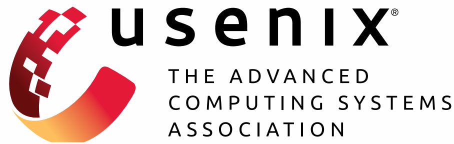

APKeep: Realtime Verification for Real Networks

Peng Zhang and Xu Liu, *Xi'an Jiaotong University;* Hongkun Yang, *Google;* Ning Kang, Zhengchang Gu, and Hao Li, *Xi'an Jiaotong University*

https://www.usenix.org/conference/nsdi20/presentation/zhang-peng

This paper is included in the Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI '20)

February 25–27, 2020 • Santa Clara, CA, USA

978-1-939133-13-7

Open access to the Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI '20) is sponsored by

APKeep: Realtime Network Verification for Real Networks

Peng Zhang*, Xu Liu*, Hongkun Yang†, Ning Kang*, Zhengchang Gu*, and Hao Li*
*Xi'an Jiaotong University, †Google

Abstract

Realtime network verification ensures the correctness of network by incrementally checking data plane updates in real time (e.g., $<$ 1ms per rule update). Even state-of-the-art methods can already achieve sub-millisecond verification time, such speed is achieved mostly for pure IP forwarding devices, and is unrealistic for real-world networks, due to two reasons. (1) Their network models cannot express the forwarding behavior of real devices, which have various functions including IP forwarding, ACL, NAT, policy-based routing, etc. (2) Their update algorithms do not scale in space and/or time: multi-field rules (e.g., ACL rules) can make these tools run out of memory and/or incur long verification time. To scale realtime verification to real networks, we propose APKeep based on a new modular network model that is expressive for real devices, and propose new algorithms that can achieve low memory cost and fast update speed at the same time. Our experiments show that for real-world update traces consisting of IP forwarding rules and ACL rules, existing methods either run out of memory or incur a prohibitively long verification time, while APKeep still achieves a sub-millisecond verification time. We also show that APKeep can verify an update of NAT rule mostly in less than 1 millisecond.

1 Introduction

Computer networks are prone to faults due to protocol misconfigurations, software bugs, and hardware failures [7,17,26,35]. Manually troubleshooting the faults often costs a network downtime up to several hours [7]. How to prevent network faults by ensuring network correctness becomes a fundamental problem posed to network operators and researchers.

Network verification seeks to automatically check network correctness at both control plane [9, 12, 13, 16, 18, 19, 30] and data plane [10, 15, 20, 22–24, 27, 36–40]. Compared to control plane verification which focuses on detecting protocol misconfigurations, data plane verification directly checks the data plane, which is closer to the actual forwarding behaviors of packets, and thus can catch a broader range of faults due to switch software bugs and hardware failures.

More recently, *realtime data plane verification* allows operators to check the correctness of data plane as it updates in realtime [20, 22, 24, 37–39]. To achieve this, realtime data plane verifiers often partition packets into *equivalence classes (ECs)*, and maintain a model of forwarding behavior for these ECs. When the data plane updates, they *incrementally* update the model, and check the updated model against correctness properties.

State-of-the-art realtime data plane verifiers have already achieved sub-millisecond verification time [20, 24]. However, such speed is mostly achieved for pure forwarding devices. For real devices consisting of various functions other than forwarding, these verifiers exhibit two fundamental limitations.

Network model is *not expressive* for real devices. Apart from IP forwarding, real devices have many other functions including access control list (ACL), network address translation (NAT), etc., which are composed in specific orders to implement various processing logic. For example, inside a typical router, multiple ACLs can be chained and applied at multiple ports to filter inbound and/or outbound packets [1]. Some routers may perform NAT on packets matching an ACL. Tools like VeriFlow [24] and Delta-net [20] assume simple models which only express forwarding functions. Models of NetPlumber [22] and AP Verifier [38, 39] can express more functions, but are hard to extend. For example, most vendors provide variants of policy-based routing [8], and adding such a feature requires heavy modification of their models. Even for the same set of functions, different devices may also have different pipelines, and writing a model for each of them is clearly not scalable.

Verification algorithms are *not scalable* for real devices. Range EC-based methods like VeriFlow and Delta-net represent each EC as a range of packet headers, thereby achieving a fast verification speed for IP forwarding rules. For example, Delta-net can check an update of IP forwarding rule in tens of microseconds on average. However, when there are multi-

field rules, e.g., ACL rules, range EC-based methods may suffer from the problem of *EC explosion*, where the number of ECs grows exponentially with the number of multi-field rules. We find that for a real-world dataset consisting of only 686 ACL rules, an open-source version of VeriFlow and our multi-field extension of Delta-net can create up to 15 million ECs, causing either prohibitively long verification times or memory overflows.

AP Verifier computes the minimum number of ECs with respect to the network behavior. The downside, however, is that the update of ECs is more difficult, and can cost up to 10 milliseconds [37].

To overcome the above limitations and bring realtime network verification closer to the real world, this paper presents APKeep, a new realtime data plane verifier.

APKeep builds on a new network model that is modular and expressive. It models networks in a granularity of logical functions instead of physical devices. Each function, e.g., forwarding, filtering, rewriting, is modeled as a logically-independent *element*, which holds a set of logical *ports* corresponding to different actions on packets. APKeep views packets forwarded to the same port (i.e., undergoing the same actions) at each element as an EC, and encodes each EC with a logical *predicate*. The modularity of our model makes it easy to support common functions and vendor-specific compositions of functions in real devices. In addition, it also reduces the update scope and makes the update more efficient.

APKeep uses novel algorithms to compute and maintain the minimum number of ECs in realtime. A key reason for EC explosion of existing methods is that they create ECs based on the match fields of rules, resulting in a lot of unnecessary ECs with the same forwarding behavior. In addition, they cannot compress these ECs after creation. APKeep significantly reduces the number of ECs based on two principles. *(1) Creating ECs only when necessary.* APKeep creates ECs only when it needs more ECs to express new forwarding behaviors. *(2) Merging ECs when possible.* APKeep tracks the forwarding behavior of each EC, and merges multiple ECs with the same forwarding behavior. *We proved that by applying the above principles, APKeep always maintains the minimum number of ECs during update.*

In summary, our contribution is three-fold:

- We introduce a new network model that is modular and expressive for modeling real network devices.
- We design APKeep, which uses novel algorithms to fast update the network model for realtime verification.
- We show APKeep achieves a sub-millisecond verification time for update traces consisting of IP forwarding rules, ACL rules, and NAT rules.

Roadmap. We present the design overview (§ 2) and details (§ 3) of APKeep, followed by a case study (§ 4). Then, we show the experiment results (§ 5). After discussing related work (§ 6) and potential issues (§ 7), we conclude (§ 8).

2 Design Overview

This section overviews the design of APKeep. We will first introduce the network model that APKeep builds on, and then show how APKeep can fast update the model.

2.1 The Modular Network Model

To achieve realtime network verification for real networks, the network model should satisfy three key requirements: (1) *expressive* for common functions in real devices, e.g., IP forwarding, ACL, NAT, policy-based routing, etc.; (2) *extensible* for different devices with different vendor-specific implementations of these functions; (3) *efficient* to update for achieving realtime verification.

We propose *Port-Predicate Map (PPM)*, a new network model that meets all the above requirements. To demonstrate how PPM works, we use the example network shown at the top-left corner of Figure 1. In this network, switch *C* has four functions or modules (two ACLs, one forwarding, and one NAT), each having its own rules. If packets arrive at *port*1, two ACLs *ACL1* and *ACL2* are applied in sequence; if they arrive at *port*2, only *ACL1* is applied. Then, the packets will be sent to an output port according to the forwarding rules. If the output port is *port*5, packets will go through an NAT.

As an alternative, we could model a device as a monolithic box. This approach has the following drawbacks. First, it will be difficult to extend the model for new functionalities. For example, a device from another vendor may have a different chaining of modules (e.g., NAT before IP forwarding), or a new function (e.g., overriding IP forwarding with user policies). Then, we need to compose another device model. In addition, it will also make the update inefficient. For example, suppose a rule is inserted into *ACL1*, then we need to update the ECs allowed by the two input ports.

Element. Instead of modeling a network as a set of devices, PPM models at a granularity of *element*, defined as a logically-independent function (e.g., IP forwarding, ACL, or NAT). Each element has its own set of *rules*, and holds a set of logical *ports*. Different from physical ports, i.e., interfaces, logical ports represent generic actions including “output to VLAN 10”, “permit SSH traffic”, “rewrite dstIP to 10.0.0.1”. This allows elements to express a broad range of functions other than IP forwarding. In specific, an element holds one port for each distinct action of rules in the element, and a special port for the default action is reserved for packets not matching any rules. When a packet arrives at an element, it will be “forwarded to” exactly one port of the element, i.e., taking the actions of that port. Currently, PPM supports three types of elements, and more types can be added in the future.

