TRACKING P4 PROGRAM EXECUTION IN THE DATA PLANE

Suriya Kodeswaran

A MASTER'S THESIS

PRESENTED TO THE FACULTY

OF PRINCETON UNIVERSITY

IN CANDIDACY FOR THE DEGREE

OF MASTER OF SCIENCE IN ENGINEERING

RECOMMENDED FOR ACCEPTANCE BY

THE DEPARTMENT OF COMPUTER SCIENCE

Adviser: Jennifer Rexford

June 2020

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License. (https://creativecommons.org/licenses/by-sa/4.0/)

ABSTRACT

While programmable switches provide operators with much needed control over the network, they also increase the potential sources of packet processing errors. Bugs can happen anywhere: in the P4 program, the controller installing rules into tables, or the compiler that maps the P4 program into the resource constrained switch pipelines. Most of these bugs manifest themselves after certain sequences of packets with certain combinations of rules in the tables. Tracking each packet's execution path through the P4 program, i.e., the sequence of tables hit and the actions applied, directly in the data plane is useful in localizing such bugs as they occur in real time. The fact that programmable data planes require P4 programs to be loop-free and can perform simple integer arithmetic operations makes them amenable to Ball-Larus encoding, a well known technique in profiling execution paths in software programs that can efficiently encode all N paths in a single $\lceil \log(N) \rceil$ bit variable. However, for real world P4 programs, the path variable can get quite large, making it inefficient for integer arithmetic at line rate. Moreover, the encoding could require a subset of tables that would otherwise have no data dependency, to update the same variable. By carefully breaking up the P4 program into disjoint partitions and tracking each partition's execution path separately, we show how to minimally augment P4 programs to track the execution path of each packet. With this system in place, we then provide intuition towards using the tracked path to detect and localize bugs in P4 programs.

ACKNOWLEDGEMENTS

I would like to thank my adviser, Jennifer Rexford for her invaluable knowledge and experience. I also want to thank my colleagues Praveen Tammana and Mina Arashloo for their significant contributions to the project.

TABLE OF CONTENTS

1. Introduction
2. Ball-Larus for P4 Programs
    a. Running example
3. Challenges
    a. Addition on large operands
    b. Extra data dependencies
4. Multi-Variable Path Encoding
    a. The partitioning problem
    b. Running example
    c. Choosing K (# path variables)
    d. The optimization problem
5. Evaluation
    a. Prototype
    b. Augmenting the program
    c. Integration with existing compilers
    d. Benchmark programs
    e. Data-plane overhead
6. Usecases: Detecting and Localizing Bugs
    a. Including ground truth in data-plane
    b. Run time path assertions
    c. Sample and check paths offline
7. Related Works
    a. Data-plane postcards
    b. Test packet generation
    c. Verification
8. Conclusion
9. References

Tracking P4 Program Execution in the Data Plane

Suriya Kodeswaran

1 Introduction

Programmable switches [16, 11, 7, 4] allow network operators to customize the switch data-plane using high-level languages such as P4 [12]. This provides much-needed flexibility and fine-grained control over switches in the network. However, compared to fixed-function switches, programmability increases the potential sources of packet processing errors as a significant portion of the data-plane behavior is only specified at program compile or run time.

To see why, consider all the complicated pieces of software involved in operating the data plane of a programmable switch. The P4 program itself, which specifies the match-action tables and the order in which they should process incoming packets, can get quite large and complicated in practice: the `switch.p4` program [15], the open-source implementation of a standard switch in P4, has $\sim10^{34}$ different control paths through 157 tables and $\sim$307 actions! To fit these programs into the extremely resource-constrained physical pipelines of programmable switches, P4 compilers implement several rounds of aggressive target-specific optimizations and code transformations, which makes the compiler software grow into large complicated pieces of software with hundreds of thousands of line of code. Finally, after the P4 program is compiled and installed on the switch, the contents of match-action tables are continuously modified by control-plane programs that add, modify, and remove rules from the tables at run-time. In practice, these control-plane programs are often large and complicated as well, as they have to deal with the intricacies of consistently transitioning the data plane from one set of rules to another in response to various run-time events.

Any bug in this complex collection of software can adversely affect how packets are processed in the data plane. Previous work has uncovered several bugs in existing P4 and control-plane programs [8, 14, 5, 3]. There are also several bug reports for existing compilers, describing non-trivial mismatches between the expected packet processing behavior described in the P4 program and the observed behavior of the compiled program running on the switch. In most cases, these bugs happen in corner cases, only manifesting themselves after certain sequences of incoming packets with certain combinations of rules in the tables. Thus, given the size and complexity of real-world P4 programs, they are typically not uncovered during testing prior to deployment. Even when they are triggered by production traffic afterwards, their subtle nature makes them difficult to reproduce for analysis. In this paper, we propose a useful data-plane primitive for detecting and localizing such bugs as they occur in real time: tracking each packet's execution path through the P4 program, i.e., the sequence of tables hit and the actions applied, directly on the data plane. This effectively turns *every* packet that goes through the switch into a potential test packet for the data plane. If there is prior knowledge about expected execution paths for certain classes of traffic, e.g., from static analysis of the program or in form of assertions from the programmers themselves, it is possible to detect when the observed execution path deviates from the expected one directly in the data plane. Alternatively, one could send some packets from every observed path to a local controller running on the switch CPU to compute their expected execution paths and compare them against the observed ones. One could even have each switch tag the packet with the observed execution path, e.g., as part of the INT header, so that the final destination can recover and analyze every decision made by every switch in processing the packet.

