SAT-based Bounded Software Model Checking for Embedded Software: A Case Study

Yunho Kim and Moonzoo Kim
CS Dept. KAIST, Daejeon, South Korea
kimyunho@kaist.ac.kr, moonzoo@cs.kaist.ac.kr

***Abstract*—Conventional manual testing often misses corner case bugs in complex embedded software, which can incur large economic loss. To overcome the weakness of manual testing, automated program analysis/testing techniques such as software model checking and concolic testing have been proposed. This paper makes a detailed report on the application of a SAT-based bounded software model checking technique using CBMC to `busybox ls` which is loaded on a large number of embedded devices such as smartphones and network equipments. In this study, CBMC demonstrated its effectiveness by detecting four bugs of `busybox ls`, but also showed limitations for the loop analysis. In addition, we report the importance of calculating minimum iterations to exit a loop (MIEL) to prevent false negatives in practice.**

I. Introduction

Although manual testing is a de-facto standard method to improve the quality of software in industry, conventional manual testing methods often fail to detect costly corner-case bugs in target programs. Two main reasons of such deficiency of manual testing are

- *Low effectiveness:*
  Human engineers are not good at thinking of exceptional scenarios. Engineers often focus on a few main execution scenarios and miss numerous exceptional scenarios caused by complex combination of unexpected conditions.
- *Low efficiency:*
  Human engineers are slow to generate test cases. Thus, it is difficult to manually generate a large number of test cases required to test a target program in fine granularity.

These limitations are serious issues in industrial projects, particularly in embedded system domains where high reliability is required and product recall for bug-fixing incurs significant economic loss.

To solve such limitations, several automated software analysis techniques such as model checking [13], software model checking [18], and concolic testing (a.k.a., dynamic symbolic execution or white-box fuzzing) [26], [15] have been developed. However, such techniques are not frequently applied to industrial software due to steep learning curve and hidden costs to apply these techniques to industrial software in practice. For example, to perform effective and efficient analysis, a user has to understand and overcome the limitations of the automated technique being applied, which are often not clearly described in related technical papers. Consequently, field engineers hesitate to adopt automated analysis techniques to their projects.

To transfer automated techniques to industry, thus, it is essential to conduct exploratory case studies applying the automated analysis techniques to real-world software since field engineers need references to estimate the required effort and the benefit of applying the automated techniques in detail. Furthermore, concrete applications of such techniques can also benefit researchers by guiding new research goals and directions to solve practical limitations observed in the studies.

In this paper, we report our experience of applying SAT-based bounded software model checking technique using CBMC [11] to `busybox ls` [1] (i.e., a tiny version of unix/linux `ls` utility to display directory/file information) which is loaded on a large number of embedded devices such as smartphones and network equipments [2]. In this study, we have checked 15 functional requirements of `busybox ls` specified in the POSIX standard specification [28] on `ls`. CBMC demonstrated its effectiveness by detecting four bugs of `busybox ls`, but also showed limitations for the loop analysis (Section V). In addition, we compare the advantages and weaknesses of the SAT-based bounded software model checking technique with those of concolic testing using CREST [19] (Section VI-D).

The main contribution of this paper is that this case study can serve as a reference to promote field engineers to adopt automated software analysis techniques. We have reported the detailed steps of applying CBMC to real-world embedded software `busybox ls` (how we obtained requirement properties, how we set loop bounds, etc.) and concretely describing the benefit (i.e., high bug detection capability (Section V-C)) and the limitation of the technique (i.e., the manual cost for loop bound analysis (Section V-A) and the scalability problem for loop unwinding (Section V-C)) in practice. Thus, field engineers can estimate necessary efforts and benefits of applying CBMC from this case study and utilize the case study as a reference to apply automated software analysis techniques to their own projects.

The paper is organized as follows. Section II overviews a SAT-based bounded model checking technique. Section III explains the background of this study. Section IV describes the experiment setting on how we applied CBMC to `busybox ls`. Section V shows the model checking results. Section VI shares the lessons learned from this case study. Section VII describes other case studies utilizing CBMC and compare those studies with our case study. Finally, Section VIII summarizes this paper with future work.

II. Background on SAT-based Bounded Model Checking

A. Bounded Model Checking

Bounded model checking [6] unwinds the control flow graph of a target program P for a fixed number of times n and then checks if an error can be reached within these n steps. SAT-based bounded model checking [10] unrolls the target program P n times, transforms this unrolled program into the SAT formula ϕ_P, and then checks whether P can reach an error within this bound n by checking the satisfiability of ϕ_P [16]. In spite of the NP-complete complexity, structured Boolean satisfiability (SAT) formulas generated from real world problems are successfully solved by SAT solvers in many cases. Modern SAT solvers, such as MiniSAT [14] and Chaff [24], exploit various heuristics [31] and can solve some large SAT formulas containing millions of variables and clauses in modest time [3].

