Automated Analysis of Industrial Embedded Software

Moonzoo Kim and Yunho Kim
Provable Software Lab
KAIST, South Korea

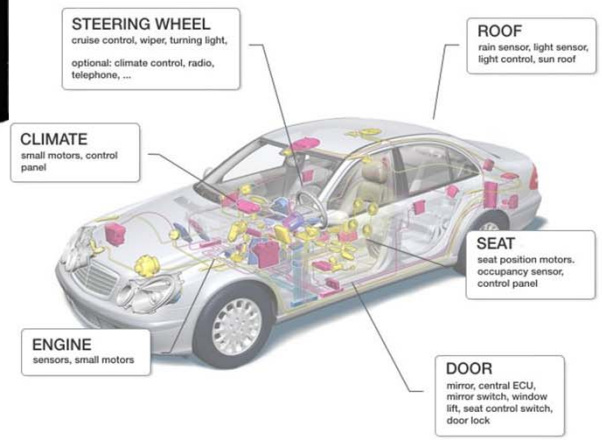
Thanks to Hotae Kim and Yoonkyu Jang
Samsung Electronics, South Korea

Strong IT Industry in South Korea

SAMSUNG

KIA

KIA MOTORS

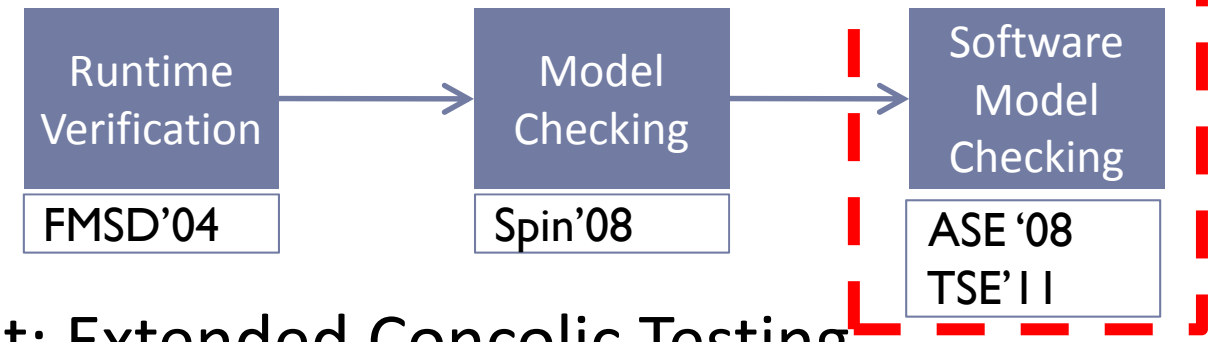


LG

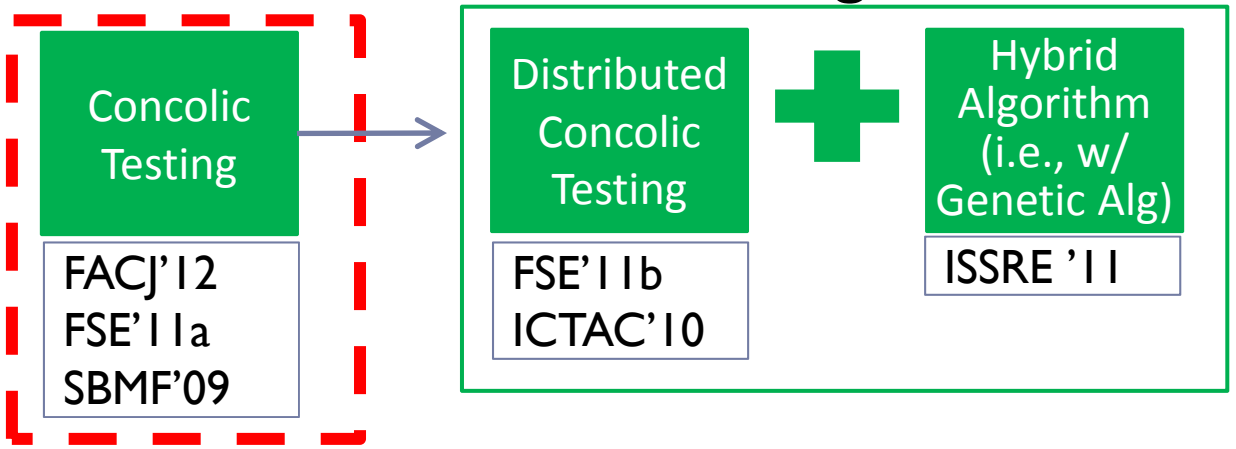
HYUNDAI

Personal Research Roadmap

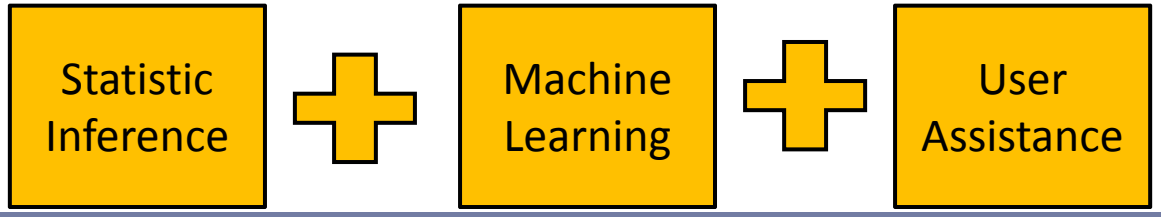
- Past: RV (dynamic) && MC (static)

