bf4: towards bug-free P4 programs

Dragos Dumitrescu Radu Stoenescu Lorina Negreanu Costin Raiciu
University Politehnica of Bucharest
firstname.lastname@cs.pub.ro

ABSTRACT

Recent verification work has made advances in finding bugs in P4 programs before deployment, but it requires that the programmer specifies table rules that are possible at runtime[32, 24, 27]. This imposes a specification burden on the programmer, while at the same time failing to guarantee that bugs will not be inserted at runtime by faulty controllers.

We present bf4, a novel verification approach for P4 programs that uses a mix of static verification, code changes and runtime checks to ensure that the deployed P4 program is bug free. To achieve this, bf4 uses static analysis to find all possible bugs in the P4 program; for each possible bug, bf4 attempts to find predicates that, when applied to table rules inserted by the controller, make that bug unreachable. If such predicates do not exist, bf4 can change the P4 code and re-run the procedure above.

We applied bf4 to a wide range of P4 programs; for all these, bf4 is able to generate controller assertions and propose fixes that guarantee no controller-induced bug is reachable. At runtime, bf4 checks that the controller does not insert faulty rules; when it does, it throws an exception which helps troubleshoot the bug.

CCS CONCEPTS

• **General and reference → Verification**; • **Networks → Network reliability**; **Programmable networks**;

KEYWORDS

Network dataplane verification, programmable networks

ACM Reference Format:
Dragos Dumitrescu Radu Stoenescu Lorina Negreanu Costin Raiciu. 2020. bf4: towards bug-free P4 programs. In *Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication (SIGCOMM '20), August 10–14, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3387514.3405888

1 INTRODUCTION

P4 and NPL are languages which can be used to program hardware dataplanes. These languages and the corresponding programmable targets enable unprecedented network flexibility, promising an ever-evolving set of network functionalities at hardware speeds. On the downside, dataplane programs written in these languages can exhibit multiple types of faulty behaviors (bugs) that include accesses to uninitialized header fields, header stack underflows or overflows, out-of-bounds regiester accesses, undefined packet forwarding behaviour, and so forth [32, 24].

Programming dataplanes is far from trivial, because programmers must separately describe the dataplane, the control plane as well as their interaction. As a result of this complexity, bugs have been found even in trivial programs [32]; left unchecked, such bugs could lead to incorrect network behaviour and potentially be exploited by malicious adversaries [14]. It therefore crucial to find and remove such bugs from production networks.

Ideally, we should find all bugs before deployment, but this is rarely possible. Only when the P4 bug is always reachable (i.e., there exists an input packet which triggers the bug) independent of the table entries, verification before deployment works well, finding all such *dataplane bugs* in seconds / minutes [24, 32, 27].

In most cases, however, the bugs are only reachable when faulty table entries are inserted at runtime by the controller—we call these *controller-induced bugs* (or controller bugs, for short). Unfortunately, the controller is general-purpose code and automatically predicting controller-induced bugs means predicting, at deployment, the table entries the controller will output at runtime—this is impossible in the general case.

To side-step this problem, existing verification works ask the programmer to specify before deployment which table entries are possible at runtime, and then use this information to find reachable bugs. This approach **adds significant burden to the dataplane programmer** (e.g., 770 lines of annotations for a 6KLOC program, for one bug type [24]) and **does not guarantee the program is bug-free**: at runtime, the controller may still insert faulty rules.

We propose bf4, an end-to-end verification approach for P4 programs that guarantees deployed P4 programs are bug-free, regardless of the source of the bug. bf4 works as follows:

(1) At compile-time, it finds all possible bugs; some of these bugs, however will not be reachable in practice, so simply presenting them to the programmer is not useful.
(2) Next, it automatically infers controller annotations—these are predicates to be obeyed by the rules inserted by the controller to avoid controller-induced bugs, removing the need for manual annotations required by existing tools.
(3) Assuming annotations hold at runtime, most bugs will now become unreachable. However, if there are still bugs reachable (e.g., dataplane bugs), bf4 proposes code changes to make the remaining bugs unreachable. If the changes are accepted by the programmer, repeat step (2).
(4) At runtime monitor the rules inserted by the controller, raising an exception when they do not satisfy the annotation.

We have implemented bf4 as a backend for p4c, the P4 compiler suite. bf4 runs in minutes even on large programs and is easy

SIGCOMM '20, August 10–14, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-7955-7/20/08…$15.00
https://doi.org/10.1145/3387514.3405888

```
control ingress(){ //...
 apply {
   if_info.apply();
   nat.apply();
   if (meta.meta.do_forward == 1w1){
     ipv4_lpm.apply();
     forward.apply();
   }
}}
```

```
table nat {
  actions = { drop();//...
    nat_hit_int_to_ext();}
  key = { //...
    hdr.ipv4.isValid(): exact
    hdr.ipv4.srcAddr  : ternary}}
action nat_hit_int_to_ext(a,p) {
  meta.meta.do_forward = 1w1;
  meta.meta.ipv4_sa = a;//...}
```

```
table ipv4_lpm {...
  actions = {set_nhop();drop();...}
  key = {meta.meta.ipv4_da: lpm;}
}

action set_nhop(nhop_ipv4, p) {
  meta.meta.nhop_ipv4 = nhop_ipv4;
  standard_metadata.egress_spec = p;
  hdr.ipv4.ttl = hdr.ipv4.ttl - 1; }
```

Figure 1: Running example: a snippet from the simple nat P4 program.

to use: it does not require manual annotations to find the bugs, and it simplifies code changes when bugs are found. We applied `bf4` to many P4 programs, including `switch.p4`. On the latter, `bf4` found 165 bugs are reachable when all table entries are feasible (similar to p4v [24]). After we inferred the necessary annotations, 55 bugs remain reachable. `bf4` automatically generated code fixes that added 26 keys to 14 tables; these are then used to eliminate all 165 bugs. At runtime, `bf4`'s shim filters rules in tens of ms.

2 MOTIVATION

P4 is a language that allows specifying programmable dataplanes which can be deployed on multiple targets including switches (Barefoot Tofino) and smartNICs (Netronome FX). P4 delivers programmability without hurting performance: the Tofino can support Tbps packet processing speeds. NPL is another dataplane programming language similar to P4 that is supported by Broadcom's Trident 4 switching chip.

For brevity, we do not include an introduction to the P4 language here; interested readers may look at the numerous tutorials available on p4.org and elsewhere. In this paper we focus P4-16, on the newest version of the P4 language. Programs written in P4-14 (the previous version) can be converted to P4-16 using the p4 compiler suite and then analyzed by `bf4`. `bf4` does not currently support NPL, but it would be straightforward to extend to do so.

To guide our discussion we rely on the example shown in Figure 1. The code is taken from the simple NAT P4 program available from p4.org, but we only show a few interesting snippets. We show the ingress pipeline (left), together with parts of the two tables and some of the actions defined in those tables (center and right). In the ingress pipeline shown in Figure 1, packets first visit the `nat` table, and then conditionally the `ipv4_lpm` and `forward` tables.

The P4 program only specifies half of the dataplane functionality; the other half is provided by the control plane (typically an SDN controller) at runtime, in the form of table rules. The intention of the programmer of our NAT, not evident from the dataplane code alone, is that packets for ongoing connections should be directly translated by the `nat_hit_` actions when there is already an installed mapping in the table. This is achieved by inserting 5-tuple rules in the nat `table` for known connections and using actions to set the appropriate metadata for the current packet: for instance, in the `nat_hit_int_to_ext` action, `meta.ipv4_sa` is set to the value of the desired IP source address, and the metadata marks that this packet should be forwarded. The metadata information will be used in the egress pipeline to set the new IP address of the packet before it leaves the switch.

When there is no matching hit entry and the traffic is coming from the internal network, the `nat` must forward the packet to the controller which will then insert a 5-tuple hit entry; if the traffic is coming from the external network, it should be dropped. This is what the `nat_miss_` rules are meant to achieve. Finally, all packets that must be forwarded (not destined to the controller) should then pass through the `ipv4_lpm` and `forward` tables.

Note that the functionality of the NAT heavily depends on the table rules, and thus correct functioning hinges on both the dataplane and the control plane being correct.

2.1 Bugs in our example program

P4 programs exhibit a range of faulty behaviors (bugs) that were also highlighted by previous works [32, 24]. The effects of these bugs range from benign to incorrect processing of packets and even malicious exploitation by attackers [14].

A common bug is accessing an invalid header. All possible headers that the P4 program can parse are declared up-front and are in scope throughout the entire processing pipeline. At some point during execution, the packet only contains a subset of all headers, but all headers may be accessed because they are syntactically in scope; when uninitialized headers are read "undefined" values will be returned.

In our example, the `set_nhop` action in the `ipv4_lpm` table accesses the TTL field in the IPv4 header without checking its validity first. A more subtle bug exists in the NAT table: consider a rule where the valid flag is invalid (implying that the `srcAddr` should not be matched) but the mask for the ternary-match `srcAddr` field is non-zero. When a packet without an IP header hits this rule, the pipeline will read the `srcAddr` field from the invalid header and match it against the entry in the table, resulting in undefined behavior.

Another type of bug is not specifying the forwarding behaviour for certain packets (also known as egress_spec not set). This means that forwarding will be target-specific, and may differ from a software to a hardware target. In our example, this bug is triggered on some paths (not shown).

Other types of bugs, not present in our simple example, include out-of-bounds accesses to register arrays, header handling errors such as adding a header that is already valid, popping an empty header stack or pushing a new header in a full header stack, as well as decapsulation errors where live headers are not deparsed on output [32, 24, 27].

3 SOLUTION SPACE

The programmer deploys a P4 program, and at runtime the controller adds and removes rules in the tables of the program, or changes other configuration parameters. A snapshot is the P4 program together with all its active table entries and configuration knobs. The controller triggers changes from one snapshot to the

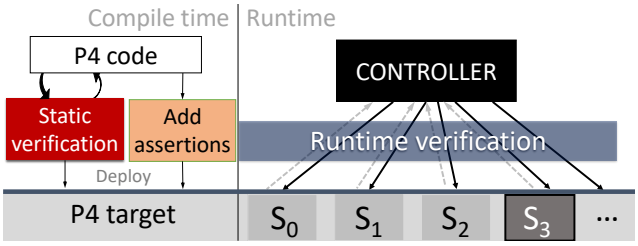

Figure 2: P4 program verification: a landscape

next at runtime.

The goal of our work is to help programmers ensure that P4 programs do not have reachable bugs when deployed for the classes of bugs that we support. This means that no active snapshot may have reachable bugs, ever.

