# Delta-net: Real-time Network Verification Using Atoms

Alex Horn
*Fujitsu Labs of America*

Ali Kheradmand
*University of Illinois at Urbana-Champaign*

Mukul R. Prasad
*Fujitsu Labs of America*

## Abstract

Real-time network verification promises to automatically detect violations of network-wide reachability invariants on the data plane. To be useful in practice, these violations need to be detected in the order of milliseconds, without raising false alarms. To date, most real-time data plane checkers address this problem by exploiting at least one of the following two observations: (i) only small parts of the network tend to be affected by typical changes to the data plane, and (ii) many different packets tend to share the same forwarding behaviour in the entire network. This paper shows how to effectively exploit a third characteristic of the problem, namely: similarity among forwarding behaviour of packets through *parts* of the network, rather than its entirety. We propose the first provably amortized quasi-linear algorithm to do so. We implement our algorithm in a new real-time data plane checker, Delta-net. Our experiments with SDN-IP, a globally deployed ONOS software-defined networking application, and several hundred million IP prefix rules generated using topologies and BGP updates from real-world deployed networks, show that Delta-net checks a rule insertion or removal in approximately 40 microseconds on average, a more than 10× improvement over the state-of-the-art. We also show that Delta-net eliminates an inherent bottleneck in the state-of-the-art that restricts its use in answering Datalog-style "what if" queries.

## 1 Introduction

In an evermore interconnected world, network traffic is increasingly diverse and demanding, ranging from communication between small everyday devices to large-scale data centres across the globe. This diversity has driven the design and rapid adoption of new open networking architectures (e.g. [41]), built on programmable network switches, which make it possible to separate the control plane from the data plane. This separation opens up interesting avenues for innovation [37], including rigorous analysis for finding network-related bugs. Finding these bugs *automatically* poses the following challenges.

Since the control plane is typically a Turing-complete program, the problem of automatically proving the presence and absence of bugs in the control plane is generally undecidable. However, the data plane, which is produced by the control plane, can be automatically analyzed. While the problem of checking reachability properties in the data plane is generally NP-hard [34], the problem becomes polynomial-time solvable in the restricted, but not uncommon, case where network switches only forward packets by matching IP prefixes [36]. This theoretical fact helps to explain why *real-time data plane checkers* [27, 25, 55] can often automatically detect violations of network-wide invariants on the data plane in the order of milliseconds, without raising false alarms.

To achieve this, most real-time network verification techniques exploit at least one of the following two observations: (i) only small parts of the network tend to be affected by typical changes to the data plane [27, 25], and (ii) many different packets often share the same forwarding behaviour in the entire network [27, 55]. Both observations are significant because the former gives rise to *incremental network verification* in which only changes between two data plane snapshots are analyzed, whereas the latter means that the analysis can be performed on a representative subset of network packets in the form of *packet equivalence classes* [27, 25, 55].

In spite of these advances, it is so far an open problem how to efficiently handle operations that involve swaths of packet equivalence classes [27]. This is problematic because it limits the real-time analysis of network failures, which are common in industry-scale networks, e.g. [13, 4]. Moreover, it essentially prevents data plane checkers from being used to answer "what if" queries in the style of recent Datalog approaches [17, 33] because these hypothetical scenarios typically involve checking the fate of many or all packets in the entire network.

To address this problem, this paper shows how to effectively exploit a third characteristic of data plane checking, namely: similarity among forwarding behaviour of packets through *parts* of the network, rather than its entirety. We show that our approach addresses fundamental limitations (§ 2) in the design of the currently most advanced data plane checker, Veriflow [27].

In this paper, we propose a new real-time data plane checker, Delta-net (§ 3). Instead of constructing *multiple forwarding graphs* for representing the flow of packets in the network [27], Delta-net incrementally transforms a *single edge-labelled graph* that represents *all* flows of packets in the entire network. We present the first provably amortized quasi-linear algorithm to do so (Theorem 1). Our algorithm incrementally maintains the lattice-theoretical concept of *atoms*: a set of mutually disjoint ranges through which it is possible to analyze all Boolean combinations of IP prefix forwarding rules in the network so that every possible forwarding table over these rules can be concisely expressed and efficiently checked. This approach is inspired by Yang and Lam's atomic predicates verifier [55]. While more general, their algorithm has a quadratic worst-case time complexity, whereas ours is quasi-linear. Since Delta-net's atom representation is based on lattice theory, it can be seen as an abstract domain (e.g. [11]) for analyzing forwarding rules. What makes our abstract domain different from traditional ones is that we dynamically refine its precision so that false alarms never occur.

For our performance evaluation (§ 4), we use data sets comprising several hundred million IP prefix rules generated from the UC Berkeley campus, four Rocketfuel topologies [49] and real-world BGP updates [46]. As part of our experiments, we run SDN-IP [31, 47], one of the most mature and globally deployed software-defined networking applications in the ONOS project [7, 42]. We show that Delta-net checks a rule insertion or removal in tens of microseconds on average, a more than 10× improvement over the state-of-the-art [27]. Furthermore, as an exemplar of "what if" scenarios, we adapt a link failure experiment by Khurshid et al. [27], and show that Delta-net performs several orders of magnitude faster than Veriflow [27]. We discuss related work in § 5.

**Contributions.** Our main contributions are as follows:

- Delta-net (§ 3), a new real-time data plane checker that incrementally maintains a compact representation about the flows of all packets in the network, thereby supporting a broader class of scenarios and queries.
- new realistic benchmarks (§ 4.2.2) with an open-source, globally deployed SDN application [47].
- experimental results (§ 4.3) that show Delta-net is more than 10× faster than the state-of-the-art in checking rule updates, while also making it now feasible to answer an expensive class of "what if" queries.

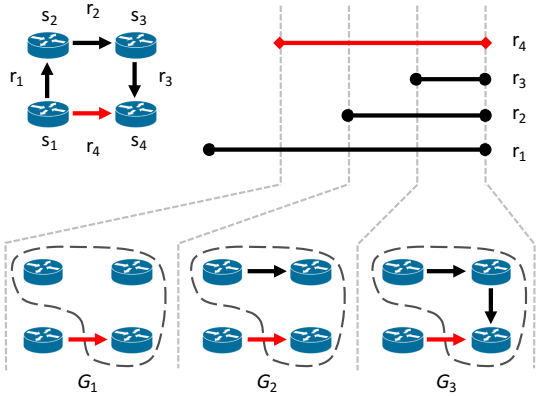



Figure 1: When rule $r_4$ (red edge) is inserted into switch $s_1$, Veriflow constructs at least three forwarding graphs, which significantly overlap with each other.

## 2 Overview of approach

In this section, we motivate and explain our approach through a simple example (§ 2.1) that illustrates how Delta-net differs from the currently most advanced data plane checker, Veriflow [27]. In addition to performance considerations, we follow three design goals (§ 2.2).

### 2.1 Example

Our example is based on a small network of four switches, shown in the upper-left corner of Figure 1. The data plane in this network is depicted as a directed graph in which each edge denotes an IP prefix forwarding rule. For example, rule $r_1$ in Figure 1 is assumed to determine the packet flow for a specific destination IP prefix from switch $s_1$ to $s_2$. Suppose the network comprises rules $r_1$, $r_2$ and $r_3$ (black edges) installed on switches $s_1$, $s_2$ and $s_3$, respectively. Since each rule matches packets by a destination IP prefix, we can represent each rule's match condition by an interval. For example, the IP prefix 0.0.0.10/31 (using the IPv4 CIDR format) corresponds to the half-closed interval $[10:12) = \{10, 11\}$ because 0.0.0.10/31 is equivalent to the 32-bit binary sequence that starts with all zeros and ends with $101*$ where $*$ denotes an arbitrary bit. Here, we depict the intervals of all three rules as parallel black lines (in an arbitrary order) in the upper-right half of Figure 1. The interpretation is that all three rules' IP prefixes overlap with each other.

Let us assume we are interested in checking the data plane for forwarding loops. Veriflow then first partitions all packets into *packet equivalences classes*, as explained next. Consider a new rule $r_4$ (red edge in Figure 1) to be installed on switch $s_1$ such that rule $r_4$ has a higher priority than the existing rule $r_1$ on switch $s_1$. As depicted in the upper half of Figure 1, the new rule $r_4$ overlaps with

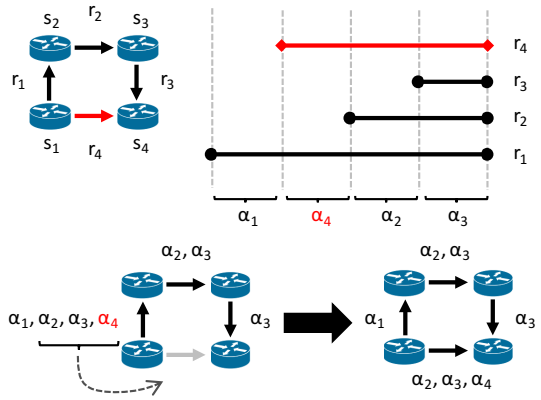



Figure 2: Rather than constructing *multiple* forwarding graphs that potentially overlap (Figure 1), Delta-net incrementally transforms a *single* edge-labelled graph.

