Software-Defined Networking (SDN)

Enables new functionality through programmability ...
... at the risk of bugs

Network Operating System

A fatal exception has occurred at 10.3.0.5/C001E36 in OF(O) + 00010E36. The current OpenFlow application will be terminated.

* Press any key to terminate the current OpenFlow application
* Press CTRL+ALT+DEL again to restart your network. Your users will lose all network connectivity.

Press any key to continue

Software Faults

• Will make communication unreliable

• Major hurdle for success of SDN

We need effective ways to test SDN networks
This talk: automatically testing OpenFlow Apps
Quick OpenFlow 101

Execute packet_in event handler program

Default: forward to controller

Install rule; forward packet

System is distributed and asynchronous → can misbehave under corner cases

Bugs in OpenFlow Apps

Controller

Install rule

Install rule Delayed!

Drop packet

Inconsistent distributed state!

Goal: systematically test possible behaviors to detect bugs
Systematically Testing OpenFlow Apps

- Carefully-crafted streams of packets
- Many orderings of packet arrivals and events

Scalability Challenges

- Data-plane driven
  - Huge space of possible packets
  - Equivalence classes of packets

- Complex network behavior
  - Huge space of possible event orderings
  - Domain-specific search strategies

Enumerating all inputs and event orderings is intractable
NICE found 11 bugs in 3 real OpenFlow Apps
Model Checking

State-Space Model

System State

Controller (global variables)

Environment:

Switches (flow table, OpenFlow agent)
  Simplified switch model

End-hosts (network stack)
  Simple clients/servers

Communication channels (in-flight pkts)
Combating Huge Space of Packets

Equivalence classes of packets:
1. Broadcast destination
2. Unknown unicast destination
3. Known unicast destination

Code itself reveals equivalence classes of packets
Code Analysis: Symbolic Execution (SE)

Symbolic packet \( \lambda \)

1 path = 1 equivalence class of packets = 1 packet to inject

Packet arrival handler

\( \text{is } \lambda . \text{dst broadcast?} \)

\( \lambda . \text{dst} \notin \{\text{Broadcast}\} \land \lambda . \text{dst} \notin \text{mactable} \)

Flood packet

Install rule and forward packet

Infeasible from initial state

Combining SE with Model Checking

Controller state changes

\( \text{discover\_packets} \) transition:

Enable new transitions:
host / send(pkt B)
host / send(pkt C)

Symbolic execution of packet_in handler

New packets

25 Apr 2012  NSDI'12
Combating Huge Space of Orderings

OpenFlow-specific search strategies for up to 20x state-space reduction:
Specifying App Correctness

- Library of **common properties**
  - No forwarding loops
  - No black holes
  - Direct paths (no unnecessary flooding)
  - Etc...
- Correctness is app-specific in nature

API to Define App-Specific Properties

```python
def init():
    init local vars
    register("packet_in")

def on_packet_in():
    check system-wide state
```

Register callbacks to observe transitions

Execute after transitions
Prototype Implementation

• Built a NICE prototype in Python
• Target the Python API of NOX

Experiences

• Tested 3 unmodified NOX OpenFlow Apps
  – MAC-learning switch
  – LB: Web server load balancer [Wang et al., HotICE’11]
  – TE: Energy-aware traffic engineering [CoNEXT’11]

• Setup
  – Iterated with 1, 2 or 3-switch topologies; 1,2,... pkts
  – App-specific properties
    • LB: All packets of same request go to same server replica
    • TE: Use appropriate path based on network load
Results

• NICE found 11 property violations → bugs
  – Few secs to find 1st violation of each bug (max 30m)
  – Few simple mistakes (not freeing buffered packets)
  – 3 insidious bugs due to network race conditions
    • NICE makes corner cases as likely as normal cases

Conclusions

NICE automates the testing of OpenFlow Apps

- Explores state-space efficiently
- Tests unmodified NOX applications
- Helps to specify correctness
- Finds bugs in real applications

SDN: a new role for software tool chains to make networks more dependable.
NICE is a step in this direction!
Related Work (1/2)

- **Model Checking**
  - SPIN [Holzmann’04], Verisoft [Godefroid’97], JPF [Visser’03]
  - Musuvathi’04, MaceMC [Killian’07], MODIST [Yang’09]

- **Symbolic Execution**
  - DART [Godefroid’05], Klee [Cadar’08], Cloud9 [Bucur’11]

- **MC+SE: Khurshid’03**
Related Work (2/2)

- OpenFlow programming
  - Frenetic [Foster’11], NetCore [Monsanto’12]
- Network testing
  - FlowChecker [Al-Shaer’10]
  - OFRewind [Wundsam’11]
  - Anteater [Mai’11]
  - Header Space Analysis [Kazemian’12]

Micro-benchmark of full state-space search

- Single 2.6 GHz core
- 64 GB RAM

Compared with
- SPIN: 7 pings → out of memory
- JPF is 5.5 x slower

<table>
<thead>
<tr>
<th>Pings</th>
<th>Transitions</th>
<th>Unique states</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>470</td>
<td>268</td>
<td>0.94 [s]</td>
</tr>
<tr>
<td>3</td>
<td>12,801</td>
<td>5,257</td>
<td>47.27 [s]</td>
</tr>
<tr>
<td>4</td>
<td>391,091</td>
<td>131,515</td>
<td>36 [m]</td>
</tr>
<tr>
<td>5</td>
<td>14,052,853</td>
<td>4,161,335</td>
<td>30 [h]</td>
</tr>
</tbody>
</table>
State space reduction by heuristics

