Dynamic Property Enforcement in Programmable Data Planes

Miguel Neves, Bradley Huffaker, Kirill Levchenko, *Member, IEEE*, and Marinho Barcellos

***Abstract*— Network programmers can currently deploy an arbitrary set of protocols in forwarding devices through data plane programming languages such as P4. However, as any other type of software, P4 programs are subject to bugs and misconfigurations. Network verification tools have been proposed as a means of ensuring that the network behaves as expected, but these tools frequently face severe scalability issues. In this paper, we argue for a novel approach to this problem. Rather than statically inspecting a network configuration looking for bugs, we propose to enforce networking properties at runtime. To this end, we developed *P4box*, a system for deploying runtime monitors in programmable data planes. *P4box* allows programmers to easily express a broad range of properties (both program-specific and network-wide). Moreover, we provide an automated framework based on assertions and symbolic execution for ensuring monitor correctness. Our experiments on a SmartNIC show that *P4box* monitors represent a small overhead to network devices in terms of latency, throughput and power consumption.**

***Index Terms*— P4, SDN, programmable networks, network debugging, monitoring.**

I. Introduction

PROGRAMMABLE data planes allow network operators to modify the packet processing pipeline of network devices to quickly deploy new protocols, customize network behavior, and implement advanced network services. The introduction of the P4 [1] programming language has greatly lowered the barriers to doing so, bringing data plane programming into the mainstream. Over the last years, an ecosystem of data plane software has emerged (e.g., [2], [3]), and we can expect to see network devices running code written by teams of developers across multiple organizations, assembled by a network operator from libraries and modules, in the near future.

Despite the simplicity of its programming model, P4 programs have demonstrated to be prone to a variety of bugs and misconfigurations [4], [5]. As a result, network operators need ways to ensure that the programs they produce behave correctly in order to reap the benefits of a data plane software ecosystem. Decades of progress in software engineering have produced mature tools and methodologies for ensuring that certain properties hold in a program, and this idea has been gradually extended to the networking domain. State-of-the-art network verification tools can take a model of the network, its configuration, and a set of properties specified using traditional formalisms (e.g., temporal logic or Datalog rules) and automatically check whether these properties hold for any packet [6], [7].

Although these tools have helped network operators to identify bugs before they manifest, they still face important issues that hinder their adoption in production networks. First, most of these tools require programmers to manually model data plane programs, which is a cumbersome and error-prone task [7]. Second, these tools are usually restricted in terms of the properties they can guarantee. For example, some of them are specialized to the verification of reachability properties in order to reduce verification times [8]. Third, more expressive tools capable of verifying multiple properties frequently face severe scalability issues (e.g., checking conformance with a protocol specification can take days even for a single data plane program [4]). Finally, programmers usually have to be proficient in formal verification techniques for correctly specifying their properties.

In this paper, we propose a novel approach to this problem which is based on dynamic (or runtime) enforcement rather than static verification. While the former cannot always provide the kind of strong correctness guarantees that the latter can, it has several practical advantages. First, we do not need to wait for the outcome of a long verification process in order to push a new configuration out to the network switches. In addition, runtime enforcement can promptly intervene if problematic situations actually occur. It means we can still extract some useful work from buggy code when it behaves correctly, and perhaps repair problems without disturbing any network service (see an example in Section IV-B.2).

In contrast to static verification, run-time enforcement also lets the developer express policy and mechanism using the same programming environment as the rest of the program. The value of this should not be underestimated: not only does it make life easier for the developer, it also prevents translation errors between implementation and policy domains. That is, rather than expressing a property, such as loop-free forwarding using a separate modeling or formal reasoning language, the programmer can write code to enforce and verify

Manuscript received March 18, 2020; revised December 5, 2020; accepted March 3, 2021; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor D. Malone. Date of publication April 1, 2021; date of current version August 18, 2021. This work was supported in part by NSF under Grant CNS-1740911, in part by Rede Nacional de Ensino e Pesquisa (RNP)/Coordenação de Tecnologia da Informação e Comunicação (CTIC) under Grant P4Sec, in part by Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) under Grant 140317/2017-1, and in part by Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES)/Brazil under Grant 001. *(Corresponding author: Miguel Neves.)*

Miguel Neves is with the Dalhousie University, Faculty of Computer Science, Halifax, NS B3H 4R2, Canada and also with the Federal University of Rio Grande do Sul (UFRGS), Porto Alegre 91501-970, Brazil (e-mail: miguelcneves2@gmail.com).

Bradley Huffaker is with the Center for Applied Internet Data Analysis (CAIDA), University of California San Diego (UCSD), La Jolla, CA 92093 USA.

Kirill Levchenko is with the Electrical and Computer Engineering Department, University of Illinois at Urbana-Champaign, Urbana, IL 61801 USA.

Marinho Barcellos is with the School of Computer and Mathematical Sciences, University of Waikato, Hamilton 3240, New Zealand.

Digital Object Identifier 10.1109/TNET.2021.3068339

the desired properties in the language of the program (i.e., P4 in our case).

To realize the benefits of our dynamic enforcement approach we developed P4box, a system for deploying runtime monitors in programmable data planes. A *program monitor* is a language construct we developed (as an extension to P4) inspired by the *Aspect-Oriented Programming* (AOP) paradigm [9] which provides language-level constructs for attaching code to designated points in an existing program without modifying the program itself. Programmers can use monitors to modify or verify the behavior of control blocks, parsers, and external functions of P4 programs, and thus ensure they respect a set of desired properties. Monitors are particularly well-suited to the context in which data plane programs are assembled from externally-maintained modules, where it may be desirable to alter or verify the behavior of these modules without modifying their code.

P4box instruments a P4 program with monitors at compile-time in such a way that the former cannot circumvent or interfere with the latter. Moreover, monitors can be combined to enforce more complex properties such as the ones involving extraction and emission of labels on packets (see an example in Section IV-B.1). In summary, we make the following contributions:

- ❖ We design an extension to the P4 data plane programming language, called a *monitor*, that allows a programmer to specify properties about the network (using P4) in the form of pre- and post-conditions to control-blocks, parsers and extern functions (Section III).
- ❖ We develop P4box, a system for deploying runtime monitors in programmable data planes by instrumenting P4 programs at compile-time in such a way that the former cannot be hindered, tampered or circumvented (Section III).
- ❖ We show how P4box can be used to enforce several networking properties, including packet well-formedness, header protection, and waypointing (Section IV).
- ❖ We provide an automated framework based on assertions and symbolic execution for allowing programmers to check properties of interest on monitors (Section VI).
- ❖ We evaluate P4box on various applications running in a SmartNIC and show that monitors impose low overhead to network devices in terms of latency, throughput and power consumption (Section VII).

This paper extends our earlier conference paper [10] by describing our automated framework for ensuring monitor correctness as well as the extensive set of experiments we performed on a commodity SmartNIC. Also, we have updated the related work to reflect the most recent advances we found in the literature. Next, we review the architecture of programmable network devices, summarize the main aspects of P4 programs, and motivate the development of property enforcement mechanisms in programmable data planes.

II. Background and Motivation

A. Programmable Network Devices

Programmable network devices (a.k.a. *targets*) are packet processing elements (i.e., switches, SmartNICs, NetFPGAs) that allow network programmers to configure their data plane. These devices implement variations of an architecture known as PISA (Protocol Independent Switch Architecture).[1] PISA-based devices contain multiple programmable blocks, which can be parsers, deparsers, match-action stages or queueing systems. Figure 1 presents an example of a PISA-based switch containing three programmable blocks (dashed boxes): a parser, a match-action pipeline and a deparser. Each programmable block can be configured by developers using a data plane programming language (typically P4), and the organization and capabilities of these blocks are abstracted to P4 programs as an interface or *architecture model*.

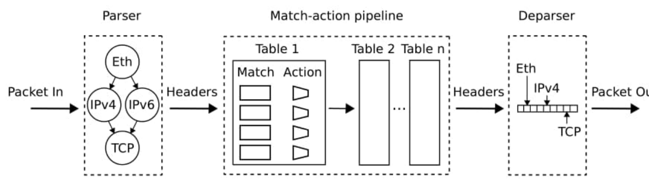

Fig. 1. Example of PISA-based switch. Dashed blocks can be programmed in P4.

```
 1 parser ParserImpl( packet_in packet ){...}
 2
 3 control Pipeline( inout headers hdr ){
 4   ...
 5   action route( bit<9> iface ){ ... }
 6
 7   /* Route IPv4 packets */
 8   table route_packet {
 9     actions = { route; }
10     key = {
11       hdr.ipv4.srcAddr : ternary;
12       hdr.ipv4.dstAddr : ternary;
13     }
14     size = 1024;
15   }
16
17   apply{ route_packet.apply(); }
18 }
19
20 control DeparserImpl( packet_out packet ){...}
21
22 Switch(ParserImpl(), Pipeline(), DeparserImpl())
```

Fig. 2. Example P4 program.

