Apoorv Shukla, Kevin Nico Hudemann, Artur Hecker, Stefan Schmid

Runtime Verification of P4 Switches with Reinforcement Learning

Shukla, A., Hudemann, K. N., Hecker, A., & Schmid, S. (2019). Runtime Verification of P4 Switches with Reinforcement Learning. Proceedings of the 2019 Workshop on Network Meets AI & ML - NetAI'19. https://doi.org/10.1145/3341216.3342206

Runtime Verification of P4 Switches with Reinforcement Learning

Apoorv Shukla
TU Berlin

Kevin Nico Hudemann
TU Berlin

Artur Hecker
Huawei Technologies

Stefan Schmid
Faculty of Computer Science, University of Vienna

ABSTRACT

We present the design and early implementation of p4rl, a system that uses reinforcement learning-guided fuzz testing to execute the verification of P4 switches *automatically* at runtime. p4rl system uses our novel user-friendly query language, *p4q* to conveniently specify the intended properties in simple conditional statements (if-else) and check the actual runtime behavior of the P4 switch against such properties. In p4rl, user-specified *p4q* queries with the control plane configuration, *Agent*, and the *Reward System* guide the fuzzing process to trigger runtime bugs automatically during *Agent* training. To illustrate the strength of p4rl, we developed and evaluated an early prototype of p4rl system that executes runtime verification of a P4 network device, e.g., L3 (Layer-3) switch. Our initial results are promising and show that p4rl automatically detects diverse bugs while outperforming the baseline approach.

KEYWORDS

Network Verification; P4; Machine Learning; Fuzzing

1 INTRODUCTION

With the emergence of P4 [1] programmable data planes, it has become possible for the network operators to develop and deploy their customized and flexible packet processing programs to achieve fine-grained custom capabilities. P4 allows the programmers to define how a data plane device should process the packets and thus, break free from the vendor-specific expensive hardware and proprietary software. P4 allows programmers to define the multiple pipeline stages of packet processing: the packet parser, the packet processing (ingress and egress), and the deparser that dictate the packet processing behavior of the data plane switch.

With the programmability, however, the verification of the runtime network behavior has become increasingly complex. Runtime bugs or faults may cause serious network outages or security threats, which is why the network verification is critical. Current approaches focus mainly on the static analysis [2–5] of P4 programs. We, however, realize that static program analysis is insufficient when it comes to extensively and automatically verifying the runtime behavior of a P4 switch as P4 programs alone do not determine the forwarding behavior. Indeed, the actual forwarding rules provided by the control plane at runtime or statically when the P4 program is deployed as well as the switch-dependent components determine the network forwarding behavior. Thus, there is a dire need for runtime verification.

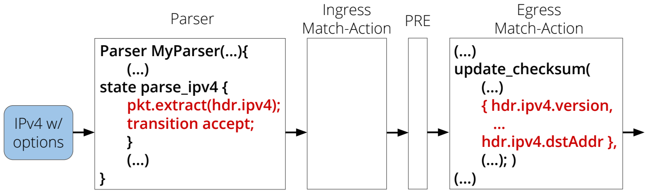
An interesting solution is fuzz testing or fuzzing [6, 7], a well-known dynamic program analysis technique that generates semi-valid, random inputs which trigger abnormal program behavior. However, in order for fuzzing to be efficient, intelligence needs to be added to input generation, to maximize the number of bugs found while providing minimal inputs. This becomes crucial especially in networking, where the input space is huge and complex, e.g., a 32-bit destination IPv4 address field in a packet header has 2^{32} possibilities and with the 5-tuple flow, the input space gets even more complex. In order to make fuzzing more effective, we consider the use of machine learning, to guide the fuzzer to generate smart inputs which trigger abnormal program behavior. In recent years, artificial intelligence and machine learning have gained attention to solve very complex problems, also in the area of networking [8, 9]. A variety of algorithms and approaches exist, which can be mainly categorized into supervised, semi-supervised, unsupervised and reinforcement learning [10]. In contrast to the other approaches, reinforcement learning aims at enabling an agent to learn how to interact with an environment, based on a series of reinforcements, meaning rewards or punishments received from the target environment. The agent observes the environment and chooses an action to be executed. After the action is executed, the agent receives a reward or punishment from the environment. While the goal of learning is to maximize the rewards, we argue it is equally crucial to design a machine learning model which is general enough for any kind of target environment.

