DBVal: Validating P4 Data Plane Runtime Behavior

K Shiv Kumar
IIT Hyderabad, India

Ranjitha K
IIT Hyderabad, India

P S Prashanth
IIT Hyderabad, India

Mina Tahmasbi Arashloo
Cornell University, USA

Venkanna U.
IIIT-NR, India

Praveen Tammana
IIT Hyderabad, India

ABSTRACT

The P4 software ecosystem to operate programmable data planes is increasingly becoming complex. The packet-processing behavior is defined by several components: the P4 program, the compiler that maps P4 programs to resource-constrained switch pipeline, the control-plane program that installs rules, and the switch software agents that configure the data plane. Bugs in any one or more of these components would potentially introduce packet-processing errors in the data plane. Prior work verifies P4 programs before deployment and found many program bugs. But bugs can happen in other components after the program deployment and may not be found during testing and only manifest themselves in production.

In this work, our goal is to detect packet-processing errors induced by bugs that are not caught (or are difficult to catch) before the P4 program deployment. Our key idea is to let P4 programmers specify the intended packet-processing behavior and validate the actual packet-processing behavior against the intended behavior at runtime. We obtain intended behavior from the P4 programmers in the form of assertions, where each assertion specifies which tables and actions should be applied and in what order on a certain subset of traffic. Next, the assertions are compiled and translated to P4 implementation such that the implementation efficiently tracks the packet execution path, that is, the set of tables applied and actions executed, and then validates the tracked behavior at line rate. We show that our techniques can be used to effectively detect bugs that are difficult, if not impossible, to catch with existing techniques for testing and verifying programmable data planes.

CCS CONCEPTS

• **Networks** → **In-network processing**; **Programmable networks**; **Middle boxes / network appliances**.

KEYWORDS

Software-Defined Networks, Programmable Data Planes, Runtime Validation, Bug Detection

ACM Reference Format:
K Shiv Kumar, Ranjitha K, P S Prashanth, Mina Tahmasbi Arashloo, Venkanna U., and Praveen Tammana. 2021. DBVal: Validating P4 Data Plane Runtime Behavior. In *The ACM SIGCOMM Symposium on SDN Research (SOSR)*

SOSR '21, September 20–21, 2021, Virtual Event, USA

ACM ISBN 978-1-4503-9084-2/21/09…$15.00
https://doi.org/10.1145/3482898.3483352

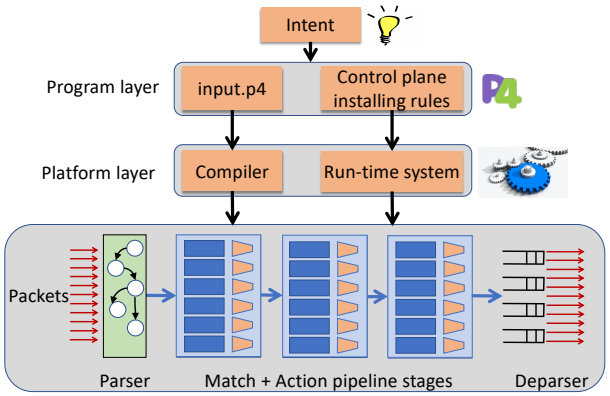


Figure 1: P4 stack

(SOSR '21), September 20–21, 2021, Virtual Event, USA. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3482898.3483352

1 INTRODUCTION

Programmable data planes allow programmers to define packet processing and develop new network functions as well as re-configure the existing functions. The functions range from traditional IP-prefix based routing to load balancing [31, 36, 42] to DDoS detection [27]. Moreover, recent works [7, 22, 37] show the performance benefits of offloading network functions from general-purpose servers to programmable data planes (*e.g.*, smartNICs [3], Intel Barefoot Tofino [17]).

To reap the benefits of programmable data planes in practice, we need to address a new set of challenges in ensuring the correctness of the underlying software and hardware ecosystem, as shown in Fig. 1. More specifically, the actual behavior of a P4 program in the data plane may mismatch with the intended behavior because of the following reasons: (1) Bugs in P4 program [29, 39, 50]; (2) Faulty match-action rules installed by the controller [26, 28]; (3) State in the data plane registers could be manipulated by adversaries by injecting crafted input packets [33, 41]; (4) Compiler bugs and switch software bugs [4, 20, 35, 46]; and (5) Hard to detect platform-dependent hardware failures [47, 48].

In brief, packet-processing errors affect how packets are processed in the data plane, which can result in violation of policies related to availability, performance, and security. This motivates the need for an ability to ensure that every packet in the switch data plane follows its intended program path, that is, the set of tables to be applied and the actions to be executed. In this paper, the key question we like to investigate is: *How can we validate the packet-processing behavior of the P4 data plane at runtime?*

> ...while network control planes are inherently complicated, a P4 data plane captures ground truth for the network—*i.e.*, how it forwards packets—and is therefore an attractive platform for deploying verification technologies. By observing and then validating behavior at the data plane level, it is possible to reduce the trusted computing base: the switch operating system, driver, and other low-level components do not need to be trusted...
>
> – *Larry Peterson* et al *[45]*

Existing works on P4 verification [28, 39, 43, 50] mostly focus on statically verifying various properties of the P4 program. They operate at the level of the P4 program, thus verify whether the software logic is bug-free for a specific set of bugs. However, they cannot be used to detect incorrect data-plane behavior due to issues at the platform layer or in the P4 pipeline, such as (a) incorrect hardware mapping for a P4 program by a compiler during compilation, especially non open-source compilers; (b) bugs in switch software; and (c) bugs in non-programmable blocks [48] of a P4 pipeline (*e.g.*, packet replication engine, buffer queuing engine).

On the other hand, automatic test packet generation approaches [44, 49] send test packets and verify the runtime behavior of a P4 program with intended behavior. This enables detecting incorrect behavior due to bugs at the platform layer or in the P4 pipeline. However, given the size of real-world P4 programs and the complex software and hardware ecosystem around them, test packets may not exercise all possible packet-processing scenarios. There are several bugs which in most scenarios happen in corner cases, manifesting themselves only after certain sequences of incoming packets with certain combinations of rules in the tables. Therefore testing prior to deployment approach may not uncover such hard-to-catch bugs that trigger at runtime.

In this work, we propose **D**ataplane **B**ehavior **V**alidator (DBVal) system that validates the actual packet-processing behavior against the intended behavior at runtime. Our approach is to let the P4 programmer tells us what is correct (expected) and use it as a reference to validate what we observed in the data plane in real time. In contrast to using test packets, we consider *every* packet that goes through the switch as a potential test packet for the data plane and validate the observed behavior. Validating the observed behavior of real packets is complementary to both the static analysis approach and the test packet generation approach; it uncovers bugs that are hard or not possible to detect by these approaches.

Designing such a system is challenging. First, doing per-packet validation at high-speed line rates (order of Tbps) is not trivial. The checks should be fast, accurate, and consume minimal data-plane resources. For example, if the validation module is placed in the switch control plane, at high line rates, the interface connecting the data plane and the control plane will become a bottleneck, thus making it infeasible to forward every packet metadata to the control plane. Instead, it is a good design choice to place the validation module in the data plane with intended behavior accessible at line rate. To do so, we design a high-level assertion language using which a P4 programmer writes assertions to define the intended behavior of incoming traffic, and the DBVal compiler automatically converts these assertions to P4 implementations that run at line rate. More specifically, each assertion specifies the tables to be applied and the actions to be executed on a certain traffic class, and the compiler automatically annotates the original P4 program such that the final annotated P4 program tracks and validates the actual packet-processing behavior at runtime (more details in Section §4.2).

Second, it is crucial to carefully design a primitive that tracks actual packet-processing behavior (*i.e.*, the sequence of tables hit and the actions applied) of every packet in a resource-constrained data plane. In our prior preliminary work [38], we developed a path tracker primitive that tracks the packet execution path in the data plane. The key idea is that Ball-Larus encoding technique, a well-known technique for profiling execution paths in software [23], is a promising fit for tracking packet execution paths in P4 programs. Since the encoding requires simple addition operations, it can be implemented on programmable switches [17]. In this paper, we build atop this primitive and integrate the primitive in the compilation process so that the DBVal automatically checks whether tracked execution path is one among the intended execution paths specified in assertions (more details in Section §4.1).

The key contributions of the paper are as follows:

- Through example scenarios, we identify the need for validating data plane runtime behavior (§2).
- We present DBVal, a system for validating the actual packet-processing behavior with the intended behavior expressed by the P4 programmers. To realize DBVal, we propose a language syntax using which P4 programmers can write assertions and express intended packet-processing behavior (§3). DBVal compiler automatically translates the assertions to corresponding P4 implementations (§4).
- We prototype DBVal for two targets: BMV2 software switch [16] and Intel Barefoot Tofino software switch model [17] (§5). The code is available at [18]. We evaluated the DBVal prototype in terms of expressiveness and data-plane resource overhead by compiling assertions written for a variety of P4 programs (§6).

