lecture 24: testing OpenFlow applications
5590: software defined networking

ando wang, Temple University
TTLMAN 401B, R 17:30-20:00
A NICE Way to Test OpenFlow Applications

https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/canini
SDN and software faults (bugs)

SDN raises the risks of bugs?
- wide range of functionality through software
- diverse collection of network operators — 3rd party

SDN depends on reliable control software
- testing OpenFlow applications
bugs in OF applications

for each event, a controller handler installs rules or collects traffic statistics

- conceptually centralized
- but the system is inherently distributed and asynchronous
- an OF application that works correctly most of the time can misbehave under certain event orderings
example bug
the challenges

OF applications execute in a larger environment
- end-hosts send and receive traffic
- switches process packets, install rules, generate events

large space of OF
- switch state
  - switches run their own programs: state for packet processing, counters, timers, priority
- input packets
  - OF match fields: MAC, IP (source, destination), port …
- event ordering
  - packet arrivals, topology changes …
NICE

Input
- OpenFlow controller program
- Network topology
- Correctness properties

NICE
- Model Checking
- Symbolic Execution
- State-space search

Output
- Traces of property violations

2.1 Background on Model Checking

Modeling the state space.

The programmer can specify correctness properties as snippets of Python code that operate on system state, or strategies to reduce the space of event orderings. The bugs in the controller program, as discussed in Section 4, generate event interleavings that are likely to uncover issues. However, applying model checking "out of the box" does not scale. While simplified models of the switches and underlying switches and end hosts; the controller, in turn, affects the behavior of these components. As such, testing is not just a simple matter of exercising every path through the controller program—we must consider the topology, as discussed in Section 2. Our NICE prototype tests unmodified applications written in Python for the popular NOX platform, producing them. The programmer can also use NICE as a debugging tool to identify bugs, forcing model checking to explore all possible in-flight states.

Bringing these ideas together, NICE combines model checking and search to systematically explore the space of possible system behaviors, and checks them against the desired correctness properties. The programmer can also configure the desired search strategy. In the end, NICE outputs property traces as evidence of violations.

- C h a p t e r 2 o f [ 1 9 ]
- s e t t i n g s , s e e C h a p t e r 2 o f [ 1 9 ]
- i n f o r m a t i o n

Figure 2: Given an OpenFlow program, a network topology, and correctness properties, NICE performs a state-space search and outputs traces of property violations. NICE identifies equivalence classes of packets—ranges of header fields that determine unique paths through the system executions or random walks on system states. NICE feeds the network a representative packet from each class by adding a state transition that injects the packet. To reduce the space of event orderings, we propose several domain-specific search strategies that don't scale or a set of "important" invariants, e.g. no forwarding loops or no black holes, and outputs provided by the developer (which is undesirable).
- test un-modified controller programs, by
- automatically generate carefully-crafted streams of packets under many possible event interleavings
NICE

address large space of switch state

- simplified switch / host models
NICE

address large space of input packets
- symbolic executing packet-arrival handlers
- identifies equivalence classes of packets
- feeds the network a representative concrete packet
address large space of event ordering

- domain specific search strategies that are likely to uncover bugs
model checking — systematically explore execution paths, customized with

- simplified model of switches, hosts
- representative packet inputs by symbolic execution
- search strategy by domain knowledge
transition model — controller

controller program

- a set of event handlers
  - events: packet arrivals, topology changes
- state
  - global variables (ctrl_state)
- transition
  - treat each handler a transition
  - (event handler, concrete input)
transition model — controller

controller program

- a set of event handlers
- events: packet arrivals, topology change
- state
- global variables (ctrl_state)
- transition
- treat each handler a transition
- (event handler, concrete input)

```
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]  # Handles packet arrivals
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,
12                       DL_TYPE: pkt.type, IN_PORT: inport}
13           actions = [OUTPUT, outport]
14           install_rule(sw_id, match, actions, soft_timer=5,
15                       hard_timer=PERMANENT)  # 2 lines optionally
16            send_packet_out(sw_id, pkt, bufid)  # combined in 1 API
17        return
18     flood_packet(sw_id, pkt, bufid)
19
def switch_join(sw_id, stats):  # Handles when a switch joins
20       if not ctrl_state.has_key(sw_id):
21           ctrl_state[sw_id] = {}  # Handles when a switch leaves
22
def switch_leave(sw_id):  # Handles when a switch leaves
23       if ctrl_state.has_key(sw_id):
24           del ctrl_state[sw_id]
```
transition model — switches

