EPVerifier: Accelerating Update Storms Verification with Edge-Predicate

Chenyang Zhao, Yuebin Guo, Jingyu Wang, Qi Qi, Zirui Zhuang, Haifeng Sun, and Lingqi Guo, *State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications;* Yuming Xie, *Huawei Technologies Co., Ltd;* Jianxin Liao, *State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications*

https://www.usenix.org/conference/nsdi24/presentation/zhao

This paper is included in the Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation.

April 16–18, 2024 • Santa Clara, CA, USA

978-1-939133-39-7

Open access to the Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation is sponsored by

EPVerifier: Accelerating Update Storms Verification with Edge-Predicate

Chenyang Zhao[§ *], Yuebin Guo[§ *], Jingyu Wang[§ †], Qi Qi[§], Zirui Zhuang[§],
Haifeng Sun[§], Lingqi Guo[§], Yuming Xie[⋆], Jianxin Liao[§ †]
[§] *State Key Laboratory of Networking and Switching Technology,*
Beijing University of Posts and Telecommunications [⋆]*Huawei Technologies Co., Ltd*

Abstract

Data plane verification is designed to automatically verify network correctness by directly analyzing the data plane. Recent data plane verifiers have achieved sub-millisecond verification for per rule update by partitioning packets into equivalence classes (ECs). A large number of data plane updates can be generated in a short interval, known as *update storms*, due to network events such as end-to-end establishments, disruption or recovery. When it comes to update storms, however, the verification speed of current EC-based methods is often slowed down by the maintenance of their EC-based network model (EC-model).

This paper presents EPVerifier, a fast, partitioned data plane verification for update storms to further accelerate update storms verification. EPVerifier uses a novel edge-predicate-based (EP-based) local modeling approach to avoid drastic oscillations of the EC-model caused by changes in the set of equivalence classes. In addition, with local EPs, EPVerifier can achieve a partition of verification tasks by switches that EC-based methods cannot to get better parallel performance. We implement EPVerifier as an easy-to-use tool, allowing users to quickly get the appropriate verification results at any moment by providing necessary input. Both dataset trace-driven simulations and deployments in the wild show that EPVerifier achieves robustly fast update storm verification and superior parallel performance and these advantages expand with the data plane's complexity and storm size growth. The verification time of EPVerifier for an update storm of size 1M is around 10s on average, a 2-10× improvement over the state-of-the-art.

1 Introduction

Network faults such as forwarding loops, black holes, and reachability issues caused by misconfigurations, hardware or software problems can result in significant economic losses and social impact [12, 16, 21]. Manual troubleshooting is inefficient and prone to introduce new errors due to oversight. Therefore, *network verification* is proposed to automatically detect network correctness and avoid error-prone manual analysis by inferring all possible network behaviors based on the network's control plane configurations [5, 10, 13, 14, 25, 26, 32, 34] or data plane forwarding state [6, 15, 17–20, 23, 33, 37], which are known as *control plane verification* and *data plane verification*, respectively.

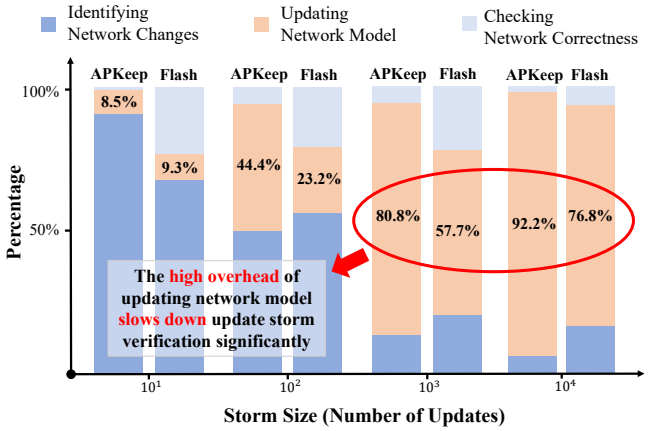

Figure 1: For the state-of-the-art equivalence-class-based data plane verifiers, the proportion of the Updating Network Model to the overall verification time explodes as the storm size (the number of updates in the update storm) expands.

In particular, this paper focuses on *data plane verification*, which directly checks the data plane to detect whether the network violates invariants (loop and blackhole) or other user-defined specifications (*e.g.,* reachability). A typical data plane verification process usually consists of three steps: 1) *Identifying network changes*, which identifies hop-by-hop forwarding behavior changes when updates come by analyzing rule dependency. 2) *Updating network model*, which updates the network model that is consistent with the data plane forwarding behavior 3) *Checking network correctness*, which checks the correctness of the data plane on a forwarding graph

*Chenyang Zhao and Yuebin Guo contributed equally to this work.
†Jingyu Wang and Jianxin Liao are the corresponding authors.

extracted from the network model. To be useful in practice, such data plane verifications need to be completed as rapidly as possible. Otherwise, the verification results may be invalid due to inconsistencies between the network state captured before verification and the actual network state that the updates are delivered into.

On the one hand, recent tools like APKeep [37] achieve sub-millisecond verification of each update when data plane updates arrive relatively slowly. Still, on the other hand, a large number of data plane updates can be arrived in a short time frame due to network events such as end-to-end establishments, disruption or recovery, etc., which is called *update storms* [15]. In this case, although Flash [15] gets substantial speed up on update storm verification by avoiding redundant computations and applying a series of engineering optimizations, updating the network model still slows down the verification speed. For example, as shown in Figure 1, in a synthetic wide-area backbone network containing 87 routers and 2308 links, the time consumption of updating network model as a percentage of the overall verification time increases with the size of the update storm and stays at a high level ($> 75\%$) regardless of whether the optimization is applied or not. This is because their network model is based on global equivalence classes (ECs) and therefore cannot avoid *network model oscillations*, *i.e.,* redundantly modify the part of the network model where no traffic change occurs. Besides, Flash divides the header space into multiple subspaces and distributes the verification of these subspaces to a cluster of verifiers in parallel. This simple and straightforward partition can be applied directly to all data plane verification systems. However, this kind of partition is prone to bottlenecks in practice because the updates that arrive are most likely *not evenly* distributed across subspaces.

To address the above problems and further accelerate update storms verification, we present EPVerifier, a fast, partitioned data plane verifier for update storms. Instead of partitioning packets into a set of ECs and maintaining an EC-based network model (EC-model) to represent the data plane forwarding state, EPVerifier maintains network packet flow with edge predicates(EPs) to form an EP-based network model, which we call *EP-model*. Each EP represents the local packet flow of one specific unidirectional edge. Although the time consumption of checking network correctness increases compared to the EC-based methods, such cost is worth it for update storms. This is because checking network correctness consumes much less time than updating the network model when there are a large number of updates, and the EP-model fundamentally avoids modifying the part of the network model where no packet flow changes, thus greatly speeding up the network model updating. For parallelism, EPVerifier can divide verification tasks in a switch-based way that EC-based methods cannot. This is because EP has a local nature that EC does not. Data plane updates that affect only the forwarding behavior of a single switch will also only affect edge predicates on that switch. However, using ECs to represent network traffic makes the same EC appear on different switches, meaning that traffic changes on one switch may also affect other switches. This local nature allows the EPVerifier to divide switches into clusters, which we call *regions*. We divide the regions after the update storms arrive to ensure that updates are *evenly* distributed among regions. We fully implement EPVerifier as an easy-to-use tool, users can quickly get the appropriate verification results at any moment by providing necessary input. The evaluation results on both datasets simulations and deployments in the wild show that EPVerifier achieves robustly fast update storm verification and superior parallel performance and these advantages expand with the data plane's complexity and storm size growth. Compared to the state-of-the-art, EPVerifier is up to more than $10\times$ faster.

Contributions. In summary, our main contributions are:

- We propose a novel network modeling approach based on edge predicates (EPs), which can quickly load a large number of data plane updates as they arrive. This approach fills the gap for fast update storm verification.
- A novel verification approach that can divide verification tasks by switches to achieve fast update storms verification in parallel.
- EPVerifier, an implementation of our approaches, evaluated on both datasets trace-driven simulations and deployments in the wild. EPVerifier achieves substantial gains compared to the state-of-the-art, and such advantages expand with the complexity of the data plane and the growth of the update storm size.

Roadmap. The problem definition for updating storm verification (§ 2) and the architecture and workflow of EPVerifier (§ 3) are given first, followed with examples to illustrate the difference between our approach and the state-of-the-art (§ 4). Then, the design details (§ 5) are presented and the experimental results (§ 6) are shown. After disscussing future (§ 7) and related work (§ 8), the conclusion is drawn (§ 9).

