Under consideration for publication in Formal Aspects of Computing

Concolic Testing of the Multi-sector Read Operation for Flash Storage Platform Software [1]

Moonzoo Kim, Yunho Kim[1] and Yunja Choi[2]

[1] CS Dept. KAIST, Daejeon, South Korea
E-mail: moonzoo@cs.kaist.ac.kr, kimyunho@kaist.ac.kr
[2] School of EECS, Kyungpook National University, Daegu, South Korea
E-mail: yuchoi76@knu.ac.kr

Abstract.

In today's information society, flash memory has become a virtually indispensable component, particularly for mobile devices. In order for mobile devices to operate successfully, it is essential that flash memory be controlled correctly through flash storage platform software such as the file system, flash translation layer, and low-level device drivers. However, as is typical for embedded software, conventional testing methods often fail to detect hidden flaws in the software due to the difficulty of creating effective test cases. As a different approach, model checking techniques guarantee a complete analysis, but only on a limited scale.

In this paper, we describe an empirical study wherein a *concolic testing* method is applied to the multi-sector read operation for flash storage platform software. This method combines a concrete dynamic execution and a symbolic execution to automatically generate test cases for full path coverage. Through the experiments, we analyze the advantages and weaknesses of the concolic testing approach on the flash storage platform software.

Keywords: Flash memory, concolic testing, unit analysis, and embedded software verification

1. Introduction

Due to the attractive characteristics, such as low power consumption and strong resistance to physical shock, flash memory has become a crucial component for mobile devices. Accordingly, in order for mobile devices to operate successfully, it is essential that the flash storage platform software (e.g., file system, flash translation layer, and low-level device driver) operates correctly. However, conventional testing methods often fail to detect hidden bugs in flash storage platform software, since it is very difficult to create effective test cases that provide a check of all possible execution scenarios generated from complex flash storage platform software. Thus, the current industrial practice of

Correspondence and offprint requests to: Moonzoo Kim, Email: moonzoo@cs.kaist.ac.kr
[1] Preliminary version appeared at SBMF'09 [24]

manual testing does not achieve high reliability or provide cost-effective testing. In another testing approach, randomised testing can save human effort for test case generation, but does not achieve high reliability, because random input data does not necessarily guarantee high coverage of a target program.

These deficiencies of conventional testing incur significant overhead to manufacturers. In particular, ensuring reliability and performance are the two most time-consuming tasks to produce high quality embedded software. For example, a multi-sector read (MSR) function was added to the flash software to improve the reading speed of a Samsung flash memory product [37]. However, this function caused numerous errors in spite of extensive testing and debugging efforts, to the extent that the developers seriously considered removing the feature. Considering that MSR is a core logic for the flash software, research on the effective analysis of MSR is desirable and practically rewarding. In addition, in spite of the importance of flash memory, little research work has been conducted to formally analyze flash storage platforms. Also, most of such work [22, 16, 9] focuses on the specification of file system design, not real implementation.

In this paper, we describe experiments we carried out to analyze the MSR code of the Samsung flash storage platform software using CREST [8], an open source concolic testing tool for C programs. *Concolic (CONCrete + symbOLIC) test* combines both a concrete dynamic analysis and a symbolic static analysis [27, 44] to automatically generate test cases to explore all possible execution paths of the target program, thus overcoming the limitation of conventional testing. Concolic testing might fail to detect bugs which can be uncovered by (state) model checkers, since it is an explicit path model checking technique. However, concolic testing can produce verification result faster with less memory than state model checking, since it does not store whole state space, but analyze each execution path one by one in a systematic manner (i.e., through depth first ordering) [40]. This can be a good trade-off and even more attractive to practitioners. In addition, concolic testing generates concrete test cases, which are invaluable assets for industrial software projects. It is, however, still necessary to check the effectiveness and efficiency of concolic testing on the flash storage platform software through empirical studies. MSR has complex environmental constraints between sector allocation maps and physical units for correct operation (see Section 3.2) and these constraints may cause insufficient path coverage and/or high runtime cost for the analysis when concolic testing is applied. In addition, we compare the empirical results of concolic testing with those yielded by model checking [23].

