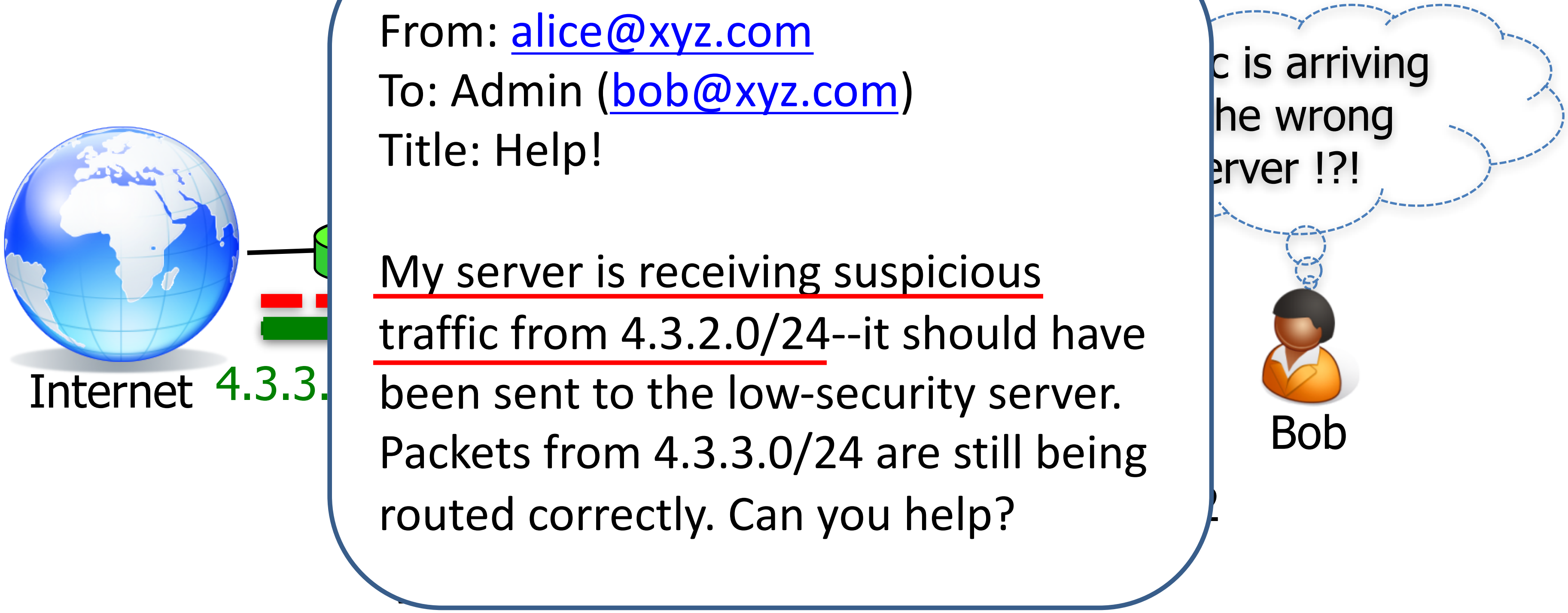
Motivation: Root cause analysis

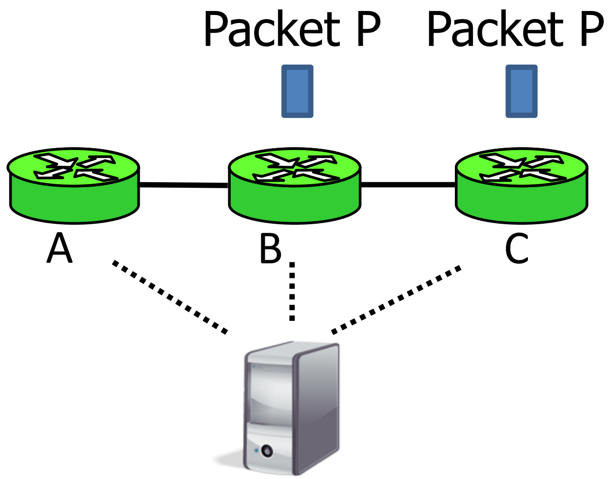


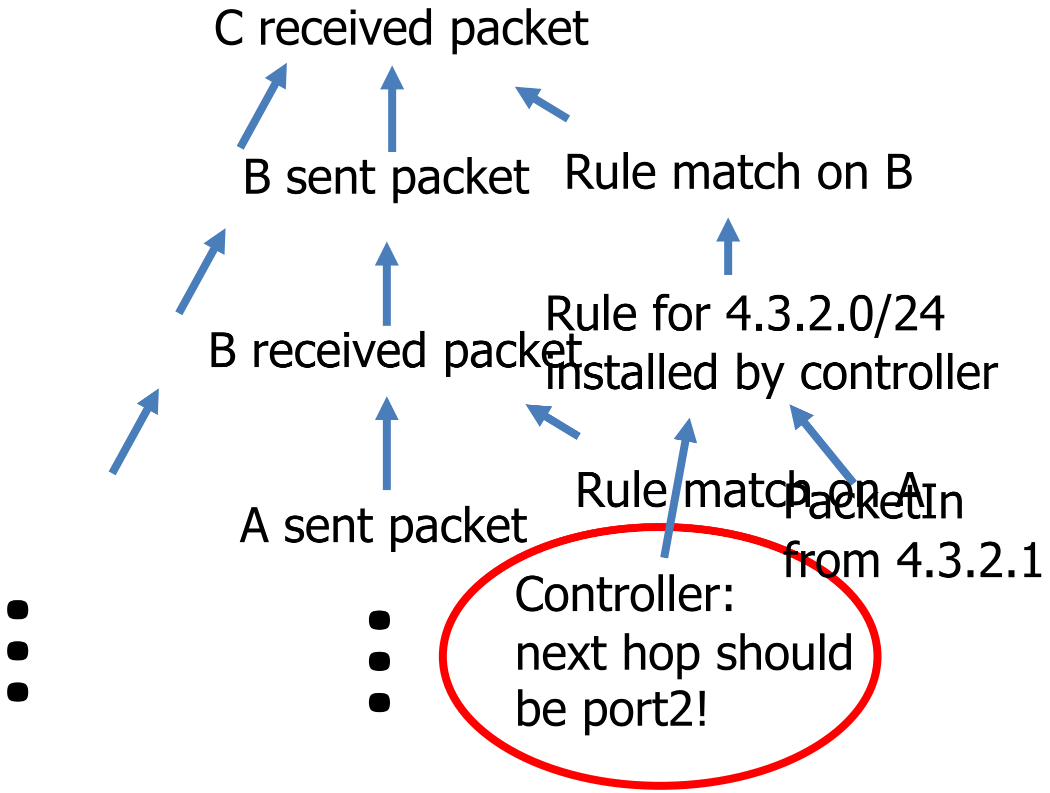
From: alice@xyz.com
To: Admin (bob@xyz.com)
Title: Help!

My server is receiving suspicious traffic from 4.3.2.0/24--it should have been sent to the low-security server. Packets from 4.3.3.0/24 are still being routed correctly. Can you help?

- Networks can (and frequently do!) have bugs
- We need a good debugger!