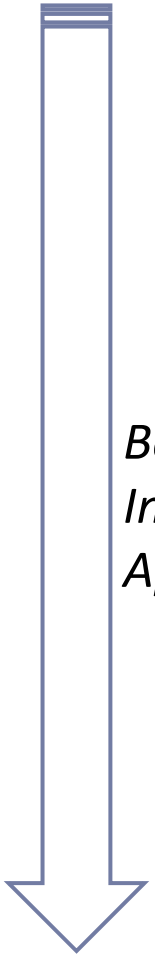


- Current: Extended Concolic Testing



- Future: Concolic Testing with Intelligence



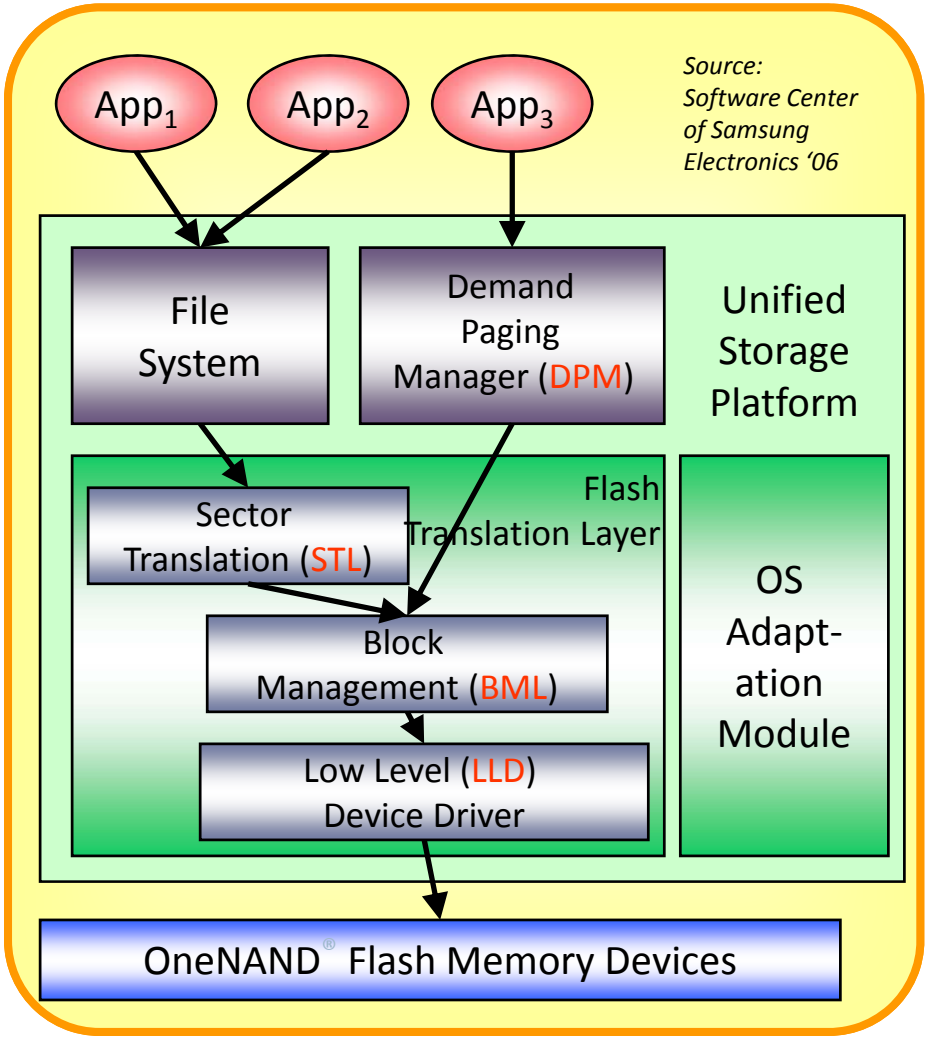


Better Industrial Application

Part I: Experience from SW Model Checking

Target system: Samsung Unified Storage Platform (USP) for OneNAND® flash memory (around 30K lines of C code)

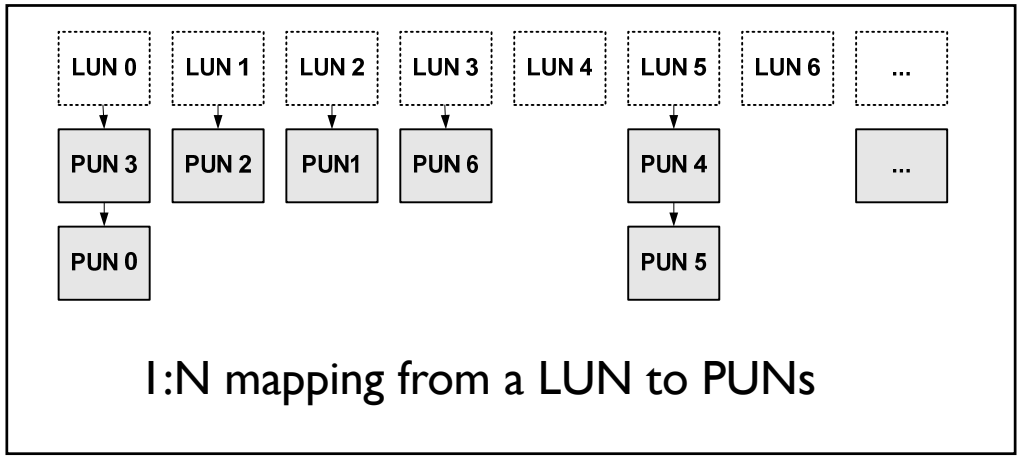
- Characteristics of OneNAND® flash mem
  - Each memory cell can be written limited number of times only
    - Logical-to-physical sector mapping
    - Bad block management, wear-leveling, etc
  - Concurrent I/O operations
    - Synchronization among processes is crucial
  - XIP by emulating NOR interface through demand-paging scheme
    - binary execution has a highest priority
  - Performance enhancement
    - Multi-sector read/write
    - Asynchronous operations
    - Deferred operation result check

