Fast BGP Simulation of Large Datacenters

Nuno P. Lopes and Andrey Rybalchenko

Microsoft Research

Abstract. Frequent configuration churn caused by maintenance, upgrades, hardware and firmware failures regularly leads to costly outages. Preventing network outages caused by misconfigurations is important for ensuring high network availability. Dealing with production datacenters with thousands of routers is a major challenge.
Network verification inspects the forwarding tables of routers. These tables are determined by the so-called control plane, which is given by the steady state of the routing protocols. The ability to simulate routing protocols given router configuration files and thus obtain the control plane is a key enabling technology.
In this paper, we present FastPlane, an efficient BGP simulator. BGP support is mandated by modern datacenter designs, which choose BGP as the routing protocol. The key to FastPlane's performance is our insight into the routing policy of cloud datacenters that allows the usage of a generalized Dijkstra's algorithm. The insight reveals that these networks are monotonic, i.e., route advertisements decrease preference when propagated through the network.
The evaluation on real world, production datacenters of a major cloud provider shows that FastPlane 1) is two orders of magnitude faster than the state-of-the-art on small and medium datacenters, and 2) goes beyond the state-of-the-art by scaling to large datacenters. FastPlane was instrumental in finding several production bugs in router firmware, routing policy, and network architecture.

1 Introduction

Preventing network outages caused by misconfigurations is important for ensuring high network availability. It is particularly relevant for public cloud infrastructures where an outage can affect thousands of customers [35].

Computing the network control plane is a crucial building block to prevent outages, as it consists of routing tables (RIBs) that determine network connectivity. These tables can be automatically inspected to check validity of configuration intents related to connectivity, as well as fault-tolerance and performance.

The ability to compute control planes from router configuration files and topology information enables static, dynamic, and design-time verification scenarios. Statically, i.e., before deploying a configuration into production, we first compute the control plane and verify its properties. If all checks pass, the configuration can be deployed with increased confidence. Some configuration intents can also be validated when the network is designed. For example, the computed

control plane can demonstrate whether the required level of fault-tolerance and load-balancing is achievable. Unfortunately, static checks are not sufficient, due to bugs in router firmware. Hence there is a need for dynamic checking as well, i.e., once a configuration is already deployed. Cross-checking the computed control plane with the one from production routers can uncover firmware bugs.

Due to lack of adequate validation tools, frequent configuration churn in datacenter networks caused by maintenance, upgrades, hardware and firmware failures regularly leads to costly outages. Scaling control plane computation to thousands of datacenter routers and network prefixes is still an open problem [42].

In this paper we present FastPlane, a tool for fast BGP simulation of large datacenters. Support for BGP is mandated by best practices in modern datacenter design, where BGP runs on each router [6, 31, 32]. The key to FastPlane's scalability is our insight into the routing policy that is revealed through a study of production configurations deployed by a major cloud provider. The insight shows that the network is monotonic, i.e., route advertisements decrease preference when propagated through the network [44]. It allows the deployment of a generalized form of Dijkstra's algorithm. FastPlane executes Dijkstra's algorithm over route advertisements instead of numeric path weights.

We adapt Dijkstra's algorithm to directly perform route advertisement propagation. Instead of numeric weight addition when traversing a graph edge, we apply routing policy determined by configuration files. The order of priority queue is no longer arithmetic comparison, but route preference order determined by BGP RFC/vendor specifications. The result corresponds to the control plane of the datacenter network once it reached a stable state [25].

We evaluated FastPlane on all production datacenters of a major cloud provider, and compared it with the state-of-the-art control plane verifier Batfish. For small and medium datacenters, FastPlane is two orders of magnitude faster than Batfish. For large datacenters, FastPlane finishes in a few minutes while Batfish either times out after one CPU week or runs out of memory.

Control planes computed by FastPlane exposed several bugs. A bug in the redistribution policy of connected routes was discovered by comparing computed RIBs with expected entries specified by network operators. This bug was fixed in production. A firmware bug that caused the RIB to contain different next-hops than the forwarding table was caught by cross-checking production against computed control planes. By similar cross-checking we also discovered a bug in high level routing architecture that causes a non-deterministic drop in fault-tolerance and load-balancing. Mitigation measures for this bug are underway.

In summary, we contribute a scalable algorithm for fast BGP simulation of datacenter networks. It exploits monotonicity of datacenter routing policy, from which we derive the applicability of a shortest path-based characterization of the control plane, yet, for the first time, expressed over route advertisements instead of numeric weights. Our implementation scales to large production datacenters, which are out of reach for the state-of-the-art.

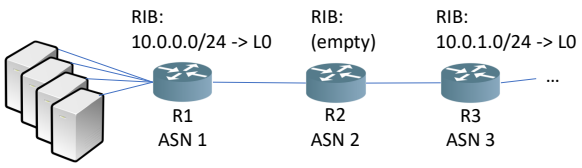

Fig. 1. Example network with three routers running BGP. We show the initial RIB of each router, i.e., before any information has been exchanged between neighbors.

```
! ---------------- Router R1 -----------------------
interface Ethernet0                 ! physical port connected to R2
  ip address 172.16.0.0/31

bgp router 1                        ! run BGP with ASN 1
  network 10.0.0.0/24               ! export prefix to neighbors
  neighbor 172.16.0.1 remote-as 2   ! peer with R2

! ---------------- Router R2 -----------------------
interface Ethernet0                 ! physical port connected to R1
  ip address 172.16.0.1/31

bgp router 2                        ! run BGP with ASN 2
  ! export prefix if any sub-prefix in RIB
  aggregate-address 10.0.0.0/16 summary-only
  neighbor 172.16.0.0 remote-as 1   ! peer with R1
  neighbor 172.16.0.3 remote-as 3   ! peer with R3
```

Fig. 2. Configuration fragments for routers R1 and R2 in Fig. 1. R1 exports the prefix used by directly connected servers. R2 aggregates and exports the prefix 10.0.0.0/16 whenever a more specific prefix exists in the RIB. At the same time, R2 blocks advertisement of the more specific prefixes.

2 Datacenters and BGP

Modern datacenter designs choose BGP as the routing protocol to compute RIBs [2, 23, 32]. By running BGP each datacenter router participates in a distributed best path computation, where information about the best paths is exchanged between direct neighbors. The cost metric is not, however, the number of hops in the path, but rather a lexicographic order of several path attributes.

Each router has an autonomous system number (ASN). ASNs are used to keep track of the path an advertisement has taken. Datacenter routers have different ASNs between layers such that external BGP (eBGP) is used.

We will now show how BGP propagates best path information. Fig. 1 shows an example network with three routers and Fig. 2 shows fragments of two configuration files. RIBs are initialized with locally exported prefixes. For example, router R1 exports 10.0.0.0/24, and therefore this prefix is inserted in its RIB.