Debugging networks with provenance





C received packet

B sent packet

Rule match on B

Rule for 4.3.2.0/24 installed by controller

B received packet

Rule match on A

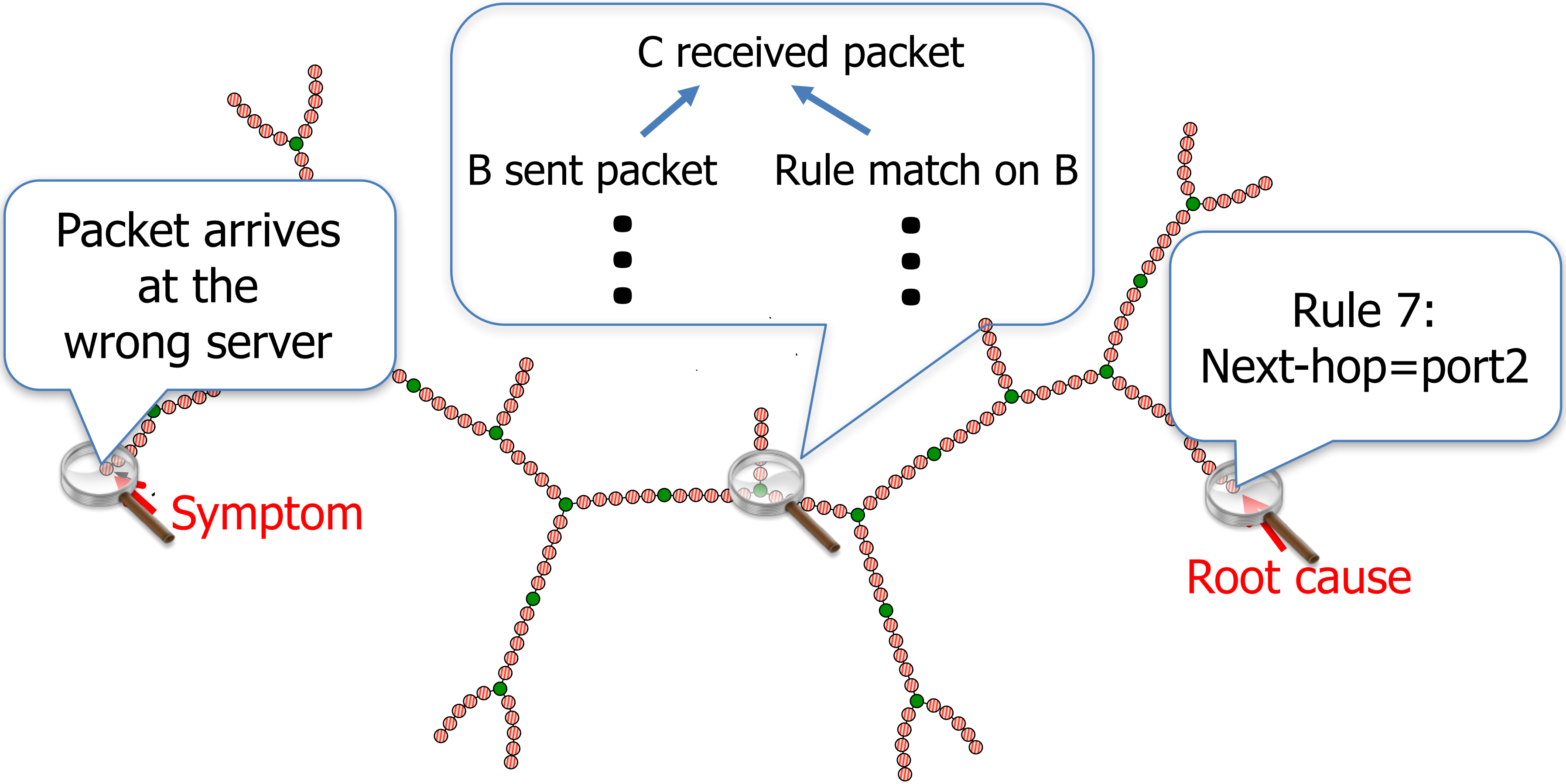
PacketIn from 4.3.2.1

A sent packet

Controller: next hop should be port2!

- Existing debuggers tell us what happened
  - Example: NetSight [NSDI'14]
- Provenance offers a richer explanation
  - Example: Y! [SIGCOMM'14]

Problem: The explanation can be too big!



Packet arrives at the wrong server

C received packet

B sent packet

Rule match on B

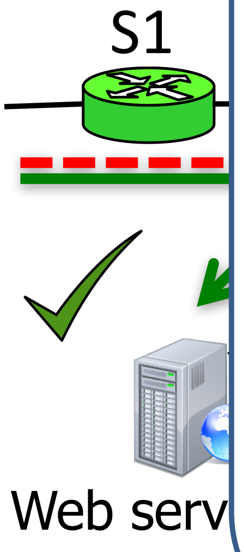
⋮ ⋮

Rule 7:
Next-hop=port2

Symptom

Root cause

What can we do?



From: alice@xyz.com
To: Admin (bob@xyz.com)
Title: Help!

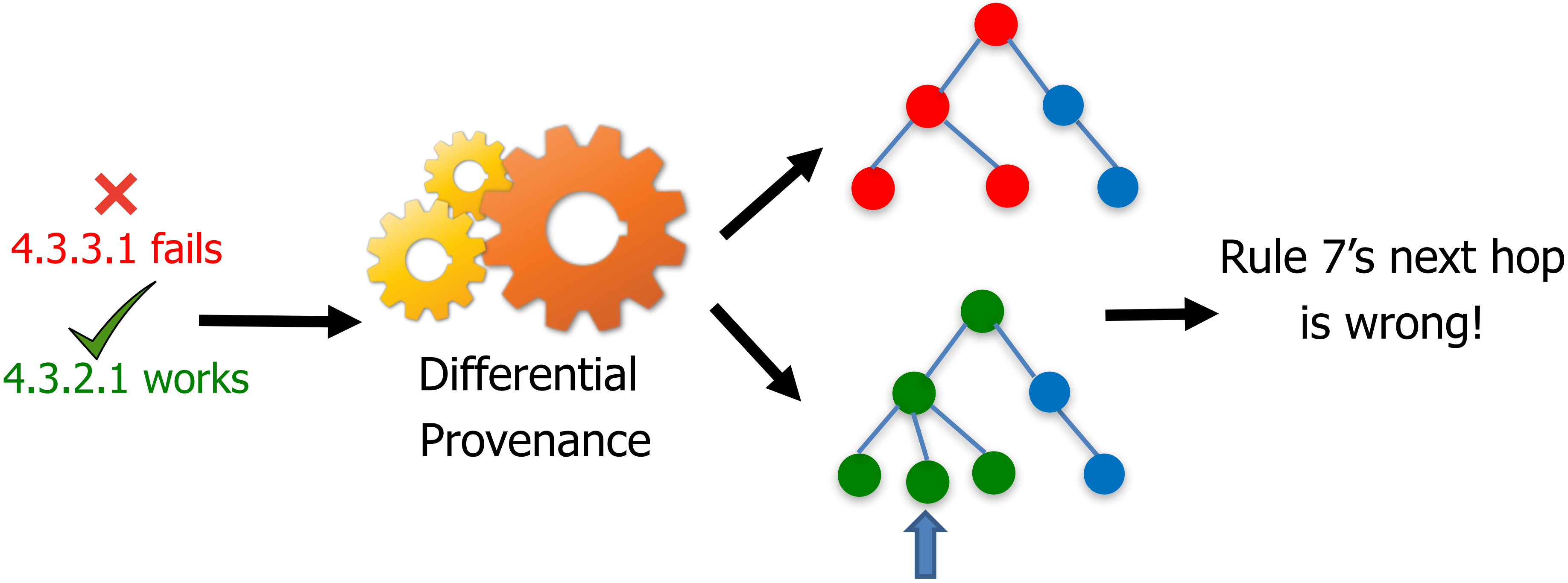
My server is receiving suspicious traffic from 4.3.2.0/24--it should have been sent to the low-security server. Packets from 4.3.3.0/24 are still being routed correctly. Can you help?

Working reference!

Outages mailing list
Sept.—Dec. 2014:
66% have references!