To use a SAT solver as a bounded model checker to verify whether a given C program P satisfies a requirement property R, it is necessary to translate both P and R into Boolean formulas ϕ_P and ϕ_R, respectively. A SAT solver then determines whether $\phi_P \wedge \neg\phi_R$ is satisfiable: if the formula is satisfiable, P violates R; if not, P satisfies R (note that each satisfying assignment to ϕ_P represents a possible execution trace in P).

A brief sketch of the translation process follows [11]. We assume that a given C program is already preprocessed. First, the C program is transformed into a canonical form, containing only `if`, `goto`, and `while` statements without side effect statements such as ++. Then, the loop statements are unwound. The `while` loops are unwound using the following transformation n times (calling n as ULB (Unwinding Loop Bound for a target loop) in this paper):

$$\texttt{while(e) stm} \Rightarrow \texttt{if(e)} \; \{\texttt{stm; while(e) stm}\}$$

After unwinding the loop n times, the remaining `while` loop is replaced by an unwinding assertion `assert(!e)` that guarantees that the program does not execute more iterations. The similar procedure is applied to loops containing the backward `goto` statements. Function calls are inlined and recursive function calls and backward goto statements are unwound in a manner similar to `while` loops. The transformed C program consists of only nested `if`, assignments, assertions, labels, and forward `goto` statements. Finally, this C program is transformed into static single assignment (SSA) form. This SSA program is converted into corresponding bit-vector equations through combinatorial circuit encoding and the final Boolean formula is a conjunction of all these bit-vector equations.

Note that if n (i.e., ULB) is smaller than the minimum number of loop iterations required to exit the loop (calling it MIEL (Minimum Iterations to Exit a target Loop) in this paper), executions after exiting the loop will not be analyzed and property violations can be missed due to the violation of the unwinding assertion (i.e., false negatives). For the following example, suppose that `f(a)` always assigns the first three elements of the array `a` as non-zero values (thus, MIEL of the loop is 3). If we set n as two (i.e., a number of loop unwinding is less than MIEL), assert statement at line 4 will not be reached/analyzed and the assert violation will not be reported consequently.

```
1:f(a); // a[]={1,2,3,...}
2:// no break inside the loop body
3:for(i=0; a[i]!=0; i++) {...}
4:assert(0);
```

Dozens of research papers focus to get loop unwinding upper bounds to obtain sound bounded model checking result. However, related papers often fail to recognize the importance of calculating MIEL to prevent false negatives in practice. Since a sound loop unwinding upper bound is often too large to unwind due to lack of memory and CPU time, calculating MIEL can be more important than calculating sound loop unwinding upper bound in practice. As far as the authors know, this paper is the first one to put emphasis on the significance of MIEL for practical application of bounded model checking techniques.

Note that bounded model checking is incomplete on infinite state systems. Thus, several approaches based on k-induction [27] or interpolation [22] have been studied to make SAT-based bounded model checking complete. Although bounded model checking may be inefficient in the presence of deep loops, it can be used as an effective verification method up to small loop bounds.

B. C Bounded Model Checker (CBMC)

CBMC [11] is a bounded model checker for ANSI-C programs. CBMC receives a C program as its input and analyzes all C statements (e.g., pointer arithmetic, arrays, structs, function calls, etc.) with bit-level accuracy and translates the C program into a SAT formula automatically. A requirement property is written as an assert statement in a target C program. The loop unwinding bound n_l for loop l can be given as a command line parameter; for simple loops with constant upper bounds, CBMC automatically calculates n_l. If $\phi_P \wedge \neg\phi_R$ is satisfiable, CBMC generates a counterexample that shows a step-by-step execution leading to the violation of the requirement property.

One distinct feature of the CBMC based analysis, compared with testing, is its capability of handling *non-deterministic values* (i.e., function parameters, uninitialized local variables, or variables explicitly assigned with non-deterministic values), which are useful in modeling unconstrained user inputs, a range of values as a whole, or return values of undefined functions. Using this feature, CBMC can conveniently analyze all execution scenarios of a target C program.

For example, if we analyze `adder(unsigned char x, unsigned char y) {...}` function, CBMC symbolically analyzes all $65536(= 256^2)$ possible cases. If we provide an explicit constraint `__CPROVER_assume(x==1)`, the total number of cases to analyze is reduced to 256, since only `y` has a non-deterministic value ranging from 0 to 255. This capability of analyzing non-deterministic values helps testing of C functions by reducing the manual effort to explicitly generate test cases. [1]

III. Study Background

This section explains the background of the study including the motivation of the case study and the rationales for applying CBMC as an automated analysis tool to detect bugs in `busybox ls`.

