2012 IEEE Fifth International Conference on Software Testing, Verification and Validation

Industrial Application of Concolic Testing on Embedded Software: Case Studies

Moonzoo Kim, Yunho Kim
Department of Computer Science
KAIST
South Korea
moonzoo@cs.kaist.ac.kr,kimyunho@kaist.ac.kr

Yoonkyu Jang
Samsung Electronics
South Korea
yoonkyu.jang@samsung.com

***Abstract*—Current industrial testing practices often build test cases in a manual manner, which is slow and ineffective. To alleviate this problem, concolic testing generates test cases that can achieve high coverage in an automated fashion. However, due to a large number of possible execution paths, concolic testing might not detect bugs even after spending significant amount of time. Thus, it is necessary to check if concolic testing can detect bugs in embedded software in a practical manner through case studies. This paper describes case studies of applying the concolic testing tool CREST to embedded C applications. Through this project, we have detected new faults in the Samsung Linux Platform (SLP) file manager, Samsung security library, and busybox `ls`.**

I. Introduction

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. Since the consumer electronics market requires short time-to-market and high reliability, 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 embedded software for three years (2010-2012).

Concolic (CONCrete + symbOLIC) [33] testing (also known as dynamic symbolic execution [22], [9], [36] or white-box fuzzing [14]) 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 operations [29]. Thus, its effectiveness (in terms of bug detection capability) and efficiency (in terms of time taken to detect a bug) must be investigated further through case studies.

This paper describes our case studies on the application of CREST [5] (an open-source concolic testing tool for C programs) to embedded C programs used in the products of Samsung Electronics including Samsung Linux Platform (SLP) file manager, Samsung security library, and busybox `ls`.[1] Through the project, we have detected new faults in the aforementioned applications. For example, we detected an infinite loop bug in the SLP file manager, invalid memory access bug in in the Samsung security library, and four command-line option related bugs in busybox `ls` that has been undetected for several years in spite of being used by millions of users.

The organization of the paper is as follows. Section II overviews this testing project. Section III explains related work on concolic testing tools and related case studies. Section IV overviews CREST. Sections V to VII describe testing experiments targeting SLP file manager, Samsung security library, and busybox `ls` respectively. Section VIII summarizes the lessons learned from the project. Section IX concludes this paper with future work.

II. Project Background

The case studies described in this paper were conducted as a part of the three-year project to apply concolic testing approach to improve the quality of the consumer electronics products of Samsung Electronics. Through the case studies, we aim to evaluate the effectiveness of concolic testing approach as a method to improve quality of embedded software. Samsung Linux Platform (SLP) file manager, Samsung security library, and busybox `ls` were selected as target programs, because they are important for the products of Samsung Electronics and written in C with modest size and complexity (see Sections V to VII).

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 Samsung security library could not join this project. In addition, there were no documents on the SLP file manager and the Samsung security library. Thus, our team had to

[1]Preliminary testing result on the SLP file manager and the Samsung security library were reported in a 4 page short paper [21].

978-0-7695-4670-4/12 $26.00 © 2012 IEEE
DOI 10.1109/ICST.2012.51

understand the requirements and target code from scratch, which took almost half the time of the project. In contrast, we could obtain detailed requirement specification of busybox `ls` from IEEE Std 1003.1 [35] that describes detailed requirements for UNIX utilities such as `ls`.

We used CREST [5] as a concolic testing tool in the project for the following reasons. First, we needed an open source concolic testing tool for C programs to control testing experiments in a refined manner, and analyze testing results in deep technical level by obtaining various internal information. KLEE [7] 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 complex symbolic path formulas supporting 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 [17], [21], [18].

We performed the 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 Samsung security library and busybox `ls`. We could not run Linux on a real machine due to Samsung's security policy for visitors.