Idea: Reason about the differences bet n
and the reference

Differential provenance



✗ 4.3.3.1 fails

✓ 4.3.2.1 works

Differential Provenance

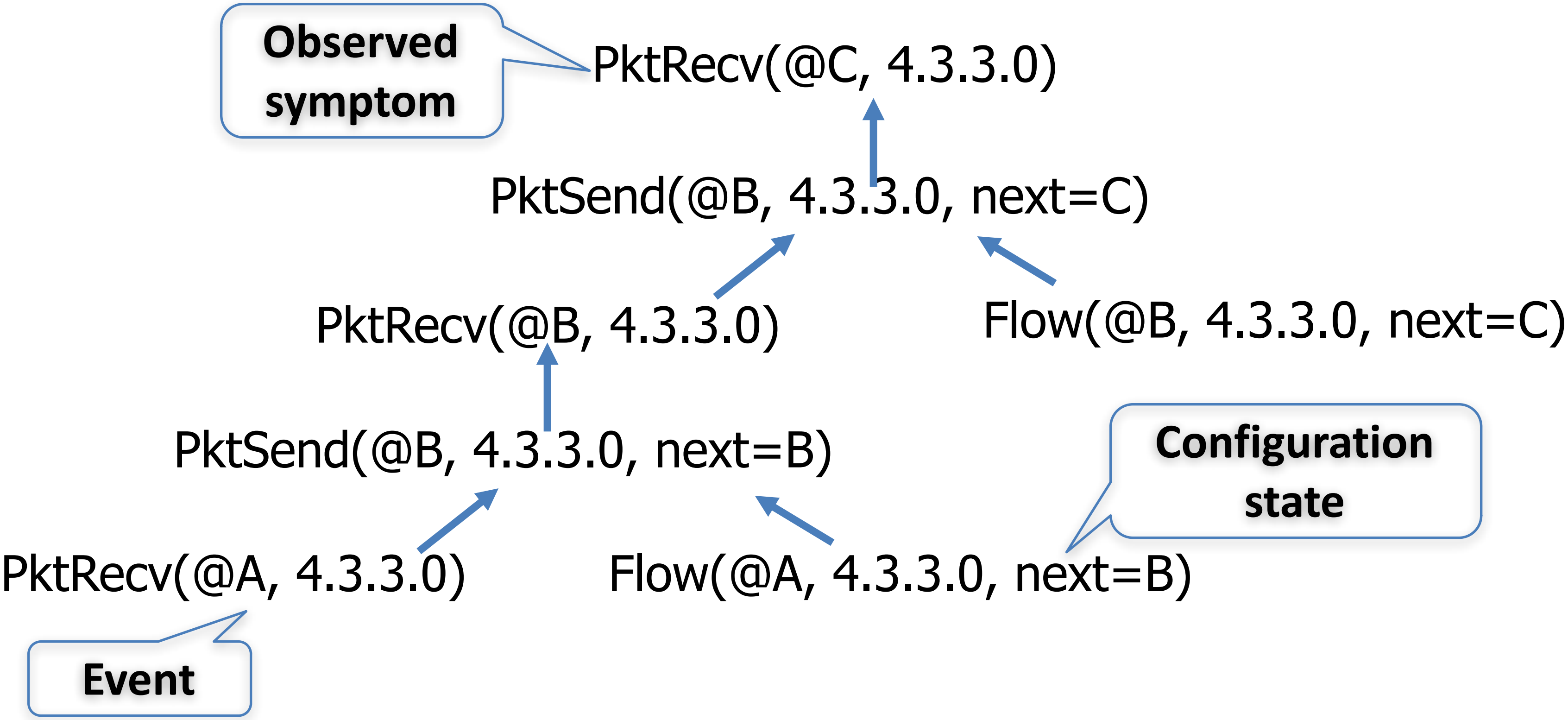
Rule 7’s next hop is wrong!

- Input: a bad symptom and a good reference
- Debugger reasons about the differences
- Output: root cause

Outline

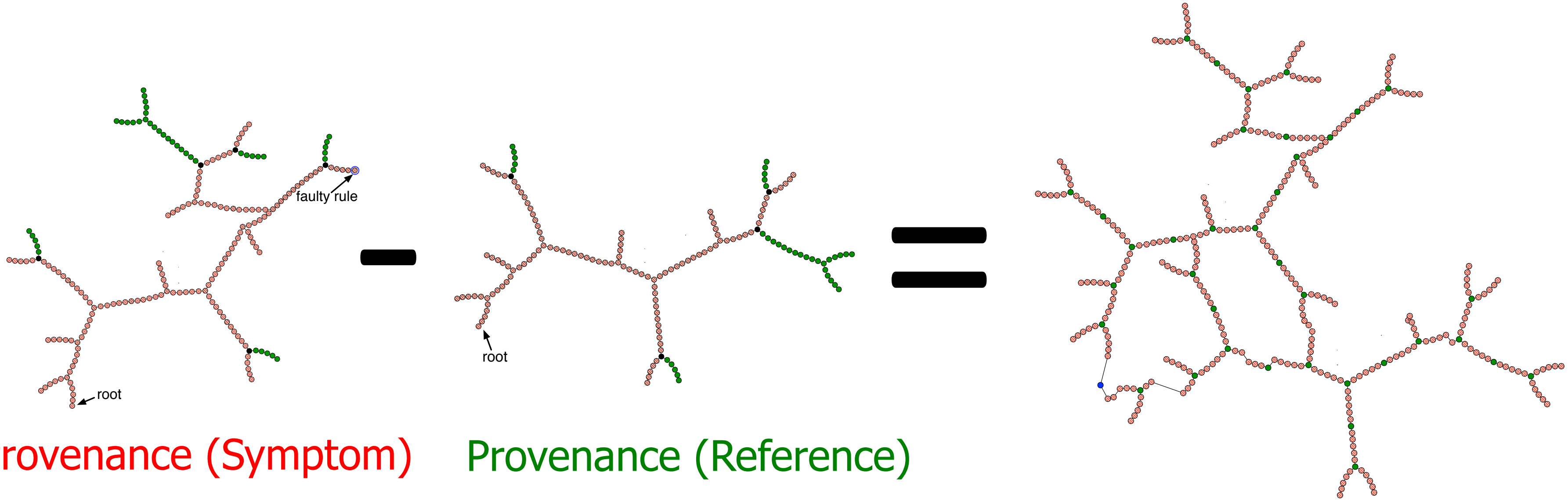
- ✓ Motivation: Root cause analysis
- ⇨ Differential provenance
  - Background: Provenance
  - Strawman solution
  - Algorithm
- Evaluation
  - Prototype implementation
  - Usability
  - Query processing speed
  - Complex network diagnostics
- Conclusion

Background: Provenance


Observed symptom
PktRecv(@C, 4.3.3.0)
PktSend(@B, 4.3.3.0, next=C)
PktRecv(@B, 4.3.3.0)
Flow(@B, 4.3.3.0, next=C)
PktSend(@B, 4.3.3.0, next=B)
Configuration state
PktRecv(@A, 4.3.3.0)
Flow(@A, 4.3.3.0, next=B)
Event

- Provenance tracks causal connections between network events and state [ExSPAN-SIGMOD'10]
  - Provenance graph: Vertexes → event/state. Edge → causality
  - Provenance tree: Recursive explanation of an event/state

Strawman solution

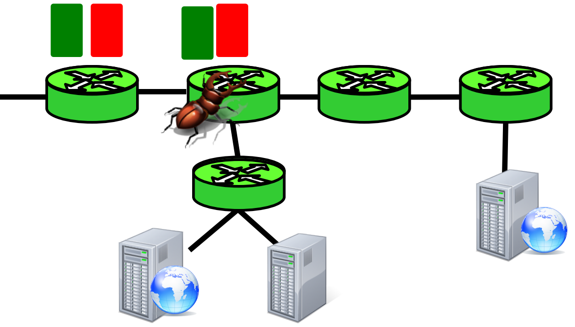


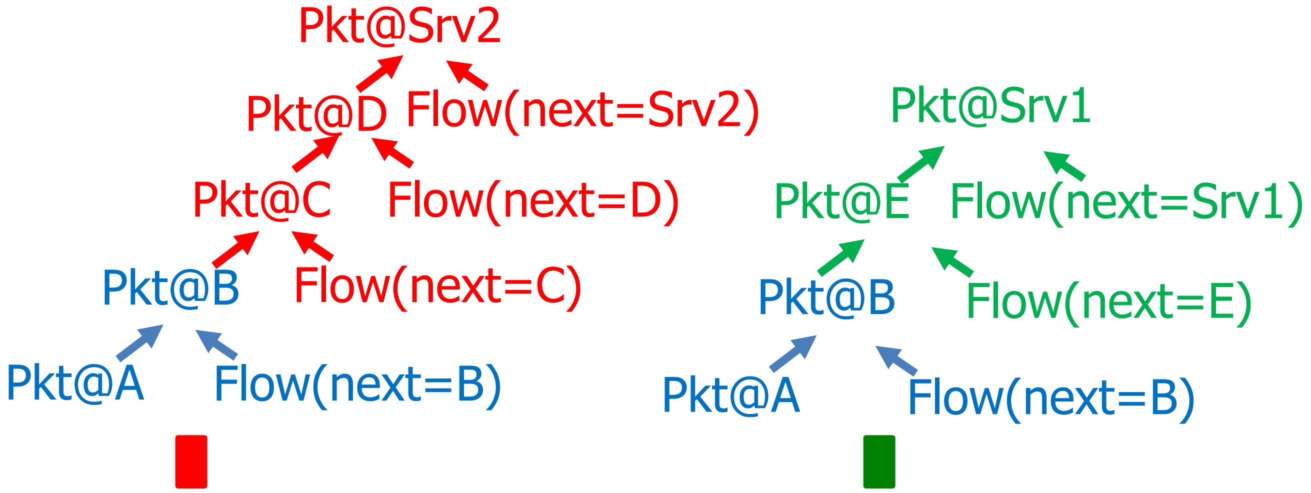
Provenance (Symptom)

Provenance (Reference)

- Strawman solution: Find vertexes that are different in the two trees
- Problem: The diff can be larger than the individual trees!

Why does the strawman solution not work?





