Detecting Routing Loops in the Data Plane

Jan Kučera
jan.kucera@cesnet.cz
CESNET, a.l.e.

In collaboration with:
Ran Ben Basat (Harvard University),
Mário Kuka (CESNET), Gianni Antichí (Queen Mary University of London),
Minlan Yu (Harvard University) and Michael Mitzenmacher (Harvard University)
Routing loops

- harm network operation, lead to losses, which
  - increase the tail latency [1]
  - are interpreted as a congestion (TCP), cause throughput reduction [2]

- significantly increase the overall traffic [3]
- affect other traffic on shared links in terms of delay and jitter [1]

- real-time detection is essential for the network performance

---

3 categories, classification of the past approaches: (based on handling the information needed for detecting loops)

1) keep flow state at switches
2) mirror information at switches
3) store information on packets

3 perspectives to evaluate them:

1) switch overhead,
2) network overhead,
3) real-time detection
1) On-switch state

- aggregate flow information at switches
- periodically export them to a collector
- keeping state, e.g., up to 100K active flows
- e.g., FlowRadar [1], Hash IP Traceback [2]

- switch overhead: high
- network overhead: low
- real-time detection: ❌

---

2) Header Mirroring

- duplicate the traffic headers
- sending it to an analyzer

- e.g., NetSight [1], Everflow [2], Trajectory Sampling [3]

- switch overhead: \textit{low}
- network overhead: \textit{high}
- real-time detection: \xmark

---

3) Full Path Encoding on Packets

- each switch records its ID in the incoming packet
- if its ID is already stored, a loop is detected
- per packet overhead cost grows linearly
- e.g., INT [1], PathDump [2], Tiny Program Packets [3]

- switch overhead: low
- network overhead: high
- real-time detection: ✓

---

Design space

- **All the existing solutions**
  - are either **unable to detect loops in real time** or
  - have a **packet overhead** that is **linear in the number of hops**

- **Can we design an algorithm that detects routing loops**
  - in the **data plane**, 
  - at **real time**, 
  - while keeping **low switch and network overheads**?

- **INT stores all switches, storing Bloom Filter** saves the bandwidth
  - reduces the **overhead**, but introduces **false positives**
  - but still encodes IDs of **all the visited switches**, is it necessary?
Designing Unroller

- **No need to store all switches on the loop!**

- **Unroller**
  - stores *only some switch* on the loop
  - the *minimum switch ID* that it has seen
  - reports the loop when we see *repeated switch ID*
  - guaranteed detection after *two iterations* thought the loop

- **A path of switches before reaching the loop!**

- **We occasionally reset the stored ID**
  - gradually increasing the resetting interval
  - reset after each *phase* -- $b^i$ hops for $i=1, 2, 3, ...$
  - e.g., for $b=2$ phases consist of 2, 4, 8, 16, ... hops
Unroller | Detection time analysis

- **B** ... the number of **hops before the loop**
- **L** ... the number of **switches in the loop**
- at least **X=B+L hops** required for any algorithm

**We showed that** (the proof presented in the paper)
- after **no more than 4.67X hops** the packet reaches a switch that reports the loop
- **lower bound** $\approx 3.73X$ (the minimal number of hops required by **any algorithm that stores a single ID**)  
- **not far from optimal** for deterministic algorithms
Unroller | Reducing per-packet overhead

- **hashing switch ID into** \( z \) **bits** and storing only hash instead
  - that introduces also **false positives** (FPs)

- **counting technique** to **reduce FPs**
  - small counter to track the number of times the switch matches the stored ID
  - once the counter reaches the threshold (Th), we report the loop

---

**FPs for compressed IDs** \((b=4, B=20, L=0)\)

**Detection time using counting technique** \((b=4, B=5)\)
Implementation

- Unroller implemented using P4 and compiled into BMv2
  - number of visited hops, minimum switch ID seen,
  - value of Th counter, encoded on packets
  - other b, z, Th values preset

- HW resources quantified
  - compiled for three FPGA-base NICs
  - two Xilinx FPGAs, and one Intel FPGA

- created Python simulator for evaluation on real topologies

Open sourced and available on GitHub: https://github.com/kucejan/unroller
Evaluation

■ **Sensitivity analysis**
how different parameters (b, B, z, ...) affect Unroller performance (presented above)

■ **Comparing to state-of-the-art solutions**
  ■ Comparison of false positives between Unroller and Bloom filter
  ■ Comparison of per-packet overhead on real topologies

<table>
<thead>
<tr>
<th>Topology</th>
<th># of Nodes</th>
<th>Diameter</th>
<th>Bloom filter Overhead</th>
<th>Unroller Avg Time</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stanford</td>
<td>16</td>
<td>2</td>
<td>171b</td>
<td>1.74X</td>
<td>25b</td>
</tr>
<tr>
<td>BellSouth</td>
<td>51</td>
<td>7</td>
<td>189b</td>
<td>1.56X</td>
<td>25b</td>
</tr>
<tr>
<td>GEANT</td>
<td>40</td>
<td>8</td>
<td>608b</td>
<td>2.13X</td>
<td>27b</td>
</tr>
<tr>
<td>ATT-NA</td>
<td>25</td>
<td>5</td>
<td>608b</td>
<td>2.13X</td>
<td>27b</td>
</tr>
<tr>
<td>UsCarrier</td>
<td>158</td>
<td>35</td>
<td>2466b</td>
<td>2.47X</td>
<td>28b</td>
</tr>
<tr>
<td>FatTree4</td>
<td>20</td>
<td>4</td>
<td>414b</td>
<td>1.73X</td>
<td>28b</td>
</tr>
</tbody>
</table>

* over 3M runs so that there are no FPs

6x-100x
Conclusion

- **Detecting Routing Loops in the Data Plane**
- **Unroller** = a lightweight **loop detection solution**
  - easily deployable on programmable switches
  - encodes only a small subset of the switches along the path
  - using a minimal **bit-overhead** on packets
  - does not store state on switches
  - identifies loops in real time
  - without a remote analysis node
  - detection in a **bounded number of hops**

Questions ?

jan.kucera@cesnet.cz
storing multiple identifiers on packets (c·H in total)

- $H \in \mathbb{N}$, the number of hash functions (parallel runs of the algorithm)
  - multiple switches can have “minimum IDs” with respect to some hash function

- $c \in \mathbb{N}$, the number of phase chunks (partitioning each phase its chunks)
  - each of the c identifier tracks the minimum only on a $1/c$ fraction of the phase

Detection time for different $c$ (number of chunks) and $H$ (number of hashes) configurations ($b=4$, $B=5$, $L=20$, $z=32$)