III. Related Work

A. Concolic Testing Tools

The core idea behind concolic testing is to obtain symbolic path formulas from concrete executions and solve them to generate test cases by using constraint solvers (see Section IV-A). Various concolic testing tools have been implemented to realize this core idea (see [28] for a survey). We can classify the existing approaches into the following three categories based on how they extract symbolic path formulas from concrete executions.

1) Static instrumentation of target programs: The concolic testing tools in this group instrument a source program to insert probes that extract symbolic path formulas from concrete executions at run-time (Section IV-A). Many concolic testing tools adopt this approach because it is relatively simple to implement and, consequently, convenient when attempting to apply new ideas in tools. In this group, CUTE [33], DART [13], and CREST [6] operate on C programs, while jCUTE [32] operates on Java programs.

2) Dynamic instrumentation of target programs: The concolic testing tools in this group instrument a binary program when it is loaded into memory (i.e., through a dynamic binary instrumentation technique [25]). Thus, even when the source code of a program is not available, its binary can be automatically tested. In addition, this approach can detect low-level failures caused by a compiler, linker, or loader. SAGE [14] is a concolic testing tool that uses this approach to detect security bugs in x86-binaries.

3) Instrumentation of virtual machines: The concolic testing tools in this group are implemented as modified virtual machines on which target programs execute. One advantage of this approach is that the tools can exploit all execution information at run-time, since the virtual machine possesses all necessary information. PEX [36] targets C# programs that are compiled into Microsoft .Net binaries, KLEE [7] targets LLVM [23] binaries, and jFuzz [15] targets Java bytecode on top of Java PathFinder [27].

B. Concolic Testing Case Studies

Concolic testing has been applied to detect bugs in various target applications such as database applications [12], [26], web application servers [3], [31], web application client [16], source code control client [24], mobile sensor network [30], network card device driver [8], and flash memory storage platform [18].

Emmi et al. [12] applied jCute [32] to generate test cases for database applications which make calls to a database through an API to execute SQL queries. Pan et al. [26] also applied the concolic testing to database applications RiskIt and UnixUsage by using PEX [36]. Artzi et al. [3] developed the concolic testing tool, Apollo, to automatically generate test cases for server-side web applications written in PHP. Kiezun et al. [16] utilized Apollo to generate test cases which trigger the SQL injection and cross-site scripting security vulnerabilities in web programs written in PHP. Saxena et al. [31] applied concolic testing to client-side web applications such as FaceBook Chat written in JavaScript. Marri et al. [24] applied PEX [36] to CodePlex, which is a source code control client written in C#. Sasnauskas et al. [30] developed a debugging environment KleeNet to detect bugs in the μIP TCP/IP protocol stack in the Contiki OS [10]. Chipounov et al. [8] developed S2E, which is a x86 binary program analysis platform based on QEMU [4] and KLEE and applied S2E to test network card device driver on the windows platform. Kim et al. [18] applied CREST [6] to Samsung flash storage platform. [18] described the importance of an environment model for the target program in terms of testing speed and conducted performance analysis based on the characteristics of generated symbolic path formulas and reduction techniques on the symbolic path formulas.

However, most related case studies mainly describe the authors' own algorithms to utilize concolic testing to detect bugs, not detailed explanation on user efforts and obstacles to apply concolic testing to real world target programs. In contrast, we describe the detailed testing experiments and lessons learned from the case studies (see Section VIII).

IV. Overview of CREST

CREST [6] is an open source concolic testing tool for C programs. To build symbolic path formulas from concrete execution paths, CREST inserts probes in a target C program before a target program executes. CREST supports linear integer arithmetic (LIA) formula and uses Yices [11] to solve symbolic path formulas to obtain next test inputs.

A. Concolic Testing Process

This section presents an overview of the original (non-distributed) concolic testing process that performs static instrumentation of a target program to extract symbolic path formulas. The concolic testing process proceeds via the following steps:

1. Declaration of symbolic variables. Initially, a user specifies which variables should be handled as symbolic variables, based on which symbolic path formulas are constructed.[2]

2. Instrumentation. A target source program is statically instrumented with probes, which record symbolic path conditions from a concrete execution path when the target program is executed. For example, at each conditional branch, a probe is inserted to record the branch condition/symbolic path condition; then, the instrumented program is compiled into an executable binary file.

