Libra: Divide and Conquer to Verify Forwarding Tables in Huge Networks

Hongyi Zeng, *Stanford University;* Shidong Zhang and Fei Ye, *Google;* Vimalkumar Jeyakumar, *Stanford University;* Mickey Ju and Junda Liu, *Google;* Nick McKeown, *Stanford University;* Amin Vahdat, *Google and University of California, San Diego*

https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/zeng

This paper is included in the Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI '14).

April 2–4, 2014 • Seattle, WA, USA

ISBN 978-1-931971-09-6

Libra: Divide and Conquer to Verify Forwarding Tables in Huge Networks

Hongyi Zeng[†][*], Shidong Zhang[§], Fei Ye[§], Vimalkumar Jeyakumar[†][*]
Mickey Ju[§], Junda Liu[§], Nick McKeown[†], Amin Vahdat[§][‡]
[†]*Stanford University* [§]*Google* [‡]*UCSD*

Abstract

Data center networks often have errors in the forwarding tables, causing packets to loop indefinitely, fall into black-holes or simply get dropped before they reach the correct destination. Finding forwarding errors is possible using static analysis, but none of the existing tools scale to a large data center network with thousands of switches and millions of forwarding entries. Worse still, in a large data center network the forwarding state is constantly in flux, which makes it hard to take an accurate snapshot of the state for static analysis.

We solve these problems with Libra, a new tool for verifying forwarding tables in very large networks. Libra runs fast because it can exploit the scaling properties of MapReduce. We show how Libra can take an accurate snapshot of the forwarding state 99.9% of the time, and knows when the snapshot cannot be trusted. We show results for Libra analyzing a 10,000 switch network in less than a minute, using 50 servers.

1 Introduction

Data center networks are immense. Modern data centers can employ 10,000 switches or more, each with its own forwarding table. In such a large network, failures are frequent: links go down, switches reboot, and routers may hold incorrect prefix entries. Whether routing entries are written by a distributed routing protocol (such as OSPF) or by a remote route server, the routing state is so large and complex that errors are inevitable. We have seen logs from a production data center reporting many thousands of routing changes per day, creating substantial opportunity for error.

Data centers withstand failures using the principles of scale-out and redundant design. However, the underlying assumption is that the system reacts correctly to failures. Dormant bugs in the routing system triggered by rare boundary conditions are particularly difficult to find. Common routing failures include routing loops and black-holes (where traffic to one part of the network disappears). Some errors only become visible when an otherwise benign change is made. For example, when a routing prefix is removed it can suddenly expose a mistake with a less specific prefix.

Routing errors should be caught quickly before too much traffic is lost or security is breached. We therefore need a fast and scalable approach to verify correctness of the entire forwarding state. A number of tools have been proposed for analyzing networks including HSA [10], Anteater [13], NetPlumber [9] and Veriflow [11]. These systems take a snapshot of forwarding tables, then analyze them for errors. We first tried to adopt these tools for our purposes, but ran into two problems. First, they assume the snapshot is consistent. In large networks with frequent changes to routing state, the snapshot might be inconsistent because the network state changes while the snapshot is being taken. Second, none of the tools are sufficiently fast to meet the performance requirements of modern data center networks. For example, Anteater [13] takes more than 5 minutes to check for loops in a 178-router topology.

Hence, we set out to create Libra, a fast, scalable tool to quickly detect loops, black-holes, and other reachability failures in networks with tens of thousands of switches. Libra is much faster than any previous system for verifying forwarding correctness in a large-scale network. Our benchmark goal is to verify all forwarding entries in a 10,000 switch network with millions of rules in minutes.

We make two main contributions in Libra. First, Libra capture stable and consistent snapshots across large network deployments, using the event stream from routing processes (Section 3). Second, in contrast to prior tools that deal with arbitrarily structured forwarding tables, we substantially improve scalability by assuming packet forwarding based on longest prefix matching.

[*]Hongyi Zeng and Vimalkumar Jeyakumar were interns at Google when this work was done. Hongyi Zeng is currently with Facebook.

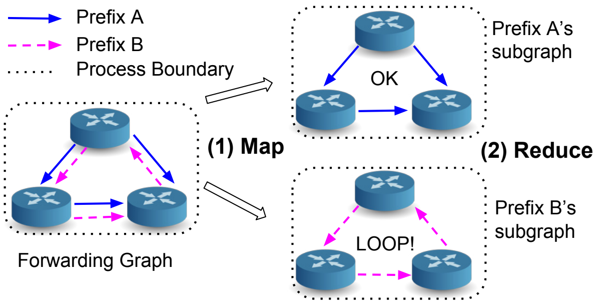
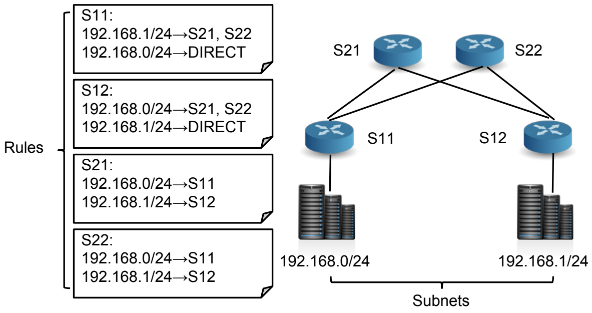


Figure 1: Libra divides the network into multiple forwarding graphs in mapping phase, and checks graph properties in reducing phase.



```
S11:
192.168.1/24→S21, S22
192.168.0/24→DIRECT
S12:
192.168.0/24→S21, S22
192.168.1/24→DIRECT
S21:
192.168.0/24→S11
192.168.1/24→S12
S22:
192.168.0/24→S11
192.168.1/24→S12
```

Figure 2: Small network example for describing the types of forwarding error found by Libra.

Libra uses MapReduce for verification. It starts with the full graph of switches, each with its own prefix table. As depicted in Figure 1, Libra completes verification in two phases. In the *map* phase, it breaks the graph into a number of slices, one for each prefix. The slice consists of only those forwarding rules used to route packets to the destination. In the *reduce* phase, Libra indepedently analyzes each slice, represented as a forwarding graph, *in parallel* for routing failures.

We evaluate Libra on the forwarding tables from three different networks. First, "DCN" is an emulated data center network with 2 million rules and 10,000 switches. Second, "DCN-G" is made from 100 replicas of DCN connected together; i.e., 1 million switches. Third, "INET" is a network with 300 IPv4 routers each contains the full BGP table with half a million rules. The results are encouraging. Libra takes one minute to check for loops and black-holes in DCN, 15 minutes for DCN-G and 1.5 minutes for INET.

2 Forwarding Errors

A small toy network can illustrate three common types of error found in forwarding tables. In the two-level tree network in Figure 2 two top-of-rack (ToR) switches (S11, S12) are connected to two spine switches (S21, S22). The downlinks from S11 and S12 connect to up to 254 servers on the same /24 subnet. The figure shows a "correct" set of forwarding tables. Note that our example network uses multipath routing. Packets arriving at S12 on the right and destined to subnet 192.168.0/24 on the left are load-balanced over switches S21 and S22. Our toy network has 8 rules, and 2 subnets.

