SCMon: Leveraging Segment Routing to Improve Network Monitoring

François Aubry, David Lebrun, Stefano Vissicchio, Minh Thanh Khong, Yves Deville, Olivier Bonaventure
ICTEAM, Université catholique de Louvain, Louvain-la-Neuve, Belgium
Email: `firstname.lastname@uclouvain.be`

***Abstract*—To guarantee correct operation of their networks, operators have to promptly detect and diagnose data-plane issues, like broken interface cards or link failures. Networks are becoming more complex, with a growing number of Equal Cost MultiPath (ECMP) and link bundles. Hence, some data-plane problems (e.g. silent packet dropping at one router) can hardly be detected with control-plane protocols or simple monitoring tools like `ping` or `traceroute`.**

In this paper, we propose a new technique, called SCMon, that enables continuous monitoring of the data-plane, in order to track the health of all routers and links. SCMon leverages the recently proposed Segment Routing (SR) architecture to monitor the entire network with a single box (and no additional monitoring protocol). In particular, SCMon uses SR to (i) force monitoring probes to travel over cycles; and (ii) test parallel links and bundles at a per-link granularity. We present original algorithms to compute cycles that cover all network links with a limited number of SR segments. Further, we prototype and evaluate SCMon both with simulations and Linux-based emulations. Our experiments show that SCMon quickly detects and precisely pinpoints data-plane problems, with a limited overhead.

I. Introduction

Monitoring is a crucial task for network operation. It is needed to ensure that all resources operate correctly (e.g., no failures) and their configuration meets operator's expectations (no congestion, required quality of service, etc.). Effective monitoring is also fundamental for management tasks like traffic engineering, maintenance and troubleshooting.

Unfortunately, even basic monitoring tasks, like checking for hardware malfunctions, are practically hard, due to the complexity of current networks. Prominently, multi-path routing is widely used, both to spread the load on multiple paths and aggregate parallel links in bundles. Figure 1 shows an overview of the European backbone of a big cloud provider, OVH. It highlights that parallel links are used at the same time between many pairs of routers. While enabling better performance and robustness, multi-path routing also poses significant obstacles to monitoring. For instance, assessing the exact path and performance of each packet becomes complex [2], [21] since such a path depends on (vendor-specific) hash functions used by routers for load-balancing.

As a consequence, not only naive approaches (e.g., based on `ping` or `traceroute`) are not sufficient, but also state-of-the-art monitoring techniques tend to be ineffective.

Stefano Vissicchio is a postdoctoral researcher of the Belgian fund for scientific research (F.R.S.-FNRS)

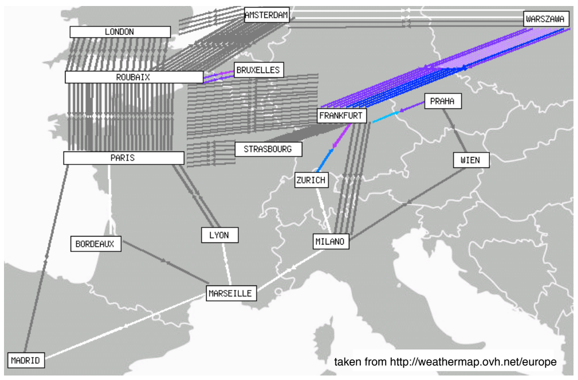

Fig. 1: European backbone of OVH. In contrast to prior techniques, SCMon can monitor health and performance of single links in bundles (e.g., between LONDON and ROUBAIX).

On the one hand, protocol-based approaches use control-plane messages to infer possible failures. For example, link-state routing protocols (like OSPF or IS-IS) or specialized ones (BFD [15]) rely on heartbeat-like mechanisms to check bi-directional connectivity among pairs of adjacent nodes. This approach only ensures detection of failures that affect control-plane messages. However, it cannot be used to detect failures that only affect data-plane traffic like: (i) corruption of an optical link that leads to framing errors and packet losses, (ii) malfunctioning of a router interface that considers the link still up but discards all the received packets, and (iii) failure of only one link among the parallel ones between two routers.

On the other hand, probe-based techniques rely on sending data-plane monitoring packets, i.e., *probes*, between fixed vantage points in the network. Vantage points typically run standard protocols (e.g., IPSLA [8]) to send probes and extract measurements from them. Unfortunately, if the probes are sent over paths used to forward regular traffic, many vantage points may be needed to obtain high coverage, and links not used by current paths (e.g., backup links) cannot be checked at all. Otherwise, probes can be sent over tunnels (e.g., RSVP-TE [3] ones) to enforce specific paths, but this is not scalable. Indeed, even for detecting single-link failures and pinpointing their position, the number of needed tunnels tends to explode, and so does the control-plane overhead (to signal tunnels) [7].

In this paper, we propose a new technique, called *SCMon*, that ensures full coverage of all network resources from a single vantage point. It is based on sending data-plane probes over carefully-chosen *cycles*. This way, a single box can both send and receive monitoring probes, avoiding the need for synchronizing and coordinating multiple vantage points, hence minimizing infrastructural costs. By relying on data-plane measurements, we support both detection of hardware failures and resource overloading (e.g., link congestion).

A key building block of our approach, used to send probes over cycles, is Segment Routing (SR) [12]. SR is a modern evolution of source routing which has recently attracted huge interest from both vendors and operators. In SR, node and adjacency segments can be added to a packet header: They instruct routers to forward the corresponding packet to intermediate nodes (node segments) or through intermediate links (adjacency segments) before sending it to its destination.

Despite its flexibility, SR also comes with challenges. First, SR paths natively use multi-path routing among intermediate nodes or links. For example, in Fig. 1, assume that a router in LONDON forces packets for FRANKFURT to cross PARIS with an SR segment. Before being forwarded to FRANKFURT, each of those packets is then sent on any shortest path from LONDON to PARIS, without any control on the link effectively crossed between LONDON and ROUBAIX. Moreover, SR generates conflicting objectives for the computation of cycles over which probes are sent. On the one hand, we would like to minimize the number of cycles, to reduce the monitoring overhead; hence, we should have cycles that are long and few in number. On the other hand, we need cycles to be enforced in practice, and long cycles may require too many segments even for the most powerful commercial routers (currently supporting less than 10 segments).

To tackle those challenges, SCMon runs original algorithms that compute cycles on a *monitoring topology*, maintained by routers in addition to the one used for user traffic. The monitoring topology spans all network links, and uses carefully-computed weights that avoid (as much as possible) ECMP paths, i.e., multiple shortest paths between a pair of nodes. Note that existing routers have already been shown to well support two topologies and their (limited) overhead [18].