The second step of BGP is to continuously exchange information with neighbor routers about newly learnt prefixes and about prefixes that the router can

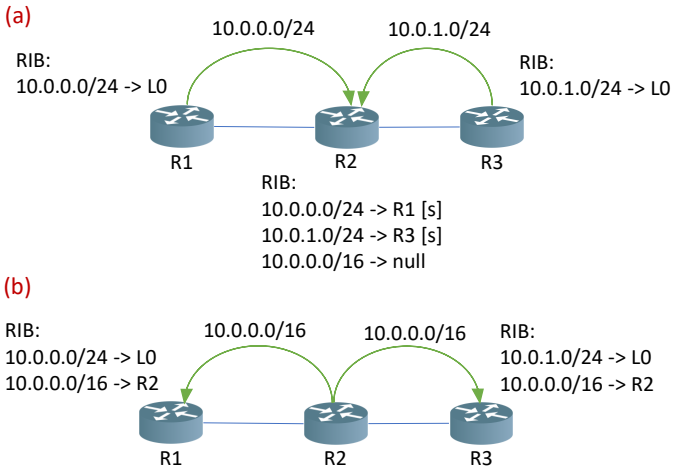

Fig. 3. Example of BGP running on the network of Fig. 1. [s] indicates a suppressed entry, which will not be advertised to the neighbors.

no longer reach. In our example, R1 advertises 10.0.0.0/24 to R2 and, similarly, R3 advertises 10.0.1.0/24 to R2, as can be seen in Fig. 3 (a). Since R2 does not block any advertisement from its neighbors, both of these prefixes are installed in the RIB of R2.

Router R2 has an "aggregate summary-only" command, which blocks any sub-prefix of 10.0.0.0/16 from being advertised to neighbors. Therefore, the prefixes received from R1 and R2 are marked with `[s]` in the RIB, meaning they are suppressed. Additionally, the aggregated prefix is installed in the RIB.

Router R2 then advertises the new entries in its RIB to its neighbors, as shown in Fig. 3 (b). The only new non-suppressed entry is 10.0.0.0/16 and it is sent to both neighbors, which install it in their RIBs.

As a final step, routers R1 and R3 try to advertise the new prefix to their neighbor (R2), but since this prefix was sent to them by R2 and BGP does not send a prefix back to the router that advertised it, routers R1 and R3 do not advertise anything further. Therefore, the RIBs in Fig. 3 (b) are the stable state of the network and no further communication occurs until some RIB changes.

Although we have presented the execution of BGP as a sequence of steps, the protocol does not run in a synchronous way: advertisements can be sent in any order.

3 Illustration

In this section we illustrate several key aspects of our algorithm. The first example introduces the algorithm through a simple step-by-step run and shows how different prefixes interact with each other. The second one focuses on how the order of propagation of route advertisements through the network is determined by our algorithm, and highlights how this order is fundamentally different from

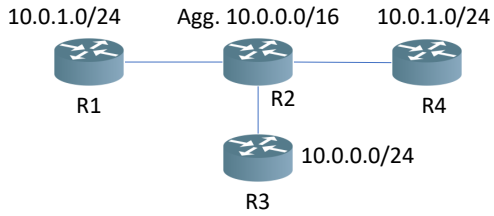

Fig. 4. Example network with four routers and prefixes they export. R2 aggregates 10.0.0.0/16.

the propagation happening during an actual, distributed execution of BGP. The last example shows that preference decrease across route advertisement is a necessary condition, as otherwise our algorithm fails to compute a correct answer.

For each example we assume that a router named Ri is configured to have the AS number i and they run eBGP.

3.1 Prefix Interaction

First we show how our algorithm computes routing tables for the example network of Fig. 4.

Each router R1, R3, and R4 exports a single prefix. Router R2 aggregates sub-prefixes of 10.0.0.0/16. This prefix is initially not exported by R2 because R2 has no sub-prefix in its RIB to trigger the aggregation.

In the first step of the algorithm, we collect all seed advertisements, i.e., all advertisements that routers in the network export on their own. In this example, we have three such advertisements that we will represent as tuples $(\mathit{router}, \mathit{prefix}, \mathit{AS\ path})$. Note that in practice BGP advertisements have many more attributes, but for the sake of simplicity we omit them. We use $\langle\rangle$ to represent the empty AS path. The seed advertisements are $a_1 = (\text{R1}, 10.0.1.0/24, \langle\rangle)$, $a_3 = (\text{R3}, 10.0.0.0/24, \langle\rangle)$, and $a_4 = (\text{R4}, 10.0.1.0/24, \langle\rangle)$. These advertisements are then grouped by prefix as follows.

$$((10.0.0.0/24, \{a_3\}),\ (10.0.1.0/24, \{a_1, a_4\}))$$

Our algorithm will now iterate over this list of seeds and consume its elements. Later we will see how additional items are placed on the list.

Routes are computed for each prefix individually, since routing policies may differ for different prefixes. We need to start with more specific prefixes and continue with less specific prefixes, for reasons that will be explained later. In our example, the list only has two prefixes and they have equal prefix length, which is 24, so they are incomparable and hence we can pick either of them arbitrarily. We chose to start with 10.0.0.0/24.

After we picked the prefix, we consider the corresponding set of seed advertisements, $\{a_3\}$. Now we propagate this set of advertisements to every router. That is, every router needs to learn a best path to reach 10.0.0.0/24 at R3.

We show our adaptation of Dijkstra's shortest path algorithm for this task. First we initialize a work list WL with the seed advertisements, i.e., $WL =$

$\{a_3\}$. Then the algorithm takes advertisements from the work list, one-by-one, processes them, and iterates until the work list becomes empty. So, we take the only present element in the work list, a_3, and re-advertise it to all neighbors of R3, which happens to be only router R2. To re-advertise, we create a new advertisement by copying a_3 and prepending R3's ASN to the AS path, and obtain $a_{2'} = (\text{R2}, 10.0.0.0/24, \langle 3 \rangle)$. This new advertisement is then added to the work list, hence we obtain $WL = \{a_{2'}\}$. After the first re-advertisement we obtain the following RIB entries.

R2	R3
$a_{2'} = (\text{R2}, 10.0.0.0/24, \langle 3 \rangle)$	$a_3 = (\text{R3}, 10.0.0.0/24, \langle\rangle)$

The algorithm then advertises $a_{2'}$ to R2's neighbors R1 and R4. Two new advertisements $a_{1'} = (\text{R1}, 10.0.0.0/24, \langle 2, 3 \rangle)$ and $a_{4'} = (\text{R4}, 10.0.0.0/24, \langle 2, 3 \rangle)$ are created. Note how R2's ASN is prepended to the AS path. The work list becomes $WL = \{a_{1'}, a_{4'}\}$.

We now reach a new case in our algorithm, in which the work list WL has more than one element. The choice of the next advertisement to process is important. Like in Dijkstra's algorithm we pick a vertex that is labeled with the smallest distance value: we need to visit a most preferred advertisement first. We assume that a partial order relation $\prec$ captures BGP's advertisement preference order. We can obtain $\prec$ from the description of the BGP's best path selection algorithm, which specifies that advertisements with shorter AS paths are preferable to advertisements with longer AS paths, among other criteria.

The order $\prec$ is partial, since BGP advertisements are not always comparable. One of the main reasons we particularly notice lack of totality is that the BGP best path selection algorithm was designed to be used within a single router, while our work list contains advertisements that reside at different routers. In our example, both advertisements in the work list have the same AS path length, so they are equally preferable. We will break the tie through an auxiliary lexicographic order on names of routers that store the advertisements, i.e., R1 and R4. As a result, our algorithm deterministically picks $a_{1'}$ from the work list.

Advertisement $a_{1'}$ can only be re-advertised back to R2 since R1 has no other neighbor. However, R2 rejects this advertisement because its own ASN occurs in the AS path $\langle 2, 3 \rangle$. A similar advertisement rejection happens with $a_{4'}$. Finally, the work list WL becomes empty and the advertisement propagation loop finishes. We computed four RIB entries, one for each of the routers in the network, since there are no policies in our example network that block advertisements of the considered prefix and all routers are reachable from R3.

