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 exhaustively verifies P4 program snapshots 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.

This technical report is complementary to the Sigcomm 2018 paper titled "Debugging P4 Programs with Vera" that describes the main components of Vera and presents its evaluation. The present technical report must be read together with the conference paper: it is not a self-contained document as it does not describe the core functionality of Vera.

This technical report describes different aspects of Vera that were not fully captured in the conference paper. These aspects include the operational semantics we have developed for SEFL and P4, a formal description of NetCTL and the guarantees it offers, a discussion of symbolic match-action rules and the coverage they provide. The report also includes proofs that Vera's loop-detection algorithm is correct and that the match-action forest data-structure enables cheap symbolic execution.

1 Introduction

Programmable network dataplanes such as those enabled by P4 [1] 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.

We argue that dataplane programs should be verified before deployment to enable safe operation. In the Sigcomm 2018 paper called "Debugging P4

Programs with Vera", we have presented 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 [11] 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 P4 program snapshot: 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. 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 find such problems it guarantees the P4 program is bug-free.

This technical report completes the conference publication by detailing the theoretical foundations of Vera. In §2, 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 P4 instruction(s) and their SEFL translation are equivalent.

Upon deployment, P4 programs are incomplete (they lack table entries). Vera can analyze such P4 programs by using symbolic table entries instead of requiring concrete table entries. In §3 we show that one symbolic rule per possible action in each table is enough to explore all possible dataplanes, as long as there are no loops.

At runtime, controllers will insert rules in the P4 program, and these must be checked against the policy of the network (which involves network-wide verification). Here, the time available for verification is constrained. Vera introduces a match forest data structure that concurrently optimizes both update time and verification time. We describe it in more detail and show it yields the minimum number of constraints in §4.

Additionally, Vera can check user-provided functional properties and thus prove the box conforms to a user-specification. In §6 we provide a detailed presentation of NetCTL, a novel specification language that allow user-specified policies. NetCTL uses the specification to drive symbolic execution, ensuring the checked properties always hold while reducing exploration time.

2 Operational semantics for SEFL and P4

The following section presents the "big-step" operational semantics for some relevant statements of P4 and SEFL as well as the proof of semantics preservation through the translation process. 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 state includes header variables and metadata. The definition of $\rightarrow$ is given by the semantics rules, written as inferences where a horizontal line separates premises (above) from

the conclusion (below). The meaning of statements is summarized as a function $\mathcal{S} : Stm \to (State \to State)$, specified as $\mathcal{S}_P$ for P4, and $\mathcal{S}_S$ for SEFL.

The rules for most of the statements either rewrite the state by updating a header field (variable) or invoke a control state. In order to deal with the transformations in terms of variables we represent the states with two mappings: a variable environment that associates a location with a variable and a store that associates a value with a location. The variable environment env_V is an element of $Env_v = Var \to Loc$, where Loc is the set of locations. A store sto is an element of $Store = Loc \to \mathbb{N}$. Under these assumptions a state is defined as $s = sto \circ env_V$ [4].

2.1 Operational semantics for SEFL

The SEFL statements can create header fields and metadata ($Allocate, Deallocate$), assign the result of an evaluated expression to a variable ($Assign$), manipulate tags to simplify access to header fields ($CreateTag, DestroyTag$). The control statements are $Fork$ and If. SEFL includes two statements that constrain the execution of the current path, $Fail$ and $Constrain$. For a full specification of the SEFL language, please see [11].

The statements *Allocate* and *Deallocate* update the variables environment and store.

[alloc] $\langle Allocate(v), env_V, sto\rangle \to (env_V[v \mapsto l], sto[l \mapsto \varepsilon])$

where l is the next available location.

[dealloc]
$\langle Deallocate(v), env_V, sto\rangle \to (env_V[v \to \varepsilon], sto[l \to \varepsilon])$
where $l = env_V v$

We used the notation $env_V[x \mapsto y]$ to denote that the environment $env_V[x \mapsto y]$ is the same as env_V except for the association x to y. Similar for the store. The symbol ε is the semantic equivalent of null.

The statements *CreateTag* and *DestroyTag* update only the variables environment, a tag being an alias for a location. $\mathcal{A}$, defined as a total function, denotes the meaning of arithmetic expressions, $\mathcal{A} : Aexp \to (State \to \mathbb{Z})$.

[create]
$\langle CreateTag(t, e), env_V, sto\rangle \to (env_V[v \mapsto l], sto)$
where $l = \mathcal{A}[e](sto \circ env_V)$

[destroy]
$\langle DestroyTag(t), env_V, sto\rangle \to (env_V[v \mapsto \varepsilon], sto)$

The semantics of the next statements is related to the variables environment, therefore we use the notation $env_V \vdash \langle S, s\rangle \to s'$ in order to emphasize the presence of the environment.

The statement *Assign(v,a)*, assigns the result of the evaluation of the expression a to the variable v.

$$[\text{assign}^{\text{t}}] \quad env_V \vdash \langle Assign(v,a), sto\rangle \to sto[l \mapsto a] \text{ where } l = env_V v \text{ and } a = \mathcal{A}[a](sto \circ env_V)$$

If the variable is not allocated then an error will be issued.

$$[\text{assign}^{\text{f}}] \quad env_V \vdash \langle Assign(v,a), sto\rangle \to \langle Errl(msg), sto\rangle \text{ where } l = env_V v \text{ and } l = \varepsilon$$

The next semantic rules describe the *sequential composition* and *If* statements. The values of boolean expressions are true values. Their meaning is defined by the total function $\mathcal{B} : Bexp \to (State \to T)$.

$$[\text{seq}] \quad \frac{env_V \vdash \langle S_1, sto\rangle \to sto' \quad env_V \vdash \langle S_2, sto'\rangle \to sto''}{env_V \vdash \langle S_1; S_2, sto\rangle \to sto''}$$

$$[\text{if}^{\text{t}}] \quad \frac{env_V \vdash \langle S_1, sto\rangle \to sto'}{env_V \vdash \langle \text{if } b \text{ then } S_1 \text{ else } S_2, sto\rangle \to sto' \quad if \mathcal{B}[b](sto \circ env_V) = \text{t}}$$

$$[\text{if}^{\text{f}}] \quad \frac{env_V \vdash \langle S_2, sto\rangle \to sto'}{env_V \vdash \langle \text{if } b \text{ then } S_1 \text{ else } S_2, sto\rangle \to sto' \quad if \mathcal{B}[b](sto \circ env_V) = \text{f}}$$

The *Fork* instruction can be expressed as a parallel execution with packet duplicates.

