Automatic Concolic Test Generation with Virtual Prototypes for Post-Silicon Validation

Kai Cong, Fei Xie, Li Lei

Presented by: Jianchao Gao, Dike Zhou
Outline

Background
Concolic Test Generation
Implementation
Experimental Results
Conclusion
Outline

Background

Concolic Test Generation

Implementation

Experimental Results

Conclusion
Concolic: CONCrete

Concrete Execution: Executed by concrete values

temperature \( t = 80 \)

- if \( t > 75 \)
  - turn on AC
  - do not turn on AC
Concolic: symbOLIC

Symbolic Execution: Executed by symbolic values

KLEE: Symbolic execution engine
Quiz: Symbolic or Concrete?

memReq(Req r, Addr a, Val& v) {
    case(LOAD):
        loadFromMem(a, v)
    case(STORE):
        storeToMem(a, v)
}

foo() {
    Req r, Addr a, Val v
    memReq(r, a, v)
}

Ans: Symbolic.
Virtual Prototypes

//device state
typedef struct E1000_st {
    ...
} E1000State;
//Interface register function
static void write_reg (...) {
    ...
}
//device transaction function, invoked by reg_func
static void start_xmit (E1000State *s) {
    ...
}
//environment function
static size_t receive (...) {
    ...
}

Virtual prototype: fast and fully-functional software models of hardware system

4 fundamental portions

QEMU (Quick EMUlator): generic & open source machine emulator and virtualizer

Virtual devices have better observability and traceability
Outline

Background

Concolic Test Generation

Implementation

Experimental Results

Conclusion
Motivation

Starting point: run symbolic execution from the reset state to explore all states

Issue?

- How many states exist for a commercial used network adapter chip?
- How many symbolic requests are needed to reach a specified state?

Combine concolic & symbolic tests together (concolic) to generate test cases in order to avoid these issues.
Concolic Test Generation

Apply concrete requests to generate a set of reproducible states \(\{S_0, \ldots, S_n\}\)

- Apply symbolic requests on each states to reach more new states.
- Find the concrete requests that correspond to each transaction.
- Cover all the states along the path
Concolic Test Generation Algorithm

C: set of all computed symbolic executions
TC: set of all generated test cases
s_v: temporary state
r_v: temporary symbolic request
r: temporary concrete request

C = {}, TC = {}, S_v = S_ut
r_v = create_symbolic_request()
C.push(S_v, r_v)

Looped all c in C?

Y

r = generate_conc_req(c)
tc = <seq_k, r>
TC.push(tc)
return TC

N
Transaction-Based Test Selection Strategy

Issues of the methodology:

- Large numbers of states generated from concolic execution
- Test cases that covers the same transactions
Transaction-Based Test Selection Strategy

Algorithm

\[
SI = \{\}, \ T = \{\}, \ i = 0, \ s_0 = \text{reset\_device()}
\]

1. \(i < \text{seq.size?}\)
   - If \(Y\): 
     - \(r_{i+1} = \text{get\_req(seq, i+1)}\)
     - \(s_{i+1} = \text{comp\_n\_state}(s_i, r_{i+1})\)
     - \(t = \text{comp\_transaction}(s_i, r_{i+1})\)
     - If \(t \in T?\): 
       - \(\text{T.push(t)}\)
     - \(\text{SI.push(i+1)}\)
     - \(i++\)

2. Return \(\text{SI, T}\)

SI: set of indices of all selected states

T: all unique transactions
Outline

Background

Concolic Test Generation

Implementation

Experimental Results

Conclusion
Framework

Recorder: Captures concrete requests

Symbolic Engine: Explore new states and create concrete test cases

Test Manager: Apply test cases to physical device and check for inconsistency
Harness

Harness: a driver for the virtual devices

Four components:

1. Declarations of the state variable and parameters of entry functions
2. Code for making the state variable and parameters of entry functions symbolic
3. Non-deterministic calls to virtual device entry functions
4. Stub functions for virtual machine API functions invoked by virtual devices
Symbolic Execution Issues

Two problems

Path explosion problem
Number of path grows exponentially...

Environment interaction problem
API functions resulting in unknown values from environments...
Proposed Solutions

Path Explosion Problem

Loop bound: applied on each loop whose loop condition is a symbolic expression

Time bound: symbolic execution will terminate in some finite time

Environment Interaction Problem

If API function calls will affect states in virtual device, implement a stub function
Quiz: Coverage

By applying symbolic execution, can we achieve 100% state coverage?