A. Target Software: `busybox ls`

For the last few years, smartphones have been prevalent in our society and become essential equipment for our daily life. Android OS has contributed significantly to the success and evolution of smartphones. As Android OS is developed based on the Linux operating system, many unix/linux utilities are running on Android smartphones. `busybox ls` is one such example and the reliability of `busybox ls` is important since it is used by many other applications such as app managers and file managers.

In addition, since `busybox ls` has clear functional requirements specified in the POSIX specification (IEEE Standard 1003.1 [28]), it can be convenient to determine whether `busybox ls` has a bug. Finally, the size of `busybox ls` is not very large (i.e., around 8200 LOC in 28 files and 63 functions). Thus, we decided to apply automated software analysis techniques to `busybox ls` to evaluate the effectiveness, efficiency and required manual effort of the automated software analysis techniques. We targeted `busybox ls` 1.17.0, which was the latest version when we started to apply automated analysis techniques to `busybox ls`.

Furthermore, we could save time and efforts to apply CBMC to `busybox ls` by utilizing the 15 `assert()`s and the symbolic environment that had been built for our previous case study of applying a concolic testing technique using CREST to `busybox ls` [19]. Furthermore, we can compare the advantages and weaknesses of CBMC and CREST more directly by targeting the same software.

B. Target Automated Technique

We decided to apply a SAT-based bounded software model checking technique using CBMC to detect bugs in `busybox ls` for the following reasons:

1) *No need to build a target program model:*
SAT-based bounded software model checking techniques can be directly applied to target C code using a tool like CBMC. In contrast, other model checkers such as Spin [17], NuSMV [9], or PAT [29] require a user to build a target model in their own modeling languages, which is not feasible in most industrial projects with tight project budget/period.

2) *High maturity of CBMC:*
CBMC can analyze target ANSI C code as it is and generate sound verification result (modulo user-given loop upper bounds). In contrast, other software model checkers targeting C code such as Blast [4] and CPAChecker [5] have limitations in analyzing complex target C code in practice (for example, Blast does not analyze array operations correctly [20]). In addition, CBMC has been developed more than 10 years and become more reliable than other research prototype tools of short development history.

3) *Sufficient scalability for* `busybox ls`*:*
`busybox ls` is written in 8273 C lines, which is a modest size to apply SAT-based software model checking from our previous experience [20]. In addition, we can expect that `busybox ls` might not suffer limited loop bounds severely because the main functionality of `busybox ls` is to align and display information of files and directories. Also, we expected that the loops of `busybox ls` might not be very complex to analyze (Section II).

4) *The authors have sufficient knowledge of CBMC:*
The authors have used CBMC for seven years and have experience to apply CBMC to low-level device driver code on flash memory platform [20]. Thus, we decided to continue to use CBMC since we can configure CBMC experiments and interpret the analysis results precisely and conveniently.

IV. Experiment Setup

In general, to apply CBMC to detect bugs in a target C program, a user should do the following tasks:

1) *Property specification:*
A user specifies requirement properties and writes down `assert()` statements to detect violations of the properties.

2) *Symbolic environment setting:*
A user specifies which variables of the target program to have symbolic/non-deterministic values to model various execution scenarios (and specify constraints on the symbolic variables if necessary)

3) *Loop bound analysis:*
A user analyzes loops of the target program to decide a proper unwinding loop bound (ULB) for each loop which should be greater or equal to MIEL (Minimum Iterations to Exit a target Loop (Section II)).

4) *Apply CBMC with parameters:*
A user executes `CBMC` with enabling/disabling default checkers such as a division-by-zero checker. In addition, a user iteratively executes `CBMC` with increasing num-

[1]A few paragraphs of the background section are excerpted from Kim et al. [20].

```
// if -F is given without -L, the information
// of a target symbolic link file (not a
// referred one) should be printed
assert(
  !( (opt & (1<<21)) && !(opt & (1<<23))) ||
  (!(all_fmt & FOLLOW_LINKS) && !force_follow)
);
```

Fig. 1. Assert to check the behavior of `busybox ls` regarding `-F` option

bers of ULBs for loops until the memory is exhausted or a given time limit is reached.

We describe the detail of these steps to analyze `busybox ls` in the following subsections.

A. Property Specification

First, we reviewed the POSIX specification (IEEE Standard 1003.1 [28]) on `ls` utility (around 10 pages long in A4 paper). The POSIX specification describes the behaviors of `ls` with various command-line options and environment conditions. For example, the POSIX specification requires that, when `-F` option is given, `ls` should print out '`@`' symbol right after a name of a symbolic link to indicate a type of the file. Based on the POSIX specification and our understanding of the target `busybox ls` code, we wrote 15 `assert()` statements to check if `busybox ls` satisfies the POSIX specification.

