# SyRep: Efficient Synthesis and Repair of Fast Re-Route Forwarding Tables for Resilient Networks

Csaba Györgyi[1] Kim G. Larsen[2] Stefan Schmid[1,3] Jiří Srba[2]
[1]University of Vienna [2]Aalborg University [3]TU Berlin

***Abstract*—In modern communication networks with stringent dependability requirements, local fast re-routing (FRR) is essential for a quick response to link failures. Configuring FRR for multiple failures is, however, challenging since a router's forwarding table may take into account only the failed links directly incident to it. We propose SyRep, an efficient method to repair and synthesize resilient FRR forwarding tables. At the heart of SyRep lies a method which identifies and removes ill-defined routing entries and employs symbolic binary decision diagram (BDD) technology to automatically replace the removed entries with correct values. SyRep cannot only be used to efficiently repair existing forwarding tables, but also to synthesize new tables from scratch, using an efficient hybrid approach: by first using fast heuristics that provide close-to-resilient routing tables and then quickly repair the ill-defined entries. We present such a fast heuristic based on novel structural reduction rules and our empirical evaluation shows that SyRep is up to three orders of magnitude faster compared to the state-of-the-art.**

***Index Terms*—Dependable communication networks, fast re-routing, synthesis, perfect resilience, binary decision diagrams**

## I. Introduction

Almost all modern communication networks support fast re-routing (FRR) mechanisms to quickly react to link failures, including IP networks [1]–[3], MPLS networks [4], [5], segment routing networks [6], [7], software-defined networks [8], [9], among many more [10]. These FRR mechanisms are based on conditional forwarding tables in routers which define where to forward a packet based on which links incident to the router failed. Such local forwarding decisions in the data plane can be orders of magnitudes faster compared to reactions to link failures in the control plane, as they do not require the communication of failure information [10].

However, while local reactions are fast, the lack of information about possible additional failures in other parts of the network renders configuring FRR mechanisms to provide a high resilience challenging: an additional link failure along a backup path can lead to a forwarding loop. And with the increasing scale of today's communication networks, multiple link failures are naturally more likely to occur [11]–[13]. Given the increasingly stringent dependability requirements, over the last years, the networking community has hence made great efforts to devise FRR algorithms which can tolerate multiple link failures [5], [10], [14]–[25].

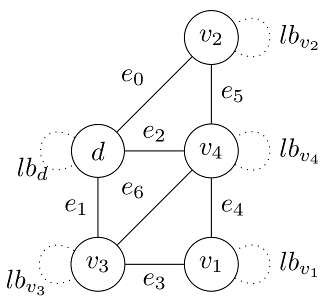
This paper explores *automated* approaches to generate highly resilient re-routing tables. Automated approaches are attractive as they relieve network operators from the complexity to reason about possible failure scenarios. In fact, generating resilient routing tables is expensive even for computers today, and recent tools such as [26] require significant time to ensure reachability.

We are particularly interested in designing *perfectly $k$-resilient FRR mechanisms*, for some parameter $k$ (which we aim to maximize): the network should re-route traffic between any pair of routers or hosts (henceforth called *nodes*) even if up to $k$ links fail, *as long as the two nodes are still connected in the underlying physical network*.

Perfect resilience has already been studied intensively in the literature [10], [22]. A common and particularly memory-efficient way to realize fast re-routing is *skipping* [10], [22], [26], [27]: the conditional failover rules at each node are organized as a priority list, which defines an order in which the next-hop links are chosen depending on their availability, and which results in small forwarding tables. For example, in software-defined networks based on OpenFlow [28], forwarding rules are organized in an ordered list of action buckets, and each bucket in the fast failover type table is associated with a parameter that determines whether the bucket is live; a switch will always forward traffic to the first live bucket.

This paper presents a new method to repair existing routing tables and an efficient algorithm to generate perfectly $k$-resilient FRR tables based on skipping. Our solution, SyRep, is using binary decision diagrams (BDDs) [29], [30] as a basis of an efficient method to *repair* existing routing entries to make the network more resilient. This is attractive for two reasons. First, repairing forwarding tables is an interesting use case on its own, as networks in practice already come with certain mechanisms and forwarding tables, e.g., populated by network protocols or manually, and our approach hence allows to quickly fortify an existing data plane. Second, this method can be used to efficiently *synthesize* forwarding tables *from scratch*, using a hybrid approach: first, using a fast heuristic that provides close-to-resilient routing tables, and then quickly repair the ill-defined entries using our rigorous BDD approach.

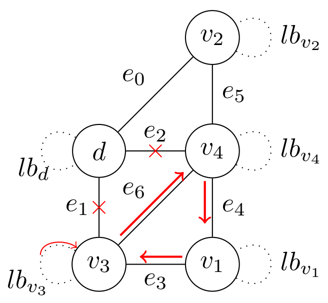
**Example.** Before we elaborate on our approach and contribution in more details, let us explain the skipping routing policy and our repair solution on a small running example. In Figure 1a we can see a network topology with five nodes (routers) connected by bi-directional edges (links). For every node $v$ we assume an implicitly present loop-back interface $lb_v$ (self-loop)—these are used to model the arrival of packets (e.g. locally generated traffic). The task is to route the packets from every node in the network to the destination node $d$. A skipping routing policy can be described using a routing table as shown



(a) Network with destination node $d$

| in-edge | node | out-edges |
|---|---|---|
| $lb_{v_1}$ | $v_1$ | $e_3$, $e_4$ |
| $e_3$ | $v_1$ | $e_4, e_3$ |
| $e_4$ | $v_1$ | $e_3$ ,$e_4$ |
| $lb_{v_2}$ | $v_2$ | $e_0, e_5$ |
| $e_5$ | $v_2$ | $e_0, e_5$ |
| $lb_{v_3}$ | $v_3$ | $e_1$, $e_6$ ,$e_3$ |
| $e_3$ | $v_3$ | $e_1$, $e_6$ ,$e_3$ |
| $e_6$ | $v_3$ | $e_1, e_3, e_6$ |
| $lb_{v_4}$ | $v_4$ | $e_2$, $e_4$ ,$e_5$ |
| $e_4$ | $v_4$ | $e_2, e_5, e_6$ |
| $e_5$ | $v_4$ | $e_2, e_4, e_6$ |
| $e_6$ | $v_4$ | $e_2$, $e_4$ ,$e_5$ |

(b) Perfectly 1-resilient routing table



(c) A forwarding loop with two failed links (starting from $v_1$ or $v_4$ has a similar effect)

Fig. 1: Example of configuration repair to synthesize a perfectly 2-resilient routing

in Figure 1b. The table for each ingress interface (in-edge) at a given node specifies a prioritized list of egress interfaces (out-edges). In case of no link failures, the first out-edge is always used for forwarding. If the first out-edge has failed, then the left-most out-edge in the list that is not failing is used for packet forwarding.

For example, a packet arriving on $lb_{v_3}$ to the node $v_3$ makes the next-hop directly to the destination node $d$ via the edge $e_1$, unless the edge $e_1$ is failing, in which case it instead uses the second priority edge $e_6$ to get to the node $v_4$; by following the routing table, the node $v_4$ then forwards the packet to $d$ via the edge $e_2$, unless $e_2$ also failed, in which case it forwards the packet along the edge $e_4$ to $v_1$, etc.

The provided routing table is perfectly 1-resilient, meaning that for any failure scenario with up to 1 broken link, a packet arriving to any node in the network is always delivered to the destination node $d$, unless the node is disconnected from $d$ (which cannot happen for one link failure only as the network is 2-connected). On the other hand, the table is not perfectly 2-resilient as shown in Figure 1c. In case of simultaneous failure of the links $e_1$ and $e_2$, even though the node $v_3$ is still connected to $d$, a packet arriving to the node $v_3$ creates a forwarding loop in the network by repeatedly visiting the nodes $v_4$, $v_1$ and $v_3$.

By exploring all failure scenarios of size at most 2, we notice that whenever the packet delivery fails then the forwarding entries highlighted in Figure 1b are used in the forwarding along the failed traces. We mark them as suspicious and using our repair algorithm, we try to replace them with alternative entries. Indeed, if the highlighted second priority edge $e_4$ in the last row of the table is dropped (i.e. replaced with $e_5$)—as fully automatically suggested by our method—we obtain a perfectly 2-resilient routing that guarantees a packet delivery with up to two failing links.

**Our contributions.** We design and implement SyRep, an efficient and automated approach to repair and synthesize highly dependable fast re-routing tables, providing perfect $k$-resilience. Our approach is based on the BDD technology, inspired by its recent deployment for computing and storing all perfectly resilient routings [26]; however, we use BDDs for repairing existing routing configurations—an application that has not been considered in [26] nor in other related work we are aware of. We shall now summarize our three main contributions.

As our first contribution, we define and implement an automatic method for repairing non-resilient routing tables. Compared to a recent work [26] which uses a symbolic method for the synthesis of all perfectly $k$-resilient routings, there are two main reasons why the repair methodology, introduced in our current work, is beneficial: (i) in case of network vulnerabilities to failures, network operators strongly prefer to implement the minimum invasive approach for fixing such issues instead of completely replacing the existing data-plane—our method identifies and replaces only the (usually very few) misbehaving entries, (ii) the scalability of the synthesis method from [26] suffers when increasing the size of the networks as well as the resilience level because it has to automatically fill in a large number of empty routing entries, which is computationally demanding—our method allows us to use several heuristic synthesis approaches that provide close-to-resilient routing tables and then quickly repair them with orders of magnitude improved CPU time.

As our second contribution, we suggest a fast heuristic algorithm for populating skipping routing tables as well as a method for reducing the size of the synthesized networks by applying structural reduction rules. For a given network topology, we first apply structural reductions in order to obtain a smaller network on which we further apply our heuristic synthesis algorithm to generate a $k$-resilient routing. If the heuristic approach fails, we repair the routing tables and then expand the routing from the reduced network to the original one. Again, we verify if the produced routing is $k$-resilient and if not, we run our repair algorithm. In many cases, this results in perfectly $k$-resilient data plane while using only a fraction of the computational time compared to the full synthesis on the original network.

