Fix with P6: Verifying Programmable Switches at Runtime

Apoorv Shukla[1,*] Kevin Hudemann[2,*] Zsolt Vági[3,*] Lily Hügerich[4]
Georgios Smaragdakis[4] Stefan Schmid[5] Artur Hecker[1] Anja Feldmann[6]

[1]Huawei Munich Research Center [2]SAP [3]Swisscom [4]TU Berlin [5]Faculty of Computer Science, University of Vienna [6]MPI-Informatics

***Abstract*—We design, develop, and evaluate P6, an automated approach to *(a)* detect, *(b)* localize, and *(c)* patch software bugs in P4 programs. Bugs are reported via a violation of pre-specified expected behavior that is captured by P6. P6 is based on machine learning-guided fuzzing that tests P4 switch non-intrusively, i.e., without modifying the P4 program for detecting runtime bugs. This enables an automated and real-time localization and patching of bugs. We used a P6 prototype to detect and patch existing bugs in various 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 large P4 programs such as switch.p4 without triggering any regressions.**

I. Introduction

Programmable networks herald a paradigm shift in the design and operation of networks. 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 [1], [2], 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 P4 switch.

However, with these new capabilities, also new challenges are unleashed, related to the P4 software verification, i.e., ensuring that the software fully satisfies all the expected requirements. 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 examine and verify the behavior of P4 switches after the P4 programs are deployed. We pose the question: *"Is it possible to* `detect`*,* `localize`*, and* `patch` *software bugs in a P4 program running on P4 switches?"*. We believe that being able to answer this question, even partially, unlocks full potential of programmable networks, improves their security, and will hence increase their penetration in operational and mission-critical networks.

Recently, a panoply of P4 program verification tools [4]–[10] 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 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 or ECMP (Equal-Cost Multi-path) hash calculations-related bugs are platform-dependent or P4 target switch implementation-specific and, thus, cannot be detected by static analysis approaches [4]–[7] or others [11]. Since, runtime verification aims to verify the *actual* behavior against the *expected* behavior of a switch by sending specially-crafted input packets to the switch and observing the behavior, such verification is complementary to static analysis. Currently, the development and testing cycles in P4-based systems are short [12] due to intense competition and need for new applications which makes runtime verification indispensable. Note; this makes the detection of bugs causing the abnormal runtime behavior a challenging task as the P4 switch does not throw any runtime exceptions. Furthermore, the detection of bugs is also challenging if there is no output, i.e., packets are dropped silently instead of being forwarded. Thus, runtime verification of switch is crucial.

A useful approach to verify the runtime behavior is fuzz testing or fuzzing [13]–[23], a well-known dynamic program 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., even a 32-bit destination IPv4 address field in a packet header introduces 2^{32} possibilities. 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. [23] have shown that Reinforcement Learning (RL) [24], [25] can be used to train a verification system. We build upon [23] 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

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

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 pre-defined test cases; this makes it difficult to pinpoint the actual faulty line/s of code. Tarantula [26]–[28] 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 also show how the detection and localization of bug makes it possible to patch a number of bugs in P4 programs.

P6. In P4, the automated program repair [29] is an uncharted territory and becomes increasingly important as the software development lifecycle in programmable networks is short [12] with insufficient testing. In this paper, we show that due to the structure of P4 programs, it is possible to automate patching of platform-independent bugs (P4 program-specific software bugs) in P4 programs, if the patch is available. To this end, we present P6, *P4* with runtime *P*rogram *P*atching, a novel runtime P4-switch verification system that (a) detects, (b) localizes, and (c) patches software bugs in a P4 program. P6 improves existing work on machine learning-guided fuzzing [23] 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.

In a nutshell, in P6, the first step is to capture the expected behavior of a P4 switch, which is achieved using information from three different sources: (i) the control plane configuration, (ii) queries in p4q (§III-B1), 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 fix 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 $P4_{16}$ application programs from switch.p4 [30], P4 tutorial solutions [31], and NetPaxos codebase [32] across two P4 switch platforms, namely, behavioral model version 2 (bmv2) [33] and Tofino [34]. Our results show that P6 successfully detects, localizes, and patches diverse bugs in all $P4_{16}$ programs while significantly outperforming bug detection baselines without introducing any regressions.