This paper presents our novel approach for P4 switch verification, p4rl, a system that relies on the combination of fuzzing and reinforcement learning techniques to automatically and efficiently verify P4 switches at runtime. Using Double Deep Q Networks (DDQN) [11], we ensure that the evaluation of an action is independent of the selection of an action. Thus, we avoid the problem of overfitting for a given target environment. Furthermore, the prioritized experience replay [12] helps to avoid oscillations or divergence of parameters in the machine learning *Agent*. In addition, to specify the intended network behavior, e.g. P4 switch, we provide an easy-to-use query language, *p4q*, so that users can conveniently specify the *expected* packet processing behavior in the conditional if-else statements and verify such behavior against the *actual* behavior. *p4q* works in conjugation with p4rl.

The three main challenges in the design of p4rl are: *1)* careful selection of a suitable machine learning solution in a scenario, where provisioning of training data in desired quality and quantity is not feasible. *2)* designing a general machine learning model for any kind of target environment. *3)* dealing with the problem of smart input generation, especially crucial for the initial phase of the fuzzing process.

Our contributions in this paper are:

• We introduce a novel machine learning-guided fuzzing system,



```
Parser MyParser(...){
      (...)
state parse_ipv4 {
      pkt.extract(hdr.ipv4);
      transition accept;
      }
      (...)
}
```

```
(...)
update_checksum(
      (...)
      { hdr.ipv4.version,
        ...
      hdr.ipv4.dstAddr },
      (...); )
(...)
```

Figure 1: Example of a target device-independent bug.

P4RL that performs automatic runtime verification of P4 switches to detect diverse runtime bugs;

- We design a novel and user-friendly query language, *p4q*, for expressing the intended P4 switch behavior;
- We develop an early prototype of P4RL and evaluate it on a P4 network running real P4 application [13]. Our initial results show that P4RL can detect different bugs, while outperforming the baseline approach by around 4 times;
- P4RL software is publicly available at: https://gitlab.inet.tu-berlin.de/apoorv/P4ML.

2 MOTIVATING EXAMPLES

Software bugs or errors can occur at any stage in the P4 processing pipeline: parser, ingress match-action, packet replication engine (PRE), egress match-action and deparser. Note all stages except the PRE, are P4 programmable while PRE remains as a vendor specific, fixed function component.

There can be, however, bugs in the parser code, e.g., when not or incorrectly checking the header fields, such as IPv4 header length (`ihl`) or `TTL` field. As an example, consider the scenario in Figure 1 that illustrates part of the implementation of L3 (Layer-3) switch, provided in the P4 language tutorial solutions [13]. Here, the parser accepts any kind of IPv4 packets and does not check if the IPv4 header contains IPv4 options or not, i.e., if IPv4 `ihl` field is not or is equal to 5. 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 *incorrectly* forwarded instead of getting dropped. This leads to anomaly in the network behavior. Currently, the detection of such runtime bugs is non-trivial. We call such errors as *target device-independent bugs*, as these only result from programming errors involving one of the P4 programmable stages.

These are, however, not the only kind of bugs that can occur. The interaction of the P4 program with the `PRE` may result in an unintended behavior as well, which we call as *target device-dependent bugs*. The `PRE` is responsible for carrying out several forwarding actions, such as `clone`, `multicast`, `resubmit` or `drop`. The programmer uses standard metadata to communicate the forwarding action to the `PRE`, which interprets it and executes the actions accordingly. It is, however, very common to have situations where conflicting forwarding actions are selected. Consider a scenario where a P4 program implementing at least two tables, where one could be an IPv4 longest prefix match (`LPM`) table and a following table an access control list (`ACL`). If packets are matched by the `LPM` table and a clone decision is made, those packets later, get dropped by the `ACL` table. In such a case, the forwarding behavior depends on the implementation of the `PRE`, which is target device-dependent. The implementation of `PRE` of the simple switch target in the behavioral model (Bmv2) would drop the original packet, however, *incorrectly* forward the cloned copy of the packet. Similar runtime bugs can be seen, if instead of `clone`, `multicast` or `resubmit` actions are chosen.

