Industrial Application of Concolic Testing Approach: A Case Study on `libexif` by Using CREST-BV and KLEE

Yunho Kim, Moonzoo Kim, YoungJoo Kim
CS Dept. KAIST, South Korea
kimyunho@kaist.ac.kr, moonzoo@cs.kaist.ac.kr, jerry88.kim@gmail.com

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

***Abstract*—As smartphones become popular, manufacturers such as Samsung Electronics are developing smartphones with rich functionality such as a camera and photo editing quickly, which accelerates the adoption of open source applications in the smartphone platforms. However, developers often do not know the detail of open source applications, because they did not develop the applications themselves. Thus, it is a challenging problem to test open source applications effectively in short time. This paper reports our experience of applying concolic testing technique to test `libexif`, an open source library to manipulate EXIF information in image files. We have demonstrated that concolic testing technique is effective and efficient at detecting bugs with modest effort in industrial setting. We also compare two concolic testing tools, CREST-BV and KLEE, in this testing project. Furthermore, we compare the concolic testing results with the analysis result of the Coverity Prevent static analyzer. We detected a memory access bug, a null pointer dereference bug, and four divide-by-zero bugs in `libexif` through concolic testing, none of which were detected by Coverity Prevent.**

I. Introduction

As smartphones become popular, rich features such as a camera and photo editing are added to the smartphone platforms. Due to high competition in the market, smartphone manufacturers such as Samsung Electronics should develop smartphones with rich functions in a short time, which accelerates the adoption of open source applications in the smartphone platform. Although the reliability of popular open source applications are high and field-proven by a large number of users, there can still be uncovered flaws in the applications. Thus, to ensure the high quality of smartphone products, manufacturers have to test open source applications adopted in the smartphone products in a careful manner. However, developers do not know the detail of open source applications, because they did not develop the applications themselves. Testing such applications in a short time is challenging given current industrial practice in which developers manually generate test cases.

As a solution to this problem, we propose applying concolic testing techniques to test open source applications. Concolic (CONCrete + symbOLIC) testing [26] (also known as dynamic symbolic execution [28] and white-box fuzzing [11]) combines concrete dynamic analysis and static symbolic analysis to automatically generate test cases to explore execution paths of a target program. 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 open source applications in a practical manner through case studies.

In this paper, we report our experience of applying concolic testing tools to test `libexif`, an open-source library to manipulate Exchangeable Image File Format (EXIF) information in image files. `libexif` is adopted on many smartphone platforms of Samsung Electronics such as Android, Samsung Linux Platform, and Samsung Bada platform. We have demonstrated that concolic testing technique is *effective* (in terms of the bug detection capability) and *efficient* (in terms of the speed of test case generation) to detect bugs in open-source applications. In addition, we compared two popular concolic testing tools, CREST-BV(an extended version of CREST with bit-vector (BV) support) and KLEE, in terms of effectiveness and efficiency in this testing project. Furthermore, we applied Coverity Prevent to `libexif` and compared the result with those of CREST-BV and KLEE. We detected one memory access bug, one null pointer dereference bug, and four divide-by-zero bugs in `libexif` by using CREST-BV and KLEE.

The organization of the paper is as follows. Section II overviews `libexif`. Section III explains related work on concolic testing tools. Section IV describes the backgrounds of CREST-BV and KLEE. Section V overviews this testing project. Section VI explains the testing methods we applied to improve the bug detection capability of concolic testing. Section VII describes the testing results by using CREST-BV, KLEE, and Coverity Prevent. Section VIII summarizes the lessons learned from the project. Section IX concludes this paper with future work.

II. Overview of `libexif`

`libexif` is an open source library to read/update/write Exchangeable Image File Format (EXIF) [13] metainformation from and to image files [1]. `libexif` contains 238

978-1-4673-1067-3/12/$31.00 © 2012 IEEE

ICSE 2012, Zurich, Switzerland
Software Engineering in Practice

functions in C (total 13,585 lines). The latest `libexif` was released on Dec 2010 (version 0.6.20).

To understand `libexif`, we need to understand the basic structure of an image file that contains the EXIF metainformation. The structure of the image files is depicted in Figure 1. The structure consists of

- a file header
- 0th image file directory (IFD) for primary image data and its value
- EXIF IFD and its value
- GPS IFD and its value
- 1st IFD for thumbnail data and its value
- thumbnail image data
- primary image data

An IFD consists of a 2 byte counter to indicate a number of tags in the IFD, tag arrays, and 4 byte offset to the next IFD. Each tag consists of tag id (2 bytes), type (2 bytes), count (i.e., a number of values) (4 bytes), and value (or offset to the value if the value is larger than 4 bytes) (4 bytes).