Besides detecting bugs, tracking packets' execution paths in the data plane is a valuable tool for localizing bugs as well. Suppose the operators detect a problem with a certain subset of traffic, either a mismatch between their expected and observed execution paths as described above or other correctness and performance problems detected through other monitoring tools. To localize the problem, operators can simply ask the switch to send the program paths that those packets are taking right then in the data plane to the controller to analyze where in the P4 program the problem is coming from. Once the problem is localized to a certain part of the program, they can then find out whether it is due to a bug in the P4 program itself, or an incorrect match-action rule installed by the control plane, or the compiler not compiling that part of the program correctly.

To track packet execution paths in the data plane, we need to augment the original P4 program to encode the sequence of tables and actions that process the packet as it goes through the switch. As table and actions are triggered, we can track their execution by updating the Packet Header Vector (PHV), the limited per-packet state that travels with the packet throughout its processing time. The PHV is a valuable fixed size resource, typically a few hundred bytes, storing parsed packet header as well as any meta data required for processing the packet. Typically, the more complex a program is, the greater number of PHV bits it requires for processing packets. A seemingly natural approach towards tracking packet execution paths is to have a flag bit for each portion of the program we want to track, setting it to one if it was used in processing the packet. This, however, can quickly deplete the available bits in the PHV.

We find that Ball-Larus encoding [2], a well-known technique in profiling execution paths in software, is a promising fit for tracking packet execution paths in P4 programs: As we show in §2, when programs are loop free, like in P4, Ball-Larus can encode all N program paths in a single $\lceil log(N) \rceil$-bit variable, thus adding minimal overhead to the per-packet metadata that is carried across data-plane stages. Moreover, Ball-Larus encoding does not require sophisticated updates to the path variable: it carefully labels every transition between program statements with an integer. As the input (packet in our setting) transitions from one statement to the next, the integer label for that transition is added to the path variable, an operation perfectly within the capabilities of programmable data planes today.

Nevertheless, if adapted naïvely, Ball-Larus encoding can have prohibitive overhead in terms of action complexity and number of stages. For large and complex P4 program such as `switch.p4`, the path variable can get as large as a few hundred bits, making it inefficient for integer arithmetic at line rate. Moreover, the encoding adds extra data dependencies between tables that have to update the path variable. These extra dependencies could force tables, that were otherwise independent and mapped by the compiler to the same stage in the hardware pipeline, to span across multiple stages instead. We show, in §4, how we overcome both of these challenges by carefully partitioning the P4 program and tracking the execution path of each partition separately.

We have implemented a prototype that takes a P4 program as input and outputs an augmented P4 program that can track packet execution paths. We augment a variety of P4 programs, including `switch.p4`, using our prototype and evaluate the amount of data-plane resource needed by our augmentation on a Barefoot Tofino [16] switch. Our preliminary results demonstrate that even for programs as large and complicated as `switch.p4` with $\sim 10^{34}$ paths, we can track packet's execution path in the data plane using only $\sim$178 bits of meta data and the same number of data-plane stages as the non-augmented program.

2 Ball-Larus for P4 Programs

Ball-Larus encoding is a well-known technique for efficiently profiling execution paths in software [2]. For loop-free programs, i.e., programs whose control flows graphs (CFGs) are directed acyclic graphs (DAGs), it can encode all N paths of a program in a single $\lceil log(N) \rceil$-bit variable. More specifically, it labels every edge in the DAG, i.e., transitions between program statements, with an integer. The pseudo-code presented in the original Ball-Larus paper is included in figure 1. The computation time for a DAG with E

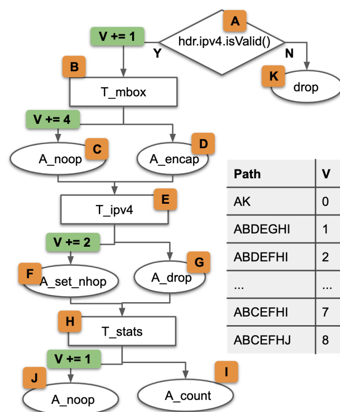
```
for each Node V in reverse topological order:
     if V is leaf node:
          NumPaths[V] = 1
     else:
          NumPaths[V] = 0
          for each edge E = V → W:
               BL[A] = NumPaths[V]
               NumPaths[V] += NumPaths[W]
```

Figure 1: Pseudo code to perform Ball-Larus algorithm included in original paper [2]. After performing, the dictionary BL will contain the update for each edge.

edges and V nodes is proportional to $O(E+V)$, as a reverse topological ordering can be implented as a simple post-order traversal of the DAG. As the input transitions from one statement to the next in the program, the integer label assigned to that transition is added to the variable that is tracking the input's execution path. When the program finishes processing the input, the value in the path variable is a number between 0 and N, uniquely identifying the path the input took through the program.

With an observed path variable number in hand, it is easy to recreate the exact path taken through a DAG. Edge updates are placed to cover continous ranges of path labels, which essentially makes the DAG into a search tree. This means, by performing simple comparisions starting from the root, we can work our way down through the DAG to discover the exact edge transitions that must have taken place to result in our obersved path variable number. In essense, this process is similar to performing a binary search, but where each node can have more than two outgoing edge transitions, thus requiring more than one comparision at each vertex.

