A Precise and Expressive Lattice-theoretical Framework for Efficient Network Verification

Alex Horn*
Apple

Ali Kheradmand*
University of Illinois at Urbana-Champaign

Mukul R. Prasad
Fujitsu Laboratories of America

***Abstract*—Network verification promises to detect errors, such as black holes and forwarding loops, by logically analyzing the control or data plane. To do so efficiently, the state-of-the-art (e.g., Veriflow) partitions packet headers with identical forwarding behavior into the same packet equivalence class (PEC).**

Recently, Yang and Lam showed how to construct the minimal set of PECs, called atomic predicates. Their construction uses Binary Decision Diagrams (BDDs). However, BDDs have been shown to incur significant overhead per packet header bit, performing poorly when analyzing large-scale data centers. The overhead of atomic predicates prompted ddNF to devise a specialized data structure of Ternary Bit Vectors (TBV) instead.

However, TBVs are strictly less expressive than BDDs. Moreover, unlike atomic predicates, ddNF's set of PECs is not minimal. We show that ddNF's non-minimality is due to empty PECs. In addition, empty PECs are shown to trigger wrong analysis results. This reveals an inherent tension between precision, expressiveness and performance in formal network verification.

Our paper resolves this tension through a new lattice-theoretical PEC-construction algorithm, #PEC, that advances the field as follows: (i) #PEC can encode more kinds of forwarding rules (e.g., ip-tables) than ddNF and Veriflow, (ii) #PEC verifies a wider class of errors (e.g., shadowed rules) than ddNF, and (iii) on a broad range of real-world datasets, #PEC is $10\times$ faster than atomic predicates. By achieving precision, expressiveness and performance, this paper answers a longstanding quest that has spanned three generations of formal network analysis techniques.

I. Introduction

In complex networks, misconfigurations continue to be common [1], [2], causing costly unscheduled outages or compromising security [3]–[5]. This has generated significant interest in formally analyzing network behavior on the control (e.g., [6]–[9]) or data plane (e.g., [10]–[15]), a class of formal methods collectively known as *network verification*. In this paper, we provide a new algorithm and data structure that can serve as a foundation for both forms of network verification.

What make network verification interesting is its predictive power: it promises to find network-related errors that traditional diagnostic tools, such as ping and traceroute, in general cannot. To accomplish this feat, network verification creates a mathematical model of the network to logically analyze the packet forwarding behavior of packets, rather than merely observing network traffic. This is an inherently difficult task: even reachability checking in the data plane is NP-hard [10]. Much research therefore has gone into making formal network analysis as efficient as possible.

* This work was completed at Fujitsu Laboratories of America.

978-1-7281-2700-2/19/$31.00 2019 © IEEE

First-generation formal network analysis tools (e.g., [10], [16]–[24]) rely on *SAT/SMT solvers*, highly optimized backtracking decision procedures for solving propositional or first-order logic problems. However, SAT/SMT solvers are too slow to enumerate all witnesses of each network property violation [25], and SAT/SMT solvers tend to perform poorly on reachability queries over many distinct network paths [15].

This bottleneck prompted *second-generation* formal network analysis techniques to use a geometric model for packet classification instead, notably in the form of *Header Space Analysis* (HSA) [11], [12], [26]. At its core, HSA represents packet headers as the difference of cubes in a multi-dimensional hyperspace. While compact, a significant drawback of HSA's difference of hypercube representation is that it is computationally expensive to evaluate in general. This explains why HSA uses a lazy evaluation strategy, which still has performance problems (akin to lazy functional languages).

By contrast, *third-generation* formal network analysis tools avoid the problems of lazy evaluation by *pre-computing* a family of disjoint sets of packet headers. We call these *packet equivalence classes* (PECs). Intuitively, each PEC contains packet headers that experience the same forwarding behavior through the network at each router—a form of lossless compression that has been shown to make formal network analysis more efficient in both time and space [27].[1]

Formal network analysis tools based on PECs include Veriflow [30], APV [27], ddNF [28] and Delta-net [14], all of which detect a myriad of network errors—such as black holes, forwarding loops, reachability and isolation violations—and PECs help to do so in a vendor-agnostic manner. In this paper, we focus on Veriflow [30], APV [27] and ddNF [28]. These tools can encode match conditions with possibly many packet header fields, so-called *multi-dimensional match conditions*.

However, reasoning about multi-dimensional match conditions in a priority-ordered list (such as a forwarding table) is challenging, because a higher-priority rule x may overlap with a lower-priority rule y. Such overlapping amounts to logical negation (i.e., $y \wedge \neg x$), because x needs to be subtracted from y. The crux of the problem is that logical negations can lead to an exponential number of case splits. Consider some packet header filter that uses the match condition $1*1*0$, an instance of a *Ternary Bit Vector* (TBV) where '$*$' matches either '1' or

[1] While, in the worst case, the number of generated PECs is exponential in the number of match conditions, in practice there are only relatively few PECs [27], [28]. In fact, in restricted, but not uncommon cases, the number of PECs is even linear in the number of match conditions [14], [29].

'0'. The number of case splits due to TBV-negation, such as ¬(1∗1∗0), is generally exponential in the length of the TBV.

Binary Decision Diagrams (BDDs) [31], [32] can efficiently represent such case splits, and APV [27] uses BDDs to compactly represent the space of packet headers, including their negation. By constructing BDDs, APV produces also canonical and optimal PECs, called *atomic predicates*, which form the unique and smallest partition of packet headers [27].

But there is a catch: BDDs incur significant overhead per bit in each packet header field, a performance bottleneck in real-world network analysis [28]. This prompted ddNF to not use BDDs. Instead, ddNF constructs PECs by only intersecting TBVs. The intersection of TBVs is very efficient due to their compact representation in memory, and experiments using Azure data center snapshots confirm that ddNF is significantly more efficient than APV, a remarkable achievement.

However, both ddNF's TBVs as well as Veriflow's multi-dimensional trie data structure have inherent limitations (§ II-B): they cannot efficiently represent match conditions over arbitrary sets and ranges of ports, and their complements. Consequently, ddNF and Veriflow cannot analyze common firewall rules in practice (§ IV), such as iptables rule-sets [33].

Furthermore, ddNF and Veriflow's PECs are not minimal (§ II-C). In the case of ddNF, we show that this non-minimality can lead to wrong analysis results, e.g., ddNF is unsuitable for detecting shadowed rules. We catalog over forty cases of such imprecision (§ IV-C). This motivates the following question:

> *Can the construction of precise and expressive packet equivalences classes be also efficient?*

Our paper answers this question in the affirmative through a new lattice-theoretical PEC-construction algorithm (§ III), #PEC, that combines the precision and expressiveness of atomic predicates with the scalability of ddNF. #PEC is more expressive than Veriflow and ddNF, because its encoding is not tied to TBVs. As a result, for instance, #PEC can check match conditions with arbitrary ranges, e.g., iptables rule-sets. Moreover, #PEC can detect errors, such as shadowed rules, that ddNF cannot in general, since its analysis is imprecise.

We show that ddNF's imprecision is due to PECs that are empty. We detect such empty PECs—a coNP-hard problem—by efficiently counting the packet headers in each PEC. This way #PEC achieves full precision, and it does so $10-100\times$ faster than SAT/SMT and BDD-based solutions that encode the PEC-emptiness problem into propositional logic. Moreover, by detecting empty PECs, #PEC constructs PECs that are unique and minimal (§ III-F), achieving the optimality of atomic predicates, but at least $10\times$ faster than APV (§ IV).

To avoid the aforementioned limitations of TBVs and multi-dimensional trie data structures, we organize packet headers in a *meet-semilattice* [34] (§ III-B). Through this lattice-theoretical framework, #PEC can formally analyze a strictly broader class of forwarding filters than ddNF and Veriflow.

By achieving precision, expressiveness and performance, we answer a longstanding quest that has spanned three generations of formal network analysis techniques.

	SOURCE	DESTINATION	PROTO	ACTION
①	0.0.0.4/30	0.0.0.0/28	!UDP	DROP
②	0.0.0.0/29	0.0.0.12/30	UDP	DROP
③	0.0.0.4/30	0.0.0.12/30	ANY	FORWARD

Fig. 1: Forwarding table (using priorities) with 3-dimensional match conditions that neither Veriflow nor ddNF can analyze

II. BACKGROUND AND MOTIVATION

We start by giving background on formal network analysis (§ II-A), illustrating why achieving expressiveness (§ II-B) and precision (§ II-C) at the same time is challenging.

A. Background: Formal Network Analysis

In this subsection, we explain through illustrations what makes multi-dimensional match conditions challenging to formally analyze. Readers familiar with PEC-based formal network analysis may wish to skip this subsection for now.

Consider two physically connected routers ν_1 and ν_2. The network operator wants to check the absence of forwarding loops between ν_1 and ν_2. Assume that ν_2 forwards packets to ν_1 according to the forwarding table in Figure 1. This forwarding table has three priority-ordered rules: ①, ② and ③, where ① has highest priority. Since the match conditions of ①, ② and ③ filter packets based on three packet header fields, they are instances of 3-dimensional match conditions.

