

# Path Splicing

Murtaza Motiwala, Megan Elmore, Nick Feamster and Santosh Vempala  
College of Computing, Georgia Tech

<http://www.gtnoise.net/splicing>

## ABSTRACT

We present *path splicing*, a new routing primitive that allows network paths to be constructed by combining multiple routing trees (“slices”) to each destination over a single network topology. Path splicing allows traffic to switch trees at any hop en route to the destination. End systems can change the path on which traffic is forwarded by changing a small number of additional bits in the packet header. We evaluate path splicing for intradomain routing using slices generated from perturbed link weights and find that splicing achieves reliability that approaches the *best possible* using a small number of slices, for only a small increase in latency and no adverse effects on traffic in the network. In the case of interdomain routing, where splicing derives multiple trees from edges in alternate backup routes, path splicing achieves near-optimal reliability and can provide significant benefits even when only a fraction of ASes deploy it. We also describe several other applications of path splicing, as well as various possible deployment paths.

**Categories and Subject Descriptors:** C.2.1 [Computer-Communication Networks]: Network Architecture and Design C.2.2 [Computer-Communication Networks]: Network Protocols—Routing Protocols

**General Terms:** Algorithms, Design, Reliability

**Keywords:** Path Splicing, path diversity, multi-path routing

## 1. INTRODUCTION

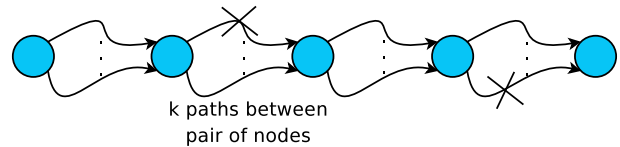
Many networked applications can benefit from access to multiple paths between endpoints. *Multipath routing*, which provides nodes access to multiple paths for each destination, can increase availability by providing fast (or simultaneous) access to backup paths; it can also improve capacity by increasing the number of paths that endpoints can use to communicate with one another. As Internet applications demand higher availability and faster recovery from failures, multipath routing and pre-computed backup paths have emerged as promising mechanisms for recovering from failures.

Despite the need for, and the promise of, multipath routing, many such schemes require considerable precomputation to achieve even a small number of paths through the network. Two obstacles have hindered many multipath routing solutions; the first is *scalability*. Existing schemes typically compute a small number of backup paths that can protect against certain failure scenarios, but they do not provide recovery from many others. Instead, the routing system should provide much stronger guarantees: Unless the under-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SIGCOMM’08, August 17–22, 2008, Seattle, Washington, USA.

Copyright 2008 ACM 978-1-60558-175-0/08/08 ...\$5.00.



**Figure 1: With  $k$  paths between the pairs of nodes, any  $k$  failures, one on each path disconnects the network. With splicing, a graph cut must be created to disconnect the network.**

lying network is partitioned, the routing system should provide at least one path that allows endpoints to communicate. The second obstacle is *control*: an endpoint (or intermediate point) should have some ability to change the path or paths that it uses to send traffic to each destination. Unfortunately, granting too much control to end systems can interfere with traffic engineering and may potentially result in traffic oscillations [24].

This paper presents the design, implementation, and evaluation of a new routing primitive called **path splicing**, a scalable mechanism for providing network nodes or endpoints access to a very large number of alternate paths. Path splicing has three key features: (1) it constructs multiple routing trees over a single fixed physical topology; (2) it allows traffic to take a path that switches between these trees at intermediate hops en route to the destination; (3) it allows end systems to change the forwarding path by changing a small number of additional bits in the packet header. Intermediate nodes can also change the path on which traffic is forwarded. These building blocks, of course, could apply to any routing protocol. In this paper, we study them in the context of intradomain and interdomain routing.

We explore how path splicing can improve availability according to two metrics: *reliability* and *recovery*. Reliability measures whether the routing information that is disseminated between routers reflects the connectivity in the underlying topology. In other words, it measures whether the paths that each router knows create a connected graph in the underlying topology, even when links or nodes in the underlying topology fail. Recovery measures how quickly endpoints can re-establish working paths with one another by finding a working path in among the available choices in the routing tables. Our evaluation demonstrates that, with just a few slices, path splicing can achieve reliability that approaches that of the underlying graph (*i.e.*, the best possible), and that, in the face of failures, end systems can discover a new working path within two trials (which are independent and can be run simultaneously), even without any knowledge about the location of the failure. The actual time to recover from a failure, of course, also comprises the time to detect the existence of a failure, which we do not consider in this work. Our results suggest that, when combined with a fast failure detection mechanism, path splicing can provide end systems with enough resilience to quickly recover from failures without waiting for dynamic routing protocols to converge to a new working path.

To illustrate why path splicing can be so effective, consider Figure 1. A conventional routing algorithm would compute one path between the nodes at each end. Multipath routing typically aims to

compute  $k$  edge-disjoint paths between these nodes. Unfortunately, if at least one link fails on each path, the nodes may become disconnected, *even if the underlying topology remains connected*. Path splicing computes multiple paths and also allows traffic to change paths at intermediate nodes, thus “splicing” paths together. By providing access to these spliced paths, path splicing can sustain connectivity in the face of many more link and node failure scenarios. In Figure 1, the pair of nodes on each side of the graph will become disconnected if a link fails on each of the  $k$  edge-disjoint backup paths. With path splicing,  $k$  links must fail in the same cut to create a disconnection, a much less likely event (since this is only one specific way in which all  $k$  paths could be broken). If we assume that links fail at random, then  $O(k \log k)$  failures will disconnect all  $k$  paths with high probability<sup>1</sup>, and the probability of a cut is exponentially small.

Despite its conceptual simplicity, path splicing faces several practical challenges. First, splicing forwards traffic along paths that do not constitute a single tree to a destination, which creates the possibility for paths to contain loops. We show, both analytically and empirically, that in practice these loops are neither persistent nor long. Second, splicing gives end hosts some control over where traffic is forwarded, which can interfere with operators’ traffic engineering goals and potentially cause oscillations if all end systems forward traffic over the same set of links. Path splicing’s interface for path selection carries no explicit semantics about the actual path, however, which means that end systems have no mechanism or incentive to select the same alternate path when a path fails. Our experiments show that spliced paths do not adversely affect the traffic distribution or load across the network links. Finally, there is an inherent tradeoff between the extent to which alternate slices provide paths with a diverse set of edges and the additional latency (“stretch”) incurred along the spliced paths. For intradomain routing, path splicing can achieve near-optimal reliability with a stretch of about 30%; for interdomain routing, splicing can achieve near-optimal reliability with negligible stretch in terms of the number of AS hops.

Although this paper focuses on how splicing applies to Internet routing (specifically, we focus on applications of splicing to both intradomain and interdomain routing), the mechanism is general and could certainly be applied in other contexts (*e.g.*, routing in wireless networks or overlays). This paper explores how path splicing can improve availability by facilitating rapid recovery from failures; however, splicing is useful in any scenario that requires access to multiple paths. In Section 8, we discuss various open issues with ultimately deploying path splicing in practice.

The rest of the paper is organized as follows. Section 2 summarizes our design goals. Section 3 presents related work. Section 4 provides an overview of path splicing and describes the high-level properties of the technique. Section 5 describes how splicing can be applied to intradomain routing, and Section 6 describes an extension of splicing to interdomain routing. Section 7 presents experiments that quantify how splicing improves both reliability and recovery, and explores splicing’s effects on and interactions with traffic. Section 8 describes a possible implementation path for splicing, as well as security concerns, and Section 9 concludes.

## 2. DESIGN GOALS

To achieve high availability, routing must exploit the underlying diversity of the network graph. Routing should maintain paths between nodes in the network unless the underlying network graph itself is disconnected. Current routing protocols, which are typi-

<sup>1</sup>This result follows from the coupon collector problem.

cally single-path, cannot achieve this. The challenge in providing multiple paths in the network to provide high path diversity is to disseminate the information about the multiple paths in a simple, scalable fashion. Specifically, a routing system should have the following design goals:

- **High reliability.** A routing protocol should allow nodes to maintain information about connectivity between pairs of network nodes, even as nodes or links in the network fail. (Section 2.1)
- **Fast recovery.** In addition to providing many alternate paths, the routing protocol should allow end systems to discover and use these alternate paths. (Section 2.2)
- **Small stretch.** The alternate paths should not be significantly longer, in terms of latency or number of hops, than the default path. (Section 2.3)
- **Control to end systems.** End systems should have some control over the paths that traffic uses. (Section 2.4)

The rest of this section describes these goals in more detail and formally defines metrics that we use to evaluate them.

### 2.1 High Reliability

Many attempts to improve reliability through diverse, multiple paths have operated without a clear definition of either reliability or path diversity, although they have typically implicitly assumed an “operational” definition of masking path failures along paths between endpoints. To capture the effect of increasing path diversity on the actual availability of the network, we introduce a formal metric for *reliability*, which describes how the graph behaves under failure. It is convenient to talk about reliability in terms of the fraction of node pairs become disconnected when a certain fraction of edges fail. We formalize this notion below.