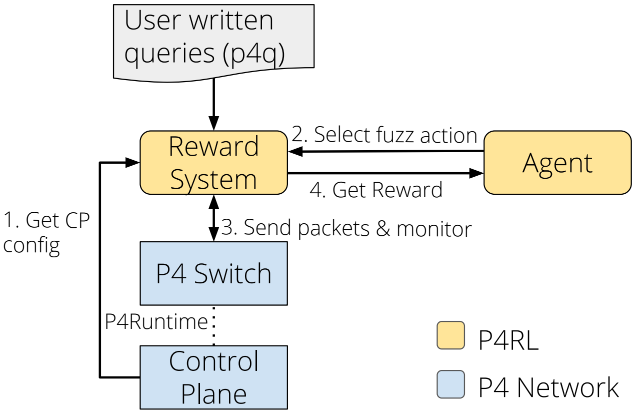


Figure 2: P4RL System Workflow.

Target device-independent or -dependent bugs are present in many real-world P4 applications. Currently, the aforementioned runtime bugs cannot be detected by the existing static analysis approaches [2–5].

3 P4RL: SYSTEM DESIGN

3.1 Overview

To address the limitations of current verification approaches illustrated by the previous example scenario (§2), we propose our novel verification system, P4RL (P4 Reinforcement Learning). Our approach is based on mutation-based fuzzing in combination with reinforcement learning techniques. We also provide a language, *p4q* for expressing the *expected* P4 switch behavior conveniently and check the *actual* runtime behavior of the P4 switch against such behavior.

Figure 2 illustrates an overview of the P4RL system. First, the user specifies the behavioral properties of the network to be verified. Together with the configuration of the control plane, it is the input for the *Reward System*, providing the basis for the verification. The reinforcement learning *Agent* defines the mutation actions to be applied for each individual packet to be generated. *Agent* adjusts its future action selection using the information returned by the *Reward System*, about the processing of packets done by the P4 switch/es.

We, now provide details on the brain of P4RL system: *Agent* (§3.2), *p4q* query language (§3.3), and P4RL workflow (§3.4).

3.2 Machine Learning-guided Fuzzing

In contrast to the static program analysis, dynamic program analysis can be used to test the forwarding behavior of a data plane device at runtime. Executing the program verification as a runtime task may lead to high costs in the dynamic network environment,

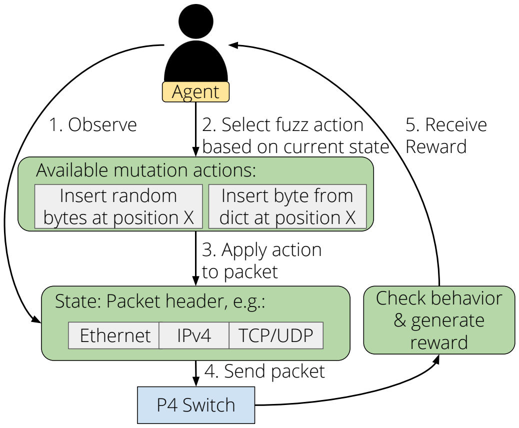


Figure 3: *Reward System* (green) and *Agent* (yellow) interactions.

if done naively. Fuzz testing or fuzzing [6, 7] is a popular dynamic testing approach, relying on generating or mutating inputs for the program under test. In the case of P4 programs or data plane devices in general, the number of possible inputs for the different header fields is huge and complex, e.g., just for one IPv4 destination address field there are 2^{32} possibilities, and with the 5-tuple flow, the input space gets even more complex. Hence, a solution for verifying P4 data plane devices, relying on fuzzing as dynamic testing technique needs to tackle these problems. The incorporation of feedback generated by the target system guides and adds intelligence to the fuzzing process. Feedback-driven fuzzing is also widely adopted, e.g., by *afl* [6], but current feedback-driven fuzzers lack the ability to reason about the relation of mutation actions and states. We, however, realized that awareness of the relationship between actions and states is highly important to reduce the number of inputs needed to trigger bugs, even though this might introduce higher complexity for the fuzzing process.

Using state of the art machine learning techniques, it is possible to create complex models and enable efficient reasoning about the connection of mutation actions and states. Relying solely on neural networks, or other common classification techniques, would require a lot of input data or training data. Together with the need for knowing the kind of packets that trigger bugs in the program, it does not prove to be a viable solution. In contrast, reinforcement learning approaches are widely adopted in the area of artificial intelligence, e.g., learning to play complex games like *Go*, making it apt for enabling intelligence to the fuzzing process.