Consider the lowest-priority rule ③ in Figure 1. It is not difficult to see that the set of packets matched by ③ correspond to the logic formula ③∧¬①∧¬②. This formula says that ③ matches only packet headers that are not matched by ① or ②, thereby encoding the fact that both ① and ② have a higher priority than ③.

To understand the significance of such logic formulas, assume that ν_1 forwards to ν_2 all packets matched by either ①, ② or ③, i.e., ①∨②∨③. Abstractly, formal analysis tools essentially reason about the forwarding behavior of a network in terms of a directed graph whose edges are annotated by such logic formulas (or PECs as we shall see), as illustrated in Figure 2. The existence of a forwarding loop between ν_1 and ν_2 depends on whether the logic formula $\phi = (① \vee ② \vee ③) \wedge (③ \wedge \neg① \wedge \neg②)$ is satisfiable or not; equivalently, does there exist a packet header such that formula ϕ can evaluate to true?

The challenge for PEC-based formal analysis tools is to be able to express complex multi-dimensional match conditions, while also being able to efficiently and precisely solve the resulting constraint systems via PECs. Unlike SAT/SMT solvers, PECs give by default the set of all such solutions (if any).

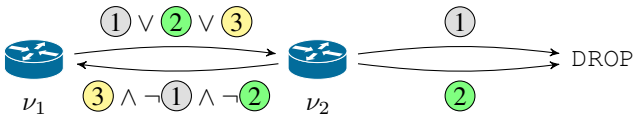
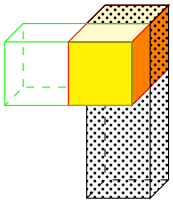
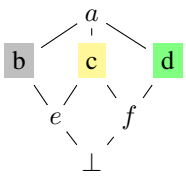


Fig. 2: Verification: is there a forwarding loop, or not?



(a)



(b)

Fig. 3: (a) Geometric view of the three 3-dimensional match conditions in Figure 1; (b) Hasse diagram of the meet-semilattice induced by these match conditions (see also § III)

What does a solution to this challenge entail? To answer this, consider Figure 3a, a geometric view of the three match conditions in Figure 1. Since there are 3-dimensional match conditions, Figure 3a has three axes: the x-axis and y-axis correspond to the range of the source and destination IP addresses, respectively, whereas the z-axis evenly divides the space into `UDP` and non-`UDP` packets. The color of each rectangular cuboid corresponds to ①, ② and ③. The key idea behind PECs is to divide the whole geometric space into disjoint sub-spaces prior to the analysis.

Note that each overlapping of cuboids corresponds to an overlapping of a pair of rules. In general, however, reasoning about the intersection of higher-dimensional cuboids, as in Figure 3a, is NP-hard. For example, there is no forwarding loop between ν_1 and ν_2, since the 3-dimensional space denoted by ϕ is in fact empty, an instance of an NP-hard query.

B. Challenge: Expressiveness

Notice that the highest-priority rule ① in Figure 1 complements an individual packet header field. That is, ① matches only *non*-`UDP` packet headers whose source and destination IP address match `0.0.0.4/30` and `0.0.0.0/28`, respectively. However, the PEC-construction schemes in Veriflow and ddNF are not designed for multi-dimensional match conditions with arbitrary ranges, sets of values, or their complements (all of which can be found in iptables rule-sets [33]).

Veriflow and ddNF's limitation is due to the fact that they are tied to TBVs, where Veriflow represents TBVs as a trie data structure with nodes that can have three children for '0', '1' and '$*$' [35]. The problem is that a single TBV cannot represent match conditions such as the non-`UDP` example above. As another instance, an arbitrary range that is not an IP prefix can only be represented by multiple TBVs. This renders the TBV representation of match conditions inefficient and impractical. By contrast, #PEC can efficiently encode such match conditions via element types (§ III-B). While APV can represent the same match conditions as #PEC, APV's reliance on BDDs makes it at least $10\times$ slower than #PEC (§ IV-D).

C. Challenge: Precision and Minimality of PECs

For ddNF to be able to analyze the network in Figure 2, let us further simplify the example by replacing the the three match conditions of the rules ①, ② and ③ with the following three IP prefixes, respectively: $x = $ `10.57.0.0/19`, $y = $ `10.57.32.0/19` and $z = $ `10.57.0.0/18`. This simplifies the forwarding table in Figure 1 accordingly, where each rule now only matches packets based on longest IP prefix matching—something that ddNF is designed to handle. We remark that our simplification preserves a vital characteristic of the example: reasoning about it requires only *two* PECs, which form atomic predicates by definition (§ III-F).

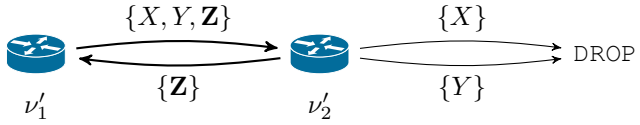


Fig. 4: Wrong result in ddNF, due to non-minimal PECs

However, ddNF constructs *three* PECs, denoted by uppercase letters: X and Y that represent all IP addresses in $x = $ `10.57.0.0/19`, $y = $ `10.57.32.0/19`, respectively, and Z for all IP addresses in $z = $ `10.57.0.0/18`, *except* those IP addresses in x and y. By construction, X, Y and Z are disjoint, so $\{X, Y, Z\}$ is indeed a set of PECs.

The crux of the problem is that $\{X, Y, Z\}$ is not minimal, because there is a PEC that is empty, namely Z. To illustrate the impact of this superfluous PEC, consider Figure 4. Notice that each edge in Figure 4 has a corresponding edge in Figure 2. The problem is that ddNF's PEC construction fails to precisely capture the Boolean formulas along the edges in Figure 2: ddNF wrongly reports a forwarding loop between ν'_1 and ν'_2, because the edges in Figure 4 labelled by Z (shown in bold) form a spurious cycle. This cycle is spurious, and therefore leads to a *false alarm*, because the conjunction of the corresponding Boolean formulas in Figure 2 is unsatisfiable.[2]

In addition to false alarms, ddNF's limitation can also manifest itself as a *failure to detect network-related errors*: in our example, ddNF will not detect that the last rule in Figure 2 is shadowed.[3] The possibility for both *false alarms* as well as *false negatives* means that ddNF comes with the overhead to always sanity check its results, a serious limitation (§ IV-C).

Our experiments (§ IV) show that #PEC is $10-100\times$ faster than alternative SAT/SMT solvers and BDD-based solutions for detecting empty PECs. To achieve this speedup, #PEC exploits the fact that it is enough to find the number of packets in a PEC to check the PEC's emptiness, rather than finding a witness for its non-emptiness. This reveals, in particular, that in the context of formal network analysis *counting* is much faster than backtracking on a wide range of realistic datasets.

III. Lattice-theoretical Framework

In this section, we highlight the technical approach behind #PEC (§ III-A), before explaining its data structures (§ III-B–III-C) and PEC-construction algorithm (§ III-D). We explain

[2] A Boolean formula is *unsatisfiable* whenever it can never evaluate to true.

[3] There are two kinds of shadowed rules: (i) a single higher-priority rule covers some lower-priority rule, or (ii) the *union* of several higher-priority rules covers some lower-priority rule. Here ddNF fails to detect the latter.

how to answer queries in #PEC (§ III-E). Finally, we show that #PEC constructs the minimal set of PECs (§ III-F).

A. Technical Approach

To illustrate our approach, reconsider the match conditions ①, ② and ③ in Figure 1. Recall that ① complements an individual packet header field. We can represent such and other match conditions by so-called *element types* (Figure 7).

The geometric interpretation (§ II-A) we considered in Figure 2 was only in three dimensions. In general, the geometric view is unfeasible, since it requires reasoning about hypercubes as the number of packet header fields increases. Instead, #PEC constructs a *meet-semilattice*, a form of partially ordered set in which every finite subset of elements has a greatest lower bound [34]. In doing so, #PEC is able to represent match conditions that ddNF and Veriflow cannot, while also achieving precision and performance, as described below.

Figure 3b shows the Hasse diagram of the meet-semilattice produced by #PEC, given the match conditions in Figure 1. A *Hasse diagram* has an edge from a vertex v to a vertex u whenever u is a subset of v (written $u \subset v$), and there is no other vertex w such that $u \subset w \subset v$. In other words, only non-transitive edges are included in the Hasse diagram.

Figure 5 gives the more familiar interpretation of the elements in the meet-semilattice as match conditions. Note that the color of the rows in Figure 5 corresponds to the coloring of the corresponding match conditions in Figure 1.

Observe that the meet-semilattice in Figure 3b contains more elements than there are match conditions. Intuitively, the reason is that the meet-semilattice describes the overlapping of all match conditions. This intuition is made precise by the requirement that every meet-semilattice is closed under intersection: it must contain every element that is the result of intersecting sets of other elements. For example, elements e and f are in the meet-semilattice because $e = \text{b} \cap \text{c}$ and $f = \text{c} \cap \text{d}$, respectively. The last two rows in Figure 5 give a more familiar interpretation of elements e and f.