To guide the discussion of possible solutions, Figure 2 summarizes alternative approaches. The ideal solution is to use static verification to find and remove all bugs from P4 programs at compile-time. The trouble with this solution is that it requires predicting, before deployment, all the P4 snapshots that will be active at runtime. Since the controller is general-purpose code, this problem can be reduced to the halting problem, which is undecidable.

To avoid predicting controller outputs, existing verification works either ask the programmer for concrete snapshots of table entries (under-approximate the set of snapshots) or assume all possible table entries are likely (over-approximate). Vera [32] takes the first approach, analyzing concrete P4 snapshots with symbolic execution; it can find a wide range of bugs, but it can be fairly slow for large programs (5-15s to check a snapshot of switch.p4, for each packet type). Vera has however limited coverage: even if one snapshot is bug-free, bugs can exist in other snapshots.

The second approach, taken by p4v [24] and ASSERT-P4 [27], assumes that all entries are possible and reports all the possible bugs; many are false positives, being reachable in snapshots that do not appear at runtime. For instance, in our running example p4v will give an example of a rule for the `nat` table where `ipv4.isValid=0` and the mask for `ipv4` is non zero, meaning that the invalid field will be read from the packet. It is clear that a sane controller should never insert faulty rules where all matching packets trigger a bug.

To reduce false positives, p4v and ASSERT-P4 require the programmer to provide control-plane assertions that limit the range of possible table entries. Both tools assume the assertions hold, and then find the bugs that are still reachable and flag them as dataplane bugs which must be fixed by the programmer. Unfortunately, both p4v and ASSERT-P4 put the annotation burden on the programmer and this may lead them to give up on verification.

Note that, even if the controller assertions guarantee the absence of bugs, there is no guarantee the controller obeys the assertions: static verification alone cannot give any guarantees w.r.t to the absence of bugs, so runtime checks are therefore needed to ensure controller outputs are not buggy. There are two ways to deploy such checks:

(1) Change the P4 program before deployment to catch all packets that trigger a fault and send them to the controller, notifying that a bug exists.
(2) Monitor the controller outputs, rejecting those that would result in buggy snapshots.

In principle, both these approaches can be used standalone to detect bugs at runtime, even without verification at compile-time. However, they are too expensive to use in practice. Using the first approach, we can change the P4 code to add runtime checks before any possible bug. For instance, we can change the set_nhop action as follows:

```
action set_nhop(nhop_ipv4, p) { ...
    if (ipv4.isValid) hdr.ipv4.ttl = hdr.ipv4.ttl - 1;
    else           meta.faulty = 1;  }
```

After the table apply call, we also add code that sends all packets with the faulty metadata set to the controller. In principle, adding assertions to the P4 program in this way can catch many bugs just before they are triggered. The downside is that every change to the control flow results in the program needing more stages; in the simple NAT example, such instrumentation doubles the number of stages needed to deploy the program on the Tofino. For large programs, the instrumented version cannot be deployed at all.

Additionally, some bugs cannot be fixed by changes in the dataplane code alone. This is the case in the `nat` table, where a faulty rule can match an invalid IPv4 header and with non-zero mask for the IPv4 source address. This P4 code is correct, so we don't need to fix it. The controller that injects such an entry is buggy, though, and we will use the second approach to catch such bugs.

To catch controller-induced bugs, we must filter updates from the controller before they are inserted in the dataplane: whenever a table is changed, use existing tools to check for reachable bugs in the new snapshot. If no bugs are reachable, allow the update; otherwise, throw an exception that a buggy snapshot exists. The problem is that snapshot verification will take minutes for large P4 programs, meaning that either (a) all updates will be delayed by said time, creating convergence and stability problems for the control-plane protocols (e.g., BGP), or (b) updates will be allowed and checked reactively, with the danger of deploying buggy snapshots.

A better approach is to use the controller annotations at runtime to filter faulty table rules, instead of verifying entire snapshots: such annotations typically apply to a single table and are independent of the rules already in the table. In our example, it would be trivial to check that `nat` rules don't set `ipv4.isValid=0` and `ipv4.srcAddrMask!=0` simultaneously; such checks can be done in milliseconds, as we show in our evaluation. Today, however, such annotations are generated manually; we generate them automatically instead.

4 BF4 OVERVIEW

`bf4` *is an end-to-end verification solution that guarantees that any active snapshot is not buggy.* `bf4` has two components:

(1) A static verification part that runs at compile-time and automatically derives controller annotations, instead of requiring manual input for programmers, with the goal of making all bugs in the P4 program unreachable. In certain cases, a minority of bugs may still be reachable after the annotations are inferred. In such cases, `bf4` also suggests changes to be made to the P4 program to ensure most/all bugs can be controlled via further annotations. In the rare cases where a bug is still reachable after fixes, it is a dataplane bug that must be fixed by the programmer.

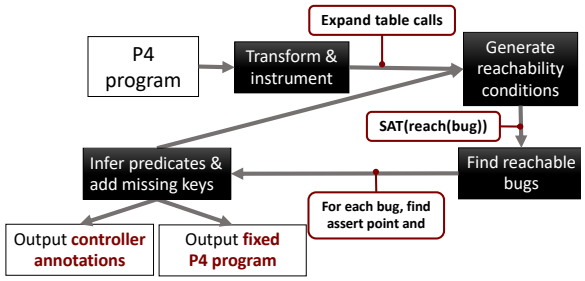

Figure 3: bf4 operation summary.

(2) A runtime component that acts as a shim sitting between the controller and the data plane, quickly checking all rules inserted in the dataplane against the annotations derived at compile time. Whenever a rule does not satisfy an annotation, the shim throws an exception notifying the controller that a bug has appeared; otherwise the rule is inserted, and the snapshot is guaranteed to be bug-free.

In the remainder of this section we describe in detail the static verification part of bf4; for brevity, we refer to it as bf4. The runtime component is described in more detail in §4.4. At compile-time, bf4 aims to find control plane annotations according to these principles:

(1) Inhibit many (ideally all) unsafe behaviors (bugs).
(2) Do not prevent *any* safe behaviors.
(3) No manual annotations required.

We also want bf4 to be practical to use: the time to infer control plane annotations should be similar to compile time.

The static verification part of bf4 can be used as a standalone tool for P4 code verification - much like p4v [24] - and P4 snapshot verification - similar to Vera [32] and P4-NOD [26]. It builds upon classic bounded model checking [10].

The way bf4 works is captured in Figure 3. As all prior works, bf4 first instruments the P4 code, adding assertions where bugs may appear. bf4 then expands table `apply` calls into their abstract semantics in a way similar to p4v, and initially assumes all table rules are possible. bf4 then builds the control-flow graph of the P4 program, generates the conditions that must be satisfied for each bug to be reachable, and uses Z3 solver to check reachability (see §4.1).

Since all table rules are likely, this results in a large number of bugs being reachable; for instance 165 bugs are reachable in `switch.p4`. Having the programmer reason about all these bugs is tedious, yet all prior works rely on the programmer to reduce the space of relevant table rules. Our goal is to *achieve the same task automatically*.

The key insight is that certain table rules are undesirable because all the packets matching them would result in buggy behavior; such table rules should be discarded by the verification tool, and should never be inserted by *any* controller. In our example, consider this rule in the `nat` table:

```
ipv4.isValid==0, ipv4.srcAddr==(192.0.0.0,mask=255/8)
```

where the associated action can be anything. *All packets* that match this rule will have an invalid `ipv4` header, and will trigger a bug because the invalid `srcAddr` field will be read. Since this rule cannot process *any* packet without triggering a bug, we tag it as *faulty* (or *buggy*) and we can safely discard it: it will never be inserted by any sane controller.

```
1       flow_def_nat_t pcn = havoc();
2       pcn.reach = true;
3       if (pcn.isHit) {
4           assert(pcn.key_ipv4_valid == ipv4.isValid() &&
            pcn.key_ipv4_src_addr_value == ipv4.srcAddr &
            pcn.key_ipv4_src_addr_mask);
5           if (pcn.action_run == drop) { drop(); }
6           else if (pcn.action_run == nat_hit_int_to_ext) {
7               meta.meta.do_forward = 1w0;
8               meta.meta.ipv4_sa = pcn.parm_a;//... }}
```

Figure 4: Expanding nat table call

bf4 automatically infers predicates which filter buggy rules (see §4.2); for the example above, bf4 generates the predicate: `ipv4.isValid==0 ∧ ipv4.srcAddr.mask!=0`.

Such predicates are in fact annotations, and we use them for two purposes: first, to reduce the number of reachable bugs in verification, thus easing the burden on the programmer. Secondly, we use them at runtime to filter the rules inserted by the controller (see §4.4).

Even after inferring predicates, there may be quite a few reachable bugs still remaining. In our example, consider the `ipv4_lpm` table: all packets hitting the `set_nhop` action will have their `TTL` field decremented. When such packets have an invalid ipv4 header, they will trigger a bug. Ideally, we would like to prohibit packets with an invalid ipv4 header to hit a rule with this action, but unfortunately the `ipv4_lpm` table does not match on the validity of the `ipv4` header.

When there are bugs still reachable after predicate inference, bf4 runs an additional pass where it proposes changes to the P4 code to remove the remaining bugs; for most bug types, it selects a small number of keys to be added to certain tables (§4.3). It then reruns the algorithm to find new predicates that can remove further (ideally all) bugs. In our example, it will add the `ipv4.isValid` key to the `ipv4_lpm` table and generate predicates that eliminate the bug.

While bf4 employs verification techniques from the PL literature [10, 11, 1], we benefit from the specifics of network programming languages - loop-freedom, simple primitive instructions, low-latency constraints which limit program sizes - to make them more efficient and useful in practice.

We have implemented bf4 as a separate backend to the p4c compiler infrastructure. It consists of a series of compiler passes which transform the initial p4 program, automatically add assertions for some categories of bugs, run the analysis phase which includes the `Infer` and Fixes algorithms (§4.2) and output a file describing the inferred necessary precondition and a set of proposed keys to bring the existing bugs under control. The latter is passed to a controller shim which enforces the rules described therein (§4.4).

Our implementation (25KLOC of C++) works against the p4c intermediate representation (IR) and re-uses many tools and passes already implemented within the p4c framework - e.g., type inference, reference resolution etc. We implement the semantics of most instructions defined in the p4 spec [7, 30] and of some of the core external primitives. We now discuss bf4's components in detail.