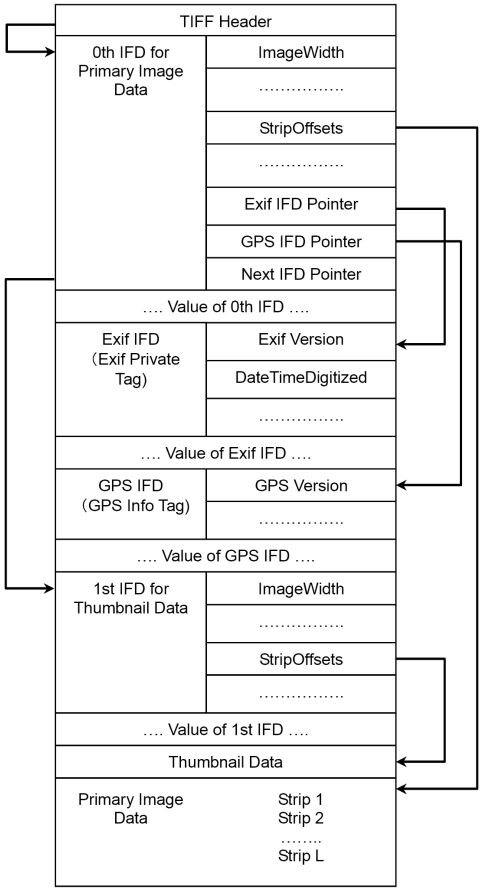

Figure 1. Basic Structure of Image Files (quoted from [6])

The EXIF specification [6] defines a large number of tags (185 pages long) for

- image data structure (image width, etc.)
- image data characteristics (gamma, etc.)
- picture-taking conditions (exposure time, etc.)
- user information (manufacturer notes, etc.)

One important tag is the manufacturer note tag that is used for manufacturers of EXIF writers to record any desired information; the contents are up to the manufacturer. In other words, camera manufacturers define their own tags (called *maker note tags*) such as a camera model number and lens type and include a set of maker note tags in the manufacturer note tag without restriction. Note that camera manufacturers define a large number of their own maker note tags and maker note tags are not specified in the official EXIF specification. For example, Canon defines 400+ maker note tags. We focus our effort to test `libexif` regarding these maker note tags (see Sections V-B, VI-B and VII-B).

III. Related Work

A. Concolic Testing Tools

The core idea of concolic testing is to obtain symbolic path formulas from concrete executions and solve them to generate test cases by using SMT solvers. Various concolic testing tools have been implemented to realize this idea (see [23] for a survey). We can classify the existing approaches into the following three categories based on how they obtain 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. 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 addition, it is easier to analyze the performance and internal behavior of the tools compared to the other approaches. In this group, CUTE [26], DART [10], and CREST [4] operate on C programs, while jCUTE [25] 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 [20]). 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 [11] 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 [28] targets C# programs that are compiled into Microsoft .Net binaries, KLEE [5] targets LLVM [18] binaries, and jFuzz [12] targets Java bytecode on top of Java PathFinder [29], [22].

B. Concolic Testing Case Studies

Concolic testing has been applied to detect bugs in various applications such as sensor networks [24], web applications [2], database applications [8], [21], embedded systems [14], [15], linux command-line applications [16], [15], etc. However, most related case studies concentrate on the evaluation of the authors' own techniques and do not describe the detailed user efforts and difficulties to apply concolic testing to real world applications. On the other hand, we not only describe the whole process of applying concolic testing to one real world application in detail, but also provide test strategies to address the difficulties to apply concolic testing to real world applications. Furthermore, we compared CREST-BV [17] and KLEE [5] in terms of test generation speed and the capability of detecting bugs and also compared the concolic testing tools and a commercial static analyzer, Coverity Prevent [3] to help practitioners to choose an adequate tool for their project.

IV. Overview of CREST-BV and KLEE

We selected CREST-BV and KLEE to apply to `libexif` for the following two reasons: (1) they can analyze target C programs, and (2) since they are open source tools, we can modify the source code to control testing experiments in a refined manner, and analyze testing results in deep technical level by obtaining various internal information.

A. CREST-BV

CREST-BV [17] is a concolic testing tool for C programs, which is an extended version of CREST [4] with bit-vector (BV) support. The original CREST supports only linear-integer arithmetic (LIA) formulas so that non-linear arithmetic operations in a target program cannot be analyzed symbolically, which often causes CREST to fail to generate test cases to reach specific branches. Therefore, to improve the bug detection capability and branch coverage of CREST, we have developed CREST-BV based on CREST 0.1.1 to support bit-vector symbolic path formulas by using Z3 2.19 SMT solver [19].[1] CREST-BV consists of the front-end for instrumentation, the middle-end for symbolic execution, and the back-end for solving the symbolic path formula and generating a new test case. First, we develop probes to symbolically record non-linear arithmetic operations and extend the instrumentation engine to insert the probes at non-linear arithmetic operations in a target program. Second, we extend the middle-end to represent and symbolically execute the non-linear path conditions. Finally, we extend the back-end to translate a symbolic path formula into a bit-vector SMT formula and solve the SMT formula with bit-level accuracy. CREST-BV uses Z3 as a SMT solver because Yices does not support divide and modular operations in bit-vector formulas through its API.

A concolic testing procedure can be described as follows:

1) *Declaration of symbolic variables*
   Initially, a user must specify which variables should be handled as symbolic variables, based on which symbolic path formulas are constructed.
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, 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*
   When a target program terminates, to obtain the next input values, ψ_i is generated by negating one path condition c_j and removing subsequent path conditions of ϕ_i (i.e., $\psi_i = c_1 \wedge c_2 ... \wedge \neg c_j$). The selection of c_j depends on a search strategy of CREST-BV. 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 [27] 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.

[1] CREST-BV is a component of distributed concolic testing tool SCORE [16]. CREST-BV can be downloaded from http://pswlab.kaist.ac.kr/tools/crest-bv.