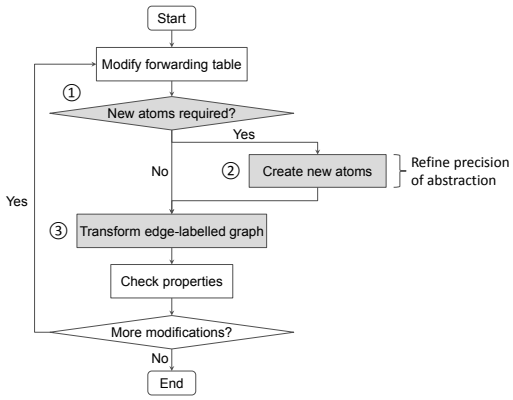



Figure 3: Delta-net incrementally maintains atoms, a family of sets of packets that can represent all Boolean combinations of IP prefix forwarding rules.

all the existing rules in the network, irrespective of the switch on which they are installed. Veriflow identifies at least three equivalence classes that are affected by the new rule, each of which denotes a set of packets that experience the same forwarding behaviour throughout the network. Here, we depict equivalence classes by three interval segments (gray vertical dashed lines).

For each equivalence class, Veriflow constructs a *forwarding graph* (denoted by $G_1$, $G_2$ and $G_3$ in Figure 1) that represent how packets in each equivalence class can flow through the network. Veriflow can now check for, say, forwarding loops by traversing $G_1$, $G_2$ and $G_3$. Note that the edge that represents the packet flow from switch $s_1$ to $s_2$ is excluded from all three forwarding graphs because on switch $s_1$, for the three depicted equivalence classes, the packet flow is determined by the higher-priority rule $r_4$ rather than the lower-priority rule $r_1$.

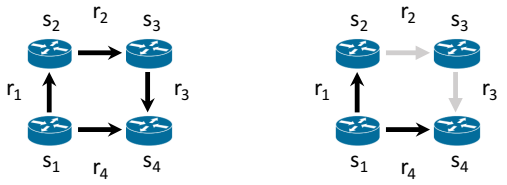
Crucially, in our example, the forwarding graphs that Veriflow constructs are essentially the same to previously constructed ones (dashed areas) except for the new edge from switch $s_1$ to $s_4$. In addition, $G_1$, $G_2$ and $G_2$ share much in common, e.g. $G_2$ and $G_3$ have the same edge from switch $s_2$ to $s_3$. As the number of rules in the network increases, so may the commonality among forwarding graphs. In real networks, this leads to inefficiencies that pose problems under real-time constraints.

We now illustrate how our approach avoids these kind of inefficiencies. For illustrative purposes, assume we start again with the network in which only rules $r_1$, $r_2$ and $r_3$ (black edges) have been installed on switches $s_1$, $s_2$ and $s_3$, respectively. The collection of IP prefixes in the network induces half-closed intervals, each of which we call an *atom*. A set of atoms can represent an IP prefix. For example, as shown at the top of Figure 2, the set $\{\alpha_2, \alpha_3\}$ represents the IP prefix of rule $r_2$.

At the core of our approach is a directed graph whose edges are labelled by atoms. The purpose of this edge-labelled graph is to represent packet flows in the entire network. For example, to represent that $r_2$ forwards packets from switch $s_2$ to $s_3$ we label the corresponding edge in the directed graph with the atoms $\alpha_2$ and $\alpha_3$.

Of course, an edge-labelled graph that represents all flows in the network may need to be transformed when a new rule is inserted or removed. The bottom of Figure 2 illustrates the nature of such a graph transformation in the case where rule $r_4$ is inserted into switch $s_1$. The point of the drawing is threefold. First, observe that the rule insertion of $r_4$ results in the creation of a new atom $\alpha_4$ (red label in the graph on the bottom-left corner). Using the newly created atom, $r_4$'s IP prefix can now be precisely represented as the set of atoms $\{\alpha_2, \alpha_3, \alpha_4\}$. Second, when a new atom, such as $\alpha_4$, is created, existing atom representations may need to be updated. For example, $r_1$'s IP prefix on the edge from switch $s_1$ to $s_2$ needs to be now represented by four instead of only three atoms. Finally, since rule $r_4$, recall, has higher priority than rule $r_1$, three of those four atoms need be moved to the newly inserted edge from switch $s_1$ to $s_4$ (as shown by a dashed arrow in Figure 2). This results in the edge-labelled graph shown in the bottom-right corner of Figure 2 where the edges from switch $s_1$ correspond to the forwarding action of the rules $r_1$ and $r_4$ and are labelled by the set of atoms $\{\alpha_1\}$ and $\{\alpha_2, \alpha_3, \alpha_4\}$, respectively. Crucially, note how our approach avoids the construction of *multiple* overlapping forwarding graphs by transforming a *single* edge-labelled graph instead.

Delta-net's key components and sequence of steps are depicted in Figure 3. In this flowchart, the steps in shaded areas — annotated by $\{①, ②\}$ and $\{③\}$ in Figure 3 — are new and described in § 3.1 and § 3.2, respectively. Here, we only highlight two main fundamental differences between Delta-net and Veriflow:




(a) Veriflow (b) Delta-net

Figure 4: Comparison of processed rules (black edges).

- Veriflow generally has to traverse rules in different switches to compute equivalence classes and forwarding graphs: in our example, when rule $r_4$ is inserted into switch $s_1$, Veriflow traverses all rules in the network (four black edges in Figure 4a). By contrast, our approach concentrates on the affected rules in the modified switch. For example, when rule $r_4$ is inserted into switch $s_1$, the two black edges in Figure 4b show that only rules $r_1$ and $r_4$ on switch $s_1$ are inspected by Delta-net to transform the edge-labelled graph.
- Veriflow recomputes affected equivalence classes and forwarding graphs each time a rule is inserted or removed, whereas Delta-net incrementally transforms a single edge-labelled graph to represent the flows of *all* packets in the entire network. This significantly broadens the scope of Delta-net (§ 2.2) because it can more efficiently handle network failures and "what if" queries regarding *many or all* packets in the network.

## 2.2 Functional design goals

In addition to more stringent real-time constraints, our work is guided by the following three design goals:

1. Similar to Datalog-based approaches [17, 33], we want to efficiently find *all* packets that can reach a node $B$ from $A$, avoiding restrictions of SAT/SMT-based data plane checkers (e.g. [34]), which can solve a broader class of problems but require multiple calls to their underlying SAT/SMT solver to find more than one witness for the reachability from $A$ to $B$.
2. Our design should support known incremental network verification techniques that construct forwarding graphs for the purpose of checking reachability properties each time a rule is inserted or removed [27]. This is important because it preserves one of the main characteristics of previous work, namely: it is practical, and no expertise in formal verification is required to check the data plane.
3. When real-time constraints are less important (as in the case of pre-deployment testing, e.g. [58]), we want to facilitate the answering of a broader class of (possibly incremental) reachability queries, such as *all-pairs reachability* queries in the style of recent Datalog approaches [17, 33]. These kind of queries generally concern the reachability between *all* packets and pairs of nodes in the network. We also aim at efficiently answering queries in scenarios that involve many or all packets, such as link failures [27].

After explaining the technical details of Delta-net, we describe how it achieves these design goals (§ 3.3).

| Priority | IP Prefix | Action |
|---|---|---|
| High | 0.0.0.10/31 | drop |
| Low | 0.0.0.0/28 | forward |

Table 1: A forwarding table for a network switch.

# 3 Delta-net

In this section, we explain Delta-net's underlying atom representation (§ 3.1), and its algorithm for modifying rules through insertion and removal operations (§ 3.2). Recall that these two subsections correspond to the steps annotated by {①,②} and {③} in Figure 3, respectively.

We illustrate the internal workings of Delta-net using the simple forwarding table in Table 1. It features two rules, $r_H$ and $r_L$, whose subscript corresponds to their priority: the higher-priority rule, $r_H$, drops packets whose destination address matches the IP prefix 0.0.0.10/31, whereas the lower-priority rule, $r_L$, forwards packets destined to the IP prefix 0.0.0.0/28. We elide details about the next hop (where a matched packet should be sent) because it is not pertinent to the example.

As alluded to in the previous section (§ 2.1), we can think of IP prefixes as half-closed intervals: $r_H$'s IP prefix, 0.0.0.10/31, corresponds to the half-closed $[10:12)$. Similarly, $0.0.0.0/28 = [0:16)$ for $r_L$'s IP prefix. Of course, this interval representation can be easily generalized to IPv6 addresses. Next, we show how Delta-net represents rules with such IP prefixes, for some fixed IP address length.

## 3.1 Atom representation

In this subsection, we describe the concept of atoms; how they are maintained is essential to the rule modifications algorithms in the next subsection (§ 3.2).

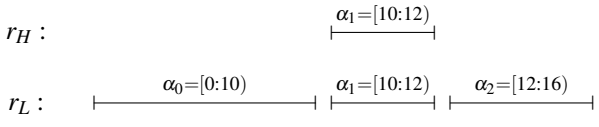



Figure 5: Atoms for the IP prefix rules in Table 1.

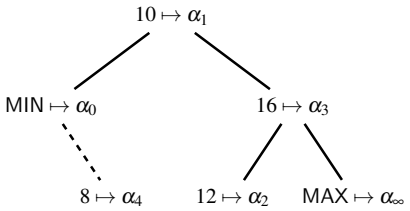



Figure 6: Balanced binary search tree of key/value pairs after inserting the half-closed intervals from Figure 5.