4.1 Finding reachable bugs

Our approach is similar to that of p4v [24]: we expand table applications depicted by abstract flow entries which match table keys and

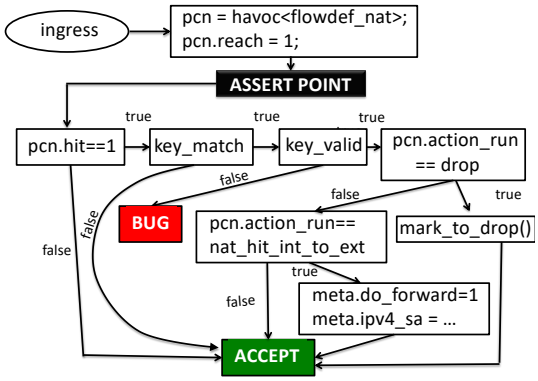

Figure 5: CFG for the running example

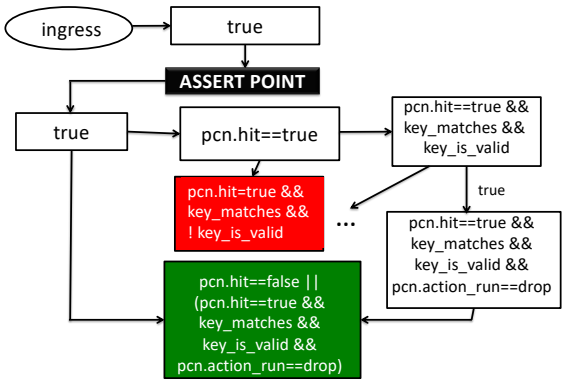

Figure 6: Weakest preconditions.

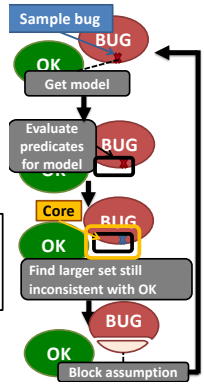

Figure 7: Infer

non-deterministically choose between one of the actions allowed by the table.

Figure 4 shows an example of table expansions which makes explicit the matched entry with all its corresponding variables. We introduce two meta-variables for each table apply call; in our example these are *pcn.reach* and *pcn.hit*. The former means that a table call was reached at some point in the execution, while the latter indicates if the matched entry was a table entry or the default *no-op* action. If a table entry is hit, we instantiate the hit condition. This is a relation between table key expressions (header values) and table entry contents (keys), as depicted by the assert statement. Each match kind has a corresponding semantics - e.g., the ternary mask is translated into a bitwise and, while the exact match is translated into an equality.

We then perform program instrumentation, where we add code to find all possible bugs of the following types: accesing invalid headers, out-of-bounds accesses to register arrays or header stacks and egress_spec not set. To instrument for header validity, we add validity checks whenever a header is read or written to. For instance, before accessing the `ipv4.ttl`, `bf4` checks the validity as follows:

```
if (!ipv4.isValid()) bug(); else ipv4.ttl = ipv4.ttl-1;
```

Similar checks are added to catch cases when the indices for array accesses are larger than the statically known size. Finally, for egres_spec not set, we add check at the end of the ingress control function if the value is different from default.

Next, we integrate the control flows (ingress and egress) and stitch them together with the parser. The resulting program is converted into a control-flow graph (CFG), a graph representation of the program that captures all possible paths. Fig. 5 depicts the CFG of a call to table *nat* in our example.

In order to obtain an acyclic CFG, we unroll possible loops. Since P4 control flows are guaranteed not to contain forwarding loops, the only component which may exhibit cyclic behavior is the parser. The P4 specification [30] states that the hardware may limit the number of times a packet visits a parser state; this means that infinite loops are not possible. We leverage this fact and unroll the looping parser states [1] a number of times larger than the size of any header stack that can be extracted therein.

Similarly to p4v [24], we transform the program to static single assignment form (SSA) in order to avoid exponential blow-up in expression sizes [15].

The resulting CFG is optimized via classic compiler optimization techniques - constant propagation, local copy propagations [2]. We then slice the CFG with respect to the reachability of bug nodes [34]. The slicing algorithm is of special importance, because it is also employed in inferring missing table keys to automatically fix the bugs remaining after Infer (§4.3).

We leverage the acyclicity of the CFG to compute weakest preconditions for the reachability of each node in the graph. We do so by iterating through the nodes of the CFG in topological order and propagating *stronger* conditions to all neighbors depending on the transition relation.

A representative example is depicted in figure 6. The example shows the calculated preconditions at each node in a subset of the CFG from Figure 5. The condition to reach the initial node is *true*; this is propagated to the `pcn.hit==1` node. On the true branch, we propagate the stronger condition `true && pcn.hit==1` to the next node, and its negation on the else branch. Whenever a node has already received a precondition from another parent (e.g., the accept node on the false branch), we merge that condition with the one from the current node. At each merge point in the graph, we perform logical or between the existing label and the one which was propagated. For each bug node, we invoke an SMT solver (in our case Z3[13]) to determine whether or not the current bug is reachable. For each satisfiable bug, it means that there is an initial input which reaches it.

Making verification faster. In order to make `bf4` run faster, we perform a number of optimization passes on the original P4 code and subsequently on the resulting CFG intermediate representation. While many of them are well-known and heavily employed by optimizing compilers - e.g., dead code elimination, constant propagation, live variable analysis etc. -, we emphasize the importance of slicing [19] as a means to reduce the number of instructions in the CFG and thus reduce verification runtime.

We implement slicing in P4 starting from the original SSA-converted CFG by computing the control dependency graph and the data dependency graph [19]. Their union yields the program dependency graph. Then, starting from each "interesting" node - in our case bug nodes -, we compute the transitive closure of the relation given by the dependency graph. The subgraph of the CFG

[1] We detect loops by finding *back edges* in the CFG. Since our CFG is reducible, only back edges can lead to loops [2].

constrained to the nodes in the transitive closure is the slice of the program with respect to the possible bugs in the program.

While the current slicing implementation is using a naive syntactic approach, the results obtained show a lot of promise. On `switch.p4`, our largest test case, slicing reduces the number of instructions relevant for bug reachability from 17155 to 7087 and the total time to model-check for reachable bugs from 36s to 11s.

A side-effect of program slicing is that it eases analysis and program understanding. Counter-example instruction traces obtained on sliced programs are significantly simpler (roughly 10x less instructions than without any slicing) and thus way easier to understand, validate and analyze.

4.2 Inferring buggy table rules

A run through a P4 snapshot is the trace of instructions executed for one input packet. *Good runs* are traces which result a well-defined forwarding behavior (accept or reject). *Bad runs* are traces which result in buggy behavior - e.g., accessing invalid headers.

To generate controller assertions, we find the **table entries that our shim must block without prohibiting any good runs while discarding many (preferably all) bad runs.**

To achieve this we rely on the concept of necessary preconditions from the programming languages literature defined as a formula whose complement captures *inputs to that program that will always trigger bugs* [11]. Note that necessary preconditions are different from *sufficient* preconditions which ensure the program is bug-free, but may inhibit good runs.

`bf4` generates necessary preconditions with a supplementary constraint that all variables used in the necessary preconditions depend solely on the table entries inserted at runtime by the controller. To generate necessary preconditions, we identify a set of nodes in the CFG of the expanded program as *assert points* where we can apply predicates on the table rules. For practical purposes, we only consider assert points to be located upon table call entry. In our example, the only assert point is the black node in Fig.5. Furthermore, we identify a number of program variables as *control variables*. Essentially, control variables are the elements of a table rule: keys, chosen action and the action parameters.

We propose Infer, a novel inference algorithm which iteratively finds controlled necessary preconditions (Algorithm 1) that can be applied in assert points by predicates on the control variables, and guarantee preservation of correct behavior. We run Infer once for every assertion point in the program (i.e., every table apply statement). It takes as input two logical formulas *OK* and *BUG* and a set *P* of atoms whose variables are *control variables*. Infer outputs a predicate (in conjunctive normal form) that is a boolean formula using solely atoms from *P*.

OK describes the set of good runs which pass through the assertion point; we compute it as the precondition to reach an accept state (e.g., the formula in the green node in Fig.6 plus the condition to reach the assert point). *BUG* specifies bad runs dominated by this assertion point. We compute *BUG* as the precondition to reach any bug dominated by this assertion point (the red node in Fig. 6). [2]

P contains boolean atoms using variables whose value we can control at the bug. The atoms are generated syntactically from program text. In our example, the following atoms are generated:

`pcn.ipv4.isValid` `pcn.ipv4.srcAddr.mask==0` `pcn.hit`
`pcn.action_run==drop` `pcn.action_run==nat_hit_int_to_ext`

[2]Dominance is computed on the CFG and it means that all runs to the bug pass through the assertion point.

Algorithm 1 Infer

1: **function** INFER(OK, BUG, $\mathcal{P}$) ▷ returns a CNF formula ϕ with atoms $\subseteq \mathcal{P}$ s.t. $OK \models \phi$ and minimizes $|\{\eta \mid \eta \models BUG \wedge \phi\}|$
2: $direct \leftarrow Solver(BUG)$, $dual \leftarrow Solver(OK)$
3: $\phi : Expression \leftarrow \top$
4: **while** direct.check() == SAT **do**
5: $model \leftarrow direct.model()$
6: $assumptions \leftarrow \{q \mid$ if $model.eval(p)$, then $(q = p)$ otherwise $(q = \neg p)$, $p \in \mathcal{P}\}$
7: **if** $dual.check(assumptions) == UNSAT$ **then**
8: $uc \leftarrow dual.unsat\_core()$
9: $\phi \leftarrow \phi \wedge \neg(\bigwedge_{a \in uc} a)$
10: $direct.add(\neg(\bigwedge_{a \in uc} a))$
11: **else**
12: $direct.add(\neg(\bigwedge_{a \in assumptions} a))$
13: **end if**
14: **end while**
15: **return** ϕ
16: **end function**

Figure 7 graphically shows the steps which make up the Infer algorithm, where we depict the *OK* and *BUG* predicates as regions in the space of all runs. The algorithm iteratively builds a predicate which covers an increasingly larger part of the bad runs but does not contain any good runs.

It first uses the SMT solver to select a sample bad run from set *BUG* - i.e., a model (Line 5). If there is a model, it will contain concrete values for all variables in the run, and can be thought of as a point in the bug region in Figure 7.

Next, Infer evaluates all predicates in *P* against the current model to yield a predicate (called *assumptions*) that describes a wider region (a box in Fig.7) that includes the model, but may also include non-buggy runs (Line 6).

Since the formula *assumptions* can be used as a controlled necessary precondition (since it contains only control variables) iff it removes no good runs. To check this, we need to see whether *assumptions* and *OK* are disjoint (line 7); if they are, we block the boxed region (i.e, we "subtract" *assumptions* from *BUG*) and restart the iteration (line 12).

