Automated Unit Testing of Large Industrial Embedded Software using Concolic Testing

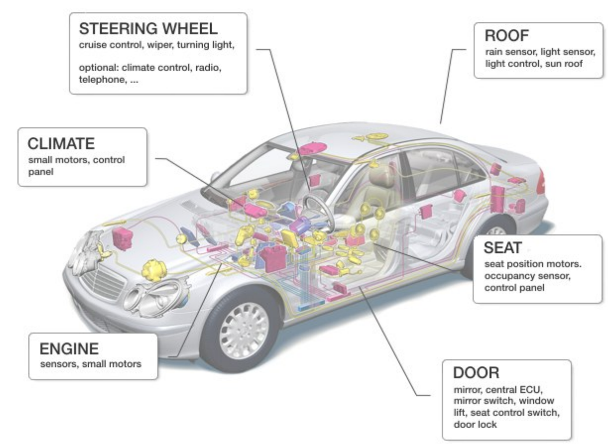
Yunho Kim, Moonzoo Kim
SW Testing & Verification Group
KAIST, South Korea

http://swtv.kaist.ac.kr

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

Strong IT Industry in South Korea

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



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
  - **Conc**rete + Symb**olic** = **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

Testing Level	Target Programs	Results	Publication
Unit-testing	Busybox ls	Detected **4 bugs** and covered 98% of branches	Kim et al. [ICST12]
	Samsung security library	Detected **1 memory bug** and covered 73% of branches	Kim et al. [ICST12]
System-testing	Samsung Linux Platform (SLP) file manager	Detected **1 infinite loop bug** and covered 20% of branches	Kim et al. [FSE11]
	10 Busybox utilities	Detected **1 bug in grep** and covered 80% of branches	
	Libexif	Detected **6 bugs including 2 security bugs** registered in Common Vulnerabilities and Exposures, and covered 43% of branches	Kim et al. [ICSE12]

Obstacles of Concolic Testing for Industrial Embedded SW

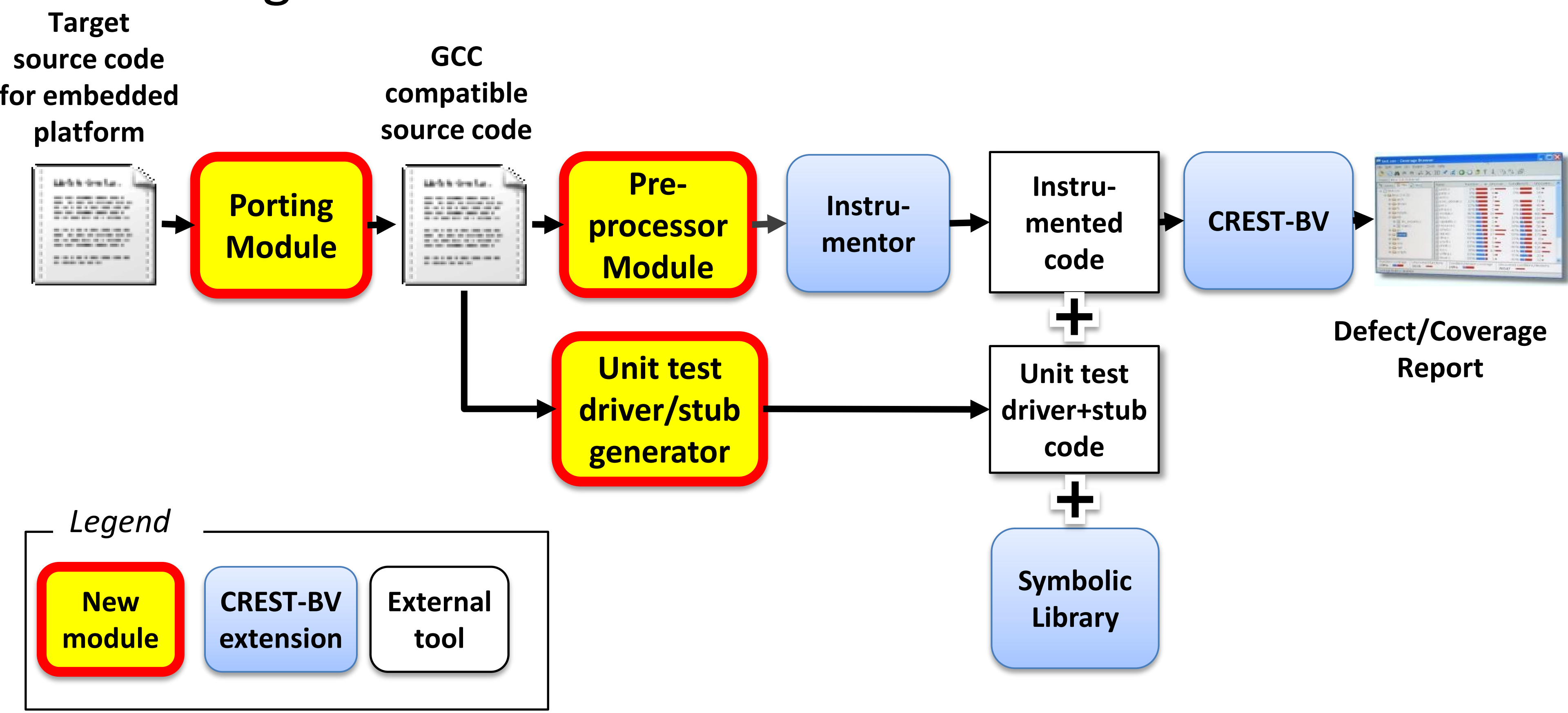
1. Each execution path can be very long, which causes a huge state space to analyze
   – Generating and running test cases on embedded platforms would take significant amount of time
2. Porting of a concolic testing tool to a target embedded OS can be difficult
   – Due to resource constraint of embedded 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

