R-MPLS: Recursive Protection for Highly Dependable MPLS Networks

Stefan Schmid
TU Berlin and University of Vienna
Germany and Austria

Morten Konggaard Schou
Dept. of Computer Science, Aalborg University
Denmark

Jiří Srba
Dept. of Computer Science, Aalborg University
Denmark

Juan Vanerio
Faculty of Computer Science, University of Vienna
Austria

ABSTRACT

Most modern communication networks feature fast rerouting mechanisms in the data plane. However, design and configuration of such mechanisms even under multiple failures is known to be difficult. In order to increase the resilience of the widely deployed MPLS networks, we propose R-MPLS, an alternative link protection mechanism for MPLS networks that uses *recursive* protection and can route around *multiple* simultaneously failed links. Our new R-MPLS approach comes with strong theoretical underpinnings, is implementable in a fully distributed way and executable on existing MPLS hardware, and formally guarantees that no forwarding loops are introduced. We implement our R-MPLS protection in an automated tool which overcomes the complexity of configuring such resilient network data planes, and report on the benefits of recursive protection in realistic network topologies. We find that R-MPLS significantly increases network robustness against multiple failures, with only moderate increase in the number of forwarding rules and communication overhead (both comparable to industry-standards like RSVP-TE FRR).

CCS CONCEPTS

• **Networks** → **Network algorithms**; **Network reliability**.

KEYWORDS

Network Reliability, Network Algorithms, Fast Reroute, MPLS

ACM Reference Format:
Stefan Schmid, Morten Konggaard Schou, Jiří Srba, and Juan Vanerio. 2022. R-MPLS: Recursive Protection for Highly Dependable MPLS Networks. In *The 18th International Conference on emerging Networking EXperiments and Technologies (CoNEXT '22), December 6–9, 2022, Roma, Italy.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3555050.3569140

1 INTRODUCTION

Network failures are inevitable: network interfaces can go down and devices crash at any time [28, 43]. Today, especially link failures are common, and with the increasing scale of communication networks, failures are likely to become more frequent [23]. In order to deal with such failures and provide the required high degree of dependability, modern networks rely on control software whose responsibility is to ensure connectivity despite these unreliable components. In particular, most mission-critical communication networks today feature fast rerouting (FRR) mechanisms in the data plane [10] which allow routers to locally and hence quickly forward traffic to alternative paths.

However, the design of fast rerouting mechanisms providing a high degree of resilience is known to be challenging, and continues to attract significant attention from the research community [10, 12, 20, 24, 35]. In particular, since routers need to react to failures *locally*, these decisions are taken without knowledge of potential failures downstream. The additional failures, however, may lead to incorrect forwarding behaviors and threaten reachability.

Many major network outages have been reported over the last years [14, 15]. While sometimes already a single link failure can lead to undesired network behaviors [7], with the increasing network scale and due to shared risk link groups, operators now even have to plan for multiple failures simultaneously [49]. Also, multiple link failures have already been studied in the literature intensively before [5, 12, 16, 37]. There results for achieving perfect resilience for many clasess of topologies and different kinds of networks. None of them is readily implementable in MPLS.

This paper is motivated by observing an opportunity to significantly improve the resilience provided by fast rerouting mechanisms. In particular, we consider the widely deployed networks based on Multiprotocol Label Switching (MPLS) [40, 44]. For example, MPLS networks are popular among ISPs for traffic engineering purposes. The fast rerouting mechanism used in MPLS relies on stacks of labels in the packet header, where a label pushed on the stack allows to route packets around failed links, creating a "backup tunnel" [38]. MPLS FRR allows protecting against individual link and node failures, and has been successfully used for two decades already. It has recently regained attention for supporting fast what-if analysis [25, 26].

Two common protection methods are standardized on MPLS for protection of traffic engineering tunnels [38]: *one-to-one backup* where one backup label switched path (LSP) is established for each protected LSP in such a way that the former intersects the latter at one of the downstream nodes and *facility backup* where each backup LSP is established to protect a set of many primary LSPs that share the same outgoing interface or next-hop, intersecting the primary paths at a shared downstream node right after the

CoNEXT '22, December 6–9, 2022, Roma, Italy

ACM ISBN 978-1-4503-9508-3/22/12.
https://doi.org/10.1145/3555050.3569140

failed link/node. However, while conceptually simple, MPLS FRR procedures are designed to protect against a single link or node failure (as a weakness and in contrast to our method).

Our main contribution is a generalization of the MPLS fast reroute mechanism, R-MPLS, which supports a *recursive* protection scheme, where additional labels are pushed on the stack whenever a packet encounters another failure, essentially creating "nested tunnels". This generalization is non-trivial: if done naively, nested tunnels may quickly lead to forwarding loops, which is a major concern of operators. Also, while R-MPLS may increase the header size, this overhead occurs only when it is needed due to multiple link failures. To this end, we believe that our approach is in line with other trends in networking, such as IPv6 or segment routing [19], which require larger headers. Although MPLS forwarding requires exact matching on labels, which is less expensive than IP ternary matches, the number of routing entries and the number of communications required to compute a protection are critical parameters to scale with the network size. Our recursive protection mechanism R-MPLS then substantially improves the resilience of the network to multiple link failures without the risk of introducing forwarding loops, while keeping low the memory and the communications overheads.

By recursively building protection tunnels, R-MPLS is able to provide alternative paths from a single router, as well as protection paths for other protection paths, neither of which is possible with standard MPLS protection mechanisms. Figure 1 exemplifies the protections provided by R-MPLS. A main path from v_0 to v_1 can be backed up with a protection path via v_2. If (v_0, v_2) is also unavailable then the main path is protected by another protection path via v_3, which rejoins the original protection path at v_2. Additionally also link (v_2, v_1) has its own protection via v_4. As a result, R-MPLS can find a path from v_0 to v_1 even when links (v_0, v_1), (v_0, v_2) and (v_2, v_1) simultaneously fail. R-MPLS's improvement is due to recursive link failure protection by design.

We evaluate the benefits of such recursive protection empirically on a large number of real network topologies, also comparing against the state-of-the-art mechanism to achieve multi-failure resiliency. We find that R-MPLS can indeed significantly increase the network resilience against multiple link failures at minimal overheads. Another attractive feature of R-MPLS is that it is compatible with and can be employed on top of any existing MPLS data plane and protocol, such as RSVP and LDP.

As a contribution to the research community, in order to ensure reproducibility and support follow-up work, we make all our experimental artifacts and implementations publicly available (as open-source code) [41].

2 MPLS NETWORK MODEL

Let us first formally define a general data plane model of MPLS networks. This model is based on prior formal models of MPLS [25, 26], though we restrict the model to the widely used per-platform label space.

In the model, an MPLS network consists of a topology and forwarding rules, where the topology is composed of routers and directed links. Bidirectional links, which are common in real networks, are modelled by two directed links. Figure 2a gives a small

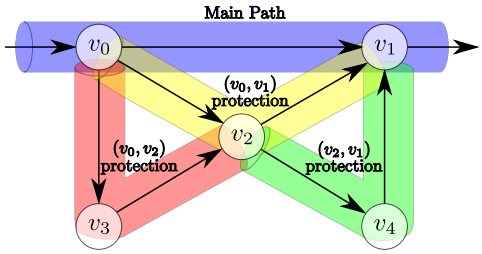


Figure 1: Example R-MPLS protection.

example topology. In the figure, links in_1, in_2 and out_1 are connected to the outside of the MPLS domain, modelled using a designated external node not shown in the figure.

Definition 1. A *network topology* is a directed multigraph (V, E, src, tgt) where V is a set of *routers*, E is a set of *links* between routers, $src : E \to V$ assigns the *source router* to each link, and $tgt : E \to V$ assigns the *target router*.

A *path* p in the directed multigraph is a sequence of links $e_1 \dots e_n \in E^*$ with $tgt(e_i) = src(e_{i+1})$ for $1 \leq i < n$. The path is *simple* if all its routers are distinct. Define $tgt(p) \triangleq tgt(e_n)$.

We assume that links in the network can fail. This is modelled by a set $F \subseteq E$ of *failed* links. In our model, this set does not change for the duration of time considered. In other words we look at a snapshot of the data plane after some failures happen and before the control plane computes new paths. A link is *active* if it belongs to $E \setminus F$. We sometimes call F the failure scenario.

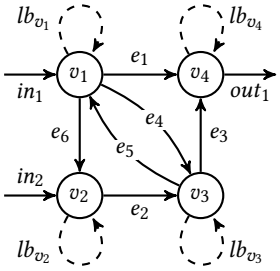
Forwarding in an MPLS network is accomplished using labels in the packet header. We denote the set of MPLS labels used in the network by L. Packet headers are modified using pop, swap and push operations. For a set of MPLS labels L, we define the set of *MPLS operations* on packet headers as $Op(L) = \{\mathsf{swap}(\ell) \mid \ell \in L\} \cup \{\mathsf{push}(\ell) \mid \ell \in L\} \cup \{\mathsf{pop}\}$.

Each router has a mapping from labels to forwarding entries. Figure 2c shows an example of a forwarding table for the topology in Figure 2a. The tables encode two Label Switched Paths (LSPs) from v_1 resp. v_2 and exiting at out_1.

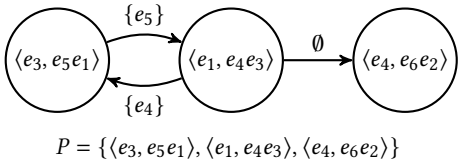
For ease of presentation, the formal model does not include how a packet enters the MPLS domain, but it is easily extendable by also mapping these external interfaces to forwarding entries. In the formal definition, these mappings for all routers are joined into one forwarding table τ:

Definition 2. An *MPLS network* $N = (V, E, src, tgt, L, \tau)$ is a tuple where (V, E, src, tgt) is a network topology, L is a finite set of MPLS labels, and $\tau : V \times L \to 2^{\mathbb{N} \times E \times Op(L)^+}$ is the forwarding table.

For every router-label pair $(v, \ell) \in V \times L$, the forwarding table returns a set $\tau(v, \ell) = \{(pr_1, e_1, \omega_1), \dots, (pr_m, e_m, \omega_m)\}$ of *forwarding entries* where, for all $1 \leq j \leq m$, pr_j is the priority, e_j is the outgoing link such that $src(e_j) = v$, and $\omega_j \in Op(L)^+$ is a nonempty sequence of MPLS operations to be performed on the packet header. We say that a forwarding entry is *active*, if its outgoing link is active.



(a) Network topology.



$P = \{\langle e_3, e_5e_1\rangle, \langle e_1, e_4e_3\rangle, \langle e_4, e_6e_2\rangle\}$

(b) Protection graph for the protections P is used for loop avoidance.

Router	Label	Prio.	e_{out}	Operation
v_1	01	1	e_1	swap(02)
v_2	05	1	e_2	swap(06)
v_3	06	1	e_3	swap(07)
v_4	02	1	out_1	pop
	07	1	out_1	pop

(c) Forwarding table, before R-MPLS, encoding flows $v_1 \to v_4$ and $v_2 \to v_3 \to v_4$.

Router	Label	Prio.	e_{out}	Operation
v_1	01	1	e_1	swap(02)
		2	lb_{v_1}	swap(02) ∘ push(10)
	10	1	e_4	swap(11)
		2	lb_{v_1}	swap(11) ∘ push(40)
	40	1	e_6	swap(41)
	31	1	e_1	pop
		2	lb_{v_1}	pop ∘ push(10)
v_2	05	1	e_2	swap(06)
	41	1	e_2	pop
v_3	06	1	e_3	swap(07)
		2	lb_{v_3}	swap(07) ∘ push(30)
	11	1	e_3	pop
		2	lb_{v_3}	pop ∘ push(30)
	30	1	e_5	swap(31)
v_4	02	1	out_1	pop
	07	1	out_1	pop

(d) Forwarding table after R-MPLS protects links e_1, e_3, and e_4 with labels (10, 11), (30, 31), resp. (40, 41). Gray rows are excluded to avoid loops.