2 MOTIVATING EXAMPLES

A P4 program and the rules installed in its tables determine how each packet should be processed by the switch. More specifically, they determine the path in the program's control flow the packet should take, including the tables it should hit, the rules in each table it should match, and the actions that should be executed on it. However, it is possible for a packet to take a different path than what it is supposed to: hit different tables, get matched by the wrong rules, or get processed incorrectly by actions. This can happen because of (i) bugs in the compiler that result in parts of the program getting translated incorrectly to hardware, (ii) bugs in switch software, or (iii) bugs in the hardware target itself. All the above are hidden from the programmer at the P4 program level and are hard to identify and resolve in an offline manner before deployment.

In this section, we present few examples to motivate the need for validation of data plane behavior. More specifically, we consider three types of bugs that cause a mismatch between the intended behavior and the actual behavior. First, we consider compiler bugs that causes incorrect mapping of a p4 program on a programmable

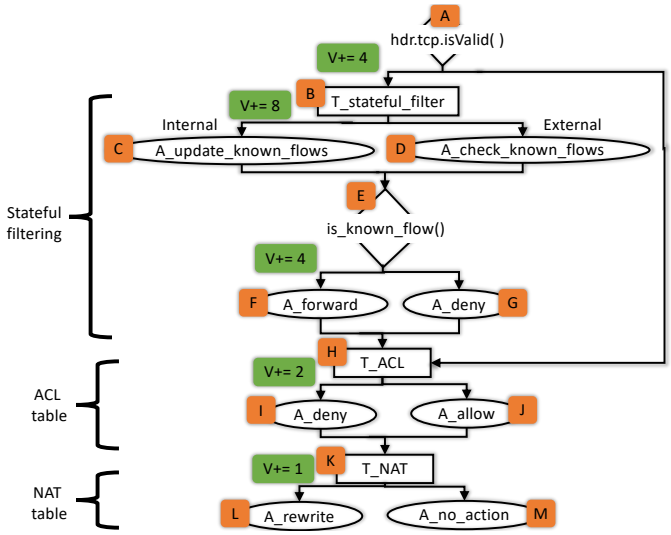
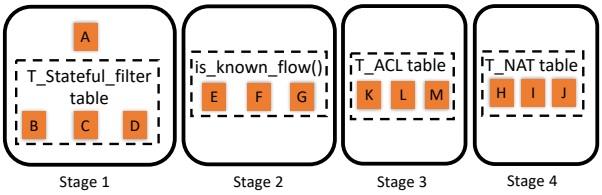


Figure 2: Stateful_firewall.p4 control flow graph

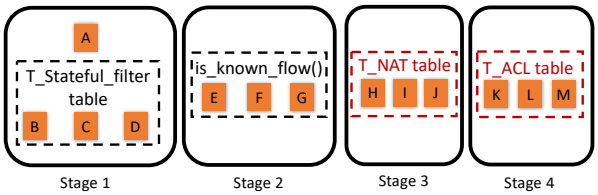
switch. Second, we consider a switch software bug causing incorrect installation of rules into tables. Before we present the details of bugs, let us consider an example P4 program, Stateful_firewall.p4, which implements a chain of network functions in the P4 data plane.

Stateful_firewall.p4. Fig. 2 shows the control flow graph (CFG) of the P4 program. The program first checks whether it has received tcp packet. If so it applies `T_stateful_filter`. `T_stateful_filter` only allows packets from an external network if they belong to a connection initiated by hosts in an internal network. The table has two actions: `A_Set_Known_Flows` action is executed for internal to external traffic and updates known flows register if the packet's syn flag is set, and `A_Check_Known_Flows` action is executed for external to internal traffic and checks whether the packet's flow is seen before. If seen, the packet will be forwarded, otherwise dropped. Next, the program applies `T_ACL` which keeps packet-filtering rules installed by control plane and executes two actions: `A_allow` to allow traffic and `A_deny` to deny traffic. Finally, `T_NAT` maps packets with private IP address space to global IP address space by executing two actions: `A_rewrite` rewrites source IP in forward direction and rewrites destination IP in reverse direction and `A_noop`.

Security policy. Consider a security policy written for a host (10.1.1.10) that stores sensitive data. The policy says the host's communication to the external network must be encrypted and the host IP address should be hidden. As shown in Fig. 3(c), ACL table has rules for only allowing encrypted traffic (*i.e.*, srcport = 443) and denying all non-encrypted web traffic (*e.g.*, srcport = any) originated from the host with IP address 10.1.1.10. To hide the host's private IP address, the NAT table has a rule that matches with the source IP address 10.1.1.10 and rewrites it to a public IP address 2.2.2.1. Now we will demonstrate possible policy violations induced by various bugs.



(a) **Correct placement of tables**



(b) **Incorrect placement of tables**

T_ACL table rules

Rule	Match	Action
R1	srcIP=10.1.1.10, srcport=443, priority=1	Allow
R2	srcIP=10.1.1.10, srcport=*, priority=2	Deny
R3	*	Deny
	⋮	

T_NAT table rules

Match	Action
srcIP=10.1.1.10	rewrite(srcIP=2.2.2.1)
⋮	

(c) **ACL table and NAT table rules**

Figure 3: Statefull_firewall tables to stage mapping and table rules

Violation due to bugs in compiler. Consider a bug in a compiler that place tables incorrectly in the data plane [4]. Fig. 3(a) shows the correct placement of the tables, whereas Fig. 3(b) shows that the order of ACL and NAT is reversed. As a result, before ACL (stage 4), NAT is applied (Stage 3). Since the NAT table rewrites the packet's source IP address with 2.2.2.1, there will be no matching rule for 2.2.2.1 in the ACL table, and hence, the encrypted packet will be denied. This demonstrates an incorrect ordering of ACL and NAT tables due to compiler bugs that can break connectivity, leading to a policy violation.

Another possible bug is related to incorrect mapping of different parsed packet header fields to the same data plane resource. Several hardware programmable switches have a fixed number of bits, called a packet header vector (PHV), that contains parsed packet headers and meta-data variables, which are passed from one stage to the next to get processed by the program. One of the responsibilities of the compiler is to map packet headers and meta-data variables defined in the program to bits in the PHV. Since PHV is a limited

resource, compilers try to use the same bits in the PHV for different packet headers whenever possible, for instance, if two packet header fields have non-overlapping lifetimes in the pipeline. While this is a crucial optimization, it can lead to non-trivial data corruption if not implemented correctly, *i.e.*, if the compiler uses the same set of bits for header fields that have overlapping lifetimes.

Suppose, in our example program, the compiler, by mistake, uses the same set of PHV bits for `dst_port` and SYN flag such that the way it assigns PHV bits causes the fourth bit of `dst_port` to overwrite the bit used for the SYN flag. If both the fields were used in the same stage, one of them could corrupt the value of another. Now, say a TCP-SYN packet comes from the internal network for which the 4th bit of the `dst_port` is zero. Then, that flow will not be saved in registers as a new flow. Because by mistake, the SYN flag has been rewritten to zero. So, the other direction of traffic will not be allowed inside, thus violates the security policy.

Violation due to bugs in the switch software. Similarly, there are other scenarios like non-deterministic communication delays between SDN controller and switches [26], or bugs in switch OS causing packets to match with wrong rules, which can lead to incorrect packet processing. More specifically, CacheFlow [35] authors observed that if the number of table rules to be installed exceeds TCAM table size, then the switch maintains the extra rules in a software agent. However, when installing rules in the TCAM, not all splits of a set of rules present at TCAM, and the agent respect the cross-rule dependencies. For example, the high priority rules are maintained by the agent, and the low priority rules are cached in TCAM. Such incorrect handling of table rules may cause a packet to hit a cached rule in the TCAM that is supposed to hit a rule in the software agent, leading to incorrect packet processing. For instance, in the running security policy example, as shown in Fig. 3(c), the user expects the ACL table to have two rules, R1 and R2, where each has a different priority. However, due to the bug, if R2 is cached in the TCAM and R1 is not, then the SSL traffic with srcport=443 will be denied, thus violates the policy.

Violation due to bugs in the P4 pipeline. Typically, a P4 pipeline can have platform-independent programmable blocks (*e.g.*, match-action tables, parser) and platform-dependent non-programmable blocks (*e.g.*, PRE, BQE). To fix issues in non-programmable blocks, the respective switch vendor has to be informed. Consider that there are two tables, longest-prefix match (LPM) and access control list (ACL), and the packets are first matched by the LPM table, and a clone decision is made. Next, the ACL table marks the packets to be dropped. A potential bug in the packet replication engine (PRE) [48] could drop the original packet, but however forward the cloned copy thus violates the policy.