Contributions. Our main contributions are:

- We design, implement, and evaluate P6, an end-to-end runtime P4 verification system that detects, localizes, and patches bugs in P4 programs non-intrusively. (§III)
- We observe that the success of P6 relies on the increased patchability of P4 program from old ($P4_{14}$) to the new version ($P4_{16}$). (§II)
- We present a P6 prototype and report on an evaluation study. We evaluate our P6 prototype on a P4 switch running eight $P4_{16}$ programs (including switch.p4 with $8,715$ LOC) from publicly available sources [30]–[32] 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. (§IV)
- For platform-independent bugs, P6 localizes bugs and fixes the P4 program, when a patch is available, without causing regressions/introducing new bugs. (§III, §IV)
- We will release the P6 software and library of ready patches for all existing bugs in the P4 programs.

II. Challenges & Opportunities

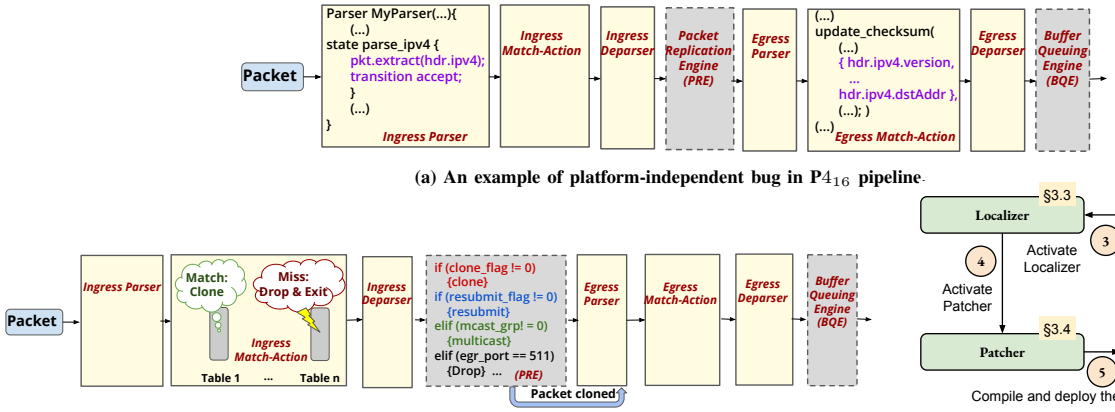
A. Primer: Packet Processing Pipeline of P4

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 [35] to its current form $P4_{16}$ [2] in generic Portable Switch Architecture (PSA) [36] switch platform, e.g., Tofino [34] (Figure 1a and 1b). In $P4_{16}$ 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 1a and 1b. There are also two platform-dependent blocks (annotated with dashed lines in Figures 1a and 1b): the packet replication engine (PRE) and the buffer queuing engine (BQE). These are non-programmable relying on proprietary implementations of hardware vendors.

B. 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 we term the bug as 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 if the implementation is hardware-related or 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.

For *platform-independent bugs*, we consider the Figure 1a (solid line blocks). It illustrates part of the implementation of Layer-3 (L3) switch provided in the P4 tutorial solutions [31]. Here, the parser does not check if the IPv4 header



(a) An example of platform-independent bug in $P4_{16}$ pipeline.

(b) An example of platform-dependent bug in $P4_{16}$ pipeline.

(c) `P6` Workflow. Modules of `P6` (in solid green boxes).

Fig. 1: Fig. 1a and Fig. 1b illustrate platform-independent and -dependent bugs respectively. Fig. 1c depicts `P6` Workflow.

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 leading to anomalies in network behavior. Other bugs that fall in this category are those related to IPv4/6 `checksum` and `ttl` in the packet. Such bugs are *platform-independent*, as they only result from programming errors.