B. P4 Programs

As a domain specific language, P4 offers many constructs to facilitate the specification of packet processing tasks. Programmers can, for example, declare packet headers, parsers, tables, actions to modify packets, and control blocks to compose sequences of tables. These abstractions are used to configure different programmable blocks in network devices, and the configuration of all blocks comprises a P4 program. Figure 2 shows an example of a program for configuring the PISA-based switch described in Section II-A. In this example, the match-action pipeline block implements a single table that routes packets based on their IPv4 addresses (l.8-15).

C. Data Plane Bugs

Although the simplicity of its programming model (e.g., P4 programs have no loops or dynamic memory allocation [1]), data plane programs have demonstrated to be prone

[1]https://p4.org/assets/p4-ws-2017-p4-architectures.pdf

to many bugs and misconfigurations. Bugs in P4 vary in nature, but overall they can be both generic bugs (i.e. well-known from other programming languages) such as information overwriting[2] and data use-before-initialization,[3] and also network specific bugs such as the creation of malformed packets [8], incorrect implementation of protocol specifications [5] or policy violations due to bad table configurations. In this context, it is essential to develop mechanisms that support the development of secure and correct network data planes. In particular, the myriad of static analysis tools available to check P4 programs [4], [8], [11] cannot detect runtime bugs (e.g., checksum implementation or platform-dependent bugs) and thus a runtime verification tool becomes necessary.

III. P4box

P4box is a system that allows network programmers to deploy runtime *monitors* in programmable data planes. Using P4box programmers can attach monitors before and after control blocks, parser state transitions, and calls to external functions of a P4 program. Each monitor can modify the input and output of the code block or function it monitors. This enables the verification of pre- and post-conditions which can be used to enforce specific properties or modify the behavior of the monitored block. P4box inclines monitor code into the monitored P4 program at the intermediate representation level (i.e., during the compilation of the latter). The resulting program (original code plus monitors) then continues the compilation as before, which allows P4box to be used with any backend compiler based on the $P4_{16}$ reference implementation. In the rest of this section, we provide an overview of P4box and its runtime monitors (Section III-A), and describe the three kinds of monitors P4box can deploy in detail (Sections III-B, III-C, and III-D).

A. Overview

A runtime monitor interposes on the interaction of a P4 control block or parser with the rest of the execution environment (Figure 3), allowing the monitor programmer to modify the behavior of the enclosed P4 block with the rest of the environment. A P4 programmable block (either a control block or parser) interfaces with the rest of the P4 execution environment at entry into the block, return from the block, and at calls to architecture-supplied external functions. In the P4box programming model, when a programmable block is invoked, control first passes to a monitor, also written in P4, before passing to the intended programmable block. Similarly, when a programmable block completes processing, control first passes to the monitor before returning to the device. This allows a monitor to modify the behavior of programmable blocks in a well-defined way.

Monitors can also interpose on calls to external functions: when a programmable block invokes an external function, control first passes to the monitor, then the function, and then back to the monitor again, before returning to the programmable

[2]https://github.com/p4lang/switch/issues/97

[3]https://github.com/p4lang/switch/pull/102

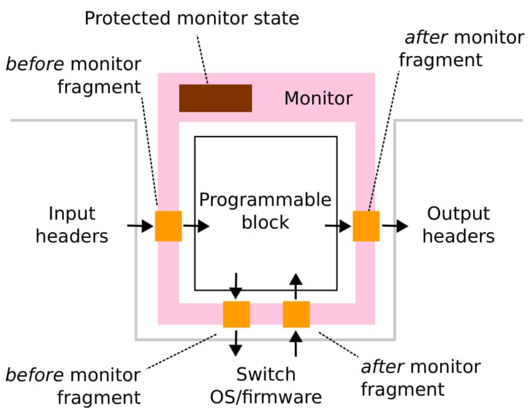

Fig. 3. P4box programming model.

block. A monitor can thus modify the apparent behavior of an external function. Monitors are declared and defined at the top level of a P4 program, alongside control blocks, parser blocks, and other top-level declarations. The syntax for a monitor is:

```
monitor <name> ( [param-list] ) on <object> {
    [local-declarations]
    (before | after) { <p4-statements> }
}
```

Each monitor is identified by a unique *<name>* and may receive additional parameters (*<param-list>*) containing headers and metadata in addition to the parameters of the monitored object. Every monitor must be associated with a data plane *<object>*, which can be a parser, control block or extern function. The resource type defines the set of *<p4-statements>* elements the monitor supports. Monitors can have two types of methods, namely: *before* and *after*, which specify code fragments that are executed before and after the monitored resource, respectively. They can also contain local declarations (e.g., actions, tables) visible inside the monitor but not the monitored block. Following the P4 semantics, a local declaration is visible only inside its enclosing monitor.[4]

Figure 4 shows the P4box workflow. The original P4 program and P4 source files defining runtime monitors are provided to P4box which combines the original program with the monitors at the intermediate level to produce a new program suitable for further compilation. At the end, machine-level code containing all monitors is generated for a variety of targets. During the instrumentation process, P4box takes advantage of language features provided by P4 such as separate scopes and namespaces in addition to static analysis to provide the following guarantees for each monitor:

- ***Complete Mediation***: The flow of execution of the original data plane program will always pass through a monitor (when one is defined by the programmer). This means it is not possible for the original program to circumvent a monitor.
- ***Non-Interference***: The original program cannot interfere in the operation of a monitor (e.g., by modifying its local

[4]It is possible to enable access from multiple monitors to the same data by passing the latter as a parameter to the desired monitors.

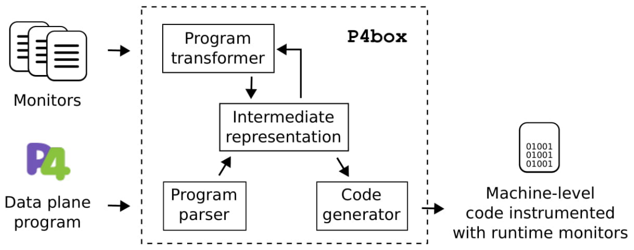
Monitors
Data plane program
P4box
Program transformer
Intermediate representation
Program parser
Code generator
Machine-level code instrumented with runtime monitors

Fig. 4. P4box workflow.

```
1  monitor hdrInvMonitor() on Pipeline {
2    ipv4_t protec_ipv4;
3    udp_t  protec_udp;
4
5    before {
6      protec_ipv4 = hdr.inner_ipv4;
7      protec_udp = hdr.inner_udp;
8    }
9
10   after {
11     if( protec_ipv4 != hdr.inner_ipv4 ||
12         protec_udp != hdr.inner_udp ){
13       /*Run enforcement action
14         (e.g., restore original header
15         value, notify the control plane,
16         write log) */
17   }}
15 }
```

Fig. 5. Example of control block monitor to enforce header protection.

variables or headers), which means monitors are completely isolated from the data plane program.

Together, the complete mediation and non-interference properties allow monitors to restrict what the original P4 program is allowed to do even when the latter is *untrusted* (e.g., a third-party program). Monitors are thus not only an aspect-oriented P4 program structuring mechanism, but also a software sandbox that can be used to encapsulate untrusted or buggy P4 code. Next, we show examples and describe each of the three kinds of monitors P4box supports in more detail.

B. Control Block Monitors

P4box can attach monitors to top-level control blocks. In this case, *before* and *after* contain statements that will be executed at the beginning and the end of block, respectively. Figure 5 shows an example of a control block monitor, which could be used to detect and process information overwriting bugs^2. This monitor is responsible for ensuring that a header is not erroneously modified by the data plane program. The monitor is attached to the processing pipeline and has two elements: i) before the programmable block, it collects state from the original packet as soon as it is parsed (l.5-8); and ii) after the block, it tests whether monitored headers were modified (l.10-17). Local variables (i.e., visible only to the monitor) are used to store protected headers (l.2-3). If the monitor detects a violation, different actions can be performed to enforce the desired property (e.g., restore the original header value, notify the network controler, log an event), being up to the programmer to decide what to do.

```
control <control_name>
    ( <combined-params> ){
  [local_elements]
  [monitor_local_elements]

  apply{
    [before_statement]
    ...
    [block_statement]
    ...
    [after_statement]
  }
}
```

```
control pipeline(inout newHeaders hdr,
                 inout metadata meta){
  ipv4_t protec_ipv4;
  ...
  apply {
    protec_ipv4 = hdr.inner_ipv4;
    ...
    if(protec_ipv4 != hdr.inner_ipv4
     || protec_udp != hdr.inner_udp){
      ...
    }
  }
}
```

Fig. 6. Instrumentation of control blocks.