#PEC bases its meet-semilattice construction on the algorithm in [36], but with a twist: we maintain the *cardinality* of each PEC—the number of packet headers in each PEC. Crucially, an empty PEC has cardinality zero. This way, #PEC detects that $e \cup f = \text{c}$, which ddNF cannot. Unlike a *per bit* combinatorial backtracking search with SAT/SMT solvers, our cardinality computation uses the structure of the semilattice and harnesses the computing power of ALUs [37] (IV-D2).

$$A \triangleq a - (\text{b} \cup \text{c} \cup \text{d}) \qquad C \triangleq \text{c} - (e \cup f) \qquad E \triangleq e$$
$$B \triangleq \text{b} - e \qquad D \triangleq \text{d} - f \qquad F \triangleq f$$

Fig. 6: PECs due to the match conditions (colored rows) in Figure 5, using the Hasse diagram in Figure 3b

Next, we discuss the technical details behind #PEC, specifically: its data structures for representing match conditions (§ III-B) and PECs (§ III-C), as well as its algorithm that use these data structures to compute PECs (§ III-D) and answer a network operator's queries about the network (§ III-E).

B. Representation of Match Conditions via Element Types

At its core, #PEC features an abstract data type for match conditions, called *element type*, which strictly generalizes the expressiveness of Veriflow and ddNF's TBVs. For example, using element types, we encode the match conditions in Figure 5 as 3-tuples $\langle F_1, F_2, F_3 \rangle$ where F_1 and F_2 denote ranges of source and destination IP addresses, respectively, whereas F_3 denotes a set of protocols where '!' on the protocol field is encoded by efficiently complementing a bitset.

For its generalization, #PEC imposes only two basic requirements on element types: elements must form a finite partially ordered set, whose cardinality must be computable in polynomial time. Figure 7 shows fundamental element types used in practice that satisfy these requirements where the partial ordering corresponds to the usual subset inclusion order. For example, if x and y are of type `ip_prefix`, $x \subseteq y$ means that every IP address in x appears also in y.

Each element type features three operators: equality ($=$), intersection ($\cap$), and cardinality. All element types in Figure 7 can be efficiently implemented using data structures that use contiguous memory, and our implementations have therefore high cache locality, similar to TBVs in ddNF. We remark that since $x \subseteq y$ holds exactly if $x \cap y = x$, the subset-inclusion operator ($\subseteq$) is derived automatically, a default implementation that can be optionally optimized.

Some element types such as `disjoint_ranges` and `set<T>` where `T` is a fixed-size type, support a complement

	SOURCE	DESTINATION	PROTOCOL
a	0.0.0.0/0	0.0.0.0/0	ANY
b	0.0.0.4/30	0.0.0.0/28	!UDP
c	0.0.0.4/30	0.0.0.12/30	ANY
d	0.0.0.0/29	0.0.0.12/30	UDP
e	0.0.0.4/30	0.0.0.12/30	!UDP
f	0.0.0.4/30	0.0.0.12/30	UDP

Fig. 5: Six semi-meetlattice elements induced by the three 3-dimensional match conditions b, c and d where b features negation of a protocol header field, e.g., '!UDP'

ELEMENT TYPE	DESCRIPTION
`ip_prefix`	IP prefix, convertible to `range`
`optional<T>`	Wildcard or a value of type `T`
`tbv<N>`	Fixed-length TBV
`range`	Half-closed interval, e.g., $[0:10)$
`disjoint_ranges`	Set of disjoint ranges
`set<T>`	Finite set of values of type `T`
`tuple<`$\mathtt{E}_1$`,...,`$\mathtt{E}_k$`>`	Tuple where $\mathtt{E}_j$ are element types

Fig. 7: Element types to form complex (i.e., multi-dimensional) packet header match conditions

('!') operator. This allows for more complex match conditions, such as complements on protocol fields as in Figure 5. By contrast, the `tuple` element type has no complement operator because it is computationally too expensive [11].

More generally, by introducing element types, #PEC can tightly control the use and effects of complements, allowing only forms of negation that can be efficiently implemented.

Example. The match conditions in Figure 1, and the corresponding meet-semilattice elements in Figure 5, can be represented by 3-tuples of element type `tuple<ip_prefix, ip_prefix, set<protocol>>` where `protocol` is an enumeration type. Alternatively, if there is no need to be able to complement the protocol header field, the last tuple component could be also replaced by `optional<protocol>`.

C. PEC-representation as a DAG

#PEC represents the Hasse diagram of a meet-semilattice as a *directed acylic graph* (DAG). Since such a Hasse diagram can be shown to be unique up to graph isomorphism [34], so is the DAG that #PEC constructs using the later algorithm.

Therefore, #PEC represents each PEC by a pointer to a DAG node. Each such PEC denotes the packet headers that are in the element associated with the pointed to DAG node, minus the elements in its children. For example, given the meet-semilattice in Figure 3b, uppercase letter A denotes the PEC that includes the packet headers in a, excluding those in b, c and d. Figure 6 defines the other PECs similarly. By construction, all PECs are pair-wise disjoint. For example, the intersection of the PECs B and C in Figure 3b is empty, whereas $\mathrm{b} \cap \mathrm{c}$ is non-empty. Each node n in the DAG has the following three fields: (i) $n.elem$ denotes the match condition associated with n; (ii) $n.children$ contains all the DAG nodes c such that $c \neq n$ and $c.elem \subseteq n.elem$ and there is no other DAG node c' such that $n \neq c' \neq c$ and $c.elem \subseteq c'.elem \subseteq n.elem$; (iii) $n.cardinality$ corresponds to the number of packet headers in the PEC denoted by n.

Example. Let n_a be the root node of the DAG in Figure 3b such that $n_a.elem = a$ and $n_a.children = \{n_b, n_c, n_d\}$. Note that neither n_f and n_e are in $n_a.children$, because they are not direct children of n_a. We shall see that #PEC computes $n_c.cardinality = 0$, i.e., the PEC denoted by n_c is empty.

D. Algorithm for Computing PECs

The algorithm of #PEC is divided into three procedures, each of which accesses the global variable *Modified_Nodes* that determines what PEC-cardinalities need to be recomputed. We explain each procedure in turn.

The main procedure, INSERT (Algorithm 1), takes as input an *element*—a match condition of the kind as explained in § III-B—that is to be added into the meet-semilattice. To do so, INSERT calls FIND_OR_CREATE_NODE(*element*) which uses a hash table (not shown) to determine when a new DAG node n, satisfying $n.elem = element$, has to be created or not. Only in the former case, when $new = \mathbf{true}$, is n inserted into *Modified_Nodes* and the subprocedures INSERT_NODE and COMPUTE_CARDINALITY are called, as discussed next.

To insert a new node n into the DAG that represents the Hasse diagram of the meet-semilattice, INSERT_NODE is called on the root of the DAG. Intuitively, INSERT_NODE (Algorithm 2) works by case analysis on the three possible partial orderings between a pair of nodes (line 4, 6 and 9). By using *max_children*, we only add a child c to a parent provided that c's element is maximal with respect to the other children elements in Γ (line 20 and 21–23), thereby ensuring that all children of a parent remain mutually incomparable.

The correctness of the induced updates to the DAG edges (line 17, 22 and 23) follows directly from the proof in [36], since we only augment the algorithm by maintaining what nodes have been modified along the way (see *Modified_Nodes* on line 18 and 15). There may be multiple such nodes when the insertion of a single new match condition requires multiple DAG nodes to be created, due to the requirement that the lattice be closed under meets, as illustrated next.

Example. Consider the DAG in Figure 8a whose nodes correspond to the elements a through e in Figure 8c, and suppose we want to insert element $\mathbf{f}$ now. For clarity in what follows, let n_p denote a DAG node that satisfies $n_p.elem = p$. As expected, the call INSERT($\mathbf{f}$) creates DAG node $n_{\mathbf{f}}$. However, it also creates $n_{\mathbf{g}}$ and $n_{\mathbf{h}}$ for elements $\mathbf{g}$ and $\mathbf{h}$ in Figure 8c, respectively, since $b \cap \mathbf{f} = \mathbf{g}$ and $d \cap \mathbf{g} = \mathbf{h}$; hence, $Modified\_Nodes = \{n_a, n_b, n_d, n_{\mathbf{f}}, n_{\mathbf{g}}, n_{\mathbf{h}}\}$. The resulting DAG is shown in Figure 8b (new elements shown in bold). □

We compute PEC-cardinalities using the set of modified nodes: once Algorithm 2 returns, the computation continues on line 8 in Algorithm 3 where COMPUTE_CARDINALITY is called for every node in *Modified_Nodes* (line 6–7), and as it does so *Modified_Nodes* shrinks after each call to COMPUTE_CARDINALITY, until it becomes empty.

