Concolic Testing on Embedded Software - Case Studies on Mobile Platform Programs

Yunho Kim†, Moonzoo Kim†, Yoonkyu Jang*

†Computer Science Department
Korea Advanced Institute of Science and Technology
kimyunho@kaist.ac.kr, moonzoo@cs.kaist.ac.kr

*Digital Media and Communication Department
Samsung Electronics
yoonkyu.jang@samsung.com

ABSTRACT

Current industrial testing practices often build test cases in a manual manner, which degrades both the effectiveness and efficiency of testing. To alleviate this problem, concolic testing generates test cases that can achieve high coverage in an automated fashion. This paper describes case studies of applying concolic testing to mobile platform C programs that have been developed by Samsung Electronics. Through this work, we have detected new faults in the Samsung Linux Platform (SLP) file manager and security library.

1. INTRODUCTION

In industry, testing is a standard method to improve the quality of software. However, conventional testing methods frequently fail to detect faults in target programs. One reason is that a program can have an enormous number of different execution paths due to conditional and loop statements. Thus, it is infeasible for a test engineer to manually create test cases sufficient to detect subtle errors in specific execution paths. In addition, it is technically challenging to generate effective test cases in an automated manner.

These limitations are manifested in many industrial projects including the mobile platform software for Samsung smartphones. Since the smartphone market requires short time-to-market and high reliability of software, Samsung Electronics decided to apply advanced testing techniques to overcome the aforementioned limitations. As a consequence, Samsung Electronics and KAIST set out to investigate the practical application of *Concolic testing* techniques to the mobile software domain for three years (2010-2012). [1] Concolic (CONCrete + symbOLIC) [10] testing (also known as dynamic symbolic execution [11] or white-box fuzzing [5]) combines concrete dynamic analysis and static symbolic analysis to automatically generate test cases to explore execution paths of a program. A drawback of concolic testing, however, is that the coverage drops if the target program has external binary libraries or complex operations such as pointer arithmetic. Thus, its effectiveness and efficiency must be investigated further through case studies.

This paper reports case studies on the application of CREST [2] (an open-source concolic testing tool) to mobile platform C programs developed by Samsung Electronics. Through the project, we have detected new faults in the Samsung Linux Platform (SLP) file manager and the security library: an infinite loop fault was detected in the SLP file manager and an invalid memory access fault was detected in the security library functions that handle large integers.

[1] In addition to concolic testing, we considered random testing and genetic algorithm-based testing [8]. However, in our experience, the former rarely generates effective test cases for exceptional scenarios and the latter consumes huge time and requires manual tuning for performance.

2. PROJECT BACKGROUND

In this project, the Samsung Linux Platform (SLP) file manager and a security library were selected as target programs, because they are important in the mobile phone products and proper targets for C-based concolic testing tool in terms of size and complexity. The SLP file manager is 18,000 lines long containing 85 functions. The security library consists of 62 functions and is 8,000 lines long.

Our team consisted of one professor, one graduate student, and one senior SQA engineer from Samsung Electronics. The original developers for the SLP file manager and the security library could not join this project due to other release deadlines. In addition, there were no documents on the SLP file manager and the security library. Thus, our team had to understand the target code from scratch, which took almost half the time of the project.

We used CREST [2] as a concolic testing tool in the project for the following reasons. First, we needed an open source concolic testing tool for C programs that can be modified for mobile platform C programs. KLEE [3] and CREST satisfy this requirement. Second, from our experience on other embedded software such as a flash memory device driver, KLEE is an order of magnitude slower than CREST due to the overhead of the LLVM virtual machine and the underlying bit-vector SMT solver. In contrast, CREST inserts probes in a target program to record symbolic path formulas at runtime and uses a linear integer arithmetic SMT solver, which achieves faster testing speed compared to KLEE. Last, we had rich experience with CREST in other industrial case studies [6, 7].

We performed experiments on a VMware 2.5 virtual machine that runs 32 bit Ubuntu 9.04, whose host machines were Windows XP SP3 machines equipped with Intel i5 2.66 GHz and 4 GBytes memory for the SLP file manager, and Intel Core2Duo 2 GHz and 2 GBytes memory for the security library. We could not run Linux on a real machine due to Samsung's security policy for visitors.

