Automatically Correcting Networks with NEAt

Wenxuan Zhou, Jason Croft, Bingzhe Liu, Elaine Ang, Matthew Caesar
University of Illinois at Urbana-Champaign
A simple idea about complexity…

Networks are so complex it’s hard to make sure they’re doing the right thing.
They instill policies:

- no untrusted traffic entering a secure zone
- the preference of one path over another
- loop & black hole avoidance
- …

What is a network supposed to do?

Intents

IPv6

OSPF-TE

VPN

RSVP-TE

MPLS

Multicast

Virtualization

Anycast

VLAN

NAT

iBGP, eBGP

Cloud

IPSec
A Typical Enterprise Network

Network errors are common

Lots of problems arise today

89% of operators are not certain their configuration changes are safe. [Kinetic NSDI’15]
Networks are so complex it’s hard to make sure they’re doing the right thing.

Let’s automate.

What to automate?
Many have tried.

Automatically identifying errors in networks (Verification)

Automatically synthesize correct networks

≈ a compiler
Repeated when policies change
Do not cooperate with users (operators/applications)
What if we can automatically correct networks on the fly?

Policies

Network abstraction

What if we can automatically correct networks on the fly?

Policies

Network abstraction
What if we can automatically correct networks on the fly?

Auto-correct is not a new idea.

Using the policy graph, NEAt's verification and correction engines ensure the network's verification and repair process by checking properties against a set of repaired updates, which the application can accept or reject. If the application accepts the changes, it sends them onto the network and updates its state, ensuring the application and network state are consistent.
What if we can automatically correct networks on the fly?

**Network Abstraction**

Operating on the data plane simplifies our work

- Diagnose problems as close as possible to actual network behavior
- Data plane is a “narrower waist” than configuration
Goal: Improve upon a manual effort with transparency in both performance and architecture.

Challenge 1: Repair speed

- Based on real-time verification technique
- Derive fixes via linear optimization, with min. changes
- Topology limitation & graph compression

Challenge 2: Zero/minimal architecture/application changes

- Minimal changes
- Pass-through mode
- Interactive mode
Design of NEAt

Controller

Stream of Updates

Corrected Updates

Policy

NEAt

Network Diagram
<table>
<thead>
<tr>
<th>Policy</th>
<th>Network Model</th>
<th>Verification Engine</th>
<th>Correction Engine</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Controller</strong></td>
<td>Stream of Updates</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td><strong>Network Events</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Design of NEAt
Application Mode

Interactive

Proposed Updates

Suggested Changes

NEAt

Controller

App

App

Stream of Updates

Pass-Through
Policy as Graphs

Graphs are **neat**

- Network state synthesis → viewing the network as a whole.
- Graphs → richer set of policies.

A *policy graph* is defined on a packet header pattern

- ip dst 10.0.1.0/24, port 443.

<table>
<thead>
<tr>
<th>Reachability</th>
<th>m = 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bounded path length (shortest path)</td>
<td>m = 1</td>
</tr>
<tr>
<td></td>
<td>n = path_length</td>
</tr>
<tr>
<td>Multipath/Resilience</td>
<td>m = k (k &gt; 1)</td>
</tr>
<tr>
<td>Isolation</td>
<td>m = 0</td>
</tr>
</tbody>
</table>
Use *policy graphs* to express both *qualitative* and *quantitative* reachability constraints.
Repair Algorithm

Cast the problem as an optimization problem:
- Map forwarding graph to policy graph
- Minimize # of changes
Cast the problem as an optimization problem:

- Map forwarding graph to policy graph
- Minimize # of changes
- boolean variable \( x_{i,j,p,q} \):
  - topology edge \((i,j)\) → policy edge \((p,q)\)
  - s.t., policy level reachability \((p,q)\)
Repair - Generalized Reachability

Basic Reachability

∀(i, j) \quad x_{i,j} \geq \sum_{(p,q)\in E_{p\leftarrow}} \frac{x_{i,j,p,q}}{N(E_{\phi})}

∀(j, q) \quad \frac{\sum_{(p,q)\in E_{p\leftarrow}} x_{j,i,p,q}}{N(E_{\phi})} - x_{j,i} \leq 1

∀(j, i) \quad x_{i,j} + x_{j,i} \leq 1

∀(p, q), \forall i \in T:

\left\{ \begin{array}{ll}
\sum_{j\in NB_T(i)} x_{i,j,p,q} = 1 & \text{if } i = p \\
\sum_{j\in NB_T(i)} x_{j,i,p,q} = 0 & \text{if } i = q \\
\sum_{j\in NB_T(i)} x_{j,i,p,q} = 1 & \text{otherwise}
\end{array} \right.

Flow conservation

\sum_{j\in NB_T(i)} (x_{i,j,p,q} - x_{j,i,p,q}) = 0

No tight loops

\sum_{j\in NB_T(i)} x_{i,j,p,q} = 1

\sum_{j\in NB_T(i)} x_{j,i,p,q} = 1

\sum_{j\in NB_T(i)} x_{i,j,p,q} = 0

\sum_{j\in NB_T(i)} x_{j,i,p,q} = 0

Flow sinks at DROP node

\sum_{j\in NB_T(i)} x_{i,j,p,q} = 0

if i = DROP

\sum_{j\in NB_T(i)} x_{i,j,p,q} = 0

if i = q

Correct waypoint order

\sum_{j\in NB_T(i)} x_{i,j,p,q} \geq m

if i = p

\sum_{j\in NB_T(i)} x_{i,j,p,q} = 0

if i = q

Bounded or Equal Path Length

\sum_{j\in NB_T(i)} x_{i,j,p,q} \geq m

# of paths \geq m

MultiPath (Resilience)

Load Balancing

Flow distribution propagates
The preceding algorithm operates on a *loop-free* graph.