SCMon supports detection and localization of *any* set of link failures/overloading. It infers single-link failures by keeping track of the cycles associated to probes sent and not received at the monitoring box. For multiple link failures, we have two cases. Some of them, e.g., affecting links in disjoint cycles, are detected directly from the set of lost probes (as single-link failures). The others, e.g., affecting two links belonging to a single monitoring cycle, are reported by SCMon one at the time: This provides operators with a debugging interface (asking to correct one error before showing another one) similar to the one of a compiler for software programs. Similar considerations apply to node failures and link congestion.

In developing SCMon, we make three main contributions.

The first complete formalization of SR, including node and adjacency segments, as needed for SCMon cycles. (§II)

Algorithms to compute both the monitoring topology and cycles. Our algorithms are parametric with respect to the number of segments supported by routers. (§III)

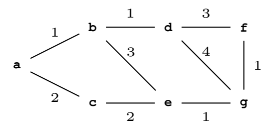

Fig. 2: Sample graph used to illustrate our examples.

Thorough evaluation of both the effectiveness of our algorithms and the practical ability of SCMon to quickly detect and troubleshoot data-plane problems. (§IV)

We also compare with prior work (§V) and conclude (§VI).

II. Segment Routing Model

In this section, we propose the first complete model of Segment Routing with both node and adjacency segments: We use this formalization as a basis for our algorithms. We start by discussing an example of how segment routing works. Then, we formalize our model and properties relevant for SCMon.

A. Segment Routing

Segment Routing is a recent routing architecture developed within the Internet Engineering Task Force. It enables to use non-shortest paths by specifying detours. Packets are forwarded through the shortest path from the source to the first detour, then to the second detour and so on. These detours are called segments and can refer to a network node or a link. In the case of a link (i.e. adjacency) segment, the shortest path to the upstream node is taken and then that link is crossed. In the case of Equal Cost Multipath (ECMP) between two segments, all the ECMP paths are used (for different packets).

Consider the network in Figure 2. If we specify the node e and the link (d, g) as segments from a to f, the path that the packets will follow is a shortest path from a to e, then a shortest path from e to d, then the link (d, g) and finally a shortest path from g to f. In this example, there are two shortest paths from a to e, namely (a, c, e) and (a, b, e), and a single shortest path from e to d and from g to f. Therefore, if a probe is sent using this list of segments, it will follow either the path (a, c, e, b, d, g, f) or the path (a, b, e, b, d, g, f), depending on which shortest path from a to e is taken.

Adjacency segments are more expensive than node segments as they require us to push more labels into the Segment Routing header. Therefore, we prefer to avoid them if possible.

B. Formalization

We model the network as a weighted directed graph $G = (V, E, w)$ where $w : E \to \mathbb{Z}^+$ is a weight function on E which corresponds to the IGP link costs. In a network where $(x, y) \in E$, we usually have $(y, x) \in E$, but sometimes $w(x, y) \neq w(y, x)$. In the figures, whenever $w(x, y) = w(y, x)$ we represent the edge (x, y) as an undirected edge. Given two paths $p_1 = (x_1, \ldots, x_n)$ and $p_2 = (x_n, \ldots, x_{n+m})$ we define the concatenation of p_1 and p_2 as $p_1 \oplus p_2 = (x_1, \ldots, x_n, x_{n+1}, \ldots, x_{n+m})$. Finally, given a path $p = (x_1, \ldots, x_n)$, we write *first*$(p) = x_1$ and *last*$(p) = x_n$.

Given a network G, we denote by $\mathcal{D}_x$ the shortest path DAG (directed acyclic graph), with path lengths measured according to the weights w, rooted at node $x \in V$. In other words, $\mathcal{D}_x$ is the sub-graph of G containing all the edges that belong to a shortest path starting at x. These DAGs can easily be computed using Dijkstra's algorithm by keeping track of all predecessors of the nodes upon shortest path computation.

We now give a formalization of Segment Routing on a network G. A segmentation of a path p is essentially a decomposition of p into a sequence of shortest paths or edges. For this reason, we define $Sp(G)$ to be the set of all shortest paths in G and the set of all adjacency segments $Adj(G) = \{\overrightarrow{(x,y)} \mid (x,y) \in E(G)\}$. Finally, we define the set $\mathcal{S}(G) = Sp(G) \cup Adj(G)$. The set $\mathcal{S}(G)$ represents the set of all possible segments. We use the arrow notation on the second set to distinguish them from shortest paths consisting of a single edge. This distinction is necessary because it may happen that an edge (x,y) is also the shortest shortest path between x and y. In this case the adjacency segment is represented by $\overrightarrow{(x,y)}$ and the shortest path by (x,y).

Definition 1. *A segmentation of a path $p = (x_1, \ldots, x_n)$ in a graph $G = (V,E)$ is a list $s_1, \ldots, s_k \in \mathcal{S}(G)$ such that $p = s_1 \oplus s_2 \oplus \ldots \oplus s_k$. We call k the length of the segmentation.*

As explained above, from the networking point of view, a segmentation is a list of nodes and edges that represents a list of detours. We now explain how to transform a segmentation into a list of detours. This is necessary because it is the data that will be added into the SR header.

If $p = s_1 \oplus s_2 \oplus \ldots \oplus s_k$ then we define

$$seg(s_i) = \begin{cases} last(s_i) & \text{if } s_i \in Sp(G) \\ s_i & \text{if } s_i \in Adj(G) \end{cases}$$

The *segment list* corresponding to $s_1 \oplus s_2 \oplus \ldots \oplus s_k$ is defined as $\langle seg(s_1), seg(s_2), \ldots, seg(s_k) \rangle$. In a segment list, nodes are called *node segments* and edges are called *adjacency segments*.

For example, in Figure 2, let $p = (a,c,e,b,d,g,f)$. A possible segmentation of p is $(a,c) \oplus (c,e) \oplus (e,b,d) \oplus \overrightarrow{(d,g)} \oplus (g,f)$. Its segment list is $\langle c, e, d, \overrightarrow{(d,g)}, f \rangle$. If we follow the shortest path from a to c, then the shortest path from c to e, then from e to d, then the edge (d,g) and finally the shortest path from g to f, we will go exactly through path p. Suppose now that, instead of this segmentation, we use the following (notice that there is no arrow on top of edge (d,g)): $(a,c) \oplus (c,e) \oplus (e,b,d) \oplus (d,g) \oplus (g,f)$. Its segment list is $\langle c, e, d, g, f \rangle$. If we proceed as before, following shortest paths between elements of this list, when we reach node d we have to go to g. But there are two shortest paths between these two nodes, namely, (d,g) and (d,f,g). Therefore, given only the segment list, we cannot recover the original path.