C. Parser Monitors

Parser monitors, on their turn, can be attached to top-level parsers. As such, *before* and *after* can contain finite state machines and both of them must have a start and accept state. It is possible to specialize a parser monitor to a specific parser state, in which case *before* and *after* are associated only to the latter. An example of a parser monitor is shown in Figure 11-lines 6 to 17, where the monitor is attached to the parse_ethernet state and used to extract an enforcement header. Parser monitors are also particularly useful for skipping the extraction of packet bits that for some reason (e.g., confidentiality) should not be visible to the data plane program.

D. Extern Monitors

Extern monitors are attached to extern calls. Their capabilities are restricted to what actions can do in P4 because of limitations the latter have on extern callers (e.g., it is not possible to make local declarations or invoke a table from inside an action). Similar to parser monitors, extern monitors can also be specialized to subgroups of a resource. In this case, a type signature is used to apply a monitor only to a subset of the extern calls. An example is presented in Figure 11-lines 20 to 24, where the extern monitor is applied only to calls for emitting headers of type ethernet_t. Unfortunately, P4box cannot distinguish between two or more calls for an extern with the same type signature. In this case, P4box emits a warning to the user whenever this situation happens, so that the user can take the appropriate action to deal with it. One option is to create a type alias (i.e., redefine the same type with a different name) and then program separate monitors to inspect them. Extern monitors are useful to mediate how the data plane program interacts with the platform underlying it.

IV. Enforcing Properties

The value of a mechanism like P4box is best seen through examples. In this section, we show how P4box can be used to enforce several kinds of properties in the data plane. Generally, these fall into two categories: program properties, which are properties of a single program's behavior, and network-wide properties, which are properties of several network devices' behavior.

A. Program Properties

Program properties concern the behavior of a program running on an individual device. These properties must hold

```
parser <parser_name>
    ( <combined-params> ){
  [local_elements]
  [monitor_local_elements]
  ...
  state <s_k-1> {
    transition [before_FSM];
  }
  [state before_FSM {
    transition <s_k> }]
  state <s_k> {
    transition [after_FSM];
  }
  [state after_FSM {
    transition <s_k+1> }]
  state <s_k+1> {
    transition <s_k+2>
  }
  ...
}
```

```
parser pipeline(packet_in packet,
             out newHeaders hdr){
  ...
  state parse_ethernet {
    transition _M_START_;
  }
  state _M_START_ {
    transition select(...){
      16w0xFFFF : parse_wp_header;
      ...
    }
  }
  state parse_wp_header {
    transition parse_ipv4;
  }
  state parse_ipv4 {
    transition parse_tcp;
  }
  ...
}
```

Fig. 7. Instrumentation of parsers.

```
control <control_name>
    ( <combined-params> ){
  action <action_name>(){
    ...
    [before_statement]
    [extern_A_call]
    [after_statement]
    ...
  }

  apply{
    ...
    [before_statement]
    [extern_A_call]
    [after_statement]
    ...
  }
}
```

```
control DeparserImpl(
          packet_out packet,
          in newHeaders hdr){
  apply{
    ...
    packet.emit(hdr.ethernet);
    packet.emit(hdr.wp_header);
    packet.emit(hdr.ipv4);
    ...
  }
}
```

Fig. 8. Instrumentation of extern calls.

regardless of how the device is configured or connected in a topology. They are also referred to as *network function properties* in the literature [12]. In this work, we consider two types of program properties: *generic safety* properties, which correspond to low-level properties related to the correct operation of a data plane program (e.g., packet formation properties and use-after-initialization), and *functional* or semantic properties, which guarantee the program conforms to a given user-specification (e.g., an RFC). Below we show how we enforce some program properties of interest, well-formedness and header protection.

1) Well-Formedness: We assume the same definition for well-formedness properties as [8]. That is, we are looking for ensuring that the packets produced (i.e., emitted) by one P4 program can be consumed by another according to a given protocol (or protocol stack) definition. In practice, this boils down to checking which headers are being emitted and their particular order (one could also consider checking the set of values each header can assume, though that also fits into the context of functional or semantic properties). Enforcing well-formedness invariants is particularly useful in hybrid networks (i.e., networks containing both P4-enabled and legacy devices), where the elements may not support the same set of protocols. P4box can enforce well-formedness properties (e.g., packets do not contain both an IPv4 and IPv6 header, ICMP packets always have an IPv4 header) with simple checks of header validity at the end of the processing pipeline.

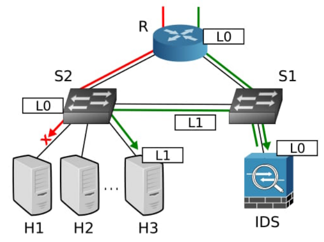

Fig. 9. Example topology for waypointing.

2) Header Protection: In some cases, it may be desirable to ensure that a header is not modified by a forwarding device or programmable block. For example, in a deployment where VLANs are used to isolate potentially untrusted domains, it may be necessary to provide strong assurance that a VLAN tag is not modified by a forwarding device. P4box can be used to ensure that headers are not modified by collecting the appropriate packet state at the beginning of the processing pipeline (e.g., the value of a VLAN tag), and comparing it against the emitted headers. Such properties can be easily extended to allow only transformations to a pre-defined domain (e.g., source MAC can be modified only to a set of output interface addresses).

B. Network-Wide Properties

Network-wide properties concern forwarding devices when configured and connected in a particular topology [12]. These properties may involve basic predicates (e.g., A can reach B) as well as state and quantities (e.g., express desired behaviors for networks containing middleboxes or having latency constraints). We now describe how P4box can enforce common network-wide properties.

1) Waypointing: Network operators may want to force packets to pass through a sequence of devices (waypoints) before the network delivers them to an end host. P4box can enforce waypoint properties by checking and updating labels whenever these packets cross a device in the chain. As an example, Figure 9 shows a scenario where packets coming from an external network (i.e., through router R) must first be inspected by an IDS system before arriving at a web server (hosts H1–H3). In this case, a P4box monitor in R introduces labels in each packet in order to enforce waypointing. These labels are then updated by another monitor at switch S1, and a third monitor checks them at switch S2 for dropping packets that are destined to the web servers and do not contain the updated tag (L1). Figure 10 shows how P4box interacts with the P4 program to enforce waypointing, where vertical arrows represent the flow of execution. Note that P4box traps the program at three points: first, between the parsing of the Ethernet and IPv4 headers, to check whether the packet contains a label and extract the latter; second, right before the beginning of the match-action pipeline, to operate on the label (e.g., check, updates or remove) depending on how the device is connected in the topology; finally, to emit the label during the deparsing phase.

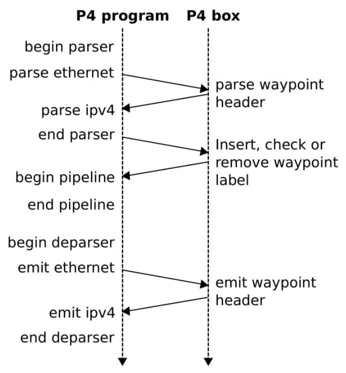

Fig. 10. Interaction between P4box and the P4 program to enforce waypointing.

Figure 11 shows a summary (with some parts omitted due to space constraints) of the code used to enforce waypoint properties. Each trap is programmed as a separate monitor. Parser (l.6-17) and extern (l.20-24) monitors are employed to extract and emit labels, which are declared in the wp_header (l.2). Moreover, a control block monitor uses match-action tables to insert, check/update and remove labels according to the incoming/outgoing ports of the packet. P4box monitors can be configured (proactive or reactively) to reroute packets on-the-fly and correct property violations. Moreover, we can extrapolate the labeling mechanism described above to enforce path conformance (i.e., to guarantee that the actual path taken by a packet conforms to the operator policy). In this case, P4box monitors check and update packet labels on every hop.