**Definition 2.1 (Reliability)** *For a given graph  $G$ , and any  $0 \leq p \leq 1$ , let  $R(p)$  denote the fraction of node pairs that are disconnected when each edge fails independently with probability  $p$ . Reliability is then represented as a function  $y = R(x)$ , where  $x$  ranges from 0 to 1.*

This metric has an edge version and a vertex version. We have stated the edge version, but the vertex variant is quite similar. Note that this metric can apply to any graph, including the underlying network graph; we can assess the reliability of a routing protocol by comparing the reliability achieved by the routing protocol to that of the underlying graph. To achieve high reliability (*i.e.*, to attain a reliability curve that mirrors as closely as possible that of the underlying graph), a routing protocol should exploit the path diversity that exists in the underlying graph.

Conventionally, previous routing protocols have achieved high path diversity by providing systems access to node-disjoint paths. However, paths do not need to be completely node disjoint to provide high reliability (particularly if edges are failing, as opposed to nodes). To capture this property, we quantify the diversity that is achieved by two paths using a notion we call *novelty*. Essentially, the novelty of two paths is the fraction of edges between the two paths that are distinct.

**Definition 2.2 (Novelty)** *Given a (source, destination) pair, let  $P_s$  be the path with fewer edges and  $P_l$  be the path with more edges. Formally, novelty is*

$$1 - \frac{|P_l \cap P_s|}{|P_s|}$$

Novelty provides a diversity metric for any two paths between a source-destination pair. Note that novelty captures disjointness in

some fashion: For example, two paths that are completely edge disjoint will have novelty 1. As with reliability, novelty has a vertex version, but we focus on the edge version in this paper. In our experiments, we use novelty to quantify the diversity of the paths in each alternate slice relative to the original shortest path.

## 2.2 Fast Recovery

Simply achieving high reliability is not of much use if the routing system cannot quickly discover working paths when nodes or edges fail. Beyond simply achieving *high reliability*, a routing system should quickly, scalably, and simply provide working paths to nodes and end systems when links or nodes fail. We define the time it takes for a pair of nodes to establish a working path after a failure has occurred the *recovery time*.

**Definition 2.3 (Recovery Time)** *Recovery time is the time that the routing system takes to re-establish connectivity between a (source, destination) pair after the existing path has failed.*

In the absence of pre-computed backup paths or other “fast recovery” techniques, the recovery time is simply the convergence time of the routing protocol (*i.e.*, the time it takes to re-establish a working path after a failure has occurred). In the case where backup paths are available, however (*e.g.*, in the cases of fast reroute and path splicing), recovery can be faster than convergence time, because a failure can trigger an immediate failover to a backup path.

When we consider recovery time for the case of path splicing, we are interested in quantifying how long it takes for end systems to discover alternate working paths after a failure occurs. Recovery time should ideally be measured in units of time and include both the detection time (*i.e.*, the time taken to detect a failure) and the time to discover a new working path. Without a complete implementation, however, it is difficult to express recovery time in units of time. For the purposes of our evaluation in Section 7, we express recovery in terms of number of trials—the number of recovery attempts before a working path is found. One could estimate recovery time as detection time plus the recovery time, where recovery time is the number of trials required for recovery divided by the number of trials that can be executed in parallel.

## 2.3 Low Stretch

Routing protocols should provide access to alternate paths that are not significantly longer than the “default” path between those nodes, both in terms of the actual latency of the alternate paths and in terms of the number of hops that they traverse. We define a notion of *stretch* to quantify the additional latency that is incurred by alternate paths over the default path.

**Definition 2.4 (Stretch)** *Stretch is defined as the ratio of the latency on a path (between a pair of nodes) in the perturbed topology to the ratio of the shortest path (between the same pair of nodes) in the original topology.*

We use total path cost as a proxy for latency. Path diversity and stretch are somewhat conflicting goals. Thus, we must generate slices to have low stretch, but high novelty. An easy approach to creating paths with high novelty with acceptable stretch is to create slices at random (*i.e.*, by using random link weights for creating each slice). Selecting link weights in this way would lead to paths with high stretch.

## 2.4 Control to End Systems

The notions of *availability* and *failure* are specific to the application sending traffic along these paths. In the case of real-time

applications such as VoIP, it matters if the packets cannot reach the destination in a certain bounded time. For other applications (*e.g.*, bulk file transfer), these constraints may matter less, but end systems may wish to find paths with high throughput. Because end systems have differing requirements for what constitutes a “good” path, building a “one size fits all” routing system that provides good paths to all applications without taking input from the end systems themselves about the quality of paths is difficult.

If an end system deems some path in the network to be non-functional or detrimental to application performance, it should be able to signal to the network the desire to send its traffic along a different path. Of course, because network operators have traffic engineering goals and constraints of their own, the routing system should provide this control without introducing too much instability to the offered traffic load in the network.

## 3. RELATED WORK

We survey related work in three areas—multihoming and multipath routing, fast recovery schemes and overlay networks—and explore the tradeoffs of each of these recovery schemes in terms of processing overhead, storage overhead, recovery time, and required modifications to existing routers.

**Multihoming and multipath routing.** Multihoming and multipath routing provide nodes multiple paths for exchanging traffic. Various mechanisms manipulate routing to take better advantage of multiple underlying network paths [9, 15]. These schemes can operate without changing hosts or routers but are more coarse-grained, since they still only forward traffic along one path to each destination at any time. Perlman designed a routing protocol that floods routes in a way that is robust to Byzantine failure [22]. MIRO [31] and R-BGP [16] allow networks to discover additional interdomain routes to recover from failure. MIRO provides more explicit control over the AS path that traffic travels to a destination (*e.g.*, it allows a network to explicitly select the ASes that its traffic traverses) and it requires no modifications to the data plane (*i.e.*, packet headers or forwarding functions), but it requires establishing additional state at routers for each alternate path and additional out-of-band control-plane signaling, which may make it too heavyweight as a general recovery mechanism. R-BGP provides similar interdomain failure recovery as splicing, without requiring any modifications to the packet headers. Like splicing, it requires additional state in forwarding tables like splicing. Unlike splicing, however, R-BGP provides only local recovery at routers.

Path splicing relates to multi-topology routing, which precomputes backup topologies for specific failures by removing edges from the underlying topology or by setting high costs on some edges [3, 12, 17]; in contrast, path splicing computes alternate paths for *arbitrary* failure combinations. Path splicing allows traffic to traverse multiple topologies along a single path, whereas multi-topology routing only allows traffic to switch topologies once en route to the destination. It also allows end systems to divert traffic along different paths. Aspects of multi-topology routing have been standardized [23], and Cisco has recently incorporated a related mechanism called multi-topology routing into their IOS routing platform [8]; a small variant could ultimately enable path splicing.

**Fast recovery and reroute.** Path splicing uses bits in the IP header to affect how routers along a path forward traffic to a destination. This mechanism is similar to the “deflection” mechanism recently proposed by Yang *et al.* [32]. Although path splicing’s mechanisms for deflecting traffic along a new end-to-end path are similar, we show in Section 7 that path splicing achieves more path

diversity than this deflection mechanism with considerably shorter paths. Establishing parallel backup paths resembles various techniques proposed by the IETF routing working group [26] and router vendors, including load balancing mechanisms such as equal-cost multipath [14], link protection mechanisms such as MPLS Fast Reroute [7], IP Fast Reroute [27] (as well as various optimizations [4, 28]), but fast reroute requires manual configuration and requires additional routing state for each link or node to be protected. Furthermore, rerouting is triggered only by local failure detection, not by end systems. Failure-carrying packets (FCPs) carry information about failed links; this information allows routers to re-route data packets around failed links [18]. Like fast reroute mechanisms, FCPs allow routers to circumvent node and link failures without waiting for the routing protocol to reconverge, but the mechanism only provides local recovery and requires inserting large amounts of information into packets as well as potentially expensive dynamic computation.

**Improving reliability with overlays.** Overlay networks can improve diversity by routing traffic on alternate paths above the network layer [1, 2, 13]. Others have investigated how to improve connectivity by strategically placing overlay nodes within a single ISP [6]. Splicing provides a similar recovery capability without requiring continual probing of alternate paths.

## 4. PATH SPLICING: MAIN IDEA

Path splicing is a general mechanism for giving end systems access to multiple paths composed from multiple routing trees. Any instantiation of path splicing relies on the following three aspects:

1. *Generate many alternate paths by running multiple routing protocol instances.*<sup>2</sup> Instead of running a single instance of a routing protocol over a topology, routers run  $k$  routing protocol instances on the same topology, each with a slightly different configuration. The goal is to design the configuration of the routing protocol instances such that the trees to each destination do not share many edges in common. Every node then stores  $k$  forwarding table entries for each destination (one corresponding to each tree).
2. *Allow traffic to switch between paths at intermediate hops.* Rather than routing traffic over a single topology, path splicing allows traffic to switch topologies at any intermediate hop along the path. Thus, rather than having  $k$  options, a source gains access to considerably more paths to a destination (in theory, as many as  $k^l$ , where  $l$  is the number of hops on a path between the source and destination).
3. *Give end systems the control to switch paths.* To select a path, an end system includes *splicing bits* in the packet, along with the packet's destination. These splicing bits control which of the  $k$  forwarding tables is used at each hop en route to the destination. In later sections, we describe several possible designs for the splicing bits.