Now we inspect if aggregation is configured on any of the routers. Router R2 has an aggregate for 10.0.0.0/16 which was not previously enabled since the RIB of R2 was empty. With the installation of $a_{2'}$ in R2's RIB, the aggregation becomes active because the prefix of $a_{2'}$ is a sub-prefix of the aggregate. Therefore, we generate a new advertisement $a_{2''} = (\text{R2}, 10.0.0.0/16, \langle\rangle)$, which tracks the enabled aggregate. The list with seed advertisements we had before is now extended to include the new advertisement and its prefix as follows.

$$((10.0.1.0/24, \{a_1, a_4\}),\ (10.0.0.0/16, \{a_{2''}\}))$$

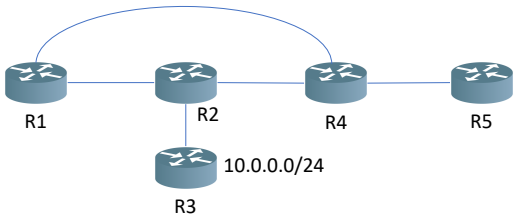

Fig. 5. Example network with five routers. Only R3 advertises a prefix.

We are now considering a case when it matters which prefix we take from the seed list to process next. Note that aggregate advertisements, like $a_{2''}$, are created and installed due to advertisements for sub-prefixes being installed in the RIB. The attributes of $a_{2''}$ are computed by applying appropriate aggregation functions on the attributes of sub-prefixes. Therefore, we advertise all sub-prefixes before advertising the aggregate, and hence avoid the problem of updating advertisement attributes and propagating the effect of such updates through additional route advertisements. This is why we iterate over the seed list by starting with more specific prefixes and proceeding with less specific prefixes.

The BGP RFC [41, § 9.2.2.2] mandates that the aggregated AS path should be the largest common prefix of the AS paths of advertisements of sub-prefixes. In our example, we set the AS path of the aggregate to $\langle\rangle$, which is not what the BGP specification mandates, but it is how it is implemented by some relevant vendors, e.g., Cisco.

Our algorithm proceeds with a new run of the modified Dijkstra's algorithm that advertises 10.0.1.0/24. The only difference to what we described previously is that now we have two seed advertisements, a_1 and a_4. These are inserted in the work list WL and the rest of the algorithm proceeds as before. Finally, all advertisements for 10.0.0.0/16 are computed and our algorithm terminates.

The forwarding tables (FIBs) of the routers can be computed from the RIBs by taking the best advertisements for each prefix. In our example, each router only has one advertisement for each prefix, so all advertisements are propagated to the FIB.

3.2 Globally vs. Locally Preferred Advertisements

In this example we show how the order of propagation of route advertisements used by our algorithm differs from what a (distributed) execution of BGP in a real network can choose. This difference is important in ensuring that any propagated route advertisement will never be superseded by a better one.

To illustrate the above point, we change our example network to include an additional link from R1 to R4 and an extra router R5, as shown in Fig. 5. We also add a route map to router R2 that applies to advertisements going out to R4. This route map augments the AS path by prepending the AS number 2 twice. The configuration change in router R2 to include this route map is as follows.

```
route-map prepend permit 10
  set as-path prepend 2 2
!
router bgp 2
  neighbor 10.1.0.4 route-map prepend out  ! R4
```

In this example, we only have one seed advertisement $a_3 = (\text{R3}, 10.0.0.0/24, \langle\rangle)$. This propagates to R2 as $a_2 = (\text{R2}, 10.0.0.0/24, \langle 3\rangle)$. Advertisement a_2 is then propagated to the neighbors of router R2, and so we obtain two new advertisements $a_1 = (\text{R1}, 10.0.0.0/24, \langle 2, 3\rangle)$ and $a_4 = (\text{R4}, 10.0.0.0/24, \langle 2, 2, 2, 3\rangle)$. The work list becomes $WL = \{a_1, a_4\}$, together with the RIB entries shown below. Note that for brevity we only show the AS path in each of the advertisements.

R1	R2	R3	R4	R5
$a_1 = \langle 2, 3\rangle$	$a_2 = \langle 3\rangle$	$a_3 = \langle\rangle$	$a_4 = \langle 2, 2, 2, 3\rangle$	

The next advertisement to explore is a_1, since it is more preferred than a_4, i.e., $a_1 \prec a_4$. Advertising a_1 to R1's neighbors results in a new advertisement $a_{4'} = (\text{R4}, 10.0.0.0/24, \langle 1, 2, 3\rangle)$, while R2 drops the advertisement from R1 due to the occurrence of its ASN in the AS path of a_1. We now have two competing advertisements at R4. One was received from R1, $a_{4'}$, and the other from R2, a_4. A router only advertises a most preferred advertisement, which in this case is $a_{4'}$ since $a_{4'} \prec a_4$ as the AS path $\langle 1, 2, 3\rangle$ for $a_{4'}$ is shorter than $\langle 2, 2, 2, 3\rangle$ for a_4. Therefore, we replace a_4 with $a_{4'}$ in the work list to get $WL = \{a_{4'}\}$. We point out that advertisement a_4 is nevertheless stored in the RIB of R4, but is not advertised further. Finally, the algorithm computes $a_5 = (\text{R5}, 10.0.0.0/24, \langle 4, 1, 2, 3\rangle)$ for R5. The final result is that each router has one entry in the RIB, except R4 which has two entries, where one is singled out as a best advertisement.

In this example we observed that since the work list stores advertisements across all routers, when we take the globally most preferred advertisement for exploration, the exploration of the most preferred advertisement within a given router may be delayed. Here we speak of a global preference order. It is essential for avoiding recomputation of advertisements due to arrival of more preferred ones, as it happens when BGP runs in a distributed setting over real networks in which a router propagates an advertisement that is most preferred among the locally present ones. In this case we speak of a local preference order.

In contrast, when running BGP on our example and following the local order on the RIBs containing advertisements $a_1, \ldots, a_5$, R4 may advertise a_4 before it receives $a_{4'}$, which leads to $a_{5'} = (\text{R5}, 10.0.0.0/24, \langle 4, 2, 2, 2, 3\rangle)$. After the advertisement of a_4, R1 may advertise a_1 to R4 which results in $a_{4'}$ appearing on R4. At this point R4 discovers that a_4 is no longer the most preferred advertisement, while $a_{4'}$ is. So it needs to ask R5 to withdraw advertisement $a_{5'}$. In a larger network, by transitivity all advertisements that were sent out because of $a_{4'}$ would need to be withdrawn, which could be a significant effort.

By following the global order, instead of the local ones, our algorithm never withdraws advertisements, which helps in scaling to large datacenter networks.

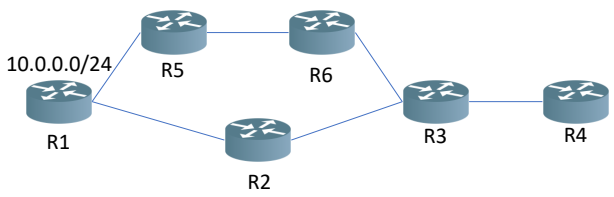

Fig. 6. Example network, where only router R1 advertises a prefix. R3 has a route map that increases local preference on incoming advertisements from R6.