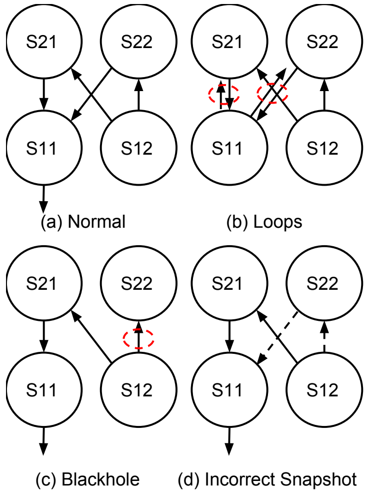


Figure 3: Forwarding graphs for 192.168.0/24 as in Figure 2, and potential abnormalities.

A *forwarding graph* is a directed graph that defines the network behavior for each subnet. It contains a list of (local_switch, remote_switch) pairs. For example, in Figure 3(a), an arrow from S12 to S21 means the packets of subnet 192.168.0/24 can be forwarded from S12 to S21. Multipath routing can be represented by a node that has more than one outgoing edge. Figure 3(b)-(d) illustrates three types of forwarding error in our simple network, depicted in forwarding graphs.

Loops: Figure 3(b) shows how an error in S11's forwarding tables causes a *loop*. Instead of forwarding 192.168.0/24 down to the servers, S11 forwards packets up, i.e., to S21 and S22. S11's forwarding table is now:

```
192.168.0/24 → S21, S22
192.168.1/24 → S21, S22
```

The network has two loops: S21-S11-S21 and S22-S11-S22, and packets addressed to 192.168.0/24 will never reach their destination.

Black-holes: Figure 3(c) shows what happens if S22 loses one of its forwarding entries: `192.168.0/24 → S11`. In this case, if S12 spreads packets destined to 192.168.0/24 over both S21 and S22, packets arriving to S22 will be dropped.

Incorrect Snapshot: Figure 3(d) shows a subtle problem that can lead to *false positives* when verifying forwarding tables. Suppose the link between S11-S22 goes down. Two events take place (shown as dashed arrows

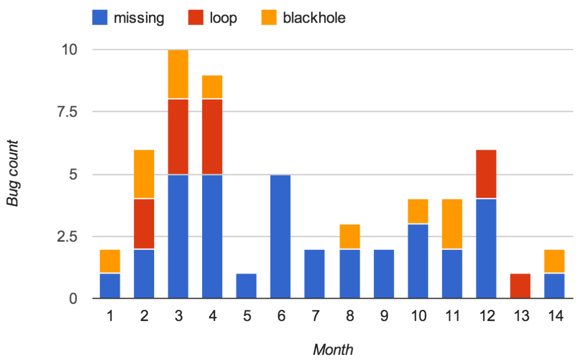


Figure 4: Routing related tickets by month and type.

in the figure): $e1$: S22 deletes `192.168.0/24` $\rightarrow$ `S11`, and $e2$: S12 stops forwarding packets to S22. Because of the asynchronous nature of routing updates, the two events could take place in either order $(e1,e2)$ or $(e2,e1)$. A snapshot may capture one event, but not the other, or might detect them happening in the reverse order.

The sequence $(e1,e2)$ creates a temporary blackhole as in Figure 3(c), whereas the desired sequence $(e2,e1)$ does not. To avoid raising an unnecessary alarm (by detecting $(e1,e2)$ even though it did not happen), or missing an error altogether (by incorrectly assuming that $(e2,e1)$ happened), Libra must detect the correct state of the network.

2.1 Real-world Failure Examples

To understand how often forwarding errors take place, we examined a log of “bug tickets” from 14 months of operation in a large Google data center. Figure 4 categorizes 35 tickets for missing forwarding entries, 11 for loops, and 11 for black-holes. On average, four issues are reported per month.

Today, forwarding errors are tracked down by hand which - given the size of the network and the number of entries - often takes many hours. And because the diagnosis is done after the error occurred, the sequence of events causing the error has usually long-since disappeared before the diagnosis starts. This makes it hard to reproduce the error.

Case 1: Detecting Loops. One type of loop is caused by prefix aggregation. Prefixes are aggregated to compact the forwarding tables: a cluster E can advertise a *single* prefix to reach all of the servers connected “below” it to the core C, which usually includes the addresses of servers that have not yet been deployed. However, packets destined to these non-deployed addresses (e.g., due to machine maintenance) can get stuck in loops. This is because C believes these packets are destined to E, while E lacks the forwarding rules to digest these packets due to the incomplete deployment, instead, E’s default rules lead packets back to C.

This failure does not cause a service to fail (because the service will use other servers instead), but it does degrade the network causing unnecessary congestion. In the past, these errors were ignored because of the prohibitive cost of performing a full cluster check. Libra can finish checking in less a minute, and identify and report the specific switch and prefix entry that are at risk.

Case 2: Discovering Black-holes. In one incident, traffic was interrupted to hundreds of servers. Initial investigation showed that some prefixes had high packet loss rate, but packets seemed to be discarded randomly. It took several days to finally uncover the root cause: A subset of routing information was lost during BGP updates between domains, likely due to a bug in the routing software, leading to black-holes.

Libra will detect missing forwarding entries quickly, reducing the outage time. Libra’s stable snapshots also allow it to disambiguate temporary states during updates from long-term back-holes.

Case 3: Identifying Inconsistencies. Network control runs across several instances, which may fail from time to time. When a secondary becomes the primary, it results in a flurry of changes to the forwarding tables. The changes may temporarily or permanently conflict with the previous forwarding state, particularly if the changeover itself fails before completing. The network can be left in an inconsistent state, leading to packet loss, black-holes and loops.

2.2 Lessons Learned

Simple things go wrong: Routing errors occur even in networks using relatively simple IP forwarding. They also occur due to firmware upgrades, controller failure and software bugs. It is essential to check the forwarding state *independently*, outside the control software.

Multiple moving parts: The network consists of multiple interacting subsystems. For example, in case 1 above, Intra-DC routing is handled locally, but routing is a global property. This can create loops that are hard to detect locally within a subsystem. There are also multiple network controllers. Inconsistent state makes it hard for the control plane to detect failures on its own.

Scale matters: Large data center networks use multipath routing, which means there are many forwarding paths to check. As the number of switches, N, grows the number of paths and prefix tables grow, and the complexity of checking all routes grows with N^2. It is essential for a static checker to scale linearly with the network.

3 Stable Snapshots

It is not easy to take an accurate snapshot of the forwarding state of a large, constantly changing network. But if Libra runs its static checks on a snapshot of the state that never actually occurred, it will raise false alarms and miss real errors. We therefore need to capture - and check - a snapshot of the global forwarding state that actually existed at one instant in time. We call these *stable snapshots*.[1]

When is the state stable? A large network is usually controlled by multiple *routing processes*,[2] each responsible for one or more switches. Each process sends timestamped updates, which we call *routing events*, to add, modify and delete forwarding entries in the switches it is responsible for. Libra monitors the stream of routing events to learn the global network state.