Path splicing has many possible realizations in various contexts. For example, it does not mandate the use of any particular routing protocol, nor does it specify how alternate topologies are generated. In the rest of this paper, we study path splicing in the context of Internet routing. Section 5 discusses the application of path splicing to intradomain routing; Section 6 discusses path splicing in the context of interdomain routing. In each case, the methods for

<sup>2</sup>We describe splicing as running  $k$  routing protocol instances for conceptual simplicity. Later, we describe how the same function can be achieved by only running a single routing protocol instance.

generating alternate paths are slightly different, but both share the above three properties.

## 5. INTRADOMAIN PATH SPLICING

In this section, we describe the design of *path splicing* in the context of intradomain routing. We also define some of the terminology we use when talking about splicing in the later sections.

### 5.1 Control Plane

The first step in splicing is to create a set of slices for the network. A *slice* is essentially a set of shortest path trees for a particular *view* of the network graph.

**Constructing slices.** The path splicing *control plane* computes multiple routing trees based on perturbations of the underlying network topology. The control plane comprises of two main components: (1) random perturbations of link weights to help deflect traffic off the shortest paths for some gains in diversity; and (2) pushing these routes in the data plane so that they can be used by the routers in making forwarding decisions.

Conventional shortest paths routing is designed to route traffic along low-cost paths, but it may create bottlenecks between various source-destination pairs. To allow endpoints to discover paths other than shortest paths between any two nodes in the network, path splicing creates routing trees that are based on *random link-weight perturbations*.

Path splicing perturbs link weights based on the original weight of the link to ensure that the length of the new shortest path is not very long compared with the original shortest path (*stretch*). The following expression defines the link weight perturbations:

$$L'(i, j) = L(i, j) + \text{Weight}(a, b, i, j) \cdot \text{Random}(0, L(i, j)) \quad (1)$$

where  $L(i, j)$  is the original link weight of the link from nodes  $i$  to  $j$ ,  $\text{Weight}(a, b, i, j)$  is a function of some properties of nodes  $i$  and  $j$  (e.g., the degrees of the nodes),  $a$  and  $b$  are constants and  $\text{Random}(0, L(i, j))$  is a random number chosen in the range of 0 to  $L(i, j)$ .

The nature of the perturbation can be changed by using different  $\text{Weight}()$  and  $\text{Random}()$  functions. The particular  $\text{Weight}()$  function used will have an effect on the types of shortest paths selected by the shortest-path algorithm.

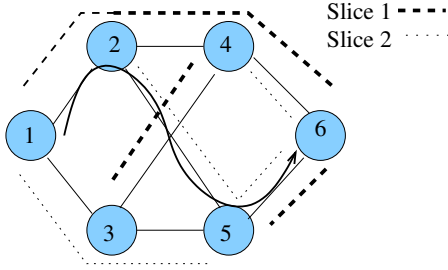
**Degree-based perturbations of link weights.** The function  $\text{Weight}(a, b, i, j)$  is selected to be a linear function of the sum of the degrees of  $i$  and  $j$ , *i.e.*

$$\forall_{i,j} \text{Weight}(a, b, i, j) = f_{ab}(\text{degree}(i) + \text{degree}(j))$$

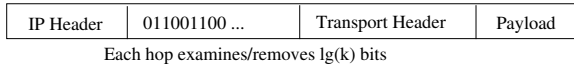
where  $f_{ab}$  is a linear function in  $\text{degree}(i) + \text{degree}(j)$  ranging from  $a$  to  $b$ . This function will cause the perturbations to depend on the end vertices  $i$  and  $j$  of a link. Links connected to nodes with a high degree may be perturbed more than links connected to nodes with smaller degree, which reduces the likelihood of many shortest paths using the same link. To describe a degree-based perturbation, we use the notation *Degree-Based*  $[a, b]$ , where  $a$  and  $b$  correspond to the minimum and maximum values that can be taken by the  $\text{Weight}(i, j)$  function. The intuition behind degree-based perturbations is to discourage the use of links between high-degree nodes, introducing more diverse path choices.

### 5.2 Data Plane

Once we have precomputed multiple slices in the network, a spliced path can be constructed by “splicing” together path segments from one or more slices. For example, as shown in Figure 2,



**Figure 2: Example of path splicing:** The two different slices shown with dotted lines on top of the original topology reflect two different trees, both rooted at node 6. Traffic can reach node 6 by traversing one or more trees.



**Figure 3: The path splicing header sits between the IP and the transport headers, facilitating incremental deployment:** routers without path splicing simply forward traffic based on the IP header.

a spliced path from node 1 to 6 is constructed by starting on slice 1 and then switching to slice 2 at the next hop (node 2). Thus, a spliced path is composed of multiple path segments from different slices. It is also easy to construct, since at each hop an independent forwarding decision could be made to either let the packet be forwarded on the same slice or switch to another slice. As we describe further, the packet could carry splicing bits (shown in Figure 3), which dictate the slice on which the packet is to be forwarded at each hop along the path. Because each hop stores the forwarding table entries (FTEs) for each slice in a separate forwarding table, the bits can index the forwarding table to use (since a forwarding table corresponds to a slice).

### 5.2.1 Header format

End systems insert a “shim” *splicing header* in between the network and transport headers. End systems can set *splicing bits* in this header to control the path taken by the packets in the network by indicating, for each hop, which forwarding table should be used to forward the packet en route to the destination.

We propose a simple encoding where the shim header contains, for  $n$  hops along the network path,  $\lg(k)$  bits that indicate an index into the forwarding table that should be used to forward the traffic at that hop, where  $k$  is the number of slices used to splice the network paths. Thus, if the size of the splicing header is  $n \cdot \lg(k)$  bits, then the header allows the packet to switch between  $k$  slices for as many as  $n$  hops along the network. Our experiments in Section 7 indicate that reliability of path splicing approaches the *best possible* reliability (as limited by the underlying network topology) with only about 4 or 5 slices. Given that most network-level paths are typically less than 30 hops [5], even this inefficient encoding would require only  $30 \lg(4) = 60$  bits. Other encodings could reduce the overall size of the splicing header.

### 5.2.2 Forwarding and failure recovery

**Forwarding algorithm.** To forward packets, each node along the path: (1) reads the rightmost  $\lg(k)$  bits from the splicing header to determine the forwarding table to use for forwarding the packet; and (2) shifts the bitstream right by  $\lg(k)$  bits to allow subsequent

hops to perform the same operation. Our previous work describes the forwarding algorithm in more detail [21].

In the default case, an end system sets the splicing bits in the splicing header to direct traffic along a path in a single routing tree (*i.e.*, as would be the case with a conventional routing protocol). A network can achieve some load balance if sources select their initial slices at random: in the absence of failure, a different subset of all sources can route traffic in each perturbed slice, achieving better “spread” of traffic across the network than could be obtained by routing all traffic along a single tree. We evaluate the effects of splicing on traffic in the network in Section 7.7.

Splicing bits carry *no explicit semantics*; this characteristic has two important implications. First, it allows path splicing to scale well, since end hosts never need to learn the details of actual paths through the network; rather, they simply use the splicing bits as an opaque identifier for some path, and they can change the path through the network simply by changing the splicing bits. We believe that this function is sufficient: end systems tend to care less about the specific hop-by-hop details about the paths their traffic is traversing than they do about whether or not they can route around a poorly performing (or faulty) path with high likelihood.

Because splicing bits control which path segments from the different slices are used to construct a spliced path, the selection of the bits determines whether an end-to-end path could be found between two nodes for which the path on the default path is disconnected. Our evaluation shows that even an extremely simple choice for the splicing bits ensures that end systems will be able to find an available path within two trials.

Because the splicing bits are opaque and have no explicit semantics (*e.g.*, they do not specify node addresses for a path), path splicing is incrementally deployable: routers that have implemented path splicing can inspect the splicing header and route packets out a different outgoing interface based on the rightmost  $\lg k$  bits in the header. Nodes along the path that do not support path splicing simply forward data packets as they normally would, based on the destination IP address in the IP header.

**Failure recovery.** When a failure occurs, traffic must be redirected to a different slice; an end host can perform this redirection simply by changing the bits in the splicing header, which will cause an end-to-end path to the destination to be spliced from a different set of slices. This redirection could be performed by either a node along the path that detects the failure or the end system, end systems can detect poorly performing paths from a variety of causes (*e.g.*, queueing, packet loss, etc.), and they are better equipped to detect when traffic should be deflected off of a poorly performing *end-to-end* path.