Ball-Larus encoding is a promising fit for tracking packet execution paths in P4 programs. First, P4 programs are restricted to be loop-free to allow for their efficient implementation on programmable switches. Thus, given a P4 program, we can construct a DAG representing the program's CFG and run the Ball-Larus algorithm on it. Moreover, programmable switches allow P4 programs to define per-packet meta-data variables carried with the packet throughout its processing, and to define actions to update them using simple arithmetic operations. Thus, once the Ball-Larus algorithm assigns labels



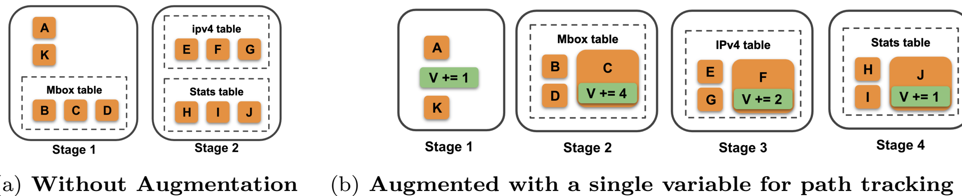
Path	V
AK	0
ABDEGHI	1
ABDEFHI	2
...	...
ABCEFHI	7
ABCEFHJ	8

Figure 2: The control flow graph of an example P4 program with edges labeled by the Ball-Larus algorithm.

to the edges in the program's DAG, we can augment the program with a meta-data variable to track the execution path, and extra actions to update its value on the DAG's transitions based on the labels.

Running example. Consider the control flow graph of the simple P4 program in figure 2. It first checks whether it has received a valid IPv4 packet (node A in the CFG). If so, it first applies the `T_mbox` table (node B), which has two actions: `A_encap` (node D) tunnels packets towards sensitive destination IP addresses specified in the table rules to a middlebox for further analysis, and `A_noop` (node C) simply lets other packets through. Next, the program applies the `T_ipv4` table (node E), which performs a longest prefix match on the packet's final destination and either sets the address of its next hop in `A_set_nhop` (node F) or drops it if it is not matched (node G). Finally, the program applies the `T_stats` table (node H), which has two actions as well: `A_count` (node I) to count certain destination IP addresses, specified in the table rules, and `A_noop` (node J) to let other packets through. Non-IPv4 packets are simply dropped (node K).

If we run the Ball-Larus algorithm on this DAG, it will mark edges $A \rightarrow B$ and $H \rightarrow J$ with number 1, edge $B \rightarrow C$ with 4, $E \rightarrow F$ with 2, and all other edges with 0. As shown in figure 2, adding up the numbers on the edges along each of the nine different paths in the CFG will lead to a unique number



(a) **Without Augmentation** (b) **Augmented with a single variable for path tracking**

Figure 3: Mapping of CFG nodes in our example programs to pipeline stages without augmentation for path tracking (3(a)), and when augmented with a single variable for tracking paths (3(b)).

between 0 and 8 which uniquely identifies that path. Now, to track each packet's execution path in the P4 program, we augment the program with (i) a meta-data variable, called V in the figure, to track the path ID, and (ii) an extra action on any transition in the CFG that is assigned a non-zero label, i.e., $A \rightarrow B$, $B \rightarrow C$, and $E \rightarrow F$, to simply add the value of the label to V. For transitions that are between a table and its actions, i.e., $B \rightarrow C$ and $E \rightarrow F$, we cannot add an extra action on the transition. Instead we can augment the table action itself to perform the addition.

With this augmentation in place, when the program finishes processing a packet on a switch, V contains the unique identifier for the path the packet has taken through the program. The path identified shows precisely which CFG nodes, i.e., conditionals, tables, and actions, the packet has hit in this switch. However, as we discuss next, such augmentations can cause non-negligible overheads when applied to real-world P4 programs.

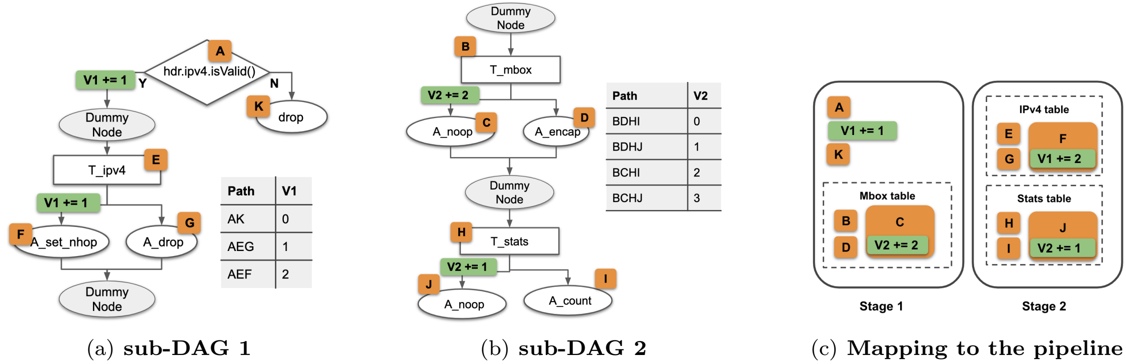
3 Challenges

By tracking all N paths of a program in a $\lceil log(N) \rceil$-bit variable, Ball-Larus has minimal overhead in terms of the amount of per-packet meta-data it needs in the data plane. However, when applied to P4 programs, using a single variable to keep track of the execution path has two negative implications.

Addition on large operands. First, for large P4 programs with many paths, such as `switch.p4`, the size of meta-data variable V that keeps track of the path ($log(N)$) can get as large as a few hundred bits. Existing programmable data planes, however, cannot perform arithmetic operations on operands larger than 64 bits in a single stage. Thus, performing addition on a few-hundred-bit-wide variable would have to span multiple stages. Ball-Larus encoding requires multiple such additions, one on *every* edge in the CFG that has a non-zero label. Thus, for large P4 programs, it can significantly increase the number of pipeline stages required for the augmented program in the data plane.