The re-computation of PEC-cardinalities works as follows. COMPUTE_CARDINALITY in Algorithm 3 traverses the DAG using a queue (line 2). We initialize the PEC-cardinality of the input DAG node n by counting the packets in its associated element (line 3), using the cardinality operator from § III-B, before subtracting the PEC-cardinality of n's descendants. To do so, the PEC-cardinality of all the modified children is computed (line 9) prior to updating n's PEC-cardinality (line 12). Since there may be multiple paths to the same node in the DAG, COMPUTE_CARDINALITY uses a local variable (line 2) to ensure it does not subtract too much (line 7) as it traverses the sub-DAG rooted at COMPUTE_CARDINALITY's input DAG node. By deferring the re-computation of PEC-cardinalities for several insertions, the computation can be

Algorithm 1 Insert new element into meet-semilattice

```
1: procedure INSERT(elem)
2:     n, new ← FIND_OR_CREATE_NODE(elem)
3:     if new then
4:         Modified_Nodes.insert(n)
5:         INSERT_NODE(Root, n)                    ▷ Root.elem = ⊤
6:         for n' ∈ Modified_Nodes do
7:             COMPUTE_CARDINALITY(n')
```

Algorithm 2 Update DAG representing meet-semilattice

```
1:  procedure INSERT_NODE(parent, n)
2:      Γ ← {}
3:      for child ∈ parent.children do
4:          if child.elem ⊆ n.elem then
5:              Γ.insert(child)
6:          else if n.elem ⊆ child.elem then
7:              INSERT_NODE(child, n)
8:              return
9:          else
10:             e′ ← n.elem ∩ child.elem
11:             if e′ is not empty then
12:                 n′, new ← FIND_OR_CREATE_NODE(e′)
13:                 Γ.insert(n′)
14:                 if new then
15:                     Modified_Nodes.insert(n′)
16:                     INSERT_NODE(child, n′)
17:     parent.children.insert(n)
18:     Modified_Nodes.insert(parent)
19:     max_children ←
20:         {c ∈ Γ | ∀c′ ∈ Γ : (c.elem ⊆ c′.elem → c = c′)}
21:     for max_child ∈ max_children do
22:         parent.children.erase(max_child)
23:         n.children.insert(max_child)
```

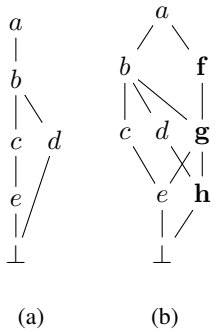
amortized, if so desired. Note that the set of generated PECs is invariant under the insertion order of elements.

Unlike SAT/SMT solvers or BDD-based solutions—which can prove that a PECs is non-empty by finding a witness—#PEC computes PEC-cardinalities instead. In the worst case, this computation is quadratic in the size N of the DAG where N can be exponential in the number of input match conditions [28].

E. Answering Operator Questions via PEC-based Queries

When applying a PEC-based formal network analysis technique to a set of packet headers decribed by a logical query, it is necessary to convert the query into a set of PECs. In #PEC, we perform this conversion as follows.

Foremost, we assume that each logical query is a Boolean combination of logical predicates that have the same element type (Figure 7) as the match conditions in the meet-semilattice.



(a)

(b)

	DESTINATION	PROTO
a	0.0.0.0/0	ANY
b	210.4.214.0/23	ANY
c	210.4.214.0/24	ANY
d	210.4.215.0/24	ANY
e	210.4.214.0/24	ICMP
f	**0.0.0.0/0**	**ICMP**
g	**210.4.214.0/23**	**ICMP**
h	**210.4.215.0/24**	**ICMP**

(c)

Fig. 8: Two meet-semilattices (a and b) for different subsets of (c) 2-dimensional match conditions

Algorithm 3 Compute and/or update cardinality of PECs

```
1:  procedure COMPUTE_CARDINALITY(n)
2:      queue ← [n]; visited ← {n}
3:      n.cardinality ← cardinality(n.elem)
4:      while queue is not empty do
5:          n′ ← queue.dequeue()
6:          for child ∈ n′.children do
7:              if child ∉ visited then
8:                  if child ∈ Modified_Nodes then
9:                      COMPUTE_CARDINALITY(child)
10:                 visited.insert(child)
11:                 n.cardinality ←
12:                     n.cardinality − child.cardinality
13:                 queue.enqueue(child)
14:     Modified_Nodes.erase(n)
```

If a predicate in the query is not present in the meet-semilattice, we first insert it via Algorithm 1.

Under these assumptions, a query is then converted into a set of PECs by invoking Algorithm 4, a recursive function on the logical structure of the input query. We remark that our last assumption ensures that line 3 in Algorithm 4 always finds a node in the DAG, i.e., n is never null. As part of our case study (§ IV-C), we give an example of a query conversion.

Algorithm 4 Convert a query to a set of PECs

```
1:  function CONVERT_TO_PECS(query)
2:      if query is an element type then
3:          n ← FIND_NODE(query)
4:          return SUBTREE(n)
5:      else if ∃ g : query = ¬g then
6:          Universe ← SUBTREE(Root)
7:          return Universe − CONVERT_TO_PEC(g)
8:      else if ∃ g, h : query = g ∧ h then      ▷ 'g ∨ h' is similar
9:          G ← CONVERT_TO_PECS(g)
10:         H ← CONVERT_TO_PECS(h)
11:         return G ∩ H                          ▷ Use '∪' for 'g ∨ h'
```

F. Minimality of PECs

Next, we show that the set of non-empty PECs produced by #PEC form atomic predicates in the following strict sense:

Definition (**Atomic Predicates [27]**). Let $\mathfrak{M}$ be a set of predicates, each of which represents a match condition of a firewall or forwarding rule. Then $\mathfrak{M}$'s set of atomic predicates, written $\mathsf{A}(\mathfrak{M}) = \{\alpha_1, \ldots, \alpha_k\}$, satisfies the following:

1) for all $i \in \{1, \ldots, k\}$, $\alpha_i \neq \mathbf{false}$;
2) $(\bigvee_{i=0}^{k} \alpha_i) = \mathbf{true}$;
3) $\alpha_i \wedge \alpha_j = \mathbf{false}$ for all $i, j \in \{1, \ldots, k\}$ such that $i \neq j$;
4) Each match condition p in $\mathfrak{M}$, where $p \neq \mathbf{false}$, is equal to the disjunction of some subset of atomic predicates:
$$p = \bigvee_{i \in S(p)} \alpha_i \text{ where } S(p) \subseteq \{0, \ldots, k\};$$
5) k is the minimal number such that the set $\mathsf{A}(\mathfrak{M}) = \{\alpha_1, \ldots, \alpha_k\}$ satisfies the above four conditions.

Yang and Lam show that the set of atomic predicates is unique [27], which they compute using BDDs. Given a set

of rule match conditions that can be expressed using element types (Figure 7), the next theorem shows how to compute this unique and minimal set through a fundamentally different algorithm that uses lattice theory and model counting.

Theorem (**Optimality of #PEC**). Given as input a set of match conditions $\mathfrak{M}$ of an element type, the set of non-empty PECs constructed by #PEC forms atomic predicates $\mathsf{A}(\mathfrak{M})$.

Proof. The proof can be found in [38]. □

Put simply, #PEC's output is as good as APV's [27]. We re-emphasize two important assumptions: (i) match conditions must be expressed as element types (Figure 7), and (ii) the same inputs are supplied to both tools. Condition (ii) is violated when, say, APV is allowed to pre-process forwarding rules by aggregating match conditions that are associated with the same output port. #PEC does not perform such port-aggregation pre-processing step, a design decision we made because the partitioning of the packet header space would need to be re-computed every time some port is changed.

IV. Evaluation

For our study, we experimentally evaluate different implementations of PEC-construction schemes (§ IV-A), namely: #PEC, APV, ddNF and Veriflow where possible. We evaluate both the SAT/SMT and BDD-based solutions of the PEC-emptiness problem as well as our counting method. As part of our evaluation, we analyze firewalls and forwarding rules collected from a variety of sources (§ IV-B). Using these datasets, we uncover real-world cases where ddNF raises false alarms and misses errors (§ IV-C), which #PEC successfully avoids. We then evaluate #PEC's performance (§ IV-D).

A. Implementations

Here we outline our implementations of PEC-emptiness checks (§ IV-A1), and APV as well as #PEC (§ IV-A2).

1) Three PEC-emptiness checking procedures: In addition to implementing #PEC's counting method, we want to evaluate the SAT/SMT and BDD-based solutions to the PEC-emptiness problem that use propositional logic to precisely encode when a PEC is empty. Their symbolic encoding works as follows.

Let x be a match condition of a rule, and denote with C_x the set of direct children of element x in the DAG constructed by #PEC (recall § III-C). By construction, every child c in C_x is a strict subset of x, i.e., $c \subset x$. We emphasize that, for efficiency reasons, we only consider the direct children of x, so all children in C_x are mutually incomparable, i.e., for any child c and c' in C_x, neither $c \subset c'$ nor $c' \subset c$. To implement the propositional logic PEC-emptiness solutions, we construct the following Boolean formula: $x \wedge \neg \left(\bigvee_{c \in C_x} c\right)$—equivalently, $x \wedge \neg c_1 \wedge \neg c_2 \wedge \ldots \wedge \neg c_n$ where c_1, c_2, …, c_n are x's direct children in C_x. For checking the formula's satisfiability, we use a SAT/SMT solver or construct a BDD, as detailed next.