Intuitively, we can segment the IP prefixes of all the rules in the network into disjoint half-closed intervals, which we call atoms. This kind of segmentation is illustrated in Figure 5 using the rules $r_H$ and $r_L$ in Table 1.[1]

By construction of atoms, we can represent an IP prefix of a rule $r$ as a *set of atoms*. We denote this IP prefix representation by $[\![\mathsf{interval}(r)]\!]$. For example, $r_H$'s IP prefix, $[\![\mathsf{interval}(r_H)]\!]$, corresponds to the singleton set consisting of the atom $\alpha_1$, whereas $r_L$'s IP prefix is $[\![\mathsf{interval}(r_L)]\!] = \{\alpha_0, \alpha_1, \alpha_2\}$. Using these atoms, we can represent, for example, the set difference $[\![\mathsf{interval}(r_L)]\!] - [\![\mathsf{interval}(r_H)]\!]$ to formalize the fact that $r_L$ can only match packets that are not matched by the higher-priority rule $r_H$. Next, we explain how to devise an efficient representation of atoms such that we can efficiently verify network-wide reachability properties when a rule is inserted or removed (§ 3.2).

At the core of our atom representation is a function, $\mathfrak{M}$, that maps non-negative integers to identifiers. Specifically, $\mathfrak{M}$ is an ordered map that contains key/value pairs $n \mapsto \alpha_i$ where $n$ is a lower or upper bound of an IP prefix of a rule $r$ (denoted by $\mathsf{lower}(r)$ and $\mathsf{upper}(r)$, respectively) and $\alpha_i$ is a unique identifier, called *atom identifier*. For example, $\mathsf{lower}(r_H) = 10$ and $\mathsf{upper}(r_H) = 12$. More generally, we ensure that $\mathsf{MIN} \leq \mathsf{lower}(r) < \mathsf{upper}(r) \leq \mathsf{MAX}$ for every rule $r$ where $\mathsf{MIN} = 0$ and $\mathsf{MAX} = 2^k$ for some fixed positive integer $k$, e.g. $k = 32$ for 32-bit IP addresses. We maintain the invariant that $\mathfrak{M}$ contains only unique keys. The interpretation of each pair $n \mapsto \alpha_i$ in $\mathfrak{M}$, for all $n < \mathsf{MAX}$, is as follows: the atom identifier $\alpha_i$ denotes the *atom* $[n : n')$ where $n'$ is the next numerically greater key in $\mathfrak{M}$. Each atom identifier, therefore, uniquely denotes a half-closed interval, i.e. an atom. For efficiency reasons, we ensure that each atom identifier is generated from a consecutively increasing counter that starts at zero. Before processing any rules, we initialize $\mathfrak{M}$ by inserting $\mathsf{MIN} \mapsto \alpha_0$ and $\mathsf{MAX} \mapsto \alpha_\infty$ where $\alpha_\infty$ is the greatest atom identifier.

We define the procedure CREATE_ATOMS($r$), where $\mathsf{interval}(r) = [\mathsf{lower}(r) : \mathsf{upper}(r))$ is the half-closed interval corresponding to $r$'s IP prefix, such that, if $\mathfrak{M}$ has not already paired $\mathsf{lower}(r)$ with an atom identifier, then it inserts into $\mathfrak{M}$ the key/value pair $\mathsf{lower}(r) \mapsto \alpha_j$ for the next available counter value $\alpha_j$; similarly, we conditionally insert into $\mathfrak{M}$ the key/value pair $\mathsf{upper}(r) \mapsto \alpha_k$ for the next available counter value $\alpha_k$. Note that after CREATE_ATOMS($r$) has been called, $\mathfrak{M}$ may contain 0, 1, or 2 new atoms (but not more). For example, IP prefixes such as 1.2.0.0/16 and 1.2.0.0/24 have the same lower bound because they only differ in their prefix lengths, and so together yield only three and not four atoms. While the values of atom identifiers depend on the order in which rules are inserted, the set of generated atoms at the end is invariant under the order in which CREATE_ATOMS is called. We also remark that the number of atoms represented by $\mathfrak{M}$ is equal to $\mathfrak{M}$'s size minus one.

For our complexity analysis, we assume that the $\mathfrak{M}$'s insertion and retrieval operations run logarithmically in the size of $\mathfrak{M}$, which could be achieved with a balanced binary-search tree such as a red-black tree. In this case, Figure 6 (excluding the leaf node connected by a dashed edge) illustrates the balanced binary search tree that results after CREATE_ATOMS($r_H$) and CREATE_ATOMS($r_L$) has been called for the rules $r_H$ and $r_L$ in Table 1. For example, $\alpha_1$ at the root of the binary search tree in Figure 6 denotes the atom $[10 : 12)$. When clear from the context, we refer to atom identifiers and atoms interchangeably.

## 3.2 Edge labelling algorithm

Using our atom representation (§ 3.1), we show how to efficiently label the edges of a directed graph that succinctly describes the flow of all packets in the entire network. Our algorithm is incremental in the sense that it only changes edge labels that are affected by the insertion or removal of a rule. Our algorithm, which achieves this incrementality, requires the following notions.

We denote an IP prefix forwarding rule by $r$, possibly with a prime symbol. Each rule $r$ is associated with $\mathsf{priority}(r)$ and $\mathsf{link}(r)$, as explained in turn. We assume that rules in the same forwarding table whose IP prefixes overlap have pair-wise distinct priorities, denoted by $\mathsf{priority}(r)$.[2] For all rules $r$ and $r'$ in the same forwarding table, $r$ has a *higher priority than* $r'$ if $\mathsf{priority}(r) > \mathsf{priority}(r')$; equivalently, $\mathsf{priority}(r) < \mathsf{priority}(r')$ means that $r$ has a *lower priority than* $r'$. Note that longest-prefix routing can be simulated by assigning rule priorities according to prefix lengths [55]. We denote by $\mathsf{link}(r)$ a directed edge in a graph that is induced by a network topology. For theoretical and practical reasons (see also § 4.1), $\mathsf{link}(r)$ is purposefully more general than a pair of, say, ports. We write $\mathsf{source}(r)$ for the node

[1] Appendix A illustrates the fact that atoms induce a Boolean lattice.

[2] This assumption is reasonable for, say, OpenFlow tables where the matching of rules with the same highest priority is explicitly undefined.

in the graph on which $\mathsf{link}(r)$ is incident. For example, $\mathsf{source}(r_1) = s_1$ and $\mathsf{source}(r_2) = s_2$ in Figure 2.

From a high-level perspective, Delta-net consists of two algorithms, one for inserting (Algorithm 1) and another for removing (Algorithm 2) a single rule. Both algorithms accesses three global variables: $\mathfrak{M}$, *label* and *owner*, as described in turn. First, $\mathfrak{M}$ is the balanced binary tree described in § 3.1, e.g. Figure 6. Second, given a *link* in the network topology, *label*[*link*] denotes a set of atoms, each of which corresponds to a half-closed interval that a designated field in a packet header $h$ can match for $h$ to be forwarded along the *link*. Finally, *owner* is an array of hash tables, each of which stores a balanced binary search tree containing rules ordered by priority. More accurately, *owner* is an array of sufficient size such that, for every atom $\alpha$, *owner*[$\alpha$] is a hash table that maps a *source* node to a balanced binary search tree, *bst*, that orders rules in the *source* node that contain atom $\alpha$ in their interval according to their priority, i.e., we maintain the invariant that *bst* contains only rules $r$ such that *source* = $\mathsf{source}(r)$ and $\alpha \in [\![\mathsf{interval}(r)]\!]$ where *bst* = *owner*[$\alpha$][*source*]. The highest-priority rule in a non-empty balanced binary search tree *bst* can be retrieved via *bst*.highest_priority_rule(). We remark that we do not use a priority queue because Algorithm 2 described later (§ 3.2.2) needs to be able to remove arbitrary rules, not just the highest-priority one. We write $r \in bst$ when rule $r$ is stored in *bst*.

### 3.2.1 Edge labelling when inserting a rule

We now explain how the INSERT_RULE procedure in Algorithm 1 works. The algorithm starts by calling CREATE_ATOMS$^+$ (line 2) that accomplishes the same as CREATE_ATOMS from § 3.1 except that CREATE_ATOMS$^+$ also returns $\Delta$, a set of *delta-pairs*, as explained next. Each delta-pair in $\Delta$ is of the form $\alpha \mapsto \alpha'$ where $\alpha$ and $\alpha'$ are atoms. The intuition is that the half-closed interval previously represented by $\alpha$ needs to be now represented by two atoms instead, namely $\alpha$ and $\alpha'$. We call this *atom splitting*. In a nutshell, this splitting provides an efficient mechanism for incrementally refining the precision of our abstract domain. This incremental abstraction refinement allows us to precisely and efficiently represent all Boolean combinations of rules in the network (see also § 1).