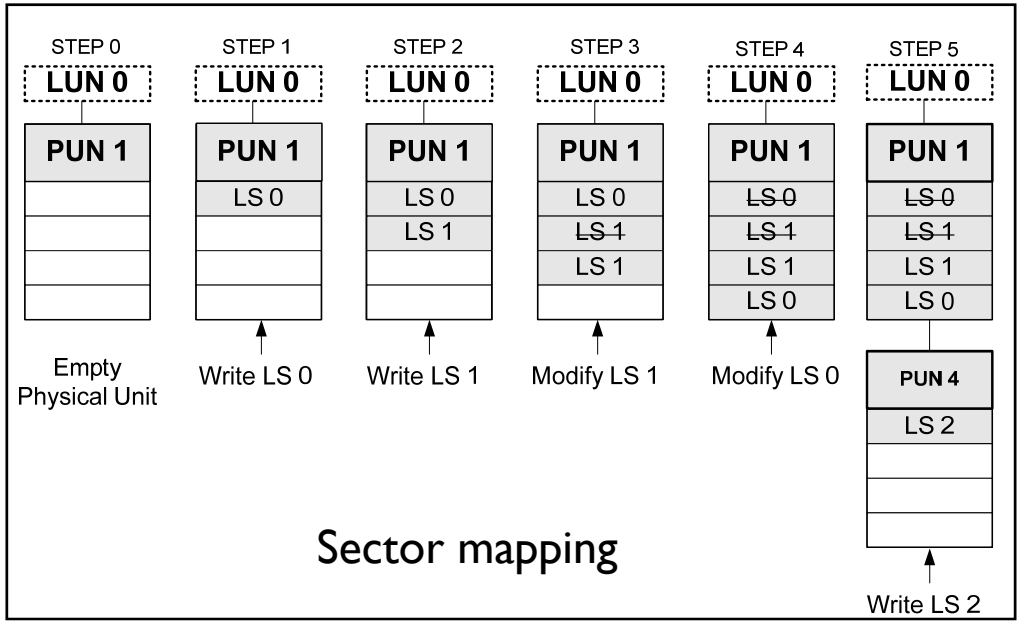


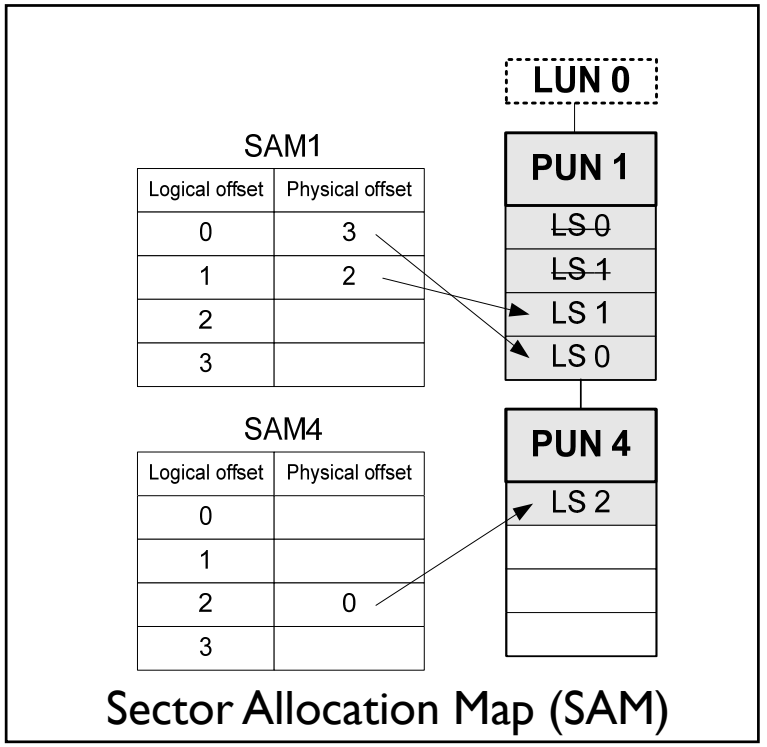
Results of Unit Analysis through CBMC and BLAST [TSE'11]

- Demand paging manager (234 LOC)
  - Detected a bug of not saving the status of suspended erase operation
- Concurrency handling
  - Confirmed that the BML semaphore was used correctly in all 14 BML functions (150 LOC on average)
  - Detected a bug of ignoring BML semaphore exceptions in a call sequence from STL (2500 LOC on average)
- Multi-sector read operation (MSR) (157 LOC)
  - Provided high assurance on the correctness of MSR
    - no violation was detected even after exhaustive analysis (at least with a small number of physical units(~10))
- In addition, we evaluated and compared pros and cons of CBMC and BLAST empirically