As our third contribution, we implement the proposed methodology in a prototype tool (to be publicly released after the acceptance of this paper) and carry on a comparative experimental evaluation on a large range of the network topologies from the Internet Topology Zoo [31]. The results document up to three orders of magnitude speedup in generating resilient routing tables compared to the state-of-the-art [26].

**Organization.** The remainder of this paper is organized as follows. We present a formal problem specification in Section II. Sections III and IV describe our repair and heuristic approach in detail, which we evaluate in Section V. After we review related work in Section VI, we conclude in Section VII.

## II. Problem Formulation

We shall now provide formal definitions of a network, routing function for packet forwarding and define the notion of perfect resilience. We model a network topology as an undirected multigraph with loop-back edges.

**Definition 1.** *A* network *is an undirected multigraph $G = (V, E, r)$ where $V$ a finite set of* nodes *(routers), $E$ a finite set of undirected* edges *(links), and $r : E \to \{\{x, y\} \mid x, y \in V\}$ is a function assigning to each edge a set of connected nodes. We assume that for every node $v \in V$ there is always a* loop-back edge *(self-loop) called $lb_v$ such that $r(lb_v) = \{v\}$.*

Figure 1a depicts an example of a network where the loop-back edges are drawn using a dotted line. The loop-back edges are used to represent ingress interfaces for packets arriving to/originating in the given node. As the loop-back edges are always implicitly present, we can omit them when drawing a network topology.

Let us assume a fixed network $G = (V, E, r)$. A finite *path* in a network is a sequence of edges $(e_0, e_1, ..., e_n)$ such that there are nodes $v_0, v_1, ..., v_{n+1}$ where $r(e_i) = \{v_i, v_{i+1}\}$ for all $0 \leq i \leq n$. The path then *connects* the node $v_0$ to $v_{n+1}$.

We assume that edges in the network can fail. A *failure scenario* $F \subseteq E$ is a subset of $E$ and contains the set of all failed edges. In fast re-route mechanisms [10], a node $v$ has to be robust to any possible set of links that may fail but the forwarding decisions have to be resolved locally, i.e. based only on the knowledge of the set $F \cap \{e \in E \mid v \in r(e)\}$ which contains only the failed links that are incident to the node $v$. Even though the node $v$ has the knowledge of all failed incident links, creating a forwarding entry for every possible subset of failed links, also called *combinatorial routing* [32], is expensive and often infeasible in practice due to exponential memory requirements [33]. An alternative way, called *skipping routing* [22], is to provide for every node $v$ and any of its ingress interfaces (in-edge), a priority list of out-edges. This requires us to store only linearly many forwarding entries with the semantics that the node $v$ forwards the packet along the first non-failing edge in the priority list.

**Definition 2** (Skipping Routing). *A skipping routing in a network $G = (V, E, r)$ is a partial function*

$$R : E \times V \rightharpoonup E^*$$

*such that if $R(e, v) = (e_1, e_2, \ldots, e_\ell)$ then $v \in r(e) \cap r(e_1) \cap \ldots \cap r(e_\ell)$, i.e. all edges in the priority list as well as the incoming edge are connected to the node $v$.*

An example of a skipping routing in a form of a table is given in Figure 1b. The intuition is that, for a given a failure scenario $F$, if $R(e, v) = (e_1, e_2, \ldots, e_\ell)$ then any packet arriving on the edge $e$ to the node $v$ is routed via the first available edge $e_i \notin F$ where all the edges with higher priority are failed, meaning that $e_1, \ldots, e_{i-1} \in F$. If $e_1, \ldots, e_\ell \in F$ then the packet is dropped. We can now define the notion of a network trace: a path in the network that in a given failure scenario follows the predefined skipping routing policy.

**Definition 3** (Network Trace). *A sequence of edges $(e_0, e_1, \ldots, e_n)$ is a network* trace *from the node $v_0$ under the skipping routing $R$ and the failure scenario $F$ if*

- *$(e_0, e_1, \ldots, e_n)$ where $e_0 = lb_{v_0}$ is a path in the network that visits the nodes $v_0, v_1, \ldots, v_{n+1}$, and*
- *for every $i$, $0 \leq i \leq n$, we have $R(e_i, v_{i+1}) = (e'_1, e'_2, \ldots, e'_\ell)$ such that $e_{i+1} = e'_j$ for some $j$, $1 \leq j \leq \ell$, where $e'_1, \ldots, e'_{j-1} \in F$ and $e'_j \notin F$.*

For a given routing $R$ and failure scenario $F$, there is a unique longest trace from any node $v$, unless the trace forms a loop, as it can be seen e.g. in Figure 1c where under the failure scenario $F = \{e_1, e_2\}$ there is a trace from the node $v_3$ of the form $(lb_{v_3}, e_6, e_4, e_3, e_6, \ldots)$. On the other hand, in the failure scenario $F = \{e_1, e_6\}$, the longest trace from the node $v_3$ is of the form $(lb_{v_3}, e_3, e_4, e_2)$ and delivers the packet to the destination node $d$ (from which no further routing rules are defined).

We are now ready to introduce the notion of a perfectly $k$-resilient routing, stating that the routing delivers a packet from any source node to its destination node whenever the two nodes are connected in a given failure scenario with up to $k$ failed edges. In the definition, for a network $G = (V, E, r)$, we let $G^F = (V, E \smallsetminus F, r|_{E \smallsetminus F})$ be a network with all failed edges in $F$ removed.

**Definition 4** (Perfect Resilience). *Let $d \in V$ be a destination node and $k$ a nonnegative integer. A routing $R$ is* perfectly $k$-resilient, *if for every node $s$ and every failure scenario $F$ where $|F| \leq k$, there exists a trace from $s$ that reaches the node $d$ whenever there is a path connecting $s$ to $d$ in the graph $G^F$. A routing is* perfectly resilient *if it is perfectly $k$-resilient for all $k$.*

The looping trace from $v_3$ in Figure 1c implies that the skipping routing in Figure 1b is not perfectly 2-resilient; the trace clearly does not deliver the packet to $d$ even though there is a path from $v_3$ to $d$ via the edges $e_6$, $e_5$ and $e_0$. The routing table is though perfectly 1-resilient, which can be verified by considering all failure scenarios with up to one failed edge and checking that the routing always delivers the packet to $d$ from any given source node.

In Definition 4 we consider a fixed destination node. This is without loss of generality because each packet header has the information about its destination node and the routing tables for nodes with different destinations are considered as independent. Hence we can synthesize the routing tables for different destination nodes independently of each other. From now on, we hence focus only on a single, fixed destination node.

In what follows, we shall see how we can automatically remove suspicious routing entries that cause that a given routing is not perfectly $k$-resilient and try to repair the routing by finding suitable alternatives to the removed entries.

## III. Verify and Repair Method

In this section, we present our approach aimed at repairing pre-existing routings. Our method focuses on the identification and elimination of potentially misbehaving routing entries, after which the missing entries are filled in correctly using BDDs.

### A. Leveraging BDDs to Synthesise Routings

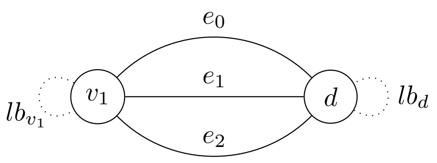
We build on the existing encoding of routing synthesis into Binary Decision Diagrams (BDD) [26] to find appropriate routing entries that provide the desired resiliency characteristics. We now recall the main building blocks of that method.

At the core of the tool lies a BDD formulation addressing the routing synthesis problem. Multiple potential routings are represented using BDDs, a data structure introduced by Lee [29], which efficiently encodes Boolean functions as rooted directed acyclic graphs (DAGs). Bryant [30] developed a reduced ordered variant of BDDs featuring fixed variable ordering and maximum sharing of isomorphic sub-graphs, thereby achieving a more compact and canonical representation. Universal and existential quantifiers are also accommodated in addition to basic logical operators, thus effectively supporting quantified Boolean formula (QBF). Furthermore, all operations of QBF exhibit polynomial complexity relative to the size of the underlying BDDs. For a more in-depth exploration of the technical intricacies regarding the formulation with BDDs, refer, for example, to [34].

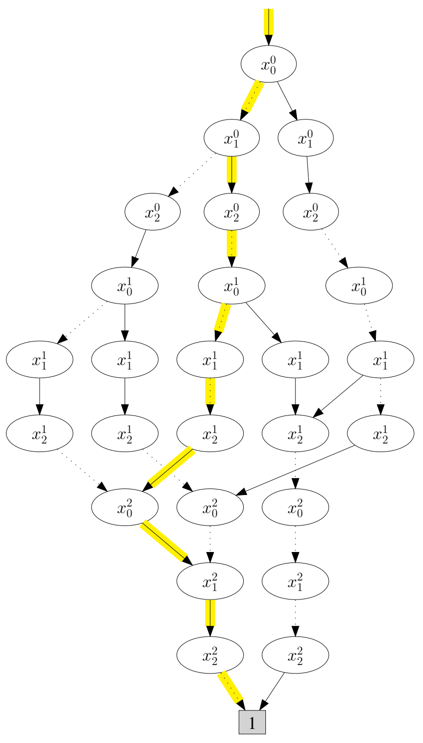
Any finite set $S$ can be efficiently represented by $n = \lceil \log(|S|) \rceil$ Boolean variables. Assuming a predetermined $s_0, s_1, ..., s_{|S|-1}$ enumeration of elements and denoting the Boolean variables as $\bar{\mathbf{x}} = (x_0, x_1, ..., x_n)$, any truth assignment $\mu$ to $\bar{\mathbf{x}}$ can be interpreted as the binary encoding of a natural number $n(\mu) \in \mathbb{N}$, representing the $n(\mu)$'th element within the set $S$.