Finding the stable state of a *single* switch is easy: each table is only written by one routing process using a single clock, and all events are processed in order. Hence, Libra can reconstruct a stable state simply by replaying events in timestamp order.

By contrast, it is not obvious how to take a *globally* stable snapshot of the state when different routing processes update their switches using different, unsynchronized clocks. Because the clocks are different, and events may be delayed in the network, simply replaying the events in timestamp order can result in a state that did not actually occur in practice, leading to false positives or missed errors (Section 2).

However, even if we can not precisely synchronize clocks, we can *bound* the difference between any pair of clocks with high confidence using NTP [15]. And we can bound how out-of-date an event packet is, by prioritizing event packets in the network. Thus, every timestamp t can be treated as lying in an interval $(t-\varepsilon, t+\varepsilon)$, where ε bounds the uncertainty of when the event took place.[3] The interval represents the notion that network state changes atomically at some unknown time instant within the interval.

Figure 5 shows an example of finding a stable snapshot instant. It is easy to see that if no routing events are recorded during a 2ε period we can be confident that no routing changes actually took place. Therefore, the snapshot of the current state is stable (i.e., accurate).[4]

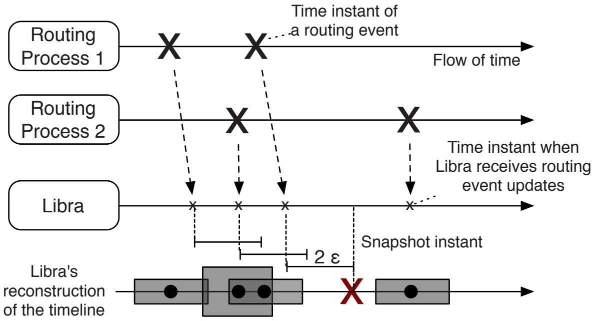

Routing Process 1
Time instant of a routing event
Flow of time
Routing Process 2
Time instant when Libra receives routing event updates
Libra
Snapshot instant
2 ε
Libra's reconstruction of the timeline

Figure 5: Libra's reconstruction of the timeline of routing events, taking into account bounded timestamp uncertainty ε. Libra waits for twice the uncertainty to ensure there are no outstanding events, which is sufficient to deduce that routing has stabilized.

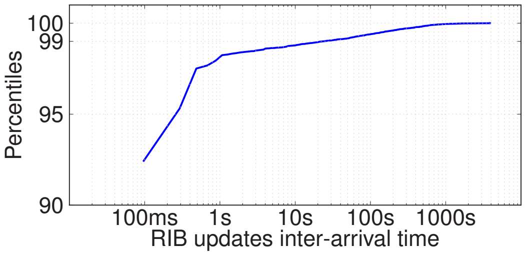


Figure 6: CDF of inter-arrival times of routing events from a large production data center. Routing events are very bursty: over 95% of events happen within 400ms of another event.

The order of any two past events from *different* processes is irrelevant to the current state, since they are applied to different tables without interfering with each other (recall that each table is controlled by only one process). So Libra only needs to replay all events in timestamp order (to ensure events for the *same* table are played in order) to accurately reconstruct the current state.

This observation suggests a simple way to create a stable snapshot by simply waiting for a quiet 2ε period with no routing update events.

Feasibility: The scheme only works if there are frequent windows of size 2ε in which no routing events take place. Luckily, we found that these quiet periods happen frequently: we analyzed a day of logs from all routing processes in a large Google data center with a few thousand switches. Figure 6 shows the CDF of the inter-arrival times for the 28,445 routing events reported by the routing processes during the day. The first thing to notice is the burstiness — over 95% of events occur within 400ms of another event, which means there are long periods when the state is stable. Table 1 shows the fraction of time the network is stable, for different values of ε. As expected, larger ε leads to fewer stable states and smaller percentage of stable time. For example, when ε=100ms, only 2,137 out of all 28,445 states are stable. However,

[1] Note that a stable snapshot is not the same as a *consistent* snapshot [3], which is only one *possible state* of a distributed system that might not actually have occurred in practice.

[2] Libra only considers processes that can directly modify tables. While multiple high-level protocols can co-exist (e.g., OSPF and BGP), there is usually one common low-level table manipulation API.

[3] The positive and negative uncertainties can be different, but here we assume they are the same for simplicity.

[4] A formal proof can be found in [14, § 3.3].

ε/ms	# of stable states	time in stable state/%
0	28,445	100.00
1	16,957	99.97
100	2,137	99.90
1,000	456	99.75
10,000	298	99.60

Table 1: As the uncertainty in routing event timestamps (ε) increases, the number of stable states decreases. However, since routing events are bursty, the state is stable most of the time.

because the event stream is so bursty, the unstable states are extremely short-lived, occupying *in total* only 0.1% (~1.5min) of the entire day. Put another way, for 99.9% of the time, snapshots are stable and the static analysis result is trustworthy.

Taking stable snapshots: The stable snapshot instant provides a reference point to reconstruct the global state. Libra's stable snapshot process works as follows:

1) Take an initial snapshot S_0 as the combination of all switches' forwarding tables. At this stage, each table can be recorded at a slightly different time.

2) Subscribe to timestamped event streams from all routing processes, and apply each event e_i, in the order of their timestamps, to update the state from S_{i-1} to S_i.

3) After applying e_j, if no event is received for 2ε time, declare the current snapshot S_j stable. In other words, S_0 and all past events e_i form a stable state that actually existed at this time instant.

4 Divide and Conquer

After Libra has taken a stable snapshot of the forwarding state, it sets out to statically check its correctness. Given our goal of checking networks with over 10,000 switches and millions of forwarding rules, we will need to break down the task into smaller, parallel computations. There are two natural ways to consider partitioning the problem:

Partition based on switches: Each server could hold the forwarding state for a cluster of switches, partitioning the network into a number of clusters. We found this approach does not scale well because checking a forwarding rule means checking the rules in many (or all) partitions - the computation is quickly bogged down by communication between servers. Also, it is hard to balance the computation among servers because some switches have very different numbers of forwarding rules (e.g. spine and leaf switches).

Partition based on subnets: Each server could hold the forwarding state to reach a set of subnets. The server computes the forwarding graph to reach each subnet, then checks the graph for abnormalities. The difficulty with this approach is that each server must hold the entire set of forwarding tables in memory, and any update to the forwarding rules affects all servers.

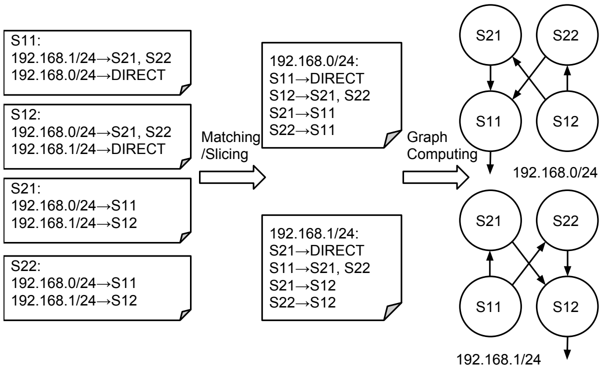


Figure 7: Steps to check the routing correctness in Figure 2.