Logical to Physical Sector Mapping

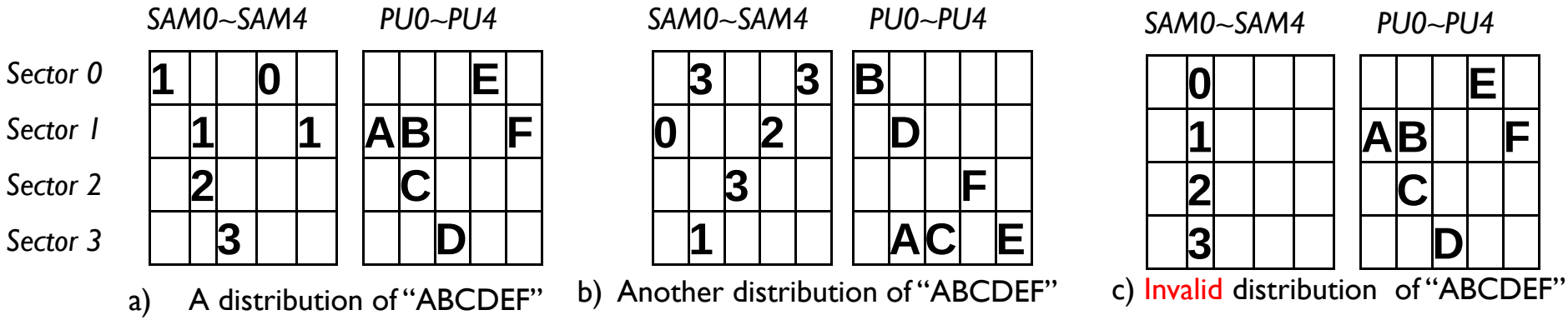






- In flash memory, logical data are distributed over physical sectors.

Multi-sector Read Operation (MSR)



- MSR reads adjacent multiple physical sectors once in order to improve read speed
  - MSR is 157 lines long, but highly complex due to its 4 level loops
  - 4 parameters to specify logical data to read (from, to, how long, read flag)
- The requirement property is to check
  - after_MSR -> ($\forall$ i. logical_sectors[i] == buf[i])
- We built a **verification environment model** for MSR

Environment Modeling

1. **One PU is mapped to at most one LU**
