Automated Analysis of Industrial Embedded Software

Moonzoo Kim and Yunho Kim

Computer Science Department
Korea Advanced Institute of Science and Technology (KAIST)
Daejeon, South Korea
moonzoo@cs.kaist.ac.kr, kimyunho@kaist.ac.kr

Abstract. For the last few decades, automated software analysis techniques such as software model checking and concolic testing have advanced in a large degree. 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. Therefore, to enable technology transfer to industry, it is essential to conduct concrete case studies applying automated techniques to real-world industrial software. These studies can serve as references for field engineers who want to improve quality of software by adopting automated analysis techniques. Furthermore, concrete applications of such techniques can guide new research goals and directions to solve practical limitations observed in the studies. In this paper, we describe our experience of applying various automated software analysis techniques to industrial embedded software such as flash memory storage platform and smartphone platform.

1 Introduction

Manual testing is a de-facto standard method to improve the quality of software in industry. However, conventional testing methods frequently fail to detect faults in target programs, since it is infeasible for a test engineer to manually create test cases sufficient to detect subtle errors in specific/exceptional execution paths among an enormous number of different execution paths. These limitations are serious issues in many 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, many researchers have worked to develop automated software analysis techniques such as model checking [5], software model checking [8], and concolic testing (a.k.a., dynamic symbolic execution) [13, 6]. 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, although model checking is a fully automated technique, model creation/extraction is a mostly manual process which causes large cost in an industrial setting. In addition, for software model checking, a user does not have to make a target model unlike model checking, but still he/she has to build a valid environment model to obtain meaningful verification results. Furthermore, to achieve effective and efficient analysis results, a user has to understand the limitations of automated techniques, which are not clearly described in related technical papers. Consequently, field engineers often hesitate to adopt automated analysis techniques in their projects.

To realize the benefits of automated software analysis techniques in practical settings, our group has worked to apply automated analysis techniques such as software model checking and concolic testing (a.k.a. dynamic symbolic execution) to industrial software by collaborating with consumer electronics companies such as Samsung Electronics. Through the collaboration, we realized that it is essential to conduct concrete case studies of applying automated techniques to real-world industrial software. These studies can serve as references for field engineers who want to improve quality of software by adopting automated analysis techniques. Furthermore, concrete applications of such techniques can guide new research directions to solve practical limitations observed in the studies. In this paper, we share our experience of applying various tools of model checking, software model checking, and concolic testing to flash memory storage platform [9, 11, 10] and smartphone platform [12].

2 Unified Storage Platform for OneNAND Flash Memory

2.1 Overview of the Unified Storage Platform

The unified storage platform (USP) is a software solution for OneNAND based embedded systems. Figure 1 presents an overview of the USP: it manages both code storage and data storage. USP allows processes to store and retrieve data on OneNAND through a file system. USP contains a flash translation layer (FTL) through which data and programs in the OneNAND device are accessed. FTL is a core part of the storage platform for flash memory, since logical data can be mapped to separated physical sectors due to the physical characteristics of flash memory (see Section 2.2). FTL consists of the three layers: a sector translation layer (STL), a block management layer (BML), and a low-level device driver layer (LLD).

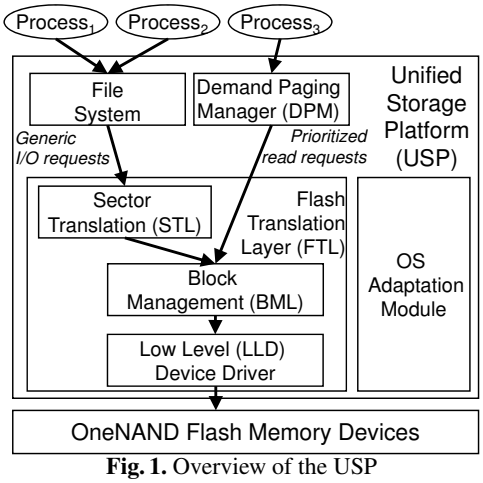

Fig. 1. Overview of the USP

Generic I/O requests from processes are fulfilled through the file system, STL, BML, and LLD, in that order. Although the USP allows concurrent I/O requests from multiple processes through the STL, the BML operations must be executed sequentially, not concurrently. For this purpose, the BML uses a binary semaphore to coordi-

nate concurrent I/O requests from the STL. In addition to generic I/O requests, a process can make a *prioritized read request* for executing a program through the demand paging manager (DPM) and this request goes directly to the BML. A prioritized read request from the DPM can preempt generic I/O operations requested by STL. After the prioritized read request is completed, the preempted generic I/O operations should be resumed again.