For example, to check the behavior of `busybox ls` with the `-F` option, we inserted the following `assert()` statement in `my_stat()` (line 299 in `coreutils/ls.c`) (see Figure 1). Suppose that a user executes `busybox ls -F` *slnk* where *slnk* is a symbolic link file pointing to a $referred$ file. `opt & (1<<21)` and `opt & (1<<23)` indicate `-F` and `-L` options, respectively. If `-L` option is given, `ls` should show the information for $referred$. `all_fmt` and `force_follow` are internal variables of `busybox ls`. `all_fmt` indicates how to display the status of file/directory and `FOLLOW_LINKS` is a constant mask to show the status of $referred$ instead of *slnk* (i.e., `!(all_fmt & FOLLOW_LINKS)` means that `busybox ls` should print out the information of *slnk*). `force_follow` is a flag variable to force `busybox ls` to utilize the status of $referred$. Thus, the above `assert()` claims that if `-F` is given without `-L`, the information of *slnk* (not $referred$) should be printed.

B. Symbolic Environment Setting

Since the main functionality of `busybox ls` is to show the status of file/directory according to the format specified through various command-line options, we sets the variables that represent *command-line options* and *file/directory status* as symbolic ones (i.e., variables to have non-deterministic values).

1) Command-line Options: To set the command-line options as symbolic inputs, we set the three `unsigned int` variables `opt`, `tabstops`, and `terminal_width` as symbolic variables, each of which represents enabled command-line options, a number of spaces between columns of output entries, and the width of the current terminal in characters, respectively. `busybox ls` parses the command-line options by calling `getopt32()` which sets the three variables based on the command-line parameter string given by a user. Instead of using the values of the three variables set by `getopt32()`, we set the three variables as *symbolic variables*. For example, to make `opt` have a non-deterministic value, we inserted `opt=non_det();` (`non_det()` returns a non-deterministic value since `non_det()` is undefined) right after the `getopt32()` is called. Other two variables were set as symbolic input in the similar manner.

2) File/directory status: To set the file/directory status as symbolic input, we replaced `stat()` and `lstat()` with `sym_stat()` and `sym_lstat()`. After reading status of a target file/directory from a file system, `stat()` and `lstat()` set `stat` data structure (defined in `sys/stat.h`) that represents the file/directory status (e.g., accessed time, owner, permission, etc). The `stat` data structure has 13 field member variables such as `mode_t st_mode` representing type and permission of file/directory and `uid_t st_uid` representing a user ID of the owner of file/directory. We wrote `sym_stat()` and `sym_lstat()` functions that set all 13 field member variables of `stat` data structure as symbolic input.

C. Loop Bound Analysis

To apply bounded model checking, a user has to decide how many times each loop of a target program should be unwound in the analysis (Section II). This task is complex and requires human effort since the task requires knowledge of target code (calculating an exact loop upper bound is an undecidable problem). In addition, the state explosion problem prohibits a user from using a large unwinding loop bound value. Thus, a user often disables unwind assertion check (e.g., `--no-unwinding-assertions` in CBMC which change an unwinding assertion to an unwinding assumption) and specifies an unwinding loop bound (ULB) as a small number such as one or two without careful loop analysis hoping that a bug might be found within the small state space generated with the small ULBs (see Section VII). However, this way of setting ULBs often makes a bounded model checker miss bugs if ULB of a loop is smaller than MIEL of the loop (see Section II and Section V-C). [2]

Thus, to detect bugs of `busybox ls` effectively, we analyzed loops of `busybox ls`. We first drew a static function call graph from the entry function of `busybox ls` (i.e., `ls_main()`) to collect a list of functions reachable from `ls_main()`. Then, we manually analyzed all 53 loops in the functions and calculated MIELs of the loops.

D. CBMC Parameter Setting and Experiment Platform

To alleviate the state explosion problem, initially we configured CBMC to check only user assertions, not other

[2]CBMC 4.6 provides `--partial-loops` option to continue the analysis by removing unwinding assertions/assumptions although the analysis can be unsound. However, CBMC generated an internal error in our study on `busybox ls` and we could not use the option.

default properties such as array bounds. Once we obtained the verification results, we enabled default checkers one by one to check array bounds (`--bounds-check`), division-by-zero (`--div-by-zero-check`), and arithmetic overflow/underflow (`--signed-overflow-check` and `--unsigned-overflow-check`). In addition, we disabled unwinding assertions (i.e., `--no-unwinding-assertions`) and set ULBs of the loops of `busybox ls` with MIELs+k using `--unwindset` where $k \in \{1, 2, 3\}$. [3] For example, we initially set `--unwindset xrealloc_vector_helper.0:81` (MIEL+1) for the loop whose id is `xrealloc_vector_helper.0` and whose MIEL is 80 (see the last row of Table I). Then, we re-ran CBMC with `--unwindset xrealloc_vector_helper.0:82` (MIEL+2) and `--unwindset xrealloc_vector_helper.0:83` (MIEL+3).