- A **forwarding element** has rules that match IP prefixes and whose actions are “output packets to a specific set of interfaces”. A forwarding element holds one port for

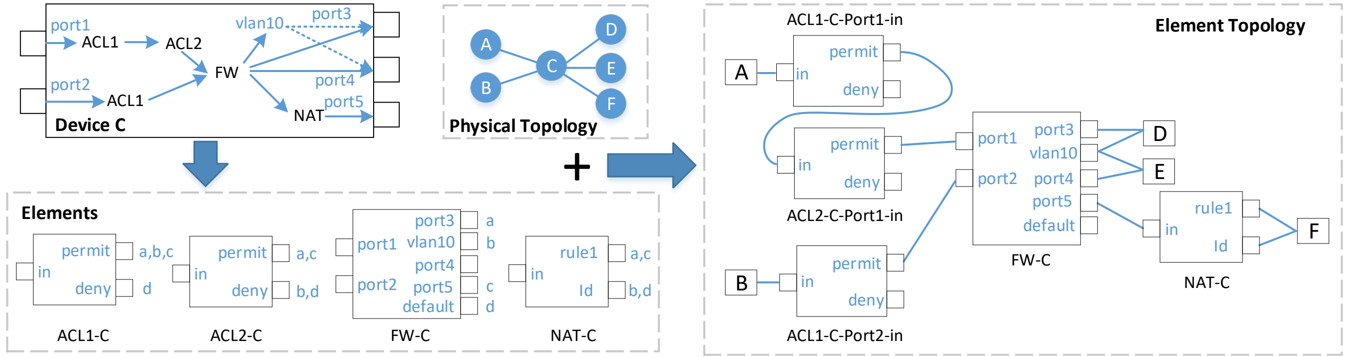

Figure 1: An example showing how APKeep divides devices into elements.

each distinct set of interfaces, and a *default* port for the default action, e.g., dropping packets.

- A **filtering element** has rules that match 5-tuples and whose actions are either "permit" or "deny". A filtering element holds exactly two ports: *permit* and *deny*.
- A **rewriting element** has rules which match 5-tuples and whose actions are "rewrite a specific header field to a specific value". A rewriting element holds one port for each distinct rewriting action, and an *id* port corresponding to no packet rewrite.

When a device has multiple functions, we break it into multiple elements. As shown in the left bottom of Figure 1, device C breaks into a forwarding element `FW-C`, two filtering elements `ACL1-C`, `ACL2-C`, and a rewriting element `NAT-C`.

Equivalence Class. Let $\mathcal{E}$ be the set of all elements in the network, and $\mathcal{H}$ be the set of all packet headers. For each header $h \in \mathcal{H}$ and element $e \in \mathcal{E}$, let $Port_e(h)$ be the port that h would be "forwarded to", assuming h has been received by e. Then, we have the following definition for equivalence class (EC).

Definition 1. *We say $\mathcal{C} = \{c_1, c_2, \ldots, c_n\}$ is a set of equivalence classes (ECs) with respect to element set $\mathcal{E}$ and header set $\mathcal{H}$ if: (1) $c_i \wedge c_j = \emptyset, i \neq j$; (2) $\vee_{i=1}^{n} c_i = \mathcal{H}$; (3) $\forall h_1, h_2 \in \mathcal{H}$, $h_1 \neq h_2$: $\exists c \in \mathcal{C}$, $h_1, h_2 \in c \Rightarrow \forall e \in \mathcal{E}, Port_e(h_1) = Port_e(h_2)$ [1]. We say $\mathcal{C}$ is the minimum set of ECs if it is the smallest set satisfying the above conditions.*

APKeep encodes an EC with a logical *predicate*, i.e., Boolean formula. The reason to use predicate instead of range as in [20, 24] is that a predicate can encode an arbitrary set of packet headers, such that multiple range-based ECs having the same forwarding behavior can be represented as a single predicate. This allows APKeep to merge ECs with the same forwarding behavior, thereby avoiding explosion of ECs (§ 2.2).

Port-Predicate Map. For each predicate c and each element e, let $Port_e(c) = Port_e(h), \forall h \in c$. Suppose $p = Port_e(c)$, then we say port p *holds* predicate c. Define the *predicate set* of port p as: $Pred(p) = \{c \in \mathcal{C} | Port_e(c) = p, e \in \mathcal{E}\}$. We can see that $Pred$ is a map from port to predicates, which encodes the network forwarding behavior: given a packet h at element e, suppose it belongs to predicate c, then h will be forwarded by e to the port p satisfying $c \in Pred(p)$.

Element Topology. PPM uses the *element topology* to describe how elements are chained to process packets in the network. The right of Figure 1 shows the element topology of the example. First, each node represents an "application" of the corresponding element. For example, since `ACL1-C` is applied to $port1$ and $port2$, there are two nodes `ACL1-C-Port1-in` and `ACL1-C-Port2-in`. The forwarding element `FW-C` is applied once, and thus it corresponds to a single node. Creating a separate node for each application allows elements to be agnostic of input ports where packets are received. Second, each node has a set of ports, each holding a set of predicates, in the same way as its corresponding element. Thus, we only need to update a single element rather than all its nodes. For example, when a rule is inserted into $ACL1$, we only update the element `ACL1`, rather than its two nodes. Third, nodes are connected based on the physical topology, and how the elements are applied inside devices. For example, a port of A is connected to $port1$ of C in the network topology. Then, in the element topology, the port of A connects to the *in* port of `ACL1-C-Port1-in`, whose *permit* port connects to the *in* port of `ACL2-C-Port1-in`, and its *permit* port connects to the $port1$ of `FW-C`. The element topology will be used to construct forwarding graphs for verification (§ 3.3).

As shown above, PPM achieves modularity by breaking the composite functions inside a device into logically-independent elements. This brings the following benefits.

Expressiveness. Using the three types of elements as building blocks, PPM can express the forwarding, ACL, and NAT functions. Besides that, we will show how PPM can express the policy-based routing function offered by a major device vendor (§ 4).

Extensibility. Even most devices share roughly the same set of functions, the implementations and compositions of these functions are often vendor-specific. Writing a model for each

[1] Condition (3) says that for each h_1 and h_2 in $\mathcal{H}$ such that $h_1 \neq h_2$, we have: if there exists an c in $\mathcal{C}$ such that h_1 and h_2 both belong to c, then for each element e in $\mathcal{E}$, $Port_e(h_1) = Port_e(h_2)$

different device wastes time and effort. PPM models each device at the function level with elements, therefore it is relatively easy to model devices with vendor-specific compositions of functions by properly chaining the elements.

Reduced update scope. First, updates of multiple elements are decoupled, and when a rule is updated, we only need to update the element where the rule is updated, without affecting other elements. For example, multiple ACLs may be chained and applied to an interface. If a rule is inserted to one ACL, we only need to update the element of that ACL. Secondly, the application of elements is decoupled away from the elements themselves. For example, an ACL can be applied to multiple interfaces, and we only need to update the element of the ACL once, instead of updating all these interfaces. As another example, an operator may activate/deactivate an existing ACL on a port, or even migrate an ACL from a port to another [33]. In this case, we do not need to update the element of the ACL, as the forwarding behavior of the element is not affected.

2.2 The Update of Network Model

In the following, we show how APKeep updates the network model using a simple example in Figure 2. As shown in (a), the device has an ACL applied to its input port, followed by a forwarding module. For simplicity, we assume there are two match fields: *dstIP* represented with 2 bits x_1, x_2, and *dstPort* represented with 2 bits y_1, y_2. The forwarding module matches only *dstIP* with longest prefix match, and the ACL matches both *dstIP* and *dstPort* according to priorities (larger number means higher priority). We assume that by default, the ACL denies all packets and the forwarding module forwards all packets to *port*1. Initially, we have one EC a, which appears at the port *port*1 of element `FW`, and the port *deny* of element `ACL`. We will insert two ACL rules $R1$ and $R2$ shown in (b), and two forwarding rules $R3$ and $R4$ shown in (c).

First, we insert an ACL rule $R1$, whose match fields are $x_1x_2 = 0*, y_1y_2 = 00$, as shown at the top of (d). APKeep analyzes how $R1$ will affect the behaviors of element `ACL`. Specifically, APKeep finds $R1$ overrides the default deny rule in the red dashed rectangle. However, since $R1$ also has a deny action, packets in the rectangle will not change. Thus, APKeep does not update the EC a, which still appears at *port*1 of `FW` and *deny* of `ACL`, as shown at the bottom of (d). In contrast, if we create range-based ECs based on match fields, we will split EC a into three ECs, each of which is a rectangle in the header space.

Suppose another ACL rule $R2$ is inserted. Since $R2$ has a lower priority than $R1$, APKeep finds $R2$ can match only the shaded area, where it overrides default deny rule. As a result, packets matching the shaded area will change their behavior from deny to permit. To reflect that change, APKeep decides to transfer those packets from port *deny* to port *permit*. Since the packets are a portion of EC a, it splits a into two ECs, i.e., b for the shaded area, and another one by subtracting b from a. Then, APKeep transfers b to port *permit*, as shown at the bottom of (e). The reason that the EC b can be a non-rectangle area is that ECs are encoded with predicates. Specifically, the match fields of $R1$ and $R2$ can be represented as predicates $\bar{x}_1\bar{y}_1\bar{y}_2$ and $\bar{y}_1$, respectively. Then, b can be calculated by logical operations as $b = \bar{y}_1 \wedge \neg(\bar{x}_1\bar{y}_1\bar{y}_2) = (\bar{y}_1x_1) \vee (\bar{y}_1y_2)$.

Suppose a forwarding rule $R3$ is inserted. $R3$ overrides the default rule in the red dashed rectangle, which changes its port from *port*1 to *port*2. APKeep creates two new EC c and d by splitting a and b, respectively, and transfers them to *port*2, as shown in (f). The insertion of another forwarding rule $R4$ is similar and shown in (g). At this time, APKeep finds two ECs d and f appear at the same port at both elements, meaning that they have the same forwarding behavior – permitted by the ACL and forwarded to *port*2. Thus, APKeep merges d and f into a single EC. Similarly, APKeep merges c and e into a single EC, as shown in (h). The merging of ECs translates into logical disjunction of predicates. For example, d and f are merged into an EC represented by $(\bar{x}_1\bar{x}_2\bar{y}_1y_2) \vee (x_1\bar{x}_2\bar{y}_1)$.