Furthermore, we performed in-depth empirical analysis on the symbolic path formulas generated from MSR. Although there are many research papers on the algorithms of SMT solvers, few papers provide case studies that explore the application of SMT solvers for concolic testing in detail. We believe that this paper can serve both practitioners and researchers as a reference to figure out the possibilities and limitations of SMT solvers for concolic testing.

The organization of this paper is as follows. Section 2 describes related literature on concolic testing technique and practical application of this technique. Section 3 overviews the flash translation layer and describes multi-sector read operation in detail. Section 4 explains a concolic testing process of CREST. Section 5 describes the experimental results obtained by applying concolic testing to MSR. Section 6 analyzes the symbolic path formulas generated from the concolic testing of MSR in detail. Section 7 discusses observations from the experiments. Section 8 concludes the paper with directions for future work.

2. Related Work

Concolic testing is also known as dynamic symbolic execution, since it utilizes both dynamic testing and symbolic execution [27]. The core idea of concolic testing is to obtain symbolic path formulas from concrete executions and solve them to generate test cases by using constraint solvers (see Section 4). Various concolic testing tools have been implemented to realize this core idea (see [38] for a complete survey). We can classify this work into the following three categories, based on how they extract symbolic path formulas from concrete executions:

1. *Static instrumentation of target programs*
   The concolic testing tools in this group instrument a target source program to insert probes to extract symbolic path formulas from concrete executions at run-time (see Section 4). Many concolic testing tools adopt this approach, since the approach 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 [40], DART [18], CREST [8] and PathCrawler [45] target C programs, while jCUTE [39] targets Java programs.
2. *Dynamic instrumentation of target programs*
   The concolic testing tools in this group instrument a binary target program when it is loaded into memory (i.e., through a dynamic binary instrumentation technique [34]). Thus, even when the source code of a target program is

not available, the target binary program can be automatically tested. In addition, this approach can detect low-level failures caused by a compiler, a linker, or a loader. SAGE [19] is a concolic testing tool that uses this approach to detect security bugs in x86-binary programs.

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[42] targets C# programs that are compiled into Microsoft .Net binaries. KLEE[10] targets LLVM [30] binaries. jFuzz [21] targets Java bytecode on top of Java PathFinder [43].

Although there have been many research articles on concolic testing techniques, there are only a few papers that report the results of empirical studies evaluating the effectiveness of these techniques in practice. Lakhotia et al. [29] applied CUTE (and AUSTIN [28], which is a test case generation tool based on genetic algorithms) to the 387 functions of four C programs (`libogg`, `plot2d`, `time`, and `zile`) and reported testing results. Marri et al. [32] used PEX to test a client program of CodePlex (a source control management system) with an intelligent mock file system. Kim et al. [24] report results of applying CREST to a flash file system. Botella et al. [6] discuss limitations of current concolic techniques in practice and suggests several solutions.

Finally, for automated testing techniques, it is important to employ a proper context when generating test cases for a given target program. Otherwise, invalid test cases (which violate preconditions/assumptions of a target program) might be given to a target program and test results cannot be trusted. In other words, a test engineer should build a testing environment that can feed only valid test cases to a target program (see Section 5.2.1 and Section 5.3.1). For this purpose, concolic testing employs "mock objects" [32]/"environment model" [24] to generate test cases that satisfy "data structure invariants" [40]/"symbolic grammar" [31]/"grammar-based constraints" [17]/"preconditions" [6] of a given target program.

3. Overview of Multi-sector Read Operation

Unified storage platform (USP) is a software solution to operate a Samsung flash memory device [37]. USP allows applications to store and retrieve data on flash memory through a file system. USP contains a flash translation layer (FTL) through which data and programs in the flash memory device are accessed. The FTL consists of three layers - a sector translation layer (STL), a block management layer (BML), and a low-level device driver layer (LLD). Generic I/O requests from applications are fulfilled through the file system, STL, BML, and LLD, in order. MSR resides in STL. [2]

3.1. Overview of Sector Translation Layer (STL)

A NAND flash device consists of a set of *pages*, which are grouped into *blocks*. A *unit* can be equal to a block or multiple blocks. Each page contains a set of *sectors*.