We performed the experiments on the machines equipped with Intel Core i5 3570K@3.8GHz and 16 gigabytes memory running 64bit Debian Linux 6.0.7. We used CBMC 4.6 64bit version. [4]

V. Experiment Result

A. Loop Analysis Result

Among the total 53 loops of `busybox ls`, we found that six loops have zero as their MIELs, 34 loops have one as MIELs, and two loops have three as MIELs, and the remaining 11 loops have four or greater numbers (up to 1320) as MIELs. Table I shows the list of the 11 loops whose MIELs are four or greater. The first column shows the loop ID (specified by CBMC). The second and third columns present a filename and a line number where the loop is located, respectively. The fourth column shows MIELs of the loops. The last column shows the descriptions of the loops. One graduate student spent three days to analyze all loops of `busybox ls`.

For example, the loop `ls_main.2` (the third row of Table I) checks whether or not each of the 26 different command-line options is enabled by checking a list of option flags. Thus, this loop iterates at least 26 times and the corresponding MIEL is 26. [5]

B. Detected Bugs

CBMC detected the four bugs in `busybox ls` regarding the `-F`, `-n`, `-i`, and `-s` options, but did not detect any violation of the default checkers being applied (see Section IV-D).

[3]We did not use k larger than three since we found that the memory was already exhausted with $k = 3$ (Table V).

[4]We tried to generate SMT formulas using CBMC, expecting to achieve faster analysis with less memory consumption, but CBMC raised an internal error.

[5]CBMC can automatically detect upper bounds of unwinding loops for simple loops. However, CBMC could detect upper bounds of unwinding loops for only 18 out of the 53 loops of `busybox ls`.

1) Bug regarding `-F`:: `-F` does not show the status of a symbolic link itself, but the file the symbolic link points to. This bug was detected through the violation of the `assert()` statement described in Section IV-A. The root cause of the bug is that the last parameter of `my_stat()` was incorrect when `ls_main()` called `my_stat()` as follows (in 1151 of `coreutils/ls.c`).

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

If `-F` is given without `-L`, the last parameter of `my_stat()` becomes non-zero (i.e., true in C). This is because `-F` does not set `all_fmt` to indicate `STYLE_LONG` or `LIST_BLOCKS` that indicate to display file/directory information in a long format and to display a block size, respectively. Thus, `force_follow` which is the last formal parameter of `my_stat()` becomes non-zero and `my_stat()` reads the status of the referred file instead of the symbolic link, which violates the POSIX specification.

2) Bug regarding `-n`:: `-n` does not show user id and group id in a numeric format. If `-n` option is given (i.e., `opt & (1<<7)`), `ls` should display user ID and group ID in a numeric format. This bug was detected through violation of the following `assert()` statement in `ls_main()` (line 1142 of `coreutils/ls.c`).

```
assert(!(opt & (1<<7)) ||
  (all_fmt & LIST_ID_NUMERIC));
```

The root cause of this bug is that `busybox ls` misses setting a bit of `all_fmt` for `LIST_ID_NUMERIC` when `-n` is given.

3) Bug regarding `-i`:: `-i` does not show space between adjacent two columns. If `-i` option is given, `ls` should display the corresponding inode number of each file. This bug was detected thorough violation of the `assert()` statement in `showfiles()` function (in line 794 of `coreutils/ls.c`).

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

`nexttab` is the start position of the next column, `tabstops` is the number of spaces between two columns, and `column` is the end position of the current column. The root cause of this bug is that `busybox ls` assumes that the inode number has maximum eight digits when `nexttab` variable is updated. Thus, `busybox ls -i file1 file2` does not show a space if `file1`'s inode number has nine or more digits.

4) Bug regarding `-s`:: `-s` does not show space between adjacent two columns. If `-s` option is given, `ls` should display the corresponding block size of each file. The root cause of this bug is similar to the cause of the bug regarding `-i`. `busybox ls` assumes that the size of block has maximum five digits, but the size of block can have six or more digits, actually.

C. Model Checking Result

Table II summarizes the experimental results. The first column shows the command-line options regarding which

TABLE I
11 LOOPS OF `busybox ls` THAT HAVE FOUR OR GREATER NUMBERS AS MIELS

Loop ID	File	Line	Min. iter. to exit a loop	Description
ls_main.0	ls.c	1032	25	The loop in `memset()` iterates over a global structure G whose size is 25 bytes
ls_main.2	ls.c	1061	26	The loop iterates over an array of 26 options
getopt32.1	getopt32.c	362	1320	The loop in `memset()` iterates over an array `complementary`.
getopt32.3	getopt32.c	373	28	The loop iterates over a string representing possible `busybox ls` options
getopt32.4	getopt32.c	465	27	The loop iterates over `busybox ls` options to check `-T` or `-w` is set
getopt32.5	getopt32.c	501	27	The loop iterates over `busybox ls` options to check `-T` or `-w` is set
getopt32.6	getopt32.c	492	5	The loop iterates over a string `opt_complementary`
getopt32.7	getopt32.c	431	5	The loop iterates over a string `opt_complementary`
getopt32.11	getopt32.c	560	33	The loop iterates over an array `complementary`
bb_verror_msg.0	verror_msg.c	30	16	The loop exists in `strlen()`
xrealloc_vector_helper.0	xrealloc_vector.c	43	80	The loop exists in `memset()`

