Efficient Multicast Forwarding

https://helix-project.org

ABSTRACT

The amount of live video traffic has been increasing at high rates due to the proliferation of online social platforms such as Facebook Live. These platforms enable *anyone* to broadcast live videos to many users *anytime*. Such new applications have introduced a renewed interest in designing efficient multicast services. Current multicast systems do not scale because of the state, processing, and communication overheads imposed on routers. We propose a new system called Helix to efficiently realize multicast trees inside ISP networks. Helix splits information about every multicast tree as a label attached to packets and state stored at few routers. Helix encodes tree links using probabilistic filters into labels, produces state to address the probabilistic nature of these filters, and reduces this state by recursively encoding it multiple times. In the data plane, Helix forwards packets based on their labels and the state. We implemented Helix in a programmable data plane testbed. Our experiments show that Helix easily saturates a 10 Gbps link. Moreover, we compared Helix against the closest systems in the literature using simulations. Our results show that Helix achieves significant gains over the closest systems in the literature.

1 INTRODUCTION

Recent large-scale applications enable online sharing of live content. Examples of such applications include live Internet broadcast (e.g., Facebook Live and Periscope), IPTV [13], video conferencing [3] and massive multiplayer games [5]. The scale of these applications is unprecedented. For instance, Facebook Live aims to stream millions of live sessions to millions of concurrent users [23, 29]. As another example, BBC reported a peak of 9.3 million TV users watching the Euro 2016 opening, and 2.3 million users watched that event through the BBC Internet streaming service [22]. The amount of live streaming traffic is expected to occupy 17% of the total Internet video traffic by 2022 [6]. These applications thus have restored the need for scalable multicast systems. In particular, to reduce the network load of such applications, ISPs use multicast to efficiently carry the traffic through their networks. For example, AT&T has deployed UVerse and BT has deployed YouView multicast services.

Large content providers require ISPs that carry their traffic to meet certain quality objectives or service level agreements. ISPs thus need to carefully direct traffic flows of multicast sessions through their networks in order to satisfy these quality objectives. That is, the traffic flow of a multicast session may not always be forwarded on the shortest network paths within the ISP. Instead, the traffic flow of a multicast session is forwarded on a *customized* multicast tree, which is a distribution tree created over routers in the ISP to reach all end users of the multicast session, while achieving the required quality objectives. That is, a customized multicast tree is a general tree and is not necessarily a minimum spanning tree. For example, different methods have been proposed in the literature to create customized multicast trees that minimize the maximum link utilization such as [4, 14, 15, 25].

ISPs need to deploy a multicast forwarding system inside their networks to implement customized distribution trees. However, designing efficient multicast forwarding systems is quite challenging, as it needs to *simultaneously* address three main issues: (i) scalability, (ii) correctness, and (iii) generality. The scalability has been a major issue since the introduction of IP multicast [8]. The main concern is that group management and tree construction protocols (i.e., IGMP [2] and PIM [12]) require maintaining *state* at routers for each session. In addition, these protocols generate control messages among routers to refresh and update this state. Maintaining and refreshing state for each session impose significant overheads on core routers that need to support high-speed links. Thus, in practice, router manufacturers tend to limit the number of multicast sessions. For example, the Cisco ASR 9000 series can maintain up to 1,024 multicast sessions [7].

The correctness of a multicast forwarding system means that the data plane should forward packets *on and only on* links of the multicast tree and should not result in any forwarding loops in the network. Although a critical requirement for bandwidth-demanding applications, correctness may not be guaranteed by all multicast systems in the literature. For example, the LIPSIN system [17] may forward packets on links that do not belong to the multicast tree (referred to as false positives) and can also introduce loops in the network. Finally, the generality of a multicast system indicates that it can support different multicast distribution trees, traffic patterns, and network topologies. This is a key requirement for implementing customized trees that satisfy various quality objectives. Not all multicast systems in the literature possess this property. For example, Elmo [24] is designed for a specific topology for data center networks and cannot be used with other network topologies.

In this paper, we propose a new multicast forwarding system, called Helix, which addresses the above three challenges. Helix: (i) is general and can be used to implement customized multicast trees in any network topology, (ii) does not introduce any loops or false positive packets, and (iii) is scalable as it requires only a small state to be maintained at a fraction of the routers and does not impose significant processing overheads on routers. The key idea of Helix is to split the information about a multicast tree into two parts: *constant-size* label attached to packets and small state at *some* routers in the tree. This information-split architecture leads to substantial improvements in the data plane performance, as well as allows Helix to strike a balance between the communication (label) overhead and the memory and processing overheads imposed on routers to handle labels and forward traffic on the multicast tree. We present proofs to show the correctness of Helix.

To show its feasibility and practicality, we have developed a proof-of-concept implementation of Helix in a testbed that uses NetFPGA. Our experiments show that Helix can easily provide line-rate throughput and it consumes a negligible amount of the hardware resources. We have also implemented a simulator to compare Helix against the closest multicast forwarding systems in the literature using real ISP topologies. Our simulation results show that Helix is efficient and scalable, and it substantially outperforms the closest systems across all considered performance metrics. For example, Helix reduces the state per session by up to 18X compared to the closest label-based multicast forwarding system. Furthermore, compared to a rule-based system implemented in OpenFlow, Helix decreases the required state per session by up to 112X.

The organization of this paper is as follows. In §2, we summarize the related work in the literature. In §3, we present the details of the proposed system, and in §4 we present our testbed implementation and experiments. In §5, we analyze Helix and compare it against others using large-scale simulations. We conclude the paper in §6.

To avoid disrupting the flow of the paper, we present the proofs of the theorems in **Appendix A**. In addition, we describe a detailed example of Helix in **Appendix B**.

Ethics Considerations. *This work does not raise any ethical issues.*

2 RELATED WORK

We divide existing multicast forwarding systems into rule-based and label-based systems. Rule-based systems maintain the forwarding state (i.e., rule) about each session at routers. Label-based systems move the forwarding information to labels attached to packets.

Rule-based Systems. IP multicast [8] does not scale in real deployments [9] due to its state and communication overheads. Specifically, its group management and tree construction protocols, e.g., IGMP [2] and PIM [12], need to maintain state at routers belonging to the multicast tree. Moreover, to refresh and update this state, these protocols generate control messages that routers need to process. Furthermore, IP multicast uses shortest paths and cannot implement customized trees.

Multicast forwarding systems based on the match-action abstraction can implement customized trees by installing rules at routers. For instance, in OpenFlow [20], the controller installs rules to forward/duplicate packets on specified paths. However, the forwarding state at routers grows with the number of multicast sessions. To reduce the state, Li et al. [19] proposed a multi-class Bloom filter (MBF) that implement multicast trees in data center networks. For every interface in a router, MBF maintains a Bloom filter to indicate whether packets of a specific session should be duplicated on that interface. MBF assumes prior knowledge of joining events and session size to calculate the number of hash functions per session, which is not realistic for multicast sessions in the general ISP setting. MBF cannot completely avoid false positives, which introduce redundant traffic in the network. Furthermore, MBF imposes additional communication overhead to keep the filters at routers updated as multicast trees change over time.

In these systems, various events, e.g., router joining/leaving and link failures, trigger changing the multicast trees. These changes may require updating the state at many routers, which imposes additional communication overheads. Furthermore, frequent router updates could introduce inconsistency in the network state, especially for large ISPs. To avoid inconsistency, the control plane has to use a scheduling algorithm to gradually update the forwarding rules at routers [16]. Rule update scheduling reduces the network agility (i.e., the ability of the network to quickly react to dynamic changes) as the updated rules take time to be installed/activated at all intended routers; greedily updating rules may result in violating service level agreements [31].

Label-based Systems. These systems use labels to represent multicast trees. Examples incude mLDP [27], BIER [28], Elmo [24], and LIPSIN [17]. mLDP [27] forwards traffic on the shortest paths and cannot support customized trees. Moreover, it requires an additional protocol to distribute labels among routers. BIER [28] uses a bitmap to encode global IDs of tree receivers in a label, and similar to mLDP only supports trees that follow the shortest paths. Elmo [24] is designed for data center networks. It attaches multiple labels to packets, where each label represents a forwarding rule at

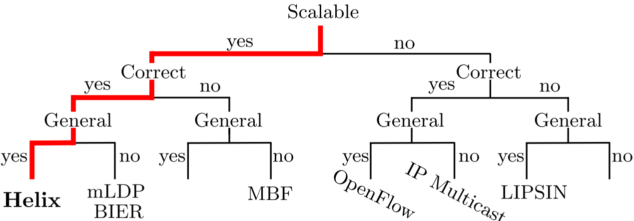


Figure 1: Properties of current multicast systems.

a router in every layer of a Clos topology. Various optimizations in Elmo rely on the considered Clos topology and thus cannot be easily used with general ISP networks.

LIPSIN [17] encodes link IDs of each tree using a Bloom filter, and attaches the resulting filter to packets. LIPSIN may result in loops and redundant traffic due to the false positives of Bloom filters. The authors of LIPSIN proposed a method to detect loops. However, this method makes each router keep a list of all packets that have recently been forwarded by that router, and it checks this list before forwarding every new packet to break any loop. This method imposes substantial memory and processing overheads on routers and thus is not scalable. LIPSIN does not have a method to eliminate false positives; it assumes that such packets will be dropped somewhere downstream. This wastes significant network resources especially for bandwidth-demanding applications such as live video streaming.

We **summarize** the properties of current multicast forwarding systems in Figure 1. Unlike Helix, current works in the literature do not *simultaneously* address the scalability, correctness, and generality challenges. In §5, we compare Helix against LIPSIN and OpenFlow because both systems can implement customized multicast trees.