We now present an extension of the BDD encoding of the routing synthesis problem with one link failure (as presented in [26]) to deal with multiple link failures.

First, we define a few helper formulae. Let $state(\bar{\mathbf{v}}, \bar{\mathbf{e}})$ denote (the encodings of) all the node and edge pairs that are connected in the network topology, and let $\mathcal{V}_{v,e}(\bar{\mathbf{e}}^0_{v,e}, \bar{\mathbf{e}}^1_{v,e}, ..., \bar{\mathbf{e}}^k_{v,e})$ encode all possible combination of $k+1$ edges (priority list)



(a) Network topology



(b) All perfectly 2-resilient routings encoded in BDD

Fig. 2: A simple network topology and a BDD encoding all possible perfectly 2-resilient routing

for each $v \in V$ and $e \in E$ such that every such edge is connected to $v$ and $\bar{\mathbf{e}}^0_{v,e}$ does not encode the edge $e$.

We now construct a Boolean formula $\mathcal{T}$ representing the valid transitions of a packet from one in-edge and node combination to another under a given failure scenario and routing entries is given as follows:

$$\begin{aligned}
&\mathcal{T}(\bar{\mathbf{e}}_{in}, \bar{\mathbf{v}}, \bar{\mathbf{e}}_{out}, \bar{\mathbf{v}}', \bar{\mathbf{f}}_1, ..., \bar{\mathbf{f}}_k, [\bar{\mathbf{e}}^0_{v,e}, \bar{\mathbf{e}}^1_{v,e}, ..., \bar{\mathbf{e}}^k_{v,e} : {}^{e \in E}_{v \in r(e)}]) = \\
&state(\bar{\mathbf{v}}, \bar{\mathbf{e}}_{in}) \wedge state(\bar{\mathbf{v}}, \bar{\mathbf{e}}_{out}) \wedge state(\bar{\mathbf{v}}', \bar{\mathbf{e}}_{out}) \wedge \\
&\bar{\mathbf{e}}_{out} \notin \{\bar{\mathbf{f}}_1, ..., \bar{\mathbf{f}}_k\} \wedge \\
&\bigwedge_{e \in E \wedge v \in r(e)} \mathcal{V}_{v,e}(\bar{\mathbf{e}}^0_{v,e}, \bar{\mathbf{e}}^1_{v,e}, ..., \bar{\mathbf{e}}^k_{v,e}) \wedge \\
&\bigwedge_{e \in E} \bigwedge_{v \in r(e)} \Big( \bar{\mathbf{e}}_{in}(e) \wedge \bar{\mathbf{v}}(v) \implies \bar{\mathbf{e}}_{out} \in \{\bar{\mathbf{e}}^0_{v,e}, ..., \bar{\mathbf{e}}^k_{v,e}\} \Big) \wedge \\
&\bigwedge_{i \in 0,..,k} \bigwedge_{e \in E} \bigwedge_{v \in r(e)} \Big( \bar{\mathbf{e}}_{in}(e) \wedge \bar{\mathbf{v}}(v) \wedge \bar{\mathbf{e}}^i_{v,e} \notin \{\bar{\mathbf{f}}_1, ..., \bar{\mathbf{f}}_k\} \wedge \\
&\{\bar{\mathbf{e}}^0_{v,e}, ..., \bar{\mathbf{e}}^{i-1}_{v,e}\} \subseteq \{\bar{\mathbf{f}}_1, ..., \bar{\mathbf{f}}_k\} \implies \bar{\mathbf{e}}_{out} = \bar{\mathbf{e}}^i_{v,e} \Big)
\end{aligned}$$

In the formula, $\bar{\mathbf{e}}_{in}$ encodes the in-edge we arrived to the current node represented by $\bar{\mathbf{v}}$. Similarly $\bar{\mathbf{e}}_{out}$ encodes the out-edge trough which we go to the next node represented by $\bar{\mathbf{v}}'$. The vectors of variables $\bar{\mathbf{f}}_1, ..., \bar{\mathbf{f}}_k$ encode the list of failed edges. Finally, $[\bar{\mathbf{e}}^0_{v,e}, \bar{\mathbf{e}}^1_{v,e}, ..., \bar{\mathbf{e}}^k_{v,e} : {}^{e \in E}_{v \in r(e)}]$ represents the elements of the priority list for each $e \in E$ in-edge, $v \in V$ node. In the second line, we require that the in-edge, the out-edge, the current node and the next node are appropriately connected. The third line enforces that the out-edge is not failed. The fourth line ensures that only valid routings are considered. The fifth line requires that the out edge is present at the priority list. In the last two lines, we make sure that the selected index at the priority list is not failed but all links before it are failed.

We now proceed by defining a sequence of Boolean formulae (represented as BDDs) starting with $\mathcal{D}_0(\bar{\mathbf{e}}_{in}, \bar{\mathbf{v}}, \bar{\mathbf{f}}_1, ..., \bar{\mathbf{f}}_k, [\bar{\mathbf{e}}^0_{v,e}, \bar{\mathbf{e}}^1_{v,e}, ..., \bar{\mathbf{e}}^k_{v,e} : {}^{e \in E}_{v \in r(e)}]) = (\bar{\mathbf{v}} = \bar{\mathbf{d}})$ where $\bar{\mathbf{d}}$ is the Boolean encoding of $d$ destination node. The sequence is built using the following rule:

$$\begin{aligned}&\mathcal{D}_{n+1}(\bar{\mathbf{e}}_{in}, \bar{\mathbf{v}}, \bar{\mathbf{f}}_1, ..., \bar{\mathbf{f}}_k, [\bar{\mathbf{e}}^0_{v,e}, \bar{\mathbf{e}}^1_{v,e}, ..., \bar{\mathbf{e}}^k_{v,e} : {}^{e \in E}_{v \in r(e)}]) = \\ &\mathcal{D}_n(\bar{\mathbf{e}}_{in}, \bar{\mathbf{v}}, \bar{\mathbf{f}}_1, ..., \bar{\mathbf{f}}_k, [\bar{\mathbf{e}}^0_{v,e}, \bar{\mathbf{e}}^1_{v,e}, ..., \bar{\mathbf{e}}^k_{v,e} : {}^{e \in E}_{v \in r(e)}]) \lor \Big(\exists \bar{\mathbf{v}}', \bar{\mathbf{e}}_{out} : \\ &\mathcal{T}(\bar{\mathbf{e}}_{in}, \bar{\mathbf{v}}, \bar{\mathbf{e}}_{out}, \bar{\mathbf{v}}', \bar{\mathbf{f}}_1, ..., \bar{\mathbf{f}}_k, [\bar{\mathbf{e}}^0_{v,e}, \bar{\mathbf{e}}^1_{v,e}, ..., \bar{\mathbf{e}}^k_{v,e} : {}^{e \in E}_{v \in r(e)}]) \\ &\land \mathcal{D}_n(\bar{\mathbf{e}}_{out}, \bar{\mathbf{v}}', \bar{\mathbf{f}}_1, ..., \bar{\mathbf{f}}_k, [\bar{\mathbf{e}}^0_{v,e}, \bar{\mathbf{e}}^1_{v,e}, ..., \bar{\mathbf{e}}^k_{v,e} : {}^{e \in E}_{v \in r(e)}])\Big)\end{aligned}$$

Let $\mathcal{D}$ be $\mathcal{D}_m$ where $\mathcal{D}_m = \mathcal{D}_{m-1}$ representing all in-edge and node pairs $(\bar{\mathbf{e}}_{in}, \bar{\mathbf{v}})$ from which packets are delivered to the destination under the failure scenario encoded by $\bar{\mathbf{f}}_1, ..., \bar{\mathbf{f}}_k$ using the routing configuration described by $[\bar{\mathbf{e}}^0_{v,e}, \bar{\mathbf{e}}^1_{v,e}, ..., \bar{\mathbf{e}}^k_{v,e} : {}^{e \in E}_{v \in r(e)}]$.

By employing $\mathcal{D}$, we can now express the Boolean formula $\mathcal{P}$ encoding all possible prefectly $k$-resilient routings as follows:

$$\begin{aligned}&\mathcal{P}([\bar{\mathbf{e}}^0_{v,e}, \bar{\mathbf{e}}^1_{v,e}, ..., \bar{\mathbf{e}}^k_{v,e} : {}^{e \in E}_{v \in r(e)}]) = \forall \bar{\mathbf{f}}_1, ..., \bar{\mathbf{f}}_k : \forall (\bar{\mathbf{e}}_s, \bar{\mathbf{s}}) : \\ &\Gamma(s(\bar{\mathbf{s}}), f_1(\bar{\mathbf{f}}_1), ..., f_k(\bar{\mathbf{f}}_k), d) \implies \\ &D(\bar{\mathbf{e}}_s, \bar{\mathbf{s}}, \bar{\mathbf{f}}_1, ..., \bar{\mathbf{f}}_k, [\bar{\mathbf{e}}^0_{v,e}, \bar{\mathbf{e}}^1_{v,e}, ..., \bar{\mathbf{e}}^k_{v,e} : {}^{e \in E}_{v \in r(e)}])\end{aligned}$$

where $\Gamma(v_1, e_1, ..., e_k, v_2) : V \times E^* \times V \longrightarrow \mathbb{L}$ is true iff there is a path between $v_1$ and $v_2$ not containing $e_1, ..., e_k$. The input parameters $\bar{\mathbf{e}}^0_{v,e}, \bar{\mathbf{e}}^1_{v,e}, ..., \bar{\mathbf{e}}^k_{v,e}$ for $\mathcal{P}$ (vectors of Boolean variables) provide a Boolean encoding of the edges in the priority list of $R(e, v)$. For the predefined routing entries, we can simply substitute the corresponding routing parameter by their concrete values during the BDD calculations. In particular, this will allow us to restrict the synthesis to only a few selected routing parameters.