3. SLP FILE MANAGER

Figure 1 shows an overview of the SLP file manager. The file manager (FM) monitors a file system and notifies corresponding applications of events in the file system. FM uses an `inotify` system call to register directories/files to monitor. When the directories and files that are being monitored change, the Linux kernel

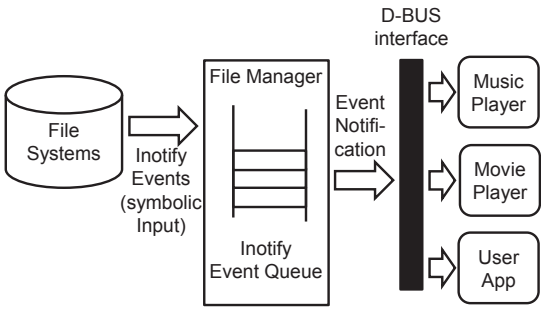

Figure 1: Overview of the SLP file manager

generates `inotify` events and adds these events to an `inotify` queue. FM reads an event from the queue and notifies corresponding programs of the event through a D-BUS inter-process communication interface. For example, when a user adds an MP3 file to a file system, FM notifies a music player to update its playlist automatically. A fault in FM can cause serious problems in SLP, since many applications depend on FM.

3.1 Difficulties of Concolic Testing For FM

Embedded software such as FM often has different development/runtime environments from those of non-embedded software. Due to limited computational power, embedded software has unique characteristics in its development/runtime environments, which causes difficulties for concolic testing. We observed the following difficulties when we applied CREST to FM:

Complex build process: To instrument FM, we had to modify the build process to use a compiler wrapper tool for CREST. The wrapper tool, however, had limitations to handle the build process for embedded software. For performance improvement, a build process for embedded software utilizes complex optimization techniques that are not normally used for non-embedded software. One example was that a build script of FM enforced a specific order of library linking options to optimize the FM binary. The CREST wrapper tool, however, did not keep the order of given options, because the order of options for compilers/linkers does not affect the build process of most non-embedded software. Thus, we had to modify the CREST wrapper tool to keep the order of options. Understanding the optimized build process and modifying the build script took around one fourth of the total project time.

Specialized execution environment: The target platform of FM was Samsung Electronics' own Linux platform based on the ARM architecture. An original test environment was constructed on the Scratchbox [1] ARM simulator, on which CREST runtime modules such as libcrest and Yices [4] could not execute, since only the x86 binary of Yices was available. Thus, with help of the original developers, we ported FM and related SLP libraries to the Scratchbox x86 simulator and applied CREST to FM on the simulator. We could not execute FM on x86 Linux directly, since FM had dependencies on libraries that could run only on Scratchbox.

3.2 Symbolic Inputs

To apply concolic testing, we must specify symbolic variables in a target program, based on which symbolic path formulas are generated at runtime. We specified `inotify_event` as a symbolic input, whose fields are defined as follows:

```
struct inotify_event {
  int wd;          /*Watch descriptor */
  uint32_t mask;   /*Event */
  uint32_t cookie;/*Unique cookie associating events*/
  uint32_t len;    /*Size of 'name' field */
  char     name[];/*Optional null-terminated name */};
```

`wd` indicates the watch for which this event occurs. `mask` contains bits that describe the type of an event that occurred such as `MOVE_IN` (a file moved in the watched directory). `cookie` is a unique integer that connects related events (e.g., pairing `IN_MOVE_FROM` and `IN_MOVE_TO`). `name[]` represents a file/directory path name for which the current event occurs and `len` indicates a length of the file/directory path name. Among the five fields, we specified `wd`, `mask`, and `cookie` as symbolic variables, since `name` and `len` are optional fields. We built a symbolic environment to provide an `inotify_event` queue that contains up to two symbolic `inotify_event`s.

3.3 Results

Two persons of our team worked to apply CREST to FM for five weeks, but only two days per week. KAIST visited Samsung Electronics every week to analyze target code, since Samsung Electronics could not release the target code to KAIST for intellectual property issues. By using CREST, we detected an infinite loop fault in FM in one second. After FM reads an `inotify_event` in the queue, the event should be removed from the queue to process the other events in the queue. For a normal event, the `wd` field of the event is positive. Otherwise, the event is abnormal. We found that FM did not remove an abnormal event from the queue and caused an infinite loop when an abnormal event was added to the queue.