TABLE II
MODEL CHECKING RESULTS OF `busybox ls`

Opt.	Unwinding loop bound	Assert violated	Time(s)			Mem(MB)	SAT formula statistics	
			Formula generation	Solving	Total		# of variables	# of clauses
`-F`	MIEL+1	Y	246.6	101.1	347.7	4300	8142514	33245688
	MIEL+2	Y	2137.4	8664.4	10801.8	15500	17453685	83456781
	MIEL+3	N/A	N/A	N/A	N/A	OOM	N/A	N/A
`-n`	MIEL+1	Y	240.7	109.7	350.4	4300	8142514	33245688
	MIEL+2	Y	2478	8543.9	11021.9	15600	17453685	83456781
	MIEL+3	N/A	N/A	N/A	N/A	OOM	N/A	N/A
`-i`	MIEL+1	N	251.0	104.5	355.5	4300	8142514	33245688
	MIEL+2	Y	2666.9	8211.7	10878.6	15700	17453685	83456781
	MIEL+3	N/A	N/A	N/A	N/A	OOM	N/A	N/A
`-s`	MIEL+1	N	251.0	104.5	355.5	4300	8142514	33245688
	MIEL+2	Y	2666.9	8211.7	10878.6	15700	17453685	83456781
	MIEL+3	N/A	N/A	N/A	N/A	OOM	N/A	N/A

`busybox ls` has a bug. The second column shows ULBs used. The third column presents a related bug is detected or not. The fourth to sixth columns present SAT formula generation time, SAT solving time, and total execution time in seconds, respectively. The seventh column shows the total amount of memory consumed (we mark Out-Of-Memory (OOM) in the table if the 16 GB memory was exhausted). The eighth and ninth columns show the number of variables and the clauses of the generated SAT formula, respectively.

Increasing ULBs makes CBMC consume significantly more amount of execution time and memory. When we increased ULBs from MIEL+1 to MIEL+2, CBMC's execution time increased almost 10 times to detect bugs (for example, to detect `-F` bug, CBMC executed 246.6 and 2137.4 seconds with ULBs as MIEL+1 and MIEL+2 respectively). Also, memory consumption increased almost four times (for example, to detect `-F` bug, CBMC consumes 4300 MB and 15500 MB with ULBs as MIEL+1 and MIEL+2 respectively).

CBMC detected the two bugs regarding the `-F` and `-n` options, but did not detect the other two bugs regarding `-i` and `-s` options with ULBs as MIELs+1. [6] When we increased ULBs as MIELs+2, the bugs regarding `-i` and `-s` were also detected. In other words, if we used a machine with 8 GB memory, we could not detect the bugs regarding the `-i` and `-s` options because we could not increase ULBs as MIELs+2 due to the memory exhaustion. Also, we could not increase ULBs as MIELs+3 since the whole 16 GB memory was exhausted during the formula generation.

One interesting observation was that CBMC did not detect any bugs when we simply set ULBs of all loops with one value uniformly, although the value is as large as 1300 using `--unwind 1300`. This is because `busybox ls` has a loop

[6]Since the bugs of `-i` option and `-s` option are detected by the same assertion violation, the experimental results of `-i` and `-s` are same.

`getopt32.1` whose MIEL is 1320 and the loop is located near the entry point of `busybox ls` (see the fourth row of Table I). Thus, unless we unwind the loop more than 1320 times, CBMC could not analyze the subsequent executions and could not detect bugs at all.

VI. Lessons Learned

A. Effectiveness of SAT-based Bounded Software Model Checking

We have shown that a SAT-based bounded model checking technique is effective to detect the four corner case bugs (Section V-B), which had not been detected by the manual testing of the original developers of `busybox ls`. Since CBMC detected all four bugs that a concolic testing tool CREST detected [19] with the same assertions and the same environment setting, we can conclude that the bug detection capability of CBMC is as high as that of CREST. However, CBMC did not detect any new bugs which CREST could not detect either. Thus, on this case study, we consider the effectiveness of CBMC and CREST are comparable to each other.

B. Manual Loop Analysis Effort for Bounded Model Checking

As described in Section II and Section IV-C, we had to analyze loops of `busybox ls` to improve the effectiveness of bug detection. In addition to the necessity of calculating sound loop unwinding upper bounds, we have to calculate MIELs to prevent false negatives, which can be more important than calculating the loop upper bounds (see Section II).