Finally, after inserting $R1$ through $R4$, APKeep creates 4 ECs. In contrast, if we create range-based ECs based on match fields, we will need 10 ECs, one for each rectangle of (h). The above is just an over-simplified example with only two fields which have 3-4 values. In real scenarios, the reduction rate can be as high as 99.99% (see Table 3). Actually, we prove that APKeep always maintains the minimum number of ECs during update (see Theorem 1).

The reason that APKeep can update such a small number of ECs is two-fold: *(1) Creating new ECs only when necessary.* APKeep creates a new EC only when part of an existing EC changes its forwarding behavior and the EC needs to be split into two ECs. In contrast, creating new ECs whenever the match fields of the new rule split some existing ECs will result in many redundant ECs. *(2) Merging ECs whenever possible.* APKeep tracks the forwarding behaviors of ECs, and merges multiple ECs if they have the same forwarding behavior. In contrast, range-based EC presentation mostly does not allow ECs to be merged.

The update of ECs in APKeep is much faster compared to AP Verifier due to the following reason. APKeep can quickly identify the changes of forwarding behaviors, and incrementally update predicates instead of re-computing them (§ 3.2). In contrast, AP Verifier maintains a port predicate for each device port, and computes atomic predicates (minimum number of ECs) based on all port predicates. When a rule is updated, it needs to first update the port predicates, and if new port predicates are created, it re-computes the atomic predicates based on the updated port predicates. We observe an up to 200× speedup in our experiments.

3 Design Details

This section presents the design of APKeep. Figure 3 shows the architecture of APKeep, which consists of three layers:

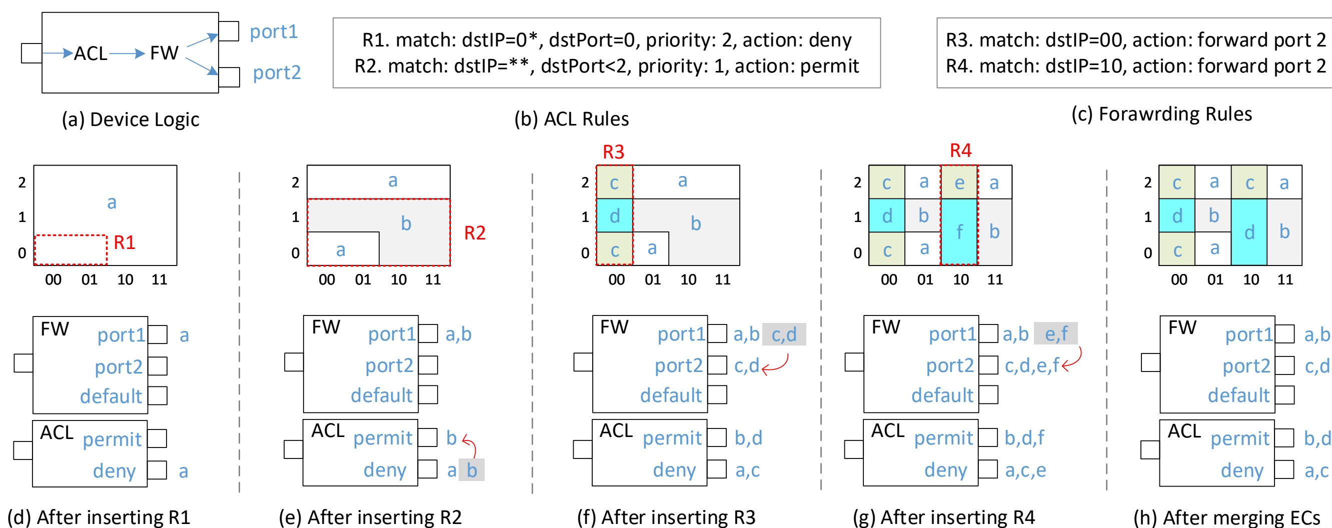

Figure 2: An example of incremental update for rule insertion.

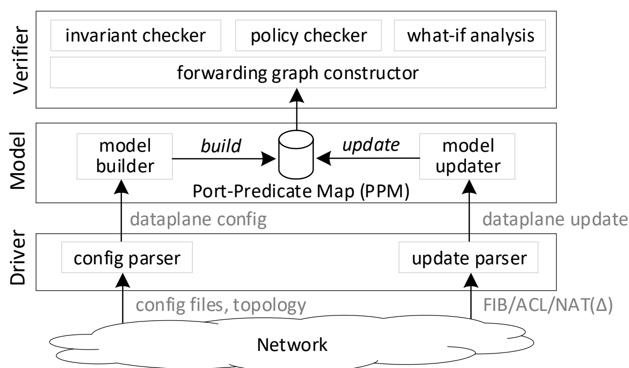

Figure 3: The architecture of APKeep.

The driver layer serves as the interface between network data plane and the model layer. In the bootstrap stage, the *config parser* reads in the network topology and configuration files, and generates the vendor-neutral *data plane config*, describing the configuration of interfaces, VLANs, ACLs, etc. for each device. The *update parser* fetches the FIB/ACL/NAT (changes) from each device and generates *data plane updates*, including insertion/deletion/modification of rules.

The model layer is the core of APKeep system. The *model builder* constructs PPM model by creating all the elements based on the data plane config. The *model updater* continuously updates the PPM model by processing each data plane update in sequence.

The verifier layer hosts verification applications on top of the model layer. The *forwarding graph constructor* generates forwarding graphs based on the PPM model, and on top of the graphs, various applications can be deployed to check network invariants, operator policies, or conduct what-if analysis.

In the following, we show how APKeep builds and updates the PPM model, and performs verification. Then, we show how APKeep supports packet rewrites, and present some optimization techniques.

3.1 Building PPM

For each device, APKeep constructs a device model based on its configuration of interfaces, ACLs, NAT, etc., and decomposes the device model into a set of elements. Currently, APKeep offers three types of elements, i.e., forwarding element, filtering element, and rewriting element. Initially without any rules, a forwarding element has a $default$ port; a filtering element has a *permit* port and a *deny* port; a rewriting element has an *id* port. For forwarding and rewriting elements, more ports can be created on-the-fly during rule insertions.

After creating elements, APKeep constructs the *element topology* by augmenting the physical topology with intra-device element connections, based on how elements are composed inside the device. For example, if an ACL $ACL1$ is declared to filter inbound traffic at port $port1$, then there is a connection from the *permit* port of $ACL1$ to the $port1$ port of the forwarding element.

Initially there is only one `True` predicate, standing for the set of all possible packets. For each element, the `True` predicate is held by its $default$, *deny*, or *id* port, depending on the element type. APKeep initializes the predicate set $Pred(p)$ (§ 2.1) to {`True`} if p is $default$, *deny*, or *id* port, and to empty set otherwise.

3.2 Updating PPM

For each rule update, APKeep updates the PPM using three steps: (1) encoding the match fields of the rule, (2) identifying the changes of forwarding behavior, and (3) updating the predicates and the map from port to predicates. The following only shows the case for rule insertion, and rule deletion differs only slightly in Step (2). Rule modification can be seen as a pair of rule deletion and insertion.

Let r be the rule to be inserted, specified as a 3-tuple $(priority, match, action)$, and let e be the element where r

Algorithm 1: `IdentifyChangesInsert`(r, $\mathcal{R}$)

Input: r: the newly inserted rule; $\mathcal{R}$: the list of existing rules, sorted by decreasing priorities.

Output: C: the set of changes due to the insertion of rule r.

```
1  C ← {};
2  r.hit ← r.match;
3  foreach r' ∈ R do
4      if r'.prio > r.prio and r'.hit ∧ r.hit ≠ ∅ then
5          r.hit ← r.hit ∧ ¬r'.hit;
6      if r'.prio < r.prio and r'.hit ∧ r.hit ≠ ∅ then
7          if r'.port ≠ r.port then
8              C ← C ∨ {(r.hit ∧ r'.hit, r'.port, r.port)};
9          r'.hit ← r'.hit ∧ ¬r.hit;
10 Insert r into R;
11 return C;
```

is inserted.

Step 1. Encoding match fields. Assume each packet header has h bits, each of which can be represented as a Boolean variable. Then, the match field of a rule corresponds to a set of packet headers, and can be represented as Boolean formula of h variables. For example, an IP match field of 128.0.0.* can be represented as $x_1 \wedge \bar{x}_2 \wedge \cdots \wedge \bar{x}_{24}$. We adopt the methods of [37] to encode the Boolean formulas of match fields based on Binary Decision Diagram (BDD [11]). BDD is a data structure that can canonically represent Boolean formulas, and it allows efficient logical operations including conjunction ($\wedge$), disjunction ($\vee$), and negation ($\neg$). By encoding the match fields with BDDs, we can efficiently compute and update predicates leveraging these logical operations. We use *r.match* to denote the match fields of r, encoded with BDD.

Step 2. Identifying changes. This step identifies the changes of forwarding behavior at element e, by analyzing how the insertion of r affects existing rules of e. Here, a behavior change takes the form of $(\delta, from, to)$, meaning packets satisfying predicate δ, which are originally forwarded to port $from$, will now be forwarded to port to. Note that this step is locally performed at e.

