Automated Unit Testing of Large Industrial Embedded Software using Concolic Testing

**Yunho Kim**, Moonzoo Kim
SW Testing & Verification Group
KAIST, South Korea

**Youil Kim**, Taeksu Kim, Gunwoo Lee, Yoonkyu Jang
Samsung Electronics, South Korea

http://swtv.kaist.ac.kr
Strong IT Industry in South Korea

Time-to-Market? v.s. SW Quality?

Samsung
LG

Automated Unit Testing of Large Industrial Embedded Software using Concolic Testing

Yunho Kim
SWTV Group
Summary of the Talk

- Embedded SW is becoming larger and more complex
  - Ex. Android: 12 MLOC, Tizen > 6 MLOC
- Smartphone development period is very short
  - No time to manually test smartphones sufficiently
- Solution: **Automated unit test generation** for industrial embedded SW using **CONBOL** (CONcrete and symBOLic testing)
  - CONBOL automatically generates unit-test driver/stubs
  - CONBOL automatically generates test cases using concolic testing
  - CONBOL targets crash bugs (i.e. null pointer dereference, etc.)
- CONBOL detected **24 crash bugs** in 4 MLOC Android SW in 16 hours
Contents

• Motivation
• Background on concolic testing
• Overview of CONBOL
  – Unit test driver/stub generator
  – Pre-processor module
• Real-world application: Project S on Samsung smartphones
• Lessons learned and conclusion
Motivation

• Manual testing of SW is often ineffective and inefficient
  – Ineffectiveness: SW bugs usually exist in corner cases that are difficult to expect
  – Inefficiency: It is hard to generate a sufficient # of test cases in a given amount of project time

• For consumer electronics, these limitations become more threatening
  – Complex control logic
  – Large software size
  – Short development time
  – Testing platform limitation
Concolic Testing

• Combine concrete execution and symbolic execution
  – Concrete + Symbolic = Concolic

• In a nutshell, concrete execution over a concrete input guides symbolic execution
  – Symbolic execution is performed along with a concrete execution path

• Automated test case generation technique
  – Execute a target program on automatically generated test inputs
  – All possible execution paths are to be explored
  – Higher branch coverage than random testing
# Industrial Experience w/ Concolic Testing

**Target platform:** Samsung smartphone platforms

<table>
<thead>
<tr>
<th>Testing Level</th>
<th>Target Programs</th>
<th>Results</th>
<th>Publication</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unit-testing</td>
<td>Busybox ls</td>
<td>Detected <strong>4 bugs</strong> and covered 98% of branches</td>
<td>Kim et al. [ICST12]</td>
</tr>
<tr>
<td></td>
<td>Samsung security library</td>
<td>Detected <strong>1 memory bug</strong> and covered 73% of branches</td>
<td>Kim et al. [ICST12]</td>
</tr>
<tr>
<td>System-testing</td>
<td>Samsung Linux Platform (SLP) file manager</td>
<td>Detected <strong>1 infinite loop bug</strong> and covered 20% of branches</td>
<td>Kim et al. [FSE11]</td>
</tr>
<tr>
<td></td>
<td>10 Busybox utilities</td>
<td>Detected <strong>1 bug in grep</strong> and covered 80% of branches</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Libexif</td>
<td>Detected <strong>6 bugs including 2 security bugs</strong> registered in Common Vulnerabilities and Exposures, and covered 43% of branches</td>
<td>Kim et al. [ICSE12]</td>
</tr>
</tbody>
</table>
Obstacles of Concolic Testing for Industrial Embedded SW

1. Each execution path can be very long, which causes a huge state space to analyze
   – Generating and running test cases on embedded platforms would take significant amount of time

2. Porting of a concolic testing tool to a target embedded OS can be difficult
   – Due to resource constraint of embedded platforms

3. Embedded SW often uses target-specific compiler extensions
Solutions of CONBOL

1. Automatically generate unit tests including test drivers/stubs
   – We can apply concolic testing on industrial embedded SW that has 4 MLOC

2. Test embedded SW on a host PC
   – Most unit functions can run on a host PC
     • Only a few unit functions are tightly coupled with target embedded platforms

3. Port target-specific compiler extensions to GCC compatible ones
Overview of CONBOL

- We have developed the CONcrete and symBOLic (CONBOL) framework: an automated concolic unit testing tool based-on CREST-BV for embedded SW.
Porting Module

• The porting module **automatically modifies the source code** of unit functions so that the code can be compiled and executed at the host PC

