(3) Naïve fuzzer. The third baseline is a simple naïve fuzzer, which is not aware of any packet header layouts. It generates and sends Ethernet frames from purely random mutation of bytes. The actual behavior is evaluated using the p4q queries.

5.2 Bugs

Table 1 provides an overview of *existing* bug types (with bug IDs) detected in the publicly available P4 programs from [28–30] by the P6 prototype. These bugs are detected as they violate the corresponding p4q queries (from Figure 9). In total, P6 prototype can detect 10 distinct bugs in the P4 programs. Out of these 10 bugs, 7 are patchable platform-independent (bugs 1 – 7), and 3 are platform-dependent bugs (bug 8 – 10).

Platform-independent bugs. The two detected bugs with bug ID 1 and 2, are related to wrong IPv4 checksum computation and missing checksum validation. P6 is able to detect, localize and patch these bugs. The four bugs with IDs 3 – 6 are missing or wrong IPv4 packet header validation. Specifically, missing validation of IP version (bug 3), IPv4 header length (bug 4), IPv4 total length (bug 5) and IPv4 time-to-live (TTL) (bug 6). P6 can detect, localize and patch these bugs. While the current approaches *may* be able to detect the bugs, they still lack localization and patching of the bugs. The last of the platform-independent bugs is faulty TTL decrement bug (bug 7). In case, the P4 program accepts packets with IPv4 TTL 0, the TTL decrement is still executed, causing an incorrect increment of TTL to 255. Note, these 7 platform-independent bugs already exist in the publicly available P4

P4 ₁₆ Applications	bmv2 (LOC)	Tofino (LOC)	% increment
basic.p4 [29]	181	257	41.9
basic_tunnel.p4 [29]	219	302	37.9
advanced_tunnel.p4 [29]	242	316	30.6
mri.p4 [29]	277	372	34.3
netpaxos-acceptor.p4 [30]	270	327	21.1
netpaxos-leader.p4 [30]	259	323	24.7
netpaxos-learner.p4 [30]	288	355	23.2
switch.p4 [28]	8715	- ²	-

Table 2: P4₁₆ Programs’ LOC in bmv2 and Tofino.

programs of [29, 30]. In `switch.p4` program [28], bugs with IDs 1,2,4 and 5 exist.

Platform-dependent bugs. In addition to the aforementioned platform-independent bugs, P6 is able to detect three platform-dependent bugs. The first bug (Bug ID 8) is described in Figure 2 and occurs when using ingress-to-egress clone action. It violates Query 7 in Figure 9 leading to incorrect forwarding of cloned packets when they are supposed to be dropped. The second bug (Bug ID 9) involves the resubmit operation. Packets with the resubmit metadata field set which are marked to be dropped in a later stage, will be incorrectly resubmitted, i.e., the packet will be processed again, starting at the ingress parser. Thus, packets that are not expected to be resubmitted will be processed again. In the worst case, this can lead to packets being resubmitted over and over again or other unexpected behavior. It violates Query 6 in Figure 9. The third bug (Bug ID 10) involves the multicast operation. If a packet is marked to be dropped and later the `mcast_grp` metadata field is set, then the multicasted copies of the packet are incorrectly forwarded and do not get dropped. It violates Query 8 in Figure 9. Note, we found all platform-dependent bugs specifically, in `basic.p4` program [29] on bmv2 platform [31]. Furthermore, these bugs *cannot* be detected by current approaches that are based on static analysis.

5.3 Experiment Strategy

For conducting our experiments and to evaluate P6 prototype, we ran P6 together with the P4 switch and control plane module in a Vagrant [48] environment with VirtualBox [49]. We emulate the network shown in Figure 13. For each program, separate Vagrant machines, each with 10 CPU cores and 7.5 GiB RAM, are used. The Vagrant machines ran on a server running Debian 9 OS (Version 4.9.110), with Intel Xeon CPU and 256 GiB RAM. Each experiment was executed ten times on each of the eight P4₁₆ programs shown in Table 2 from P4 tutorials [29], NetPaxos [30] and `switch.p4` [28] repository. For each of the ten runs, 9 test-cases were executed, where each test-case corresponds to one condition of the queries 1 – 6 illustrated in Figure 9. Note, `basic.p4` program has 10 test-cases as it is also tested using query 7 for the platform-dependent bug. Furthermore, we currently trained the *Agent* separately for each test-case and sequentially execute the test cases. This can, however, be parallelized easily. We observed

²`switch.p4` is not publicly available for Tofino.

that for only two conditions of the p4q queries, no bugs were detected.

5.3.1 Experiment Topology

Figure 13 shows the topology used for the experiments as part of the P6 system evaluation. In total, five virtual machines are used. One of the machines is running the P4 switch, which is connected to all other machines. The controller is connected to the P4 switch, in order to deploy the P4 program and fill the forwarding tables. The P6 system is connected to one port of the switch for sending the packets generated by Fuzzer. Two virtual machines act as the receiver for these packets. The feedback of the receivers is used by the P6 system to evaluate the actual behavior of the P4 switch. In addition, the machine running P6 is connected to the controller to trigger re-deployment of the patched P4 program, in case a bug is detected.