There are many possible ways to attempt recovery. Perhaps the simplest approach is for an end host to select a new random bit-string for the splicing header upon detection of a failure, which will cause traffic to be sent, with high probability, along a completely different path, thus avoiding the cause of the faulty path. If an end system were able to determine the location of a failure, however, it could change only the bits in the splicing header that were needed to divert traffic around the failure. As a third option, an end system could divert traffic to a different slice at an early point along the path (*i.e.*, close to the source) so as to divert traffic to a network slice that avoided the failure with high likelihood.

Nodes in the network can also take advantage of splicing to divert traffic from default paths during network failures or high congestion. If a router detects that the next-hop for a particular destination is unreachable, it can send the packet on some other connected slice while waiting for the routing protocol to converge.

## 5.3 Optimizations

**Single routing protocol instance.** It is easy to think of path splicing as running multiple instances of the routing protocol, where each instance runs with a slightly perturbed version of the topology. Unfortunately, running multiple instances of a routing protocol introduces additional unnecessary overhead including additional routing messages, as well as resource consumption on the nodes running multiple instances of the routing software.

Instead, we can implement path splicing within the context of a single routing protocol instance, with a few minor modifications. As in any intradomain routing protocol, each node would discover the complete network topology via link-state advertisements. Each node could then generate multiple variants of this topology by perturbing the weights on each edge in the graph *in the same way as on other nodes in the topology* and could compute forwarding tables for each slice locally, without having to run multiple routing protocol instances to advertise perturbed link costs.

**Single forwarding table.** The basic splicing scheme requires inserting FTEs corresponding to each slice in a separate forwarding table at each node, essentially having a forwarding table for every slice. Given that every node has a fixed number of neighbors, there could be many common entries for a particular destination among the different forwarding tables. For example, if a node has only two neighbors and we compute 3 slices, then at least two of them will have the same next hop for any destination. Thus, maintaining separate forwarding tables for every slice can lead to inefficient use of memory. One possible optimization involves having only a single forwarding table for all slices and maintain a separate column which records the different slices for which a particular entry is valid.

**Embed splicing bits into the IP header.** As we have described path splicing, the splicing bits explicitly control which slice each node on the path should use to forward traffic. In this case, the size of the shim header is proportional to the length of the path. To reduce this overhead, the splicing bits could instead be encoded in a smaller number of bits and embedded into the type-of-service and IP ID fields in the IP header; each router could then select the slice on which to forward traffic based on, say, a hash of these bits (and possibly also the source and destination IP address).

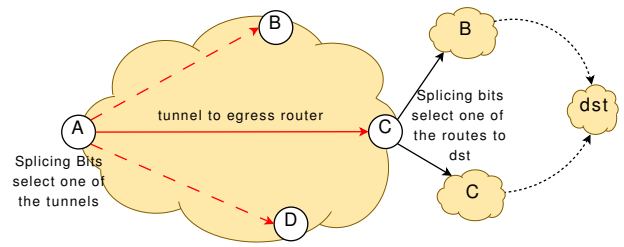
## 6. INTERDOMAIN PATH SPLICING

This section describes the application of path splicing to interdomain routing. Interdomain splicing can be deployed without modifying BGP’s message format and with no additional routing messages. In fact, it can be deployed using only a single BGP instance.

The key idea involves exploiting the fact that each router learns one BGP route to each destination *per session*, and most BGP-speaking routers already have multiple BGP sessions to neighboring routers. Rather than selecting a single best route per destination, a router inserts the best  $k$  routes for each destination; a packet’s splicing bits can then directly indicate which of these  $k$  routes a router should use to forward traffic to each destination. This section describes the control-plane and data-plane modifications to routers, and practical considerations (*e.g.*, ensuring that spliced BGP routes do not violate business policy).

### 6.1 Control Plane

Routers typically learn multiple routes to any given destination prefix both from neighboring ASes and from other routers within the same AS (via internal BGP), as summarized in Figure 4. Some of these routing table entries may correspond to alternate highly disjoint paths in the network. Routers may thus *already* learn mul-



**Figure 4: Interdomain splicing.** The bits at the ingress router select the egress router to use. The packet is tunneled to the egress router and from there one of the external routes is used to forward the packet to a neighboring AS.

iple diverse routes for each destination. Today, BGP selects only a single best route for each destination prefix. Instead, a router could select the best  $k$  routes and push them into the forwarding table. The splicing bits in a packet then index to the appropriate FTE at each hop. Using splicing bits to access alternate FTEs contrasts with existing multipath interdomain routing schemes (*e.g.*, MIRO [31], R-BGP [16]), which rely on the control plane to discover and exercise these alternate routes.

A naïve approach for selecting the top  $k$  best routes would be to repeat the route selection  $k$  times, each time removing the best route and pushing it into the IP routing table of the router. A more efficient approach would be to modify the BGP decision process to select the  $k$  best routes instead of a single best route.

### 6.2 Data Plane

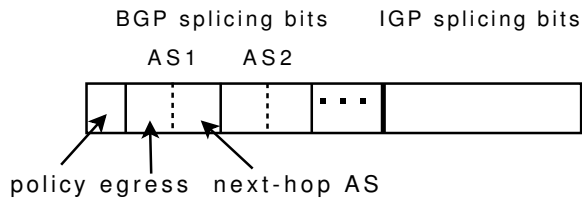
Unlike intradomain splicing, interdomain splicing uses alternate routes already in the BGP routing tables to achieve path diversity. However, the forwarding plane of the router needs to be modified to support path splicing.

#### 6.2.1 Splicing bits

As before, an end system inserts splicing bits into the packet header; the ingress and egress routers in each AS read these bits to determine how to forward the packet, as shown in Figure 4. The ingress router learns multiple paths to a destination prefix from the various border routers using iBGP (either via a “full mesh” iBGP or via its connections to multiple route reflectors) and thus may learn multiple exit points (“egress routers”) from the network for each destination prefix. For each packet, an ingress router reads the rightmost  $\lg(k)$  routing bits to determine which egress router should receive the packet and tunnels the packet to one of the egress routers. Similarly, an egress router learns multiple routes to a destination from the various border routers of the neighboring ASes via eBGP. It uses the rightmost  $\lg(k)$  routing bits to determine which of the  $k$  eBGP-learned routes (*i.e.*, which FTE) to use.

As with intradomain splicing, the ingress or egress router removes the rightmost bits from the splicing header to allow the next router that supports interdomain splicing to read the next rightmost bits. Using this approach, an  $n$ -hop AS path requires  $2n \cdot \lg(k)$  routing bits. To further reduce overhead, interdomain splicing can also use an encoding that is similar to those described in Section 5.2.

Creating  $k$  copies of the forwarding tables could introduce significant memory overhead on line cards, given the large (and growing) size of the default-free BGP routing tables. However, note that in many cases, the next-hop for a destination may be the same for different slices. In these cases, FTEs could be coalesced to save space, similar to how routers can coalesce FTEs for contiguous IP prefixes that use the same outgoing interface. In future work, we



**Figure 5: Structure of splicing bits for intradomain and interdomain splicing.**

will study the extent to which this coalescing can reduce this overhead.

### 6.2.2 Interdomain and intradomain splicing

Path splicing’s *splicing bits* must direct traffic along an end-to-end path that ultimately traverses multiple domains. To achieve this function, these bits must carry semantics for both interdomain and intradomain paths. Additionally, the interdomain paths that splicing takes must also comply with ISPs’ business policies. To achieve this function, we divide the splicing bits into several segments. The first segment is for interdomain routing (*i.e.*, selecting at both ingress and egress routers which alternate paths to use); the second segment is for intradomain routing. We envision that the interdomain splicing bits will be used at each hop along the path to the destination; in contrast, the same intradomain bits can be re-used in different ISPs along the end-to-end path.

Finally, we use a single bit in the packet header to indicate whether the packet has traversed a “peer” or “customer” edge (in the parlance of Gao-Rexford [11]); if this bit is set, the interdomain bits can *only* be used to select a BGP route through a customer AS. Routers can easily implement this mechanism by dividing the forwarding table into two separate tables: routes to provider and peer ASes, and routes to customer ASes. A router sets this bit before it sends a packet along a customer or peer edge. With this additional bit set, the interdomain splicing bits will be used to select only routes from the latter forwarding table. This mechanism ensures that all interdomain paths are valley-free.

## 6.3 Practical Concerns

This section discusses two practical concerns: the potential for interdomain forwarding loops, and the possibility that the AS-level forwarding path may not match the AS path for a destination.