The *Constrain* statement ensures that the variable satisfies the specified constrain. The execution path fails if it doesn't.

$$[\text{constr}^{\text{t}}] \quad env_V \vdash \langle Constrain(v,c), sto\rangle \to sto \text{ if } \mathcal{B}[c](sto \circ env_V) = \text{t}$$

$$[\text{constr}^{\text{f}}] \quad env_V \vdash \langle Constrain(v,c), sto\rangle \to fail \quad if \mathcal{B}[c](sto \circ env_V) = \text{f}$$

$$[\text{fail}] \quad env_V \vdash \langle Fail(msg), sto\rangle \to fail$$

where *fail* is the fail state.

2.2 Operational semantics for P4

The P4 statements follow the behavior already introduced. They rewrite the state by updating a field or invoke a control state in the parser, table or action component.

The $modify\_field$ statement sets the field using a passed parameter or a field in the metadata. In our case the environment keeps the variables (including local) as well as the metadata and their values, therefore the semantics evaluates the expression in the environment.

[mod_field$^{\mathrm{t}}$]
$$env_V, env_A \vdash \langle modify\_field(x, a), sto\rangle \to sto[l \mapsto v]$$
where $l = env_V x$ and $v = \mathcal{A}[a](sto \circ env_V)$

If the parent header instance of the field is not valid the action has no effect. The metadata is always valid.

[mod_field$^{\mathrm{f}}$]
$$env_V, env_A \vdash \langle modify\_field(x, a), sto\rangle \to sto$$
where $l = env_V x$ and $l = \varepsilon$

In order to specify the semantics of actions we introduce the action environment env_A, which is an element of $Env_A = Aname \hookrightarrow Stm \times Env_V \times Env_A$. The action environment allows us to associate action names with their body as well as the action environment and variable environment at the point of the declaration. The transitions will have the form $env_V, env_A \vdash \langle S, sto\rangle \to sto'$.

[act] $$\frac{env'_V, env'_A \vdash \langle S, sto\rangle \to sto'}{env_V, env_A \vdash \langle action\ a, sto\rangle \to sto'} \quad \text{where } env_A\ a = (S, env'_A, env'_V)$$

Actions that modify fields can be combined in parallel or sequentially. The semantic rules follow:

[seq] $$\frac{env_V, env_A \vdash \langle S_1, sto\rangle \to sto' \quad env_V, env_A \vdash \langle S_2, sto\rangle \to sto''}{env_V, env_A \vdash \langle S_1; S_2, sto\rangle \to sto''}$$

[par] $$\frac{env_V, env_A \vdash \langle S_1, sto\rangle \to sto', \langle S_2, sto'\rangle \to sto''}{env_V, env_A \vdash \langle S_1 \text{ par } S_2, sto\rangle \to sto''}$$

$$\frac{env_V, env_A \vdash \langle S_2, sto\rangle \to sto', \langle S_1, sto'\rangle \to s''}{env_V, env_A \vdash \langle S_1 \text{ par } S_2, sto\rangle \to sto''}$$

The $apply(table)$ statement applies the actions in the initial state followed by the control statements in the modified state.

$$[\text{apply}] \quad \frac{\begin{array}{c} env_V, env_A \vdash \langle S, sto\rangle \to sto' \\ env'_V, env'_A \vdash \langle S', sto'\rangle \to sto'' \end{array}}{\begin{array}{l} env_V, env_A \vdash \langle apply(table), sto\rangle \to sto'' \\ \quad \text{where } table = (r, a) \\ \quad \text{and } env_A\, a = (S, env'_V, env'_A) \\ \quad \text{and } S' \in \{apply, noop\} \end{array}}$$

$$[\text{if}^{\text{t}}] \quad \frac{env_V, env_A \vdash \langle S_1, sto\rangle \to sto'}{\begin{array}{c} env_V, env_A \vdash \langle \text{if } b \text{ then } S_1 \text{ else } S_2, sto\rangle \to sto' \\ \text{if } \mathcal{B}[b](sto \circ env_V) = \text{t} \end{array}}$$

$$[\text{if}^{\text{f}}] \quad \frac{env_V, env_A \vdash \langle S_2, sto\rangle \to sto'}{\begin{array}{c} env_V, env_A \vdash \langle \text{if } b \text{ then } S_1 \text{ else } S_2, sto\rangle \to sto' \\ \text{if } \mathcal{B}[b](sto \circ env_V) = \text{f} \end{array}}$$

2.3 Correctness of the translation

We introduced the semantics of the SEFL and P4 languages in order to prove that the translation process ensures the semantics preservation. The correctness of the translation amounts to show that if we translate a P4 statement into SEFL code and execute that code, then we obtain the same result as specified by the operational semantics of P4.

The translation of P4 statements into SEFL code is given by the function $\mathcal{TS} : Stm_P \to Stm_S$.

We give the translation rules for non trivial P4 statements:

$\mathcal{TS}[modify\_filed(x, a)] = Assign(x, a)$

$\mathcal{TS}[apply(table \equiv (S, S'))] = \mathcal{TS}[S]; \mathcal{TS}[S']$

$\mathcal{TS}[S_1; S_2] = \mathcal{TS}[S_1]; \mathcal{TS}[S_2]$

Theorem

For every statement S of P4, $\mathcal{S}_P[S] = \mathcal{S}_S[S]$.

The theorem relates the behavior of statements in P4 and SEFL under the "big-step" operational semantics. It expresses the property that if the execution of statement S from some state terminates in one of the semantics it also terminates in the other and the resulting states will be equal. In order to prove the theorem we must first prove the following Lemma.

Lemma

For every statement S of P4 and states s and s', if $\langle S, s\rangle \to s'$ then $\langle \mathcal{TS}[S], s\rangle \to s'$.

Proof: the proof is by induction on the shape of the derivation tree for $\langle S, s\rangle \to s'$.

The case [mod_field$^{\text{t}}$]: we assume that

$\langle modify\_field(x, a), s\rangle \to s'$, where $s' = s[x \mapsto \mathcal{A}[a]s]$.

From the translation rules we have

$\mathcal{TS}[modify\_field(x, a)] = Assign(x, a)$

According to the *Assign* semantic rule

$\langle Assign(x, a), s\rangle \rightarrow s'$ where $s' = s[x \mapsto \mathcal{A}[a]s]$.