When *assumptions* and *OK* do not intersect, we could simply add *assumptions* to our preconditions and iterate, but it would take a long time to cover the entire *BUG* region. To reduce the number of steps, we would like to expand this region as much as possible as long as it does not intersect *OK*. For this, we use the *unsat-core* implementation of the solver, obtaining *uc* which is a minimal subset of *assumptions* which is unsatisfiable in *OK* (Line 8).

In other words, this finds the largest region which includes the box while still not intersecting *OK* - the core. We then update the set $BUG = BUG \setminus uc$ (Line 10) and go to the first step. The algorithm ends when set *BUG* is empty. In our running example, the following predicate is inferred: `pcn.hit` $\wedge$ `¬pcn.ipv4.valid` $\wedge$ `¬(pcn.ipv4.srcAddr.mask==0)`.

The predicate states that all rules with an invalid `ipv4` header and a zero mask are buggy and thus should be blocked. These predicates are then used at runtime, to filter buggy rules (see 4.4) and in the verification process: the predicate is asserted and we recompute the reachability of the bugs it dominates. Many bugs become unreachable, and we say we can "control" these bugs; some are

Algorithm 2 FAST-INFER

1: **function** FAST-INFER(cfg, t, $controlled$) ▷ returns a controlled necessary precondition ▷ cfg is the subgraph of table t, $table\_begin$ the start of the $t.apply()$, $controlled$ a set of control variables of t - keys and action data
2: $(p2oks, p2bugs) \leftarrow symbex(cfg, table\_begin)$
3: $\phi \leftarrow \top$
4: **for** $path \in p2oks$ **do**
5: $pc \leftarrow pathCondition(path)$
6: $v \leftarrow Vars(pc)$
7: **if** $v \subseteq controlled$ **then**
8: $\phi \leftarrow \phi \wedge (\neg pc)$
9: **end if**
10: **end for**
11: **return** ϕ
12: **end function**

still reachable and in which case we invoke the Fix algorithm that changes the P4 program to make the bug unreachable, described in §4.3.

Even though we provide no guarantees on its minimality, in practice, Infer can be used to control many reachable bugs in a program, as we show in §5.

Safety of Infer. In the Appendix we provide a theoretical formulation of the necessary preconditions `Infer` computes and prove that the algorithm never removes good runs.

Speeding up Infer. While Infer is reasonably fast in practice (10 minutes against switch.p4), we propose an approximate version of it that makes it much faster. The biggest issue with Infer is the size of the formulae we pass to the solver. The set *OK* includes the path conditions for all good runs through that entire program, with the only condition that they pass through the table we are analyzing.

Our fast algorithm reduces this set by only examining paths through the table in question, from the assertion point to when control exits the table call: it thus assumes that any packet can reach the assertion point, and that any packet leaving the table will be a good run.

Fast-Infer (Algo.2) finds all possible paths from the assertion points to the bug(s) in the table using symbolic execution. If all the variables in a path condition are controlled (i.e., all packets hit that bug), then the negation of this path condition is a necessary precondition (i.e., a controller assertion). We first compute cfg, the control-flow graph for table t. The set *controlled* is the set of keys and action parameters of table t. Then, symbolically execute starting with unconstrained variables from the start of the graph (*table_begin*). The result of symbolic execution is a set of paths to terminals (either good or bad). For each path to a bug, take its path condition *pc*. If the variables involved in *pc* are included within *controlled*, conclude that $\neg pc$ is a necessary condition.

As each table's CFG is small and the number of conditions inside a table are few (in the order of the number of table actions), symbolic execution explores all paths to terminals very fast (1ms per table in switch.p4). Furthermore, the check in line 7 of Alg.2 is also fast. The end result is an algorithm which computes a set of all necessary table preconditions in just 1.5s for switch.p4.

How does Fast-Infer relate to Infer? Assume that ϕ_{fast} is the result of Fast-Infer and ϕ that of Infer. We prove in the Appendix that Fast-Infer is an over-approximation of Infer ($\phi \models \phi_{fast}$): running Fast-Infer may fail to produce preconditions in certain cases where Infer will succeed.

`bf4` uses both algorithms: it first runs Fast-Infer; only if it does not control all bugs we call Infer to cover the bugs not controlled by Fast-Infer.

Multi-table rules. Both algorithms Infer and Fast-Infer reason about single-table preconditions. These preconditions can be quickly checked at runtime and remove many bugs from the data-plane code, but they do not remove all bugs.

Is it possible to infer multi-table preconditions while keeping runtime complexity at bay? We attempted a generalization of Infer to multiple tables but this resulted in non-tractable runtimes. We then turned to heuristics that capture some classes of multi-table assertions. Take the following snippet inspired by switch.p4:

```
table t1 {key={k1:exact}, action={validate_H; nop;}}
table t2 {key={k1:exact, k2:exact}, action={use_H;}}
action validate_H() { H.setValid(); }
action use_H(){ if (!H.isValid()) bug(); else ...;
  H.setInvalid(); t1.apply(); t2.apply(); ...}
```

Assume that the following two entries exist in tables $e1 = (k1 = v, action = nop) \in t1$ and $e2 = (k1 = v, k2 = *, action = use\_H) \in t2$. Here, v stands for an arbitrary value in the domain of instr1, while * denotes an unconstrained value. Whenever a packet hits entry $e2$ in $t2$, it will definitely hit rule $e1$ in $t1$. Since the statement right before the application of t1 sets H to invalid, then a bug will be triggered in action *use_H*. This means that this combination of rules is bound to be buggy if ever inserted by the controller.

To generate these multi-table assertions, we use the results of Fast-Infer for table $t1$ to start exploration for table $t2$. When we have uncontrollable bugs in $t2$, if the keys of $t2$ are a superset of $t1$, we try to control the bugs in $t2$ using the assertion point of $t1$.

Our multi-table heuristic generates assertions that control 7 bugs in switch.p4 that cannot be controlled by single-table assertions. To understand if there are other types of multi-table assertions that could control the remaining bugs in switch.p4, we tested all possible combinations of two and three tables, asking the solver to provide for each combination of matching rules a matching good and bad run. If the solver returns unsat, it means that we could control more bugs with better multi-table algorithms. For switch.p4, however, the solver confirmed that none of the remaining bugs can be controlled this way.

Increasing bug coverage. Infer produces specs which cover a fair amount of bugs (79 bugs previously reachable are suppressed in switch.p4). Can we do better while ensuring maximal permissiveness with respect to good runs through the program? Note that the limiting factor with respect to the maximum coverage attainable by Infer is the set of OK runs.There are however situations where this condition is too strong, as in the following example inspired by switch.p4:

```
table encap{key={ipv4.valid:exact},actions={do_encap; nop;}}
action do_encap() { inner_ipv4 = ipv4;}
```

Our instrumentation backend converts the header assignment in action do_encap into:

```
if (ipv4.isValid()) {... copy fields}
else { if (inner_ipv4.isValid()) { bug(); }
       else {  /* no op */; }}
```

We claim that the bug in the else branch is legitimate (though not an undefined behavior per se), because it performs a destructive operation against a previously valid header without the user explicitly

mandating so. Had the programmer actually intended to invalidate inner_ipv4, he should have explicitly called the *setInvalid*() primitive on inner_ipv4.

Now, assume that table encap has an entry e=(ipv4.valid=false, action=do_encap). Then there are two possibilities: either inner_ipv4 is invalid, in which case nothing happens or it is valid and a bug gets triggered. Running Infer against this table produces no spec, since `inner_ipv4.isValid()` is not among its keys and thus could take an arbitrary value. Had Infer produced a spec precluding the existence of e, then it would only have suppressed the "no op" branch. So this means that keeping entry e in table `encap` is just a "fancy" way to do a no-op or trigger a bug. We classify such rules as *controller-induced bugs*.

To find such cases, we introduce a `dontCare` instruction to our intermediate code. Thus, the "no op" branch above becomes: `... else dontCare(); ...`

Now, we tailor Infer to take `dontCare`s into account. We first compute a formula *reach*(*dontCare*) as the disjunction of the reachability formulas of all *dontCare* statements and constrain the *OK* formula s.t. $OK \leftarrow OK \wedge \neg reach(dontCare)$. This change reflects the following rationale: we would like to suppress as many bugs as possible without precluding any good runs that we care about. This simple heuristic trims down 31 extra encapsulation bugs in switch.p4.

4.3 Fixing P4 programs

Our end goal is bug-free programs, but the Infer algorithm may fail to control bugs in certain cases. One such case is in our example: the bug is accessing the `ipv4` header in `set_nhop` action for packets with an invalid `ipv4` header. This bug cannot be controlled because the only match key in the `ipv4_lpm` (metadata `meta.ipv4_da`) is in no way connected to the validity of header ipv4. What are the options to handle this kind of uncontrolled bugs?

The simplest way is to just output a path through the program which triggers the bug and let the programmer fix it - either change his dataplane code or come up with a stronger controller constraint which rules it out. Another solution to generate a sufficient precondition to suppress the bug; unfortunately this may prune good runs. It is then up to the programmer to decide whether it is ok to rule out those; unfortunately, this task has complexity similar to manually generating assertions, which we want to avoid.

We want to automatically deduce fixes to the dataplane code. All code changes are subject to the following constraint: all good runs in the fixed program should produce the same result as the original one. Furthermore, the fixes should be small enough to avoid hurting performance. An obvious approach is to place if statements (guards) around all possible bugs. However, most hardware targets do not allow ifs within actions; worse, even when available, such guards increase the number of stages needed to deploy the program.

Our solution fixes the program by adding missing table keys. This approach can solve all controller-induced bugs, but will fail to fix dataplane bugs. Initially, we modified the Infer algorithm to this end, relying on unsat-core to find missing keys. While correct, that version was also fairly slow. That is why we used ideas from Fast-Infer to generate fixes.

Our Fixes algorithm (Algo. 3) uses data flow analysis to find the

Algorithm 3 Fixes

```
1:  function TableKeys(bug, t, controlled) ▷ returns a set of keys to t such that
    t controls bug; controlled is a set of control variables of t
2:      node2fact : Map[node, L] ← {}
3:      node2fact[t] = (∅, ∅)
4:      node2fact ← fw_dataflow(t, node2fact)
5:      return vars(node2fact[bug]) \ controlled
6:  end function
7:  function Fixes(bug) ▷ returns a mapping from tables to extra keys to be added
8:      s ← slice(bug)
9:      T ← last_resort_tables(bug)
10:     υ : Map[node, P(V)]
11:     for t ∈ T do
12:         υ[t] ← TableKeys(bug, t)
13:     end for
14:     return υ
15: end function
```

set of keys needed to control a bug. Let *bug* be a bug point which is still reachable. We start by computing the slice of the initial CFG to yield a smaller graph relevant to the reachability of *bug* (line 8 in Alg.3). This line is not necessary in the computation of extra keys, but is used as a means to reduce the number of computed keys.