To illustrate the splitting of atoms, let $r_M$ be a new medium-priority rule to be inserted into Table 1 such that $\mathsf{priority}(r_L) < \mathsf{priority}(r_M) < \mathsf{priority}(r_H)$. Assume $r_M$'s IP prefix is 0.0.0.8/30; hence, $\mathsf{interval}(r_M) = [8 : 12)$. If $\mathfrak{M}$ is the binary search subtree in Figure 6 consisting of undashed edges, then CREATE_ATOMS$^+(r_M)$ returns a single delta-pair, namely $\Delta = \{\alpha_0 \mapsto \alpha_4\}$, where $\alpha_0$ is the atom identifier denoting the atom $[\mathsf{MIN} : 10)$ before

**Algorithm 1** Inserts rule $r$ into a forwarding table.

```
1:  procedure INSERT_RULE(r)
2:    Δ ← CREATE_ATOMS+(r)                          ▷ |Δ| ≤ 2
3:    for α ↦ α′ in Δ do
4:      owner[α′] ← owner[α]
5:      for source ↦ bst in owner[α] do
6:        r′ ← bst.highest_priority_rule()
7:        label[link(r′)] ← label[link(r′)] ∪ {α′}
8:      end for
9:    end for
10:   for α in [[interval(r)]] do
11:     r′ ← null
12:     bst ← owner[α][source(r)]
13:     if not bst.is_empty() then
14:       r′ ← bst.highest_priority_rule()
15:     end if
16:     if r′ = null or priority(r′) < priority(r) then
17:       label[link(r)] ← label[link(r)] ∪ {α}
18:       if r′ ≠ null and link(r) ≠ link(r′) then
19:         label[link(r′)] ← label[link(r′)] − {α}
20:       end if
21:     end if
22:     bst.insert(r)
23:   end for
24: end procedure
```

$r_M$ has been inserted, and $\alpha_4$ is a new atom identifier, depicted as a dashed leaf in Figure 6. Here, $\Delta = \{\alpha_0 \mapsto \alpha_4\}$ means that the existing atom $[\mathsf{MIN} : 10)$ needs to be split into $\alpha_0 = [\mathsf{MIN} : 8)$ and $\alpha_4 = [8 : 10)$. Note that there are always at most two delta-pairs in $\Delta$. Thus, since $|\Delta| \leq 2$, we can effectively update the atom representation of forwarding rules in an incremental manner.

The splitting of atoms is effectuated by updating the labels for some links in the single-edged graph that represents the flow in the entire network (line 7). To quickly determine these links, we exploit the highest-priority matching mechanism of packets. For this purpose, we use the array of hash tables, *owner*: it associates an atom $\alpha$ and *source* node with a binary search tree *bst* such that *bst*.highest_priority_rule() determines the next hop from *source* of an $\alpha$-packet (line 6). Since $|\Delta| \leq 2$, the doubly nested loop (line 3–9) runs at most twice. For each delta-pair $\alpha \mapsto \alpha'$ in $\Delta$, the array of hash tables is updated so that *owner*[$\alpha'$] is a copy of *owner*[$\alpha$] (line 4). Therefore, since $r' \in owner[\alpha][\mathsf{source}(r')]$ holds for the existing atom $\alpha$, it follows that $r' \in owner[\alpha'][\mathsf{source}(r')]$ holds for the new atom $\alpha' \in [\![\mathsf{interval}(r')]\!]$, thereby maintaining the invariant of the *owner* array of hash tables (§ 3.2). We adjust the labels accordingly (line 7). The remainder of Algorithm 1 (line 10–23) reassigns atoms based on the priority of the rule that 'owns' each atom, as explained next.

**Algorithm 2** Removes rule $r$ from a forwarding table.

```
1:  procedure REMOVE_RULE(r)
2:    for α in [[interval(r)]] do
3:      bst ← owner[α][source(r)]
4:      r' ← bst.highest_priority_rule()
5:      bst.remove(r)
6:      if r' = r then
7:        label[link(r)] ← label[link(r)] − {α}
8:        if not bst.is_empty() then
9:          r'' ← bst.highest_priority_rule()
10:         label[link(r'')] ← label[link(r'')] ∪ {α}
11:       end if
12:     end if
13:   end for
14: end procedure
```

The algorithm continues by iterating over all atoms that collectively represent $r$'s IP prefix (line 10), possibly including the newly created atom(s) in $\Delta$ (see previous paragraphs). For each such atom $\alpha$ in $[\![\mathsf{interval}(r)]\!]$, we find the highest-priority rule $r'$ (line 14) that determines the flow of an $\alpha$-packet at the node $\mathsf{source}(r)$ into which rule $r$ is inserted. We say such a rule $r'$ *owns* $\alpha$. If no such rule exists or its priority is lower than $r$'s (line 16), we assign $\alpha$ to the set of atoms that determine which network traffic can flow along the link of $r$ (line 17–20), i.e. $label[\mathsf{link}(r)]$. Finally, we insert $r$ into the binary search tree for atom $\alpha$ and node $\mathsf{source}(r)$ (line 22), irrespective of which rule owns atom $\alpha$.

#### 3.2.2 Edge labelling when removing a rule

Algorithm 2 removes a rule $r$ from a forwarding table. Similar to Algorithm 1, Algorithm 2 iterates over all atoms $\alpha$ that are needed to represent $r$'s IP prefix (line 2). For each such atom $\alpha$, it retrieves the *bst* that is specific to the node from which $r$ should be removed (line 3). After finding the highest-priority rule $r'$ in *bst* (line 4), it removes $r$ from *bst* (line 5). If $r'$ equals $r$ (line 6), we need to remove $\alpha$ from the label of $\mathsf{link}(r)$ because the rule that needs to be removed, $r$, owns atom $\alpha$ (as described in § 3.2.1). In addition, we may need to transfer the ownership of the next higher priority rule (line 8-11).

We remark that after the removal of a rule, it may be that some (at most two) atoms are not needed any longer. In this case, akin to garbage collection, we could reclaim the unused atom identifier(s). This 'garbage collection' mechanism is omitted from Algorithm 2.

#### 3.2.3 Complexity analysis

We now show that each rule update is amortized linear time in the number of affected atoms and logarithmic in the maximum number of overlapping rules in a single switch. While in the worst-case there are as many atoms as there are rules in the network, our experiments (§ 4) show that the number of atoms is typically much smaller in practice, explaining why we found Delta-net to be highly efficient in the vast majority of cases.

**Theorem 1** (Asymptotic worst-case time complexity)**.** *To insert or remove a total of $R$ rules, Algorithm 1 and 2 have a $O(RK \log M)$ worst-case time complexity where $K$ is the number of atoms and $M$ is the maximum number of overlapping rules per network switch.*

*Proof.* The proof can be found in Appendix B. □

The space complexity of Delta-net is $O(RK)$ where $R$ and $K$ are the total number of rules and atoms, respectively. We recall that $K$ is significantly smaller than $R$. We also experimentally quantify memory usage (§ 4).

### 3.3 Revisited: functional design goals

From a functionality perspective, recall that our work is guided by three design goals (§ 2.2). In this subsection, we explain how Delta-net achieves these goals.

**API for persistent network-wide flow information.** Delta-net provides an exact representation of all flows through the entire network. For this purpose, Delta-net maintains the atom labels for every edge in the graph that represents the network topology. From a programmer's perspective, this edge-centric information can be always retrieved in constant-time through $label[link]$ where *link* is a pair of nodes in this graph. This way, our API allows a programmer to answer reachability questions about packet flow through the entire network irrespective of the rule that has been most recently inserted or removed. This makes Delta-net different from Veriflow [27]. Architecturally, our generalization is achieved by decoupling packet equivalence classes (whether affected by a rule update or not) from the construction of their corresponding forwarding graphs, cf. [27].

**Incremental network verification via delta-graphs.** Similar to Veriflow [27], Delta-net can build forwarding graphs, if necessary, to check reachability properties that are suitable for incremental network verification, such as checking the existence of forwarding loops each time a rule is inserted or removed. In fact, the concept of atoms has as consequence a convenient algorithm for computing a compact edge-labelled graph, called *delta-graph*, that represents all such forwarding graphs. We can generate a delta-graph as a by-product of Algorithm 1 for all atoms $\alpha$ whose owner changes (line 16-21); similarly for Algorithm 2. If so desired, multiple rule updates may be aggregated into a delta-graph.

**Algorithm 3** Compute all-pairs reachability of all atoms.

```
for k, i, j in V do                          ▷ Triple nested loop
    label[i, j] ← label[i, j] ∪ (label[i,k] ∩ label[k, j])
end for
```

**Easier checking of other reachability properties.** Delta-net's design provides a lattice-theoretic foundation for transferring known algorithmic techniques to the field of network verification. For example, Algorithm 3 adapts the Floyd–Warshall algorithm to compute the transitive closure of packet flows between all pairs of nodes in the network. Note that our adaptation interchanges the usual maximum and addition operators with union and intersection of sets of atoms, respectively. This way, Algorithm 3 process multiple packet equivalence classes in each hop.[3] Veriflow has not been designed for such computations, and Algorithm 3 illustrates how Delta-net facilitates use cases beyond the usual reachability checks, cf. [27, 25, 55]. This algorithm could be run either on the edge-labelled graph that represents the entire network or only its incremental version in form of a delta-graph (see previous paragraph).

While decision problems such as all-pairs reachability have a higher computational complexity (e.g., Algorithm 3's complexity is $O(K|V|^3)$ where $K$ and $V$ is the number of atoms and nodes in the edge-labelled graph, respectively), they are relevant and useful during pre-deployment testing of SDN applications, as demonstrated by recent work on Datalog-based network verification, e.g. [17, 33]. The fact that our design makes it possible to verify network-wide reachability by intersecting or taking the union of sets of atoms [55] is also relevant for scenarios that involve many or all packet equivalence classes at a time, such as "what if" queries, network failures, and traffic isolation properties, e.g. [3, 18].

# 4 Performance evaluation

In this section, we experimentally evaluate our implementation of Delta-net (§ 4.1) on a diverse range of data sets (§ 4.2) that are significantly larger than previous ones (see also Appendix C). Our experiments provide strong evidence that Delta-net significantly advances the field of real-time network verification (§ 4.3).