3 PROPOSED HELIX SYSTEM

3.1 Overview

Multicast services can be used in various scenarios. A well-known use-case is when a major ISP, e.g., AT&T, manages multicast sessions for its own clients. Clients in this case can be end users in applications such as IPTV and live streaming. Clients could also be caches for content providers such as Netflix, where the contents of such caches are periodically updated using multicast. Another common use-case for multicast services happens when large-scale content providers, such as YouTube, Facebook, Periscope, and Twitch partner with ISPs to deliver live streams to millions of users.

Figure 2 provides a high-level overview of the proposed Helix system. Helix is designed for ISPs to manage different use cases of multicast within their networks. The considered ISP network has data and control planes. The data plane is a

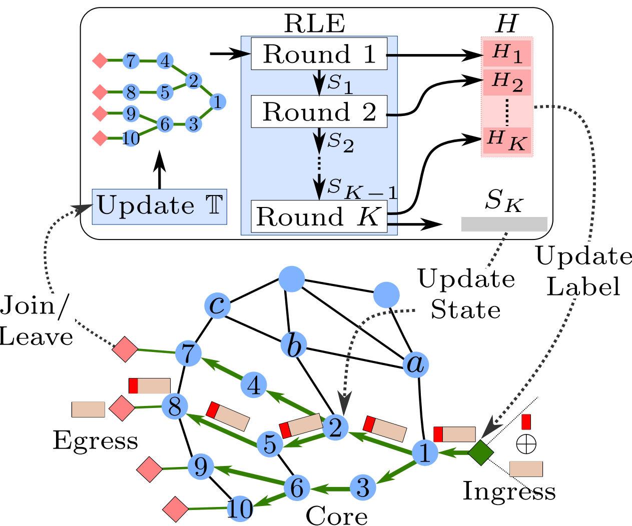


Figure 2: High-level overview of Helix.

set of routers and directed links connecting them. Each link l has a global ID $l.id$. The control plane is a centralized controller that computes and updates packet labels and states at routers to forward the traffic of multicast sessions according the quality objectives of the ISP. Examples of these objectives include minimizing the maximum link utilization and minimizing the delay.

A multicast session within the ISP has a source and multiple destinations. We refer to the source as the ingress router and the destinations as egress routers. End users (receivers of the multicast traffic) are typically connected to egress routers. Each multicast session has an ID included in all packets belonging to that session. This ID can be inserted, for example, in the MAC destination address. A multicast session is represented by a distribution tree $\mathbb{T}$, which spans the ingress and egress routers of the session. The set of routers that $\mathbb{T}$ spans are denoted by $\mathbb{R}$. Links of this tree (referred to by $\mathbb{L}$) do not necessarily follow the shortest paths; rather they are chosen to achieve the quality objectives of the ISP.

The ISP controller and content providers communicate through application-layer APIs to initiate the session and calculate the desired trees. When a multicast session changes (e.g., starts, ends, or user joins/leaves), the corresponding ingress or egress router sends a request to the controller. The controller updates the distribution tree $\mathbb{T}$ based on the service level agreements and the session events. Various algorithms can be used to compute customized trees such as [4, 14, 15, 25]. In this paper, the desired trees are given to the controller and our work is to efficiently realize such trees in the network and support their dynamic changes.

There are two modules in Helix: (i) Creating Labels, which runs at the controller, and (ii) Processing Labeled Packets,

which runs in the data plane at individual routers. For every tree $\mathbb{T}$, the controller calculates a label to be attached to packets of the session and small state at some routers. It uses a probabilistic set membership data structure (aka *filter*) to encode the tree link IDs in a relatively small label using hash functions. Since these filters trade off membership accuracy for space efficiency, they may result in *false positives*, which occur when some links that do not belong to the multicast tree are incorrectly included in the computed filter. False positives waste network resources, overload routers, and could create loops. In Section §3.2, we present the details of our method of creating labels while eliminating false positives.

In Helix, various routers process packets as follows. The ingress router of a multicast session attaches the label to packets of that session. Egress routers receive joining/leaving messages from end-users, and send them to the controller to update the multicast trees. Moreover, egress routers detach labels from packets coming from the core network to transmit them to end users. Core routers, on the other hand, use our algorithm (in §3.3) to forward multicast packets. The algorithm checks labels attached to packets to decide which interfaces to duplicate these packets on. In our design, labels do not change as packets traverse the network. Furthermore, labels are carefully structured so that they can be processed at high speed without overloading routers.

We describe a detailed example in Appendix B to illustrate the various components of Helix.

3.2 Creating Labels

Designing a label creation algorithm is a challenging task. This is because it needs to achieve the generality, scalability and correctness requirements. To achieve generality, the proposed algorithm encodes the tree link IDs of every tree using filters, which allow the multicast tree to include links not on the shortest paths. Although these probabilistic filters produce relatively small labels and can be processed efficiently in the data plane, they may result in false positives (i.e., incorrect forwarding). To ensure correctness, the data plane needs to store these false positives and check them before making decisions. This impacts the scalability of the data plane as it needs to maintain more state.

The key idea to achieve these objectives is to encode both tree and false-positive link IDs inside the label. Thus, the data plane does not need to maintain *all* false positives. We design an algorithm based on *recursive encoding*. Our recursive encoding algorithm works in rounds to successively reduce the final state to be maintained at routers. Each round encodes a given set of link IDs based on the outputs of previous rounds, and produces a fixed-size label and an intermediate state. The proposed algorithm carefully controls the inputs of each round to guarantee the correctness of the final outputs. It has provable guarantees on its correctness as well as time and space complexities. In addition, by parameterizing the number of rounds, the algorithm can control the trade-off between the label size and maintained state.

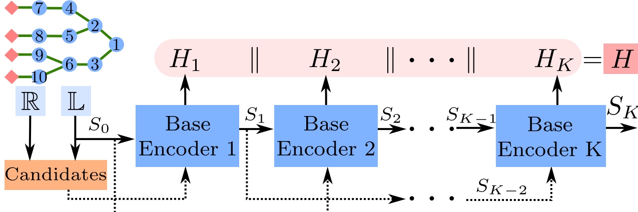


Figure 3: Overview of the proposed Recursive Label Encoder (RLE).

The algorithm for creating labels takes as input a multicast tree $\mathbb{T}$, and produces two outputs: label H and state S. **??** shows a high-level overview of the proposed algorithm, which we call Recursive Label Encoder (RLE). RLE *recursively* calls the BaseEncoder function K times. The BaseEncoder encodes a given set of link IDs into a label of size B bits and a state S. The state can have zero or more entries, and each takes the form $\langle r,\ linkID \rangle$, where r is the router that should maintain this state and $linkID$ is the ID of the link identified as a false positive during the encoding process.

The inputs to each recursive call of the BaseEncoder are carefully selected to successively reduce the number of entries in the final state as well as facilitate the packet processing at core routers. The label of the session is created by concatenating all intermediate labels produced from the K recursive calls. The state remained after the Kth call is the final state used in the forwarding decisions. The BaseEncoder uses a probabilistic set membership data structure (filter) to produce small labels. The BaseEncoder supports any filter that can: (1) add an item to an existing filter, (2) test whether an item exists (potentially with false positives), and (3) avoid false negatives. A false negative happens when a link in the multicast tree $\mathbb{T}$ is not represented in the filter. These features are important to create labels and to develop our packet processing algorithm. Examples of such data structures are Bloom [1] and Cuckoo [11] filters.

We define two functions that the filter supports: (i) $H = E_B(l; h)$ to encode an input item l (link ID in our case) into a bit string H of size B bits using hash function h, and (ii) $C_B(l, H; h)$ to check whether a given item l belongs to H using hash function h. Since the BaseEncoder is recursively called K times and each with a different hash function, we use a set of K independent hash functions denoted by $\mathcal{H}_k = \{h_1, \ldots, h_K\}$.

Algorithm 1 lists the pseudo code of the RLE algorithm, which implements the ideas in **??**. RLE algorithm recursively

Algorithm 1 The Recursive Label Encoder (RLE) algorithm.

```
Input: 𝕋: multicast tree
Input: K: number of filtering rounds
Input: B: number of bits per filter
Input: ℋ = h_1, ..., h_K: set of K hash functions
Output: H: label to be attached to the session packets
Output: state: state sent to a subset of the core routers
 1: function RLE(𝕋, K, B)
 2:   H = {}
 3:   state = 𝕃
      // initial false positive candidates
 4:   candidates = FindFPCandidates(𝕋.ℝ, 𝕋.𝕃)
 5:   for (k = 1; k ≤ K; k++) do
 6:     ⟨H_k, S_k⟩=BaseEncoder(state, candidates, B, h_k)
 7:     H = H ∪ H_k
 8:     candidates = state
 9:     state = S_k
10:   return ⟨H, state⟩
11: function BaseEncoder(links, candidates, B, h)
12:   label = BitString(size=B)
13:   state = {}
14:   for (l ∈ links) do
15:     label = label ∪ E_B(l.id; h)
      // Calculate state
16:   for (l ∈ candidates) do
17:     if (C_B(l.id, label; h)) then // false positive
          // add (router ID, link ID) to state
18:       state = state ∪ {⟨l.src, l.id⟩}
19:   return ⟨label, state⟩
20: function FindFPCandidates(ℝ, 𝕃)
21:   cand = {} // calculated false positive candidates
22:   𝕃̄ = {(l.dst → l.src) for l in 𝕃} // upstream links
23:   for (u ∈ ℝ) do
24:     for (l ∈ u.links) do
25:       if (l ∉ {𝕃 ∪ 𝕃̄}) then
26:         cand = cand ∪ (u → l.dst)
27:   return cand
```

calls the BaseEncoder K times with different inputs. The main building block is the BaseEncoder (Lines 11–19). It encodes every link l in the set of links passed to it using the specified hash function h. Then, it calculates the state that needs to be maintained at routers to avoid false positives. It does so by checking all false positive candidates passed in the *candidates* variable and adding only the entries that collide with the tree links encoded in *label* to the returned state.