5.4 Metrics

In particular, we ask the following questions:

- Q1. How much time does P6 take to detect, localize, and patch all bugs? (§5.4.1)
- Q2. How does P6 perform against the baselines? (§5.4.2)
- Q3. How many rewards does P6 generate against the baseline of an Advanced Agent for Agent training? (§5.4.3)
- Q4. How many packets does P6 generate to detect bugs against the baselines? (§5.4.4)
- Q5. What is the accuracy of P6? (§5.4.5)

5.4.1 Performance of P6

To evaluate the performance of P6, we execute the detection, localization, and patching on 8 publicly available P4 programs from the P4 tutorials [29], NetPaxos [30] and `switch.p4` [28] repository with minimal manual efforts.

Figure 11a and 11d show the median bug detection time of P6 over ten runs for the different programs using bmv2 SimpleSwitchGrpc and Barefoot Tofino Model, respectively. Note, `switch.p4` program is only available for bmv2 and was not tested using Tofino. In all runs on bmv2 except for `switch.p4` program, P6 was able to detect all bugs in less than two seconds. In `switch.p4`, P6 was able to detect all bugs in less than ten seconds. The detection time is higher for `switch.p4` as compared to the other tested programs since more packets get dropped making bug detection more difficult. On Tofino, the median detection time was slightly higher for four out of seven programs. The reason for the increased bug detection time with the NetPaxos programs [30] can be due to the instrumentation of these programs by us to make them run on CPU intensive Tofino Model.

Figures 11b and 11e illustrate the median bug localization time of P6 for the different programs using bmv2 SimpleSwitchGrpc and Barefoot Tofino Model. Overall, all bugs

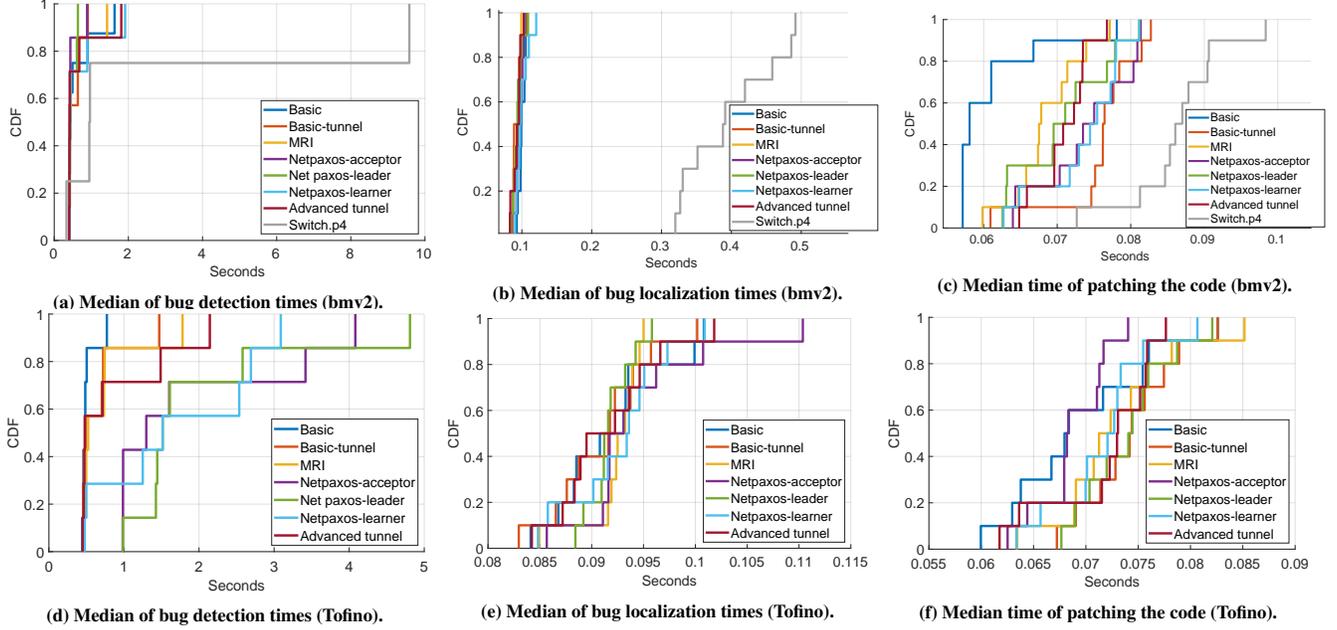


Figure 11: Bug detection, localization and patching times of different P4 programs in bm2 and Tofino. Each plot represents a median over 10 runs.

