Debugging P4 programs with Vera

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

ABSTRACT

We present Vera, a tool that verifies P4 programs using symbolic execution. Vera automatically uncovers a number of common bugs including parsing/deparsing errors, invalid memory accesses, loops and tunneling errors, among others. Vera can also be used to verify user-specified properties in a novel language we call NetCTL.

To enable scalable, exhaustive verification of P4 program snapshots, Vera automatically generates all valid header layouts and uses a novel data-structure for match-action processing optimized for verification. These techniques allow Vera to scale very well: it only takes between 5s-15s to track the execution of a purely symbolic packet in the largest P4 program currently available (6KLOC) and can compute SEFL model updates in milliseconds. Vera can also explore multiple concrete dataplanes at once by allowing the programmer to insert symbolic table entries; the resulting verification highlights possible control plane errors.

We have used Vera to analyze many P4 programs including the P4 tutorials, P4 programs in the research literature and the switch code from https://p4.org. Vera has found several bugs in each of them in seconds/minutes.

CCS CONCEPTS

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

1 INTRODUCTION

Programmable network dataplanes such as those enabled by P4 [2] promise to help networks meet ever-increasing application demands. On the downside, unverified changes to network functionality can introduce bugs that may cause great damage. Recently, faulty routers in two airline networks have grounded airplanes for days (for both Delta and Southwest Airlines), showing just how disruptive the effects of incorrect network behavior can be. Given the momentum behind programmable networks, we expect such faults and many others will cripple programmable networks.

In this paper, we argue that dataplane programs should be verified before deployment to enable safe operation. We present Vera, a verification tool that enables debugging of P4 programs both before deployment and at runtime. At its core, Vera translates P4 to SEFL, a network language designed for verification, and relies on symbolic execution with Symnet [31] to analyze the behavior of the resulting program. Vera incorporates a set of novel techniques that together enable scalable and easy-to-use P4 verification.

Vera exhaustively verifies a snapshot of a running P4 program (i.e. the program and a snapshot of all its table rules): it uses the parser of the P4 program to generate all parsable packet layouts (e.g. header combinations), and makes all header fields symbolic (i.e. they can take any value). It then tracks the way these packets are processed by the program, following all branches to completion. To improve scalability, Vera introduces a novel match-forest data structure that concurrently optimizes both update and verification time.

Vera automatically checks for common problems in P4 programs including loops, parsing/deparsing errors, tunneling bugs, overflows and underflows, among others. Since verification is exhaustive, if Vera does not report problems, it guarantees the P4 program snapshot does not include bugs from these categories.

Even if a snapshot of a P4 program is bug-free, there is no guarantee that the same holds for other snapshots (i.e. sets of table rules). With Vera, users can explore multiple table snapshots by specifying symbolic table rules instead of concrete ones. Such exploration is much more costly, because in the extreme it can test *all possible dataplane behaviours of the P4 program*, but it is also very powerful: if it finishes, it can prove that the program does not have any of the bugs Vera catches, regardless of the dataplane. Symbolic table entries are also useful in checking whether a P4 program conforms to some user-specified policy that is specified in a subset of CTL (§4.1).

SIGCOMM '18, August 20–25, 2018, Budapest, Hungary

ACM ISBN 978-1-4503-5567-4/18/08…$15.00
https://doi.org/10.1145/3230543.3230548

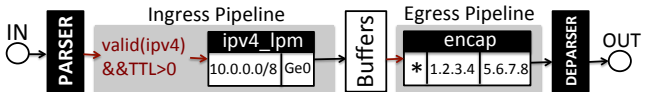

Figure 1: ENCAP: P4 that encapsulates IP packets.

We have used Vera to analyze many P4 programs including the P4 tutorials, a load balancer (Beamer [27]), a packet-trimming switch (NDP [12]), P4xos [7], an implementation of Paxos, and the complete datacenter switch implementation provided by p4.org. In seconds, Vera has found bugs in each of these programs, with little or no specification effort on our side. In an accompanying technical report, we manually prove that Vera correctly translates from P4 to SEFL by defining the operational semantics for both P4 and SEFL and by proving that core P4 instruction(s) and their SEFL translation are equivalent [30].

2 BACKGROUND AND MOTIVATION

An example P4-14 program[1] is shown in Figure 1 and has a few main parts: the parser, the ingress pipeline, the egress pipeline and the deparser. The parser transforms the packet from bits into headers according to a parser specification provided by the programmer (see Fig. 2). The parser specification also dictates how packets are *deparsed* from separate headers into a bit representation on output. After input parsing, a ingress control function decides how the packet will be processed; the control instructions in our example are shown in red. Then, the packed is queued for egress processing. Upon dequeuing, it is processed by an egress control function and then it leaves the switch.

Both ingress and egress control functions direct the packet through any number of match-action tables. The control functions conditionally steer the packet to various tables based on header values, metadata or table match outcomes; packet contents cannot be changed in the control function. In our example, there is one table used on input, `ipv4_lpm`, and one table on output called `encap`. In the ingress control function, packets with valid IPv4 headers and strictly positive TTL values are sent to the `ipv4_lpm` table; on egress, all packets are matched against the `encap` table.

Match-action tables are where the bulk of packet processing takes place. The P4 program defines which fields will be matched in any given table; for instance, the `ipv4_lpm` table matches the destination address field in the IPv4 header. The actual table rules are provided at runtime by a controller or statically, when the P4 is deployed; in our example, there is a single rule for prefix 10.0.0.0/8.

When a packet visits a table, it is matched against the existing table rules and upon a match the action associated

[1]Vera only supports P4-14 at this point; adding P4-16 support is part of our future work.

```
parser start {
 extract(eth);
 return select(eth.type){
  0x800 : parse_ipv4;
  default: ingress;
}}
parser parse_ipv4 {
 extract(ipv4);
 return ingress;
}
```

Figure 2: Parser.

```
action ipip_encap(srcIP,dstIP) {
  add_header(inner);
  //copy outer to inner header
  copy_header(inner, ipv4);
  //set outer header addr. and proto
  modify_field(ipv4.src, srcIP);
  modify_field(ipv4.dst, dstIP);
  modify_field(ipv4.proto, 0x5E);
}
```

Figure 3: Encap action.

with the rule is executed. Actions can change packet contents (modify fields, add or remove headers) or metadata, drop or clone the packet. For instance, when a packet destined to 10.0.1.1 visits the `ipv4_lpm` table, it will match the 10/8 rule and execute the associated action. This action is not shown in the example, but it sets the *egress_spec* metadata to that of interface "Ge0", among other changes. After table matching, packet processing resumes in the control function where the outcome of the match—the executed action, if any—can be used to decide how the packet will be processed next.

After ingress processing, if the packet is not dropped, it is placed in one of the queues for the egress interface based on the *egress_spec* metadata (this and other implicit edges are shown as black arrows in Fig.1). It then continues to egress processing, as dictated by the egress control function. In Fig. 1 the packet will visit the `encap` table where all packets execute the `ipip_encap` action (code in Fig. 3) that adds an inner IP header, copies the outer header to the inner header, and then modifies the outer header with addresses given as parameters. On egress, other processing includes changing the ethernet addresses and computing checksums.

Debugging P4 programs is hard. Despite its apparent simplicity, programming P4 is often counter-intuitive and unexpectedly difficult, and debugging P4 programs is particularly challenging. In traditional programming, hardware traps interrupt the program when a critical error such as unmapped memory access or division by zero is attempted, and debuggers can quickly track the source of such errors. Unfortunately, this is not the case with P4 programmable switches: when present, errors are handled silently at runtime: the packet triggering the offending behavior is either dropped or modified in unspecified ways; in both cases, tracking the location of the bug is very difficult, even when using the software P4 switch (the behavioral mode switch). Loops are also very difficult to catch: a single looping packet will slightly decrease the throughput; only when the pipeline fills with looping packets the throughput will collapse and the effects will be easily visible externally.

We have discovered a variety of bugs in the P4 programs we have access to. The most common mistakes seem to be parser bugs, invalid header accesses and encapsulation bugs, but we have also seen out-of-bounds register accesses and loops caused by recirculated packets. Most of these bugs are facilitated by traits of the P4 language that we discuss next.