$(in_1, 01)(e_1, 02)(out_1, \varepsilon)$	$F = \emptyset$
$(in_1, 01)(lb_{v_1}, 10 \circ 02)(e_4, 11 \circ 02)(e_3, 02)(out_1, \varepsilon)$	$F = \{e_1\}$
$(in_1, 01)(lb_{v_1}, 10 \circ 02)(lb_{v_1}, 40 \circ 11 \circ 02)(e_6, 41 \circ 11 \circ 02)(e_2, 11 \circ 02)(e_3, 02)(out_1, \varepsilon)$	$F = \{e_1, e_4\}$
$(in_1, 01)(lb_{v_1}, 10 \circ 02)(e_4, 11 \circ 02)(lb_{v_3}, 30 \circ 02)(e_5, 31 \circ 02)(lb_{v_1}, 10 \circ 02)\ldots$	$F = \{e_1, e_3\}$

(e) Traces through the network in different failures scenarios. The last looping trace (notice the repeated hop $(lb_{v_1}, 10 \circ 02)$) is avoided by excluding the grayed forwarding rules in Figure 2d.

Figure 2: A simple network with a routing table before and after R-MPLS protection.

The semantics of a set of forwarding entries is to choose an active entry with the highest priority (lowest natural number). If several active entries have the same highest priority, we nondeterministically pick one, hence abstracting away from various specific routing policies like e.g. ECMP that allow splitting a flow along multiple paths.

Definition 3. For a set of failed links $F \subseteq E$ we define the *active forwarding table* $\tau_F : V \times L \to 2^{E \times Op(L)^+}$ as $\tau_F(v, \ell) = \{(e, \omega) \mid (pr, e, \omega) \in \tau(v, \ell),\ e \in E \setminus F \text{ and } pr = pr_{min}\}$, where pr_{min} is the highest priority (minimal value) of an active forwarding entry in $\tau(v, \ell)$, or define $\tau_F(v, \ell) = \emptyset$ if $\tau(v, \ell)$ has no active forwarding entries given F.

As an example with a single protection entry, if $\tau(v_1, 01) = \{(1, e_1, \mathsf{swap}(02)), (2, e_2, \mathsf{swap}(02) \circ \mathsf{push}(10))\}$, then given the failure scenario $F = \{e_1\}$, the corresponding entry in the active forwarding table is $\tau_F(v_1, 01) = \{(e_2, \mathsf{swap}(02) \circ \mathsf{push}(10))\}$. In this case forwarding is deterministic, since $\tau_F(v_1, 01)$ is a singleton set.

Definition 4. The semantics of MPLS operations is a partial *header rewrite function* $\mathcal{H} : L^* \times Op(L)^* \rightharpoonup L^*$, where $\omega, \omega' \in Op(L)^*$, $h \in L^*$ and ε is the empty sequence of operations:

$$\mathcal{H}(h, \omega) = \begin{cases} h & \text{if } \omega = \varepsilon \\ \mathcal{H}([op](\ell) \circ h', \omega') & \text{if } \omega = op \circ \omega' \text{ and } h = \ell \circ h' \\ & \text{with } \ell \in L,\ h' \in L^* \\ \textit{undefined} & \textit{otherwise} \end{cases}$$

where we define $[\mathsf{pop}](\ell) = \varepsilon$, $[\mathsf{swap}(\ell')](\ell) = \ell'$ and $[\mathsf{push}(\ell')](\ell) = \ell'\ell$ for all $\ell, \ell' \in L$.

As an example, applying the operation sequence $\mathsf{swap}(02) \circ \mathsf{push}(10)$ to the header 01, yields $\mathcal{H}(01, \mathsf{swap}(02) \circ \mathsf{push}(10)) = 10 \circ 02$. The forwarding of a packet proceeds by (i) selecting an entry from the active forwarding table that corresponds to the top-most label on the packet label-stack, (ii) applying the header operations, and (iii) sending the packet on the outgoing link.

Definition 5. A *trace* in a network $N = (V, E, src, tgt, L, \tau)$, given a set of failed links $F \subseteq E$, is any (finite or infinite) sequence of link-header pairs $(e_1, h_1)(e_2, h_2)\ldots$ with each $(e_i, h_i) \in (E \setminus F) \times L^*$, where for each $i > 1$, $h_i = \mathcal{H}(h_{i-1}, \omega)$ for some $(e_i, \omega) \in \tau_F(tgt(e_{i-1}), head(h_{i-1}))$, where $head(h)$ is the top (left-most) label of h.

Figure 2e shows traces under different failure scenarios using the forwarding table in Figure 2d. The first trace is a primary path in the original data plane. The next two use R-MPLS protection in two different failure scenarios. The last one shows looping behavior, where the gray parts correspond to the gray forwarding entries in Figure 2d, which are excluded from the forwarding table by our loop avoidance algorithm.

3 R-MPLS PROTECTION

Our recursive MPLS protection (R-MPLS) is designed as a protection layer that enhances an existing data plane. That is, it takes a topology and a data plane as inputs and returns an augmented version of the same data plane as output. This operation is performed regardless of the protocols involved in the creation of the original one. Hence our R-MPLS implementation can be used for postprocessing and data plane augmentation.

We generalize the notion of link protection and node protection (Definition 6) and address which forwarding entries can be protected by a given protection path in Section 3.1. Next we solve in Section 3.2 the issue of avoiding the introduction of forwarding loops—which occurs if naively applying recursive protection. The high-level pseudocode of our protection algorithm is described in Algorithm 2 and Section 3.3, while Section 3.5 provides details on its distributed implementation.

3.1 Protectable Forwarding Entries

Definition 6. A *protection* is a pair $\langle e, p\rangle$ where $e \in E$ is the link being protected and p is a simple path $e_1 \ldots e_n \in (E \setminus \{e\})^*$, with $src(e_1) = src(e)$.

If $tgt(p) = tgt(e)$, then $\langle e, p\rangle$ is a *link protection*. Figure 3 shows a network with two main LSPs (dotted and dashed lines), and three protections P for e_1. The protection $\langle e_1, e_3e_4\rangle$ is a link protection. A *node protection* routes around not just the failing link, but also the neighboring node. In the example, $\langle e_1, e_3e_7\rangle$ is a node protection. Note that this protection can only be used for the LSP going through v_5. The other LSP (going through v_3) has no node protection, but we can protect it by a path merging further down the LSP, namely the protection $\langle e_1, e_3e_7e_8\rangle$.

Not all forwarding entries can be protected by a given protection. We only install protections on forwarding entries where the original path merges with the protection path.

Definition 7. An entry $(pr, e, \omega) \in \tau(src(e), \ell)$ for a label $\ell \in L$ is *protectable* by a protection $\langle e, p\rangle$ using operations $\omega' \in Op(L)^*$, if there exists $e' \in E$ with $tgt(e') = tgt(p)$ such that for all $h \in L^*$ there is a trace

$$(e, \mathcal{H}(\ell \circ h, \omega)) \ldots (e', \mathcal{H}(\ell \circ h, \omega'))$$

in the network under no failures.

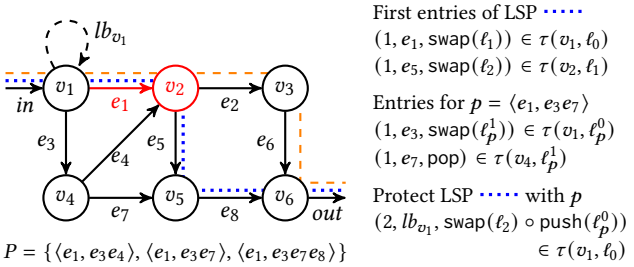


Figure 3: A network with two main forwarding paths (dotted and dashed line), and three protections P of e_1. Right side shows node protection of the dotted LSP.

The R-MPLS algorithm only installs protections for protectable entries, and it needs to know the operation sequence ω' to use. For link protection, all entries are protectable using $\omega' = \omega$.

In Figure 3, the entry $(1, e_1, \mathsf{swap}(\ell_1)) \in \tau(v_1, \ell_0)$ is protectable by $\langle e_1, e_3e_7\rangle$ using $\omega' = \mathsf{swap}(\ell_2)$, since $(e_1, \ell_1 \circ h)(e_5, \ell_2 \circ h)$ is a valid network trace for all $h \in L^*$. Note that applying $\omega' = \mathsf{swap}(\ell_2)$ before pushing the protection label, ensures that a packet using the protection path arrives at v_5 with the same header $\ell_2 \circ h$ as if using the main LSP.

3.2 Loop Avoidance

When using a protection $\langle e, p\rangle$, in case of a failure of a link e' on the path p, the R-MPLS algorithm allows to recursively switch to a new protection $\langle e', p'\rangle$. As shown in the last trace in Figure 2e this can sometimes result in a loop. In this example the failure of e_1 causes the packet to use the backup path e_4e_3, where the failure of e_3 makes the packet switch to the backup path e_5e_1, hence looping back to the failed e_1 and the backup path e_4e_3.

To avoid introducing forwarding loops, we need to understand the interactions between different protection paths, and then avoid installing the recursive protection in some cases. For this, we use the following graph.

Definition 8. Given a set of protections P, we define a *protection graph* with nodes P and edges called *protection-pairs* such that there is an edge $(\langle e, p\rangle, \langle e', p'\rangle) \in$ *protection-pairs* whenever the protected link e' is on the protection path p, and p' merges downstream on p, i.e. there is $e_i = e'$ and e_j with $j \geq i$ and $tgt(e_j) = tgt(p')$ such that $p = e_1 \ldots e_i \ldots e_j \ldots e_n$. Moreover, we annotate using the function α every such edge with the set of links that must be active before the link e' is used, i.e. $\alpha(\langle e, p\rangle, \langle e', p'\rangle) = \{e_1, e_2, \ldots, e_{i-1}\}$ where $p = e_1e_2 \ldots e_{i-1}e_i \ldots e_n$ and $e' = e_i$.

Figure 2b gives the protection graph for the running example, when all link protections are used. The edges in the graph are labelled by their α annotations. We use the annotation to keep track of which links on the protection path p must be active, since the packet has already traversed them, when the failure of e_i makes the packet move onto the protection path p'.

Algorithm 1 Computing bad protection pairs to avoid loops

```
1:  function FindBadProtectionPairs(protections P)
2:      Let G be the protection graph of P (see Definition 8)
3:      B ← ∅                                  ▷ Initialize bad protection-pairs
4:      while there is a bad simple cycle of length n in G do
5:          if n > 2 then
6:              Pick an edge (⟨e, p⟩, ⟨e′, p′⟩) from the cycle
7:              Update B ← B ∪ {(⟨e, p⟩, ⟨e′, p′⟩)}
8:              Remove the edge (⟨e, p⟩, ⟨e′, p′⟩) from G
9:          else
10:             Add both protection pairs in the cycle to B
                and remove the two edges from G
11:     return B
```

Note that in our example protection graph in Figure 2b, the cycle between $\langle e_1, e_4e_3\rangle$ and $\langle e_3, e_5e_1\rangle$ corresponds to a possible forwarding loop in the failure scenario $F = \{e_1, e_3\}$ as shown by the last trace in Figure 2e. We define such bad cycles that lead to forwarding loops in some failure scenarios.

Definition 9. A *bad simple cycle* in the protection graph is a sequence of distinct protections $\langle e_1, p_1\rangle \dots \langle e_n, p_n\rangle$ s.t.
$C \triangleq \big\{(\langle e_1, p_1\rangle, \langle e_2, p_2\rangle), \dots, (\langle e_{n-1}, p_{n-1}\rangle, \langle e_n, p_n\rangle),$
$(\langle e_n, p_n\rangle, \langle e_1, p_1\rangle)\big\} \subseteq$ *protection-pairs*
and $\{e_1, \dots, e_n\} \cap (\bigcup_{(\langle e,p\rangle, \langle e',p'\rangle) \in C} \alpha(\langle e, p\rangle, \langle e', p'\rangle)) = \emptyset$.

The requirement in Definition 9 on the property of the cycle states that the paths in the cycle do not protect links that appear in the annotations of the protection-pairs involved in the cycle. If this property does not hold, a link is assumed to be both active and failed, hence the cycle does not correspond to a routing loop in any possible failure scenario.

Algorithm 1 removes edges, i.e. protection-pairs, from the protection graph, until there are no more bad simple cycles in the graph. The set of removed edges is returned as the bad protection-pairs, where R-MPLS should avoid adding recursive protection entries for these specific cases. We note that for cycles of length 3 and more, we only break one protection-pair on that cycle, while for shorter cycles we remove all of them (a minor optimization of the algorithm). In our example from Figure 2b, we identify the set $\{(\langle e_1, e_4e_3\rangle, \langle e_3, e_5e_1\rangle), (\langle e_3, e_5e_1\rangle, \langle e_1, e_4e_3\rangle)\}$ as the set of bad protection-pairs which form a cycle of length 2 and the links e_1 and e_3 do not appear on the annotations of the protection-pairs in the cycle.