3.3 Necessity of Monotonic Increase of Preference

We showed how our algorithm avoids recomputation of advertisements by propagating only globally optimal advertisements. However, this procedure is only correct if the routing policy produces advertisements that never increase in preference. This means that the preference of a route advertisement at the destination router cannot be higher than the preference of the originating advertisement at the source router. However, some features supported by BGP routing policies can lead to violation of this property.

The following example illustrates the necessity of the monotonic increase property, and shows that without it our algorithm computes an incorrect result.

We consider the network in Fig. 6. Router R3 has a route map that increases the local preference of advertisements incoming from R6 to 200, while the default value is usually 100. The route map is as follows.

```
route-map in_r6 permit 10
  set local-preference 200
!
router bgp 3
  neighbor 10.1.0.3 route-map in_r6 in
```

Note that the BGP best path selection algorithm states that the advertisement with the highest local preference is preferred. If advertisements have an equal value of the local preference attribute, then the advertisement with the shortest AS path is preferred.

Since we now need to track the local preference attribute, we will represent advertisement as tuples $(router, prefix, local\ pref, AS\ path)$.

The seed advertisement is $a_1 = (\text{R1}, 10.0.0.0/24, 100, \langle\rangle)$. Our algorithm propagates a_1 from R1 to R2 and R5 resulting in $a_2 = (\text{R2}, 10.0.0.0/24, 100, \langle 1\rangle)$ and $a_5 = (\text{R5}, 10.0.0.0/24, 100, \langle 1\rangle)$. The resulting work list is $WL = \{a_2, a_5\}$.

As a_2 and a_5 are equally preferred, our algorithm picks the advertisement located at the router with the lowest identifier (in order to stay deterministic), which is a_2 in this case. We propagate a_2 to R3 and obtain $a_3 = (\text{R3}, 10.0.0.0/24, 100, \langle 2, 1\rangle)$ and $WL = \{a_3, a_5\}$. We then take a_5 from the work list and compute $a_6 = (\text{R6}, 10.0.0.0/24, 100, \langle 5, 1\rangle)$ and $WL = \{a_3, a_6\}$. Afterward we take a_3 and compute $a_4 = (\text{R4}, 10.0.0.0/24, 100, \langle 3, 2, 1\rangle)$ and $WL = \{a_4, a_6\}$. The resulting RIBs (with just the local preference and AS path attributes) are shown below.

R1: $a_1 = (100, \langle\rangle)$	R2: $a_2 = (100, \langle 1\rangle)$	R3: $a_3 = (100, \langle 2, 1\rangle)$
R4: $a_4 = (100, \langle 3, 2, 1\rangle)$	R5: $a_5 = (100, \langle 1\rangle)$	R6: $a_6 = (100, \langle 5, 1\rangle)$

We now arrive at the problematic part. When we take a_6 from the work list and advertise it to R3, the incoming route map at R3 sets the local preference of the incoming advertisement to 200. Therefore we obtain $a_{3'} = (\text{R3}, 10.0.0.0/24, 200, \langle 6, 5, 1\rangle)$. This advertisement is more preferred than a_3 that was received previously from R2, i.e., $a_{3'} \prec a_3$ since $200 > 100$. This means that R3 now has a more preferred advertisement than the one previously present in its RIB and therefore the new advertisement needs to be propagated, with all the related withdrawals and re-advertisements, while a_3 is still kept in the RIB.

Unfortunately, we already propagated a_3 to R4 by following the global preference order. To fix the problem, we would need to remove a_4 from R4's RIB, as well as remove any advertisements transitively derived from a_4, potentially spanning the whole network. However, due to the monotonic increase assumption, our algorithm does not anticipate such an issue and hence is not able to delete RIB entries. As a consequence, we obtain a wrong result for this network.

To summarize, our algorithm only produces a correct result when the network's routing policies ensure monotonic increase of preference. Fortunately, several studies (including ours) confirm that industrial datacenter networks have this property.

4 Algorithms

In this section we describe an algorithm for efficient simulation of BGP in datacenter networks. Our algorithm is based on Dijkstra's shortest path algorithm, and adapts it to our setting by using BGP route advertisements to track distance, comparing distances using BGP path selection function, and updating distance using BGP route maps. We also show how to deal with equal-cost multi-path routing (ECMP) and aggregation.

4.1 Generalizing Dijkstra's Algorithm

We begin by revisiting Dijkstra's algorithm, in order to fix a particular version as there are different ways of setting up and maintaining the distance and work list data structures. See Fig. 7.

Dijkstra's algorithm works as follows. It initializes the distance from the source vertex to itself as zero and adds the vertex to the queue (lines 1–2). Then it iterates over the queue until it is empty. At each iteration of the loop it picks the vertex u from the queue with the smallest distance from the source vertex (lines 3–5). The algorithm then iterates over each neighbor v of vertex u and updates the best known distance so far to v if it is the first path we discover to v or if the previously known path was longer (lines 6–10). When the queue becomes empty, the function returns function *dist* which contains the shortest distance from vertex v_0 to all the other reachable vertexes in the graph (line 12).

```
    function DIJKSTRA
    input
      E : V × V – set of edges
      v0 : V – initial vertex
      length : V × V → ℕ – edge length
    vars
      dist : V → ℕ – distance from source to other vertexes
      queue : 𝒫(V) – queue with vertexes pending processing
    begin
1     dist(v0) := 0
2     queue := {v0}
3     while queue ≠ ∅ do
4       u = arg min^<_{w∈queue} dist(w)
5       queue := queue \ {u}
6       for each v ∈ E(u) do
7         if v ∉ dom(dist) ∨ dist(u) + length(u, v) < dist(v) then
8           dist(v) := dist(u) + length(u, v)
9           queue := queue ∪ {v}
10      done
11    done
12    return dist
    end
```

Fig. 7. DIJKSTRA computes the shortest path between a source vertex in the graph and all other vertexes. $E(u)$ is the set of neighbors of u. $\arg\min^{<}$ chooses a minimum with respect to the relation $<$.

We gave a brief description of how Dijkstra's algorithm works. It is important to note that the result is a labeling of vertexes with a natural number: the shortest distance from the source to that vertex. We will now consider a few operations in Dijkstra's algorithm in a more general setting. In particular, we will consider the labels of vertexes to be of an arbitrary type D, with the ordering $\prec$ between these labels. We assume that a function *trans* can be used to compute labeling of a neighboring vertex. The generalized version of Dijkstra's algorithm is shown in Fig. 8.

The deviations from Dijkstra's algorithm are as follows. For initialization (lines 1–2), we now take the initial label of the source vertex d_0 as input, instead of setting it to zero. Secondly, the order of extraction of vertexes from the queue is given by a label order $\prec$ given as input (line 4). Finally, the new label computed for a neighbor is computed by the *trans* function given as input instead of computing a path length explicitly (line 7–8). Old and new labels are compared with $\prec$ as well.