Before introducing the algorithm, we define the *hit* and *port* fields for each rule. First, note that multiple rules may have overlapping match fields, and packets will take the action of the rule with the highest priority. Thus, some headers in *r.match* may not "hit" rule r due to the presence of some higher-priority rules. To represent the headers that actually "hit" a rule, we define the *hit field* for each rule r as:

$$r.hit \triangleq \neg(\vee_{r'.prio > r.prio} r'.match) \wedge r.match \tag{1}$$

If $h \in r.hit$, we know that h will take the action of r. Initially when there is only one default rule, the hit field of the default rule is equal to its match field, i.e., `True`. Second, recall that each element has a port corresponding to each distinct action of rules in the element. We use *r.port* to denote the port corresponding to the action field of r. As an example, if r is a forwarding rule whose action field is "output to interface eth0/0", then $r.port = eth0/0$.

Algorithm 1 summarizes the procedure to identify the set of all behavior changes when a rule is inserted. It calculates the hit field *r.hit* by subtracting the match fields of higher-priority rules from *r.match* (Line 3-5), and identifies all behavior changes by analyzing how *r.hit* "overrides" lower-priority rules with different ports (Line 6-9). The algorithm for rule deletion differs only slightly and is not given here.

Step 3. Updating predicates. In this stage, APKeep takes the set of behavior changes caused by the inserted rule, denoted by C, and computes the set of transferred predicates, denoted by D. The process is summarized in Algorithm 2. In order to track which ports hold a given predicate, the algorithm maintains a map *Port* from each predicate c to the set of ports holding c, defined as $Port(c) = \{Port_e(c) | e \in \mathcal{E}\}$. We term $Port(c)$ as the *port set* of predicate c.

Initially, the set of transferred predicates D is set to empty (Line 1). For each change $(\delta, from, to)$, we iterate over each predicate p in the predicate set of $from$, and check whether p overlaps with δ (Lines 2-4). If so, we further perform the following three steps (Lines 5-10).

(1) Splitting predicates. In this step, we check whether p belongs to δ (Line 5). If not so, we need to split p into two new predicates $p \wedge \delta$ and $p \wedge \neg\delta$. by invoking the `Split` function (Line 6). As shown in Lines 11-17, the function `Split`$(p, p1, p2)$ first updates the predicate set of each *port* in $Port(p)$, by replacing p with $p1$ and $p2$ (Lines 12-13). Then, it initializes the port set of $p1$ and $p2$ with that of p (Lines 14-15). Finally, it updates the set of transferred predicates if needed (Lines 16-17).

(2) Transferring predicates. This step transfers the predicate $p \wedge \delta$ from port $from$ to port to by invoking the `Transfer` function (Line 7), as shown in Lines 18-22.

(3) Merging predicates. This step checks whether each predicate p' held by port to has the same port set with p (Line 8). If so, p' and p have the same forwarding behavior, and we merge them into a new predicate $p \vee p'$, by invoking the `Merge` function (Line 9), as shown in Lines 23-28.

After the above three steps, we update δ by subtracting p from it, and proceed to the next predicate of port $from$ (Line 10).

Theorem 1. *APKeep maintains the minimum set of equivalence classes after each rule update.*

The proof is given in Appendix A.

3.3 Verification

Checking Invariants. APKeep can check network invariants including loop-freedom and blackhole-freedom, which are defined as follows.

Algorithm 2: `Update`(C)

```
Input: C: the set of changes identified in the first stage.
Output: D: the set of transferred predicates.
1  D ← {};
2  foreach (δ, from, to) ∈ C do
3      foreach p ∈ Pred(from) do
4          if p ∧ δ ≠ ∅ then
5              if p ∧ δ ≠ p then
6                  Split(p, p ∧ δ, p ∧ ¬δ);
7              Transfer(p ∧ δ, from, to);
8              if ∃p' ≠ p, Port(p') = Port(p) then
9                  Merge(p, p', p ∨ p');
10             δ ← δ ∧ ¬p;
11 Function Split(p, p1, p2):
12     foreach port ∈ Port(p) do
13         Pred(port) ← Pred(port) ∪ {p1, p2} \ {p};
14     Port(p1) ← Port(p);
15     Port(p2) ← Port(p);
16     if p ∈ D then
17         D ← D ∪ {p1, p2} \ {p};
18 Function Transfer(p, from, to):
19     Pred(from) ← Pred(from) \ {p};
20     Pred(to) ← Pred(to) ∪ {p};
21     Port(p) ← Port(p) ∪ {to} \ {from};
22     D ← D ∪ {p};
23 Function Merge(p1, p2, p):
24     foreach port ∈ Port(p1) do
25         Pred(port) ← Pred(port) ∪ {p} \ {p1, p2};
26     Port(p) ← Port(p1);
27     if p1 ∈ D or p2 ∈ D then
28         D ← D ∪ {p} \ {p1, p2};
29 return D;
```

- **Loop.** A packet traverses the same device for the second time, without being modified.
- **Blackhole.** A packet arrives at a device but does not match any forwarding rule.

Similar to Delta-net, APKeep checks invariants by constructing and traversing a *delta forwarding graph (DFG)*, a graph with each edge labeled with the ECs allowed on the edge. The difference is that APKeep updates the PPM model rather than DFG, and only constructs DFG based on the PPM model when checking invariants. Specifically, given a set of transferred predicates, APKeep constructs the DFG by adding the transferred predicates and the corresponding edges on the element topology. Then, APKeep traverses the DFG with a set of predicates P, which is initialized to the transferred predicates. When an edge is visited, P is intersected with the set of predicates on that edge. The traversal terminates when P becomes empty, reaching an edge with no next hop (blackhole detected), or the same node is visited twice (loop detected). The construction and traversal algorithms of DFG are given in Appendix B.

Checking policies. Operators may need to check user-defined policies such as hosts in a specific prefix can or cannot access a web server, traffic from subnet1 to subnet2 should pass the firewall, etc. We show how APKeep can support this task. Here, we define a policy as a pair of *match condition* and *path constraint*, where the match condition can be specified by header fields (e.g., 5-tuple), and a path constraint can be specified by a regular expression. Given a policy, APKeep can convert its match condition into a *policy predicate*, i.e., a BDD denoted as q, and its path constraint into an automata denoted as A. APKeep can check whether the policy is satisfied after an update as follows.

Let D be the transferred predicates after an update. APKeep computes a new set of predicates $D_q \leftarrow \{\delta \in D | \delta \wedge q \neq false\}$, and constructs the DFG G_q based on D_q. Then, APKeep traverses G_q while updating an instance of automata A for each $p_i \in D_q$, denoted as A_i. Specifically, APKeep updates the automata A_i if the predicate p_i visits a new node in DFG. The policy is satisfied if after traversal, all the automata enter the absorbing states; otherwise, the policy is violated. Note here multiple policies can be checked in parallel, and for each policy, the updating of each automata can also be parallelized.

What-if analysis. Operators can use APKeep to conduct "what-if analysis", e.g., will the invariants break if a specific link fails? APKeep answers such a query by retrieving all the predicates traversing the link, constructing a DFG using these predicates, and traversing the DFG to check invariants. The time to answer such a query heavily depends on the total number of ECs. We will show APKeep achieves a much shorter running time than Delta-net (§ 5.4).

3.4 Supporting Packet Rewrites

APKeep supports packet rewrites with *rewriting elements*. A rewriting element consists of a list $\mathcal{T}$ of rewrite rules, where each $T \in \mathcal{T}$ matches on 5-tuples, and rewrites the header to a specific value. APKeep creates a port for each rule in the rewriting element.

Based on the match fields of rewrite rules, we can compute predicates, and assign them to each port of the rewriting element, just as the forwarding and filtering element. The different part is: (1) how to encode packet rewrites using logical operations; (2) how to update predicates in the presence of rewrites.

Encoding packet rewrites. we adopt the methods in [39] to encode packet rewrites with logical operations as follows. First, it uses the existential quantification on predicate. Let p be a predicate, and x be one of the Boolean variables that p is defined on. The existential quantification of x is defined as:

$$\exists x.p = p|_{x=true} \wedge p|_{x=false} \quad (2)$$

, where $p|_{x=true}$ sets the value of variable x in p as `true`. Suppose the header has two bits x_1, x_2, then an NAT rule T that rewrites it to $x_1 = 1, x_2 = 0$ can be encoded as a logical function:

$$T(p) = (\exists x_1 \exists x_2.p) \wedge (x_1 \wedge \bar{x}_2) \quad (3)$$

The existential quantification operation is supported by BDD.

Updating predicates in the presence of rewrites. Recall that for verification, we need to traverse a DFG which is constructed based on PPM. When there are only forwarding and filtering elements, we only need to perform intersections on predicate sets during traversal. However, when there are rewriting elements, predicates need to be transformed, and we need to ensure two conditions:

(1) Each predicate should be unambiguously transformed, i.e., the transformation should be defined for each predicate in PPM. For example, suppose p is split into $p1$ and $p2$, we should know how to transform each of them; otherwise, when traversing with only $p1$ or $p2$ in the predicate set, the rewriting element does not know how to transform it.

(2) The result of transformation should be represented by a set of predicates in PPM such that the traversal can proceed. For example, suppose a predicate p is held by the port of rewrite rule T, and $T(p) = p'$. If p' cannot be represented by a set of predicates in PPM, the traversal cannot continue since p' is not "recognized" by other elements.