For a *platform-dependent bug*, consider the scenario shown in Figure 1b (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 1b. It would drop the original packet, however, forward the cloned copy of the packet. Similar bugs can occur, if instead of the clone action, resubmit action is chosen (blue) or 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*.

C. Opportunities for Patching: Structure of a P4 Program

In the evolution of P4, there are two recent versions: $P4_{14}$ [37] and $P4_{16}$ [2]. $P4_{16}$ allows programmers to use definitions of a target switch-specific architecture, e.g., PSA (Portable Switch Architecture) [36], [38]. $P4_{16}$ is an upgraded version of $P4_{14}$. In particular, a large number of language features have been eliminated from $P4_{14}$ and moved into libraries including counters, checksum units, meters, etc., in $P4_{16}$. $P4_{14}$ 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 $P4_{16}$ allows to explicitly program six programmable blocks (Figure 1a).

By analyzing programs in the $P4_{14}$ and $P4_{16}$ 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. We observe that the potentially patchable code percentage increases from $P4_{14}$ to $P4_{16}$ in all applications (excluding calculator) from P4 tutorial solutions [31] and NetPaxos codebase [32] in behavioral model (bmv2) switch platform [33] and other generic PSA switch platforms [36], [38], e.g., Tofino [34] respectively. The patchable code percentage comes from the six programmable blocks in $P4_{16}$. Roughly, whatever is programmable, is patchable. In principle, around 40-45% of a P4 program is patchable in $P4_{16}$ programs for behavioral model (bmv2) switch platform [33]. This increases to 50-55% if the ingress deparser and egress parser are programmable for other target switch platforms, e.g., Tofino [34]. In particular, the parser and header definitions account for 20-40% of the total patchable code. If there is no bug in parser or header, packets with incorrect header get dropped. However, the bug still can be either in the non-patchable platform-dependent block or in the application code logic or deparser which is patchable as it is platform-independent.

Observation: *From* $P4_{14}$ *to* $P4_{16}$, P4 *program possesses twice as many programmable blocks increasing the chances for patchability. Bugs detected in the platform-independent part can be localized and patched; a platform-dependent*

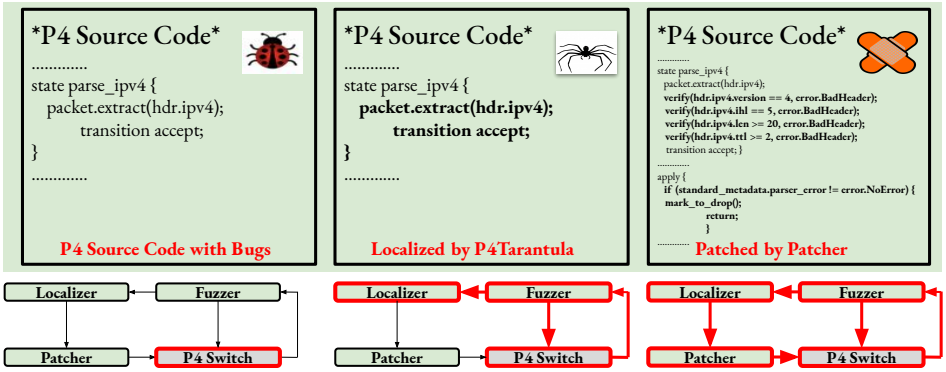


Fig. 2: P6 in Action: depicting the automated detection, localization and patching of a bug in a L3 switch P4 program [31].

bug may not be patchable if it is hardware-related.

III. P6: System Design

A. P6: Overview

P6's goal (see Figure 1c) is to detect, localize, and patch the software bugs in a P4 program at runtime. 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 (§III-B1) to the P4 switch running the pre-compiled P4 program. (§III-B)
(2) Localizer: P4Tarantula is the Localizer which pinpoints faulty lines of code causing bugs in the P4 program. (§III-C)
(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. (§III-D)
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* (§III-B). As shown in Figure 1c, 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 during the training process by selecting better mutation actions on the packet (2). After the bug detection, the Fuzzer automatically triggers Localizer (§III-C), 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 (§III-D) 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 2 illustrates how P6 detects, localizes, and patches an existing bug in a layer-3 (L3) switch P4 source code (program) from [31] in an automated fashion. The left part of Figure 2 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 2) 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 [39] 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.