Our example network in Figure 2a has only two nodes, where $d$ denotes the destination. Since the graph is so small, we only need to synthesise the $R(lb_{v_1}, v_1)$ priority list. Without going deeper into the details of the computation, it is easy to see that the only way of providing the required resiliency is if we try all 3 possible out-edges in some order. We use the $lb_d, lb_{v_1}, e_0, e_1, e_2$ ordering of edges for their Boolean representation. Since we have 5 edges, we need 3 bits to encode an edge. For example, 011 encodes the third edge in our ordering which is $e_1$ since we index from 0. Figure 2b depicts the BDD representation of the Boolean formula $\mathcal{P}(\bar{\mathbf{e}}^0_{v_1,lb_{v_1}}, \bar{\mathbf{e}}^1_{v_1,lb_{v_1}}, \bar{\mathbf{e}}^2_{v_1,lb_{v_1}})$ encoding all possible perfectly 2-resilient routings. One can get a satisfying assignment by following the arrows from the top to the 1 node, such that the dotted arrows denote that the variable in that node is set to 0 (false) and a solid arrow means that it gets the value 1 (true). For simplicity, the $\bar{\mathbf{e}}^0_{v_1,lb_{v_1}}, \bar{\mathbf{e}}^1_{v_1,lb_{v_1}}, \bar{\mathbf{e}}^2_{v_1,lb_{v_1}}$ parameters are denoted by variable vectors $\bar{\mathbf{x}}^0$, $\bar{\mathbf{x}}^1$ and $\bar{\mathbf{x}}^2$ and we use the $x^i_j$ to denote the *j-th* bit of the vector $\bar{\mathbf{x}}^i$ where the least significant bit is with index 0. Figure 2 depicts a simple network topology and a BDD encoding all possible perfectly 2-resilient routing. Now the highlighted path encodes the values 2, 4 and 3 corresponding to the priority list $R(lb_{v_1}, v_1) = (e_0, e_2, e_1)$. Similarly, we can observe all six permutations of these edges as correct perfectly 2-resilient routings, all compactly encoded in the BDD data structure.

## B. Routing Verification

We rely on the fact that, for small $k$ values, verification of a routing can be efficiently achieved using a conventional brute-force approach. Systematically evaluating all possible failure scenarios and following traces from all nodes to the destination node can be done with an acceptable runtime. During this process, we store the failing scenarios and the routing entries used. If a certain routing entry was firing during the routing trace, where the packet was not delivered to its destination, we label that routing entry as *suspicious*. Let us call the pair $(v, F)$ a *failing delivery* if the packet starting from node $v$ cannot be delivered under the failure scenario $F$.

In case of our running example depicted in Figure 1, the routing presented in Figure 1b fails to deliver the packets starting from $v_1, v_3$ and $v_4$ if $e_1$ and $e_2$ fail, since the packets enter a loop. In fact, $(v_1, F)$, $(v_3, F)$ and $(v_4, F)$ where $F = \{e_1, e_2\}$ are exactly all failed deliveries with up to 2 failed links. During these failed deliveries, we used the six entries highlighted with grey in Figure 1b and labeled these entries as *suspicious*.

## C. Routing Repair

After collecting the *suspicious* entries, we remove these (and only these) entries from the routing leaving them as parameters (or *holes*) to be synthesized using the BDD-based method described in Section III-A. Here, the BDD-based algorithm will propose new routing entries that make the current routing $k$-resilient if such changes are possible.

Unfortunately, the repair mechanism is not complete, meaning that there are cases, when after removing the suspicious entries there are still no possible routing entries that can make the routing $k$-resilient. The core of the weakness lies in some particular ill-defined entries that make the discovery of remaining ill-defined entries impossible. For example, the priority list $R(e, v) = (e, e')$ sends back every packet on the in-edge $e$. Since the packet arrived on $e$, the edge $e$ can not be failed, thus $e'$ will never be used as it has a lower priority.

If $e'$ is ill-defined, we have no chance of removing it and suggesting an alternative after the suspicious entry with the edge $e$ is removed.

Removing every *suspicious* entry may also result in too many missing entries, thus slowing down the BDD calculation. Gradually removing *suspicious* routing entries is also possible and can be beneficial in certain cases. However, one should be careful when selecting the removed *suspicious* entries. First, for each *failing delivery* we must remove at least one firing entry, otherwise the BDD-based calculation has no chance of fixing the corresponding *failing delivery*. Moreover, not every failure is relevant for a given packet, e.g., if a packet starting from node $v$ is using the same entries in failure scenarios $F_1$ and $F_2$ where $F_1 \subseteq F_2$, then the *failing delivery* $(v, F_2)$ does not provide any additional insight.

Continuing the repair of the routing table from our running example presented in Figure 1b, we remove all six suspicious entries from the routing and let the BDD calculations fill them in. The BDD solution yields a perfectly 2-resilient solution, where every routing entry is the same except that the second priority of $R(e_6, v_4)$ is now $e_5$. Note that using only $e_5$ as a backup-edge does not impact perfect 2-resilience, since we use this entry only if we already encountered 2 failures.

## IV. Fast Generation of Routing Tables

We shall now present two approaches that allow us to synthesize close-to-resilient routing tables in polynomial time.

### A. Heuristic Routing Generator

First, we suggest a construction of skipping routings for a given network topology that, as we demonstrate in our experiments, often provide routing tables with high resilience.

Let $G = (V, E, r)$ be a network topology together with the destination node $d$. We first fix for each node $v \in V$ a primary next-hop edge called $e_v$ such that $v \in r(e_v)$, for example by following one of the shortest paths from $v$ to the destination node $d$. We call the fixed path from $v$ to the destination the *default path*. The choice of the default path can be used to minimize additional quantitative properties of the routing, like strech or congestion, in the failure-free scenario.

By $post(v)$ we denote the set of all nodes that are on the default path from $v$ to $d$. By $pre(v)$ we denote the set of all nodes such that their default paths to $d$ contain the node $v$.

We now recall the notion of node's level and the definition of a backup edge from [26]. We say that a node $v \in V$ is of *level* $\ell$, if there is an edge $e$ where $r(e) = \{v, v'\}$ such that the default path from $v'$ to $d$ intersects $post(v)$ in at most $\ell$ nodes. The minimum (lowest) level of a node $v$ is denoted by $mlevel(v)$ and the corresponding edge $e$ is called its *mlevel edge*. A *backup edge* for $v$ is either an *mlevel edge* in case $v$ has the smallest *mlevel* among all nodes in $pre(v)$, or the backup edge is a defaulte edge $e_{v'}$ for some node $v' \in pre(v)$ such that some smallest *mlevel* node from $pre(v)$ is also in $pre(v')$.

We note that there can be several different backup edges for a given node $v$. These are our primal candidates for

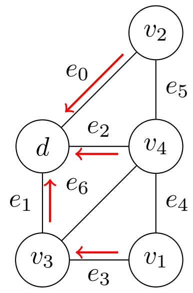


Fig. 3: Default paths for the running example from Figure 1

appropriate high priority edges in the skipping routing (that we shall construct) because by following them we minimize the possibility that we meet a failed edge that was encountered on the default path from $v$ to $d$.

Our heuristically constructed routing $R$ is now defined so that from every node $v$, the first priority always follows the default path edge $e_v$, unless we arrive to $v$ via the edge $e_v$ itself (getting ourselves further away from $d$), in which case we select as the first priority edges the backup edges for $v$, followed by the remaining edges. Formally, we define a skipping routing $R$ such that for every edge $e \in E$ and every node $v \in V \setminus \{d\}$

- if $e \neq e_v$ then
$$R(e, v) := (e_v, e_1, e_2, \ldots, e_\ell, e'_1, e'_2, \ldots, e'_m, e)$$
- otherwise
$$R(e_v, v) := (e_1, e_2, \ldots, e_\ell, e'_1, e'_2, \ldots, e'_m, e_v)$$

where $e_1, e_2, \ldots, e_\ell$ are all backup edges for $v$ (in an arbitrary order) and $e'_1, e'_2, \ldots, e'_m$ are all the remaining edges (different from $e_v$ and the backup edges) connected to $v$. As the very last resort, we add the edge on which the packet arrived at the end of the priority list (meaning that we return the packet back to the sender only if all other options fail).

If we restrict the constructed routing $R$ to contain only the first backup edge, we obtain with guarantee a perfectly 1-resilient routing as proven in [26]. Our extension of the construction to higher levels of resiliency is though not guaranteed to be safe. In fact, this cannot even be achieved in general, as there exist networks for which there are no perfectly 3-resilient routings [22] and the question of whether for any network there exists a perfectly 2-resilient routing is still an open problem (see e.g. [26]). On the other hand, we provide experimental evidence that our fast (polynomial) heuristic construction of skipping routings either already provides 2- or 3-resilient routings and if not, it can often be repaired using our method from Section III.

Figure 3 depicts the primary next-hops for all nodes in our running example (and the default paths to the destination). In Figure 1b we can see a routing table $R$ constructed using our heuristic approach. For example for the entry $R(lb_{v_3}, v_3) = (e_1, e_6, e_3)$ we only have one option of edge ordering. The first edge $e_1$ is the default next-hop (that delivers to $d$) and the second edge is $e_6$ as it is the only backup (mlevel) edge for $v_3$ that has the level 1 (the default path from $v_4$ intersects the default path from $v_3$ only at the destination). Finally, $e_3$ is the last edge in the priority list as it is not a backup edge for $v_3$ because the default path from $v_1$ goes via $v_3$ and hence intersects the $v_3$ default path at two nodes ($v_3$ and $d$).