2 Problem Definition

Network model. On a data plane consists of N devices, the forwarding behavior of each device i is controlled by its routing table T_i, where each item $T_i[k]$ is a 3-tuple (m,a,p). Here m denotes the packets matched by this item, a denotes where the packets matched by this item will be forwarded to, and p denotes the priority of a routing table item. Since the m of different items in the same routing table may overlap, the packets governed by each item are determined by both m and p. When a packet arrives, device i iterates through each item on T_i in decending order of priority to get the highest priority item $T_i[k]$ that matches the packet and forwards the packet to $T_i[k].a$. The data plane forwarding behavior then can be modeled using an edge-labeled directed graph $G(V,E,l)$, where the nodes V represent devices, and edges E represent simplex

links. The function $l : E \rightarrow Label$ assigns to each edge a label that represents the packet flow on this edge. Note that different network models have different labels, for example, for the EC-model, the label is a set of equivalence classes, while for the EP-model it is an edge predicate.

Identifying network changes. Consider an update storm S consisting of K data plane updates, where each update is denoted as an insertion or deletion of a 4-tuple FIB rule $r(match, from, to, priority)$. And the installation process of these updates in the data plane is the insertion or deletion of the item $(match, to, priority)$ from the routing table T_{from}. However, the change of packet flow due to the entry insertions or deletions is not straightforward because of the priority, *i.e.*, we cannot directly modify the edge label based on $r.match$. Therefore, to specify the set of packets that a rule actually affects, we need a new field *hit*, introduced by [37], to represent the set of packets that a rule actually affects in the network. The *hit* is defined as follows:

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

Here r' denotes other FIB rules that share the routing table with r. With *hit* field, the network change of S is defined as follows:

$$S.change = (r_1.hit, ..., r_K.hit) \quad (2)$$

Updating network model. The update process of the network model G is actually the modification process of the edge label l. For the insertion of rule r, there are two parts of the edge label that need to be modified: 1) $r.hit$ need to be merged into the label on edge$(r.from, r.to)$ and 2) deleted from the label on edge$(r.from, *)$, here $*$ means any value that is not $r.to$. The similar goes for rule deletion.

Checking network correctness. After updating the network model, the verifier uses graph algorithms to check the correctness of the network on a forwarding graph $G'(V, E, l')$ extracted from the network model. Similar to G, the forwarding graph G' is also an edge-labeled directed graph, but l' is only a subset of l that affected by update storm S. Specifically, l' satisfies the following constraint:

$$l'[(v, v')] = l[(v, v')] \wedge (\vee_{i=1}^{i=K} r_i.hit) \quad \forall (v, v') \in E \quad (3)$$

Verification partition. There are two natural ways to decompose complex verification tasks into smaller, parallel computations: *subspace-based partition* and *switch-based partition*, as described in turn. First, the subspace-based partition achieves parallelism by dividing the forwarding state of the overall network into multiple subspaces. Namely, subspace-based partitions build multiple network models, each representing the forwarding behavior in a subspace, just like network slices. Second, the purpose of the switch-based partition is to divide the network model into different clusters of switches, which we call *regions*, and manage the forwarding state of different regions in parallel. In other words, switch-based partition only constructs one network model but maintains different parts simultaneously to keep the overall network model consistent with the data plane forwarding behavior.

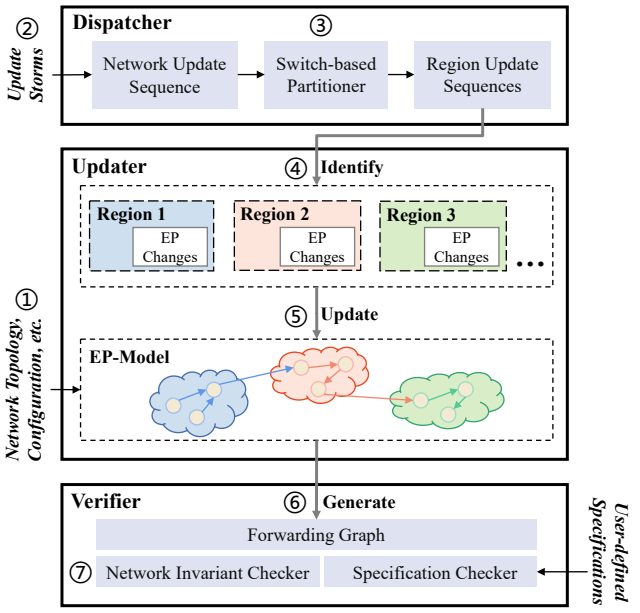

Figure 2: The architecture and workflow of EPVerifier, which requires three inputs: 1) the data plane topology and configuration provided by the controller for building the EP-model 2) Controller-generated update storms that will be delivered into the data plane 3) the user-defined specifications (reachability, waypointing, *etc.*)

3 Architecture and Workflow

As shown in Figure 2, the EPVerifier is composed of the following three parts:

The Dispatcher aims to guarantee the data plane updates in the update storms evenly across the different regions. When an update storm arrives, it divides the EP-model into multiple regions based on the distribution of data plane updates on each switch, ensuring that the number of updates in each region is as equal as possible.

The Updater corresponds to the identifying network change and updating network model of the data plane verification process. More specifically, it identifies the EP changes in each region in parallel and applies these changes to the EP-model.

The Verifier is responsible for checking network correctness. It first extracts the forwarding graph from the EP-model, and then applies the corresponding algorithm on this graph to verify the network invariants or user-defined specifications.

Workflow. A typical workflow of using the EPVerifier is as illustrated in Figure 2. First, the controller * needs to pro-

*Note that the EPVerifier can run in any environment that can provide data plane updates. For simplicity, we use "controller" to refer to the assisted system that provides data plane updates to it.

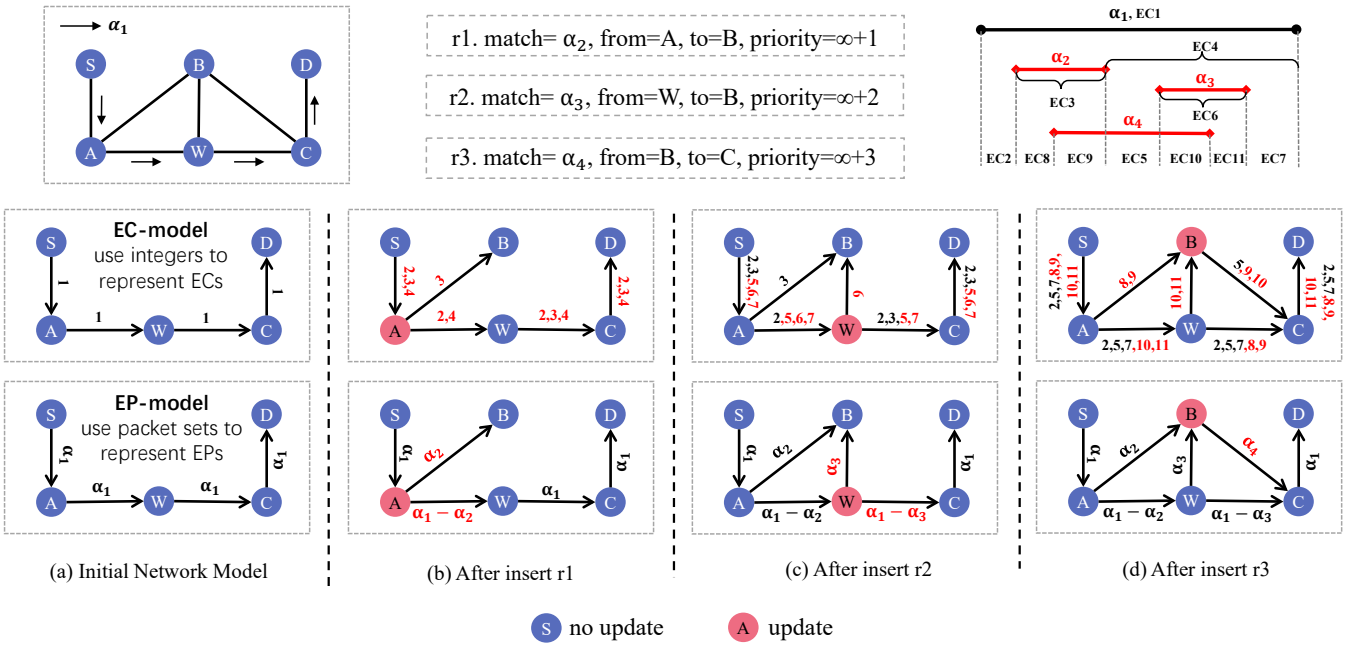

Figure 3: EPVerifier uses edge predicates to avoid the network model oscillations.

vide the data plane topology and configuration information to EPVerifier for initializing an EP-based network model in the bootstrap stage (①). Then, EPVerifier converts each update storm obtained from the controller into a network update sequence (②), where each item is a data plane update representing the insertion or deletion of a 4-tuple FIB rule (See § 5.2 for more details). The Dispatcher then divides the EP-model into multiple regions according to the distribution of updates (③). After that, the Updater identifies the EP changes in each region in multiple threads (④) and updates the EP-model to maintain its consistency with data plane forwarding behavior in parallel (⑤). Finally, the Verifier extracts the forwarding graph from the updated EP-model (⑥) and uses it to verify different specifications (⑦), note that here EPVerifier only verifies network invariants (loop and black-hole) by default, but users also have the option to provide additional user-define specfications such as reachability, waypointing, etc. (See § 5.3 for more details.)