When new data is written to flash memory, rather than overwriting old data directly, 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 this mapping from logical sectors (LS) to 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. [3]

Figure 1 illustrates a process transforming a mapping from logical sectors to physical sectors, where 1 unit consists of 1 block and 1 block contains 4 pages, each of which consists of 1 sector. We use the following notations:

- PU$<i>$and LU$<j>$stands for the $<i>$th physical unit and the $<j>$th logical unit, respectively.
- PS$<i>$and LS$<j>$stands for the $<i>$th physical sector and the $<j>$th logical sector, respectively.
- SAM$<i>$[$<j>$] stands for the $<j>$th offset of the $<i>$th SAM, which is the SAM for PU$<i>$.

[2] Most description in this section is taken from [23].
[3] Section 5.2.1 describes the relationship between these entities of NAND flash memory in detail.

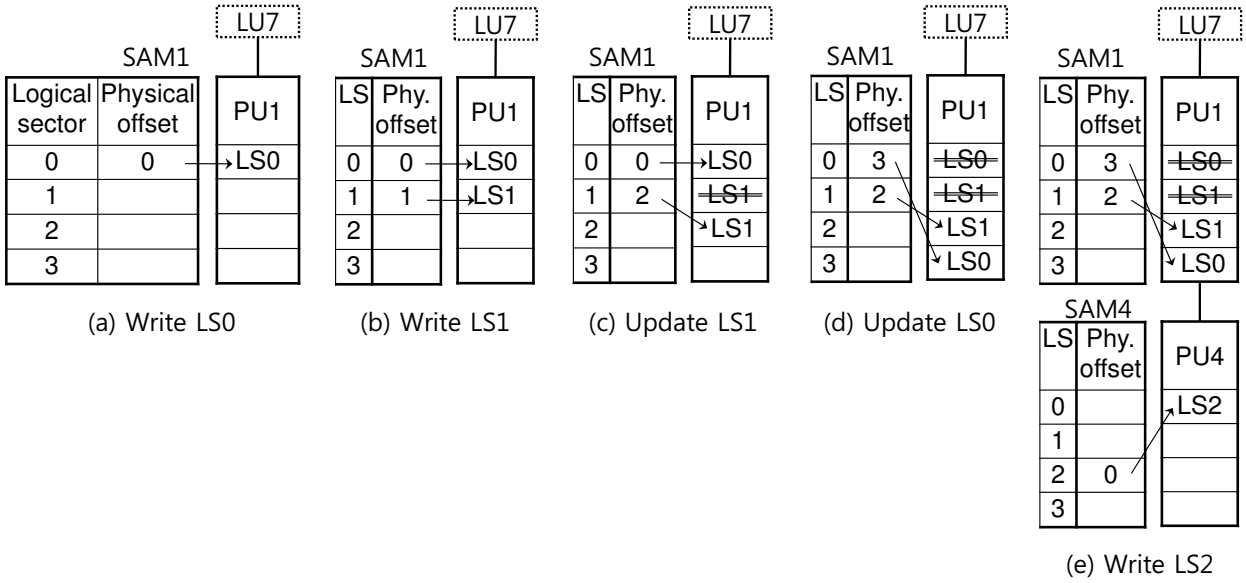


Fig. 1. Mapping from logical sectors to physical sectors

Suppose that a user writes LS0 of LU7. An empty physical unit PU1 is then assigned to LU7, and LS0 is written into PS0 of PU1 (SAM1[0]=0) (Figure 1(a)). The user continues to write LS1 of LU7, and LS1 is subsequently stored into PS1 of PU1 (SAM1[1]=1) which is the next available (i.e., empty) physical sector (Figure 1(b)). The user then updates LS1 and LS0 in order, which results in SAM1[1]=2 and SAM1[0]=3 (see Figure 1(c) and Figure 1(d)). Finally, the user adds LS2 of LU7, which adds a new physical unit PU4 to LU7 (since there is no empty PS in PU1) and yields SAM4[2]=0 (Figure 1(e)).

3.2. Multi-sector Read Operation

USP provides a mechanism to simultaneously read as many multiple sectors as possible in order to increase the reading speed. The core logic of this mechanism is implemented in a single function in STL. The function for MSR is 157 lines long and receives three parameters.