3. Concrete execution. The instrumented binary is executed with given input values. For the first execution of the program, initial input values are assigned randomly. From the second execution onwards, input values are obtained from Step 6.

4. Obtain a symbolic path formula ϕ_i. The symbolic execution part of the concolic execution collects symbolic path conditions over the symbolic input values at each branch point encountered for along the concrete execution path for a test case tc_i. Whenever each statement s of the target program is executed, a corresponding probe inserted at s updates the map of symbolic variables if s is an assignment statement, or collects a corresponding symbolic path condition, c, if s is a branch statement. Thus, a symbolic path formula ϕ_i is built at the end of the ith execution as $c_1 \wedge c_2 ... \wedge c_n$ where c_n is the last path condition executed and c_k is executed earlier than c_{k+1} for all $1 \leq k < n$.

5. Generate a new symbolic path formula ψ_i. Given a symbolic path formula ϕ_i obtained in Step 4, to obtain the next input values, ψ_i is generated by negating one path condition c_j and removing subsequent path conditions (i.e., $\psi_i = c_1 \wedge c_2 ... \wedge \neg c_j$). For example, if a depth first search (DFS) strategy is used, as it often is, to explore the symbolic path formula, then c_j is the last symbolic path condition in ϕ_i whose negated path condition has not been executed previously. If ψ_i is unsatisfiable, another path condition $c_{j'}$ is negated and subsequent path conditions are removed until a satisfiable path formula is found. If there are no further new paths to try, the algorithm terminates.

6. Select the next input values tc_{i+1}. A constraint solver such as a Satisfiability Modulo Theory (SMT) solver [34] generates a model that satisfies ψ_i. This model determines the next concrete input values to try (i.e., tc_{i+1}), and the concolic testing procedure iterates from Step 3 using these input values.

[2]Proper selection of symbolic input is important for effective and efficient testing, since search space is decided based on the symbolic input and the search space is usually very large to cover completely. This step requires human expertise on a target domain. For example, Kim et al. [19] demonstrated that testing strategies on selecting symbolic input affects bug detection capability through a case study on the open source `libexif` library.

```
01:int main() {
02:  int x, y,z, max_num=0;
03:  CREST_int(x); // Declaration of x, y, z
04:  CREST_int(y); // as symbolic integer
05:  CREST_int(z); // variables
06:
07:  if(x >= y) {
08:    // SYM_COND(x,y, ">=");
09:    if(y >= z) {
10:      // SYM_COND(y,z, ">=");
11:      max_num = x;
12:    } else {
13:      // SYM_COND(y,z, "<");
14:      if (x >= z){
15:        // SYM_COND(x,z, ">=");
16:        max_num = x;
17:      } else {
18:        // SYM_COND(x,z, "<");
19:        max_num = z;
20:      }
21:    }
22:  } else { ...}
23:  printf("%d is the largest number among\
24:          {%d,%d,%d}", max_num, x,y,z);
25:  // SMT_Solve();
26:}
```

Figure 1. Example used to illustrate concolic testing

B. Example of Concolic Testing Process

We illustrate this process through an example involving Figure 1, which returns the largest number from three given integers `x`, `y`, and `z`.

1. Declaration of symbolic variables. A user declares `x`, `y`, and `z` as symbolic integer variables by using `CREST_int()` (lines 3-5).

2. Instrumentation. A concolic testing tool inserts a probe to record a corresponding path condition at each `then` branch in an automated manner. Similarly, at each `else` branch, a probe is inserted to record a corresponding path condition. In Figure 1, probes inserted through instrumentation are shown as comments. For example, at line 10, `SYM_COND(y,z,">=")` is inserted to record path condi-

tion $y >= z$. Similarly, `SYM_COND(y,z,"<")` is inserted at line 13 to record path condition $y < z$.