Pkt@Srv2

Pkt@D Flow(next=Srv2)

Pkt@C Flow(next=D)

Pkt@B Flow(next=C)

Pkt@A Flow(next=B)

Pkt@Srv1

Pkt@E Flow(next=Srv1)

Pkt@B Flow(next=E)

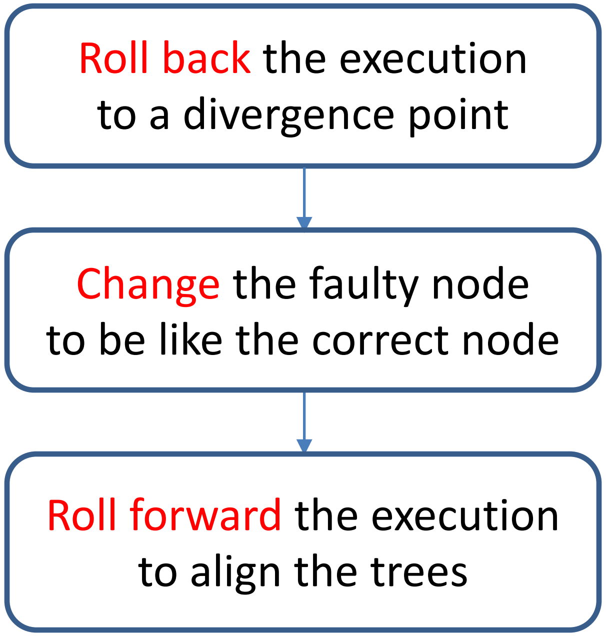
Pkt@A Flow(next=B)

- Observation: The diff can be larger than the individual trees
- Reason: "Butterfly effect"
  - A small initial difference can lead to drastically different events later on

Outline

- ✓ Motivation: Root cause analysis
- ✓ Key insight
- ✓ Differential provenance
  - ✓ Background: Provenance
  - ✓ Strawman solution
  - ⇨ Algorithm
- Evaluation
  - Prototype implementation
  - Usability
  - Query processing speed
  - Complex network diagnostics
- Conclusion

Algorithm: Refinement #1



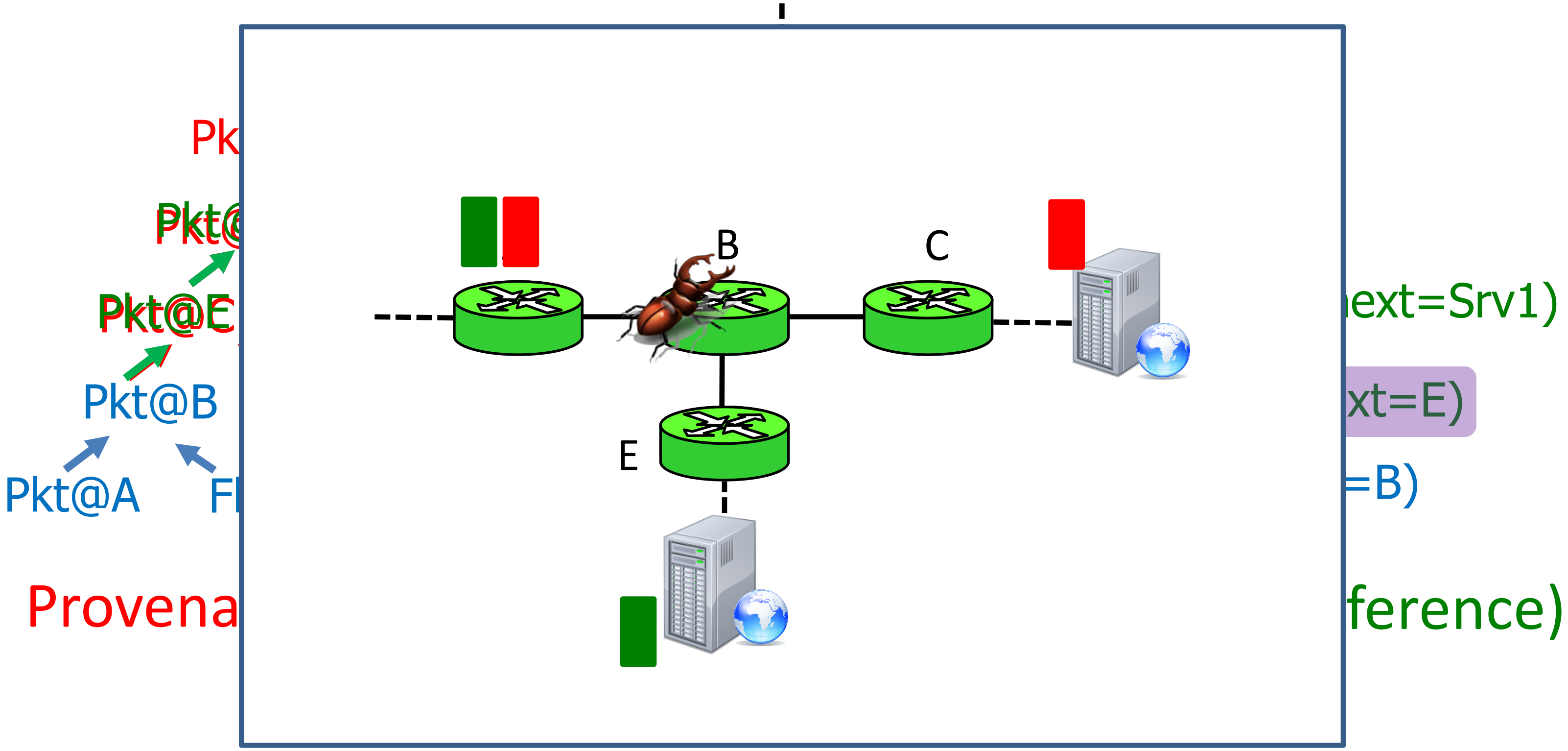
Roll back the execution to a divergence point

Change the faulty node to be like the correct node

Roll forward the execution to align the trees

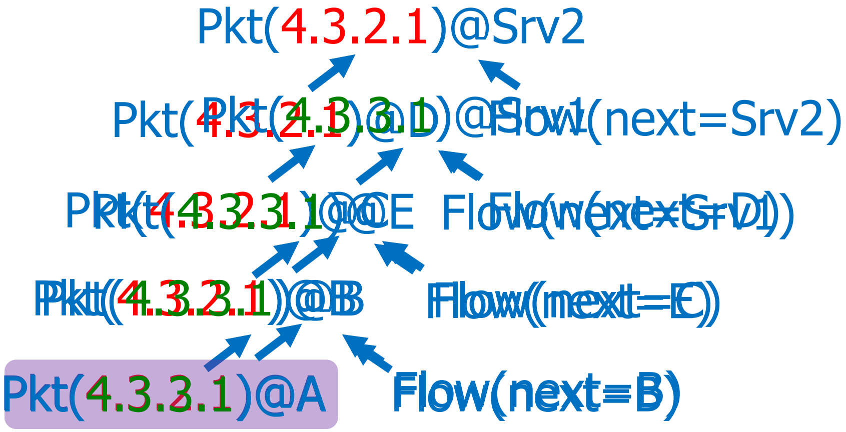
- This approach finds only the (small) initial differences
  - The (potentially large) consequences are ignored

Algorithm: Refinement #1 (Cont'd)



- Approach: Roll back the execution, change the first faulty node, and roll forward again to align the trees

How to preserve crucial differences?



Pkt(4.3.2.1)@Srv2

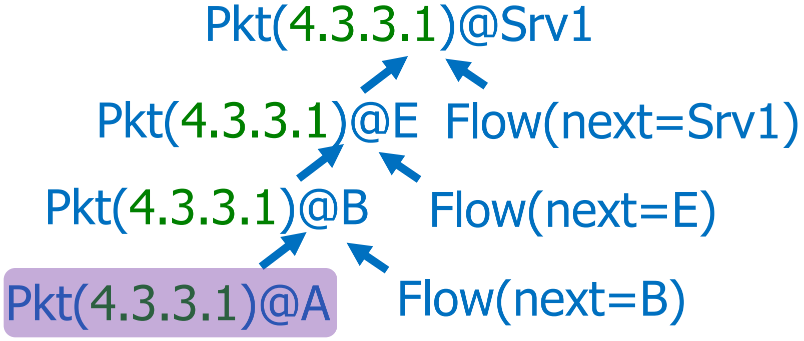
Pkt(4.3.2.1)@Srv1 Pkt(4.3.3.1)@D Flow(next=Srv2)

Pkt(4.3.2.1)@C Pkt(4.3.3.1)@E Flow(next=Srv1) Flow(next=D)

Pkt(4.3.2.1)@B Pkt(4.3.3.1)@B Flow(next=C) Flow(next=E)

Pkt(4.3.2.1)@A Pkt(4.3.3.1)@A Flow(next=B)