An overarching problem of P4 is that it is not memory-safe. When an uninitialized header field is read, the switch behavior is unspecified; the packet could be dropped, or a quasi-random value returned. Since the switch does not throw any runtime exceptions, catching and fixing such bugs, especially if they are triggered rarely, is a nightmare.

Another fundamental problem is that P4 programs are underspecified. First, functionality is split between the P4 program and the match-action rules inserted at runtime. At compile time only the program is available and, sometimes, a few static rules that will be inserted in the P4 tables at startup. The runtime rules are unknown as they will be generated and inserted by the control plane, a separate piece of software. The programmer thus has to understand the behavior of a seemingly underspecified program, which is far from easy.

Secondly, a lot of P4 processing is performed implicitly, without being explicitly requested by the programmer. In our example, packets without a valid IPv4 header, or with zero TTL will arrive at the buffers without the `egress_spec` metadata value set, and will be implicitly dropped. Such implicit drop behavior seems convenient, but it can result in the program dropping useful packets and is difficult to debug. To avoid such errors, the programmer should explicitly drop unwanted packets instead, but reasoning about all such packets is difficult at compile time.

Thirdly, dropped packets in the ingress pipeline continue match-action processing (because of the difficulty of removing packets from the pipeline), and they may match entries in downstream tables. Such behavior may lead to errors where packets dropped by one table (e.g. `acl`) are later revived unintentionally by another table (e.g. `lpm`).

Another peculiarity that is unique to P4 is that parsing must account for all header layouts the P4 program can accept on input *and* emit on output, despite the fact that the code is written in terms of ingress parsing (see Fig. 2). A common mistake, also present in our example, is when the header layout of outgoing packets is not captured in the parser spec: in our case, we do not parse the inner IP header. At deployment, this bug will make our program output packets that only contain the outer (encapsulation) header. Finding the root cause of the bug is quite difficult, but fixing it is easy enough. For this, the `parse_ipv4` code in Fig.2 could be replaced by:

```
parser parse_ipv4 {
  extract(ipv4);
  select (ipv4.protocol){
    0x5E: parse_inner_ipv4;
    default: ingress;}
parser parse_inner_ipv4 {
  extract(inner_ipv4);
  return ingress;}
```

The new code correctly *deparses* encapsulated packets, but has an unintended consequence: incoming IP-in-IP packets will have their inner header overwritten: in the encapsulation action, when a new inner header is added and its fields modified, they will overwrite the existing inner header. We can fix this bug by checking that the inner header is invalid before encapsulation, or by using a header array instead.

`encap` shows how easy it is to make mistakes even in very simple P4 programs. Our analysis has found bugs in all the P4 tutorials (found at https://p4.org/code/), despite their simplicity and reduced size. As runtime debugging of P4 programs is tricky, developing even simple, correct P4 programs is a challenging task.

Verification approaches. If P4 is to meet its goal of enabling programmable networks that replace today's reliable, ossified ones, we must ensure P4 programs are easy to debug and fix. An ambitious goal is to *catch all bugs at compile time*.

In verification, there is a fundamental trade-off between specification effort and verification complexity. Iterative design and specification approaches such as Cocoon [29] require massive input from the programmer/verifier but are feasible computationally and guarantee that the generated code matches the specification. As network processing changes quickly, such approaches are both unlikely to keep up and will be too expensive to use in most networks.

At the other end of the spectrum, we can use testing and simply inject all the packets that the program's parser will accept and then check if their processing leads to problematic behavior. Such testing requires almost zero specification and it covers all possible behaviors, but will scale poorly.

Symbolic execution is an excellent middle-ground: it can potentially explore the processing of all possible packets and does not require programmer input for verification. For traditional programs, symbolic execution offers much better code coverage compared to testing [3], but it rarely explores all possible paths, and thus it is not exhaustive—it does not guarantee absence of bugs. Network dataplanes are however much simpler than standard C code, e.g. they do not include loops. Exhaustive dataplane symbolic execution has been shown to be feasible for moderate sized networks [31, 8].

3 VERA: SYMBOLIC EXECUTION FOR P4

We present Vera, a verification tool that uses symbolic execution to test the behavior of P4 programs for all possible packets. To run Vera, the user passes as arguments the name of the P4 source file together with a set of commands that insert table rules at program startup. Vera first translates the P4 program together with the provided table rules to the SEFL language, obtaining an equivalent program (see §3.2 for details). The resulting SEFL program is a collection of virtual ports, each with associated SEFL instructions.

A sketch of the SEFL version of the `encap` program is given in Figure 4, where edges denote how packets may flow

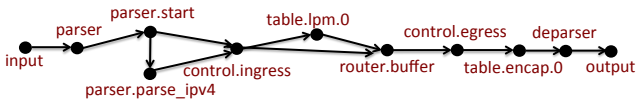

Figure 4: Ports generated by Vera for Encap P4.

through the program. Snippets of SEFL code generated by Vera for encap are given in Figures 6 and 7.

Vera examines the P4 parser state machine and generates one symbolic packet for each header layout that can be accepted by the program; in the encap example, it will generate two possible packets: one containing an ethernet header and another one containing an ethernet header followed by an IP header. The symbolic packets have all their fields set to symbolic values, meaning they take any value in their domain. Vera injects these packets into the input port of the program and uses Symnet [31] for symbolic execution.

In Symnet, one execution path represents one symbolic packet and its associated metadata, possibly with constraints for the header or metadata values (we use packet and path inter-changeably). As long as it has active packets, Symnet selects one of them and executes the next SEFL instruction for that packet. All packets are processed until completion. Completed packets can either be failed or successful. Failed packets were either dropped or have triggered a bug and were terminated by Symnet. Successful packets have fields with feasible constraints or concrete values, and no more instructions to execute (e.g. at the output port).

When symbolic execution finishes, the output is a list of packets. For each packet we know if it is failed or successful, the ports it has visited, the instructions it has run, and the state of each header field and metadata: concrete values or constraints for the symbolic header fields.

In Fig. 5 we show the output of Vera when injecting an ethernet/IP packet in the encap program. The initial packet is shown as a white box. Path constraints are shown as annotations on the different edges. All final states are shown in red and represent failed packets. Vera finds four different packets; each of these takes a different path from the input to the output states in Fig. 5. Three of these paths result in implicit packet dropping in the buffering mechanism: when eth.type is set to a value different from 0x800, when ipv4.TTL is zero or less, and when the destination address does not match 10/8. A single packet makes it to the deparser; its constraints are eth.type=0x800, ipv4.TTL>0 and ipv4.dst matches 10/8. Despite this, the packet fails in the deparsing stage because the inner IPv4 does not exist in the parser specification.

In the rest of this section we describe in more detail why translating P4 to SEFL is the right choice (§3.1), how the translation is performed (§3.2), how we can deal with missing table entries (§3.4) and how symbolic execution can besped up by smartly generating the match-action table code (§3.3).

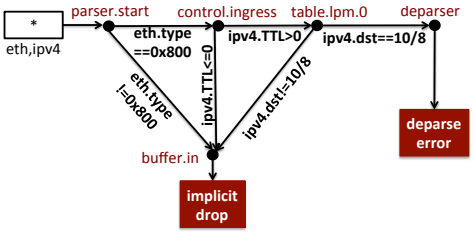

Figure 5: Symbolic execution of encap finds four failed paths for ethernet/ip packets.

3.1 Why Symnet/SEFL?

In principle, any symbolic execution engine can be used with similar results. We chose SEFL/Symnet because it has been optimized for network dataplanes, showing superior performance to llvm/Klee [31]. Secondly, Symnet offers a memory model very similar to that of P4 (packet headers and metadata) and, in addition, it offers memory safety—unallocated or misaligned header accesses are automatically caught, as are header overlaps. These traits considerably simplify bug catching during symbolic execution, allowing us to focus most of our effort a correct translation from P4 to SEFL. Finally, there already exists a wide range of compilers that take FIB snapshots, Click modular router configurations and output SEFL, meaning we can integrate our P4 models in larger legacy networks and perform network-wide dataplane verification without any added cost.