3. Concrete execution. Initial input values for the symbolic variables are randomly chosen. We assume that `x`, `y`, and `z` are assigned 1, 1, and 0 as initial random values, respectively (i.e., $tc_1 =< 1, 1, 0 >$). Then, the instrumented target program executes lines 2-11 and lines 23-26.

4. Obtain a symbolic path formula ϕ_i. During the concrete execution of lines 2-11, the probes record two symbolic path conditions $x >= y$ and $y >= z$ through `SYM_COND(x,y,">=")` (line 8) and `SYM_COND (y,z,">=")` (line 10) respectively. Thus, the symbolic formula $\phi_1 = (x >= y) \wedge (y >= z)$ is obtained for the first iteration.

5. Generate a new symbolic path formula ψ_i. If a DFS algorithm is used, ψ_1 is $(x >= y) \wedge \neg(y >= z)$.

6. Select the next input values. At line 25, the target program finishes its first iteration and invokes a constraint solver to solve ψ_1. Suppose that an SMT solver solves ψ_1 and generates 1, 1, and 2 for `x`, `y`, and `z` as a solution (i.e., $tc_2 =< 1, 1, 2 >$). Then, the target program starts the second iteration with these values, and the entire process from Step 3 is repeated.

V. SLP File Manager

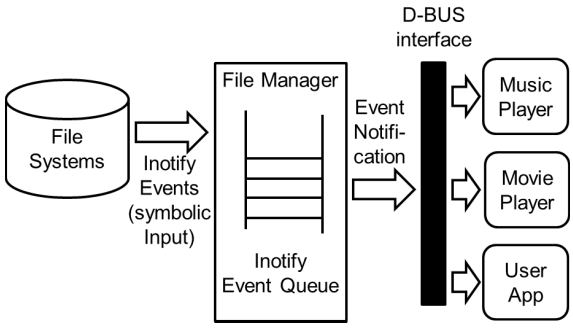

Figure 2. Overview of the SLP file manager

Figure 2 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 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. The SLP file manager is 18000 lines long containing 85 functions.

A. Difficulties of Concolic Testing For SLP FM

Embedded software such as FM often has different development and runtime environments from those of non-embedded software. Due to limited computational power, embedded software has unique characteristics in its development/build-process and runtime environments, which causes difficulties for concolic testing. Since concolic testing is involved with build-process and runtime environment (see Section IV), these tool problems can be critical in industrial setting. We observed the following difficulties when we applied CREST to FM:

1) 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.

2) 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 [2] ARM simulator, on which CREST runtime modules such as libcrest and Yices [11] could not execute, since only the x86 binary of Yices was available. Thus, 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.

B. 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 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.

C. Results

Two persons of our team worked to apply CREST to FM for ten days. 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. We added 14 assertions to check return values of the FM functions for a basic sanity check. 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. We found that FM did not remove an abnormal event whose `wd` is zero or negative from the queue and caused an infinite loop when an abnormal event was added to the queue.

The FM code in Figure 3 handles `inotify_event`. 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 is 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. These 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.[3] These test cases did not violate any of the 14 assertions. Due to the limited time for the project (i.e., ten days), 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
10:  else if (ev->mask & MOVE_IN){
11:    ... //notify registered programs
12:    i+=ev_len(ev);//ev is removed
13:  } else if (ev->mask & DELETE){...
```

Figure 3. FM code to handle `inotify_events`

VI. Samsung Security Library

The Samsung 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.

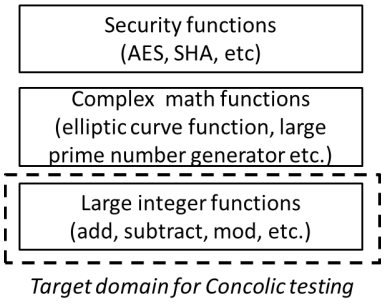

Figure 4. Structure of the Samsung security library

The security library consists of 62 functions and is 8000 lines long.

A. 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). This is because concolic testing cannot solve a symbolic path formula that

[3]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.

contains binary library calls on symbolic variables. 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.

B. Symbolic Inputs

A large integer is represented by the `L_INT` data structure in Figure 5.

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

Figure 5. Large integer data structure

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

C. Test Drivers

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 7, which generates symbolic large integers whose values are between