2) Traffic Locality: Sometimes operators want to preserve traffic locality, e.g., packets flowing between two VMs in the same rack must not leave the top-of-rack switch in a data center, or traffic between two hosts in the same autonomous system should not leave its borders [7]. P4box can enforce traffic locality by controlling the set of output ports a packet can take. For example, packets from host A to B in Figure 12 are not allowed to be forwarded to upper ports. Figure 13 shows how P4box interacts with the P4 program to enforce traffic locality. First, it hooks the flow of execution at the beginning of the processing pipeline to save the state of required headers (e.g., MPLS or IPv4) before the program can modify them. Then, at the end of the pipeline, it uses the saved state as well as information about the outgoing port to check whether the packet can be forwarded. Figure 14 shows relevant parts of the monitor used to enforce traffic locality. It contains a single table that matches a set of control headers and the outgoing port (l.8-16), and runs an enforce_locality action (e.g., send the packet to a different outgoing port) when a violation is detected (l.4).

```
1  struct p4boxState {
2    waypoint_t wp_header;
3  }
4
5  //Parser monitor to extract enforcement header
6  monitor wpParser(inout p4boxState pstate) on ParserImpl {
7    after parse_ethernet {
8      state start {
9        transition select(packet.lookahead<bit<32>>()){
10         16w0xFFFF : parse_wp_header;
11         default : accept;
12       }
13     }
14     state parse_wp_header {
15       packet.extract(pstate.wp_header);
16       transition accept;
17 }}}
18
19 //Extern monitor to emit enforcement header
20 monitor wpExtern(inout p4boxState pstate)
21                            on emit<ethernet_t>{
22   after {
23     packet.emit(pstate.wp_header);
24 }}
25
26 monitor wpControl(inout p4boxState pstate) on Pipeline {
27   ...
28   table check_waypoint {...}
29   ...
30
31   after {
32     //Enforce waypointing property
33     insert_label.apply();
34     check_waypoint.apply();
35     remove_label.apply();
36 }}
```

Fig. 11. Supervisor to enforce waypointing.

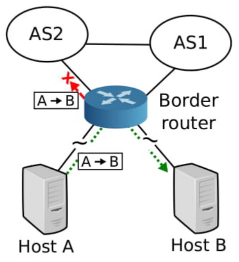

Fig. 12. Example topology for traffic locality.

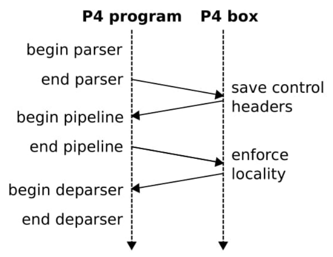

Fig. 13. Interaction between P4box and the P4 program to enforce traffic locality.

V. Program Instrumentation

Control Block Monitors: P4box performs the instrumentation of control blocks in three steps: first, monitor parameters containing headers and metadata are merged with parameters of the monitored block (e.g., joining the fields of two structs to create a super struct). If during this process P4box identifies there is no feasible mapping, a message is emitted and the instrumentation process is aborted. This is the case, for example, when a monitor receives an "inout" parameter, but the monitored block only supports "in" ones and proceeding with the instrumentation would create a semantic mismatch; second, *before* and *after* blocks as well as local declarations are inserted in the monitored block; finally, a name resolution

```
monitor tlMonitor(inout p4boxState pstate)
                              on Pipeline {
  //Run enforcement action
  action enforce_locality(){ ... }

  //Check if packet violates locality
  //(i.e., tries to leave AS)
  table traffic_locality_table {
    actions = { NoAction; enforce_locality; }
    key = {
      hdr.ipv4.srcAddr : ternary;
      hdr.ipv4.dstAddr : ternary;
      standard_metadata.egress_port : exact;
    }
    size = 512;
  }

  after { traffic_locality_table.apply(); }
}
```

Fig. 14. Supervisor to enforce traffic locality.

pass maps monitor names to their new namespaces. The left part of Figure 6 illustrates this transformation, where a generic control block is instrumented with its monitoring primitives. A corresponding example is shown on the right, representing the instrumentation performed to the monitor specified in Figure 5. As a result of this transformation, all packets crossing the control block also pass through the monitor since P4 assumes network devices execute statements in order.

Parser Monitors: To instrument parsers, P4box takes into account if *before* and *after* are attached to states or not. If not, it assumes the start and end (i.e., accept) states of the monitored parser as its hooking points. The left part of Figure 7 shows the transformations P4box applies. Assuming state S_k is being monitored, P4box links the finite state machine specified inside *before* (before_FSM) between states S_{k-1} and S_k by modifying state transitions. An analogous process is performed for the finite state machine specified inside *after* (after_FSM), linking it between states S_k and S_{k+1}. The right part of Figure 7, on its turn, shows an example of these transformations, where P4box performs the instrumentation to the parser monitor specified in Figure 11. Instead of transitioning directly from state parse_ethernet to parse_ipv4, the execution flow goes through states _M_START_ and parse_wp_header.

Extern Monitors: Finally, P4box instruments extern calls by adding *before* and *after* blocks right before and after every monitored call, respectively. The left part of Figure 8 illustrates this transformation, where the same extern call appears twice (inside an action and directly in the control block body). For the particular case in which a monitor has a type signature, only calls with that signature are instrumented. As an example, the right part of Figure 8 shows the instrumentation to the extern monitor specified in Figure 11.

VI. Checking Monitor Correctness

Monitors are less likely to contain bugs compared to P4 programs due to their smaller size. For example, a monitor to enforce header protection has no more than a dozen of lines of code while traditional P4 programs usually have hundreds to thousands of lines (two to three orders of magnitude larger) [4], [11]. Despite their simplicity, monitors are still subject to bugs and misconfigurations though. For this reason, we developed an automated framework for allowing programmers to check invariants in their specified monitors.[5]

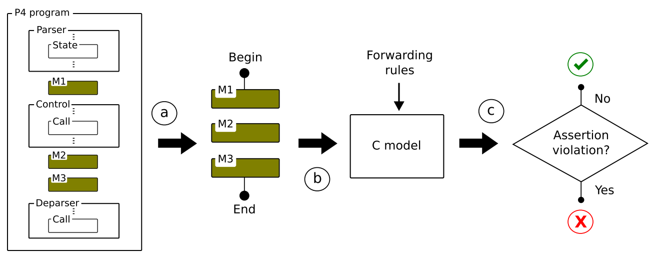

Fig. 15. Workflow for checking monitor correctness. M1, M2, M3 = annotated monitors. a = monitor assembling. b = model extraction. c = symbolic execution.

Our framework is inspired by assert-p4 [5], a state-of-the-art tool for checking invariants in P4 programs. As for assert-p4, our framework is also based on assertions and symbolic execution (see Figure 15 for its workflow). First, programmers annotate monitors with assertions expressing properties of interest. For that, we consider the same assertion language as proposed in [5], which is also a good fit to our problem since monitors are comprised of P4 constructs. The language enables the specification of basic predicates (there is no support for quantifiers) and includes elements for expressing logical, relational and arithmetic expressions, as well as conditional statements and basic tests involving packet headers (e.g., whether a header was extracted from a packet or not). As shown in our previous work [5], this can be used to specify a wide set of properties, both program (or monitor) dependent (e.g., output port is set to "local" port) and generic ones (e.g., packet header is not modified). Assertions can only contain monitor-private variables. However, it is possible for a monitor to hook the state of a program variable (e.g., through an assignment) and store it in a monitor-private one. In such a case, the program variable is treated as input to the monitor and thus made symbolic at the beginning of the verification process.

Once annotated, monitors are assembled in a "virtual program" respecting the same order of execution as the monitored code. This means if monitors A and B are monitoring programmable blocks X and Y, respectively, and X runs before Y, then A will precede B. In addition, the assembled code also contains all header and metadata definitions from the original program, which similarly to program variables are also treated as symbolic inputs by the verification engine and enable programmers to check invariants on monitors that manipulate program state (e.g., change a header value). After the assembling phase, the new virtual program is translated into an equivalent model in C, and assertions are checked using a symbolic execution tool.