  1. The porting module removes unportable functions
     • Inline ARM assembly code, hardware dependent code, unportable RVCT(RealView Compilation Tools) extensions

  2. The porting module translates target code to be compatible with GCC and CIL(C Intermediate Language) which is an instrumentation tool

<table>
<thead>
<tr>
<th>RVCT</th>
<th>Translation for GCC</th>
</tr>
</thead>
<tbody>
<tr>
<td>__asm {...}</td>
<td>Not Portable</td>
</tr>
<tr>
<td>__swi (0x01)</td>
<td>Not Portable</td>
</tr>
<tr>
<td>__align(8)</td>
<td><strong>attribute</strong>((aligned(8)))</td>
</tr>
<tr>
<td>__packed</td>
<td><strong>attribute</strong>((packed))</td>
</tr>
</tbody>
</table>
The unit test driver/stub generator **automatically generates unit test driver/stub** functions for unit testing of a target function.

- A unit test driver symbolically sets all visible global variables and parameters of the target function.

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Code Example</th>
</tr>
</thead>
</table>
| Primitive | set a corresponding symbolic value                    | int a;  
SYM_int(a);                                           |
| Array     | set a fixed number of elements                        | int a[3];  
SYM_int(a[0]); ... SYM_int(a[2]);                       |
| Structure | set NULL to all pointer fields and set symbolic value to all primitive fields | struct _st{int n,struct _st*p}a;  
SYM_int(a.n);  
a.p=NULL; |
| Pointer   | allocate memory for a pointee and set a symbolic value of corresponding type of the pointee | int *a;  
a = malloc(sizeof(int));  
SYM_int(*a); |

- The test driver/stub generator replaces sub-functions invoked by the target function with symbolic stub functions.
Example of an automatically generated unit-test driver

```c
typedef struct Node_{
  char c;
  struct Node_ *next;
} Node;
Node *head;

// Target unit-under-test
void add_last(char v){
  // add a new node containing v
  // to the end of the linked list
  ...
}

// Test driver for the target unit
void test_add_last(){
  char v1;
  head = malloc(sizeof(Node));
  SYM_char(head->c);
  head->next = NULL;
  SYM_char(v1);
  add_last(v1); }
```

**Unit Test Driver**

- Generate symbolic inputs for global variables and a parameter
- Call target function
- Set global variables
- Set parameter
Pre-processor Module

• The pre-processor module inserts probes for three heuristics **to improve bug detection precision**
  1. `assert()` insertion to detect more bugs
  2. Scoring of alarms to reduce false alarms
  3. Pre-conditions insertion to reduce false alarms
Unit-testing Strategy to Reduce False Alarms

• CONBOL raises a false NPD alarm because $ctx$(line 6) is not correctly initialized by $init_ctx()$(line 8)
  
  – $init_ctx()$ is replaced with a symbolic stub function
    
    01:int $init_ctx$(struct CONTEXT &$ctx$){
    02:  $ctx$.f = malloc(...);
    03:  ...
    04:  return 0;}
    05:void $f$(){
    06:    struct CONTEXT $ctx$;
    07:    int ret;  \[ init_ctx() \] is replaced with a symbolic stub that does not initialize
    08:    ret = $init_ctx$(&$ctx$);  \[ $ctx$ \]
    09:    if (ret == -1){
    10:      return;}
    11:    if ($ctx$.f[1] > 0){  \[ A false NPD alarm is raised at line 11 because $ctx$ is not properly initialized \]
    12:      /* Some code */
    13:    }
    14:} 

• We are developing a technique to automatically identify sub-functions that should not be replaced with stub functions
Inserting `assert()` Statements

- The pre-processor module automatically inserts `assert()` to cause and detect the following three types of run-time failures
  - Out-of-bound memory access bugs (OOF)
    - Insert `assert(0<=idx && idx<size)` right before array access operations
  - Divide-by-zero bugs (DBZ)
    - Insert `assert(denominator!=0)` right before division operators whose denominator is not constant
  - Null-pointer-dereference bugs (NPD)
    - Insert `assert(ptr!=NULL)` right before pointer dereference operations
Scoring of Alarms (1/2)

• CONBOL assigns a score to each alarm as follows:

  1. Every violated assertion (i.e., alarm) gets 5 as a default score.
  2. The score of the violated assertion **increases by 1** if the assertions contains a variable \( x \) which is checked in the target function containing the assertion (e.g., \( \text{if}(x<y+1) \) ...)