In line 9, we compute the set of last-resort tables with respect to node bug; a table is of last resort for a given node if it is the last which may prevent the node from being reached (in most cases, this is the table which dominates the bug).

From each $t \in T$, we perform a forward dataflow analysis [2] with lattice given by tuple $L = (vars : \mathcal{P}(V), terms : \mathcal{P}(V))$, and partial order given by pairwise set inclusion. The goal is to find the smallest set of variables which are live at the assertion point and control the bug. Take this example:

```
// assertion_point for table t
if (y == 0) { x = 3; } else { x = z; }
if (x == 10) { bug(); }
```

Since x is the only variable read in the condition which guards bug(), it is tempting to think x is the only extra key required for table t. However, this is not right, since the value of x at condition `x==10` has been rewritten since the assertion point. Our algorithm tracks the fact that x depends on y and z, so the minimal set of keys to add to table t is $\{y, z\}$.

The transfer function $tr : Statement \rightarrow (L \rightarrow L)$ wrt to a P4 program statement is defined as:

$$vars(tr(stat)(l)) = vars(l) \cup (reads(stat) \setminus terms(l)),$$
$$terms(tr(stat)(l)) = terms(l) \cup writes(stat),$$

where $reads, writes : Statement \rightarrow \mathcal{P}(V)$ denote the variables that are read or written in that statement.
The meet operator is: $l \sqcup r = (vars(l) \cup vars(r), terms(l) \cup terms(r))$. The analysis begins at node $t \in T$ with initial value $(\emptyset, \emptyset)$. Let f the dataflow fact obtained for node *bug*. Then $Miss = vars(f) \setminus controlled$ are the missing keys of t. Since we are working in SSA, we state that the variables in *Miss* are available at node t, by construction. For our running example, Fixes adds key ipv4.isValid() to table `ipv4_lpm`.

4.4 Sanitizing table rules

Our shim transparently intercepts the messages between the controller and the dataplane, as proposed in prior SDN verification works [8, 21]. We have implemented and integrated the dataplane update sanitization module in ONOS[29] (a popular P4 controller). We changed to the P4Runtime component which is in charge of

pushing updates to the switches.

bf4 outputs table assertions using a SQL-like syntax. Each assertion has two parts: the condition header which states the variables it references (P4 match keys, or action arguments) and the condition body which is a predicate over the variables in the header. A condition may also refer variables from the currently running switch configuration; to fetch these values in a timely manner, the shim maintains shadow copies of the table contents for every P4 switch it controls, thus avoiding to query the switches. The shim loads and parses the assertions and creates in-memory data structures. For every dataplane update request, the validation algorithm a) first detects which conditions the update might violate, based on the update and the condition headers; this takes constant time since one update can modify only one table and we can cluster conditions based on the table id; b) next, for every such condition its body is rewritten using the concrete values from the update being tested; c) if there are any unbound variables, the shadow tables are queried; for every table, we deploy a set of hash tables, one for every variable referenced by the table; lastly, since we are only dealing with exact matches, querying the shadow tables is linear in the number of unbound variables left after step (b).

When a rule is safe, it will be inserted in the switch. Otherwise, the rule will not be inserted, and an exception will be returned to the controller. At this point, the controller should be debugged and the source of the bug corrected; if the exception is ignored, we now have a mismatch between the controller(which thinks the rule update went ahead) and the dataplane - this is a recipe for confusion and further bugs.

A special case is handling default rules: the shim rejects all attempts to set a default rule which contains an action that has a reachable bug, since there is no way to guarantee that packets will not hit that rule at runtime.

When fixes have been applied to the P4 program, we assume the controller has been updated to reflect these changes; the new rules will be filtered at runtime as discussed above. We acknowledge that controller changes are cumbersome, and may take time to develop. We have considered the alternative of having the shim transparently "fix" the rules from an old controller by adding safe values for the new keys. However, the new and the old dataplanes one are no longer equivalent which raises interesting questions around controller correctness and corner-case behaviour. We intend to further explore this direction in future work.

4.5 Shim impact on correctness.

Our sanitization shim logically operates between the controller and the P4 dataplane and performs input validation on whatever comes in from the controller. To make bf4 useful in practice, its impact on the original P4 code and controller must be as small as possible. We analyze the shim's impact in terms of: (i) *correctness*, (ii) *performance* and (iii) *code*. We leave performance impact analysis for §5.3.

We now turn our attention to correctness. In this setting, correctness refers to semantics preservation with respect to the original P4 program and controller. Adding the shim means altering the controller. We call this alteration semantics preserving if the P4 program + the modified controller are behaving the same as the old one with respect to packet processing.

Ideally, for a bug-free program, the shim bears no impact on the original P4 program and controller. However, when bugs start showing up, things get trickier. Assume that the shim is presented with a new table rule. We pose the following restriction: if the rule is correct - i.e., it does not lead to bugs - the program behaves exactly the same as the original. If, however, the rule is not correct, then the new controller's behavior is undefined. We say that the shim is partially semantics preserving wrt the original program.

We now examine what happens when the controller attempts to insert a buggy rule. There are two extreme solutions and combinations thereof:

(1) allow the rule to get in - then the resulting program is identical to the original. But then, the program becomes a ticking bomb waiting for the right packet to trigger dataplane bugs
(2) block the input and raise an exception - in this case the two programs are not identical anymore. It is now the job of the controller to perform error-handling, which may imply changing the original controller code to handle this situation.

We believe that the latter approach is the right one to ensure correct dataplanes. We also claim that the code impact for this kind of error handling is not dramatic. This is because the code to handle input validation exceptions is already baked in the original controller even when the shim is not there. For instance, if a controller attempts to insert a duplicate rule in a P4 table, an exception will be thrown. We find that this situation is, in principle, identical to the shim throwing a sanitization exception on a bad input and should be treated in a similar fashion.

4.6 Limitations

Our approach has a series limitations which fall in two classes: technical and conceptual.

The most noteworthy technical limitation is that bf4 doesn't fully support the PSA and TNA architectures (only works for the ingress and egress pipelines in separation). The reason for this shortcoming is that, in order to capture the interaction between ingress and egress, packets need to be precisely modeled (e.g., via strings of bits). This model makes calls to the underlying solver highly inefficient. Finding an good model for packet processing primitives is our future work.

Even though bf4 is a procedure to automatically infer missing controller annotations, it does not preclude adding user-defined annotations. Nevertheless, the current implementation of bf4 provides no way for the user to specify user-defined annotations.

Conceptually, bf4 is first of all limited by the kind of properties it is able to capture. The static verification part of bf4 is a bounded model checker which only tackles safety properties. For the moment, we instrument for invalid header accesses, "egress spec not set" bugs and array indices out of bounds. One could also envisage higher-level properties by placing assertions in their code.

While in principle bf4 should be able to infer fixes/specs for any kind of safety property, it is not effective for specs of the form: "there must exist a rule subject to c in table t". bf4 is mostly efficient to infer specs of the form: "there must not be any rules in table t subject to c", corresponding to safety invariants.

A noteworthy example is the "egress spec not set" bug. In this case, one may envisage a property enforcing the control plane

to always take a forwarding decision. This does not work well with bf4. The reason lies in the way "egress spec not set" bug is instrumented. Much like p4v [24], we keep a shadow boolean variable together with the *egress_spec* metadata which indicates whether or not *egress_spec* was set. At the end of ingress, we check if this shadow variable is set. If it isn't, then we signal a bug. This means that for all feasible sequence of tables from the start of the program to the end of ingress, at least one chosen action sets *egress_spec*. But if on that path there are more than one possible actions which set *egress_spec*, then bf4 can't say for certain which of them must be set, since this would preclude good runs passing through the other.

Moreover, even bf4's fixing strategy is problematic in this case. The extra key needed to control this kind of bug is the ghost variable used to track the assigndness state of *egress_spec*. Since this variable doesn't really exist as part of the original program, the reason for its existence may seem meaningless to the programmer.

To handle this particular situation, we specialize our fixing strategy: whenever an "egress spec not set" bug is reachable, bf4 suggests to the programmer a "safe" fix - e.g., drop the packet at the end of the pipeline.

A note on result minimality. Both Infer and Fast-Infer are used to synthesize invariants which rule out unsafe behaviors. We claim no minimality on the filters inferred by either of them. So far, the filters were simple enough and reasonably few to be efficiently implemented by the controller shim. However, this may become a bottleneck as more complex properties will be added.

In what the Fixes algorithm is concerned, its output is also not guaranteed to be minimal. However, our evaluation shows that it never adds more than two keys per table for a single bug and that, in most cases the set of keys is minimal.

Generating a minimal number of fixes as well as minimal number of invariants is important both for performance (table width/height increase - some programs may actually fail to compile) and diagnostics (making fixes/annotations easier to understand by programmers). We leave these interesting ideas as future work.

In the following section, we show how to make use of the specs inferred via algorithm Infer and enforce controller-level safety.

5 EVALUATION

bf4 computes necessary preconditions to control bugs and suggests key additions which transform most uncontrolled bugs into controlled bugs. Assume that a P4 program only contains controlled bugs, and that our sanitization shim is deployed. We prove in the Appendix that bf4 guarantees that all deployed snapshots are bug free, i.e., no packet exists which can reach a bug.

We ran bf4 on a wide range of P4 programs to test its usability and efficiency. bf4 differs from existing P4 verification tools because it does not stop after discovering possible bugs: it generates predicates that eliminates these bugs and fixes the P4 program when predicates cannot be generated with existing keys. We therefore wish to find out how many bugs bf4 controls immediately, what keys it adds when needed, and how many bugs remain reachable after fixing.

We evaluated bf4 on a 1.70GHz Intel(R) Xeon(R) CPU E5-2650L server with 32GB of memory. We take as inputs a set of 94 openly available P4 programs in the V1Model. For the moment, our implementation includes full support for P4-14 (or V1Model programs) and only limited support for PSA and TNA. Full support for these architectures is our future work.

Table 1 contains a subset of our results, showing the total number of bugs found, the number of bugs which may be controlled using only existing keys (results of Infer), the number of extra keys added by our fixing algorithm, and the number of bugs that are reachable after fixing. The most complex program we ran bf4 on was the open-source version of switch.p4, a production-grade datacenter router implemented in P4.

A large fraction of the bugs bf4 found are invalid header accesses (90%), followed by egress_spec not set and, less frequently, out-of-bounds register accesses. Infer manages to generate assertions to remove 39% of reachable bugs on average, and all bugs in the best case. For switch.p4 it removes 100 bugs out of a total of 155. This verifies our hypothesis that simple table-level checks on existing keys are sufficient to remove many bugs.