Extra data dependencies. Second, all the augmented actions, and their corresponding tables, would have extra data dependencies with each other as they all update the same path variable. This can cause the augmented P4 program to use up more stages on the data plane compared to the original one. Consider `T_ipv4` and `T_stats` for instance. Without any augmentation, they have no data dependencies as both just read the destination IP and do not write to it. Thus, as shown in figure 3(a), both tables and their corresponding actions can reside in the same stage when the program is compiled and installed on the switch. After augmentation, however, both `A_set_nhop` from `T_ipv4` and `A_noop` from `T_stats` update V, and therefore, as shown in figure 3(b), can no longer be placed on the same stage. Similarly, the extra action that increments V on transition from node A to node B cannot be placed on the same stage as `T_mbox` and its actions.

Thus, the augmented program needs four stages in the data plane, two more than the original program: A, K, and the action incrementing V on $A \rightarrow B$ all reside on the first stage, `T_mbox` and its actions are on the second stage, `T_ipv4` and `T_stats` and their actions each get their own stage as they were both dependent on `T_mbox` before and are now dependent on each other as well. As the number of pipeline stages on existing switches is typically small (¡32),



(a) **sub-DAG 1** (b) **sub-DAG 2** (c) **Mapping to the pipeline**

Figure 4: The example program partitioned into two sub-DAGs (4(a) and 4(b)), the execution path of each tracked by a different meta-data variable. 4(c) shows how the program augmented with multiple variables is mapped to the same number of stages as the non-augmented program in figure 3(a).

these extra dependencies quickly become problematic for large programs such as `switch.p4` that have hundreds of mostly-independent tables and heavily rely on the compiler placing multiple tables on the same stage to fit in the data plane.

4 Multi-Variable Path Encoding

To make Ball-Larus feasible for large P4 programs, we make the following observation: if we break-up the program's CFG into multiple sub-DAGs, we can concurrently track the execution path in each sub-DAG using *independent* meta-data variables. This helps alleviate both of the challenges in §3. First, each sub-DAG has fewer paths compared to the original DAG. Thus, its path variable can potentially stay within the bit-width limits of arithmetic operands in each stage. Moreover, augmented actions and their corresponding tables in different sub-DAGs update different path variables. As such, they can co-exist in the same stage and no longer need to span across multiple stages.

The partitioning problem. Given a DAG D, we want to find K sub-DAGs $D_1, \cdots, D_K$ that respectively have $P_1, \cdots, P_K$ paths tracked by variables $V_1, \cdots, V_K$, such that (i) the bit-width of each V_i, i.e., $len(V_i) = \lceil log(P_i) \rceil$ is within the limits of arithmetic operands in programmable switches, and (ii) once the program is augmented to track the execution path of these K sub-DAGs, it still uses the same number of data-plane stages as the non-augmented program. Note that the tuple $(V_i, \cdots, V_K)$ still uniquely identifies the execution path throughout the entire program.

Running example. Suppose we break up the CFG in figure 2 into two sub-DAGs as shown in figures 4(a) and 4(b). The first sub-DAG contains the conditional (node A), the drop action (node K), and table `T_ipv4` and its actions (nodes E, F, and G) while the second sub-DAG contains the rest, i.e., table `T_mbox` and `T_stats` and their actions. In each sub-DAG, connected sets of the nodes from the original DAG that are not present are replaced with dummy nodes. We run Ball-Larus independently on each sub-DAG to mark the edges with labels, and use two separate meta-data variables to track the execution path in each sub-DAG: V_1 will updated on transitions in the first sub-DAG, and V_2 is updated on transitions in the second sub-DAG.

This partitioning satisfies our conditions: V_1 and V_2 track 3 and 4 paths, respectively, and each need two bits. Thus, they are within the bit-width limits of arithmetic operands on programmable switches. Moreover, there is no extra dependency between `T_ipv4` and `T_stats` as each are updating a different path variable. Similarly, there is no extra dependency between `T_mbox` and the action updating the path variable for the edge between A and B. Thus, as depicted in figure 4(c), the augmented program

Programs	Program Statistics				Path Encoding Statistics			
	Paths (N)	Tables	Stages	Actions	Path Vars (K)	Added Actions	Added Metadata (bits)	
							our approach	optimal ($\lceil log(N) \rceil$)
tna-action-selector.p4	6	2	2	6	1	0	3	3
source-routing.p4	5	1	7	6	3	3	3	3
tna-multicast.p4	36	6	4	15	2	2	6	6
fabric-bng.p4	1.01×10^9	67	11	73	20	35	46	30
simple switch.p4	1.76×10^9	35	7	157	8	11	47	31
switch.p4	1.75×10^{34}	157	12	307	21	71	178	114

Table 1: A summary of our benchmark programs and the extra data-plane resource needed for multi-variable path encoding.

can be mapped to two stages, not using any extra stages compared to the original program.