On the other hand, there are two options for the routing entry at node $v_4$ when arriving on the edge $e_6$: either $R(e_6, v_4) = (e_2, e_4, e_5)$ (the one chosen in the table from Figure 1b) or $R(e_6, v_4) = (e_2, e_5, e_4)$. This is because both $e_4$ and $e_5$ are backup edges for $v_4$ — the default paths from the target nodes of these two edges intersect the default path from $v_4$ only at the destination node $d$. We can see that our choice of the first option caused that the routing table is not perfectly 2-resilient (but can be quickly repaired), while the other choice produces already a perfectly 2-resilient routing.

## B. Structural Reduction Methods

Reducing the size of the network can speed up the synthesis of resilient routing configurations. In this subsection we leverage structural reduction rules to get $k$-resilient routings for smaller graphs that can be extended and used for the original network. We present two reduction rules with different guarantees.

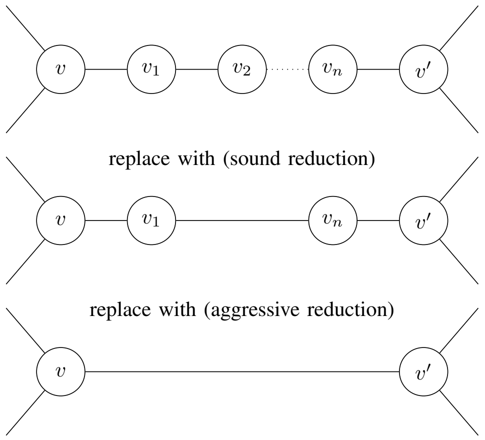
**Structural reduction rule 1: sound chain-reduction.** In the sound reduction, we reduce every chain with at least four edges to only three edges as depicted in Figure 4. This means that we can remove a node $v_i$, $1 < i < n$, from the network topology with the destination node $d$ if

- $v_i$ has exactly two incident edges $e'$ and $e''$ where $r(e') = \{v_i, v_{i-1}\}$, $r(e'') = \{v_i, v_{i+1}\}$ such that $v_{i-1}, v_i, v_{i+1}$ and $d$ are all different nodes, and
- $v_{i-1}$ and $v_{i+1}$ have exactly two incident edges too.

After removing $v_i$, we introduce a new edge $e$ that connects the nodes $v_{i-1}$ and $v_{i+1}$ such that $r(e) = \{v_{i-1}, v_{i+1}\}$. We repeat the application of this reduction rule until no further nodes can be removed.

We can now compute a resilient routing $R$ on the reduced network (which can take less time than on the original network) and expand $R$ to a routing on the original network as follows (we again refer to Figure 4):

- if $R(e''', v_1)$ contains a new edge $e$ not present in the original network, we replace it with an original edge between $v_1$ and $v_2$ and similarly if $R(e''', v_n)$ contains a newly added edge, we replace it with the edge between $v_n$ and $v_{n-1}$, and
- for every removed node $v_i$ with incident edges $e'$ and $e''$, we define $R(e', v_i) = (e'', e')$, $R(e'', v_i) = (e', e'')$ and $R(lb_{v_i}, v_i) = (e', e'')$ where
  - $e'$ connects $v_{i-1}$ and $v_i$ in case that $R(lb_{v_1}, v_1)$ contains as the first priority an edge between $v$ and $v_1$ (i.e. the default edge of $v_1$ points to the left), otherwise
  - $e'$ is the edge that connects $v_i$ and $v_{i+1}$ (i.e. the default edge of $v_1$ points to the right).


$v$
$v_1$
$v_2$
$v_n$
$v'$
replace with (sound reduction)
$v$
$v_1$
$v_n$
$v'$
replace with (aggressive reduction)
$v$
$v'$
Fig. 4: Sound and aggressive structural reduction rules

We can now prove the following soundness theorem.

**Theorem 1.** *Let $R$ be a routing on the reduced network after any number of applications of the sound reduction rule and $R'$ be the routing that expands $R$ to the original network. If $R$ is perfectly $k$-resilient on the reduced network then $R'$ is perfectly $k$-resilient on the original network.*

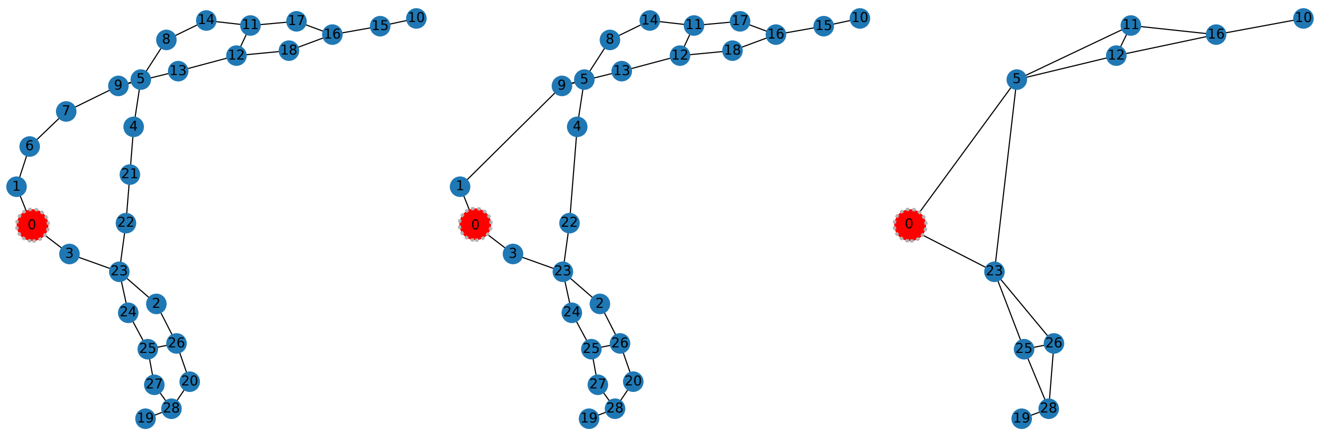
*Proof sketch.* Let us assume that in the reduced network $R$ delivers a packet arriving at any source node to the destination node $d$ under any failure scenario of size at most $k$, whenever the source and destination nodes are connected. We shall argue that $R'$ also delivers a packet from any source node to the destination under any failure scenario $F'$, $|F'| \leq k$, where the source and destination node remain connected.

Let us first construct a failure scenario $F$ in the reduced network such that $F$ contains all edges from $F'$ that appear also in the reduced network and if $F'$ contains an edge that is not in the reduced net then we add to $F$ the newly added edge that connects the nodes $v_1$ and $v_n$ directly (refer to Figure 4).

If a packet now arrives in the original network under the failures $F'$ to some source node that is also present in the reduced network, the same routing as in $R$ is applied in $R'$. Should the network trace traverse the nodes $v_2, \ldots, v_{n-1}$ (say from left to right) that were removed, the expanded routing $R'$ will deliver the packet to the end node $v'$ as in the reduced network, unless some of the edges are failed, in which case the packet returns back to $v$ (the same behaviour as in the reduced net in case that the newly added edge between $v_1$ and $v_n$ is failed). Clearly, if the reduced routing $R$ deliveres the packet to $d$, so will the expanded routing $R'$.

If a packet arrives in original network under $F'$ to a source node $v_i$, $1 < i < n$, that is not present in the reduced network then we in $R'$ forward in the same direction as the node $v_1$ does in the routing $R$ (either to the left via $v$ or to the right via $v_n$). As the packet arriving at $v_1$ under $R$ will be delivered to the destination node $d$, so will be the packet arriving at the node $v_i$.

This concludes that $R'$ is perfectly $k$-resilient whenever $R$ is perfectly $k$-resilient. $\square$



(a) Original (29 nodes) (b) After sound reduction (26 nodes) (c) After aggressive reduction (11 nodes)

Fig. 5: Effect of the two structural reduction rules on the *BizNet* topology where node 0 is the destination node.

The sound reduction rule allows us to extend any perfectly $k$-resilient routing in the reduced network to the original one while preserving its resilience. However, the reduction rule only applies to networks that contain chains of nodes with at least four edges, which does not happen too often in the typical ISP topologies. Reducing chains with less then three edges will imply that the endpoints of the reduced chain are able to see a possible link failure inside the chain which can help them to make more qualified routing decisions (something that is not possible in the unreduced network).

In order to extend the applicability of the reduction rule, we suggest an aggressive variant of the reduction rule which is more often applicable on typical ISP topologies, however, it does not guarantee the correctness of the reduction as shown for the sound reduction rule in Theorem 1. On the other hand, if the reduced routing cannot be expanded to a resilient routing on the original net, we can use our repair method. Advantages of this approach are discussed in our experimental section.

**Structural reduction rule 2: aggressive chain-reduction.** This rule removes a super-set of nodes removed by the previous safe rule thus allowing for smaller reduced networks. An application of the aggressive reduction rule is shown at the bottom of Figure 4. We remove a node $v_i$, $1 \leq i \leq n$, from the network if

- $v_i$ has exactly two incident edges $e'$ and $e''$ where $r(e') = \{v_i, v'\}$, $r(e'') = \{v_i, v''\}$ and $v_i, v', v''$ and $d$ are different nodes.

After removing the node $v_i$, the nodes $v'$ and $v''$ are connected with a newly added edge $e$ and as before the reduction rule can be applied multiple times. We can now construct a perfectly $k$-resilient routing on the reduced network and try to extend it to the original one using the same approach as for the sound reduction rule. The only issue is how to define the expanded routing entry for $R(lb_{v_i}, v_i)$ for a removed node $v_i$. In case of the sound reduction rule, we overtook the forwarding direction of the node $v_1$, however, in the aggressive reduction the node $v_1$ is removed as well. We shall instead forward the packet in the direction of a shortest path to the destination of $d$ as it is experimentally confirmed to return close-to-resilient routings.

If the resulting extended routing is not perfectly $k$-resilient, we apply our repair method from Section III. In Figure 5 we can see the effect of the sound and aggressive reduction on the *BizNet* topology from the topology Zoo database [31].