Therefore the resulting states are equal.

The case [mod_field$^{\text{f}}$]: we assume that

$\langle modify\_field(x, a), s\rangle \rightarrow s$, meaning that the field is not allocated (parent header instance is invalid)

From the translation rules we have

$\mathcal{TS}[modify\_field(x, a)] = Assign(x, a)$

According to the *Assign* semantic rule for the situation in which the variable is not allocated:

$\langle Assign(x, a), s\rangle \rightarrow s$.

Therefore the resulting states are equal.

The case [apply]: we assume that

$\langle apply(table), s\rangle \rightarrow s''$ holds, where $table = (r, a)$, because $\langle S; S', s\rangle \rightarrow s''$ holds, where $env_A[a \mapsto S]$ and $S' \in \{apply, noop\}$.

From the translation rules we have that $\mathcal{TS}[S; S'] = \mathcal{TS}[S]; \mathcal{TS}[S']$.

The case [seq]: we assume that

$\langle S_1; S_2, s\rangle \rightarrow s''$ holds because

$\langle S_1, s\rangle \rightarrow s'$ and $\langle S_2, s'\rangle \rightarrow s''$.

From the translation rules we have $\mathcal{TS}[S_1; S_2] = \mathcal{TS}[S_1]; \mathcal{TS}[S_2]$. By applying the induction hypotesis on the premises, we have $\langle \mathcal{TS}[S_1], s\rangle \rightarrow s'$ and $\langle \mathcal{TS}[S_2], s'\rangle \rightarrow s''$.

This proves the Lemma.

3 Symbolic match-action rules and coverage

Vera can quickly explore snapshots of P4 programs where a snapshot includes the P4 code and the concrete match-action rules for all the tables in the program. Unfortunately, though, verifying a single (or a few) snapshot to be bug free does not mean the P4 program behaves correctly for other table rules.

Vera allows users to insert symbolic match-action rules where both the matching fields and the parameters for the actions can be unconstrained symbolic variables. Intuitively, by adding enough symbolic rules, it should be possible to explore all possible dataplanes, thus achieving the goal of exhaustively verifying a P4 program, without verifying the controller program itself (a very difficult task since the controller is a general-purpose program).

What is the minimum number of rules that must be inserted to achieve exhaustive verification? We answer this question in this section.

To guide the exposition we use the running example in figure 1 that shows a P4 match-action table. with two actions: rewrite the IPv4 source field to a value specified as an action parameter, or drop the packet. This table matches *exactly* on the IPv4 destination field. The table has two entries: i. one matching

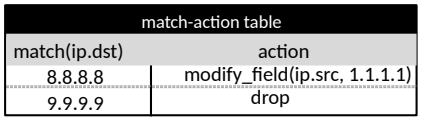

Figure 1: Concrete match-action table

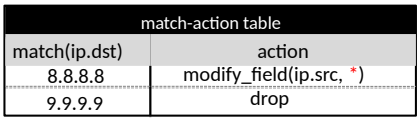

Figure 2: Symbolic action parameter

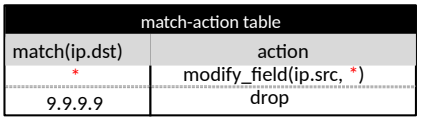

Figure 3: Fully symbolic table entry

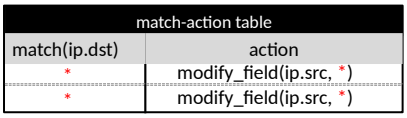

Figure 4: Duplicated symbolic entry

the address '8.8.8.8' in which case it rewrites the IPv4 source field to '1.1.1.1'
ii. another matching the address '9.9.9.9' in which case the traffic is dropped.

Based on this concrete example we state the following:

1. *A single symbolic table entry per action can model the behavior of the whole set of concrete entries that employ the same action.*

   In our example (figure 2), if we swap the concrete value of the parameter for 'modify_field' action for a symbolic one and then perform symbolic execution we will get back a set of symbolic execution paths. Each resulting execution path will state a different branching condition on the rewritten source address field, covering the entire space of possible values.

   Furthermore (figure 3), we can make the match value symbolic instead of '8.8.8.8'. This will now match any packet. While this may seem useless, as the constraint 'IPv4.dst == *' will always be satisfiable, looking closer, it has a subtle, yet very important side-effect: from this point until the field gets rewritten, any subsequent constraint imposed on the 'IPv4.dst' field will be transitively imposed on the symbolic match value from the symbolic table as well. In the end, this will partition the space of the possible match values according to the symbolic paths discovered.

2. *There is no need to use multiple symbolic entries for the same action, when recirculation is not used.*

   To see why it suffices to use one symbolic entry per type of action, let us consider the case in which we added another extra one, with different symbolic entries for the same type of action. In figure 4 the first entry will behave the same as before. In the case of the second one, the action effects will be indistinguishable from the ones belonging to the first entry as they both rewrite the source address to an unconstrained symbolic value. When it comes to the symbolic match value, there will be an extra

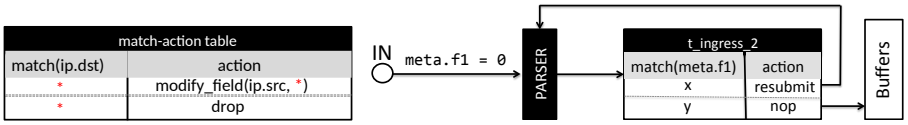

Figure 5: Symbolic table rules with one rule per action.

Figure 6: Using symbolic table entries to analyze the Resubmit P4 tutorial.

constraint that the two symbolic values must be different (to avoid an overlap), which brings no information, since this is already part of the P4 spec itself. Thus we conclude there is no information gain from deploying multiple symbolic entries per type of action.

On the other hand, a second symbolic entry corresponding to the 'drop' action will suffice to cover all the possible behavior for this table (figure 5).

3. *When recirculation is used, one rule per symbolic table per action is insufficient for exhaustive exploration.*

   Consider now the match-action table in Figure 6 where the parameterless resubmit action overwrites `meta.f1` to 1 and recirculates the packet, sending it to parser input. A valid question about programs using recirculation is whether they can have loops, as loops are very difficult to debug in practice.