In order to satisfy these two conditions, we apply the following two operations: (1) when a predicate p of a rewriting port is split into $p1$ and $p2$, we compute $p1' = T(p1)$ and $p2' = T(p2)$, and apply operation (2). (2) if the transformation result p cannot be represented as a set of predicates, we create new predicates to represent p. Note that this may split a predicate of some rewriting port and trigger operation (1).

Algorithm 3 summarizes how APKeep handles rule updates for rewriting elements. First, it updates the predicates with Algorithm 2 (Line 1). The difference lies in that the algorithm also maintains a *rewrite table*, where for each entry (k, v), k is a predicate before rewrite, and v is a set of predicates after rewrite. After transferring one predicate p to the port of another rule r', we need to apply the rewrite rule r' on p, and ensure the values in the rewrite table are still predicates (Lines 2-12).

3.5 Optimization

Delayed predicate merging. In Algorithm 2, APKeep merges two predicates instantly if they have the same port set. However, for some datasets, we find that some predicates are repeatedly merged and split, resulting in a waste of time. Thus, we adopt a delayed predicate merging: when a predicate can be merged, we record it, and when the total number of predicates exceeds a threshold (500 by default), we merge all the recorded predicates. To fast determine whether a predicate can be merged, we maintain a hash table where the key is an ordered list of ports, and the value is a set of predicates that appear at all these ports.

Algorithm 3: `UpdateRewrite`$(C, \mathcal{RT})$

Input: C: the set of changes identified in the first stage; $\mathcal{RT}$: the rewrite table.
Output: D: the set of transferred predicates.

```
1  D ← UpdateRW(C);
2  while true do
3      updated ← false;
4      foreach (k, v) ∈ RT do
5          foreach p ∈ v do
6              if p ∉ P then
7                  foreach p' ∈ P do
8                      if p' ∧ p ≠ false and p' ∧ ¬p ≠ false
                       then
9                          SplitRW (p', p' ∧ p, p' ∧ ¬p);
10                 updated ← true;
11     if updated = false then
12         break;
13 Function SplitRW(p, p1, p2):
14     Split(p, p1, p2);
15     foreach (k, v) ∈ RT do
16         if p ∈ v then
17             v ← v ∪ {p1, p2} \ {p};
18     RT.remove(p);
19     RT.add(p1, {T(p1)});
20     RT.add(p2, {T(p2)});
21 return D;
```

Separate update for different types of elements. Updating both forwarding rules and ACL rules may result in a large number of predicates. For example, suppose there are n ECs generated by forwarding rules, and an ACL rule matching a destination port range will create n new ECs. AP Verifier [37] proposed to compute the atomic predicates for forwarding and ACL rules, separately. We adopt this approach and update two sets of predicates, one for forwarding elements, and one for ACL elements. When traversing the forwarding graph, we need to carry two sets of predicates, and set intersection only happen between the same set of predicates. Different from [37], our algorithm avoids false positives when verifying invariants. For example, when a node is visited twice, we evaluate whether there exist two predicates, one from each set, that have non-empty conjunction. If so, the loop exists; otherwise, the loop is a false positive.

4 Case Study

We study the expressiveness of our PPM model by showing how to model a vendor-specific function with the three built-in

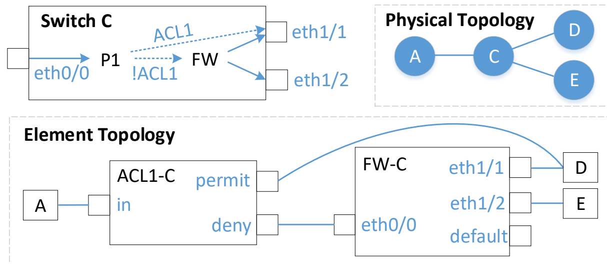

Figure 4: Modeling traffic policy in APKeep.

element types.

Policy-Based Routing (PBR) is a function commonly available in many routers and switches. It allows operators to override the IP forwarding rules such that packets are forwarded based on criteria other than destination IP address. Different vendors may implement their own version of PBR, and here we study one such implementation offered by a large device vendor.

The vendor offers a function named *traffic policy*, defined as a set of classifier-behavior pairs. The following shows a traffic policy `p1` applied to inbound traffic of interface `eth0/0` at switch *C*. `p1` is defined by a classifier `c1` and a behavior `b1`, meaning that packets satisfying `c1` will be forwarded according to `b1`. `c1` is defined using an ACL `ACL1`, and the behavior is redirecting traffic to interface `eth1/1`. The top-left and top-right of Figure 4 show the processing logic of switch *C* and the network topology, respectively.

```
interface eth0/0
traffic-policy p1 inbound
#
traffic policy p1 match-order config
classifier c1 behavior b1
#
traffic classifier c1 operator or precedence 5
if-match acl ACL1
#
traffic behavior b1
permit
redirect interface eth1/1
```

In our PPM model, the above traffic policy can be easily modeled by creating an ACL element `ACL1-C`, and properly chaining it into the forwarding graph, as shown in the bottom of Figure 4: (1) connecting its `in` port to the upstream port originally connected to `eth0/0`, (2) connecting its `permit` port to the downstream port originally connected to `eth1/1`, (3) connecting its `deny` port to the `eth0/0` port of `FW-C`.

The above is just the simplest form of traffic policy, and in a more general case, a policy can contain multiple classifier-behavior pairs, and each classifier can contain multiple ACLs. Then, we need to create multiple elements, one for each ACL, and cascade them together.

In addition to 5-tuples, a traffic policy also matches various information including VLAN ID, layer-3 packet length, time ranges, etc. Since the predicate-based EC representation has no restriction on the match fields, we can encode these match conditions by adding more fields. For example, we can add a 16-bit field to encode the packet length from 0 to 65535, and a 5-bit field to encode the hour-level time range. Note since PPM models the packet forwarding behaviors of symbolic packets, the fields to add do not have to be packet headers.

Apart from PBR, the traffic policy function also supports other behaviors including traffic statistics, flow mirroring, etc., which do not change the forwarding behaviors, and rate limiting, congestion avoidance, which selectively drop packets. As all previous data plane verifiers, PPM cannot model these features.

Table 1: Dataset statistics.

Network	Nodes	Links	Forwarding rules	ACL rules	Updates
Airtel1	68	260	6.89×10^4	0	1.42×10^7
Airtel2	68	260	9.84×10^4	0	5.05×10^8
4Switch	12	16	1.12×10^6	0	1.12×10^6
Internet2	9	56	1.26×10^5	0	2.52×10^5
Stanford*	16	74	3.84×10^3	0	7.68×10^3
Purdue*	1,646	3,094	3.52×10^6	0	7.04×10^6
Stanford	124	182	3.84×10^3	686	9.05×10^3
Purdue	2,159	3,607	3.52×10^6	2,707	7.05×10^6

5 Evaluation

5.1 Setup

Implementation. We implemented APKeep with around 5K lines of Java code. Currently, we have implemented config parsers for three different vendors, which translate vendor-specific configuration files into a unified representation in JSON format. We also implemented an update parser for one vendor, whose devices support fetching data plane state including FIBs and ACLs. For verification, we implemented an invariant checker that can detect loop and blackhole, and a what-if analyzer that can reason about the possible impact of link failures. For BDD operations, we use JDD, a BDD library for Java [34].

Dataset. Table 1 shows the datasets we use. The first six consist of updates of IPv4 forwarding rules, and the last two consist of updates of both IPv4 forwarding rules and ACL rules. The first three datasets are generated by Delta-net [20] using the ONOS SDN-IP application [6], and the Internet2 dataset is from [5]. The Stanford dataset [2] consists of both IPv4 forwarding rules and ACL rules, and the original Purdue dataset [32] consists of only ACL rules. We generate forwarding rules for the Purdue dataset, using shortest path routing. Finally, we remove the ACL rules from these two datasets, and obtain another two pure-IP datasets Stanford* and Purdue*. Since the last five datasets are snapshots of rules, we generate a sequence of updates from each of them as follows. First, we add all the ACL rules (if any), one rule per ACL each time, and then all the forwarding rules, one rule per device each time. After that we delete these rules in the reverse order as they are inserted.

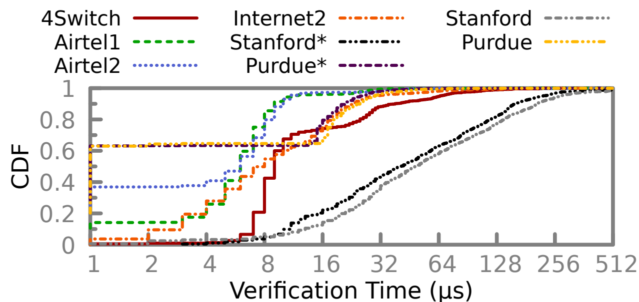

Figure 5: The distribution of verification time for APKeep.

Methods to compare. We compare APKeep with four data plane verification tools.

AP Verifier [37]. We use its open-source implementation in Java [4], and also the authors' implementation of incremental update algorithms [3]. We modify it to process our rule updates, and implement an incremental loop checker for it.

Delta-net [20]. We implement an extended version of Delta-net using C++, referred to as **Delta-net$^{\text{MF}}$**. It handles single-field IP forwarding rules in the same way as Delta-net, and handles multi-field ACL rules using a multi-layered tree approach as in VeriFlow. Note, Delta-net$^{\text{MF}}$ may not be the best approach for extending Delta-net so that it can apply to multiple fields.