## 4.1 Implementation

We implemented Algorithm 1 and 2 in C++14 [22]. Our implementation is single-threaded and comprises around 4,000 lines of code that only depend on the C++14 standard library. In particular, we use the standard hashmap, balanced binary search tree and resizeable array implementations. We implement edge labels as customized dynamic bitsets, stored as aligned, dynamically allocated, contiguous memory. We detect forwarding loops via an iterative depth-first graph traversal.

We remark that while Algorithm 1 and 2 focus on handling IP prefix rules, our approach can be extended for other packet header fields. For non-wildcard (i.e. concrete) header fields, our implementation achieves this by encoding composite match conditions as separate nodes in the single edge-labelled graph. For example, if a switch $s$ contains rules that can match three input ports, we encode $s$ as three separate nodes in the edge-labelled graph. It is for this reason that we report the number of graph nodes rather than the number of switches when describing our data sets in the next subsection.

[3] A routine proof by induction on $k$ (the outermost loop) shows that Algorithm 3 computes the all-pairs reachability of every $\alpha$-packet.

| Data set | Nodes | Max Links | Operations |
|---|---|---|---|
| Berkeley | 23 | 252 | $25.6 \times 10^6$ |
| INET | 316 | 40,770 | $249.5 \times 10^6$ |
| RF 1755 | 87 | 2,308 | $67.5 \times 10^6$ |
| RF 3257 | 161 | 9,432 | $149.0 \times 10^6$ |
| RF 6461 | 138 | 8,140 | $150.0 \times 10^6$ |
| Airtel 1 | 68 | 260 | $14.2 \times 10^6$ |
| Airtel 2 | 68 | 260 | $505.2 \times 10^6$ |
| 4Switch | 12 | 16 | $1.12 \times 10^6$ |

Table 2: Data sets used for evaluating Delta-net.

## 4.2 Description of data sets

Our data sets are publicly available [14] and can be broadly divided into two classes: data sets derived from the literature (§ 4.2.1), and data sets gathered from an ONOS SDN application (§ 4.2.2). Both are significant as the former avoids experimental bias, whereas the latter increases the realism of our experiments. To achieve reproducibility, we organize our data sets as text files in which each line denotes an *operation*: an insertion or removal of a rule. So all operations can be easily replayed.

Table 2 summarizes our data sets in terms of three metrics.The second and third column in Table 2 correspond to the maximum number of nodes and links in the edge-labelled graph, respectively. We recall that the number of nodes is proportional to the number of ports and switches in the network (§ 4.1). The total number of operations is reported in the last column. Note that most of our data sets are significantly larger than previous ones, cf. [27, 10, 25, 55] (see also Appendix C). Next, we describe the main features of our data sets.

### 4.2.1 Synthetic data sets

To avoid experimental bias, our experiments purposefully include data sets from the literature [59, 39] that

feature network topologies from the UC Berkeley campus and the Rocketfuel (RF) project [49], namely ASes 1755, 1239, 6257 and 6461. Note that the RF topologies in [39] correspond to those used by [21, 19, 51]. For each of these five network topologies, we generate forwarding rules following the same mechanism as in [59], namely: we gather IP prefixes from over a half a million of real-world BGP updates collected by the Route Views project [46], and compute the shortest paths in a network topology [30]. For example, for the network topology RF 1239, this results in the INET data set [59], a synthetic wide-area backbone network that contains approximately 300 routers, 481 thousand subnets and 125 million IPv4 forwarding rules. We modify the data sets so that rules are inserted with a random priority. After rules have been inserted, we remove them in random order. The first five rows in Table 2 show the resulting data sets, which contain up to 125 million rules. Due to rule removals, the total number of operations is twice the maximum number of rules. Collectively, the Berkeley, INET and RF 1755, 3257 and 6461 data sets comprise around 640 million rule operations. Next, we explain the remaining three data sets in Table 2.

### 4.2.2 SDN-IP Application

In addition to synthetic data sets (§ 4.2.1), we run experiments with ONOS [7, 42], an open SDN platform used by sizeable operator networks around the globe [7, 42].

To obtain a relevant and realistic experimental setup, we run SDN-IP [31, 47], an important ONOS application that allows an ONOS-controlled network to interoperate with external autonomous networks (ASes). This interoperability is achieved as follows (Figure 7). Inside the ONOS-controlled network reside *Border Gateway Protocol* (BGP) speakers (in our experimental setup there is exactly one internal BGP speaker) that use eBGP to exchange BGP routing information with the border routers of adjacent external ASes. This information, in turn, is propagated inside the ONOS-controlled network via iBGP. As sketched in the upper half of Figure 7, SDN-IP listens to these iBGP messages and requests ONOS to dynamically install IP forwarding rules such that packets destined to an external AS arrive at the correct border router. In doing so, SDN-IP sets the priority of rules according to the longest prefix match where rules with longer prefix lengths receive higher priority. For each rule insertion and removal (depicted by $+r_1$ and $-r_2$ in Figure 7), Delta-net checks the resulting data plane.

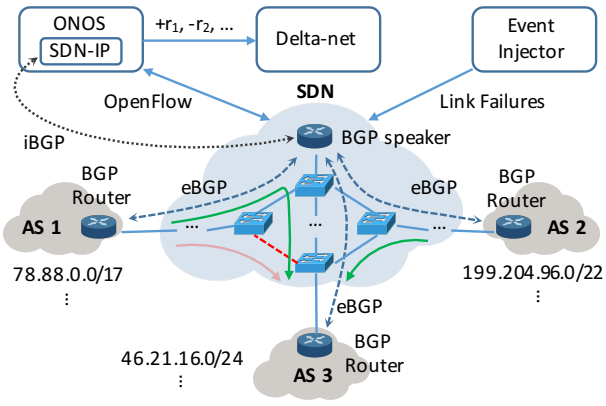



Figure 7: Experimental setup with SDN-IP application.

For our experiments, we run SDN-IP in a single ONOS instance. We use Mininet [29] to emulate a network of sixteen Open vSwitches [43], configured according to the Airtel network topology (AS 9498) [28]. We connect each of these OpenFlow-compliant switches [38] to an external border router that we emulate using Quagga [45]. We configure Quagga such that each border router advertises one hundred IP prefixes, which we randomly select from over half a million real-world IP prefixes gathered from the Route Views project [46], resulting in a total of 1,600 unique (but possibly overlapping) IP prefixes.

Our experiments in § 4.3.1 exploit the fact that SDN-IP relies on ONOS to reconfigure the OpenFlow switches when parts of the network fail. Since network failures happen frequently [4] and pose significant challenges for real-time data plane checkers [25, 27], we can generate interesting data sets by systemically failing links, controlled by the 'Event Injector' process in the upper right half of Figure 7. In particular, the Airtel 1 data set contains the rule insertions and removals triggered by failing a single inter-switch link at a time, recovering each link before failing the next one. Such a link failure (dashed red edge) is illustrated in the left half of Figure 7, causing ONOS to reconfigure the data plane so that a new path is established (green arrow on the left) that avoids the failed link, which caused disruption to earlier network traffic (red arrow). In the case of Airtel 2, we automatically induce all 2-pair link failures (separately failing the first link and then the second one), including their recovery.

We also wanted to study a larger number of rules and IP prefixes, but were limited due to technical issues with ONOS. We worked around these limitations by using a 4-switch ring network. In this smaller ring topology, we configure each Quagga instance to advertise 5,000 IP prefixes (rather than only 100 IP prefixes as in the Airtel experiments), again randomly selected from the Route Views project [46]. We do not fail any links. Instead, we only collect the rules generated by SDN-IP, a process we repeat fourteen times with different IP prefixes. This workaround yields the 4Switch data set in Table 2, comprising 1.12 million rules. In contrast to the previously

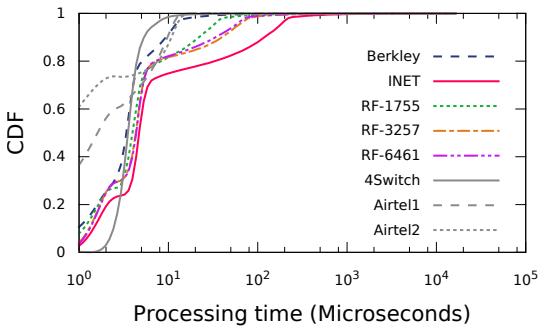



Figure 8: CDF of combined time (microseconds) for processing a rule update and checking for forwarding loops.

described data sets, all of the operations in the 4Switch data set are rule insertions.

## 4.3 Experimental results

Our experiments separately measure Delta-net's performance in checking individual rule updates (§ 4.3.1) and handling a "what if" scenario (§ 4.3.2). In both cases, at the cost of higher memory usage, Delta-net is more than 10× faster than the state-of-the-art. We run our experiments on an Intel Xeon CPU with 3.47 GHz and 94 GB of RAM. Since our implementation is single-threaded (§ 4.1), we utilize only one out of the 12 available cores.

### 4.3.1 Checking network updates

To evaluate Delta-net's performance with respect to rule insertions and removals, we build the delta-graph (§ 3.3) for each operation, and find in it all forwarding loops, a common network-wide invariant [26, 25, 55, 27, 59]. We process the rules in each data set in the order in which they appear in the data sets (§ 4.2).