4 Example

In this section, we use examples to illustrate 1) the difference between EP-model and EC-model (§ 4.1) and 2) the difference between switch-based partition for EP-model and subspace-based partition for EC-model (§ 4.2).

4.1 Network Model

As shown in Figure 3, our example is based on a small network of six switches. We use a directed edge-labeled graph to represent the forwarding state of the data plane, and the label on each edge indicates the packet flow on that edge. For simplicity, let us assume that an update storm consisting of three rules arrives at some point. The details of the three rules $r1$, $r2$ and $r3$ are shown in the top-middle of the Figure 3. The three rules are delivered to switches A, W and B and the set of matched packets are $\alpha_2, \alpha_3, \alpha_4$, respectively. For priority, $r1$, $r2$ and $r3$ have increasing priority and are higher than all existing rules in the network.

The initial state of the data plane is shown in the top-left corner of the Figure 3, where the packets matched by α_1 from S are forwarded to D via A, W and C. We show how α_1, α_2, α_3 and α_4 overlap with each other by parallel lines in the top-right corner of the Figure 3.

We now discuss how the EC-based method maintains EC-model. Since each equivalence class(EC) is a set of packets that experience the same forwarding actions throughout the network [20], thus, in our example, the network initially contains only one equivalence class, EC1, which denotes the identical group of packets represented by α_1. However, EC1 is continuously split into the final 7 ECs after inserting all three rules: First, with the insertion of $r2$, the action of all packets matched by α_2 on switch A changes from W to B because the priority of $r1$ is greater than all the old rules on switch A. This results in splitting the EC1 into EC2, EC3 and EC4 in the EC-model. After this, with the insertion of rule $r2$, EC4 is split into EC5, EC6 and EC7. Finally, after the insertion of rule $r3$, EC3 and EC6 are split into EC8, EC9 and EC10, EC11 respectively.

Network model oscillations. Note that although each rule insertion only affects the packet flow on 1-2 edges, each EC

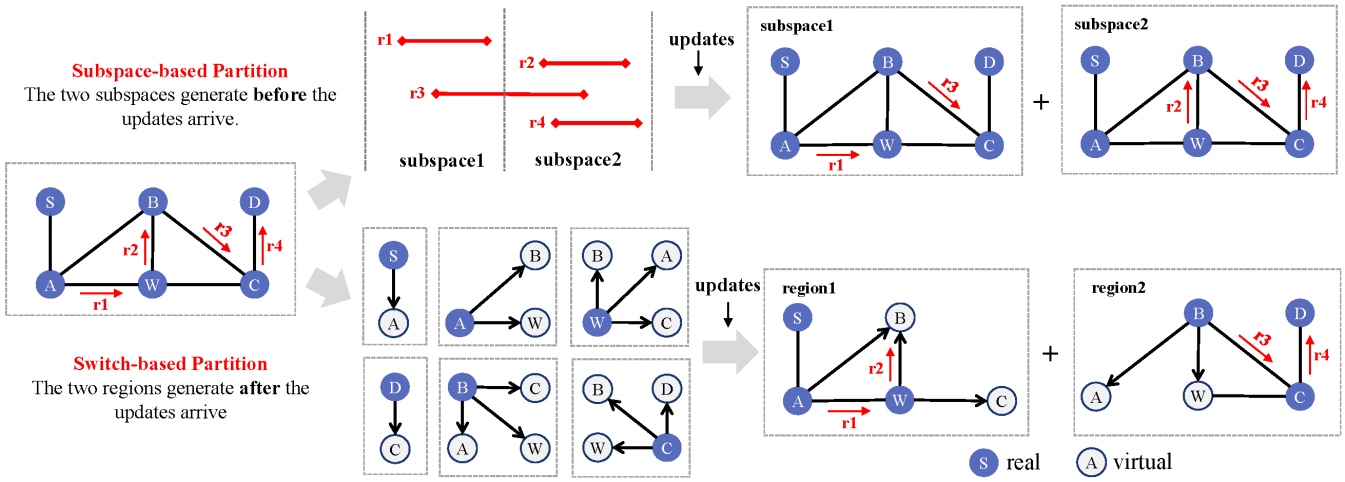

Figure 4: Two natural ways for the data plane verification partition.

split caused by rule insertions results in a large number of modifications to the EC model (red label in the forwarding graph). This is due to the fact that the equivalence classes of the entire network appear on a large number of edges in the EC-Model, *i.e.*, the global nature of ECs. Although the EC-based method can be improved in practice to avoid problems such as the explosion of the number of equivalence classes [37], redundant computation [15], the dramatic oscillation of the EC-model caused by changes in the set of equivalence classes are fundamentally unavoidable. Thus, we conclude that the global nature of EC fundamentally limits the speed of updating the network model and thus slows down the update storms verification.

We then illustrate how EPVerifier avoids the above oscillations of the network model and accelerates the verification time of update storms. Unlike the EC-model that divides packets into a set of ECs and uses ECs to label each directed edge in the network forwarding graph, the EP-model of EPVerifier represents packet flow in the network by maintaining edge predicates (EPs). An edge predicate is defined as follows.

Definition (Edge Predicate): An edge predicate(EP) is a 3-tuple$(from, to, P)$, indicating that any packet $p \in P$ will be forwarded by device $from$ to device to.

As the definition describes, each directed edge $(from, to)$ in the EP-model corresponds to each EP individually, determining that each EP in the network model will only appear once, *i.e.*, the local nature of EP. When a rule update comes, such local EPs allow EPVerifier to keep the network model consistent with the data plane by only modifying those EPs whose bound edges have packet flow changes. For example, as shown in Figure 3, we use the packet flow on each edge to represent the EPs in the network model. In this case, when rule $r1$ arrives, EP-model only needs to modify the EP on edge (A, W) and (A, B) because there are packet flow changes on them due to the installation of rule $r1$.

In summary, for the network model, the EC-based methods cannot avoid the oscillations in the network model caused by changes in the EC set due to the global nature of EC, which slows down the verification time of update storms. To further accelerate and achieve fast verification of update storms, EPVerifier uses EPs to represent packet flow in the network instead of global ECs to ensure that only the part of the network model with packet flow changes is modified.

4.2 Partition

Suppose we want to divide the overall verification task into two smaller tasks for parallel computation. As described in § 2, there are two natural ways to achieve parallelism: *subspace-based partition* and *switch-based partition*. We now illustrate the difference between them.

The subspace-based partition achieves parallelism by slicing the forwarding state of the overall network into multiple subspaces. Specifically, as shown in the left side of the Figure 4, The subspace-based partition creates two network models, each accepting the updates from subspace1 and subspace2, respectively. Ideally, when the updates are evenly distributed among different subspaces, this simple and straightforward partition can significantly reduce the number of updates a single network model needs to handle. However, since the subspaces and the corresponding network models are built *before* the arrival of update storms, the subspace-based partition will easily encounter the following limitations in practice: 1) updates are biased towards one subspace, *e.g.*, as shown in Figure 4, $r1, r2, r3$ and $r4$ are generally biased towards subspace2. 2) Some updates may affect multiple subspaces at the same time, *e.g.*, $r3$ affects both subspace1 and subspace2.