We can find and eliminate all bad cycles using a depth-first-search approach starting from each protection, where the annotations are continuously checked to not intersect with the protected links on the search stack.

3.3 R-MPLS Algorithm

Given an existing MPLS network, with its own topology and forwarding tables, we initialize the execution of Algorithm 2 by adding *loopback* links to each router as an abstraction of instructing the router to run a packet through its forwarding processes again. The R-MPLS algorithm then installs the LSP for each protection (loop in Lines 4–10). We here give the protections as input to the algorithm. They can be computed e.g. as link or node protections along shortest paths. Each router along the protection allocates a local label $\ell^i_{\langle e,p\rangle}$ to it and records that the label is on the LSP of protection $\langle e, p\rangle$. It then creates new entries in its forwarding table to use the protection path. Notice that the last router in the path uses a pop instruction while the others just swap labels. No router along this path records a push instruction, as these LSPs are only used as protection paths.

Next, we compute a set of bad protection-pairs based on the protection graph using Algorithm 1. In order to avoid introducing forwarding loops, we disable the recursive protection for these specific pairs of protections.

After completing the previous process, each router proceeds to execute the loop in Lines 14–21 which augments the original forwarding table by adding lower priority entries. Note that this loop does not iterate over the new entries created inside the loop. For each previously existing highest priority forwarding entry, and for each protection that can protect that forwarding entry (Line 16), the router creates a new lower priority entry (Line 17). The new entry performs the operations ω' that will make the packet arrive at the merge point router with the same header as under the original forwarding. These operations are followed by pushing the label that encodes the protection path. The new protection entries forward to the router's loopback link. For link protection, the sequence ω' is just the original operations ω for the entry; for other protections this information needs to be retrieved from e.g. the control plane.

To achieve recursive protection, the same is done for the entries created on the loop of Lines 4–10, unless the protection that the incoming label encodes, paired with the protection we are about to use, are part of the bad protection-pairs. Again, we check if the entry is protectable by the protection, i.e. that the new protection intersects downstream with the current protection. The operation ω' is computed based on where the two protection paths intersect.

We apply Algorithm 2 on the network topology from Figure 2a and forwarding table in Figure 2c that encodes two flows. We use the link protections P in Figure 2b. The resulting forwarding table is shown in Figure 2d, where the links e_1, e_3, and e_4 are protected. Note that in this small example, no other links can have link protection due to the direction of the links; however, in real networks usually all links get protection paths. Figure 2e shows four possible traces under different failure scenarios. Notice how the third trace uses the recursive protection to recover from both e_1 and e_4 failing. Due to the failure of e_1, it first tries to use the protection path starting at e_4 encoded by the label 10. Using the loopback link, it then tests if link e_4 is also failing, and then it uses the path through e_6 until that path joins the first protection path at router v_3.

3.4 Recursive Link and Node Protection

Algorithm 2 takes the set of protections P as input. We now show two instantiations of computing P: link protection and node protection.

Given a network topology (V, E, src, tgt) where the links are annotated with weights, we compute for each link $e \in E$ the shortest path p from $src(e)$ to $tgt(e)$ in the graph $(V, E \setminus \{e\}, src, tgt)$, and if p exists add $\langle e, p\rangle$ to the set of protections P. This gives link protection of all links.

Algorithm 2 Recursive protection algorithm

Input: Network $N = (V, E, src, tgt, L, \tau)$, finite set of protections $P \subseteq (E \times E^*)$
Output: Protected network $N' = (V, E', src', tgt', L', \tau')$ with R-MPLS protection

1: $src' \leftarrow src$, $tgt' \leftarrow tgt$, $L' \leftarrow L$, $\tau' \leftarrow \tau$ ▷ Initialize N'
2: **for** $v \in V$ **do** create loopback link lb_v such that $src'(lb_v) = v$ and $tgt'(lb_v) = v$
3: $E' \leftarrow E \cup \{lb_v \mid v \in V\}$
4: **for** each protection $\langle e, p \rangle \in P$ **do** (let $e_1, \ldots, e_n$ be the links on the protection path p)
5: $e_0 \leftarrow lb_{src(e)}$ ▷ Start from loopback link
6: $\ell^0_{\langle e,p \rangle}, \ldots, \ell^{n-1}_{\langle e,p \rangle} \leftarrow$ fresh labels, add each $\ell^i_{\langle e,p \rangle}$ to L'
7: $protects(\ell^i_{\langle e,p \rangle}) \leftarrow \langle e, p \rangle$ for $0 \leq i < n$ ▷ Remember protection for label with the mapping $protects : (L' \setminus L) \to P$
8: **for** $i \in \{0, 1, \ldots, n-2\}$ **do**
9: $\tau'(tgt(e_i), \ell^i_{\langle e,p \rangle}) \leftarrow \{(1, e_{i+1}, \mathsf{swap}(\ell^{i+1}_{\langle e,p \rangle}))\}$ ▷ Use the labels to encode the protection path
10: $\tau'(tgt(e_{n-1}), \ell^{n-1}_{\langle e,p \rangle}) \leftarrow \{(1, e_n, \mathsf{pop})\}$ ▷ Pop the label on last hop
11: $bad\text{-}protection\text{-}pairs \leftarrow$ FindBadProtectionPairs(P) ▷ Call Algorithm 1
12: Let M be larger than any priority occurring in τ'
13: Let $\tau'_{min}(v, \ell) = \{(pr, e, \omega) \in \tau'(v, \ell) \mid pr = pr_{min}\}$ where pr_{min} is the highest priority in $\tau'(v, \ell)$
14: **for** $v \in V$, $\ell \in L'$, $(pr, e, \omega) \in \tau'_{min}(v, \ell)$ and $\langle e, p \rangle \in P$ **do** ▷ Iterate over routers, labels, entries and protections
15: **if** $\ell \in L$ **then** ▷ Protection of original data plane
16: **if** the entry $(pr, e, \omega) \in \tau'(v, \ell)$ is protectable by $\langle e, p \rangle$ using operations $\omega' \in Op(L)^*$ **then**
17: $\tau'(v, \ell) \leftarrow \tau'(v, \ell) \cup \{(M,\ lb_v,\ \omega' \circ \mathsf{push}(\ell^0_{\langle e,p \rangle}))\}$ ▷ Push backup path
18: **else** (let $\langle e', p' \rangle = protects(\ell)$, let $e'_1, \ldots, e'_n$ be the links on p', and let j be the index where $tgt(e'_j) = v$)
19: **if** $(\langle e', p' \rangle, \langle e, p \rangle) \notin bad\text{-}protection\text{-}pairs$ **and** there exists index i such that $tgt(e'_i) = tgt(p)$ and $i > j$ **then**
20: $\omega' \leftarrow \begin{cases} \mathsf{pop} & \text{if } i = n \\ \mathsf{swap}(\ell^i_{\langle e',p' \rangle}) & \text{otherwise} \end{cases}$
21: $\tau'(v, \ell) \leftarrow \tau'(v, \ell) \cup \{(M,\ lb_v,\ \omega' \circ \mathsf{push}(\ell^0_{\langle e,p \rangle}))\}$ ▷ Push recursive backup path
22: **return** $N' = (V, E', src', tgt', L', \tau')$

To compute node protection: for each $v \in V$ and $e, e' \in E$ with $tgt(e) = v = src(e')$, compute the shortest path p from $src(e)$ to $tgt(e')$ in the graph with v and all of v's incident edges removed. If p exists, add $\langle e, p \rangle$ to the set of protections P.

The standard FRR facility protection uses node protection when possible and link protection only as a fallback. We can achieve the recursive version of this, by adding both link and node protections to P, and then extending Line 16 and 19 to filter out link protections in case the entry is protectable by a node protection. The R-MPLS framework also allows for more general sets of protections, e.g. to optimize link capacity usage in failure scenarios.

3.5 Distributed R-MPLS Implementation

The pseudocode from Algorithm 2 and the computation of protections P for common protection schemes like link and node protection, can be implemented in a fully distributed fashion, and it is hence compatible with traditional MPLS routers. In particular, each router can compute the protection paths for each of its outgoing links. The topology knowledge required to compute paths is provided on traditional MPLS networks by the Interior Gateway Protocol (IGP), typically OSPF-TE [51] or ISIS-TE [33]. The information exchanged by the IGP is stored by each router in a local database, so no central entity with a complete view of the topology is required. Each router along the computed protection path is notified by the originating router, and then Lines 4–10 of Algorithm 2 are executed locally.

Notice that Line 11 and Line 18 (with the mapping defined on Line 7) require knowledge of the full protection path $p = e_1, \ldots, e_n$. To obtain this information, the intermediate routers along the path $src(e_2), \ldots, src(e_n)$ query $src(e_1)$, which is responsible for computing the protection path. Such a query can be performed e.g. using the RSVP Diagnose facility [52], by which any network element sends a request message to another router and inquires information about computed paths. This request uses only existing RSVP primitives so the communication can be implemented completely in software.

The loop on Lines 14–21 of Algorithm 2 requires only local operations on each router, when link protection is used. For node protections, Line 20 queries the merge point router for its label allocated to the protection, and Line 16 needs to query the next-hop router $tgt(e)$ for its forwarding entries of the top label left by ω. In other words, each router only needs to know about the labels of neighboring routers to implement link protection and their next-hop forwarding entries to implement node protection. No other information nor central controller is needed.

R-MPLS has then all the information available, when it finishes computing its protection paths. So, whichever path is provided by the underlying protocols, at present or in the future, as long as

R-MPLS finished computing its own (topology dependent only) protection paths, then the router can derive a protecting entry for each original data plane route, loop-freedom guaranteed; (though for node protections, Line 16 still needs to query the neighboring router to get ω'). And if or when new forwarding paths result from the underlying protocols, these can also be protected against link failures. Since the priority range is partitioned, all R-MPLS routing entries are guaranteed to have lower priority than all original ones, ensuring no interference on the networks' basic routing.

Execution of Algorithm 1 can also be implemented in a distributed fashion. Again, each router only performs computations for its own outgoing links. Routers query each other for the protections paths they have previously computed. The rest of the algorithm is computed locally from those elements. To ensure that computations made on Line 6 are identical on all routers, it suffices to assume a total order on the set of links and choose the protection-pair deterministically regardless of the router.

3.6 Properties of the R-MPLS Protection

We shall now argue that our R-MPLS protection preserves all the connections of the original MPLS data plane and does not introduce any forwarding loops. For this we first need to define the subset of traces that corresponds to a full run of a packet.

Definition 10. Let $N = (V, E, src, tgt, L, \tau)$ be an MPLS network and let $F \subseteq E$ be the set of failed links. A *maximum trace* in N under F is either any infinite trace or a finite trace that is not a prefix of any other trace in N under F.

Hence a finite trace $(e_1, h_1) \dots (e_n, h_n)$ is maximum if its last link-header pair (e_n, h_n) satisfies either $h_n = \varepsilon$, $\tau_F(tgt(e_n), head(h_n)) = \emptyset$, or $\mathcal{H}(h_n, \omega)$ is undefined for all $(e, \omega) \in \tau_F(tgt(e_n), head(h_n))$.

We now formally define the three properties of MPLS networks. Due to the support of nondeterministic forwarding, there is a difference between possibility and certainty of connectivity in a given scenario.

Definition 11. For a network $N = (V, E, src, tgt, L, \tau)$ and set of failed links $F \subseteq E$, define the predicates $\textit{no-loops}^F_N$, $\textit{can-reach}^F_N$, and $\textit{must-reach}^F_N$ such that for $e, e' \in E$ and $h, h' \in L^*$:

- $\textit{can-reach}^F_N(e, h, e', h')$ is true iff there exists a trace $(e, h) \dots (e', h')$ in N under F,
- $\textit{must-reach}^F_N(e, h, e', h')$ is true iff every maximum trace starting at (e, h) contains (e', h'), and
- $\textit{no-loops}^F_N$ is true iff every maximum trace in N under F is finite.

We now show that Algorithm 2 preserves all *can-reach* and *must-reach* properties, i.e. our protection never removes connectivity. We refer to the appendix for the proofs of the theorems.

Theorem 1. Let N' be the result of applying Algorithm 2 for recursive protection to an MPLS network $N = (V, E, src, tgt, L, \tau)$. For all possible failure scenarios $F \subseteq E$, for all $e, e' \in E$ and $h, h' \in L^*$:

(1) if $\textit{can-reach}^F_N(e, h, e', h')$ then $\textit{can-reach}^F_{N'}(e, h, e', h')$,
(2) if $\textit{must-reach}^F_N(e, h, e', h')$ then $\textit{must-reach}^F_{N'}(e, h, e', h')$.