To summarize, bugs at the lower layers (*i.e.*, compiler, switch software, P4 pipeline) may lead to incorrect packet-processing behavior in the data plane causing policy violations. In this work, our key focus is to check whether the actual packet-processing behavior is equivalent to the intended behavior. If not, we will detect and raise alerts.

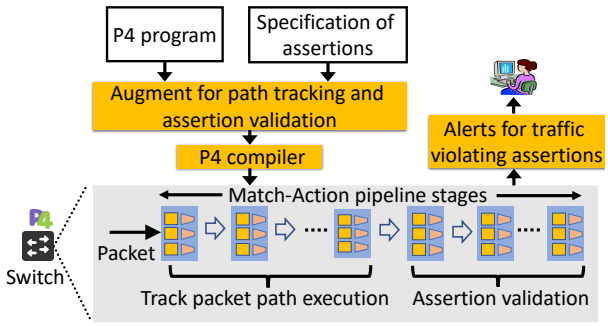


Figure 4: DBVal system architecture

Field f	**::=**	**metadata_field | packet_hdr_field**
Predicate Pred	**::=**	**f op v | f in L | Pred & Pred | Pred || Pred | !(Pred)**
Operator op	**::=**	**== | != | < | > | <= | >=**
Value v	**::=**	**int | hex | ip-address | true | false**
Const List L	**::=**	**nil | v,L**
Path P	**::=**	**.*S.***
Sequence S	**::=**	**table | action | table@action | S*S | S^S**
Assert AT	**::=**	**assert(Pred) | assert(P)**
Filter FT	**::=**	**filter(Pred) | filter(P)**
Assertion A	**::=**	**FT ~ AT | AT**

Figure 5: The language syntax for DBVal assertions.

3 VALIDATION WITH ASSERTIONS

Suppose a P4 programmer wants to ensure that certain traffic is processed in the data plane at runtime as intended. Here, we assume that the P4 program is bug-free, but the packet-processing in the data plane at runtime may be incorrect because of bugs in other software or hardware components (*e.g.*, control plane, compiler, switch software, P4 pipeline). To validate packet-processing at runtime, we propose DBVal system as shown in Fig. 4. Using DBVal, the programmer expresses intended packet-processing behavior through assertions, where each assertion specifies which tables and actions should be applied to a subset of traffic. That is, the assertion describes the intended execution path in terms of a sequence of tables to be applied and actions to be executed for a subset of traffic. DBVal translates these assertions to P4 implementations, which execute assertions on the incoming traffic at line rate to check whether the specified traffic indeed followed the specified path in the data plane. Finally, DBVal raises alerts whenever the expected path is different from the observed path (violation). To summarize, we let the P4 programmer tell us what is correct and use it as a reference to validate what we observed in the data plane.

3.1 The DBVal language

We identify that the language syntax should be expressive enough to support two key constructs: One construct for expressing the subset of traffic on which assertions should be applied. Another for expressing the intended packet execution path in a P4 program. DBVal's assertion language is inspired by SDN programming languages [21, 24] developed for OpenFlow settings. We adapt them

for P4 settings. Recent works like P4Assert [29] and Poise [34] proposed languages for P4-based programmable data planes. Though they have constructs to filter packets based on packet header fields and meta-data fields, but lack the constructs necessary to express packet execution paths in a P4 program. In this work, we introduce the syntax necessary to express packet execution paths.

Fig. 5 shows proposed language syntax for DBVal assertions. It comprises of the following key constructs:

- **Path (P):** Specifies the intended packet execution path in the form of regular expressions comprising a sequence of tables and associated actions in a P4 program.
- **Predicate (Pred):** In a predicate *Pred*, operator *op* indicates comparisons such as <, >, and so on, and *in* is used to test whether field f matches with one of the values v in constant list L. For instance, one could define a list of ports *portlist* as ["port1", "port2"] and use it to filter traffic (`filter(md.ingressport in portlist)`) or to assert traffic (`assert(md.ingressport in portlist)`).
- **Filter (FT):** Filters traffic on which *assert* should be applied. A packet is filtered under two conditions: (1) predicate in the *filter* construct evaluates to true, or (2) traverses the specified intended execution path P.
- **Assert (AT):** If intended path P is present in *assert*, it checks whether the filtered traffic is indeed traversing the path P at runtime (*e.g.*, assertion **A1** in the next section). Instead, if *Pred* is present in *assert*, it ensures only a subset of traffic that satisfies predicate is traversing certain packet execution path (*e.g.*, **A3** in the next section).

3.2 Example Assertions

Consider the running security policy example presented in §2. Suppose a programmer wants to ensure that encrypted traffic from the host with IP 10.1.1.10 should be allowed. This property can be expressed using an assertion:

A1: filter(pkt.srcIP == 10.1.1.10 & pkt.srcPort == 443)
~ **assert**($\cdot^*$ ACL@allow $\cdot^*$)

In this assertion, DBVal first filters traffic with pkt.srcIP = 10.1.1.10 and pkt.srcPort = 443, and then checks whether the allow action is executed by the ACL table. Note that the path specified in the *assert* construct is a regular expression over a sequence of tables and actions. Through path constructs, the programmer can specify the expected set of tables to be applied and actions to be executed. This means that the regular expression $.*$ preceding the specified path indicates that the filtered packets may be processed by any other set of tables and actions before the ACL table.

Now consider the problem that ACL and NAT tables are placed incorrectly due to bugs, because of which the default deny action is executed by the ACL table. Similarly, if rule R1 is not cached in the TCAM and kept in the software agent by mistake (as shown in Fig. 3(c)), then the deny action in rule R2 will be applied. In both cases, instead of allowing, encrypted packets will be denied, that is the deny action is executed by the ACL table. DBVal detects this assertion failure and raises an alert.

Consider another intended behavior that all new TCP connections initiated from external network must be denied by one of the rules in the ACL table. More specifically, TCP SYN packets from the external network to internal hosts must be denied by the ACL table. This can be expressed via an assertion:

A2: filter(tcp.flags==SYN & md.ing_port in [port1,port2,..])
~ **assert**(.*T_ACL@A_deny.*)

This assertion verifies whether TCP connections initiated from the external network are denied by the ACL table.

Suppose a programmer wants to ensure that new connections from a specific set of internal hosts should be remembered (using stateful registers), so that packets in reverse direction can be checked and allowed. To keep the state secure, the programmer may want to ensure only new TCP connections from the internal hosts should access and update known-flows register (see Fig. 2). This assertion can be written as:

A3: filter(.*A_update_known_flows.*)
~ **assert**(hdr.tcp.flags==SYN & hdr.srcIP in [IP1, IP2,..])

This assertion filters packets that traverse the paths with node `A_update_known_flows` and then *assert* whether the filtered packets are indeed SYN packets originated from the given list of internal hosts. This assertion shows that intended paths can also be specified as part of *filter* construct.

Consider a case where the programmer wants to ensure that certain packets should be processed by a set of tables and actions in certain order – if packets are not processed by any one of the tables then it may result in incorrect packet-processing behavior. More specifically, consider an assertion that every packet allowed by the ACL table must be rewritten by the NAT table. It can be written as:

A4: filter(.*T_ACL@A_allow.*)
~ **assert**(.*T_NAT@A_rewrite.*)

This assertion first filters packets that are allowed by the ACL table, and then assert whether rewrite action is executed by the NAT table. This assertion shows that intended paths can be specified in both *filter* construct and *assert* construct.

Goal. Given the intended packet-processing behavior of a P4 program in the form of assertions, our goal is to detect assertion violations that happen at runtime and raise alerts along with the actual packet execution path, that is, the set of tables applied and the actions executed. The alerts could be later used for debugging the cause of the violation.

4 ASSERTION COMPILATION AND EXECUTION

Now, we briefly describe two main components of DBVal assertion compilation process. Fig. 9 shows the code snippet at each step for the running example.

- **Track packet execution path:** A packet might take any path (*i.e.*, a set of tables applied and actions executed) in a P4 program. To track every path in the data plane, we need to augment the original p4 program such that the augmented P4 program maintains and updates the execution path of packets in the per-packet state as they go through the program. In section §4.1, we provide more details on how P4 program is augmented for tracking packet execution path.