3.2 Translating P4 to SEFL

The expressive power of P4 is very similar to that of SEFL. This is perhaps not surprising as both languages have been designed to capture data plane processing, albeit for different end goals: SEFL aims to enable cheap verification while P4 aims to be easily deployable in hardware. Neither SEFL nor P4 have loop instructions.

SEFL allows creating any number of named ports, which have associated SEFL instructions. The packet gets directed to these ports explicitly, by using the forward instruction, or implicitly, via directional links where each link connects two ports. Our parser uses ports to preserve the layout of the P4 program in the SEFL program it outputs.

SEFL has two types of variables: packet headers and metadata. Metadata are key/value pairs where the keys are strings and values can take any type. Packet headers are allocated in a linear address space, and all fields have absolute starting addresses. SEFL offers memory safety: any access to unallocated metadata or packet header fields terminate the current execution path. Additionally, packet header accesses must always be aligned to be allowed, and field allocation ensures neighboring fields do not overlap. P4 also has two types of variables: metadata, which we naturally map to SEFL metadata, and packet header fields.

```
parser.parser_start:
    exists(current,48);
    exists(current+48,48);
    exists(current+96,16);
    eth.src = @current;
    eth.dst = @(current+48);
    eth.type = @(current+96);
    current += 112;
    valid.eth = 1;
    If (eth.type==0x800)
        Forward(extract_ipv4);
    Else
        Forward(control_ingress);
```

```
parser.parse_ipv4:
    exists(current,4);
    ...
    exists(current+128,32);
    ipv4.version = @current;
    ...
    ipv4.dst = @(current+128);
    current += 160;
    valid.ipv4 = 1;
    Forward(control_ingress);
```

Figure 6: Generated code for the P4 parser.

When a packet enters a SEFL-P4 program, it is encoded as a series of successive header fields (as it is in practice). Our translation for the P4 parser code generates one port for each path in the parse tree. The packet is directed from input to the `parser.start` port. The `current` parse location in the header is remembered as a SEFL tag which is a pointer to a location in the packet. When the packet enters the P4 box, we create the current tag as follows: `CreateTag(current, START)`; START is a tag maintained by SEFL that points to the beginning of the packet.

A known problem with any verification approach is that checksums cannot be verified when header fields are symbolic; however this verification is a crucial part of the parsing process. Instead, we validate headers by checking that header fields are allocated at the right locations before we extract a P4 header, as suggested by Symnet [31]. In particular, to implement the `extract` call, we generate one check for each header field to be extracted (see Fig. 6). For this, we use the `exists` instruction in SEFL that checks for an allocated variable of given size at a certain position in the header.

If the packet header does not match the expected layout, one of the `exists` functions will fail, and the error will be logged. If all the header fields exist, the parser implementation in SEFL then proceeds to create a SEFL metadata value for each parsed header field; the name of the metadata is *header_instance.field_name*. When parsing is conditional, constraints are added as shown in the example for the *eth.type* field. After parsing has finished, all parsed headers are available as SEFL metadata. In addition, the SEFL code initializes P4 metadata and other information from the parser (e.g. which headers are valid), and forwards the packet to the `control.ingress` port that contains the SEFL code for the ingress control function. Translating control flow instructions is straightforward, as there is a one-to-one mapping between SEFL and P4 instructions here. We show the code for the ingress pipeline of `encap` in Fig.7.

To translate match-action table processing, Vera generates a new port for each `apply` call and associates with it the SEFL code implementing match-action processing (e.g. `table.lpm.0` in Fig.7). The port name is guaranteed to be unique as it is formed by concatenating the table name and a per table sequence number. The parser also creates a new port in the control function (`control.ingress.1` in our example), where processing will resume after table processing. To execute an apply call, we insert in the control function a `forward` to the respective table invocation port. Table processing will finish with a `forward` to the control function. Vera directly translates all P4 primitive actions to SEFL.

```
control.ingress:
  If (valid.ipv4==1&&ipv4.ttl>0)
      Forward(table.lpm.0);
  Else
      Forward(buffer.in);
control.ingress_1:
   Forward(buffer.in);
```

```
table.lpm.0:
  If (ipv4.dst match 10/8){
       lpm.0.Hit = 1;
       lpm.0.action.lpm = 1;
       ipv4.ttl--;
       meta.egress_spec = 1;
  } Else lpm.0.Hit = 0;
  Forward(control.ingress.1);
```

Figure 7: Generated code for the ingress pipeline.

Registers are arrays of predefined *size*, and translating them is tricky because SEFL does not have array support for metadata. Vera creates one metadata for each array entry; the name of each metadata includes its location in the array (e.g. the variable `a[0]` holds the first value in the array, `a[1]` the second, and so forth). To access a register value, we insert a series of `if/else` instructions that successively test the value of the index at runtime against all possible locations, and then access the correct array location. While inneficient, this solution does not increase the number of explored paths when the index is concrete.

The various `clone` actions are implemented using the `fork` instruction which creates a new execution path that is a copy of the current path. On the cloned path the `instance_type` metadata is set appropriately. and the packet is then redirected either to buffering or parser input. Packet redirection actions such as `resubmit` or `recirculate` are implemented by `forward` to the parser input port.

The `drop` action, when applied on ingress, is implemented by setting the `egress_spec` to a predefined value (511). The packet is only dropped when it reaches the buffering mechanism, as per the P4 spec. In the egress pipeline, `drop` is implemented using the `fail` instruction that terminates the current path and prints an error message.

Whenever a header instance must be added or removed, our SEFL code first checks that the header is valid (for removal) or invalid (for addition), throwing an exception otherwise. Creating the header instance is easy: we generate SEFL code to allocate SEFL metadata for each field; the name will be *instance.field*, and is guaranteed to be unique by the P4 compiler. Copying headers is similarly trivial to translate.

Dealing with header arrays, however, is a bit trickier because SEFL does not have support for arrays. To circumvent this problem we have implemented a working but inefficient solution that relies on the fact that header arrays have a predefined size and additions/removals are always done at locations known at compile time. Below is the implementation for `add_header(ip[0])` for an array of maximum two

headers: the code simply treats all possibilities, in parallel. The first path will succeed if there is no header in the header array yet, otherwise it will silently fail. The second path will only succeed when the first header is valid but the second one is not, and the third will throw an exception when both headers are allocated and adding a third is not possible.

```
Fork { Path1: If (!Exists(ip[0])) { ip[0].valid = 1;}
       Path2: If (Exists(ip[0]) && !Exists(ip[1])) {
                ip[1].valid = 1;copy_header(ip[1],ip[0]);
                forall f in fields(ip[0]):
                  ip[0].f = 0;}
       Path3: If (Exists(ip[0]) && Exists(ip[1])){
                Fail("Attempting to add header in a full array")}}
```

While inneficient, this solution works well enough for the examples we tested, mostly because header array sizes used in practice are fairly small. A more elegant solution requires array support in SEFL—we leave this to future work.

The final step in the pipeline, deparsing, is the inverse process of parsing. First, the SEFL program searches for a path through the topologically sorted parse tree which matches valid header fields. If such a path is not found, a deparsing exception is raised; otherwise, the P4 spec guarantees that a single path is possible, and the code simply copies the metadata to the packet layout, before releasing the packet to the specified egress interface. The deparser also raises an error when it finds an extra valid header that does not match the selected parse tree path. This catches a common class of deparser bugs, such as those discovered in Beamer (see §5).

Unsupported features. Currently, Vera does not handle hash computations (e.g. for ECMP). As a result, all checksums are always set to symbolic values, and header validity does not validate checksums, using the header field alignment to detect bad packets instead. Vera does not support ECMP either; a possible solution is to fork the packet on all possible ECMP paths, and explore each path independently.

3.3 Fast verification of match-action tables

The match code for match-action processing is easy to translate into a series of `If/Else` SEFL instructions. The resulting code will have at least as many `If` instructions as table entries, and this will make symbolic execution explore a separate execution path for each table entry. With enough table entries, symbolic execution will be infeasible to use.

To ensure symbolic execution actually scales to large P4 programs, we need to optimize the match code while preserving its functionality. There are two general directions of optimization: a) reducing the number of paths explored by symbolic execution and b) reducing the number of constraints that need to be checked by the solver on each path.