**Forwarding loops.** Because interdomain splicing constructs a single end-to-end interdomain path from multiple routing trees, interdomain paths can also have loops. As with intradomain splicing, none of the loops formed are persistent because the splicing bits are finite. We propose two simple techniques to mitigate the occurrence of even transient loops. The first requires the introduction of a small counter in the shim header, *deflection counter*. This counter limits the number of times a packet can switch slices. Because most packet traverse only about four ASes [19], most potential loops will be small. Recording the last four ASes traversed by a packet in the packet header to restrict the ASes to which packets are deflected could also prevent interdomain loops.

**AS-level forwarding consistency.** In interdomain splicing, traffic might be forwarded along any of the top  $k$  best routes for a prefix, but the AS announces only a single best route to its neighbors. Some might view using a route that was not announced to its neighbors as a violation of protocol semantics, but we note that an AS will use a non-default path only if the splicing bits in the packet explicitly request this behavior or if the default path has failed. We

<b>Reliability with splicing approaches optimal.</b> For intradomain splicing, 5 slices and for interdomain splicing, only 2 slices achieve near-optimal reliability.	7.1
<b>Splicing has fast recovery.</b> An end system can recover from failure in about 2 trials when trying splicing bits at random.	7.2
<b>Perturbations achieve high novelty with low stretch.</b> Intradomain splicing has an average stretch of 20% while gaining 80% paths which are different from the original. For interdomain, the average hop stretch is only 3.8% when 5% of AS links have failed.	7.3
<b>Splicing provides better recovery than routing deflections.</b> Path splicing with only 5 slices can provide better recovery than routing deflections [32] with bounded stretch. Path splicing generally provides much shorter recovered paths, and the recovered paths have much lower variance in terms of stretch.	7.4
<b>Splicing is incrementally deployable.</b> Splicing offers significant benefits even if only a fraction of ASes deploy it.	7.5
<b>Loops are rare.</b> Forwarding loops are transient and infrequent. In intradomain splicing, we observe only 1 loop longer than 2 hops and <i>no persistent loops</i> , even with 10% of links failed.	7.6
<b>Splicing causes minimal disruption to traffic.</b> Splicing does not have much adverse effect on traffic in the network. Our evaluation using real traffic data on Abilene shows that total load on links increases only by 4% on average.	7.7

**Table 1: Summary of results.**

also note that, even today, the AS-level forwarding path is by no means guaranteed to match the advertise AS path, and many such violations occur in practice [30].

## 7. EVALUATION

This section evaluates path splicing in terms of the reliability it achieves, the ability to allow paths to quickly recover from failures of nodes and links, the latency stretch of the resulting paths, the reliability when only a fraction of ASes deploy it, the frequency of loops in spliced paths, and the effects on traffic. Table 1 summarizes the results of our evaluation. We find that path splicing provides high reliability and rapid recovery from failures and provides end systems access to a large number of low-latency, relatively loop-free paths. We also find that path splicing balances traffic across links in the network in much the same fashion as the “base” set of link weights and, to some extent, even balances this traffic slightly more evenly.

### 7.1 High Reliability

This section presents the results for reliability experiments performed with splicing for intradomain and interdomain networks. We find that, in both cases, path splicing achieves reliability that approaches that of the underlying network.

#### 7.1.1 Intradomain splicing

To evaluate the reliability of path splicing under a variety of link-failure scenarios, we implemented a simulator that takes as input a “base” network topology (with link weights) and outputs the different shortest paths trees for that network using degree-based perturbations. To simulate link failures, we removed each edge from the underlying graph with a fixed failure probability. We used the Sprint backbone network topology inferred from Rocketfuel, which

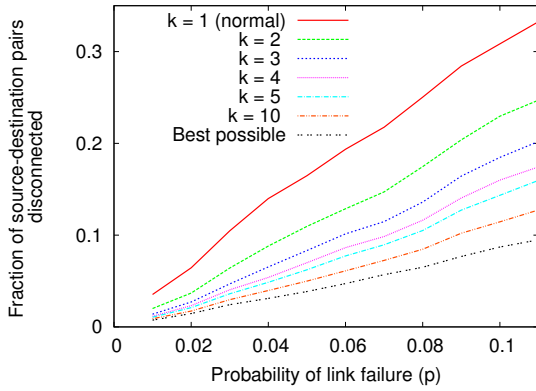


Figure 6: Reliability of path splicing for the Sprint topology.

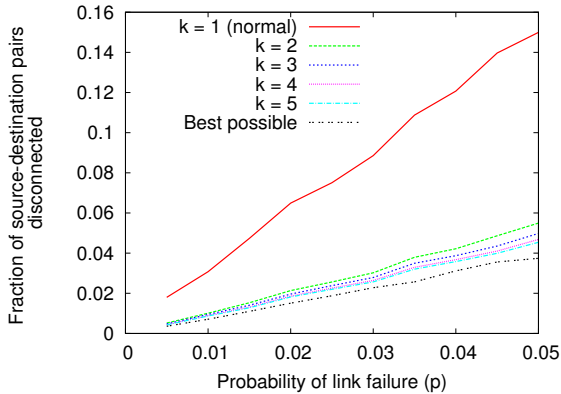


Figure 7: Reliability using a 2,500 node policy-annotated Internet AS graph.

has 52 nodes and 84 links [29]. We computed the *reliability curves* for graphs generated using path splicing and compared this characteristic both to “conventional” shortest paths routing and to that of the original underlying graph, whose reliability reflects the *best possible* reliability that could be achieved by any routing protocol.

A spliced graph with  $k$  slices is constructed by taking the union of the  $k$  slices, each of which is a random perturbation, generated as described in the previous section. Next, we remove each edge from the graph independently with probability  $p$ . We start with  $k = 1$ , evaluate the reliability for the resulting graph, increase  $k$  to 2 (*i.e.*, add edges to the graph by taking the union of the two graphs) and evaluate the reliability of the resulting graph by failing the *same set of links* (simulating the effects of a link failure in the underlying network). We perform this process 1,000 times; in other words, for each  $k$  and  $p$ , we construct a  $k$ -slice graph with appropriate edges “failed”, and compute the average reliability for those 1,000 trials.

Figure 6 shows the reliability curves for Sprint using degree-based perturbations with Degree-based(0, 3). Adding just one slice (*i.e.*, increasing  $k$  to 2) significantly improves reliability; adding more slices improves reliability further. Figure 6 demonstrates that even with just a few slices (*i.e.*, 5) and a simple scheme for generating alternate graphs (*i.e.*, link-weight perturbations), the *reliability of path splicing approaches the reliability of the original underlying network*. We also performed a reliability experiment for single node failures and found similar results.

### 7.1.2 Interdomain splicing

To evaluate the reliability of interdomain splicing, we used C-BGP [25], an open-source BGP routing solver. C-BGP takes as input a policy-annotated graph of ASes and calculates the interdomain routes for each AS. For our experiments, we use a 2,500 node policy-annotated AS graph generated by Dimitropoulos *et al.* [10]. Once C-BGP computes the interdomain routes, we removed AS edges at random with probability  $p$ . Next, on this modified AS graph, we checked for connectivity between random pairs of ASes in the graph (testing reliability for all pairs is not tractable).

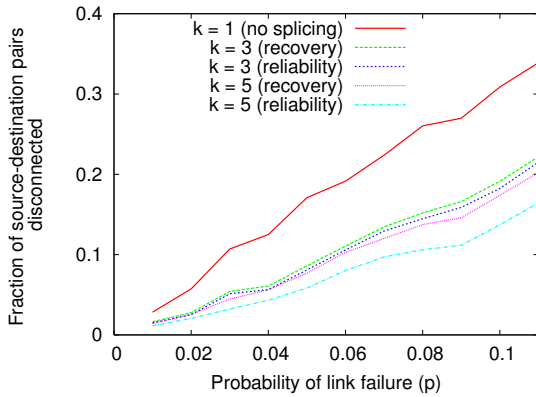
In cases where the default path was disconnected, we checked to see if a “spliced” path existed for the disconnected AS pair using up to  $k$  choices for the next-hop. We repeated this process 50 times for each value of  $p$  and  $k$ . Figure 7 shows the average fraction of pairs disconnected for a range of values for  $p$  and  $k$ . We observe that adding just one more slice significantly improves the reliability of the AS graph. For the “best possible” case, we evaluated reliability for the base graph (without policy restrictions). The reliability curve for interdomain splicing that respects policy is so close to the *best possible* reliability curve, which demonstrates that BGP, even with policy restrictions, has near-optimal path diversity if multiple routes are used. Path splicing can thus exploit this diversity without violating AS-level policies or any modifications to BGP message format.