B. KLEE

KLEE [5] is an open source symbolic execution tool to automatically generate test cases. KLEE is implemented as a modified LLVM virtual machine targeting LLVM bytecode programs. In addition to symbolic declaration of program variables (see Step 1 in Section IV-A), KLEE provides a symbolic POSIX library to enable analysis of programs that utilize environment intensively in a convenient manner. For example, a user can test a target program with a symbolic input file with a command line option `--sym-files <num of files> <max file size>`. KLEE supports bit-vector (BV) formulas and uses STP [9] as a constraint solver.

A concolic testing procedure of KLEE is as follows. While CREST-BV builds symbolic path formulas ϕ_is from concrete execution paths one by one, KLEE builds a symbolic execution tree by forking new symbolic processes for symbolic executions of different branching decisions. When a symbolic process p_i of KLEE interprets a conditional LLVM instruction corresponding to `if(b) ... else ...`, p_i creates two symbolic queries $\phi_i \wedge b$ and $\phi_i \wedge \neg b$ (where ϕ_i is a symbolic path formula from the initial/root instruction to the current instruction in p_i) to check whether a current execution path can proceed to the true branch and/or the false branch by solving the queries using STP. If only one query is satisfiable, current symbolic process p_i proceeds to execute a corresponding branch. If both queries are satisfiable, p_i proceeds to a true branch (ϕ_i is updated with $\phi_i \wedge b$) and forks a new symbolic child process p_j, which proceeds to a false branch (ϕ_j is updated with $\phi_i \wedge \neg b$). Since there can be multiple symbolic processes, a KLEE's search strategy decides which symbolic process to proceed. KLEE repeats this procedure until it covers all possible symbolic execution paths. If a time limit given by a user is reached, KLEE stops execution of all symbolic processes, and creates test cases by solving queries for all leaves in the current symbolic execution tree.

V. Project Overview

A. Project Goal and Scope

This project was conducted to evaluate *effectiveness* (in terms of the bug detection capability) and *efficiency* (in terms of the speed of test case generation) of concolic testing technique on open source applications that are used in Samsung smartphones. We selected `libexif` as a target application among several open source applications in Samsung smartphones for the following reasons:

1) We targeted open source applications written in C, since such applications can be used for multiple smartphone platforms including Samsung Bada OS, Samsung Linux Platform (SLP), and Android, thus maximizing the benefits of the improved quality through bug detection.
2) We excluded applications that contain heavy floating point arithmetic operations, since branches of conditional statements that contain floating point arithmetic operations may not be covered completely. This is because most SMT solvers do not solve path formulas containing floating point arithmetic operations.

We focused on detecting the following *run-time failure* bugs that cause a target program to crash.

- *Divide-by-zero bugs:*
  Although divide-by-zero bugs can be detected without explicit assert statements, we mechanically added `assert(<denominator>!=0)` right before the statements containing division operators whose denominators are not constant. This is because these assert statements increase the probability of detecting the divide-by-zero bugs by forcing a concolic testing tool to generate test inputs that make the denominator variables zero to exercise false branches of the assert statements.
- *Null pointer dereference bugs:*
  This bug is one of the main causes to crash a target program. Unlike divide-by-zero, we did not add assert statements such as `assert(<pointer>!=NULL)` because current concolic testing tools cannot analyze pointer variables symbolically, thus do not generate test inputs that make the pointer variables null.
- *Out-of-bound memory access bugs:*
  Out-of-bound memory access crashes a target program with high probability, although this violation may not always crash a target program.

The primary reason to concentrate on these run-time failure bugs is that these bugs can disable normal operations of smartphones and damage the user experience severely. Another reason is that these run-time failure bugs can be detected conveniently without manually inserting explicit assert statements. It takes a large amount of time for human engineers to understand detailed functionalities of `libexif` and manually define/insert corresponding assert statements to check the correctness of the functionalities.

For the testing experiments, one graduate student and one engineer of Samsung Electronics took five days to apply KLEE and CREST-BV to `libexif` to detect the run-time failure bugs, not including exploratory experiments and time taken to modify KLEE and CREST-BV for the testing experiments.

B. Testing Target and Testing Environment

We selected `test-mnote.c` in `libexif` as a main target program to test. `test-mnote.c` reads an image file as an input and exercises many functionalities of `libexif` including read/write of EXIF data from and to the image file, creation/initialization of a `libexif` parser, manipulations of EXIF data and so on. `test-mnote.c` calls 206 functions among total 238 functions in `libexif`. Furthermore,

test-mnote.c focuses to test a portion of libexif code that handles maker note tags, which seems less reliable compared to other parts of libexif, since maker note tags are defined in an ad-hoc manner by camera manufacturers (see Section II).

All experiments were performed on 64 bit Fedora Linux 9 equipped with a 3.6 GHz Core2Duo processor and 16 gigabytes of memory. For the testing experiments, we used CREST-BV with Z3 2.19 SMT solver and KLEE rev.136605 [5]. KLEE supports bit-vector formulas and uses STP rev.1398 [9]. In these series of experiments, we measured the branch coverage via gcov by executing test-mnote.c on the generated test cases.

