Program Synthesis for Network Updates

Pavol Černý
University of Colorado Boulder
Networks
Distributed Firewall example

Traffic: (u)known, (g)uest, (s)tudent, (f)aculty

Properties
a) All SSH traffic from Unknown and Guest is dropped
b) All other traffic gets to the World
Network Updates

- The situation has changed: (more Unknown and Guest traffic)
- Need to update the network configuration from Initial to Final configurations.
- Assumption: single switch updates happen atomically and instantaneously.
What is the problem with this update step?

Recall the properties of interest:

- All SSH traffic from Unknown, Guest is dropped
- All other traffic gets to the World
A solution

Properties

- All SSH traffic from Unknown, Guest is dropped
- All other traffic gets to the World
Synthesis of Updates

Input:
- initial network configuration
- final network configuration
- set of path properties (in LTL)

Output:
- sequence of switch updates
  such that the path properties hold for every packet that traverses the network while updates are performed

a) All SSH traffic from Unknown and Guest is dropped
b) All other traffic gets to the World
Algorithm

- **High-level structure:** Depth-first search with counterexamples.
  Tries to update switches one-by-one, looking for the correct order. If a configuration is found to be wrong, we get a counterexample.

Two main ideas

I. **Incremental** model checking (for LTL)

II. **Heuristic:** use (partial) model checking results to direct the DFS
No forwarding loops

Critical Observation:
Correct network configurations do not produce forwarding loops.

Therefore:
We obtain tree-like Kripke structures.

The observation greatly simplifies (incremental) model checking.
Model checking for LTL on tree-like structures

One sentence summary:
The idea is the same as in LTL-to-Büchi construction, but on tree-like structures it is possible to check all constraints locally (no need for the Büchi condition).
Model checking for LTL on tree-like structures

One sentence summary:
The idea is the same as in LTL-to-Buechi construction, but on tree-like structures it is possible to check all constraints locally (no need for the Buechi condition).

One more level of details:
Labeling by maximally consistent sets of subformulas (and their negations)

Temporal properties checked at sink nodes
sink nodes $s$ labeled by $M$ implies that $\varphi_1 \mathbf{U} \varphi_2 \in M$ iff $\varphi_2 \in M$

and between parent and child in the tree:
a node $n$ is labeled by $M$ implies that $\varphi_1 \mathbf{U} \varphi_2 \in M$ iff
either $\varphi_2 \in M$
or there is a child $n'$ of $n$ labeled by $M'$ and $\varphi_1 \in M$ and $\varphi_1 \mathbf{U} \varphi_2 \in M$
Incremental model checking LTL

Example: We are updating the state K.

The label at state K has changed.
Example: We are updating the state K.

The label at state K has changed.
The label at its parent has not changed.

We can stop propagating the update.
Heuristic synthesis search chooses updates that cause “minimal waves” in the model checking process.

How do we choose which switch to update?

Sending $s$ traffic to F3 does not cause labels to change.

(all the other updates cause “waves”)

a) All SSH traffic from Unknown and Guest is dropped

b) All other traffic gets to the World
Examples for evaluation

1. Topology Zoo

2. Fat Tree Topologies

3. Small World Topologies
Experiments: Incremental vs Monolithic (NuSMV)

<table>
<thead>
<tr>
<th>Total</th>
<th>Count</th>
<th>Incr</th>
<th>NuSMV</th>
<th>Win</th>
</tr>
</thead>
<tbody>
<tr>
<td>212</td>
<td>186</td>
<td>Yes</td>
<td>Yes</td>
<td>160</td>
</tr>
<tr>
<td></td>
<td>25</td>
<td>No</td>
<td>No</td>
<td>25</td>
</tr>
<tr>
<td>12</td>
<td>4</td>
<td>Yes</td>
<td>timeout</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>No</td>
<td>timeout</td>
<td>8</td>
</tr>
<tr>
<td>14</td>
<td>9</td>
<td>Yes</td>
<td>crash</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>No</td>
<td>crash</td>
<td>5</td>
</tr>
<tr>
<td>21</td>
<td>21</td>
<td>timeout</td>
<td>timeout</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>timeout</td>
<td>crash</td>
<td>0</td>
</tr>
</tbody>
</table>

Desktop 2.2GHz processor, 16GB RAM

Topology Zoo dataset
Experiments: scalability

Small world Topologies
Acknowledgments

Andrew Noyes, Todd Warszawski, Pavol Cerny, Nate Foster
*Toward Synthesis of Network Updates*, SYNT 2013

I. Counterexample-guided search

Jedidiah McClurg, Nate Foster, Pavol Cerny
*Efficient Synthesis of Network Updates*, submitted

I. Incremental model checking (for LTL)

II. Heuristic: use (partial) model checking results to direct the DFS
Summary

Two main ideas

I. Incremental model checking (for LTL)

II. Heuristic: use (partial) model checking results to direct the DFS

Work in progress:
- Fast updates (eliminating wait commands)
- Failure recovery, robustness
- Bandwidth constraints
Program Synthesis for Network Updates

Pavol Černý
University of Colorado Boulder
LTL Properties

- Reachability
  Every packet that starts in $s_i$ reaches $d_i$
  \[ \forall_i (\text{port} = s_i \rightarrow F \text{ port } d_i) \]

- Waypointing
  "a packet does not exit the network without passing through $w$"
  \[ \neg g \ U \ w \]
  \[ F \ g \land \neg g \ U \ w \]

- Service chaining
  "all packets go first through $w_1$ and then through $w_2$, before exiting the network"
  \[ \neg g \ U \ w_2 \land \neg w_2 U \ w_1 \]
Counterexamples

Use of counterexamples.
If a configuration is found to be wrong, we get a counterexample.

Counterexample: (sequence of pairs (node;bool); bool indicates whether node has been updated)

(Start,false) (F2,true)

Use the counterexample to avoid model checking calls.
Restrictions

Two restrictions on the search space

i. Every node updated at most once. Simple sequence of updates.

ii. Wait between every two updates. Careful sequence of updates.
   a) Enables checking configurations in isolation.
   b) Requires checking loop-freedom. (At each step, we check that the node we updated is not a part of a loop.)
In-flight packets

What is the problem with this step?
Waiting

Solution: wait between updates until packets exit.

Caveat: works only if there are no loops in the network policy.
Without waiting, a packet can experience three configurations.

With waiting, it experiences at most two. Careful sequence of updates. Enables checking configurations in isolation.

ii. Wait between every two updates. Careful sequence of updates. Enables checking configurations in isolation.
Existing Solution

Consistent updates [Reitblatt, SIGCOMM 2012]

- **Goal**: make sure that every packet gets processed by only switches in the old configuration, or only switches in the new configuration; but not a mixture of both
- **Technique**: distribute versioning (two-phase updates)
- **Problem**: requires large storage on the switches (two routing tables) as it preserves all properties of initial and final configurations
- **Our approach**: preserves only specified properties

---

Diagram:

![Diagram showing network flow](Diagram.png)

- a) All SSH traffic from Unknown and Guest is dropped
- b) All other traffic gets to the World