## 7.2 Fast Recovery

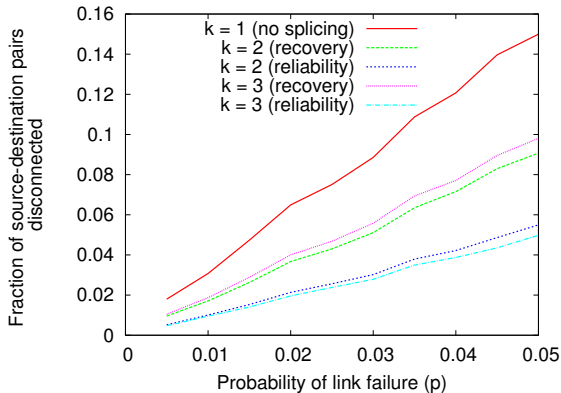
In this section, we demonstrate how an end system or a network node can quickly recover from failures by selecting spliced paths in the network at random. We evaluate two approaches to recovery: *end-system recovery* is network-agnostic and relies on the end system (*e.g.*, user, proxy, edge router) to initiate recovery; *network-based recovery* assumes that the node in the network can detect a failure on an incident link and initiate recovery by diverting traffic to a different slice. To generate a spliced graph with failures on the Sprint topology, we use a simulation setup similar to the one for the reliability experiment in Section 7.1.1. We only show results from *end-system recovery*.

For all disconnected source-destination pairs, we evaluate whether splicing allows pairs of nodes to discover working alternate paths. If splicing can recover the path in five or fewer trials (we assume that the end system or node could run these trials either in sequence, in parallel, or even in advance), we consider the path recoverable. As discussed in Section 2, our simulations do not allow us to explicitly compute recovery time in terms of seconds, but we can estimate what this time might be from the number of trials: Because it would take about one round-trip time to estimate whether a new set of splicing bits resulted in a functional path, we can estimate the recovery time as the number of trials times the round-trip time, divided by the number of trials that the system makes in parallel.

**End-system recovery.** Figure 8 shows the recovery where the end system controls the spliced path to the destination. In our experiments, we used a header that allows 20 hops to be spliced. For a failed path, the new shim header (*i.e.*, the splicing bits) is constructed as follows: A coin is tossed for every hop in the shim header; if the result is a head, a different slice is selected at random for that hop (*i.e.*, at every hop we switch slices with 0.5 probability). We check to see if a failed path can be recovered in fewer than 5 trials. The average number of trials in any case where splicing could recover from the failure was slightly more than 2. Paths were on average 1.3 times longer (in terms of path cost) compared to the shortest path in the “base” topology; the resulting paths typically used about 50% more hops compared to the original shortest path.



**Figure 8: Recovery using *end-system recovery* and Sprint topology.**



**Figure 9: Recovery using *end-system recovery* and a 2,500 node policy-annotated Internet AS graph.**

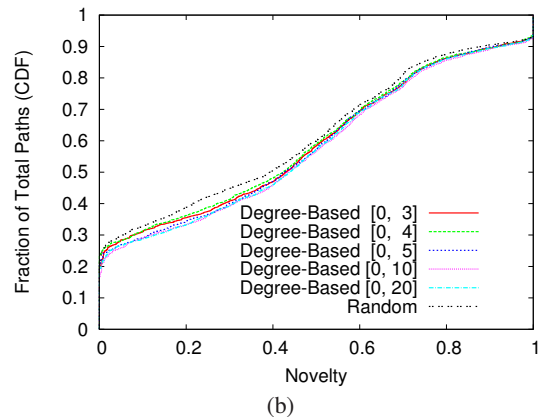
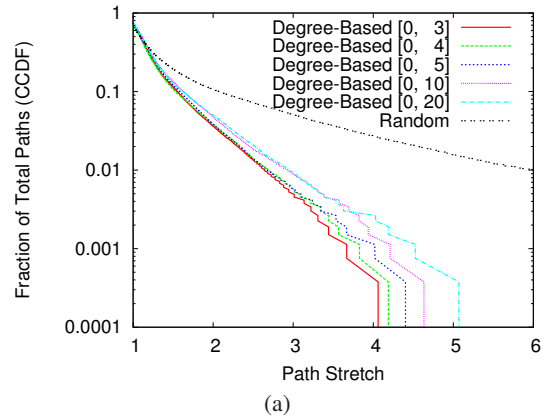
In any particular slice, 99% of all paths in each tree had stretch less than 2.6. Figure 9 shows recovery for interdomain splicing. The recovery is slightly worse because we consider only policy-compliant paths as recoverable. These results show that splicing provides effective recovery, even with the simplest possible recovery scheme and no knowledge about the location of failures.

To understand how these recovery numbers compare to a simpler scheme that simply tries to recover by using one of  $k$  paths at the source (closer to what a simple multipath scheme might do), we compared path splicing to a recovery scheme that selects one slice at the first hop and does not switch at intermediate hops. We found that splicing’s end-system recovery still exhibits slightly better recovery: With 2 slices and a 10% failure probability, splicing was able to recover about 7% more paths. This margin may, in some cases, not justify the additional cost of path splicing, but path splicing may also be able to perform better with a more sophisticated recovery scheme that uses specific information about the location of network failures.

### 7.3 High Novelty, Low Stretch

Recall from our design goals in Section 2 that the paths generated in each slice should have low stretch and high novelty. Our evaluation shows that, for intradomain splicing, random perturbations achieve reasonable novelty while keeping the stretch of each slice—and the stretch of the overall spliced paths—low.

**Intradomain splicing.** We show the results of our stretch and novelty experiments using the Sprint topology. We vary the



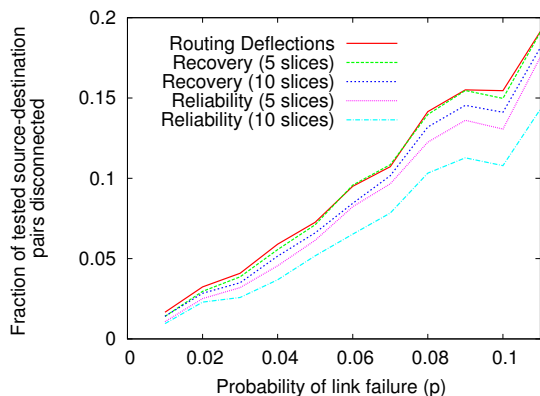
**Figure 10: Stretch and novelty for *degree-based perturbations* of the paths in the Sprint topology.**

Weight( $a, b, i, j$ ) function from Equation 1 (Section 2) and observe its effects on novelty and stretch. We also compared the results of degree-based perturbations with the random case in which link weights are set randomly in the range of  $[0, 5000]$ . For these experiments, we ran the simulator to generate 100 different slices for different values of  $b$  with  $a = 0$ , in Weight( $a, b, i, j$ ), which controls the magnitude of the perturbations.

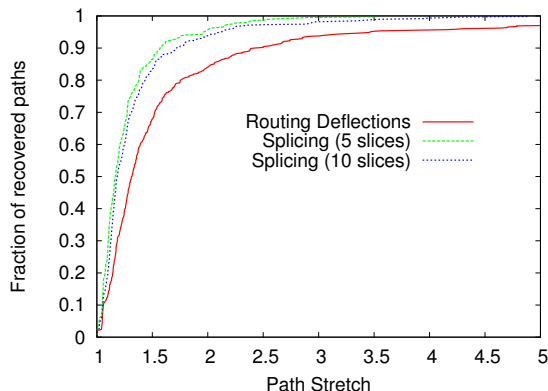
Figure 10 shows the stretch and novelty for the Sprint topology with degree-based perturbations; each line reflects a different Weight( $a, b, i, j$ ) function. Degree-based perturbations achieve almost as much novelty as random link weight settings, but with far less stretch (particularly in the worst case). For example, in the case of Degree-Based[0, 3], the average stretch is only 1.2; the worst-case stretch is also far better than the random link-weight settings. In fact, only about 3.5% of paths have stretch of more than 2. The corresponding average novelty value for the slices for degree-based perturbations is 0.41 and 80% of paths have one or more links different than those in the original shortest paths. Increasing the value of the Weight() function results in small improvements in novelty but higher stretch.

Uniform perturbations also have low stretch, but they provide less novelty than degree-based perturbations. For example, the average stretch for the case of Weight() = 1 is only 1.03. The corresponding average novelty for this case is 0.22. On average, 57% of paths differ by one link or more from the original shortest paths.

Not only is the stretch of the paths in each slice low, but the stretch of the actual spliced paths after recovery is also low. In the case of end-system recovery, paths were on average 1.3 times



**Figure 11: Comparison of recovery for splicing vs. routing deflections with stretch  $< 2$ .**



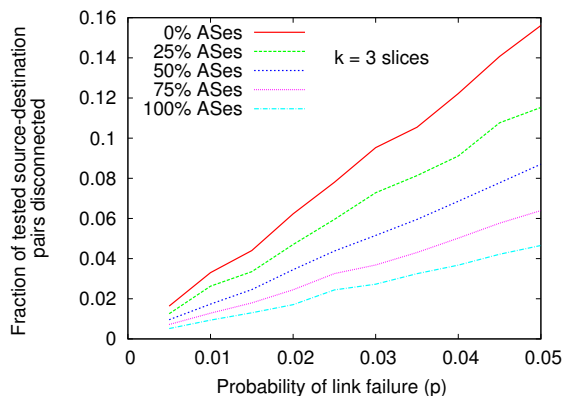
**Figure 12: Comparison of stretch for recovered paths for splicing vs. routing deflections.**

longer in delay compared to the shortest path in the “base” topology; the resulting paths typically use about 50% more hops compared to the original shortest path. In any particular slice, 99% of all paths in each tree have stretch of less than 2.6. The average stretch network-based recovery was 1.33, while there were 55% more hops in the recovered paths; these numbers are slightly higher compared to the end-system recovery scheme.