For example, as shown in Section V-C, without careful loop analysis, even a large unwinding loop bound value such as 1300 make CBMC fail to detect a bug in `busybox ls`. Thus, although software model checking technique like CBMC can analyze a target C program automatically, still manual effort of a user is required for effective model checking. Since a real-world target program usually has many loops each of which has different characteristics, loop analysis is unavoidable for effective bounded model checking. In contrast, CREST usually does not require manual effort to analyze loops of target code.

C. Limited Scalability of SAT-based Bounded Model Checking

Through the study, we observed that the limited scalability of SAT-based bounded model checking regarding loop unwinding and SAT formula generation is a major obstacle to apply SAT-based bounded model checking to industrial embedded software. For example, if we did not have a machine with large memory (i.e., 16 GB), SAT-based bounded model checking (i.e.,CBMC) would miss two bugs (i.e., the bugs regarding `-i` and `-s` options) in `busybox ls` due to the memory exhaustion during the SAT formula generation (Section V-C). Thus, to achieve higher effectiveness in terms of analysis coverage and bug finding, the SAT formula generation algorithm should be optimized further as the first step to improve the scalability of SAT-based bounded model checking.

D. Weaknesses of SAT-based Bounded Model Checking for Real-world Projects

Through the study, we have observed that SAT-based software model checking has a few weakness as an analysis tool for whole program in industrial setting.

First, CBMC requires manual effort to analyze loops of target code for high bug detection capability (Section VI-B). Since most field engineers are sensitive to the required manual effort to adopt a new technique, they might not favor to apply CBMC to their projects.

Second, CBMC does not report intermediate verification progress explicitly until the whole verification is completed, which may take arbitrarily long time and make field engineers uncomfortable. In industrial projects, it is crucial to measure the intermediate progress of verification and validation (V&V) activities to satisfy the quality goal and the project deadline. Most model checking techniques produces all-or-nothing verification results. In comparison, concolic testing can be more suitable for industrial projects since concolic testing gradually increases the analysis coverage and bug detection capability and continuously reports *intermediate testing results* through generated test cases and their results so far. [7]

Finally, software model checking does not utilize a concrete external environment. For example, `busybox ls` invokes 14 external functions such as `readlink()` which accesses the information of a target file system. We modeled some external functions such as `stat()` and `lstat()` (Section IV-B2) but not all of them due to limited project time. If a user does not explicitly model an external function, CBMC considers the return value of the external function like `readlink()` can be arbitrary (i.e., over-approximation), which can cause false positives (this case study happened to escape this problem) and worsen the state explosion problem. In contrast, concolic testing can invoke external binary library functions and utilize their concrete return values to analyze a corresponding execution path and does not raise false alarms (although testing coverage can be low).

VII. Related Work

CBMC has been applied to various target applications such as mobile sensor network applications [30], [8], embedded OS scheduler [21], embedded software for medical devices [12], automotive systems [23], [25]. Werner and Farago [30] applied CBMC to verify a mobile sensor network protocol called *ESAWN* [7]. Bucur and Kwiatkowska [8] also applied CBMC to model check sensor network applications to check array out-of-bounds access and NULL-pointer dereference. Ludwich and Frohlich [21] used CBMC to verify a process scheduler in embedded operating system written in C++. Cordeiro et al. [12] verified embedded software in a medical device using CBMC and SATABS model checkers. Metta [23] verified CRC (Cyclic Redundancy Check) computation functions in an

[7]Concolic testing analyzes execution paths *one by one independently* and symbolic analysis of each execution path consumes only a small amount of memory and computation time.

automotive system. The author optimized the implementation and verified that the optimized code is equivalent to the original code in terms of functionality. Park et al. [25] applied CBMC to verify OSEK/VDX operating system for automotive systems. To construct environment model for model checking, the authors sliced the target program using the variables in requirement properties.

Most of the above case studies did not perform detailed loop analysis as we did in this case study. Those case studies simply/optimistically set unwinding loop bounds for loops in target code with small values such as one or two and increase the bounds until computational resource was exhausted or given time limit was reached. Thus, their case studies might miss bugs (see Section VI-B). This case study may be the first one to emphasize the necessity of manual loop analysis to avoid hidden false negatives due to an unwinding loop bound (ULB) smaller than the minimum iterations to exit a target loop (MIEL).

VIII. Conclusion

In this project, we applied CBMC that utilizes a SAT-based bounded model checking technique to detect bugs in `busybox ls`. Through the case study, we have confirmed that an automated software analysis technique such as CBMC could successfully detect hidden bugs in real-world software like `busybox ls`. However, this success can be obtained only with non-trivial manual effort due to the necessity of analyzing the loops of a target program. In this study, we demonstrate the importance of calculating MIELs to prevent false negatives in practice, which is one of the contributions of this paper.