Libra partitions the network based on subnets, for reasons that will become clear. We observe that the route checker's task can be divided into two steps. First, Libra *associates* forwarding rules with subnets, by finding the set of forwarding rules relevant to a subnet (i.e., they are associated if the subnet is included in the rule's prefix). Second, Libra builds a forwarding graph to reach each subnet, by assembling all forwarding rules for the subnet. Both steps are embarrassingly parallel: matching is done per (subnet, forwarding rule) pair; and each subnet's forwarding graph can be analyzed independently.

Libra therefore proceeds in three steps using N servers:

Step 1 - Matching: Each server is initialized with the *entire* list of subnets, and each server is assigned $1/N$ of all forwarding rules. The server considers each forwarding rule in turn to see if it belongs to the forwarding graph to a subnet (i.e. the forwarding rule is a prefix of the subnet).[5] If there is a match, the server outputs the (subnet, rule) pair. Note that a rule may match more than one subnet.

Step 2 - Slicing: The (subnet, rule) pairs are grouped by subnet. We call each group a *slice*, because it contains all the rules and switches related to this subnet.

Step 3 - Graph Computing: The slices are distributed to N servers. Each server constructs a forwarding graph based on the rules contained in the slice. Standard graph algorithms are used to detect network abnormalities, such as loops and black-holes.

Figure 7 shows the steps to check the network in Figure 2. After the slicing stage, the forwarding rules are organized into two slices, corresponding to the two subnets 192.168.0/24 and 192.168.1/24. The forwarding graph for each slice is calculated and checked in parallel.

[5] Otherwise, a subnet will be fragmented by a more specific rule, leading to a complex forwarding graph. See the last paragraph in Section 9 for detailed discussion.

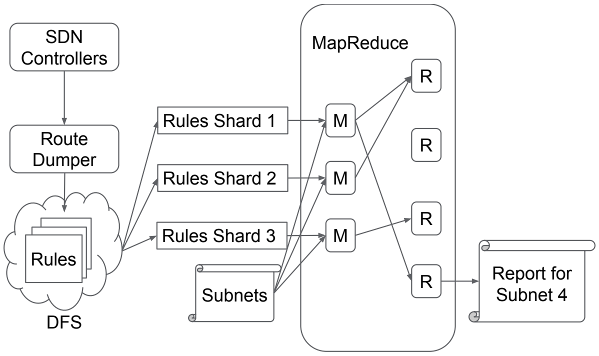


Figure 8: Libra workflow.

If a routing error occurs and the second rule in S11 becomes `192.168.0/24` → `S21, S22`, the loop will show up in the forwarding graph for 192.168.0/24. S11 will point back to S21 and S22, which will be caught in graph loop detection algorithm.

Our three-step process is easily mapped to MapReduce, which we describe in the next section.

5 Libra

Libra consists of two main components: a *route dumper* and a MapReduce-based *route checker*. Figure 8 shows Libra's workflow.

The route dumper takes stable snapshots from switches or controllers, and stores them in a distributed file system. Next, the snapshot is processed by a MapReduce-based checker.

A quick review of MapReduce: MapReduce [5] divides computation into two phases: *mapping* and *reducing*. In the mapping phase, the input is partitioned into small "shards". Each of them is processed by a mapper in parallel. The mapper reads in the shard line by line and outputs a list of `<key, value>` pairs. After the mapping phase, the MapReduce system *shuffles* outputs from different mappers by sorting by the key. After shuffling, each reducer receives a `<key, values>` pair, where `values=[value1, value2, ...]` is a list of all values corresponding to the `key`. The reducer processes this list and outputs the final result. The MapReduce system also handles checkpointing and failure recovery.

In Libra, the set of forwarding rules is partitioned into small shards and delivered to mappers. Each mapper also takes a full set of subnets to check, which by default contains all subnets in the cluster, but alternatively can be subsets selected by user. Mappers generate intermediate keys and values, which are shuffled by MapReduce. The reducers compile the values that belong to the same subnet and generate final reports.

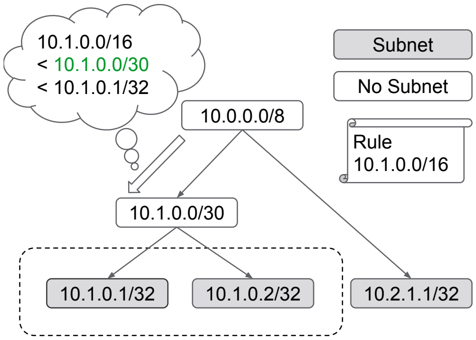


Figure 9: Find all matching subnets in the trie. `10.1.0.0/30` (X) is the smallest matching trie node bigger than the rule `10.1.0.0/16` (A). Hence, its children with subnets `10.1.0.1/32` and `10.1.0.2/32` match the rule.

5.1 Mapper

Mappers are responsible for slicing networks by subnet. Each mapper reads one forwarding rule at a time. If a subnet matches the rule, the mapper outputs the subnet prefix as the intermediate key, along with the value `<rule_mask_len, local_switch, remote_switches, priority>`. The following is an example (`local_switch`, `remote_switches`, `priority` is omitted):

Subnets:	`192.168.1.1/32`
	`192.168.1.2/32`
Rules:	`192.168.1.0/28`
	`192.168.0.0/16`
Outputs:	`<192.168.1.1/32, 28>`
	`<192.168.1.1/32, 16>`
	`<192.168.1.2/32, 16>`

Since each mapper only sees a portion of the forwarding rules, there may be a longer and more specific—but unseen—matching prefix for the subnet in the same forwarding table. We *defer* finding the longest matching to the reducers, which see all matching rules.

Mappers are first initialized with a full list of subnets, which are stored in an in-memory binary trie for fast prefix matching. After initialization, each mapper takes a shard of the routing table, and matches the rules against the subnet trie. This process is different from the conventional longest prefix matching: First, in conventional packet matching, rules are placed in a trie and packets are matched one by one. In Libra, we build the trie with *subnets*. Second, the goal is different. In conventional packet matching, one looks for the longest matching rule. Here, mappers simply output *all* matching subnets in the trie. Here, matching has the same meaning—the subnet's prefix must fully fall within the rule's prefix.

We use a trie to efficiently find "all matching prefixes," by searching for the *smallest matching trie node* (called node X) that is *bigger or equal to* the rule prefix (called node A). Here, “small” and “big” refer to the lexicographical order (not address space size), where for each bit in an IP address, $wildcard < 0 < 1$. X may or may not contain a subnet. If X exists, we enumerate all its nonempty decedents (including X itself). Otherwise, we declare that there exist no matching subnets in the trie. Figure 9 shows an example. `10.1.0.0/30` (X) is the smallest matching trie node bigger than the rule `10.1.0.0/16` (A). Hence, its children with subnets `10.1.0.1/32` and `10.1.0.2/32` match the rule.

Proof: We briefly prove why this algorithm is correct. In an uncompressed trie, each bit in the IP address is represented by one level, and so the algorithm is correct by definition: if there exist matching subnets in the trie, A must exist in the trie and its descendants contain all matching prefixes, which means $A = X$.