Provenance (symptom)



Pkt(4.3.3.1)@Srv1

Pkt(4.3.3.1)@E Flow(next=Srv1)

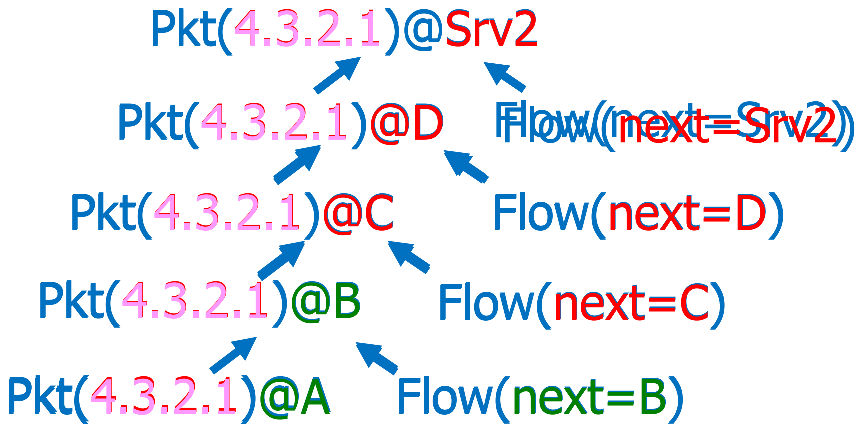
Pkt(4.3.3.1)@B Flow(next=E)

Pkt(4.3.3.1)@A Flow(next=B)

Provenance (reference)

- Problem: There are differences that we need to preserve
  - Example: The packets whose provenance we are looking at

Solution: Establish equivalence



Pkt(4.3.2.1)@Srv2

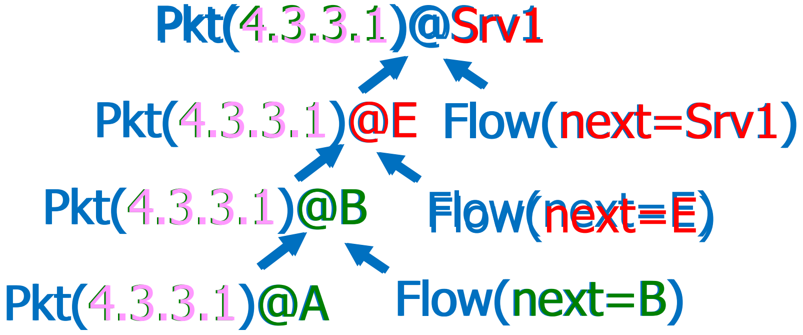
Pkt(4.3.2.1)@D Flow(next=Srv2)

Pkt(4.3.2.1)@C Flow(next=D)

Pkt(4.3.2.1)@B Flow(next=C)

Pkt(4.3.2.1)@A Flow(next=B)

Provenance (symptom)



Pkt(4.3.3.1)@Srv1

Pkt(4.3.3.1)@E Flow(next=Srv1)

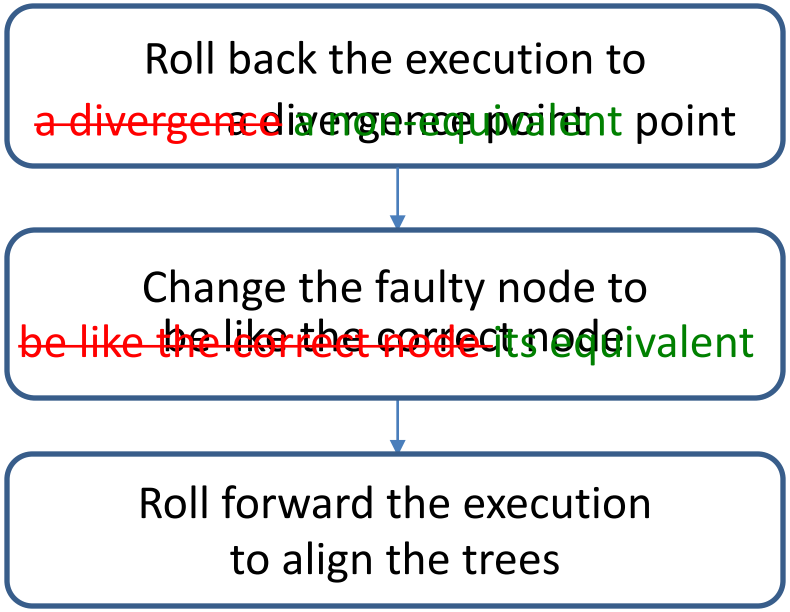
Pkt(4.3.3.1)@B Flow(next=E)

Pkt(4.3.3.1)@A Flow(next=B)

Provenance (reference)

- Establish an equivalence relation between the trees
  - Example: IP addresses 4.3.2.1 and 4.3.3.1
  - Values on the trees can be identical, equivalent, or different
- Goal: Make the trees equivalent, not necessarily identical!

Algorithm: Refinement #2



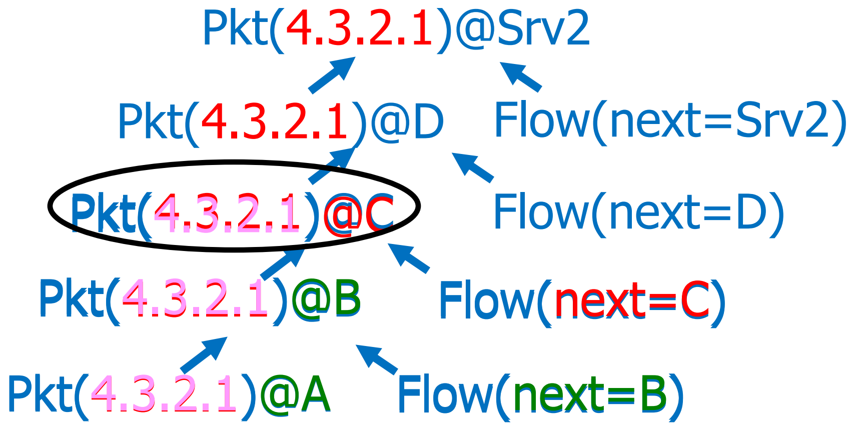
Roll back the execution to
a divergence point
~~a divergence~~ a non-equivalent point

Change the faulty node to
be like the correct node
~~be like the correct node~~ its equivalent

Roll forward the execution
to align the trees

- Benefit: Preserves the crucial differences between the trees

Establishing and propagating equivalence



Pkt(4.3.2.1)@Srv2

Pkt(4.3.2.1)@D

Flow(next=Srv2)

Pkt(4.3.2.1)@C

Flow(next=D)

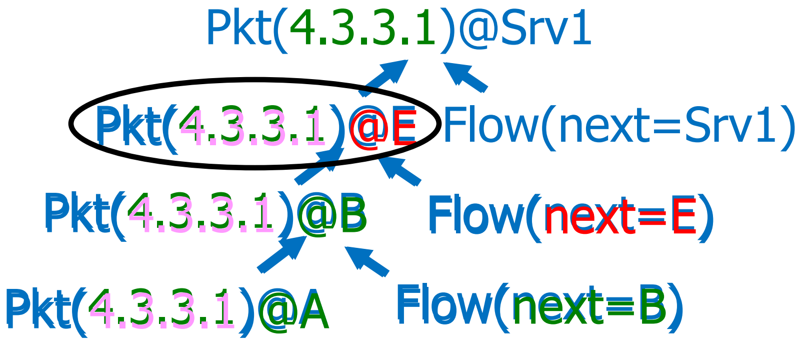
Pkt(4.3.2.1)@B

Flow(next=C)

Pkt(4.3.2.1)@A

Flow(next=B)

Bad provenance



Pkt(4.3.3.1)@Srv1

Pkt(4.3.3.1)@E

Flow(next=Srv1)

Pkt(4.3.3.1)@B

Flow(next=E)

