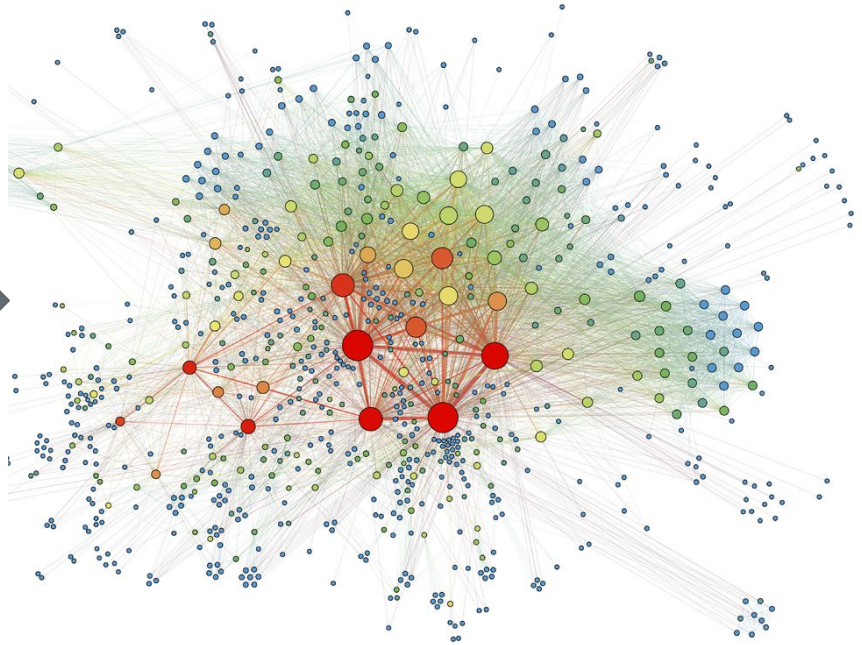
Libra: Divide and Conquer to Verify Forwarding Tables in Huge Networks

Hongyi Zeng, Shidong Zhang, Fei Ye, Vimalkumar Jeyakumar, Mickey Ju, Junda Liu, Nick McKeown, Amin Vahdat

Modern data center networks

- Few thousands of switches each with its own forwarding table.
- Managed by some distributed routing protocol eg. OSPF.
- The routing state is so large and complex that errors are inevitable.

Solution: analyze the network to find errors



Problems with known tools

Tools like Anteater, NetPlumber or Veriflow analyzes snapshot of the network either by converting the checking problem into a Boolean satisfiability problem or by tracking the dependency between rules.

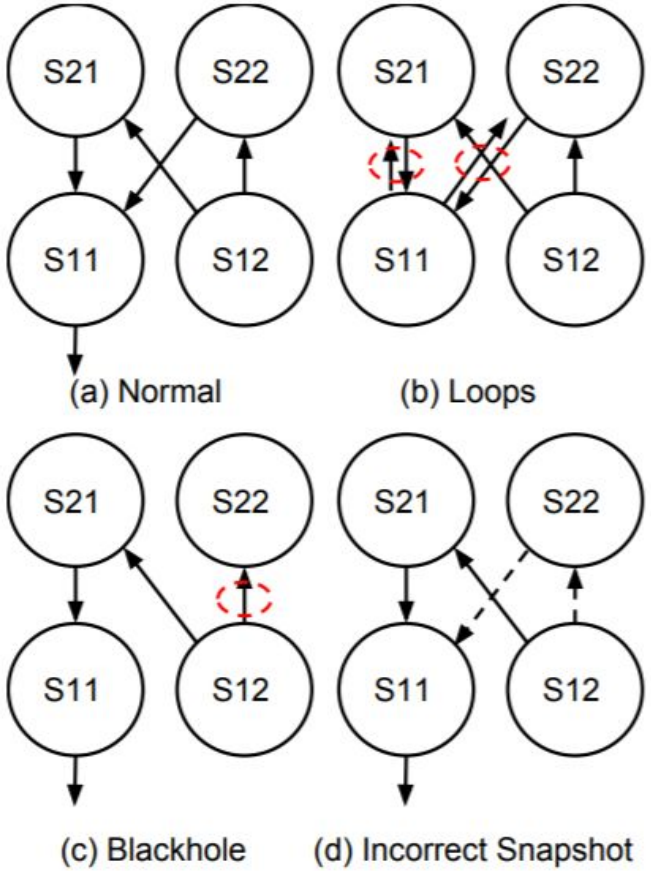
This works fine for smaller networks but:

- Lacks scalability
- Assumes that snapshot is consistent
- For big networks, requires many hours to finish analysis.

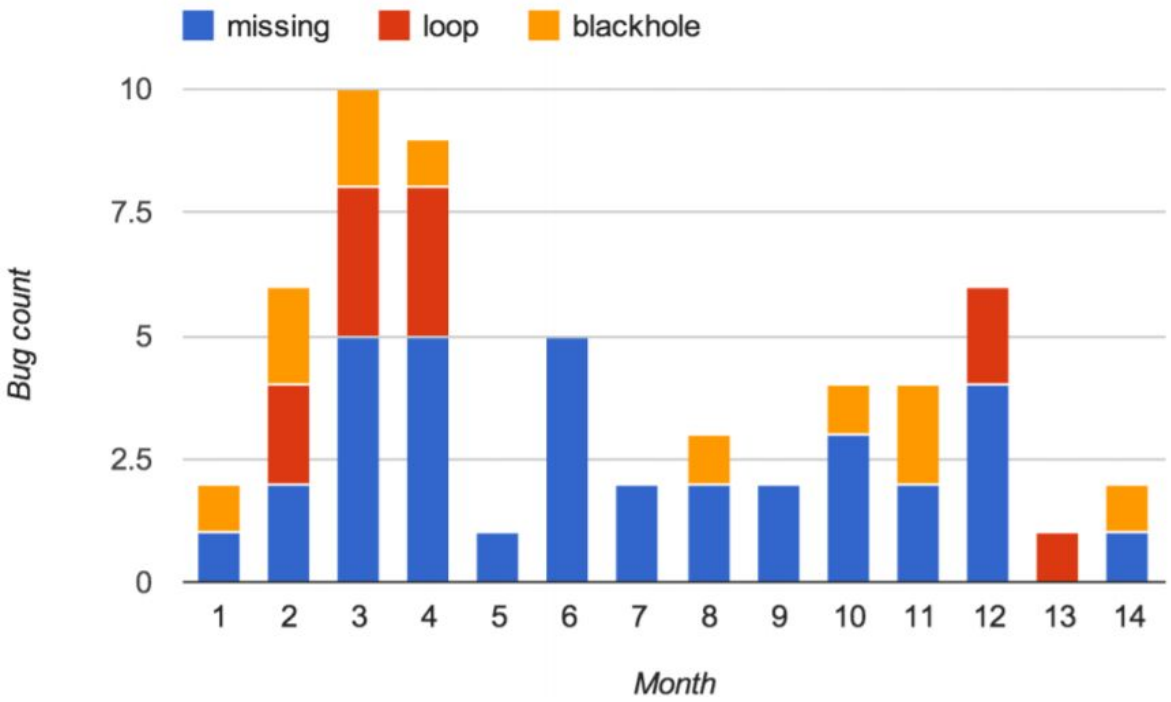
Libra designers goal

Verify all forwarding entries in a 10,000 switch network with millions of rules in minutes.

Typical routing errors



Bug tickets from 14 months of operation in a large Google data center



How to make stable snapshot

Problem 1 - Taking snapshot takes some time, we can look at different switches in different points in time.

Problem 2 - Different routing processes using different unsynchronized clocks.

Problem 1 - Taking snapshot takes some time, we can look at different switches in different points in time.