Choosing K. Setting a value for K is not straightforward as the benefits of partitioning come at a cost. After partitioning, the total number of bits used for path encoding across all sub-DAGs is $\Sigma_{i=1}^{k} \lceil log(P_i) \rceil$. Depending on the partitioning, this can be larger than the optimal $\lceil log(N) \rceil$ that is achievable without partitioning and using a single variable for tracking the execution path. This can happen for two reasons. First, not all combinations of paths in different sub-DAGs construct a valid execution path in the entire program. For instance, in our example in figure 4, V_1 can track AK and V_2 can track $BDHI$, but their combination, i.e., $(AK, BDHI)$, will never happen in the program as a whole. More generally, $(V_1, \cdots, V_K)$ encodes $P_1 \times P_2 \times \cdots \times P_K$ paths which, due to partitioning, can become larger than N, the total number of valid paths in the program. Second, suppose we manage to partition the original DAG such that $P_1 \times P_2 \times \cdots \times P_K$ is equal to N, for instance by using a larger K and partitioning the program into more sub-DAGs. Then, $\lceil log(N) \rceil$ will be equal to $\lceil \Sigma_{i=1}^{k} log(P_i) \rceil$, and $\Sigma_{i=1}^{k} \lceil log(P_i) \rceil$ can become K bits larger than $log(N)$ due to rounding.

To find a suitable value for K, we exploit the mapping of tables and conditionals in the original non-augmented P4 program to the switch pipeline. More specifically, suppose T_s denotes the set of tables and conditionals that are mapped to stage s when we compile the original P4 program to the switch. This information is available from the output of P4 compilers for existing programmable switches. Suppose T_{max} is the size (in number of tables) of the largest T_s. If we set K to T_{max}, it is possible to assign all tables and conditionals in the same T_s to different sub-DAGs. As a result, after encoding, their augmented actions will not be dependent on each other, and therefore, they will remain on the same stage. Thus, the augmented program will use the same number of data-plane stages as the non-augmented program[1].

The Optimization Problem. To decide how to assign tables and conditionals in each T_s to the K sub-DAGs, we use an integer linear program (ILP). The ILP takes the CFG and T_ss as input, and outputs a_{ij}, which is set to one if node i is assigned to sub-DAG j, and is zero otherwise. Here, a node is either a table or a conditional (which is treated similar to a table by existing programmable switches) together with its actions. The objective is to minimize $\Sigma_{i=1}^{K} v_i$, where v_i is the number of bits required to track all the paths in sub-DAG i.

The first set of constraints are of the form $0 \leq v_i \leq MAX_W$, where MAX_W is the maximum number of bits allowed in arithmetic operands in each stage on the switch. Next, as discussed above, for each stage s and sub-DAG j, we ensure that only one node from T_s is assigned to sub-DAG j using the following constraint: $\Sigma_{i \in T_s} a_{ij} = 1$. For each node i, we have constraints of the form $\Sigma_{j=1}^{K} a_{ij} = 1$ that it is assigned to only one sub-DAG. Finally, suppose $p_i \in R$ is the

[1] Unless the augmented actions of the same T_i do not fit in the same stage anymore due to the extra ALUs used for addition. We have not observed this corner case even in our most complicated evaluated programs (§5). But, even if it happens, our approach still correctly tracks execution paths and the augmented program just spans over one or few extra stages.

log of the number of outgoing edges of CFG node i. We estimate log of the number of paths in sub-DAG j as $\Sigma_{i=1}^{N} p_i \cdot a_{ij}$, and relate that to v_i using the constraint $\Sigma_{i=1}^{N} p_i \cdot a_{ij} \leq v_j$.

5 Evaluation

Prototype. Using Barefoot Tofino [16] as target, we have implemented a prototype that takes a P4 program as input and outputs an augmented P4 program that can track packet execution paths. More specifically, given an input program `prog.p4`, we first compile the program using the Tofino compiler to extract the program's control flow graph, and the mapping from the the program's tables and conditionals to the pipeline stages, both useful by-products of the compilation process. We have developed a python script that takes the CFG and the mapping as input, and partitions the CFG into multiple sub-DAGs by solving the optimization problem discussed in §4 using the `puLP` package. The script then runs the Ball-Larus algorithm on each sub-DAG to obtain the integer labels for each transition. Even for our largest example programs, we found that the time required to solve the optimization problem and obtain the integer labels, takes under 5 seconds. A reference to the entire system is provided in figure 5.

Augmenting the program. Next, we augment `prog.p4` in the following way. First, we add a meta-data variable `v_i` to track the execution path of each sub-DAG. Second, for each action of each table that has a non-zero integer label on its incoming edge, we add a single instruction to add the label to the path variable for the corresponding sub-DAG. Finally, in Ball-Larus encoding, only one of the two outgoing edges of each conditional will have a non-zero label. Thus, for each conditional, we add an action on the branch with the non-zero label to update the path variable for the sub-DAG assigned to that conditional accordingly.

Integration with existing compilers. An ideal starting point for augmenting programs is to modify its intermediate representation (IR) typically used by compilers during the compilation process. The program's IR is its parsed representation, typically stored in a graph. It is a more detailed version of its control graph with each node corresponds to some piece of syntax in the original program. During the compilation process, the IR goes through a sequence of passes that transform it to a more optimized version, some performing target-independent optimizations such as removing unreachable pieces of code, and others transforming the IR to better fit on the specified target. Our augmentations can be another pass in the process, adding extra nodes in the graph for the meta-data variables and actions required to keep track of the execution paths in the computed sub-DAGs. In fact, for our initial prototype, we implemented our encoding as an IR pass in P4's open-source compiler [13]. However, we did not have access to add passes to Tofino's compiler. Thus, our prototype for Tofino parses the input program line by line and injects the extra meta-data variables and actions directly into the program's code.