Two important aspects need to be addressed: (i) determining the *initial* set of false positive candidates, which is the set used in the first round of encoding, and (ii) controlling the input parameters to the BaseEncoder in each of the K encoding rounds. Both directly impact the correctness and efficiency of the proposed label creation algorithm.

Computing Initial False Positive Set. The challenge of finding the initial false positive candidates is to produce a complete *and* minimal set. An incomplete set may result in false positives that are not accounted for in the state, which means forwarding multicast packets to links that do not belong to the multicast tree. A non-minimal set increases the state overhead by including state entries that will never be used. For example, choosing a candidate set of all links in the network clearly results in a complete set. However, this set is not minimal and many non-tree links may needlessly collide with the tree links in the calculated label. The key idea of finding this minimal set is that only outgoing links attached to routers belonging to the tree nodes $\mathbb{R}$ could be false positives. This is because routers in $\mathbb{R}$ do not forward packets on non-tree links since they store these links as state or these non-tree links are encoded in the label. Thus, if packets of a session would not reach a non-tree router, there is no need to check links attached to that router. For example, in Figure 2, link $(a \to b)$ is removed from the set because router 1 will not forward packets to a.

The following theorem states the conditions for the initial false positive candidate set to be complete and minimal.

Theorem 1. *The initial false positive candidate set is complete and minimal if it is composed of every link $l = (u \to v) \notin \mathbb{L}$ provided that $u \in \mathbb{R}$ and $(v \to u) \notin \mathbb{L}$, where $\mathbb{R}$ and $\mathbb{L}$ are the sets of nodes and links of the multicast tree $\mathbb{T}$, respectively.*

Proof. The details of the proof are given in the Appendix. The idea of the proof is to derive the necessary conditions for a link to be a false positive using basic routing constraints, such as a router should not forward a packet back on the same interface it received the packet on. We enumerate all possible links in the network, and keep the ones that meet the necessary conditions in the false positive candidate set. □

The above theorem allows us to remove many links from the false positive candidate set. For example, links $(2 \to 1)$, and $(a \to b)$ in Figure 2 cannot be false positives and thus are not considered in the candidate set.

Given Theorem 1, the function FindFPCandidates (Lines 20–27) in Algorithm 1 calculates the initial false positive candidate set. It initializes an empty set *cand*, reverses every link in $\mathbb{L}$ from $(l.src \to l.dst)$ to $(l.dst \to l.src)$, and stores this new set in $\overline{\mathbb{L}}$. Note that every non-tree link in $\overline{\mathbb{L}}$ is an upstream link that cannot carry traffic of $\mathbb{T}$. FindFPCandidates then iterates over all routers in $\mathbb{R}$ (Line 23), and for

every router u, it adds to *cand* the non-tree links that are adjacent to u and do not belong to $\mathbb{L}$ or $\overline{\mathbb{L}}$.

Controlling Inputs for BaseEncoder. The second important aspect in the RLE algorithm is controlling the input parameters to the BaseEncoder to successively reduce the state across the K encoding rounds, without complicating the packet processing algorithm at core routers. As illustrated in ??, every round k of the algorithm produces a label H_k and an intermediate state set S_k. Each round expects two inputs: (1) links to be encoded S_{k-1} (solid lines), and (2) false positive candidates S_{k-2} that may collide with S_{k-1} in H_k (dotted lines). These inputs are the resulting states from previous rounds, and they are subsets of the tree links $\mathbb{L}$ and the false positive candidates. By including H_k in the final label, the algorithm removes the false positive candidates in S_{k-2} that are not stored in S_k. That is, in each round k, the algorithm decides that some links from that round false positive candidates S_{k-2} are not needed in the final state S_K. This is because filters in Helix do not produce false negatives, and H_k does not include them. As a result, the algorithm removes these links from the resulting state of this round, and only keeps the links S_k that may be needed in the final state.

Formally, to remove tree and non-tree links from the final state, we extend the definitions of the links to be encoded and the false positive candidates as follows. We denote the tree links $\mathbb{L}$ by S_0, and the initial false positive candidate set by S_{-1}. For round k, the links to be encoded are the resulting state from the previous round S_{k-1}. The false positive candidates are the encoded links of the previous round S_{k-2}. We define the outputs of round k as:

$$H_k = \text{Encode link IDs in } S_{k-1} \text{ using } E_B, \text{ and} \quad (1)$$

$$S_k = \text{Calculate state using } C_B, H_k, \text{and } S_{k-2} \quad (2)$$

Generally, RLE uses K labels to remove links belonging to $S_{-1}, S_0, \ldots, S_{K-2}$ from S_K. For example, when $K = 1$, RLE removes links from the initial candidate set S_{-1} (i.e., the BaseEncoder). When $K = 5$, RLE uses five labels to remove links belonging to $S_{-1}, S_0, S_1, S_2, S_3$ from the final state S_5.

Finally, given the structure of RLE in ??, the resulting state in each round contains either tree or non-tree links (but not a mixture of them). To elaborate, in the first round, RLE encodes tree links into H_1. It takes S_{-1} as candidates and produces S_1 as state. Note that S_1 is a subset of S_{-1}, hence, the resulting state S_1 contains non-tree links only. Similarly, in the second round, RLE encodes S_1 into H_2 and uses S_0 as false positive candidates. Thus, the state S_2 contains tree links that is a subset of S_0, and collides with S_1 in H_2. Formally, the following theorem states this property.

Theorem 2. *In the Recursive Label Encoder (RLE) algorithm, if round k is even, the links encoded in the output state S_k are tree links. Otherwise, these links do not belong to the multicast tree.*

Proof. The details of the proof are given in the Appendix. The idea is to derive a relationship across resulting states, and prove the theorem by induction. □

Time and Space Complexities. It is straightforward to show that the time complexity of the RLE algorithm is $O(NI + KM)$ and its space complexity is $O(NI + M)$, where N is the number of routers, M is the number of links, I is the maximum number of interfaces per router, and K is number of filtering rounds. The number of routers, interfaces and links in ISP networks are usually in the orders of tens to hundreds. And from our experiments (§5), K in the range of 4 to 7 results in good performance for most practical multicast trees and network sizes. Recall that the RLE algorithm runs in the control plane and invoked only at the creation of a multicast tree and whenever it changes. Thus, the proposed label creation algorithm can easily be deployed in real scenarios.

3.3 Processing Labeled Packets

The ingress and egress routers of every multicast session perform simple operations on packets. The ingress router attaches the label H received from the controller to every packet in the multicast session. Egress routers detach labels from packets before forwarding them towards end-users. In this section, we focus on the packet processing algorithm needed by core routers to forward packets of multicast trees.

This algorithm needs to achieve two goals. First, it has to forward packets on and only links belonging to the tree (i.e., correctness). It does so by exploiting the structure of RLE and the absence of false negatives to make its decisions. Second, it should perform at line rates and consume small amount of hardware resources. We design the algorithm to achieve high performance per packet by executing its operations across links and rounds at same time. In addition, we design the algorithm to use simple instructions, e.g., bitwise operations, that do not require large hardware resources. Moreover, the algorithm only maintains small data structures (i.e., bit-vectors) to keep intermediate computation results per packet.

Upon receiving a packet at a core router, the algorithm reads both the session ID and the attached label H. The algorithm does not share information across different packets. That is, the algorithm checks the label and the state (if any) maintained by the router to decide which of the router's links belong to the multicast tree. The pseudo code of the proposed packet processing algorithm is shown in Algorithm 2. To efficiently utilize the available hardware resources and accelerate processing, the algorithm makes decisions about all links at same time. Specifically, for each link l, the algorithm checks all the K components $H_1, \ldots, H_K$ of the label H as

follows. The algorithm searches for the *first* label component H_k that does *not* contain the input link ID, $l.id$ (Line 3). If the algorithm finds such label and k is even, the algorithm duplicates the packet on the link l. This decision is driven by the design of RLE. Since filters in Helix do not produce false negatives, the algorithm decides (without error) that the input link does not belong to this label component. Moreover, given the structure of RLE, the candidates of the round S_{k-2} contain the link $l.id$, because H_k is the first label that does not contain $l.id$. The algorithm decides whether these candidates are tree or non-tree links based on the value of k. If k is even, $k-2$ is even as well. Hence, the links in S_{k-2} are tree links (Theorem 2). If k is odd, these candidates are non-tree links. Checking whether k is even in Line 4 can be done by testing the first bit of k, which can be easily implemented in hardware. Notice that routers can use the results of Theorem 2 without storing the actual resulting states $S_1, \ldots, S_{K-1}$.

When $l.id$ exists in all label components, the algorithm needs to check the maintained state $State$. This is because filters in Helix may result in false positives. Recall that this maintained state is a subset of S_K that is calculated by RLE. The algorithm uses the results of Theorem 2 in this case as well. If K is even, the router knows that the links in $State$ are tree links. Thus, the algorithm duplicates the packet if $l.id$ exists in $State$[ID] (Line 6). On the other hand, if K is odd, the algorithm realizes that the maintained state contains non-tree links. In this case, the algorithm duplicates the packet only if $l.id$ does not exist in $State$[ID].