## V. Implementation and Evaluation

We implemented the heuristic synthesis methods as well as our repair procedure in the tool SyRep (that will be made publicly available as open source after the acceptance of the paper). Our implementation relies on the CUDD [35] backend of the omega [36] Python library.

### A. SyRep *Architecture*

The overall flow diagram of SyRep is depicted in Figure 6. We can either start with network topology without any existing routing information, apply the aggressive structural reduction rule and generate (using our fast heuristic method) a skipping routing for the reduced network. We then verify if the routing is $k$-resilient and try to repair it otherwise. After this, the resilient routing on the reduced network is expanded to the full network and passed to the verify/repair module. As an alternative, an existing (less resilient) routing on the network can be collected and passed directly to the verify/repair module.

The verify/repair module first checks if the routing is already perfectly $k$-resilient and if not, it initiates the repair procedure. As the outcome, we get either perfectly $k$-resilient routing on the original input network, or the information that the generated or existing routing cannot be repaired in order to achieve $k$-resilience.

### B. *Experimental Evaluation*

We run our experiments on the publicly available benchmark of ISP topologies called topology Zoo [31], using all connected networks from the benchmark. Each of the topologies are for a single AS (Autonomous System). We consider up to 3

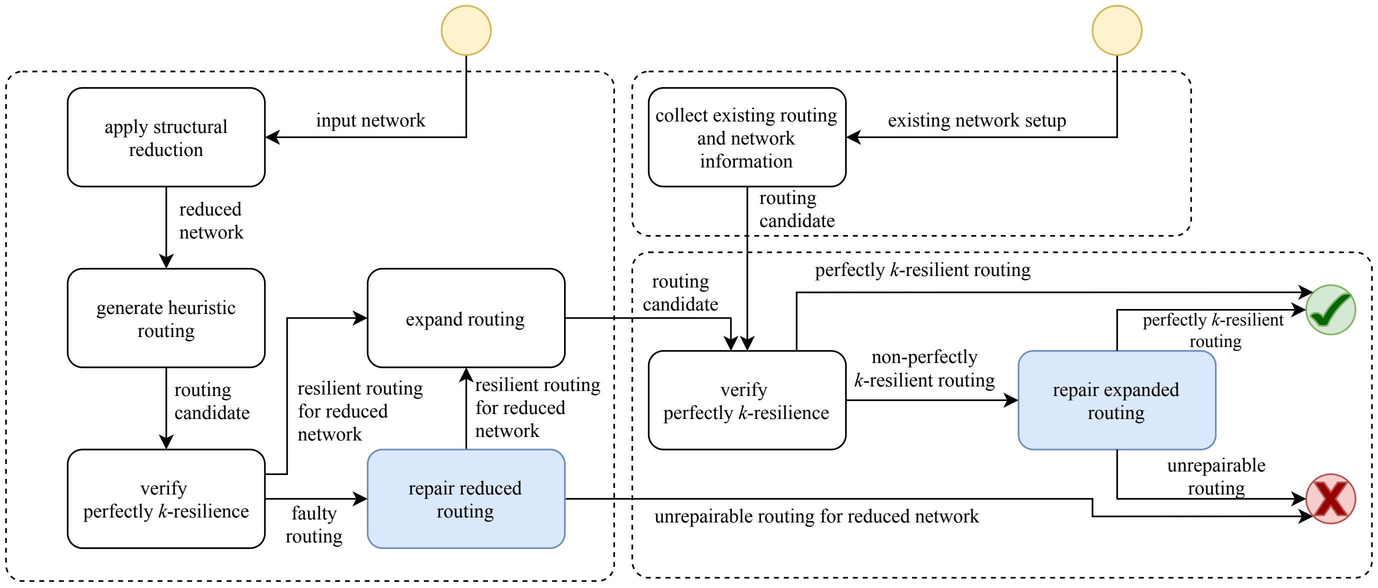

input network
apply structural reduction
reduced network
generate heuristic routing
routing candidate
verify perfectly $k$-resilience
resilient routing for reduced network
expand routing
resilient routing for reduced network
faulty routing
repair reduced routing
existing network setup
collect existing routing and network information
routing candidate
routing candidate
perfectly $k$-resilient routing
verify perfectly $k$-resilience
non-perfectly $k$-resilient routing
repair expanded routing
perfectly $k$-resilient routing
unrepairable routing
unrepairable routing for reduced network

Fig. 6: Modular architecture of SyRep

failed links in these ISP topologies, as this is often a sufficient resiliency level in IP backbone networks (very few topologies from the benchmark are more than 3-connected anyway).

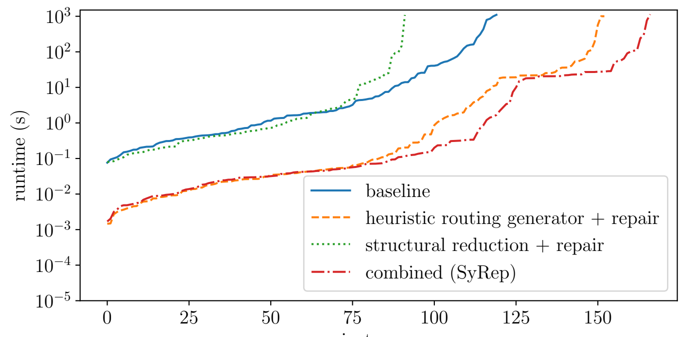
All experiments are conducted utilizing Intel Xeon Gold 6209U CPUs (2.10GHz) with a memory limit of 128GB and a timeout set to 20 minutes. We note that our algorithms are guaranteed to terminate, however, for some larger networks, they may exceed the given timeout. On the other hand, sometimes a network routing is not repairable by our method, even though a perfectly resilient routing may exist.

In Figure 7a we compare the baseline method from [26] for the synthesis of perfectly 2-resilient routings with our combined approach that uses both the aggressive structural reduction as well as heuristic routing generator. For completeness, we also add the results where the heuristic generator and structural reduction are used on their own. Independently for each of the method, the network instances (on x-axis) are ordered by the CPU time (on y-axis) that is needed to find a perfectly 2-resilient routing. We notice that our combined method achieves about two orders of magnitude speedup (the y-axis is logarithmic) compared to the baseline method. Interestingly, the structural reduction method on its own performs worse than the baseline (likely because it has to compute both the reduced solution BDD and then the BDD solution for the whole network, which does not pay off in some cases), however, in combination with the heuristic routing generator it achieves superior performance compared only to the heuristic routing generator alone. In total, the baseline solved 120 instances while our combined method solved 167. The expansion succeeded in 126 cases and repair was initiated only for 41 networks where in all cases it successfully repaired the routing to be perfectly 2-resilient.

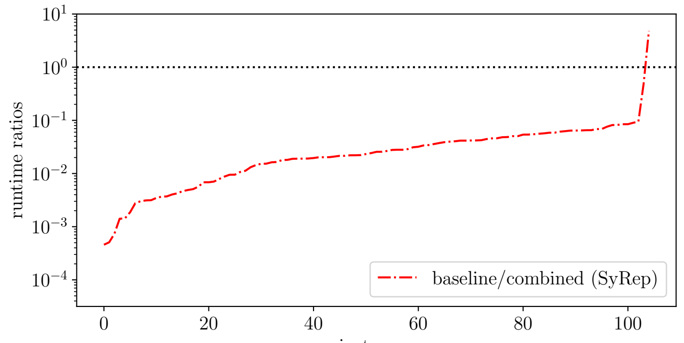
The plot in Figure 7a highlights the main overall performance trends but does not show instance to instance comparison of the methods (all methods are sorted independently). In Figure 7b we instead show the ratios (sorted on x-axis) of our combined runtime divided by the baseline time. Only the ratios where both methods finished before the timeout are shown. The ratios are reported for the same instance of the problem and values smaller than 1 imply that our method is faster than the baseline on the concrete instance of the problem. This is in fact always the case except for a single network *Renater2001* on which the combined method used 21.9 seconds while the baseline found the solution in 4.6 seconds. This is likely due to the fact that in this singular case computing the two BDDs (first the reduced one and then repairing the expanded one) is more expensive than computing the final BDD with all solutions directly. However, on most of the networks, our method provides between one to three orders of magnitude speedup.

Similarly, Figure 7c shows the plots for generating perfectly 3-resilient routings. The baseline solved 89 instances while we solved 106 and 6 of those needed a repair that was successful. The number of solved instances is lower than for 2-resilient routing generation as the problems become combinatorially more challenging. The main observation is that our combined method is even faster than the baseline. Again, structural reduction helps less than the heuristic routing generation but their combination solves larger number of instances compared to 2-resilient routings. The ratio plot in Figure 7d demonstrates also larger instance per instance improvement in the running time (up to four orders of magnitude). There were only two instances where our method is slightly slower than the baseline, namely *Cesnet200511* and *Cesnet200603*, on both of which we saw less than 6% deterioration in the running time.