Dataset	Short Description
REANNZ-IP [39], [40]	1,159 distinct IP prefixes
REANNZ-Full [39], [40]	1,170 OpenFlow rules
Azure-DC [25]	2,942 ternary 128-bit vectors
Berkeley-IP [14], [41]	584,944 distinct IP prefixes
Stanford-IP [11]	197,828 distinct IP prefixes
Stanford-Full [11]	2,732 ternary 128-bit vectors
Diekmann [33]	Thousands of 8-tuples

Fig. 9: Summary of datasets

Our BDD implementation uses the C++ BuDDy library. We set the intial node number and cache size by manual tuning and choosing values that yield better results. In the case of the SAT/SMT implementation, we call Z3 [42]. To avoid additional parsing overhead, we use Z3's C++ API to construct the Boolean formulas, rather than using the more standard SMT-LIB [43] or DIMACS format for SAT solvers.

As part of the Boolean encoding of element types (recall § III-B), we convert `tbv<N>` elements into `N` Boolean variables, one for each non-$*$ ternary bit. For the conversion of `set<T>`, which is implemented using bitsets, we encode the disjunction of the indexes of the set bits using $\lceil \log_2 K \rceil$ Boolean variables where K is the length of the bitset. For the `tuple<E1,...,Ek>` encoding, we designate $b = b_1 + \ldots + b_k$ Boolean variables where b_j is the number of Boolean variables needed to represent $\mathtt{E}_j$. The final encoding is the conjunction of the Boolean encoding of each tuple coordinate.

2) Implementation of APV, ddNF and #PEC: To rigorously evaluate the performance of our tool against others, we implement a version of APV and #PEC within the same framework: we opted for Z3 [42]. Our re-implementation of APV applies the same optimizations as proposed in [28]. We do not have to re-implement ddNF, since it is already available as an open-source module in Z3. Similar to ddNF, our implementation of #PEC leverages Z3's highly optimized TBV implementation. We implement the other element types as a C++11 library, which we describe in more detail in [38].

B. Datasets

Our evaluation uses 64 different datasets, extracted from five independently collected routing tables and firewalls collections [11], [14], [25], [33], [39]. Figure 9 categorizes our datasets according to their source of origin. Each dataset is encoded as a list of rule match conditions of a specific element type (recall Figure 7 in § III-B). Since ddNF only supports TBVs, we encode the match conditions in our datasets as TBVs whenever possible, which is feasible for all datasets except the 'Diekmann' dataset, as described below. Irrespective of the element type, we ensure all match conditions per dataset are unique, since duplicates could be processed with almost zero cost. We describe each category of dataset in turn.

a) REANNZ: The REANNZ-Full dataset [39] contains more than a thousand OpenFlow rules, extracted from a single routing table that was used in the Cardigan deployment [40]. The OpenFlow rules in the REANNZ dataset use the following header fields: source and destination MAC addresses, ether-type, source and destination IP address, IP protocol field, and source and destination TCP ports. We convert each match condition in the rules to a ternary 216-bit vector. From the

full dataset, we extract REANNZ-IP which contains only IP prefixes, but also encoded as TBVs.

b) Berkeley-IP: The Berkeley-IP dataset originates from [14] where IPv4 prefixs from the RouteViews project [41] were evaluated in the context of the UC Berkeley campus network topology. Our dataset focuses only on the IPv4 prefixes, which we encode as 32-bit long TBVs.

c) Azure-DC: The Azure-DC dataset [25] contains synthetic FIBs that simulate Azure-like data centers as deployed by Microsoft at that time. It contains a total of nearly 3000 match conditions, each of which is a ternary 128-bit vector.

d) Stanford: The Stanford dataset originates from Stanford's backbone network [11], which contains configurations of sixteen Cisco routers. For each router, we generate its transfer function [11] which models the static behavior of the router (including forwarding and ACLs). We then use the match conditions in the transfer function, encoded as ternary 128-bit vectors, to produce a dataset for that router (e.g Stanford-Full/boza). To measure the effect of analyzing a network containing all sixteen routers, we also combine all sixteen datasets into a single one, Stanford-Full, which contains a total of 2,732 unique ternary 128-bit vectors. In our Stanford-IP dataset, we extract the IP prefixes directly from the raw router configurations, thereby avoiding the IP prefix compression feature in HSA's transfer functions. As a result, our Stanford-IP datasets are significantly larger than the datasets used in the evaluation of HSA [11] and ddNF [28].

e) Diekmann: The Diekmann datasets contains match conditions from real-world Linux iptables rule-sets [33]. We parse the following packet header matching fields: source and destination IP prefix, source and destination port, protocol, connection state, input and output interface. We encode these as a mixture of TBVs and regular bitsets, which we combine into 8-tuples. We ignore wildcard characters for interfaces. We simplify each original iptables rule-set through a pre-processor that propagates match conditions along iptables chains in a depth-first manner, similar to function inlining. This essentially flattens a multi-chain iptables configuration into a list of match conditions without jumps and returns, so they conform to the same format as the other datasets.

C. Case Study

In this subsection, we describe real-world cases of imprecision in ddNF, all of which #PEC handles successfully. Due to space, we only illustrate a few examples (in our full study, we encountered over three dozen cases of imprecision in ddNF).

To begin with, ddNF misses 35 shadowed rules in the REANNZ dataset. We found that ddNF misses four shadowed rules in the Stanford datasets, one in each of the 'soza', 'sozb', 'yoza', and 'yozb' Cisco routers. Furthermore, in the Stanford dataset, ddNF fails to check that every packet whose destination IP address matches the IP prefix `171.64.79.160/24` is forwarded from router 'yozb' to router 'yoza'. For this query, ddNF wrongly reports that some packets with such a destination IP address are dropped. The slightly simplified relevant rules in the dataset for the 'yozb' router are as follows:

```
Destination=171.64.79.160/28 => yoza
Destination=171.64.79.176/28 => yoza
Destination=171.64.79.128/27 => yoza
Destination=171.64.79.192/27 => yoza
Destination=171.64.79.224/27 => yoza
Destination=171.64.79.0/25   => yoza
Destination=171.64.79.0/24   => DROP
```

Here, ddNF produces this wrong result, because the union of IP prefixes that forward to 'yoza' equals the IP prefix of the last rule that drops packets: the match condition of the last rule, therefore, is encoded as a singleton set that contains an empty PEC—the same underlying cause as described in § II-C.

As a more complicated example, consider the following human-readable form of the OpenFlow rules part in the REANNZ dataset (slightly simplified to help with readability), ordered from highest to lowest priority:

```
Protocol=ICMP              => Controller
Destination=210.4.214.0/24 => Port 1
Destination=210.4.215.0/24 => Port 1
Destination=210.4.214.0/23 => Port 2
Destination=ANY            => DROP
```

The match conditions associated with these OpenFlow rules induce the DAG shown in Figure 8. Suppose a network operator wants to answer the following query:

> "Are all non-`ICMP` packets destined to IP prefix `210.4.214.0/23` sent to Port 1?"

Formally, this query is a Boolean combination of the form $\texttt{210.4.214.0/23}, \texttt{ANY} \wedge \neg(\texttt{0.0.0.0/0}, \texttt{ICMP})$ where the first and second conjunct are elements b and c in the DAG in Figure 8, respectively. Using Algorithm 4 in § III-E, we convert the query into the set of PECs $\{B, C, D\}$. Since B is a PEC associated with a rule that outputs the packet at port 2, ddNF concludes that the above property is violated. However, ddNF's verification result is incorrect: since B is empty no such violation can be realized in the actual network. #PEC correctly detects that the property holds. For the sake of brevity, we omit the discussion of five other, but similar, examples of imprecision in the REANNZ dataset.

D. Performance Evaluation

We evaluate #PEC's performance along two dimensions, namely: (i) time and memory usage to construct #PEC's meet-semilattice; (ii) time and memory usage for detecting empty PECs. We discuss our results in turn.[4]

1) PEC-construction: We compare #PEC to APV, and Z3's implementation of ddNF. We ensure that every implementation benefits from the same optimizations (§ IV-A2). We find that #PEC consistently outperforms APV and ddNF in Z3 where, on larger datasets, the speed-up is more than $10\times$. For example, on the Azure-DC dataset, our re-implementation of #PEC in Z3 is approximately $30\times$ faster than ddNF. APV times out on the Berkeley-IP dataset after 10 hours, whereas #PEC completes the PEC-construction in 45 minutes. We include in #PEC's total run-time the time it takes to check PEC-emptiness, when comparing #PEC and APV. For this

[4]All experiments are run on a Linux machine with an Intel Xenon CPU ES-1660 3.30GHz and 32GB DDR3 1333MHz RAM.