Table 3 summarizes our experimental results for measuring the checking of rule insertions and removals. The first row in Table 3 shows that the total number of atoms is much smaller than the total number of rules in the network (recall Table 2), suggesting a significant degree of commonality among IP prefix rules that atoms effectively exploit. Furthermore, for all data sets, the median and average rule processing time is less than 5 and 41 microseconds, respectively, which includes the checking of forwarding loops. On closer inspection, as shown in the last row of Table 2, Delta-net processes rule updates and checks for the existence of forwarding loops in less than 250 microseconds for at least 98.5% of cases. The combined time for processing a rule update and finding *all* forwarding loops in the corresponding delta-graph (§ 3.3) is visualized by the cumulative density function (CDF) in Figure 8. It shows that the INET data set [59] (solid red line) is one of the more difficult ones for Delta-net. We remark that Delta-net's memory usage never exceeds the available memory on our machine (Appendix D).

Our measurements are significant because earlier experiments with Veriflow [27] result in an average verification time of 380 microseconds, whereas Delta-net verifies rule insertions and removals in often tens of microseconds, and 41 microseconds on average even on the largest data set, INET. This comparison is meaningful because our data sets are significantly larger than previous ones [27, 10, 25, 55]. Moreover, two of our data sets (Airtel 1 and 2) are derived from a real-world software-defined networking application while causing an extensive number of link failures in the network, which were previously shown to lead to longer verification times [25, 27]. Our experiments therefore provide strong evidence that Delta-net can be at least one order of magnitude faster compared to Veriflow [27]. Since neither Veriflow's implementation (or its algorithm) nor any of the data sets used for its experimental evaluation are publicly available, and neither its time nor space complexity is specified, we further quantify the differences between Delta-net and Veriflow by re-implementing a consistent interpretation of Veriflow, as described next.

Our re-implementation of Veriflow, which we call *Veriflow-RI*, is not intended to be a full-feature copy of Veriflow, but rather a re-implementation of their core idea to enable an honest comparison with Delta-net. Specifically, Veriflow-RI is designed for matches against a single packet header field. This explains why Veriflow-RI uses a one-dimensional trie data structure in which every node has at most two children (rather than three [27]). We optimize the computation of equivalence classes and construction of forwarding graphs. Note that these optimizations may not be possible in the original Veriflow implementation with its ternary trie data structure, and Veriflow-RI may therefore be faster than Veriflow [27]. We remark that Veriflow-RI's space complexity is linear in the number of rules in the network, whereas its time complexity is quadratic, rather than quasi-linear as in the case of Delta-net (Theorem 1).

While Delta-net is only approximately 4× faster than Veriflow-RI on the Airtel data set, on the INET data set, Delta-net is approximately 6× faster than Veriflow-RI. This gap widens on the RF 3257 and 6461 data sets where Delta-net is approximately 7× faster than Veriflow-RI. In turn, however, Veriflow-RI consumes $5-7\times$ less memory than Delta-net (Appendix § D).

It is therefore natural to ask whether this trade-off in space and time is worth it. Next, we answer this question affirmatively by showing that Delta-net can check properties for which Veriflow often times out. This difference in run-time performance is due to the fact that Delta-net incrementally maintains flow information of every packet in the entire network, whereas Veriflow recom-

| | Berkeley | INET | RF 1755 | RF 3257 | RF 6461 | Airtel 1 | Airtel 2 | 4Switch |
|---|---|---|---|---|---|---|---|---|
| Total number of atoms | 668,520 | 563,480 | 726,535 | 726,535 | 726,535 | 2,799 | 2,799 | 443,443 |
| Median rule processing time | 4 $\mu s$ | 5 $\mu s$ | 4 $\mu s$ | 5 $\mu s$ | 5 $\mu s$ | 2 $\mu s$ | 1 $\mu s$ | 4 $\mu s$ |
| Average rule processing time | 5 $\mu s$ | 41 $\mu s$ | 11 $\mu s$ | 22 $\mu s$ | 20 $\mu s$ | 3 $\mu s$ | 3 $\mu s$ | 5 $\mu s$ |
| Percentage $< 250\,\mu s$ | 99.9% | 98.5% | 99.8% | 99.6% | 99.7% | 99.9% | 99.9% | 99.9% |

Table 3: Experimental results using Delta-net, measuring rule insertions and removals.

putes the forwarding graph for each affected equivalence class. What is remarkable is that Delta-net achieves this extra bookkeeping without limiting the checking of individual network updates (see previous paragraph).

### 4.3.2 Beyond network updates

We show how Delta-net can go beyond traditional data plane checks per network update. To do so, we consider the following question, which was previously posed by [27], as an exemplar of a "what if" query: What is the fate of packets that are using a link that fails? We interpret their question to mean that Veriflow has to construct forwarding graphs for all packet equivalence classes that are affected by a link failure. This is known to be a difficult task for Veriflow since it requires the construction of at least a hundredfold more forwarding graphs compared to checking a rule insertion or removal (§ 4.3.1). Here, our experiment quantifies how much Delta-net gains by incrementally transforming a single-edge labelled graph instead of constructing multiple forwarding graphs.

For our experiments, we generate a consistent data plane from all the rule insertions in the five synthetic and 4Switch data sets in Table 2, respectively. And in the case of Airtel, we extract a consistent data plane snapshot from ONOS. The total number of resulting rules in each data plane is shown in the second column of Table 4. For all of these seven data planes, we answer which packets and parts of the network are affected by a hypothetical link failure. The verification task therefore is to represent via one or multiple graphs all flows of packets through the network that would be affected when a link fails. The third column in Table 2 (number of links) corresponds to the number of queries we pose, except for the new Airtel data plane snapshot where we pose 158 queries.

Since Delta-net already maintains network-wide packet flow information, we expect it to perform better than Veriflow-RI.[4] The third and fourth column in Table 4 quantify this performance gain by showing the average query time of Veriflow-RI and Delta-net, respectively. On three data planes, Veriflow-RI exceeds the total run-time limit of 24 hours, whereas the longest running Delta-net experiment takes a total of 3.2 hours. When these time outs in Veriflow-RI occur, we report its incomplete average query time $t$ as '$t^{\dagger}$'. We find that Delta-net is usually more than 10× faster than Veriflow-RI (even if Delta-net checks for forwarding loops, as reported in the last column). Since Delta-net is very fast in maintaining the flow of packets, the difference between the last two columns in Table 4 shows that Delta-net's processing time is dominated by the property check (here, forwarding loops). In contrast to Delta-net, Veriflow's processing time is reportedly dominated by the construction of forwarding graphs [27].

[4] Recall from previous experiments (§ 4.3.1), Delta-net's extra bookkeeping poses no performance problems for checking network updates.

## 5 Related work

In this section, we discuss related works in the literature.

**Stateful networks.** One of the earliest stateful network analysis techniques [9] proposes symbolic execution of OpenFlow applications using a simplified model of OpenFlow network switches. VeriCon [5] uses an SMT solver to automatically prove the correctness of simple SDN controllers. FlowTest [15] investigates relevant AI planning techniques. SymNet [50] symbolically analyzes stateful middleboxes through additional fields in the packet header. Unlike [9], BUZZ [16] adopts a symbolic model-based testing strategy [52] as a way to capture the state of forwarding devices. Most recent complexity results [53] are the first step towards a taxonomy of decision procedures in this research area. Real-time network verification techniques (see next paragraph) can be extended to check safety properties that depend on the state of the SDN controller [6].

**Stateless networks.** The seminal work of Xie et al. [54] introduces stateless data plane checking to which Delta-net belongs. The research that emerged from [54] can be broadly divided into offline [57, 2, 24, 40, 1, 34,

| Data plane | Rules | Average query time (*ms*) | | |
|---|---|---|---|---|
| | | Veriflow-RI | Delta-net | +Loops |
| Berkeley | 12,817,902 | 3,073.0 | **4.7** | 93.3 |
| INET | 124,733,556 | 29,117.5$^{\dagger}$ | **0.7** | 2,888.6 |
| RF 1755 | 33,732,869 | 8,100.6 | **1.3** | 897.4 |
| RF 3257 | 74,492,920 | 17,645.3$^{\dagger}$ | **1.0** | 2.6 |
| RF 6461 | 75,005,738 | 17,594.5$^{\dagger}$ | **0.4** | 0.4 |
| Airtel | 38,100 | 4.5 | **0.04** | 2.3 |
| 4Switch | 1,120,000 | 433.4 | **21.1** | 128.1 |

Table 4: Experimental results for "what if" link failures.

48, 26, 35, 17, 33] and online [27, 25, 55] approaches. The offline approaches encode the problem into Datalog [17, 33] or logic formulas that can be checked for satisfiability by constructing a Binary Decision Diagram [57, 2] or calling an SAT/SMT solver [24, 40, 1, 34, 48, 23, 35]. By contrast, all modern online approaches [27, 25, 55] partition in some way the set of all network packets. In particular, the partitioning scheme described in [26], on which [27] is based, dynamically computes equivalence classes by propagating ternary strings in the network, whereas more recent work [25, 55, 8], including ours, pre-compute network packet partitions prior to checking a verification condition. Our work could be used in conjunction with network symmetry reduction techniques [44]. Custom network abstractions can be very useful for restricted cases [20]. While potentially less efficient, our work is more general than [20], and most closely related to [27, 10, 25, 55, 59, 8], which we discuss in turn. The complexity of the most prominent of these works, including Veriflow [27] and NetPlumber [25], is summarized in work [32, Section II] that is independent from ours.

