Dynamic Property Enforcement in Programmable Data Planes

Miguel Neves*, Bradley Huffaker†, Kirill Levchenko‡ and Marinho Barcellos*
UFRGS*, CAIDA/UCSD†, UIUC‡

***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 typically require programmers to manually model P4 programs, are limited in terms of the properties they can guarantee and 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. Our results show that *P4box* allows programmers to easily express a broad range of properties. Moreover, we demonstrate that runtime monitors represent a small overhead to network devices in terms of latency and resource consumption.**

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].

ISBN 978-3-903176-16-4

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-B3).

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 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 emition of labels on packets (see an example in Section IV-B1). 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 show that monitors impose low overhead to network devices in terms of latency and memory consumption (Section V).

The remaining of this paper is organized as follows. Section II reviews the architecture of programmable network devices, summarize the main aspects of P4 programs, and motivates the development of property enforcement mechanisms in programmable data planes. Section VI discusses key aspects of runtime enforcement, P4box and program monitors. Section VII compares our proposal with related work, and finally Section VIII concludes the paper.

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*.

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

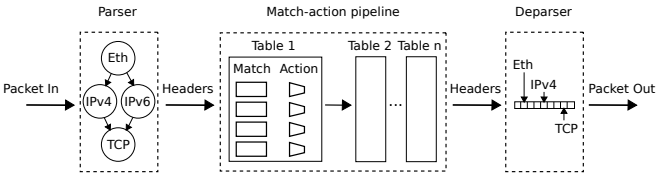

Fig. 1. Example of PISA-based switch. Dashed blocks can be programmed in P4.

```
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 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.

[2]https://github.com/p4lang/switch/issues/97
[3]https://github.com/p4lang/switch/pull/102

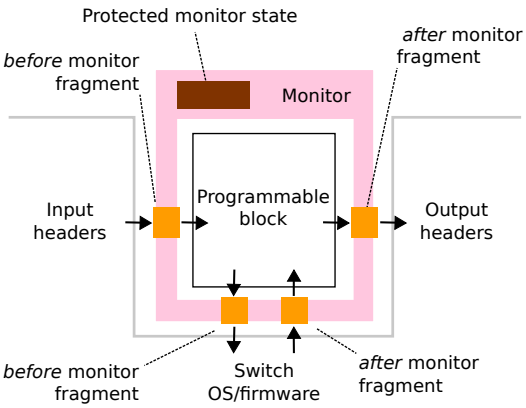

Fig. 3. P4box programming model.

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), describe the three kinds of monitors P4box can deploy in detail (Sections III-B, III-C, and III-D) and present our prototype implementation (Section III-E).

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 block. A monitor can thus modify the apparent behavior of a 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:

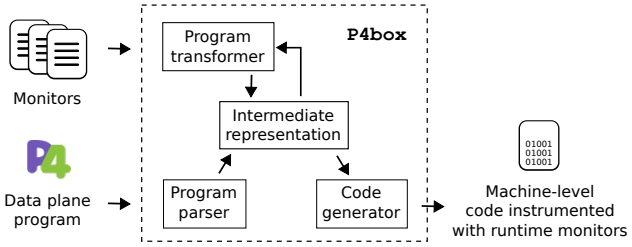

Fig. 4. P4box workflow.

```
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. Finally, they can also contain local declarations (e.g., actions, tables) visible inside the monitor but not the monitored block.

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 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.

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 (e.g., because there is no parameter in the monitored block that supports the merge operation), a message is emitted and the instrumentation process is aborted; second, *before* and *after* blocks as well as local declarations are inserted in the monitored block; finally, a name resolution 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.

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

```
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.

bits that for some reason (e.g., confidentiality) should not be visible to the data plane program.

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.

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. Extern monitors are useful to mediate how the data plane program interacts with the platform underlying it.

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.