2. **Valid correspondence between SAMs and PUs:**

   If the *i* th LS is written in the *k* th sector of the *j* th PU, then the *i* th offset of the *j* th SAM is valid and indicates the k'th PS ,

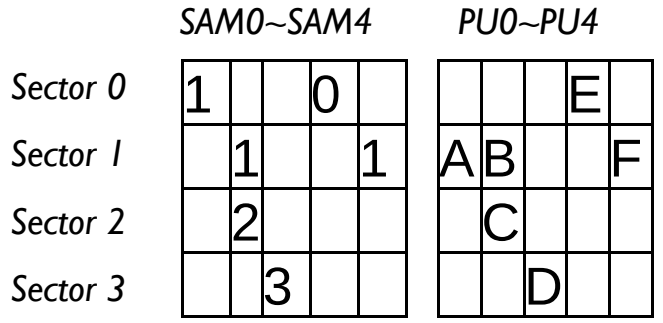
   Ex> 3rd LS ('C') is in the 3rd sector of the 2nd PU, then SAM1[2] ==2

   i=2 k=2 j=1

3. **For one LS, there exists only one PS that contains the value of the LS:**

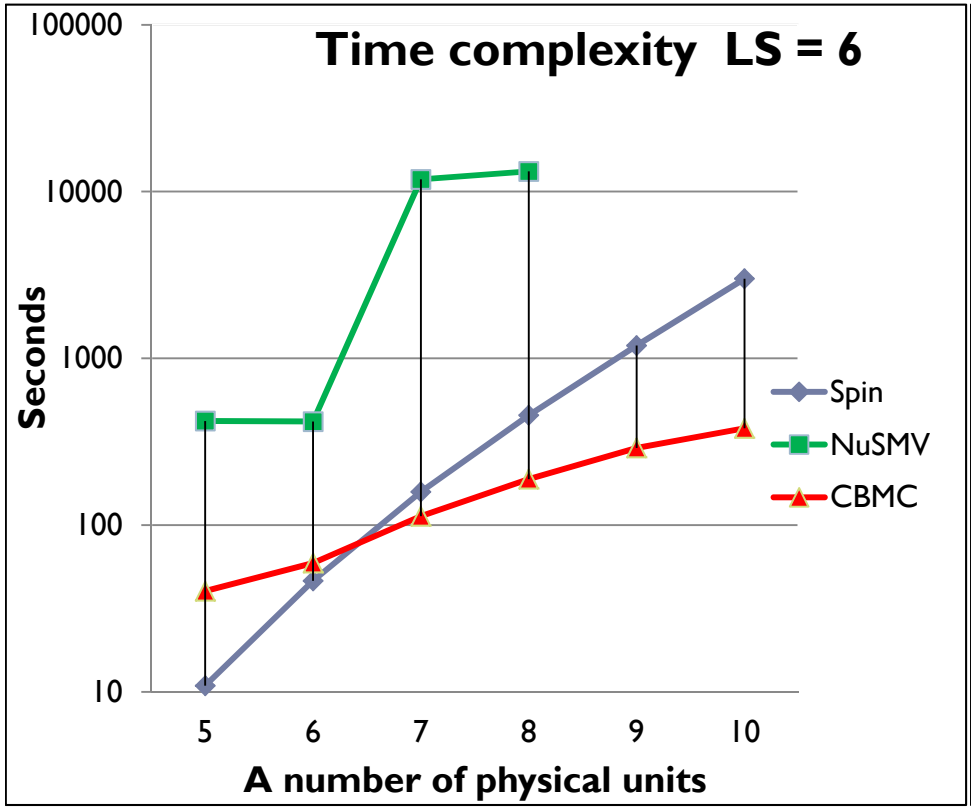
   The PS number of the *i* th LS must be written in only one of the (*i* mod *4*) th offsets of the SAM tables for the PUs mapped to the corresponding LU.