Veriflow [27] constructs multiple forwarding graphs that may significantly overlap (§ 2.1). Our algorithm exploits this overlapping and transforms a single edge-labelled graph instead. Moreover, Veriflow relies on the fact that overlapping IP prefixes can be efficiently found using a trie data structure [27]. By contrast, atoms are generally not expressible as a single IP prefix. For example, atom $[0:10)$ in Figure 5 can only be represented by the union of at least two IP prefixes.

Chen [10] shows how to optimize Veriflow [27], while retaining its core algorithm. Similar to [10], we represent IP prefixes in a balanced binary search tree. Unlike [10], however, our representation serves as a built-in index of half-closed intervals through which we address fundamental limitations of Veriflow (§ 2.1).

NetPlumber [25] incrementally creates a graph that, in the worst case, consists of $R^2$ edges where $R$ is the number of rules in the network. In contrast to NetPlumber, Delta-net maintains a graph whose size is proportional to the number of links in the network, which is usually much smaller than $R$. Since the number of atoms tends to be much less than $R$ (§ 4), Delta-net has an asymptotically smaller memory footprint than NetPlumber.

Yang and Lam [55] propose a more compact representation of forwarding graphs that reduces the task of data plane checking to intersecting sets of integers. For the restricted, but common, case of checking IP forwarding rules, our algorithm is asymptotically faster than theirs. Our algorithm, however, does not find the unique minimal number of packet equivalence classes, cf. [55].

More recent work for stateless and non-mutating data plane verification [8] encodes a canonical form of ternary bit-vectors, and shows on small data sets with a few thousand rules that their encoding performs better than Yang and Lam [55]'s algorithm. It would be interesting to repeat these experiments on our, significantly larger, data sets.

Finally, Libra [59] may be used for incrementally checking network updates, but it requires an in-memory "streaming" MapReduce run-time, whereas Delta-net avoids the overheads of such a distributed system. Since Libra's partitioning scheme into disjoint subnets is orthogonal to our algorithm, however, it would be interesting to leverage both ideas together in future work.

## 6 Concluding remarks

In this paper, we presented Delta-net (§ 3), a new data plane checker that is inspired by program analysis techniques in the sense that it automatically refines a lattice-theoretical abstract domain to precisely represent the flows of all packets in the entire network. We showed that this matters from a theoretical and practical point of view: Delta-net is asymptotically faster and/or more space efficient than prior work [27, 25, 55], and its new design facilitates Datalog-style use cases [17, 33] for which the transitive closure of many or all packet flows needs to be efficiently computed (§ 3.3). In addition, Delta-net can be used to analyze catastrophic network events, such as link failures, for which current incremental techniques are less effective. To show this experimentally (§ 4), we ran an adaptation of the link failure experiments by Khurshid et al. [27] on data sets that are significantly larger than previous ones. For this exemplar "what if" scenario, we found that Delta-net is several orders of magnitude faster than the state-of-the-art (Table 4). Our work therefore opens up interesting new research directions, including testing scenarios under different combinations of failures, which have been shown to be effective for distributed systems, e.g. [56].

**Future work.** One advantage of Delta-net is that its main loops over atoms in Algorithm 1 and 2 are highly parallelizable. In addition, (stateless) packet modification of IP prefixes can be easily supported without substantial changes to the data structures by augmenting the edge-labelled graph with the necessary information on how atoms are transformed along hops. We are also studying an improved version of Delta-net that avoids the quadratic space complexity by exploiting properties of IP prefixes. Finally, since a naive implementation of Delta-net is exponential in the number of range-based packet header fields (as is Veriflow's [32, Section II]), it would be interesting to guide further developments into multi-range support in higher dimensions using the 'overlapping degree' among rules [32].

**Acknowledgements.** We would like to thank Sho Shimizu, Pingping Lin and members of the ONOS developer mailing list for technical support. We thank Rao Palacharla, Nate Foster and Mina Tahmasbi for their invaluable feedback on an early draft of this paper. We also would like to thank Ratul Mahajan and the anonymous reviewers of NSDI for their detailed comments and helpful suggestions.

## References

[1] AL-SHAER, E., AND AL-HAJ, S. FlowChecker: Configuration analysis and verification of federated OpenFlow infrastructures. In *SafeConfig* (2010).

[2] AL-SHAER, E., MARRERO, W., EL-ATAWY, A., AND EL-BADAWI, K. Network configuration in a box: towards end-to-end verification of network reachability and security. In *ICNP* (2009).

[3] ANDERSON, C. J., FOSTER, N., GUHA, A., JEANNIN, J.-B., KOZEN, D., SCHLESINGER, C., AND WALKER, D. NetKAT: Semantic foundations for networks. In *POPL* (2014).

[4] BAILIS, P., AND KINGSBURY, K. The network is reliable. *Queue 12*, 7 (July 2014), 20:20–20:32.

[5] BALL, T., BJØRNER, N., GEMBER, A., ITZHAKY, S., KARBYSHEV, A., SAGIV, M., SCHAPIRA, M., AND VALADARSKY, A. VeriCon: Towards verifying controller programs in software-defined networks. In *PLDI* (2014).

[6] BECKETT, R., ZOU, X. K., ZHANG, S., MALIK, S., REXFORD, J., AND WALKER, D. An assertion language for debugging SDN applications. In *HotSDN* (2014).

[7] BERDE, P., GEROLA, M., HART, J., HIGUCHI, Y., KOBAYASHI, M., KOIDE, T., LANTZ, B., O'CONNOR, B., RADOSLAVOV, P., SNOW, W., AND PARULKAR, G. ONOS: Towards an open, distributed SDN OS. In *HotSDN* (2014).

[8] BJØRNER, N., JUNIWAL, G., MAHAJAN, R., SESHIA, S. A., AND VARGHESE, G. ddNF: An efficient data structure for header spaces. In *HVC* (2016).

[9] CANINI, M., VENZANO, D., PEREŠÍNI, P., KOSTIĆ, D., AND REXFORD, J. A NICE way to test openflow applications. In *NSDI* (2012).

[10] CHEN, Z. Veriflow system analysis and optimization. Master's thesis, University of Illinois Urbana-Champaign, 2014.

[11] COUSOT, P., AND COUSOT, R. Systematic design of program analysis frameworks. In *POPL* (1979).

[12] DAVEY, B. A., AND PRIESTLEY, H. A. *Introduction to Lattices and Order*, second ed. Cambridge University Press, 2002.

[13] DEAN, J. Underneath the covers at Google, 2008. Google I/0.

[14] DELTA-NET. https://github.com/delta-net/datasets.

[15] FAYAZ, S. K., AND SEKAR, V. Testing stateful and dynamic data planes with FlowTest. In *HotSDN* (2014).

[16] FAYAZ, S. K., YU, T., TOBIOKA, Y., CHAKI, S., AND SEKAR, V. BUZZ: Testing context-dependent policies in stateful networks. In *NSDI* (2016).

[17] FOGEL, A., FUNG, S., PEDROSA, L., WALRAED-SULLIVAN, M., GOVINDAN, R., MAHAJAN, R., AND MILLSTEIN, T. A general approach to network configuration analysis. In *NSDI* (2015).

[18] FOSTER, N., KOZEN, D., MILANO, M., SILVA, A., AND THOMPSON, L. A coalgebraic decision procedure for NetKAT. In *POPL* (2015).

[19] FRENETIC TOPOLOGIES. https://github.com/frenetic-lang/pyretic/tree/master/pyretic/evaluations. Tree ac942315136e.

[20] GEMBER-JACOBSON, A., VISWANATHAN, R., AKELLA, A., AND MAHAJAN, R. Fast control plane analysis using an abstract representation. In *SIGCOMM* (2016).

[21] HARTERT, R., VISSICCHIO, S., SCHAUS, P., BONAVENTURE, O., FILSFILS, C., TELKAMP, T., AND FRANCOIS, P. A declarative and expressive approach to control forwarding paths in carrier-grade networks. In *SIGCOMM* (2015).

[22] ISO. *International Standard ISO/IEC 14882:2014(E) Programming Language C++*. 2014.

[23] JAYARAMAN, K., BJØRNER, N., OUTHRED, G., AND KAUFMAN, C. Automated analysis and debugging of network connectivity policies. Tech. rep., Microsoft Research, 2014.

[24] JEFFREY, A., AND SAMAK, T. Model checking firewall policy configurations. In *POLICY* (2009).

[25] KAZEMIAN, P., CHANG, M., ZENG, H., VARGHESE, G., MCKEOWN, N., AND WHYTE, S. Real time network policy checking using header space analysis. In *NSDI* (2013).

[26] KAZEMIAN, P., VARGHESE, G., AND MCKEOWN, N. Header space analysis: Static checking for networks. In *NSDI* (2012).

[27] KHURSHID, A., ZOU, X., ZHOU, W., CAESAR, M., AND GODFREY, P. B. VeriFlow: Verifying network-wide invariants in real time. In *NSDI* (2013).

[28] KNIGHT, S., NGUYEN, H., FALKNER, N., BOWDEN, R., AND ROUGHAN, M. The internet topology zoo. *IEEE Journal on Selected Areas in Communications 29*, 9 (Oct. 2011), 1765–1775.

[29] LANTZ, B., HELLER, B., AND MCKEOWN, N. A network in a laptop: Rapid prototyping for software-defined networks. In *SIGCOMM Workshop on Hot Topics in Networks* (2010).

[30] LIBRA. https://github.com/jvimal/libra-data.