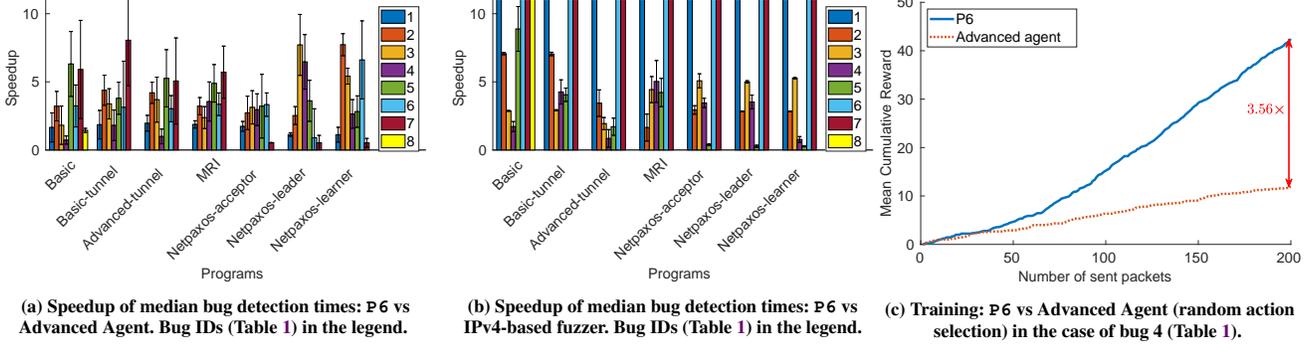


Figure 12: P6 vs Baselines. Each plot represents a median over 10 runs.

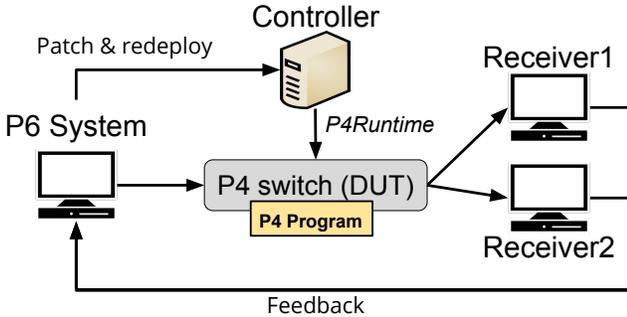


Figure 13: Experiment Topology

for 7 of the programs were localized by P6 in just above 0.12 seconds on bm2 and Tofino. To our surprise, the bug localization time for switch.p4 program running on bm2 is only increased by a factor of 4 \times , even though the program has about 30 \times more lines of code compared to the other tested programs (see Table 2). The median time of patching the code is shown in Figures 11c and 11f for bm2 and Tofino respectively. P6 is able to patch the P4 programs with millisecond

scale performance (max. 98 milliseconds).

5.4.2 P6 vs Baselines: Detection Time

We compare P6 against the three baseline approaches in terms of bug detection time. We observe that the Advanced Agent baseline, see Figure 12a (with quartiles), was able to detect all the bugs present in the tested programs, which is due to the similarity with the P6 Agent. Advanced Agent, however, cannot learn from the rewards, hence generates more packets and thus, takes more time to detect the bugs than P6 Agent. IPv4-based fuzzer was only able to detect 4 out of 10 bugs in the seven programs from [29, 30]. For switch.p4 program [28], IPv4-based fuzzer was able to detect 3 out of 4 bugs which were IPv4-based. In Figure 12b (with quartiles), the speedup is defined as infinite for the test-cases where IPv4-based fuzzer could not detect the bug. Accordingly, the bars representing these test-cases range until the top of the figure. Note, Naive fuzzer was not able to detect bugs at all, even though generating 16k packets.

Figure 12a shows the speedup (Advanced Agent/P6 Agent)

P4 ₁₆ Applications	P6	Advanced Agent	IPv4-based	Naïve
basic.p4 [29]	13	59	8,035	16,000
basic_tunnel.p4 [29]	11	59	6,044	14,000
advanced_tunnel.p4 [29]	12	57	6,038	14,000
mri.p4 [29]	10	61	6,058	14,000
netpaxos-acceptor.p4 [30]	11	52	6,021	14,000
netpaxos-leader.p4 [30]	9	49	6,024	14,000
netpaxos-learner.p4 [30]	12	44	6,026	14,000
switch.p4 [28]	28	113	2,132	14,000

Table 3: P6 vs Baselines. Median #packets sent per run over 10 runs.

for all bugs detected in the seven tested programs from [29,30]. The results show that P6 Agent can detect bugs up to $10.96\times$ faster than the Advanced Agent baseline. Only bug 7 was detected faster by the Advanced Agent in 3 of the 7 P4₁₆ applications tested as the Advanced Agent needs less time for random action selection than P6 Agent for intelligent action selection, based on its neural networks. In addition, Advanced Agent can make use of the same mutation actions and the pre-generated dict, hence when triggering the bug, the overall execution time will be slightly lower than that of P6 Agent. In 94% of the test-cases, Advanced Agent required more time and packets to detect the bugs than the P6 Agent. For switch.p4 program [28], the results show that P6 Agent is able to detect bugs up to $30\times$ faster than the Advanced Agent baseline.