The fixes algorithm manages to eliminate **all** bugs in most target programs (bug types are header validity, registers out-of-bounds, and egress spec not set). For switch.p4, it adds 23 table keys to 13 separate tables to eliminate all remaining bugs. Assuming our implementation of bf4 is correct, and that the sanitization shim works correctly, these results imply that we can deploy these programs with the guarantee that no buggy snapshot will ever be deployed.

The worst case runtime of bf4 is 4 minutes on switch.p4. This includes instrumentation, translating the original program to a verification-friendly form, integrating the packet flow between the components of the P4 program (42% of the time) and the actual bf4 algorithms (58% of the total time). Given the complexity of this program, and the runtime of previous tools that requires annotations (2.5 mins for p4v), we believe the runtime is competitive.

We found that many programs in our test suite (43 out of 94 tested programs) exhibit "uncontrollable" bugs, which are still reachable after running Fixes. These are genuine dataplane bugs that we report to the programmer in order to get them fixed. One example is in the mplb_router which reads from the tcp header inside an if condition. This is clearly a problem since no prior table is able to rescue it.

To understand the overhead of key additions, we note that the 23 keys added to switch.p4 are a 6% increase compared to the original program (372 keys). However, the average increase in the size of a table entry is less than 1% because most of the keys added are validity checks which only require one bit (i.e., on average less than one bit is added to each rule).

Another interesting metric impacting bf4's deployability in production settings is the impact of key addition to underlying controller code. Adding new keys changes the runtime API [31] for the table being controlled. In our largest example, switch.p4, there are 13 tables which get modified as a result of bf4 out of a total of 129 (10%). Since our changes are additive, the newly generated runtime API inherits from the old one, so controller changes should be minimal.

Program	LoC	#bugs	bugs after Infer	runtime (s)	bugs after fixes	keys added
07-MultiProtocol	371	2	2	6.977	0	2
arp	217	6	0	5.214	0	0
ecmp_2	204	2	2	4.575	0	1
flowlet	290	2	2	5.464	0	2
flowlet_switching	288	2	2	5.633	0	2
hash_action_gw2	135	2	2	3.397	0	1
heavy_hitter_1	211	5	4	4.579	0	2
heavy_hitter_2	246	5	5	5.352	0	6
hula	321	6	3	8.563	0	3
issue894	254	5	5	5.362	0	1
linearroad_16	991	20	20	26.773	1	20
mc_nat_16	181	2	1	4.013	0	1
mplb_router-ppc	264	2	2	6.89	1	0
ndp_router_16	287	4	4	5.69	0	3
netchain	1849	4	4	402.79	0	5
netchain_16	497	6	6	11.258	0	5
netpaxos_accept_16	234	2	2	5.58	0	1
resubmit	124	2	0	3.197	0	0
simple_nat	372	7	2	7.174	0	1
switch	6234	155	55	222.003	0	23
ts_switching_16	142	4	3	4.036	0	2

Table 1: Experimental results on a wide range of P4 programs, showing verify removes all bugs.

5.1 Interesting bugs discovered by bf4

The purpose of verification tools is to provide strong correctness guarantees and detect possibly faulty behaviors. Thus, a detailed understanding about the bugs reported by bf4 is crucial to ensure that it works correctly and to capture interesting behaviors that may impact the network dataplane. We describe such bugs next.

Missing assumptions. Let's take as an example table *validate_outer_ethernet* in the ingress control flow of switch.p4. Its structure is described below (details omitted for brevity).

```
action doubletagged() {
  meta.l2_metadata.lkp_pkt_type = 3w1;
  meta.l2_metadata.lkp_mac_type = hdr.vlan_tag_[1].etherType;
  meta.l2_metadata.lkp_pcp = hdr.vlan_tag_[0].pcp; }
table validate_outer_ethernet {
  actions = { ...
    doubletagged();
    // stands for _set_valid_outer_unicast_packet_double_tagged  }
  key = { ...
    hdr.vlan_tag_[0].isValid(): exact;
    hdr.vlan_tag_[1].isValid(): exact; } }
```

Both actions read field the pcp from header `vlan_tag_[0]`, whereas action *_set_valid_outer_unicast_packet_double_tagged* also reads from header `vlan_tag_[1]`. Infer outputs that all bugs within this table are completely controllable when the following assertion holds for table entries:

$$\neg(\neg key(hdr.vlan\_tag\_[0].isValid()) \wedge action = doubletagged) \wedge \neg(\neg key(hdr.vlan\_tag\_[1].isValid()) \wedge action = doubletagged)$$

The inferred condition is natural as it implies that whenever there is no vlan encapsulation, then one should simply not call action *doubletagged*. Uncorrected, this bug could cause significant problems: packets without the VLAN tags might be validated using header values from previous packets, allowing attackers to break VLAN isolation.

Missing validity checks. Table *fabric_ingress_dst_lkp* in the ingress control flow of switch.p4 is a good example where a bug is reachable regardless of controller assumptions.

```
action unicast() { standard_metadata.egress_spec =
          (bit<9>)hdr.fabric_header.dstPortOrGroup;
   meta.tunnel_metadata.tunnel_terminate =
          hdr.fabric_header_unicast.tunnelTerminate; ... }
table fabric_ingress_dst_lkp { actions = { ...  unicast();
// stands for _terminate_fabric_unicast_packet_0 }
    key = { hdr.fabric_header.dstDevice: exact; } }
```

First of all, notice that since there is no key matching against the validity of *fabric_header*, exact matching against an invalid value will lead to undefined behavior. Thus, a packet may match an arbitrary table entry. Assume for the moment that there is an entry with action *unicast* and we get a match against this particular entry. This now means that *egress_spec* will be set to an arbitrary value.

Furthermore, since there is no control assertion to filter this behavior, bf4 outputs that there is indeed a bug within table *fabric_ingress_dst_lkp* and deems it uncontrollable using the current keys. Then, using the fixes algorithm, bf4 is able to determine that an extra exact match on *hdr.fabric_header.isValid()* is necessary to control the undefined behavior specified above.

To emphasize the severity of validity errors, notice that if header *fabric_header* reaches action *unicast* with unitialized values, the entire forwarding behavior of the switch will also be undefined (due to the assignment to *egress_spec* field). Since some target architectures do not zero out invalid headers [14] and keep values from previous packets, an exploit is possible whereby an attacker sends out a valid *fabric_header* packet on some port, then subsequent invalid *fabric_header* packets from the same port will be forwarded to whatever port the attacker had previously chosen. By adding extra validity matches at table level and enforcing controller assertions, bf4 makes this attack impossible.

Egress spec not set. A very common bug occurring in P4 programs (especially those converted from P4-14) is not setting the egress spec by the end of the ingress pipeline. This implies programmers assume that packets are implicitly dropped if no forwarding action is specified. However, this is not the actual behavior. The egress_spec field is initialized to the 0 unless the pipeline actually sets it to some other value (such as drop). In our running example, assume a packet hits table `if_info` and then hits table `nat` with action `miss_ext_to_int`. The action correctly sets `meta.do_forward` to 0, and the packet finishes processing. However, since no action sets the value of `egress_spec`, the packet gets forwarded to port zero of the switch.

An easy fix is to have the target reserve port 0 as drop port (as in Tofino). However, we believe that a better way to solve this is to explicitly set the `egress_spec` to drop or some other value at beginning of ingress. Thus, the programmer's intention would become transparent from the beginning.

We have encountered this bug in most V1 programs we have examined (except for switch.p4). With this bug we cannot simply apply the fixing strategy described in §4.3. We have observed that Fixes infers a very large number of extra keys spanning multiple tables to handle this behavior. Instead of doing this, we special case such bugs: we suggest the user to mark the packet as drop at the beginning of the pipeline.

5.2 bf4 vs. existing P4 verification tools

To our best knowledge, bf4 is the first tool that automatically infers control-plane assertions for a P4 dataplane and adds keys to fix bugs. However, we experimentally compare bf4 to p4v [24] and Vera [32] on switch.p4 to gain confidence in bf4's correctness and performance characteristics.

Since p4v is not open-source, we will contrast it to bf4 in two ways: by comparing the results reported in the paper, and by implementing an approximation of p4v. For Vera, we use the publicly available code.

We approximate p4v's functionality by combining the weakest preconditions for all the bugs computed by bf4 and then running a single solver query to check whether any of the bugs is reachable. This approach confirms the existence of bugs in roughly 2 minutes (28s for instrumentation and 85s for optimizations, query creation and solving) The p4v paper reports a similar runtime (2min36s). p4v is meant to be run with a human in the loop: after a bug is reported, the human adds control plane assertions that "block" that bug, and p4v is then rerun to check for other bugs. To check header safety, 700 lines of annotations were added to switch.p4; this is both time consuming (as a solver call is made after each bug is blocked) and error-prone. bf4 takes under 4 minutes to verify the same program for invalid headers, out-of-bounds indices and "egress spec not set" bugs, generates all the annotations and derives the missing table keys automatically.

Vera [32] uses symbolic execution and takes 15s on a concrete switch.p4 snapshot. To find bugs in all dataplanes, we run Vera using symbolic table entries, as suggested by [32]. In this case, Vera didn't finish exploring all paths in the P4 program and obtained a code-coverage of roughly 30%, finding 32 bugs after 7 hours of running. Vera has the advantage of finding all paths that lead to a given bug. However, making sense of the verbose output is a big challenge, and increasing coverage to find all bugs seems a hard task. In contrast, bf4 is not as fine-grained as Vera, but scales indeed much better.

5.3 Sanitizing controller inputs

To assess the feasibility of deploying our modified ONOS controller in production environments, we benchmarked it using the largest P4 program we analyzed (switch.p4). We stress tested the shim by simulating table rule insertions for the 79 tables in switch.p4 which contain assertions using production traces containing 2000 updates to the dataplane. The updates were performed one-by-one and the shim validation time was measured for each particular update.

For switch.p4, bf4 generated 370 assertions that our shim must verify. We tested all updates against every rule, finding that it takes less than 2ms in 90% percentile to decide if the update satisfies the assertion (maximum 20ms for the largest assertion). We then examined the time needed to decide whether an update is safe, which meant matching against all the relevant assertions for the target table. In this case, the median time is 42ms, the 90th percentile is 70ms, with a maximum of 250ms for the `int_bos` table that has 16 assertions over 30 keys.