B. Fuzzer: RL-guided Fuzzing

The goal of Fuzzer is to detect the runtime bugs discussed in §II-B. We improve [23] 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 [40], which can be augmented to static analysis. We guide the mutation-based white-box fuzzing [15] via RL [24], [25]. 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 (§III-B1).

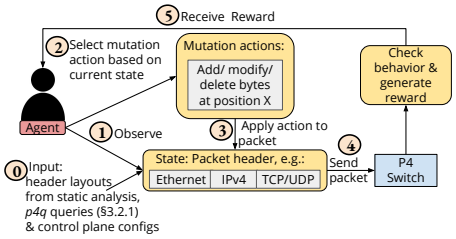


Fig. 3: Fuzzer. *Reward System* (in yellow) and *Agent* (in pink).

p4q queries are conditional queries (if-then-else) where each query has multiple conditions and each condition acts as a test case. A test case violation represents a bug detection.

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* immediately receives 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 it 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 (§III-B1). 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 3 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 §III-B2). The observed state is the input for the neural networks of the *Agent* (§III-B2), which outputs the appropriate mutation action. The selected action is applied for the given packet, and the packet is sent to the 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

```
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
15 (ing.hdr.ipv4,                                         Query 6
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
22 (ing.hdr.ipv4,                                         Query 7
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
29 (ing.hdr.ipv4,                                         Query 8
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()}, )
```

Fig. 4: p4q Queries. Queries 1-6 represent platform-independent, and Query 7-8 represent platform-dependent queries respectively.

violated. Then, *Agent* (§III-B2) uses the reward to improve action selection in subsequent executions (exploitation).

1) p4q: Query Language: Before diving deep in the *Agent* training, we explain the query language, p4q [23], written by the tester or programmer and used for specifying the *expected* switch behavior. To achieve the goal of an automated runtime verification system, the 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 [23], 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 branch is not fulfilled at egress. To define these conditions, the p4q syntax and grammar are used.

Figure 4 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. 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, the p4q queries are pre-specified as platform-independent or -dependent.*

2) Agent: The *Agent* houses the RL algorithm, which is inspired by Double Deep Q Network (Double DQN) [41], an improved version of Deep Q Networks (DQN) [42] and based on Q-learning [25], hence a *model-free* RL algorithm. *Model-free* means that 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 making DDQN an apt 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. Furthermore, to counter the scenario of sparse rewards, a simple form of prioritized experience replay, inspired by [43], is applied. The memory is sorted by absolute reward and each experience is prioritized by a configurable factor and the index.

For our Agent algorithm, we leverage Multi-Layer Perceptron (MLP) [44]. Hereby, we apply an ϵ-greedy policy, i.e., during training of the *Agent*, an action is selected randomly by the *Agent* with probability ϵ to ensure sufficient exploration. Once trained on a P4 program, a trained *Agent* can be used on other P4 programs to find bugs.

C. 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 [26], [28]. 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 in the P4 program.* 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 (if p4q query is not violated) or fail *verdict* (if p4q query is violated) corresponding to those sent packets. Recall, a *verdict* corresponds to a condition of the p4q query which acts as a test case.