- Single 2.6 GHz core
- 64 GB RAM

Compared to base model checking

Transitions # / run time [s] to 1st property violation of each bug

<table>
<thead>
<tr>
<th>BUG</th>
<th>PKT-SEQ only</th>
<th>NO-DELAY</th>
<th>FLOW-IR</th>
<th>UNUSUAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>23 / 0.02</td>
<td>23 / 0.02</td>
<td>23 / 0.02</td>
<td>23 / 0.02</td>
</tr>
<tr>
<td>II</td>
<td>18 / 0.01</td>
<td>18 / 0.01</td>
<td>18 / 0.01</td>
<td>18 / 0.01</td>
</tr>
<tr>
<td>III</td>
<td>11 / 0.01</td>
<td>16 / 0.01</td>
<td>11 / 0.01</td>
<td>11 / 0.01</td>
</tr>
<tr>
<td>IV</td>
<td>386 / 3.41</td>
<td>1661 / 9.66</td>
<td>321 / 1.1</td>
<td>64 / 0.19</td>
</tr>
<tr>
<td>V</td>
<td>22 / 0.05</td>
<td>Missed</td>
<td>21 / 0.02</td>
<td>60 / 0.18</td>
</tr>
<tr>
<td>VI</td>
<td>48 / 0.05</td>
<td>48 / 0.06</td>
<td>31 / 0.04</td>
<td>49 / 0.07</td>
</tr>
<tr>
<td>VII</td>
<td>297k / 1h</td>
<td>191k / 39m</td>
<td>Missed</td>
<td>26.5k / 5m</td>
</tr>
<tr>
<td>VIII</td>
<td>23 / 0.03</td>
<td>22 / 0.02</td>
<td>23 / 0.03</td>
<td>23 / 0.02</td>
</tr>
<tr>
<td>IX</td>
<td>21 / 0.03</td>
<td>17 / 0.02</td>
<td>21 / 0.03</td>
<td>21 / 0.02</td>
</tr>
<tr>
<td>X</td>
<td>2893 / 35.2</td>
<td>Missed</td>
<td>2893 / 35.2</td>
<td>2367 / 25.6</td>
</tr>
<tr>
<td>XI</td>
<td>98 / 0.67</td>
<td>Missed</td>
<td>98 / 0.67</td>
<td>25 / 0.03</td>
</tr>
</tbody>
</table>
OpenFlow Switch Model

Example: adding Rule 1 and Rule 2

1) Flow Table

2) Flow Table
   Rule 1

3) Flow Table
   Rule 1
   Rule 2

MAC-learning switch (3 bugs)

BUG-I: Host unreachable after moving
MAC-learning switch (3 bugs)

OpenFlow program

Host A

B->A | port 1
A->B | port 2

Host B

B->A | port 2
A->B | port 1

BUG-I: Host unreachable after moving
BUG-II: Delayed direct path

BUG-I: Host unreachable after moving
BUG-II: Delayed direct path
BUG-III: Excess flooding
Web Server Load Balancer (4 bugs)

Custom property: all packets of same request go to same server replica

**BUG-IV: Next TCP packet always dropped after reconfiguration**
**BUG-V: Some TCP packets dropped after reconfiguration**
**BUG-VI: ARP packets forgotten during address resolution**
**BUG-VII: Duplicate SYN packets during transitions**

Energy-Efficient TE (4 bugs)

- Precompute 2 paths per <origin,dest.>
  - Always-on and on-demand
- Make online decision:
  - Use the smallest subset of network elements that satisfies current demand

**BUG-VIII: The first packet of a new flow is dropped**
**BUG-IX: The first few packets of a new flow can be dropped**
**BUG-X: Only on-demand routes used under high load**
**BUG-XI: Packets can be dropped when the load reduces**
Results

• Why were mistakes easy to make?
  – Centralized programming model only an abstraction

• Why the programmer could not detect them?
  – Bugs don’t always manifest
  – TCP masks transient packet loss
  – Platform lacks runtime checks

• Why NICE easily found them?
  – Makes corner cases as likely as normal cases

Example: MAC-learning Switch

```python
1 ctrl_state = {} # State of the controller is a global variable (a hashtable)
2 def packet_in(sw_id, inport, pkt, bufid): # Handles packet arrivals
3     mactable = ctrl_state[sw_id]
4     is_bcast_src = pkt.src[0] & 1
5     is_bcast_dst = pkt.dst[0] & 1
6     if not is_bcast_src:
7         mactable[pkt.src] = inport
8     if (not is_bcast_dst) and (mactable.has_key(pkt.dst)):
9         outport = mactable[pkt.dst]
10    if outport != inport:
11        match = [DL SRC: pkt.src, DL DST: pkt.dst, DL TYPE: pkt.type, IN PORT: inport]
12        actions = [OUTPUT, outport]
13        install_rule(sw_id, match, actions, soft_timer=5, hard_timer=PERMANENT)
14        send_packet_out(sw_id, pkt, bufid)
15    return
16    flood_packet(sw_id, pkt, bufid)
```
Causes of Corner Cases
(Examples)

• Multiple packets of a flow reach controller
• No atomic update across multiple switches
• Previously-installed rules limit visibility
• Composing functions that affect same packets
• Assumptions about end-host protocols & SW