Unlike subspace-based partition, the purpose of the switch-based partition is to divide the forwarding graph into different clusters of switches, which we call *regions*, and manage the forwarding state of different regions in parallel. The key

Algorithm 1: Divide update storm R into N region update sequences

```
1 for i ∈ Range(len(V)) do
2     v ←the unvisited switch that has the maximum
        number of updates;
3     r ← the region sequence index that has the
        minimum number of updates;
4     add all updates on v to the sequence S_r;
5 return S;
```

Algorithm 2: GetHit(r)

```
1 r.hit ← r.match;
2 for r' ∈ r.from.rules do
3     if r'.prio > r.prio && r'.hit ∧ r.hit ≠ False then
4         r.hit ← r.hit ∧ ¬r'.hit;
5     if r.hit = BDD.False then break;
```

difference between switch-based partition and subspace-based partition is that the regions are divided *after* the update storms arrive. Therefore, switch-based partitioning can modify the partition of regions according to the update storms to make the number of updates in different regions as even as possible. For example, as shown in the right side of Figure 4, the switch-based partition first divides the network model in terms of switches. Then it combines the switches into region1 and region2 according to the rules distribution to ensure that the four rules are evenly distributed among the two regions.

While switch-based partition performance is better than subspace-based partition, EC-based methods are unable to apply this partition. This is because the edge label of the EC-model is a set of equivalence classes, which re-couples the various regions that were intended to be isolated from each other, and the synchronization costs associated with this coupling are unacceptable. Hence the state-of-the-art EC-based data plane verifier [15] only divides the network into multiple subspaces to squeeze out the parallel potential of the EC-model. In contrast, after employing edge predicates in place of equivalence classes, we make a key observation that EP's inherently local properties make it highly conducive for switch-based partition. This is because each update can affect only the EP of those edges emanating from the switch where the update is being installed, so EPVerifier can easily divide the EP-model into isolated regions and maintain them in parallel for fast verification of update storms.

5 Design Detail

In this section, we discuss the design detail of EPVerifier. We first show how to divide the network model into multiple regions by switches (§ 5.1). And then illustrate the EP-model updating process (§ 5.2), followed with an introduction on how to use the EP-model to check the correctness of the data plane (§ 5.3).

5.1 Switch-based Partition

Representation of the packet set P. Considering a packet with a match field of h bits, we can think of this packet as a boolean formula consisting of h boolean variables. For example, an IP match field of 128.0.0.0/16 can be represented as $x_1 \wedge \overline{x_2} \wedge \cdots \wedge \neg x_{16}$. Therefore, the EP, which represents a set of packets that can traverse through an edge, can also be represented as a Boolean formula. We adopt the methods of [33], use binary decision diagram (BDD [9]) to encode the packet matching field. This enables us to take advantage of the efficient logical operations provided by BDD, such as conjunction ($\wedge$), disjunction ($\vee$), and negation ($\neg$), in order to compute and update EPs.

EPs that belong to different switches. The prerequisite for efficient operations between different BDDs is to share the same boolean variables, so existing BDD libraries usually use BDD managers to organize BDDs that share the same boolean variables, and only BDDs that belong to the same manager can perform logical operations with each other. However, in such cases, BDDs that belong to the same BDD manager incur significant synchronization costs in multi-threaded situations. Therefore, to accelerate update storms verification, EPVerifier employs distinct BDD managers for EPs in different switches to circumvent synchronization costs. With distinct switch EPs, EPVerifier can then freely combine the switches in the network into multiple regions.

According to the above description, for a certain update storm R, in addition to the set of edges E and the set of points V that represent the topological information of the data plane, the EPVerifier uses three variables to maintain the EP model with N regions: S, B and M, as described in turn. First, $S_1 - S_N$ indicates the update sequences of each region, where N indicates the maximum number of regions in the network. Note that N should be smaller than the number of CPU cores to avoid frequent context switching. Second, B is an array of BDD managers. For each $v \in V$, $B[v]$ is the BDD manager of switch v. Namely, All EPs on switch v are encoded by $B[v]$. Finally, M is a hash table, which takes the network model edge as the key. For each edge $e(v,v')$, $M[e]$ is a BDD encoded by $B[v]$, indicating the set of packets that can pass on e. Note that each $M[e]$ is initialized to $BDD.False$ at the beginning to indicate that there is no traffic on e.

The algorithm 1 shows a sample program for dividing the update storm R into N region update sequences. It uses a greedy algorithm to generate N update sequences as evenly as possible. Specifically, the algorithm picks the switch with the maximum number of updates in the update storm (Line 2) and adds all the updates to the sequence with the fewest updates (Line 3-4). Finally, after all the switches have been

Algorithm 3: UpdateEP(S_i)

```
1  for (r, isInsert) ∈ S_i do
2      if isInsert then RuleInsert(r, i) ;
3      else RuleRemove(r, i) ;
4  Function RuleInsert(r, i):
5      GetHit(r);
6      M[(r.from, r.to)] ← M[(r.from, r.to)] ∨ r.hit;
7      for r' ∈ r.from.rules do
           // sorted by decreasing priorities
8          if r'.prio ≤ r.prio && r'.hit ∧ r.hit ≠
            BDD.False then
9              if r'.to ≠ r.to then
10                 M[(r.from, r'.to)] ←
                    M[(r.from, r'.to)] ∧ ¬(r'.hit ∧ r.hit);
11             r'.hit ← r'.hit ∧ ¬r.hit;
12     r.from.rules ← r.from.rules ∨ r;
13 Function RuleRemove(r, i):
14     M[(r.from, r.to)] ← M[(r.from, r.to)] ∧ ¬r.hit;
15     r.from.rules ← r.from.rules \ r;
16     for r' ∈ r.from.rules do
           // sorted by decreasing priorities
17         if r'.prio ≤ r.prio && r'.match ∧ r.hit ≠
            BDD.False then
18             if r'.to ≠ r.to then
19                 M[(r.from, r'.to)] ←
                    M[(r.from, r'.to)] ∨ (r'.match ∧ r.hit);
20             r.hit ← r.hit ∧ ¬r'.hit;
21             r'.hit ← r'.hit ∨ (r.hit ∧ ¬r'.hit);
22         if r.hit = BDD.False then break;
```

visited, the final set of region update sequences S is returned (Line 5).

5.2 Updating EP-model

As described in § 2, rules sharing a routing table may have overlapping match fields, so the set of packets actually affected by a rule's insert or remove is determined by both match and priority, *i.e.*, the hit field.

As shown in Algorithm 2, we calculate the hit field of each arriving rule update to obtain the impact of this update on the forwarding behavior of the data plane. And this procedure illustrates that a rule update may directly or indirectly affect two types of EPs: 1) the EPs on edge$(r.from, r.to)$, which are directly affected by rule r, 2) and the EPs on other edges' $from = r.from$ which are indirectly affected by rule r, note that these EPs are directly affected by other rules that share the routing table with the rule r.

Each region rule sequence S_i is an array of 2-tuple$(r, isInsert)$ items. Each item indicates an insertion ($isInsert = True$) or deletion ($isInsert = False$) of a rule r in the network model. Each rule r is specified as a 4-tuple $(from, to, match, priority)$, where $match$ means the set of packets matched by this rule, $from$ and to means that this rule will be installed to device $from$ and its action is forwarded to device to.

Algorithm 3 summarizes the update process of EPs in the region i. For all rules that have an impact on region i, the EPVerifier calls different methods for rule insertion and deletion separately (Line1-3). For the insertion of rule r (Line 4-12), EPVerifier first calculates the hit field of r according to the definition of hit (Line 5) and updates the EPs directly affected by r (Line 6). After that, the algorithm iterates through all the rules r' that share a routing table with r in descending order of priority (Line 8) and updates the EPs indirectly affected by r (Line 8-10), and maintains $r'.hit$ (Line 11). Finally, EPVerifier adds rule r to the rule set $rules$ of $r.from$ to complete the insertion of rule r (Line 12). For the deletion of rule r (Line 13-22), since the hit field is already computed during rule insertion, the EPs on edge$(r.from, r.to)$ can be updated directly at this point using $r.hit$ (Line 14) and the rule r deleted from $r.from.rules$ (Line 15). Subsequently, similar to rule insertion, EPVerifier updates the EPs directly affected by those rules with lower priority than r (Line 16-22). Besides, we find that a rule r affects or is affected by another rule r' when their match fields consisting of h bits overlap. So we can use Trie [22] instead of BDD conjunction to speed up the rule update installation. Such optimization is omitted from Algorithm 3.