   Symbolic table entries are perfect to explore this question, but if we simply apply the rule above and insert one rule per action, we might reach the wrong conclusion. In our example, when we insert any symbolic packet it will reach the `t_ingress_2` table with the metadata field set to zero. Then, it will either miss the resubmit entry and be dropped (explicitly or implicitly), or match the resubmit entry which implies that the symbolic entry `x` must be equal to 0. In the resubmit action, f1 is set to 1 and cannot match the same entry again, thus there can be no loop with the current table rules.

   Consider now what would happen if we add a different symbolic rule for the same resubmit action; say the symbolic variable is called `z`. When the packet gets recirculated, we know that `x` must be zero and that `f1` is set to 1; the second resubmit rule will be matched if `z=1` (a satisfiable constraint since `z` is unbound), and then we will have a loop.

   The take away is that to exhaustively explore P4 programs that use recirculation in an action, we need to add at least symbolic rules for the said action, as well as for all actions that can be matched *before* the recirculation (i.e. in other preceding tables).

4. *Any action can act as the default but there is no need to model this explicitly by further insertions of symbolic table entries.*

The rationale behind this can be regarded as a corollary of the previous observation. This default action will collect the negated constraints of all previous symbolic match values - which states nothing more than what the definition of the default action in the P4 spec already does.

Known modeling limitations:

1. Only parameters of immediate value type can be substituted by symbolic values. We mention that no production-grade P4 program we used for evaluation broke this assumption.

2. In the case of overlapping match conditions different entries must state different priorities. Our model does not cover such cases which means in practice, a symbolic path explored by the analysis might not be executed in reality, being shadowed by another one. In other words, multiple execution paths carry overlapping match conditions, the one executed in the concrete case is the one with the highest priority and this behavior should be considered as an additional post-symbolic execution step.

3. Even if a concrete table entry is validated according to Vera, it might get rejected by the switch at runtime because it either overlaps another entry with the same priority, or the table is full. Such corner-cases should also be considered outside Vera.

4 Fast verification of match-action tables

To ensure symbolic execution actually scales to large P4 programs, we need to optimize the match code while preserving its functionality. Consider a router FIB where we run symbolic execution with a symbolic destination address. If we generate code with one if/else per FIB entry, this will result in an infeasible number of paths. 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. We discuss this second part below. To achieve for the router (a) we can create a path per output port (i.e. groupping all packets that might leave on any given port); we can do this by forking the packet, and then applying the appropriate constraints.

Part (b) minimizes the number of constraints while ensuring the overall behaviour is equivalent to the if/else case. 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. In the router example, for instance, 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. Looking at the entire FIB, if we have many overlapping prefixes, 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.

This looks very discouraging given the commonality of FIBs with sizes in the hundreds of thousands. An additional complication comes from P4's support

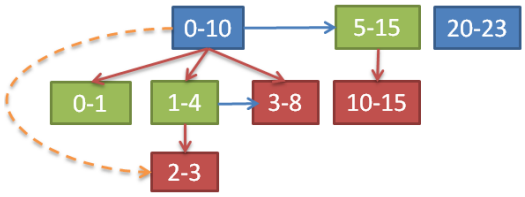

Figure 7: Match condition forest

for matching on multiple fields at once. Even the way matching can be specified evolved, ranging from exact matching to more complex matching schemes, such as: bit masks, longest-prefix matching, ranges. Furthermore any modelling technique must work well in the dynamic context in which data planes operate in production. Model computation should happen at the same time scales as table updates occur.

Our solution builds upon existing work [8, 2] and is applicable to a wide range of matching strategies including longest-prefix match, range, etc. We could not use an off-the-shelf solution coming from the study of packet classification algorithms because our optimization criteria are different. While data plane packet classification algorithms look to optimize classification speed while maintaining the feasibility of hardware-based implementation, in our case we try to reduce the number of symbolic constraints while preserving model correctness.

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.7. In the figure, one node represents the match condition for one table rule and the colours 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. (disconnected nodes in the figure).

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 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 an instance of this data structure, it is trivial to construct the minimal constraint required to match any given node but no other node of lower priority: add the node's constraint and the negated constraints of all its children and neighbors.

This data structure holds two benefits regarding model construction and update, while ensuring model optimality(fewest number of constraints being generated): i. constraint computation efficiency - given a reference to the subject node - the minimal set of conditions that have to be considered requires no tree traversal, thus requiring constant time (assuming the 'child-of' and 'neighbor-of' collections are attributes of every node and require a constant number of memory accesses). ii. tree construction efficiency - 'neighbor-of' links will not span across tree boundaries, they can only link nodes in the same tree, situated at the same level. This reduces the number of nodes to be checked for overlaps, making the node inserts very efficient in practice.

Next we show that our algorithm generates the theoretical minimum number of constraints. Given a match-action table entry (C, A) where C is the set of conditions that must be true in order to apply action A, we determine a set $O = \{(C_1, A_1), ..., (C_n, A_n)\}$ such that C overlaps with every C_i (for every given i, there is a packet P such that both C and C_i hold) and every math-action pair in O has a higher priority than (C, A). To mitigate the overlaps of a condition C, we could compute O and then the formula $F_O = \neg C_1 \wedge \neg C_2 \wedge, ..., \wedge \neg C_n$. In this complete set O, however, there are constraints that can fully overlap others, we say this is the case with two constraints C_i, C_j when $C_i \implies C_j$ for any packet P. An example of such a case is C_i being $TCPPort \in [1, 512]$ and C_j being $TCPPort \in [1, 1024]$. Taking this into consideration, in the equivalent form $\not{C}_j \implies \not{C}_i$, we can simplify F_O by removing C_i, for every pair (C_i, C_j) while preserving equivalence. This relation between C and C_i is reflected in our data structure by the *ancestor-of* links, which are not considered when building the reduced logical formula for mitigating overlaps.

For example, in figure 7 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.

5 Loop detection

Loop detection in programs is already a well-known and thoroughly investigated problem [6]. The trivial solution is to perform symbolic execution on the program at hand and store a snapshot of the memory state, together with the program instruction that ran to produced it. Whenever an instruction is executed again, the new state is compared against all the states observed in the

past. The state comparison routine iterates over all the program variables that make up the program state and checks whether they differ or not. Program variable x *differs* in two program states if x is bound to symbolic expression e_1 in one state, to e_2 in the other and the symbolic expression $e_1 \neq e_2$ is satisfiable. To perform this test we rely on a SMT solver, namely Z3[2].