Path	Value
AHJKM	0
AHJKL	1
AHIKM	2
AHIKL	3
…….	…….
ABCEFHJKM	16
ABCEFHJKL	17
ABCEFHIKM	18
ABCEFHIKL	19

Figure 6: BL values of paths in statefull_firewall.p4 CFG

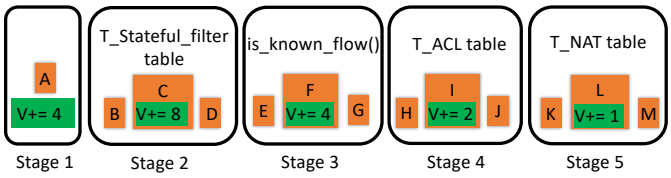


Figure 7: Mapping augmented CFG to the pipeline

- **Assertion execution at line rate:** DBVal translates assertions to corresponding P4 implementations (*e.g.*, match-action table rules, P4 if/else constructs) and annotates the original P4 program such that the annotated P4 program validates observed packet execution paths at line rate (more details in section §4.2).

4.1 Tracking packet execution path

To track every path, we augmented the original p4 program such that the per-packet state like Packet Header Vector (PHV) is updated as the packet travels. However, PHV is a scarce resource, thus the encoding technique should operate under limited memory available and update PHV with a limited set of operations supported in the data plane. In our preliminary work, we showed that Ball-Larus encoding technique, a well-known technique for profiling execution paths in software [23], is a promising fit for tracking packet execution paths in P4 programs. This is because P4 programs are loop free and the encoding does not require sophisticated updates: addition operation on path variable is sufficient. We use this technique for tracking path and execute assertions on expected paths of filtered traffic.

Ball-Larus encoding. To keep this paper self-contained, we briefly present the core idea of Ball-Larus encoding technique. Since P4 programs are loop free, control flow graph (CFG) of a P4 program is a directed acyclic graph (DAG) where each node represents a program statement such as table, action, or conditional. The Ball-Larus encoding algorithm performs reverse topological ordering of the DAG and assigns a label to each edge such that given a packet as input, as it transition from one program statement to the next, the associated edge label is added to the packet's path variable. Finally, at the end of DAG (or program) processing, the path variable value uniquely represents the path the packet has taken in the program.

```
table membershipCheck
{
    keys = {
        md.ingress_port : exact;
    }
    actions = {
        membership_check_pass;
        membership_check_fail;
    }
    const entries = {
        port1 : membership_check_pass
        port2 : membership_check_pass
        ...
        ...
    }
}
```

Figure 8: Membership check

Moreover, BL encoding technique can encode all N program paths in a single $log(N)$-bit variable, thus adding minimal overhead to per-packet meta-data that is carried across data-plane stages. After finishing the program processing, the path variable has a unique value between 0 to $N-1$.

Running example. Fig. 2 shows CFG of stateful_firewall.p4, whose edges are labeled after running the Ball-Larus (BL) encoding technique. Unlabeled edges have 0 by default, so only non-zero edge values are shown. As the packet traverses an edge, the associated edge label is added to the packet's path variable V. Fig. 6 shows BL value of each path in the CFG which has a total of 20 different paths from root node A to leaf nodes (node M and node N). When a program completes packet processing, the value V will be in the range of 0 to 19 where each value uniquely identifies the path the packet has taken. Fig. 7 shows the mapping of P4 program CFG to stages in PISA pipeline [17].

Limitation. For large P4 programs such as `switch.p4` [2] the path variable size can go as large as few hundred bits, making integer arithmetic at line rate challenging. Also, updating the same path variable by multiple tables would force compiler to put the tables across multiples stages, which are otherwise mapped to the same stage; this will increase the number of stages. One can handle both of these challenges by carefully partitioning the original DAG and assigning a different variable to each partition (*i.e.*, sub-DAG), and tracking packet execution path separately. DBVal currently uses single variable BL encoding and it works well for small programs. In our future work, we plan to extend DBVal for large programs augmented using the multi-variable BL encoding technique.

4.2 Compiling assertions

In the previous section, we demonstrated a technique to track packet execution path where at the end of packet-processing, the value in the path variable uniquely identifies the path a packet has taken in a P4 program. Now, we present how our assertion compiler translates assertions to corresponding P4 implementation. More specifically, each P4 implementation checks whether the path the packet has taken is one among the expected paths specified in an assertion. If

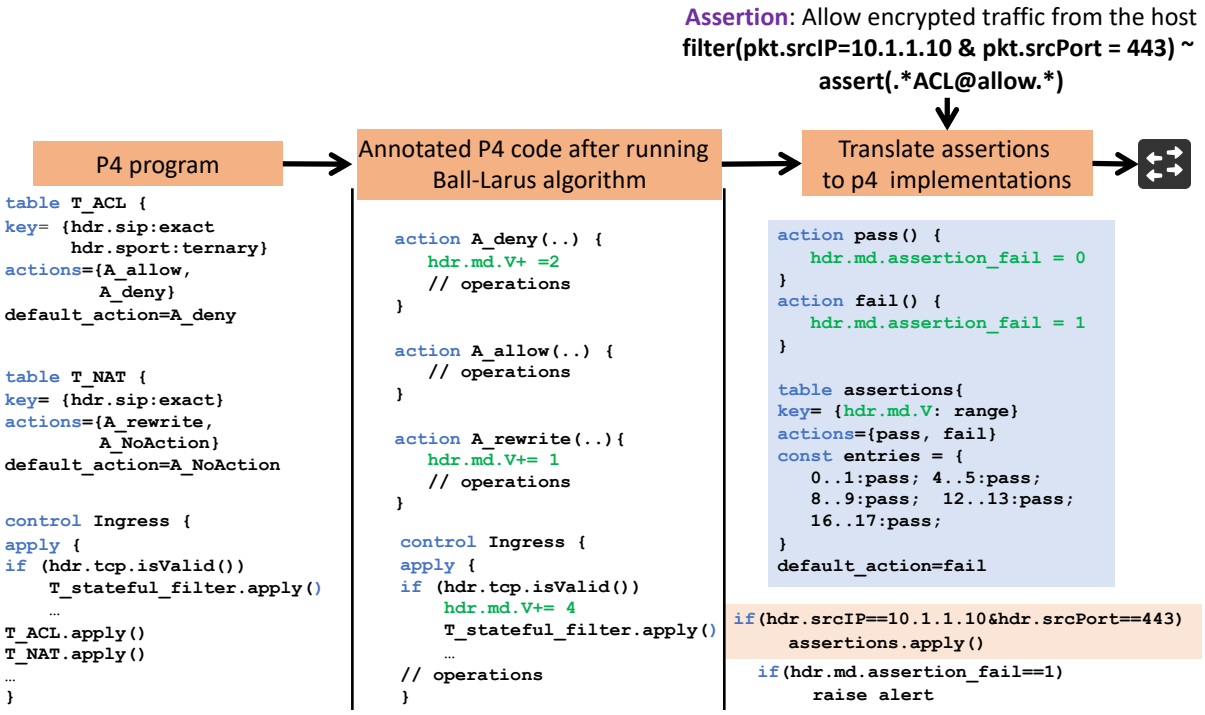


Figure 9: DBVal applies Ball-Larus encoding technique on P4 program control flow graph and track packet execution path in the P4 program. DBVal installs match-action rules, where each rule match on an intended path specified in an assertion. At the end of packet processing, the program checks whether the tracked path maintained in hdr.md.V matches with one of the path values of the intended paths.

not, it indicates an assertion failure, which in turn is reported as an alert to the control plane for further analysis.

Compiling predicates. As shown in Fig. 5, a predicate can have comparison operators, membership checks, or both. For instance, consider a predicate with only comparison operators (=, <, >). The predicate is translated to a P4 *if-else* condition to check whether it evaluates to true or false. Further, the inside of the *if-else* block contains the code for either filter or assert. For example, consider filter in assertion **A1** described in section §3. Fig. 9 (right) shows *filter(hdr.srcIP == 10.1.1.10 and hdr.srcPort == 443)* is implemented using the *if* construct in P4 language, and the if block contains the code for assertion on the packet execution path, that is, *assert(.*ACL@allow.*)*. Note, a predicate can also be part of an *assert* as in assertion **A3** described in section §3. Here, the *if* block contains code for setting a flag, which indicates that the assertion is passed.

Now consider a predicate with a membership check, that is, check whether the header value matches with one of the values in a list. The membership check predicate is translated to match-action table rules where the table key is the field in the left-hand side of the operator 'in', and the table has one rule for each member in the list on the right-hand side. That is, a membership check on a list with n items $[k_1,k_2,...,k_n]$ is converted to a match-action table with n entries. All table entries have the same action, which sets a flag to indicate that the membership check is successful. For example, Fig. 8 has P4 implementation of assertion **A2** described in §3. The assertion has a predicate as part of the filter: (*(filter(md.ingress_port in [port1,port2,..])*). The P4 implementation has a table named *membershipCheck* with *md.ingress_port* as key and one match rule for each port in the list.