• An explicit check of \( x \) indicates that the developer considers \( x \) important, and the assertion on \( x \) is important consequently.

```c
01: void f(int x, int y) {
02:     int array[10];
03:     if (x < 15) {
04:         assert(x<10);
05:         array[x]++;
06:         assert(y<10);
07:         array[y]++;
08:     } }
```

<table>
<thead>
<tr>
<th>No</th>
<th>Type</th>
<th>Location</th>
<th>Assert Expression</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>OOB</td>
<td>src.c:f():4</td>
<td>x&lt;10</td>
<td>6(=5+1)</td>
</tr>
<tr>
<td>2</td>
<td>OOB</td>
<td>src.c:f():6</td>
<td>y&lt;10</td>
<td>5</td>
</tr>
</tbody>
</table>
3. For each violated assertion \texttt{assert (expr)}, the score of the assertion decreases by 1, if \texttt{expr} appears five or more times in other violated assertions in the entire target software.

- Developers write code correctly most of the time: target code that is repeated frequently is not likely to be buggy.

<table>
<thead>
<tr>
<th>No</th>
<th>Type</th>
<th>Location</th>
<th>Assert Expression</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>OOB</td>
<td>src.c:f():1287</td>
<td>A.index - 1 &gt;= 0</td>
<td>4(=5-1)</td>
</tr>
<tr>
<td>2</td>
<td>OOB</td>
<td>src.c:g():1300</td>
<td>A.index - 1 &gt;= 0</td>
<td>4(=5-1)</td>
</tr>
<tr>
<td>3</td>
<td>OOB</td>
<td>src.c:h():1313</td>
<td>A.index - 1 &gt;= 0</td>
<td>4(=5-1)</td>
</tr>
<tr>
<td>4</td>
<td>OOB</td>
<td>src.c:x():1326</td>
<td>A.index - 1 &gt;= 0</td>
<td>4(=5-1)</td>
</tr>
<tr>
<td>5</td>
<td>OOB</td>
<td>src.c:y():1339</td>
<td>A.index - 1 &gt;= 0</td>
<td>4(=5-1)</td>
</tr>
</tbody>
</table>

- CONBOL reports alarms whose scores are 6 or above.
Inserting Constraints to Satisfy Pre-conditions

• The pre-processor module automatically inserts `assume()` to avoid false alarms due to violation of implicit pre-conditions

  – Pre-conditions for array indexes
    • Insert array pre-conditions if the target function does not check an array index variable

  – Pre-conditions for constant parameters
    • Insert constant parameter pre-conditions if the parameter of the target function is one of some constant values for all invocations
      – Ex.) the third parameter of `fseek()` should be one of `SEEK_SET`, `SEEK_CUR`, or `SEEK_END`

  – Pre-conditions for `enum` values
    • CONBOL considers an `enum` type as a special `int` type and generates concrete test cases defined in the corresponding `enum` type
• An automatically generated unit test driver can violate implicit pre-conditions of a target unit function
  – Violation of implicit pre-conditions raises false alarms

```c
01: int array[10];
02: void get_ith_element(int i){
03:   return array[i];
04:
05: }// Test driver for get_ith_element()
06: void test_get_ith_element(){
07:   int i, idx;
08:   for(i=0; i<10; i++){
09:     SYM_int(array[i]);
10:   }
11:   SYM_int(idx);
12:   get_ith_element(idx);
13: }
```

Line 3 can raise an OOB alarm because \( i \) can be greater than or equal to 10.

However, developers often assume that `get_ith_element()` is always called under a pre-condition \((0\leq i \&\& i<10)\).
Inserting Constraints to Satisfy Pre-conditions (3/3)

• An example of pre-conditions for array index

```
01: int array[10];
02: void get_ith_element(int i){
03:   return array[i];
04: }
05: // Test driver for get_ith_element()
06: void test_get_ith_element(){
07:   int i, idx;
08:   for(i=0; i<10; i++){
09:     SYM_int(array[i]);
10:   }
11:   SYM_int(idx);
12:   assume(0<=idx && idx<10);
13:   get_ith_element(idx);
14: }
```

Developers assume that callers of `get_ith_element()` performs sanity checking of the parameter before they invoke `get_ith_element()`.
Statistics of Project S

- Project S, our target program, is an industrial embedded software for smartphones developed by Samsung Electronics
  - Project S targets ARM platforms