The FM code in Figure 2 handles `inotify_event`s. FM moves `BUF_LEN` bytes from the `inotify_event` queue (`event_queue`) to `buf` (line 1). Then, it processes all events in `buf` through the `while` loop (lines 3-13). Line 7 checks whether or not a current event (`ev`) is normal. If `ev` is normal (line 10), FM sends notifications to corresponding programs (line 11) and removes `ev` by increasing `i` to indicate the next event (line 12). If `ev` is abnormal, line 9 continues the loop *without* increasing `i`. Thus, at the next iteration of the loop, FM reads the same abnormal `ev` again, which causes an infinite loop. The original developers of FM confirmed that this fault was real and fixed it. They had failed to detect this fault for long time, because they had created only a dozen test cases for FM in a manual manner. Those manual test cases did not include test cases with abnormal events that were difficult to generate for a real file system.

After the fault was corrected, CREST generated 138 test cases in five minutes, which covered around 1750 branches among 8152 branches of FM. [2] These test cases did not violate any of the 14 assertions used to check return values of the FM functions for a basic sanity check. Due to the limited time for the project (i.e., 2 days $\times$ 5 weeks), we could not perform more elaborate concolic testing with more assertions and sophisticated symbolic inputs.

```
01:length = read(event_queue, buf, BUF_LEN);
02:i=0;
03:while( i<length ){
04:  struct inotify_event *ev =
05:         (struct inotify_event*)&buf[i];
06:  ...
07:  if (ev->wd<1) {
08:    ERROR("invalid wd : %d",ev->wd);
09:    continue;} //ev is NOT removed from the queue
10:  else if (ev->mask & MOVE_IN){
11:    ... // notify registered programs
12:    i+=ev_len(ev);//ev is removed from the queue
13:  } else if (ev->mask & DELETE){ ...
```

Figure 2: FM code to handle `inotify_event`s

[2]CREST transforms a target program to an equivalent extended version whose branches contain only one atomic condition per branch. The branch coverage data in this paper is based on the extended target program.

4. SECURITY LIBRARY

The security library provides API functions for various security applications on mobile platforms such as SSH (secure shell) and DRM (digital right management). The security library consists of the following three layers:

- *Security functions:*
  This top layer provides security APIs such as AES (advanced encryption standard) or SHA (secure hash algorithm) that are frequently used by applications that handle security operations such as encryption and description.
- *Complex math functions:*
  This middle layer provides complex mathematical functions such as elliptic curve functions and large prime number generators that are used by the security functions.
- *Large integer functions:*
  This bottom layer provides data structures for large integers that cannot be represented by `int` and related operations such as addition and subtraction of two large integers.

4.1 Difficulties of Concolic Testing for the Security Library

We targeted the large integer function layer in the security library, since the security function and complex math function layers were not proper for us to apply concolic testing to for the following reasons. First, these two layers frequently use external binary math functions such as `pow()` and `sqrt()`, which decreases the effectiveness of concolic testing (i.e., resulting in low coverage). Second, the security function and complex math function layers are hard for us to understand due to complex algorithms. Consequently, it would be difficult to specify test oracles and to develop appropriate symbolic inputs for these layers. In contrast to FM, the security library could be compiled and tested on x86 Linux without difficulty.

4.2 Symbolic Inputs

A large integer is represented by the `L_INT` data structure:

```
struct L_INT {
unsigned int size;//Allocated mem size in 32 bits
unsigned int len; //# of valid 32 bit elements
unsigned int *da; //Pointer to the dynamically
                  //allocated data array. da[len-1]
                  //are the most-significant bytes.
unsigned int sign;//0:non-negative, 1: negative }
```

For example, 4294967298 (=$2 + 2^{32}$) can be represented by a `L_INT` data structure that contains `size=3`, `len=2`, `da={2,1,0}` (i.e., $2\times2^{(32\times0)} + 1\times2^{(32\times1)} + 0\times2^{(32\times2)}$), and `sign=0`. Large integers are passed as operands to large integer functions such as `L_INT_ModAdd(L_INT d,L_INT n1,L_INT n2,L_INT m)` that performs `d=(n1+n2)%m`.