RVCT	Translation for GCC
__asm {…}	Not Portable
__swi (0x01)	Not Portable
__align(8)	__attribute__((aligned(8)))
__packed	__attribute__((packed))

Unit Test Driver/Stub Generator(1/2)

- 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

Type	Description	Code Example
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

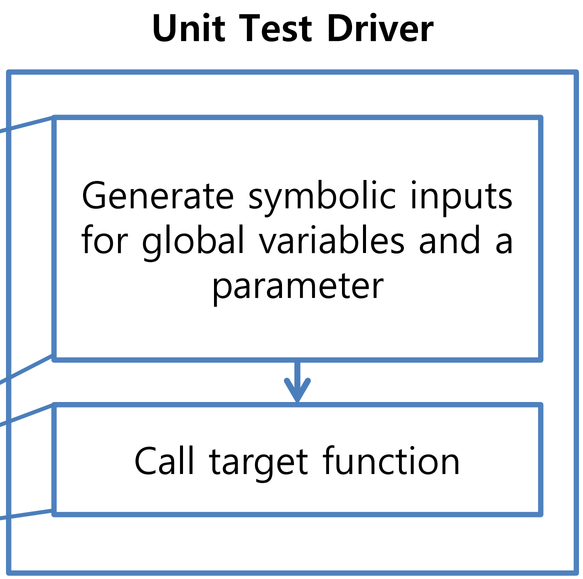
Unit Test Driver/Stub Generator(2/2)

- Example of an automatically generated unit-test driver

```
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); }
```

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;
08:  ret = init_ctx(&ctx);
09:  if (ret == -1){
10:    return;}
11:  if (ctx.f[1] > 0){
12:    /* Some code */
13:  }
14:}
```

`init_ctx()` is replaced with a symbolic stub that does not initialize `ctx`

A false NPD alarm is raised at line 11 because `ctx` is not properly initialized

- 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(OOB)
    - 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., `if(x<y+1)`...)
     - An explicit check of x indicates that the developer considers x important, and the assertion on x is important consequently.

```
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:}}
```

No	Type	Location	Assert Expression	Score
1	OOB	src.c:f():4	x<10	6(=5+1)
2	OOB	src.c:f():6	y<10	5

Scoring of Alarms (2/2)

3. For each violated assertion `assert(`$expr$`)`, the score of the assertion **decreases by 1**, if $expr$ 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

No	Type	Location	Assert Expression	Score
1	OOB	src.c:f():1287	`A.index - 1 >= 0`	4(=5-1)
2	OOB	src.c:g():1300	`A.index - 1 >= 0`	4(=5-1)
3	OOB	src.c:h():1313	`A.index - 1 >= 0`	4(=5-1)
4	OOB	src.c:x():1326	`A.index - 1 >= 0`	4(=5-1)
5	OOB	src.c:y():1339	`A.index - 1 >= 0`	4(=5-1)

- 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 the `enum` type

Inserting Constraints to Satisfy Pre-conditions(1/3)

- An automatically generated unit test driver can violate implicit pre-conditions of a target unit function
  - Violation of implicit pre-conditions raises false alarms

```
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:
13:  get_ith_element(idx);
14:}
```

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<=i && i<10)`

Inserting Constraints to Satisfy Pre-conditions(3/3)

- An example of pre-conditions for array index

