Toward Synthesis of Network Updates

Andrew Noyes – Cornell University
Todd Warszawski – Cornell University
Pavol Cerny – University of Colorado Boulder
Nate Foster - Cornell University

Presented by: Ronen Jacobi.
Introduction

- Most networks are updated frequently.
- Implementing network updates while traffic continues to flow is very difficult.
- Bad implementing can cause severe damages.
- Mechanisms that were invented are too general and hard to implement or very limited.
Example: firewall

World

S

F1

F2

F3

Internal
• MODULE main
• VAR
• port : {S_0, F1_0, F2_0, F3_0, START, WORLD, DROP};
• src : {Auth, Guest};
• purpose : {Web, Other};
• ASSIGN
• next(port) := case
  • port = START : S_0;
  • port = S_0 & src = Auth : {F1_0, F2_0};
  • port = S_0 & src = Guest : F3_0;
  • port = F1_0 : WORLD;
  • port = F2_0 : WORLD;
  • port = F3_0 & purpose = Web : WORLD;
  • port = F3_0 & purpose = Other : DROP;
  • port = WORLD : WORLD;
  • port = DROP : DROP;
• esac;
• INIT port = START;

• LTLSPEC (purpose = Other & src = Guest -> port = DROP) &
• ((src = Auth | src = Guest & purpose = Web) -> F port = WORLD);
Example: firewall
Example: firewall
Example: Cycle
Example: Cycle

N1

A

P

B

N3

N2
A New Idea

- We’ll use synthesis to generate update mechanisms automatically.

- **Input:**
  - Current configuration
  - Target configuration
  - Invariants.

- **Output:** Sequence of modifications to the forwarding rules of individual switches
Network Model

- **Network topology** = (Sw,P,inport,outport,ingress)
- Sw = Switches
- P = Ports
- \( \text{inport} \in P \times Sw \)
  - for \( p \in P \) there is a unique \( s \in Sw \) with \( \text{inport}(p,s) \).
- \( \text{outport} \in Sw \times P \)
  - for \( p \in P \) there is a unique \( s \in Sw \) with \( \text{outport}(s,p) \).
- Ingress = ports that get packets from WORLD.
Policies

- A *switch policy* *(switch’s forwarding rules)* is a policy of switch *s* if:
  \[ \text{SwitchPol}(p,pt) = (p',pt') \text{ s.t.} \]
  - *pt* is a packet
  - \((p,s) \in \text{inport}\)
  - \((s,p') \in \text{outport}\)

- A *network policy* *(network’s forwarding rules)* is a function
  \[ \text{NetPol}: S \rightarrow \text{SwitchPol} \text{ s.t} \]
  \[ \text{NetPol}(s) = \text{SwitchPol} \Rightarrow \text{SwitchPol} \text{ is the switch policy of } s. \]
Updates

- An *update* is a pair \((s, \text{SwitchPol})\) s.t. SwitchPol is a policy of \(s\).

\[\text{NetPol}(s \leftarrow \text{SwitchPol}) \Rightarrow \text{NetPol}'(s) = \text{SwitchPol} \text{ and for all } s' \neq s, \text{NetPol}'(s')=\text{NetPol}(s').\]
Commands

- A *command* is either an update or a wait command.

- A *wait* command:
  - Disables the ability to update network policy
  - Limits the network to one update while there’s a packet in it.
Why Wait is Important?

S3 checks whether it’s Auth or guest before sending to N3.
Why Wait is Important?

S3 checks whether it’s Auth or guest before sending to N3.
Why Wait is Important?

S3 checks whether it’s Auth or guest before sending to N3
Why Wait is Important?

S3 does not check Packet’s src before sending to N3.
Network States & Transition

- A network state $ns = ((p,pt),NetPol,\text{wait-flag},\text{comSeq})$
- A network transition is a relation $ns \rightarrow ns'$.
- There are 4 types of transitions:
  1. A packet move:
     $ns = ((p,pt),NetPol,\text{wait-flag},\text{comSeq})$
     $ns' = ((p',pt'),NetPol,\text{wait-flag},\text{comSeq})$
  2. An update transition:
     $ns = ((p,pt),\text{NetPol, false}, (s, \text{SwitchPol}).\text{comSeq})$
     $ns' = ((p,pt),\text{NetPol}[s \leftarrow \text{SwitchPol}],\text{false},\text{comSeq})$
  3. A wait transition:
     $ns = ((p,pt),\text{NetPol,wait-flag,wait.comSeq})$
     $ns' = ((p,pt),\text{NetPol,true,comSeq})$
Network Transition

4. A new packet transition:
   ns = (((p,pt),NetPol,wait-flag,comSeq)
   ns' = (((p',pt'),NetPol,false,comSeq)
   where p' ∈ ingress.
Network Trace

- A network trace $nt$ is an infinite sequence $ns_0ns_1ns_2 \ldots$ s.t. for all $i \geq 0$ $ns_i \rightarrow ns_{i+1}$ is a network transition.