2.2 Logical-to-Physical Sector Translation

A NAND flash device consists of a set of *pages* that are grouped into *blocks*. A *unit* can be equal to a block or multiple blocks. Each page contains a set of *sectors*. Operations are either read/write operations on a page, or erase operations on a block. NAND can write data only on an empty page and the page can be emptied by erasing the block containing the page. Therefore, when new data is written to the flash memory, rather than directly overwriting old data, the data is written on empty physical sectors and the physical sectors that contain the old data are marked as invalid. Since the empty physical sectors may reside in separate physical units, one logical unit (LU) containing data is mapped to a linked list of physical units (PU). STL manages the mapping from the logical sectors (LS) to the physical sectors (PS). This mapping information is stored in a sector allocation map (SAM), which returns the corresponding PS offset from a given LS offset. Each PU has its own SAM. Figure 2 illustrates the mapping from logical sectors to physical sectors where one unit contains one block and a block consists of four pages, each of which has one sector.

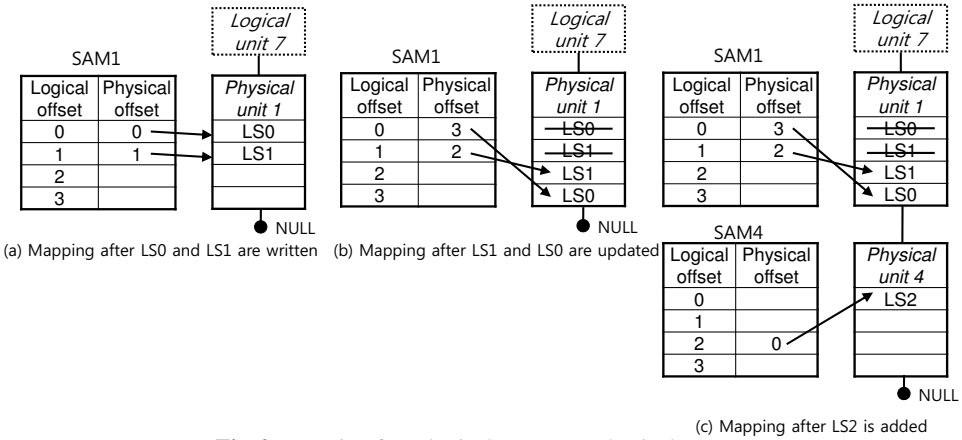

Fig. 2. Mapping from logical sectors to physical sectors

2.3 Analysis Results and Discussions

Multi-sector Read Function We began by analyzing a multi-sector read (MSR) function in STL (see Section 2.2) that reads multiple physical sectors that correspond to logical sectors specified by a user. We selected MSR as it is a core function of USP and relatively small (157 lines of C code) but with complex control (i.e., four-level nested loops) and data structure (i.e., LU, PU, and SAM). The requirement property we checked is that the read buffer of MSR should contain corresponding data in physical

sectors at the end of MSR. In addition, to obtain valid verification results, we had to provide an operational environment of MSR such as following:

1. For each logical sector, at least one physical sector that has the same value exists.
2. If the i_{th} LS is written in the k_{th} sector of the j_{th} PU, then the $(i \ mod \ m)_{th}$ offset of the j_{th} SAM is valid and indicates the PS number k, where m is the number of sectors per unit.
3. The PS number of the i_{th} LS must be written in *only* one of the $(i \ mod \ m)_{th}$ offsets of the SAM tables for the PUs mapped to the $\lfloor \frac{i}{m} \rfloor_{th}$ LU.