VI. Testing Methods

The number of feasible execution paths of a target program is very large even for a small program due to loops. For example, if each execution path consists of 100 branch conditions on average, in theory, there can be $1.26 \times 10^{30} (= 2^{100})$ execution paths to analyze. Consequently, a concolic testing tool can analyze only a small subset of the entire execution paths in practice. Therefore, we have to develop *testing strategies* to focus on a set of execution paths that may expose symptoms of bugs. We have developed the following two testing strategies described in Sections VI-A and VI-B respectively.

A. Baseline Concolic Testing

We feed a small symbolic file as an input to test-mnote.c. We set the size of the input symbolic file at 244 bytes, since another test program (test-parser.c) in libexif that tests only minimal functionalities of libexif (i.e., creation and destruction of a EXIF data parser) uses a 244 byte long image file. In other words, we use a symbolic file that consists of 244 symbolic variables whose size is 1 byte each. Note that a larger symbolic file would enlarge the size of search space significantly, to decrease the probability of detecting bugs in a given limited time.

B. Concolic Testing with Focus on Maker Note Tags with Concrete Image Files

We focus on analyzing execution paths that are exercised based on the maker note tags. A rationale for this strategy is that the portion of libexif that handles maker note tags may be buggy due to ad-hoc definitions of maker note tags (see Section II). Another reason is that such code takes a large portion of libexif; five functions among the largest ten in libexif are designed to handle maker note tags and they comprise 27% of the total libexif branches.

To focus on handling maker note tags, we declare the bytes of the input image file that correspond to maker note tags symbolically; the other bytes in the image file are handled concretely. By concentrating on the maker note tags, we expect to detect bugs related to the maker note tags with high probability within a given limited time. We used the following six input image files that contain the maker note tags (downloaded from [13]). For each search strategy, we executed CREST-BV and KLEE six times for the six image files respectively.

- sanyo-vpcg250.jpg
- sanyo-vpcsx550.jpg
- canon-ixus.jpg
- nikon-e950.jpg
- fujifilm-finepix40i.jpg
- olympus-c960.jpg

C. KLEE and CREST-BV Settings

We applied KLEE to libexif with depth first search (DFS), random path, random search, covering new, interleaving of DFS + covering new search strategies as these five search strategies may cover different paths and different branches (see [5] for the detail of the search strategies) [2]

We executed KLEE with --max-time as 900, 1800, and 3600 seconds (15, 30, and 60 minutes respectively) for each testing strategy. [3] The --max-time option restricts only symbolic execution time, not including time spent to generate test cases from the current symbolic execution tree when the target program is forced to stop. If this test case generation process takes more than a given amount of time, KLEE halts without generating possible test cases further. We modified the KLEE source code to generate all possible test cases without time limit; thus, the actual amounts of testing time were larger than 15, 30, and 60 minutes respectively. In addition, we disabled counter example cache (CEC) unit of KLEE, since CEC slowed down KLEE several times compared to KLEE without CEC. [4]

Then, we applied CREST-BV to libexif with DFS, random path, and control flow graph based search (CFG) search strategies (see [4] for the detail of the search strategies). We executed CREST-BV with DFS and random path for the equal amount of time spent by KLEE. For the CFG search strategy, we executed CREST-BV in 15, 30, and 60 minutes, since no search strategy of KLEE is equivalent to the CFG search strategy. In addition, we added source code of small but frequently used five library functions (memcmp(), memcpy(), bsearch(), qsort() and

[2]We selected "covering new" among the six non-uniform random search strategies of KLEE, since this strategy likely increases the branch coverage quickly. Also, we performed the interleaved combination of DFS and the covering new search strategies similarly to that of [5].

[3]The complete options given to KLEE are as follows:
```
--simplify-sym-indices --emit-all-errors
--disable-inlining --use-forked-stp
--optimize --libc=uclibc --posix-runtime
--max-time=[900,1800,3600] --watchdog
--allow-external-sym-calls --use-batching-search
--batch-instructions=10000 --sym-args 1 1 1
--sym-files 1 244
```

[4]We contacted the KLEE development team regarding this issue, but have not received a response.

strcmp() (total 173 lines)) to test-mnote.c, since CREST-BV does not support symbolic C library automatically. For a symbolic file input, we modified a file read function in libexif so that every byte read from an input file was declared as a symbolic variable.

We repeated all these series of experiments five times to get execution time, a number of generated test cases, and branch coverage on average. In addition, we applied Coverity Prevent [3] to libexif and compared the result with those of CREST-BV and KLEE. Coverity Prevent is a static analyzer to detect the run-time failure bugs in large programs.

VII. Experimental Results

A. Baseline Concolic Testing

1) KLEE Results: Table I shows *time* taken for each of the five search strategies of KLEE with --max-time 900, 1800, and 3600 seconds . This table also shows *numbers of test cases* generated and *branch coverage ratios* achieved for each search strategy. In addition, the third rightmost to the fifth rightmost columns of the table show the total amount of time spent for the five search strategies, total number of test cases generated, and the total branch coverage obtained based on the all test cases. The rightmost two columns show a number of bugs detected (effectiveness) and test case generation speed (efficiency).

For example, with DFS and max-time=3600, KLEE spent 3705 seconds to generate 4868 test cases that covered 8.1% of the branches of libexif (see the last row and the second to fourth columns of Table I). Through the all five search strategies with max-time=3600, KLEE generated 34125 test cases to cover 20.4% of the branches of libexif in 12 hours (=46244 seconds) (see the last row and the third rightmost to the fifth rightmost columns of Table I).

