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 \texttt{switch.p4}. 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 \texttt{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
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 Program Patching, 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\textsubscript{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\textsubscript{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\textsubscript{14}) to the new version (P4\textsubscript{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\textsubscript{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 \cite{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 \cite{35} to its current form P4\textsubscript{16} \cite{2} in generic Portable Switch Architecture (PSA) \cite{36} switch platform, e.g., Tofino \cite{34} (Figure 1a and 1b). In P4\textsubscript{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 \cite{31}. 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 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...
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 consists of 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.

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).
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 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
ports 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 Per-
ceptron (MLP) [44]. Hereby, we apply an ε-greedy policy,
i.e., during training of the Agent, an action is selected ran-
domly 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 \( V[p] == \text{pass} \) then
4 \( \text{totalPassed} += 1 \)
5 else
6 \( \text{totalFailed} += 1 \)
7 end
8 end
9 foreach \( SC \) do
10 if \( SC[j].\text{pass} == 1 \) then
11 \( \text{totalPassed} += 1 \)
12 else
13 \( \text{totalFailed} += 1 \)
14 end
15 \( S[j] = \frac{SC[j].\text{fail}}{\text{totalFailed}} \)
16 end
17 call Patcher

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 Suspicousness score ≥ 0.5 then
4     check corresponding lines of code pinpointed by P4Tarantula
5     if the patch is missing and violating q then
6       apply the preferred patch
7       inform the programmer
8     end
9   else
10    Goto next line
11  end
12 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, 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 ones 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 the information, 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 file 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 ≈ 3,100 lines of code (LOC); Fuzzer with ≈ 2,200 LOC, P4Tarantula with ≈ 490 LOC and Patcher with ≈ 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. Naive 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
TABLE I: Bugs (with Bug IDs) detected by the P6 prototype through the violation of the corresponding p4q queries (in Figure 4).
Note, P1/PD refer to platform-independent and-dependent.

<table>
<thead>
<tr>
<th>Bug IDs</th>
<th>Bugs</th>
<th>Queries (Figure 4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Accepted wrong checksum (PI)</td>
<td>Query 1</td>
</tr>
<tr>
<td>2</td>
<td>Unrecognized frame (PI)</td>
<td>Query 2</td>
</tr>
<tr>
<td>3</td>
<td>TTL value of 0 or 1 is accepted (PI)</td>
<td>Query 3</td>
</tr>
<tr>
<td>4</td>
<td>IP IHL value out of bounds (PI)</td>
<td>Query 4</td>
</tr>
<tr>
<td>5</td>
<td>TTL not decremented (PI)</td>
<td>Query 5</td>
</tr>
<tr>
<td>6</td>
<td>Clone not dropped (PD)</td>
<td>Query 6</td>
</tr>
<tr>
<td>7</td>
<td>Multicast packet not dropped (PD)</td>
<td>Query 7</td>
</tr>
<tr>
<td>8</td>
<td>Resubmitted packet not dropped (PD)</td>
<td>Query 8</td>
</tr>
</tbody>
</table>

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 different programs using bm2 SimpleSwitchGrpc and Barefoot Tofino Model, respectively. Note, switch.p4 program is only available for bm2 and was not tested using Tofino. In all runs on bm2 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 different programs using bm2 SimpleSwitchGrpc and Barefoot Tofino Model. Overall, all bugs 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 4x, even though the program has about 30x more lines of code compared to the other tested programs. Figures 5c and 5f illustrate the median time of patching code for bm2 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 programs 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 8.8× 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.
TABLE II: P6 vs Baselines. Median #packets sent over 10 runs.

\begin{tabular}{|c|c|c|c|c|}
\hline
P4 Application & Baseline & Advanced Agent & IP4-based & Native \\
\hline
Basic & 15 & 59 & 8,933 & 16,000 \\
Basic-tunnel & 12 & 76 & 8,933 & 14,000 \\
MRI & 12 & 55 & 6,024 & 14,000 \\
NetPaxos-acceptor (p4) & 11 & 92 & 6,024 & 14,000 \\
NetPaxos-leader (p4) & 9 & 87 & 6,024 & 14,000 \\
Switch (p4) & 12 & 113 & 9,132 & 16,000 \\
\hline
\end{tabular}

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

\begin{tabular}{|c|c|c|c|c|c|c|}
\hline
Project & IPv4 Detection & IPv4 Localization & IPv4 Verification & IPv4 Patching & Platform-Dependent Bugs & Native \\
\hline
CoCo [50] & \(\checkmark\) & \(\checkmark\) & \(\checkmark\) & \(\checkmark\) & \(\checkmark\) & \\
Cocoon [50] & \(\checkmark\) & \(\checkmark\) & \(\checkmark\) & \(\checkmark\) & \(\checkmark\) & \\
Vera [4] & \(\checkmark\) & \(\checkmark\) & \(\checkmark\) & \(\checkmark\) & \(\checkmark\) & \\
Cocoon [50] & \(\checkmark\) & \(\checkmark\) & \(\checkmark\) & \(\checkmark\) & \(\checkmark\) & \\
Switch [50] & \(\checkmark\) & \(\checkmark\) & \(\checkmark\) & \(\checkmark\) & \(\checkmark\) & \\
\hline
\end{tabular}

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 ResolutionNet (ERC-StG-679158), and the Vienna Science and Technology Fund project ICT19-045, 2020-2024.
REFERENCES


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