C. Properties for single-box monitoring

In general, many segmentations exist for a single path. Given a path p, an easy segmentation consists in assigning adjacency segments to every edge of the path. For example, $(a,b,d,f) = \overrightarrow{(a,b)} \oplus \overrightarrow{(a,b)} \oplus \overrightarrow{(b,d)} \oplus \overrightarrow{(d,f)}$. For SCMon to be practical, we focus on segmentations with specific properties.

Primarily, we consider **minimal** segmentations.

Definition 2. *A segmentation S of length k of a path p is said to be* minimal *iff any other segmentation of p has length at least k.*

Moreover, since adjacency segments are more expensive than node segments, we try to compute **simple** segmentations that contain only node segments.

Definition 3. *A segmentation S is said to be* simple *iff its segment list contains only node segments.*

Unfortunately, it is not always possible to compute simple segmentations. For example, a path may contain edges not belonging to any shortest path. The same applies to paths with an edge (x,y) that is one among multiple shortest paths between x and y (as for (d,g) in Figure 2).

Finally, to ease troubleshooting, we would like that every segmentation is **ECMP-free**, i.e., it is biunivocally associated with one segment list; otherwise, we have cannot know which path actually corresponds to the segmentation.

Definition 4. *A segmentation $p = s_1 \oplus s_2 \oplus \ldots \oplus s_k$ is said to be* ECMP-free *iff each s_i is either an element of $Adj(G)$ or the unique shortest path between its endpoints.*

III. Computing Probing Cycles

Our approach to compute monitoring cycles consists of four steps. The first steps computes IGP weights for the monitoring topology, in order to minimize ECMP paths. The second step models link bundles aggregated into a single IGP link. The third step calculates a set of cycles covering the whole network and sharing the node from which probes are sent. The last step computes the SR segments to send probes over those cycles.

A. Optimizing the Monitoring Topology

We propose an algorithm to compute IGP weights for the monitoring topology (only used for monitoring) so that

1) every link belongs to some shortest path,
2) there is as few ECMP paths as possible.

Let m denote the number of edges in the network and let $\mathcal{P}_m = \{p_1, p_2, \ldots, p_m\}$ be the first m prime numbers.

The key idea of the algorithm is to use the logarithm of these prime numbers as IGP weights. Indeed, these weights would guarantee that any two paths have a distinct cost, since $\ln(x) + \ln(y) = \ln(x \cdot y)$ and $\ln$ is injective. Thus, the cost of the first path is the logarithm of some product of prime numbers whereas the weight of the second path is the logarithm of the product of some other prime numbers. Since these products are distinct, their logarithms must also be distinct.

Since the IGP weights must be integers, we use a truncated logarithm $\overline{\ln}^s$, defined as follows:

$$\overline{\ln}^s(x) = \lfloor 10^s \cdot \ln(x) \rfloor$$

For instance, $\overline{\ln}^4(5) = \lfloor 10^4 \cdot 1.60943... \rfloor = 16094$.

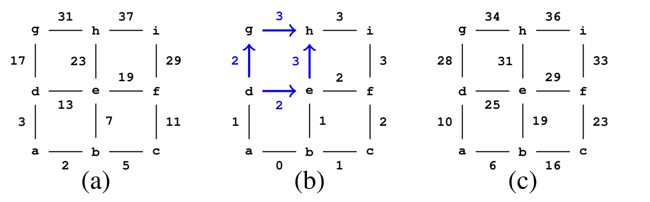

Fig. 3: First two iterations of the execution of the algorithm on a 3 by 3 grid.

In our algorithm, we start with $s = 0$ and increase s until there is no ECMP. The following proposition proves that the algorithm finds a solution in a finite number of iterations.

Proposition 1. *Let A and B be two distinct subsets of $\mathcal{P}_m$ and $P = \prod_{x \in \mathcal{P}_x} x$. If $s > \ln(m \cdot P)$, $\sum_{x\in A} \overline{\ln}^s(x) \neq \sum_{x \in B} \overline{\ln}^s(x)$.*

Proof. Given any s and $X \subseteq \mathcal{P}_m$, we have

$$10^s \cdot \ln\left(\prod_{x\in X} x\right) \geq \sum_{x \in X} \overline{\ln}^s(x) \geq 10^s \cdot \ln\left(\prod_{x \in X} x\right) - |X| \cdot \quad (1)$$

Write $a = \prod_{x\in A} x$ and $b = \prod_{x \in B} x$. Assume without loss of generality that $a > b$ (they are not equal because they are the product of distinct prime numbers). Then by (1)

$$\sum_{x\in A} \overline{\ln}^s(x) - \sum_{x \in B} \overline{\ln}^s(x) \geq 10^s \cdot \ln(a) - |A| - 10^s \cdot \ln(b) =$$
$$10^s \cdot \ln(a/b) - |A| \geq 10^s \cdot \ln(a/(a-1)) - |A| \geq$$
$$10^s/a - |A| \geq 10^s/P - m.$$

Which is positive as long as $s > \log_{10}(m \cdot P)$. □

Further, we ensure that each edge belongs to a shortest path by adding a large enough constant to each edge. Let w be the weights obtained from the iterative procedure above, and C be any constant that is bigger than the diameter of the graph. Denote by w_C the cost resulting from adding C to the cost of every edge, that is $w_C(e) = w(e) + C$. Then for any given edge $e = (v, u)$ we have $w_C(e) = C + w(e)$. If P is any path other than e from v to u then

$$w_C(P) = C \cdot |E(p)| + w(P) \geq 2 \cdot C \geq C + w(e)$$

since $C > w(e)$. Thus e is the shortest path from v to u.

Also, adding this constant does not create additional ECMP. If P_1 and P_2 are two paths with $w(P_1) \neq w(P_2)$ then

$$w_C(P_1) - w_C(P_2) = C(|E(P_1)| - |E(P_2)|) + w(P_1) - w(P_2)$$

If $|E(P_1)| = |E(P_2)|$ then $w_C(P_1) - w_C(P_2) = w(P_1) - w(P_2) > 0$ we defintion of w. Otherwise, we can assume that $|E(P_1)| > |E(P_2)|$ and we have $w_C(P_1) - w_C(P_2) \geq C - w(P_2) > 0$ since C is larger than the diameter of G.