In a compressed trie, nearby nodes may be combined. A may or may not exist in the trie. If it exists, the problem reduces to the uncompressed trie scenario. If A does not exist in the trie, X (if it exists) contains all matching subnets in its descendants. This is because:

a) Any node Y smaller than X does not match A. Because there is no node bigger than A and smaller than X (otherwise X is not the smallest matching node), $Y < X$ also means $Y < A$. As a result, Y cannot fall within A's range. This is because for Y to fall within A, all A's non-wildcard bits should appear in Y, which implies $Y \geq A$.

b) Any node Y bigger than the biggest descendants of X does not match A. Otherwise, X and Y must have a common ancestor Z, where Z matches A because both X and Y match A, and $Z < X$ because Z is the ancestor of X (a node is always smaller than its descendants). This contradicts the assumption that X is smallest matching node of A.

Time complexity: We can break down the time consumed by the mapping phase into two parts. The time to construct the subnet trie is $O(T)$, where T is the number of subnets, because inserting an IP prefix into a trie takes constant time ($\leq$ length of IP address). If we consider a single thread, it takes $O(R)$ time to match R rules against the trie. So the total time complexity is $O(R+T)$. If N mappers share the same trie, we can reduce the time to $O(R+\frac{T}{N})$. Here, we assume $R \gg T$. If $T \gg R$, one may want to construct a trie with rules rather than subnets (as in conventional longest-prefix-matching).

5.2 Reducer

The outputs from the mapping phase are shuffled by intermediate keys, which are the subnets. When shuffling finishes, a reducer will receive a subnet, along with an unordered set of values, each containing `rule_mask_len`, `local_switch`, `remote_switches`, and `priority`. The reducer first selects the highest priority rule per `local_switch`: For the same `local_switch`, the rule with higher `priority` is selected; if two rules have the same priority, the one with larger `mask_len` is chosen. The reducer then constructs a directed forwarding graph using the selected rules. Once the graph is constructed, the reducer uses graph library to verify the properties of the graph, for example, to check if the graph is loop-free.

Time complexity: In most networks we have seen, a subnet matches at most 2-4 rules in the routing table. Hence, selecting the highest priority rule and constructing the graph takes $O(E)$ time, where E is the number of physical links in the network. However, the total runtime depends on the verification task, as we will discuss in Section 6.

5.3 Incremental Updates

Until now, we have assumed Libra checks the forwarding correctness from scratch each time it runs. Libra also supports incremental updates of subnets and forwarding rules, allowing it to be used as an independent “correctness checking service” similar to NetPlumber [9] and Veriflow [11]. In this way, Libra could be used to check forwarding rules quickly, *before* they are added to the forwarding tables in the switches. Here, a in-memory, “streaming” MapReduce runtime (such as [4]) is needed to speed up the event processing.

Subnet updates. Each time we add a subnet for verification, we need to rerun the whole MapReduce pipeline. The mappers takes $O(\frac{R}{N})$ time to find the relevant rules. And a single reducer takes $O(E)$ time to construct the directed graph slice for the new subnet. If one has several subnets to add, it is faster to run them in a batch, which takes $O(T+\frac{R}{N})$ instead of $O(\frac{RT}{N})$ to map.

Removing subnets is trivial. All results related to the subnets are simply discarded.

Forwarding rule updates. Figure 10 shows the workflow to add new forwarding rules. To support incremental updates of rules, reducers need to store the forwarding graph for each slice it is responsible for. The reducer could keep the graph in memory or disk—the trade-off is a larger memory footprint.[6] If the graphs are in disk, a fixed number of idle reducer processes live in the memory and fetch graphs upon request. Similarly, the mappers need to keep the subnet trie.

To add a rule, a mapper is spawned just as it sees another line of input (Step 1). Matching subnets from the trie are shuffled to multiple reducers (Step 2). Each reducer reads the previous slice graph (Step 3), and recalculates it with the new rule (Step 4).

Deleting a rule is similar. The mapper tags the rule as “to be deleted” and pass it to reducers for updating the

[6] At any time instance, only a small fraction of graphs will be updated, and so keeping all states in-memory can be quite inefficient.

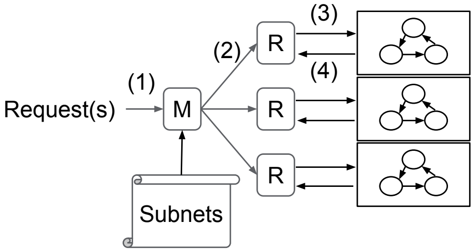


Figure 10: Incremental rule updates in Libra. Mappers dispatch matching <subnet, rule> pair to reducers, indexed by subnet. Reducers update the forwarding graph and recompute graph properties.

slice graph. However, in the graph's adjacency list, the reducer not only needs to store the highest priority rule, but also *all* matching rules. This is because if a highest priority rule is deleted, the reducer must use the second highest priority rule to update the graph.

Besides updating graphs, in certain cases, graph properties can also be checked incrementally, since the update only affects a small part of graph. For example, in loop-detection, adding an edge only requires a Depth-First-Search (DFS) starting from the new edge's destination node, which normally will not traverse the entire graph.

Unlike NetPlumber and Veriflow, Libra does not need to explicitly remember the dependency between rules. This is because the dependency is already encoded in the matching and shuffling phases.

5.4 Route Dumper

The route dumper records each rule using five fields: `<switch, ip, mask_len, nexthops, priority>`. `switch` is the unique ID of the switch; `ip` and `mask_len` is the prefix. `nexthops` is a list of port names because of multipath. `priority` is an integer field serving as a tie-breaker in longest prefix matching. By storing the egress ports in `nexthops`, Libra encodes the topology information in the forwarding table.

Although the forwarding table format is mostly straightforward, two cases need special handling:

Ingress port dependent rules. Some forwarding rules depend on particular ingress ports. For example, a rule may only be in effect for packets entering the switch from port `xe1/1`. In reducers we want to construct a simple directed graph that can be represented by an adjacency list. Passing this ingress port dependency to the route checker will complicate the reducer design, since the next hop in the graph depends not only on the current hop, but also the *previous hop*.

We use the notion of *logical switches* to solve this problem. First, if a switch has rules that depend on the ingress port, we split the switch into multiple logical switches. Each logical switch is given a new name and contains the rules depending on one ingress port, so that the port is "owned" by the new logical switch. We copy rules from the original switch to the logical switch. Second, we update the rules in upstream switches to forward to the logical switch.

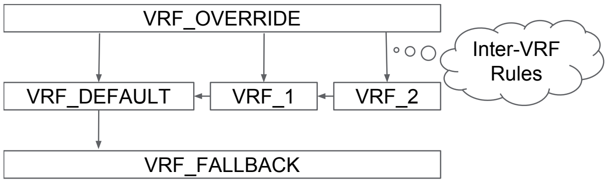


Figure 11: Virtual Routing and Forwarding (VRFs) are multiple tables within the same physical switch. The tables have dependency (inter-VRF rules) between them.