5.3 Verification

EP transfer algorithm. As described in § 5.1, EPVerifier employs distinct BDD managers for EPs in different switches to circumvent synchronization costs. However, BDDs that belong to different BDD managers cannot operate logical operations with each other. This is unacceptable for checking network correctness, which requires the operation of EPs on different switches. Therefore EPVerifier needs to use the EP transfer algorithm to migrate the packet flow changes in different regions to the same BDD manager and generate a forwarding graph for checking network correctness.

The migration process of EP is the process of migrating BDDs from a regional BDD manager to a network BDD manager. Specifically, EPVerifier uses two steps to complete the migration of BDDs: 1) In the region BDD manager, a BDD is transformed into a set of formulas it represents. For instance, a BDD representation of $(x_1 \wedge \overline{x_2} \wedge \overline{x_3}) \vee (\overline{x_1} \wedge \overline{x_2} \wedge x_3)$ would yield a formula set: $[x_1 \wedge \overline{x_2} \wedge \overline{x_3}, \overline{x_1} \wedge \overline{x_2} \wedge x_3]$. 2) In the Network BDD Manager, the formulas are re-converted into binary decision diagrams (BDDs). While in the worst-case rule, updates may affect all formulas in the formula set of EPs and thus cause excessive EP transfer overhead, our experiments (§ 6.4) show that most rule updates have little impact on EP

Algorithm 4: TransferEP(e, old, i)

```
1  delta ← old \ e;
2  for formula ∈ delta.formulas do
3      M'[e.edge] ←
         M'[e.edge] \ B0.encoding(formula);
4      affected ← affected ∨ B0.encoding(formula);
5  delta ← e \ old;
6  for formula ∈ delta.formulas do
7      M'[e.edge] ←
         M'[e.edge] ∨ B0.encoding(formula);
8      affected ← affected ∨ B0.encoding(formula);
```

formulas, *i.e.* only 1-2 atomic formulas change. Therefore, EPVerifier performs incremental migration of changed EPs rather than starting from scratch to avoid redundant calculations. In other words, in addition to maintaining a hash table M that stores network traffic managed by different BDD managers, EPVerifier also maintains an M' that stores network traffic encoded by a same BDD manager. Meanwhile, the process of maintaining M' is incremental, *i.e.*, after the updating of M, the EP transfer algorithm is used to migrate the changes in M into M'. We use B_0 to represent the BDD manager of M'. The algorithm 4 summarizes the incremental migration of an EP change $old \rightarrow e$ from M to the M'. Note that update storms may have both rule insertions and deletions, so the algorithm 4 migrates the decreases (Line 1-4) and increases (Line 5-8) of e into M' in two parts and records the set of packets $affected$ by this update storm for verification.

Algorithm 5: CheckInvariants(G)

```
1  for s ∈ V do
2      Traverse(n, affected, {});
3  Function Traverse(s, p, history):
4      if p = B0.False then return;
5      if s ∈ history then Alert('loop');
6      for (s, s') ∈ E do
7          if M'[(s, s')] ≠ B0.False then
8              p ← p \ M'[(s, s')];
9              history ∨ s';
10             Traverse(s', M'[(s, s')] ∧ p, history);
11             history \ s';
12             M'[(s, s')] ← M'[(s, s')] \ (M'[(s, s')] ∧ p);
13     if p ≠ B0.False then Alert('blackhole');
```

Network invariants. With the forwarding graph $G(V,E,M')$, operators can check network invariants, including loop and blackhole, by traversing G. The algorithm 5 summarizes the process by which EPVerifier verifies network invariants using the network forwarding graph. The algorithm starts the depth-first search (DFS) traversal from each node $n \in V$(Line1). In addition to the s representing the current node, the DFS traversal process carries two parameters: p and $history$, as described in turn. First, p represents the packets carried during the current traversal, encoded by BDD. Second, $history$ stores the nodes that have been visited. p is initialized to the set of packets $affected$ by this update storm at the beginning of the traversal, while $history$ is initialized to the empty set (Line2). The traversal stops when p is empty (Line 4). When the traversal reaches a previously traversed node, a forwarding loop exists in the network (Line 5). Otherwise, the algorithm starts with the neighbor node s' of s in the G for DFS (Line 6). For every s', the algorithm updates history and carries $M'(s,s') \wedge p$ for DFS (Line 8-11), and after the traversal of s', the algorithm avoids repeated traversals by updating M' (Line 12). Finally, when all neighboring nodes are traversed, a blackhole exists if packets remain in s (Line 13).

Algorithm 6: CheckOtherSpecification(G, l)

```
1  M'[l] ← B0.False;
2  Reach ← {};
3  Traverse(source, match, waypoints, dst);
4  for node ∈ (waypoints ∨ dst) do
5      if Reach[node] ≠ match then Alert('Failure');
6  Function Traverse(source, p, waypoints, dst):
7      if match = B0.False then return;
8      if source = dst then
9          Reach[source] ← Reach[source] ∨ match;
10         return;
11     if source ∈ waypoints then
         Reach[source] ← Reach[source] ∨ match;
12     for (source, next) ∈ E do
13         p ← p \ M'[(source, next)];
14         Traverse(next, M'[(source, next)] ∧
             p, waypoints, dst);
```

Other specifications. In addition to network invariants, Operators may need to check other specifications, such as whether certain packets can be forwarded from device a through device b to device c or whether network invariants are violated when a link is broken, etc. We now explore how EPVerifier verifies these specifications. Here, we define a specification as a 4-tuple$(match, source, waypoints, dst)$, *i.e.*, all packets matched by a match from device source must be forwarded in the network via $waypoints$ must be forwarded to device dst. For such a policy, we carry the packets representing all the matched packets from the source and traverse G to simulate the packet forwarding process in the network. This policy is not violated only when all packets can pass through $waypoints$ to dst intact. In the case of a change in network state, such as a broken link, we modify the network forwarding graph before traversal.

Table 1: Dataset Information.

Network	Node	Links	FIB rules	Storm Size
Airtel1	68	220	6.89×10^4	10K
Airtel2	68	260	9.84×10^4	10K
Berkeley	23	252	1.28×10^7	1M
INET	315	40770	2.49×10^8	1M
RF1755	87	2308	3.37×10^7	1M
RF3257	161	9432	7.45×10^7	1M
RF6461	138	8140	7.50×10^7	1M

The algorithm 6 summarizes how EPVerifier uses the network forwarding graph $G(V,E,M')$ to verify that when the link l fails, all packets from *source* that are matched by *match* can are forwarded to device *dst* via *waypoints*. The algorithm first sets the EP on l to False to simulate a link failure (Line 1), and then initializes the variable *Reach* (Line 2), which records the packets that can be reached on each *waypoint*. Unlike the verification invariant, the algorithm directly carries the set of packets *match* of interest to the user for DFS traversal (Line 3). During the subsequent traversal (Line 6-14), the algorithm keeps track of the packet forwarding process by maintaining the *Reach* variable. After the traversal is completed, the specification is verified by checking whether all packets in *match* reach *waypoints* and *dst* (Line 4-5).

6 Evaluation

We fully implement EPVerifier as an easy-to-use tool and test it exhaustively with trace-driven simulations and deployments in the wild.

6.1 Setup

Implementation. We implemented EPVerifier as an easy-to-use tool in ∼4000 lines of Java code. In particular, EPVerifier exposes a ***init()*** function that accepts data plane topology and configuration information for initializing the EP-model and a ***verify()*** function that accepts update storms with user-defined specifications (if any) for verifying data plane correctness. For the parallelism part, we use the standard asynchronous thread of OpenJDK. For BDD operations, we use JDD, a BDD library for Java [30].

Dataset. Table 1 summarizes the information about the dataset we use. They all come from the open source dataset [1,17]. The Airtel1 and Airtel2 are generated using the ONOS SDN-IP application [3,11]. And the remaining five datasets are synthetic datasets generated using the mechanism in [36]. Specifically, [17] collects IP prefixes from real-world BGP updates collected by the Route Views project [4] and computes the shortest paths in the network topology [27].

Methods to compare. We compare EPVerifier with two state-of-the-art data plane verifiers, APKeep [37] and Flash [15], because their evaluation identifies that they achieve the most generic and fastest data plane verification. 1) **APKeep***: Since we did not find an open-source implementation of APKeep, we implement APkeep ourselves following the pseudocode in [37], referred to as **APKeep***. 2) **Flash**: we use its open-source implementation in Java [2]. Apart from the above two methods, we also consider **EPVerifier***, standing for EP-model with subspace-based partition instead of switch-based partition to evaluate the performance gap between the subspace-based partition and our switch-based partition.