We relate Dijkstra's algorithm with the generalized version as follows.

$$\textsc{Dijkstra}(E, v_0, \mathit{length}) = \textsc{GDijkstra}\langle\mathbb{N}\rangle(E, v_0, 0, \lambda d\, u\, v.d + \mathit{length}(u, v), <)$$

Here we set the initial label of the source vertex to zero. The label of a neighbor is the label of the current vertex u, i.e., the distance between source and u, plus the length of the path from u to v.

```
    function GDijkstra⟨D⟩
    input
      E : V × V – set of edges
      v0 : V – initial vertex
      d0 : D – initial label
      trans : D × V × V → D – transform label along an edge
      ≺ : P(D × D) – label ordering
    vars
      dist : V → D – distance from v0 to other vertexes
      queue : P(V) – queue with vertexes pending processing
    begin
1     dist(v0) := d0
2     queue := {v0}
3     while queue ≠ ∅ do
4       u = arg min^≺_{w∈queue} dist(w)
5       queue = queue \ {u}
6       for each v ∈ E(u) do
7         if v ∉ dom(dist) ∨ trans(dist(u), u, v) ≺ dist(v) then
8           dist(v) := trans(dist(u), u, v)
9           queue := queue ∪ {v}
10      done
11    done
12    return dist
    end
```

Fig. 8. GDijkstra computes min. labels that reach each vertex from v_0 labeled by d_0.

We now state the correctness of the generalized Dijkstra's algorithm.

Theorem 1. *If $\prec$ is a strict partial order and function trans is monotonically increasing, i.e.,*

$$\forall d\ \forall (u,v) \in E : \neg(trans(d,u,v) \prec d)\ ,$$

then GDijkstra *labels each vertex with a minimal label that can be computed by traversing the set of edges E starting at vertex v_0 with label d_0 and using function trans to label edges.*

4.2 Advertising a Single Prefix

We now show how to simulate BGP for the advertisement of a single prefix using our generalized version of Dijkstra's algorithm.

Vertexes correspond to routers and edges the peering relations established between them. The label type D will be route advertisements. The source vertex will be the router that exports the prefix. The source label will be an initial advertisement as mandated by the BGP standard, e.g., with empty AS path, with the origin type indicating how this advertisement was produced, etc.

The order between advertisements is given by $\prec_{BGP}$. For example, $a \prec_{BGP} a'$ holds if the local preference of a is greater than that of a'. If $a \prec_{BGP} a'$ holds, we say that a is preferred to a'. Order $\prec_{BGP}$ corresponds to the best past selection algorithm of BGP, which is a lexicographic order on advertisement attributes.

The transform function *trans* has to do several things. Firstly, it needs to check if the advertisement can be propagated any further. One example of an advertisement that is blocked is when there is a summary-only aggregate whose prefix intersects with the prefix being advertised. This type of aggregates blocks contributing advertisements (i.e., advertisements of more specific prefixes) from being propagated to neighbors. Secondly, this function needs to transform the advertisement for the given neighbor, e.g., prepend its own ASN to the AS path, and then apply the outgoing route map of the sender and the incoming route map of the neighbor (if any). Any of these route maps may rewrite some fields of the advertisement or even block it from being advertised or added to the RIB, respectively for outgoing and incoming route maps. We need to compute a new advertisement for each neighbor because routers can have different policies for different neighbors and incoming route maps may also differ between neighbors.

A simplified version of the transform function can be represented by the following pseudo code. We refer to [47] for an example of a formal discussion. In the pseudo code we use ∞ to denote a least preferred advertisement with respect to $\prec_{BGP}$. We use ∞ to model the case when a route map rejects an advertisement. Such advertisements can be ignored upon the termination of the algorithm, when installing advertisements into the RIBs of their respective routers.

$trans_{BGP}(a, u, v) :=$
if u should not advertise a **then**
return ∞
$a' :=$ create advertisement for v from a
$a'' :=$ OutRouteMap(u, a')
if v should not accept a'' **then**
return ∞
return InRouteMap(v, a'')

In practice, function $trans_{BGP}$ can be quite complicated and needs to faithfully implement vendor-specific details. For example, there are more cases that block advertisements from being propagated besides summary-only aggregates, such as when an advertisement is tagged with the "no export" or "no advertise" communities, and when an advertisement is received from an iBGP peer it cannot be advertised to other iBGP peers. Also, some vendors do not support the advertisement of IPv4 prefixes to neighbor routers that are connected over IPv6.

A reason to reject an incoming advertisement is, e.g., if the AS path contains the ASN of the receiving router. This check can only be performed after the outgoing transformations, since outgoing route maps are allowed to change the AS path.

Putting everything together, we define a function BGP_{One} that computes a RIB for a given prefix. Here v_0 is the router that exports the initial advertisement d_0 for the prefix, and E is defined by the BGP peering between routers.

$$\text{BGP}_{\text{One}}(E, v_0, d_0) := \text{GDijkstra}(E, v_0, d_0, trans_{BGP}, \prec_{BGP})$$

To be able to use GDijkstra and obtain a correct result, we need to establish the two assumptions made by the algorithm: (1) $\prec_{BGP}$ is a strict partial order, and (2) $trans_{BGP}$ is monotonically increasing. Assumption (1) holds because $\prec_{BGP}$ is a lexicographic order on advertisement attributes.

In general, $trans_{BGP}$ is not monotonically increasing. For example, route maps may increase local preference, which ranks higher in the best path selection than the AS path length which usually increases by one when an advertisement is propagated to a neighbor. In this work, since we target datacenter networks, we deal with $trans_{BGP}$ that is monotonically increasing.

4.3 Computing All Advertisements for a Single Prefix

In the previous section, we showed how to use our generalized version of Dijkstra's algorithm to compute advertisements that are propagated to every router from a given prefix. This is very close to what BGP actually computes, but not exactly. BGP records at each router not only the most preferred advertisement it has received for a given prefix, but also all the received advertisements. This way the router can, e.g., promote the second best advertisement to become the most preferred one if the neighbor that sent the original most preferred advertisement becomes unreachable. As we have done, a router only propagates most preferred advertisements to its neighbors.

We need a further extension in Dijkstra's algorithm to keep track of all advertisements, including non-best ones. The new (and final) generalization is shown in Fig. 9. This algorithm tracks distance from the source as a set of labels instead of a single label. It stores at vertex v all labels computed by traversing paths from the neighbors of v to v, instead of keeping only the smallest label. The creation of a new label for a neighbor of vertex v continues to depend only on a minimal label of v as previously (c.f. arguments to *trans* function).

We note that $\min^{\prec} dist(v)$ in this algorithm in line 7 is exactly the same value as $dist(v)$ in the previous algorithm GDijkstra in line 7. Therefore, the only change in behavior of GDijkstraSet is in line 9. Previously we only stored the minimal label found so far, so the assignment of $dist(v)$ was inside the if statement. Now, we moved the assignment out of the if statement such that the assignment is executed regardless whether the new label is $\prec$-better than the previous one.

We now define BGP tracking all advertisements in terms of the set-tracking generalization of Dijkstra's algorithm.

$$\mathrm{BGP}_{\mathrm{All}}(E, v_0, d_0) := \mathrm{GDijkstraSet}(E, v_0, d_0, trans_{BGP}, \prec_{BGP})$$

The function $\mathrm{BGP}_{\mathrm{All}}$ correctly computes propagation of a single prefix in an efficient way. The network must, however, respect the monotonic increase property we mentioned previously.

We now state the correctness of $\mathrm{BGP}_{\mathrm{All}}$.