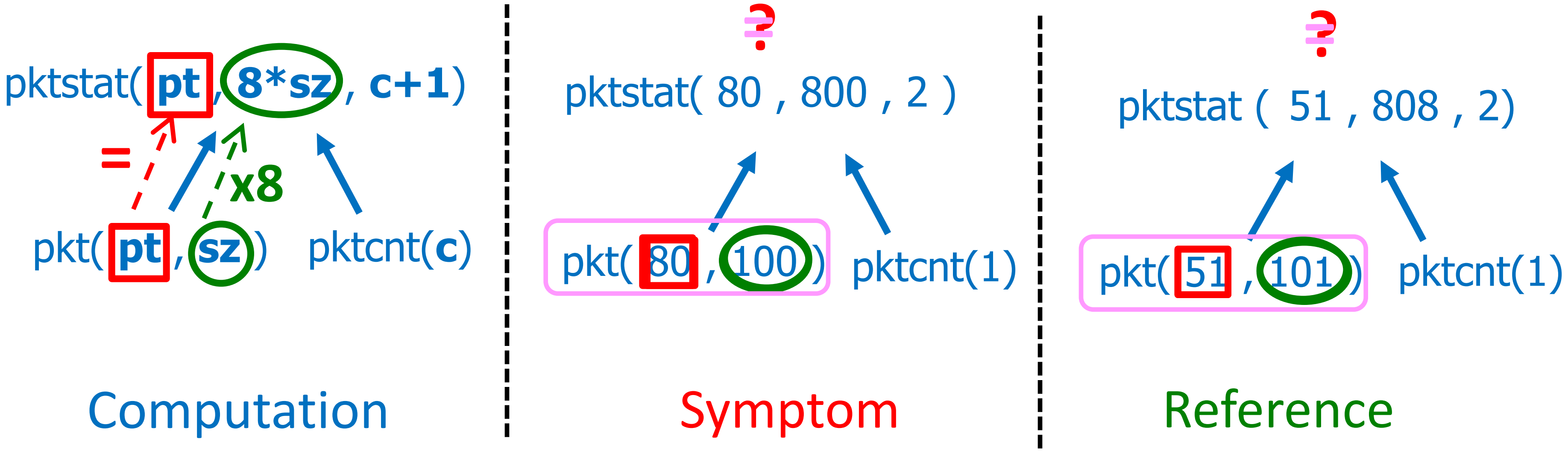
Pkt(4.3.3.1)@A

Flow(next=B)

Reference provenance

- Start with an initial equivalence relation between the packets
  - Establish a mapping between packet fields that are different
- Keep track of the mapping while going up the tree
  - Stop at the first non-equivalent(!) node
  - More general approach: taint analysis

Propagating equivalence with taints



pktstat(pt , 8*sz , c+1)

pkt(pt , sz) pktcnt(c)

Computation

pktstat(80 , 800 , 2)

pkt(80 , 100) pktcnt(1)

Symptom

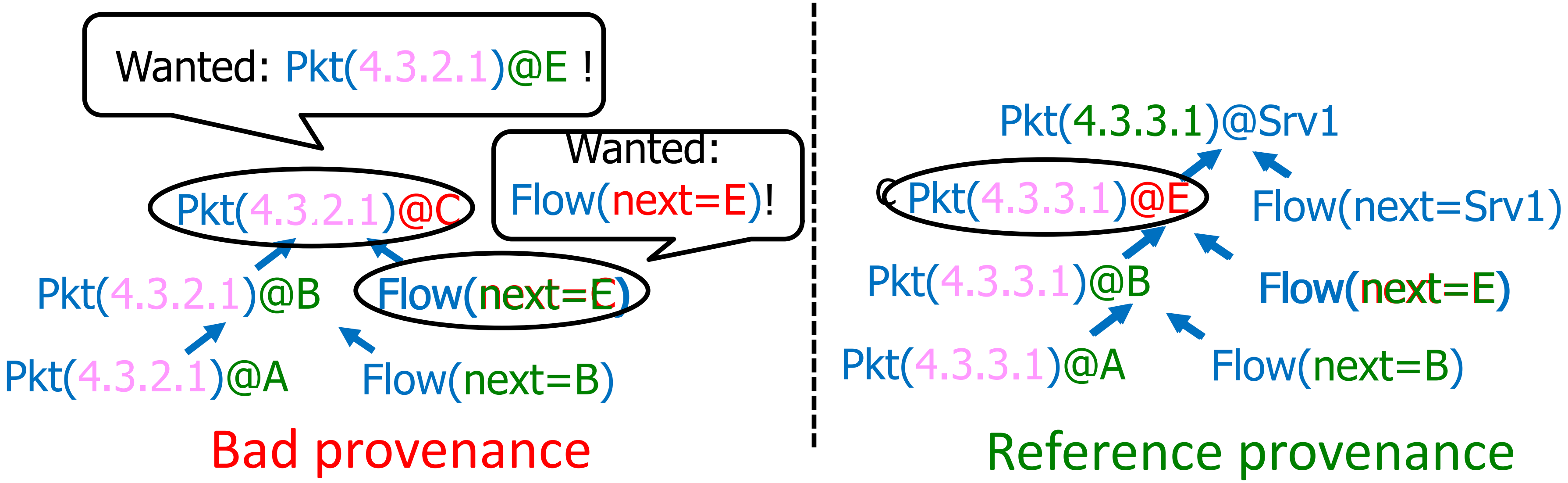
pktstat (51 , 808 , 2)

pkt(51 , 101) pktcnt(1)

Reference

- Approach:
  - Create taints for equivalent fields
  - Propagate taints up the tree
  - Repeat until we find a non-equivalent node

Changing the faulty node



Wanted: Pkt(4.3.2.1)@E !

Wanted: Flow(next=E)!

Pkt(4.3.2.1)@C

Pkt(4.3.2.1)@B

Flow(next=E)

Pkt(4.3.2.1)@A

Flow(next=B)

Bad provenance

Pkt(4.3.3.1)@Srv1

Pkt(4.3.3.1)@E

Flow(next=Srv1)

Pkt(4.3.3.1)@B

Flow(next=E)

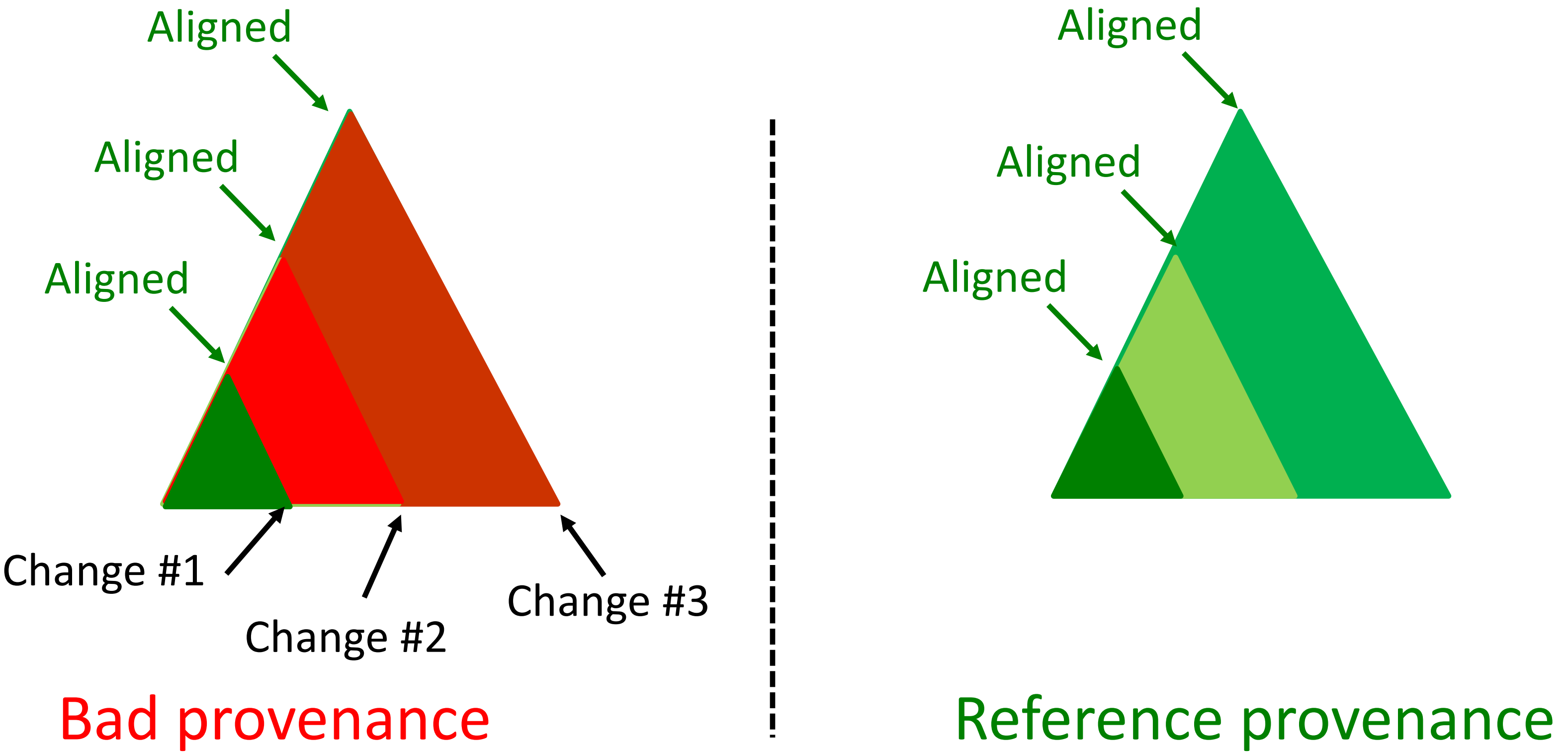
Pkt(4.3.3.1)@A

Flow(next=B)