We applied model checking techniques to MSR through a symbolic model checker NuSMV [3], an explicit model checker Spin [7], and C-bounded model checker CBMC [4] (more detail can be found in [9]) using 64 bit Linux machine equipped with 3 Ghz Xeon dual-core cpu. For NuSMV and Spin, we built a model for MSR manually. We found that it was a highly challenging task to build a NuSMV model for a C program with complex control and data structure (a corresponding MSR model is 1000 lines long). The above model checkers did not detect a violation of the requirement property in problem instances up to 10 physical units and 6 logical sectors. Figure 3 shows the verification performances of the above model checkers in terms of time and memory. NuSMV spent more than 90% of time in dynamic reordering of BDD variables due to hard-to-abstract SAMs and showed an order-of-magnitude slower speed than Spin. For memory consumption, NuSMV showed better performance than Spin. CBMC showed better performance in terms of both time and memory than Spin and NuSMV. Note that CBMC demonstrated relatively slow increases of time/memory cost as the problem size grows up (i.e., scalability of CBMC is better than NuSMV and Spin due to the underlying industrial-strength SAT solver). Though the verification was conducted on a small-scale, this exhaustive result provided good confidence on the correctness of MSR. Thus, we found that a software model checker could be used as an effective unit-testing tool for embedded software.

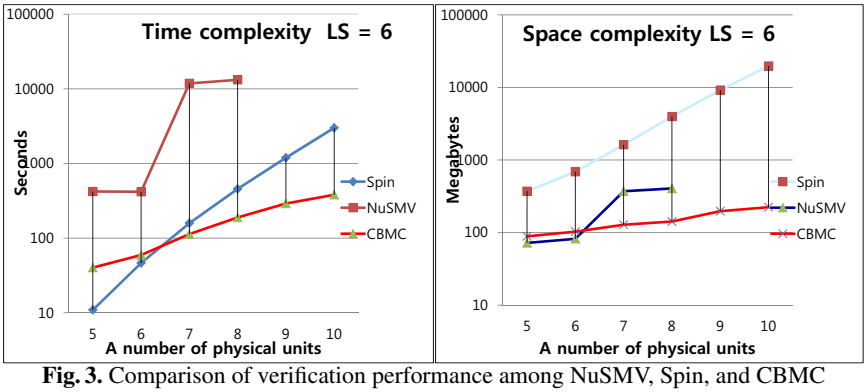

Fig. 3. Comparison of verification performance among NuSMV, Spin, and CBMC

BML and LLD Layers We applied software model checkers Blast [1] and CBMC to several components in the BML and LLD layers (we could not apply Spin and NuSMV, since translation from BML/LLD C code to formal models would require large human effort). In these experiments, we had to build valid environment models for target units

as we did for MSR. We found several bugs including a preemption error caused by a prioritized read operation and an error that does not propagate a BML semaphore exception to STL, which is required to reset USP (Section 2.1). Figure 4 shows a call graph of the topmost STL functions toward BML functions. When a BML function such as `BML_GetVolInfo` raises a semaphore exception for any reason, that exception should be handled by STL functions, but `_GetSInfo` does not pass the exception to its caller in some cases. Total size of all functions from STL to BML is around 2500 lines of C code on average. Blast failed to detect the error and raised false alarms due to its limitations on handling bitwise operators and nested data structures. CBMC detected this error in 12 minutes with consuming 3 Gbyte of memory on average (details of the experiments can be found in [11]).