To test large integer functions, we built a symbolic large integer generator that returns a symbolic large integer `n` (line 12) as shown in Figure 3. Lines 3-5 allocate memory for `n` (line 5). Line 3 declares the `size` of `n` as a symbolic variable of `unsigned char` type. Note that line 4 enforces a constraint on `size` such that `min`$\leq$`size`$\leq$`max`. Without this constraint, `size` can be 255, which will generate unnecessarily many large integers, since the number of generated large integers increases as the `size` increases. Line 5 allocates memory for `n` using `L_INT_Init()`. For simple analysis, we assume that `len==size` (line 6). Lines 9-10 fill out a data array of `n`, if necessary (line 8). For example, we do not need to fill out a data array for `d` that is a result of `L_INT_ModAdd(L_INT d,...)`. Since we assume that

```
01:L_INT* gen_s_int(int min,int max,int to_fill) {
02:  unsigned char size, i;
03:  CREST_unsigned_char(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;}
```

Figure 3: Symbolic large integer generator

`size==len`, we do not allow the most-significant bytes to be 0 (line 11).

Using `gen_s_int()`, we developed test drivers for all 14 large integer functions. For example, a test driver for `L_INT_ModAdd()` is described in Figure 4, which generates symbolic large integers whose values are between $2^{(32\times1)} - 1$ and $2^{(32\times4)} - 1$ (lines 2-4). [3] `dest` and `dest2` do not need to have symbolic values (lines 5-6), since they will be assigned new values by `L_INT_ModAdd()`. This test driver checks whether or not `(n1+n2)%m == (n2+n1)%m` at line 11.

```
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), // to_fill=0
06:  *dest2=gen_s_int(1,4, 0); // to_fill=0
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);}
```

Figure 4: Test driver for `L_INT_ModAdd()`

4.3 Results

Two persons of our team worked to apply CREST to the security library for five weeks, but only one day per week. We inserted 40 assertions in the 14 large integer functions and found that all 14 large integer functions violated some assertions. CREST generates 7537 test cases for the 14 large integer functions in five minutes, that cover 1284 of 1753 branches in the target functions. For example, `test_L_INT_ModAdd()` generated 831 test cases that covered 129 of 150 branches in `L_INT_ModAdd()`. 17 of the 831 test cases violated the `assert()` at line 11 of Figure 4.

We analyzed `L_INT_ModAdd(L_INT d,L_INT n1,L_INT n2,L_INT m)` and found that this function did not check the `size` of `d`. Thus, if the `size` of `d` is smaller than `(n1+n2)%m`, this function writes beyond the allocated memory for `d`, which may corrupt `d` later by other memory writes. To analyze the fault further, we checked all functions in the security function and complex math function layers that invoke `L_INT_ModAdd()` and found that those functions set the `size` of `d` as equal to `n1` and pass `d` to `L_INT_ModAdd()`. We suspect that this fault has not been detected, because `(n1+n2)%m < m` and `m` is usually smaller than `n1` in most mathematical formulas used in security applications. Furthermore, many security algorithms assume that the bit size of operands and the bit size of the result are fixed and same. However, the large integer library should handle exceptional cases properly, since there is no such guarantee in general. Failure to handle such

[3] 4 is the smallest number for the `len` that can represent all possible relations between `len`s of `d`, `n1`, `n2`, and `m`. For example, `1=len(m)<len(d)<len(n1)<len(n2)=4` where `len(x)` is the `len` of a large integer `x`.

exceptional scenarios can cause serious problems and the original developers confirmed their mistakes.

5. LESSONS LEARNED

5.1 Covering Exceptional Scenarios

A main reason why the original developers could not detect the faults discovered in this work is that these faults cause errors only in *corner-case/unexpected scenarios*. For example, the fault in the SLP file manager triggers errors only when a file system error occurs (i.e., when an abnormal event is generated). It is very difficult for a human engineer to detect faults that are manifest only in exceptional scenarios through manual testing. This is because developers tend to concentrate on the expected behaviors of the target programs and often miss testing unexpected behaviors in a systematic manner. Another reason is that manual test case generation consumes a large amount of time and there can be too many exceptional test cases.

Concolic testing aims to automatically generate test cases that cover all possible execution paths including unexpected execution scenarios of a target program. Thus, concolic testing can test unexpected execution scenarios in an effective and efficient manner. Through this work, we demonstrated that concolic testing could detect corner-case faults in industrial software successfully.