Algorithm 1: 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] = SC[j].fail/totalFailed / (SC[j].pass/totalPassed + SC[j].fail/totalFailed)
16     end
17 end
18 call Patcher
```

Algorithm 1 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 of the P4 source code 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* [28] 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 of a line of code 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 program lines are ordered as per their suspiciousness score to localize the bug. Then, Patcher is notified.

D. 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.

Algorithm 2: 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* ≥ 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

From the results of the static analysis, Patcher extracts the required 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 [30]–[32]) 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 code analysis.

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 automatically 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.

Algorithm 2 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 0.5 (threshold is configurable as per deployment scenario) (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).

IV. P6 Prototype & Evaluation

We developed a P6 prototype using Python version 3.6 with $\approx 3,100$ lines of code (LOC); Fuzzer with $\approx 2,200$ LOC, P4Tarantula with $\approx$ 490 LOC and Patcher with $\approx$ 430 LOC. Fuzzer is implemented using Keras [45] library with Tensorflow [46] backend and Scapy [47] for packet generation and monitoring. The *Agent* was trained separately, for each condition of each query written in p4q.

A. Baselines

We compare P6 against three baseline fuzzers:
(1) Advanced Agent. It represents the intelligent baseline as it only relies 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.
(2) IPv4-based fuzzer. It 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 packets from being dropped by the P4 switch forwarding rules.
(3) Naïve fuzzer. It is not aware of any packet header layouts. It generates and sends Ethernet frames from purely random mutation of bytes.

B. Bugs

Table I provides an overview of *existing* bug types (with bug IDs) detected in the publicly available P4 programs from [30]–[32] by the P6 prototype. These bugs are detected as they violate the corresponding p4q queries (from Figure 4). 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 $8-10$).

C. 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]. 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

Bug IDs	Bugs	Queries (Figure 4)
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 I: Bugs (with Bug IDs) detected by the P6 prototype through the violation of the corresponding p4q queries (in Figure 4). Note, PI/PD refer to platform-independent and -dependent.

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 $P4_{16}$ programs [30]–[32].

D. Metrics

In particular, we ask the following questions:

Q1. *How much time does* P6 *take to detect, localize, and patch all bugs?* (§IV-D1)

Q2. *How does* P6 *perform against the baselines?* (§IV-D2)

Q3. *How many rewards does* P6 *generate against the baseline of an Advanced Agent for Agent training?* (§IV-D3)

Q4. *How many packets does* P6 *generate to detect bugs against the baselines?* (§IV-D4)