The following theorem proves the correctness of Algorithm 2.

THEOREM 3. *The packet processing algorithm, Algorithm 2, duplicates packets on and only on links that belong to the multicast tree.*

PROOF. We present the proof in the Appendix. The idea of the proof is to utilize some properties of the used filters and the structure of RLE. For example, a resulting state at any round contains either tree or non-tree links. We enumerate all cases in which a link belongs to the label, and derive the required conditions to duplicate a packet on a given link. □

Implementation Notes. We perform two further optimizations to execute our algorithm at line rate. First, we do not make routers execute the membership checking function C_B. Instead, each router stores K bit strings for every link attached to it. These bit strings are calculated by the controller once. We use Bloom filter in our experiments, but other filters can be used as well. Second, we check the K label components at same time by unfolding the logic inside the loop in Lines 2–4. For a representative platform with programmable processing pipeline such as FPGA, the loop can be unfolded across two clock cycles as follows. For each link, the first clock cycle checks whether this link belongs to all label components. For a given link, the second clock cycle finds the first label component where that link does not belong to. This fast implementation comes at a simple cost of storing a small bit-vector when running our algorithm. We discuss the details in §4.

Algorithm 2 Process labeled packets at core router.

```
Input: l: link attached to the router
Input: H: Helix label
Input: State: a subset of S_K stored at the router
Input: K: number of filtering rounds
Input: 𝓗 = h_1, ..., h_K: set of K hash functions
Output: true if duplicating a pkt on link l, else false
   // Runs for every link l attached to the core router
1: function ProcessLabeledPacket(l, H, State, K)
2:   for (k = 1; k ≤ K; k++) do
3:     if (not C_B(l.id, H_k; h_k)) then
4:       return k % 2 == 0
     // link l exists in H_1, H_2, ..., H_K
     // ID is the session ID included in the packet header
5:   if (K is even) && (l.id ∈ State[ID]) then
6:     return true
7:   if (K is odd) && (l.id ∉ State[ID]) then
8:     return true
9:   return false
```

4 EVALUATION IN A TESTBED

We present a proof-of-concept implementation of the packet processing algorithm of Helix in a testbed. We realize that core routers have many different hardware designs and software stacks. The goal of this section is to show that our proposed ideas can be implemented in a representative platform (NetFPGA). For instance, the line cards of Huawei NetEngine NE5000E core router are implemented using FPGA [30]. Although the implementation of Helix in other platforms will be different, the conceptual ideas are the same.

We use this testbed to show that Helix can sustain line-rate performance with minimal overhead on the hardware resources, and to demonstrate the correctness of Helix.

4.1 Testbed Setup and Algorithm Implementation

Hardware. We implemented a router with a programmable processing pipeline using NetFPGA SUME [32]. The router has four 10GbE ports and it is clocked at 156.25 MHz. The testbed also has a 40-core server with an Intel X520-DA2 2x10GbE NIC, which is used to generate and consume traffic

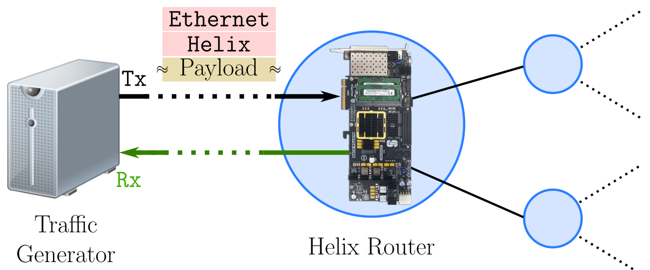


Figure 4: Setup of our testbed.

	nf1	nf2	nf3
Expected	38	7	10
Helix	38	7	10
LIPSIN [17]	38	10	37

Table 1: Validation of forwarding decisions made by our packet processing algorithm.

at high rate. The router and the server are connected with two SFP+ fiber optic cables. As shown in Figure 4, our implementation represents a core router in an ISP topology. Since the router has four ports, we use one port to transmit packets from the traffic generator to the router. Another port receives packets at the traffic generator after being processed by the router. The remaining two ports are traffic sinks.

Implementation. Our implementation is based on the `reference_switch_lite` project for NetFPGAs [21], which we synthesize using the Xilinx Vivado design suite. This router contains three main modules: `input_arbiter`, `output_port_lookup` and `output_queues`. The `input_arbiter` dispatches one packet at a time from the input queues to the `output_port_lookup`. The `output_port_lookup` decides the ports to duplicate packets on by creating a bitmap of the destination ports. The `output_queues` module forwards the packet based on the bitmap. Our implementation modifies the `output_port_lookup` module to calculate the bitmap based on the packet processing algorithm as follows. It contains a memory block to store the state. If the controller instructs the router to maintain state for a specific multicast session, our implementation maps the session ID to an I-bit vector, where I is the number of links attached to the router. For every session maintained in memory, if a link $i \in \{0, \ldots, I-1\}$ is to be stored in this router, the corresponding bit at index i is set to 1. For every port, the router knows the mapping between this port and link ID. In addition, the router stores K bit strings each of size B bits for every port. These bit strings are the hashed link IDs created by the controller. We use Bloom filter and Murmurhash3 as the hashing function.

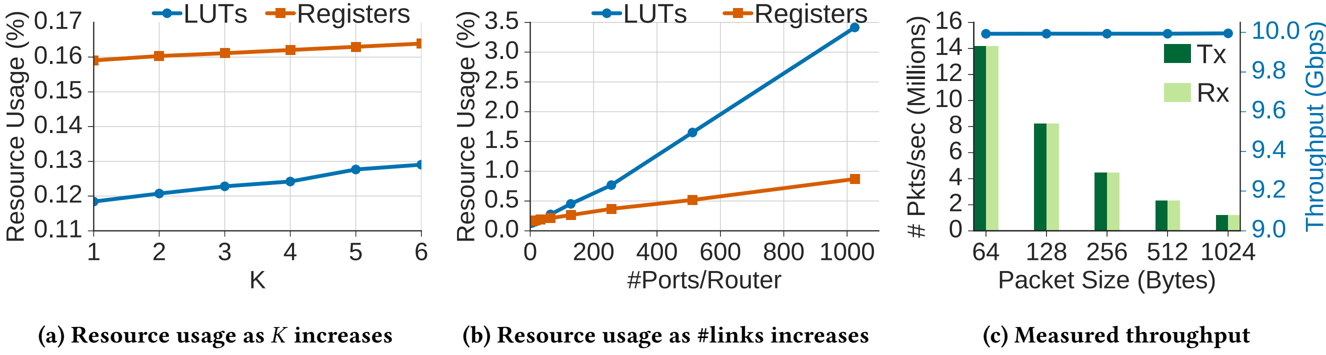
When our implementation receives a labeled packet, it parses the K label components by reading the first $K \times B$ bits that follow the Ethernet header. Once the label components are parsed, our implementation runs in three clock cycles (19.2ns) to determine the output ports. In the first clock cycle, the module checks in parallel for each link whether it belongs to every label component H_k. Checking whether a link belongs to a label component H_k in Bloom filter is a bitwise-AND operation between the kth hashed link ID and H_k. This operation is done in one clock cycle. Then, the algorithm stores these results in an $(I \times K)$-bit vector. For every link, the second clock cycle uses the resulting bit vector to detect the index k of the first label where this link does not exist. If one link ID exists in all label components, the algorithm requests a memory read using the session ID. The third clock cycle specifies the final output ports based on the value of k (even/odd) and the state maintained at the router for this session (if any).

Traffic Generation and ISP Topology. The 40-core server is used to generate traffic using MoonGen [10], which allows creating new network protocols such as Helix using Lua language. Since layer-3 protocols are the payload of Helix, we do not generate any layer-3 specific headers. MoonGen can measure the throughput of the received traffic at the server as well. The arrows in Figure 4 show the direction of the traffic. We transmit traffic of sessions on one port of the NIC, and receive it through the router on the other port.

We consider a real ISP network topology with 125 routers, chosen from the Internet Topology Zoo [26]. We make our router act as one of these routers. Using this topology, we randomly generate 40 multicast sessions. Every session has a multicast tree covering various routers in the topology. For each session, we use RLE to encode the tree into a label. We set K to 4 rounds and B to 32 bits in our algorithm.

4.2 Results

Correctness of Forwarding Decisions. We first validate the forwarding decisions made by the proposed packet processing algorithm. We wrote a Python script that uses the NetFPGA testing APIs. This script transmits one packet on interface `nf0` for each of the 40 multicast sessions. It compares the observed forwarding decision of the algorithm against the expected behavior knowing the multicast tree of each session. Our results confirmed that all packets were forwarded as expected, with no false positives (i.e., no packet was forwarded to any link that does not belong to the corresponding multicast session) and no false negatives (i.e., all links that belong to a multicast session received the packet of that session).



(a) Resource usage as K increases **(b) Resource usage as #links increases** **(c) Measured throughput**

Figure 5: Performance of Helix using the testbed.

We summarize the packet counts on the different interfaces of the router in Table 1. For comparison, we also implemented the closest label-based multicast forwarding system in the literature, called LIPSIN [17] which is briefly described in Section 2. The first row in the table shows the expected number of packets on each interface for the 40 sessions passing through the NetFPGA router, which exactly matches the number resulted by running our algorithm (second row). The third row (obtained by running LIPSIN) indicates that LIPSIN results in many false positives. For example, interface `nf3` received 37 packets instead of the expected 10 packets.