[31] LIN, P., HART, J., KRISHNASWAMY, U., MURAKAMI, T., KOBAYASHI, M., AL-SHABIBI, A., WANG, K.-C., AND BI, J. Seamless interworking of SDN and IP. In *SIGCOMM* (2013).

[32] LINGUAGLOSSA, L. *Two challenges of Software Networking: Name-based Forwarding and Table Verification*. PhD thesis, Paris Diderot University, France, 2016.

[33] LOPES, N. P., BJØRNER, N., GODEFROID, P., JAYARAMAN, K., AND VARGHESE, G. Checking beliefs in dynamic networks. In *NSDI* (2015).

[34] MAI, H., KHURSHID, A., AGARWAL, R., CAESAR, M., GODFREY, P. B., AND KING, S. T. Debugging the data plane with Anteater. In *SIGCOMM* (2011).

[35] MALDONADO-LOPEZ, F. A., CALLE, E., AND DONOSO, Y. Detection and prevention of firewall-rule conflicts on software-defined networking. In *RNDM* (2015).

[36] MCGEER, R. Verification of switching network properties using satisfiability. In *ICC* (2012).

[37] MCKEOWN, N. How SDN will shape networking, 2011. Open Networking Summit.

[38] MCKEOWN, N., ANDERSON, T., BALAKRISHNAN, H., PARULKAR, G., PETERSON, L., REXFORD, J., SHENKER, S., AND TURNER, J. OpenFlow: Enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review 38*, 2 (Mar. 2008), 69–74.

[39] NARAYANA, S., TAHMASBI, M., REXFORD, J., AND WALKER, D. Compiling path queries. In *NSDI* (2016).

[40] NELSON, T., BARRATT, C., DOUGHERTY, D. J., FISLER, K., AND KRISHNAMURTHI, S. The Margrave tool for firewall analysis. In *LISA* (2010).

[41] NUNES, B. A. A., MENDONCA, M., NGUYEN, X. N., OBRACZKA, K., AND TURLETTI, T. A survey of software-defined networking: Past, present, and future of programmable networks. *IEEE Communications Surveys Tutorials 16*, 3 (2014), 1617–1634.

[42] ONOS DEPLOYMENTS. https://wiki.onosproject.org/display/ONOS/Global+SDN+Deployment+Powered+by+ONOS.

[43] PFAFF, B., PETTIT, J., KOPONEN, T., JACKSON, E. J., ZHOU, A., RAJAHALME, J., GROSS, J., WANG, A., STRINGER, J., SHELAR, P., AMIDON, K., AND CASADO, M. The design and implementation of open vswitch. In *NSDI* (2015).

[44] PLOTKIN, G. D., BJØRNER, N., LOPES, N. P., RYBALCHENKO, A., AND VARGHESE, G. Scaling network verification using symmetry and surgery. In *POPL* (2016).

[45] QUAGGA. http://www.nongnu.org/quagga/.

[46] ROUTE VIEWS. http://www.routeviews.org/.

[47] SDN-IP APPLICATION. https://wiki.onosproject.org/display/ONOS/SDN-IP.

[48] SON, S., SHIN, S., YEGNESWARAN, V., PORRAS, P. A., AND GU, G. Model checking invariant security properties in OpenFlow. In *ICC* (2013).

[49] SPRING, N., MAHAJAN, R., AND WETHERALL, D. Measuring ISP topologies with Rocketfuel. In *SIGCOMM* (2002).

[50] STOENESCU, R., POPOVICI, M., NEGREANU, L., AND RAICIU, C. SymNet: Scalable symbolic execution for modern networks. In *SIGCOMM* (2016).

[51] TAHMASBI, M. personal communication.

[52] UTTING, M., PRETSCHNER, A., AND LEGEARD, B. A taxonomy of model-based testing approaches. *Software Testing, Verification & Reliability 22*, 5 (Aug. 2012).

[53] VELNER, Y., ALPERNAS, K., PANDA, A., RABINOVICH, A., SAGIV, M., SHENKER, S., AND SHOHAM, S. Some complexity results for stateful network verification. In *TACAS* (2016).

[54] XIE, G. G., ZHANM, J., MALTZ, D. A., ZHANG, H., GREENBERG, A., HJALMTYSSON, G., AND REXFORD, J. On static reachability analysis of ip networks. In *INFOCOM* (2005).

[55] YANG, H., AND LAM, S. S. Real-time verification of network properties using atomic predicates. In *ICNP* (2013).

[56] YUAN, D., LUO, Y., ZHUANG, X., RODRIGUES, G. R., ZHAO, X., ZHANG, Y., JAIN, P. U., AND STUMM, M. Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In *OSDI* (2014).

[57] YUAN, L., MAI, J., SU, Z., CHEN, H., CHUAH, C.-N., AND MOHAPATRA, P. FIREMAN: A toolkit for firewall modeling and analysis. In *SP* (2006).

[58] ZENG, H., KAZEMIAN, P., VARGHESE, G., AND MCKEOWN, N. Automatic test packet generation. In *CoNEXT* (2012).

[59] ZENG, H., ZHANG, S., YE, F., JEYAKUMAR, V., JU, M., LIU, J., MCKEOWN, N., AND VAHDAT, A. Libra: Divide and conquer to verify forwarding tables in huge networks. In *NSDI* (2014).

# A Illustration of Boolean lattice

Delta-net is based on ideas from lattice theory.[5] In particular, Delta-net leverages the concept of atoms, a form of mutually disjoint ranges that make it possible to analyze all Boolean combinations of IP prefix forwarding rules in a network. The fact that atoms induce a Boolean lattice is illustrated by the Hasse diagram [12] in Figure 9 where atoms (depicted in bold) correspond to $\alpha_0$, $\alpha_1$ and $\alpha_2$ in Figure 5, respectively.

[5] For interested readers, a good introduction to lattice theory, whose applications in computer science are pervasive, can be found in [12]

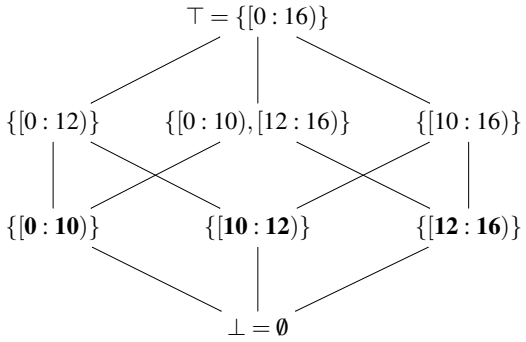



Figure 9: Boolean lattice induced by the atoms (bold) in Figure 5, assuming 4-bit numbers for simplicity.

# B Proof of complexity analysis

In this appendix, we sketch the proof of the asymptotic worst-case time complexity of Algorithm 1 and 2.

*Proof of Theorem 1.* We analyze INSERT_RULE. Each atom split (line 2-9) requires copying the owner information from an existing atom to a newly created atom. For insertion of $R$ rules, resulting in $K$ atoms, this requires $O(RK)$ steps in the worst-case. In each insertion, the adjustment of labels and retrieval of the balanced binary search tree (BST) (line 12) are amortized constant-time operations per atom. Inserting each rule into the BST and finding the highest-priority rule per atom (line 14) are $O(\log M)$. By the loop (line 10-23), we get $O(RK + RK \log M) = O(RK \log M)$, concluding the proof. A similar argument proves the claim for REMOVE_RULE. □

# C Comparison to previous data sets

In this appendix, we discuss how our data sets compare to previous ones used in the experimental evaluation of Veriflow [27].

In particular, it is natural to ask how our RF 1755 data set in Table 2 compares to the one used in a previous Veriflow experiment [27], which was constructed from 5 million BGP RIP entries and by 'replaying' 90,000 BGP updates. While the resulting total number of IP prefix rules in the original RF 1755 data set is not reported, the authors of the Veriflow paper note that "[t]he largest number of ECs (equivalence classes) affected by a single rule was 574; the largest verification latency was 159.2 *ms* due to an update affecting 511 ECs." For our experiments, we expect this number to be different, since we had to generate a new data set.

Running Veriflow-RI (§ 4.3.1) on our RF 1755 data set, we find that the maximum number of affected ECs on rule insertions is 319,681, which is significantly larger than the original experimental evaluation of Veriflow [27].

| Data set | Memory usage (*MB*) | |
|---|---|---|
| | Veriflow-RI | Delta-net |
| Berkeley | 1,089 | 6,208 |
| INET | 9,776 | 63,563 |
| RF 1755 | 2,713 | 16,937 |
| RF 3257 | 5,882 | 40,716 |
| RF 6461 | 5,920 | 39,481 |
| Airtel 1 | 7 | 61 |
| Airtel 2 | 9 | 74 |
| 4Switch | 154 | 785 |

Table 5: Memory usage of Delta-net and Veriflow-RI.

## D Memory usage

In this appendix, we report the detailed memory consumption of Delta-net (§ 3) and Veriflow-RI (§ 4.3.1) using our eight data sets (§ 4.2, see Table 2).

Table 5 quantifies the memory usage of Delta-net and Veriflow-RI. In all cases, Delta-net consumes between 5 and 7 times more space than Veriflow-RI. This increase in memory consumption is offset, however, by the fact that Delta-net keeps track of the forwarding behaviour of all packets, and as a result can check properties that Veriflow-RI cannot. Nevertheless, as discussed for future work (§ 5), we are actively working on asymptotically reducing the memory consumption of Delta-net.