The above procedure is guaranteed to terminate for SEFL models of P4 programs: program variables model header fields and metadata over a finite set, whose size is known in advance. The input of a P4 program (i.e. valid packet layouts) and the auxiliary per-packet variables (meta-data fields) are already known from the parser section of the P4 program and cannot be arbitrarily large (tens of primitive variables, in practice). Also, each header field has a fixed size, hence it can carry a limited number of values. Symbolic expressions (e.g. e_1) model *sets* of possible header field values. Thus, when running symbolic execution with loop detection, worst-case, all possible value-sets (encoded by symbolic expressions) will be explored for each variable. At this point it is guaranteed that a newly-constructed program state was already encountered, and the algorithm will terminate by reporting a loop.

The clear downside of this approach is complexity. Assuming the loop occurs after executing N instructions, the worst case time and space and complexity is quadratic w.r.t N.

However, SEFL models of a P4 program follow a structure that offers great room for optimization:

- loops cannot have arbitrary structure - there are a select few instructions that can cause loops (*resubmit*, *recirculate*). This insight can greatly reduce complexity. We only check memory state snapshots when these instructions are executed;
- in order to detect the loop formed by instructions $i_0, i_1, \ldots, i_n, i_0$, it is not necessary to start symbolic execution with instruction i_0. One can choose any instruction of the loop and invoke the store and compare routine there, instead of calling it across the whole loop. In our particular case, we know what P4 instructions can cause loops and the symbolic executor is altered to check for loops only there.

In the table below we show real-world benchmark results for the loop detection algorithm applied on a model containing a loop. The loop size is larger than 10 instructions and it can be detected by applying the loop detection check in any of its instructions. However, to emphasize the economy that is achieved by limiting the number of instructions considered for loop detection, we have reported the run time of the algorithm when using between 1 and 10 instructions. In this example, one can see that even for relatively small loops (10 instructions) a 10x improvement can be achieved.

Instructions	1	2	3	4	5	6	7	8	9	10
Time(ms)	165	261	373	498	632	774	924	1080	1242	1410

6 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 as intended 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 (which we call *state properties*) with *constraints over the possible paths* which the packets may take between ports (henceforth called *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.

To express path properties, we have considered a wide range of SDN policy languages, e.g. the Kinetic [9] family, FatTire [10],
NetPlumber [7], as well as approaches relying on logic programming (e.g. FML [5]). 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) [3]. CTL is a logical language designed for expressing properties over *computation trees*, which traditionally model all possible behaviours of a finite-state system. The main ingredients of the language are *branching operators* e.g. $\exists\varphi$ which states that on some execution trace of the system φ is true, and *temporal operators* e.g. $\mathbf{F}\varphi$ which states that in some state of the execution trace φ is true. NetCTL uses the very same temporal and branching operators together with SEFL instructions, in order to express state properties. For instance, the policy: $\forall\mathbf{F}$`destTCP == 80` evaluated at some port P of a box, expresses that on all possible packet paths from P, `destTCP` will eventually become 80.

Unlike Merlin, FatTree or NetPlumber, NetCTL is compositional: starting from simple properties, we can construct more complex 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: $\forall\mathbf{G}$(`ip != 192.168.0.0/16` $\rightarrow$ $\exists\mathbf{F}$ `port == Internet`). NetCTL can express many other properties including TCP connectivity and invariance across tunnels.

Although not as expressive as other temporal logics (e.g. CTL*), CTL benefits from fast model checking algorithms which run in linear time with respect to the size of the formula and to that of the modelled system. NetCTL inherits the same traits, although it does not explicitly rely on model checking.

The rest of this section is organised as follows: we first discuss the "*state-explosion problem*", and explain how our approach avoids it and how it is different from standard model checking. Next, we formally review the syntax of CTL and introduce a semantics on finite computation trees — this semantics is almost identical to the traditional one (defined over finite-state systems) and it allows us to prove that our NetCTL verification algorithm is correct. Fi-

nally, we discuss two algorithm optimizations: "*lazy-symbolic execution*" and "*conjunction verification*".

6.1 NetCTL and the state explosion problem

Model checking (with CTL or other temporal logics) is known to suffer from the "*state-explosion problem*" — describing programs, especially ones which involve concurrency, as finite-state systems requires a huge number of states. CTL model checking must process all of these states even though some may prove to be irrelevant for the truth-value of the formula. Symbolic model checking is a considerable improvement: it puts forth a compact representation for models — Ordered Binary Decision Diagrams (OBDD). In a nutshell, OBDDs use boolean encodings for states, thus scaling to models with hundreds of different parameters per state. While this adds an order of magnitude to the size of verifiable models it does not scale to arbitrary programs.

A different approach - Bounded Model Checking (BMC), unrolls the model under scrutiny for a number of k transitions, and describes it as a SAT formula. While BMC is fast, it sacrifies precision — unrolling only k transitions may not cover the complete behaviour of the model.

Our approach avoids the state-explosion problem altogether by relying on symbolic execution. Instead of building a finite-state system to exhaustively cover all possible behaviours of the program, we use symbolic execution to find *only those behaviours which may actually occur at runtime*. Moreover, we take a *lazy* approach in building and exploring those behaviours. The following example is an informal illustration:

```
void f (int x) {
  if (x>1)
    x = x - 1;
    if (x < 0)
      return x;
      else return 0;
  else return 1;
}
```

The program contains two imbricated `if` statements, and without making any reasoning on the possible values of `x`, we have three possible output states once `f` is called. Using symbolic execution, we can infer that the *then* branch of the inner `if` cannot be reached at execution, and thus rule it out as a possible state altogether. Moreover, if our aim is to find whether `f` can return `0`, we can stop symbolic execution once the *else* branch of the inner `if` (i.e. `return 0;`) is explored.

To conclude, NetCTL verification differs from standard model checking in that: (i) we handle the state explosion problem by relying on symbolic execution to reveal all program states which may be realized at execution and (ii) we explore only those states which are relevant for the formula (property) under scrutiny.

The rest of the section is organized as follows. First, we present the standard CTL syntax and semantics. Next, we introduce an algorithm which performs CTL verification on *finite computation trees*. The latter are obtained by performing symbolic execution on SEFL programs. We show the algoritm to be correct. Finally, we show how our algorithm is deployed for NetCTL verification — we show how state properties are verified and explain *lazy symbolic execution*. This optimization allows verification to stop once the formula is found true (or false), without exploring the entire symbolic execution tree of the program.