```
01:L_INT* gen_s_int(min,max,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 6. Symbolic large integer generator

$2^{(32\times1)}-1$ and $2^{(32\times4)}-1$ (lines 2-4).[4] `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 7. Test driver for `L_INT_ModAdd()`

D. Results

Two persons of our team worked to apply CREST to the security library for ten days. 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 (73%) in the target functions. [5]

For example, `test_L_INT_ModAdd()` generated 831 test cases that covered 129 of 150 branches (86%) in `L_INT_ModAdd()`. 17 of the 831 test cases violated the `assert()` at line 11 of Figure 7.

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

[4] 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`.

[5] Large integer functions contain several non-linear arithmetic operations, which prevents CREST from reaching high branch coverage. In addition, the reported branch coverage is based on the extended target program, which is equivlanet to the condition/decision coverage on the original target program.

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

VII. BUSYBOX LS

Busybox [1] is a one-in-all command-line utility that combines tiny versions of many common UNIX utilities such as ls and grep. Busybox is written for minimal size and limited resources. Also, it is modular to customize busybox to be adopted in various embedded systems.

We selected ls as our target utility among the busybox utilities, since ls is the most frequently used utility in busybox and used/tested by millions of users. Thus, we can evaluate the effectiveness of a concolic testing approach to improve the reliability of field-proven application furthermore. We targeted busybox 1.17.0, the latest version at the time of the project. Busybox ls consists of 16 functions and it is 1100 lines long in C.

To apply concolic testing effectively, we reviewed the POSIX specification (IEEE Std 1003.1 [35]) on ls that describes expected behaviors of ls with various command-line options and environment conditions. For example, the POSIX specification requires that ls should print out the '@' symbol right after a symbolic link file $slnk$ when -F option is given (i.e., ls should obtain the status of $slnk$, not the status of the file $slnk$ points to). Based on the POSIX specification for various options, we declared related variables of busybox ls as symbolic and inserted assert() statements correspondingly.

A. Difficulties of Concolic Testing for busybox ls

Since CREST can analyze only linear integer arithmetic (LIA) expression symbolically, CREST cannot test a target C program effectively that utilizes bit-wise operations (e.g., &, |, <<) heavily. We found that busybox ls stores command-line options in a 32 bit unsigned integer variable opt as a bit sequence. In other words, each bit in opt indicates if a corresponding command-line option is on or off. For example, the 21st bit of opt indicates -F option that appends the indicator symbol '@' to a file name to display.

Thus, we had to develop a work-around solution to analyze bit-wise operations symbolically. We modified busybox ls by replacing operations on a bit sequence with operations on an integer array each element of which corresponds to each bit of the bit sequence as shown in Figure 8. For example, we converted opt into int opt_list[32] by bs2ia(opt, opt_list) and replaced & with bit_and().

```
//a number of bits used in a bit sequence
int BITSIZE;

//a bit sequence bs->an integer array ia
static int* bs2ia(unsigned bs,int *ia){
  int i;
  for(i=BITSIZE-1;i>=0;i--,bs=bs/2)
    ia[BITSIZE-i-1] = bs%2;
  return ia; }

// Replacement of bitwise &
static int bit_and(int *a, int *b) {
  int i;
  for(i=0 ; i<BITSIZE ; i++)
    if(a[i]!=0 && b[i]!=0) return 1;
  return 0; }

// Replacement of bitwise |
static int bit_or(...) {...}
...
```

Figure 8. Transformation of operations on a bit sequence into operations on an integer array

B. Symbolic Inputs

Since a main task of busybox ls is to display the status/meta-data of files/directories, we declared stat (defined in sys/stat.h) data structure symbolically that represents a status of a file/directory. In other words, we declared all 13 members of stat as symbolic variables such as mode_t st_mode (mode of a file), ino_t st_ino (inode number), and uid_t st_uid (user ID of file).