The next theorem states that for any loop-free input data plane, R-MPLS guarantees to produce a loop-free protected data plane.

Theorem 2. Let $N' = (V, E', src', tgt', L', \tau')$ be the result of applying Algorithm 2 for recursive protection to an MPLS network $N = (V, E, src, tgt, L, \tau)$. If for all failure scenarios $F \subseteq E$ the network N satisfies $\textit{no-loops}^F_N$ then for all failure scenarios $F' \subseteq E$ the protected network also satisfies $\textit{no-loops}^{F'}_{N'}$.

4 EVALUATION OF R-MPLS

In this section we describe the experimental evaluation of R-MPLS. We compare its protection performance as well as memory and communication overhead against the unprotected data plane, the industry standard FRR protection and the optimal protection achieved by the tool Plinko [45].

4.1 MPLS Generation and Simulation

For evaluation of MPLS data planes we use MPLS-Kit [46], a tool and library for data plane generation and simulation. It includes utilities for automation of execution and analysis. Specifically, MPLS-Kit provides two main functionalities:

- **Data Plane Generation.** Allows for the computation of the converged data plane, mimicking a real network by running the industry standard control protocols Label Distribution Protocol (LDP)[2] and the Resource ReSerVation Protocol (RSVP) [38]. It does so by exchanging the same information that these protocols do in real networks, yet without engaging in a simulation of the actual message passing. The user inputs the network topology and the required control protocol parameters. With the information at hand, each router in the topology allocates MPLS labels and populates its forwarding tables accordingly to provide the intended reachability.
- **Simulation.** Once the data plane is generated, the library also provides functionality to perform simple packet-level simulations in order to test reachability. For this purpose, packet-level simulators model and mimic the packet delivery through the network on a hop by hop basis. The resulting trajectory of the packet along the network serves as a witness in testing that the packet arrives to its intended destination. In particular, our simulator initializes an MPLS packet with a valid header to be handled by routers in the topology. The packet is then forwarded according to the data plane rules until there are no more labels on the header or the time-to-live field is exceeded (i.e, a forwarding loop).

Further details regarding both core functionalities are provided in the appendix. The code, dataset and experimental setup is publicly available as an artifact [41], and details on the artifact are given in the artifact appendix.

4.2 Methodology

To empirically evaluate the reliability achieved by our proposed recursive protection method, we perform a series of experiments that can be decomposed in two sets:

- **RSVP experiments**: Adding R-MPLS protection on top of RSVP-based data planes.

- **LDP experiments**: Adding R-MPLS protection on top of LDP-based data planes.

The topologies used as input for MPLS-Kit [46] are real-world networks from the topology Zoo dataset [27]. For the topologies in the dataset, we first generate data planes and then we enumerate all failure scenarios (sets of failed links) with up to 4 failed links. Then, for each combination of topology, data plane and failure scenario, we run a set of packet-level simulations for all labels representing valid user traffic. As the LDP data plane is nondeterministic, we run here multiple packet simulations and take the average.

4.2.1 Data Plane Generation. On RSVP experiments, we compute the following data planes (referred as *RSVP-based data planes*):

- **RSVP:** Data plane containing n^2 unprotected RSVP tunnels between random endpoints, where n is the number of nodes.
- **RSVP + R-MPLS {Link,Node}:** Data plane containing the exact same tunnels as RSVP and additional R-MPLS recursive protection on top with either single link protections (Link) or node protections with link protection as fallback (Node).
- **RSVP + FRR:** Data plane containing the exact same tunnels as RSVP and additional RSVP-TE Fast Reroute node protection [10].
- **RSVP + Plinko {2,4}:** Data planes containing the exact same tunnels as RSVP and additional Plinko [45] path protection, with resiliency levels 2 and 4.

This set of six data planes per topology allows for direct evaluation of the impact of adding R-MPLS on top of an unprotected RSVP data plane, and to compare against the benchmarks of industry-standard RSVP-TE FRR and the state-of-the-art high-resiliency approach of Plinko.

RSVP-TE FRR is widely used on real MPLS networks to provide temporary protection against networking failures. It has been in use for more than a decade and is well understood. It is designed to provide sub 50ms local responses by diverting traffic in case of link failure and comes in two modes, facility (node) protection where the precomputed protection path avoids sending traffic through the next hop of the failed link, and link protection where other links to the same next hop can be used.

Plinko is a state-of-the-art technique for achieving optimal resiliency, i.e., it provides protection to existing routes on scenarios of up to t link failures, as long as there exists a path in the topology, without introducing loops. Plinko can be implemented using RSVP-TE FRR primitives, so we extended the RSVP class of MPLS-Kit to provide protection as specified by Algorithm 1 in [45]. A drawback hindering widespread adoption of Plinko on real networks is its high memory consumption. The original paper proposes a non-MPLS forwarding model that allows a reduction of required storage space but this improvement is impossible to employ on traditional networking devices. In our experiments, we use Plinko with resiliency levels 2 and 4. This means that for value 4, Plinko computes a protection path for up to 4 failed links (provided that the topology remains connected).

On LDP experiments, we compute just two data planes (referred as *LDP-based data planes*):

- **LDP** Data plane containing LDP generated labels for reaching every link and node in the topology from all routers.
- **LDP + R-MPLS {Link,Node}** Same data plane as LDP but with additional R-MPLS recursive protection on top with either link protection (Link) or node protection with link protection as fallback (Node).

In this case, we evaluate the effect of adding the recursive protection on top of an unprotected data plane. Notice that as FRR is a RSVP TE specific protection mechanism, it cannot be applied to LDP.

4.2.2 Failure Scenarios. We generate the failure scenarios by choosing all possible combinations of k links on all the topologies, for all values of k between 0 and 4 (included). This means that all failure scenarios have a number of total failed edges of at most 4. When we generate a failure on a topology link, we remove both directed links between the corresponding nodes, thus provoking a disconnection in both directions. This is usually the case in real networks. To restrict the total number of cases to test, if the number of total combinations to evaluate is larger than $\binom{40}{4}$, we randomly choose that number of scenarios from the set of all possible ones with at most than 4 failed links. In this case no distinction is made between LDP and RSVP experiments. Notice that when available, Plinko with resiliency level 4 provides protection for the maximum number of failed links we consider. This implies that Plinko (level 4) always achieves the optimal protection level in our simulations, however, at the expense of exponentially large communication and memory overheads as demonstrated by our experiments.

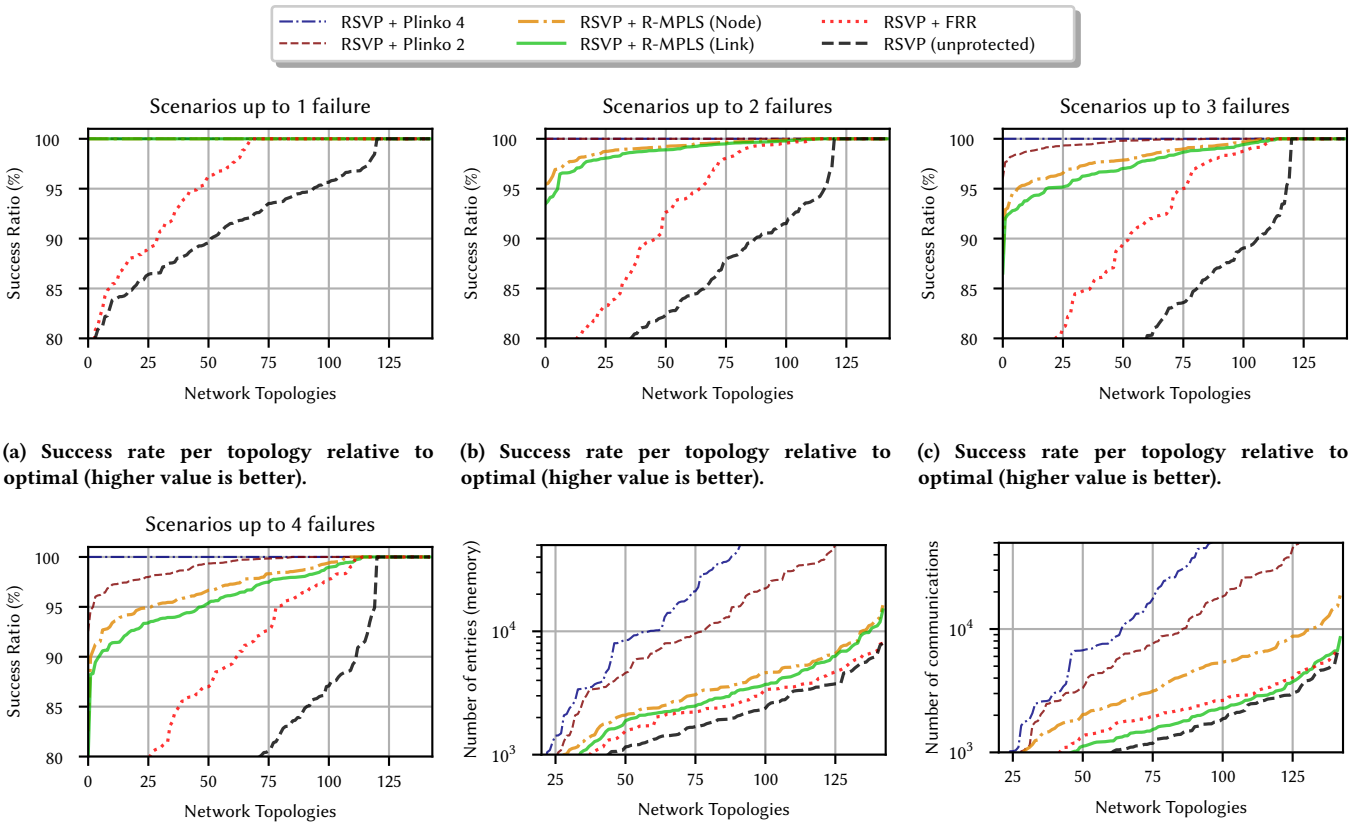
4.2.3 Execution Details. All simulations are executed making use of the command line tools and the library we implemented. In order to evaluate the topology as an MPLS network, we only simulate packets that can be part of user-generated traffic. A key advantage that this method provides, is that it allows to test the exact same packets on all data planes for each kind of experiment (RSVP or LDP), simplifying the comparison of the results.

During the simulations we also count how many times the simulation of a packet forwarding ends up in a successful forwarding towards its intended destination and how many times it ends in a failure. Given the large number of experiments, we execute the experiments on a compute cluster with 9 machines, each with 64 cores. We conduct the experiments on all topologies with a single connected component of up to 40 links.

The size of the topologies in our experiments is constrained by the computation times required to compute Plinko and LDP data planes. These two protocols have poor space scalability (exponential for Plinko and quadratic for LDP), therefore using larger topologies leads to excesively large delays in the tool MPLS-Kit [46] just to obtain the baselines, without contributing significantly to the results. This criterion results in a subset of 143 topologies.

4.3 Results of RSVP Experiments

In our experiments, we analyze the success rates of the protection, memory overhead (related to the number of added protection rules) and communication overhead (number of messages needed to establish the protection in a distributed way).



(a) Success rate per topology relative to optimal (higher value is better).

(b) Success rate per topology relative to optimal (higher value is better).

(c) Success rate per topology relative to optimal (higher value is better).

(d) Success rate per topology relative to optimal (higher value is better).

(e) Number of total forwarding table entries per topology (lower value is better).

(f) Number of communication messages among nodes (lower value is better).

Figure 4: Results for R-MPLS on all RSVP based data planes. Note that in (e) and (f) the y-axis is logarithmic.

From the experience using our data plane generation tool, on the large majority of topologies the time required to compute R-MPLS entries is comparable to the time to compute RSVP+FRR entries. Additionally, the observed average number of additional hops is almost identical between RMPLS and RSVP+FRR protections.

4.3.1 Success Rates. Figures 4a-4d show, for increasing numbers of failed links, plots for the success rates achieved by each RSVP-based data plane on each topology. Results are averaged over all failure scenarios considered, providing a measurement on the fraction of successful cases for each network topology and the network topologies are sorted in non-decreasing order (on x-axis) according to their success rates (y-axis). We can observe that Plinko 4 indeed provides optimal protection upto 4 link failures, and Plinko 2 up to two link failures. Our R-MPLS provides perfect protection for 1 link failure and slighly deteriorates with the increasing number of failures. In the rest we focus on the discussion of Figure 4d, which contains all the simulated scenarios with up to 4 link failures.