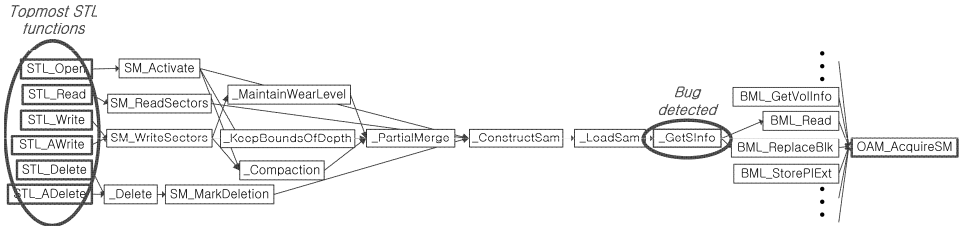

Fig. 4. Call graph of the topmost STL functions using the BML semaphore

In these analyses, however, we found that both Blast and CBMC had limitations for complex embedded C programs. For example, Blast often analyzed array operations incorrectly and its result could not be trusted. In contrast, CBMC did not suffer accuracy problems, but due to its loop unwinding scheme, extensive loop analysis (i.e., unwinding many times) was infeasible. In addition, when a target code invokes external libraries, the analysis accuracy decreases unless a user makes an environment model for such libraries. Consequently, we decided to focus on more scalable and automated analysis techniques and concentrated on concolic testing techniques (see Section 3).

3 Concolic Testing Technique

Concolic (CONCrete + symbOLIC) testing [13, 6] combines both a concrete dynamic analysis and a symbolic static analysis to automatically explore all possible execution paths of a target program by negating every branching decision in execution paths. Thus, concolic testing aims to overcome the limitation of conventional testing as well as software model checking. Concolic testing can analyze a target program with less memory than state model checking, since it does not store the entire state space, but analyzes each execution path one by one in a systematic manner (i.e., through depth first search strategy). In addition, concolic testing can analyze a target program faster than state model checking, since search space of concolic testing (i.e., explicit path model checking) is usually smaller than that of state model checking. Although concolic testing may fail to detect bugs which can be discovered by state model checkers, concolic testing techniques can be a good trade-off between effectiveness and efficiency. In addition, unlike model checking, external library calls can be handled using concrete input and output values, thus achieving better applicability. Lastly, concolic testing generates

concrete test cases, which are invaluable assets for industrial software projects (i.e., through conventional testing, regression testing, and product line testing, etc.).

It is, however, still necessary to check the effectiveness and efficiency of concolic testing on industrial software through concrete case studies, since this technique is relatively new and depends on many other static and dynamic components. These components potentially include SMT solvers, virtual machines, code instrumenters, compilers, etc. In addition, in our experience we found that successful application of concolic testing depends on the expertise of a human engineer, as they must determine what should be declared as symbolic input and what should be the initial input from which symbolic analysis begins, which search strategy should be chosen, etc.

4 Samsung Linux Platform (SLP) for Smartphones

4.1 File Manager

The SLP file manager (FM) monitors a file system and notifies corresponding applications of events in the file system. FM uses an `inotify` system call to register directories/files to monitor. When the directories and files that are being monitored change, the Linux kernel generates `inotify` events and adds these events to an `inotify` queue. FM reads an event from the queue and notifies corresponding programs of the event through a D-BUS inter-process communication interface. For example, when a user adds an MP3 file to a file system, FM notifies a music player to update its playlist automatically. A fault in FM can cause serious problems in SLP, since many applications depend on FM. FM is written in C and around 10,000 lines long.

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

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

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

Analysis Results By using a concolic testing tool CREST [2], we detected an infinite loop fault in FM in one second. After FM reads an `inotify_event` in the queue, the event should be removed from the queue to process the other events in the queue. For a normal event, the `wd` field of the event is positive. Otherwise, the event is abnormal. We found that FM did not remove an abnormal event from the queue and caused an infinite loop when an abnormal event was added to the queue. This bug had not been detected before because original developers did not make test cases that contained abnormal events, which were hard to trigger. Note that external SLP libraries used by FM could be handled by CREST without difficulty (but with decreased path coverage), since CREST simply used concrete values for library calls without building a corresponding symbolic path formula.

4.2 Security Library

The security library in SLP provides API functions for various security applications on mobile platforms such as SSH (secure shell) and DRM (digital right management). The security library consists of security function layer (security APIs such as AES or SHA), complex math function layer (elliptic curve, large prime number generators, etc), and a large integer function layer. Most functions in the library are well documented and its input/output behaviors are clearly defined based on mathematical semantics. We analyzed a large integer function layer (around 2500 lines long) using CREST.