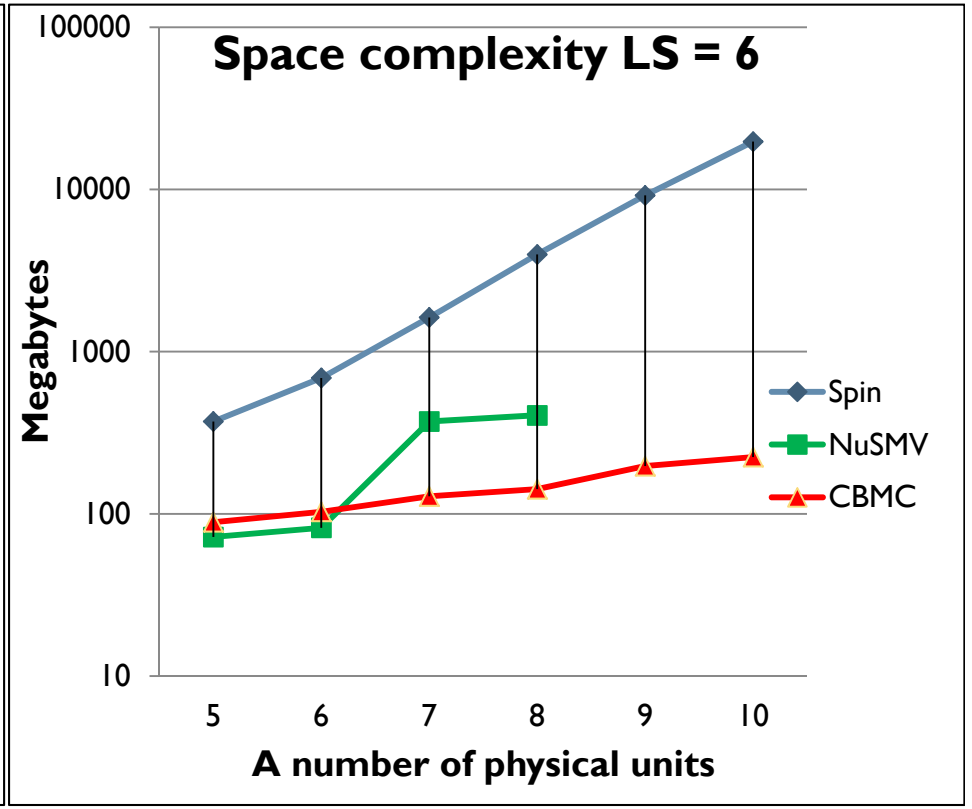
$$\forall i, j, k\ (LS[i] = PU[j].sect[k] \rightarrow (SAM[j].valid[i \bmod m] = true \\ \& SAM[j].offset[i \bmod m] = k \\ \& \forall p.(SAM[p].valid[i \bmod m] = false) \\ \text{where } p \neq j \text{ and } PU[p] \text{ is mapped to} \lfloor \frac{i}{m} \rfloor_{th} LU))$$



Model Checking Results of MSR [Spin'08]