```
int array[10];
void get_ith_element(int i){
  return array[i];
}
// Test driver for get_ith_element()
void test_get_ith_element(){
  int i, idx;
  for(i=0; i<10; i++){
    SYM_int(array[i]);
  }
  SYM_int(idx);
  assume(0<=idx && idx<10);
  get_ith_element(idx);
}
```

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

`assume(expr)` enforces symbolic values to satisfy `expr`

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

Metric		Data
Total lines of code		About 4,000,000
# of branches		397,854
# of functions	Total	48,743
	Having more than one branch	29,324
# of files	Sources	7,243
	Headers	10,401

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**

Statistics	Number
Total # of test cases generated	About 800,000
Branch coverage (%)	59.6
Execution time (hour)	15.8
# of functions reaching timtout1 (30s)	742
# of functions reaching timtout2 (15s)	134
Execution time w/o timeout (hour)	9.0

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

# of reported alarms	Out-of-bound		NULL-pointer-dereference		Divide-by-zero		Total	
	# of alarms	Ratio (%)	# of alarms	Ratio (%)	# of alarms	Ratio (%)	# of alarms	Ratio (%)
W/O any heuristics	3235	100.0	2588	100.0	61	100.0	5884	100.0
W/ inserted pre-conditions	2486	76.8	2511	97.0	58	95.1	5055	85.9
W/ inserted pre-conditions + scoring rules	220	6.8	42	1.6	15	24.6	277	4.7
Confirmed and fixed bugs	**13**	**0.4**	**5**	**0.2**	**6**	**9.8**	**24**	**0.4**

Lessons Learned

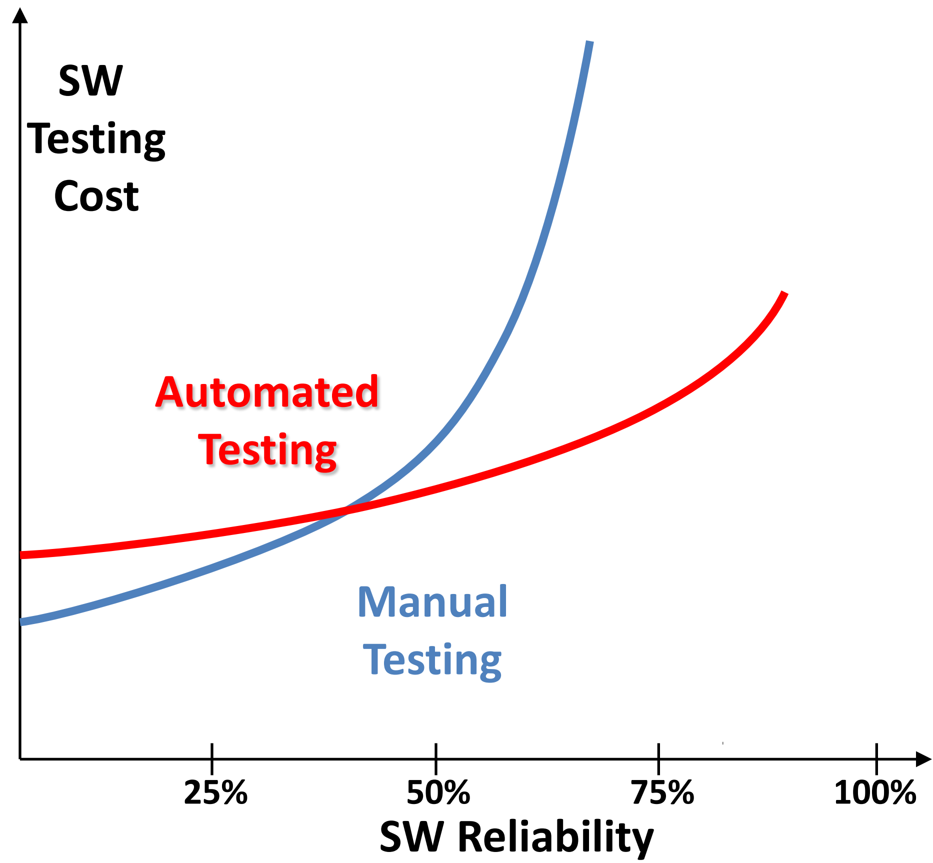
- Effective and efficient automated concolic unit testing approach for industrial embedded software
  - Detected 24 critical crash bugs in 4 MLOC embedded SW
- Samsung engineers were sensitive to false positives very much
  - False alarm reduction techniques are very important
- Technical challenges 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