Solution:

Take initial (maybe not stable) snapshot and subscribe to timestamped stream of events from all routing processes. Apply events up to some specific point in time to the snapshot.

New problem to solve - how to determine this point in time?

Problem 2 - Different routing processes using different unsynchronized clocks.
1.1 - When to stop applying events from stream.

We can not precisely synchronize clocks, but we can bound the difference between any pair of clocks with high confidence using NTP. Let's call this difference ε.

Now all events with timestamp t happened between t-ε and t+ε.

We can wait for a window without events of length 2ε, after that time we can be sure there was some actual time frame without updates, so we have our stable snapshot.

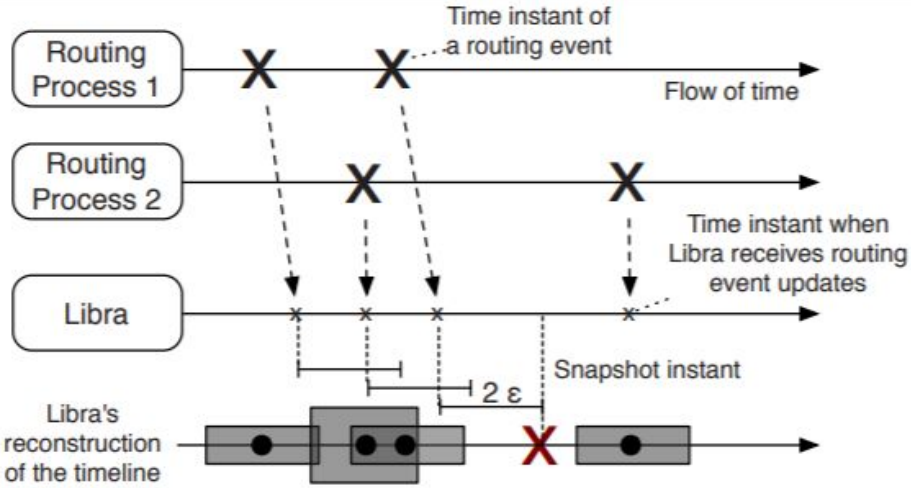


Figure 5: Libra's reconstruction of the timeline of routing events, taking into account bounded timestamp uncertainty ε. Libra waits for twice the uncertainty to ensure there are no outstanding events, which is sufficient to deduce that routing has stabilized.

Real life validation

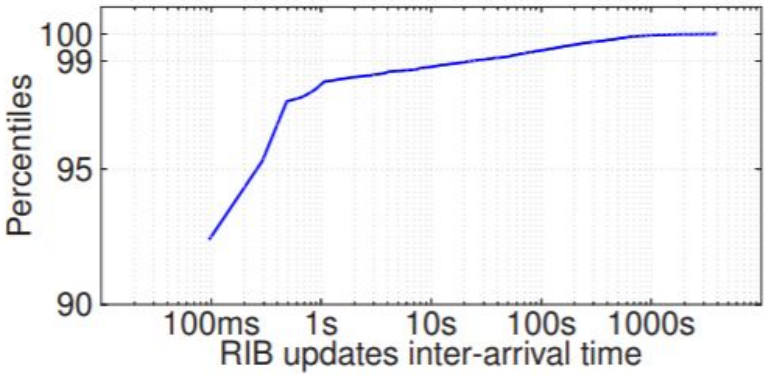


Figure 6: CDF of inter-arrival times of routing events from a large production data center. Routing events are very bursty: over 95% of events happen within 400ms of another event.

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.

After preparing stable snapshot we can begin the analysis.