```
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.

```
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.

E. Implementation

We implemented a prototype of P4box by extending the $P4_{16}$ reference compiler[4]. Our system has around 1.5K lines of C++ code and is publicly available[5]. 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.

[4]https://github.com/p4lang/p4c

[5]https://github.com/mcnevesinf/p4box

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 regardless of how the device is configured or connected in a topology. They are also referred to as *network function properties* in the literature [10]. 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: The output of a data plane program is *well-formed* if it complies with relevant protocol standards. *Well-formedness* determines the interoperability between multiple implementations of a protocol stack. In terms of programmable data planes, this means that the packets produced by one data plane program can be processed by another, and vice-versa. 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.

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 an 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 [10]. These properties may involve basic predicates (e.g., A can reach B) as well as state and quantities (e.g., express desired behaviors

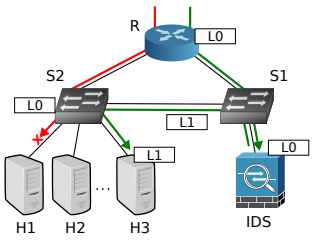

Fig. 9. Example topology for waypointing.

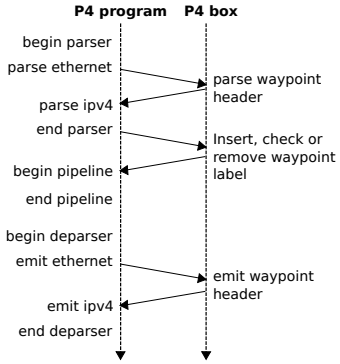

Fig. 10. Interaction between P4box and the P4 program to enforce waypointing.

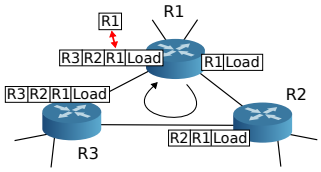

Fig. 12. Example topology for loop detection.

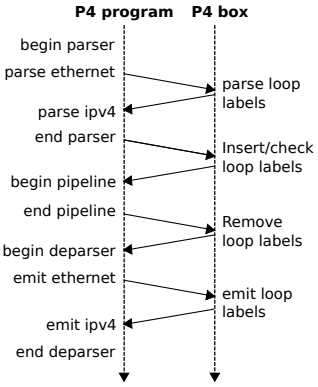

Fig. 13. Interaction between P4box and the P4 program to enforce loop detection.

```
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   before {
32     //Enforce waypointing property
33     insert_label.apply();
34     check_waypoint.apply();
35     remove_label.apply();
36 }}
```

Fig. 11. Supervisor to enforce waypointing.

```
1  struct p4boxState {
2    ...
3    //Header stack to store sequence of labels
3    loop_header_t[10] loopHeader;
4  }
5
6  monitor loopMonitor(inout p4boxState pstate)
7                                     on Pipeline{
8    ...
9    action loop_detected(){ ... }
10   action insert_label( bit<32> label ){ ... }
11
12   /*Check if sequence of labels in a packet
13     contains router ID (i.e., has a loop)*/
14   table check_loop {
15     actions = { insert_label; loop_detected; }
16     key = {
17       pstate.loopHeader[0].label : ternary;
18       ...
19       pstate.loopHeader[9].label : ternary;
20     }
21     size = 10;
22   }
23
24   before {
25     check_loop.apply();
26   }
27 }
```

Fig. 14. Supervisor to detect forwarding loops.

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.

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) Loop detection: P4box can also detect forwarding loops by adding labels to packets. However, unlike waypointing, it appends a new label rather than updating a single one whenever the packet traverses a different hop. Figure 12 illustrates this idea, where labels contain router IDs. To detect a loop, a P4box monitor compares the sequence of labels already in the packet with the new one. If there is a match, then a loop is identified. Figure 13 shows the interaction between P4box and the P4 program in order to enforce loop detection. Similar to waypointing, P4box first hooks the program parser in order to extract the sequence of labels attached to the packet. However, two (rather than one) traps are needed during the match-action processing, one before and another after the pipeline. The former ensures the device does not waste time processing a packet that is in a loop and will be discarded anyway, while the latter is used to guarantee that the labels are only removed after a packet gets its output port in the last hop.

Figure 14 summarizes monitors for enforcing loop detection. Parser and extern monitors, which are used to extract and emit the sequence of labels, are omitted due to space constraints. Moreover, the sequence of labels is manipulated using a header stack (l.3). A control block monitor contains the match-action tables to check, insert and remove labels (l.6-27). Entries to these tables place the router ID in each position of the stack in order to detect a loop.

3) 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 15 are not allowed to be forwarded to upper ports. Figure 16 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 17 shows

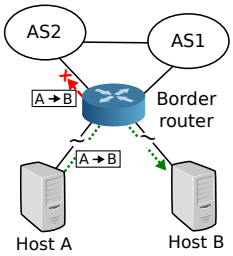

Fig. 15. Example topology for traffic locality.

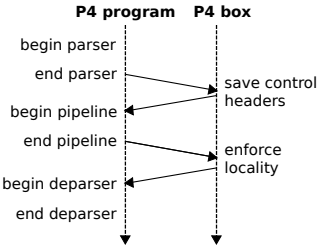

Fig. 16. Interaction between P4box and the P4 program to enforce traffic locality.