5.2 A Concolic Testing Approach Suitable for Embedded Software

Through this work, we identified issues to consider for successful application of concolic testing to embedded software that runs on specialized platforms (see Section 3.1). First, a concolic testing approach that instruments a target source code is more appropriate for embedded software than virtual machine based approach, since the former is lighter than the latter in terms of porting efforts. A virtual machine based approach [3, 11, 9] has an advantage in terms of applicability; it can be applied to target programs in various high-level languages, since the virtual machine works on low-level bytecodes (e.g., LLVM bit-code, Java bytecode). However, for an embedded target program, a virtual machine/emulator of a specific target OS/HW platform (e.g., SLP or Samsung Bada OS on ARM architecture) should be modified to add concolic testing capability, which can cause huge overhead or may not be feasible. Second, it is advantageous to separate the symbolic path formula construction/solving mechanism from the runtime information extraction mechanism (i.e., probes). Due to the limited computing power of an embedded target platform, heavy computing activities (the former) need to run on a powerful machine while the probes (the latter) run on the embedded target platform and communicate with the former. In this regard, an instrumentation based concolic testing approach has benefits for embedded software.

5.3 Limitations of CREST

As noted in Sections 3.1 and 4.1, concolic testing in general has limitations. We also noted specific limitations in CREST as follows. CREST uses a linear integer arithmetic (LIA) SMT solver to solve generated symbolic path formulas. Thus, CREST cannot handle full ANSI C semantics, especially those related to bit-level representations. The first limitation we observed was that CREST did not support bit-wise operators (`&`, `|`, «, etc) in a target program. If a branch condition contains a bit-wise operator, that branch condition cannot be negated to generate a new test case that will execute an uncovered path. For example, the SLP file manager used bit-wise operators to check the type of an `inotify` event (see line 10 of Figure 2). As a workaround, we replace bit-wise operators with functions that contain loops to handle each bit of the operands explicitly (we modified 11 lines in FM for this purpose). The second limitation was that CREST could not analyze integer overflow semantics of C programs as a default. The security library utilizes integer overflow explicitly (e.g., large integer functions contain `if(x+y >= x) {...} else {...}` where `x` and `y` are `unsigned int` types). In C semantics, `x+y>=x` is always true for `unsigned int x` and `y` except when integer overflow occurs (e.g., when `x`=$2^{32}-1$ and `y`=2). However, CREST cannot generate a test case representing this integer overflow scenario, since it generates symbolic path formulas in LIA only.

6. CONCLUSION AND FUTURE WORK

We reported our case studies to apply CREST on the SLP file manager and the security library that were developed by Samsung Electronics. We found new faults in both programs, which were difficult to find through manual testing, since human engineers often miss such exceptional scenarios. Samsung Electronics and KAIST will continue collaboration for the next two years to overcome the limitations of CREST observed in this work (see Section 5.3) by modifying CREST to generate and solve symbolic path formulas in bit-vector representations.

Acknowledgments

This work was supported by Samsung Electronics, the ERC of Excellence Program of Korea Ministry of Education, Science and Technology(MEST) / National Research Foundation of Korea) (Grant 2011-0000978), and Basic Science Research Program through the NRF funded by the MEST (2010-0005498).

7. REFERENCES

[1] Scratchbox - cross-compilation toolkit. `http://www.scratchbox.org/`.

[2] J. Burnim. CREST - automatic test generation tool for C. `http://code.google.com/p/crest/`.

[3] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.

[4] B. Dutertre and L. Moura. A fast linear-arithmetic solver for DPLL(T). In *CAV*, 2006.

[5] P. Godefroid, M. Y. Levin, and D. Molnar. Automated whitebox fuzz testing. In *NDSS*, 2008.

[6] M. Kim and Y. Kim. Concolic testing of the multi-sector read operation for flash memory file system. In *SBMF*, 2009.

[7] M. Kim, Y. Kim, and Y. Choi. Concolic testing of the multi-sector read operation for flash storage platform software. *Formal Aspects of Computing*. to be published.

[8] Roy P. Pargas, Mary Jean Harrold, and Robert R. Peck. Test-data generation using genetic algorithms. *Journal of Software Testing, Verification, and Reliability*, 9:263–282, September 1999.

[9] C. Pasareanu, P. Mehlitz, D. Bushnell, K. Gundy-burlet, M. Lowry, S. Person, and M. Pape. Combining unit-level symbolic execution and system-level concrete execution for testing nasa software. In *ISSTA*, 2008.

[10] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *ESEC/FSE*, 2005.

[11] N. Tillmann and W. Schulte. Parameterized unit tests. In *ESEC/FSE*, 2005.