Resource Usage and Scalability. We measure the resource usage of the packet processing algorithm, in terms of the number of used look-up tables (LUTs) and registers in the NetFPGA. These numbers are generated by the Xilinx Vivado tool after synthesizing and implementing the project. We vary the number of filtering rounds K from 1 to 6 and plot the numbers of LUTs and registers used by our algorithm in Figure 5a. The figure shows that our algorithm utilizes a tiny amount of the available hardware resources. For example, when K is 6, our algorithm requires 559 LUTs and 1,420 registers, which represent only 0.13% and 0.16% of the available LUTs and registers, respectively. Moreover, increasing K from 1 to 6 requires only additional 46 LUTs and 42 registers. Thus, our packet processing algorithm scales well as the number of rounds increases.

Next, we analyze the performance of our algorithm as we increase the number of ports from 4 to 1,024. We increase the number of ports by controlling a parameter in our Verilog implementation. Since our router has only four ports, we map each of the additional ports to a physical port in a round-robin fashion. We set the number of filtering rounds K to 6. The results of this experiment are shown in Figure 5b, which shows that Helix scales as we increase the number of ports. For example, for a large router with 1,024 ports, our algorithm uses only 3.4% and 0.87% of the available LUTs and registers, respectively.

Throughput Measurement. We show that our packet processing algorithm can easily handle packets at line rate. In this experiment, we use MoonGen to generate labeled packets of multicast traffic at 10 Gbps. We transmit these packets from the traffic-generating server to the router on interface `nf0` for 60 sec. We use MoonGen to count the number of packets per second received at each of the other interfaces of the router. We vary the packet size from 64 to 1,024 bytes. We run the experiment five times for every packet size and compute the average across them. In Figure 5c, we compare the average number of input packets per second to the router (received on interface `nf0`) against the average number of packets per second observed on one of the output interfaces (interface `nf1`); the results for other interfaces are the same. The figure shows that the numbers of transmitted and received packets per second are the same (i.e., no packet losses). We plot the achieved throughput in the same figure. The figure shows that our algorithm can sustain the required 10 Gbps throughput for all packet sizes.

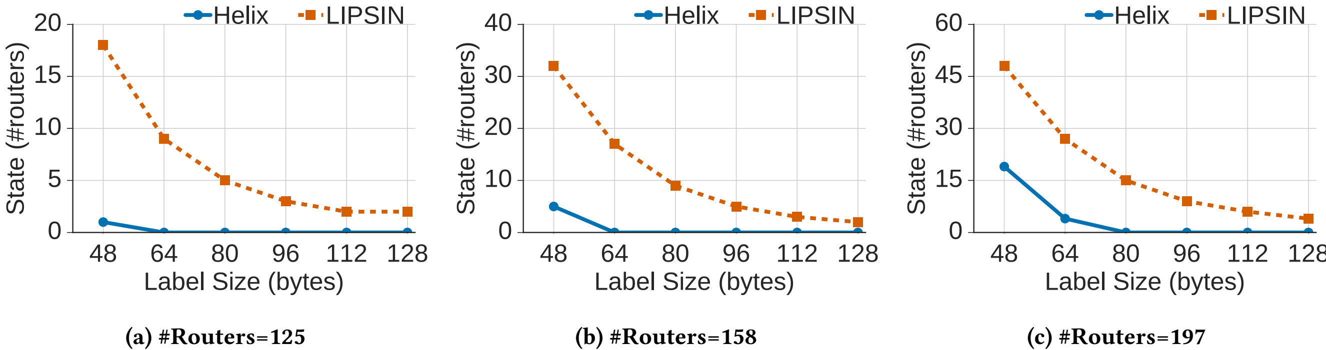
5 EVALUATION USING SIMULATION

In this section, we compare Helix against the closest approaches in the literature in large-scale simulations. And we analyze the effect of varying various Helix parameters.

5.1 Setup

We implemented a Python-based simulator that acts as a what-if scenario analysis tool. The simulator allows us to evaluate the performance of different multicast forwarding systems in large setups in terms of label size, topology size, receiver density and number of sessions. The core component of the simulator implements the Helix controller. It receives an event such as a router joining/leaving a session, updates the corresponding multicast tree, and then generates labels and states based on the used algorithm.

Performance Metrics and Control Parameters. We consider two main performance metrics:



(a) #Routers=125

(b) #Routers=158

(c) #Routers=197

Figure 6: The effect of label size on the session state for Helix and LIPSIN.

(1) *Session state:* number of routers that need to maintain state for a session.
(2) *Number of messages per tree update:* number of messages sent to routers by the controller when a multicast tree is updated to modify the state at these routers.

We report the 95-percentile of these metrics, because it reflects the performance over extended number of sessions or period of time. We repeat every experiment five times and report the average of the 95-percentile of every metric across these repetitions.

The main parameters we control in the experiments are:

(1) *Label size:* We vary the label size from 48 to 128 bytes.
(2) *Receiver density:* It is defined as the number of routers that join a session divided by the total number of routers. We vary the maximum receiver density from 10% to 40%.
(3) *Topology size:* We use 14 real ISP topologies from the Internet Topology Zoo datasets [26]. They represent a wide range of ISPs, where the number of core routers ranges from 36 to 197, and the number of links ranges from 152 to 486.

Systems Compared Against. We compare Helix versus the closest label-based multicast forwarding system, which is LIPSIN [17]. LIPSIN encodes the tree link IDs of a session using one filter. For every link, the LIPSIN controller maintains D link ID tables with different hash values. LIPSIN creates the final label by selecting the table that results in the minimum false positive rate. Every router maintains D tables for every link attached to it. Since LIPSIN may result in false positives, each router maintains state about incoming links and the label that may result in loops for every session passing through this router. To ensure fairness, we use the same parameters proposed by LIPSIN: we set D to 8 tables and use five hash functions per link. For both Helix and LIPSIN, we use Bloom filters and Murmurhash3 hashing functions. When Helix uses K filtering rounds and B bits per round, LIPSIN encodes the links in a label of size $K \times B$ bits. Thus, both systems use the same label size.

In addition, we implemented a rule-based multicast forwarding system using OpenFlow [18], because rule-based is a general packet processing model that is supported in many networks. The rule-based system installs match-action rules in routers to implement the multicast trees.

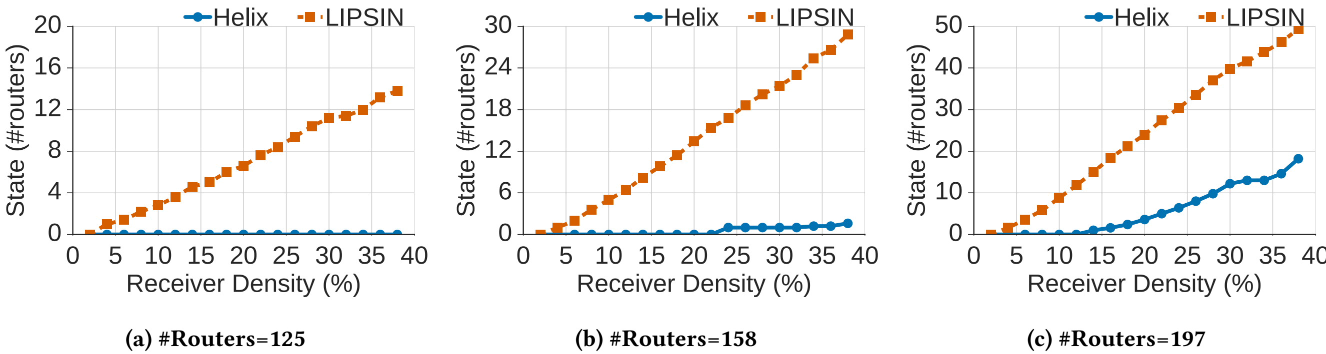
Session Dynamics. For every topology, we simulate 2,000 dynamic multicast sessions for 12 hours, where receivers join and leave sessions over time. The sources of multicast sessions are uniformly distributed among the routers. For each session, the maximum receiver density is randomly chosen to be either 10%, 20%, 30% or 40% of routers in that topology. The session bandwidth is randomly assigned to one of {0.5, 1, 2, 5, 10} Mbps values.

We simulate multicast session dynamics such as router joining/leaving as follows. For every router, we generate events following a Poisson distribution with an average rate of λ events per minute, where λ is computed by dividing the number of sessions by the number of routers. For every event at a router, we randomly pick a multicast session, and the router randomly joins or leaves the multicast session with probabilities 0.6 and 0.4, respectively. Since Helix does not dictate how multicast trees are computed, we use an algorithm similar to [15] to calculate multicast trees based on link utilization.

Due to space limitation, we only present a representative sample of our results.

5.2 Comparison against LIPSIN

Label Size. We first analyze the impact of varying the label size on the session state and number of messages per tree update. As illustrated in Figure 6 for three representative topologies, our results show that Helix requires maintaining state at fewer routers compared to LIPSIN, and it eliminates the state completely in many cases There are two points to be observed from the figure. First, Helix achieves better



(a) #Routers=125 **(b) #Routers=158** **(c) #Routers=197**

Figure 7: Impact of receiver density on the session state for Helix and LIPSIN. The label size is 64 bytes.

performance using the same label size. For example, when the label size is 48 bytes, 95% of the sessions in LIPSIN need to store state at 18, 32, and 48 routers for the three topologies, respectively. Whereas for the same label size, Helix requires state at 1, 5, and 19 routers for the three topologies, respectively. That is, Helix achieves at least 2.5X and up to 18X reduction in the state compared to LIPSIN. Second, Helix eliminates the need to maintain state with much smaller label sizes compared to LIPSIN. For example, when the topology size is 158 routers and the label size is 64 bytes, Helix does not require any router to store state. LIPSIN, on the other hand, maintains state at 17 routers for the same case. A side benefit of reducing the number of routers maintaining state is reducing the number of messages per tree update sent from the controller (figures are omitted). For example, for topology of size 158 routers and label size of 64 bytes, LIPSIN sends 19 messages to update each session, whereas Helix sends only 1 message. Reducing number of messages improves network agility in case of routers joining/leaving sessions and network failures.