6.2 Background: CTL syntax and semantics

Let $Props$ be a finite set of propositions. The syntax of CTL is recursively defined as follows:

$$\varphi ::= p \in Props \mid \neg\varphi \mid \varphi \wedge \varphi \mid YZ[\varphi] \mid Y[\varphi \mathcal{U} \varphi]$$

$$\text{where } Y \in \{\exists, \forall\} \text{ and } Z \in \{\mathbf{X}, \mathbf{F}, \mathbf{A}\}$$

whenever the scoping of operators is clear (e.g. as in $\exists\mathbf{F}p$), we omit the square brackets. $\exists$ and $\forall$ are the standard branching operators, while $\mathbf{X}$ (*next*), $\mathbf{F}$ (*future*, or *sometime*), $\mathbf{A}$ (*always*) and $\mathcal{U}$ (*until*) are temporal operators.

Traditionally, the semantics of CTL is given over finite-state labelled transition systems. These objects are *models of programs*. Thus, in the (standard) entailment $M \models \varphi$, the formula φ is a property of a program M. Instead, we define the CTL syntax over *finite computation-trees* T obtained by the symbolic execution of a SEFL program P, on a symbolic input packet σ_0. Thus, in the entailment $T \models \varphi$, the formula φ is a property of the symbolic execution of program P, and not P itself. In SEFL, different symbolic input packets (e.g. with different header layouts) may produce different execution trees on the same SEFL program.

Also, note that, formally - transition systems are compact representations of infinite computation trees. In our semantics, computation trees are finite, since the SEFL programs on which we perform symbolic execution are expected to always terminate. Technically, we could easily construct an infinite computation tree from a finite one by adding *loop*-transitions on each leaf state, however this is not useful for our framework.

While the interpretation of a model has slightly changed in NetCTL, the underlying intuition of the operators remains the same. We continue with a few preliminary definitions.

A finite computation tree is a tuple $T = (Props, Q, L, next, q_0)$ where Q is a finite set whose elements we call *states*, $L : Q \to 2^{Props}$ is a state labelling function, $next : Q \to 2^Q$ is a *successor* function - $next(q) \subseteq Q$ are the *children* states of q and $q_0 \in Q$ is the initial state. If $next(q) = \emptyset$ then q is a *leaf-state*.

A ***trace*** in a tree T is a *maximal* finite sequence $q_1 \dots q_n$ (i.e. q_n is a leaf state) such that $q_{i+1} \in next(q_i)$ for each $1 \leq i \leq n-1$. A trace models a path discovered by symbolic execution of a program. Such a path is *maximal*, in the sense that symbolic execution either ends successfully or by reporting an error

(e.g. a constraint violation). Let $\Lambda_T(q)$ denote the set of traces in T starting with state q. The CTL semantics is given below:

$T, q \models p$ iff $p \in L(q)$

$T, q \models \neg\varphi$ iff $T, q \models \varphi$ is false

$T, q \models \varphi_1 \wedge \varphi_2$ iff $T, q \models \varphi_1$ and $T, q \models \varphi_2$.

$T, q \models \exists\mathbf{X}\varphi$ iff there exists $q' \in next(q)$ such that $T, q' \models \varphi$

$T, q \models \forall\mathbf{X}\varphi$ iff $T, q' \models \varphi$ for all $q' \in next(q)$

$T, q \models \exists\mathbf{F}\varphi$ iff there exists $q_1 \ldots q_n \in \Lambda_T(q)$ such that $T, q_i \models \varphi$ for some $1 \leq i \leq n$

$T, q \models \forall\mathbf{F}\varphi$ iff for all traces $q_1 \ldots q_n \in \Lambda_T(q)$, $T, q_i \models \varphi$ for some $1 \leq i \leq n$

$T, q \models \exists\mathbf{A}\varphi$ iff there exists $q_1 \ldots q_n \in \Lambda_T(q)$ such that $T, q_i \models \varphi$ for all $1 \leq i \leq n$

$T, q \models \forall\mathbf{A}\varphi$ iff for all traces $q_1 \ldots q_n \in \Lambda_T(q)$ we have $T, q_i \models \varphi$ for all $1 \leq i \leq n$

$T, q \models \exists[\varphi_1 \mathcal{U} \varphi_2]$ iff there exists $q_1 \ldots q_n \in \Lambda_T(q)$ such that $T, q_i \models \varphi_2$ for some $1 \leq i \leq n$, and $T, q_j \models \varphi_1$ for each $1 \leq j \leq i-1$

$T, q \models \forall[\varphi_1 \mathcal{U} \varphi_2]$ iff for all traces $q_1 \ldots q_n \in \Lambda_T(q)$, there exists an index $1 \leq i \leq n$ such that $T, q_i \models \varphi_2$ and $T, q_j \models \varphi_1$ for each $1 \leq j \leq i-1$

Note that the above semantics also covers the *leaf-state* case. For instance, consider evaluating $\forall\mathbf{A}p$ in a leaf state q from a computation tree T. Also, let p be true in state q. Then $\Lambda_T(q)$ is a singleton set which contains the single-state trace q. Since $T, q \models p$, by our definition it follows that $T, q \models \forall\mathbf{A}p$.

6.3 An algorithm for NetCTL verification

We introduce an algorithm which performs CTL verification on finite computation trees (i.e. checks $T, q \models \varphi$). We first introduce the procedures $check_\exists, check_\forall$ and $check\_next$, which are later used in our verification. We later show how symbolic execution can be used construct computation trees.

Algorithm 1: CTL verification of the $\exists$ path quantifier

Data: $check_\exists(T, q_0, \varphi)$
Result: Verifies if φ is true on some trace starting in q_0

```
for q ∈ next(q_0) do
    if check(T, q, φ) = true then
        return true
    end
end
return false
```

Algorithm 1 takes as input a tree T, initial state q_0 and formula φ and returns true if φ holds on *some* path starting at q_0.

Algorithm 2: CTL verification of the $\forall$ path quantifier

Data: $check_{\forall}(T, q_0, \varphi)$
Result: Verifies if φ is true on all traces starting in q_0

```
for q ∈ next(q0) do
    if check(T, q, φ) = false then
        return false
    end
end
return true
```

Conversely, Algorithm 2 takes as input a tree T, initial state q_0 and formula φ and returns true if φ holds on *all* paths starting at q_0.