P4RL Reinforcement Learning: Our novel methodology aims at overcoming the problems discussed by interpreting mutation-based fuzzing as reinforcement learning problem. Furthermore, we aim at designing a solution able to generalize for different target environments. Feedback is generated using the control plane configuration and queries defined with *p4q* (§3.3). The control plane configuration contains not only the forwarding table contents but also information about the P4 program generated by the compiler. Doing so enables the system to determine if the data plane device behaves as expected or a bug was triggered. To formalize mutation-based fuzzing as a reinforcement learning problem, states, actions and rewards are defined. *States:* The states are defined as the sequence of bytes forming the packet header. *Actions:* The set of actions is defined as the set of mutation actions for each individual header field. It can be either inserting random bytes or bytes from a pre-generated dictionary. *Reward:* The reward can be immediately received by the *Agent*, after the mutated packet was sent to the data plane device (switch) and the results of the execution are evaluated. In any scenario, it is likely to experience sparse rewards, so to send a lot of packets that will not trigger any bug. Accordingly, the reward is defined as 0 if the packet did not trigger a bug and 1 if the packet successfully triggered a bug to avoid divergence and oscillations during *Agent* training.

The pre-generated dictionary (dict) is generated using the control plane configuration and queries defined with *p4q*. The control plane configuration comprises the table contents, target-dependent configuration and the compiled P4 program in JSON format. From these, the packet header fields and layouts can be derived. Available boundary values for the header fields are extracted from the *p4q* queries (§3.3). Figure 3 illustrates the combination of reinforcement learning and mutation-based fuzzing. First, the *Agent* observes the current state of the environment, hence, the current packet header. The observed state is the input for the algorithm of the *Agent*, which outputs the fuzz action. The selected action is applied for the given packet, and the packet is sent to the P4 network. After the packet is processed, the behavior is evaluated and the reward is generated and returned to the *Agent*. The *Agent* uses the received reward to improve the action selection in subsequent executions.

3.2.1 Agent. The *Agent* houses our novel reinforcement learning algorithm (§3.2.2), which is inspired by Double Deep Q Network (Double DQN) [11], an improved version of Deep Q Networks (DQN) [14]. The key idea of the algorithm is to feed the current state of the environment to a neural network, to predict the action the *Agent* shall select to maximize future rewards. We used Double DQN algorithm [11] as it splits the selection of an action in a certain state, from the evaluation of that action. To realize this, it uses two neural networks. The online network executes the action selection and the target network evaluates that action. Doing so, significantly reduces the problem of overoptimism in action selection during learning. Overoptimism means, to overestimate the future rewards of certain actions. Accordingly, reducing overoptimism improves the learning process of the *Agent*, helps in avoiding overfitting and thus, help in creating a model able to generalize for different target environments. In order to apply the algorithm to our scenario, several customizations and improvements were necessary.

Experience replay [15] is a technique used to eliminate problems of oscillation or divergence of parameters, resulting from correlated data. Experiences of the *Agent*, hence, a tuple comprising the current state, predicted action, reward received and resulting state are saved in the memory of *Agent*. For learning by experience replay, random samples from past experiences are selected to update the neural network model. Problems arise in the case of sparse rewards, meaning when the *Agent* only receives rewards in a small number of trials. When randomly selecting experiences in such a case, most likely an experience that did not generate reward is returned. To overcome this, we apply a simple form of prioritized experience reply, inspired by [12]. In a nutshell, we sort the memory of the

Agent by absolute reward, and weight (prioritize) each experience by a configurable factor and the index.

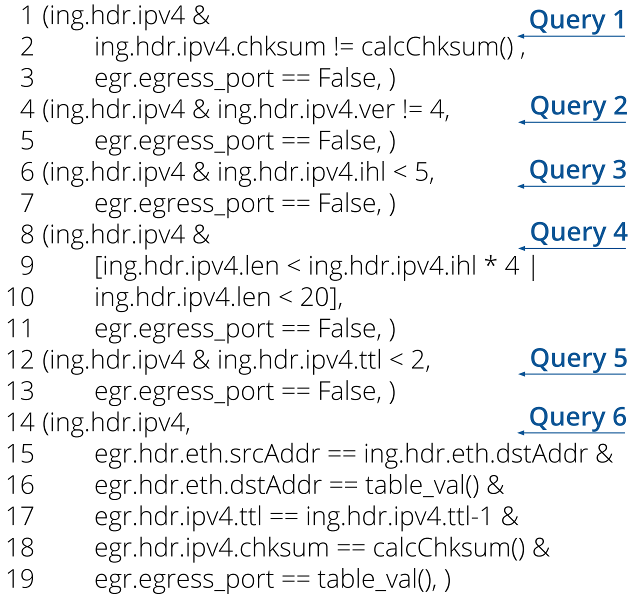
3.2.2 Agent Training Algorithm: Now, we will present the training algorithm of the *Agent* in P4RL system. As a first step, the weights of online and target networks are initialized. For each trial of the *Agent*, a packet header in byte representation is chosen from a pre-generated set of packet headers randomly. This byte sequence will then be converted to a series of float representations. To ensure diverse action selection, the *Agent* either selects the action for the current state randomly with a configurable probability or uses its online network to predict the action to be executed. The next step is to execute the selected action, observe and save the result in the experience memory. A sample is selected out of memory M to calculate a `value`, which is used to calculate the categorical cross entropy loss and perform the stochastic gradient descent step for updating the network weights.