Translating monitors to C allows us to use an off-the-shelf symbolic execution engine, e.g., KLEE [13], to check the desired properties. Moreover, tools to ensure the correctness of the translation process are also available.[6] As an example,

[5]The procedure our framework implements is restricted to monitors and should not be used to check invariants about the program as a whole.
[6]https://github.com/gnmartins/assert-p4

```
 1 #include "klee.h"
 2
 3 //Model monitor locals
 4 ipv4_t protec_ipv4;
 5 udp_t protec_udp;
 6
 7 //Make monitor inputs symbolic
 8 void symbolizeInputs(){
 9   klee_make_symbolic(&hdr, sizeof(hdr), "hdr");
10   klee_make_symbolic(&meta, sizeof(meta), "meta");
11 }
12
13 //Model monitor logic
14 void hdrInvMonitor_before(){
15   protec_ipv4 = hdr.inner_ipv4;
16   constant_protec_var = protec_ipv4;
17   protec_udp = hdr.inner_udp;
18 }
19
20 void hdrInvMonitor_after(){
21   if( protec_ipv4 != hdr.inner_ipv4 ||
22       protec_udp != hdr.inner_udp ){ ... }
23 }
24
25 int main(){
26   symbolizeInputs();
27   hdrInvMonitor_before();
28   hdrInvMonitor_after();
29   //Model assertions
30   if( constant_protec_var != protec_ipv4 ){
31     //Assertion is violated }
32   return 0;
33 }
```

Fig. 16. Equivalent model in C to the monitor described in Section III-B.

Figure 16 shows the resulting model for the monitor described in Section III-B (we omit some parts for the sake of simplicity). The *main* code (lines 25-32) controls the call order for the monitors, which are on their turn modeled as additional functions (lines 14-23). We make all monitor inputs (i.e., packet headers, metadata and protected state) symbolic (lines 8-11), so that they can be comprehensively checked by the symbolic execution engine. Local monitor definitions (e.g. variables and match-action tables) are modeled as unique global constructs (lines 4-5). Finally, each assertion is modeled independently, and usually involves variables that are set and tested at relevant points in the program. For example, the assertion modeled in lines 16 and 30 checks whether the monitor, which should ensure a packet header is not modified, is not itself erroneously modifying the header. We refer to [5] for more details on the translation process.

Performance: One of the key concerns in automated testing is performance as not rarely the cost of checking a program invariant becomes prohibitive in practice. For example, symbolic execution is particularly known for its path explosion issue [14] and other techniques also have their own drawbacks (e.g., the state space explosion problem in model checking [15] or large logical formulas in SMT solving [11]). A few techniques (e.g., program slicing and directed symbolic execution) have been proposed to reduce this burden in the context of P4 programs, but it still takes hours or even days to check a relatively complex program instance [4], [5].

To demonstrate the scalability of checking monitor invariants using our approach, we applied our framework to check basic semantic properties (i.e., show that monitors in fact meet their desired behavior) on the monitors described in Section IV. We run our experiments in a single-core virtual machine equipped with 4GB of RAM and Ubuntu 18.04.

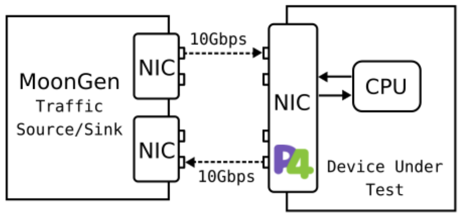

Fig. 17. Testbed topology. Dashed arrows represent the data flow. Solid arrows indicate control traffic (e.g., for programming the NIC firmware using P4 and collecting statistics).

We used KLEE 2.0, the Z3 solver, and LLVM 6.0 as the symbolic execution engine. In each case, our framework was able to check the whole input space in less than a second. This is mainly because of the small size of monitors, which typically result in no more than a few hundred execution paths.

VII. Evaluation

Because dynamic enforcement happens at run time, it may impose a performance penalty compared with static verification. In this section, we analyze the performance overhead of P4box and show it is small for many useful properties and applications. We implemented a prototype of P4box by extending the $P4_{16}$ reference compiler.[7] Our system has around 1.5K lines of C++ code and is publicly available.[8] We modified the front-end compiler to instrument programs by adding additional passes over their intermediate representation. Our examples and the workloads used in our experiments are also available online.

Figure 17 shows the topology of the setup for evaluating P4box. The device under test (DuT) is equipped with a 4-core Intel Core i3 530 2.93GHz CPU and a single-port 40G Agilio CX smart NIC running in breakout mode (i.e., 4×10G virtual interfaces). The traffic generator, on its turn, contains a 4-core Intel Xeon E31220 3.1GHz CPU and two dual-port 10G Agilio CX NICs. We configure the traffic generator with MoonGen [16] and use a single interface in each NIC for sending and receiving traffic respectively, leaving the other interfaces unused. Unless explicitly mentioned otherwise, our analyses consider the traffic generator creates a 10 Gbps stream of 64-byte UDP packets ($\sim$14.8 million packets per second).

All P4 programs run as embedded firmware in the DuT NIC and are isolated from other end host resources (e.g., CPU, memory and operating system). We use P4box to create instrumented P4 programs and then the Netronome P4 compiler with MAC timestamps and shared content stores enabled to convert instrumented programs into target specific code. Except for Section VII-A, in which we analyze the cost of enforcing each property separately, all our experiments assume P4box instruments data plane programs with the four properties described in Section IV, so that we could measure overheads in more demanding conditions.

We measure throughput, latency and power consumption to compare the forwarding performance of the device under test with and without P4box. To measure throughput, we count the

[7]https://github. com/p4lang/p4c
[8]https://github.com/mcnevesinf/p4box

TABLE I

AVERAGE, 5TH AND 95TH-PERCENTILE LATENCY COST OF THE PROPERTIES DESCRIBED IN SECTION IV

Property	Latency (us)		
	Avg	5th	95th
Well formedness	1.91	1.24	3.61
Header protection	1.32	1.02	2.30
Traffic locality	1.25	1.02	1.80
Waypointing	0.97	0.87	1.40
All 4 properties	2.35	1.74	3.12

number of packets processed in the NIC each second using a P4 counter. We report the average of 10 runs where each run lasts for 30 seconds. To measure the packet processing latency, we collect NIC ingress/egress timestamps and report results over 100 packets. Finally, we use the automated script provided by Netronome (*nic-power*) to read the board power consumption every 100 milliseconds, and similarly to latency measurements also report results over 100 reads. All measurements are performed after a 5 seconds warm-up interval.

A. Property Overhead

We start looking at the overhead of each property in isolation. To evaluate this overhead, we instrumented a very simple data plane program (L3 routing – see Table II) with P4box configured to enforce a single property, and measure the performance drawback compared to a baseline (i.e., the same program without any instrumentation). Table I shows the latency overhead, in microseconds, for enforcing the properties described in Section IV.[9] As we can see, the overhead is under 5 μs even when we consider all properties together – last line in the table. This is at least one order of magnitude smaller than the latency cost for processing a packet in many data plane applications (see Section VII-B). Also, the overhead is clearly not additive, meaning the cost for enforcing a combination of properties is not the same as the sum of the cost for enforcing the individual ones. This is because P4box can employ resource sharing among monitors (i.e., the same monitor can be used to ascertain more than one property) in order to optimize their performance. For example, two properties that involve storing and checking the same program state (e.g., IP source value) could take advantage of a common monitor to store this information.

B. Application Performance

Next, we evaluate the forwarding performance of the device under test while running real-world applications instrumented with P4box. We select instances of 4 popular applications across different domains: (1) L3 routing, which forwards packets based on destination IP addresses [17]; (2) Load balancing, which uses Othello hashes for mapping virtual IPs (VIPs) to destination servers (DIPs) [18]; (3) DDoS detection, which adopts counting sketches to identify malicious flows [19]; and (4) Surveillance protection, which encrypts IP addresses to

[9]For well-formedness we are checking header emissions (i.e., whether incoming and outgoing packets have the same headers) as no header should be removed or added in the application we run.

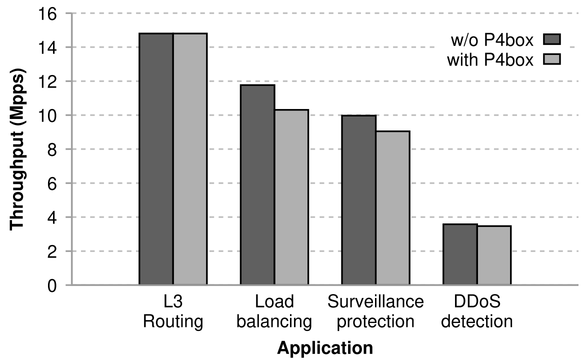

Fig. 18. Average throughput for the evaluated applications. Standard deviation is less than 0.1 Mpps.