1) Performance of P6: To evaluate P6 performance, we execute the detection, localization, and patching on 8 publicly available P4 programs from the P4 tutorials [31], NetPaxos [32] and switch.p4 [30] repository.

Figure 5a and 5d 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 increased bug detection time in the NetPaxos programs [32] can be due to the required instrumentation to make them run on CPU intensive Tofino Model.

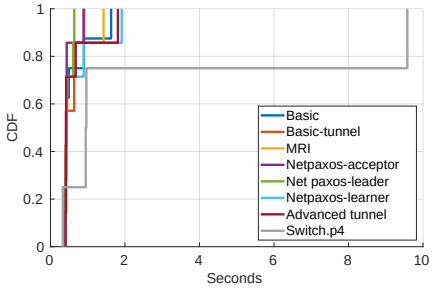
Figures 5b and 5e illustrate the median bug localization time of P6 for the different programs using bmv2 SimpleSwitchGrpc and Barefoot Tofino Model. Overall, all bugs for 7 of the programs were localized by P6 in just above 0.12 seconds on bmv2 and Tofino. To our surprise, the bug localization time for switch.p4 program running on bmv2 is only increased by a factor of 4×, even though the program has about 30× more lines of code compared to the other tested programs. Figures 5c and 5f illustrate the median time of patching code for bmv2 and Tofino respectively. P6 is able to patch the P4 programs with millisecond scale performance (max. 98 ms).

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 6a (with quartiles), was able to detect all the bugs present in the tested programs, which is due to the similarity with the P6 *Agent* but it cannot learn from the rewards, hence generates more packets and is slower 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 [31], [32]. For switch.p4 program [30], IPv4-based fuzzer was able to detect 3 out of 4 bugs which were IPv4-based. In Figure 6b (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, Naïve fuzzer was not able to detect any bugs, even with 16k packets.*

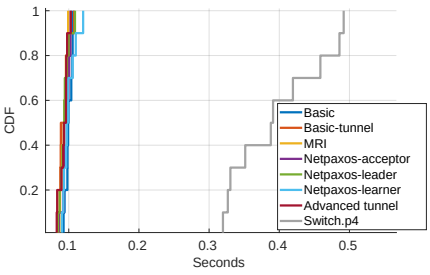
Figure 6a shows speedup (Advanced Agent/P6 *Agent*) for all bugs detected in the seven tested programs from [31], [32]. The results show that P6 *Agent* can detect bugs up to 10.96× faster than the Advanced Agent baseline. Only bug 7 was detected faster by the Advanced Agent in 3 of the 7 $P4_{16}$ applications 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 requires more time and packets to detect the bugs than the P6 *Agent*. For switch.p4 program [30], P6 *Agent* detects bugs 30× faster than the Advanced Agent.

Figure 6b shows the speedup (IPv4-based fuzzer/P6 *Agent*) for all bugs detected in the 7 tested programs from [31], [32]. For test-cases where IPv4-based fuzzer detects the bugs, we observe that in 89% of the test-cases P6 *Agent* detects the bugs faster while sending significantly fewer packets. P6 *Agent* outperforms IPv4-based fuzzer by up to 8.88× even when IPv4-based fuzzer sends packets at a higher rate. For switch.p4 program, P6 *Agent* detects bugs up to 30× faster than IPv4-based fuzzer.

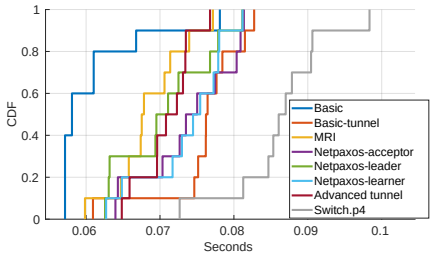
3) P6 vs Advanced Agent Training: To verify that P6 *Agent* is able to effectively learn to detect bugs, we compare P6 *Agent* against an Advanced Agent, that only relies on random action selection. Advanced Agent is just not as intelligent as P6 *Agent*. Advanced Agent can still execute same mutation actions but is not able to reason about which actions lead to maximized rewards. Figure 6c shows a comparison of the *mean cumulative reward* (MCR) of the training process of both agents for bug ID 4 of Table I. We observe that the P6 *Agent* is able to outperform the baseline by a factor of 3.56× 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 shown in Figure 4 using same set of hyper-parameters. Overall, P6 *Agent* outperforms baselines.