Theorem 2. *Given a monotonically increasing BGP network,* $\mathrm{BGP}_{\mathrm{All}}$ *computes a stable state of RIBs in the network.*

```
    function GDijkstraSet⟨D⟩
    input
      E : V × V – set of edges
      v0 : V – initial vertex
      d0 : D – initial label
      trans : D × V × V → D – transform label along an edge
      ≺ : P(D × D) – label ordering
    vars
      dist : V → P(D) – distance from v0 to other vertexes
      queue : P(V) – queue with vertexes pending processing
    begin
1     dist(v0) := {d0}
2     queue := {v0}
3     while queue ≠ ∅ do
4       u = arg min≺_{w∈queue}(min≺ dist(w))
5       queue := queue \ {u}
6       for each v ∈ E(u) do
7         if v ∉ dom(dist) ∨ trans(min≺ dist(u), u, v) ≺ (min≺ dist(v)) then
8           queue := queue ∪ {v}
9         dist(v) := {trans(min≺ dist(u), u, v)} ∪ (dist(v) if v ∈ dom(dist) else ∅)
10      done
11    done
12    return dist
    end
```

Fig. 9. GDijkstraSet computes a set of minimal labels at each vertex, as well as keeps track of all labels that are propagated to a vertex, a so called one-hop history.

In this section we assumed that there is only one source router for each prefix. This is not true in general, however. For example, we may want to load balance traffic for a service between different racks in a datacenter, and so the routers of all such racks have to advertise the same prefix corresponding to the service.

Extending the given algorithm for multiple sources is straightforward. Instead of taking a single source vertex and advertisement, the algorithm can take a set instead. Then the queue is populated with all the advertisements and these will be explored in order.

4.4 Computing RIBs for All Prefixes

In the previous section we presented an algorithm to compute BGP advertisements for a single prefix. We now show an algorithm that computes BGP advertisements for all prefixes originating in a monotonic network, and produces the RIBs for all the routers. The algorithm consists of a loop invoking the single prefix-propagating algorithm for each prefix and a prefix composition step.

We compute a separate control plane for each prefix since prefixes are exported at varying locations. Moreover, different routers in a network are often configured to accept and/or modify advertisements differently depending on the prefix. Therefore we cannot simply run the set-generalized Dijkstra algorithm for all prefixes at once.

The algorithm to compute the RIB for all routers is as follows.

RIB := ∅
seeds := InitSeeds()

while *seeds* ≠ ∅ **do**
  (*prefix*, *adverts*) := take most specific prefix from *seeds*
  RIB := RIB ∪ {(*r*, *prefix*) ↦ *a* | (*r*, *a*) ∈ $\mathrm{BGP}_{\mathrm{ALL}}(E, adverts)$}
  seeds := UpdateSeeds(RIB, *prefix*, *seeds*)
done
return RIB

The procedure starts by computing the set of seed advertisements. grouped by prefix. Seed adverts consist of the prefixes advertised by each router through, e.g., the `network` command, or via aggregation of locally installed routes.

The order of iteration through prefixes is relevant for features where there is a dependency between different prefixes, i.e., features that make advertisement of prefixes to not be independent of each other. For example, an aggregated prefix, say 10.0.0.0/8, depends on contributing prefixes, say 10.0.1.0/24. In this case we need to iterate through more specific prefixes before the less specific ones, e.g., we need to execute $\mathrm{BGP}_{\mathrm{ALL}}$ on 10.0.1.0/24 before executing it on 10.0.0.0/8.

Function UpdateSeeds creates and updates existing seed advertisements. These new and/or updated seeds need to be iterated over later. It is guaranteed, however, that any new seed is of a less specific prefix than any other already processed. Since we iterate from more specific to less specific prefixes, we never miss any update to a seed or explore the same prefix more than once.

4.5 Updating Seed Advertisements

Sometimes there are dependencies between different IP prefixes, and installing an entry in the RIB may automatically trigger the installation (or update) of an entry for another prefix.

One such case is aggregation. For example, if a router is configured to aggregate 10.0.0.0/16 but has no initial seed with a sub-prefix, initially 10.0.0.0/16 will not be installed in the RIB since there is no contributing advertisement. If later this router receives an advertisement for, e.g., 10.0.0.1/32, the aggregated prefix becomes active and thus it becomes a seed since it needs to be advertised to the neighbors.

Another case is when an aggregated prefix is already active and the router installs another sub-prefix. In this case, we may need to update the seed advertisement for the aggregated prefix since it depends on all contributing advertisements. For example, the origin type of an aggregated advertisement is the result of combining the origin type of all contributing advertisements. Other attributes of advertisements are often combined using vendor-specific functions.

Function UpdateSeeds takes the last prefix that was advertised as input and checks if that prefix is a potential contributor to any aggregated prefix in

the routers. If so, it creates a new seed advertisement for the aggregated prefix in case it does not exist yet, or updates the existing seed.

It is guaranteed that any created or updated seed advertisement has not been visited yet by the BGP_{ALL} algorithm. This is because the main loop traverses prefixes from more specific to less specific, and the created/updated seeds have a less specific address than in the current loop iteration, otherwise the advertisements created in the current iteration could not possibly be contributors to the created/updated seeds of aggregated prefixes.

5 Evaluation

To evaluate the proposed algorithm, we implemented a prototype called FastPlane in C++17. It supports several router vendors, including Arista, Cisco (IOS and Nexus), Force10, and Juniper. The range of implemented features includes BGP (internal and external), communities, BGP multipath, route maps, prefix aggregation, ACLs, ECMP, static routes, IPv4, and IPv6.

We compare the running time of FastPlane with Batfish [20], which is the state-of-the-art tool for RIB computation supporting general networks (as opposed to FastPlane, which only supports monotonic ones). As far as we are aware, Batfish is the only publicly available tool that can parse significant portions of industrial router configurations and that scales to thousands of routers.

Setup We took the configuration files for all datacenters (DCs) of a major public cloud provider. Overall we collected a few (single digit) GBs of configuration files containing hundreds of millions of lines.

The network architecture of these DCs is a fat-tree running eBGP between all routers [23]. The dataset contains DCs with several variants of the architecture, depending on the DC size and age (since the architecture keeps evolving). We validated that the monotonicity property holds for all DCs in our dataset.

The machine used to run the experiments had 2x Intel Xeon E5-2660 CPUs (16 cores total), with 112 GBs of RAM. We used Batfish revision `b004dff` from 11/Jan/2018, with a limit of 100 GBs of memory for the JVM.

Performance Results For each datacenter, we computed RIBs and FIBs for all routers using FastPlane and Batfish, and measured the table size and the running time. The total number of entries in the RIBs of all devices of a single datacenter varied between several thousands and hundreds of millions.

We present the CPU time taken to compute the RIBs and FIBs in Fig. 10. Datacenters are grouped into five buckets, according to their number of routers. For each bucket we show the average time for the datacenters in that bucket.

Fig. 10 shows that FastPlane is about two orders of magnitude faster than Batfish. Given 100 GBs of memory, Batfish does not scale beyond 2,000 routers. Moreover, Batfish only supports IPv4, while FastPlane supports IPv6 as well.