3.3 Query Language: *p4q*

Together with the goal of automating the P4 network verification process, it is indispensable to provide a language to query the P4 network behavioral properties. To this end, we propose *p4q*, a query language to specify the *expected* network behavior and check it against the *actual* behavior. One of the major design goals of *p4q* was to provide a user-friendly interface for P4RL. To achieve this, the *p4q* syntax is kept simple. Each property to be checked is described as a tuple, in an `if-then-else` conditional statement. The user specifies the conditions to be fulfilled by the packet at switch ingress (`if`), together with conditions the packet should fulfill at egress (`then`). Optionally, the user can describe alternative conditions e.g., when the conditions in the "then" branch are not fulfilled at egress (`else`). Each of these conditions are defined by using the specified *p4q* syntax and grammar.

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

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 14-19). 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 out the correct port as instructed by the control plane.



```
 1 (ing.hdr.ipv4 &
 2        ing.hdr.ipv4.chksum != calcChksum() ,
 3        egr.egress_port == False, )
 4 (ing.hdr.ipv4 & ing.hdr.ipv4.ver != 4,
 5        egr.egress_port == False, )
 6 (ing.hdr.ipv4 & ing.hdr.ipv4.ihl < 5,
 7        egr.egress_port == False, )
 8 (ing.hdr.ipv4 &
 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,
13        egr.egress_port == False, )
14 (ing.hdr.ipv4,
15        egr.hdr.eth.srcAddr == ing.hdr.eth.dstAddr &
16        egr.hdr.eth.dstAddr == table_val() &
17        egr.hdr.ipv4.ttl == ing.hdr.ipv4.ttl-1 &
18        egr.hdr.ipv4.chksum == calcChksum() &
19        egr.egress_port == table_val(), )
```

Figure 4: *p4q* L3 (Layer 3) Switch Example.

3.4 P4RL Workflow

As illustrated in Figure 2, to initialize P4RL, the intended behavior of the P4 switch is described using *p4q*. The queries are imported by the *Reward System* to determine the boundary values for certain header fields and later on, enable the verification. If a query specifies to compare a header field with a specific value, the boundary values are slightly below and slightly above the specified value. Note boundary values have a higher probability to trigger bugs: they are valid enough to pass through the parser, but invalid enough to trigger problems further. In addition, the control plane configuration or the table contents as well as target-dependent configuration, e.g., clone or multicast, is used as well. The control plane configurations contain the compiled P4 program in JSON representation, so the *Reward System* can determine the supported header layouts.

With the available information, the *Reward System* generates two sets of packets. The first set comprises packets with mostly valid headers. It is used as a sample for the initial environment states (seeds). The second set contains packets with headers containing available boundary values. This set is the dictionary (dict) used by the *Agent* if action of inserting bytes from dict is chosen.

As soon as the initialization is done, the *Agent* observes the initial state of the environment. A packet header is randomly chosen from the set of initial environment states. The *Agent* uses this as input for its online network to predict an action to be selected, or an action is selected randomly, to ensure diverse action selection. The *Reward System* executes the action on behalf of the *Agent*, and sends the packet with the mutated packet header, to the ingress port of the P4 switch. The *Reward System* monitors the ports of the switch to capture the packet after it is processed. It uses the imported queries and a copy of the packet which was sent to the P4 switch to determine if the processing was as expected. In case a bug was triggered, the corresponding packet is saved to help the user find the source of the bug. Based on that information, *Reward System* generates the reward and returns it to the *Agent*. The reward is then used by the *Agent* to update its neural networks and the process is executed again. After a configurable timeout, if no bug is detected anymore, the process ends and the bug-triggering packets will be returned together with the violated properties to guide the programmer in localizing the potential faults, e.g., faulty header fields.