Algorithm 3: CTL verification of path quantifiers

Data: $check\_next(T, q_0, \varphi)$
Result: Verifies if φ is true on some or all traces starting in q_0, depending on the path quantifier in scope from φ

```
case φ ≡ ∃ψ do
    return check∃(T, q0, φ)
end
case φ ≡ ∀ψ do
    return check∀(T, q0, φ)
end
```

Finally, Algoritm 3 checks the temporal operator in scope, from the input formula φ, by relying on $check_{\exists}$ and $check_{\forall}$.

The following is our algorithm for CTL verification on trees:

Algorithm 4: CTL verification on trees

Data: check(T, q_0, φ)
Result: $T, q_0 \models \varphi$

case $\varphi \equiv p$ **do**
  return $p \in L(q_0)$
end
case $\varphi \equiv \neg\psi$ **do**
  return check(T, q_0, ψ) $== false$
end
case $\varphi \equiv \psi_1 \wedge \psi_2$ **do**
  return check(T, q_0, ψ_1) *and* check(T, q_0, ψ_2)
end
case $\varphi \equiv Y\boldsymbol{X}\psi$ *with* $Y \in \{\exists, \forall\}$ **do**
  if $next(q_0) = \emptyset$ **then return** *false*;
  else
    case $\varphi \equiv \exists\boldsymbol{X}\psi$ **do**
      return $check_{\exists}(T, q_0, \psi)$
    end
    case $\varphi \equiv \forall\boldsymbol{X}\psi$ **do**
      return $check_{\forall}(T, q_0, \psi)$
    end
  end
end
case $\varphi \equiv Y\boldsymbol{F}\psi$ *with* $Y \in \{\exists, \forall\}$ **do**
  if check$(T, q_0, \psi) = true$ **then return** *true*;
  else
    if $next(q_0) = \emptyset$ **then return** $false$;
    else
      return $check\_next(T, q_0, \varphi)$
    end
  end
end
case $\varphi \equiv Y\boldsymbol{A}\psi$ *with* $Y \in \{\exists, \forall\}$ **do**
  if check$(T, q_0, \psi) = false$ **then return** $false$;
  else
    if $next(q_0) = \emptyset$ **then return** *true*;
    else
      return $check\_next(T, q_0, \varphi)$
    end
  end
end
case $\varphi \equiv Y[\psi_1 \mathcal{U} \psi_2]$ *with* $Y \in \{\exists, \forall\}$ **do**
  if check$(T, q_0, \psi_2) = true$ **then return** *true*;
  else
    if check$(T, q_0, \psi_1) = true$ *and* $next(q_0) \neq \emptyset$ **then return** $check\_next(T, q_0, \varphi)$;
    else
      return $false$
    end
  end
end

The algorithm relies on the observation that, on trees, the formula $Y\mathbf{F}\psi$ is true iff:

(i) ψ is true in the current state q **or**

(ii) $Y\mathbf{F}\psi$ is true on some (for $Y = \exists$) or all (for $Y = \forall$) children states of q.

Similarly, $Y\mathbf{A}\psi$ is true iff:

(i) ψ is true in the current state q **and**

(ii) $Y\mathbf{A}\psi$ is true on some (for $Y = \exists$) or all (for $Y = \forall$) children states of q.

The operator $\mathcal{U}$ receives the very same treatment. Note that, to verify $Y\mathbf{X}\psi$ in state q it suffices to verify ψ in some or all children of q. For this reason, on lines 12-17 of the Algorithm, we call $check_\exists$ (resp. $check_\forall$) with the inner formula ψ. We do not use *check_next*, since this call will trigger the verification of $Y\mathbf{X}\psi$ in the children states of q as well, thus violating the semantics of the $\mathbf{X}$ operator.

In order to prove correctness, we rely on a few observations captured by the following lemma.

Lemma 1 *Let T be a finite computation tree and $q_0 \dots q_k \dots q_n \in \Lambda_T(q_0)$ be a trace. If* check$(T, q_k, \exists \boldsymbol{F}\psi)$ *returns true for some state q_k, then* check$(T, q_0, \exists \boldsymbol{F}\psi)$ *also returns true.*

Lemma 1 states that if our algorithm reports the formula $\exists\mathbf{F}\phi$ as being true on some state q_k along a trace, it will also report it to be true at the beginning of the trace.

Proof: It is sufficient to prove the implication

$$\text{check}(T, q_i, \exists\mathbf{F}\psi) = true \implies \text{check}(T, q_{i-1}, \exists\mathbf{F}\psi) = true$$

for all $0 < i \leq n$. Suppose check$(T, q_i, \exists\mathbf{F}\psi)$ returns true. Then, the call check$(T, q_{i-1}, \exists\mathbf{F}\psi)$ will either return true if check(T, q_{i-1}, ψ) (and the lemma follows) or trigger the call $check\_next(T, q_{i-1}, \exists\mathbf{F}\psi)$ and followed by the call $check_\exists(T, q_{i-1}, \exists\mathbf{F}\psi)$. Subsequently, c.f. Algorithm 1, the call check$(T, q, \exists\mathbf{F}\psi)$ is evaluated for the succesor states q of q_{i-1}. As long as this latter call returns false, verification continues with another succesor state. Since q_i is a succesor of q_{i-1} and check$(T, q_i, \exists\mathbf{F}\psi)$ returns true, it follows that $check_\exists(T, q_{i-1}, \exists\mathbf{F}\psi)$ returns true.

By using the implication for $i = k, k-1, \leq 1$, the lemma follows. □

Proposition 6.1 (correctness) *The procedure* check(T, q_0, φ) *always terminates and is correct.*

Proof: Part 1 — termination: The algorithm performs a deph-first search of the tree T. Each state in T will be explored by a number of times bounded by the size of the formula (e.g. to check the conjunction $\psi_1 \wedge \psi_2$, we need to

check both ψ_1 and ψ_2 in the current state). Thus, the algorithm is guaranteed to terminate for any tree and initial state.

Part 2 — correctness: We prove $T, q_0 \models \varphi \iff \text{check}(T, q_0, \varphi) = true$, by induction over the formula structure of φ.