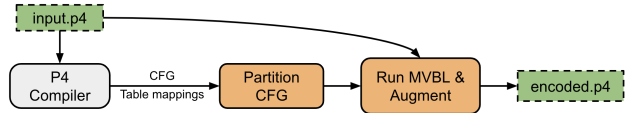


Figure 5: End to end system to augment an input P4 program with multi-variable Ball-Larus path encoding.

Benchmark Programs. We augment six P4 programs of varying size and complexity, listed in table 1, using our prototype and evaluate the amount of data-plane resource needed by our augmentation on a Tofino switch. The first three, two taken from the examples included with the Tofino compiler and one from P4 tutorials [1], are smaller, with a few tables and actions, and not more than a few tens of execution paths. The last three are much larger and more complicated with hundreds of tables and actions and more than billions of paths: `fabric-bng.p4` and `switch.p4`, specifically, are production-quality programs implementing a Broadband Network Gateway (BNG) and a standard switch, respectively.

Data-Plane Overhead. Table 1 summarizes the extra data-plane resources caused by program augmentation. Our partitioning strategy (§4) ensures no extra dependencies between tables and conditionals that are mapped to the same stage after the compilation of the original program. Thus, as expected, augmented programs do not use any extra stages compared to the original programs.

There is a slight increase in the number of actions in the augmented program. This is expected: as discussed above, while we can augment existing table actions to update path variables, we have to add extra actions to perform one addition to update path variables for transitions out of conditionals. These extra light-weight actions, however, do not stop our augmented benchmarks from fitting in the switch. This is because the mapping from tables and conditionals to the stages does not change after augmentation and the extra actions merely use the extra ALUs in the stages that were previously unused by the original program.

Finally, recall from §4 that while partitioning makes the augmentation feasible for large programs, both in terms of the number of stages and complexity of addition operations to path variables, it comes at the cost of using extra bits to encode paths that cannot occur in the program. Using extra bits for encoding execution paths means leaving fewer bits in the PHV for the original P4 program. Thus, we use our benchmark programs to ensure that our multi-variable approach can still efficiently encode the execution paths for the program without exhausting the available PHV bits.

More specifically, as shown in Table 1, we compare the optimal number of bits required to encode all N paths of the program, i.e., $\lceil log(N) \rceil$, to the number of bits used by our multi-variable encoding, i.e., the sum of the sizes of the K path variables tracking execution paths in the K different partitions of the program. The most complicated program, `switch.p4`, can be encoded with 178 bits, and the other two large programs can be encoded with 47 and 46 bits. For comparison, KeySight, the closest related work that can encode such information [18] uses 32K bits of metadata for encoding in the worst case (see §7). Moreover, recall that the optimal encoding would require a single Ball-Larus variable, creating extra data-dependencies between otherwise independent tables when their actions update that variable. The naïve encoding, which is to update a corresponding flag bit on every action in the program, is similarly prohibitive. In existing programmable switches, these approaches only work for encoding the small and simple input programs, and do not scale efficiently to larger real-world programs such as `fabric-bng.p4` and `switch.p4`. Thus, we believe our multi-variable encoding overhead, and its difference from optimal, is not considerable given its significant benefits in terms of number of stages and action complexity and as the amount of per-packet meta-data on existing switches is a few thousand bits.

6 Usecases: Detecting and Localizing Bugs

We have so far described and evaluated an effecient system to implement path profiling for P4 programs which will work on hardware targets overcoming the numerous restrictions.

Even without any further instrumentation in place, the multi-variable Ball-Larus path encoding could prove a useless tool to P4 developers. Every packet that it processed by the encoded program will be tagged with the path it took, akin to debugging print statements used for control flows in conventional code. In addition, in creating the encoded program, we provide the user with the expected CFG for their code and can enumerate over all possible paths. P4 programs are dense with conditional executions due to being dominated by match action units, making it possible to lose track of the control flow while coding. For a novice P4 developer, the path encoding instrumentation would serve as a sanity check where they can verify if the expected paths in the P4 code allign with the vision they had for the program.

As discussed, knowing the set of Ball-Larus variable values output for a packet as well as the CFG of the P4 program will be enough to recreate the path those variables represent. This is just the set of hit match-action units and conditionals. Note that our technique will only tell which units were hit, not the specific order they were processed in the hardware. A compiler bug occurs when the observed path during run-time for a set of packets is not equal to the expected path on the annoted CFG. If there is a mismatch, we can conclude that the compiler introduced a bug when mapping resources to the hardware pipeline, which manifests as unexpected behavior for some subset of packets. Thus, we will need some mechanism in place to compare the expected paths

versus observed paths for packets to easily determine if any mismatches occur.

In the following section, we discuss the benefits and limitations of three approaches to detect mismatches, which would allow the user to localize bugs.

- Expected path in data-plane lookup table (Full coverage)
- Run time path assertion checks (Static coverage)
- Sample and check paths offline (Best attempt coverage)

Including ground truth in data-plane. If the expected program execution path for packets is located in the data plane, we can compare a packet's evaluated execution path in the pipeline vs the expected path. Doing this check for every packet that traverses the switch effectively treats every incoming packet as a test packet. This allows the user to either be confident that all production traffic is being properly handled, or at least able to identify which parts of the execution path are corrupted, providing full debugging coverage over all possible paths in the program.

Packets are typically processed in a switch via header transformations. In a stateless program, two packets sharing identical header fields will follow the same execution path in a program. Thus, we can introduce a "ground truth" table to the pipeline of augmented programs, wherein the look-up keys are the pertinent header fields, and the values are the expected program execution path, calculted beforehand. Unfortunately, certain hardware limitations can make calculating and maintaining the ground truth table prohibitively costly for complex programs.