6 RELATED WORK

Network verification research has traditionally focused on checking network-wide properties such as reachability or isolation, and falls in two broad classes: dataplane and control plane verification. Dataplane verification tools [20, 21, 25, 33] take a network snapshot and check the desired properties. Control-plane verification answers similar questions by taking network configurations and either simulating the functionality of the control plane [16, 4, 17] or runs it in an emulated environment [23] to produce the dataplane which is then verified. bf4 is a network dataplane analysis tool, specialized for finding bugs in P4 programs.

bf4 shares the similar goal of reducing human input to network configuration synthesis. Work by Beckett et al. [5] and NetComplete [18] aims to automatically generate router configurations that achieve stated network-wide properties. While [5, 18] generate control plane inputs, bf4 generates assertions for the control plane outputs and fixes for the P4 dataplane program. NICE [8] proposes a methodology to test SDN controllers with increased coverage. While NICE uncovers many controller bugs, it assumes the dataplane implementation is correct. With P4 this assumption does not hold, and bf4 helps by eliminating all dataplane bugs.

Existing P4 verification tools include p4v [24], Vera [32], ASSERT-P4 [27], P4-NOD [26] and p4-pktgen [28] We've discussed p4v, Vera and ASSERT-P4 in detail throughout this paper. P4-NOD assumes concrete snapshots and translates P4 to NOD[25]; it has reduced coverage as a consequence [26]. p4pktgen[28] uses symbolic execution to generate test packets and predict the expected result, which it compares to the bmv2 software switch results; it finds bugs in the compiler and software switch. bf4 is complementary to p4pktgen.

The works of Cousot et.al.[11, 12] and CEGIS [1] inspired our work. However, the algorithms described therein mostly use word-level abstractions, whereas most network functions are expressed in terms of bit vectors. PREInfer[3] takes known concrete input examples which lead to failures and generalizes them as function-level preconditions. Since the presence of a bug in p4 is not transparent to the user, we found this approach to be unsuitable for our goals. We follow the technique described in [9] and also use notions from IC3[6] and [22] to strengthen over-approximations of the expected result until fixed point.

7 CONCLUSIONS

bf4 is a tool that instruments P4 programs to find all instances of various types of bugs (invalid header accesses, out-of-bounds accesses, etc.). It is also the first tool that guarantees that such bugs do not appear in practice by automatically generating controller assertions, fixes to the program when needed, and by enforcing the assertions at runtime.

bf4 is ready for deployment: it can fix and check the largest public P4 program in a few minutes, and can verify controller assertions at runtime using our sanitization shim with negligible overhead. Further experiments are needed with real controllers to find out how often buggy entries are inserted. Integrating correctness specifications into bf4 is another interesting avenue of research.

ETHICS ISSUES

This work does not raise any ethical issues.

ACKNOWLEDGEMENTS

This work was funded by CORNET H2020, a research grant of European Research Council (no. 758815).

REFERENCES

[1] Alessandro Abate, Cristina David, Pascal Kesseli, Daniel Kroening, and Elizabeth Polgreen. "Counterexample Guided

Inductive Synthesis Modulo Theories". In: *Computer Aided Verification*. Ed. by Hana Chockler and Georg Weissenbacher. Springer International Publishing, 2018.

[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, Aug. 2006. ISBN: 0321486811. URL: http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20%5C&path=ASIN/0321486811.

[3] A. Astorga, S. Srisakaokul, X. Xiao, and T. Xie. "PreInfer: Automatic Inference of Preconditions via Symbolic Analysis". In: *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. June 2018, pp. 678–689. DOI: 10.1109/DSN.2018.00074.

[4] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. "A General Approach to Network Configuration Verification". In: *SIGCOMM*. 2017.

[5] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. "Network Configuration Synthesis with Abstract Topologies". In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2017. Barcelona, Spain: ACM, 2017, pp. 437–451. ISBN: 978-1-4503-4988-8. DOI: 10.1145/3062341.3062367. URL: http://doi.acm.org/10.1145/3062341.3062367.

[6] Aaron R. Bradley. "SAT-Based Model Checking without Unrolling". In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by Ranjit Jhala and David Schmidt. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 70–87. ISBN: 978-3-642-18275-4.

[7] Mihai Budiu and Chris Dodd. "The P416 Programming Language". In: *SIGOPS Oper. Syst. Rev.* 51.1 (Sept. 2017), pp. 5–14. ISSN: 0163-5980. DOI: 10.1145/3139645.3139648. URL: http://doi.acm.org/10.1145/3139645.3139648.

[8] Marco Canini, Daniele Venzano, Peter Peresini, Dejan Kostić, and Jennifer Rexford. "A NICE Way to Test Openflow Applications". In: *Proc. NSDI'12*.

[9] Hana Chockler, Alexander Ivrii, and Arie Matsliah. "Computing Interpolants without Proofs". In: *Hardware and Software: Verification and Testing*. Ed. by Armin Biere, Amir Nahir, and Tanja Vos. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013.

[10] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. "Bounded Model Checking Using Satisfiability Solving". In: *Form. Methods Syst. Des.* 19.1 (July 2001), pp. 7–34. ISSN: 0925-9856. DOI: 10.1023/A:1011276507260. URL: https://doi.org/10.1023/A:1011276507260.

[11] Patrick Cousot, Radhia Cousot, Manuel Fähndrich, and Francesco Logozzo. "Automatic Inference of Necessary Preconditions". In: *Proceedings of the 14th International Conference on Verification, Model Checking, and Abstract Interpretation - Volume 7737*. VMCAI 2013. Rome, Italy: Springer-Verlag, 2013, pp. 128–148. ISBN: 978-3-642-35872-2. DOI: 10.1007/978-3-642-35873-9_10. URL: https://doi.org/10.1007/978-3-642-35873-9%5C_10.

[12] Patrick Cousot, Radhia Cousot, and Francesco Logozzo. "Precondition Inference from Intermittent Assertions and Application to Contracts on Collections". In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by Ranjit Jhala and David Schmidt. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 150–168. ISBN: 978-3-642-18275-4.

[13] Leonardo De Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Proc. TACAS'08*.

[14] Mihai Dumitru, Dragos Dumitrescu, and Costin Raiciu. "Is it possible to exploit buggy P4 programs?" In: *Symposium on SDN Research (SOSR)*. 2020.

[15] Cormac Flanagan and James B. Saxe. "Avoiding Exponential Explosion: Generating Compact Verification Conditions". In: *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '01. London, United Kingdom: ACM, 2001, pp. 193–205. ISBN: 1-58113-336-7. DOI: 10.1145/360204.360220. URL: http://doi.acm.org/10.1145/360204.360220.

[16] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. "A General Approach to Network Configuration Analysis". In: *NSDI*. 2015.

[17] Aaron Gember-Jacobson, Raajay Viswanathan, Aditya Akella, and Ratul Mahajan. "Fast Control Plane Analysis Using an Abstract Representation". In: *SIGCOMM*. 2016.

[18] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin Vechev. "NetComplete: Practical Network-Wide Configuration Synthesis with Autocompletion". In: *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. Renton, WA: USENIX Association, 2018, pp. 579–594. ISBN: 978-1-931971-43-0. URL: https://www.usenix.org/conference/nsdi18/presentation/el-hassany.

[19] Susan Horwitz, Thomas W. Reps, and David Binkley. "Interprocedural Slicing Using Dependence Graphs". In: *ACM Trans. Program. Lang. Syst.* 12.1 (1990), pp. 26–60. DOI: 10.1145/77606.77608. URL: http://doi.acm.org/10.1145/77606.77608.

[20] Peyman Kazemian, George Varghese, and Nick McKeown. "Header Space Analysis: Static Checking for Networks". In: *Proc. NSDI'12*.

[21] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. "VeriFlow: Verifying Network-Wide Invariants in Real Time". In: *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. Lombard, IL: USENIX, 2013, pp. 15–27. ISBN: 978-1-931971-00-3. URL: https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/khurshid.

[22] Shuvendu K. Lahiri, Robert Nieuwenhuis, and Albert Oliveras. "SMT Techniques for Fast Predicate Abstraction". In: *Computer Aided Verification*. Ed. by Thomas Ball and Robert B. Jones. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 424–437. ISBN: 978-3-540-37411-4.

[23] Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Jiaxin Cao, Sri Tallapragada, Nuno P. Lopes, Andrey Rybalchenko, Guohan Lu, and Lihua Yuan. "CrystalNet: Faithfully Emulating Large Production Networks". In: *Proc. of the 26th Symposium on Operating Systems Principles (SOSP)*.

[24] Jed Liu, William Hallahan, Cole Schlesinger, Milad Sharif, Jeongkeun Lee, Robert Soule, Han Wang, Calin Cascaval,

Nick McKeown, and Nate Foster. "p4v: Practical Verification for Programmable Data Planes". In: *Proceedings of ACM SIGCOMM 2018*.

[25] Nuno P. Lopes, Nikolaj Bjørner, Patrice Godefroid, Karthick Jayaraman, and George Varghese. "Checking Beliefs in Dynamic Networks". In: *Proc. NSDI'15*.

[26] Nick McKeown, Dan Talayco, George Varghese, Nuno Lopes, Nikolaj Bjorner, and Andrey Rybalchenko. *Automatically verifying reachability and well-formedness in P4 Networks*. Tech. rep. 2016. URL: https://www.microsoft.com/en-us/research/publication/automatically-verifying-reachability-well-formedness-p4-networks/.

[27] Miguel Neves, Lucas Freire, Alberto Schaeffer-Filho, and Marinho Barcellos. "Verification of P4 Programs in Feasible Time Using Assertions". In: *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*. CoNEXT '18. Heraklion, Greece: ACM, 2018, pp. 73–85. ISBN: 978-1-4503-6080-7. DOI: 10.1145/3281411.3281421. URL: http://doi.acm.org/10.1145/3281411.3281421.

[28] Andres Nötzli, Jehandad Khan, Andy Fingerhut, Clark Barrett, and Peter Athanas. "P4Pktgen: Automated Test Case Generation for P4 Programs". In: *Proceedings of the Symposium on SDN Research*. SOSR '18. Los Angeles, CA, USA: ACM, 2018, 5:1–5:7. ISBN: 978-1-4503-5664-0. DOI: 10.1145/3185467.3185497. URL: http://doi.acm.org/10.1145/3185467.3185497.

[29] *ONOS*. Peacock. Open Network Operating System. 2019.

[30] *P4_16 Language Specification*. v1.1.0. The P4 Language Consortium. Nov. 2018.

[31] *P4Runtime Specification*. v1.0.0. The P4.org API Working Group. Jan. 2019.

[32] Radu Stoenescu, Dragos Dumitrescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. "Debugging P4 Programs with Vera". In: *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. SIGCOMM '18. Budapest, Hungary: ACM, 2018, pp. 518–532. ISBN: 978-1-4503-5567-4. DOI: 10.1145/3230543.3230548. URL: http://doi.acm.org/10.1145/3230543.3230548.