Dataset	Insertions	PECs	Empty PECs	Atomic Preds.	PEC-construction time (s)		PEC-emptiness check (s)			APV (s)	Memory (MB)			
					Z3 ddNF	#PEC	BDD	SAT	Card.		BDD	SAT	Card.	APV
REANNZ-IP	1,159	1,160	25	1,135	<1ms	<1ms	0.016	0.414	<1ms	0.001	6	6	3	5
REANNZ-Full	1,170	12,783	275	12,508	0.112	0.009	2	9	0.018	3	14	26	9	10
Azure-DC	2,942	5,096,869	10,450	5,086,419	3301	121	20112	47829	30	25669	4,429	5,797	2,365	2,517
Berkeley-IP	584,944	584,945	29,813	Timeout	Timeout	2709	1553	460	0.515	Timeout	302	701	227	Timeout
Stanford-IP/soza	184,682	184,682	4,841	179,841	471	347	7	82	0.119	4951	102	251	69	49
Stanford-IP/yoza	4,746	4,746	3	4,743	<1ms	<1ms	0.076	2	0.002	2	8	9	4	6
Stanford-IP/All	197,828	197,828	4,874	192,954	266	199	19	89	0.156	5149	122	265	85	53
Stanford-Full/soza	524	16,764	81	16,683	0.056	<1ms	0.668	9	0.024	2	18	19	10	13
Stanford-Full/yoza	507	60,363	231	60,132	5	0.17	4	38	0.17	20	46	65	31	28
Stanford-Full/All	2,732	1,176,095	48,906	1,127,189	560	28	692	1958	4	2314	895	1,077	544	439
Diekmann/G	5,321	889,646	40	889,606	-	39	413	4729	10	2385	3,843	3,854	3,924	608
Diekmann/J	6,004	1,058,897	56	1,058,841	-	71	486	5654	13	2936	4,558	4,573	4,656	700
Diekmann/K	3,242	400,911	257	400,654	-	18	157	2084	3	732	1,997	2,006	2,031	233
Diekmann/P	578	492,378	4	492,374	-	47	168	1837	4	635	1,563	1,573	1,606	324
Diekmann/Q	307	4,626	38	4,588	-	0.087	0.763	17	0.016	0.94	21	29	18	7

Fig. 10: Evaluation results for a subset of datasets. The full experimental results can be found in our technical report [38].

comparison, we use the 39 datasets in which either APV or #PEC runs for more than $100\,ms$, excluding the Berkeley-IP dataset where APV times out. In 95% of these 39 cases, despite #PEC's PEC-emptiness check, #PEC is at least $10\times$ faster than APV, and 25% of this time #PEC's speed-up is at least $100\times$. On average, #PEC is at least $80\times$ faster than APV. Finally, APV and #PEC's memory usage averages out to be the same across these datasets. Figure 10 shows parts our experimental results, see [38] for the full details.

The fact that #PEC outperforms APV is expected, since #PEC eliminates the per-bit overhead of BDDs. The performance difference between #PEC and Z3's implementation of ddNF, in turn, can be explained in terms of the number of intersection and subset operations required to insert a new match condition into their respective data structure: their total run-time is proportional to these operations. For example, in the Stanford-Full dataset, #PEC requires 0.4 million whereas ddNF in Z3 takes 8 million such operations, a $20\times$ improvement. #PEC's improvement over Z3's implementation of ddNF are similar on the other datasets.

2) PEC-emptiness checking: We compare #PEC's counting method to the SAT/SMT and BDD-based solutions to checking PEC-emptiness. We evaluate the performance of PEC-emptiness checking using the 24 datasets in which #PEC runs for more than $100\,ms$. We perform the PEC-emptiness check after the PEC-construction has completed. We take extra precautions in our implementations to ensure a fair comparison (§ IV-A1). Figure 10 shows that #PEC's counting method significantly outperforms the SAT/SMT and BDD-based approaches: #PEC achieves at least a $10\times$ speed-up compared to the SAT/SMT and BDD-based approach in over 95% of cases. On average, #PEC is at least $500\times$ and $200\times$ faster than the SAT/SMT and BDD-based approaches, respectively.

To understand why #PEC's cardinality-based approach outperforms the SAT/SMT and the BDD-based approaches, reconsider the IP prefixes in § II-C. Representing x, y, and z in propositional logic requires 19, 19, and 18 variable assignments respectively, corresponding to their non-wildcard bits. Just encoding $Z = z - (x \cup y)$ in SAT requires near 60 logic gates, excluding the task of checking satisfiability. Representing the predicates using BDDs requires the same number of BDD nodes. Assuming logical BDD operations are linear in their operand size, computing Z at least requires CPU cycles proportional to the cumulative size of the three BDDs. On the other hand, the cardinality of each predicate in the example fits into a single machine word. We need only 2 arithmetic CPU operations to compute the cardinality of Z (i.e $|z| - |x| - |y|$), and then check if it is zero. While in theory there are still near 60 operations performed (at the bit level), #PEC harnesses the computing power of ALUs to finish the operations in fewer CPU cycles. For example, in the Stanford-Full dataset where each node in the DAG has 3 children and 12 nodes in its subtree on average, the BDD-based approach requires 3×128 low-level BDD operations on average (each spanning tens of CPU instructions). By contrast, our cardinality-based approach needs at most 3 ALU operations for each subtraction. So #PEC should be at least $(3{\times}128)/(12{\times}3) \approx 10\times$ faster than the BDD-based approach, and our experiments show indeed at least a $127\times$ speed-up.

3) Comparison with Veriflow: We compare #PEC to the original implementation of Veriflow [44]. Since that implementation of Veriflow only supports a restricted form of OpenFlow rules where arbitrary per-field bitmasks are disallowed [35], it cannot analyze the majority of our datasets. We therefore restrict our experiments with Veriflow to a simplified version of the Stanford-Full dataset. We use the default packet header field ordering in Veriflow. We ask Veriflow to only find 'Equivalence Classes' (ECs), rather than each EC's forwarding graph. In this restricted setting, Veriflow takes $41\,s$ to create 3,778,324 ECs, using $1\,GB$ of memory. Despite #PEC's support for arbitrary bitmasks, it is still more efficient than Veriflow, in both time ($30\,s$) and space ($0.5\,GB$): specifically, #PEC constructs only 1,066,645 PECs in $27\,s$, and finds 44,418 empty PECs in 3s.

E. Discussion: Importance of Empty PECs

We showed that ddNF's non-minimality of PECs is due to PECs that are empty. In our case study (§ IV-C), we exemplified real-word cases where empty PECs lead to wrong

analysis results, which are very likely to hinder technology adoption [45]. We emphasize that we only gave illustrative examples; our list is not exhaustive, and it includes cases where ddNF misses errors. In practice, therefore, ddNF is only as fast as the slowest decision procedure needed to sanity check its results, a fundamental limitation. By contrast, #PEC's analysis is correct by construction (§ III-F), and its performance is *not* dependent on BDDs or SAT/SMT solvers, which are orders of magnitude slower in finding empty PECs (§ IV-D2).

V. Related Work

Similar to APV [27] and ddNF [28], #PEC has many potential applications in the field of network correctness. The literature in this field is vast and includes BGP configuration checking (e.g., [7], [46]–[53]), ACL misconfiguration detection (e.g., [54], [55]), firewall checking (e.g., [17], [18], [21], [56]), SDN verification (e.g., [20], [23], [57], [58]), testing (e.g., [2], [59]–[62]), debugging (e.g., [63], [64]), differential analysis (e.g., [65]), concurrency analysis (e.g [66], [67]), automatic repair (e.g., [68]–[70]), synthesis (e.g. [71]–[73]), programming languages (e.g. [74]–[78]), safe network updates (e.g., [79]–[82]), data plane checking (e.g., [10], [11], [16], [25]), real-time checkers [12], [14], [30], [83], and more general network analyses (e.g., [6], [8], [9], [19], [84], [85]) together with suitable levels of abstractions (e.g., [86], [87]).

Our work is most closely related to ddNF [28], APV [27], Delta-net [14] and Veriflow [30], since they all partition packet headers somehow. However, these formal network analysis tools also differ in important ways, as summarized by Figure 11 using characteristics, which are divided into three blocks: (i) whether the analysis is precise PECs remain the same when the priority or output port of rules change; (ii) common kinds of match conditions of practical interest; and (iii) finally, attributes of the underlying algorithms. We discuss each of these tools in turn:

a) ddNF [28]: #PEC achieves precision when ddNF cannot. Furthermore, we have shown that #PEC can detect shadowed rules, whereas ddNF cannot in general. As a result, #PEC can verify equivalence of forwarding tables, whereas ddNF cannot. We have also shown that #PEC is more expressive than ddNF in the kind of match conditions supported, e.g., iptables rule-sets. The DAG produced by #PEC can be shown to be isomorphic to ddNF's, but #PEC is up to $30\times$ faster than ddNF in constructing it (§ IV).