Receiver Density. We next study the the impact of the receiver density on the session state and the number of messages per tree update when the label size is 64 bytes. Figures 7a–7c show that the session state of LIPSIN increases linearly as the receiver density increases. On the other hand, the session state in Helix increases at slow rate with increasing the receiver density. For the topology of 125 routers, for example, Helix does not result in state at any router. LIPSIN, however, requires maintaining state even when the receiver density is 10%. For the topology of 158 routers, Helix does not maintain state at any router for receiver density up to 22%. For the largest topology, Helix reduces the session state by up to 15X compared to LIPSIN. As a result of reducing the session state, Helix reduces the number of message per tree update by up to 19X, 20X and 16X for the three sample topologies, respectively (figures are not shown).

False Positives and Loops. We analyze the false positive overheads of LIPSIN. We calculate this overhead by dividing the number of sessions that are falsely passing through a link by the exact number of sessions that should pass through the same link. Our results show that LIPSIN imposes large overheads due to false positives. For example, when the topology size is 158 routers and label size is 64 bytes, LIPSIN results in a false positive rate of 37%. For the same topology, LIPSIN needs to increase the label size to 128 bytes to eliminate this overhead. These false positives not only impose redundant traffic, but they also can introduce forwarding loops. For the same topology and label size, if LIPSIN routers do not store state as well as additional per-packet information to detect loops, our simulation shows that there would be 250 sessions with loops (i.e., 12% of sessions).

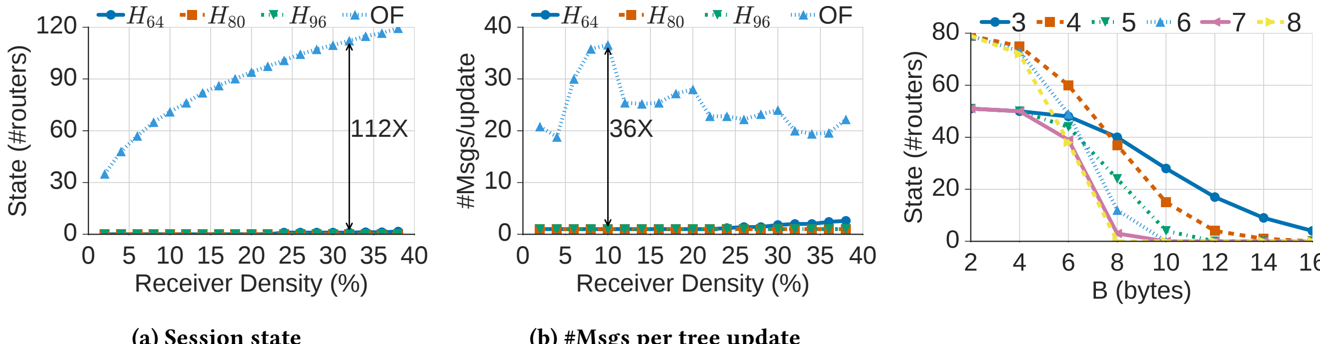
Helix does not incur these overheads because it eliminates false positives.

5.3 Comparison against OpenFlow

We compare Helix versus a rule-based approach implemented using OpenFlow. Since OpenFlow does not use labels, we only analyze the impact of receiver density on the performance. We use three label sizes for Helix: 64, 80 and 96 bytes.

Figure 8a depicts the session state for one sample topology of 158 routers. The figure shows that Helix outperforms OpenFlow for all receiver densities. For example, when the receiver density is 38%, the 64-byte label results in storing state at up to 2 routers (1.3% of the routers) for 95% of the sessions. Using 80- and 96-byte labels do not result in any state for any receiver density. For the same topology, OpenFlow requires installing match-action rules at up to 120 routers when the receiver density is 38%.

Increasing the session state at OpenFlow routers not only consumes their limited memory resources, but also increases the communication overhead when updating this state (i.e., when a session is updated). Figure 8b depicts the number of messages per tree update for Helix and OpenFlow systems



(a) Session state

(b) #Msgs per tree update

Figure 8: Performance of Helix vs OpenFlow.

Figure 9: Impact of filter size B on the session state for multiple values of K.

for the sample topology. The figure shows that OpenFlow requires sending 36 messages to update state when the receiver density is 10%. For the same receiver density, Helix sends only one message to update the label at the ingress router. When the receiver density is 38%, the number of messages per tree update of Helix is only 3 when the label size is 64 bytes. OpenFlow, on the other hand, requires sending 22 messages to update state at corresponding routers.

5.4 Analysis of Helix Parameters

Choosing working ranges for Helix parameters is important in order to achieve the expected performance gains. We study the impact of choosing B and K on the session state for six topologies of different sizes. In Figure 9, we show the session state for one of the topologies (158 routers) for six values of K while increasing B from 2 to 16 bytes. Other figures are similar and omitted due to space limitations. We can draw three conclusions from the results.

First, choosing a proper value for B is more important than increasing K. For example, in Figure 9, increasing K to 8 when B is small (i.e., 2–4 bytes) does not reduce the session state. This is because small filters cannot encode tree and non-tree links without resulting in many false positives. Second, once B is properly chosen, increasing K systematically reduces the session state. In the same figure, when B is 8 bytes, increasing K from 3 to 6 reduces the session state from 40 to 12. Setting K to 8 eliminates the session state.

Finally, choosing good ranges for B and K depends on the topology size. For small topologies (35–75 routers), B can be set to 4–6 bytes, and K to 4–6 rounds. For medium topologies (75–150 routers), B can be set to 8–12 bytes, and K to 5–7 rounds. For large topologies (150+ routers), we can set B to 12–16 bytes, and K to 5–7 rounds. For a given label size, choosing B within these ranges results in better performance than smaller values with large K. For example, in the 158-router topology, setting B to 6 bytes and K to 8 results in maintaining state at 38 routers per session. However, setting B to 12 bytes and K to 4 reduces the session state to 4.

6 CONCLUSIONS

Recent large-scale live broadcast applications have introduced a renewed interest in designing efficient multicast services. Current multicast systems cannot implement general multicast trees, do not scale in terms of state maintained at routers, or may introduce loops and redundant traffic. We designed, implemented, and evaluated a new multicast forwarding system, called Helix, which is general (can be used with any topology), scalable (minimizes the size and update frequency of the state maintained at routers), and efficient (does not introduce loops or forward packets on links not belonging to the multicast tree). Helix has control plane and data plane components. In the control plane, Helix uses a Recursive Label Encoder (RLE) algorithm to encode multicast trees into labels. RLE uses probabilistic set membership data structures to encode tree links as labels. While these probabilistic data structures reduce the label size, they introduce false positives. To address this problem, RLE calculates a set of links that are false positives, and sends them to the routers that handle these links as a state. RLE minimizes this state by recursively encoding both tree and non-tree links across multiple rounds. This allows RLE to encode more information in the label. In the data plane, core routers execute a simple processing algorithm on each packet, which utilizes the information embedded in the labels by RLE. We analytically proved the correctness of Helix. In addition, we demonstrated the practicality of Helix by implementing it in a testbed and showing that it can provide line-rate throughput and it consumes a small fraction of the hardware resources. We have also compared Helix against the closest multicast forwarding systems in the literature using large-scale simulations with real ISP topologies. Our simulation results showed that Helix can achieve significant

performance gains over the other systems. For example, Helix reduces the state per session by up to 18X compared to the closest label-based multicast forwarding system. In addition, Helix eliminates the required state for many sessions using smaller labels across different topologies. Furthermore, compared to OpenFlow, Helix decreases the required state per session by up to 112X.

REFERENCES

[1] Burton H. Bloom. 1970. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM* 13, 7 (July 1970), 422–426.

[2] Bradley Cain, Steve E. Deering, Bill Fenner, Isidor Kouvelas, and Ajit Thyagarajan. 2002. Internet Group Management Protocol, Version 3. RFC 3376. (Oct. 2002). https://doi.org/10.17487/RFC3376

[3] X. Chen, M. Chen, B. Li, Y. Zhao, Y. Wu, and J. Li. 2013. Celerity: A Low-Delay Multi-Party Conferencing Solution. *IEEE Journal on Selected Areas in Communications* 31, 9 (September 2013), 155–164.

[4] S. H. Chiang, J. J. Kuo, S. H. Shen, D. N. Yang, and W. T. Chen. 2018. Online Multicast Traffic Engineering for Software-Defined Networks. In *Proc. of IEEE INFOCOM'18*. Honolulu, HI.

[5] T. W. Cho, M. Rabinovich, K. K. Ramakrishnan, D. Srivastava, and Y. Zhang. 2009. Enabling Content Dissemination Using Efficient and Scalable Multicast. In *Proc. of IEEE INFOCOM'09*. Rio de Janeiro, Brazil, 1980–1988.

[6] Cisco 2018. Cisco Visual Networking Index: Forecast and Trends, 2017–2022. https://bit.ly/2TYstY8. (November 2018). [Online; accessed January 2019].

[7] Cisco 2018. Multicast Command Reference for Cisco ASR 9000 Series Routers. https://bit.ly/2QAiLwk. (November 2018). [Online; accessed January 2019].