An intuitive process to calculate the expected execution paths is through symbolic execution, which discovers the output corresponding to every input to the program. We can utilize symbolic execution of P4 programs to create a set of "packet classes", consisting of ranges of the headers field values that influence the execution path. Members of the same packet class are expected to take the same execution path.

One of the limitations of this approach is the time required for symbolic execution. Symbolic execution is an exhaustive process that tests all possible execution paths, which will not scale to large programs such as switch.p4. Previous work on symbolic execution in P4, such as in p4pktgen [10], show that it takes greater than 30 mins to perform on `switch.p4`. In addition, the resulting packet class information may be to too large to fit into the data plane. As `switch.p4` has $\sim 10^{34}$ paths, we expect there to be at least as many packet classes, and it is improbable that a table of that size can fit on modern switches. For simple programs however, it may be the case that the ground truth table created by symbolic execution can be inserted into the pipeline and matched with during run-time.

One solution to ensure that the ground truth table does not contain too many entries is to populate it reactively during run-time. In this case, the ground truth table in the pipeline contains a subset of all the expected execution paths. When a packet enters and does not match with the table, we send the packet, along with its evaluated execution path, to the controller to perform symbolic execution on. The packet's packet class and expected execution path can then be installed onto the table by the controller.

Another limitation to the aproach via symbolic execution is that is does not take any table rules into account. During compilation of a P4 program, we only know which keys are matched upon in any given table, as well as all the possible P4 actions that can be taken from this table after matching. We know that some action will happen dependant on the table's keys, but we do not know which action will be selected just yet. The user installs rules into tables after the fact, which determine the specific action that specific key values take. This implies that a reactive solution may be necessary, as the controller will need to populate the ground truth table as rules change.

A final limitation to symbolic execution is that the number of keys needed to identify a packet class may become too large. This becomes a problem when we preform the lookup in the ground truth table. In the naive case, a single packet class will need to represent every header field that is matched by any table in the program, as all these fields will jointly determine the execution path. Similar to the previous argument, it is unlikely this will scale to hardware for large pro-

grams. The key space for the ground truth table could be at least as long as packets' header space. In addition, this makes the problem much harder to solve for packet's with header stacks.

Run time path assertions. In many cases, a network operator may know that paths must be malformed through packet analysis, although it may be difficult to identify exactly how this manifested in the hardware. For situations such as this, we could use run time assertion checks on the path variables to ensure that only valid packets are on specific paths. This approach requires you to statically specifcy some path suspected of contiaining a bug to be checked in the data plane. This can be thought of as a smaller scale version of the full coverage mechanism discussed in the previous subsection, wherein the ground truth packet class for the specified path will be computed using symbolic execution. Then, we would know what set of headers should result in packets to take that path (i.e. the packet class). We can perform two different kinds of assertion checks with this information during run time. One would be to ensure that every packet part of the computed packet class takes the desired path, as seen in the path variables at the end of the pipeline. The other would be to ensure that only packets part of the packet class take the path we wish to verify. Any mismatch will be flagged. The user can then compute the expected path offline, which will provide explicit information of how the the suspected path is malformed, and which set of packets may trigger the bug.

For example, consider a P4 program containing an ACL table. The rule for the ACL table may be to deny non-IPv4 packets, by performing the drop action. Packets are classified as IPv4 through a field in the header. An operator may notice that non-IPv4 packets are incorrectly making it through the switch without being denied. A run time path assertion for this problem would be to check the path of non-IPv4 packets at the end of the pipeline, and send any which do not take the ACL deny action to the controller. Due to the nature of the Ball-Larus edge increments, this can be performed as a simple range check on our Ball-Larus variables. This will allow the user to build a set of buggy packets and determine their expected packets, which will provide valuable debugging insight.

Sample and check paths offline. For large programs, obtaining a full coverage solution strictly in the data plane is difficult. A solution to this would be to verify the evaluated paths outside of the dataplane, and provide a best attempt at coverage using a sampling mechanism, to avoid excessive computational costs from exhaustive sampling. Instead of treating every packet as a test packet for mismatches, we use a smaller subset. Then we would be able to sample packets and check against the expected path independent of run time at the collector, where the observed path could be verified for correctness using a software model of the program. An ideal sampling strategy would have near full coverage in verifying paths, while limiting the number of unnecessary checks performed at the collector. In this way, we would build a set of buggy paths at the collector, identifying the sampled packets that have a mismatch between observed path and expected path.

Different sampling mechanisms have different tradeoffs. At the simplest level, the strategy could be to utilize a simple probability with which we wish to sample packets randomly. Unfortunately, this method is unlikely to find any bugs present in "mice" paths, which do not trigger often. Instead, we would do frequent checks for packets that take heavy hitter paths. Resampling from heavy hitter paths is usually a waste of resources, as we would have already verified that path using a previously sampled packet.

To remedy this, we can use a smart data structure to keep track of the heavy hitter paths during run time. Then, we would only choose to sample packets from paths that are not already present in the heavy hitter structure. To this end, we can utilize a hash-based cache or bloom filters to ensure that our observed paths are not oversampled from. The index to these hash structures will be dependant on the observed path variables as well as the IP flow tuple of the packet to be sampled. These values will be hashed together to uniquely idenitfy individual IP flows taking a unique path in the program. As a packet finishes the pipeline of the switch, it performs a lookup in our sampling data structure to check if the same path/tuple combination was recently sampled. Packets that were not recently sampled will be

sent to collector to be verified against the software model.