Clearly, the unprotected RSVP has the smallest success rates, and all protected data planes achieve higher success rates. The standard FRR node protection on top of RSVP achieves, as expected, a considerably better success rate. Adding our R-MPLS protection on top of the unprotected RSVP data plane alone clearly outperforms RSVP with the standard FRR node protection, both for the link and node protection. This is because RSVP+FRR has only one option to provide a protection while R-MPLS adds recursive (multiple edge) protection. Our R-MPLS protections gets closer in success rate to the optimal protection achieved by Plinko level 4 (protection is guaranteed anytime there is physical connectivity), while Plinko's level 2 success rate is between R-MPLS and the optimal protection. As expected, the node protection R-MPLS achieves better success ratio than the link protection. In all experiments, as formally proved earlier, we confirm that R-MPLS does not create any forwarding loops.

Table 1 highlights the 5 topologies (from the tested 143) that achieved the largest improvement by adding R-MPLS protection on top of the unprotected RSVP data plane, as well as the 5 in the middle and the 5 topologies with the lowest improvement. It also provides further details about the size of the topologies. Note that the 5 bottom topologies are trees, and hence cannot be protected against link failures by any method. For the top 5 topologies in the table, large improvements are achieved by all protection schemes.

Topology	#nodes	#links	RSVP	FRR	R-MPLS (Link)	R-MPLS (Node)	Plinko 2	Plinko 4
Ans	18	25	61 %	70 %	87 %	88 %	90 %	94 %
Heanet	7	11	55 %	77 %	80 %	83 %	85 %	88 %
Uninet	13	18	58 %	81 %	83 %	84 %	87 %	88 %
EliBackbone	20	30	68 %	85 %	93 %	94 %	96 %	98 %
Abvt	23	31	65 %	74 %	90 %	91 %	93 %	95 %
Cesnet200304	29	33	69 %	80 %	82 %	82 %	83 %	83 %
Nextgen	17	19	44 %	48 %	56 %	58 %	60 %	60 %
Harnet	21	23	59 %	71 %	71 %	71 %	73 %	74 %
Getnet	7	8	39 %	50 %	51 %	52 %	54 %	54 %
GtsRomania	21	24	64 %	72 %	76 %	76 %	77 %	77 %
Nordu1997	14	13	49 %	49 %	49 %	49 %	49 %	49 %
Arn	30	29	67 %	67 %	67 %	67 %	67 %	67 %
Reuna	37	36	57 %	57 %	57 %	57 %	57 %	57 %
Amres	25	24	46 %	46 %	46 %	46 %	46 %	46 %
Basnet	7	6	31 %	31 %	31 %	31 %	31 %	31 %

Table 1: Success rates for topologies using different protections ordered by the improvment R-MPLS (Link) gives on unprotected RSVP. The table shows the five top, five middle and five bottom rows.

Yet clear differences are present between the R-MPLS protected data planes and FRR: at least 2% for link protection (3% for node protection) and up to 17% (resp 18%) on the first 5 topologies. On many occasions, the R-MPLS solutions get closer to the optimal value achieved by Plinko (at level 4) than to the standard FRR.

4.3.2 Memory and Communication Overhead. Figure 4e shows the accumulated number of entries in the forwarding tables (corresponding to the required memory) of the routers, where the topologies are sorted (on the x-axis) according to the number of entries (y-axis). We can see that both FRR and our R-MPLS approach add only a moderate number of additional forwarding rules to the existing data plane (with only small differences between node and link protection). On average 21% of the memory on FRR protected data planes are used for the protection, where for R-MPLS the number is 35% and 44% for link and node protection respectively. Plinko protects (whenever possible) against all failure scenarios with up to 4 failed links, but it requires exponentially many more entries in the forwarding tables to do so (note that the y-axis is logarithmic). This is also the case if we consider Plinko only at level 2; now Plinko does not provide the optimal protection for 4 link failures anymore but at the same time it still has an exponential overhead for establishing the protection.

Similarly, Figure 4f shows the amount of required communication (message exchanges between the nodes) for all considered network topologies, showing only a negligible overhead for establishing FRR and R-MPLS link protection (on average 31% resp. 23% of the communications are used for protection) but a large communication penalty for adding the optimal protection by Plinko, both for level 4 and 2. We also notice that computing the R-MPLS node protection requires larger number of communications (on average 60% of the communications) compared to the link protection. This is due to the fact that routers must query the neighboring routers about the labels used for encoding downstream header rewriting.

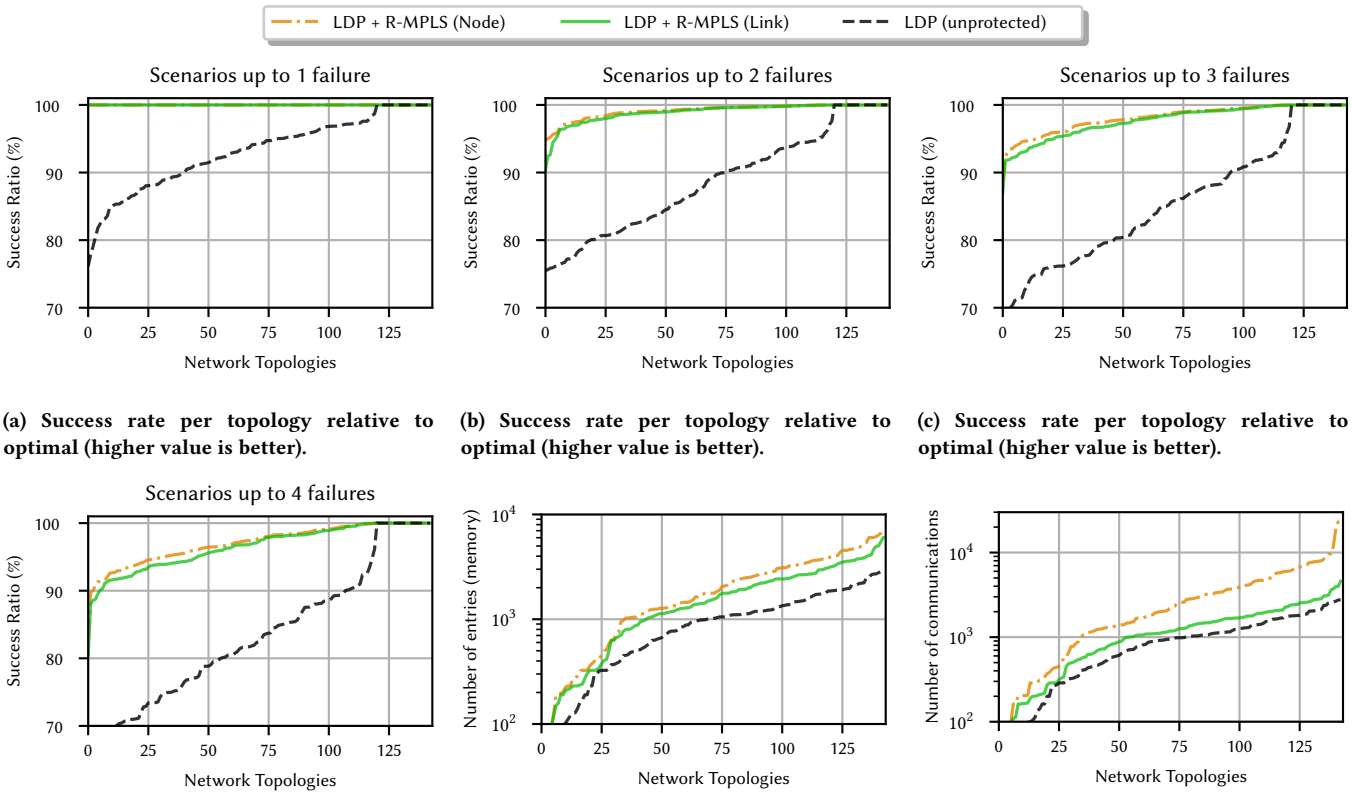
As a conclusion, the memory overhead to establish our R-MPLS protection in a distributed way is small and comparable to the widely used FRR protection, however, the success rate of the R-MPLS protection is significantly higher than for FRR. R-MPLS link protection requires fewer message exchanges between the routers compared to the node protection. Plinko achieves the optimal success rate, however, at the expense of unrealistic demands on the available memory and with a large communication overhead.

4.4 Results of LDP Experiments

We first focus on the success rate achieved by both protected (R-MPLS) and unprotected LDP data planes and benchmark against the optimum which indicates a success if the failure scenario still allows at least one path from source to destination. We then consider the memory and communication overhead. As discussed earlier, FFR and Plinko are not applicable for protecting an LDP data plane.

4.4.1 Success Rates. Figures 5a-5d show, for increasing numbers of failed links, plots with sorted success rates achieved by the LDP-based data plane and its R-MPLS protection relative to the optimum achievable protection. As before, we can observe that R-MPLS protects optimally up to 1 link failure and with the increasing number of link failures, it provides significant improvement over the unprotected data plane. The curves in Figure 5d confirm the observations from the RSVP experiments, showing significantly improved success rates when the basic LDP data plane is protected using R-MPLS and we are also relatively close to the optimum protection. As before, R-MPLS node protection is slightly more successful than the link protection.

4.4.2 Memory and Communication Overhead. The plots in Figure 5e and 5f follow the same trend as for protection of RSVP data plane and show that the overhead both for the number of entries in the forwarding tables and in the communication overhead is



(a) Success rate per topology relative to optimal (higher value is better).

(b) Success rate per topology relative to optimal (higher value is better).

(c) Success rate per topology relative to optimal (higher value is better).

(d) Success rate per topology relative to optimal (higher value is better).

(e) Number of total forwarding table entries per topology (lower value is better).

(f) Number of communication messages among nodes (lower value is better).

Figure 5: Results for R-MPLS on LDP based data planes. Note that in (e) and (f) the y-axis is logarithmic.

moderate and proportional to the overhead for establishing the unprotected data plane using LDP. Out of all created forwarding rules, only 39% (on average) are used for the additional recursive link protection and 49% for node protection, with about 27% of message exchanges needed to establish the link protection and 61% for node protection.

5 DISCUSSION

Our R-MPLS protection is designed for working on top of an arbitrary MPLS data plane. To realize this efficiently we have to address the issue of packet recirculation. Our solution uses the logical loopback link to check link failures one at a time. In case of failures, this induces a runtime overhead that is linear in the number of failed links at the router. The advantage, however, is that only one path needs to be added per link we protect, and it only takes one entry to add recursive protection for each existing entry, so the memory overhead is minimal. An alternative approach is to compute protection paths for each router and for each subset of its links that can fail. With this approach, there is no time overhead, however, an explosion in the number of necessary entries in the forwarding tables of the routers.

In our R-MPLS implementation, we hence resort to packet recirculation (where packets are sent to the loopback interface) inside the router to provide resiliency against failures. Yet, recirculating packets requires these to be sent through a slow processing path to a control element that introduces the packet again into forwarding hardware. This can hurt the throughput and cause a spike in the router's CPU. To avoid these effects, we propose a further R-MPLS enhancement without altering its inner working by recirculating the first few packets of a flow after an adjacent link failure and caching the set of header operations and the outgoing interface through which the packet finally gets transmitted. This information is then used to insert a new temporary routing entry in the forwarding table, with a priority such that it matches the following packets of the same flow. The new entry is valid until the router detects a new local link up or link down event. This operation avoids further recirculation of packets while preserving the same protection intended by R-MPLS. We plan to develop this concept in future work. In the case of non-traditional MPLS devices, it is possible to implement this caching mechanism e.g. using PURR [13], a technique devised specifically to provide packet recirculation-free primitives for path protections on programmable routers.

Although requiring fewer routing entries, R-MPLS may result in deep label stacks in multiple failure scenarios, leading to potential fragmentation or maximum label depth issues. The former can be alleviated with jumbo frames [17] without requiring lowering the MTU, and the latter with label replacement techniques in which a label stack is replaced by a new, shallower stack.

6 RELATED WORK

To provide high availability in the presence of failures, most modern communication networks support fast recovery in the data plane [11, 35, 38], see [10] for a recent survey.