Simulation objective. We use the above dataset traces to discuss the following questions:

- what is the overall performance gap between EPVerifier and the state-of-the-art data plane verifiers? (§ 6.2)
- what is the performance gap between the EP-based network model and the EC-based network model? (§ 6.3)
- what is the performance gap between switch-based partition and subspace-based partition? (§ 6.4)

Simulation testbed. We run all our simulation experiments on a Ubuntu 20.04(x64) LTS with a 3.4 GHz Intel 8-core CPU and 32 GB of RAM. The OpenJDK v19.0.2 is installed for Java support.

Deployment. Apart from simulations, We deploy EPVerifier and Flash onto a network control plane of a commercial data center backbone network for testing the real-world performance of EPVerifier (§ 6.5). The placement and operations are similar to those described in § 3, operators provide information about the update storms they are interested in, as well as a snapshot of the network before the update storms are issued, to the verifiers, which then parse these information and check whether the delivery of each update storm violates network invariants or user-defined specifications.

6.2 Overall Performance

We now evaluate the overall performance of EPVerifier in dealing with update storms. For network correctness, we verify the presence of forwarding loops in the network because the verification of forwarding loops requires completely traversing the entire forwarding graph while other specifications only need to traverse part of it. Since we use an 8-core CPU, we divide the network into eight subspaces or eight regions to make full use of the eight threads. As described in Table 1, for each dataset, we randomly select 10K or 1M rule insertions to form a update storm and deliver it to EPVerifier. After this update storm is installed and the loop is verified, we record the memory consumption and install another update storm, which is composed of 10K or 1M rule deletions in the same order.

The overall time, memory consumption and the number of BDD operations for the above process under different scenarios and datasets are summarized in Table 2.

In terms of time consumption, we can see that EPVerifier achieves significant performance improvements on most datasets, especially on the last four datasets where the network topology is dense and prone to update storms; for example,

Table 2: Time, memory cost and BDD operations of different methods

Network	Time cost (s)				Memory Usage (MB)				BDD Operations			
	APKeep* (speedup)	Flash (speedup)	EPVerifier* (speedup)	EPVerifier (speedup)	APKeep*	Flash	EPVerifier*	EPVerifier	APKeep*	Flash	EPVerifier*	EPVerifier
Airtel1	0.44 (4.9×)	0.10 (1.1×)	0.03 (0.3×)	**0.09** (1×)	63.76	64.15	65.67	407.41	1×10^6	3×10^5	8×10^4	5×10^5
Airtel2	0.23 (2.6×)	0.08 (0.9×)	0.03 (0.3×)	**0.09** (1×)	66.37	63.78	65.37	399.48	1×10^6	3×10^5	1×10^5	6×10^4
Berkeley	2262.15 (157×)	34.29 (2.4×)	41.57 (2.9×)	**14.42** (1×)	1434.57	1312.56	1866.29	2829.26	7×10^9	1×10^7	3×10^6	4×10^6
INET	14458.69 (557×)	800.65 (31×)	38.60 (1.5×)	**25.98**(1×)	8178.28	6124.18	2149.38	5253.45	2×10^{10}	3×10^9	7×10^7	7×10^7
RF1755	10196.17 (856×)	100.08 (8.4×)	48.03 (4×)	**11.91** (1×)	2924.07	2419.23	2018.21	3364.27	2×10^{10}	2×10^8	4×10^6	5×10^6
RF3257	10473.87 (785×)	208.95 (15.7×)	37.05 (2.8×)	**13.34** (1×)	4947.75	3876.56	2191.80	3987.75	2×10^{10}	7×10^8	1×10^7	1×10^7
RF6461	8736.91 (668×)	157.57 (12×)	34.51 (2.6×)	**13.07** (1×)	4502.41	3567.80	2135.69	3987.23	2×10^{10}	6×10^8	1×10^7	1×10^7

EPVerifier is 557× and 31× faster than APKeep and Flash respectively on the INET dataset, and at least 10x faster on the last three datasets. The overhead of the network model oscillations (§ 4.1) depend mainly on the number of edges in the network model, so for the Airtel1 and Airtel2 with fewer edges and small storm sizes, the acceleration of update storm verification by EPVerifier is insignificant because the advantage gained by EPVerifier in the updating network model phase is overshadowed by the subsequent EP transfer (§ 5.3). In the meantime, the EPVerifier* using subspace-based partition does not require EP transfer and therefore performs best in these two datasets.

In terms of memory usage, EPVerifier uses more memory when the network is small because it uses a separate BDD manager on each switch and each BDD manager needs to allocate a certain size of cache when it is initialized. However, this memory consumption does not explode as the complexity of the network rises because the total number of updates as the source of BDDs is the same regardless of the number of BDD managers. It can be seen that in the last four dense networks, EPVerifier's memory consumption compares to that of Flash and APKeep.

Finally, the number of BDD operations is good evidence of the time performance improvement of EPVerifier. In all datasets, the number of BDD operations of EPVerifier* and EPVerifier is at least one magnitude less than that of Flash and APKeep.

In summary, we can conclude that EPVerifier achieves robustly fast update storm verification. In dense networks that are prone to update storms, EPverifier is more than 10× faster than state-of-the-art.

6.3 Performance of EP-model

We now consider the performance gap between the equivalence-class-based network model (EC-model) and our edge-predicate-based network model (EP-model). For all datasets, we randomly select updates at different sizes (starting from 1) to form update storms and use Flash and EPVerifier to verify the forwarding loop, respectively. We do not apply any partition to these verifiers to focus on the effects of the network model.

The result is shown in Figure 5. We find that the advantage of EP-model is not obvious when the storm size is small. This is because when the storm size is small, the additional BDD operations brought about by EP transfer and traversing the forwarding graph are not negligible compared to the improvement brought about by the model update. However, as the storm size increases, the EP-model advantage starts to emerge and this advantage increases with the size of the update storm. Notice that the acceleration effect of EP-model is not the same for different datasets. This is because the number of edges is different in each dataset. When the number of edges is higher, the number of labels that need to be modified for the network oscillation (§ 4.1) of EC-model is also higher. Therefore, the improvement brought by EP-model is more obvious at this time. For example, for the INET dataset with tens of thousands of edges, EP-model is much faster than EC-model even when the storm size is very small. Thus, we can conclude that the advantage of EP-model expands with the complexity of the data plane and the growth of storm size.

6.4 Performance of Switch-based Partition

Compared to subspace-based partition. We evaluate the performance gap between our switch-based partition and subspace-based partition. Specifically, we divide the network into two subspaces and manually place updates into the two subspaces to achieve a different distribution of updates in the subspaces. We then compare the performance of verifying the update storm composed of these updates under the two partitioning approaches with EP-model. The result is shown in Figure 6. The x-axis is the ratio of the variance of the number of updates in the two subspaces to the total number of updates, which indicates the average distribution degree of updates in the subspaces. Obviously, when the value of x is between 0 and 1, when x is 0, it means that the updates are evenly distributed in two subspaces, and when it is 1, it means that all updates are in one subspaces. The y-axis is the ratio of the verification time of the two partition methods. Thus, the switch-based partition performs better when y is greater than 1. We can see that the verification time of the subspace-based partition rises as the variance of the rules in the two subspaces increases. For the last five datasets in Table 1 with complex topology and large storm sizes, the switch-based partition still outperforms the subspace-based partition even if the updates are evenly distributed among the two subspaces. This is because the subspace-based partition uses only two BDD managers, one for each subspace, while the switch-based partition assigns a BDD manager to each switch.