- `buf`: a pointer to the read buffer where read data is stored
- `startLUN`: an index to the starting logical unit to read
- `len`: a number of sectors to read from `startLUN`'th LU

Figure 2 describes a simplified pseudo code of MSR. The outermost loop iterates over LUs of data (lines 8-27) until the `len` amount of the logical sectors are read completely. The second outermost loop iterates until the LSes of the current LU are completely read (lines 11-25). The third loop iterates over PUs mapped to the current LU (line 13-24). The innermost loop identifies consecutive PSes that contain consecutive LSes in the current PU (lines 14-17). This loop calculates `conScts` and `offset`, which indicate the number of such consecutive PSes and the starting offset of these PSes, respectively. Once `conScts` and `offset` are obtained, `BML_READ` rapidly reads these consecutive PSes as a whole and copies these PSes into the read buffer `buf` (line 20).

The requirement property for MSR is that the content of the read buffer should be equal to the original data in the flash memory, when MSR finishes reading. This property can be specified as follows:

$\forall 0 \leq i < \texttt{len}.(\texttt{buf}[i]==\texttt{LS}[i])$

At testing time, this property can be checked through the `assert` statement with the loop at line 29 before MSR returns.

The operation of MSR can be illustrated as follows. Suppose that, as depicted in Figure 3(a), the data is "ABCDEF" and each unit consists of four sectors and PU0, PU1, and PU2 are mapped to LU0 ("ABCD") in order and PU3 and

```
01:// Read the len amount of sectors starting from startLUN'th LU
02:// and store the data into buf
03:MSR(byte *buf, int startLUN, int len) {
04:  curLUN = startLUN;
05:  numScts = len;
06:  pBuf = buf;
07:
08:  while(numScts > 0) {
09:    readScts = # of sectors to read in the current LU
10:    numScts -= readScts;
11:    while(readScts > 0) {
12:      curPU = LU[curLUN]->firstPU;
13:      while(curPU != NULL) {
14:        while(...) {
15:          conScts = # of consecutive PSes to read in curPU
16:          offset = the starting offset of these consecutive PSes
17:        }
18:        // Copy data in the corresponding physical sectors
19:        // into the read buffer
20:        BML_READ(pBuf, curPU, offset, conScts);
21:        readScts = readScts - conScts;
22:        pBuf = buf + conScts;
23:        curPU = curPU->next;
24:      }
25:    }
26:    curLUN++;
27:  }
28:  // check the requirement property
29:  for(i=0;i<len;i++) { assert (buf[i]==LS[i]); }
30:}
```

Fig. 2. Psuedo code of MSR

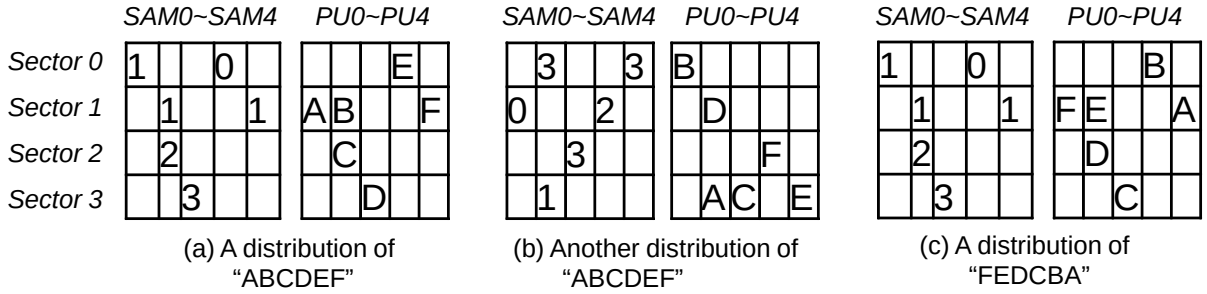


Fig. 3. Possible distributions of data "ABCDEF" and "FEDCBA" to physical sectors

PU4 are mapped to LU1 ("EF") in order. Initially, MSR accesses SAM0 to find which PS of PU0 contains LS0('A'). It then finds SAM0[0]=1 and reads PS1 of PU0. Since SAM0[1] is empty (i.e., PU0 does not have LS1('B')), MSR moves to the next PU, which is PU1. For PU1, MSR accesses SAM1 and finds that LS1('B') and LS2('C') are stored in PS1 and PS2 of PU1 consecutively. Thus, MSR reads PS1 and PS2 of PU1 altogether through `BML_READ` and continues its reading operation.