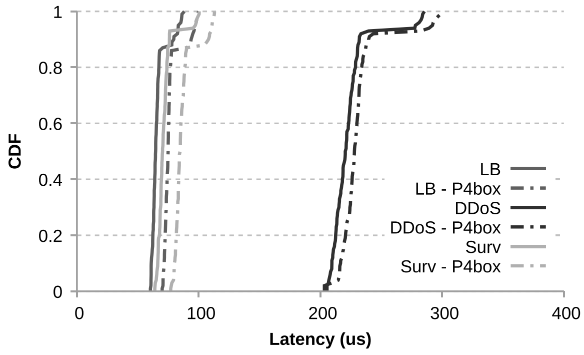

Fig. 19. CDF of the packet latency for the evaluated applications.

obfuscate information about Internet users and devices [20]. Table II summarizes the P4 programs implementing these applications. Each program has a distinct number of matching tables, which results in different pipeline depths. Moreover, three of the programs do not manipulate any persistent state in the device while the remaining one uses registers for storing packet counts.

Figure 18 compares the throughput of the device under test for the evaluated applications. In all cases, the overhead for running P4box is small, representing a throughput drop of about 9% (1.4 Mpps) for load balancing, 6% (0.9 Mpps) for surveillance protection and 0.7% (0.1 Mpps) for DDoS detection. Interestingly, there was no noticeable overhead for L3 routing as this application was able to achieve line rate in both scenarios.

Figure 19 compares the cumulative distribution of the packet processing latency for the different applications. As can be seen, P4box implies a small latency overhead for packets. For example, the increase in the median latency is below 20% in all cases (4% for DDoS detection, 15% for load balancing and 19% for surveillance protection). Results are similar when we look at the tail latencies, with an overhead smaller than 15% at the 99th percentile in the worst case (for load balancing). Overall, the more complex the application the lower the penalty for running P4box. That is, the cost of a P4 program (e.g., in terms of latency and throughput) is directly proportional to the number of elements it contains, which is also true for P4box monitors. In this case, the bigger the difference between a P4 program and a set of monitors in

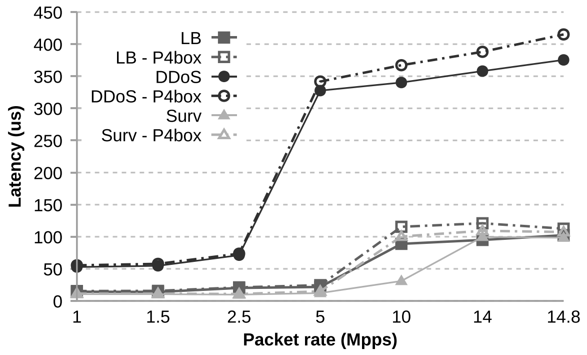

Fig. 20. 95-percentile tail latencies at different packet rates.

TABLE II
EVALUATED APPLICATIONS. LoC = LINES OF CODE

Application	#Tables	Stateful	LoC
L3 routing [17]	3	N	160
Load balancing [18]	11	N	420
Surveillance protection [20]	6	N	480
DDoS detection [19]	2	Y	540

terms of the number of operations they perform, the lower the performance penalty the set of monitors implies.

C. Effect of Packet Rate

We now turn our attention to examining how different packet rates affect P4box. We consider a maximum load scenario in which the traffic generator sends traffic at the constant rate of 10Gbps, but changes the packet size and consequently the number of packets sent per unit of time. For example, the traffic generator can send up to 14.8 million 64-byte packets per second, but this number reduces to approximately 1 million if it instead sends packets of 1500 bytes.

Figure 20 compares the 95-percentile tail latency for different applications as a function of the packet rate. P4box overhead is negligible up to 5 Mpps. This is because NIC resources are not overloaded at low rates. Above 5 Mpps, P4box increases tail latencies around 20% as a result of bottleneck on NIC. This bottleneck is more prominent in computing-intensive applications such as DDoS detection, where higher processing demands per packet induce a head-of-line (HOL) blocking and consequently queueing formation at input ports [21].

D. Power Consumption

Finally, we evaluate how P4box affects the SmartNIC power consumption. First, we measure the overhead for different link utilizations. We start with an idle system, and gradually increase the input rate until it achieves full link capacity (10 Gbps). Figure 21 shows the results for the L3 routing application. As we can see, P4box overhead is smaller than 5% (0.4W) even in the worst case (i.e., when link utilization is maximum). Moreover, this overhead slightly decreases for lower utilizations.

We also measure the overhead for different applications and packet rates. In this case, we consider a line rate scenario where different packet sizes result in different packet rates,

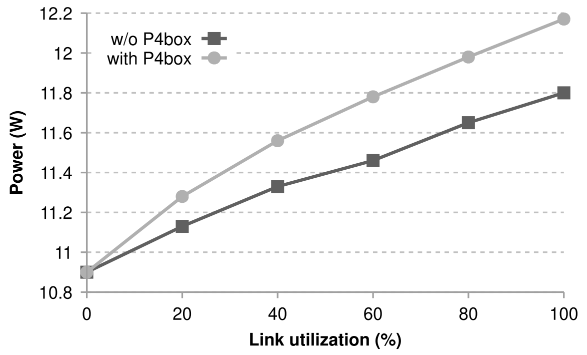

Fig. 21. Average SmartNIC power consumption for different link utilizations. Standard deviation is less than 0.1W.

but do not affect the link utilization (always 100%) - similarly to the analysis performed in Section VII-C. Table III shows that P4box increases power consumption less than 0.5W for all applications. Interestingly, the overhead is smaller for higher packet rates. We believe this is because of the increased packet processing demand, which keeps more processing units (called Micro Engines - MEs in Netronome ASICs [22]) active/occupied along time for both approaches (i.e., with and without P4box).

E. Scalability

The SmartNICs (Netronome NFP 4000) available to the experiments did not have enough resources to run switch.p4,[10] so we could not directly assess the impact of P4box monitors on the most complex existing data plane program. Nevertheless, we show in the following that monitors have low impact, by looking at the amount of operations they perform [23]. Table IV shows the P4box performance overhead for enforcing different properties compared to switch.p4. Column *field writes* corresponds to operations such as adding and removing headers as well as field assignments in actions. Also, we use variables to indicate parameters that can be adjusted when enforcing each property. For example, header protection requires one field write for saving the state of each protected header (see lines 5-8 in Figure 5), in which case we represent the number of protected headers as m. This number may change from program to program. Other variables include the number of header validity checks for enforcing well-formedness, n, and the size of the labels attached to packets for enforcing waypointing, p.

To put the numbers from Table IV into perspective, switch.p4 has over 6K lines of code and, to process a traditional IPv4 packet, requires parsing 384 bits and applying 40 tables. In order to enforce waypointing, P4box requires parsing only 8 bits (assuming $p = 8$) and applying 3 tables which are specified in 80 lines of code. In terms of resource consumption, this represents less than 3% additional memory blocks, flip-flops and lookup tables, according to the literature [24], if we consider a NetFPGA-based switch (assuming key sizes of 72 bits and a hash-based associative memory implementation).

[10]https://github.com/p4lang/switch/

TABLE III

AVERAGE POWER CONSUMPTION (IN WATTS) AT LINE RATE FOR DIFFERENT APPLICATIONS. STANDARD DEVIATION IS LESS THAN 0.1W

Application	Packet size / rate					
	64 bytes / 14 Mpps			1500 bytes / 900 Kpps		
	w/o P4box	with P4box	%	w/o P4box	with P4box	%
Load balancing	12.79	12.92	+1.01	11.25	11.25	0
Surveillance protection	12.70	12.74	+0.31	11.21	11.32	+0.98
DDoS detection	12.63	12.71	+0.63	11.21	11.77	+4.99

TABLE IV

P4box PERFORMANCE OVERHEAD COMPARED TO SWITCH.P4. n = #CHECKS, m = #PROTECTED HEADERS, p = LABEL SIZE

Property	#Parsed bits	#Tables	#Field writes	#Lines of code
Well formedness (Sec. IV-A1)	0	0	1	$n+4$
Header protection (Sec. IV-A2)	0	0	m	$2m+12$
Waypointing (Sec. IV-B1)	p	3	5	80
Traffic locality (Sec. IV-B2)	0	1	2	25
switch.p4 - IPv4	384	40	≈ 50	$\approx 6K$

VIII. DISCUSSION

Concurrency Bugs: P4box cannot detect concurrency bugs as it works at the P4 level and thus cannot control access from program blocks to shared state (the only mechanism P4 allows for that is the @atomic annotation which must be specified at compile time). One option could be creating externs to perform concurrency control and then invoke them inside monitors, but those would be target-dependent mechanisms. Regarding concurrency bugs in monitors themselves, *before* and *after* blocks are comprised of P4 constructs (e.g., actions and block statements) and thus also support @atomic annotations. However, at present our monitor verification process cannot ensure concurrency properties (e.g., safety and liveness properties). We leave both points for future investigation.