In addition, we made a symbolic directory environment that has two files, whose file name lengths are declared symbolically but less than nine characters (for example, "aa" and "b"). Also, we feed these two file names as command-line arguments to busybox ls by setting argv[] and argc correspondingly.

To apply these symbolic settings, we modified static struct dnode * my_stat(const char *fullname, const char *name, int force_follow) that returns status of a file pointed by name in dnode structure; modified my_stat() invokes sym_stat() and sym_lstat() to return symbolic status of files instead of stat() and lstat() that access status of a concrete file.

Furthermore, we declared opt_list[32] as symbolic variables, which corresponds to opt that is a bit sequence indicating a list of given command-line options. Thus, we analyzed all possible combinations of command-line options through concolic testing.

C. Test Oracles

For each command-line option, we inserted corresponding `assert()` as test oracles. For example, for `-F`, we inserted the following `assert()` in `my_stat()` (line 292 in `ls.c`.

```
assert(!(opt_list[21] && !opt_list[23])||
!((all_fmt & FOLLOW_LINKS)||force_follow))
```

`opt_list[21]` and `opt_list[23]` indicate `-F` and `-L` options respectively. `-L` forces busybox `ls` to follow a symbolic link *slnk* to display the status of the file *slnk* points to (call the file *linked*) instead of that of *slnk*. `all_fmt` indicates how to display files and `FOLLOW_LINKS` is a constant mask to display *linked* instead of *slnk*. `force_follow` is a flag to force `my_stat()` to return the status of *linked*. Thus, the `assert()` claims that if `-F` without `-L` is given, `all_fmt` should *not* indicate to display *linked* and `force_follow` should be false.

In a similar manner, we defined and inserted 15 `assert()` statements.

D. Results

Two persons of our team worked to apply CREST to busybox `ls` for ten days. Most of time were spent to define `assert()` statement by understanding the IEEE Std 1003.1 specification and identifying corresponding segments in the busybox `ls` code.

CREST detected the following four bugs in 15 minutes by generating 13000 test cases and covered 68.6% of the branches in busybox `ls` (188 out of 274 branches). We reported these bugs to the busybox development team. All of these bugs were confirmed by the busybox developers and fixed in busybox 1.19.

1) `-F` does not show the status of slnk itself, but the file slnk points to: This bug was detected through the violation of the `assert()` statement in Section VII-C. The bug was caused because the last parameter to `my_stat()` in `ls_main()` (at line 1074 of `ls.c`) was incorrect:

```
1:cur = my_stat(*argv, *argv,
2:!(all_fmt & (STYLE_LONG|LIST_BLOCKS)));
```

With `-F` without `-L`, the last parameter of `my_stat()` at line 2 becomes true, since `STYLE_LONG` and `LIST_BLOCKS` are constant masks to display files in a long format and with a block size respectively and none of them are enabled by `-F`. Then, `force_follow` (the last formal parameter of `my_stat()`) becomes true and `my_stat()` obtains the status of *linked* instead of *slnk*, which violates the requirement for `-F` option. For example, busybox `ls -F -i` *slnk* does not display the '`@`' mark for *slnk* and displays the inode number of *linked*.