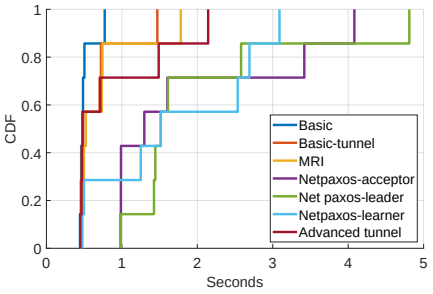
(a) Median of bug detection times (bmv2).



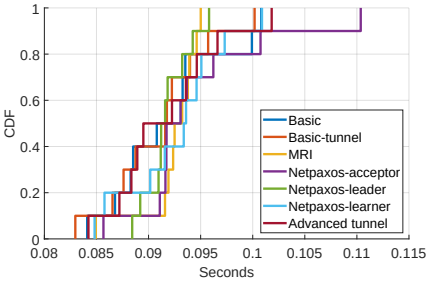
(b) Median of bug localization times (bmv2).



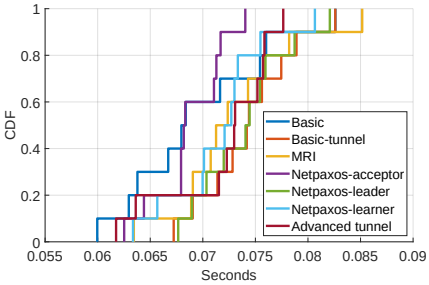
(c) Median time of patching the code (bmv2).



(d) Median of bug detection times (Tofino).

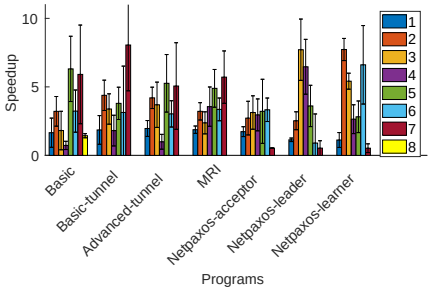


(e) Median of bug localization times (Tofino).

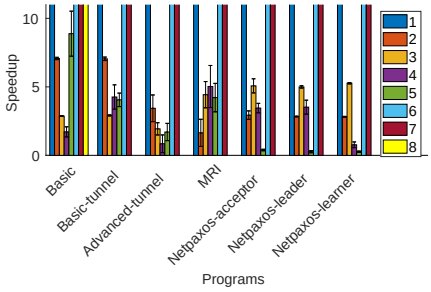


(f) Median time of patching the code (Tofino).

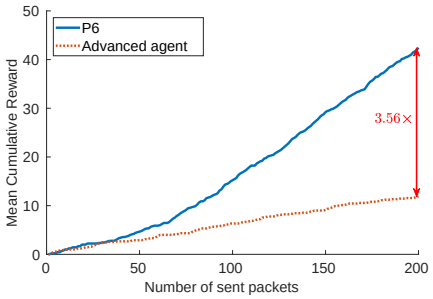
Fig. 5: Bug detection, localization and patching times of P4 programs in bmv2 and Tofino. Each plot represents a median over 10 runs.



(a) Speedup of median bug detection times: P6 vs Advanced Agent. Bug IDs (Table I) in the legend.



(b) Speedup of median bug detection times: P6 vs IPv4-based fuzzer. Bug IDs (Table I) in the legend.



(c) Training: P6 vs Advanced Agent (random action selection) in the case of bug 4 (Table I).

Fig. 6: P6 vs Baselines. Each plot represents a median over 10 runs.

$P4_{16}$ Applications	P6	Advanced Agent	IPv4-based	Naïve
basic.p4 [31]	13	59	8, 035	16, 000
basic_tunnel.p4 [31]	11	59	6, 044	14, 000
advanced_tunnel.p4 [31]	12	57	6, 038	14, 000
mri.p4 [31]	10	61	6, 058	14, 000
netpaxos-acceptor.p4 [32]	11	52	6, 021	14, 000
netpaxos-leader.p4 [32]	9	49	6, 024	14, 000
netpaxos-learner.p4 [32]	12	44	6, 026	14, 000
switch.p4 [30]	28	113	2, 132	14, 000

TABLE II: P6 vs Baselines. Median #packets sent over 10 runs.

Related work in P4	Runtime Verification	Detection	Localization	Patching	Detection of PD bugs
Cocoon [50]	✗	✓	✓	✗	✗
Vera [4]	✗	✓	✗	✗	✗
p4v [5]	✗	✓	✗	✗	✗
ASSERT-P4 [6], [7]	✗	✓	✗	✗	✗
P4NOD [51]	✗	✓	✗	✗	✗
p4pktgen [8]	✗	✓	✗	✗	✗
P4CONSIST [9]	✓	✓	✗	✗	✗
P4RL [23]	✓	✓	✗	✗	✗
P6	✓	✓	✓	✓	✓

TABLE III: Related work in P4 verification. PD corresponds to the platform-dependent bugs. Note, ✓denotes the capability, (✓) denotes a part of full capability, and × denotes the missing capability.

4) P6 vs Baselines: Dataplane Overhead: Table II 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 could not trigger any bug.

Related Work. Table III illustrates capabilities of other P4 verification tools as compared to P6.

V. Conclusion

We presented P6, a system that enables runtime verification of P4 switches in a non-intrusive fashion. We believe P6 is an important foray into self-driving networks [52], which come with stringent requirements on dependability and automation. As a part of our future agenda, we plan to apply P6 on commercial-grade P4 programs and networks to report on our experience.

Acknowledgement. We thank Lalith Suresh, Bhargava Shastry, and our anonymous reviewers for their helpful feedback. This work and its dissemination efforts were conducted as a part of Verify project supported by the German Bundesministerium für Bildung und Forschung (BMBF) Software Campus grant 01IS17052, the European Research Council (ERC) Starting Grant ResolutioNet (ERC-StG-679158), and the Vienna Science and Technology Fund project ICT19-045, 2020-2024.

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. $P4_{16}$ 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] A. Shukla. *Towards runtime verification of programmable networks*. Doctoral thesis, Technische Universität Berlin, Berlin, 2020.

[11] S. Kodeswaran, M. Arashloo, P. Tammana, and J. Rexford. Tracking p4 program execution in the data plane. In *SOSR*, 2020.