Thus, our algorithm to compute the IGP weights for the monitoring topology consists in (i) iterating over s until no two shortest paths have the same weight and (ii) adding the value of the diameter plus one to the cost of each edge.

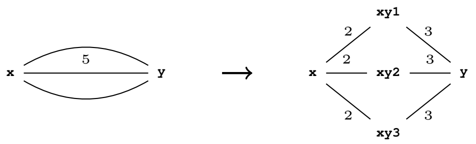

Fig. 4: Link bundle transformation.

Figure 3 shows the execution of the algorithm on a 3×3 grid. Figure 3(a) displays the initial prime numbers associated to each edge, and Figure 3(b) conveys the result of computing $\overline{\ln}^0$ on these numbers. Note that there is still ECMP (in blue), so we iterate. Figure 3(c) reports the result of the second iteration, using $\overline{\ln}^1$: There is no more ECMP. To ensure that each edge belongs to a shortest path, we then add the value of the diameter plus one (154) to each edge.

In practice, we cannot configure weights that are larger than 65535, so we stop our algorithm before this threshold is reached. This may leave some ECMP, but our experiments show that the remaining ECMP is likely zero or negligible in few corner cases. We ran the algorithm on all 243 connected topologies in the Topology Zoo [16] and the Rocketfuel [23] topologies. The average number of iterations (value of s) was 1.06 with maximum value 5. Only two instances, namely `Kdl` from the Topology Zoo and `RF1239` from the Rocketfuel topologies, still had ECMP after the execution of our algorithm. In those topologies, only 0.4% and 0.06% of source-destination pairs had ECMP, respectively.

To implement this algorithm, we need to be able to find the first m prime numbers. We can find these by iterating over the integers $2, 3, 5, 7, 9, \ldots$ and using any primality test algorithm to check which ones are prime numbers. The Prime number theorem [9] tells us that the m-th prime number is close to $m \ln(m)$ so we find them in a small number of steps.

B. Dealing with link bundles

Several physical cables between the same pair of nodes can be aggregated into a single logical (IGP) link. In those cases, we cannot avoid multi-path routing, not even setting appropriate IGP weights in the monitoring topology. However, in order to monitor each physical cable, we need to explicitly represent all cables in a bundle and the corresponding multi-path routing. We therefore derive a graph model of the monitoring topology. We start from the IGP graph. Then, for each link between u to v with several physical cables, we remove (u, v) from our model; Moreover, for every physical cable in (u, v), we add one fake node f and two edges (u, f) and (f, v) and set their costs so that their sum is equal to the IGP cost $w(u, v)$. An example of this transformation is shown in Figure 4. Note that fake links intuitively translated into the need for using adjacency segments in our monitoring probes.

C. Computing the Cycle Cover

From the graph model built as described in the previous section, we finally compute a cycle cover of the network. Our algorithm is based on the two following observations:

1) A shortest path can be represented by a single segment, provided that there is no ECMP.
2) The set of shortest paths from one node to all other nodes in a graph forms a DAG. The longest paths in DAGs can be computed in linear time [22].

The algorithm takes as input the weighted graph representing the monitoring topology and a parameter k specifying the maximum number of segments that can be used for each cycle.

To find a cycle cover, we start by assigning a binary variable u_e to each edge e such that it has value 1, if the edge is not yet covered by any cycle, and 0 otherwise. Then we call the *cycle-finding algorithm* (described next) until all edges are covered, that is, while $\sum_{e \in E} u_e > 0$. Every time we find a cycle we set $u_e = 0$ for each edge of that cycle.

The cycle-finding algorithm is outlined in Figure 5. Starting from a source node s, it computes the longest path $\bar{P}$ (with respect to the number of uncovered edges) in $\mathcal{D}_s$. After finding $\bar{P}$, it then computes the longest path starting from the last node of $\bar{P}$, and iterates on the latter found path for a given number of iterations (depending on k). To avoid repeated edges in the built cycle, the algorithm also keeps track of the edges traversed in previous iterations, and discards paths (i) crossing already-traversed edges, or (ii) terminating in a node with no path to s disjoint from the already-traversed edges.

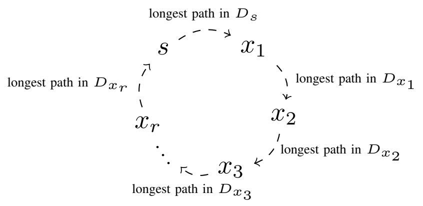
longest path in D_s
s
x_1
longest path in D_{x_1}
x_2
longest path in D_{x_2}
x_3
longest path in D_{x_3}
x_r
longest path in D_{x_r}

Fig. 5: Cycle construction outline

The pseudocode of the cycle-finding algorithm is given in Algorithm 1. The input is the graph G of the monitoring topology, the shortest path DAG D computed on G, the node $source$ from which probes will be sent and the segment budget k. We maintain the current node (corresponding to the x_i's in Figure 5), the set of forbidden edges $\mathcal{F}$ to avoid repeating edges, the binary variables u indicating whether an edge has already been covered and the cycle itself. We use an algorithm DAGLongestPath that receives as input a shortest path DAG, a starting node x, the variables u to use as path costs, the remaining segment budget and the forbidden edges. It outputs the set of all the longest paths from the starting node to the other nodes in the graph. For each of those paths, it provides a tuple ($\textit{ending node}, \textit{path}, \textit{number of segments}$). Paths are sorted by non-increasing number of uncovered edges. In the case of ties, they are broken by the number of uncovered edges in the DAG of the ending node (the more the better). This ensures that if all the edges of a DAG are covered we go to a node that is adjacent to an uncovered edge. From this set of solutions, we select the best one that has a path to the source with the remaining segment budget (lines 10-18). Finally, we close the cycle by finding a path from the last node to the starting node (if they differ). The last step is always possible since we only choose nodes that have a path to s as x_i nodes.