Figure 12b shows the speedup (IPv4-based fuzzer/P6 Agent) for all bugs detected in the seven tested programs from [29,30]. For the test-cases where IPv4-based fuzzer was able to detect the bug, we observe that in 89% of the test-cases P6 Agent is able to detect the bugs faster while sending significantly fewer packets. P6 Agent outperforms IPv4-based fuzzer by up to $8.88\times$ even though IPv4-based fuzzer sends packets at a higher rate. For switch.p4 program, the results show that P6 Agent is able to detect bugs up to $30\times$ faster than IPv4-based fuzzer.

5.4.3 P6 vs Advanced Agent Training

To verify that P6 Agent is able to effectively learn to detect bugs, we compare P6 Agent against an Advanced Agent, that only relies on random action selection. This makes Advanced Agent similar, but not as intelligent as P6 Agent. Advanced Agent can still execute the same mutation actions but is not able to reason about which actions lead to maximized rewards. Figure 12c shows a comparison of the *mean cumulative reward* (MCR) of the training process of both agents for bug ID 4 of Table 1. We observe that the P6 Agent is able to outperform the baseline by a factor of $3.56\times$ for the mentioned case. Especially, the *prioritized experience replay* helps the P6 Agent to quickly learn about which actions lead to reward, hence trigger bugs in the program. Since, the P6 Agent is trained only using experiences which are *valuable* for the training.

P6 Agent is trained for each condition of each query described by Figure 9 using the same set of hyper-parameters.

More results. Figure 14 shows the training comparison re-

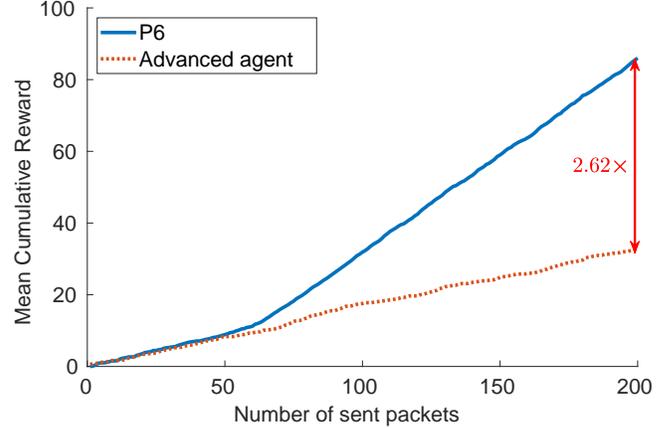


Figure 14: Training: P6 vs Advanced Agent (random action selection) in the case of bug ID 2 in Table 1.

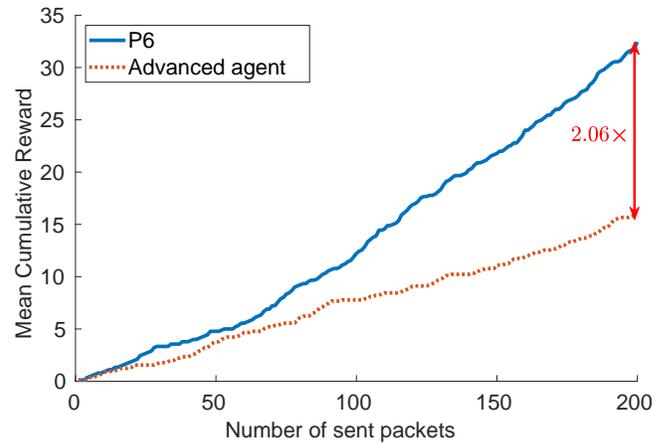


Figure 15: Training: P6 vs Advanced Agent (random action selection) in the case of bug ID 5 in Table 1.

sults for the bug ID 2 (Generated wrong checksum) in Table 1. Also, in this case, the P6 Agent is able to outperform the Advanced Agent baseline by a factor of $2.62\times$.

The training comparison results for the bug ID 5 (IP TotalLen Value is too small) can be seen in Figure 15, showing that P6 Agent is able to outperform Advanced Agent baseline by a factor of $2.06\times$.

In the case of the bug ID 6 (TTL 0 or 1 is accepted), P6 Agent is able to outperform Advanced Agent baseline by a factor of $2.11\times$, as illustrated by Figure 16.

Figure 17 shows that P6 Agent outperforms Advanced Agent for the bug ID 7 (TTL not decremented) by a factor of $2.03\times$. These results show the clear advantage of P6 Agent over the Advanced Agent baseline.

Above results show P6 Agent consistently outperforms Advanced Agent in MCR for other queries. Overall, this shows the clear advantage of P6 Agent over the baselines and the ability to detect bugs with fewer packets.