**Interdomain Splicing.** We computed the average hop-count stretch for the interdomain reliability experiment in Section 7.1.2. The hop-count stretch with 5% of the AS links failed was only 1.038, or 3.8% more hops than in the default AS paths.

## 7.4 Comparison to Routing Deflections

We compared the end-system recovery achieved by intradomain path splicing to that achieved by the routing deflection mechanism proposed by Yang *et al.* [32]. We re-implemented the deflection routing system and compared the reliability achieved by this scheme to that achieved by path splicing. Previous work on routing deflections does not consider the stretch of the resulting paths and considers *all* possible recovered paths. With routing deflections, the number of neighbors that a node can potentially send a packet to is not bounded, whereas in path splicing it is bounded by the number of slices; hence, routing deflections may require significantly more storage. To provide a fair comparison between the two schemes, we consider a path “recovered” only if it has a stretch of less than 2. Figure 11 shows the recovery achieved by path splicing for different numbers of slices compared to routing deflections.



**Figure 13: BGP Splicing: Incremental deployment.**

Path splicing recovers more paths than routing deflections using just five slices.

In addition to directly comparing recovery, we compared the stretch of the recovered paths using each of the schemes for this experiment. Figure 12 shows the resulting statistics. The results show that path splicing can recover paths that have lower stretch than the stretch of the paths recovered using routing deflection. Path splicing generated paths with an average stretch of 1.26, whereas the path stretch using routing deflections was 1.78. Path splicing also generates shorter paths more consistently: the variance of stretch values for paths generating using path splicing was 0.09; in contrast, the variance of stretch for recovered paths using routing deflections was 4.83.

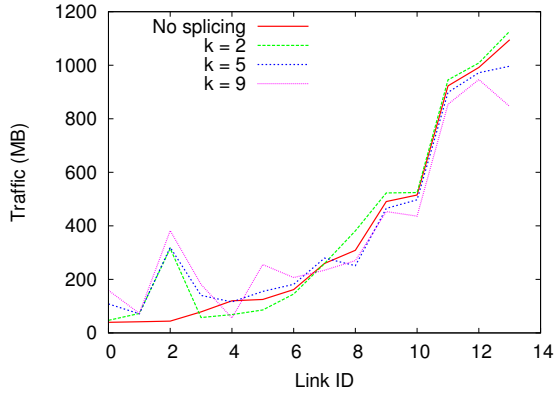
## 7.5 Incremental Deployability

Interdomain splicing requires ASes to independently decide to deploy additional functionality. It is reasonable to ask, then, how well interdomain splicing would perform if only a fraction of ASes deployed it. Our experiments show that path splicing provides significant benefits even if only a small fraction of a fraction of ASes deploy it. To evaluate the benefits of partial deployment, we use the same AS topology as in the interdomain reliability experiments. We fixed the number of slices and performed the reliability experiment as before; for each experiment, we let only a fraction of ASes select an alternate AS-level path if the next-hop on the default route has failed. We evaluate reliability for five levels of deployment: 0% to 100% with 3 slices, as shown in Figure 13. Reliability improves significantly even if only 25% of the ASes deploy interdomain splicing. We expect that the benefits might be even higher if all “Tier-1” ISPs deployed splicing.

## 7.6 Infrequent (and avoidable) Loops

Because traffic is not forwarded along a single routing tree, splicing does create the potential for transient forwarding loops if some precautions are not taken. Forwarding loops are a concern because they increase the total length of the end-to-end path, and they also unnecessarily use extra network capacity and node resources (note that these detriments are the same as paths with longer stretch; we have already shown that spliced paths have reasonable stretch).

Fortunately, certain recovery strategies can avoid *persistent* forwarding loops entirely. First, a persistent loop would require the splicing bits to be repeated in exactly the right sequence. Second, in the design we presented in Section 5.2, the splicing header will eventually run out of splicing bits as each node shifts  $\lg(k)$  bits from the header; at this point, the packet stays in the same tree to the destination. Second, paths that never switch back to a previously



**Figure 14: Abilene Network: Effect of splicing on traffic in the network using real traffic traces.**

used slice would never contain persistent forwarding loops of any length; recovery strategies could pick only these paths. Although it would not necessarily prevent transient loops entirely, restricting the number of switches between slices that any packet takes would also limit the likelihood of loops significantly. Our evaluation shows that loops were quite infrequent. Using network-based recovery, there was less than 1 loop on average with length greater than 2 when recovering from the case where the network had 10% of links failed. Two-hop loops occurred more frequently (about one per 100 trials for  $k = 2$ , and about one in ten trials for higher values of  $k$ ). Using any of the schemes discussed above could eliminate loops entirely, at the cost of restricting the paths available for recovery.

### 7.7 Minimal Disruption to Traffic

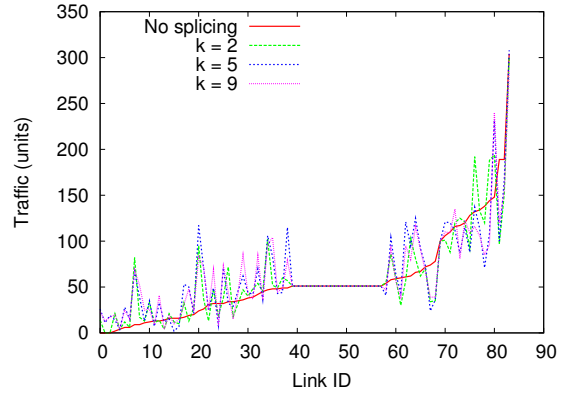
We studied the effects of splicing on traffic loads within a single ISP. We extended C-BGP to support intradomain path splicing and provided C-BGP with BGP routing tables, IGP configurations, and NetFlow traffic traces for the Abilene network; we then used it to determine the traffic load on each link in the network in the default case and for various instantiations of splicing. Abilene has only 11 nodes and 14 links, but we ran our experiments using this network because it makes routing and traffic data publicly available.

For the experiment, we create  $k$  slices for the Abilene topology in C-BGP; we used degree-based perturbations to generate the slices. C-BGP computes shortest paths for each slice and loads the routes into the respective forwarding tables on each of the nodes. Next, we load the BGP routing table dumps obtained from Abilene on each of the nodes. We then “play” 5-minute NetFlow traces through the network; we load a NetFlow trace onto each node that corresponds to the traffic collected from the node in the actual Abilene network.

For every packet reflected in the trace statistics, C-BGP selects a slice based on the hash value of the source and destination IP addresses in the packet. So traffic is split randomly among the  $k$  slices. Figure 14 shows the resulting link loads.<sup>3</sup> We also performed a similar experiment using the Sprint topology and a synthetic traffic matrix, which consisted of unit traffic for all node pairs. Figure 15 shows the results of this experiment.

The plots sort links on the basis of their load in the case without splicing and show the corresponding load on the same links using splicing. The plots demonstrate that splicing does not cause significant adverse effects on traffic. Splicing can increase stretch if

<sup>3</sup>We repeated the experiment with different 5-minute NetFlow packet traces and found similar results.



**Figure 15: Sprint Network: Effect of splicing on traffic in the network using synthetic traffic.**

traffic is routed on paths other than the shortest path in the network. As a result, the sum of the load on the links in the network will be higher when using splicing. Fortunately, the utilization is not that much greater: the sum of the load on the links is on average only about 4% higher (and never more than 10% higher) than without splicing. In the Sprint network, traffic under splicing is 9% higher on average (and never more than 12%).

## 8. DISCUSSION AND OPEN ISSUES

This section explores the changes both to hosts and to routers that would be required to deploy and evaluate various aspects of splicing in practice (*e.g.*, recovery time).

**Changes to routers.** Path splicing requires changes to the forwarding plane in routers in order to support multiple routes for a destination and the ability to select one of those routes based on the splicing bits. Recently, multi-topology routing has been standardized [23], and router vendors are also supporting this function [8, 20]. The basic forwarding mechanism required for splicing is very similar to multi-topology routing. We expect that the data-plane implementation of splicing will entail only a small extension to MTR. Additionally, we have developed a Click element that uses bits in the IP ID and type of service fields and to index into separate forwarding tables generated by the path splicing control plane; we plan to use this in conjunction with the changes to end systems described below to evaluate the recovery time of splicing in practice.