Algorithm 1 FindCycle$(G, \mathcal{D}, \textit{source}, k)$

```
1:  $cur \leftarrow source$
2:  $\mathcal{F} \leftarrow \emptyset$
3:  $u_e \leftarrow 1$ **forall** $e \in E(G)$
4:  $cycle \leftarrow null$
5:  $S_1 \leftarrow$ DAGLongestPath$(\mathcal{D}, cur, u, k, \mathcal{F})$
6:  **while** $|S_1| > 0$ **do**
7:      $(next, path, nseg) \leftarrow$ (**null**, **null**, **null**)
8:      **while** $|S_1| > 0$ **and** $next =$ **null do**
9:          $(x, p, s) \leftarrow S_1$.RemoveBest()
10:         $S_2 \leftarrow$ DAGLongestPath$(\mathcal{D}, x, u, k - s, \mathcal{F} \cup E(p))$
11:         **if** $|S_2| > 0$ **then**
12:             $(next, path, nseg) \leftarrow (x, p, s)$
13:     **if** $next \neq$ **null then**
14:         $k \leftarrow k - nseg$
15:         $\mathcal{F} \leftarrow \mathcal{F} \cup E(path)$
16:         $cur \leftarrow next$
17:         $u_e \leftarrow 0$ **forall** $e \in E(path)$
18:         $cycle \leftarrow cycle$.Append$(path)$
19:         $S_1 \leftarrow$ DAGLongestPath$(\mathcal{D}_{cur}, cur, u, k, \mathcal{F})$
20: **if** $cycle$.Last() $\neq source$ **then**
21:     $S_1 \leftarrow$ DAGLongestPath$(\mathcal{D}_{cur}, cur, u, k, \mathcal{F})$
22:     $(x, p, s) \leftarrow S_1$.RemoveBest()
23:     $cycle \leftarrow cycle$.Append$(path)$
24: **return** $cycle$
```

The algorithm DAGLongestPath works in a similar fashion as the standard dynamic programming algorithm for computing longest paths in directed acyclic graphs [22]. We extended it to also compute the number of segments needed in the path. It does so by using the shortest path DAGs given as input to check for ECMP. For all ECMP cases, it decreases the current segment budget (since a new segment is needed). The time complexity of DAGLongestPath is $O(E \cdot k)$.

D. Segmenting Cycles

Once we have a cycle cover of the graph, we need to be able to represent them as a list of segments. Our *segmenting algorithm* computes ECMP-free segmentations so that (i) it uses adjacency segments only when any other segmentation would also use one, and (ii) it minimizes the remaining node segments. Those two properties imply that if the input path admits a simple segmentation, our algorithm will produce a minimal simple segmentation.

The pseudo-code of the algorithm is reported in Algorithm 2. It receives as input a path p and a shortest path DAG D and outputs an ECMP-free segmentation of p. Its complexity is linear with respect to the length of p.

We now show that the algorithm produces a valid segmentation.

Lemma 1. *Let seg_i and seg_{i+1} be two consecutive elements in the segment list produced by Algorithm 2. Suppose that seg_i and seg_{i+1} are both node segments. Then there is a unique shortest path from seg_i to seg_{i+1}.*

Proof. Suppose that there exist two shortest paths from seg_i to seg_{i+1}. Then the in-degree of seg_{i+1} in the shortest path

Algorithm 2 MinSegECMP($(x_1, x_2, \ldots, x_n), \mathcal{D}$)

```
$r \leftarrow 1$
$S \leftarrow \langle\rangle$
for $i \leftarrow 1$ to $n - 1$ do
    if $(x_i, x_{i+1}) \notin E(D_r)$ then
        if $d^-_{D_{x_i}}(x_{i+1}) = 1$ then
            $S \leftarrow S + x_i$
            $r \leftarrow x_i$
        else
            $S \leftarrow S + \langle x_i, (x_i, x_{i+1})\rangle$
            $r \leftarrow x_{i+1}$
    else if $d^-_{D_r}(x_{i+1}) > 1$ then
        if $d^-_{D_{x_i}}(x_{i+1}) > 1$ then
            $S \leftarrow S + \langle x_i, (x_i, x_{i+1})\rangle$
            $r \leftarrow x_{i+1}$
        else
            $S \leftarrow S + x_i$
            $r \leftarrow x_i$
return $S$
```

DAG rooted at *seg*$_i$ is larger than one. But, in this case, the algorithm would have either produced an adjacency segment on the edge of the input path ending at *seg*$_{i+1}$ (on line 19) or a node segment at the last node of p before node *seg*$_{i+1}$ (on line 22). In either case, this contradicts the fact that *seg*$_{i+1}$ is the element after *seg*$_i$ in the segment list. □

When *seg*$_i$ is an adjacency segment and *seg*$_{i+1}$ is a node segment, a similar argument applies. In the other cases there is nothing to show. This gives the following corollary.

Corollary 1. *The segmentations produced by Algorithm 2 are ECMP-free.*

Now we establish that some adjacency segments must be in any ECMP-free segmentation and that those are the only adjacency segments present in the segmentation produced by Algorithm 2.

Lemma 2. *Let $p = (x_1, x_2, \ldots, x_n)$ be a path on a graph G. If the segment list of p produced by Algorithm 2 possesses an adjacency segment (x_i, x_{i+1}) then the segment list of any ECMP-free segmentation of p possesses this adjacency segment.*

Proof. Algorithm 2 produces adjacency segments in two cases: the edge (x_i, x_{i+1}) does not belong to any shortest path DAG or there exists a shortest path from x_i to x_{i+1} different than (x_i, x_{i+1}). It is obvious that any segmentation contains adjacency segments on all edges that do not belong to any shortest path since it is the only way we can cross them. In the latter case, if a segmentation passes through the edge via a shortest path p, then p cannot be a unique shortest path since its last edge is (x_i, x_{i+1}). We can get another one by removing the node x_{i+1} from p and concatenating the result with the other shortest path from x_i to x_{i+1}. □

The next lemma shows that Algorithm 2 minimizes the remaining node segments.

Lemma 3. *Let $p = (x_1, x_2, \ldots, x_n)$ be a path in a graph G and let $S = s_1 \oplus s_2 \oplus \ldots \oplus s_k$ be the segmentation corresponding to the segment list produced by Algorithm 2. Let S' be any ECMP-free segmentation of p having the exact same set of adjacency segments. Then, for each i such that $s_i \in Sp(G)$, there exists a node in s_i that is a node segment in the segment list of S'.*

Proof. Suppose that no node in s_i is a node segment on the segment list of S'. Then, since both segmentations have the same adjacency segments, no edge of s_i is an adjacency segment in the segment list of S'. Therefore, there must be a shortest path s' starting before (or at) *first*(s_i) in S'. This contradicts the fact that s_i was produced by Algorithm 2 in the first place because when *last*(s_i) was added to the segment list, the condition on line 4 must have been false meaning that the edge is not a shortest path edge in the shortest path DAG of *first*(s_i). Thus that edge cannot also be a shortest path edge in the *first*(s'), since the node does not come after *first*(s_i). This contradicts the fact that s' is a shortest path. □

A corollary of Lemma 3 is the following proposition, stating that Algorithm 2 uses the strictly-minimum number of adjacency segments and also minimizes node segments.