By applying symbolic execution proposed in this paper, can we achieve 100% state coverage?
Test with Generated Test Cases

Application of test cases

1. Reset both real and virtual devices to desired state
2. Apply test cases to both devices and capture their concrete next-states
3. Check for any inconsistency

1. Reset both real and virtual devices to desired state
   - $(S_{R,0},S_{V,0}) = \text{Reset\_Device}()$
2. Apply test cases to both devices and capture their concrete next-states
   - $(S_{R,i+1},S_{V,i+1}) = \text{Compute\_Next\_State}(S_{R,i},S_{V,i},r_{i+1})$
3. Check for any inconsistency
   - $(S_{R,i+1},S_{V,i+1}) = \text{Check\_State}(S_{R,i+1},S_{V,i+1})$
Test Case Replay

Upon the detection of an inconsistency, the triggering test case can be replayed on the virtual device.

Better observability and controllability of transactions.
Outline

- Background
- Concolic Test Generation
- Implementation
- Experimental Results
- Conclusion
Compare Transaction-Based Strategy with Random Strategy

Proposed selection strategy can generate many more tests than random strategy

<table>
<thead>
<tr>
<th>Requests in Trace</th>
<th>Transaction Strategy</th>
<th>Random Strategy</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>States</td>
<td>Tests</td>
</tr>
<tr>
<td>E1000</td>
<td>60</td>
<td>774</td>
</tr>
<tr>
<td>Tigon3</td>
<td>52</td>
<td>175</td>
</tr>
<tr>
<td>EEPro100</td>
<td>54</td>
<td>357</td>
</tr>
</tbody>
</table>
Proposed selection strategies take a reasonable amount of time.

Only two more new tests found on 6000 states while taking one day of work. Not cost-effective to capture all states. Proposed selection strategy is efficient enough.

### Time usage of transaction-based selection strategy

<table>
<thead>
<tr>
<th>States</th>
<th>Time (Min)</th>
<th>Selection</th>
<th>Generation</th>
<th>Overall</th>
</tr>
</thead>
<tbody>
<tr>
<td>E1000</td>
<td>60</td>
<td>3.5</td>
<td>26.5</td>
<td>30</td>
</tr>
<tr>
<td>Tigon 3</td>
<td>52</td>
<td>2</td>
<td>17</td>
<td>19</td>
</tr>
<tr>
<td>EE Pro 100</td>
<td>54</td>
<td>2</td>
<td>91</td>
<td>93</td>
</tr>
</tbody>
</table>
Test Case Redundancy Elimination

Number of test cases is reduced by one order of magnitude

Number of Tests

<table>
<thead>
<tr>
<th>Device</th>
<th>Before Elimination</th>
<th>After Elimination</th>
</tr>
</thead>
<tbody>
<tr>
<td>EEPro100</td>
<td>2400</td>
<td>200</td>
</tr>
<tr>
<td>Tigon3</td>
<td>2000</td>
<td>200</td>
</tr>
<tr>
<td>E1000</td>
<td>8000</td>
<td>800</td>
</tr>
</tbody>
</table>
Coverage Improvement

Coverage improved in all cases
For E1000 and Tigon3, function coverage can achieve 100%
For Tigon3, branch coverage can be improved by more than 30%
Inconsistencies

Two types of inconsistencies:

Devices are not initialized correctly

Devices update reserved registers
Outline

Background

Concolic Test Generation

Implementation

Experimental Results

Conclusion
Conclusion

ACTG with virtual prototypes can leverage observability and traceability of virtual prototypes.

This approach can generate effective test cases in a modest amount of time using transaction-based test selection strategy, and improve coverage as well.

Several inconsistencies found by tests generated under proposed approach.
Debate:

1. The proposed method requires a virtual device which should be exactly the same in transactions in order to generate a valuable feedback in replay. This can be very effort consuming. Worth it?

1. Can the proposed method replace ordinary test case selection methods, even though the proposed method cannot cover all possible transactions?
Experiment Setup

Summary of three virtual prototypes

<table>
<thead>
<tr>
<th>Virtual Prototype</th>
<th>Harness</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lines</td>
<td>Functions</td>
</tr>
<tr>
<td>E1000</td>
<td>2099</td>
</tr>
<tr>
<td>Tigon3</td>
<td>4648</td>
</tr>
<tr>
<td>EEPro100</td>
<td>2178</td>
</tr>
</tbody>
</table>

Summary of test suite

<table>
<thead>
<tr>
<th>Category</th>
<th>Commands</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic Programs</td>
<td>ifup</td>
<td>Bring a network interface up</td>
</tr>
<tr>
<td></td>
<td>ifdown</td>
<td>Take a network interface down</td>
</tr>
<tr>
<td></td>
<td>ifconfig</td>
<td>Configure a network interface</td>
</tr>
<tr>
<td></td>
<td>ping</td>
<td>Send ICMP ECHO_REQUEST</td>
</tr>
<tr>
<td></td>
<td>scp</td>
<td>Copy files between network hosts</td>
</tr>
</tbody>
</table>

Lightweight harnesses: <100 lines of code

Capture this test suite from concrete executions of virtual devices in QEMU