Note that predicates in assertions use header values that a packet has on its arrival at a switch. However, the header values may change during the packet processing. To ensure assertions are executed on the original header values, before the execution of instructions in a P4 program control block starts, we copy the values of headers specified in the predicates to temporary meta-data variables. These meta-data variables are carried along with the packet (PHV) till the end of the pipeline where the predicates are executed. Similarly, if a predicate contains meta-data variable(s) defined in the original program, we maintain a separate copy for it. By doing so will ensure assertion checks are done on the original header values that a packet has on its arrival at a switch.

Compiling packet execution path. The path construct in an assertion is in the form of a regular expression over a sequence of tables and actions. Consider that a path construct is used in *assert* which indicates the expected packet execution path. Since a regular expression may correspond to more than one path in a P4 program CFG, we define the expected set of paths as E. When we apply Ball-Larus encoding, at the end of packet processing in the data plane, the path variable (V) in the packet's metadata has a path ID. V represents one of all possible paths (U) in the program CFG. To summarize, E is a subset of U, and our objective is to check whether $V \in E$ (assertion

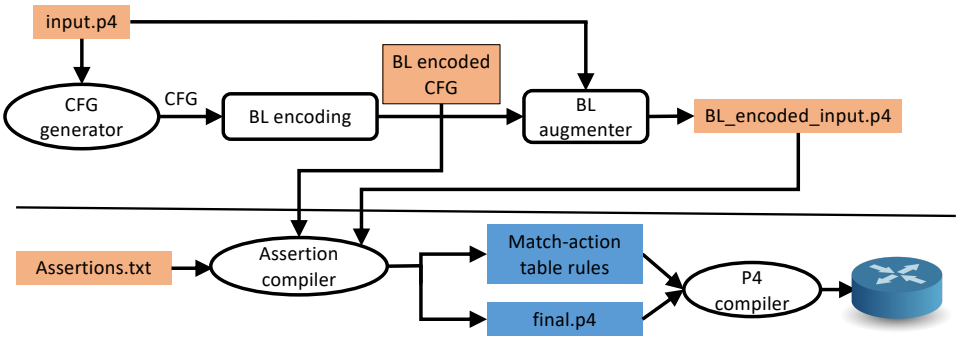


Figure 10: DBVal Implementation

pass) or not (assertion fail). On the other hand, if path construct is present in *filter*, then $V \in E$ result evaluates to true or false.

Similar to the membership check, we implement $V \in E$ check using a match-action table where the key is V. The BL values of the expected set of paths in E form table rules where each rule is either an exact match on a BL value or a range match on BL values. All rules have the same action which sets a flag to indicate either the assertion is passed (for *assert*) or the condition is true (for *filter*). For example, consider *assert(.*T_ACL@A_allow.*)* part in assertion **A1**. From the BL encoded CFG (as shown in Fig. 2), we retrieve all BL values of paths traversing the node *A_allow* and program assertion table with rules matching on the retrieved BL values (*i.e.*, *E*). Fig. 9 (right) shows its P4 implementation, where there are 10 different paths with BL values {0,1,4,5,8,9,12,13,16,17}. *hdr.md.V* is a per-packet meta-data variable that carries the BL value of the path the packet has traversed in the P4 program. We follow the same steps if the path is specified in a filter, except that the action part now sets a flag to indicate that the packet is filtered.

Optimizations. Certain assertions on large P4 programs may need many TCAM entries, especially to perform range match. Since TCAM is a scarce resource, one can explore SRAM-based packet classification algorithms like BDDs [32] at the cost of an increase in the number of stages in the pipeline. Other alternatives are using bloom filters for membership check at the cost of false positives, or perfect hash functions [1] at the cost of need for custom hash functions in the data plane. Another optimization is, if too many filtered packets fail assertion, an alert for each packet may congest the data- and control plane interface. To address this issue, we should consider restricting alert frequency either by sampling 1 out of N packets that fail assertions or by maintaining per-assertion per-epoch alerts count and use the count to decide whether to forward the alert or not, both can be implemented in the data plane.

Limitation. Currently, our design requires assertions to be written before the P4 program deployment. This design choice is made as it is simple and uses optimal data-plane resources. More specifically, DBVal filters traffic based on packet header fields that are present in assertions. In our future work, we plan to provide support for filtering traffic based on standard packet headers, so that one can add new assertions by simply adding rules to an existing match-action table at the cost of using extra data-plane resources. This will avoid recompiling the P4 program and enables adding or deleting assertions dynamically.

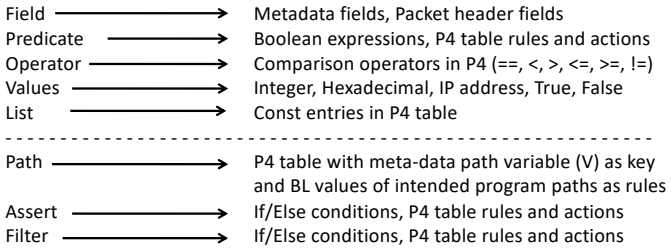


Figure 11: Mapping of assertion language constructs to P4 language constructs

5 IMPLEMENTATION

This section presents the implementation details of DBVal prototype[1] working on two targets: BMv2 [16] and Intel Barefoot Tofino [17]. Fig. 10 shows the end-to-end workflow of our system. It mainly comprises of two modules: Ball-Larus encoding module and Assertion compiler module. Implementation details of each module are given below. Fig. 11 shows the mapping of constructs in the assertion language to constructs in P4 language.

Ball-Larus (BL) encoding. This module takes the original P4 program as input and generates a BL-encoded P4 program. More specifically, for a given P4 program, CFG generator generates CFG from *.dot* and *.json* files generated by two compilers: open-source P4 compiler p4c [6] and Tofino switch compiler. Instead of relying on the P4 compiler, we can also generate the CFG by parsing the P4 program control blocks. Next, we run the Ball-Larus algorithm on the CFG and get the BL-encoded CFG. Finally, the BL augmenter parses the original P4 program line by line and augments the conditional statements and table actions in the control block with metadata variable and the arithmetic addition operation on this variable. The code is about 500 lines written in Python and the CFG is represented using the NetworkX2.2 library.

Assertion compiler. This module adds the code for assertion execution to the BL-encoded P4 program generated in the prior step.

[1] https://github.com/networked-systems-iith/DBValidator.git

More specifically, the assertion compiler takes three inputs: (1) Assertions written following the assertion language syntax (Fig. 5); (2) BL-encoded CFG; and (3) BL-encoded P4 program. The assertion compiler translates each assertion to P4 code snippet where match-action table rules are derived after parsing BL encoded CFG, and then annotate the BL-encoded P4 program with table definitions and static rules. To summarize, the final annotated p4 program maintains packet execution path in BL variable (V) and uses V along with other packet header and metadata fields as table match keys. The code for the assertion language compiler is written in C language using two libraries: Bison 3.0.4 as the syntax parser and flex 2.6.0 as the lexer. The code for assertion augmentation is written in Python with around 350 lines of code.

6 EVALUATION

Our main goal for evaluation is to study (1) how effective DBVal is in detecting packet-processing errors induced by bugs at runtime, (2) how expressive the DBVal assertion language is, and (3) what is the DBVal overhead in terms of the data-plane resources required to execute assertions.

6.1 Bug detection using DBVal

Now we demonstrate how DBVal is able to detect incorrect packet-processing behavior induced by the bugs described in the motivation section (§2). We compiled assertion **A1** written for stateful_firewall.p4 program, annotate the P4 program, and deployed the annotated P4 program on the Tofino switch software model target that is part of the software development environment (SDE). We synthetically created the compiler bug and switch software bug described in Section §2 and checked whether DBVal successfully detects incorrect packet-processing.

According to the assertion **A1**, filtered packets must traverse *T_ACL@A_allow* edge in the CFG shown in Fig. 2. So the *assertions* table is programmed with a set of expected BL values (E) {0,1,4,5,8,9,12,13,16,17}. Now, consider a security policy violation due to a compiler bug where ACL and NAT tables are placed in reverse order. We emulate incorrect table placement bug by modifying the annotated Stateful_firewall.p4 program such that the program applies NAT table before ACL table. We also install the table rules shown in Fig. 3(c). As a consequence, a packet will now traverse *T_NAT@A_rewrite* edge followed by *T_ACL@A_deny* edge. So the set of observed BL values (O) is {3,7,11,15,19}, and the packet meta-data variable *hdr.md.V* (in short V) will now contain one of the values in set O (*i.e.*, $V \in O$). Since $E \cap O = null$, $V \notin E$ which means V would not match with any rule in *assertions* table, hence, an assertion failure alert is sent to the control plane.