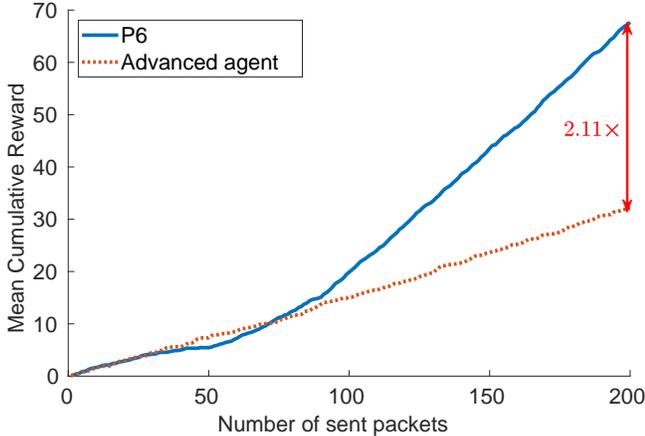


Figure 16: Training: P6 vs Advanced Agent (random action selection) in the case of bug ID 6 in Table 1.

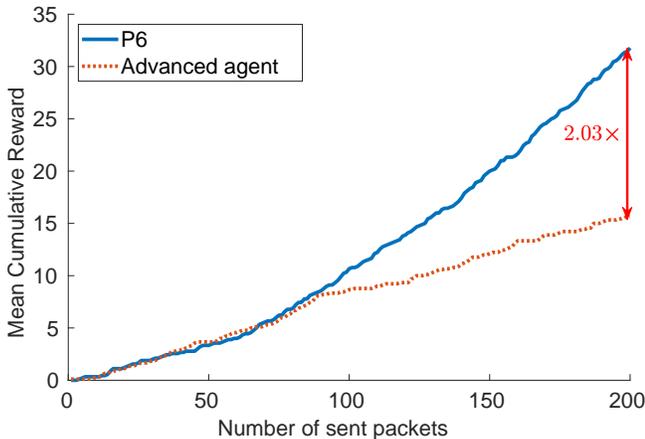


Figure 17: Training: P6 vs Advanced Agent (random action selection) in the case of bug ID 7 in Table 1.

5.4.4 P6 vs Baselines: Dataplane Overhead

Table 3 illustrates the number of packets sent by P6 and the baselines. This shows the usefulness of the P6 Agent which generates less packets by learning about rewards, and generates packets that trigger bugs. In this case, the Advanced Agent is almost similar. IPv4-based fuzzer can detect 4 out of 10 bugs, but generates around 6k packets per run. For each test-case, naïve fuzzer sends around 2k packets (in total between 12k and 16k) but it was not able to trigger any bug.

5.4.5 P6 Accuracy

During the experiments, we could not observe any false positives in bug detection. Note, false positives can occur if the p4q queries are not correct and complete. In the case of bug localization by P4Tantula, false positives are not observed. We did not observe any false positives in patching by the Patcher as it only fixes the code if the correct code is missing. Note, if localization results by P4Tantula contain false positives, then Patcher is prone to false positives as it makes the patching decisions based on the localization results.

Summary. Our results show that P6 due to RL-guided fuzzing significantly outperforms the baselines across two different platforms: bmv2 and Tofino in terms of detecting runtime bugs (including platform-dependent bugs) with minimal data-plane overhead non-intrusively. We observe that most of the platform-independent bugs existed in the parser or header part, otherwise packets with invalid headers get rejected. P6 accurately and swiftly localizes and patches (millisecond scale) the bugs due to the P4 program structure in an automated fashion.

6 Related Work

Verification of programmable networks has been in a constant state of flux. Approaches like [50–52] perform modeling of the network from the control plane to check the reachability, loop freedom, and slice isolation. ATPG [53] generates test packets based on control plane configuration using [50] for functional and performance verification in traditional networks and SDNs (Software-defined Networks). All of the aforementioned tools [50–53], however, assume that the control plane has a consistent or correct view of the data plane in traditional IP-based networks or SDNs only. P6 does not assume the correctness of the control plane and observes the runtime behavior to detect, localize and patch the software bugs in P4 switches. [54–56] use different machine learning approaches for finding security vulnerabilities or compiler specific-bugs which cause crashes, however, they are insufficient for network-related verification. P6 executes switch verification to identify the bugs in a P4 switch.