FastPlane only executes one round of BGP propagation, since it stratifies the computation. This is possible for monotonic networks. Batfish, on the other

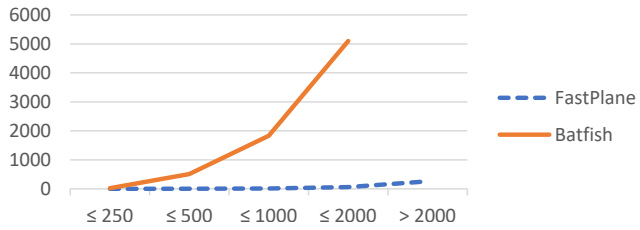

Fig. 10. CPU time (in seconds) to compute the RIBs and FIBs of all routers in each datacenter vs. datacenter size (number of routers).

hand, does not pick any particular propagation order, which leads to several iterations. In our dataset, we see Batfish requiring up to eight BGP iterations. This shows that choosing the right propagation order has significant impact on the efficiency of the algorithm.

Besides the higher number of BGP rounds, Batfish is fundamentally slower than FastPlane for two other reasons: 1) Batfish supports generic BGP networks while FastPlane only supports monotonic networks, and 2) Batfish's fixed-point check resembles the so-called naive Datalog evaluation (as opposed to the more efficient semi-naive [13]).

Batfish does not simulate BGP through message passing like C-BGP. Instead, a router's RIB is computed by peeking into the neighbor's RIBs and importing those (subject to routing policies, and so on). Batfish keeps two sets of RIBs per router: one from the previous iteration and another for the current iteration, which is computed based on the neighboring RIBs from the previous iteration. Therefore, Batfish keeps two RIBs per router in memory at a time.

Validation of BGP Semantics To increase confidence in our implementation of BGP, in particular in vendor-specific features, we compared the RIBs and FIBs computed by FastPlane with the ones from production routers in the datacenters.

Since datacenters operate in an open environment and receive external advertisements, we had to define a boundary delimiting what we would simulate. Routers outside of the given datacenter, i.e., the Internet, other datacenters, and load balancers, were modeled as dummy BGP neighbors that replayed the advertisements received by the production routers at the boundary.

This validation was effective. We found several bugs in our semantics of BGP, differences between the BGP implementations of different vendors, as well as bugs in the network. After validation, FastPlane computes FIBs and RIBs that are equivalent to those of several thousand routers we compared against.

One interesting bug we found was a difference in the behavior of BGP aggregation between Cisco and Arista: Arista follows the RFC and sets the AS path to the longest common prefix of the contributing advertisements' AS paths, while Cisco always creates aggregates with empty AS paths.

Production Bugs Found We give a high-level description of some of the bugs found in datacenter networks while doing the cross-checking explained in the previous section.

One of the bugs was in the redistribution policy of connected routes. A network operator specified how the RIB of each device type is expected to look like. For example, ToRs must have all prefixes exported by load balancers. We then checked if the computed RIBs matched the expectations, and the check failed. In particular, there were unexpected advertisements. The routing policy was fixed to block them, and FastPlane was used to validate the fix before deployment.

We also found a bug in a router's firmware that resulted in the FIB's next-hops to be incorrect for some prefixes, due to a race condition in the code that updates the FIB. This bug would have been hard to find without FastPlane, which provides the ground truth for the router behavior.

Another type of bug was a problem with the network architecture. The architecture allows the network control plane to converge to different stable states, due to non-determinism. We found that some of these states have reduced load balancing and fault tolerance. We confirmed that the problem manifests in production and a fix is underway.

6 Related Work

Control plane verification is closely related to our work. Existing tools use a variety of techniques to compute the control plane, including simulation of message passing of routing protocols, Batfish [20] and C-BGP [40], and SMT encodings of BGP, Bagpipe [47], MineSweeper [8]. ERA [17] uses BDDs for reachability analysis between endpoints. ARC [22] and [48] compute an abstraction of the control plane. These tools are less scalable than FastPlane when applied to obtain the entire control plane, but they often support more BGP features and more complex interactions between routing protocols. We believe that our algorithm could be used to scale existing tools to large datacenters, while keeping the applicability of general methods when needed.

CrystalNet [34] uses the router's firmware in a virtualized environment to compute the control plane. It is bug-compatible with production networks, but it is significantly more resource intensive and slower than FastPlane.

There also exists static analysis of configuration files, similarly to compiler warnings. Such tools, e.g., rcp [18], do not compute the control plane.

Another area of network verification is data plane verification [52]. These tools operate over given FIBs, which can be either be computed from RIBs, or obtained directly from production routers, which unfortunately precludes verification before deployment. Tools for data plane verification employ a range of techniques including specialized algorithms and data structures, e.g., HSA [29], NetPlumber [28], VeriFlow [30], ddNF [12], TenantGuard [46], Datalog solvers, e.g., NoD [36], predicate abstraction, e.g., AP [49], SAT solvers, e.g., Anteater [37] and NetSAT [51], BDDs, e.g., FlowChecker [3], symmetry reduction [39], localized, per router, properties, e.g., SecGuru [11], and symbolic execution [14].

Software defined networks (SDNs) offer an alternative to BGP or OSPF, however they are not yet deployed at datacenter scale. There exist model checkering tools for SDN controllers, e.g., Kuai [38], VeriCon [7], and SDNRacer [16].

Correct by construction is an alternative approach to network reliability. Tools for configuration synthesis include Propane [9,10], and Genesis [45]. There is also work on synthesizing ACLs [27, 50]. We anticipate that synthesis tools could improve scalability by applying our monotonicity observation.

There are new languages to declaratively specify routing behavior, e.g., NetKAT [5], and firewalls, e.g., Mignis [1].

There is related work in the area of routing algebras [4, 24, 25, 44]. For example, [26] proves that monotonicity of the edge labeling function, which corresponds to our $trans_{BGP}$, with respect to the label order, which is our $\prec_{BGP}$, ensures convergence of the routing protocol. [43] gives a generalization of Dijkstra's algorithm, but using numerical weights, while the generalization in [33] is for arbitrary, totally ordered, cost functions. [15] also gives a generalization of Dijkstra's algorithm, but does not handle aggregation, unlike our algorithm.

[19] gives an algorithm to compute the control plane of an iBGP mesh with several routers peering with other ASs. [21] gives guidelines for configuring routers that peer with other organizations to ensure convergence of BGP.

7 Future Work

In this paper we presented an algorithm for computing routing tables that is applicable only when a certain subset of features of BGP is used. Further research is needed to broaden and precisely characterize what is the set (or sets) of features that can be used together and is still compatible with the proposed algorithm (or similar monotonic reasoning approach).

Dually, further research is needed to characterize protocol features to avoid in order to support efficient verification. Moreover, there is little understanding of if/how to replace non-monotonic features by monotonic ones. This could not only improve efficiency of network verification, but also speed up convergence time in production networks, since fewer advertisements would be withdrawn.

Another avenue is a study of non-determinism in control planes. $\prec$ is sometimes not a total order, which means there may exist different stable states in the network. This has disadvantages, such as making troubleshooting more difficult. Our current prototype deliberately computes a single stable state in a consistent, deterministic way so that the results are reproducible. However, this stable state may not be identical to the state in which the real network stabilizes.

8 Conclusion

We studied datacenter networks of a major cloud provider and confirmed their monotonicity. We then presented an efficient algorithm that leverages this fact to compute routing tables of that kind of networks. The evaluation shows that our prototype, FastPlane, scales to large production datacenters.