VeriFlow [24]. We use its open-source implementation in C. Since VeriFlow only supports match fields expressed with prefixes, an ACL rule matching port ranges may be split into multiple ones which match prefixes.

NetPlumber [22]. We use its open-source implementation in C++ [2]. Since NetPlumber takes transfer function (TF) rule as input, we translate our rule updates into equivalent TF rule updates. Prior to update, we insert all the default rules created by NetPlumber since the insertion and deletion of them take a long time, as confirmed by the paper. We attach one source node to each device for NetPlumber to inject "flows" in the network model.

Apart from the above four methods, we also consider **APKeep^{-}**, standing for APKeep without merging predicates. For benchmark purpose, we let each method check loops after each update. All the experiments run on a Linux desktop with a 3.0GHz Intel Core i5 CPU and 32GB RAM.

5.2 Verification Time

Figure 5 shows the verification time of APKeep. We can see for all datasets, the verification time is less than 250μs for 90% of updates. Table 2 compares the average running time of APKeep with the other methods. For datasets with only IP forwarding rules, the running time of APKeep is comparable to Delta-net$^{\text{MF}}$, and much shorter than the other methods. For the 4Switch dataset, APKeep is 253×, 128×, and 937× faster than AP Verifier and VeriFlow, and NetPlumber, respectively. Note that NetPlumber is relatively slow since it models each rule as a node, and computes all the flows through these rules. Thus, its model is more fine-grained than APKeep, but incurs a relatively high cost. Surprisingly, APKeep^{-} is even faster than APKeep on some datasets. The reason is that these datasets have a rather small number of ECs (see Table 3), and therefore merging ECs incurs additional overhead without paying off. However, for the 4Switch dataset, APKeep^{-} is much slower as it has 271,793 ECs, while APKeep has only 557 ECs. For datasets with multiple match fields, all other methods including APKeep^{-} either incur a prohibitively long running time or run out of memory. For the Purdue dataset, only APKeep runs to completion, with an average running time of 13μs; all other methods either time out or run out of memory. This demonstrates existing methods can hardly meet the realtime requirement when the rules to update match multiple fields.

5.3 Number of Equivalence Classes

We observe that the number of ECs heavily impacts the running time of realtime data plane verifiers. To confirm this, we report the number of ECs maintained by APKeep, APKeep^{-}, and Delta-net$^{\text{MF}}$ in Table 3.

We can see that when there is a single match field, APKeep^{-} computes slightly fewer ECs than Delta-net$^{\text{MF}}$. The reduction is due to the fact that predicates can encode arbitrary packet sets, rather than ranges. By merging predicates, APKeep computes much fewer ECs than APKeep^{-}. This indicates that using predicates alone cannot efficiently reduce the number of ECs.

The number of ECs computed by Delta-net$^{\text{MF}}$ grows from 2283 to 15 million after only 686 ACL rules are inserted in the Stanford dataset, and reaches over 100 million after 2,707 ACL rules are inserted in the Purdue dataset. Note that Delta-net$^{\text{MF}}$ actually does not run to completion for Stanford and Purdue datasets, and the numbers are counted by running only the functions related to the creation of ECs. In contrast, APKeep computes only 515 and 4,160 ECs for these two datasets, a 99.99% reduction compared with Delta-net$^{\text{MF}}$.

The above results show that range-based EC representation easily leads to an explosion of ECs when there are only a small number of rules with multiple match fields. On the other hand, by representing ECs with predicates, and updating the minimum number predicates, APKeep can dramatically reduce the total number of ECs, thereby achieving a fast verification speed with small memory footprint.

Figure 6 shows the number of ECs maintained by APKeep and Delta-net$^{\text{MF}}$ during the updates. The Airtel1 dataset consists of rule insertions and deletions which are generated to react to link failures. Thus, the number of rules is small during update, and the total number of ECs is also quite small. The 4Switch dataset only has rule insertions; and the last three datasets insert all rules and remove them later. Thus, for Internet2 and Stanford, APKeep finally has only one and two predicates, respectively. For the Purdue dataset, since both

Table 2: Average verification time of different methods (D^{MF} is shorthand for Delta-net^{MF}). TO means timeout (> 24h), and MO means memory overflow (> 32GB).

Network	Average verification time (μs)						Percentage < $250\mu s$					
	AP Verifier	VeriFlow	NetPlumber	D^{MF}	APKeep$^-$	**APKeep**	AP Verifier	VeriFlow	NetPlumber	D^{MF}	APKeep$^-$	**APKeep**
Airtel1	80	59	3,804	3	5	**7**	91.3%	99.9%	3.8%	99.9%	99.9%	**99.8%**
Airtel2	135	48	TO	4	4	**6**	77.4%	99.9%	TO	99.9%	99.9%	**99.9%**
4Switch	5,316	2,706	19,678	4	2,190	**21**	7.8%	8.2%	0.8%	99.9%	75.1%	**99.8%**
Internet2	1,660	144	2,123	3	9	**12**	24.2%	93.3%	9.9%	99.9%	99.5%	**99.7%**
Stanford*	1,953	468	8,700	9	98	**94**	13.3%	96.1%	23.6%	99.9%	93.6%	**96.4%**
Purdue*	777	648	MO	15	2	**9**	83.7%	66.5%	MO	99.9%	99.9%	**99.9%**
Stanford	2,072	4.8×10^6	9,532	MO	3.1×10^5	**127**	24.3%	0.4%	34.0%	MO	11.8%	**91.7%**
Purdue	TO	TO	MO	MO	MO	**13**	TO	TO	MO	MO	MO	**99.8%**

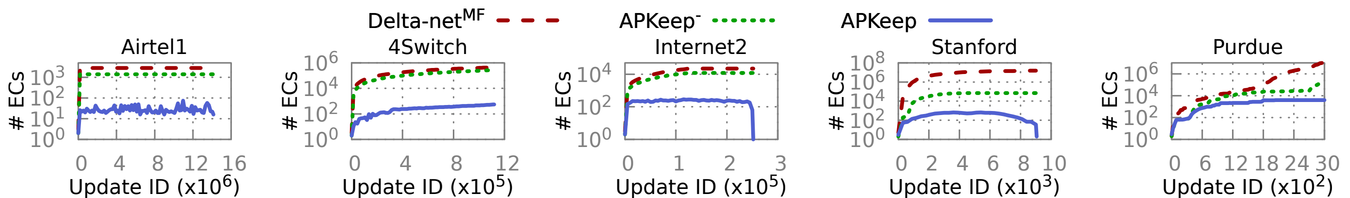

Figure 6: Number of equivalence classes maintained by APKeep and Delta-net^{MF} during updates.

Table 3: Number of ECs for APKeep and Delta-net^{MF}.

Network	Delta-net^{MF}	APKeep$^-$	APKeep	Reduction Rate
Airtel1	2,799	1,401	16	99.4%
Airtel2	2,799	1,401	64	97.8%
4Switch	443,443	271,793	557	99.9%
Internet2	22,212	14,819	216	99.0%
Stanford*	2,283	1,515	494	78.4%
Purdue*	1,176	939	267	77.4%
Stanford	15,100,968	842,734	515	99.99%
Purdue	>104,743,229	>168,891	4,160	99.99%

Table 4: Verification time for "what if" queries. The results for Delta-net were from paper [20], whose experiments ran on a 3.47GHz Intel Xeon CPU.

Network	# Rules	Average query time (ms)		+Loops (ms)	
		Delta-net	APKeep	Delta-net	APKeep
Airtel	38,100	0.04	0.02	2.3	0.13
4Switch	1,120,000	21.1	0.48	128.1	1.37

Delta-net^{MF} and APKeep$^-$ cannot run to completion, we only show the number for the first 3000 updates.

5.4 Answering "What if" Queries

We evaluate the running time for APKeep to answer "what if" queries. In particular, we consider the query "what is the fate of packets that use a link if the link fails?". To answer this query, Delta-net constructs a forwarding graph using those ECs on that link, and is reported to be 10× faster than VeriFlow. Thus, we only compare our results to those of Delta-net. In Table 4, Columns 3-4 show the average query time, and Columns 5-6 show the average query time if we additionally check loops. We can see that APKeep is 17× and 93× faster than Delta-net in overall query time for the Airtel and 4Switch datasets, respectively. The reason is that the number of ECs in APKeep is much smaller than that in Delta-net.

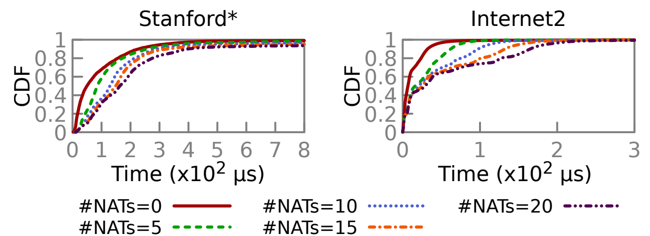

Figure 7: The cumulative distribution of verification time when different number of NATs are added.

5.5 Updating Rewrite Rules

We evaluate the time for APKeep to handle updates of rewrite rules. We use the Stanford* and Purdue* dataset, and add NAT rules into the network as follows. First, for each dataset, we find all the edge ports: an edge port holds a non-empty set of predicates, and is not connected to any other switches. Then, for each edge port, we randomly select a predicate associated with it, and compute an IP prefix that satisfies the predicate. Finally, for each IP prefix, we generate 25 NAT rules, each of which translates an IP address to another address belonging to a different IP prefix. We place the updates of NAT rules after the updates of forwarding rules.