Evaluation of the effectiveness and efficiency of KLEE on libexif is as follows:

- *Effectiveness in terms of bug detection capability*: KLEE detected an *out-of-bound memory access bug* within the first 50 test cases generated via the random path, the random search, the covering new, and the DFS+covering new strategies; it took less than 60 seconds for each search strategy to detect the bug. This bug is located in exif_data_load_data () of exif-data.c as follows (line 2):

  ```
  ...
  1:if (offset + 6 + 2 > ds) { return; }
  2:n = exif_get_short(d+6+offset, ...);
  ...
  ```

  offset is an unsigned 32 bit integer variable to indicate an offset to the value of a tag (see Section II). libexif ignores offset if it is out of bound (see line 1). However, if offset $\geq 2^{32} - 8$, integer overflow occurs and libexif continues to line 2, where exif_get_short() accesses out-of-bound memory address pointed by d+6+offset.
- *Efficiency in terms of the speed of test case generation*: KLEE generated 0.9 test case per one second on average (=(1.1+0.9+0.7)/3) (see the rightmost column of Table I). We observed that the speed of test case generation decreased as execution time increased. For example, KLEE generated 1.1 test cases per one second when KLEE executed for 10400 seconds. But KLEE generated 0.7 test cases per one second for 46244 seconds.

  This is because executions go deeper/longer as testing time increases, which increases the sizes of symbolic path formulas consequently. Because SMT solvers take longer time to solve longer formula in general, the time taken to analyze each symbolic execution formula increases as the length of the execution path increases. Thus, the average speed of test case generation decreases as the testing time increases.

In addition, we observed that the total branch coverage achieved through all five search strategies is not larger than that of the covering new strategy. For example, with max-time as 900, 1800, and 3600 seconds, the covering new search strategy covered 11.1%, 19.7%, and 20.4% of the libexif branches respectively (see the 13th column of Table I) and the total branch coverages achieved via all five search strategies were the same (i.e., 11.1%, 19.7%, and 20.4% respectively) (see the third rightmost column of Table I). This is because the five search strategies covered execution paths that executed almost same statements in this baseline concolic testing setting on libexif.

One reason for this result is that, in this baseline concolic testing setting on libexif, test-mnote.c will terminate quickly by rejecting the symbolic input file because the file does not conform to the image structure shown in Figure 1. Thus, most executions exercised only simple validity check routines and had little chance to generate diverse symbolic execution paths via different search strategies. In other words, this baseline concolic testing setting on libexif could not exploit the diversity of the five different search strategies much.

We also observed that branch coverage did not increase much over increasing number of test cases. For example, for DFS, KLEE covered 8.1% of the branches through 1289 test cases, but covered the same 8.1% of the branches through 4868 test cases (see the second and the fourth columns of Table I). A reason for the non-increasing branch coverage over the increasing number of test cases is that libexif has a large number of execution paths that execute the same statements through loops.

2) CREST-BV Results: Table II shows that, through DFS, random path, and CFG based search strategies, CREST-BV spent 17811 seconds to generate 367768 test cases to cover

Table I
STATISTICS ON THE BASELINE CONCOLIC TESTING EXPERIMENTS BY USING KLEE

Time option (sec)	DFS			Random path			Random search			Covering new			DFS + covering new			Total of the 5 search strategies			# of bugs detected	TC gen. speed (#/sec)
	time (sec)	# of TC	Br.cov. (%)	time (sec)	# of TC	Br.cov. (%)	time (sec)	# of TC	Br.cov. (%)	time (sec)	# of TC	Br.cov. (%)	time (sec)	# of TC	Br.cov. (%)	time (sec)	# of TC	Br.cov. (%)		
900	1001	1289	8.1	2577	2280	11.1	2294	3192	11.1	2574	3072	11.1	1954	2022	11.1	10400	11855	11.1	1	1.1
1800	1903	2450	8.1	5530	4121	11.1	4832	5277	11.1	4944	4083	19.7	4928	3089	19.7	22137	19020	19.7	1	0.9
3600	3705	4868	8.1	10506	7084	11.1	9406	9945	19.1	12609	4543	20.4	10018	7685	19.7	46244	34125	20.4	1	0.7

Table II
STATISTICS ON THE BASELINE CONCOLIC TESTING EXPERIMENTS BY USING CREST-BV

Corresponding KLEE time option (sec)	DFS			Random path			Control flow graph (CFG) based			Total of the 3 search strategies			# of bugs detected	TC gen. speed (#/sec)
	time (sec)	# of TC	Br.cov. (%)	time (sec)	# of TC	Br.cov. (%)	time (sec)	# of TC	Br.cov. (%)	time (sec)	# of TC	Br.cov. (%)		
900	1001	12671	20.2	2577	100934	9.3	900	25191	21.8	4478	138796	22.3	1	31.0
1800	1903	22317	20.2	5530	171531	9.3	1800	25752	21.8	9233	219600	22.3	1	23.8
3600	3705	42499	20.3	10506	259625	10.3	3600	65644	21.8	17811	367768	22.3	1	20.6

22.3% of the branches in `libexif` (see the last row and the third rightmost to the fifth rightmost columns of Table II).