[8] Stephen E. Deering and David R. Cheriton. 1990. Multicast Routing in Datagram Internetworks and Extended LANs. *ACM Transactions on Computer Systems* 8, 2 (May 1990), 85–110.

[9] C. Diot, B. N. Levine, B. Lyles, H. Kassem, and D. Balensiefen. 2000. Deployment issues for the IP multicast service and architecture. *IEEE Network* 14, 1 (January 2000), 78–88.

[10] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. 2015. MoonGen: A Scriptable High-Speed Packet Generator. In *Proc. of ACM IMC'15*. Tokyo, Japan, 275–287.

[11] Bin Fan, Dave G. Andersen, Michael Kaminsky, and Michael D. Mitzenmacher. 2014. Cuckoo Filter: Practically Better Than Bloom. In *Proc. of ACM CoNEXT'14*. Sydney, Australia, 75–88.

[12] Bill Fenner, Mark J. Handley, Hugh Holbrook, Isidor Kouvelas, Rishabh Parekh, Zhaohui (Jeffrey) Zhang, and Lianshu Zheng. 2016. Protocol Independent Multicast - Sparse Mode (PIM-SM): Protocol Specification (Revised). RFC 7761. (March 2016). https://doi.org/10.17487/RFC7761

[13] V. Gopalakrishnan, B. Bhattacharjee, K. K. Ramakrishnan, R. Jana, and D. Srivastava. 2009. CPM: Adaptive Video-on-Demand with Cooperative Peer Assists and Multicast. In *Proc. of IEEE INFOCOM'09*. Rio de Janeiro, Brazil, 91–99.

[14] L. H. Huang, H. C. Hsu, S. H. Shen, D. N. Yang, and W. T. Chen. 2016. Multicast traffic engineering for software-defined networks. In *Proc. of IEEE INFOCOM'16*. San Francisco, CA.

[15] M. Huang, W. Liang, Z. Xu, W. Xu, S. Guo, and Y. Xu. 2016. Dynamic routing for network throughput maximization in software-defined networks. In *Proc. of IEEE INFOCOM'16*. San Francisco, CA.

[16] Xin Jin, Hongqiang Harry Liu, Rohan Gandhi, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Jennifer Rexford, and Roger Wattenhofer. 2014. Dynamic Scheduling of Network Updates. In *Proc. of ACM SIGCOMM'14*. Chicago, IL, 539–550.

[17] Petri Jokela, András Zahemszky, Christian Esteve Rothenberg, Somaya Arianfar, and Pekka Nikander. 2009. LIPSIN: Line Speed Publish/Subscribe Inter-networking. In *Proc. of ACM SIGCOMM'09*. Barcelona, Spain, 195–206.

[18] D. Kreutz, F. M. V. Ramos, P. E. VerÃŋssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig. 2015. Software-Defined Networking: A Comprehensive Survey. *Proc. IEEE* 103, 1 (January 2015), 14–76.

[19] D. Li, H. Cui, Y. Hu, Y. Xia, and X. Wang. 2011. Scalable data center multicast using multi-class Bloom Filter. In *Proc. of IEEE ICNP'11*. Vancouver, BC, Canada, 266–275.

[20] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. 2008. OpenFlow: Enabling Innovation in Campus Networks. *ACM SIGCOMM Comput. Commun. Rev.* 38, 2 (March 2008), 69–74.

[21] NetFPGA 2017. NetFPGA SUME Reference Learning Switch Lite. https://bit.ly/2UrUFlx. (July 2017). [Online; accessed January 2019].

[22] RadioTimes 2016. England v Wales Euro 2016 match sets new BBC online viewer record. https://bit.ly/2G7gViq. (June 2016). [Online; accessed January 2019].

[23] Aravindh Raman, Gareth Tyson, and Nishanth Sastry. 2018. Facebook (A)Live?: Are Live Social Broadcasts Really Broadcasts?. In *Proc. of the International World Wide Web Conference (WWW'18)*. Lyon, France.

[24] Muhammad Shahbaz, Lalith Suresh, Nick Feamster, Jen Rexford, Ori Rottenstreich, and Mukesh Hira. 2018. Elmo: Source-Routed Multicast for Cloud Services. (2018). http://arxiv.org/abs/1802.09815

[25] S. H. Shen, L. H. Huang, D. N. Yang, and W. T. Chen. 2015. Reliable multicast routing for software-defined networks. In *Proc. of IEEE INFOCOM'15*. Hong Kong, China, 181–189.

[26] The University of Adelaide 2012. The Internet Topology Zoo. http://www.topology-zoo.org/dataset.html. (July 2012). [Online; accessed January 2019].

[27] Bob Thomas, IJsbrand Wijnands, Ina Minei, and Kireeti Kompella. 2011. Label Distribution Protocol Extensions for Point-to-Multipoint and Multipoint-to-Multipoint Label Switched Paths. RFC 6388. (Nov. 2011). https://rfc-editor.org/rfc/rfc6388.txt

[28] IJsbrand Wijnands, Eric C. Rosen, Andrew Dolganow, Tony Przygienda, and Sam Aldrin. 2017. *Multicast using Bit Index Explicit Replication*. Internet-Draft. Internet Engineering Task Force. https://datatracker.ietf.org/doc/html/draft-ietf-bier-architecture-06

[29] Wired 2016. Zuckerberg really wants you to stream live video on Facebook. https://bit.ly/2v6uHqF. (April 2016). [Online; accessed January 2019].

[30] Xilinx 2014. Huawei and Xilinx unveil prototype 400GE Core Router at OFC 2104. FPGAs do the heavy lifting. https://bit.ly/2RYXMEf. (March 2014). [Online; accessed January 2019].

[31] Jiaqi Zheng, Bo Li, Chen Tian, Klaus-Tycho Foerster, Stefan Schmid, Guihai Chen, and Jie Wu. 2018. Scheduling Congestion-Free Updates of Multiple Flows with Chronicle in Timed SDNs. In *Proc. of IEEE ICDCS'18*. Vienna, Austria.

[32] N. Zilberman, Y. Audzevich, G. A. Covington, and A. W. Moore. 2014. NetFPGA SUME: Toward 100 Gbps as Research Commodity. *IEEE Micro* 34, 5 (Sept 2014), 32–41.

A PROOFS

A.1 Theorem 1: Completeness and minimality of calculated candidates

THEOREM 1. *The initial false positive candidate set is complete and minimal if it is composed of every link* $l = (u \rightarrow v) \notin \mathbb{L}$ *provided that* $u \in \mathbb{R}$ *and* $(v \rightarrow u) \notin \mathbb{L}$*, where* $\mathbb{R}$ *and* $\mathbb{L}$ *are the sets of nodes and links of the multicast tree* $\mathbb{T}$*, respectively.*

PROOF. Recall that the initial set of false positive candidates is the set used in the first round of encoding. We find this set by enumerating all possible combinations of every link $(u \rightarrow v)$ in the network. Then, we derive the sufficient conditions for a link to be a false positive candidate. To achieve this, we check for each link $(u \rightarrow v)$ whether u and

#	u	v	Condition	Decision	Example
1	$\notin \mathbb{R}$	$\notin \mathbb{R}$	None	Not included	$b \rightarrow c$
2	$\notin \mathbb{R}$	$\in \mathbb{R}$	None	Not included	$b \rightarrow 2$
3	$\in \mathbb{R}$	$\notin \mathbb{R}$	None	Included	$1 \rightarrow a$
4	$\in \mathbb{R}$	$\in \mathbb{R}$	$(u \rightarrow v) \notin \mathbb{L}$	Included	$8 \rightarrow 9$
5	$\in \mathbb{R}$	$\in \mathbb{R}$	$(v \rightarrow u) \in \mathbb{L}$	Not Included	$4 \rightarrow 2$

Table 2: Possible combinations of every link $(u \rightarrow v)$ based on its source u and destination v routers. The table shows the conditions to include links in the initial false positive candidate set. The examples are from Figure 11.

v belong to the tree nodes $\mathbb{R}$. Table 2 lists these combinations with examples from Figure 11.

Note that the outgoing links attached to a router u that does not belong to $\mathbb{R}$ cannot be false positives (rows 1 and 2 in Table 2). This is because a packet can only reach a router $u \notin \mathbb{R}$ from a router $p \in \mathbb{R}$ by passing through a false positive link l_{fp} attached to p. In this case, p should have already stored that the link l_{fp} is a false positive, and it would not forward packets on this link. Hence, a link $(u \rightarrow v)$ where $v \in$ neighbors(u) cannot be a false positive, and consequently is not a candidate.

Only links connected to a core router in $\mathbb{R}$ may be a candidate (rows 3, 4, and 5) as explained below:

(1) $u \in \mathbb{R}$ and $v \notin \mathbb{R}$ (row 3): When a packet reaches a router u, this router does not know if $(u \rightarrow v)$ is a non-tree link. This is because this link may collide with the tree links in the first round of encoding. Thus, the BaseEncoder needs to check whether this link collides with the tree links in H_1. Thus, $(u \rightarrow v)$ is a candidate.

(2) Both u and v belong to $\mathbb{R}$. There are two possibilities of links connecting these two routers:
   - (a) $(u \rightarrow v) \notin \mathbb{L}$ (row 4). This link can result in forwarding loops because it can be a false positive. Hence it is a candidate, or
   - (b) $(v \rightarrow u) \in \mathbb{L}$ (row 5). Upon receiving packets at router u from v (i.e., through $(v \rightarrow u)$), router u should not forward them back on the upstream link. Hence, $(u \rightarrow v)$ cannot be a candidate.