- Verification of MSR by using NuSMV, Spin, and CBMC
- No violation was detected within |LS|<=8, |PU| <=10
  - 10^{10} configurations were exhaustively analyzed for |LS|=8, |PU|=10





Feedbacks from Samsung Electronics

Main challenge :

- IT industry is not mature enough to conduct unit testing

1. Current SW development of Samsung is not ready to apply unit testing
   - Tight project deadline does not allow defining detailed asserts and environment models
2. Needs large scalability even at the cost of accuracy
   - Rigorous automated tools for small unit (i.e., SW model checker) is of limited practical value
3. Many embedded SW components have dependency on external libraries
   - Pure analysis methods on source code only are of limited value
4. It is desirable to generate test cases as a result of the analysis.
   - Current SW V&V practice operates on test cases

Background on Concolic Testing

- **Conc**rete runtime execution guides symb**olic** path analysis
  - a.k.a. dynamic symbolic execution (DSE), white-box fuzzing
- Automated test case (TC) generation technique
  - Applicable to a large target program (no memory bottleneck)
  - Applicable to testing stages seamlessly
  - External binary library can be handled (partially)
- Explicit path model checker
  - All possible execution paths are explored based on the generated TCs
  - Anytime algorithm
    - User can get partial analysis result (i.e., TCs) anytime
  - Analysis of each path is independent from each other
    - Parallelization for linear speed up
    - Ex. Scalable COncolic testing for Reliable Embedded Software (SCORE) on thousands of Amazon EC2 cloud computing nodes [FSE'11b]

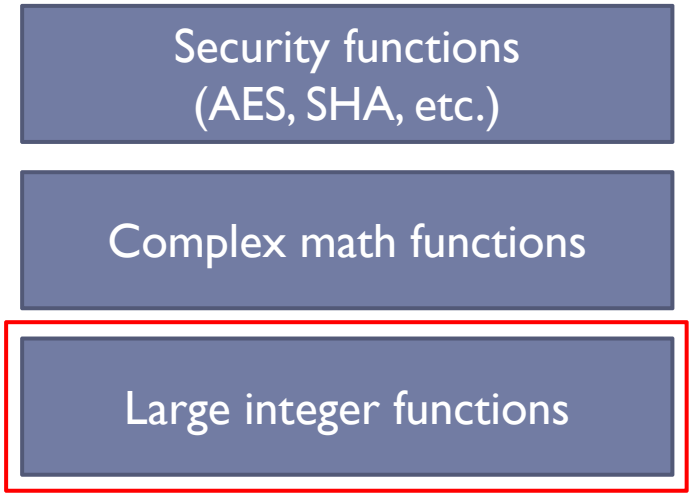
Part II: Experience from Concolic Testing using CREST

Target system: Samsung Smartphone Platform

- Unit-level testing
  1. Busybox ls (1100 LOC)
     - 98% of branches covered and 4 bugs detected
  2. Samsung security library (2300 LOC)
     - 73% of branches covered and a memory violation bug detected
- System level testing
  1. Samsung Linux Platform (SLP) file manager
     - Covered 20% of the branches and detected an infinite loop bug
  2. 10 Busybox utilities
     - Covered 80% of the branches with 4 different search strategy and 10,000 TCs in 20 min each
     - A buffer overflow bug in grep was detected
  3. Libexif
     - Covered 43% of the branches with 4 different search strategy and 10,000 TCs in 8 hours each
     - 2 null pointer dereferences and 5 divide-by-0 bugs were detected

Samsung Security Library [FSE'11a]

- Providing complete security protocol APIs
- 3 level layered structure
- Complex mathematical operation involved



- We targeted the large integer functions layer using the CREST tool
  - Upper 2 layers heavily use external math library functions and are hard to understand due to complex algorithms