Basis: $\varphi = p$ *- straightforward.*

Induction step: $\varphi = \exists \boldsymbol{F}\psi$. Direction "⇒". Suppose $T, q_0 \models \exists \mathbf{F}\psi$. By the semantics definition, there exists $q_0 q_1 \dots q_n \in \Lambda_T(q_0)$ such that $T, q_i \models \psi$ for some $0 \leq i \leq n$. By induction hypothesis, the call check(T, q_i, ψ) returns *true*. Then, the call check$(T, q_i, \exists \mathbf{F}\psi)$ will also return *true* by lines 20-21 of the algorithm. By Lemma 1, we have that check$(T, q_0, \exists \mathbf{F}\psi)$ returns true, since q_i is some state from the trace $q_0 \dots q_n$.

Direction "⇐". Suppose check$(T, q_0, \exists \mathbf{F}\psi) = true$. Then, either we have (i) check$(T, q_0, \psi) = true$, and by induction hypothesis $T, q_0 \models \psi$ thus $T, q_0 \models \exists \mathbf{F}\psi$ by the semantics definition, or (ii) check$(T, q_1, \exists \mathbf{F}\psi) = true$, for some state $q_1 \in next(q)$. The same line of reasoning follows for q_1: either check(T, q_1, ψ) returns true or check$(T, q_2, \exists \mathbf{F}\psi)$ returns true, where q_2 is some successor state of q_1. The sequence of check calls must terminate, therefore check$(T, q_i, \psi) = true$ for some state q_i which is the ith successor of q_0. The sequence $q_0 \dots q_i$ is not necessarily maximal, but we can extend it to $q_0 \dots q_i q_{i+1} \dots q_n$ by arbitrarily selecting successor states $q_k \in next(q_{k-1})$ for $i + 1 \leq k \leq n$. By induction hypothesis, we have $T, q_i \models \psi$. Finally, by the CTL semantics $T, q_0 \models \exists \mathbf{F}\psi$.

The induction steps corresponding to other formulae can be shown in the exact same way. □

Theorem 1 (Complexity) check(T, q_0, φ) *takes* $O(|T| \cdot |\varphi|)$ *time to complete.*

Proof: Worst-case, the algorithm will construct all possible paths from $\Lambda_T(q_0)$. For each temporal formula, the algorithm first proceeds to examine the subformula at hand in the *current*-state and then may continue the verification of the formula, in the *next*-state. Thus, for each state, at most $|\varphi|$ verifications may be performed. Informally, this amounts to performing symbolic execution **once for each subformula** of φ. □

6.4 Algorithm implementation

Propositions in NetCTL. A proposition in NetCTL expresses a state-property, i.e. **a propery of a packet** at a **specific** location in the network. We use a subset of SEFL to express such properties. For instance, the instruction `Constrain(IPDst,==10.0.0.1)` is used to check if the IP destination field satistifies the respective constraint. Thus, propositions $p \in Props$ are abstractions for such instructions. In order to check $p \in L(q)$ (i.e. p is true in state q), we proceed as follows. Suppose p is a SEFL instruction and q is a state obtained via symbolic execution. We construct the complement $\overline{p}$ of p, by adding negation to the constraint from p. We then perform symbolic execution on program $\overline{p}$ in symbolic state q. If this yields successful paths, then it is possible for some concrete packet to satisfy the constaints in $\overline{p}$. Thus p is false in state q. Conversely, if no succesful paths are found, p is true.

Lazy state exploration. First, note that Algorithm 4 works on the symbolic execution tree T. Building it in advance is not always useful — some traces may not be explored during verification, hence there is no need to produce them. Second, note that our algorithm does not deal specifically with how symbolic execution is performed — this is actually *hidden* by how the set *next* of succesor states is built.

To make NetCTL fast, we construct the symbolic execution tree T step-by-step. For a given current state q, each successor state $q' \in next(q)$ is obtained per-need basis. Symbolic execution executes the program at hand until some successor q' is found, and then stops and stores the current execution context. A successor is found when:

- a `Forward` instruction is executed (i.e. the packet is forwarded to some port of the network)
- symbolic execution stops with failure
- the current program terminates (e.g. the packet is not forwarded or no links to the forwarding port exist)

If verification does not complete for q', then symbolic execution continues from the current execution context, and a new succesor state is built. In this way, instead of instrumenting a complete execution tree T, our implementation only stores a stack of execution contexts.

Checking conjunctions efficiently. In order to verify properties containing binary boolean operations e.g. $\psi_1 \wedge \psi_2$ in state q, Algorithm 4 would perform verifications check(T, q, ψ_1) and check(T, q, ψ_2). This amounts to performing two symbolic executions from state q, one for each subformula. This is inefficient since some successor states may be constructed and examined twice. To avoid this, we keep a flag for each boolean operator of the formula at hand. If, for instance, ψ_1 (resp.ψ_2) is found true in the current state, then verification continues with ψ_2 (resp. ψ_1). If ψ_1 or ψ_2 is false, verification stops. If ψ_1 and ψ_2 require building a successor state (i.e. they contain temporal operators), then verifying the conjunction continues.

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. Vera is a tool that translates P4 to SEFL and then uses symbolic execution to find potential bugs. Vera can also check correctness of properties expressed in NetCTL.

In this technical report, we have provided detailed information about certain aspects of Vera that are not thoroughly captured in the conference publication. These aspects include the "big-step" operational semantics for P4 and SEFL,

correctness of translation, a description of the loop detection algorithm, further details on the match-action data structure and a formal explanation of NetCTL.

References

[1] Pat Bosshart et al. "P4: Programming Protocol-independent Packet Processors". In: *SIGCOMM Comput. Commun. Rev.* 44.3 (July 2014).

[2] Marco Canini et al. "A NICE Way to Test Openflow Applications". In: *Proc. NSDI'12*.

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

[4] Flemming Nielson Hanne Riis Nielson. "Semantics with Applications: An Appetizer". In: *Springer Verlag, London, 2007*.

[5] Timothy L. Hinrichs et al. "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.

[6] Antoine Joux. *Algorithmic Cryptanalysis*. Chapman & Hall/CRC, 2009.

[7] Peyman Kazemian et al. "Real Time Network Policy Checking Using Header Space Analysis". In: *Proc. NSDI'13*.

[8] Ahmed Khurshid et al. "VeriFlow: Verifying Network-wide Invariants in Real Time". In: *Proc. NSDI'13*.

[9] Hyojoon Kim et al. "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.

[10] Mark Reitblatt et al. "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.

[11] Radu Stoenescu et al. "SymNet: Scalable symbolic execution for modern networks". In: *SIGCOMM*. 2016. DOI: 10.1145/2934872.2934881. URL: http://doi.acm.org/10.1145/2934872.2934881.