This paper focuses on conventional MPLS networks, which are widely deployed today. Compared to alternative network types [1, 6, 21, 22, 39], a particular property and challenge of MPLS networks is that the header size is dynamic and potentially unbounded. The ability to fast reroute traffic (i.e., to protect LSPs) is a key feature of MPLS [38, 40, 44]. Most work has, however, been on single failure protection techniques, e.g, RSVP-TE FRR [38], LFA[8] and TI-LFA, [34]. Limitations of these techniques in multi-failure scenarios have already been observed [3, 4, 30–32, 47, 48, 50]. Besides RSVP-TE FRR, which has already been discussed in this article, LFA is a solution LDP Fast ReRoute. LFA requires knowledge of the paths to destination, so it cannot be used independently of its specific control protocol, while R-MPLS works for any control protocol—it uses the LFIB entries but is not concerned about how they were generated. Hence, R-MPLS does not interact with any other control protocol. Additionally, both LFA and TI-LFA have been designed to protect against a single failure, while R-MPLS is more general, so it is expected that R-MPLS outperforms both on multi-link failure scenarios. Alternatively, one can consider the resilience provided by Equal Cost Multi-Path (ECMP), a load-balancing data plane mechanism. As ECMP is not a Fast ReRoute protection scheme, it cannot be directly directly compared with R-MPLS. Furthermore, our model and simulator support ECMP that is abstracted as nondeterministic forwarding.

The approach suggested in [36] runs a data plane re-convergence algorithm by reversing the directions of links upon failures, while modifying the routing tables. This approach is orthogonal to ours as we preinstall the failover directly in the data plane.

Although there are proposals for achieving forwarding resilience up to a maximum number of link failures that do not disconnect the topology (*perfect forwarding resiliency*[18]) on top of MPLS primitives, we are not aware of a solution that achieves such resilience in a conventional MPLS network. Some existing proposals, like R3 [48] have a mandatory centralized stage and require additional traffic demand information, which is usually not available. R-MPLS achieves such resilience while being fully distributed and not requiring external information.

Protecting the protection paths is mentioned in RFC6981 [9], where the issue of mutually looping protection paths is addressed by putting such links into a secondary shared risk link group (SRLG), but—to our knowledge—this has not been implemented. Compared to [9], we extend with multiple protection paths, provide a complete algorithm for eliminating loops and support node protection. Further, we implement our algorithm along side with existing protocols, and run experiments to compare the performance.

R-MPLS is attractive for its ability to reinforce an existing forwarding data plane independently of how it was built. This is in stark contrast with Plinko [45], which is the only state-of-the-art proposal we know of capable of achieving perfect forwarding resilience that can be applied to conventional MPLS. However, Plinko requires control plane knowledge, i.e. the information on how the forwarding paths were originally computed, and thus cannot be easily integrated with traditional MPLS control protocols. Moreover, Plinko brute-force enumerates all hypothetical failure scenarious that must be encoded into (exponentially large) label space, causing a combinatorial number of inserted forwarding entries and exchanged messages among the routers. Our R-MPLS does not introduce this explosion in the number of labels and rules and hence scales memory- and communication-wise better than Plinko, whereas Plinko on the other hand achieves better connectivity. The requirement on the knowledge of the control plane also affects *Failure Carrying Packets* (FCP) [29], a classical proposal similar to Plinko.

Recent work showed how to provably verify the resilience and policy-compliance of MPLS networks under multiple failures. In particular, tools such as P-Rex [25, 42] and AalWines [26] allow verifying the reachability of MPLS data planes even under failures in polynomial time. However, in contrast to R-MPLS, these approaches cannot be used to improve the resilience of the data plane.

Last but not least, our work is orthogonal to solutions such as PURR [13], which allows to avoid overheads of recirculation in the switch during failover.

7 CONCLUSION

Motivated by uncovering the opportunity to increase the resilience of MPLS networks, we suggest a recursive MPLS data plane protection, allowing us to provably route traffic around *multiple* simultaneously failed links without creating any forwarding loops. Contrary to other existing approaches, R-MPLS is *fully distributed* solution and hence it is compatible with existing MPLS hardware employed in current networks.

We evaluate R-MPLS on protecting real-world networks with realistic data planes and show that our approach is efficient and significantly increases network robustness compared to the state-of-the-art FRR protection, at similar memory and communications cost. Another feature of our solution is that it is orthogonal and can be combined with existing and future protocols, such as RSVP or LDP, serving as an "extra resilience" layer, while requiring only minimal increase in memory and communication overhead.

Our work opens several interesting directions for future research. We plan to extend R-MPLS to Segment Routing networks and to evaluate its performance with respect to the standard Segment Routing's protection TI-LFA. Also, we plan to study how to further improve the performance of our algorithms. Our approach is also readily available to account for link congestion when fast reroute takes over because in our protection algorithm, we can select an arbitrary protection path for a link.

Acknowledgements. This research is funded by the Vienna Science and Technology Fund (WWTF), project WHATIF (ICT19-045), and DFF project QASNET.

REFERENCES

[1] Anubhavnidhi Abhashkumar, Aaron Gember-Jacobson, and Aditya Akella. 2020. Tiramisu: Fast Multilayer Network Verification. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX, 201–219.

[2] L. Andersson, I. Minei, and B. Thomas. 2007. *Multiprotocol Label Switching Architecture*. RFC 5036. RFC Editor. 1–135 pages. https://doi.org/10.17487/RFC5036

[3] David Applegate and Edith Cohen. 2003. Making Intra-domain Routing Robust to Changing and Uncertain Traffic Demands: Understanding Fundamental Tradeoffs. In *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (Karlsruhe, Germany) *(SIGCOMM'03)*. ACM, New York, NY, USA, 313–324. https://doi.org/10.1145/863955.863991

[4] Jasbir Singh Arora. 2017. *Introduction to Optimum Design* (fourth ed.). Academic Press, Boston. https://doi.org/10.1016/B978-0-12-800806-5.00002-0

[5] Alia K Atlas and Alex Zinin. 2008. Basic specification for IP fast-reroute: loop-free alternates. IETF RFC 5286.

[6] R. Beckett, A. Gupta, R. Mahajan, and D. Walker. 2017. A General Approach to Network Configuration Verification. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM'17)*. ACM, 155–168. https://doi.org/10.1145/3098822.3098834

[7] R. Beckett, R. Mahajan, T. Millstein, J. Padhye, and D. Walker. 2016. Don't Mind the Gap: Bridging Network-Wide Objectives and Device-Level Configurations. In *Proc. ACM SIGCOMM*. ACM, 328–341.

[8] S. Bryant, C. Filsfils, S. Previdi, M. Shand, and N. So. 2015. *Remote Loop-Free Alternate (LFA) Fast Reroute (FRR)*. RFC 7490. RFC Editor. https://doi.org/10.17487/RFC7490

[9] S. Bryant, S. Previdi, and M. Shand. 2013. *A Framework for IP and MPLS Fast Reroute Using Not-Via Addresses*. RFC 6981. RFC Editor. https://doi.org/10.17487/RFC6981

[10] Marco Chiesa, Andrzej Kamisinski, Jacek Rak, Gabor Retvari, and Stefan Schmid. 2021. A Survey of Fast-Recovery Mechanisms in Packet-Switched Networks. *IEEE Communications Surveys and Tutorials (COMST)* (2021).

[11] Marco Chiesa, Ilya Nikolaevskiy, Slobodan Mitrović, Andrei Gurtov, Aleksander Madry, Michael Schapira, and Scott Shenker. 2016. On the resiliency of static forwarding tables. *IEEE/ACM Transactions on Networking* 25, 2 (2016), 1133–1146.

[12] Marco Chiesa, Ilya Nikolaevskiy, Slobodan Mitrović, Aurojit Panda, Andrei Gurtov, Aleksander Maidry, Michael Schapira, and Scott Shenker. 2016. The quest for resilient (static) forwarding tables. In *Proc. IEEE INFOCOM*.

[13] Marco Chiesa, Roshan Sedar, Gianni Antichi, Michael Borokhovich, Andrzej Kamisiński, Georgios Nikolaidis, and Stefan Schmid. 2019. Purr: A primitive for reconfigurable fast reroute: Hope for the best and program for the worst. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*. 1–14.

[14] Richard Chirgwin. 2017. Google routing blunder sent Japan's Internet dark on Friday. In *https://www.theregister.co.uk/2017/08/27/google_routing_blunder_sent_japans_internet_dark/*.

[15] Duluth News Tribune. 2018. Human error to blame in Minnesota 911 outage. In *https://www.ems1.com/911/articles/389343048-Officials-Human-error-to-blame-in-Minn-911-outage/*.

[16] Theodore Elhourani, Abishek Gopalan, and Srinivasan Ramasubramanian. 2014. IP fast rerouting for multi-link failures. In *Proc. IEEE INFOCOM*. ACM, 2148–2156.

[17] EthernetAlliance.org. 2009. Ethernet Jumbo Frames. http://www.ethernetalliance.org/wp-content/uploads/2011/10/EA-Ethernet-Jumbo-Frames-v0-1.pdf

[18] Joan Feigenbaum, Brighten Godfrey, Aurojit Panda, Michael Schapira, Scott Shenker, and Ankit Singla. 2012. Brief announcement: On the resilience of routing tables. In *Proceedings of the 2012 ACM symposium on Principles of distributed computing*. 237–238.

[19] Clarence Filsfils, Nagendra Kumar Nainar, Carlos Pignataro, Juan Camilo Cardona, and Pierre Francois. 2015. The segment routing architecture. In *2015 IEEE Global Communications Conference (GLOBECOM)*. IEEE, 1–6.

[20] Klaus-Tycho Foerster, Juho Hirvonen, Yvonne-Anne Pignolet, Stefan Schmid, and Gilles Tredan. 2021. On the Feasibility of Perfect Resilience with Local Fast Failover. In *Proc. SIAM Symposium on Algorithmic Principles of Computer Systems (APOCS)*.

[21] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. 2015. A General Approach to Network Configuration Analysis. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 469–483.

[22] Aaron Gember-Jacobson, Raajay Viswanathan, Aditya Akella, and Ratul Mahajan. 2016. Fast Control Plane Analysis Using an Abstract Representation. In *Proceedings of the 2016 ACM SIGCOMM Conference*. ACM, 300–313.

[23] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. 2011. Understanding network failures in data centers: measurement, analysis, and implications. In *ACM SIGCOMM Computer Communication Review*, Vol. 41 (4). 350–361.

[24] Thomas Holterbach, Edgar Costa Molero, Maria Apostolaki, Alberto Dainotti, Stefano Vissicchio, and Laurent Vanbever. 2019. Blink: Fast connectivity recovery entirely in the data plane. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 161–176.

[25] Jesper Stenbjerg Jensen, Troels Beck Krøgh, Jonas Sand Madsen, Stefan Schmid, Jiří Srba, and Marc Tom Thorgersen. 2018. P-Rex: Fast Verification of MPLS Networks with Multiple Link Failures. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies (CoNEXT)*. ACM, 217–227. https://doi.org/10.1145/3281411.3281432

[26] Peter Gjøl Jensen, Dan Kristiansen, Stefan Schmid, Morten Konggaard Schou, Bernhard Clemens Schrenk, and Jiri Srba. 2020. AalWiNes: A Fast and Quantitative What-If Analysis Tool for MPLS Networks. In *Proceedings of the 16th International Conference on Emerging Networking EXperiments and Technologies (CoNEXT)*. ACM, 474–481.

[27] Simon Knight, Hung X. Nguyen, Nickolas Falkner, Rhys Bowden, and Matthew Roughan. 2011. The Internet Topology Zoo. *IEEE Journal on Selected Areas in Communications* 29, 9 (2011), 1765–1775. https://doi.org/10.1109/JSAC.2011.111002

[28] Craig Labovitz, G Robert Malan, and Farnam Jahanian. 1998. Internet routing instability. *IEEE/ACM transactions on Networking* 6, 5 (1998), 515–528.

[29] Karthik Lakshminarayanan, Matthew Caesar, Murali Rangan, Tom Anderson, Scott Shenker, and Ion Stoica. 2007. Achieving Convergence-Free Routing Using Failure-Carrying Packets. In *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (Kyoto, Japan) *(SIGCOMM '07)*. Association for Computing Machinery, New York, NY, USA, 241–252. https://doi.org/10.1145/1282380.1282408

[30] O. Lemeshko and K. Arous. 2014. Fast ReRoute model for different backup schemes in MPLS-network. In *2014 First International Scientific-Practical Conference Problems of Infocommunications Science and Technology*. 39–41. https://doi.org/10.1109/INFOCOMMST.2014.6992292

[31] O. Lemeshko, A. Romanyuk, and H. Kozlova. 2013. Design schemes for MPLS Fast ReRoute. In *2013 12th International Conference on the Experience of Designing and Application of CAD Systems in Microelectronics (CADSM)*. 202–203.