Currently, most of the P4-related verification techniques, use static analysis of P4 programs using symbolic execution [4, 6, 7] or Hoare logic [5]. The static analysis is prone to false positives as it analyzes the P4 program without passing any real inputs, e.g., packets. Therefore, checksum-related bugs where computations are required on input packets and platform-dependent bugs cannot be detected. Such bugs require P6-like runtime verification. In addition, [5–7] require a P4 program to be manually annotated by the programmer which is cumbersome and prone to manual errors whereas P6 is non-intrusive as it does not require to modify P4 program for bug detection and localization. p4pktgen [8] focuses on locating errors in the toolchains used to compile and run P4 code, e.g., p4c, and uses symbolic execution to create exemplary packets which can execute a selected path in the program. However, it cannot detect platform-dependent bugs or egress pipeline bugs. Such a verification method can complement our solution. P4NOD [57] statically models the network, however, it does not check how the actual P4 switches behave upon receiving the malformed packets e.g., incorrect IPv4 checksum. Cocoon [58] suggests refinement-based programming for network verification. While this approach tries to ensure that programs match their specification, it requires a huge amount of additional and manual user input. For runtime veri-

Related work in P4	Runtime Verification	Detection	Localization	Patching	Detection of PD bugs
Cocoon [58]	×	✓	✓	×	×
Vera [4]	×	✓	×	×	×
p4v [5]	×	✓	×	×	×
P4-ASSERT [6, 7]	×	✓	×	×	×
P4NOD [57]	×	✓	×	×	×
p4pktgen [8]	×	✓	×	×	×
P4CONSIST [9]	✓	✓	×	×	×
P4RL [20]	✓	✓	×	×	×
P6	✓	✓	✓	✓	✓

Table 4: Related work in P4 verification. PD corresponds to the platform-dependent bugs. Note, ✓ denotes the capability, and × denotes the missing capability.

fication, such a formal method is insufficient. Recently, [59] propose data-plane primitive for detecting and localizing bugs: tracking each packet’s execution path through the P4 program by augmenting P4 programs. However, this remains an in-progress and intrusive approach as it requires augmenting P4 code whereas P6 does not change P4 program. In-band network telemetry (INT) [60, 61] enables to collect telemetry data from each switch, however, unlike P6, it cannot localize or patch bugs if packets get dropped. Recently, Shukla et al. proposed P4CONSIST [9], a system that gathers the control and data plane states independently for comparison to verify the control-data plane consistency of P4 SDNs by detecting the path violations for critical flows in P4, however, without localization or patch support. In [20], a machine learning guided-fuzzing system is used to only detect platform-independent bugs in P4 programs. In the context of fuzzing, two approaches like [10] and [17] are worth-considering. [10] is insufficient as it uses program coverage feedback to guide fuzzing without knowing which mutations lead to bugs. [17] transforms the target program to remove sanity checks for fuzzing, however, it is intrusive as the target program requires modification for testing.

Unlike P6, P4-based verification approaches [4–9, 20, 57, 58, 60, 61] are insufficient in localizing and patching the runtime bugs in P4 programs. Besides, they cannot detect the platform-dependent bugs. Table 4 illustrates the capabilities of other P4 verification tools as compared to P6.

7 Discussion

Traditionally, fuzz testing or fuzzing is known to offer a partial testing solution as it is prone to false negatives. Rice’s theorem [62] states that all the non-trivial, semantic properties of a program are undecidable. Semantic properties refer to the behavior of a target program for all inputs. Therefore, if fuzzing does not detect any problems, it does not ensure that there is no problem at all. Statistical techniques like Good-Turing frequency estimations [63–65] for fuzz testing partially, aid in inferring the probability that the next generated test input leads to the discovery of a previously unseen species.

In general, the quality of the input seeds, e.g., the relevance of mutations to the target and program coverage serve as good indicators to assess the quality of a fuzzer. However, there is a tradeoff between speed and precision of fuzz testing as there is an instrumentation overhead involved in generating

a dictionary of meaningful inputs for effective testing and significantly reducing the input search space as compared to random mutations of bits in the inputs.

Machine learning techniques like reinforcement learning [21, 22] helps to some extent with the training of the models based on the feedback from the target. However, *what is a good feedback?* is debatable. Traditionally, feedback depends on the coverage but it all boils down to the program under test. In addition, machine learning is as good as the input training data and thus, offers insufficient guarantees as instead of learning, the model may memorize. A generalized model applicable to *any* kind of training data is highly desirable as it avoids the problems of overfitting and underfitting [66].

Dynamic program analysis technique for fault-localization like Tarantula [23–25] benefits from utilizing the information from multiple failed test cases which helps it in leveraging the richer information base. Therefore, it helps to have at least one failed test case. In addition, allowing tolerance for passed test cases that occasionally execute faults is essential for an effective fault-localization technique.

Software patching facilitates in fixing the software code errors. The patches, however, *may* cause regressions which reflect in the form of abnormal behavior of the software. Sanity testing and modular code design facilitate in ensuring that basic functionality is not affected by the patch. However, one cannot assure that there are no other problems (false negatives) caused by the fix.

We note that leveraging programmability, future programmable networks will encompass even more possibilities of faults with a mix of vendor-code, reusable libraries, and in-house code. As such the general problem of network verification will persist and we will have to explore how to extend the P6 system to traditional IP-based networks.