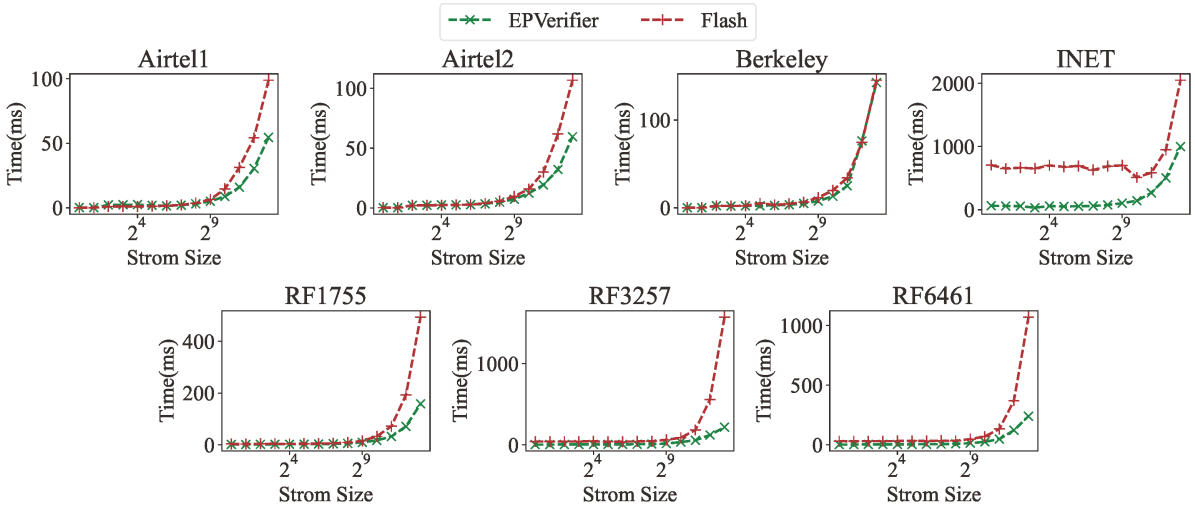

Figure 5: Verification time for different storm sizes

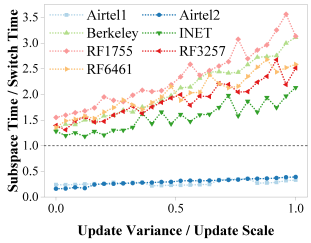

Figure 6: Effects of updates distribution on subspace-based partition verification time.

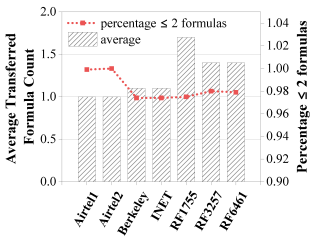

Figure 7: Each update only affects a few formulas.

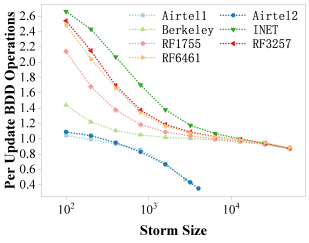

Figure 8: Per update BDD operations decrease due to the possible overlap of formulas affected by each update.

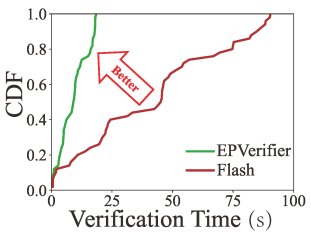

Figure 9: The update storms verification time in real-world network.

Too many BDDs in the BDD manager will slow down the BDD operations. In addition, for Airtel1 and Airtel2 datasets with simple topology and small storm size, the performance of the subspace-based partition is better because the number of BDDs in the model is limited in this case and the switch-based partition needs to go through a redundant EP transfer process.

Cost of EP transfer. Since switch-based partitions have to suffer additional EP transfer progress due to the limitation of BDD (§ 5.3), we now discuss the overhead of it. First, the complexity of the EP transfer algorithm depends mainly on the number of formulas that need to be transferred to the forwarding graph. Therefore, for each dataset, we select the device with the highest number of updates and apply these updates one by one to the EP-model and transfer them to the forwarding graph. We record the number of formulas affected by each update. Figure 7 shows the results. We can see that for all datasets, most of the updates (more than 97%) affect only one or two formulas, and the average number of transferred formulas does not exceed 2. Second, the main time of the EP transfer process is spent on the BDD operations. We generate update storms of different sizes and calculate the per update BDD operations during the EP transfer. As shown in Figure 8, for all datasets, with storm size increases, per update BDD operations tend to decrease due to the possible overlap of formulas affected by each update. Then, we can say that the overhead of EP transfer is acceptable, i.e., it does not explode with the complexity of the data plane and the storm size.

6.5 Deployment in the Wild

We evaluate the performance of EPVerifier by deploying it and Flash onto network control plane of a commercial data center backbone network with hundreds of switches and thousands of links. We verify 50 update storms with EPVerifier and Flash respectively and statistics the verification time of the both. The results are shown in Figure 9. The x-axis is the verification time and the y-axis is the Cumulative Distribution Function (CDF). Similar to the simulation results, EPVerifier still greatly accelerates update storm verification in real deployments.

7 Discussion and Future Work

The update storms. The concept of update storms is introduced by Flash [15]. However, a similar phenomenon of network update aggregation is prevalent in networks because the source of network updates, a change in the user's intent, generally affects a certain number of devices (virtual or physical), resulting in a large number of network updates. For example, in modern multi-tenant data centers, fast programming interfaces in the forwarding plane may result in thousands of updates per second [18].

Packet transformations. In addition to normal forwarding rules, there may be rules in the real network that can modify packet information, such as NAT. For these transformation rules, we can extend our design to handle them just like the fine-grained PPM network model of APKeep [37]. More specifically, we can model each transformation rule as a node in the network model to simulate the packet rewrite triggered by the corresponding rule. We leave a full design and implementation to future work.

Distributed verification. Most data plane verification tools use a centralized architecture where the server collects all data plane information and verifies it. While this architecture is inherently non-scalable, EPVerifier's switch-based partition is very suitable for implementing distributed, on-device data plane verification. A preliminary idea is that we can maintain the individual EP information of each switch and verify it centrally. Detailed design is left as one of our future works.

8 Related Work

Data plane verification. There is a long line of research on data plane verification. The early data plane verifiers [6, 7, 19, 23] use different formal methods to analyze the data plane snapshot. While useful, these verification tools are too poor to detect network errors in a timely manner and cannot handle real-time changes in the data plane to ensure that the results are correct in real-time. To overcome the above limitations, [17, 18, 20, 33] use novel algorithms and achieve per rule update real-time verification. Veriflow [20] is the first data plane verification tool that implements real-time verification by dividing packets into equivalence classes to split the verification task of the entire network into multiple independent verification tasks for the equivalence classes. Concurrent with Veriflow, Netplumber [18] implements incremental data plane verification based on Header Space Analysis (HSA). It is worth noting that Netplumber's HSA-based scheme for modeling network traffic information as a plumbing graph has similar localization properties as EP-model, but when the traffic in the network becomes progressively more complex, the plumber graph's node-by-node approach to the traffic limits its speed of verification, as demonstrated by experiments by [37]. Although these real-time data plane verifiers can achieve millisecond or even sub-millisecond verification speed, problems such as the exploding number of equivalence classes, excessive redundant computations, and inability to handle multiple match field still limit them. To further extend data plane verification, APKeep [37] solves the equivalence class explosion problem by merging equivalence classes, but when there are plenty of rule updates, many redundant computations in APKeep's PPM model still slow down the verification speed. Flash [15] avoids the redundant computations in the equivalence class model through a series of improvements and engineering optimizations. Inspired by [36], Flash divides the verification task according to subspace to speed up the verification of update storms. However, Flash still cannot solve the network oscillation problem caused by the equivalence class itself.

Control plane verification analyzes control plane information, network topology, and environmental context to verify the forwarding behavior of all packets and network intent under the data plane, which is generated by combining this information [5, 10, 13, 14, 25, 26, 32, 34]. They are complementary to EPVerifier and can use EPVerifier to accelerate the verification of the generated data planes.

Stateful network verification. Compared to data plane verification, the progress of stateful network verification is at a lower level and can only accomplish offline verification tasks. [28, 29] model stateful middleware by adding state information to packet headers based on symbolic execution techniques, and propose a modeling language SEFL to speed up the symbolic execution process. [24] proposes a middleware abstraction modeling method based on an SMT solver for verifying reachability and improves the verification speed by reducing the verification network size. [31] solves the verification problem of traffic isolation property by abstracting the processing order of packets. [8] further abstracts the state change of middleware based on [31] to achieve complexity optimization. [35] is implemented based on a symbolic model detection technique, which abstracts the network into individual packet forwarding models and achieves high scalability with a customized symbolic model detection algorithm.

9 Conclusion

This paper presented EPVerifier, a fast, partitioned data plane verification for update storms. EPVerifier achieves robustly fast update storm verification compared to the state-of-the-art. To achieve this, EPVerifier introduces an EP-based network model to avoid the network model oscillation and a switch-based Partitioning scheme to partition the update storm verification task into smaller parts for parallel computation. Both dataset trace-driven simulations and the deployments in the wild show that EPVerifier achieves substantial gains compared with the state-of-the-art. We believe the design of EP-model and switch-based partition can help data plane verification scale to large, complex data planes, which are prone to generate update storms.