4 PROTOTYPE & EVALUATION

4.1 P4RL Prototype

We implemented a prototype of P4RL using Python version 3.6. The implementation of the *Agent* uses Keras [16] library with Tensorflow backend. The monitoring and packet generation is implemented using Scapy [17]. Currently, *Agent* is trained individually for every condition of each query. In addition, the individual training runs are executed sequentially, however, it could be parallelized. behavioral model (Bmv2) [18] with simple switch target, for P4$_{16}$ programs, is executed using Mininet. P4Runtime [19] implements the control plane module. For sending the control plane configurations to the *Reward System*, we provide a module to be imported and invoked by the control plane module. Currently, the P4 switch runs in a Virtual Box VM [20]. All experiments were conducted on an 8 core 1.80 GHz Intel Core I7 CPU machine, with 24 GB of RAM and running Ubuntu 18.04.2 LTS operating system. The P4 switch (P4$_{16}$, Bmv2 simple switch target) deployed in Mininet [21], the control plane component (P4Runtime), as well as the monitoring instances, run in a Virtual Box VM on the same machine. The *Reward System* and *Agent* are executed on the host machine natively. The Virtual Box VM runs Lubuntu (Light version of Ubuntu) 16.04.4 LTS operating system, using 2 Cores of the CPU and 2 GB RAM.

4.2 Evaluation

To evaluate P4RL, the verification of an L3 switch implementation in P4$_{16}$ [22] is executed, but the P4 network is initially limited to a single P4 switch only. We rely on the openly available L3 switch example, provided as part of the P4 language tutorial solutions [13]. For querying the P4 switch behavior, we implement the queries defined in Figure 4. Furthermore, we limit the maximum number of packets to be sent for each training run to 200. We execute the experiment 10 times, to account for its stochastic nature. Our metrics for evaluation are: *mean cumulative reward (MCR)* and *bug detection time*. Note P4RL *Agent* uses exactly the same set of hyperparameters and neural network architecture during the experiment, as we are aiming for a generalized model.

Baseline: We compare P4RL against the baseline of an *Agent* relying on random action selection. Similar but not as intelligent as P4RL *Agent*, i.e., it can still execute the same mutation actions without learning which actions lead to reward.

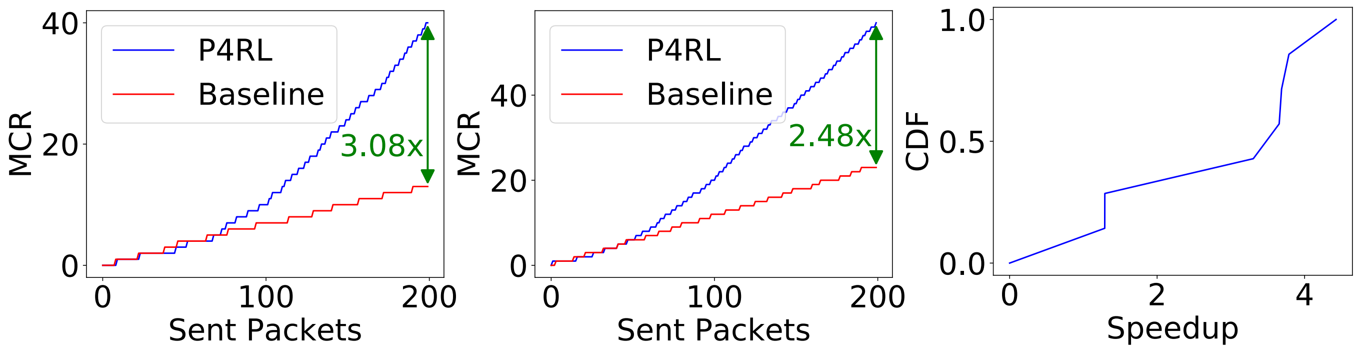
Figure 5a, 5b show the *mean cumulative reward (MCR)* of P4RL compared to the baseline, for Query 3 and 6 (line 15) in Figure 4. *In total, 7 distinct and target device-independent bugs were found by P4RL, violating the queries defined by Query 1-5, and the conditions defined in lines 17, 18 of Query 6.* Two of them, violating Query 1 and Query 6 (line 18), are checksum related bugs, current solutions could not have detected. Four bugs, violating the queries defined by Query 2-5, are related to wrong IPv4 header validation. Namely, not validated IPv4 version, TTL, header length and total length fields. The remaining bug is about the wrong IPv4 TTL decrement in case of $TTL = 0$, violating Query 6 (line 17). The baseline was also able to detect these bugs, due to the availability of the smart inputs. *Note the purely random packet generation approach was not able to trigger any bug*, given the number of packets was limited to 200 for each run, and the huge state space of IPv4 destination address field, i.e., 2^{32}. Therefore, we decided to omit the results as the cumulative reward remained 0 over all executions. Our results demonstrate that P4RL *Agent* is able to learn a strategy for triggering the runtime bugs. *Note the motivating example (§2), describing target device-dependent bugs involving clone, resubmit or multicast operations, cannot be detected by P4RL.*