[32] O. Lemeshko and O. Yeremenko. 2018. Linear optimization model of MPLS Traffic Engineering Fast ReRoute for link, node, and bandwidth protection. In *2018 14th International Conference on Advanced Trends in Radioelecrtronics, Telecommunications and Computer Engineering (TCSET)*. 1009–1013. https://doi.org/10.1109/TCSET.2018.8336365

[33] Tony Li and Henk Smit. 2008. *IS-IS Extensions for Traffic Engineering*. Technical Report 5305. https://doi.org/10.17487/RFC5305

[34] Stephane Litkowski, Pierre Francois, Ahmed Bashandy, Clarence Filsfils, and Bruno Decraene. 2018. *RFC draft: Topology Independent Fast Reroute using Segment Routing*. Technical Report. https://tools.ietf.org/html/draft-bashandy-rtgwg-segment-routing-ti-lfa-02

[35] J. Liu, A. Panda, A. Singla, B. Godfrey, M. Schapira, and S. Shenker. 2013. Ensuring connectivity via data plane mechanisms. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 113–126.

[36] Junda Liu, Aurojit Panda, Ankit Singla, Brighten Godfrey, Michael Schapira, and Scott Shenker. 2013. Ensuring Connectivity via Data Plane Mechanisms. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. USENIX Association, 113–126.

[37] Michael Menth, Michael Duelli, Ruediger Martin, and Jens Milbrandt. 2009. Resilience analysis of packet-witched communication networks. *IEEE/ACM transactions on Networking (ToN)* 17, 6 (2009), 1950–1963.

[38] P. Pan, G. Swallow, and A. Atlas. 2005. *Fast Reroute Extensions to RSVP-TE for LSP Tunnels*. RFC 4090. RFC Editor. 1–38 pages. https://doi.org/10.17487/RFC4090

[39] Santhosh Prabhu, Kuan Yen Chou, Ali Kheradmand, Brighten Godfrey, and Matthew Caesar. 2020. Plankton: Scalable network configuration verification through model checking. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 953–967.

[40] E. Rosen, A. Viswanathan, and R. Callon. 2001. *Multiprotocol Label Switching Architecture*. RFC 3031. RFC Editor. 1–61 pages. https://doi.org/10.17487/RFC3031

[41] Stefan Schmid, Morten Konggaard Schou, Jiri Srba, and Juan Vanerio. 2022. *Artifact for "R-MPLS: Recursive Protection for Highly Dependable MPLS Networks"*. https://doi.org/10.5281/zenodo.7191618

[42] Stefan Schmid and Jiří Srba. 2018. Polynomial-Time What-If Analysis for Prefix-Manipulating MPLS Networks. In *IEEE International Conference on Computer Communications (INFOCOM)*. IEEE, 1–9.

[43] Nick Shelly, Brendan Tschaen, Klaus-Tycho Förster, Michael Chang, Theophilus Benson, and Laurent Vanbever. 2015. Destroying networks for fun (and profit). In *Proceedings of the 14th ACM Workshop on Hot Topics in Networks*. 1–7.

[44] Steve Smith. 2003. *Introduction to MPLS*. https://www.cisco.com/c/dam/global/fr_ca/training-events/pdfs/Intro_to_mpls.pdf. Visited: 19/05/2020.

[45] Brent Stephens, Alan L Cox, and Scott Rixner. 2016. Scalable multi-failure fast failover via forwarding table compression. In *Proceedings of the Symposium on SDN Research*. 1–12.

[46] Juan Vanerio, Stefan Schmid, Morten K Schou, and Jiri Srba. 2022. MPLS-Kit: An MPLS Data Plane Toolkit. In *2022 Global Internet (GI) Symposium (GI 2022)*. IEEE, Paris, France. (To appear).

[47] Hao Wang, Haiyong Xie, Lili Qiu, Yang Richard Yang, Yin Zhang, and Albert Greenberg. 2006. COPE: Traffic Engineering in Dynamic Networks. In *Proceedings of the 2006 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (Pisa, Italy) *(SIGCOMM '06)*. ACM, 99–110. https://doi.org/10.1145/1159913.1159926

[48] Ye Wang, Hao Wang, Ajay Mahimkar, Richard Alimi, Yin Zhang, Lili Qiu, and Yang Richard Yang. 2010. R3: Resilient Routing Reconfiguration. *ACM SIGCOMM Computer Communication Review* 40, 4 (Aug. 2010), 291–302. https://doi.org/10.1145/1851275.1851218

[49] Dahai Xu, Yizhi Xiong, Chunming Qiao, and Guangzhi Li. 2004. Failure protection in layered networks with shared risk link groups. *IEEE Network* 18, 3 (2004), 36–41.

[50] O. S. Yeremenko, O. V. Lemeshko, and N. Tariki. 2017. Fast ReRoute scalable solution with protection schemes of network elements. In *2017 IEEE First Ukraine Conference on Electrical and Computer Engineering (UKRCON)*. 783–788. https://doi.org/10.1109/UKRCON.2017.8100353

[51] Derek M. Yeung, Dave Katz, and Kireeti Kompella. 2003. *Traffic Engineering (TE) Extensions to OSPF Version 2.* Technical Report 3630. https://doi.org/10.17487/RFC3630

[52] Lixia Zhang, Robert T. Braden, Andreas Terzis, and Subramaniam Vincent. 2000. *RSVP Diagnostic Messages.* RFC 2745. RFC Editor. https://doi.org/10.17487/RFC2745

A PROOFS FOR SECTION 3.6

Theorem 1. Let N' be the result of applying Algorithm 2 for recursive protection to an MPLS network $N = (V, E, src, tgt, L, \tau)$. For all possible failure scenarios $F \subseteq E$, for all $e, e' \in E$ and $h, h' \in L^*$:

(1) if $can\text{-}reach^F_N(e, h, e', h')$ then $can\text{-}reach^F_{N'}(e, h, e', h')$,
(2) if $must\text{-}reach^F_N(e, h, e', h')$ then $must\text{-}reach^F_{N'}(e, h, e', h')$.

Proof. Before we prove (1) and (2), we first consider for any failure scenario F, the active forwarding tables τ_F and τ'_F for N and N', respectively. We argue by considering Line 17 and 21 in Algorithm 2 together with the definition of M that for any $v, \ell \in V \times L$,

(a) if $\tau_F(v, \ell) \neq \tau'_F(v, \ell)$ then $\tau_F(v, \ell) = \emptyset$,

since all new rules in τ' are appended with lower priority.

To prove (1), we assume that $can\text{-}reach^F_N(e, h, e', h')$ is true for some $e, e' \in E$ and $h, h' \in L^*$. Then there must exist a trace $(e_1, h_1) \dots (e_n, h_n)$ in N under F s.t. $(e_1, h_1) = (e, h)$ and $(e_n, h_n) = (e', h')$. For each step i, $1 \leq i < n$, we must have $\tau_F(tgt(e_i), head(h_i)) \neq \emptyset$, and hence due to (a) we have $\tau_F(tgt(e_i), head(h_i)) = \tau'_F(tgt(e_i), head(h_i))$. This means that the same trace is valid in N' under F, so $can\text{-}reach^F_{N'}(e, h, e', h')$ is also true.

To prove (2), we assume $must\text{-}reach^F_N(e, h, e', h')$ is true for some $e, e' \in E$ and $h, h' \in L^*$. Now all maximum traces in N under F starting from (e, h) contain (e', h'). Assume (to reach a contradiction) that some maximum trace $(e_1, h_1)(e_2, h_2) \dots$ in N' under F with $(e_1, h_1) = (e, h)$ does not contain (e', h'). Then for some step i, (e_i, h_i) is contained in some maximum trace in N, while (e_{i+1}, h_{i+1}) is not. Hence, $\tau_F(tgt(e_i), head(h_i)) \neq \tau'_F(tgt(e_i), head(h_i))$, so due to (a) $\tau_F(tgt(e_i), head(h_i)) = \emptyset$. But then $(e_1, h_1) \dots (e_i, h_i)$ is a maximum trace in N that does not contain (e', h'), which is a contradiction. □

Theorem 2. Let $N' = (V, E', src', tgt', L', \tau')$ be the result of applying Algorithm 2 for recursive protection to an MPLS network $N = (V, E, src, tgt, L, \tau)$. If for all failure scenarios $F \subseteq E$ the network N satisfies $no\text{-}loops^F_N$ then for all failure scenarios $F' \subseteq E$ the protected network also satisfies $no\text{-}loops^{F'}_{N'}$.

Proof. Assume (to reach a contradiction) that there exists a failure scenario F' such that $no\text{-}loops^{F'}_{N'}$ does not hold. Then there must be some infinite trace in the protected network $(e_1, h_1)(e_2, h_2) \dots$ and because there are only finitely many links and labels, the infinite trace must consist of finitely many repeating heads $(e, head(h))$. Consider the first repeating head, i.e. the smallest b such that there exists $a < b$ where $(e_a, head(h_a)) = (e_b, head(h_b))$. The sequence $loop = (e_a, h_a) \dots (e_b, h_b)$ is the (first) forwarding loop, and we shall now argue that it cannot exist.

Let $H = \{head(h_a), \dots, head(h_b)\}$ be the set of head labels in the loop. Consider two cases: a) H consists only of protection labels, i.e. $H \subseteq (L' \setminus L)$, and b) H contains labels from the original data plane, i.e. $H \cap L \neq \emptyset$. Note that these two cases cover all possibilities.

For case a), since all protection labels are fresh, the loop must be only traversing protection paths. A single protection path does not contain a loop (assured by Definition 6), so it must be due to recursive protection moving the trace from one protection path to another eventually making a loop. Formally, let $loop^e_i$ denote the ith link e_{a+i} in the $loop$ sequence, let $loop^h_i$ denote the ith header h_{a+i} in the $loop$ sequence, and let p_j denote the jth link in the protection path p, i.e. $p_j = e_j$ if $p = e_1 \dots e_j \dots e_n$. Let $\langle e, p \rangle = protects(head(h_a))$ be the protection on which the loop starts, so e is a failed link. Let $e_a \dots p_j$ be the longest part of p that coincides with $loop^e_1 \dots loop^e_i$ such that $p_{j+1} \neq loop^e_{i+1}$. Since $head(loop^h_{i+1})$ is also a protection label, this must indicate a failure on $e' = p_{j+1}$, so we must be using a new protection $\langle e', p' \rangle = protects(head(loop^h_{i+1}))$, and the loop trace continues with $p'_1 \dots p'_{j'} = loop^e_{i+1} \dots loop^e_{i+j'}$ such that $p'_{j'+1} \neq loop^e_{i+j'+1}$. Here either $loop$ merges back into the protection path p at some point after p_j, or $e'' = p'_{j'+1}$ is the next failed link. In the former case we can just forget the fully traversed protection $\langle e', p' \rangle$ and only consider the last recursive protection of a link e' on p. Since each protection path does not contain a loop, and the recursive protections always merge downstream (ensured by Line 19), the looping trace must eventually move to a new protection that is not fully traversed. This goes on until we reach the first failed link e on a protection path, which will complete the loop. In the protection graph there must be edges $(\langle e, p \rangle, \langle e', p' \rangle)$ annotated $\alpha(\langle e, p \rangle, \langle e', p' \rangle) = \{p_1, \dots, e_a, \dots, p_j\}$, and $(\langle e', p' \rangle, \langle e'', p'' \rangle)$ annotated $\alpha(\langle e', p' \rangle, \langle e'', p'' \rangle) = \{p'_1, \dots, p'_{j'}\}$, and so on until the edge $(\langle e^{(k)}, p^{(k)} \rangle, \langle e, p \rangle)$, which forms a cycle. Note that all links in the annotations of the edges are part of the trace and hence cannot be failed, so the cycle is a bad cycle. Since the call to FindBadProtectionPairs on Line 11 of Algorithm 2 returns a set of protection-pairs that break all bad cycles, and Line 19 removes protection based on this set, there must be some (e_i, h_i) in $loop$ where the given recursive protection is not installed, and hence the loop cannot exist.

For case b), the loop includes some routing from the original network, and hence the protection paths are fully traversed, so we can iteratively remove protection paths and corresponding failures and find a loop in the original network. Line 16 and Definition 7 along with Lines 4–10 and Line 19 ensures that a protection is only used if a higher priority entry has a path to the target of the protection path in the network with no failures. If that higher priority entry is part of another protection path, we will inductively

remove that, eventually removing all failures, or else the entry is part of the original forwarding; hence, the original network will have a loop in some failure scenario $F \subseteq F'$. This contradicts the assumption that *no-loops*$_N^F$ holds for all F. □

B ELABORATION ON SECTION 4.1

B.1 Data Plane Generation

MPLS-Kit [46] implements an abstraction of the distributed MPLS control plane, in which each router has its own protocol processes, yet these can directly access the memory of each other when required. This abstracts away communications.