Figure 7 shows the running time of APKeep for different numbers of NATs ranging from 0 to 20. Since each NAT has 25 rules, the number of NAT rules ranges from 0 to 1000. We can see that the running time of APKeep is mostly less than 1ms, and scales well with the number of NAT rules.

6 Related Work

Offline data plane verification was originally studied by Xie et al. [36], and later advanced by FlowChecker [10],

Anteater [27], HSA [23], and NoD [25]. These tools take a snapshot of the data plane state, and check whether it satisfies network invariants like blackhole-freedom, loop-freedom, etc. AP Verifier [37–39] uses Binary Decision Diagraph (BDD) to compute a predicate for each port, and uses all port predicates to generate atomic predicates, which are the minimum set of ECs. APKeep differs in that it builds on a modular element-level model that is much more expressive than the monolithic model used by AP Verifier. In addition, APKeep incrementally updates the ECs instead of re-computing them from scratch, thereby achieving up to 200× speedup compared to the incremental version of AP Verifier (Table 2). To scale verification to large networks, Libra [40] uses MapReduce to parallelize verification. Plotkin et al. [29] propose to transform large networks into smaller ones for scalable verification, based on network surgery and symmetry. APKeep can leverage this technique to reduce network size, thereby scaling to larger networks. RCDC [15] decomposes data plane verification into the validation of local contracts. However, RCDC assumes structured datacenter networks so as to track the topology and address locality, while APKeep targets general networks. SymNet [31] and VMN [28] focus on verifying stateful data planes with middleboxes.

Realtime data plane verification incrementally checks the network data plane for each update in real time. Net-Plumber [22] builds on the plumbing graph model, where each node is a rule and a flow is a set of packets traversing the same sequence of rules. Thus, the model has a finer grain than PPM, while the downside is that updating the model is relatively slow (Table 2). VeriFlow [24] achieves a smaller verification time ($<$ 1ms) by computing the equivalence classes (ECs) affected by an update, and checking the forwarding graph of each affected EC. Delta-net [20] further reduces the verification time by incrementally maintaining a single EC-labelled graph, rather than constructing multiple graphs for each update. VeriFlow and Delta-net can achieve sub-millisecond verification time for updates of single-field IP forwarding rules. Howerver, they may suffer from the problem of EC explosion when there are multi-field rules, and cannot handle updates of rewriting rules.

Representation of ECs. Bjørner et al. propose ddNF [14], a new data structure for representing ECs, and show it outperforms BDD on datasets consisting of forwarding rules. However, the set of ECs represented using ddNF may not be minimal. #PEC [21] introduces a new lattice-theoretic method which can construct the minimum number of ECs, faster than using BDD. #PEC may serve as a better foundation for multi-field extension of Delta-net, and it would also be interesting to study how to leverage #PEC to further speed up APKeep.

Control plane verification checks whether protocol configurations are correct [9, 12, 13, 16, 18, 19, 30]. They are orthogonal to APKeep, while tools like Batfish [18] may use APKeep to speed up the verification of generated data planes.

7 Discussion

Model modularity vs. number of ECs. Modeling the network at a fine granularity can make the update more efficient, while may also increase the number of ECs. The reason is that the model may have more different forwarding behaviors, which need to be represented with more ECs. For example, even two packets behave the same at a device level, their behaviors may differ in the intra-device processing, and thus should be represented with different ECs. Thus, there is a tradeoff between the model granularity and the number of ECs. It will be interesting to further navigate such tradeoff in the future.

Fetching data plane state. APKeep fetches the whole data plane state only once in the bootstrap stage, and only fetches data plane updates afterwards, whose cost can be much less compared to fetching the whole data plane. To ensure timeliness, APKeep needs to fetch the updates from devices at a sufficient frequency. We are aware some new devices have already provided APIs for fetching FIB updates, and we expect this feature will be supported by more devices in the future.

Ensuring update consistency. Since the data plane state is in continuous transition, a violation can be falsely triggered by a transient state. APKeep can be made robust to such inconsistency as follows. If an update fails the verification of an invariant, APKeep flags it as suspicious without raising an alarm. After a configured time window, APKeep checks the invariant again to confirm whether the update is a true violation. Detailed design is left as one of our future work.

Why microsecond-level verification. One major purpose of speeding up incremental verification is to ensure the network model, which verification is based on, can keep up with fast network updates. For example, in a large datacenter with 1k devices, a 1ms model update time only allows the verifier to keep up with an average network update rate of 1 update per device every second. Thus, further speeding up data plane verifiers can scale the verification to larger network size and higher data plane update rate.

8 Conclusion

This paper presented APKeep, a new realtime data plane verifier. APKeep builds atop PPM, a modular network model that is expressive for real devices, and incrementally maintains the minimum number of equivalence classes in realtime. We showed that for real updates consisting of both forwarding and ACL rules, all other methods either ran out of memory or incurred a prohibitively long verification time, while APKeep still achieved a sub-millisecond verification time.

Acknowledgments. We thank the anonymous NSDI reviewers and our shepherd Behnaz Arzani for their valuable feedback. We thank Zunying Qin for help on collecting device configurations. This work is supported by NSFC (No. 61772412 and 61702407) and the K. C. Wong Education Foundation.

References

[1] Configuring IP Access Lists - Cisco. https://www.cisco.com/c/en/us/support/docs/security/ios-firewall/23602-confaccesslists.html.

[2] Hassel, the header space library. https://bitbucket.org/peymank/hassel-public.

[3] Real-time Verification of Network Properties Using Atomic Predicates (technical report). http://www.cs.utexas.edu/users/lam/NRL/TechReports/Yang_Lam_TR-13-15.pdf.

[4] Scalable Network Verification Using Atomic Predicates. http://www.cs.utexas.edu/users/lam/NRL/Atomic_Predicates_Verifiers.html.

[5] The Internet2 Observatory. http://www.internet2.edu/research-solutions/research-support/observatory.

[6] The ONOS project. https://onosproject.org/.

[7] Troubleshooting the Network Survey. http://eastzone.github.io/atpg/docs/NetDebugSurvey.pdf.

[8] Understanding Policy Routing - Cisco. https://www.cisco.com/c/en/us/support/docs/ip/border-gateway-protocol-bgp/10116-36.html.

[9] A. Abhashkumar, A. Gember-Jacobson, and A. Akella. Tiramisu: Fast and general network verification. In *USENIX NSDI*, 2020.

[10] E. Al-Shaer and S. Al-Haj. Flowchecker: Configuration analysis and verification of federated openflow infrastructures. In *ACM workshop on Assurable and usable security configuration*, 2010.

[11] H. R. Andersen. An introduction to binary decision diagrams. *Lecture notes, available online, IT University of Copenhagen*, 1997.

[12] R. Beckett, A. Gupta, R. Mahajan, and D. Walker. A general approach to network configuration verification. In *ACM SIGCOMM*, 2017.

[13] R. Beckett, A. Gupta, R. Mahajan, and D. Walker. Control plane compression. In *ACM SIGCOMM*, 2018.

[14] N. Bjørner, G. Juniwal, R. Mahajan, S. A. Seshia, and G. Varghese. ddNF: An efficient data structure for header spaces. In *Haifa Verification Conference*, 2016.

[15] N. Bjørner, J. Padhye, A. Agrawal, A. Bhargava, P.-A. C. Bissonnette, S. Foster, A. Helwer, M. Kasten, I. Lee, A. Namdhari, H. Niaz, A. Parkhi, H. Pinnamraju, A. Power, N. M. Raje, and P. Sharma. Validating datacenters at scale. In *ACM SIGCOMM*, 2019.

[16] 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 *USENIX OSDI*, 2016.

[17] N. Feamster and H. Balakrishnan. Detecting bgp configuration faults with static analysis. In *USENIX NSDI*, 2005.

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

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

[20] A. Horn, A. Kheradmand, and M. R. Prasad. Delta-net: Real-time network verification using atoms. In *USENIX NSDI*, 2017.

[21] A. Horn, A. Kheradmand, and M. R. Prasad. A precise and expressive lattice-theoretical framework for efficient network verification. In *IEEE ICNP*, 2019.

[22] P. Kazemian, M. Chan, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real time network policy checking using header space analysis. In *USENIX NSDI*, 2013.

[23] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: Static checking for networks. In *USENIX NSDI*, 2012.

[24] A. Khurshid, W. Zhou, M. Caesar, and P. Godfrey. VeriFlow: Verifying network-wide invariants in real time. In *USENIX NSDI*, 2013.

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

[26] R. Mahajan, D. Wetherall, and T. Anderson. Understanding bgp misconfiguration. In *ACM SIGCOMM*, 2002.

[27] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. Godfrey, and S. T. King. Debugging the data plane with Anteater. In *ACM SIGCOMM*, 2011.

[28] A. Panda, O. Lahav, K. Argyraki, M. Sagiv, and S. Shenker. Verifying reachability in networks with mutable datapaths. In *USENIX NSDI*, 2017.

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

[30] S. Prabhu, K.-Y. Chou, A. Kheradmand, P. Godfrey, and M. Caesar. Plankton: Scalable network configuration verification through model checking. In *USENIX NSDI*, 2020.

[31] R. Stoenescu, M. Popovici, L. Negreanu, and C. Raiciu. Symnet: Scalable symbolic execution for modern networks. In *ACM SIGCOMM*, 2016.