[12] Cisco Systems. daPIPE: DAta Plane Incremental Programming Environment. https://p4.org/assets/$P4WS_2019/p4workshop19-final16.pdf$, 2019.

[13] M. Zalewski. American Fuzzy Lop: A Security-oriented Fuzzer. http://lcamtuf. coredump. cx/afl/, (visited on 6/4/2019), 2010.

[14] Llvm Compiler Infrastructure: libfuzzer: a library for coverage-guided fuzz testing. http://llvm.org/docs/LibFuzzer.html.

[15] P. Godefroid, M. Y. Levin, and D. Molnar. SAGE: whitebox fuzzing for security testing. *Comm. of the ACM*, 55(3), 2012.

[16] Peach Fuzzer. https://www.peach.tech/.

[17] OpenRCE: sulley. https://github.com/OpenRCE/sulley.

[18] Radamsa. https://gitlab.com/akihe/radamsa.

[19] ZZUF - MULTI-PURPOSE FUZZER. http://caca.zoy.org/wiki/zzuf.

[20] H. Peng, Y. Shoshitaishvili, and M. Payer. T-fuzz: fuzzing by program transformation. In *IEEE Symposium on Security and Privacy*, 2018.

[21] 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.

[22] 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.

[23] A. Shukla, K. N. Hudemann, A. Hecker, and S. Schmid. Runtime Verification of P4 Switches with Reinforcement Learning. In *ACM SIGCOMM NetAI*, 2019.

[24] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[25] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, 3rd edition, 2009.

[26] J. A. Jones, M. J. Harrold, and J. T. Stasko. Visualization for Fault Localization. In *ACM/IEEE ICSE Workshops*, 2001.

[27] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *ACM/IEEE ICSE*, 2002.

[28] J. A. Jones and Mary J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *IEEE/ACM ASE*, 2005.

[29] C. Le Goues, M. Pradel, and A. Roychoudhury. Automated program repair. In *Comm. of the ACM*, 2019.

[30] switch.p4. https://github.com/p4lang/switch.

[31] P4 Tutorial. https://github.com/p4lang/tutorials.

[32] NetPaxos. https://github.com/usi-systems/p4xos-public.

[33] P4.org. Behavioral model repository. https://github.com/p4lang/behavioral-model, October 2015. Accessed: 2018-12.

[34] Tofino. https://www.barefootnetworks.com/products/brief-tofino.

[35] 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.

[36] P416 Portable Switch Architecture (PSA)- Version 1.1: Programmable Blocks. https://p4.org/p4-spec/docs/PSA-v1.1.0.htmlsec-programmable-blocks.

[37] P4 Language Consortium. the p4 language specifications, version 1.0.5.

[38] P416 Portable Switch Architecture (PSA)- Version 1.1: Ingress Deparser and Egress parser. https://p4.org/p4-spec/docs/PSA-v1.1.0.htmlappendix-rationale-ingress-deparser-egress-parser.

[39] P4 Language Community. P4c, 2019.

[40] 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.

[41] H. van Hasselt, A. Guez, and D. Silver. Deep Reinforcement Learning with Double Q-Learning. In *AAAI*, 2016.

[42] 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.

[43] T. Schaul, J. Quan, I. Antonoglou, and D. Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

[44] 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.

[45] Keras: The Python Deep Learning library. https://keras.io/.

[46] TensorFlow. https://www.tensorflow.org/.

[47] Scapy. https://scapy.net/.

[48] Vagrant. https://www.vagrantup.com/.

[49] VirtualBox. https://www.virtualbox.org/.

[50] 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.

[51] 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.

[52] Nick Feamster and Jennifer Rexford. Why (and how) networks should run themselves. In *arXiv preprint arXiv:1710.11583*, 2017.