Reference provenance

- Change the faulty node to its equivalent: Pkt(4.3.2.1)@C → Pkt(4.3.2.1)@E
  - Have dependent nodes → Create their equivalents recursively
    - Example: Flow(next=C) → Flow(next=E)
  - No dependent nodes → Insert its equivalent
    - Example: Insert Flow(next=E)
  - See paper for how to propagate taints in the reverse direction

Problem: Multiple faults



Aligned

Aligned

Aligned

Change #1

Change #2

Change #3

Bad provenance

Aligned

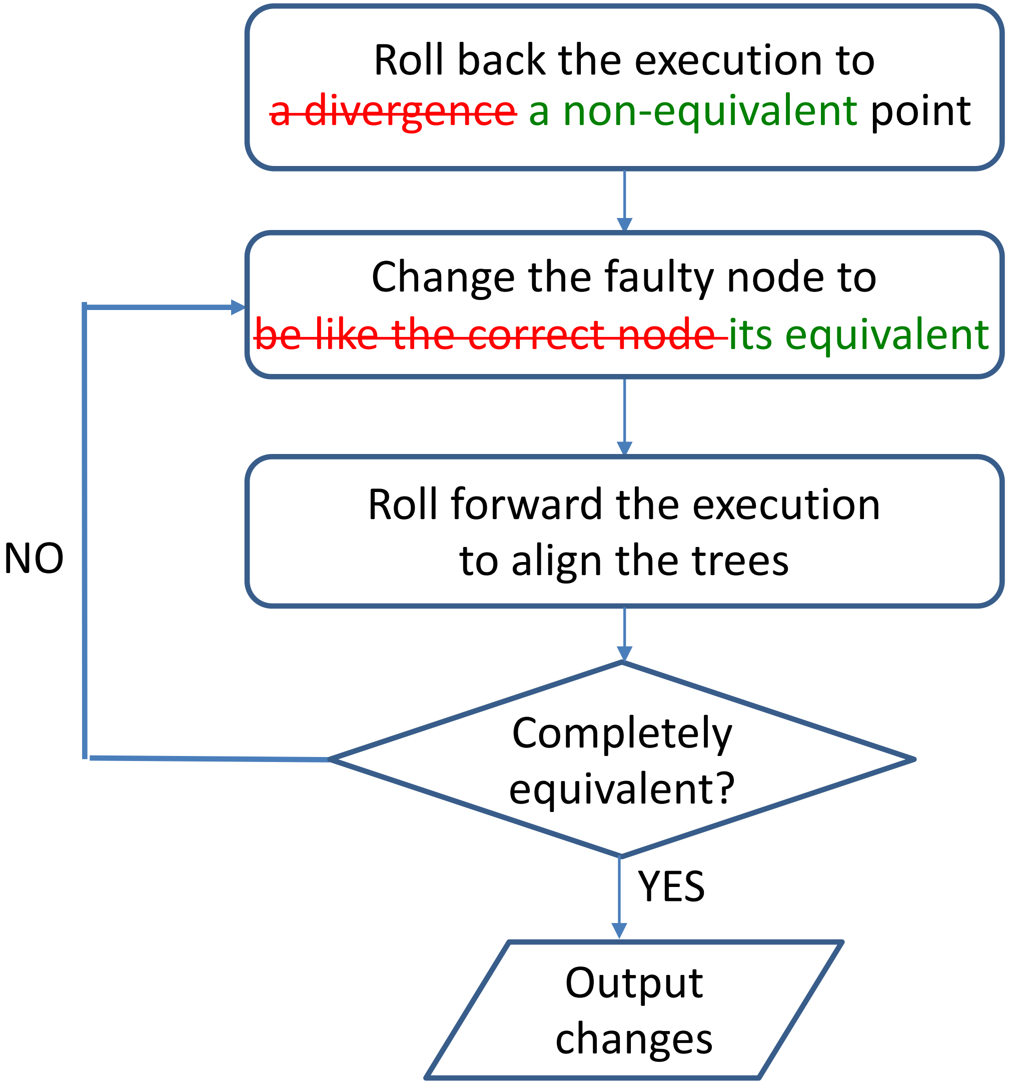
Aligned

Aligned

Reference provenance

- Problem: There could be more than one difference between the two trees
- Solution: Repeat until the trees are completely aligned

Refinement #3: Final algorithm



Roll back the execution to ~~a divergence~~ a non-equivalent point

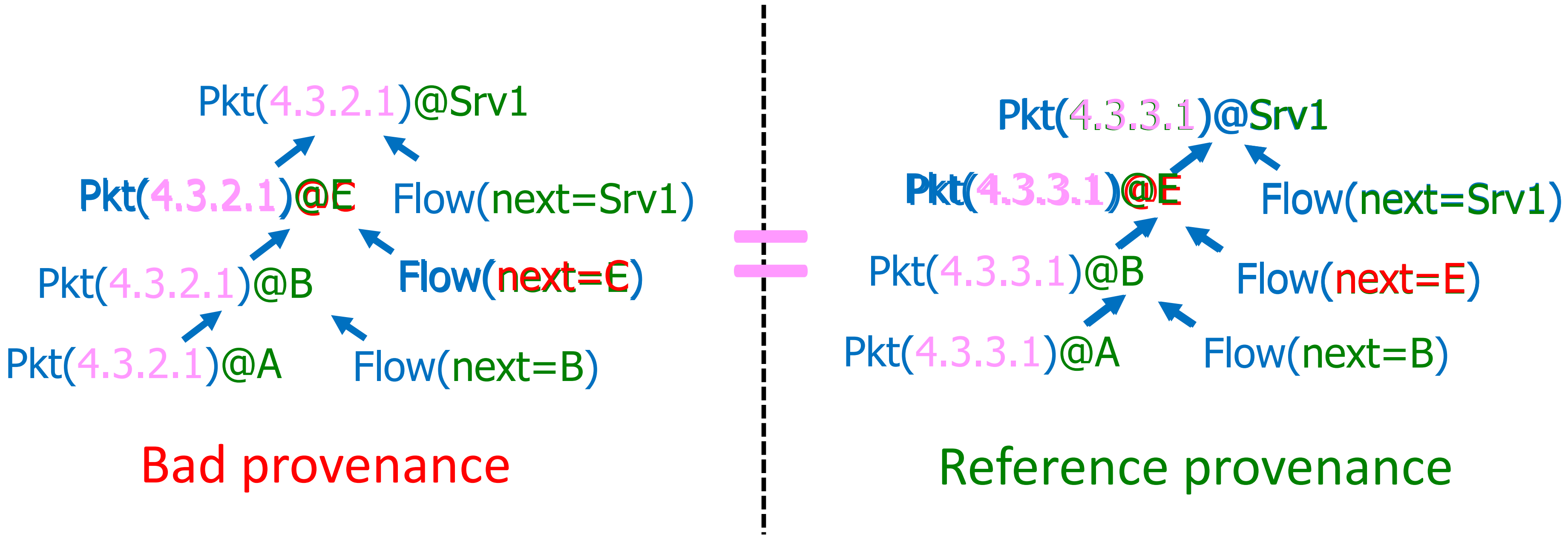
Change the faulty node to ~~be like the correct node~~ its equivalent

Roll forward the execution to align the trees

Completely equivalent?

Output changes

Rolling forward the execution



Bad provenance

Reference provenance

- Roll the execution forward to align the trees
  - Output the accumulated change(s): Flow(next=C) →Flow(next=E)!

Outline

- ✓ Motivation: Root cause analysis
- ✓ Differential provenance
  - ✓ Background: Provenance
  - ✓ Strawman approach
  - ✓ Algorithm
- ⇨ Evaluation
  - Prototype implementation
  - Usability
  - Query processing speed
  - Complex network diagnostics
- Conclusion

Prototype implementation: DiffProv

- Mostly focuses on Network Datalog (NDlog) [CACM ’2009] programs, where provenance is easy to see
- NetCore [NSDI ’13] programs are also supported
- Applicable beyond SDN: Hadoop MapReduce
- Integrated with Mininet + the Beacon controller; based on Rapidnet

Evaluation: Overview

✓ Q1: How well does DiffProv find the root cause?