Two limitations for this approach appear in the form of false positives and false negatives. False positives occur due to oversampling paths previously checked to be correct, and are considered a waste of resources. False negatives occur when we do not sample an incorrect, buggy path. If an expected path p_e is buggy, it would manifest as some packet having an incorrect observed execution path during run time, say p_o. The packet would then be indexed with p_o in the sample data structure, and would only be sampled in relation to the presence of packets in the incorrect path. There will be no knowledge of the exepected path p_e in the data plane. Thus if a path bug causes the path id to become an incorrect, yet recently sampled path, the bug would be missed. In the same regard, careful care must be taken to tuning the refresh rate as well as size of our sampling structure in relation to the number of overall paths, as this impacts the rate of hash collisions, which can be another source for false negatives causing incorrectly evaluated path could collide with a recently sampled path.

7 Related Work

Data-Plane Post-Cards. Previous work 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 help debug network problems. NetSight's post-card [6] 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. KeySight [18] copies every packet field that is read and written to in each match-action table into the post-card, but at the cost of using ∼32K bits of meta-data for large programs such as `switch.p4`. In our approach, on the other hand, the "post-card" contains the unique identifier for the path the packet has taken through the program and uses only a few hundred bits for `switch.p4`.

Test Packet Generation. Previous work such as ATPG [17] and P4pktgen [10] study automatic generation of test packets from specifications of network devices. Test packets cannot exercise every packet processing scenarios for specifications of real-world switches such as `switch.p4`. Our approach is complementary to these efforts since, by enabling operators to trace packet's execution path on the data plane at run time, it enables them to detect and localize bugs on execution paths not exercised during testing.

Verification. Recent work [14, 8, 5, 9] explores automatic verification of various properties about P4 programs using techniques such as static analysis or symbolic execution. However, not all properties can yet be verified by existing tools and these tools still operate at the level of the P4 program. That is, they can verify if the software logic is bug free for a specific set of bugs. However, the hardware mapping for the program may have been incorrectly performed by the compiler during compilation. Thus, these tools cannot be used to find compiler bugs, especially for compilers that are not open-source. As a result, our approach can complement these works by enabling the detection and localization of bugs in the P4 programs that are missed by the verification tools, or bugs in the compiler or the controller installing rules on the data plane.

8 Conclusion

We propose a useful data-plane primitive for detecting and localizing bugs as they occur in real time: tracking each packet's execution path through the P4 program, i.e., the sequence of tables hit and the actions applied, directly in the data plane. In our ongoing work, we plan to design, implement, and evaluate end-to-end monitoring and debugging systems, both in the data plane and control plane, that use the path information to detect and localize bugs in P4 programs, the compiler, and the controller.

References

[1] P4 Tutorials. https://github.com/p4lang/tutorials/. Accessed: November 2019.

[2] T. Ball and J. R. Larus. Efficient Path Profiling. In *International Symposium on Microarchitecture*, 1996.

[3] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford. A NICE Way to Test OpenFlow Applications. In *NSDI*, 2012.

[4] Cisco Catalyst 9300 Programmable Switches. https://www.cisco.com/c/en/us/products/switches/catalyst-9300-series-switches/index.html. Accessed: May November.

[5] L. Freire, M. Neves, L. Leal, K. Levchenko, A. Schaeffer-Filho, and M. Barcellos. Uncovering Bugs in P4 Programs with Assertion-Based Verification. In *SOSR*, 2018.

[6] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown. I Know What Your Packet Did Last Hop: Using Packet Histories to Troubleshoot Networks. In *NSDI*, 2014.

[7] EX9200 Programmable Switches. https://www.juniper.net/us/en/products-services/switching/ex-series/ex9200/. Accessed: November 2019.

[8] 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 *SIGCOMM*, 2018.

[9] N. Lopes, N. Bjørner, N. McKeown, A. Rybalchenko, D. Talayco, and G. Varghese. Automatically Verifying Reachability and Well-Formedness in P4 Networks. *MSR Technical Report, MSR-TR-2016-65*, 2016.

[10] A. Nötzli, J. Khan, A. Fingerhut, C. Barrett, and P. Athanas. P4pktgen: Automated Test Case Generation for P4 Programs. In *SOSR*, 2018.

[11] Advanced Programmable Switch. https://www.stordis.com/products/. Accessed: November 2019.

[12] P4 Language Consortium. https://p4.org/. Accessed: November 2019.

[13] P4_16 Reference Compiler. https://github.com/p4lang/p4c. Accessed: November 2019.

[14] R. Stoenescu, D. Dumitrescu, M. Popovici, L. Negreanu, and C. Raiciu. Debugging P4 Programs with Vera. In *SIGCOMM*, 2018.

[15] P4_16 Reference Compiler: switch.p4. https://github.com/p4lang/switch/tree/master/p4src. Accessed: November 2019.

[16] Tofino, World's Fastest P4-Programmable Ethernet Switch ASICs. https://www.barefootnetworks.com/products/brief-tofino/. Accessed: November 2019.

[17] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown. Automatic Test Packet Generation. In *CONext*, 2012.

[18] Y. Zhou, J. Bi, T. Yang, K. Gao, C. Zhang, J. Cao, and Y. Wang. KeySight: Troubleshooting Programmable Switches via Scalable High-Coverage Behavior Tracking. In *International Conference on Network Protocols (ICNP)*, 2018.