In Table 2, links that satisfy the conditions in rows 3–5 are complete because there is no other link combination not mentioned in the table. They are minimal because if we exclude any link from rows 3–5, this will result in incomplete set. □

A.2 Theorem 2: Classifying state sets to tree or non-tree links

THEOREM 2. *In the Recursive Label Encoder (RLE) algorithm, if round k is even, the links encoded in the output state S_k are tree links. Otherwise, these links do not belong to the multicast tree.*

PROOF. We first state two properties about any three consecutive sets. Given S_{k-2}, S_{k-1}, and S_k, the following two properties hold:

$$S_k \subseteq S_{k-2}, \text{ and} \tag{3}$$

$$S_k \cap S_{k-1} = \phi. \tag{4}$$

This means that, for round k, the state S_k is a subset of the round candidates S_{k-2}. In addition, the intersection of S_k and the round encoded links S_{k-1} is an empty set. The first property holds because, in RLE, the false positive candidate set of a round k is the set S_{k-2}. RLE keeps some links from S_{k-2} and removes others. Then it produces S_k as a resulting state. The second property holds because S_{k-1} is the input links to be encoded for round k, and S_k is the resulting state from the round candidates. In the first round of RLE, links to be encoded and candidates are non-overlapping sets because they are subsets of $\mathbb{L}$ and the initial false positive candidate set, respectively. Since RLE controls the inputs to BaseEncoder by swapping links and candidates, these two sets do not intersect.

We next define S_k in terms of the previous states using this recurrence:

Base case:

$k = 0,\ S_0 = \mathbb{L}$ are tree links, and

$k = 1,\ S_1$ are non-tree links.

For $k \geq 2$:

$\forall l \in S_k,\ l \in S_{k-2}$ and $l \notin S_{k-1}$.

For the base case, links in S_0 are tree links by definition. The calculated state S_1 are the non-tree links that collide with S_0 in H_1. The recurrence case (i.e., $k \geq 2$) holds because any link in S_k belongs to the set S_{k-2} (Eq. 3), and does not belong to the input set S_{k-1} (Eq. 4).

By substituting a given k in this recurrence, we can always check if the set S_k is a subset of the tree links S_0 or the non-tree links S_1. That is, if k is even, $l \in S_k \subseteq \cdots \subseteq S_0$. Otherwise, $l \in S_k \subseteq \cdots \subseteq S_1$. In addition, a state S_k cannot include tree and non-tree links because of Eq. 4. □

A.3 Theorem 3: Correctness of the packet processing algorithm

THEOREM 3. *The packet processing algorithm, Algorithm 2, duplicates packets on and only on links that belong to the multicast tree.*

Proof. The idea of the proof is to use the maintained state and exploit the properties of the RLE and used filters to determine if a link belongs to the multicast tree. To achieve this, we list all cases where a link belongs (or does not belong) to label components.

Given label components, there are three possibilities for a link l attached to a router:

1. this link does not belong to at least one label component H_k but belongs to previous ones $H_1, \ldots, H_{k-1}$,
2. it does not belong to any label component, or
3. it belongs to all label components $H_1, \ldots, H_K$.

In the first case, our proof relies on two properties. First, the candidates of round k in RLE is S_{k-2} as stated in Eq. 2. Second, filters in Helix do not produce false negatives. Let H_k be the *first* label where l does not belong to. The router knows that link l: (1) does not belong to the links in $S_{k-1} \cup S_k$ (i.e., links that exist in H_k), and (2) belongs to the candidates S_{k-2}. Otherwise, H_k would not be the first label where l does not belong to. This is because RLE removes links from the candidates only if they do not collide in H_k with the links to be encoded. If k is even, the router knows that the candidate set S_{k-2} of this round is a subset of the tree links (Theorem 2). This means that $l.id \in S_{k-2} \subseteq \cdots \subseteq S_0$. Thus, the packet is duplicated on l. If k is odd, the opposite result holds, where $l.id \in S_{k-2}$ and $S_{k-2} \subseteq \cdots \subseteq S_1$. Thus, the packet is not duplicated.

In the second case, if a link does not belong to any label component, this link does not belong to the first label as well. Thus, it is easy to see that this case is a special instance of the first case when $k = 1$. Thus, the link belongs to S_{-1} which are non-tree links.

The third case is when l belongs to all labels $H_1, \ldots, H_K$. This means that RLE could not remove this link from all previous states $S_1, \ldots, S_{K-1}$. And since the filters may result in false positives, it is not conclusive whether l belongs to the tree links. Thus, we need to look at the content of the maintained state S_K. In this case, we derive the conditions of duplicating packets by using the results of Theorem 2 to decide whether S_K contains tree or non-tree links as follows:

1. K is odd. The stored state S_K is a set of non-tree links (Theorem 2). If $l.id \notin S_K$, it means that $l.id$ does not belong to the non-tree links, the packet is duplicated on l. Otherwise, the packet is not duplicated.
2. K is even. The stored state S_K is a set of tree links. If $l.id \in S_K$, then $l.id \in \mathbb{L}$ and the router duplicates a packet on l. Otherwise, l is a non-tree link, and the packet is not duplicated on it.

□

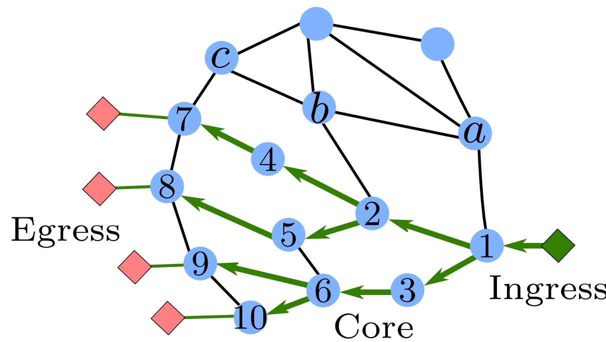
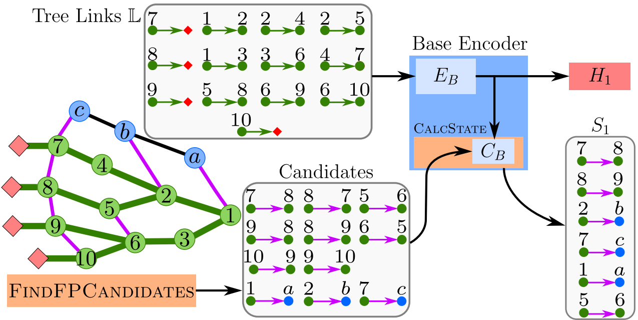
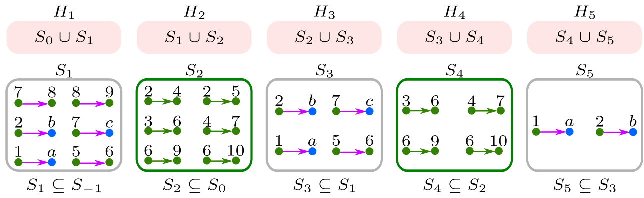


Figure 10: The topology and multicast tree used in the example.



(a) The first round of encoding using BaseEncoder.



(b) Resulting labels and states of running RLE with $K = 5$.

Figure 11: Illustrative example of the RLE algorithm.

B ILLUSTRATIVE EXAMPLE

We present a simple example in Figure 11 to illustrate the various components of Helix. This example uses the topology and multicast tree in Figure 2 (repeated in Figure 10 for quick reference). Figure 11a shows a multicast tree that spans the routers numbered 1 to 10. Routers labeled a, b, c do not belong to the tree. The figure illustrates the first round of encoding based on the tree links and the initial false positive candidate set. Figure 11b shows the labels and states after applying RLE on the same tree when K is 5. For simplicity, we show the content of a label H_i as the union of two sets: links encoded in that round S_{i-1}, and false positive links S_i that collide with these encoded links.

In the first round, RLE encodes the tree links S_0 into H_1, and produces state S_1. Note that H_1 encodes all links in S_0 and the links from S_{-1} that collide with S_0 in H_1. Thus, S_1 is the state of the first round. The second round takes S_1 as the links to be encoded, and S_0 as the false positive candidates. This round produces label H_2 that includes S_1 and S_2. Note that in this round, the tree links $(1 \rightarrow 2)$ and $(1 \rightarrow 3)$ (among other tree links) do not collide with any link in S_1. Hence, RLE decides that these links are not needed in S_2, because H_2 does not include them. The final round takes S_4 as links to be encoded, and S_3 as false positive candidates. Both non-tree links $(7 \rightarrow c)$ and $(5 \rightarrow 6)$ do not collide with any link in H_5, and thus they are removed from S_5.

Notice that the resulting states at even rounds S_2 and S_4 are tree links, and the resulting states at odd rounds S_1, S_3 and S_5 are non-tree links.

Finally, we illustrate the decisions made by the packet processing algorithm. First, we show the decisions for some non-tree links. For the link $(7 \rightarrow 8)$, the algorithm finds that it exists in both H_1 and H_2, but it does not belong to H_3 (Figure 11b). Thus, the algorithm does not duplicate packets on this link. For the link $(1 \rightarrow a)$, it exists in all labels. In this case, K is odd and the link exists in S_5. As a result, the algorithm does not duplicate the packet on this link. Second, we describe the decisions for some tree links. For the link $(1 \rightarrow 2)$, the algorithm finds that it exists in H_1 but not in H_2. Thus, the algorithm duplicates packets on this link. In the case of link $(3 \rightarrow 6)$, the algorithm finds that it exists in all labels. Since K is odd and the link does not exist in S_5, the algorithm duplicates packets on this link.