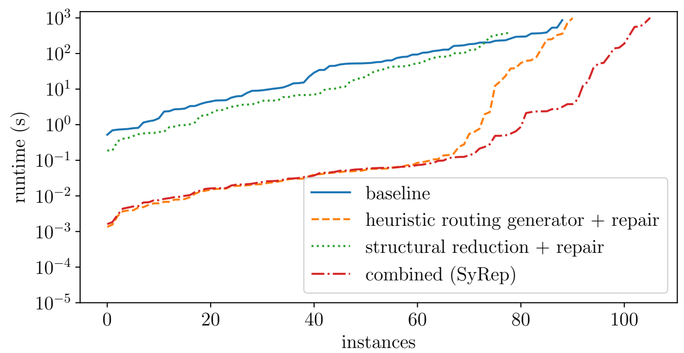
Finally, in Figures 8 and 9 we can see how the running time for constructing resilient routings for $k = 2$ and $k = 3$



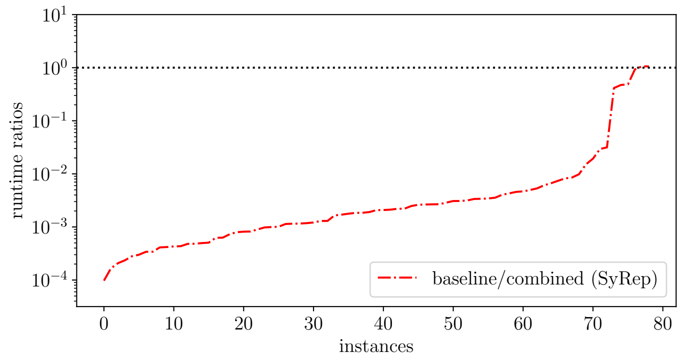
(a) Sorted runtimes for $k = 2$



(b) Sorted runtime ratios for $k = 2$



(c) Sorted runtimes for $k = 3$



(d) Sorted runtime ratios for $k = 3$

Fig. 7: Topology Zoo benchmark results

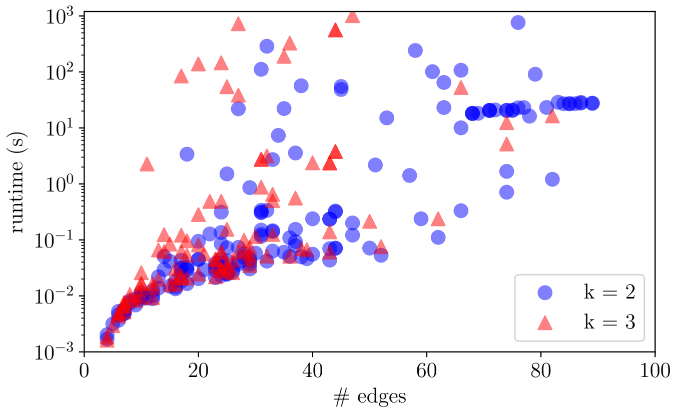


Fig. 8: Network size (**number of edges**) vs. runtime

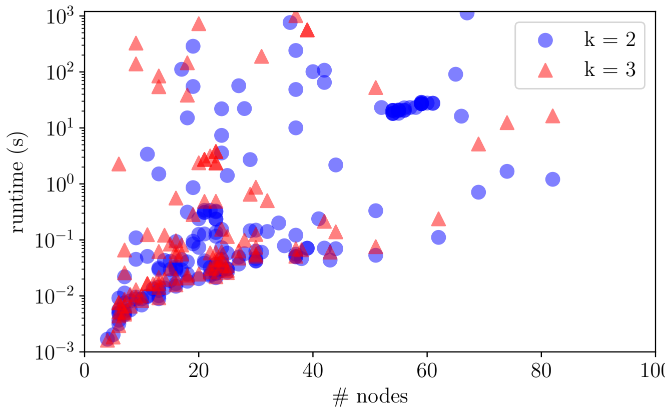


Fig. 9: Network size (**number of nodes**) vs. runtime

depends on the number of edges and nodes in the network. We can see that the required time is growing with the number of edges and nodes and we reach the limit of our method at around 80 edges/nodes. The running time also clearly depends on the structure of the network, sometimes even relatively small networks can be difficult to synthesize routings for. Also, as expected, constructing 3-resilient routings is more difficult than constructing 2-resilient routings and in particular for networks with 50 edges are more, there are several instances where the computation for $k = 3$ did not finish within the timeout while for $k = 2$ the result was computed relatively fast.

## VI. Related Work

Fast re-routing mechanisms have been studied intensively in the literature and are already widely deployed, see also the survey by Chiesa et al. [10]. Fast re-routing is attractive as it is a purely local mechanism and reactions can be an order of magnitude faster than in the control plane [14] which requires the communication of failure information, e.g. [37], [38].

The challenges introduced by the inherent locality of fast re-routing have been subject to much research already as well. Feigenbaum et al. [23] proved the negative result that it is not always possible to ensure a failover route between two routers even if they are physically connected (known as perfect resilience); on the positive side, they presented a deterministic algorithm which is provably 1-resilient for any network (graph). This result has later been generalized by Dai et al. [25] who presented an algorithm which is always 2-resilient, but requires source matching, however, as stated in [26], the question of 2-resilience without source matching remains open. This is interesting as in more restricted models (namely circular routing), it has been known for several years that 2-resilience is not possible [27]. Borokhovich et al. [32] showed that locality not only limits the achievable reachability on the routing layer, but also the achievable link utilization.

There also exists work on the achievable resilience against multiple link failures on special graphs. Foerster et al. [22] studied how the achievable resilience depends on the graph minors of the network. They proved that perfect resilience (even under multiple link failures) is always achievable in outerplanar graphs, but impossible on non-planar graphs. The authors further showed that all graph families closed under link subdivision allow for a simple and efficient failover algorithm, using skipping of failed links, though they do not provide any implementation that we can compare against. For the special case of $k$-connected graphs (for some $k$), Chiesa et al. [19], [20], [27] showed that it is always possible to tolerate $k-1$ link failures, using arc-disjoint arborescence covers. This approach generalizes the widely-used approaches based on spanning trees [39] and can also be implemented with skipping. Yang et al. [15] initiated the study of approaches beyond spanning trees and acyclic graphs, however, without providing strong formal connectivity guarantees.

Fast re-routing solutions have further also been studied for networks which support dynamic packet headers modifications [16], [17], [40], source matching [18], [19], [27], or per-packet randomization on routers [20], [21].

However, all the above results are either limited in the number of link failures, the graph type they support, or the technology they assume. To the best of our knowledge, the only general approach to compute perfectly resilient routings on arbitrary graphs and under arbitrary failures is [26], which is the closest work to ours. One exception is Grafting [41], however, this algorithm does not provide any formal guarantees. The authors of [26] present a BDD-based method to synthesize resilient forwarding tables from scratch using binary decision diagrams. In contrast, we in this paper propose a novel repair methodology allowing us to fix ill-defined routing entries, and we also suggest heuristic approaches that can generate close-to-resilient routing tables. Being able to repair existing tables is not only practically relevant, but also enables a new and significantly faster methodology to synthesize entire tables, as we have shown in this paper empirically: using heuristics to fill tables and then improve them formally, using BDDs, is orders of magnitudes more efficient than the approach from [26] in terms of CPU time. We are not aware of other tools (except for the one we compare with) supporting a fully automated synthesis of perfectly resilient routings.

More generally, great efforts have been made over the last years by the networking community, to automate and simplify the operation of communication networks. Many interesting synthesis and formal methods approaches have been presented for other aspects of networking [42]–[52]. For example, SyNET [42] synthesizes correct network configurations for routing protocols such as BGP and OSPF, and AalWines [34] provides an automated what-if verification of the policy-compliance of routes under multiple failures in MPLS networks. There are also tools to automatically and correctly update network configurations, such as NetComplete [50] and AllSynth [53], which ensure policy compliance.

## VII. Conclusion

We presented an automated approach to synthesize and repair fast re-routing tables for highly dependably communication networks. Our tool, SyRep, is significantly faster compared to the state-of-the-art solution, as it introduces the verify and repair method for fixing of existing forwarding tables. This hence also supports a hybrid approach where first "fairly resilient" tables can quickly (in polynomial time) be computed using heuristics, which can then be improved (repaired) efficiently to provide provable resilience. Our method scales well for two link failures and becomes slower for three link failures. Even though for many ISP networks, considering two to three concurrent link failures seems sufficient, scalability for larger sets of failed links is still to be explored.

On the other hand, our repair method seems promising for dynamically changing networks. If we already have a resilient routing and the networks introduces (or removes) an additional link or even a node, our method should be able to fill in (or correct) the routing tables in the area where the network has changed, while preserving most of the routing information that is not affected by this change. Experimental evaluation of the repair method for dynamically changing networks will be a part of our future work.

There exist other heuristic approaches for routing synthesis like e.g. Grafting [41], which however do not guarantee to output perfectly resilient routings but they can be also used as input to our method as we are often able to repair such routing tables.

As another future work, it will be interesting to extend our tool to provide additional desirable properties beyond connectivity, e.g., accounting for link utilization or congestion along backup paths.

## Acknowledgment

Research supported by the Vienna Science and Technology Fund (WWTF) project, grant 10.47379/ICT19045 (WHATIF), 2020-2024, by the Danish DFF project QASNET, and by the Villum Investigator Grant S4OS, 2020–2027.

## References

[1] A. Atlas and A. Zinin, "Basic specification for IP fast reroute: Loop-free alternates," *RFC*, vol. 5286, pp. 1–31, 2008.

[2] G. Rétvári, J. Tapolcai, G. Enyedi, and A. Császár, "IP fast reroute: Loop free alternates revisited," in *INFOCOM*. IEEE, 2011, pp. 2948–2956.

[3] A. Bashandy, C. Filsfils, B. Decraene, S. Litkowski, P. Francois, D. Voyer, F. Clad, and P. Camarillo, "Topology independent fast reroute using segment routing," *Working Draft*, 2018.

[4] P. Pan, G. Swallow, and A. Atlas, "Fast reroute extensions to RSVP-TE for LSP tunnels," *RFC*, vol. 4090, pp. 1–38, 2005.

[5] J. S. Jensen, T. B. Krøgh, J. S. Madsen, S. Schmid, J. Srba, and M. T. Thorgersen, "P-Rex: Fast verification of MPLS networks with multiple link failures," in *CoNEXT*. ACM, 2018, pp. 217–227.

[6] K.-T. Foerster, M. Parham, M. Chiesa, and S. Schmid, "TI-MFA: Keep calm and reroute segments fast," in *IEEE INFOCOM 2018-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE, 2018, pp. 415–420.