Symbolic Inputs

- All 14 functions in the large integer functions layer receive struct L_INT as inputs
- struct L_INT {
  unsigned int size; // Allocated memory size in 32 bits
  unsigned int len; // # of valid 32 bit elements, thus len <= size
  unsigned int *da; // Actual data, da[len-1] are the most-significant bytes
  unsigned int sign; // 0: non-negative, 1: negative
  }
- Ex. 4294967298 (=$2*2^0 + 1*2^{32}$) is represented by

unsigned int size=3;
unsigned int len=2;
unsigned int *da ={2,1,0}; // $2*2^0 + 1*2^{32} + 0*2^{64}$
unsigned int sign=0;

da[0]	da[1]	da[2]
2	1	0

Symbolic L_INT Generator

- `gen_s_int()` generates symbolic L_INT whose size is between min and max
  - Allocate memory of L_INT with symbolic size of data buffer (line 2~6)
  - Fill L_INT data when `to_fill` is not 0 (line 8~12)

```
01: L_INT* gen_s_int(int min,int max,int to_fill) {
02:   unsigned int size, i;
03:   CREST_unsigned_int(size); //sym. var.
04:   if(size> max || size< min) exit(0);
05:   L_INT *n=L_INT_Init(size);
06:   n->len=size;
07:
08:   if(to_fill){// sym. value assignment
09:     for(i=0; i < size; i++) {
10:       CREST_unsigned_int(n->da[i]);}
11:   if(n->da[size-1]==0) exit(0); }
12:   return n;}
```

Test Driver for L_INT_ModAdd()

- L_INT_ModAdd(dest, n1, n2, m)
  - dest := (n1+n2)%m // (6+7)% 10 = 3
- Check (n1+n2)%m == (n2+n1)%m (line 11)
  - Generate 3 symbolic L_INT operands for n1, n2, and m (line 2~4)
  - Generate 2 symbolic L_INT dest and dest2 to store the results

```
01: void test_L_INT_ModAdd() {
02:   L_INT *n1 = gen_s_int(1,4,1),
03:          *n2 = gen_s_int(1,4,1),
04:           *m = gen_s_int(1,4,1),
05:       *dest = gen_s_int(1,4,0), // Do not fill *da
06:      *dest2 = gen_s_int(1,4,0); // Do not fill *da
07:
08:   L_INT_ModAdd(dest,n1,n2,m);
09:   L_INT_ModAdd(dest2,n2,n1,m);
10:   // (n1+n2)%m == (n2+n1)%m
11:   assert(L_INT_Cmp(dest,dest2)==0);}
```

Results

- We tested all 14 functions and all of them violated assertions
  - 7537 TCs generated in 5 mins
  - 1284/1953 branches covered(73.2%)
- L_INT_ModAdd()
  - 831 TCs generated
  - 129 among 150 branches covered (86%)
  - 17 violations of assert (dest == dest2)

Correct behavior

```
------- Test input ---------
n1 : 2 :7f7d4b02 6b702b0d
n2 : 1 :3787923c
m: 1 :777d0295
dest.size=1
dest2.size=1
--------Test output ---------
dest: 1 :539b103d
dest2: 1 :539b103d
L_INT_Cmp(dest,dest2)==0
```

Violation of assert(dest1=dest2)

```
------- Test input ---------
n1 : 4 :777d0295 3787923c 7f7d4b02 6b702b0d
n2 : 3 :513a3234 7d0b4f12 5789fd36
m: 3 :08cae318 61d52574 73331ffa
dest.size=1
dest2.size=1
--------Test output ---------
dest: 3 :00000001 dc5f0f9e a0862e99
dest2: 3 :0874a808 dc5f0f9e a0862e99
L_INT_Cmp(dest,dest2)==-1
test_L_INT_ModAdd: Assertion `result == 0' failed.
```

Observations from these Verification Projects

Main challenge:
- State space explosion problem

1. Expensive computational cost
   - Huge state space (|TCs| = ~ $2^{|exec|}$.)
   - ~90% of time spent by a SMT solver
     - SMT solvers seem good at solving a complex formula but not good at solving millions of similar short formulas
2. Proper selection of symbolic input to reduce state space
   - requires deep knowledge of a target program
3. Build process and runtime environment dependence causes additional burden

Conclusion and Future Work

- Formal verification techniques really work in IT industry !
  - Software model checking and concolic testing detected hidden bugs in industrial embedded software
- To alleviate the limitations of concolic testing
  - Fault-tolerance for distributed concolic testing (SCORE framework [FSE'11b])
  - External function summaries through dynamic invariance generation
  - Develop a new search strategy for fast branch coverage
- Data mining on a huge set of runtime execution information
  - Automated oracle generation through dynamic invariant generation
  - Automated debugging
- Technical papers can be downloaded at http://pswlab.kaist.ac.kr