Q2: How much overhead does DiffProv incur at runtime?

✓ Q3: How quickly does DiffProv answer diagnostic queries?

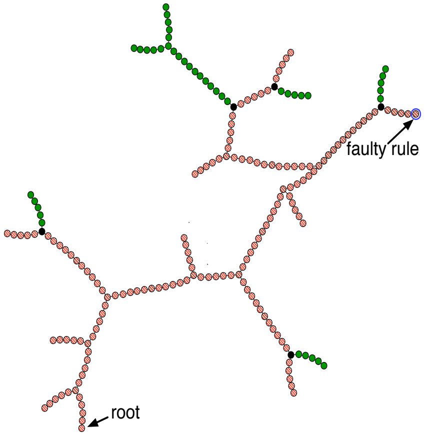
Q4: How well does DiffProv recognize bad reference events?

✓ Q5: How well does DiffProv work for complex networks?

Experimental setup

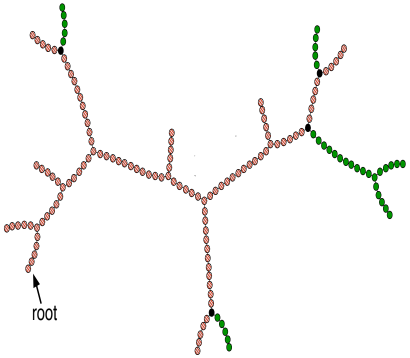
- We adapted seven diagnostic scenarios:
  - SDN1: Broken flow entry [Empr.Soft.Eng.'09]
  - SDN2: Multi-controller inconsistency [CoNEXT'14]
  - SDN3: Unexpected rule expiration [P2P'13]
  - SDN4: Multiple faulty entries [Empr.Soft.Eng.'09]
  - Complex network diagnostics [CoNEXT'12]
  - MR1: Configuration changes [Industry collaborators]
  - MR2: Code changes [Industry collaborators]
- Baseline: Y!, a provenance debugger without reference support [SIGCOMM'14]

How well does DiffProv find the root cause?



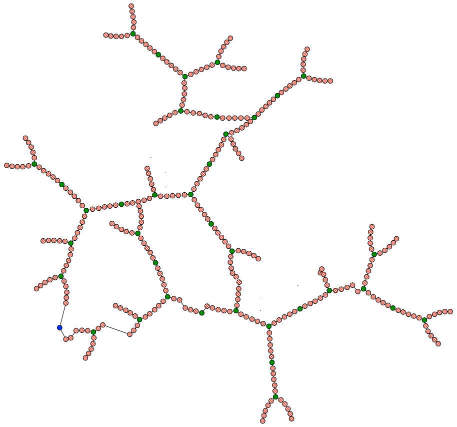
Provenance (symptom)

201 nodes



Provenance (reference)

156 nodes



Naïve diff

278 nodes

Next hop of rule 7 is wrong

DiffProv

1 node!

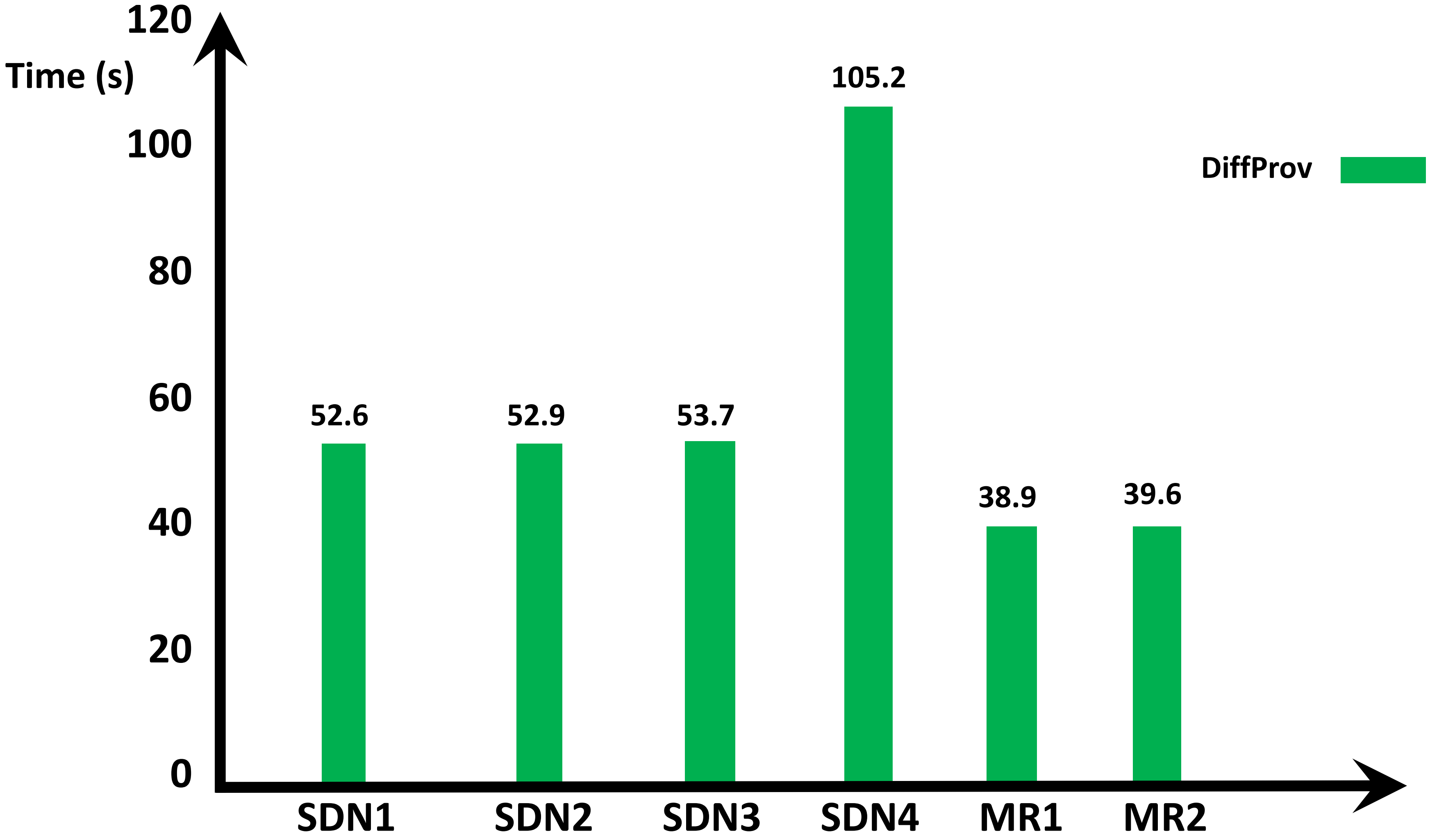
How well does DiffProv find the root cause? (Cont'd)

Query	SDN1	SDN2	SDN3	SDN4
Num. of faults	1	1	1	2
DiffProv	1	1	1	2
Reference	156	156	156	201/201
Symptom	201	156	201	156/145
Plain tree diff	278	238	74	278/218

Query	MR1-D	MR2-D	MR1-I	MR2-I
Num. of faults	1	1	1	1
DiffProv	1	1	1	1
Reference	1051	1001	588	588
Symptom	1051	848	588	438
Plain tree diff	164	306	240	216

- DiffProv finds one or two nodes (the faulty rules or MapReduce configuration entries), which are the actual root cause

How long does DiffProv take to find the root causes?



- DiffProv answered most of our queries within one minute!

How well does DiffProv work in complex networks?

- Setup: larger topology, complex config, background traffic
  - ‘Forwarding error’ scenario [ATPG-CoNEXT’12]
  - Stanford network: 757,000 forwarding entries and 1,500 ACL rules
  - Multiple faults: Injected 20 additional faulty entries
  - Background traffic: 12GB traffic, 69 protocol types
- Results:
  - DiffProv: the faulty entry for misconfigured subnet – one node
  - Identified the root cause despite heavy interference
- Why is DiffProv not confused by the interference?
  - Provenance captures causality, not merely correlations!

Summary

- Debugging networks is hard
  - Need good debuggers to find root causes!
- Key insight: We can use reference events
  - We often have more information than we are using
  - Idea: Reason about the differences between bad events and reference events
- Approach: Differential provenance
  - We have built a prototype debugger for SDNs
  - Applicable to other distributed systems beyond SDNs
- Result: Very precise diagnostics
  - Differential provenance can often identify a single root cause

Thank you!