As future work, we plan to develop a heuristic to get MIEL automatically because manual effort required to get MIEL can be a large obstacle to adopt bounded model checking techniques in practice.

Acknowledgement

This work is supported in part by the National Research Foundation of Korea (NRF) Mid-career Research Program funded by the Ministry of Science, ICT and Future Planning (MSIP), Korea (NRF-2012R1A2A2A01046172), the IT R&D Program of Ministry of Knowledge Economy (MKE)/Korea Evaluation Institute of Industrial Technology(KEIT), Korea (10041752), the Information Technology Research Center (ITRC) support program funded by MSIP and supervised by the National IT Industry Promotion Agency (NIPA), Korea (NIPA-2014-H0301-14-1023), and the Engineering Research Center of Excellence Program of Korea MSIP/NRF (Grant NRF-208-062609).

References

[1] Busybox home page. http://www.busybox.net/.
[2] Products/projects using busybox. http://www.busybox.net/products.html.
[3] Twelfth intl. conference on theory and applications of satisfiability testing, 2009.
[4] D. Beyer, T. Henzinger, R. Jhala, and R. Majumdar. The software model checker Blast: Applications to software engineering. *Software Tools for Technology Transfer (STTT)*, 2007.
[5] D. Beyer and M. E. Keremoglu. CPAchecker: A tool for configurable software verification. In *Computer Aided Verification*, 2011.
[6] A. Biere, A. Cimatti, E. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. *Advances in Computers*, 58:117–148, 2003.
[7] E. Blaß, J. Wilke, and M. Zitterbart. Relaxed authenticity for data aggregation in wireless sensor networks. In *International Conference on Security and Privacy in Communication Netowrks*, 2008.
[8] D. Bucur and M. Kwiatkowska. On software verification for sensor nodes. *Journal of Systems and Software*, 84(10):35–45, 2011.
[9] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An opensource tool for symbolic model checking. In *Computer Aided Verification (CAV)*, 2002.
[10] E. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design (FMSD)*, 19(1), 2001.
[11] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2004.
[12] L. Cordeiro, B. Fischer, H. Chen, and J. Marques-Silva. Semiformal verification of embedded software in medical devices considering stringent hardware constraints. In *International Conference on Embedded Software and Systems*, 2009.
[13] E. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, January 2000.
[14] N. Eén and N. Sörensson. An extensible SAT-solver. In *Theory and Applications of Satisfiability Testing (SAT)*, 2003.
[15] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Programming Language Design and Implementation (PLDI)*, 2005.
[16] J. Gu, P. W. Purdom, J. Franco, and B. W. Wah. Algorithms for the satisfiability (SAT) problem: A survey. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 1996.
[17] G. Holzmann. *The Spin Model Checker*. Wiley, New York, 2003.
[18] R. Jhala and R. Majumdar. Software model checking. *ACM Computing Surveys*, 41(4):21–74, 2009.
[19] M. Kim, Y. Kim, and Y. Jang. Industrial application of concolic testing on embedded software: Case studies. In *International Conference on Software Testing, Verification and Validation (ICST)*, 2012.
[20] M. Kim, Y. Kim, and H. Kim. Comparative study on software model checkers as unit testing tools: An industrial case study. *IEEE Transactions on Software Engineering (TSE)*, 37(2):146–160, March 2011.
[21] M. K. Ludwich and A. A. Frohlich. On the formal verification of component-based embedded operating systems. In *Brazilian Symposium on Computing Systems Engineering*, 2012.
[22] K. L. McMillan. Interpolation and SAT-based model checking. In *Computer Aided Verification (CAV)*, 2003.
[23] R. Metta. Verifying code and its optimizations: An experience report. In *International Conference on Software Testing, Verification and Validation Workshops*, 2011.
[24] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient sat solver. In *Design Automation Conference*, June 2001.
[25] M. Park, T. Byun, and Y. Choi. Property-based code slicing for efficient verification of osek/vdx operating systems. In *International Workshop on Formal Techniques for Safety-Critical Systems*, 2012.
[26] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE)*, 2005.
[27] M. Sheeran, S. Singh, and G. Stalmarck. Checking safety properties using induction and a SAT-solver. In *Formal Methods in Computer Aided Design (FMCAD)*, 2000.
[28] IEEE Computer Society. Standard for information technology-portable operating system interface (POSIX), 2008.
[29] J. Sun, Y. Liu, J. S. Dong, and J. Pang. PAT: Towards flexible verification under fairness. In *Computer Aided Verification*, 2009.
[30] F. Werner and D. Farago. Correctness of sensor network applications by software bounded model checking. In *International Workshop on Formal Methods for Industrial Critical Systems*, 2010.
[31] L. Zhang and S. Malik. The quest for efficient boolean satisfiability solvers. In *Computer Aided Verification (CAV)*, 2002.