[32] Y.-W. E. Sung, S. G. Rao, G. G. Xie, and D. A. Maltz. Towards Systematic Design of Enterprise Networks. In *ACM CoNEXT*, 2008.

[33] B. Tian, X. Zhang, E. Zhai, H. H. Liu, Q. Ye, C. Wang, X. Wu, Z. Ji, Y. Sang, D. Y. Zhang, Ming, C. Tian, B. Y. Zhao, and H. Zheng. Safely and automatically updating in-network acl configurations with intent language. In *ACM SIGCOMM*, 2019.

[34] A. Vahidi. JDD, a pure Java BDD and Z-BDD library. https://bitbucket.org/vahidi/jdd/.

[35] A. Wool. A quantitative study of firewall configuration errors. *Computer*, 37(6):62–67, 2004.

[36] G. G. Xie, J. Zhan, D. A. Maltz, H. Zhang, A. Greenberg, G. Hjalmtysson, and J. Rexford. On static reachability analysis of ip networks. In *IEEE INFOCOM*, 2005.

[37] H. Yang and S. S. Lam. Real-time verification of network properties using atomic predicates. In *IEEE ICNP*, 2013.

[38] H. Yang and S. S. Lam. Real-time verification of network properties using atomic predicates. *IEEE/ACM Transactions on Networking*, 24(2):887–900, 2016.

[39] H. Yang and S. S. Lam. Scalable verification of networks with packet transformers using atomic predicates. *IEEE/ACM Transactions on Networking*, 25(5):2900–2915, 2017.

[40] 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 *USENIX NSDI*, 2014.

A Proof of Theorem 1

Proof. According to Definition 1, it is easy to see that a set of ECs is minimum if the other direction of condition (3) also holds, i.e., if two packets are forwarded to the same port at each element, then they must belong to the same EC. Therefore, define (3)' by replacing "⇒" with "⇔" in (3), and we need to prove that conditions (1)(2)(3)' hold for all predicates in the PPM model.

Clearly, these conditions hold initially when there are no updates: each element has a single default rule, and there is only one `True` predicate, which is assigned to the default/deny/id port depending on element type. We prove the theorem by induction: if the conditions hold before an update, then they still hold after the update.

Let e be the element whose rule is updated. For condition (1)(2), it is clear that `transfer` operations do not modify predicates, and thus have no effect on these conditions. Also, since `split` and `merge` only move part of one predicate into another one, they will not break these conditions, either. In the following, we show both directions of (3)' hold.

⇒: Suppose $h_1, h_2 \in c$ after update, we show they appear in the same set of ports. There are two cases. (1) h_1 and h_2 belong to the same predicate before update. Then, we know $Port_s(h_1) = Port_s(h_2), \forall s$, and the update only changes $Port_e(h_1)$ and $Port_e(h_2)$. Then, we only need to show $Port_e(h_1) = Port_e(h_2)$ after update. Clearly this holds if neither h_1 or h_2 changes its port at e. Suppose at least one of them changes its port at e, and let $Port_e(h_1) = Port_e(h_2) = p_a$ before update. Without loss of generality, suppose h_1 changes its port to $p_b \neq p_a$, we prove by contradiction that h_2 must have also changed its port to p_b. Assume h_2 either keeps its port unchanged or changes its port to $p_c \neq p_b$. Suppose $h_1 \in c_1$ and $h_2 \in c_2$ after transferring predicates, then c_1 appears at port p_b, and c_2 appears at port p_a or p_c. Since c_1 and c_2 appear in different ports at e, they cannot be merged, contradicting our assumption that $h_1, h_2 \in c$ after update. (2) h_1 and h_2 belong to different predicates before update. Let $h_1 \in c_1$ and $h_2 \in c_2$ before update, then c_1 and c_2 must have been merged into c. That is, c_1 and/or c_2 must have been transferred to the same port after update, which implies that h_1 and h_2 must have changed their ports to the same one at e.

⇐: Suppose $Port_s(h_1) = Port_s(h_2), \forall s$ after update, there are two cases. (1) $h_1, h_2 \in c'$ before update. Then, $Port_s(h_1) = Port_s(h_2), \forall s$ before update, implying that both h_1 and h_2 do not change their ports or change to the same port. If they do not change their ports, then they will still belong to the same predicate (either c' if c' is not split or $c' \wedge \neg\delta$ if c is split). If they change their ports, then a predicate including h_1, h_2 (either c' if c' is not split or $c' \wedge \delta$ if c' is split) will be transferred to a new port, and they will still belong to the same predicate, no matter whether the transferred predicate is merged or not. (2) $h_1 \in c_1, h_2 \in c_2$ before update, then we know $Port_e(h_1) \neq Port_e(h_2)$ before update, and thus at least one of them must have changed its port. Without loss of generality, suppose h_1 has changed its port such that $Port_s(h_1) = Port_s(h_2)$ after the change, and let c' be the transferred predicate satisfying $h_1 \in c'$. This will trigger our algorithm to merge c' and c_2. □

Algorithm 4: `ConstructDeltaForwardingGraph`(D)

```
Input: D: the set of transferred predicates.
Output: G: the forwarding graph used for invariant checking.
1  V ← {}, E ← {}, A ← ⊥;
2  foreach δ ∈ D do
3      foreach port ∈ Port(δ) do
4          s1 ← the node holding port;
5          if s1 ∉ V then
6              V ← V ∪ {s1};
7          s2 ← the node connected to port;
8          if s2 ∉ V then
9              V ← V ∪ {s2};
10         if (s1, s2) ∉ E then
11             A(s1, s2) ← {};
12             E ← E ∪ {(s1, s2)};
13         A(s1, s2) ← A(s1, s2) ∪ {δ};
14 return G(V, E, A);
```

B Algorithms for Constructing Delta Forwarding Graphs and Checking Invariants

Constructing delta forwarding graphs. Algorithm 4 summarizes the process of constructing delta forwarding graphs. It takes the set of transferred predicates, denoted as D, computed by Algorithm 2, and returns a delta forwarding graph, denoted as $G(V,E,A)$. Here, V is the set of nodes, E is the set of edges, and $A: E \to 2^C$ is a map from each edge to the set of predicates allowed on that edge (C denotes the set of all predicates). That is, $A(s1,s2)$ is the set of predicates that can be sent from switch $s1$ to switch $s2$. First, V, E, and A are initialized (Line 1). For each predicate δ in D, APKeep iterates over each *port* in the port set of δ (Lines 2-3). If the node $s1$ holding *port* is not in the node set V, then $s1$ is added into V (Lines 4-6). Similarly, if the node directly connected to *port* is not in V, then $s2$ is also added into V (Lines 7-9). Note here if the *port* is "default", we assume it is connected to a virtual node named "default". If the edge $(s1,s2)$ is not in the edge set E, then it is added into E, and the mapping $A(s1,s2)$ is initialized to empty set (Lines 10-12). Finally, the transferred predicate δ is added into the set $A(s1,s2)$ (Line 13).

Checking invariants. With the delta forwarding graph G, operators can check network invariants by traversing G. Algorithm 5 shows an example program for checking blackhole-freedom and loop-freedom (defined in § 3.3). The algorithm starts the traversal from each node $s \in V$. Here V includes all nodes corresponding to the element where the rule is updated. Before each traversal, the algorithm initializes *pset*, the current set of predicates, to D, and *history*, the nodes that have been visited, to empty set (Lines 2-3). The traversal stops when *pset* becomes empty (Line 6-7) or the visited node is already in *history* (Line 8-10), meaning a loop is detected, or the next hop is *default* (Line 12-14), meaning packets in *pset* match no forwarding rule in the corresponding device, i.e., a blackhole is detected. Otherwise, the algorithm updates *pset* and *history* and traverses the next hop (Line 15).

Algorithm 5: `CheckInvariants`(G,D,V)

```
Input: G: the forwarding graph used for invariant checking, D:
       the set of transferred predicates, V: the set of nodes to
       start check.
1  foreach s ∈ V do
2      pset ← D;
3      history ← {};
4      Traverse (s, predicates, history);
5  Function Traverse (s, pset, history):
6      if pset = ∅ then
7          return;
8      if s ∈ history then
9          Alter('loop');
10         return;
11     foreach (s, s′) ∈ E do
12         if s′ = default then
13             Alter('backhole');
14             return;
15         Traverse (s′, pset ∧ A(s, s′), history ∪ {s});
16     return;
```

Table 5: Memory cost. TO means timeout (> 24h), and MO means memory overflow (> 32GB).

Network	Memory cost (MB)				
	AP Verifier	VeriFlow	NetPlumber	Delta-net^{MF}	APKeep
Airtel1	417	508	4,283	61	180
Airtel2	5170	16,049	TO	74	193
4Switch	7,722	2,520	1,749	785	936
Internet2	360	206	613	44	87
Stanford*	6	16	4,971	25	3
Purdue*	1,465	1,414	MO	3,414	648
Stanford	6	98	8,607	MO	3
Purdue	TO	TO	MO	MO	744

C Memory Cost

We compare the memory cost of APKeep with the other four methods. Not surprisingly, for single-field datasets, the memory cost of APKeep is comparable to Delta-net^{MF}, and both of them have smaller memory footprint than others. For the multi-field Stanford dataset, Delta-net^{MF} runs out of 32GB memory. For the multi-field Purdue dataset, AP Verifier and VeriFlow do not run to completion within 24 hours; NetPlumber and Delta-net^{MF} run out of 32GB memory. APKeep still maintains a small memory footprint for these two multi-field datasets.