switch software
- complex but irrelevant

state
- values of all variables

transition
- identify the portions of the code that process packets or control (OpenFlow) messages
transition model — switches

communication channels
- first-in, first-out buffer

transition driven by data packets and OF messages

state
- process_pkt
- process_of

merging equivalent flow tables
- canonical representation, unique order of rules
symbolic executing event handlers

symbolic execution is the natural choice for exploring code paths, but

- limitation
  - NOT scale well because the number of code paths can grow exponentially with
  - branches, inputs

challenges of symbolic executing OpenFlow Apps

- diverse inputs to packet_in handler
  - solution: symbolic packets

- controller state
  - solution: concrete (rather than symbolic) representation
SE + model checking

- model checker runs until
  - visiting all the states
  - or, detecting a first error
SE + model checking

- concrete controller state
- embed the controller state in symbolic execution
- use concrete variables rather than symbolic ones
at any controller state, add a special transition
- **discover_packets**
  - identify the packets that each client should send
  - symbolically executes **packet_in** handler
- at any controller state, add a special transition
  - `discover_states`
  - similar SE technique to deal with traffic statistics
SE + model checking

- for every code path
- instantiate one concrete packet
search strategies

use domain knowledge to reduce the space of event ordering

- focus on those that are likely to uncover bugs

PKT-SEQ: relevant packet sequences

- discover_packets and send can generate a unbounded tree of packet sequences
- bound the tree
  - depth: maximum length of the sequence
  - length of a packet burst: maximum number of outstanding packets

NO-DELAY: instantaneous rule update

- treat communication between a switch and the controller an atomic action
search strategies — continued

UNUSAL: unusual delay and reordering
- eg: if controller event handler installs rules in switches 1, 2, and 3; explores transitions that reverse the order by allowing switch 3 to install its rule first

FLOW-IR: flow independence reduction
- handling of one group is not affected by the presence of another
- explore only one relative ordering between the events affecting each group
correctness properties
correctness properties

correctness property in NICE

- a module defines robust communication delays
- e.g., intentionally wait until a “safe” time to test the property to prevent natural delays from erroneously triggering false violation
correctness properties

correctness property in NICE

- a module defines robust communication delays
  - e.g., intentionally wait until a “safe” time to test the property to prevent natural delays from erroneously triggering false violation

a library

- no forwarding loops
- no black holes
- direct paths
  - once a packet reaches its destination, future packets of the same flow do not go to the controller
- no forgotten packets
  - all switch buffers are empty at the end of system execution
implementation highlights

NICE consists of three parts

- a model checker; a symbolic execution engine; a collection of models
NICE consists of three parts

- a model checker; a symbolic execution engine; a collection of models

model checker

- system state checkpoint and restore
- remember the sequence of transition that created the state and restore it by replaying such sequence
- state-matching
- compare and store the hashes of the explored state
symbolic execution engine

- a concolic-execution engine, a derivative technique of SE
- execute code with concrete instead of symbolic inputs
  - avoids modify Python interpreter, still tracks constraints along the code path
- implement in Python a new “symbolic integer” data type to track constraints; a new arrays of the symbolic integers
- pre-processing — normalize and instrument
- convert Python app into abstract syntax tree (AST)
- manipulate the AST tree:
  - split composite branch of predicates
  - move function calls before conditional expression
  - instrument branches to inform the councilor engine on which branch is take
  - intercept and remove nondeterminism
  - ...
## Performance

**NICE-MC**: full search model checking without SE  
**NO-SWITCH-REDUCTION**: model checking without simplified switch model