Regarding effectiveness, CREST-BV detected the same out-of-bound memory access bug via random path and CFG based search strategies within the first 100 test cases (in less than 10 seconds). Regarding efficiency, however, CREST-BV demonstrated far better performance than KLEE. The speed of test case generation by CREST-BV was 28 times faster than that of KLEE, since CREST-BV generated 25.1 test cases per one second (= (31.0+23.8+20.6)/3) while KLEE generated 0.9 test cases per one second on average.

One reason for the speed difference is that symbolic path formulas of KLEE were almost three times longer than those of CREST-BV. For example, with DFS and `max-time=1800`, the average number of clauses in each query of KLEE was 141 while the average number of clauses in each symbolic path formula of CREST-BV with the equivalent setting was 55. This is because KLEE analyzes not only the executions of a target program, but also the executions of the external symbolic C libraries used by the target program. Another reason was that since KLEE utilizes SMT array theory [9] to support symbolic arrays [7], a symbolic execution query that contains array operations is more difficult to solve than a symbolic path formula of CREST-BV that does not support symbolic arrays.

Also note that the total branch coverage achieved by CREST-BV through the three search strategies (22.3%) was higher than that of KLEE through the five search strategies (20.4%) for `max-time=3600`. This result is not surprising, since CREST-BV generated 10 times more test cases (367768) than KLEE did (34125) with `max-time=3600`.

In addition, we made the following observations, which are similar to the observations with KLEE.

- Three different search strategies together did not improve the branch coverage much. For example, with the CFG-based search strategy and `max-time=3600`, CREST-BV covered 21.8% of the `libexif` branches. But with all the three search strategies, CREST-BV covered 22.3% of the branches in total (0.5% improvement).
- We observed that the speed of test case generation decreased as execution time increased. For example, CREST-BV generated 31.0 test cases per one second when CREST-BV executed for 4478 seconds. But it generated 20.6 test cases per one second for 17811 seconds.

B. Concolic Testing with Focus on Maker Note Tags with Concrete Image Files

1) KLEE Results: Table III shows *total time* taken for each of the five search strategies of KLEE with `--max-time` 900, 1800, and 3600 seconds (15, 30, and 60 minutes respectively) on the six image files in total. This table also shows *total numbers of test cases* generated and *total branch coverage ratios* achieved for each search strategy on the six image files. Table III shows that KLEE spent 31 hours (=112506 seconds) to generate 144248 test cases that achieved 49.5% of `libexif` branches with `max-time=3600`.

The evaluation of effectiveness and efficiency of KLEE in this experiment is as follows:

- *Effectiveness in terms of bug detection capability*: KLEE detected a null pointer dereference bug via the random search, the covering new, and the DFS+covering new strategies with `canon-ixus.jpg` within the first 700 test cases (in 10 minutes). This bug is located in `mnote_canon_tag_get_description()` of `mnote_canon_tag.c` as follows (see line 5):
  ...

Table III
STATISTICS ON THE CONCOLIC TESTING WITH FOCUS ON MAKER NOTE TAGS WITH 6 IMAGE FILES BY USING KLEE

Time option (sec)	DFS (Sum on the 6 files) time (sec)	DFS # of TC	DFS Br.cov. (%)	Random path (Sum on the 6 files) time (sec)	Random path # of TC	Random path Br.cov. (%)	Random search (Sum on the 6 files) time (sec)	Random search # of TC	Random search Br.cov. (%)	Covering new (Sum on the 6 files) time (sec)	Covering new # of TC	Covering new Br.cov. (%)	DFS+covering new (Sum on the 6 files) time (sec)	DFS+covering new # of TC	DFS+covering new Br.cov. (%)	Total of the 5 strategies on the 6 files each time (sec)	Total of the 5 strategies on the 6 files each # of TC	Total of the 5 strategies on the 6 files each Br.cov. (%)	# of bugs detected	TC gen. speed (#/sec)
900	5424	15804	44.5	5526	4800	44.3	5592	6684	44.7	5994	20454	44.7	5880	23424	44.7	28416	71166	49.2	1	2.5
1800	10830	24936	44.7	11010	8172	44.7	11154	10758	44.7	11646	24492	44.7	11652	34890	44.7	56292	103248	49.2	1	1.8
3600	21642	39270	44.7	21996	11342	44.7	22416	15378	45.0	23142	29988	45.0	23310	48270	45.0	112506	144248	49.5	1	1.3

Table IV
STATISTICS ON THE CONCOLIC TESTING WITH FOCUS ON MAKER NOTE TAGS WITH 6 IMAGE FILES BY USING CREST-BV

Corresponding KLEE time option (sec)	DFS (Sum on the 6 files) time (sec)	DFS # of TC	DFS Br.cov. (%)	Random path (Sum on the 6 files) time (sec)	Random path # of TC	Random path Br.cov. (%)	CFG based (Sum on the 6 files) time (sec)	CFG based # of TC	CFG based Br.cov. (%)	Total of the 3 strategies on the 6 files each time (sec)	Total of the 3 strategies on the 6 files each # of TC	Total of the 3 strategies on the 6 files each Br.cov. (%)	# of bugs detected	TC gen speed (#/sec)
900	5424	93645	48.7	5526	130387	58.7	5400	98800	56.1	16350	322832	66.2	5	19.7
1800	10830	174173	48.7	11010	245873	59.3	10800	181362	56.5	32640	601408	67.1	5	18.4
3600	21642	309931	48.7	21996	433570	59.6	21600	325261	57.4	65238	1068762	68.1	5	16.4

```
1:for(i=0;i<sizeof(table)/sizeof(table[0]);i++)
2:  //t is a maker note tag read from an image
3:  if (table[i].tag==t) {
4:    //Null-pointer dereference occurs!!!
5:    if(!*table[i].description)
6:    return "";
...
```