Acknowledgements. We thank the anonymous NSDI review-

ers and our shepherd Brighten Godfrey for their valuable feedback. We thank Bokun Li, Jinming Wu and Ruilong Ma for their invaluable feedback on an early draft of this paper. This work is supported in part by the National Natural Science Foundation of China under Grants (62101064, 62171057, 62201072, 62071067), in part by the Ministry of Education and China Mobile Joint Fund (MCM20200202), Beijing University of Posts and Telecommunications-China Mobile Research Institute Joint Innovation Center.

References

[1] Delta-net. https://github.com/delta-net/datasets.

[2] Flash artifact for sigcomm22. https://github.com/snlab/flash.

[3] The onos project. https://opennetworking.org/onos/.

[4] Route views. http://www.routeviews.org/.

[5] ABHASHKUMAR, A., GEMBER-JACOBSON, A., AND AKELLA, A. Tiramisu: Fast and general network verification. *arXiv preprint arXiv:1906.02043* (2019).

[6] AL-SHAER, E., AND AL-HAJ, S. Flowchecker: Configuration analysis and verification of federated openflow infrastructures. In *Proceedings of the 3rd ACM workshop on Assurable and usable security configuration* (2010), pp. 37–44.

[7] AL-SHAER, E., MARRERO, W., EL-ATAWY, A., AND ELBADAWI, K. Network configuration in a box: Towards end-to-end verification of network reachability and security. In *2009 17th IEEE International Conference on Network Protocols* (2009), IEEE, pp. 123–132.

[8] ALPERNAS, K., MANEVICH, R., PANDA, A., SAGIV, M., SHENKER, S., SHOHAM, S., AND VELNER, Y. Abstract interpretation of stateful networks. In *Static Analysis: 25th International Symposium, SAS 2018, Freiburg, Germany, August 29–31, 2018, Proceedings 25* (2018), Springer, pp. 86–106.

[9] ANDERSEN, H. R. An introduction to binary decision diagrams. *Lecture notes, available online, IT University of Copenhagen* (1997), 5.

[10] BECKETT, R., GUPTA, A., MAHAJAN, R., AND WALKER, D. A general approach to network configuration verification. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (2017), pp. 155–168.

[11] BERDE, P., GEROLA, M., HART, J., HIGUCHI, Y., KOBAYASHI, M., KOIDE, T., LANTZ, B., O'CONNOR, B., RADOSLAVOV, P., SNOW, W., ET AL. Onos: towards an open, distributed sdn os. In *Proceedings of the third workshop on Hot topics in software defined networking* (2014), pp. 1–6.

[12] DONNELLY, C. Microsoft 365 outage affecting teams, outlook and azure users blamed on 'networking fault'. Website, 2023. https://www.computerweekly.com/news/252529561/Microsoft-365-outage-affecting-Teams-Outlook-and-Azure-users-blamed-on-networking-fault.

[13] FAYAZ, S. K., SHARMA, T., FOGEL, A., MAHAJAN, R., MILLSTEIN, T. D., SEKAR, V., AND VARGHESE, G. Efficient network reachability analysis using a succinct control plane representation. In *OSDI* (2016), vol. 16, pp. 217–232.

[14] GEMBER-JACOBSON, A., VISWANATHAN, R., AKELLA, A., AND MAHAJAN, R. Fast control plane analysis using an abstract representation. In *Proceedings of the 2016 ACM SIGCOMM Conference* (2016), pp. 300–313.

[15] GUO, D., CHEN, S., GAO, K., XIANG, Q., ZHANG, Y., AND YANG, Y. R. Flash: fast, consistent data plane verification for large-scale network settings. In *Proceedings of the ACM SIGCOMM 2022 Conference* (2022), pp. 314–335.

[16] HERN, A. Google suffers global outage with gmail, youtube and majority of services affected. Website, 2020. https://www.theguardian.com/technology/2020/dec/14/google-suffers-worldwide-outage-with-gmail-youtube-and-other-services-down.

[17] HORN, A., KHERADMAND, A., AND PRASAD, M. R. Delta-net: Real-time network verification using atoms. In *NSDI* (2017), vol. 17, pp. 735–749.

[18] KAZEMIAN, P., CHAN, M., ZENG, H., VARGHESE, G., MCKEOWN, N., AND WHYTE, S. Real time network policy checking using header space analysis. In *NSDI* (2013), pp. 99–111.

[19] KAZEMIAN, P., VARGHESE, G., AND MCKEOWN, N. Header space analysis: Static checking for networks. In *Presented as part of the 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)* (2012), pp. 113–126.

[20] KHURSHID, A., ZOU, X., ZHOU, W., CAESAR, M., AND GODFREY, P. B. VeriFlow: Verifying Network-Wide invariants in real time. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)* (Lombard, IL, Apr. 2013), USENIX Association, pp. 15–27.

[21] LABUSCHAGNE, H. Big rsaweb outage. Website, 2023. https://mybroadband.co.za/news/fibre/478521-big-rsaweb-outage.html.

[22] MAABAR, M. Trie data structure. https://bioinformatics.cvr.ac.uk/trie-data-structure/, 2014.

[23] MAI, H., KHURSHID, A., AGARWAL, R., CAESAR, M., GODFREY, P. B., AND KING, S. T. Debugging the data plane with anteater. *ACM SIGCOMM Computer Communication Review 41*, 4 (2011), 290–301.

[24] PANDA, A., LAHAV, O., ARGYRAKI, K. J., SAGIV, M., AND SHENKER, S. Verifying reachability in networks with mutable datapaths. In *NSDI* (2017), vol. 17, pp. 699–718.

[25] PEDROSA, A. F. S. F. L., WALRAED-SULLIVAN, M., AND MILLSTEIN, R. G. R. M. T. A general approach to network configuration analysis. In *12th USENIX symposium on networked systems design and implementation (NSDI 15)* (2015), pp. 469–483.

[26] PRABHU, S., CHOU, K. Y., KHERADMAND, A., GODFREY, B., AND CAESAR, M. Plankton: Scalable network configuration verification through model checking. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)* (2020), pp. 953–967.

[27] SPRING, N., MAHAJAN, R., AND WETHERALL, D. Measuring isp topologies with rocketfuel. *ACM SIGCOMM Computer Communication Review 32*, 4 (2002), 133–145.

[28] STOENESCU, R., POPOVICI, M., NEGREANU, L., AND RAICIU, C. Symnet: static checking for stateful networks. In *Proceedings of the 2013 workshop on Hot topics in middleboxes and network function virtualization* (2013), pp. 31–36.

[29] STOENESCU, R., POPOVICI, M., NEGREANU, L., AND RAICIU, C. Symnet: Scalable symbolic execution for modern networks. In *Proceedings of the 2016 ACM SIGCOMM Conference* (2016), pp. 314–327.

[30] VAHIDI, A. Jdd: a pure java bdd and z-bdd library. https://bitbucket.org/vahidi/jdd, 2003.

[31] VELNER, Y., ALPERNAS, K., PANDA, A., RABINOVICH, A., SAGIV, M., SHENKER, S., AND SHOHAM, S. Some complexity results for stateful network verification. In *Tools and Algorithms for the Construction and Analysis of Systems: 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings 22* (2016), Springer, pp. 811–830.

[32] WEITZ, K., WOOS, D., TORLAK, E., ERNST, M. D., KRISHNAMURTHY, A., AND TATLOCK, Z. Scalable verification of border gateway protocol configurations with an smt solver. In *Proceedings of the 2016 acm sigplan international conference on object-oriented programming, systems, languages, and applications* (2016), pp. 765–780.

[33] YANG, H., AND LAM, S. S. Real-time verification of network properties using atomic predicates. *IEEE/ACM Transactions on Networking 24*, 2 (2015), 887–900.

[34] YE, F., YU, D., ZHAI, E., LIU, H. H., TIAN, B., YE, Q., WANG, C., WU, X., GUO, T., JIN, C., ET AL. Accuracy, scalability, coverage: A practical configuration verifier on a global wan. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication* (2020), pp. 599–614.

[35] YUAN, Y., MOON, S.-J., UPPAL, S., JIA, L., AND SEKAR, V. Netsmc: A custom symbolic model checker for stateful network verification. In *NSDI* (2020), pp. 181–200.

[36] 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 *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)* (2014), pp. 87–99.

[37] ZHANG, P., LIU, X., YANG, H., KANG, N., GU, Z., AND LI, H. Apkeep: Realtime verification for real networks. In *NSDI* (2020), pp. 241–255.