Proposition 2. *Algorithm 2 computes an ECMP-free segmentation of the input path such that: (1) it contains an adjacency segment if and only if all segmentations of the input path contain that adjacency segment; (2) the number of node segments used is minimal.*

IV. Evaluation

We implemented the SCMon algorithms in approximately 1,000 lines of Java code. To evaluate their performance, we focus on realistic topologies. We provide simulation results that show the number of cycles and segments that are required to cover various topologies. In addition, we also implement SCMon in `python` and evaluate its performance on emulated networks. We publicly released both the code for our algorithms and our prototype implementation of SCMon at `http://inl.info.ucl.ac.be/softwares/scmon`.

A. Network topologies

We experiment with two real topologies, a large ISP and the European backbone of OVH [1], plus all the realistic topologies included in the Rocketfuel [23] and in the Internet Topology Zoo [16] datasets. For brevity, we mainly focus on results for the Rocketfuel and OVH topologies, and provide aggregated results for the 243 connected topologies in the Internet Topology Zoo. The OVH topology is especially interesting since it contains many link bundles that are difficult to test with current monitoring techniques (see Figure 1). In all our experiments, we compute IGP weights with the algorithm described in Section III-A and select the monitoring box as the center of the graph with respect to those weights.

[1] Available at http://weathermap.ovh.net

Topology	$\lvert V\rvert$	$\lvert E\rvert$
OVH Europe	≈ 57	≈ 216
RF AS1239	153	1010
RF AS1755	67	248
RF AS3257	103	484
RF AS3967	57	208

TABLE I: Topologies used for experiments.

B. Probing cycles

The effectiveness of our cycle-cover algorithm is related to the number of cycles used to cover the input topology. Indeed, it aims at minimizing such number, in order to reduce the monitoring overhead. However, as discussed in Section III, this number depends on the maximum number of admited segments (parameter k in Algorithm 1).

Table II shows the number of cycles found by Algorithm 1 in function of k. With the exception of the OVH topology, we find solutions for all values of $k = 3, \dots, 11$ (we limited k to be 11 at most, since this is the maximum number of segments supported by current commercial routers). Indeed, in the absence of multi-path routing, any edge can be covered by at most 3 segments: the first one is a node segment used to reach the node connected to the edge we would like to cover, the second one is an adjacency segment used to traverse the edge, and the third one is a node segment used to come back to the vantage point. The OVH network is a special case because it contains a lot of parallel links, which force some multi-path routing. We also observe that the number of cycles quickly decreases with the increase of k in our experiments.

Topology	3	4	5	6	7	8	9	10	11
OVH Europe	-	-	-	-	-	87	72	66	56
RF AS1239	580	295	195	145	116	98	84	74	65
RF AS1755	98	53	34	26	23	18	15	14	12
RF AS3257	217	110	76	55	44	38	33	29	25
RF AS3967	65	35	24	18	15	13	11	10	9

TABLE II: Number of cycles in function of k for IGP weights that are ECMP-free. The first row shows the value of parameter k and then each row shows the number of cycles in the cover for each k.

To analyse the performance of our algorithm in a larger number of topologies, we now consider the 243 network topologies collected by the Internet topology Zoo project.

In Figure 6, we provide a CDF of the number of cycles that are required to cover each of these 243 topologies. Since these networks have different numbers of nodes and links, the x-axis represents the ratio between the number of cycles in the cover and the total number of links in the topology. For instance, with $k = 3$, we see that some topologies have a very bad ratio, meaning that we need a number of cycles that is close to the number of edges. For $k = 4, 5$, more than 80% of the instances have a cover that has a 20% ratio. When we increase k to 6 or 7, we obtain cycle covers with 10% ratios for more than 80% of the topologies. In other words, we find cycle covers that are small compared to the number of edges for all values of k higher than 3.

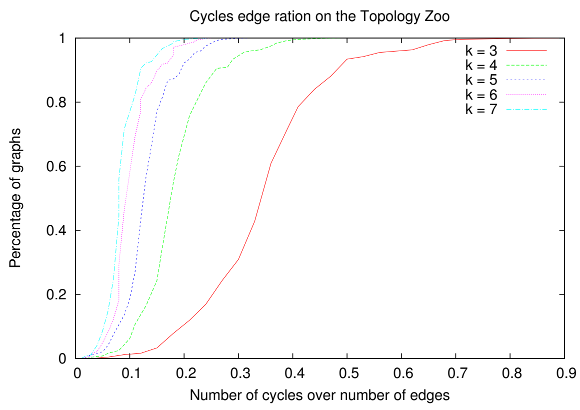

Fig. 6: Percentage of networks from the Internet topology zoo that are covered with a given ratio of the number of cycles over the number of edges for $k = 3, \dots, 7$.

C. Practical Evaluation

To evaluate the practical deployability of SCMon, we implemented a prototype in `python` and tested it in emulated networks running on Linux servers.

We emulate the RocketFuel and the OVH Europe topologies. We rely on two software components, (i) a framework that emulates large topologies and (ii) an implementation of SCMon that sends and receives probes over the computed cycles. Our framework is Linux-based and uses named namespaces and virtual Ethernet interfaces to simulate network nodes/links, IGP costs and link delays. This framework is similar to Mininet [13] with the difference that our implementation uses named namespaces instead of anonymous ones, allowing easier debugging. For Segment Routing, our prototype leverages the open-source implementation in the Linux kernel[2]. A similar implementation could be developed for the MPLS-based implementation in ONOS [10].

Our SCMon implementation (also denoted as SCMon in this section) takes as input the list of cycles computed by Algorithm 1 for the emulated topology and periodically sends UDP probes over each cycle. SCMon probes each cycle according to the Finite State Machine shown in Fig. 7. We provide more details on its behavior below.

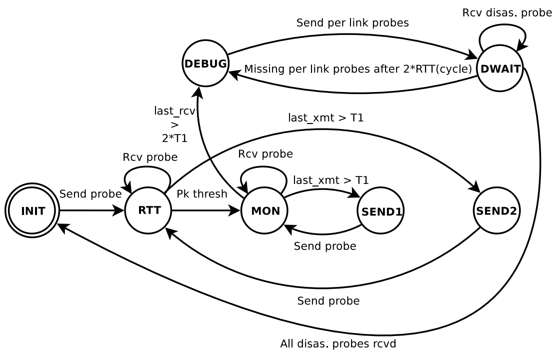

Fig. 7: Cycle state diagram in SCMon

[2]See http://www.segment-routing.org

In our implementation, we use the parameter T_1 to set the delay between successive probes. This parameter is bounded by the maximum rate at which SCMon is able to send probes. In our prototype implementation, we measured that our `python` code was able to send at most one probe every 0.2 milliseconds, i.e., 5,000 probes per second. We thus define T_1 as the number of cycles in a topology times 0.2 milliseconds, plus a 25% (added as a safety margin).