Optimizing Monitors: P4box relies on network programmers to manually optimize monitors (e.g., by eliminating duplicate operations such as storing the same packet state multiple times) in order to reduce their overhead. However, it would be feasible to automate such optimizations (or at least part of them) by using techniques such as static analysis [25] and program profiling [26]. This would however introduce new challenges in terms of ensuring semantic equivalence among original and transformed monitors, whose solution we leave as future work. Code optimization techniques could also be used to offload parts of the P4 program (or a monitor) to the control plane when dealing with resource-constrained targets.

Enforcing Other Properties: Overall, monitors have the same expressiveness as a P4 program, since both are comprised of P4 blocks. This means it would be possible, for example, to enforce stateful properties using registers or quantitative ones by using monitors to implement rate limiters. Some properties may be more difficult to express/enforce than others. For instance, enforcing network isolation can be as easy as checking an output port, while enforcing reachability among end hosts may require deploying a whole routing policy inside the monitor space. Expressing low level-properties (e.g., basic safety checks or platform-dependent properties) can be particularly challenging as P4 does not have much control of a target's operation. One option could be implementing externs to check this kind of property in a particular target and then invoking them inside monitors. We leave a detailed investigation on how to use P4box to enforce other relevant properties as future work.

Mining Monitor Specifications: The time needed for specifying properties using P4box actually depends on the expertise of the network programmer as well as the number and kind of properties she wants to specify. In this sense, we believe P4box can greatly reduce specification costs compared to other tools (e.g., theorem provers) as it allows programmers to express properties using a toolset they are already familiar with (i.e., the P4 language). Moreover, programmers can create libraries of monitors that might be reused for different programs depending on the enforced property (e.g., isolation among end hosts does not require any knowledge about the underlying packet processing logic). Even when specifying a property does require advanced knowledge about the P4 program (e.g., in a semantic property), there exist alternatives for automatically mining specifications from network configurations/programs (e.g., [27], [28]) that could be used to synthesize monitors, but this investigation is out of scope.

IX. RELATED WORK

Network Verification: Many tools have been proposed for verifying that a network behaves as expected. Moreover, these tools focus on either the control or the data plane. Minesweeper [6], Tiramisu [29] and Plankton [15] use models of networking protocols (e.g., BGP and OSPF) to analyze the network control plane. Although they can check multiple data plane configurations with this approach (i.e., the ones resulting from different protocol interactions), they are either restricted to a limited number of protocols or require long times for verifying large networks. Veriflow [30], APKeep [31], NoD [7] and SymNet [32], on the other hand, are data plane verifiers. They take a single data plane configuration (i.e., set of forwarding rules) as input, and check whether certain properties hold for all possible packets. Data plane verification approaches are typically not tied to any specific protocol, but network programmers need to manually build a separate model for each data plane program, which may be a cumbersome and error prone task.

P4v [11] and ASSERT-P4 [5] can automatically verify P4 programs, but they are able to check only program-specific properties. Vera [4], P4Nod [8] and P4K [33] create models for data plane programs that can be used as input to SymNet, NoD and the K framework, respectively. Although they can quickly verify small data plane programs (i.e., in the order of seconds), the verification time grows exponentially with both the program and the network size. bf4 [34] combines static verification, code changes and runtime checks, using static analysis to create a set of possible bugs and attempting to find predicates that, when applied to table rules inserted by a SDN controller, makes bugs in question unreachable. Finally, p4pktgen [35] and p4rl [36] generates test packets for P4 programs. As for P4box, they can detect runtime bugs that are hard to find using static analysis techniques. However, these approaches are focused on a single data plane program.

Network Debugging: Another dynamic approach to ensure security and correctness properties in networks is debugging. This approach is essentially based on monitoring and collecting statistics from network devices to perform an offline analysis. For example, Marple [37] proposes a query language for specifying monitoring tasks. Stroboscope [38] extends this idea and also considers scheduling to meet resource constraints. KeySight [39] aggregates packets with identical behaviors and generates one "postcard" per behavior. PacketScope [40] allows queries to affect packets inside the data plane, helping with debugging. In [41] authors present a data-plane primitive which encodes metadata in packets to track their path. Instead of monitoring and collecting data, P4box processes information embedded on packets in switches at runtime. This design enables our mechanism to promptly react to property violations, containing them before they compromise a network policy. In-band Network Telemetry (INT)[11] provides flexibility similar to ours. However, it assumes information embedded on packets cannot be compromised by buggy or malicious data plane programs. P4box, on the other hand, creates an isolated environment that can be used by network programmers to securely enforce policies of interest.

Runtime Enforcement: The idea of using runtime monitors to enforce properties was first introduced by [42] in the context of system security more than forty years ago. In computer networks, FlowTags is a seminal work that proposed to extend middleboxes to add tags on packets which would be used by switches to enforce path conformance and origin binding [43]. However, unlike P4box, it does not take data plane programs and all possible bugs that come with them into account.

X. Conclusion

P4 and programmable data planes lowered the barrier for innovation in networking, but at the same time also made networks more prone to bugs and misconfigurations. To solve this problem we proposed P4box, a system for dynamically enforcing properties in programmable data planes through runtime monitors. P4box can enforce both program and network-wide properties while requiring a small effort from network programmers. Moreover, it represents a small overhead to network devices in terms of latency, throughput and power consumption.

[11]https://p4.org/assets/INT-current-spec.pdf

References

[1] P. Bosshart *et al.*, "P4: Programming protocol-independent packet processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, pp. 87–95, Jul. 2014.

[2] X. Jin *et al.*, "NetCache: Balancing key-value stores with fast in-network caching," in *Proc. 26th Symp. Operating Syst. Princ.*, Oct. 2017, pp. 121–136.

[3] X. Jin *et al.*, "Netchain: Scale-free sub-RTT coordination," in *Proc. 15th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2018, pp. 35–49.

[4] R. Stoenescu, D. Dumitrescu, M. Popovici, L. Negreanu, and C. Raiciu, "Debugging P4 programs with vera," in *Proc. Conf. ACM Special Interest Group Data Commun.*, Aug. 2018, pp. 518–532.

[5] M. Neves, L. Freire, A. Schaeffer-Filho, and M. Barcellos, "Verification of P4 programs in feasible time using assertions," in *Proc. 14th Int. Conf. Emerg. Netw. Exp. Technol.*, Dec. 2018, pp. 73–85.

[6] R. Beckett, A. Gupta, R. Mahajan, and D. Walker, "A general approach to network configuration verification," in *Proc. Conf. ACM Special Interest Group Data Commun.*, Aug. 2017, pp. 155–168.

[7] N. P. Lopes, N. Bjørner, P. Godefroid, K. Jayaraman, and G. Varghese, "Checking beliefs in dynamic networks," in *Proc. 12th Symp. Netw. Syst. Design Implement. (NSDI)*, 2015, pp. 499–512.

[8] N. Lopes, N. Bjorner, N. McKeown, A. Rybalchenko, D. Talayco, and G. Varghese, "Automatically verifying reachability and well-formedness in P4 networks," Microsoft Res., Cambridge, U.K., Tech. Rep. MSR-TR-2016-65, Sep. 2016.

[9] G. Kiczales *et al.*, "Aspect-oriented programming," in *Proc. Eur. Conf. Object-Oriented Program. (ECOOP)*, 1997, pp. 220–242.

[10] M. Neves, B. Huffaker, K. Levchenko, and M. Barcellos, "Dynamic property enforcement in programmable data planes," in *Proc. IFIP Netw. Conf. (IFIP Netw.)*, May 2019, pp. 1–9.

[11] J. Liu *et al.*, "P4V: Practical verification for programmable data planes," in *Proc. Conf. ACM Special Interest Group Data Commun.*, Aug. 2018, pp. 490–503.