<table>
<thead>
<tr>
<th>Metric</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total lines of code</td>
<td>About 4,000,000</td>
</tr>
<tr>
<td># of branches</td>
<td>397,854</td>
</tr>
<tr>
<td># of functions</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>48,743</td>
</tr>
<tr>
<td>Having more than one branch</td>
<td>29,324</td>
</tr>
<tr>
<td># of files</td>
<td></td>
</tr>
<tr>
<td>Sources</td>
<td>7,243</td>
</tr>
<tr>
<td>Headers</td>
<td>10,401</td>
</tr>
</tbody>
</table>
Test Experiment Setting

• CONBOL uses a DFS strategy used by CREST-BV in Kim et al. [ICSE12 SEIP]

• Termination criteria and timeout setting
  – Concolic unit testing of a target function terminates when
    • CONBOL detect a violation of an assertion, or
    • All possible execution paths are explored, or
    • Concolic unit testing spends 30 seconds (Timeout1)
  – In addition, a single test execution of a target unit should not spend more than 15 seconds (Timeout2)

• HW setting
  – Intel i5 3570K @ 3.4 GHz, 4GB RAM running Debian Linux 6.0.4 32bit
Results (1/2)

• Results of branch coverage and time cost
  – CONBOL tested $86.7\% (=25,425)$ of target functions on a host PC
    • 13.3% of functions were not inherently portable to a host PC due to inline ARM assembly, direct memory access, etc
  – CONBOL covered $59.6\%$ of branches in $15.8$ hours

<table>
<thead>
<tr>
<th>Statistics</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total # of test cases generated</td>
<td>About 800,000</td>
</tr>
<tr>
<td>Branch coverage (%)</td>
<td>59.6</td>
</tr>
<tr>
<td>Execution time (hour)</td>
<td>15.8</td>
</tr>
<tr>
<td># of functions reaching timtout1 (30s)</td>
<td>742</td>
</tr>
<tr>
<td># of functions reaching timtout2 (15s)</td>
<td>134</td>
</tr>
<tr>
<td>Execution time w/o timeout (hour)</td>
<td>9.0</td>
</tr>
</tbody>
</table>
Results (2/2)

- CONBOL raised 277 alarms
- 2 Samsung engineers (w/o prior knowledge on the target program) took 1 week to remove 227 false alarms out of 277 alarms
  - We reported 50 alarms and 24 crash bugs were confirmed by the developers of Project S
- Pre-conditions and scoring rules filtered out 14.1% and 81.2% of likely false alarms, respectively
- Note that Coverity prevent could not detect any of these crash bugs

<table>
<thead>
<tr>
<th># of reported alarms</th>
<th>Out-of-bound</th>
<th>NULL-pointer-dereference</th>
<th>Divide-by-zero</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td># of alarms</td>
<td>Ratio (%)</td>
<td># of alarms</td>
<td>Ratio (%)</td>
</tr>
<tr>
<td>W/O any heuristics</td>
<td>3235</td>
<td>100.0</td>
<td>2588</td>
<td>100.0</td>
</tr>
<tr>
<td>W/ inserted pre-conditions</td>
<td>2486</td>
<td>76.8</td>
<td>2511</td>
<td>97.0</td>
</tr>
<tr>
<td>W/ inserted pre-conditions + scoring rules</td>
<td>220</td>
<td>6.8</td>
<td>42</td>
<td>1.6</td>
</tr>
<tr>
<td>Confirmed and fixed bugs</td>
<td>13</td>
<td>0.4</td>
<td>5</td>
<td>0.2</td>
</tr>
</tbody>
</table>
Lessons Learned

• Effective and efficient automated concolic unit testing approach for industrial embedded software
  – Detected 24 critical crash bugs in 4 MLOC embedded SW

• Samsung engineers were sensitive to false positives very much
  – False alarm reduction techniques are very important

• Technical challenges for achieving high coverage, low false alarm, low testing time
  – Support for efficient unit test driver/stub generation
  – Support for complex symbolic data
  – Support for functional oracle specifications
Recognition of Success of CONBOL at Samsung Electronics

- Bronze Award at Samsung Best Paper Award
- Oct’s Best Practice Award
- Team leader Dr. Yoonkyu Jang received Samsung Award of Honor
Conclusion

• Automated concolic unit testing was effective and efficient for testing industrial embedded software
  – Detected 24 crash bugs in 4MLOC embedded SW

• CONBOL has been successfully adopted by the original development team
  – Applied weekly and detected +40 more bugs so far

Traditional testing
• Manual TC gen
• Testing main scenarios
• System-level testing
• Small # of TCs

CONBOL
• Automated TC gen
• Testing exceptional scenarios
• Unit-level testing
• Large # of TCs