Now, consider another violation due to a bug in the switch software. We synthetically created this bug by deleting allow rule $R1$ in Fig. 3(c). As a consequence, a packet with source IP address 10.1.1.10 and source port 443 will now hit the ACL deny rule $R2$. In other words, the packet traverses *T_ACL@A_deny* edge with O = {2,3,6,7,10,11,14,15,18,19}. As $V \in O$ and $E \cap O = null$, $V \notin E$ will become true, that is, no rule in *assertions* table matches with V, hence, the data plane raises assertion failure alert to the controller.

Limitation. DBVal can detect incorrect packet-processing behavior only if the set of tables and actions in the observed path does not match with any of the sets in the expected paths. In a situation where the set of tables and actions in the observed path matches with a set in the expected paths, but is executed in a different order (possibly due to bugs), DBVal cannot detect such bugs. This is because DBVal tracks a packet execution path by adding augmented BL value along the path the packet traverses. Since addition is commutative, for a given set of tables and actions, the BL value will be the same for all possible execution orders in the set.

For instance, in the running stateful_firewall.p4 example with assertion **A1**, consider that the control plane has installed an arbitrary set of rules (includes faulty rules) into ACL and NAT tables, such that ACL table has a single rule allowing all packets and NAT table has a single rule that rewrites all packets. Then, though ACL and NAT are reversed due to the compiler bug, DBVal cannot detect this bug because all packets execute NAT@rewrite followed by ACL@allow; the corresponding BL value matches with the BL values of one of the expected paths that have ACL@allow followed by NAT@rewrite. Here, though the order is different, the set of tables and actions are the same. Note that in real networks, an ACL table typically has a default deny and allow rules for each traffic class; a default allow used in this example can be considered as a faulty rule installed by mistake.

Summary. To summarize, the use cases demonstrate DBVal is able to validate actual packet-processing behavior and raise alerts whenever the actual behavior (interms of a set of tables and actions) does not match with the intended behavior.

6.2 Assertion language expressiveness

We compiled assertions written for a total of eight programs of varying size and complexity, listed in Table 1. The programs are taken from three main sources: P4 learning [12], P4 programs survey [13], and P4 tutorials [5]. We carefully go through each P4 program and identify the intended packet-processing behavior of certain traffic classes. For P4 programs ①-⑦, we identify intended behavior, especially path-specific properties to be satisfied in the data plane. For ⑧, we derive assertions based on the comments available in the program.

Next, we expressed the intended behavior in terms of assertions using DBVal assertion language. Each assertion is written using three kinds of per-packet information: packet headers fields (PHF), meta-data fields (MF), and packet execution path (P). The conditions on these field values must be satisfied (assert) to execute an action, or to update a register. For example, the second assertion written for ① heavy_hitter.p4 program checks whether bloom filter maintained in the switch registers (update_bloom_filter) is updated by tcp traffic only; the second assertion for ⑦ stateful_firewall checks whether all tcp syn packets arriving on port 2 (external network) are blocked; the first assertion for ⑧ hula.p4 program checks whether best path to a destination is updated whenever queue depth carried in a hula packet is less than the current queue depth.

The last column in Table 1 shows which information is used by *filter* and *assert* constructs in the assertions written for respective P4 program. To summarize, the flexibility to use different kinds of information in *filter* and *assert* constructs enables the programmers to check different kinds of properties at runtime.

S.No.	P4 Program	List of assertions	(filter, assert)
1.	heavy_hitter.p4 [10]	1. filter(meta.counter_one >THRESHOLD & meta.counter_two >THRESHOLD) ~ assert(.*MyIngress@drop.*) 2. filter(.*update_bloom_filter.*) ~ assert(hdr.tcp.isValid() == true)	1. (MF, PHF) 2. (P, MF)
2.	traceroutable.p4 [15]	1. filter(.*set_nhop.*) ~ assert(hdr.ipv4.isValid() == true & hdr.ipv4.ttl >1) 2. filter(hdr.ipv4.ttl == 1) ~ assert(hdr.ipv4.protocol == 1)	1. (P, MF and PHF) 2. (PHF, PHF)
3.	flowlet_switching.p4 [8]	1. filter(.*MyIngress@drop.*) ~ assert(hdr.ipv4.ttl == 0)	1. (P, PHF)
4.	ip_forwarding.p4 [9]	1. filter(.*ipv4_lpm@ipv4_forward.*) ~ assert(hdr.ipv4.isValid() == true) 2. filter(.*ipv4_lpm@ipv4_forward.*) ~ assert(hdr.ipv4.ttl >0)	1. (P, MF) 2. (P, PHF)
5.	fast_reroute.p4 [19]	1. filter(meta.linkState >0) ~ assert(.*read_alternativePort.*)	1. (MF, P)
6.	mpls.p4 [11]	1. filter(dstIP == X)~ assert(.*fec_to_label@ add_mpls_header * mpls_tbl@mpls_forward.*) 2. filter(.*fec_to_label.*) ~ assert(meta.is_ingress_border == 1 & hdr.ethernet.etherType == TYPE_IPV4) 3. filter(.*mpls_forward.*) ~ assert(hdr.ethernet.etherType == TYPE_MPLS)	1. (PHF, P) 2. (P, MF and PHF) 3. (P, PHF)
7.	stateful_firewall.p4 [14]	1. filter(.*set_allowed.*) ~ assert(hdr.tcp.syn == 1 & standard_metadata.ingress_port == 1) 2. filter(hdr.tcp.syn == 1 & standard_metadata.ingress_port == 2) ~ assert(.*acl@deny.*) 3. filter(.*MyIngress@allow.*) ~ assert(hdr.tcp.flags != 2 & meta.ingress_port == 2 & meta.register_cell_one == 1)	1. (P, PHF and MF) 2. (PHF and MF, P) 3. (P, PHF and MF)
8.	hula.p4 [36]	1. filter(hdr.hula.isValid() == true & hdr.hula.qdepth < meta.old_qdepth) ~ assert(.*MyIngress.change_best_path_at_dst.*) 2. filter(hdr.ipv4.isValid() == true & meta.port == 0) ~ assert(.*MyIngress.hula_nhop@MyIngress.hula_get_nhop.*) 3. assert(.*MyIngress.change_best_path_at_dst^MyIngress.return_hula_to_src.*)	1. (MF, P) 2. (MF, P) 3. (-, P)

Table 1: A summary of our benchmark programs and assertions. P:Packet execution path, MF:Meta-data field, PHF:Packet header field.

S.No.	P4 Program	(LOC in original P4 program, Total #Paths)	(LOC, #Tables, #Actions, #If/Else blocks) added	(#PHV bits, #Range rules, #ALUs) used	Compilation time (in secs)
1.	heavy_hitter.p4	(228, 9)	(72, 2, 4, 2)	(6 bits, 2, 4)	0.0832
2.	traceroutable.p4	(236, 5)	(57, 1, 4, 3)	(5 bits, 1, 4)	0.0843
3.	flowlet_switching.p4	(239, 6)	(37, 1, 2, 1)	(4 bits, 1, 2)	0.0804
4.	ip_forwarding.p4	(152, 3)	(67, 2, 4, 2)	(4 bits, 2, 3)	0.0663
5.	fast_reroute.p4	(190, 8)	(36, 1, 2, 1)	(4 bits, 2, 3)	0.074
6.	mpls.p4	(244, 32)	(119, 3, 6, 3)	(8 bits, 12, 8)	0.0827
7.	stateful_firewall.p4	(318, 190)	(190, 3, 6, 3)	(11 bits, 77, 47)	0.0992
8.	hula.p4	(449, 15)	(45, 3, 6, 2)	(7 bits, 3, 13)	0.087

Table 2: Extra data-plane resources needed for executing assertions and the time taken to compile the assertions.

6.3 Data-plane Overhead

We deployed annotated P4 programs on the BMv2 switch and calculated the overhead in terms of data-plane resources required both to track packet execution path and execute assertions written for respective P4 program (as shown in Table 1). To the best of the authors knowledge, P4 programs written for the Tofino software switch model are not publicly available. So the evaluation is performed on P4 programs written for the BMV2 software switch.

Data-plane Overhead. The third and fourth columns in Table 2 summarize the data-plane resources required to execute assertions together for each program. One column shows the lines of code (LOC), tables, actions, and if/else blocks added to the original program. They are needed to filter packets and to update flags that indicate whether

the assertion is pass or fail. The other column shows the number of bits required in per-packet metadata (PHV), number of ranges rules, and ALUs. More specifically, the number of PHV bits is equal to the sum of $\log(\#Paths)$-bits and the number of assertions (one bit per assertion). Range rules are required to match on path ID carried in Ball-Larus variable (V). Finally, we calculate required ALUs as the sum of the number of updates to BL variable (V) and the number of comparison operators in if/else conditions.