Multiple tables. Modern switches can have multiple forwarding tables that are chained together by arbitrary matching rules, usually called "Virtual Routing and Forwarding" (VRF). Figure 11 depicts an example VRF set up: incoming packets are matched against VRF_OVERRIDE. If no rule is matched, they enter VRF_1 to VRF_16 according to some "triggering" rules. If all matching fails, the packet enters VRF_DEFAULT.

The route dumper maps multiple tables in a physical switch into multiple logical switches, each containing one forwarding table. Each logical switch connects to other logical switches directly. The rules chaining these VRFs are added as lowest priority rules in the logical switch's table. Hence, if no rule is matched, the packet will continue to the next logical switch in the chain.

6 Use cases

In Libra, the directed graph constructed by the reducer contains *all* data plane information for a particular subnet. In this graph, each vertex corresponds to a forwarding table the subnet matched, and each edge represents a possible link the packet can traverse. This graph also encodes multipath information. Therefore, routing correctness directly corresponds to graph properties.

Reachability: A reachability check ensures the subnet can be reached from any switch in the network. This property can be verified by doing a (reverse) DFS from the subnet switch, and checking if the resulting vertex set contains all switches in the network. The verification takes $O(V+E)$ time where V is the number of switches and E the number of links.

Loop detection: A loop in the graph is equivalent to at least one *strongly connected component* in the directed graph. Two vertices `s1` and `s2` belong to a strongly connected component, if there is a path from `s1` to `s2` and a path from `s2` to `s1`. We find strongly connected components using Tarjan's Algorithm [21] whose time complexity is $O(V+E)$.

Black-holes: A switch is a black-hole for a subnet if the switch does not have a matching route entry for the subnet. Some black-holes are legitimate: if the switch is the last hop for the subnet, or there is an explicit drop rule. Implicit drop rules need to be checked if that is by design. Black-holes map to vertices with zero out-degree, which can therefore be enumerated in $O(V)$ time.

Waypoint routing: Network operators may require traffic destined to certain subnets to go through a "waypoint," such as a firewall or a middlebox. Such behavior can be verified in the forwarding graph by checking if the waypoint exists on all the forwarding paths. Specifically, one can remove the waypoint and the associated links, and verify that no edge switches appear any more in a DFS originated from the subnet's first hop switch, with the runtime complexity of $O(V+E)$.

7 Implementation

We have implemented Libra for checking the correctness of Software-Defined Network (SDN) clusters. Each cluster is divided into several domains where each domain is controlled by a controller. Controllers exchange routing information and build the routing tables for each switch.

Our Libra prototype has two software components. The route dumper, implemented in Python, connects to each controller and downloads routing events, forwarding tables and VRF configurations in Protocol Buffers [17] format in parallel. It also consults the topology database to identify the peer of each switch link. Once the routing information is downloaded, we preprocess the data as described in Section 5.4 and store it in a distributed file system.

The route checker is implemented in C++ as a MapReduce application in about 500 lines of code. We use a Trie library for storing subnets, and use Boost Graph Library [1] for all graph computation. The same binary can run at different levels of parallelism—on a single machine with multiple processes, or on a cluster with multiple machines, simply by changing command line flags.

Although Libra's design supports incremental updates, our current prototype only does batch processing. We use micro-benchmarks to evaluate the specific costs for incremental processing in Section 8.5, on a simplified prototype with one mapper and one reducer.

8 Evaluation

To evaluate Libra's performance, we first measure start-to-finish runtime on a single machine with multi-threading, as well as on multiple machines in a cluster. We also demonstrate Libra's linear scalability as well as its incremental update capability.

Data set	Switches	Rules	Subnets
DCN	11,260	2,657,422	11,136
DCN-G	1,126,001	265,742,626	1,113,600
INET	316	151,649,486	482,966

Table 2: Data sets used for evaluating Libra.

8.1 Data Sets

We use three data sets to evaluate the performance of Libra. The detailed statistics are shown in Table 2.

DCN: DCN is an SDN testbed used to evaluate the scalability of the controller software. Switches are emulated by OpenFlow agents running on commodity machines and connected together through a virtual network fabric. The network is partitioned among controllers, which exchange routing information to compute the forwarding state for switches in their partition. DCN contains about 10 thousand switches and 2.6 million IPv4 forwarding rules. VRF is used throughout the network.

DCN-G: To stress test Libra, we replicate DCN 100 times by shifting the address space in DCN such that each DCN-part has a unique IP address space. A single top-level switch interconnects all the DCN pieces together. DCN-G has 1 million switches and 265 million forwarding rules.

INET: INET is a synthetic wide area backbone network. First, we use the Sprint network topology discovered by RocketFuel project [20], which contains roughly 300 routers. Then, we create an interface for each prefix found in a full BGP table from Route Views [18] ($\sim$500k entries as of July 2013), and spread them randomly and uniformly to each router as "local prefixes." Finally, we compute forwarding tables using shortest path routing.

8.2 Single Machine Performance

We start our evaluation of Libra by running loop detection locally on a desktop with Intel 6-core CPU and 32GB memory. Table 3 summarizes the results. We have learned several aspects of Libra from single machine evaluation:

I/O bottlenecks: Standard MapReduce is disk-based: Inputs are piped into the system from disks, which can create I/O bottlenecks. For example, on INET, reading from disk take 15 times longer than graph computation. On DCN, the I/O time is much shorter due to the smaller number of forwarding rules. In fact, in both cases, the I/O is the bottleneck and the CPU is not fully utilized. The runtime remains the same with or without mapping. Hence, the mapping phase is omitted in Table 3.

Memory consumption: In standard MapReduce, intermediate results are flushed to disk after the mapping phase before shuffling, which is very slow on a single machine. We avoid this by keeping all intermediate states

Threads	1	2	4	6	8
Read/s	13.7				
Shuffle/s	7.4				
Reduce/s	46.3	25.8	15.6	12.1	11.1
Speedup	1.00	1.79	2.96	3.82	4.17

a) DCN with 2000 subnets

Threads	1	2	4	6	8
Read/s	170				
Shuffle/s	3.8				
Reduce/s	11.3	5.9	3.2	2.7	2.1
Speedup	1.00	1.91	3.53	4.18	5.38

b) INET with 10,000 subnets

Table 3: Runtime of loop detection on DCN and INET data sets on single machine. The number of subnets is reduced compared to Table 2 so that all intermediate states can fit in the memory. Read and shuffle phases are single-threaded due to the framework limitation.

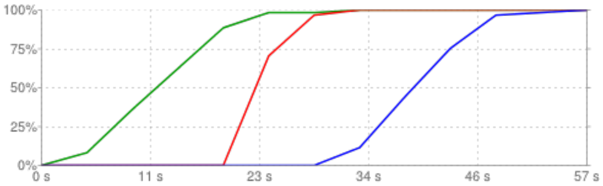


Figure 12: Example progress percentage (in Bytes) of Libra on DCN. The three curves represent (from left to right) Mapping, Shuffling, and Reducing phases, which are partially overlapping. The whole process ends in 57 seconds.

in-memory. However, it limits the number of subnets that can be verified at a time—intermediate results are all matching (subnet, rule) pairs. On a single machine, we have to limit the number of subnets to 2000 in DCN and 10,000 in INET to avoid running out of memory.