- A network state $ns$ is wait-correct if there are no loops in its network policy.
Update Synthesis Problem

Given:
- an initial network policy $NetPol_i$,
- a final network policy $NetPol_f$,
- specification $\varphi$,

construct a sequence of commands $\text{comSeq}$ such that:

• $NetPol_i \to NetPol_f$ and

• $\text{comSeq}$ is correct with respect to $\varphi$ and $NetPol_i$. 
Procedure \text{ORDERUPDATE}(NetPol_i, NetPol_f, \varphi)

Input: Initial network policy $NetPol_i$, final network policy $NetPol_f$, and LTL specification $\varphi$.

Output: Simple and careful sequence of switch updates $L$, if it exists

1: if hasLoops($NetPol_i$) \lor hasLoops($NetPol_f$) then
2: \hspace{1em} return "Loops in initial or final configuration."
3: else
4: \hspace{1em} $W \leftarrow$ false \hspace{2em} \text{Wrong configurations.}
5: \hspace{1em} $V \leftarrow$ false \hspace{2em} \text{Visited configurations.}
6: \hspace{1em} (ok, $L$) \leftarrow \text{DFSforOrder}(NetPol_i, \bot)
7: \hspace{1em} if ok then
8: \hspace{2em} return $L$
9: \hspace{1em} else
10: \hspace{2em} return "No simple and careful update sequence exists."
Procedure DFSforOrder($NetPol, cs$)

Input: Current network policy $NetPol$, most recently updated switch $cs$.

Output: Boolean $ok$ if a correct update sequence exists; $L$ correct sequence of switch updates

11: if $NetPol = NetPol_f$ then
12: return (true, [NetPol])  \> Reached final configuration.
13: if $NetPol \models V$ then
14: return (false, [])  \> Already visited $NetPol$.
15: $V \leftarrow V \lor NetPol$  \> Add to visited configurations.
16: if $NetPol \models W$ then
17: return (false, [])  \> Previous counterexample applies.
18: if $cs \neq \bot$ then
19: (ok, cex) $\leftarrow$ hasNewLoops($NetPol, cs$)
20: if ($\neg$ ok) then
21: $W \leftarrow W \lor$ analyzeCex(cex)
22: return (false, [])  \> Learn from loop counterexample.
23: (ok, cex) $\leftarrow$ ModelCheck($NetPol, \varphi$)
24: if ($\neg$ ok) then
25: $W \leftarrow W \lor$ analyzeCex(cex)
26: return (false, [])  \> Learn from property counterexample.
27: for all ($NetPol_{next}, cs$) $\in$ NextPolicies($NetPol$) do
28: (ok, $L$) $\leftarrow$ DFSforOrder($NetPol_{next}, cs$)
29: if ok then
30: return (true, NetPol :: wait :: $L$)
31: return (false, [])  \> Try to update one more switch. \> Recursive call.
Running Example

S3 checks whether it’s Auth or guest before sending to N3
S3 checks whether it’s Auth or guest before sending to N3
Running Example

S3 checks whether it’s Auth or guest before sending to N3.
S3 does not check Packet’s src before sending to N3
Example: Cycle

N1

A

C

B

N3

N2
Experiments
Time of updating with varying number of switches in network

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>100 Nodes</th>
<th>250 Nodes</th>
<th>500 Nodes</th>
<th>1000 Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>ORDERUPDATE</td>
<td>10</td>
<td>83</td>
<td>355</td>
<td>1101</td>
</tr>
</tbody>
</table>
Time of updating with varying number of switches to update

<table>
<thead>
<tr>
<th>Nodes</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>30</th>
<th>60</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>165</td>
<td>142</td>
<td>166</td>
<td>222</td>
<td>222</td>
<td>205</td>
<td>273</td>
<td>276</td>
<td>354</td>
<td>339</td>
<td>370</td>
<td>611</td>
<td>2106</td>
</tr>
</tbody>
</table>
Impossible updates

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>100 Nodes</th>
<th>250 Nodes</th>
<th>500 Nodes</th>
<th>1000 Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>ORDERUPDATE</td>
<td>19</td>
<td>170</td>
<td>900</td>
<td>3963</td>
</tr>
<tr>
<td>ORDERUPDATE w/o counterexamples</td>
<td>101</td>
<td>1793</td>
<td>6269</td>
<td>Timeout</td>
</tr>
</tbody>
</table>
Summary

• The running times are too large for online use.
• The purpose was to build a prototype tool to confirm the feasibility of the new approach.
• The counterexample analysis has great influence on the running times, though.
• If there is no way of safe update it takes much longer to finish running, because there is a large number of update sequences possible