[12] A. Zaostrovnykh, S. Pirelli, L. Pedrosa, K. Argyraki, and G. Candea, "A formally verified NAT," in *Proc. Conf. ACM Special Interest Group Data Commun.*, Aug. 2017, pp. 141–154.

[13] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proc. USENIX OSDI*, 2008, pp. 209–224.

[14] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Comput. Surv.*, vol. 51, no. 3, pp. 1–39, Jul. 2018.

[15] S. Prabhu, K. Y. Chou, A. Kheradmand, B. Godfrey, and M. Caesar, "Plankton: Scalable network configuration verification through model checking," in *Proc. 17th Symp. Netw. Syst. Design Implement. (NSDI)*, Santa Clara, CA, USA, 2020, pp. 953–967.

[16] P. Emmerich, S. Gallenmÿller, D. Raumer, F. Wohlfart, and G. Carle, "MoonGen: A scriptable high-speed packet generator," in *Proc. Internet Meas. Conf.*, Oct. 2015, pp. 275–287.

[17] P4 Consortium. (2018). *Simple Router*. [Online]. Available: https://github.com/p4lang/p4app/tree/master/examples/simple_router.p4app

[18] S. Shi, C. Qian, Y. Yu, X. Li, Y. Zhang, and X. Li, "Concury: A fast and light-weighted software load balancer," 2019, *arXiv:1908.01889*. [Online]. Available: http://arxiv.org/abs/1908.01889

[19] A. C. Lapolli, J. A. Marques, and L. P. Gaspary, "Offloading real-time DDoS attack detection to programmable data planes," in *Proc. IFIP/IEEE Symp. Integr. Netw. Service Manage. (IM)*, Apr. 2019, pp. 19–27.

[20] T. Datta, N. Feamster, J. Rexford, and L. Wang, "SPINE: Surveillance protection in the network elements," in *Proc. USENIX Workshop Free Open Commun. Internet (FOCI)*, 2019, pp. 1–7.

[21] B. Stephens, A. Akella, and M. M. Swift, "Your programmable NIC should be a programmable switch," in *Proc. 17th ACM Workshop Hot Topics Netw.*, Nov. 2018, pp. 36–42.

[22] Netronome. (2014). *The Joy of Micro-C*. [Online]. Available: https://open-nfp.org/m/documents/the-joy-of-micro-c_fcjSfra.pdf

[23] H. Harkous, M. Jarschel, M. He, R. Priest, and W. Kellerer, "Towards understanding the performance of P4 programmable hardware," in *Proc. ACM/IEEE Symp. Archit. Netw. Commun. Syst. (ANCS)*, Sep. 2019, pp. 1–6.

[24] H. Wang *et al.*, "P4FPGA: A rapid prototyping framework for P4," in *Proc. Symp. SDN Res.*, Apr. 2017, pp. 122–135.

[25] H. Soni, M. Rifai, P. Kumar, R. Doenges, and N. Foster, "Composing dataplane programs with μP4," in *Proc. Annu. Conf. ACM Special Interest Group Data Commun. Appl., Technol., Archit., Protocols Comput. Commun.*, Jul. 2020, pp. 329–343.

[26] P. Wintermeyer, M. Apostolaki, A. Dietmüller, and L. Vanbever, "P2GO: P4 profile-guided optimizations," in *Proc. 19th ACM Workshop Hot Topics Netw.*, Nov. 2020, pp. 146–152.

[27] R. Birkner, D. Drachsler-Cohen, L. Vanbever, and M. Vechev, "Config2Spec: Mining network specifications from network configurations," in *Proc. 17th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2020, pp. 969–984.

[28] A. Kheradmand, "Automatic inference of high-level network intents by mining forwarding patterns," in *Proc. Symp. SDN Res.*, Mar. 2020, pp. 27–33.

[29] A. Abhashkumar, A. Gember-Jacobson, and A. Akella, "Tiramisu: Fast multilayer network verification," in *Proc. 17th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, Santa Clara, CA, USA, 2020, pp. 201–219.

[30] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey, "Veriflow: Verifying network-wide invariants in real time," in *Proc. 10th USENIX Symp. Netw. Syst. Design Implement.(NSDI)*, 2013, pp. 15–27.

[31] P. Zhang, X. Liu, H. Yang, N. Kang, Z. Gu, and H. Li, "APKeep: Realtime verification for real networks," in *Proc. 17th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, Santa Clara, CA, USA, 2020, pp. 241–255.

[32] R. Stoenescu, M. Popovici, L. Negreanu, and C. Raiciu, "SymNet: Scalable symbolic execution for modern networks," in *Proc. ACM SIGCOMM Conf.*, Aug. 2016, pp. 314–327.

[33] A. Kheradmand and G. Rosu, "P4K: A formal semantics of P4 and applications," 2018, *arXiv:1804.01468*. [Online]. Available: http://arxiv.org/abs/1804.01468

[34] D. Dumitrescu, R. Stoenescu, L. Negreanu, and C. Raiciu, "Bf4: Towards bug-free P4 programs," in *Proc. Annu. Conf. ACM Special Interest Group Data Commun. Appl., Technol., Archit., Protocols Comput. Commun.*, Jul. 2020, pp. 571–585.

[35] A. Nötzli, J. Khan, A. Fingerhut, C. Barrett, and P. Athanas, "P4pktgen: Automated test case generation for P4 programs," in *Proc. Symp. SDN Res.*, Mar. 2018, pp. 1–7.

[36] A. Shukla, K. N. Hudemann, A. Hecker, and S. Schmid, "Runtime verification of P4 switches with reinforcement learning," in *Proc. Workshop Netw. Meets AI ML (NetAI)*, 2019, pp. 1–7.

[37] S. Narayana *et al.*, "Language-directed hardware design for network performance monitoring," in *Proc. Conf. ACM Special Interest Group Data Commun.*, Aug. 2017, pp. 85–98.

[38] O. Tilmans, T. Bühler, I. Poese, S. Vissicchio, and L. Vanbever, "Stroboscope: Declarative network monitoring on a budget," in *Proc. 15th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2018, pp. 467–482.

[39] Y. Zhou *et al.*, "KeySight: Troubleshooting programmable switches via scalable high-coverage behavior tracking," in *Proc. IEEE 26th Int. Conf. Netw. Protocols (ICNP)*, Sep. 2018, pp. 291–301.

[40] R. Teixeira, R. Harrison, A. Gupta, and J. Rexford, "PacketScope: Monitoring the packet lifecycle inside a switch," in *Proc. Symp. SDN Res.*, Mar. 2020, pp. 76–82.

[41] S. Kodeswaran, M. T. Arashloo, P. Tammana, and J. Rexford, "Tracking P4 program execution in the data plane," in *Proc. Symp. SDN Res.*, Mar. 2020, pp. 117–122.

[42] J. P. Anderson, "Computer security technology planning study," Air Force Electron. Syst. Division, Bedford, MA, USA, Tech. Rep. ESD-TR-73-51, 1972.

[43] S. K. Fayazbakhsh, L. Chiang, V. Sekar, M. Yu, and J. C. Mogul, "Enforcing network-wide policies in the presence of dynamic middlebox actions using flowtags," in *Proc. 11th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2014, pp. 543–546.

Miguel Neves received the Ph.D. degree from the Federal University of Rio Grande do Sul (UFRGS), Brazil, in 2020. He is currently a Postdoctoral Researcher at Dalhousie University, Canada. His research interests include interplay of program analysis, networking, security, and distributed systems.

Bradley Huffaker received the master's degree in mathematics and computer science from UCSD. Since 2015, he has been working as a Senior Research Programmer with the Center for Applied Internet Data Analysis (CAIDA), UC San Diego. His research interests include visualization, DNS, geolocation, and Internet topology.

Kirill Levchenko (Member, IEEE) received the B.A. degree in mathematics and computer science from the University of Illinois at Urbana-Champaign, in 2001, and the Ph.D. degree from the University of California, San Diego, in 2008. He is currently an Associate Professor with the University of Illinois at Urbana-Champaign. His research interests include evidence-based techniques to a broad range of computer and network security domains.

Marinho Barcellos was an Associate Professor with the Federal University of Rio Grande do Sul (UFRGS), from 2010 to 2019. Since October 2019, he has been with the University of Waikato, New Zealand. He has contributed with research in a wide range of topics, including programmable data planes, network security, and Internet measurements. As for public service, he is currently a member of the SIGCOMM Executive Committee, a Co-Chair of its CARES Committee, and a member of CoNEXT and PAM steering committees.