MPLS introduces the concept of Forward Equivalence Class (FEC), which stands for the set of packets that should be forwarded in the same fashion; sent through the same outgoing interface to the same next-hop and executing the same set of header operations. Essentially, each FEC is identified with a local label on each router. When a packet arrives to a router, the latter determines to which FEC the former belongs to, and forwards it accordingly.

In MPLS-Kit, the two main control protocols that create Label Switched Paths (LSPs) by introducing forwarding entries on the router's tables, are LDP and RSVP. Both are industry standard, fully distributed protocols. Each LSP is related to a single FEC. LDP associates FECs with IP protocol prefixes and propagates labels through the network in order for the other routers to build their own LSPs to reach said prefixes. RSVP builds tunnels (and associates them with FECs) from a given starting node (the *headend*) towards a final node (the *tailend*) over a path allowing for fine-grained packet steering. The routers along said path locally allocate labels to represent the LSP.

Given a weighted network topology and parameters for the control protocols as inputs, the tool outputs the data plane that results from letting the control protocols converge. As MPLS-Kit implements functionalities commonly used on ISP networks, the resulting data plane is then a *realistically-looking* MPLS data plane.

B.2 Simulation

As MPLS is a transport network, each user data packet (also called user-generated traffic) that enters the MPLS domain should follow an LSP and eventually exit the network. No successfully delivered data packets may be generated or terminated *inside* the MPLS domain.

To model the connections to the outside, we add a special node θ to V, and we add links (with infinite weight) between θ and a subset of MPLS routers that have interfaces with the outside. Such routers are known as Label Edge Routers (LERs). The links from θ are used to model the possible incoming packets, and the links to θ model the points where packets can leave the MPLS network.

Algorithm 3 shows how we simulate a packet, given the link where the packet starts and the intended exit link. Line 3 considers the possible next link-header pairs given the current link and header. Line 4 reports a failure if this set is empty, i.e. there is no valid rule for the current header. When there are multiple options due to nondeterminism, Line 5 randomly picks one. A maximum number of iteration is used to determine if the packet entered a forwarding loop.

Algorithm 3 Simulation of a packet starting from e_s.

Input: Network $N = (V, E, src, tgt, L, \tau)$, failures $F \subseteq E$, start $(e_s, h_s) \in E \times L^*$, final link $e_f \in E$
Output: Exit code (in {SUCCESS, FAILURE, LOOP})

1: $(e, h) \leftarrow (e_s, h_s)$, $n \leftarrow 0$ ▷ Initial packet
2: **while** $n <$ MAX_TTL **do**
3: $nexts \leftarrow \{(e', h') \mid (e', \omega) \in \tau_F(tgt(e), head(h)), h' = \mathcal{H}(h, \omega)\}$ ▷ Compute all next hops
4: **if** $nexts = \emptyset$ **then return** FAILURE
5: Pick at random $(e', h') \in nexts$
6: **if** $e' = e_f$ and $h' = \varepsilon$ **then return** SUCCESS ▷ Success if egress router is reached
7: $(e, h) \leftarrow (e', h')$, $n \leftarrow n + 1$ ▷ Update packet
8: **return** LOOP

The start and final links in the calls to Algorithm 3 are determined from the protocols used to create LSPs and their FECs. For the purpose of this section, a FEC f defines a mapping from a subset of routers V_f to the corresponding local labels for that FEC: $f : V_f \to L$.

For RSVP, each tunnel corresponds to a single FEC f in this protocol, and is implemented with an LSP from v to v'. We define links e, e' s.t. $src(e) = \theta$, $tgt(e) = v$, $src(e') = v'$, and $tgt(e') = \theta$, and we define $\tau(v', f(v')) = \{e', \mathsf{pop}\}$ and initial header $h = f(v)$, where $\tau(v, f(v))$ contains the forwarding entries for the first step of the LSP. This encodes the behavior of the tunnel at the border of the MPLS domain. Simulating the header h is here an abstraction over how the forwarding is implemented on a real router. For the simulation with Algorithm 3, we use initial packet (e, h) and final link e', and we run this simulation for each tunnel.

In LDP, for each LDP FEC f (corresponding to an IP destination prefix ip) announced by router v', we define a link e' s.t. $src(e') = v'$, $tgt(e') = \theta$, and we define $\tau(v', f(v')) = \{e', \mathsf{pop}\}$. Let $X \subseteq V$ be the set of label edge routers. Then for each such router $v \in X$ we define a link e_v s.t. $src(e_v) = \theta$ and $tgt(e_v) = v$, and we define initial header $h_v = f(v)$, where $\tau(v, f(v))$ contains the forwarding entries of FEC f for packets entering the MPLS network at v with destination ip. We run for each v a simulation with Algorithm 3 using initial packet (e_v, h_v) and final link e', and we run such simulations for each LDP FEC. In our simulation we use $X = V$, which implies that all MPLS routers are LERs, and results in the maximum number of possible LDP generated LSPs.

In a nutshell, our simulator initializes an MPLS packet with a valid header to be handled by routers in the topology. For RSVP tunnels, this means a packet with proper headers on each headend router. For LDP entries, the simulator just initializes a packet with the corresponding label on each router.

C ARTIFACT APPENDIX

C.1 Abstract

This appendix describes software artifacts associated with this work; the python source code of R-MPLS implemented on top of the MPLS data plane generator and simulator MPLS-Kit [46] (accepted for Global Internet 2022), along with a topology dataset derived from the original topology-zoo [27] with an adapted JSON format.

These artifacts come with scripts to reproduce the experiments described in the evaluation section and a Jupyter notebook to process the result files and produce the paper statistics, Table 1, and Figures 4 and 5. The scripts are written in Bash and automate the execution of the MPLS-Kit [46] python code. Additionally, we include a dataset containing the results files we obtained from executing the artifact's scripts on our computation cluster. Finally, we provide instructions for executing the scripts and reproducing the results.

C.2 Artifact check-list (meta-information)

- **Algorithm:** The code provided implements Algorithms 1 and 2 from the RMPLS paper on the file "rmpls.py", leveraging the MPLS-Kit [46] code base.
- **Data set:** A topology dataset in JSON format derived from the topology-zoo [27] dataset is provided. For the reviewers' convenience, a dataset with the results from executing the scripts of this artifact is also provided. The dataset is approximately 3.2GB in size.
- **Run-time environment:** The artifact should run on any Linux machine with Python3 and the required libraries. This setting is recommended: Linux kernel version 5.4.0 or later, Python 3.10. It may also require root/sudo access to install python modules.
- **Hardware:** The scripts use only CPU, memory, and I/O access, so in principle, they can be run on any Linux machine without tuning. Some simulations may require up to 20 GB RAM. As a reference, our installation used a cluster of 16 computing nodes with 1TB RAM each and 1248 CPU cores in total.
- **Execution:** Handled by the Bash scripts provided.
- **Experiments:** After installing the artifact, the experiment workflow can be reproduced by executing the provided scripts in the following order:
  (1) Run "create_confs.sh {light|full}": creates configuration files and failure scenarios.
  (2) Run "run.sh {light|full}": uses the configuration files and the topology dataset to create MPLS data planes and run simulations on each data plane and failure scenario.
  (3) Activate the python virtual environment (e.g., running "source .venv/bin/activate").
  (4) Run "jupyter-notebook make_plots.ipynb" and follow the instructions to open it in a browser. The notebook loads result files and produces figures and tables. In the "Options" cell, specify the parameters as appropriate.
- **Output:** The script "create_confs.sh" creates configuration specifications to be used as inputs by the data plane generator under the folder "confs/{light|full}/<topology name>". Each configuration file instructs the generation of an MPLS data plane from a specific set of protocols, as described in the paper. It also creates files describing the different possible failure scenarios up to $k = 2$ ("light") or $k = 4$ ("full") under a subfolder called "failure_chunks" for each topology. An additional folder "confs/conext22artifact/" contains the dataset of configurations and failure scenarios computed for the "full" case on our cluster.
  The script "run.sh" takes the output from the previous script and generates the MPLS data planes, which include a topology, and based on the protocols in use, the forwarding tables on each router and a set of valid traffic source and destination pairs. Immediately after the script executes the simulation (tracing) on each data plane in each of the failure scenarios. The results are stored in JSON files under the "results/{light|full}/<topology name>" directory. There will be one result file for each configuration, summarizing results across all failure scenarios, according to the following format:

```
{"preamble":
  {"benchmark": <topology name>,
   "prog_alias": <configuration name>,
   "program": "Comparison of MPLS
              and R-MPLS"
  },
"stats":
  {"<code>":   {"comms": 532.0,
              "entries": 835.0,
              "k": 0,
              "loops": 0,
              "optimal": 121,
              "success": 121,
              "total": 121},
  ...
}
```

  Code uniquely identifies each failure scenario. The stats are, respectively: no. of control-plane communications among routers, no. of LFIB entries, no. of failed links, no. of packet traces ended in forwarding loops, no. of possible successful traces, no. of actual successful traces, number of attempted packet traces.
  An additional folder "confs/conext22artifact/" contains the dataset of configurations and failure scenarios as computed for the "full" case on our cluster for the evaluator's convenience.
  The Jupyter notebook "make_plots.ipynb" loads the JSON files summarizing the results and the topology dataset and produces the PDF files with the plots from Figures 4 and 5 (stored in the "plots/" folder). It will also generate Table 1 showing success rates (percentages) and other performance statistics mentioned in the article.
- **How much disk space required (approximately)?:** At least 9 GB.
- **How much time is needed to complete experiments (approximately)?:** On our cluster installation, running the whole batch of experiments ("full") took five days; running on a standard laptop may take weeks. A smaller set of experiments ("light"), also included for the evaluator's convenience, can be run, taking up to 30 mins to complete in our installation while using 200MB RAM.
- **Publicly available?:** Yes.
- **Code and data licenses:** GNU General Public License v3.0
- **Archived:** https://doi.org/10.5281/zenodo.7191618

C.3 Description

C.3.1 How to access. The artifact is publicly available on Zenodo https://zenodo.org/record/7191618 The provided results dataset uses 3.2GB of space. Reproducing the results using the "full" option will create an additional 3.2 GB of data. The remaining datasets and packages are usually below 2 GB. The total would be approximately 9 GB.

C.3.2 Software dependencies. This artifact assumes execution with a Debian-based OS machine or similar, with recommended requirements: Linux kernel version 5.4.0 or later, Python 3.8.10, jupyter-notebook 6.0.3.

The following Python libraries are also required: matplotlib 3.3.4, NetworkX 2.5, numpy 1.17.4, PyYAML 5.3.1, jsonschema 3.2.0, pandas 1.3.3, ujson 5.5.0.

C.3.3 Data sets. A JSON-formatted version of the publicly available topology-zoo [27] dataset is provided with the artifact. Also, a dataset containing the JSON result files we got from our execution of the experiment workflow is included.

C.4 Installation

After downloading and decompressing the artifact, change to the main artifact folder and run:

```
./install-dependencies.sh
```

C.5 Experiment workflow

Described in Section C.2. Detailed instructions can also be found on the README file.

C.6 Evaluation and expected results

After successful installation, the evaluator can run the artifacts' scripts as described in Section C.2. Detailed instructions can be found in the README file.

There are three different evaluation options, already mentioned in Section C.2. First, the "full" option will replicate the whole set of experiments and produce their results, although it can take a significant amount of time. Alternatively, the "conext2artifact" can be used directly on the final jupyter notebook to generate the tables and figures from the article using our provided results dataset. Finally, the "light" option allows the evaluator to run a smaller and faster set of experiments to validate the artifact: no use of Plinko4, no LDP, failure scenarios considering up to two simultaneous link failures, and just the first 10 topologies from the dataset by alphabetical order. The results of this option will naturally differ from the published results, although the main findings hold.

Once the instructions from the README file have been completed (with the "full" option), the user will have launched a set of experiments for each kind of data plane included in the paper (RSVP, RSVP+R-MPLS Link, RSVP+R-MPLS Node, RSVP+FRR, RSVP+Plinko2, RSVP+Plinko4, LDP, LDP+R-MPLS Link, LDP+R-MPLS Node). They will also have reproduced Figures 4 and 5, Table 1, and other performance statistics from the article using the included jupyter notebook. The produced files' location is described in item C.2.

C.7 Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-badging
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html