The first part is fairly easy: we generate exactly one path for each distinct action invocation in the table rules. For an ACL table that has two actions (`nop` and `drop`), we will generate the following code:

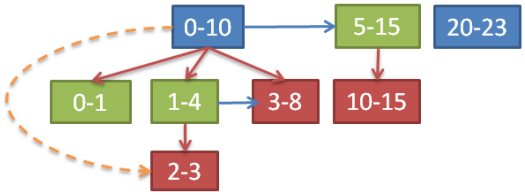

Figure 8: Match condition forest

```
Fork { Path1(nop):  Constrain(...);
       Path2(drop): Constrain(...); drop();}
```

For routing, the forward action has a parameter that specifies the output interface. In this case, our code will group all forward actions that have the same parameter into a single path, generating in effect one path for each output interface.

The second part of code generation is to ensure the path constraints are correct. This step is not trivial as the code must not only include the constraints for the associated rules, but also negated constraints for higher priority rules. As an example, the default route should forward a packet only when no other forwarding rule matches; the constraints in this case must include the negation of all other forwarding rules which have higher priority. From a table-wide perspective, if we have many overlaps, the worst case number of constraints is quadratic in the number of entries. With FIB sizes in the order of 100K, this is a show-stopper.

Our solution builds upon existing work [17, 4] and is applicable to a wide range of matching strategies including longest-prefix match, range, etc. At the core of our solution lies a data structure consisting of a forest of trees. To ease presentation, we show an example that matches a single field in Fig.8. In the figure, one node represents the match condition for one table rule and the colors represent rule priority (red>green>blue). In this data structure any pair of nodes falls in one of the four situations below:

(1) The nodes are completely **independent** if their corresponding conditions do not overlap.
(2) **Parent-of** - a node is the parent of another if the value domain corresponding to the child is strictly a subset of the one for the parent (red links). The parent must have lower priority than the child.
(3) **Ancestor-of**(dotted line) - when two nodes are connected in the same tree by several 'parent-of' links.
(4) **Neighbor-of**(blue) - nodes that have overlapping conditions, but neither condition is fully contained by the other; and neither node has an ancestor linked to the other node via a 'neighbor-of' link. As with 'parent-of' links, the source has priority lower than the destination.

To add one node to our data structure, we start at the top of the forest, checking which nodes overlap with the new one (we implement this efficiently using interval trees). If there is no overlap, we add the new node as a standalone one. Otherwise, the new node will become either a parent, a child or

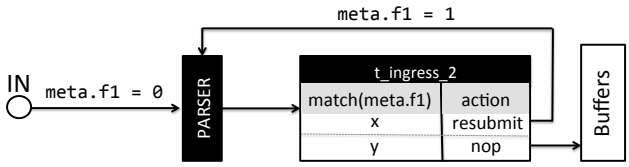

Figure 9: Using symbolic table entries to analyze the Resubmit P4 tutorial.

a neighboring node. If it becomes a parent or neighbor node, the node is inserted at the current level and the appropriate links are created. If it is a child, then the algorithm continues recursively in the subtree rooted at its newly found ancestor. Complexity is logarithmic in the number of nodes.

Given this forest, it is trivial to construct the minimal constraint required to match any given node: add the node's condition and the negated constraints of all its children and neighbors. We have proven that our algorithm generates the theoretical minimum number of constraints. Intuitively, this is because we only add the negated constraints of the direct children and neighbors, and not those of ancestors. In figure 8 red nodes have the highest match priority, then green and lastly blue ones. At first, node [0-10] should add negated constraints for all its sub nodes, plus all the nodes in the tree rooted at [5-15]. Looking closer, negated constraints for [10-15] and [2-3] are redundant since the constraints corresponding to their 'parent' nodes mitigate the overlap.

We implemented the forest construction algorithm in its most general form, that can handle range, longest-prefix and wild card matching and analyze its scalability in §5.

3.4 Symbolic table entries

Vera provides exhaustive program analysis for any given table rule snapshot, but such snapshots do not cover all possible rules that may be inserted; worse, at compile time the only available rules are static rules that help test common-case functionality, so catching difficult bugs requires carefully chosen rules by the programmer or some form of continuous snapshotting in deployments; the first approach doesn't scale, while the latter will lead to faulty dataplanes being actively used. Worse, even if a P4 program is deemed correct for many interesting snapshots of its match-action tables, it may not be correct for other snapshots.

To enable exploring a larger space of possible table snapshots, Vera allows the programmer to insert symbolic entries in match-action rules for both field matches and action parameters. A symbolic rule uses the same format as its concrete counterpart with the difference that both the match entries and action parameters can take symbolic values.

Consider the example in Fig. 9 where we show a part of the ingress pipeline of the `Resubmit` P4 example application. The match criterion for the table `t_ingress_2` is the metadata field `meta.f1`. The corresponding action can either do nothing (sending the packet to buffering) or resubmit the packet to the parser, also modifying `meta.f1` to 1.

To better understand the behavior of this program, the programmer has added two symbolic rules in the table, one for each possible action, and `x` and `y` are the respective symbolic match values. When exploring this P4 program, Vera will treat `x` and `y` as any other symbolic variable, collecting constraints and checking their feasibility. The two correct paths correspond to the packet directly hitting the `nop` action ($x! = 0$, $y == 0$), or being resubmitted and hitting the `nop` action (x==0,y==1). Vera also finds two faults, both corresponding to cases where the packet is implicitly dropped because it doesn't match any rule; the constraints for the two faulty paths are {x!=0, y!=0} and {x==0, y!=1} respectively.

Using this information, the developer of the controller program has valuable insights regarding which rules can break the intended functionality for the `t_ingress_2` table, and is assured that its code does not loop for *any* values of `x` and `y` for the existing rules. Note that this does not mean the program can never loop, regardless of its table entries. If we add one more symbolic rule with field `z` and action `resubmit`, Vera finds a loop where $x == 0$, $z == 1$ and $y! = 0, 1$.

Symbolic table entries allow the programmer to explore a wide range of dataplane behaviors and to reason about the actions the controller must take without verifying the controller itself (such verification is much more difficult because the controller is a general-purpose program).

In our example, we have manually inserted rules. Which rules are needed to explore all possible dataplane behaviors for generic P4 programs? To answer this question, first consider a program that does not *recirculate packets* and contains a single table T_1 that has n possible actions a_1, a_2, ..., a_n. In [30], we show that it suffices to add n symbolic entries (one per action) to explore all possible behaviors. If the program has another table T_2, we can apply the same rule: one symbolic entry per action will explore all possible behaviors. In fact, as long as we keep adding different tables and there is no recirculation, adding one symbolic entry per action per table is guaranteed to explore all possible behaviors. To achieve the same goal with recirculation, we need to add one additional symbolic rule for each recirculation. Symbolic table entries can produce all the possible behaviors of an arbitrary P4 pipeline, without knowledge about the controller behavior. The resulting constraints on the symbolic rules can pinpoint what concrete table entries will reproduce the behavior captured by Vera (see [30] for details).

3.5 Translator correctness

Guaranteeing correctness of code translation is a major precondition for correct software. In our work correctness is based on the formalization of P4 and SEFL semantics and

the proof of operational correspondence between the two semantics. We used the "big-step" operational semantics, for both P4 and SEFL statements. The semantics defines a relation of the form $\langle S, s\rangle \rightarrow s'$ where the pre-state s and post-state s' represent the states before and after execution of the program statement S. The definition of $\rightarrow$ is given by the associated semantic rules. The proof relies on the semantic equivalence of the statements. We consider two statements to be semantically equivalent if for all states s and s' $\langle S_1, s\rangle \rightarrow s'$ if and only if $\langle S_2, s\rangle \rightarrow s'$. We developed the "big-step" operational semantics for P4 and SEFL core statements and showed the operational correspondence between the two semantics. A specification of the semantics of the relevant statements of P4 and SEFL together with the proof of the operational correspondence can be found in [30].