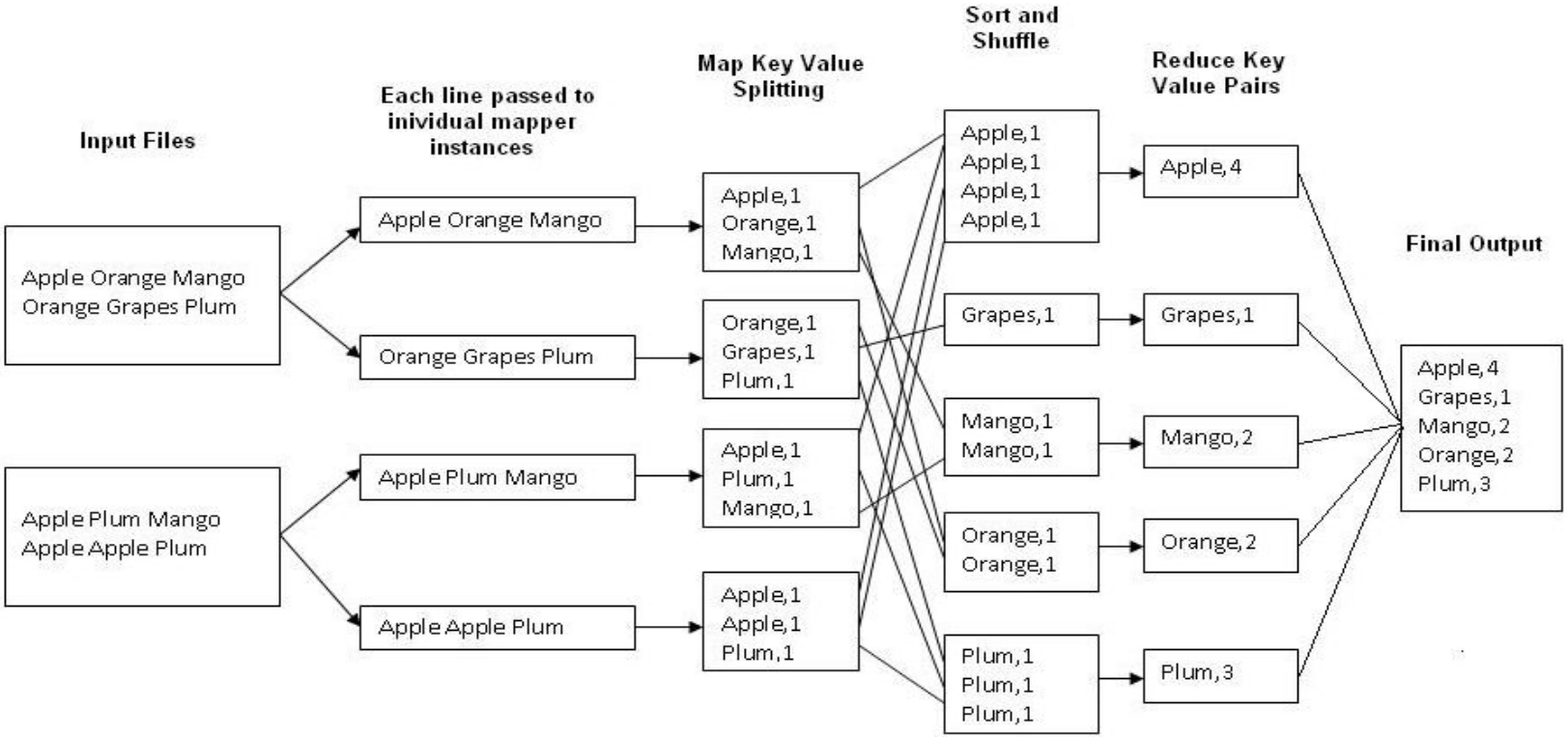
Since the goal is to check network with thousands of switches in minutes, some form of parallel computing has to be used.

Problem will be divided into smaller pieces and solved using MapReduce.

Outline of MapReduce

MapReduce 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 pairs. After the mapping phase, the MapReduce system shuffles outputs from different mappers by sorting by the key. After shuffling, each reducer receives a 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.

MapReduce example



The key idea behind Libra’s parallelism – build and analyze each sub network's graph separately.