**Changes to end systems.** Path splicing relies on a failure detection mechanism before it can find a new working path. As we discussed in Section 7, detection could take place either at the routers themselves (as it is done today with other recovery mechanisms, such as fast reroute) or at end hosts (which might allow for recovery from different classes of “failures”, such as paths that exhibit high packet loss or jitter, as well as those that might exhibit complete outages). Instrumenting applications to take advantage of path splicing will require designing and developing mechanisms for receiving information about path quality as well as an extension to the sockets API for setting splicing bits in the packet headers.

**Adversarial concerns.** An adversary could set splicing bits that send packets into a forwarding loop, thus wasting resources. This attack seems unlikely, because it requires an adversary to actually discover splicing bits that will induce a loop. An adversary cannot use the splicing bits to create arbitrary loops. Our previous work discusses defenses in more detail [21].

## 9. CONCLUSION

This paper presented the design and evaluation of *path splicing*, a primitive for increasing reliability by composing routes from multiple routing protocol instances. We have applied path splicing to both intradomain and interdomain routing and evaluated its ability to allow end systems to find alternate paths when links fail. Our experiments show that running just a few slices in parallel allows path splicing to achieve reliability that is close to that of the underlying graph (*i.e.*, as long as endpoints remain connected in the underlying graph, there will be some spliced path that connects them). We have also demonstrated that even simple recovery schemes, such as randomly selecting splicing bits, allows end systems to realize this reliability using alternate paths with small stretch. Path splicing can be deployed on existing routers with small modifications to existing multi-topology routing functions. We also foresee many possible applications to other routing protocols (*e.g.*, wireless, overlay routing) and to many other applications that could take advantage of having access to multiple paths in parallel.

## Acknowledgments

This work was funded by NSF Awards CNS-0626950, CCR-0307536, NSF CAREER Award CNS-0643974, ARC ThinkTank at Georgia Tech and a Raytheon fellowship. We thank our shepherd Alex Snoeren for helping us improve the paper. We thank the Algorithms and Randomness Center at Georgia Tech for inspiring discussions. We also thank David Andersen, Hari Balakrishnan, Andy Bavier, Navin Goyal, Roch Guerin, Dick Karp, Amund Kvalbein, Dick Lipton, Bruno Quoitin, Luis Rademacher, Jennifer Rexford, Stefan Savage, Scott Shenker, Geoff Voelker, and Muhammad Mukarram bin Tariq for helpful feedback and discussion.

## 10. REFERENCES

- [1] D. G. Andersen, H. Balakrishnan, M. F. Kaashoek, and R. Morris. Resilient Overlay Networks. In *Proc. 18th ACM Symposium on Operating Systems Principles (SOSP)*, pages 131–145, Banff, Canada, Oct. 2001.
- [2] D. G. Andersen, A. C. Snoeren, and H. Balakrishnan. Best-path vs. multi-path overlay routing. In *Proc. ACM SIGCOMM Internet Measurement Conference*, Miami, FL, Oct. 2003.
- [3] G. Apostolopoulos. Using multiple topologies for ip-only protection against network failures: A routing performance perspective. Technical Report 377, ICS-FORTH, Apr. 2006.
- [4] A. Atlas and A. Zinin. Basic Specification for IP Fast-Reroute: Loop-free Alternates. <http://tools.ietf.org/html/draft-ietf-rtgwg-ipfrr-spec-base-10>, Nov. 2007.
- [5] A. Brodia and kc claffly. Topological Resilience in IP and AS Graphs. <http://www.caida.org/analysis/topology/resilience/>, 2006.
- [6] M. Cha, S. Moon, C.-D. Park, , and A. Shaikh. Placing Relay Nodes for Intra-Domain Path Diversity. In *Proc. IEEE INFOCOM*, Barcelona, Spain, Mar. 2006.
- [7] MPLS Traffic Engineering Fast Reroute – Link Protection. <http://www.cisco.com/univercd/cc/td/doc/product/software/ios120/120newft/120limit/120st/120st16/frr.htm>.
- [8] Cisco Multi-Topology Routing. [http://www.cisco.com/en/US/products/ps6922/products\\_feature\\_guide09186a00807c64b8.html](http://www.cisco.com/en/US/products/ps6922/products_feature_guide09186a00807c64b8.html).
- [9] Cisco Optimized Edge Routing (OER). [http://www.cisco.com/en/US/products/ps6628/products\\_ios\\_protocol\\_option\\_home.html](http://www.cisco.com/en/US/products/ps6628/products_ios_protocol_option_home.html), 2006.
- [10] X. A. Dimitropoulos, D. V. Krioukov, A. Vahdat, and G. F. Riley. Graph Annotations in Modeling Complex Network Topologies. *CoRR*, abs/0708.3879, 2007.
- [11] L. Gao and J. Rexford. Stable Internet routing without global coordination. *IEEE/ACM Transactions on Networking*, pages 681–692, Dec. 2001.
- [12] S. Gjessing. Implementation of two Resilience Mechanisms using Multi Topology Routing and Stub Routers. In *International Conference on Internet and Web Applications and Services/Advanced*, Feb. 2006.
- [13] K. P. Gummadi, H. V. Madhyastha, S. D. Gribble, H. M. Levy, and D. Wetherall. Improving the reliability of Internet paths with one-hop source routing. In *Proc. 6th USENIX OSDI*, San Francisco, CA, Dec. 2004.
- [14] C. Hopps. *Analysis of an Equal-cost Multi-Path algorithm*. Internet Engineering Task Force, Nov. 2000. RFC 2992.
- [15] Internap. <http://www.internap.com/>, 2006.
- [16] N. Kushman, S. Kandula, D. Katabi, and B. M. Maggs. R-BGP: Staying connected in a connected world. In *Proc. 4th USENIX NSDI*, Cambridge, MA, Apr. 2007.
- [17] A. Kvalbein, A. F. Hansen, T. Cicic, S. Gjessing, and O. Lysne. Fast IP Network Recovery using Multiple Routing Configurations. In *Proc. IEEE INFOCOM*, pages 23–26, Barcelona, Spain, Mar. 2006.
- [18] K. Lakshminarayanan, M. Caesar, M. Rangan, T. Anderson, S. Shenker, and I. Stoica. Achieving Convergence-Free Routing with Failure-Carrying packets. In *Proc. ACM SIGCOMM*, Kyoto, Japan, Aug. 2007.
- [19] J. Leskovec, J. Kleinberg, and C. Faloutsos. Graphs over time: Densification laws, shrinking diameters and possible explanations. In *Proc. 11th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, Chicago, IL, Aug. 2005.
- [20] Juniper Networks: Intelligent Logical Router Service. [http://www.juniper.net/solutions/literature/white\\_papers/200097.pdf](http://www.juniper.net/solutions/literature/white_papers/200097.pdf).
- [21] M. Motiwala, N. Feamster, and S. Vempala. Path Splicing: Reliable Connectivity with Rapid Recovery. In *Proc. 6th ACM Workshop on Hot Topics in Networks (Hotnets-VI)*, Atlanta, GA, Nov. 2007.
- [22] R. Perlman. *Network Layer Protocols with Byzantine Robustness*. PhD thesis, Massachusetts Institute of Technology, Oct. 1988. MIT-LCS-TR-429. <http://www.lcs.mit.edu/publications/specpub.php?id=997>.
- [23] P. Psenak, S. Mirtorabi, A. Roy, L. Nguyen, and P. Pillay-Esnault. *Multi-Topology Routing in OSPF*. Internet Engineering Task Force, June 2007. RFC 4915.
- [24] L. Qiu, Y. R. Yang, Y. Zhang, and S. Shenker. On selfish routing in Internet-like environments. In *Proc. ACM SIGCOMM*, Karlsruhe, Germany, Aug. 2003.
- [25] B. Quoitin and S. Uhlig. Modeling the routing of an autonomous system with C-BGP. *Network, IEEE*, 19(6):12–19, 2005.
- [26] Routing Area Working Group (rtgwg). <http://www.ietf.org/html.charters/rtgwg-charter.html>.
- [27] M. Shand and S. Bryant. IP Fast Re-route framework. <http://www3.tools.ietf.org/html/draft-ietf-rtgwg-ipfrr-framework-07>, June 2007.
- [28] M. Shand and S. Bryant. IP Fast Reroute Using Not-via Addresses. <http://www3.tools.ietf.org/html/draft-ietf-rtgwg-ipfrr-notvia-addresses-01>, July 2007.
- [29] N. Spring, R. Mahajan, and D. Wetherall. Measuring ISP topologies with Rocketfuel. In *Proc. ACM SIGCOMM*, Pittsburgh, PA, Aug. 2002.
- [30] R. White and B. Akyol. Considerations in Validating the Path in BGP. IETF Draft, 2007.
- [31] W. Xu and J. Rexford. MIRO: Multi-path Interdomain ROuting. In *Proc. ACM SIGCOMM*, Pisa, Italy, Aug. 2006.
- [32] X. Yang, D. Wetherall, and T. Anderson. Source selectable path diversity via routing deflections. In *Proc. ACM SIGCOMM*, Pisa, Italy, Aug. 2006.