When SCMon starts, it enters an initial state (`INIT`) which calibrates the RTT of each cycle. SCMon sends and receives probes over each cycle at the rate defined by T_1 and does not consider any late arrival as a cycle timeout. The number of calibrating probes is defined by a configuration variable P_k. We set $P_k = 10$ since it proved large enough to account for unexpected jitter in our experiments. Once SCMon has received P_k probes, it considers that the cycle is up and enters the actual monitoring state (`MON`). For a given cycle, if a probe is not received within $2 * T_1$ milliseconds, then the cycle is considered as timed out and SCMon enters a debugging state for this cycle (`DEBUG`). In this state, SCMon sends one probe for each segment composing the cycle to determine the faulty one (for our prototype, we make the assumption of a single failure). If SCMon does not receive all the debugging probes within $2 * RTT_{cycle}$ milliseconds, it outputs the first segment that has timed out and starts the debugging state over again. If all the debugging probes are received within $2 * RTT_{cycle}$ milliseconds, the cycle is considered back up and SCMon re-enters the initial state for this cycle. Re-entering the initial state, and thus temporarily not reacting to timeout events allows the cycle RTT to get smoother if the detected fault was caused by a temporary jitter in the cycle RTT.

Our experiments then consist in simulating link failures, and evaluate the effectiveness of SCMon to detect and correctly locate them. For simplicity, we focus on single failures.

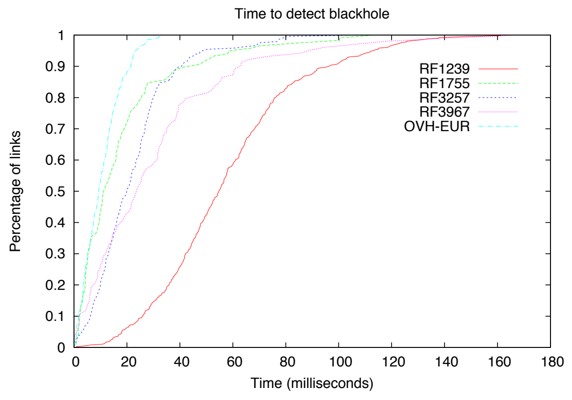

Fig. 8: Blackhole detection time for each link. The cycles were computed with $k = 5$ for all RocketFuel topologies and with $k = 8$ for the OVH topology.

In Fig. 8, we plot the time needed to detect a blackhole as cumulative distribution functions for each edge. The figure clearly shows that most of the blackholes are detected within less than 100 milliseconds. Table III and Table IV show for each topology, the blackhole detection time for several percentiles of the links, and the average and maximum RTT of the cycles. We see that the detection time is correlated mainly with the delay of the cycles in a topology, and with the number of those cycles. The RF1239 topology has a large number of cycles and large cycle RTTs, yielding a higher blackhole detection time. The topologies RF1755 and RF3257 have very similar cycle delays but the latter has a higher number of cycles, yielding a slightly higher detection time than RF1755. The RF3967 topology has a small number of cycles but very large cycle delays, yielding a higher detection time than topologies RF1755 and RF3257. The OVH topology has a relatively large number of cycles but very small cycle delays (due to missing link delays of the OVH topology, we set the delay of all the links at one millisecond), yielding the fastest detection time of all tested topologies.

Topology	20th	40th	60th	80th	90th
RF1239	35 ms	48 ms	61 ms	76 ms	96 ms
RF1755	4 ms	10 ms	16 ms	26 ms	47 ms
RF3257	10 ms	16 ms	24 ms	30 ms	40 ms
RF3967	8 ms	19 ms	31 ms	46 ms	63 ms
OVH-EUR	3 ms	7 ms	12 ms	16 ms	21 ms

TABLE III: This table shows, for each topology, the time needed to detect a blackhole at the 20th, 40th, 60th, 80th and 90th percentiles of the links

Topology	Cycles	Avg RTT	Max RTT
RF1239	195	83 ms	360 ms
RF1755	34	49 ms	130 ms
RF3257	76	48 ms	127 ms
RF3967	24	109 ms	206 ms
OVH-EUR	87	18 ms	28 ms

TABLE IV: This table shows, for each topology, the number of cycles, the average RTT of a cycle and the maximum RTT of the cycles

V. Related Work

There is a huge literature about monitoring and fault detection, including pioneering work published almost three decades ago [27]. Previous work typically start from the consideration that queries to devices (e.g., through SNMP) cannot always be trusted [7], and analyses of control-plane messages (e.g., OSPF or IS-IS hello packets) do not provide enough information on data-plane performance. These limitations are also faced by most commercial products (e.g., Tivoli NetView) that aggregate basic tools, from IP SLA to SNMP traps and Syslog collection into a common framework.

Many prior works (see, e.g., [28]) focus on data correlation and statistical techniques for detecting faults and service disruptions. For example, [17] studied how to detect silent (hardware) failures with active measurements and their post-elaboration using a greedy heuristic. Similarly, many contributions have been made in the area of network tomography where topology and link performance are inferred from end-to-end measurements (e.g., [11]). However, all prior work overlooked the impact of multi-path routing, that can make failures much harder to detect and troubleshoot. SCMon tackles those scenarios using an additional topology and segment routing to avoid multi-path routing for monitoring probes.

In addition, previous contributions typically assumed multiple vantage points, and tried to optimize their position in order to minimize their number while guaranteeing high network coverage (e.g., [20]). The presence of multiple vantage points is costly and requires coordination (time synchronization, probe identification and so on). An exception is represented by [6], which is based on a single monitoring point. However, that methodology needs unreliable tools like SNMP or Netflow to collect information on the traversed routers. SCMon effectively used a single monitoring box both to send probes (over cycles) and extract measurements from them.