[7] P. François, C. Filsfils, A. Bashandy, B. Decraene, S. Litkowski *et al.*, "Topology independent fast reroute using segment routing," 2014.

[8] Switch Specification 1.3.1, "OpenFlow," in *https://bit.ly/2VjOO77*, 2013.

[9] M. Chiesa, R. Sedar, G. Antichi, M. Borokhovich, A. Kamisinski, G. Nikolaidis, and S. Schmid, "PURR: a primitive for reconfigurable fast reroute: hope for the best and program for the worst," in *CoNEXT'19*. ACM, 2019, pp. 1–14.

[10] M. Chiesa, A. Kamisiński, J. Rak, G. Rétvári, and S. Schmid, "A survey of fast recovery mechanisms in the data plane," *TechRxiv*, May 2020.

[11] A. Markopoulou, G. Iannaccone, S. Bhattacharyya, C. Chuah, Y. Ganjali, and C. Diot, "Characterization of failures in an operational IP backbone network," *IEEE/ACM Trans. Netw.*, vol. 16, no. 4, pp. 749–762, 2008.

[12] R. Govindan, I. Minei, M. Kallahalla, B. Koley, and A. Vahdat, "Evolve or die: High-availability design principles drawn from googles network infrastructure," in *SIGCOMM*, 2016, p. 58–72.

[13] P. Gill, N. Jain, and N. Nagappan, "Understanding network failures in data centers: measurement, analysis, and implications," in *SIGCOMM*. ACM, 2011, pp. 350–361.

[14] J. Liu, A. Panda, A. Singla, B. Godfrey, M. Schapira, and S. Shenker, "Ensuring connectivity via data plane mechanisms," in *NSDI*, 2013.

[15] B. Yang, J. Liu, S. Shenker, J. Li, and K. Zheng, "Keep forwarding: Towards k-link failure resilient routing," in *INFOCOM*, 2014, pp. 1617–1625.

[16] M. Canini, P. Kuznetsov, D. Levin, and S. Schmid, "A distributed and robust SDN control plane for transactional network updates," in *INFOCOM*. IEEE, 2015, pp. 190–198.

[17] M. Borokhovich, L. Schiff, and S. Schmid, "Provable data plane connectivity with local fast failover: introducing openflow graph algorithms," in *HotSDN*. ACM, 2014, pp. 121–126.

[18] M. Chiesa, I. Nikolaevskiy, A. Panda, A. V. Gurtov, M. Schapira, and S. Shenker, "Exploring the limits of static failover routing (v4)," vol. abs/1409.0034.v4, 2016.

[19] M. Chiesa, I. Nikolaevskiy, S. Mitrovic, A. Panda, A. V. Gurtov, A. Madry, M. Schapira, and S. Shenker, "The quest for resilient (static) forwarding tables," in *INFOCOM*, 2016, pp. 1–9.

[20] M. Chiesa, A. V. Gurtov, A. Madry, S. Mitrovic, I. Nikolaevskiy, M. Schapira, and S. Shenker, "On the resiliency of randomized routing against multiple edge failures," in *ICALP*, 2016, pp. 134:1–134:15.

[21] G. Bankhamer, R. Elsässer, and S. Schmid, "Local fast rerouting with low congestion: A randomized approach," in *ICNP*, 2019.

[22] K.-T. Foerster, J. Hirvonen, Y.-A. Pignolet, S. Schmid, and G. Tredan, "On the feasibility of perfect resilience with local fast failover," in *Proc. SIAM APOCS*, 2021, pp. 55–69.

[23] J. Feigenbaum, B. Godfrey, A. Panda, M. Schapira, S. Shenker, and A. Singla, "Brief announcement: on the resilience of routing tables," in *PODC*. ACM, 2012, pp. 237–238.

[24] T. Holterbach, E. C. Molero, M. Apostolaki, A. Dainotti, S. Vissicchio, and L. Vanbever, "Blink: Fast connectivity recovery entirely in the data plane," in *USENIX NSDI*, 2019, pp. 161–176.

[25] W. Dai, K.-T. Foerster, and S. Schmid, "A tight characterization of fast failover routing: Resiliency to two link failures is possible," in *Proc. ACM SPAA*, 2023, p. 153–163.

[26] C. Gyorgyi, K. Larsen, S. Schmid, and J. Srba, "SyPer: Synthesis of perfectly resilient local fast re-routing rules for highly dependable networks," in *IEEE INFOCOM'24*. IEEE, 2024, pp. 1–10, to appear.

[27] M. Chiesa, I. Nikolaevskiy, S. Mitrovic, A. V. Gurtov, A. Madry, M. Schapira, and S. Shenker, "On the resiliency of static forwarding tables," *IEEE/ACM Trans. Netw.*, vol. 25, no. 2, pp. 1133–1146, 2017.

[28] M. Borokhovich, C. Rault, L. Schiff, and S. Schmid, "The show must go on: Fundamental data plane connectivity services for dependable sdns," *Computer Communications*, vol. 116, pp. 172–183, 2018.

[29] C. Y. Lee, "Representation of switching circuits by binary-decision programs," *The Bell System Technical Journal*, vol. 38, no. 4, pp. 985–999, 1959.

[30] Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Transactions on Computers*, vol. C-35, no. 8, pp. 677–691, 1986.

[31] S. Knight, H. X. Nguyen, N. Falkner, R. Bowden, and M. Roughan, "The internet topology zoo," *IEEE Journal on Selected Areas in Communications*, vol. 29, no. 9, pp. 1765–1775, 2011.

[32] M. Borokhovich and S. Schmid, "How (not) to shoot in your foot with SDN local fast failover - A load-connectivity tradeoff," in *OPODIS*, 2013, pp. 68–82.

[33] B. Stephens, A. L. Cox, and S. Rixner, "Plinko: Building provably resilient forwarding tables," in *HotTopics in Networks*, 2013, pp. 1–7.

[34] P. G. Jensen, D. Kristiansen, S. Schmid, M. K. Schou, B. C. Schrenk, and J. Srba, "AalWiNes: A fast and quantitative what-if analysis tool for mpls networks," in *ACM CoNEXT*, 2020, p. 474–481.

[35] F. Somenzi, "CUDD: CU decision diagram package release 3.0.0," *University of Colorado at Boulder*, 2015. [Online]. Available: http://vlsi.colorado.edu/ fabio/CUDD/

[36] "Omega Python package," 2023. [Online]. Available: https://github.com/tulip-control/omega

[37] C. Busch, S. Surapaneni, and S. Tirthapura, "Analysis of link reversal routing algorithms for mobile ad hoc networks," in *ACM SPAA*, 2003, p. 210–219.

[38] E. Gafni and D. P. Bertsekas, "Distributed algorithms for generating loop-free routes in networks with frequently changing topology," *IEEE Trans. Communications*, vol. 29, no. 1, pp. 11–18, 1981.

[39] J. Tapolcai, "Sufficient conditions for protection routing in ip networks," *Optimization Letters*, vol. 7, no. 4, pp. 723–730, 2013.

[40] T. Elhourani, A. Gopalan, and S. Ramasubramanian, "IP fast rerouting for multi-link failures," in *INFOCOM*, 2014.

[41] K.-T. Foerster, A. Kamisinski, Y.-A. Pignolet, S. Schmid, and G. Trédan, "Grafting arborescences for extra resilience of fast rerouting schemes," in *INFOCOM*. IEEE, 2021, pp. 1–10.

[42] A. El-Hassany, P. Tsankov, L. Vanbever, and M. Vechev, "Network-wide configuration synthesis," in *CAV'17*. Springer, 2017, pp. 261–281.

[43] T. Schneider, R. Schmid, and L. Vanbever, "On the complexity of network-wide configuration synthesis," in *ICNP*. IEEE, 2022, pp. 1–11.

[44] L. Beurer-Kellner, M. Vechev, L. Vanbever, and P. Veličković, "Learning to configure computer networks with neural algorithmic reasoning," *Advances in Neural Information Processing Systems*, vol. 35, pp. 730–742, 2022.

[45] K. G. Larsen, A. Mariegaard, S. Schmid, and J. Srba, "AllSynth: A BDD-based approach for network update synthesis," in *Science of Computer Programming (SCICO)*, vol. 230, 2023.

[46] S. Steffen, T. Gehr, P. Tsankov, L. Vanbever, and M. Vechev, "Probabilistic verification of network configurations," in *Proc. ACM SIGCOMM*, 2020, p. 750–764.

[47] S. Prabhu, K. Y. Chou, A. Kheradmand, B. Godfrey, and M. Caesar, "Plankton: Scalable network configuration verification through model checking," in *17th USENIX Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, 2020, pp. 953–967.

[48] J. McClurg, H. Hojjat, P. Černỳ, and N. Foster, "Efficient synthesis of network updates," in *ACM Sigplan Notices*, vol. 50, no. 6. ACM, 2015, pp. 196–207.

[49] B. Finkbeiner, M. Gieseking, J. Hecking-Harbusch, and E.-R. Olderog, "Model checking data flows in concurrent network updates (full version)," *arXiv preprint arXiv:1907.11061*, 2019.

[50] A. El-Hassany, P. Tsankov, L. Vanbever, and M. Vechev, "Netcomplete: Practical network-wide configuration synthesis with autocompletion," in *USENIX NSDI*, 2018, pp. 579–594.

[51] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker, "Netkat: Semantic foundations for networks," *ACM sigplan notices*, vol. 49, no. 1, pp. 113–126, 2014.

[52] S. Schmid, M. K. Schou, J. Srba, and J. Vanerio, "R-MPLS: Recursive protection for highly dependable MPLS networks," in *Proc. ACM CoNEXT*, 2022, p. 276–292.

[53] K. G. Larsen, A. Mariegaard, S. Schmid, and J. Srba, "AllSynth: Transiently correct network update synthesis accounting for operator preferences," in *Proc. TASE*, 2022, pp. 344–362.