Graph size dominates reducing phase: The reducing phase on DCN is significantly slower than on INET, despite INET having 75 times more forwarding rules. With a single thread, Libra can only process 43.2 subnets per second on DCN, compared with 885.0 subnets per second on INET (20.5 times faster). Note that DCN has 35.6 times more nodes. This explains the faster running time on INET, since the time to detect loops grows linearly with the number of edges and nodes in the graph.

Multi-thread: Libra is multi-threaded, but the multi-thread speedup is not linear. For example, on DCN, using 8 threads only resulted in a 4.17 speedup. This effect is likely due to inefficiencies in the threading implementation in the underlying MapReduce framework, although theoretically, all reducer threads should run in parallel without state sharing.

	DCN	**DCN-G**	**INET**
Machines	50	20,000	50
Map Input/Byte	844M	52.41G	12.04G
Shuffle Input/Byte	1.61G	16.95T	5.72G
Reduce Input/Byte	15.65G	132T	15.71G
Map Time/s	31	258	76.8
Shuffle Time/s	32	768	76.2
Reduce Time/s	25	672	16
Total Time/s	**57**	**906**	**93**

Table 4: Running time summary of the three data sets. Shuffle input is compressed, while map and reduce inputs are uncompressed. DCN-G results are extrapolated from processing 1% of subnets with 200 machines as a single job.

8.3 Cluster

We use Libra to check for loops against our three data sets on a computing cluster. Table 4 summarizes the results. Libra spends 57 seconds on DCN, 15 minutes on DCN-G, and 93 seconds on INET. To avoid overloading the cluster, the DCN-G result is extrapolated from the runtime of 1% of DCN-G subnets with 200 machines. We assume 100 such jobs running in parallel—each job processes 1% of subnets against all rules. All the jobs are *independent* of each other.

We make the following observations from our cluster-based evaluation.

Runtime in different phases: In all data sets, the sum of the runtime in the phases is larger than the start-to-end runtime. This is because the phases can overlap each other. There is no dependency between different mapping shards. A shard that finishes the mapping phase can enter the shuffling phase without waiting for other shards. However, there is a global barrier between mapping and reducing phases since MapReduce requires a reducer to receive all intermediate values before starting. Hence, the sum of runtime of mapping and reducing phases roughly equals the total runtime. Table 4 shows the overlapping progress (in bytes) of all three phases in an analysis of DCN.

Shared-cluster overhead: These numbers are a lower bound of what Libra can achieve for two reasons: First, the cluster is shared with other processes and lacks performance isolation. In all experiments, Libra uses 8 threads. However, the CPU utilization is between 100% and 350% on 12-core machines, whereas it can achieve 800% on a dedicated machine. Second, the machines start processing at different times—each machine may need different times for initialization. Hence, all the machines are not running at full-speed from the start.

Parallelism amortizes I/O overhead: Through detailed counters, we found that unlike in the single machine case (where I/O is the bottleneck), the mapping and reducing

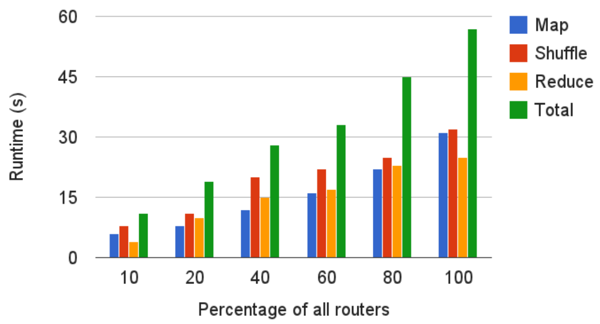


Figure 13: Libra runtime increases linearly with network size.

time dominates the total runtime. We have seen 75%–80% of time spent on mapping/reducing. This is because the aggregated I/O bandwidth of all machines in a cluster is much higher than a single machine. The I/O is faster than the computation throughput, which means threads will not starve.

8.4 Linear scalability

Figure 13 shows how Libra scales with the size of the network. We change the number of devices in the DCN network, effectively changing both the size of the forwarding graph and the number of rules. We do not change the number of subnets. Our experiments run the loop detection on 50 machines, as in the previous section. The figure shows that the Libra runtime scales linearly with the number of rules. The reduce phase grows more erratically than the mapping time, because it is affected by both nodes and edges in the network, while mapping only depends on the number of rules.

Libra's runtime is not necessarily inversely proportional to the number of machines used. The linear scalability only applies when mapping and reducing time dominate. In fact, we observe that more machines can take *longer* to finish a job, because the overhead of the MapReduce system can slow down Libra. For example, if we have more mapping/reducing shards, we need to spend an additional overhead on disk shuffling. We omit the detailed discussion as it depends on the specifics of the underlying MapReduce framework.

8.5 Incremental Updates

Libra can update forwarding graphs incrementally as we add and delete rules in the network, as shown in Section 5.3. To understand its performance, we can breakdown Libra's incremental computation into two steps: (1) time spent in prefix matching (map phase) to find which subnets are affected, and (2) time to do an incremental DFS starting from the node whose routing entries have changed (reduce phase). We also report the total heap memory allocated.

	Map (μs)	**Reduce (ms)**	**Memory (MB)**
DCN	0.133	0.62	12
DCN-G	0.156	1.76	412
INET	0.158	<0.01	7

Table 5: Breakdown of runtime for incremental loop checks. The unit for map phase is microsecond and the unit for reduce phase is millisecond.

We measured the time for each of the components as follows: (1) for prefix matching, we randomly select rules and find out all matching subnets using the algorithm described in Section 5.1, and (2) for incremental DFS, we started a new DFS from randomly chosen nodes in the graph. Both results are averaged across 1000 tests. The results are shown in Table 5.

First, we verified that no matter how large the subnet trie is, prefix matching takes almost constant time: DCN-G's subnet trie is 100 times larger than DCN-G's but takes almost the same time. Second, the results also show that the runtime for incremental DFS is likely to be dominated by I/O rather than compute, because the size of the forwarding graph does not exceed the size of the physical network. Even the largest dataset, DCN-G, has only about a million nodes and 10 million edges, which fits into 412MBytes of memory. This millisecond runtime is comparable to results reported in [9] and [11], but now on much bigger networks.

9 Limitations of Libra

Libra is designed for static headers: Libra is faster and more scalable than existing tools because it solves a narrower problem; it assumes packets are only forwarded based on IP prefixes, and that headers are not modified along the way. Unlike, say HSA, Libra cannot process a graph that forwards on an arbitrary mix of headers, since it is not obvious how to carry matching information from mappers to reducers, or how to partition the problem.

As with other static checkers, Libra cannot handle non-deterministic network behavior or dynamic forwarding state (e.g., NAT). It requires a complete, static snapshot of forwarding state to verify correctness. Moreover, Libra cannot tell *why* a forwarding state is incorrect or how it will evolve as it does not interpret control logic.