```
1   monitor tlMonitor(inout p4boxState pstate)
2                                    on Pipeline {
3     //Run enforcement action
4     action enforce_locality(){ ... }
5
6     //Check if packet violates locality
7     //(i.e., tries to leave AS)
8     table traffic_locality_table {
9       actions = { NoAction; enforce_locality; }
10      key = {
11        hdr.ipv4.srcAddr : ternary;
12        hdr.ipv4.dstAddr : ternary;
13        standard_metadata.egress_port : exact;
14      }
15      size = 512;
16    }
17
18    after { traffic_locality_table.apply(); }
19  }
```

Fig. 17. Supervisor to enforce traffic locality.

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).

V. Performance

Because dynamic enforcement happens at run time, it may impose a performance penalty compared with static verification techniques. In this section, we analyze the performance overhead of P4box in terms of logical resources (i.e., tables, actions, headers) required for enforcing each property. We favor this kind of evaluation in a preliminary analysis because these metrics are target-independent and thus can be used to estimate the overhead for different types of network devices (e.g., hardware and software switches, SmartNICs and NetFPGAs). Moreover, they are not associated with any specific data plane program running on these devices, which could affect metrics such as latency and throughput. Overall, the higher the number of logical units in a P4 program, the higher the overhead in the data plane. For example, packet parsing latency increases with the number of headers (or bits) to be extracted, and a match-action stage takes longer to process a packet if we increase the number of tables or the complexity of the actions to be performed.

Table I summarizes the overhead of P4box for enforcing the properties described in Section IV. The column *key size*

TABLE I
P4BOX PERFORMANCE OVERHEAD. n = #CHECKS, m = #PROTECTED HEADERS, p = LABEL SIZE, q = #LABELS, s = LENGTH OF CONTROL FIELDS

Property	#Parsed bits	#Tables	Key size (bits)	#Field writes	#Lines of code
Well formedness (Sec. IV-A1)	0	0	0	1	$n+4$
Header protection (Sec. IV-A2)	0	0	0	m	$2m+12$
Waypointing (Sec. IV-B1)	p	3	$p+s$	5	80
Loop detection (Sec. IV-B2)	qp	3	qp	$4q$	$5q+80$
Traffic locality (Sec. IV-B3)	0	1	s	2	25
switch.p4 - IPv4	384	40	280	≈ 50	$\approx 6K$

reflects the size of the largest matching key when multiple tables are applied, and the column *field writes* corresponds to operations such as adding and removing headers as well as field assignments in actions. 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, the size of the labels attached to packets for enforcing waypointing and loop detection, p, the maximum amount of labels, q, and the total length (in bits) of the fields used to control the operation of a monitor (e.g., IP addresses in traffic locality), s.

To put the numbers from Table I in perspective, we compare them with switch.p4[6], a widespread data plane program that implements a top-of-rack switch for data centers. Switch.p4 has more than 6K lines of code, and requires parsing 384 bits and applying 40 tables to process a traditional IPv4 packet. In order to enforce waypointing for example, P4box requires parsing only 8 bits (assuming $p = 8$) and applying 3 tables which are specified in 80 lines of code. In practice, this represents an increase lower than 5% in the packet processing latency according to the experiments we performed in a software switch[7]. Regarding resource consumption, if we consider hardware-based devices such as NetFPGAs, waypointing requires less than 3% additional memory blocks, flip-flops and lookup tables according to the literature [11] (assuming key sizes of 72 bits and a hash-based associative memory implementation).

In our ongoing work, we are investigating optimizations for enforcing each property (e.g., combining tables among them) in order to reduce even more these overheads. Moreover, P4box could benefit from parallelizations available in network devices to process monitors concurrently [12]. We plan to extend the evaluation for including measurements performed on high-performance P4-enabled devices (e.g., SmartNICs and NetFPGAs) as a future work.

VI. DISCUSSION

Monitor correctness. Although monitors can also contain bugs themselves, that is less likely to happen compared to the original program due to their intentionally small code base. Moreover, their simplicity makes them suitable to formal analysis (e.g., model checking or theorem proving) when security or reliability are important. In our ongoing work, we are exploring automatically converting monitors into an equivalent model in C and using an off-the-shelf symbolic execution engine (e.g., KLEE [13]) to prove their correctness.

High-level abstractions. While P4box allows programmers to use P4 for specifying properties, it is still necessary to think about each monitor individually. For example, programmers may need to create multiple monitors to enforce a network-wide property (e.g., a monitor for inserting and other for removing a label from packets). This can easily become a tedious process in large networks containing thousands of devices. Recent research efforts have proposed to automatically synthesize network configurations from higher-level abstractions (e.g., graphs or intents) [14]. We plan to extend P4box to support these abstractions in order to facilitate the enforcement of more complex properties or their combination.