2) `-i` does not show space between adjacent two columns: `-i` forces busybox `ls` to display inode number of each file. This bug was detected from the violation of the following `assert()` statement inserted in `showfiles()` (line 857 in `ls.c`).

```
assert(nexttab >= column + tabstops);
```

`nexttab` is the start position of the next column, `column` is the end position of the previous column, and `tabstops` is the number of spaces between columns. This bug was caused because `showfiles()` assumed that an inode number can be displayed in eight digits (i.e., inode number $\leq$ 99999999), which is *not* true. Thus, busybox `ls -i aa b` displays no space between the adjacent columns, if `b` has a large inode number (i.e., nine digits).

3) `-s` does not show space between adjacent two columns: `-s` forces busybox `ls` to display block size of each file. This bug was caused by the similar reason of the bug in Section VII-D2; `showfiles()` assumed that maximum block size of a file can be displayed in five digits, which is *not* true, either.

4) `-n` does not show user id and group id in a numeric format: This bug was detected from the violation of the following `assert()` statement inserted in `ls_main()` (line 1166 of `ls.c`) where `opt_mask[7]` is `-n` that forces busybox `ls` to display user id and group id in a numeric format.

```
assert(!opt_mask[7] ||
    (all_fmt & LIST_ID_NUMERIC));
```

VIII. Lessons Learned

A. 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.

B. Concolic Testing Approach 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 V-A).

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 [7], [36], [27] 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 bitcode, 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 requires huge effort 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.

C. Limitations of CREST

As noted in Sections V-A, VI-A, and VII-A, 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 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 unexplored path. For example, busybox `ls` used bit-wise operators to check given command-line options. As a workaround, we replace bit-wise operators with functions that contain loops to handle each bit of the operands explicitly as shown in Section VII-A. 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.

D. Importance of Detailed Requirement Specifications

A main reason that we could detect the four hidden bugs in busybox `ls` that had not been detected for several years by millions of users, is that we had a detailed requirement specification for busybox `ls`. Based on the detailed requirements (i.e., IEEE Std 1003.1 [35]), we declared relevant command line options as symbolic input and inserted a corresponding `assert()` for each option to detect bugs. In contrast, we could detect only an infinite loop bug in the SLP file manger, since we did not have a detailed requirement specification; consequently, less elaborated symbolic analysis was conducted. Thus, we need a detailed requirement specification for effective concolic testing.

However, most software development projects in IT industry are under time pressure due to heavy competition in the market and often do not have a detailed written requirement specification. Thus, it can be also practically useful to develop a testing method to focus on detecting runtime failure bugs such as null-pointer dereference, divide-by-zero, and out-of-bound memory access, which can be detected without explicit requirement specifications.

IX. Conclusion and Future Work

We reported our case studies to apply CREST on the SLP file manager and the Samsung security library that were developed by Samsung Electronics. In addition, we applied CREST to busybox `ls` that is known as a field-proven reliable utility. As results, we detected new bugs in the all target applications, which were difficult to find through manual testing, since human engineers often miss such exceptional scenarios. Thus, we confirmed that concolic testing approach can improve the quality of industrial embedded applications, even a field-proven reliable one such as busybox `ls`.

Through the case studies, however, we also observed several limitations of a current concolic testing approach and a tool (i.e., CREST). Samsung Electronics and KAIST will continue collaboration to overcome such limitations (see Section VIII-C) [20], [19].

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).

References

[1] Busybox home page. http://www.busybox.net/.

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

[3] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. D. Ernst. Finding bugs in dynamic web applications. In *International Symposium on Software Testing and Analysis*, 2008.

[4] F. Bellard. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference Freenix Track*, 2005.

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

[6] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. Technical Report UCB/EECS-2008-123, EECS Department, University of California, Berkeley, Sep 2008.

[7] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Operating System Design and Implementation*, 2008.

[8] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: A platform for in-vivo multi-path analysis of software systems. In *ASPLOS*, 2011.

[9] R. A. DeMillo and A. J. Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering (TSE)*, 17(9):900–910, 1991.

[10] A. Dunkels. Full tcp/ip for 8-bit architectures. In *MobiSys*, 2003.