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

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

To test large integer functions, we built a symbolic large integer generator that returns a symbolic large integer `n` (line 12) as shown in Figure 5. Lines 3-5 allocate memory for `n` (line 5). Line 3 declares the `size` of `n` as a symbolic variable of `unsigned char` type. Note that line 4 enforces a constraint on `size` such that `min`$\leq$`size`$\leq$`max`. Without this constraint, `size` can be 255, which will generate unnecessarily many large integers, since the number of generated large integers increases as the `size` increases. Line 5 allocates memory for `n` using `L_INT_Init()`. For simple analysis, we assume that `len==size` (line 6). Lines 8-9 fill out a data array of `n`, if necessary (line 7). Since we assume that `size==len`, we do not allow the most-significant bytes to be 0 (line 10). Using `gen_s_int()`, we developed test drivers for all 14 large integer functions to checks their basic mathematical properties such as $(n1 + n2)\%m == (n2 + n1)\%m$ for `L_INT_ModAdd`$(n1, n2, m)$.

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

Fig. 5. Symbolic large integer generator

Analysis Results We inserted 40 assertions in the 14 large integer functions and found that all 14 large integer functions violated some assertions. CREST running on a Linux machine (3.6Ghz Core2Duo) generates 7537 test cases for the 14 large integer functions in five minutes. For example, `test_L_INT_ModAdd()` generated 831 test cases to test `L_INT_ModAdd()`. 17 of the 831 test cases violated $(n1 + n2)\%m == (n2 + n1)\%m$. We analyzed `L_INT_ModAdd(L_INT d,L_INT n1,L_INT n2,L_INT m)` and found that this function did not check the `size` of `d` (destination). Thus, if the `size` of `d` is smaller than `(n1+n2)%m`, this function writes beyond the allocated memory for `d`, which may corrupt `d` later by other memory writes. This bug had not been caught before, since high level security functions invoked `L_INT_ModAdd()` with `m`

that is smaller than `n1` and `n2`, thus escaping from exceptional error-triggering scenarios. Automated analysis techniques are very effective to explore such corner case scenarios and detect hidden bugs.

5 Conclusion and Future Work

We have shown that difficult verification problems in industrial software can be handled successfully using automated formal analysis tools. Though the projects were conducted on a small-scale, Samsung Electronics highly valued the analysis results. At the same time, the experience gained in these projects led the authors to realize the practical limitations on the scalability and applicability of software model checking and the necessity of conducting further research to develop an advanced concolic testing technique for complex embedded software such as smartphone platforms. Currently, we are working with University of Nebraska to develop a hybrid concolic testing technique that utilizes a genetic algorithm to cover the weaknesses of pure concolic testing. In addition, Samsung Elctronics and KAIST continue collaboration to analyze Android 2.3 platform using concolic testing techniques.

Acknowledgements This research was partially supported by the ERC of Excellence Program of Korea Ministry of Education, Science and Technology(MEST) / National Research Foundation of Korea) (Grant 2011-0000978).

References

1. D. Beyer, T. Henzinger, R. Jhala, and R. Majumdar. The software model checker Blast: Applications to software engineering. *Software Tools for Technology Transfer*, 2007.
2. J. Burnim. CREST - automatic test generation tool for C. `http://code.google.com/p/crest/`.
3. 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*, 2002.
4. 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*, 2004.
5. E. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, January 2000.
6. P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Programming Language Design and Implementation*, 2005.
7. G. Holzmann. *The Spin Model Checker*. Wiley, New York, 2003.
8. R. Jhala and R. Majumdar. Software model checking. *ACM Computing Surveys*, 41(4):21–74, 2009.
9. M. Kim, Y. Choi, Y. Kim, and H. Kim. Formal verification of a flash memory device driver - an experience report. In *Spin Workshop*, 2008.
10. M. Kim, Y. Kim, and Y. Choi. Concolic testing of the multi-sector read operation for flash storage platform software. *Formal Aspects of Computing*. to be published.
11. M. Kim, Y. Kim, and H. Kim. A comparative study of software model checkers as unit testing tools: An industrial case study. *IEEE Transactions on Software Engineering (TSE)*, 37(2), 2011.
12. Y. Kim, M. Kim, and Y. Jang. Concolic testing on embedded software - case studies on mobile platform programs. In *European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE) Industrial Track*, 2011.
13. 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.