This bug crashes `libexif` if an input image file contains an unknown maker note tag ID for Canon cameras. This is because the last element of `table` contains its `description` as null to indicate the end of `table`, which contains a list of maker note tags and corresponding descriptions for Canon cameras. Similar functions for other camera companies such as Fujifilm or Sanyo check if `table[i].description` becomes null, but `mnote_canon_tag_get_description()` does not, which can cause null pointer dereference.

- *Efficiency in terms of the speed of test case generation*: KLEE generated 1.9 test case per one second on average (= (2.5+1.8+1.3)/3).

In addition, we observed that the five search strategies together improved branch coverage around 1.1 times, in contrast to the baseline concolic testing. For example, KLEE covered 44.7% of the `libexif` branches via DFS with `max-time=3600` seconds on the six image files. But the five search strategies together covered 49.5% of the `libexif` branches. This is because KLEE generated normal/meaningful symbolic executions based on the concrete input image files that conform to the structure of Figure 1, and utilized diversity of the five different search strategies.

2) CREST-BV Results: Table IV shows that CREST-BV generated 1068762 test cases to cover 68.1% of the branches in `libexif` in 18 hours (=65238 seconds) via the three search strategies on the six input image files.

The evaluation of effectiveness and efficiency of CREST-BV in this experiment is as follows:

- *Effectiveness in terms of bug detection capability*: CREST-BV detected the same null pointer dereference bug as KLEE did via all three search strategies with `canon-ixus.jpg` within the first 100 test cases (in one minute). In addition, CREST-BV detected four divide-by-zero bugs in `mnote_olympus_entry_get_value()` in `mnote-olympus-entry.c` as follows (see line 4):

```
...
1:vr=exif_get_rational(...);
2://Added for concolic testing
3:assert(vr.denominator!=0);
4:a = vr.numerator / vr.denominator;
...
```

`vr` is a rational number that is read from an input image file (line 1), which consists of `numerator` and `denominator`. An assert statement at line 3 is mechanically inserted to force CREST-BV to generate symbolic inputs to make `vr.denominator` zero (see Section V-A). If an input image file contains `vr` whose `denominator` is zero, `libexif` crashes due to the divide-by-zero bug. `mnote-olympus-entry.c` has three more similar divide-by-zero bugs. CREST-BV detected these bugs within the first 23000 test cases in 30 minutes. Note that KLEE could not detect these divide-by-zero bugs, since none of the test cases generated by KLEE reached the bug locations.

- *Efficiency in terms of the speed of test case generation*: CREST-BV generated 18.2 test case per one second on average (= (19.7+18.4+16.4)/3), which is 10 times faster than KLEE (KLEE generated 1.9 test cases per one second on average).

C. Comparison between CREST-BV and Coverity Prevent

Coverity Prevent [3] is a commercial static analyzer that can analyze a large source code in C/C++/Java quickly at

static time. Since Coverity Prevent also targets the run-time failure bugs, we compared the concolic testing results with the static analysis result of Coverity Prevent. We applied Coverity Prevent to `libexif` with a maximal warning level option to detect as many bugs as possible. Coverity Prevent reported the following run-time failure warnings after analyzing `libexif` in 5 minutes:

- *Null pointer dereference (3 warnings)*:
  Coverity Prevent detected a null-pointer dereference bug in `exif_loader_get_buf()` of `exif-loader.c`, which crashes `libexif` if `loader` is null (line 2).

```
1:if(!loader||(loader->data_format ... ) {
2:  exif_log(loader->log, ...);
```

  We could not detect this bug using concolic testing, because `test-mnote.c` does not call `exif_loader_get_buf()`. The other 2 warnings were false alarms.
- *Out-of-bound memory access (1 warning)*:
  After manual analysis, this warning turned out to be a false alarm.

In addition, Coverity Prevent generated the following warnings other than the run-time failure bugs:

- *Operands do not affect result (1 warning):*
  A condition in one conditional statement is evaluated as false regardless of the values of its operands.
- *Recursion in included headers (6 warnings):*
  Coverity Prevent generated warnings regarding recursively nested header files.
- *Calling risky function (4 warnings)*:
  Coverity Prevent considers `strcpy()` and `sprintf()` risky, since they can access illegal memory if the allocated memory space for destination is smaller than that of the source. After manual analysis, however, we found that these functions were used correctly in `libexif`.

Note that Coverity Prevent could not detect any bugs CREST-BV detected. Therefore, even after applying Coverity Prevent, it will be beneficial to apply a concolic testing tool to detect the run-time failure bugs.

VIII. Lessons Learned

In this section, we summarize the lessons learned from this testing project, which, we believe, can be applied to other projects of similar domains.