As expected, an assertion written for a program with more LOC needs more PHV bits, table rules, and ALUs. This is because the control flow graph has more nodes, thereby more PHV bits are required to uniquely identify packet path. This overhead is significantly small compared to the existing switches with a few thousand PHV bits. The number of range-based table rules required would depend on the number of paths to the table or action specified in an assertion. Finally, the number of ALUs required is proportional to the number of edges as the BL variable (V) has to be updated on each edge transition. Moreover, as the number of per-program assertions increases, the rules, PHV bits, and ALUs also increase. Currently, we assign one bit (flag) for each assertion, and the bit value determines whether an assertion is pass or fail. Predicates on packet header fields and meta-data fields in an assertion are implemented using if/else statements. The comparison operations in an if/else condition uses ALUs on the switch hardware, therefore the number of ALUs required increases with the number of comparison operations in each assertion and the number of assertions.

Compilation time. We compiled assertions on a VM equipped with i5-9300H 2-core 2.44 GHz CPU and 4 GB of RAM. The fifth column in Table 2 shows the time taken by the assertion compiler to translate assertions in Table 1 to P4 implementations for each P4 program. We observe that assertions are compiled in less than 0.1 seconds.

7 RELATED WORK

Assertions. P4Assert [29, 43] performs static verification on P4 programs using symbolic execution technique; it translates P4 program annotated with assertions to C model, and verify C model to find bugs before the program deployment. In contrast, DBVal assertions validate observed behavior in the data plane at runtime after the deployment. Moreover, our language allows users to express fine-grained data-plane behavior at the level of tables applied and actions executed on a subset of traffic. SDN-Assert [24] proposes an assertion-based debugging language to verify dynamic properties of controller applications written for OpenFlow-based switches. Whereas DBVal is designed for P4-based switches.

P4 Program Verification. Recent work [28, 29, 39, 40, 50] performs static analysis to verify P4 program properties and find bugs before the program is deployed. However, not all properties can yet be verified by the existing tools, and these tools still operate at the level of the P4 program. That is, they can verify if the P4 program logic is bug-free for a specific set of bugs. Thus, these tools are not designed to detect bugs in switch operating systems, P4 pipeline, or compilers, especially non open-source compilers that cause incorrect packet-processing behavior. DBVal is complementary to these works.

P4 Compiler Bug Detection. P4Fuzz [20] automatically fuzzes P4 compilers written for different targets and find bugs and vulnerabilities in compilers. It generates syntactically and semantically valid P4 programs, and sends test packets to check whether programs are compiled and deployed correctly. Gauntlet [46] finds crash bugs and semantic bugs in P4 compilers via random program generation, translation, validation, and model-based testing. Our approach is different from Gauntlet and P4Fuzz as we focus on detecting packet-processing errors due to bugs in different components (*e.g.*, compiler, control-plane programs, data-plane state, run-time system) after the program is deployed. Hence DBVal nicely complements existing works by helping to uncover bugs in other components.

Test Packet Generation. Previous work such as ATPG [51] and P4pktgen [44] study the automatic generation of test packets from specifications of network devices. P4RL [49], as an enhancement to these methods, uses reinforcement learning-enabled fuzzing to validate switch behavior at runtime. A follow-up work P6 [48], reduces input search space and efficiently detect, localize, and patch bugs at runtime. PTA [25] proposes a portable test architecture framework that allows existing verification tools to re-configure and tests the hardware. The automatic test packet generation approach may not exercise every packet-processing scenario possible after deployment in useful time, especially for large programs with table rules not known before the deployment. In contrast to these works, DBVal considers the actual packet as a test packet, and it can potentially detect packet-processing errors that trigger only with a certain sequence of packets, table rules, and data-plane state in registers, which may not be exercised during testing. As a result, DBVal can complement these works by enabling the detection of bugs on paths not exercised during testing.

Data-Plane Post-Cards. Previous work [30, 47, 52] has explored collecting information about a packet into a "post-card" as it traverses the switch and sending relevant post-cards to a controller to validate observed behavior. NetSight's post-card [30] includes the packet header, its outgoing port, and the version number for the rules installed on the switch that processed the packet but does not track which tables and actions have been hit by the packet. P4Consist [47] copies in-band telemetry data and multi-hop route inspection data comprising switch id, port id, and rule id, onto its postcard. However, this work focuses on validation of paths at network level. In contrast, DBVal design focuses on doing validation at switch level efficiently. KeySight [52] copies every packet field read and written at each match-action table into a post card, which has high overhead in terms of bits required in packet metadata, especially for large P4 programs. DBVal significantly reduces metadata bits required for tracking by using Ball-larus encoding technique. One can extend DBVal to validate network-level paths by efficiently tagging switch-level execution path to packet headers and detag at the edge.

8 DISCUSSION

Minimal trusted base. DBVal depends on the P4 compiler to compile the instructions of the BL augmentation part and the assertions execution in the data plane. The minimum trust expected from the P4 compiler is to compile correctly (a) the Ball-Larus encoding instructions required for path tracking, and (b) the P4 code that executes

assertions on the observed path. However, a bug in the P4 compiler could not only lead to incorrect packet processing behaviors discussed in the paper, but also lead to incorrect implementation of DBVal components, that is, path tracking and assertion execution in the data plane. Since DBVal captures the intended behavior independent of the underlying toolchain (P4 compiler), DBVal can still detect whenever policy violation happens through an assertion failure alert. But it cannot pinpoint whether the failure is because of the incorrect compilation of the DBVal components or the original P4 program. In either way, the policy violation detection would help in detecting P4 compiler bugs.

Bug localization. DBVal design aims for detecting a mismatch between the expected packet execution path and the observed path for a certain traffic class. It tracks the execution path of every packet and raises an alert (along with path ID) whenever the observed path is not in the list of expected paths. From the observed path ID and the expected path IDs, one can localize to a point (table and action) at which the observed path deviates from the expected ones. Though this is useful information, it may not be sufficient to localize and identify the cause (bug) of the deviation. A recent work [48] localizes and patches software bugs in P4 programs. Extending DBVal for bug localization can be potential future research.

Extending DBVal's assertion language. Using DBVal assertion language, a P4 program can write assertions on the packet execution path of a certain traffic class that he/she is interested in. The assertion language can be extended to support assertions on expected relations among header fields. For example, to check whether the TTL value was decremented as expected, or whether the source MAC address at egress is the same as the destination MAC at ingress, etc. Such assertions on packet relations can be reduced to path-based assertions; check whether the action that decrements TTL, or the action that updates MAC address is present in the observed packet execution path.

9 CONCLUSION

We present DBVal system for detecting packet-processing errors induced by bugs in the P4 software and hardware ecosystem. Through assertions, we let the P4 programmers specify intended behavior interms of assertions on which tables and actions should be applied on a subset of traffic. DBVal compiler automatically translates assertions to P4 implementations and validates the actual behavior, that is, packet execution path at line rate. To track execution path of every packet, we apply Ball-Larus encoding technique that works well under the switch resource constraints. We prototype DBVal for two P4 targets: BMV2 and Tofino software switch, and successfully compiled assertions written for a variety of P4 programs.

10 ACKNOWLEDGEMENTS

We thank the shepherd of our paper, Andreas Voellmy, and the anonymous reviewers for their thoughtful feedback. The paper writing has improved substantially by addressing their comments. We also thank Jennifer Rexford, Suriya Kodeshwaran, Harish S A, for their valuable feedback on the earlier drafts and for their participation in the discussions. This work is supported by a startup grant awarded by IIT Hyderabad and a fellowship by DST NM-ICPS TiHAN.