Figure 5c shows the cumulative distribution function (CDF) for the speedup (Baseline/P4RL bug detection time) to quantify the gains of the intelligence of trained P4RL *Agent*. We observe that P4RL is up to 4.42× faster than the baseline. For about 57% of the time, P4RL was able to perform 3.3× faster than the baseline. For about 28%, P4RL only provided a speedup of 1.3×, which are the cases when no bug was present. Since, P4RL *Agent* chose to send packets with IPv4 destination address outside of the accepted subnet ranges less frequently, it was able to complete the runs faster.

Other results, not included due to space constraints, show that P4RL consistently outperforms baseline in MCR and bug detection time over other queries mentioned in Figure 4.

5 RELATED WORK

Recently, multiple approaches for the P4 program verification have been proposed. Majorly, the existing tools are based on static analysis [2–5] of the P4 program and thus, fail to detect the runtime bugs or faults. Considering the example of a target device-independent or -dependent bugs illustrated in §2, the current solutions [2–5] are not able to detect such bugs, as they rely on static analysis and thus, executing runtime verification to observe the device behavior on various inputs is not feasible. In particular, if the parser in a P4 switch fails to check the IPv4 `ihl` field, the `checksum` is updated incorrectly leading to anomaly in the network behavior. For instance, the runtime verification of hash computations cannot be supported by the symbolic execution solutions [2, 4, 5]. Similarly, if the parser in a P4 switch fails to check the IPv4 `ihl` field, the `checksum` is updated incorrectly leading to anomaly in the network behavior. To detect such a problem is non-trivial, especially if the verification tools assume the faulty programs to be correct. P4RL detects such runtime faults. In-band network telemetry (INT) [23] only collects the data plane information, such as the traversed path of a packet, the ports taken, queue lengths or latency, however, unlike P4RL, INT cannot verify the correct forwarding behavior. Cocoon [24] aims at iterative verification as a part of the software design process using stepwise refinement approach. While this approach leads to programs matching their specification, it requires huge amount of additional and manual user input. In dynamic environments, where requirements can change quickly, such a manual approach is cumbersome and error-prone. [25–27] perform modelling of the



(a) Query 3 in Fig. 4

(b) Query 6 (line 18) in Fig. 4

(c) P4RL speedup against the baseline

Figure 5: Evaluation Results

network from the control plane to check the reachability, loop freedom, and slice isolation. ATPG [28] generates test packets based on control plane configuration for functional and performance verification in traditional networks and SDNs. [25–28], however, assume that control plane has consistent view of the data plane. [29–31] use different machine learning approaches for finding vulnerabilities or compiler specific-bugs which cause crashes, however, they are insufficient for network-related verification. P4RL executes verification to identify the bugs in a P4 switch.

6 DISCUSSION

Generalization: In order to greatly reduce the chance of overfitting, we train P4RL *Agent* using a single algorithm, the same set of hyper-parameters, as well as neural network architecture, for detecting different kinds of bugs or errors. Otherwise, it would not be possible to apply P4RL system to other P4 applications and packet headers.

Learning New Reward Functions: Reinforcement learning agents are only able to learn if the defined reward function reflects the learning goal correctly. In case of training the *Agent* for each query individually, a simple reward function was shown to be sufficient for effective learning. We consider adjusting parts in the developed model, e.g., reward function to optimize the *Agent*.

Improvement in the guiding of fuzzing process: Reinforcement learning and *p4q* guide the fuzzing process and make it focused. There is, however, still room to improve the guiding of the fuzzing process to make it even more effective. Fuzzing process can be augmented with software testing techniques like static analysis to reduce the huge and complex input search space.