References

1. P. Adão, C. Bozzato, G. D. Rossi, R. Focardi, and F. L. Luccio. Mignis: A semantic based tool for firewall configuration. In *CSF*, 2014.
2. M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *SIGCOMM*, 2008.
3. E. Al-Shaer and S. Al-Haj. FlowChecker: configuration analysis and verification of federated openflow infrastructures. In *SafeConfig*, 2010.
4. M. A. Alim and T. G. Griffin. On the interaction of multiple routing algorithms. In *CoNEXT*, 2011.
5. C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker. NetKAT: semantic foundations for networks. In *POPL*, 2014.
6. A. Andreyev. Introducing data center fabric, the next-generation Facebook data center network, 2014.
7. T. Ball, N. Bjørner, A. Gember, S. Itzhaky, A. Karbyshev, M. Sagiv, M. Schapira, and A. Valadarsky. Vericon: Towards verifying controller programs in software-defined networks. In *PLDI*, 2014.
8. R. Beckett, A. Gupta, R. Mahajan, and D. Walker. A general approach to network configuration verification. In *SIGCOMM*, 2017.
9. R. Beckett, R. Mahajan, T. Millstein, J. Padhye, and D. Walker. Don't mind the gap: Bridging network-wide objectives and device-level configurations. In *SIGCOMM*, 2016.
10. R. Beckett, R. Mahajan, T. Millstein, J. Padhye, and D. Walker. Network configuration synthesis with abstract topologies. In *PLDI*, 2017.
11. N. Bjørner and K. Jayaraman. Checking cloud contracts in Microsoft Azure. In *ICDCIT*, 2015.
12. N. Bjørner, G. Juniwal, R. Mahajan, S. A. Seshia, and G. Varghese. ddNF: An efficient data structure for header spaces. In *HVC*, 2016.
13. S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about datalog (and never dared to ask). *IEEE Trans. on Knowl. and Data Eng.*, 1(1):146–166, Mar. 1989.
14. M. Dobrescu and K. Argyraki. Software dataplane verification. In *NSDI*, 2014.
15. S. Dynerowicz and T. G. Griffin. On the forwarding paths produced by internet routing algorithms. In *ICNP*, 2013.
16. A. El-Hassany, J. Miserez, P. Bielik, L. Vanbever, and M. Vechev. SDNRacer: Concurrency analysis for software-defined networks. In *PLDI*, 2016.
17. S. K. Fayaz, T. Sharma, A. Fogel, R. Mahajan, T. Millstein, V. Sekar, and G. Varghese. Efficient network reachability analysis using a succinct control plane representation. In *OSDI*, 2016.
18. N. Feamster and H. Balakrishnan. Detecting BGP configuration faults with static analysis. In *NSDI*, 2005.
19. N. Feamster and J. Rexford. Network-wide prediction of BGP routes. *IEEE/ACM Trans. Netw.*, 15(2):253–266, Apr. 2007.
20. A. Fogel, S. Fung, L. Pedrosa, M. Walraed-Sullivan, R. Govindan, R. Mahajan, and T. Millstein. A general approach to network configuration analysis. In *NSDI*, 2015.
21. L. Gao and J. Rexford. Stable internet routing without global coordination. In *SIGMETRICS*, 2000.
22. A. Gember-Jacobson, R. Viswanathan, A. Akella, and R. Mahajan. Fast control plane analysis using an abstract representation. In *SIGCOMM*, 2016.

23. A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. Vl2: A scalable and flexible data center network. In *SIGCOMM*, 2009.
24. T. G. Griffin, F. B. Shepherd, and G. Wilfong. The stable paths problem and interdomain routing. *IEEE/ACM Trans. Netw.*, 10(2):232–243, Apr. 2002.
25. T. G. Griffin, F. B. Shepherd, and G. T. Wilfong. Policy disputes in path-vector protocols. In *ICNP*, 1999.
26. T. G. Griffin and J. L. Sobrinho. Metarouting. In *SIGCOMM*, 2005.
27. W. T. Hallahan, E. Zhai, and R. Piskac. Automated repair by example for firewalls. In *FMCAD*, 2017.
28. P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real time network policy checking using header space analysis. In *NSDI*, 2013.
29. P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: Static checking for networks. In *NSDI*, 2012.
30. A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. Veriflow: Verifying network-wide invariants in real time. In *NSDI*, 2013.
31. P. Lahiri, G. Chen, P. Lapukhov, E. Nkposong, D. Maltz, R. Toomey, and L. Yuan. Routing design for large scale data centers: BGP is a better IGP. In *NANOG'55*, 2012.
32. P. Lapukhov, A. Premji, and J. Mitchell. RFC 7938: Use of BGP for Routing in Large-Scale Data Centers, 2016.
33. T. Lengauer and D. Theune. Efficient algorithms for path problems with general cost criteria. In *ICALP*, 1991.
34. H. Liu, Y. Zhu, J. Padhye, J. Cao, S. Tallapragada, N. P. Lopes, A. Rybalchenko, G. Lu, and L. Yuan. CrystalNet: Faithfully emulating large production networks. In *SOSP*, 2017.
35. Lloyd's. Failure of a top cloud service provider could cost US economy $15 billion, 2018.
36. N. P. Lopes, N. Bjørner, P. Godefroid, K. Jayaraman, and G. Varghese. Checking beliefs in dynamic networks. In *NSDI*, 2015.
37. H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King. Debugging the data plane with Anteater. In *SIGCOMM*, 2011.
38. R. Majumdar, S. D. Tetali, and Z. Wang. Kuai: A model checker for software-defined networks. In *FMCAD*, 2014.
39. G. D. Plotkin, N. Bjørner, N. P. Lopes, A. Rybalchenko, and G. Varghese. Scaling network verification using symmetry and surgery. In *POPL*, 2016.
40. B. Quoitin and S. Uhlig. Modeling the routing of an autonomous system with C-BGP. *IEEE Network*, 19(6):12–19, Nov 2005.
41. Y. Rekhter, T. Li, and S. Hares. RFC 4271: A Border Gateway Protocol 4 (BGP-4), 2006.
42. M. Rusinovich. TechEd 2013: Windows Azure Internals, 2013.
43. J. L. Sobrinho. Algebra and algorithms for QoS path computation and hop-by-hop routing in the internet. *IEEE/ACM Trans. Netw.*, 10(4):541–550, Aug. 2002.
44. J. L. Sobrinho. An algebraic theory of dynamic network routing. *IEEE/ACM Trans. Netw.*, pages 1160–1173, 2005.
45. K. Subramanian, L. D'Antoni, and A. Akella. Genesis: Synthesizing forwarding tables in multi-tenant networks. In *POPL*, 2017.
46. Y. Wang, T. Madi, S. Majumdar, Y. Jarraya, A. Alimohammadifar, M. Pourzandi, L. Wang, and M. Debbabi. TenantGuard: Scalable runtime verification of cloud-wide vm-level network isolation. In *NDSS*, 2017.

47. K. Weitz, D. Woos, E. Torlak, M. D. Ernst, A. Krishnamurthy, and Z. Tatlock. Scalable verification of border gateway protocol configurations with an SMT solver. In *OOPSLA*, 2016.
48. 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 *INFOCOM*, 2005.
49. H. Yang and S. S. Lam. Real-time verification of network properties using atomic predicates. In *ICNP*, 2013.
50. S. Zhang, A. Mahmoud, S. Malik, and S. Narain. Verification and synthesis of firewalls using SAT and QBF. In *ICNP*, 2012.
51. S. Zhang and S. Malik. SAT based verification of network data planes. In *ATVA*, 2013.
52. S. Zhang, S. Malik, and R. McGeer. Verification of computer switching networks: An overview. In *ATVA*, 2012.