Libra is designed to slice the network by IP subnet: If headers are transformed in a deterministic way (e.g., static NAT and IP tunnels), Libra can be extended by combining results from multiple forwarding graphs at the end. For example, 192.168.0/24 in the Intra-DC network may be translated to 10.0.0/24 in the Inter-DC network. Libra can construct forwarding graphs for both 192.168.0/24 and 10.0.0/24. When analyzing the two subgraphs we can add an edge to connect them.

Forwarding graph too big for a single server: Libra scales linearly with both subnets and rules. However, a single reducer still computes the entire forwarding graph, which might still be too large for a single server. Since the reduce speed depends on the size of the graph, we could use distributed graph libraries [16] in the reduce phase to accelerate Libra.

Subnets must be contained by a forwarding rule: In order to break the network into one forwarding graph per subnet, Libra examines all the forwarding rules to decide which rules *contain* the subnet. This is a practical assumption because, in most networks, the rule is a prefix that *aggregates* many subnets. However, if the rule has a longer, more specific prefix (e.g., it is for routing to a specific end-host or router console) than the subnet's, the forwarding graph would be complicated since the rule, represented as an edge in the graph, does not apply to all addresses of the subnet. In this case, one can use Veriflow [11]'s notion of equivalence classes to acquire subnets directly from the rules themselves. This technique may serve as an alternative way to find all matching (subnet, rule) pairs. We leave this for future work.

10 Related Work

Static data plane checkers: Xie et. al introduced algorithms to analyze reachability in IP networks [22]. Anteater [13] makes them practical by converting the checking problem into a Boolean satisfiability problem and solving it with SAT solvers. Header space analysis [10] tackles general protocol-independent static checking using a geometric model and functional simulation. Recently, NetPlumber [9] and Veriflow [11] show that, for small networks (compared to the ones we consider here) static checking can be done in milliseconds by tracking the dependency between rules. Specifically, Veriflow slices the network into *equivalence classes* and builds a *forwarding graph* for each class, in a similar fashion to Libra.

However, with the exception of NetPlumber, all of these tools and algorithms assume centralized computing. NetPlumber introduces a "rule clustering" technique for scalabilty, observing that rule dependencies can be separated into several relatively *independent* clusters. Each cluster is assigned to a process so that rule updates can be handled individually. However, the benefits of parallelism diminish when the number of workers exceeds the number of natural clusters in the ruleset. In contrast, Libra scales linearly with both rules and subnets. Specifically, even two rules have dependency, Libra can still place them into different map shards, and allow reducers to resolve the conflicts.

Other network troubleshooting techniques: Existing network troubleshooting tools focus on a variety of network components. Specifically, the explicitly layered design of SDN facilitates systematic troubleshooting [8]. Efforts in formal language foundations [6] and model-checking control programs [2] reduce the probability of buggy control planes. This effort has been recently extended to the embedded software on switches [12]. However, based on our experience, multiple simultaneous writers in a dynamic environment make developing a bug-free control plane extremely difficult.

Active testing tools [23] reveal the inconsistency between the forwarding table and the actual forwarding state by sending out specially designed probes. They can discover runtime properties such as congestion, packet loss, or faulty hardware, which cannot be detected by static checking tools. Libra is orthogonal to these tools since we focus on forwarding table *correctness*.

Researchers have proposed systems to extract abnormalities from event histories. STS [19] extracts "minimal causal sequences" from control plane event history to explain a particular crash or other abnormalities. NDB [7] compiles packet histories and reasons about data plane correctness. These methods avoid taking a stable snapshot from the network.

11 Conclusion

Today's networks require way too much human intervention to keep them working. As networks get larger and larger there is huge interest in automating the control, error-reporting, troubleshooting and debugging. Until now, there has been no way to automatically verify all the forwarding behavior in a network with tens of thousands of switches. Libra is fast because it focuses on checking the IP-only fabric commonly used in data centers. Libra is scalable because it can be implemented using MapReduce allowing it to harness large numbers of servers. In our experiments, Libra can meet the benchmark goal we set out to achieve: it can verify the correctness of a 10,000-node network in 1 minute using 50 servers. In future, we expect tools like Libra to check the correctness of even larger networks in real-time.

Modern large networks have gone far beyond what human operators can debug with their wisdom and intuition. Our experience shows that it also goes beyond what single machine can comfortably handle. We hope that Libra is just the beginning of bringing distributed computing into the network verification world.

References

[1] Boost Graph Library. `http://www.boost.org/libs/graph`.

[2] M. Canini, D. Venzano, P. Peresini, D. Kostic, and J. Rexford. A NICE Way to Test OpenFlow Applications. *NSDI*, 2012.

[3] K. M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM ToCS*, 1985.

[4] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. MapReduce Online. *NSDI*, 2010.

[5] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *OSDI*, 2004.

[6] N. Foster, A. Guha, M. Reitblatt, A. Story, M. Freedman, N. Katta, C. Monsanto, J. Reich, J. Rexford, C. Schlesinger, D. Walker, and R. Harrison. Languages for Software-Defined Networks. *IEEE Communications Magazine*, 2013.

[7] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown. Where is the Debugger for my Software-Defined Network? *HotSDN*, 2012.

[8] B. Heller, C. Scott, N. McKeown, S. Shenker, A. Wundsam, H. Zeng, S. Whitlock, V. Jeyakumar, N. Handigol, J. McCauley, K. Zarifis, and P. Kazemian. Leveraging SDN layering to systematically troubleshoot networks. *HotSDN*, 2013.

[9] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real Time Network Policy Checking using Header Space Analysis. *NSDI*, 2013.

[10] P. Kazemian, G. Varghese, and N. McKeown. Header Space Analysis: Static Checking for Networks. *NSDI*, 2012.

[11] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. VeriFlow: Verifying Network-Wide Invariants in Real Time. *NSDI*, 2013.

[12] M. Kuzniar, P. Peresini, M. Canini, D. Venzano, and D. Kostic. A SOFT Way for OpenFlow Switch Interoperability Testing. *CoNEXT*, 2012.

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

[14] K. Marzullo and G. Neiger. Detection of global state predicates. *Springer*, 1992.

[15] D. L. Mills. Internet time synchronization: the network time protocol. *IEEE Transactions on Communications*, 1991.

[16] The Parallel Boost Graph Library. `http://osl.iu.edu/research/pbgl/`.

[17] Protocol Buffers. `https://code.google.com/p/protobuf/`.

[18] Route Views. `http://www.routeviews.org/`.

[19] C. Scott, A. Wundsam, S. Whitlock, A. Or, E. Huang, K. Zarifis, and S. Shenker. How Did We Get Into This Mess? Isolating Fault-Inducing Inputs to SDN Control Software. *Technical Report UCB/EECS-2013-8*, 2013.

[20] N. Spring, R. Mahajan, D. Wetherall, and T. Anderson. Measuring ISP topologies with rocketfuel. *IEEE/ACM TON*, 2004.

[21] R. Tarjan. Depth-first search and linear graph algorithms. *12th Annual Symposium on Switching and Automata Theory*, 1971.

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

[23] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown. Automatic Test Packet Generation. *CoNEXT*, 2012.