8 Conclusion

We presented P6, the first system that enables runtime verification of P4 switches in a non-intrusive fashion. P6 uses static analysis- and machine learning-guided fuzzing to detect multiple runtime bugs which are then, localized and patched on the fly with minimal human effort. Through experiments on existing P4 application programs, we showed that P6 significantly outperforms the baseline bug detection approaches to detect existing platform-independent and -dependent bugs. In the case of platform-independent bugs, P6 takes advantage of the increased programmable blocks to localize them and repair the P4₁₆ programs, if and when a patch is available.

We believe P6 is an important foray into self-driving networks [67], which come with stringent requirements on dependability and automation. With P6, developers of P4 programs and operators of P4-enabled devices can improve the security of their products. As a part of our future agenda, we plan to apply P6 on commercial-grade P4 programs and networks to report on our experience. We will also release the

P6 software and library of ready patches that we, respectively, developed and used in this work.

References

- [1] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming Protocol-independent Packet Processors. *ACM CCR*, 44(3), 2014.
- [2] P4 Language Consortium. P4₁₆ language specs, version 1.1.0, 2018.
- [3] P. Kazemian. Network path not found? <https://bit.ly/2FzpEEZ>, 2017.
- [4] R. Stoenescu, D. Dumitrescu, M. Popovici, L. Negreanu, and C. Raiciu. Debugging P4 programs with Vera. In *ACM SIGCOMM*, 2018.
- [5] J. Liu, W. Hallahan, C. Schlesinger, M. Sharif, J. Lee, R. Soulé, H. Wang, C. Caşcaval, N. McKeown, and N. Foster. P4v: Practical Verification for Programmable Data Planes. In *ACM SIGCOMM*, 2018.
- [6] M. Neves, L. Freire, A. Schaeffer-Filho, and M. Barcellos. Verification of P4 Programs in Feasible Time Using Assertions. In *ACM CoNEXT*, 2018.
- [7] L. Freire, M. Neves, L. Leal, K. Levchenko, A. Schaeffer-Filho, and M. Barcellos. Uncovering Bugs in P4 Programs with Assertion-based Verification. In *ACM SOSR*, 2018.
- [8] A. Nötzli, J. Khan, A. Fingerhut, C. Barrett, and P. Athanas. P4pktgen: Automated test case generation for p4 programs. In *ACM SOSR*, 2018.
- [9] A. Shukla, S. Fathalli, T. Zinner, A. Hecker, and S. Schmid. P4CONSIST: Towards Consistent P4 SDNs. In *IEEE Journal on Special Areas in Communication (JSAC)- NetSoft*, 2020.
- [10] M. Zalewski. American Fuzzy Lop: A Security-oriented Fuzzer. <http://lcamtuf.coredump.cx/afl/>, (visited on 6/4/2019), 2010.
- [11] LlvM Compiler Infrastructure: libfuzzer: a library for coverage-guided fuzz testing. <http://llvm.org/docs/LibFuzzer.html>.
- [12] P. Godefroid, M. Y. Levin, and D. Molnar. SAGE: white-box fuzzing for security testing. *Comm. of the ACM*, 55(3), 2012.
- [13] Peach Fuzzer. <https://www.peach.tech/>.
- [14] OpenRCE: sulley. <https://github.com/OpenRCE/sulley>.
- [15] Radamsa. <https://gitlab.com/akihe/radamsa>.
- [16] ZZUF - MULTI-PURPOSE FUZZER. <http://caca.zoy.org/wiki/zzuf>.
- [17] H. Peng, Y. Shoshitaishvili, and M. Payer. T-fuzz: fuzzing by program transformation. In *IEEE Symposium on Security and Privacy*, 2018.
- [18] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, 2016.
- [19] A. Shukla, S. J. Saidi, S. Stefan, M. Canini, T. Zinner, and A. Feldmann. Towards Consistent SDNs: A Case for Network State Fuzzing. In *IEEE Transactions on Network and Service Management*, 2019.
- [20] A. Shukla, K. N. Hudemann, A. Hecker, and S. Schmid. Runtime Verification of P4 Switches with Reinforcement Learning. In *ACM SIGCOMM NetAI*, 2019.
- [21] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. MIT press, 2018.
- [22] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, 3rd edition, 2009.
- [23] J. A. Jones, M. J. Harrold, and J. T. Stasko. Visualization for Fault Localization. In *ACM/IEEE ICSE Workshops*, 2001.
- [24] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *ACM/IEEE ICSE*, 2002.
- [25] J. A. Jones and Mary J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *IEEE/ACM ASE*, 2005.
- [26] C. Le Goues, M. Pradel, and A. Roychoudhury. Automated program repair. In *Comm. of the ACM*, 2019.
- [27] Cisco Systems. daPIPE: DATA Plane Incremental Programming Environment. https://p4.org/assets/P4WS_2019/p4workshop19-final16.pdf, 2019.
- [28] switch.p4. <https://github.com/p4lang/switch>.
- [29] P4 Tutorial. <https://github.com/p4lang/tutorials>.
- [30] NetPaxos. <https://github.com/usi-systems/p4xos-public>.
- [31] P4.org. Behavioral model repository. <https://github.com/p4lang/behavioral-model>, October 2015. Accessed: 2018-12.