VII. 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. ERA [15] and Minesweeper [6] 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 restricted to a limited number of protocols. Veriflow [16], NoD [7] and SymNet [17], 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 [18] and ASSERT-P4 [5] can automatically verify P4 programs, but they are able to check only program-specific properties. Finally, Vera [4] and P4Nod [8] create models for data plane programs that can be used as input to SymNet and NoD, 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.

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

[7]https://github.com/p4lang/behavioral-model

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 [19] proposes a query language for specifying monitoring tasks. Stroboscope [20] extends this idea and also considers scheduling to meet resource constraints. 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)[8] provides flexibility similar to ours. However, it assumes information embedded on packets can not 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 [21] 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 [22]. However, unlike P4box, it does not take data plane programs and all possible bugs that come with them into account.

VIII. 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 modest overhead to network devices in terms of latency and memory consumption. As future work, we plan to combine static verification and dynamic enforcement to build efficient, correct-by-construction programmable data planes.

Acknowledgments. This work has been supported by grants from NSF (CNS-1740911), RNP/CTIC (P4Sec), CNPq (140317/2017-1), and also by CAPES/Brazil – Finance Code 001.

References

[1] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming Protocol-independent Packet Processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, Jul. 2014.

[2] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica, "Netcache: Balancing key-value stores with fast in-network caching," in *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, 2017, pp. 121–136.

[3] X. Jin, X. Li, H. Zhang, N. Foster, J. Lee, R. Soulé, C. Kim, and I. Stoica, "Netchain: Scale-free sub-rtt coordination," in *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2018, pp. 35–49.

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

[4] R. Stoenescu, D. Dumitrescu, M. Popovici, L. Negreanu, and C. Raiciu, "Debugging p4 programs with vera," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 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 *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies (CoNEXT)*, 2018, pp. 73–85.

[6] R. Beckett, A. Gupta, R. Mahajan, and D. Walker, "A general approach to network configuration verification," in *Proceedings of the ACM SIGCOMM Conference*, 2017, pp. 155–168.

[7] N. P. Lopes, N. Bjørner, P. Godefroid, K. Jayaraman, and G. Varghese, "Checking beliefs in dynamic networks," in *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (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," Tech. Rep., September 2016.

[9] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-Oriented Programming," in *Proceedings of the European Conference on Object-Oriented Programming*, 1997, pp. 220–242.

[10] A. Zaostrovnykh, S. Pirelli, L. Pedrosa, K. Argyraki, and G. Candea, "A formally verified nat," in *Proceedings of the ACM SIGCOMM Conference*, 2017, pp. 141–154.

[11] H. Wang, R. Soulé, H. T. Dang, K. S. Lee, V. Shrivastav, N. Foster, and H. Weatherspoon, "P4fpga: A rapid prototyping framework for p4," in *Proceedings of the ACM Symposium on SDN Research (SOSR)*, 2017, pp. 122–135.

[12] L. Jose, L. Yan, G. Varghese, and N. McKeown, "Compiling packet programs to reconfigurable switches," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, 2015, pp. 103–115.

[13] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *8th USENIX Conference on Operating Systems Design and Implementation (OSDI 08)*, 2008, pp. 209–224.

[14] A. Abhashkumar, J.-M. Kang, S. Banerjee, A. Akella, Y. Zhang, and W. Wu, "Supporting diverse dynamic intent-based policies using janus," in *Proceedings of the International Conference on Emerging Networking EXperiments and Technologies (CoNEXT)*, 2017, pp. 296–309.

[15] S. K. Fayaz, T. Sharma, A. Fogel, R. Mahajan, T. Millstein, V. Sekar, and G. Varghese, "Efficient network reachability analysis using a succinct control plane representation," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 217–232.

[16] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey, "Veriflow: Verifying network-wide invariants in real time," in *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013, pp. 15–27.

[17] R. Stoenescu, M. Popovici, L. Negreanu, and C. Raiciu, "Symnet: Scalable symbolic execution for modern networks," in *Proceedings of the ACM SIGCOMM Conference*, 2016, pp. 314–327.

[18] 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 *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2018, pp. 490–503.

[19] S. Narayana, A. Sivaraman, V. Nathan, P. Goyal, V. Arun, M. Alizadeh, V. Jeyakumar, and C. Kim, "Language-directed hardware design for network performance monitoring," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '17, 2017, pp. 85–98.

[20] O. Tilmans, T. Bühler, I. Poese, S. Vissicchio, and L. Vanbever, "Stroboscope: Declarative network monitoring on a budget," in *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2018, pp. 467–482.

[21] J. P. Anderson, "Computer security technology planning study," Air Force Electronic Systems Division, Tech. Rep., 1972.

[22] 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 *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2014, pp. 543–546.