4 DEBUGGING P4

By default, Vera inserts checks that capture a wide range of bugs in P4 programs and flags such bugs to the user as failed paths. For each failed path, Vera also generates a concrete packet that matches the path constraints which can be used to test the bug in P4 switches, be they software or hardware. Below are the types of errors that Vera catches automatically:

- **Implicit drops** are flagged when a packet reaches the buffering mechanism without having an `egress_spec` set. Vera catches this by adding an assertion that the `egress_spec` must be non-zero when it reaches the `buffer.in` port.
- **Table rules that match dropped packets** are flagged as errors by adding an assertion that *egress_spec* $\neq 511$ in the preamble of all actions.
- **Invalid memory accesses** are frequent P4 mistakes, when users do not test the validity of a header before using its fields. Vera relies on Symnet's memory safety guarantees to capture these errors; when accessing an unallocated field, Symnet will fail the current path.
- **Header errors** Malformed headers are captured during parsing by using the `exists` SEFL instruction. Adding an existing header or removing an inexistent one are also caught automatically as deparsing errors.
- **Scoping and unallowed writes** Certain metadata values are read-only in P4, yet the P4 compiler allows the program to write them (e.g. the `egress_port` metadata). Further, static registers can only be read from one table according to the spec, yet the compiler allows such reads. Vera catches such errors during translation.
- **Out-of-bounds array accesses** are caught automatically by Vera by adding, before each array access, an out-of-bounds check for the index. At runtime, the solver will check if the constraint is satisfiable and if it is the user will get a failed path providing an example packet that triggers a possible out-of-bounds access.
- **Field overflows/underflows** are the only arithmetic exceptions possible in P4 (because division is not supported) and Vera catches them by adding a check before each addition/subtraction operation.

Loops are also caught automatically. The loop detector runs by default on the parser input port and on the egress input port which are the two places where packets can be redirected backwards in the P4 pipeline. Whenever a packet enters one of these ports, Vera remembers the entire memory state (i.e. the values and constraints or all the metadata and header fields). When a packet revisits the same port, its memory state is compared to *all* the previous saved memory states. Two memory states are different iff at least one symbol has a different value in the two states. Note that we compare not only concrete values, but also symbolic ones: if a metadata is bound to the same symbolic value in both states, it is deemed to be equal. Whenever Vera discovers two memory states that are equal, it fails the current path with the "loop detected" message. We provide a proof that our loop detection will always detect loops on the main input port, as long as there is enough memory to remeber all previous memory states in [30].

We note that even if a P4 program does not have any of the bugs above, it may not do the job it is supposed to. For instance, a router that explicitly drops all packets it receives is bug free but it also doesn't do anything useful. It is therefore important to also be able to reason about the **correctness** of the P4 program. The correctness properties each box has to conform to depend on its intended use, and differ among different network boxes. In the next section we describe a specification language that enables specifying a wide-range of correctness properties for network boxes and explain how Vera automatically verifies whether these properties hold.

4.1 Correctness verification with NetCTL

For any given P4 program, Vera will explore a large number of paths, many of which are successful. In our evaluation, we typically see hundreds such paths. Examining them manually to decide whether the behavior is correct is time consuming and error-prone. We wish to specify desirable properties and have Vera check them automatically.

The specification must combine *packet constraints at specific ports* of the P4 switch (or *state properties*) with *constraints over the possible paths* which the packets may take between ports (or *path properties*). We can already express state properties via SEFL instructions. For instance, the property '*destination IP is always* `X` *at port* `out`' can be verified by placing the SEFL instruction `Dest-IP != X` at port `out` and observing that no successful paths from port `out` are possible.

In order to express path properties, we have considered a wide range of SDN policy languages, e.g. the Kinetic[18]

family, FatTire[28], NetPlumber[14], as well as approaches relying on logic programming (e.g. FML[13]). We have found that all such languages are limited in their ability to express compositional constraints.

We have thus turned to Computation Tree Logic (CTL) [6]. In CTL, temporal operators such as **F** (i.e. sometime in the future) and **G** (i.e. always in the future) are combined with path quantifiers: ∃ (on some path) and ∀ (on all paths). For instance, the policy: ∀**F**`destTCP == 80` evaluated at some port P of a box, expresses that on all possible packet paths from P, `destTCP` will eventually become 80.

The syntax of NetCTL is given below:

$$\varphi ::= SEFL \mid \neg\varphi \mid \varphi \wedge \varphi \mid XY\varphi$$

where where $X \in \{\exists, \forall\}, Y \in \{\mathbf{F}, \mathbf{G}\}$.

Unlike Merlin, FatTree or NetPlumber, in NetCTL we can construct more complex properties starting from simpler ones. For instance, we can express that ''*whenever the IP destination of a packet becomes a public address, port P is reachable*" via the formula:

$$\forall\mathbf{G}(\texttt{ip != 192.168.0.0/16} \rightarrow \exists\mathbf{F}\ \texttt{port == Internet})$$

CTL can express many other properties such as invariance across tunnels and TCP connectivity.

Checking NetCTL with Vera. To check that a property written in NetCTL holds for a given P4 program, it suffices to run Vera on that program—i.e. inject all accepted header layouts, making all fields symbolic, including table entries—and then check the property against all resulting symbolic execution paths. Since the reunion of all execution paths accurately describes the behaviour of the P4 program, NetCTL verification is guaranteed to provide an accurate answer.

However, this approach is quite inefficient because in many cases we can check a property without exploring all possible paths; for instance $\exists\varphi$ is satisfied as soon as the φ holds on one symbolic execution path, without requiring further exploration of the remaining paths. That is why Vera checks NetCTL properties during symbolic execution.

In our implementation, NetCTL verification is performed as added checks on the packets after each SEFL code block is executed; the overhead of these checks is very small in practice. After every check we can decide to prioritize a certain path or stop execution altogether. Because of this, in most cases, Vera checks NetCTL properties faster than exhaustive symbolic execution (see §5). We have proven that the resulting verifier is correct, i.e. a formula is reported true by the verifier iff it is true c.f. our NetCTL semantics [30].

In Fig 10, we briefly illustrate NetCTL verification. The figure describes two symbolic execution traces performed on the same topology — a simplistic illustration of the P4 NAT model, described in more detail in the subsequent sections. Boxes represent SEFL code blocks and solid lines — links

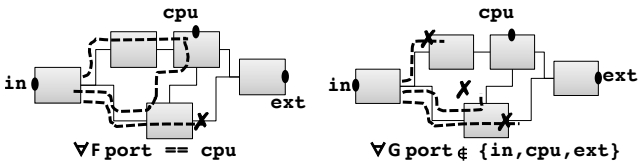

Figure 10: NetCTL example

between boxes. Dashed lines describe the paths explored by our verifier. The formula $\varphi_1 = \forall\mathbf{F}(\texttt{port == cpu})$ (left) expresses that all paths eventually reach port cpu. In order to evaluate it, our checker performs symbolic execution starting at port `in`. The checker will explore each encountered path until `port == cpu` is satisfied, the path ends, or it becomes unsatisfiable. Suppose the checker explores three paths, as shown in the figure. Since the formula **F**(`port == cpu`) is true on the first two paths only, φ_1 is false.

The formula $\varphi_2 = \forall\mathbf{G}(\texttt{port} \notin \{\texttt{in,cpu,ext}\})$ (right) expresses that all packets are dropped by the NAT. To verify it, our checker will determine if $\texttt{port} \notin \{\texttt{in,cpu,ext}\}$ is true on *each* execution path, and after each SEFL code-block. In our example, this is indeed the case, thus the policy is true.

5 EVALUATION

In our evaluation we seek to understand the coverage Vera provides and its scalability to large P4 programs (LOC) as well as large match-action tables. Tests where run on a server with a quad-core i5 processor and 8GB of RAM.

We discuss the bugs we found in a series of available P4 programs in §5.1. We then use NetCTL to express the correctness properties of a NAT and verify whether the simple NAT tutorial has these properties in §5.2. Finally, we examine how verification time scales with the number of rules in match-action tables in §5.3.

5.1 Bugs caught

We have used Vera to examine many P4 programs, but we do not claim our evaluation is exhaustive in any way. We ran tests in two ways: first, with command files supplied by the authors and using symbolic entries.