[33] Radu Stoenescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. "SymNet: Scalable symbolic execution for modern networks". In: *SIGCOMM*. 2016. DOI: 10.1145/2934872.2934881. URL: http://doi.acm.org/10.1145/2934872.2934881.

[34] Mark Weiser. "Program Slicing". In: *Proceedings of the 5th International Conference on Software Engineering*. ICSE 81. San Diego, California, USA: IEEE Press, 1981, pp. 439–449. ISBN: 0897911466.

APPENDIX

Appendices are supporting material that has not been peer-reviewed.

The central point of our solution consists in processing a P4 program and synthesizing the formulas OK describing the set of good runs, and BAD describing the set of bad runs. The formulas are computed as weakest preconditions to reach a specific state. We claim to preserve the semantics of the program in terms of the formula OK, meaning that all the packets that did not trigger a fault in the original program will be processed in exactly the same way in the modified program. To this end we developed the algorithms Infer along with the optimized version Fast-Infer that compute the necessary preconditions to control bug, and Fixes that infer a set of extra keys which transform all uncontrolled bugs in the program into controlled bugs. The formalization and theorems that state their correctness follow.

Theoretical framework

We represent a program as a control flow graph (CFG)[2] where each node represents a program location and each edge $u \rightarrow v$ represents a possible transfer of control between nodes. Denote by $nodes(CFG)$ the set of all nodes in the CFG and by $edges(CFG) \subseteq nodes(CFG) \times nodes(CFG)$ its edges.

A p4 program defines a finite set of program variables - usually fixed-width bit vectors, booleans and enumerations. During the execution of a program, the variables bear values in their declared domains. Denote by $V = v_1, v_2, ..., v_n$ the set of all program variables. A state is a mapping between program variables V and their domains and an extra variable depicting the current program location in the CFG. Formally, $\Sigma = (v_1 \rightarrow dom(v_1)) \times (v_2 \rightarrow dom(v_2)) \times ... \times (v_n \rightarrow dom(v_n)) \times nodes(CFG)$ is the set of all possible states. Define accessor function $loc : \Sigma \rightarrow nodes(CFG)$ the control location of a state.

Each edge $u \rightarrow v$ is labeled with a transformation of the program state - $\tau : edges(CFG) \rightarrow (\Sigma \rightarrow \Sigma)$. Denote by $start \in nodes(CFG)$ the start node of the program. A terminal node in CFG is a node which has no outgoing edge. Denote by E the set of terminal nodes representing error nodes, and by G the set of all terminal nodes other than error nodes.

The semantics of a packet processing pipeline is given by its set of admissible traces. A trace π is a list of states $[\pi_1, \pi_2, ..., \pi_n]$ such that:

(1) $(loc(\pi_i), loc(\pi_{i+1})) \in edges(CFG)$
(2) $\pi_{i+1} = \tau[loc(\pi_i), loc(\pi_{i+1})](\pi_i)$
(3) $loc(\pi_1) = start$
(4) $loc(\pi_n)$ is a terminal node

We call a trace bad if $loc(\pi_n) \in E$ and good if $loc(\pi_n) \in G$. Let S denote the set of good traces and F the set of bad traces.

Let $\Gamma \subseteq V$ denote the set of control variables and $A \subseteq nodes(CFG)$ the set of assertion points.

Let $X \subseteq V$. Denote by $\Phi[X]$ the set of all formulas ϕ such that all free variables of ϕ are in X. The function $eval : \Phi[X] \times \Sigma \rightarrow \mathbb{B}$ defines the usual boolean expression evaluation modulo some underlying theories.

Definition 7.1. *Let $\alpha \in \Phi[V]$ and $a \in nodes(CFG)$. A trace π is said to be assertion compliant to α at point a iff $\forall \pi_i \in \pi$, then $(loc(\pi_i) = a \rightarrow eval(\alpha, \pi_i) = true)$*

Definition 7.2. *A mapping $\phi : A \rightarrow \Phi[\Gamma]$ is a **controlled necessary condition** iff $\forall a \in A, \forall \pi \in S$, π is assertion compliant to $\phi(a)$ at point a.*

Definition 7.3. *A controlled necessary condition ϕ is minimal if $|\{\pi \mid \pi \in F \wedge \forall a \in A, \pi$ is assertion compliant to $\phi(a)$ at point $a\}|$ is minimal.*

Informally, the above two definitions mean: we need to place assertions at table call entry points in such way as to *kill* away as many

failing paths as possible while not *killing* any of the good states. Furthermore, we constrain the assertions to only be expressed in terms of control variables Γ.

Assume that for all $b \in E$ bug points, there exists a unique $a \in A$ such that a dominates b. A node x is said to dominate a node y iff all traces which contain location y also contain x. Define $controlled(a)$ the set of bug points dominated by assert point a.

Definition 7.4. *Let $\rho : nodes(CFG) \rightarrow \Phi[V]$ a formula with $\forall q \in \Sigma.\ eval(\rho(n), q) \rightarrow \exists$ a trace π such that $\pi_1 = q$ and $loc(\pi_k) = n$ for some k. We call $\rho(n)$ the reachability condition of n.*

The formula $OK = \bigvee_{g \in G} \rho(g)$ (G is the set of all *good* terminal nodes) is called weakest precondition and describes the set of all initial states from which the program terminates successfully.

Definition 7.5. *Let $\mathcal{P}$ a subset of atomic formulas in $\Phi[\Gamma]$. A $\mathcal{P}$-approximation of formula $\Omega \in \Phi[V]$ w.r.t. $\mathcal{P}$ is a CNF formula H where all atoms are $\subseteq \mathcal{P}$ and $\Omega \models H$.*

Theorem 7.1. *A mapping $\phi : A \rightarrow \Phi[V]$ such that $\phi(a)$ is a $\mathcal{P}$-approximation of formula $OK \wedge \rho(a)$ with all free variables in Γ is a controlled necessary condition.*

Theorem 7.2. *Algorithm 1 computes a $\mathcal{P}$-approximation of formula OK.*

Proof. We begin by proving the following loop invariant at line 4: $OK \models \phi_n$, where ϕ_n is the value of variable ϕ after n iterations through the loop.

Initial step: $\phi_0 = \top$. All formulas $\models \top \Rightarrow$ so does OK.

Induction hypothesis: $OK \models \phi_n$ holds after n iterations.

Induction step: Assume that $OK \models \phi_{n+1}$ does not hold. If the control flow follows the else branch at line 11, then $\phi_n = \phi_{n+1}$, then this contradicts the induction hypothesis.

Assume that the control flow follows line 8. This means that, by definition of the unsatisfiable core, $\bigwedge_{a \in uc} a$ is inconsistent with $OK \Leftrightarrow$

$$OK \models \neg \left(\bigwedge_{a \in uc} a \right) \tag{1}$$

But it is also the case that: $\phi_{n+1} = \phi_n \wedge \neg (\bigwedge_{a \in uc} a)$. But due to the induction hypothesis, $OK \models \phi_n \Rightarrow OK \models \phi_n \wedge \neg (\bigwedge_{a \in uc} a)$, contradiction. □

Theorem 7.3. *Algorithm 2 computes a necessary precondition of table t.*

Proof. We prove that each $\neg pc$ from line 8 is a necessary condition for table t. Furthermore, since ϕ is a conjunction of necessary preconditions, it is also a necessary precondition.

Take pc as above the path condition of path π starting from the assertion point and ending in bug. Furthermore assume all variables of pc are controlled. Assume that $\neg pc$ is not a necessary condition. Then for an entry satisfying pc there must exist a "good" run from the assertion point. But pc is a sufficient condition to go down π, which implies that all paths starting at entries satisfying pc end up on path π, which is a buggy run. Since a run cannot be both good and buggy, this contradicts our assumption and proves the fact that $\neg pc$ is a necessary precondition. □

Theorem 7.4. *Algorithm 2 computes multi-table necessary preconditions of simple dependencies.*

Proof. For a table t, let $\Gamma_t \subseteq \Gamma$ be the set of control variables of table t, and let $V_t \subseteq V$ be the set of program variables s.t. $\forall v \in V_t \wedge reach(t) \implies \exists \tau$ s.t. $v = \tau$, where τ is a first-order term in the theory of bit-vectors. The set X_t of free variables occurring in τ are such that $X_t \subseteq \Gamma_t$. Denote by $reach(t)$ the first-order formula for the reachability of table t.

Let $Y1 = t1.apply(X_{t1})$ and $Y2 = t2.apply(X_{t2})$ be two table applications of tables $t1$ and $t2$ with keys X_{t1} and X_{t2} respectively such that $reach(t2) \models reach(t1)$ and $X_{t1} \subseteq X_{t2}$. Let $\phi_{12} \leftarrow$ fast-Infer$(cfg_{t2}, t2, \Gamma_{t2} \cup V_{t1})$. Then ϕ_{12} is a necessary condition wrt $t2$.

We focus on the path condition pc inferred at line 8 in fast-Infer. Assume that the control flow reaches the application of $t2$ and pc holds. By the assumption that $reach(t2) \models reach(t1)$, it follows that the control reaches the application of $t2$. Given that the values of all variables in V_{t1} are functionally dependent on X_{t1} (because of the semantics of table applications) and $X_{t1} \subseteq X_{t2}$, then their values are functionally dependent on the keys of table $t2$. This is equivalent to saying that $\Gamma_{t1} \leftarrow \Gamma_{t1} \cup V_{t2}$. So now, the path condition only contains variables from table $t2$ and all paths hitting an entry matching pc will go to a bug. □

Global correctness

Assume that a P4 program only contains controlled bugs. Let *Specs* denote the set of inferred table filters by algorithm Infer. Our shim can be seen as a function: $accept : Snapshots \rightarrow \mathbb{B}$ defined by

$$accept(snap) = \bigwedge_{s \in Specs} eval(snap, s)$$

If the shim accepts a given snapshot, then we will show that this guarantees that there is no packet which can trigger a bug.

Theorem 7.5 (Global correctness). $\forall snap \in Snapshots\ accept(snap) \Rightarrow \forall p \in Packets\ \neg eval((snap, p), bug)$.

Proof. Assume that $accept(snap)$ holds and there is a packet p such that $eval((snap, p), bug)$ holds. This means that there exists a bug b in the program which is reachable (that is $reach(b)$ holds). By definition, a controlled bug is such that there exists a set of conditions $Specs_b \subseteq Specs$ such that $Specs_b \models \neg reach(b)$. By definition of function accept $accept(snap) = \bigwedge_{s \in Specs} s \models \bigwedge_{s \in Specs_b} s \models \neg bug$. So it means that b is both reachable and unreachable, contradiction. □