Graph construction workflow

Dump snapshot into distributed file system as a list of routing rules.
Distribute rules to mappers, they will output list of pairs <subnet, rule> for subnets matching to the rule.
Reducers get all rules matching a given subnet and construct forwarding graph for this subnet.

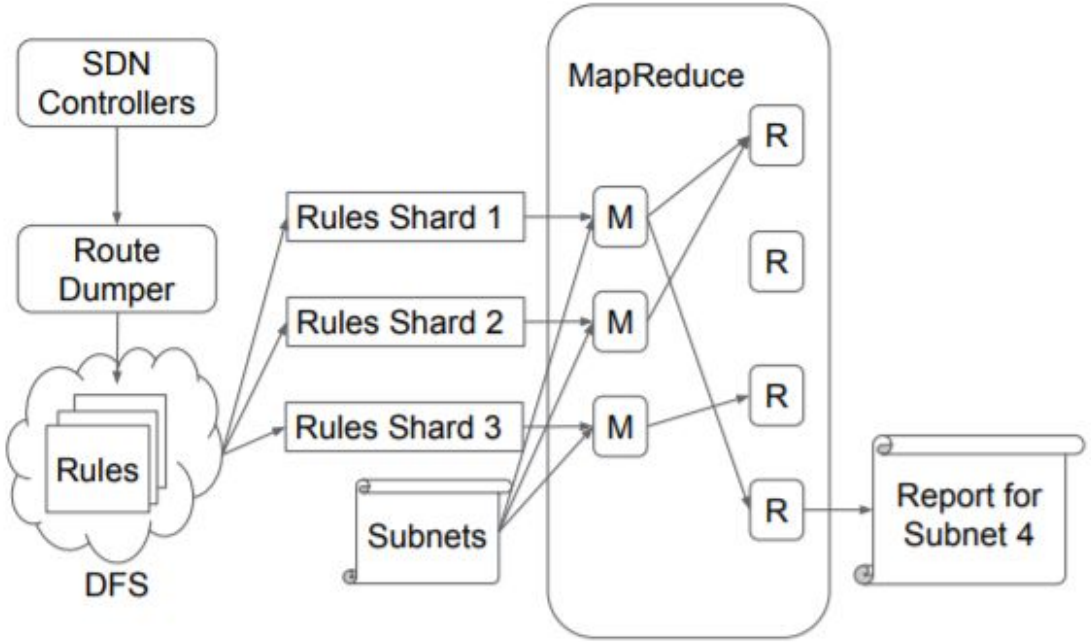


Figure 8: Libra workflow.

Mapper

Each of N mappers:

Is assigned 1/N of routing rules.

Builds prefix matching binary tree of all subnets in the network.

Reads one rule at the time and using it's ip prefix it finds a node in this tree corresponding to that prefix. All subnets in this node's subtree matches the rule's prefix.

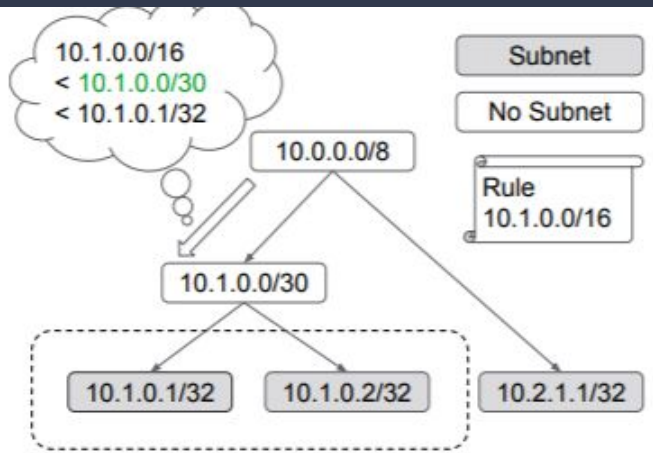


Figure 9: Find all matching subnets in the trie. `10.1.0.0/30` (X) is the smallest matching trie node bigger than the rule `10.1.0.0/16` (A). Hence, its children with subnets `10.1.0.1/32` and `10.1.0.2/32` match the rule.