- [32] Tofino. <https://www.barefootnetworks.com/products/brief-tofino>.
- [33] P. Bosshart, G. Gibb, H. S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN. *ACM CCR*, 43(4), 2013.
- [34] P416 Portable Switch Architecture (PSA)- Version 1.1: Programmable Blocks. <https://p4.org/p4-spec/docs/PSA-v1.1.0.html#sec-programmable-blocks>.
- [35] P416 Portable Switch Architecture (PSA)- Version 1.1: Ingress Deparser and Egress parser. <https://p4.org/p4-spec/docs/PSA-v1.1.0.html#appendix-rationale-ingress-deparser-egress-parser>.
- [36] P4 Language Consortium. the p4 language specifications, version 1.0.5.
- [37] P4 Language Community. P4c, 2019.
- [38] S. Moon, J. Helt, Y. Yuan, Y. Bieri, S. Banerjee, V. Sekar, W. Wu, M. Yannakakis, and Y. Zhang. Alembic: automated model inference for stateful network functions. In *NSDI*, 2019.
- [39] H. van Hasselt, A. Guez, and D. Silver. Deep Reinforcement Learning with Double Q-Learning. In *AAAI*, 2016.
- [40] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.
- [41] L.-J. Lin. Reinforcement learning for robots using neural networks. Technical report, CMU School of Computer Science, 1993.
- [42] T. Schaul, J. Quan, I. Antonoglou, and D. Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.
- [43] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1. chapter Learning Internal Representations by Error Propagation. 1986.
- [44] Keras: The Python Deep Learning library. <https://keras.io/>.
- [45] TensorFlow. <https://www.tensorflow.org/>.
- [46] Scapy. <https://scapy.net/>.
- [47] P4Runtime. <https://p4.org/p4-runtime/>.
- [48] Vagrant. <https://www.vagrantup.com/>.
- [49] VirtualBox. <https://www.virtualbox.org/>.
- [50] P. Kazemian, G. Varghese, and N. McKeown. Header Space Analysis: Static Checking for Networks. In *NSDI*, 2012.
- [51] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real Time Network Policy Checking Using Header Space Analysis. In *NSDI*, 2013.
- [52] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. VeriFlow: Verifying Network-Wide Invariants in Real Time. In *NSDI*, 2013.
- [53] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown. Automatic test packet generation. In *ACM CoNEXT*, 2012.
- [54] M. Rajpal, W. Blum, and R. Singh. Not all bytes are equal: Neural byte sieve for fuzzing. *arXiv preprint arXiv:1711.04596*, 2017.
- [55] P. Godefroid, H. Peleg, and R. Singh. Learn&fuzz: Machine learning for input fuzzing. In *IEEE/ACM Automated Software Engineering*, 2017.
- [56] C. Cummins, P. Petoumenos, A. Murray, and H. Leather. Compiler Fuzzing Through Deep Learning. In *ACM Softw. Testing & Anal.*, 2018.
- [57] N. Lopes, N. Bjørner, N. McKeown, A. Rybalchenko, D. Talayco, and G. Varghese. Automatically verifying reachability and well-formedness in P4 Networks. *Microsoft Technical Report, Tech. Rep*, 2016.
- [58] L. Ryzhyk, N. Bjørner, M. Canini, J. Jeannin, C. Schlesinger, D. B. Terry, and G. Varghese. Correct by construction networks using stepwise refinement. In *NSDI*, 2017.
- [59] S. Kodeswaran, M. Arashloo, P. Tammana, and J. Rexford. Tracking p4 program execution in the data plane. In *SOSR*, 2020.
- [60] C. Kim, A. Sivaraman, N. Katta, A. Bas, A. Dixit, and L. J. Wobker. In-band network telemetry via programmable dataplanes. In *ACM SIGCOMM*, 2015.
- [61] Int specification. <https://github.com/p4lang/p4-applications/blob/master/docs>.
- [62] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Automata theory, languages, and computation*, volume 24. Pearson, 2006.
- [63] I. J. Good. The population frequencies of species and the estimation of population parameters. *Biometrika*, 40(3-4):237–264, 1953.

- [64] W. A. Gale and G. Sampson. Good-turing frequency estimation without tears. *Journal of quantitative linguistics*, 2(3):217–237, 1995.
- [65] M. Böhme. Assurance in software testing: a roadmap. In *International Conference on Software Engineering*, 2019.
- [66] K. P. Burnham, D. R. Anderson, and K. P. Huyvaert. AIC model selection and multimodel inference in behavioral ecology: some background, observations, and comparisons. *Behavioral ecology and sociobiology*, 65(1), 2011.
- [67] Nick Feamster and Jennifer Rexford. Why (and how) networks should run themselves. In *arXiv preprint arXiv:1710.11583*, 2017.