A summary of results is shown in Table 11 where the programs without a citation are from the official P4 codebase. We list the size of each P4 program, the time it takes Vera to exhaustively verify it, and the bugs we have found. Vera has found multiple bugs in each program we have examined. When using concrete table entries, the verification time ranges from 1s for all possible packets for toy programs (P4 tutorial) to around 15 seconds for `switch.p4` for one symbolic packet. With symbolic table entries, the runtime for all programs except the switch is under 10s; for the switch with fully symbolic table rules, Vera does not finish in three hours of running (details below).

Program	*Size (LOC)*	*Verification time (sec)*	*Implicit drop*	*Parsing*	*Deparsing*	*Header ops.*	*Invalid access*	*Underflow / overflow*	*Loop*	*Processing dropped packets*
copy-to-cpu	70	0.1		•		•				
resubmit	70	0.4							•	
encap	130	0.45	•		•	•				
simple router	145	0.55	•							
simple NAT	290	1.25	•			•				
simple router + ACL	200	0.8	•							•
Axon	100	14		•			•			
Switch	6000	5-15/sym.pkt.				•	•			
Beamer mux[27]	340	1.4	•		•		•			
NDP switch[12]	210	0.8					•			
P4xos[7]	650	13.4	•				•			

Figure 11: Bugs found by Vera in P4 programs available publicly.

The severity of the bugs we found differs: some bugs are critical and will impact heavily the operation of the program (for instance out-of-bounds accesses or deparsing errors) while others are more benign. For instance, implicit drops are present in many programs, but their severity is not as high so they can be considered "warnings".

We now discuss some of the more interesting bugs in more detail. The `copy-to-cpu` tutorial is meant to show how a packet can be forwarded to the controller; a 16 bit cpu_header is added to the packet and sent to the CPU. The program has a parser bug in the following piece of code:

```
return select(current(0, 64)) {
     0 : parse_cpu_header;
     default: parse_ethernet;
}
```

Vera found two bugs in this code. When the input packet contains a single ethernet header, if the first 64 bits are set to zero this the parser will try to extract the cpu_header which will fail. However, even when the packet contains a cpu_header the code is wrong: the header is only 16bits, so the check will also include the ethernet source address. When the latter is not zero (most often), the parser will assume this packet is pure ethernet and try to extract the ethernet header, which will fail. In other tutorials which parse the cpu_header, the cpu header definition includes a 64bit preamble which must be zero; updating the header definition would also fix the `copy-to-cpu` example.

Beamer [27] is a load balancer: it takes packets, encapsulates them with an IP-IP header, and sends them to a backend server. In Beamer, Vera has found a typical deparsing bug. The exact encapsulation depends on the TCP destination port in packets: if the port is less than 1024, the output packet layout is *eth,ip,ipopt,ip*, and this is deparsed correctly. If the port is larger than 1024, the output packet does not include the *ipopt* header and this leads to a deparsing error because this packet is not correctly parsed by Beamer. When we shared our findings with the Beamer authors, they mentioned that their prototype also had a deparsing error on the first branch which they caught after some effort, but they had missed this bug that was not tested by their unit tests.

P4xos [7] is a P4 implementation of the Paxos protocol. Vera has found an out-of-bounds static register access in action `read_round` in the following instruction:

```
register_read(local_meta.round, rounds_register, paxos.inst);
```

The problem is that the `inst` header field can take any value, leading to faulty accesses (a *todo* in the code acks this issue).

Switch is the largest P4 program available today; it implements the full stack of protocols needed to operate a datacenter top-of-rack switch and is thus a good benchmark for our verification tool. Verification of a `switch.p4` snapshot takes between 5 to 15 seconds *per packet type*, when all packets fields are made symbolic, but the table rules are concrete. As there are 60.000 possible header layouts given by the parser, total verification for all header types for one snapshot of this P4 program would take 170 hours (a week) on a single machine, but this can be easily parallelized.

We have analyzed, however, only tens of packet headers because after each run we need to manually check the outputs (two-three hundred paths, typically), sift through the failed paths, and decide which ones are novel bugs and which represent bugs we already know about. This process is very time consuming. Exhaustive verification of one snapshot is therefore feasible computationally, but we need to design more tools to automatically interpret outputs; this is our future work.

Overall, the switch code is much cleaner than all the other examples we have looked at, reflecting the fact that this is production quality software. Below we discuss three of the more interesting bugs we found in the switch. The first bug is in the `remove_vlan_double_tagged` action which is triggered in the `vlan_decap` table:

```
remove_header(vlan_tag_[0]);
remove_header(vlan_tag_[1]);
```

The code above first removes the header at position 0 in the array. This makes all headers at higher indices shift their position to the left; in other words, `vlan_tag_[1]` becomes `vlan_tag_[0]`, and the second remove instruction fails silently. After the action we are left with one active header, instead of having both removed. Depending on

packet processing downstream, this bug will result in outgoing packets having a vlan tag when they shouldn't have.

Another bug is accessing an invalid field in table `l3_rewrite` where both the ipv4 and ipv6 source addresses are matched, and they can't be valid simultaneously.

A third bug appears when the data-plane is configured to allow layer 3 VXLAN encapsulation/decapsulation. The switch correctly behaves when input an `Ethernet/IP/TCP` - i.e. it VXLAN encapsulates the frame. Now, assume a VXLAN encapsulated frame is input from the non-tunnel interface of the switch. The expectation is that frames be further encapsulated within a new VXLAN header, while keeping the existing one intact. However, the actual implementation of the reference switch has a single VXLAN header and the switch attempts to add a header which is already valid. Vera quickly discovers the offending operation, while providing important insights into the error location within the P4 program - i.e. match-action table trace, offending table name and match conditions.

Switch.p4 with symbolic table rules. We injected a fully symbolic TCP packet into switch.p4, while inserting one symbolic rule for each action in each tables of switch.p4 (163 tables in total). After some changes (fixing a few bugs and dumping completed paths on disk to save RAM), Vera using the default DFS exploration strategy ran for 67 minutes on one of our testbed machines (16GB of RAM, 100GB of HDD) until it completely filled the disk with completed states. In total, Vera explored 900K paths, of which 85K where succesful and the other ones failed.

To analyze the failures, we simply groupped them by port and error message; we were surprised to find that only 32 distinct errors where captured by these 800K+ paths. We also found that overall coverage (port-wise) was quite low. We manually examined the found errors, and found they were trivial: header fields being accessed when not present, because of wrong entries in table rules. For instance, one failure was reported in the *validate_outer_header* table when accessing vlan_tag[x]. When the vlan tags are not valid, any action that attempts to use the tag will fail. While possible, this error will only appear with a faulty controller, and can be considered a controller bug.

To increase the coverage of our exploration, we also experimented with breadth-first exploration of paths. Vera achieved around 30% coverage in around 7 hours of running on a machine in AWS with 256GB of RAM; in total it reported 7 million failed states, all instances of one of 20 (trivial) bugs.

Increasing coverage with fully symbolic table entries is an interesting avenue of further development, which will require adopting techniques to increase coverage from traditional symbolic execution. Another direction of research is automating the verification of these bugs beyond grouping.

5.2 Correctness verification of simple NAT

Vigor is a provably correct NAT implementation in C [32] that required an intense verification effort from networking researchers. We want to check if the simple NAT implementation from P4 offers similar correctness guarantees. The first step is to express the properties a NAT should follow. All our verification rules specify (i) a port where the symbolic packet will be injected; (ii) init code: the header layout and other instrumentation to be performed the initial *symbolic* packet and (iii) the appropriate policy in the NetCTL language. The NAT defines an interior (`in`) and exterior port (`ex`), as well as a port for packets sent to the controller (`cpu`). We also use a meta-variable `port` which stores the current port during symbolic execution. The NAT table has *hit* and *miss* actions for the in/ext ports which handle the situation when mappings exist or don't exist for the current packet.

We list in Fig. 12 a subset of the policies we have specified to describe correct NAT behavior , in the form '`port` : φ', where `port` is the input port, and φ is a NetCTL formula. We omit describing the init code in most cases, as it is less important for understanding our methodology.

We acknowledge that specification in NetCTL is not an easy undertaking, especially for programmers not familiar with CTL. Specifying the required properties and verifying them using Vera required around one day of work for one person. The NAT, however, is a fairly simple program and specifying larger programs will be much more difficult. This is an area we intend to pursue in the future.