The approach closest in spirit to SCMon is [7], where monitoring paths are source-routed thanks to either explicit tunnels (i.e., RSVP-TE) or static routes. However, [7] can explore only layer3 paths (hence, failures on aggregated link bundles are impossible to detect) and tends to create a lot of state (especially if RSVP-TE is used). SCMon avoids those limitations: It relies on Segment Routing that requires no state on the routers and can pinpoint a layer2 failure. In addition, SCMon improves debugging time from order of minutes (as taken by [7]) to milliseconds (see §IV).

Segment Routing has been used by recent works (e.g., [5], [1], [14]) on traffic engineering. To the best of our knowledge, this is the first paper about monitoring with Segment Routing. Moreover, in contrast to the mentioned previous works (which only considered node segments), we presented the first complete model for Segment Routing.

Finally, previous works on cycle covers mainly focused on covering undirected graphs with a minimal number of edges (MCCP cover problem) [24]. For MCCP, a heuristic algorithm based on the Chinese postman problem is provided in [19]. Moreover, a polynomial time algorithm to achieve small covers is given in [4]. In SCMon, we cannot reuse those algorithms because network topologies are directed graphs. Moreover, we have additional constraints due to the maximum number of SR segments that can used for implementing every cycle.

VI. Conclusion

In this paper, we presented SCMon, a new monitoring technique which relies on Segment Routing to send probes over cycles. SCMon allows any single box to effectively monitor all the network resources, including single links in bundles. We described algorithms to compute probe-traversed cycles inducing a limited overhead and the corresponding SR configurations for the probes. Further, we implemented an SCMon prototype and evaluated its performance on publicly-available topologies and emulated networks. Our experiments show that SCMon works well in practice: By using a limited number of cycles, it takes milliseconds to pinpoint the location of failures, like packets silently discarded by router hardware, that cannot be detected by existing techniques.

In future work, we plan to deploy SCMon in real networks, and investigate possibilities (e.g., using Fibbing [26], [25]) to program ECMP-free paths while avoiding a dedicated monitoring topology.

Acknowledgements

This work is partially supported by the ARC grant 13/18-054 (ARC-SDN) from Communauté française de Belgique.

References

[1] F. Aubry, D. Lebrun, Y. Deville, and O. Bonaventure. Traffic duplication through segmentable disjoint paths. In *IFIP Networking*, 2015.
[2] B. Augustin, T. Friedman, and R. Teixeira. Measuring Load-balanced Paths in the Internet. In *IMC*, pages 149–160, 2007.
[3] D. Awduche, L. Berger, D. Gan, T. Li, V. Srinivasan, and G. Swallow. RSVP-TE: Extensions to RSVP for LSP Tunnels. RFC 3209, 2001.
[4] J. C. Bermond, B. Jackson, and F. Jaeger. Shortest coverings of graphs with cycles. *Journal of Combinatorial Theory*, 35(3):297 – 308, 1983.
[5] R. Bhatia, F. Hao, M. Kodialam, and T. Lakshman. Optimized Network Traffic Engineering using Segment Routing. In *INFOCOM*, 2015.
[6] Y. Breitbart, C.-Y. Chan, M. Garofalakis, R. Rastogi, and A. Silberschatz. Efficiently monitoring bandwidth and latency in IP networks. In *INFOCOM*, 2001.
[7] R. Cartlidge and N. Guilbaud. Topology Aware Blackbox Monitoring. NANOG presentation, 2013.
[8] M. Chiba, A. Clemm, S. Medley, J. Saloway, S. Thombare, and E. Yedavalli. Cisco service-level assurance protocol. RFC 6812, 2013.
[9] T. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. McGraw-Hill, 2001.
[10] S. Das. Segment routing in onos. https://wiki.onosproject.org/display/ONOS10/Segment+Routing.
[11] N. Duffield, F. Lo Presti, V. Paxson, and D. Towsley. Network loss tomography using striped unicast probes. *IEEE/ACM Transactions on Networking*, 14(4):697–710, Aug 2006.
[12] C. Filsfils et al. Segment Routing Architecture. Internet draft, 2014.
[13] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, , and N. McKeown. Reproducible network experiments using container-based emulation. In *CoNEXT*, 2012.
[14] R. Hartert, S. Vissicchio, P. Schaus, O. Bonaventure, C. Filsfils, T. Telkamp, and P. Francois. A Declarative and Expressive Approach to Control Forwarding Paths in Carrier-Grade Networks. In *SIGCOMM*, 2015.
[15] D. Katz and D. Ward. Bidirectional forwarding detection (bfd). RFC 5880, 2010.
[16] S. Knight, H. Nguyen, N. Falkner, R. Bowden, and M. Roughan. The Internet Topology Zoo. *IEEE J. Sel. Areas Commun.*, 29(9):1765–1775, Oct 2011.
[17] R. R. Kompella, J. Yates, A. Greenberg, and A. C. Snoeren. Detection and Localization of Network Black Holes. In *INFOCOM*, 2007.
[18] A. Kvalbein et al. Fast IP Network Recovery Using Multiple Routing Configurations. In *INFOCOM*, 2006.
[19] M. Labbe, G. Laporte, and P. Soriano. Covering a graph with cycles. *Computers & Operations Research*, 25(6):499 – 504, 1998.
[20] L. Ma, T. He, A. Swami, D. Towsley, and K. K. Leung. On optimal monitor placement for localizing node failures via network tomography. *Performance Evaluation*, 2015.
[21] C. Pelsser, L. Cittadini, S. Vissicchio, and R. Bush. From Paris to Tokyo: On the Suitability of ping to Measure Latency. In *IMC*, 2013.
[22] R. Sedgewick and K. Wayne. *Algorithms*. Pearson Education, 2011.
[23] N. Spring, R. Mahajan, D. Wetherall, and T. Anderson. Measuring isp topologies with rocketfuel. *IEEE/ACM Trans. Netw.*, 12(1):216, 2004.
[24] C. Thomassen. On the complexity of finding a minimum cycle cover of a graph. *SIAM J. Comput.*, 26(3):675–677, 1997.
[25] S. Vissicchio, O. Tilmans, L. Vanbever, and J. Rexford. Central Control over Distributed Routing. In *SIGCOMM*, 2015.
[26] S. Vissicchio, L. Vanbever, and J. Rexford. Sweet Little Lies: Fake Topologies for Flexible Routing. In *Hotnets*, 2014.
[27] P. Wu, R. Bhatnagar, L. Epshtein, M. Bhandaru, and Z. Shi. Alarm correlation engine (ACE). In *NOMS*, 1998.
[28] H. Yan et al. G-RCA: A Generic Root Cause Analysis Platform for Service Quality Management in Large IP Networks. In *CoNEXT*, 2010.