Feature / Characteristic	#PEC	ddNF	APV	Delta-net	Veriflow
Precise network analysis	●	○	●	●	●
Rule priority & action invariant	●	●	○	●	●
Match conditions with bit masks	●	●	●	○	◐
Wildcard on packet header fields	●	●	●	○	●
Match conditions with sets of values	●	○	●	○	○
Negation on packet header fields	●	○	●	○	○
Range filters beyond IP prefixes	●	○	●	●	○
PEC-cardinalities	●	○	○	○	○
Negation-free PEC-construction	●	●	○	●	○
Canonical PEC-representation	●	●	●	●	○
Minimal and unique set of PECs	●	○	●	○	○

Fig. 11: Feature comparison of closest related work

b) APV [27]: APV produces PECs in the form of atomic predicates, the smallest partition of the packet header space. #PEC also constructs the fewest PECs (§ III-F), and it does so $10\times$ faster than APV (§ IV-D). Through an optional pre-processing step, APV may further reduce the problem size by aggregating match conditions per output port. However, when the priority or the output port of a rule changes, so would atomic predicates for the *entire* network then. By contrast, #PEC and ddNF only create PECs that are invariant under changes to the priority of rules and/or their actions. As explained in the introduction, APV's PEC-construction algorithm is not negation-free, explaining why it relies on BDDs [31], [32], whereas neither #PEC nor ddNF do.

c) Delta-net [14]: Delta-net is specifically designed for real-time analysis of large-scale BGP-controlled data centers [88]. It only supports forwarding rules that match packets based on ranges, possibly with arbitrary lower and upper bounds (unlike ddNF). Due to its limited expressiveness, Delta-net achieves quasi-linear time complexity, whereas ddNF and #PEC's higher expressiveness has an exponential worst-case time complexity. In addition, unlike ddNF and #PEC, Delta-net's run-time is independent of the order in which match conditions are inserted. Delta-net exploits the fact that the negation of a range can be efficiently computed, so its PEC-construction scheme is not negation-free.

d) Veriflow [30]: Veriflow uses a multi-dimensional trie data structure to represent PECs. To do so efficiently, Veriflow imposes assumption that prevent it from analyzing most multi-dimensional match conditions in our datasets (§ IV-D). The PECs constructed by Veriflow depend on the order of levels in the multi-dimensional tries, which can render its memory usage and run-time performance unpredictable.

VI. Concluding Remarks

Our case study (§ IV-C) and experiments (§ IV-D) reveal the tension between precision, expressiveness and performance: Veriflow and ddNF impose assumptions that prevent them from analyzing most of our dataset, and ddNF's analysis is imprecise. By contrast, APV is very expressive and precise but significantly slower than Veriflow and ddNF. Our work offers a new lattice-theoretical, algorithmic framework for formal network analysis that is expressive, precise and fast, thereby addressing a longstanding problem that has spanned three generations of formal network analysis tools.

To achieve this, we identified and efficiently solved the coNP-hard problem [38] of deciding whether a PEC is empty or not. We showed that both SAT/SMT and BDD-based solutions to this problem perform poorly. This lead to #PEC, which uses a model counting method that is $10-100\times$ faster than the SAT/SMT and BDD-based solutions. In addition, #PEC constructs the unique minimal number of PECs, and it does so $10\times$ faster than APV's atomic predicates.

REFERENCES

[1] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar, "Making middleboxes someone else's problem: Network processing as a cloud service," in *SIGCOMM*, 2012.

[2] H. H. Liu, Y. Zhu, J. Padhye, J. Cao, S. Tallapragada, N. P. Lopes, A. Rybalchenko, G. Lu, and L. Yuan, "CrystalNet: Faithfully emulating large production networks," in *SOSP*, 2017.

[3] A. Wool, "A quantitative study of firewall configuration errors," *Computer*, vol. 37, no. 6, Jun. 2004.

[4] H. Hamed and E. Al-Shaer, "Taxonomy of conflicts in network security policies," *IEEE Comm. Mag.*, vol. 44, no. 3, Mar. 2006.

[5] A. Wool, "Trends in firewall configuration errors: Measuring the holes in swiss cheese," *IEEE Internet Computing*, vol. 14, no. 4, Jul. 2010.

[6] A. Fogel, S. Fung, L. Pedrosa, M. Walraed-Sullivan, R. Govindan, R. Mahajan, and T. Millstein, "A general approach to network configuration analysis," in *NSDI*, 2015.

[7] A. Gember-Jacobson, R. Viswanathan, A. Akella, and R. Mahajan, "Fast control plane analysis using an abstract representation," in *SIGCOMM*, 2016.

[8] S. K. Fayaz, T. Sharma, A. Fogel, R. Mahajan, T. D. Millstein, V. Sekar, and G. Varghese, "Efficient network reachability analysis using a succinct control plane representation," in *OSDI)*, 2016.

[9] R. Beckett, A. Gupta, R. Mahajan, and D. Walker, "A general approach to network configuration verification," in *SIGCOMM*, 2017.

[10] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King, "Debugging the data plane with Anteater," in *SIGCOMM*, 2011.

[11] P. Kazemian, G. Varghese, and N. McKeown, "Header space analysis: Static checking for networks," in *NSDI*, 2012.

[12] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte, "Real time network policy checking using header space analysis," in *NSDI*, 2013.

[13] C. Zhongbo, "Veriflow system analysis and optimization," Master's thesis, University of Illinois Urbana-Champaign, 2014.

[14] A. Horn, A. Kheradmand, and M. Prasad, "Delta-net: Real-time network verification using atoms," in *NSDI*, 2017.

[15] J. Backes, S. Bayless, B. Cook, C. Dodge, A. Gacek, A. J. Hu, T. Kahsai, B. Kocik, E. Kotelnikov, J. Kukovec, S. McLaughlin, J. Reed, N. Rungta, J. Sizemore, M. A. Stalzer, P. Srinivasan, P. Subotic, C. Varming, and B. Whaley, "Reachability analysis for AWS-based networks," in *CAV*, 2019.

[16] G. G. Xie, J. Zhanm, D. A. Maltz, H. Zhang, A. Greenberg, G. Hjalmtysson, and J. Rexford, "On static reachability analysis of IP networks," in *INFOCOM*, 2005.

[17] A. Jeffrey and T. Samak, "Model checking firewall policy configurations," in *POLICY*, 2009.

[18] T. Nelson, C. Barratt, D. J. Dougherty, K. Fisler, and S. Krishnamurthi, "The Margrave tool for firewall analysis," in *LISA*, 2010.

[19] E. Al-Shaer and S. Al-Haj, "FlowChecker: Configuration analysis and verification of federated OpenFlow infrastructures," in *SafeConfig*, 2010.

[20] S. Son, S. Shin, V. Yegneswaran, P. A. Porras, and G. Gu, "Model checking invariant security properties in OpenFlow," in *ICC*, 2013.

[21] S. Zhang, A. Mahmoud, S. Malik, and S. Narain, "Verification and synthesis of firewalls using sat and qbf," in *ICNP*, 2012.

[22] K. Jayaraman, N. Bjørner, G. Outhred, and C. Kaufman, "Automated analysis and debugging of network connectivity policies," Microsoft Research, Tech. Rep., 2014.

[23] T. Ball, N. Bjørner, A. Gember, S. Itzhaky, A. Karbyshev, M. Sagiv, M. Schapira, and A. Valadarsky, "VeriCon: Towards verifying controller programs in software-defined networks," in *PLDI*, 2014.

[24] F. A. Maldonado-Lopez, E. Calle, and Y. Donoso, "Detection and prevention of firewall-rule conflicts on software-defined networking," in *RNDM*, 2015.

[25] N. P. Lopes, N. Bjørner, P. Godefroid, K. Jayaraman, and G. Varghese, "Checking beliefs in dynamic networks," in *NSDI*, 2015.

[26] H. Hu, W. Han, G.-J. Ahn, and Z. Zhao, "FLOWGUARD: Building robust firewalls for software-defined networks," in *HotSDN*, 2014.

[27] H. Yang and S. S. Lam, "Real-time verification of network properties using atomic predicates," in *ICNP*, 2013.

[28] N. Bjørner, G. Juniwal, R. Mahajan, S. A. Seshia, and G. Varghese, "ddNF: An efficient data structure for header spaces," in *HVC*, 2016.

[29] R. McGeer, "Verification of switching network properties using satisfiability," in *ICC*, 2012.

[30] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey, "VeriFlow: Verifying network-wide invariants in real time," in *NSDI*, 2013.

[31] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Trans. Comput.*, vol. 35, no. 8, pp. 677–691, Aug. 1986.

[32] D. E. Knuth, *The Art of Computer Programming, Volume 4, Fascicle 1: Bitwise Tricks & Techniques; Binary Decision Diagrams*, 12th ed. Addison-Wesley, 2009.

[33] C. Diekmann, J. Michaelis, M. Haslbeck, and G. Carle, "Verified iptables firewall analysis," in *IFIP Networking*, 2016.

[34] B. A. Davey and H. A. Priestley, *Introduction to Lattices and Order*, 2nd ed. Cambridge University Press, 2002.

[35] A. Khurshid, personal communication, Aug. 2019.

[36] D. G. Kourie, S. Obiedkov, B. W. Watson, and D. van der Merwe, "An incremental algorithm to construct a lattice of set intersections," *Sci. Comput. Program.*, vol. 74, no. 3, Jan. 2009.