The first policy requires that the NAT drops all packets if there are no table entries. The policy states: *on all execution paths, at* no point *does the packet reach any of the NAT output ports.* Vera confirms that this policy holds in 2.2s.

We have checked policies (2-5) with and without symbolic-table entries, but present only results corresponding to symbolic entries because they have greater coverage and give insights into the correct behavior of the controller.

The second policy expresses that all packets from the input ports that do not match hit rules will reach the controller. To check this policy we add symbolic entries for the *miss* actions and Vera confirms the policy is true in around 1s.

The third policy verifies that the NAT translates packets before sending them to the output interface. To verify it, we insert symbolic entries for the hit actions and call Vera. Vera finds an example where this policy is violated: after a hit action is executed, the destination IP field is unconstrained and it can reach both `in` and `ext` during the routing phase. Concretely, this means that the NAT will also translate packets destined for the LAN, which is not intended.

In order to check policies (4-5), we have built a simple TCP responder in P4 which flips the IP and TCP source/destination

	Policy	NetCTL formula and input port	Verification time and explored paths
(1)	*A NAT without entries drops all packets*	p : ∀**G**(port ∉ {in,ex,cpu}) where $p \in$ {in,cpu}	708 paths in 2234 ms
(2)	*If only miss entries exist, in-packets reach the controller and ext-packets are dropped*	in: ∀**F**(port == cpu) ex: ∀**G**(port ∉ {in,ex,cpu})	354 paths in 1034 ms 285 paths in 1157ms
(3)	*With hit entries, matching in- and cpu-packets reach ext*	p : ∀**F**(port == ex) where $p \in$ {in,cpu}	232 paths in 816 ms
(4)	*A response to a in-packet reaches* in	in: ∃**F** (port==in)	113 paths in 1413 ms
(5)	*The NAT performs a correct IP mapping*	in: ∀**G** (port==in)⟶ (ini_dstIP == srcIP)	304 paths in 1913 ms

Figure 12: Verification of a P4 NAT

addresses. Policy (4) checks that the NAT enables for bidirectional connectivity. We start with symbolic entries for the hit action and Vera verifies that bidirectional connectivity exists, and the successful path shows how the table rules must look like: a `hit-int-to-ext` rule must exist which matches the packet's 5-tuple and has the `is-ext-if` fields set to false, and a `hit-ext-to-int` rule must exist which matches the translated packet's reverse 5-tuple, with the `is-ext-if` field set to true. Further, this rule also restricts to possible action parameters for the `hit-int-to-ext` rule: the srcIP must be the IP of the router's external interface.

Policy (5) further verifies that the translation works correctly. In order to verify it, in the init code we create a new variable `ini_dstIP` which stores the initial destination IP header field. Thus (5) expresses that whenever the packet reaches `in`, the current source IP field must be equal to the initial destination IP. The actual policy we verified checks the entire 5-tuple, not just the destination IP.

Taken together, our verification gives a clear controller spec. When a new connection arrives from the LAN, it will be sent to the controller (cf. policy 2) which *must* insert two hit rules in the NAT table (cf. policy 4) to enable translation; the contents of the rules are completely specified (except the source port). Finally, policy (3) specifies that, if we want the initial packet to be translated too, the controller *must* inject it *after* it inserts the `hit-int-to-ext` rule. Using these rules to develop a correct controller is our future work.

Vera partially explores the NAT model for policies 4-5. Verification stops as soon as a successful path is found, and this significantly reduces the number of execution paths (100-200). In contrast, unconstrained symbolic execution explores more than 2000 paths— twenty-times more—in 3.6s.

5.3 Scalability of match-action processing

All our examples so far were run with a few concrete or symbolic table entries. Here we want to examine the performance of our match-action algorithm at scale, focusing on table with many concrete rules.

We used the Stanford dataset [15] containing router FIBs of 180K entries and compared the Naive If/Else implementation with Vera and a SEFL model hand-optimized for routing from prior work [31]. We show the results in Table 13, highlighting the model generation time and the symbolic execution time of the resulting P4 program. The naive model chokes at just 10K entries in the FIB, while Vera and the hand crafted model give good performance even for large routing tables containing 180K entries. Compared to [31], Vera is slightly faster in building the model, and is faster during model updates: it takes just 2.5ms to update a model with 100K entries, whereas the hand-crafted model takes around 100ms.

Achieving good performance when analyzing longest-prefix match is possible because the number of actual paths explored is (roughly) bounded by the number of interfaces of the router, so these results are not necessarily surprising.

To test more complex match-action processing, we implemented and populated an ACL table that matches on ethernet and IP addresses, TCP ports, input interface and protocol type, and inserted this table before lpm in the simple router P4 tutorial. We then populated the ACL table using rules from Classbench-ng[24], a tool that generates realistic Openflow rules. The time required to build the match-forest data structure is under 100ms in all cases.

Fig. 14 shows how verification scales with the number of rules inserted in the ACL table; note that the maximum number of rules Classbench-ng can create is around 2.2K. We compare the fork approach, where we create one path for each possible rule, against the naive if-else implementation; the results show that if/else does not scale because it creates one branch for each field of each rule, resulting in an exponential number of paths.

For 2.2K rules, Vera exhaustively explores all paths in around 1 minute. This is a worst case because the experiment assumes each rule has a different action/parameter combination. In practice, most rules choose between a small number of actions (e.g. allow or drop); in this case Vera will create a path per group of rules instead, increasing scalability. In Figure 15 we vary the number of rules which can be groupped, measuring the verification time (with a symbolic packet) and the average time needed to push a concrete packet through the same SEFL-P4 program. The results show that grouping rules helps massively, with large 100-rule groups cutting the runtime to around 1 second. The concrete execution results

Algorithm	Action	1K	10K	100K	180K
Naive	model	2ms	5ms		
	symbex	34s	1 hour		
Hand-crafted[31]	model	23ms	1.5s	218s	242s
	symbex	220ms	2.7s	12s	16s
Vera	model	36ms	1.5s	83s	178s
	symbex	230ms	2.7s	12s	16s

Figure 13: Match-action performance for IP forwarding.

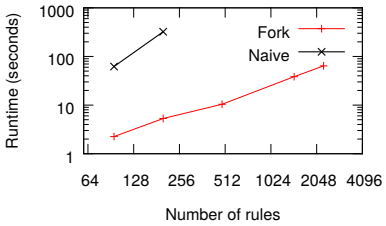

Figure 14: (verification time for Openflow-like rules.

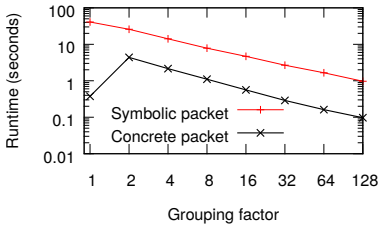

Figure 15: Effect of grouping on verification time.

are shown to contrast the cost of matching symbolic headers against that of concrete values and as expected the runtime is much smaller. The anomaly at grouping factor 1 happens because Symnet special-cases equality, and avoids calling Z3; when we have groups of rules, this optimization is no longer applied and Z3 is always called.

6 RELATED WORK

Network verification research mostly focuses on understanding whether network-wide properties such as reachability and isolation hold. Dataplane analysis tools such as HSA[15], NOD[22], Veriflow [17], Anteater[23] and Symnet[31] require a snapshot of the network dataplane, including the processing done by each box, links between boxes and forwarding rules, and test whether the desired end-to-end properties hold. Control plane verification aims to answer the same network-wide questions but without requiring the dataplane snapshot: tools like Arc[11], Batfish[9], Minesweeper[1] or CrystalNet [20] can predict how a control plane change will affect the dataplane, flagging property violations when they occur. Vera is complementary to this work: it generates SEFL models of P4 programs and these can be used in network-wide dataplane analysis with Symnet [31].

Verifying if the implementation of a networking element is correct is another area of research. Dobrescu et al. [8] use symbolic execution with S2e [5] to analyze the C++ implementation of Click modular router elements [19]. Vigor [32] is a formally verified NAT implementation written in C. Vera is complementary to these works: it can guarantee safety of P4 code, and in conjunction with NetCTL it can also prove the correctness of an implementation according to some specification (see our NAT example).