A. Practical Application of Concolic Testing

Through the project, we demonstrated that concolic testing techniques are effective to detect corner case bugs by detecting one out-of-bound memory access bug (see Section VII-A), one null pointer dereference bug and four divide-by-zero bugs (see Section VII-B). Compared to the random testing using a 244 byte random image file for 3600 seconds, which covered only 3.8% of the branches and detected no bug, this bug detection result of concolic testing was significant. Furthermore, it took only one week for us to detect such critical bugs without much knowledge on `libexif` (see Section V-A). Thus, concolic testing technique can be a practical solution to test open source applications in industrial setting.

Another interesting observation was that CREST-BV and KLEE were more effective to detect the run-time failure bugs than Coverity Prevent in this testing project. This is because concolic testing techniques perform *precise* context sensitive analysis with concrete run-time information while Coverity Prevent performs context insensitive static analysis for a large code fast. Thus, it is a good idea to apply both static analyzers such as Coverity Prevent and concolic testing tools such as KLEE and CREST-BV together.

B. Importance of Testing Methodology

Although concolic testing is an automated test case generation technique, through the project, we confirmed that good testing methodology designed by a human engineer can improve the effectiveness of concolic testing in a large degree. For example, if we performed only the baseline concolic testing (see Section VI-A), we would not detect the null pointer dereference bug and the four divide-by-zero bugs (see Section VII-B).

A main reason for the necessity of carefully designed testing strategy for concolic testing is that a target search space is very large for most target programs and concolic testing can analyze only a small portion of the space in practice. Therefore, developers or testing engineers should devise smart testing strategies to maximize the effectiveness of concolic testing.

C. Advantages of CREST-BV over KLEE

We found that CREST-BV has several advantages over KLEE in terms of both effectiveness and efficiency. First of all, CREST-BV is 10 to 28 times faster than KLEE in terms of test case generation speed in this project (see Section VII). Second, branch coverage achieved in a given amount of time by CREST-BV is higher than KLEE as CREST-BV generates test cases much faster than KLEE (for example, CREST-BV covered total 68.1% of the branches in `libexif` while KLEE did total 49.5% in the concolic testing with focus on maker note tags with `max-time=3600` as described in Table III). Finally, based on these two aforementioned advantages, CREST-BV has better bug detecting capability than KLEE does. For example, CREST-BV detected four divide-by-zero bugs, which KLEE did not (see Section VII-B).

IX. Conclusion and Future Work

We have applied two concolic testing tools, CREST-BV and KLEE, to open source application `libexif` to

detect run-time failure bugs. We found corner case bugs in `libexif` such as a memory access bug, a null pointer dereference bug, and divide-by-zero bugs, which crash `libexif` at run-time, thus causing serious problems. Considering that `libexif` is a popular open source application and field-proven by many users, it is interesting that we could detect such critical bugs in one week without much knowledge of `libexif`. Thus, we believe that concolic testing technique can be a practical solution to test open source applications in industrial setting.

In addition, we made several interesting observations through the project. Although concolic testing is an automated technique, human engineer can improve the effectiveness of concolic testing by devising smart testing strategies, since we need to focus search space to analyze out of a large number of possible execution paths. Samsung Electronics and KAIST will continue collaboration to improve the concolic testing techniques and apply CREST-BV to more applications on Samsung smartphone platforms.

ACKNOWLEDGEMENT

This work was supported by Samsung Electronics, the ERC of MEST/NRF (Grant 2012-0000473), and Basic Science Research Program of MEST/NRF (2010-0005498).

REFERENCES

[1] The libexif C EXIF library. http://libexif.sourceforge.net/.

[2] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. Ernst. Finding bugs in dynamic web applications. In *ISSTA*, 2008.

[3] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A few billion lines of code later: Using static analysis to find bugs in the real world. *CACM*, 53:66–75, February 2010.

[4] 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.

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

[6] CIPA and JEITA. Exchangeable image file format for digital still cameras: Exif version 2.3. http://www.cipa.jp/english/hyoujunka/kikaku/pdf/DC-008-2010_E.pdf.

[7] B. Elkarablieh, R. Godefroid, and M. Levin. Precise pointer reasoning for dynamic test generation. In *ISSTA*, 2009.

[8] M. Emmi, R. Majumdar, and K. Sen. Dynamic test input generation for database applications. In *ISSTA*, 2007.

[9] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *CAV*, 2007.

[10] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *PLDI*, 2005.

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

[12] K. Jayaraman, D. Harvison, V. Ganesh, and A. Kiezun. jFuzz: A concolic whitebox fuzzer for Java. In *NFM*, 2009.

[13] JEITA. Exif.org. http://www.exif.org/.

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

[15] M. Kim, Y. Kim, and Y. Jang. Industrial application of concolic testing on embedded software: Case studies. In *ICST*, 2012.

[16] M. Kim, Y. Kim, and G. Rothermel. A scalable distributed concolic testing approach: An empirical evaluation. In *ICST*, 2012.

[17] Y. Kim and M. Kim. CREST-BV: a concolic testing tool for C programs with bit-vector support, 2012. http://pswlab.kaist.ac.kr/tools/crest-bv/.

[18] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, 2004.

[19] L. Moura and N. Bjorner. Z3: An efficient SMT solver. In *TACAS*, 2008.

[20] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *PLDI*, 2007.

[21] K. Pan, X. Wu, and T. Xie. Generating program inputs for database application testing. In *ASE*, 2011.

[22] 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.

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

[24] 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 *IPSN*, 2010.

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

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

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

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

[29] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *ASE*, Sept. 2000.