First check for and remove loops before repairing other type violations.

Objective: Minimize changes

- Remove the minimal # of rules.
- Affect few packets as possible.
  - E.g. remove a rule matching 10.0.0.1/32 over one for 10.0.0.0/8.
Scalability Challenge and Solution

Scalability challenge

- $\# \text{ of variables} \approx |E(G_{\text{topo}})| \times |E(G_{\text{policy}})|$
- Easily exceeds 100k

Solution: ?
A Typical Enterprise Network

Scalability Challenge and Solution

Scalability challenge

- # of variables $\approx |E(G_{topo})| \times |E(G_{policy})|$
- Easily exceeds 100k

Solution

- Topology Limitation
- Graph Compression \textit{w.r.t} policy
  - Key: Compressed graph $\equiv$ original graph
  - Bisimulation Based Graph Compression
Prototype implementation in Python

- Use Gurobi within optimization engine
- Pass-through mode: proxy
- Interactive mode: XML-RPC API

Datasets:

- Synthetic fat-tree configurations
- SDN applications
- 244-router enterprise network trace
Pox + Mininet

- Learning switch app (pass-through)
- Load balancing app (interactive)
244 routers, one million forwarding rules

Policy: loop freedom & reachability

Issues found and repaired:

• Loops caused by default route
• Load balancing shouldn’t be turned on
Repair vs. Synthesis

Synthesizer (NetGen) as repair tool

<table>
<thead>
<tr>
<th>#TopoLinks</th>
<th>NEAt</th>
<th>NetGen</th>
<th>NetGen-C</th>
</tr>
</thead>
<tbody>
<tr>
<td>96</td>
<td>5.9ms</td>
<td>743.2ms</td>
<td>513.2ms</td>
</tr>
<tr>
<td>324</td>
<td>7.2ms</td>
<td>4404.0ms</td>
<td>1160.8ms</td>
</tr>
<tr>
<td>768</td>
<td>9.0ms</td>
<td>16337.7ms</td>
<td>2056.3ms</td>
</tr>
</tbody>
</table>

NEAt as synthesizer

<table>
<thead>
<tr>
<th>#TopoLinks</th>
<th>NEAt</th>
<th>NetGen</th>
</tr>
</thead>
<tbody>
<tr>
<td>96</td>
<td>921.7ms</td>
<td>7.1min</td>
</tr>
<tr>
<td>324</td>
<td>16.3s</td>
<td>381.7min</td>
</tr>
<tr>
<td>768</td>
<td>2.9min</td>
<td>173.2hrs</td>
</tr>
</tbody>
</table>
Isn’t that NEAt?

**NEAt**, a system analogous to a smartphone’s autocorrect.

- Casting the problem as an optimization problem
- Millisecond to second repair speed
- Generic policy support

Future work:

- Evolving & richer policies
- Different optimization goals
- Repair relevance study
Graph Compression

\textit{w.r.t policy}

Key: Compressed graph == original graph

Major building block:

- Bisimulation Based Graph Compression*

*Query preserving graph compression. \textit{SIGMOD 2012. W. Fan et al}
Graph Compression (Cont’d)

- Customized policy-preserving compression
  
<table>
<thead>
<tr>
<th>Topology</th>
<th>Compression Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fattree (6750 hosts, 1125 switches)</td>
<td>99.38%</td>
</tr>
<tr>
<td>Enterprise (236 routers)</td>
<td>88.98%</td>
</tr>
</tbody>
</table>

- Incremental Compression

- Repair compressed graphs

  Modified mutipath constraint

- Mapping back

- Proved Policy Perseverance
Model packet space as a set of Equivalence Classes

*Equivalence class (EC):* Packets experiencing the same forwarding actions throughout the network.

0.0.0.0/1  64.0.0.0/3

Fwd’ing rules
Equiv classes

Model forwarding behavior of each EC as a directed graph
Insights Behind NEAt

Preventing errors at run-time

- Allow arbitrary SDN applications to run on top
- Not restricted to any programming language
- Influence updates, rather than synthesize from scratch

Graphs are neat

- Networks are graphs
- Model network forwarding behaviors as directed graphs
- Represent operator intents as a policy graph

Discovering repairs

- Equivalent to modifying network state graph so that there exists a mapping between it and the policy graph
Approach: **Data plane verification & repair**

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Data-plane</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prediction is difficult:</td>
<td>Closer to actual network behavior</td>
</tr>
<tr>
<td>• Various configuration languages</td>
<td>Unified analysis for multiple control-plane protocols</td>
</tr>
<tr>
<td>• Dynamic distributed protocols</td>
<td>Can catch control-plane bugs</td>
</tr>
<tr>
<td>Misses control-plane bugs</td>
<td>Only detects bugs that are present in the data plane</td>
</tr>
<tr>
<td>Test prior to deployment</td>
<td></td>
</tr>
</tbody>
</table>

Operating on the data plane simplifies our work

- *Diagnose problems as close as possible to actual network behavior*
- *Data plane is a “narrower waist” than configuration*