7 CONCLUSION & FUTURE WORK

We have presented P4RL, a system for executing runtime verification of P4 switches automatically. P4RL via machine learning-guided fuzzing detects complex runtime bugs which cannot be detected by static analysis techniques. Through experiments on existing P4 applications, we show that P4RL outperforms the baseline approach.

As a part of the future work, processing of the queries defined using *p4q* and optimizations to P4RL *Agent* will be studied. We plan to extend the device-dependent queries to the *p4q* repertoire. The localization of the faults detected by P4RL and the corresponding corrective measures lays the groundwork of our future work.

8 ACKNOWLEDGEMENT

We thank Anja Feldmann, Georgios Smargadakis, 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 and by the European Research Council (ERC) grant ResolutioNet (ERC-StG-679158).

REFERENCES

[1] P4 Language Consortium. https://p4.org/specs/.
[2] R. Stoenescu, D. Dumitrescu, M. Popovici, L. Negreanu, and C. Raiciu. Debugging P4 programs with Vera. In *ACM SIGCOMM*, 2018.
[3] 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.
[4] M. Neves, L. Freire, A. Schaeffer-Filho, and M. Barcellos. Verification of P4 Programs in Feasible Time Using Assertions. In *ACM CoNEXT*, 2018.
[5] 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.
[6] Michal Zalewski. American fuzzy lop: a security-oriented fuzzer. *URl: http://lcamtuf. coredump. cx/afl/(visited on 06/21/2017)*, 2010.
[7] P. Godefroid, M. Y. Levin, and D. Molnar. SAGE: whitebox fuzzing for security testing. *Comm. of the ACM*, 55(3), 2012.
[8] A. Sapio, I. Abdelaziz, A. Aldilaijan, M. Canini, and P. Kalnis. In-network Computation is a Dumb Idea Whose Time Has Come. In *ACM HotNets*, 2017.
[9] S. Salman, C. Streiffer, H. Chen, T. Benson, and A. Kadav. DeepConf: Automating Data Center Network Topologies Management with Machine Learning. In *ACM NetAI*, 2018.
[10] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, 3rd edition, 2009.
[11] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.
[12] T. Schaul, J. Quan, I. Antonoglou, and D. Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.
[13] P4 Tutorial. https://github.com/p4lang/tutorials.
[14] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.
[15] Long-Ji Lin. Reinforcement learning for robots using neural networks. Technical report, CMU PA School of Computer Science, 1993.
[16] Keras: The Python Deep Learning library. https://keras.io/.
[17] Scapy. https://scapy.net/.
[18] P4 Behavioural model. https://github.com/p4lang/behavioral-model.
[19] P4Runtime. https://p4.org/p4-runtime/.
[20] VirtualBox. https://www.virtualbox.org/.

[21] Mininet. http://mininet.org/.

[22] P4 Language Consortium. $P4_{16}$ language specs, version 1.1.0, 2018.

[23] Changhoon Kim et al. Inband Network Telemetry (INT). Technical specification, Barefoot Networks, Jun 2016.

[24] Leonid Ryzhyk, Nikolaj Bjørner, Marco Canini, Jean-Baptiste Jeannin, Cole Schlesinger, Douglas B Terry, and George Varghese. Correct by construction networks using stepwise refinement. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, pages 683–698, 2017.

[25] Peyman Kazemian, George Varghese, and Nick McKeown. Header Space Analysis: Static Checking for Networks. In *Proc. USENIX NSDI*, 2012.

[26] Peyman Kazemian, Michael Chang, Hongyi Zeng, George Varghese, Nick Mckeown, and Scott Whyte. Real Time Network Policy Checking Using Header Space Analysis. In *Proc. USENIX NSDI*, 2013.

[27] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. VeriFlow: Verifying Network-Wide Invariants in Real Time. In *NSDI*, 2013.

[28] Hongyi Zeng, Peyman Kazemian, George Varghese, and Nick McKeown. Automatic test packet generation. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, pages 241–252. ACM, 2012.

[29] Mohit Rajpal, William Blum, and Rishabh Singh. Not all bytes are equal: Neural byte sieve for fuzzing. *arXiv preprint arXiv:1711.04596*, 2017.

[30] Patrice Godefroid, Hila Peleg, and Rishabh Singh. Learn&fuzz: Machine learning for input fuzzing. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 50–59. IEEE Press, 2017.

[31] Chris Cummins, Pavlos Petoumenos, Alastair Murray, and Hugh Leather. Compiler fuzzing through deep learning. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 95–105. ACM, 2018.