REFERENCES

[1] 2012. *C Minimal Perfect Hashing Library*. Retrieved June 2021 from http://cmph.sourceforge.net/
[2] 2015. *BMv2 switch.p4*. Retrieved May 2021 from https://github.com/p4lang/switch
[3] 2016. *Netronome Agilio CX SmartNICs*. Retrieved June 2021 from https://www.netronome.com/products/agilio-cx/
[4] 2017. *Network path not found? Forward Networks Blog*. Retrieved February 2021 from https://bit.ly/2FzpEEZ
[5] 2017. *P4 tutorials*. Retrieved February 2020 from https://github.com/p4lang
[6] 2017. *p4c issues*. https://github.com/p4lang/p4c/issues
[7] 2018. *Intel FPGAs and Programmable Devices*. https://www.intel.in/content/www/in/en/products/programmable.html
[8] 2019. *flowlet_switching.p4*. Retrieved June 2021 from https://github.com/nsg-ethz/p4-learning/blob/master/exercises/05-Flowlet_Switching/solution/p4src/flowlet_switching.p4
[9] 2019. *forwarding.p4*. Retrieved June 2021 from https://github.com/nsg-ethz/p4-learning/blob/master/examples/ip_forwarding/forwarding.p4
[10] 2019. *heavy_hitter.p4*. Retrieved June 2021 from https://github.com/nsg-ethz/p4-learning/blob/master/examples/heavy_hitter/heavy_hitter.p4
[11] 2019. *mpls.p4*. Retrieved June 2021 from https://github.com/nsg-ethz/p4-learning/blob/master/exercises/04-MPLS/mpls_basics/solution/basics.p4
[12] 2019. *P4 learning*. Retrieved January 2021 from https://github.com/nsg-ethz/p4-learning
[13] 2019. *P4 programs survey*. Retrieved June 2021 from https://github.com/muhe1991/p4-programs-survey
[14] 2019. *stateful_firewall.p4*. Retrieved June 2021 from https://github.com/nsg-ethz/p4-learning/blob/master/examples/stateful_firewall/stateful_firewall.p4
[15] 2019. *traceroutable.p4*. Retrieved June 2021 from https://github.com/nsg-ethz/p4-learning/blob/master/exercises/09-Traceroutable/solution/p4src/traceroutable.p4
[16] 2020. *BMV2 software switch*. Retrieved April 2021 from https://github.com/p4lang/behavioral-model
[17] 2020. *Tofino, World's Fastest P4-Programmable Ethernet Switch ASICs*. Retrieved November 2019 from https://www.barefootnetworks.com/products/brief-tofino/
[18] 2021. *DBVal code*. https://github.com/networked-systems-iith/DBValidator.git
[19] 2021. *fast_reroute.p4*. Retrieved June 2021 from https://github.com/nsg-ethz/p4-learning/blob/master/exercises/12-Fast-Reroute/solution/p4src/fast_reroute.p4
[20] Andrei-Alexandru Agape, Madalin Claudiu Danceanu, Rene Rydhof Hansen, and Stefan Schmid. 2021. P4Fuzz: Compiler Fuzzer for Dependable Programmable Dataplanes. In *International Conference on Distributed Computing and Networking 2021*.
[21] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. 2014. NetKAT: Semantic Foundations for Networks. In *POPL*.
[22] Jiasong Bai, Menghao Zhang, Guanyu Li, Chang Liu, Mingwei Xu, and Hongxin Hu. 2020. FastFE: Accelerating ML-based Traffic Analysis with Programmable Switches. In *ACM SIGCOMM SPIN workshop*.
[23] Thomas Ball and James R. Larus. 1996. Efficient Path Profiling. In *MICRO*.
[24] Ryan Beckett, Xuan Kelvin Zou, Shuyuan Zhang, Sharad Malik, Jennifer Rexford, and David Walker. 2014. An Assertion Language for Debugging SDN Applications. In *Workshop on Hot topics in software defined networking*.
[25] Pietro Bressana, Noa Zilberman, and Robert Soulé. 2020. Finding hard-to-find data plane bugs with a PTA. In *CoNEXT*.
[26] Marco Canini, Daniele Venzano, Peter Perešíni, Dejan Kostić, and Jennifer Rexford. 2012. A NICE Way to Test OpenFlow Applications. In *USENIX NSDI*.
[27] Marinos Dimolianis, Adam Pavlidis, and Vasilis Maglaris. 2020. A Multi-Feature DDoS Detection Schema on P4 Network Hardware. In *ICIN*.
[28] Dragos Dumitrescu, Radu Stoenescu, Lorina Negreanu, and Costin Raiciu. 2020. bf4: towards bug-free P4 programs. In *ACM SIGCOMM*.
[29] Lucas Freire, Miguel Neves, Lucas Leal, Kirill Levchenko, Alberto Schaeffer-Filho, and Marinho Barcellos. 2018. Uncovering Bugs in P4 Programs with Assertion-Based Verification. In *ACM SOSR*.
[30] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Mazières, and Nick McKeown. 2014. I Know What Your Packet Did Last Hop: Using Packet Histories to Troubleshoot Networks. In *USENIX NSDI*.
[31] Kuo-Feng Hsu, Ryan Beckett, Ang Chen, Jennifer Rexford, and David Walker. 2020. Contra: A Programmable System for Performance-aware Routing. In *USENIX NSDI*.
[32] Theo Jepsen, Ali Fattaholmanan, Masoud Moshref, Nate Foster, Antonio Carzaniga, and Robert Soulé. 2020. Forwarding and Routing with Packet Subscriptions. In *CoNEXT*.
[33] Qiao Kang, Jiarong Xing, and Ang Chen. 2019. Automated Attack Discovery in Data Plane Systems. In *USENIX CSET*.
[34] Qiao Kang, Lei Xue, Adam Morrison, Yuxin Tang, Ang Chen, and Xiapu Luo. 2020. Programmable In-Network Security for Context-aware BYOD Policies. In *USENIX Security*.

[35] Naga Katta, Omid Alipourfard, Jennifer Rexford, and David Walker. 2016. CacheFlow: Dependency-Aware Rule-Caching for Software-Defined Networks. In *ACM SOSR*.

[36] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. 2016. HULA: Scalable Load Balancing UsingProgrammable Data Planes. In *ACM SOSR*.

[37] Daehyeok Kim, Zaoxing Liu, Yibo Zhu, Changhoon Kim, Jeongkeun Lee, Vyas Sekar, and Srinivasan Seshan. 2020. TEA: Enabling State-Intensive Network Functions on Programmable Switches. In *ACM SIGCOMM*.

[38] Suriya Kodeswaran, Mina Tahmasbi Arashloo, Praveen Tammana, and Jennifer Rexford. 2020. Tracking P4 Program Execution in the Data Plane. In *ACM SOSR*.

[39] Jed Liu, William Hallahan, Cole Schlesinger, Milad Sharif, Jeongkeun Lee, Robert Soulé, Han Wang, Călin Caşcaval, Nick McKeown, and Nate Foster. 2018. P4V: Practical Verification for Programmable Data Planes. In *ACM SIGCOMM*.

[40] Nuno Lopes, Nikolaj Bjørner, Nick McKeown, Andrey Rybalchenko, Dan Talayco, and George Varghese. 2016. Automatically Verifying Reachability and Well-Formedness in P4 Networks. *MSR Technical Report, MSR-TR-2016-65* (2016).

[41] Roland Meier, Thomas Holterbach, Stephan Keck, Matthias Stähli, Vincent Lenders, Ankit Singla, and Laurent Vanbever. 2019. (Self) Driving Under the Influence: Intoxicating Adversarial Network Inputs. In *ACM HotNets*.

[42] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. 2017. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *ACM SIGCOMM*.

[43] Miguel Neves, Lucas Freire, Alberto Schaeffer-Filho, and Marinho Barcellos. 2018. Verification of P4 Programs in Feasible Time using Assertions. In *CoNEXT*.

[44] Andres Nötzli, Jehandad Khan, Andy Fingerhut, Clark Barrett, and Peter Athanas. 2018. P4pktgen: Automated Test Case Generation for P4 Programs. In *ACM SOSR*.

[45] O'Connor Vachuska Peterson, Cascone and Davie. 2020. *Software-Defined Networks: A Systems Approach*. https://sdn.systemsapproach.org/ Accessed: June 2021.

[46] Fabian Ruffy, Tao Wang, and Anirudh Sivaraman. 2020. Gauntlet: Finding Bugs in Compilers for Programmable Packet Processing. In *USENIX OSDI*.

[47] Apoorv Shukla, Seifeddine Fathalli, Thomas Zinner, Artur Hecker, and Stefan Schmid. 2020. P4CONSIST: Toward Consistent P4 SDNs. *IEEE Journal on Selected Areas in Communications* 38, 7 (2020), 1293–1307.

[48] Apoorv Shukla, Kevin Hudemann, Zsolt Vági, Lily Hügerich, Georgios Smaragdakis, Artur Hecker, Stefan Schmid, and Anja Feldmann. 2021. Fix with P6: Verifying Programmable Switches at Runtime. In *IEEE INFOCOM*.

[49] Apoorv Shukla, Kevin Nico Hudemann, Artur Hecker, and Stefan Schmid. 2019. Runtime Verification of P4 Switches with Reinforcement Learning. In *Workshop on Network Meets AI & ML*.

[50] Radu Stoenescu, Dragos Dumitrescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. 2018. Debugging P4 Programs with Vera. In *ACM SIGCOMM*.

[51] Hongyi Zeng, Peyman Kazemian, George Varghese, and Nick McKeown. 2012. Automatic Test Packet Generation. In *CoNEXT*.

[52] Yu Zhou, Jun Bi, Tong Yang, Kai Gao, Cheng Zhang, Jiamin Cao, and Yangyang Wang. 2018. KeySight: Troubleshooting Programmable Switches via Scalable High-Coverage Behavior Tracking. In *ICNP*.