[11] B. Dutertre and L. Moura. A fast linear-arithmetic solver for DPLL(T). In *Computer Aided Verification*, 2006.

[12] M. Emmi, R. Majumdar, and K. Sen. Dynamic test input generation for database applications. In *International Symposium on Software Testing and Analysis*, 2007.

[13] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Programming Language Design and Implementation*, 2005.

[14] P. Godefroid, M. Y. Levin, and D. Molnar. Automated whitebox fuzz testing. In *Network and Distributed Systems Security*, 2008.

[15] K. Jayaraman, D. Harvison, V. Ganesh, and A. Kiezun. jFuzz: A concolic whitebox fuzzer for Java. In *NASA Formal Methods Symposium*, 2009.

[16] A. Kiezun, P. J. Guo, k. Jayaraman, and M. D. Ernst. Automatic creating of SQL injection and Cross-Site scripting attacks. In *International Conference on Software Engineering*, 2009.

[17] M. Kim and Y. Kim. Concolic testing of the multi-sector read operation for flash memory file system. In *Brazilian Symposium on Formal Methods*, 2009.

[18] M. Kim, Y. Kim, and Y. Choi. Concolic testing of the multi-sector read operation for flash storage platform software. *Formal Aspects of Computing*, 24(2), 2012.

[19] M. Kim, Y. Kim, and Y. Kim. Industrial application of concolic testing approach: A case study on libexif by using CREST-BV and KLEE. In *International Conference on Software Engineering*, 2012. under review.

[20] M. Kim, Y. Kim, and G. Rothermel. A scalable distributed concolic testing approach: An empirical evaluation. In *International Conference on Software Testing, Verification and Validation (ICST)*, 2012.

[21] Y. Kim, M. Kim, and Y. Jang. Concolic testing on embedded software - case studies on mobile platform programs. In *European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE) Industrial Track*, 2011.

[22] B. Korel. Automated software test data generation. *IEEE Transactions on Software Engineering (TSE)*, 16(8):870 –879, aug 1990.

[23] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Intl. Symp. on Code Generation and Optimization*, 2004.

[24] M. Marri, T. Xie, N. Tillmann, J.de Halleux, and W. Schulte. An empirical study of testing file-system-dependent software with mock objects. In *Automation of Software Test*, 2009.

[25] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Programming Language Design and Implementation*, 2007.

[26] K. Pan, X. Wu, and T. Xie. Generating program inputs for database application testing. In *Automated Software Engineering*, 2011.

[27] 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 *International Symposium on Software Testing and Analysis*, 2008.

[28] C. Pasareanu and W. Visser. A survey of new trends in symbolic execution for software testing and analysis. *Software Tools for Technology Transfer*, 11(4):339–353, 2009.

[29] X. Qu and B. Robinson. A case study of concolic testing tools and their limitations. In *Empirical Software Engineering and Measurement (ESEM), 2011 International Symposium on*, pages 117 –126, sept. 2011.

[30] R. Sasnauskas, O. Landsiedel, M. h. Alizai, C. Weise, S. Kowalewski, and K. Wehrle. KleeNet: Discovering insidious interaction bugs in wireless sensor networks before deployment. In *ACM/IEEE International Conference on Information Processing in Sensor Networks*, 2010.

[31] P. Saxena, D. Akhawa, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for JavaScript. In *IEEE Symposium on Security and Privacy*, 2010.

[32] K. Sen and G. Agha. CUTE and jCUTE : Concolic unit testing and explicit path model-checking tools. In *Computer Aided Verification*, 2006.

[33] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *European Software Engineering Conference/Foundations of Software Engineering*, 2005.

[34] SMT-LIB: The satisfiability module theories library. http://combination.cs.uiowa.edu/smtlib/.

[35] IEEE Computer Society. Standard for information technology-portable operating system interface (POSIX), 2008.

[36] N. Tillmann and W. Schulte. Parameterized unit tests. In *European Software Engineering Conference/Foundations of Software Engineering*, 2005.