Vera is not the first verification effort geared at P4 programs. Lopez et al. [25] translate P4 to the NOD language [22] and then perform end to end dataplane reachability tests, as well as testing a form of header errors they call "well-formedness". NOD does not offer support for dynamic encapsulation, so P4NOD cannot capture many of the header errors that Vera can. Lopez et al. only examine two small programs, and do not find problems in the programs themselves, focusing on end-to-end verification instead.

p4v is concurent work by Foster et al.[21]; p4v proposes that programmers annotate P4 programs with Hoare logic clauses (pre and post conditions) to enable static verification. Their approach targets catching many of the bugs that Vera catches automatically without any specification, with the caveat that Vera cannot currently fully explore all table snapshots in switch.p4. p4v requires human specification for the possible table entries thus easing the work of the verifier, and it scales well. On the downside, human specification is both cumbersome and error prone.

p4pktgen[26] also uses symbolic execution to generate test packets and predict the expected processing. It then inserts these packets into the bmv2 software switch, and sees if the behaviour is the expected one. Using this approach they uncover a number of bugs in the tools (both the compiler and the software switch). Such toolstack verification is complementary to the one we take in Vera.

ASSERT-P4 [10] is another work that was developed concurrently to Vera. ASSERT-P4 requires programmers to add assertions to P4 programs, and then translates both program and assertions to C and checks them via symbolic execution. While also relying on symbolic execution, Vera does not require annotations to find many types of bugs, easing the job of the programmer.

Rosu et al [16] have created an executable formal semantics in K for P4 that allows verification with a number of tools including symbolic execution.

7 CONCLUSIONS

P4 promises to enable truly flexible networks that can adapt to application needs, but P4 programming is not as easy as it may seem at first sight due to language features stemming from its close relationship to switch hardware.

In this paper we have implemented Vera, a tool that uses symbolic execution to exhaustively verify snapshots of large P4 programs. Vera relies on a number of innovations including automatic policy verification and a novel match-action data structure, and can explore a multitude of table snapshots via symbolic rules. Together, these have helped Vera to catch many interesting bugs in all programs we have analyzed, with modest runtimes. Full exploration of large P4 programs with symbolic rules is still not possible, and more work is needed to increase coverage.

ACKNOWLEDGEMENTS

This work was jointly funded by CORNET H2020, a research grant of European Research Council (no. 758815) and by SUPERFLUIDITY H2020 (no. 671566).

REFERENCES

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

[2] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. "P4: Programming Protocol-independent Packet Processors". In: *SIGCOMM Comput. Commun. Rev.* 44.3 (July 2014).

[3] Cristian Cadar, Daniel Dunbar, and Dawson Engler. "KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs". In: *Proc. OSDI'08*.

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

[5] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. "S2E: A Platform for In-vivo Multi-path Analysis of Software Systems". In: *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XVI. Newport Beach, California, USA: ACM, 2011, pp. 265–278. ISBN: 978-1-4503-0266-1. DOI: 10.1145/1950365.1950396. URL: http://doi.acm.org/10.1145/1950365.1950396.

[6] Edmund M. Clarke Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. Cambridge, MA, USA: MIT Press, 1999. ISBN: 0-262-03270-8.

[7] Huynh Tu Dang, Marco Canini, Fernando Pedone, and Robert Soule. "Paxos Made Switch-y". In: *SIGCOMM Comput. Commun. Rev.* 46.2 (May 2016).

[8] Mihai Dobrescu and Katerina Argyraki. "Software Dataplane Verification". In: *Proc. NSDI'14*. NSDI'14.

[9] 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.

[10] Lucas Freire, Miguel Neves, Lucas Leal, Kirill Levchenko, Alberto Schaeffer-Filho, and Marinho Barcellos. "Uncovering Bugs in P4 Programs with Assertion-based Verification". In: *Proceedings of the Symposium on SDN Research*. SOSR '18. Los Angeles, CA, USA: ACM, 2018, 4:1–4:7. ISBN: 978-1-4503-5664-0. DOI: 10.1145/3185467.3185499. URL: http://doi.acm.org/10.1145/3185467.3185499.

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

[12] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichi, and Marcin Wójcik. "Re-architecting Datacenter Networks and Stacks for Low Latency and High Performance". In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. SIGCOMM '17.

[13] Timothy L. Hinrichs, Natasha S. Gude, Martin Casado, John C. Mitchell, and Scott Shenker. "Practical Declarative Network Management". In: *Proceedings of the 1st ACM Workshop on Research on Enterprise Networking*. WREN '09. Barcelona, Spain: ACM, 2009, pp. 1–10. ISBN: 978-1-60558-443-0. DOI: 10.1145/1592681.1592683. URL: http://doi.acm.org/10.1145/1592681.1592683.

[14] Peyman Kazemian, Michael Chang, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. "Real Time Network Policy Checking Using Header Space Analysis". In: *Proc. NSDI'13*.

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

[16] Ali Kheradmand and Grigore Rosu. *Executable Formal Semantics of P4 and Applications*. 2017.

[17] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. "VeriFlow: Verifying Network-wide Invariants in Real Time". In: *Proc. NSDI'13*.

[18] Hyojoon Kim, Joshua Reich, Arpit Gupta, Muhammad Shahbaz, Nick Feamster, and Russ Clark. "Kinetic: Verifiable Dynamic Network Control". In: *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*. NSDI'15. Oakland, CA: USENIX Association, 2015, pp. 59–72. ISBN: 978-1-931971-218. URL: http://dl.acm.org/citation.cfm?id=2789770.2789775.

[19] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. "The click modular router". In: *ACM Trans. Comput. Syst.* 18.3 (Aug. 2000), pp. 263–297. ISSN: 0734-2071. DOI: 10.1145/354871.354874. URL: http://doi.acm.org/10.1145/354871.354874.

[20] 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)*.

[21] 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*.

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

[23] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P. Brighten Godfrey, and Samuel Talmadge King. "Debugging the data plane with anteater". In: *Sigcomm*. 2011.

[24] Jiří Matoušek, Gianni Antichi, Adam Lučanský, Andrew W. Moore, and Jan Kořenek. "ClassBench-ng: Recasting ClassBench After a Decade of Network Evolution". In: *Proceedings of the Symposium on Architectures for Networking and Communications Systems*. ANCS '17. Beijing, China: IEEE Press, 2017, pp. 204–216. ISBN: 978-1-5090-6386-4. DOI: 10.1109/ANCS.2017.33. URL: https://doi.org/10.1109/ANCS.2017.33.

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

[26] 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.

[27] Vladimir Olteanu, Alexandru Agache, Andrei Voinescu, and Costin Raiciu. "Stateless Datacenter Load-balancing with Beamer". In: *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. Renton, WA: USENIX Association, 2018. URL: https://www.usenix.org/conference/nsdi18/presentation/olteanu.

[28] Mark Reitblatt, Marco Canini, Arjun Guha, and Nate Foster. "FatTire: Declarative Fault Tolerance for Software-defined Networks". In: *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*. HotSDN '13. Hong Kong, China: ACM, 2013, pp. 109–114. ISBN: 978-1-4503-2178-5. DOI: 10.1145/2491185.2491187. URL: http://doi.acm.org/10.1145/2491185.2491187.

[29] Leonid Ryzhyk, Nikolaj Bjørner, Marco Canini, Jean-Baptiste Jeannin, Cole Schlesinger, Douglas B. Terry, and George Varghese. "Correct by Construction Networks Using Stepwise Refinement". In: *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, 2017, pp. 683–698. ISBN: 978-1-931971-37-9. URL: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/ryzhyk.

[30] Radu Stoenescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. *Debugging P4 Programs with Vera*. Tech. rep. June 2018. URL: http://nets.cs.pub.ro/~costin/files/vera-tr.pdf.

[31] 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.

[32] Arseniy Zaostrovnykh, Solal Pirelli, Luis Pedrosa, Katerina Argyraki, and George Candea. "A Formally Verified NAT". In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. SIGCOMM '17. Los Angeles, CA, USA: ACM, 2017, pp. 141–154. ISBN: 978-1-4503-4653-5. DOI: 10.1145/3098822.3098833. URL: http://doi.acm.org/10.1145/3098822.3098833.