[37] A. Fog, "Instruction latencies, throughputs and micro-operation breakdowns for intel, amd and via cpus," https://www.agner.org/optimize/instruction_tables.pdf.

[38] A. Horn, A. Kheradmand, and M. R. Prasad, "A precise and expressive lattice-theoretical framework for efficient network verification," https://arxiv.org/abs/1908.09068, Tech. Rep., August 2019.

[39] N. Katta, O. Alipourfard, J. Rexford, and D. Walker, "Cacheflow: Dependency-aware rule-caching for software-defined networks," in *SOSR*, 2016.

[40] J. Stringer, D. Pemberton, Q. Fu, C. Lorier, R. Nelson, J. Bailey, C. N. Correa, and C. E. Rothenberg, "Cardigan: Sdn distributed routing fabric going live at an internet exchange," in *ISCC*, 2014.

[41] Route Views, http://www.routeviews.org/.

[42] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *TACAS*, 2008.

[43] C. Barrett, P. Fontaine, and C. Tinelli, "The SMT-LIB Standard: Version 2.6," Department of Computer Science, The University of Iowa, Tech. Rep., 2017, available at `www.SMT-LIB.org`.

[44] A. Khurshid and B. Godfrey, personal communication, Jan. 2019.

[45] C. Sadowski, E. Aftandilian, A. Eagle, L. Miller-Cushon, and C. Jaspan, "Lessons from building static analysis tools at google," *Commun. ACM*, vol. 61, no. 4, pp. 58–66, Mar. 2018.

[46] S. Prabhu, A. Kheradmand, B. Godfrey, and M. Caesar, "Predicting network futures with plankton," in *APNet*, 2017.

[47] T. G. Griffin and G. Wilfong, "An analysis of BGP convergence properties," in *SIGCOMM*, 1999.

[48] N. Feamster and H. Balakrishnan, "Detecting BGP configuration faults with static analysis," in *NSDI*, 2005.

[49] B. Quoitin and S. Uhlig, "Modeling the routing of an autonomous system with C-BGP," *IEEE Network*, vol. 19, no. 6, Nov. 2005.

[50] A. Wang, L. Jia, W. Zhou, Y. Ren, B. T. Loo, J. Rexford, V. Nigam, A. Scedrov, and C. Talcott, "FSR: Formal analysis and implementation toolkit for safe interdomain routing," *IEEE/ACM Transactions on Networking*, vol. 20, no. 6, Dec. 2012.

[51] K. Weitz, D. Woos, E. Torlak, M. D. Ernst, A. Krishnamurthy, and Z. Tatlock, "Formal semantics and automated verification for the border gateway protocol," in *NetPL*, 2016.

[52] S. K. Fayaz, T. Sharma, A. Fogel, R. Mahajan, T. Millstein, V. Sekar, and G. Varghese, "Efficient network reachability analysis using a succinct control plane representation," in *OSDI*, 2016.

[53] S. Prabhu, K. Y. Chou, A. Kheradmand, B. Godfrey, and M. Caesar, "Plankton: Scalable network configuration verification through model checking," in *NSDI*, 2020.

[54] L. Bauer, S. Garriss, and M. K. Reiter, "Detecting and resolving policy misconfigurations in access-control systems," *ACM Transactions on Information and System Security*, vol. 14, no. 1, Jun. 2011.

[55] K. Jayaraman, V. Ganesh, M. Tripunitara, M. Rinard, and S. Chapin, "Automatic error finding in access-control policies," in *CCS*, 2011.

[56] L. Yuan, J. Mai, Z. Su, H. Chen, C.-N. Chuah, and P. Mohapatra, "FIREMAN: A toolkit for firewall modeling and analysis," in *SP*, 2006.

[57] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford, "A NICE way to test openflow applications," in *NSDI*, 2012.

[58] L. Ryzhyk, N. Bjørner, M. Canini, J.-B. Jeannin, C. Schlesinger, D. B. Terry, and G. Varghese, "Correct by construction networks using stepwise refinement." in *NSDI*, 2017.

[59] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown, "Automatic test packet generation," in *CoNEXT*, 2012.

[60] D. Lebrun, S. Vissicchio, and O. Bonaventure, "Towards test-driven software defined networking," in *NOMS*, 2014.

[61] M. A. Chang, B. Tschaen, T. Benson, and L. Vanbever, "Chaos monkey: Increasing sdn reliability through systematic network destruction," in *SIGCOMM*, 2015.

[62] S. K. Fayaz, T. Yu, Y. Tobioka, S. Chaki, and V. Sekar, "BUZZ: Testing context-dependent policies in stateful networks," in *NSDI*, 2016.

[63] R. Beckett, X. K. Zou, S. Zhang, S. Malik, J. Rexford, and D. Walker, "An assertion language for debugging SDN applications," in *HotSDN*, 2014.

[64] C. Scott, A. Wundsam, B. Raghavan, A. Panda, A. Or, J. Lai, E. Huang, Z. Liu, A. El-Hassany, S. Whitlock, H. Acharya, K. Zarifis, and S. Shenker, "Troubleshooting blackbox SDN control software with minimal causal sequences," in *SIGCOMM*, 2014.

[65] T. Nelson, A. D. Ferguson, and S. Krishnamurthi, "Static differential program analysis for software-defined networks," in *FM*, 2015.

[66] J. Miserez, P. Bielik, A. El-Hassany, L. Vanbever, and M. Vechev, "SDNRacer: Detecting concurrency violations in software-defined networks," in *SOSR*, 2015.

[67] R. May, A. El-Hassany, L. Vanbever, and M. Vechev, "BigBug: Practical concurrency analysis for SDN," in *SOSR*, 2017.

[68] H. Hojjat, P. Rümmer, J. McClurg, P. Černỳ, and N. Foster, "Optimizing Horn solvers for network repair," in *FMCAD*, 2016.

[69] A. Gember-Jacobson, A. Akella, R. Mahajan, and H. H. Liu, "Automatically repairing network control planes using an abstract representation," in *SOSP*, 2017.

[70] Y. Wu, A. Chen, A. Haeberlen, W. Zhou, and B. T. Loo, "Automated bug removal for software-defined networks." in *NSDI*, 2017.

[71] R. Beckett, R. Mahajan, T. Millstein, J. Padhye, and D. Walker, "Don't mind the gap: Bridging network-wide objectives and device-level configurations," in *SIGCOMM*, 2016.

[72] A. El-Hassany, P. Tsankov, L. Vanbever, and M. Vechev, "Network-wide configuration synthesis," in *CAV*, 2017.

[73] R. Birkner, D. Drachlser-Cohen, L. Vanbever, and M. Vechev, "Net2Text: Query-Guided Summarization of Network Forwarding Behaviors," in *NSDI*, 2018.

[74] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: A network programming language," in *ICFP*, 2011.

[75] C. Schlesinger, M. Greenberg, and D. Walker, "Concurrent NetCore: From policies to pipelines," in *ICFP*, 2014.

[76] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker, "NetKAT: Semantic foundations for networks," in *POPL*, 2014.

[77] H. Kim, J. Reich, A. Gupta, M. Shahbaz, N. Feamster, and R. J. Clark, "Kinetic: Verifiable dynamic network control." in *NSDI*, 2015.

[78] A. Kheradmand and G. Rosu, "P4K: a formal semantics of P4 and applications," *CoRR*, vol. abs/1804.01468, 2018.

[79] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for network update," in *SIGCOMM*, 2012.

[80] L. Vanbever, J. Reich, T. Benson, N. Foster, and J. Rexford, "HotSwap: Correct and efficient controller upgrades for software-defined networks," in *HotSDN*, 2013.

[81] S. Vissicchio, L. Vanbever, L. Cittadini, G. G. Xie, and O. Bonaventure, "Safe update of hybrid SDN networks," *IEEE/ACM Transactions on Networking*, vol. 25, no. 3, Jun. 2017.

[82] T. D. Nguyen, M. Chiesa, and M. Canini, "Decentralized consistent updates in SDN," in *SOSR*, 2017.

[83] H. Zeng, S. Zhang, F. Ye, V. Jeyakumar, M. Ju, J. Liu, N. McKeown, and A. Vahdat, "Libra: Divide and conquer to verify forwarding tables in huge networks," in *NSDI*, 2014.

[84] E. Al-Shaer, W. Marrero, A. El-Atawy, and K. El-Badawi, "Network configuration in a box: towards end-to-end verification of network reachability and security," in *ICNP*, 2009.

[85] K. Jayaraman, N. Bjørner, G. Outhred, and C. Kaufman, "Automated analysis and debugging of network connectivity policies," Microsoft Research, Tech. Rep., 2014.

[86] G. D. Plotkin, N. Bjørner, N. P. Lopes, A. Rybalchenko, and G. Varghese, "Scaling network verification using symmetry and surgery," in *POPL*, 2016.

[87] R. Beckett, A. Gupta, R. Mahajan, and D. Walker, "Control plane compression," in *SIGCOMM*, 2018.

[88] P. Lapukhov, A. Premji, and J. Mitchell, "Use of BGP for routing in large-scale data centers," RFC 7938, Aug. 2016.