<table>
<thead>
<tr>
<th>Pings</th>
<th>Transitions</th>
<th>Unique states</th>
<th>CPU time</th>
<th>Transitions</th>
<th>Unique states</th>
<th>CPU time</th>
<th>( \rho )</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>470</td>
<td>268</td>
<td>0.94 [s]</td>
<td>760</td>
<td>474</td>
<td>1.93 [s]</td>
<td>0.38</td>
</tr>
<tr>
<td>3</td>
<td>12,801</td>
<td>5,257</td>
<td>47.27 [s]</td>
<td>43,992</td>
<td>20,469</td>
<td>208.63 [s]</td>
<td>0.71</td>
</tr>
<tr>
<td>4</td>
<td>391,091</td>
<td>131,515</td>
<td>36 [m]</td>
<td>2,589,478</td>
<td>979,105</td>
<td>318 [m]</td>
<td>0.84</td>
</tr>
<tr>
<td>5</td>
<td>14,052,853</td>
<td>4,161,335</td>
<td>30 [h]</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
- setup
  - host A pings B, which replies with a packet to A, the controller runs MAC learner
  - Linux 2.6.32, 64GB RAM, clock speed of 2.6GHz
- measure metrics as input packets (concurrent pings) increase
  - number of transitions and unique states
  - execution time

### Table: Performance Comparison

<table>
<thead>
<tr>
<th>Pings</th>
<th>Transitions</th>
<th>NICE-MC</th>
<th>CPU time</th>
<th>NO-SWITCH-REDUCTION</th>
<th>CPU time</th>
<th>$\rho$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Unique states</td>
<td></td>
<td>Transitions</td>
<td>Unique states</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>470</td>
<td>268</td>
<td>0.94 [s]</td>
<td>760</td>
<td>474</td>
<td>1.93 [s]</td>
</tr>
<tr>
<td>3</td>
<td>12,801</td>
<td>5,257</td>
<td>47.27 [s]</td>
<td>43,992</td>
<td>20,469</td>
<td>208.63 [s]</td>
</tr>
<tr>
<td>4</td>
<td>391,091</td>
<td>131,515</td>
<td>36 [m]</td>
<td>2,589,478</td>
<td>979,105</td>
<td>318 [m]</td>
</tr>
<tr>
<td>5</td>
<td>14,052,853</td>
<td>4,161,335</td>
<td>30 [h]</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

**NICE-MC:** full search model checking without SE
**NO-SWITCH-REDUCTION:** model checking without simplified switch model
performance

NICE-MC: full search model checking without SE
NO-SWITCH-REDUCTION: model checking without simplified switch model

<table>
<thead>
<tr>
<th>Pings</th>
<th>Transitions</th>
<th>NICE-MC</th>
<th>CPU time</th>
<th>NO-SWITCH-REDUCTION</th>
<th>CPU time</th>
<th>ρ</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Unique states</td>
<td>CPU time</td>
<td>Transitions</td>
<td>Unique states</td>
<td>CPU time</td>
</tr>
<tr>
<td>2</td>
<td>470</td>
<td>268</td>
<td>0.94 [s]</td>
<td>760</td>
<td>474</td>
<td>1.93 [s]</td>
</tr>
<tr>
<td>3</td>
<td>12,801</td>
<td>5,257</td>
<td>47.27 [s]</td>
<td>43,992</td>
<td>20,469</td>
<td>208.63 [s]</td>
</tr>
<tr>
<td>4</td>
<td>391,091</td>
<td>131,515</td>
<td>36 [m]</td>
<td>2,589,478</td>
<td>979,105</td>
<td>318 [m]</td>
</tr>
<tr>
<td>5</td>
<td>14,052,853</td>
<td>4,161,335</td>
<td>30 [h]</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

- **setup**
  - host A pings B, which replies with a packet to A, the controller runs MAC learner
  - Linux 2.6.32, 64GB RAM, clock speed of 2.6GHz
- **measure metrics as input packets (concurrent pings) increase**
  - number of transitions and unique states
  - execution time

\[
\rho = \frac{Unique(NO-SWITCH-REDUCTION) - Unique(NICE-MC)}{Unique(NO-SWITCH-REDUCTION)}
\]
- reduction relative to full-search NICE-MC
- state space reduction is significant
- for three pings, switch model + heuristics results in a 28-fold reduction
comparison with SPIN

on full search
- SPIN outperforms NICE

state-space explosion
- NICE outperforms SPIN
- SPIN
  - with 7 pings, SPIN runs out of memory
  - SPIN partial-order reduction decreases the growth rate of explored transition by 18%
- NICE
  - simplified switch models, hashing explored states, search strategies