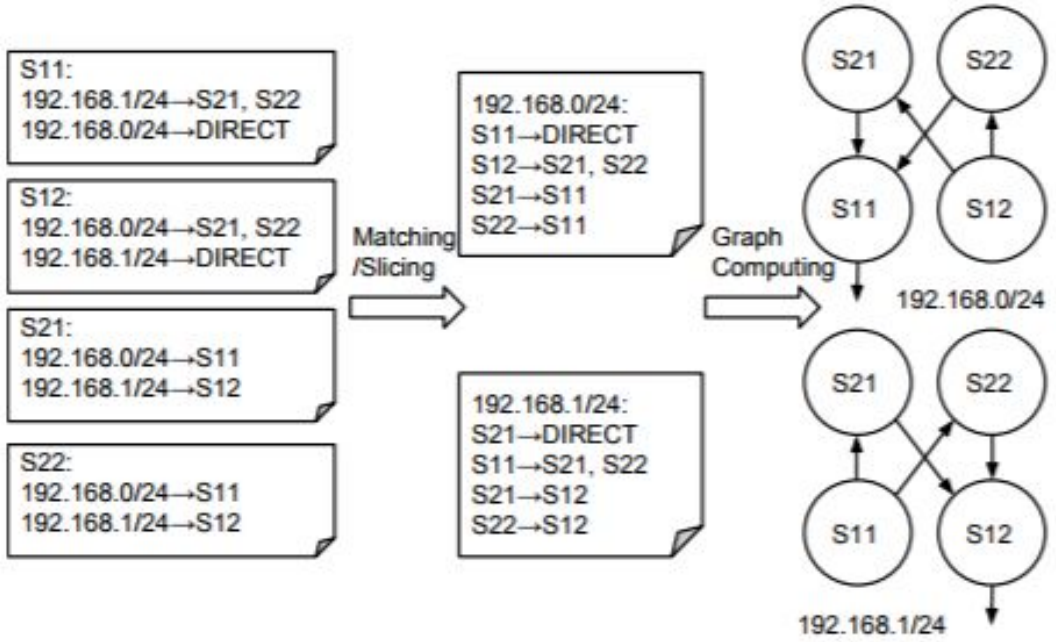
Outputs pairs <subnet, rule> for each subnet matched with a rule.

Reducer

Pairs produced by mappers are sorted by subnet.

One reducer receives all rules matched to one of the subnets.

Then, it constructs a graph describing how packets destined to this particular subnet are routed in the whole network.



Reducer

Using our subnets graphs we can check:

- Reachability: A reachability check ensures the subnet can be reached from any switch in the network.
- Loop detection.
- Black-holes: A switch is a black-hole for a subnet if the switch does not have a matching route entry for the subnet.

This can be done using some basic graph algorithms.

Libra performance

On single machine disk I/O is strongly limiting the performance.

On 50 machines cluster I/O bandwidth is much bigger - the mapping and reducing time dominates the total runtime.

Libra performance on computing cluster

DCN: Google's emulated data center network
DCN-G: 100 DCNs connected in a star topo
INET: 300 Internet routers with full BGP table

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.

	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.

Additional feature – Incremental Updates

Libra design allows reevaluation of the network after update without repeating the whole computation.

	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.

New subnets - Repeat the mapping stage, scan all rules in the network to find ones matching new subnets, then use one reducer for each new subnet. Works best with bigger batches of subnets (mapping takes around the same time for 1 and N new subnets).

Rules change - Repeat mapping only on new rules, reducers update already done graphs using new matched rules.

Limitations

- 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*.
- 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.

*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.

In conclusion

Libra was able to achieve its goal of analyzing big networks in minutes, but had to make a few assumptions about the network and use powerful, parallel machine.