In this project, we assume that each sector is 1 byte long and each unit has four sectors. Also, we assume that data is a fixed string of distinct characters (e.g., "ABCDE" if we assume that data is 5 sectors long, and "ABCDEF" if we assume that data is 6 sectors long). We can apply this data abstraction without loss of precision, since the values of physical sectors (i.e., data) does not affect the operations of MSR (see Figure 2), but the distribution of data into

Table 1. Total number of the distribution cases

PUs	4	5	6	7	8
$l = 5$	61248	290304	9.8×10^5	2.7×10^6	6.4×10^6
$l = 6$	239808	1416960	5.8×10^6	1.9×10^7	5.1×10^7
$l = 7$	8.8×10^5	7.3×10^6	3.9×10^7	1.5×10^8	5.0×10^8
$l = 8$	3.4×10^6	4.2×10^7	2.9×10^8	1.4×10^9	5.6×10^9

physical sectors does. For example, for the same data "ABCDEF", the reading operations of MSR are different for Figure 3(a) and Figure 3(b), since they have different SAM configurations (i.e., different distributions of "ABCDEF"). However, for "FEDCBA" in Figure 3(c), which has the same SAM configuration as the data shown in Figure 3(a), MSR operates in exactly same manner as for Figure 3(a). Thus, if MSR reads "ABCDEF" in Figure 3(a) correctly, MSR reads "FEDCBA" in Figure 3(c) correctly too.

In addition, we assume that data occupies 2 logical units. The number of possible distribution cases for l LSes and n physical units, where $5 \leq l \leq 8$ and $n \geq 2$, increases exponentially in terms of both n and l, and can be obtained by

$$\sum_{i=1}^{n-1} ({}_{(4\times i)}C_4 \times 4!) \times ({}_{(4\times(n-i))}C_{(l-4)} \times (l-4)!)$$

where ${}_nC_i$ indicates a number of possible ways to get i elements among n elements. For example, if a flash memory device has 5 physical units with data occupying 6 LSes, there exist 1416960 different distributions of the data in total. Table 1 shows the total number of possible cases for 5 to 8 logical sectors and various numbers of physical units, respectively, according to the above formula.

4. Overview of the Concolic Testing Process

This section presents an overview of the concolic testing process which performs static instrumentation of a target program to extract symbolic path formula, which is the way CREST operates (see Section 2). The concolic testing process proceeds in the following six steps:

1. *Declaration of symbolic variables*
   Initially, a user has to 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.
3. *Concrete execution*
   The instrumented program is compiled and executed with given input values. For the first execution of the target program, initial input values are assigned with random values. For the second execution onwards, input values are obtained from step 6.
4. *Obtaining 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 along the concrete execution path. Whenever each statement s of the target program is executed, a corresponding probe inserted at s updates the symbolic 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 complete symbolic path formula ϕ_i is built at the end of the ith execution by combining all path conditions $c_1, c_2, ...c_n$ where c_j is executed earlier than c_{j+1} for all $1 \leq j < n$.
5. *Generating a symbolic path formula ϕ'_i for the next input values*
   Given a symbolic path formula ϕ_i obtained in Step 4, to get the next input values, ϕ'_i is generated by negating

```
01:#include<crest.h>
02:int main() {
03:  int x, y,z, max_num=0;
04:  CREST_int(x); CREST_int(y); CREST_int(z); // symbolic input declaration
05:  if(x >= y) {     // CREST_sym_cond(x,y, ">=");
06:    if(y >= z) {   // CREST_sym_cond(y,z, ">=");
07:      max_num = x;
08:    } else {       // CREST_sym_cond(y,z, "<");
09:      if (x >= z){ // CREST_sym_cond(x,z, ">=");
10:        max_num = x;
11:      } else {     // CREST_sym_cond(x,z, "<");
12:        max_num = z;
13:      }
14:    }
15:  } else { ...}
16:  printf("%d is the largest number among {%d,%d,%d}", max_num, x,y,z);
17:  // CREST_Solve();
18:}
```

Fig. 4. Example for illustrating the concolic testing process

one path condition c_j and removing subsequent path conditions (i.e., $c_{j+1}, ...c_n$) of ϕ_i. For example, if depth first search (DFS) strategy is used, ϕ'_i is generated by negating c_j and removing $c_{j+1}, ...c_n$, where c_j is the last symbolic path condition of ϕ_i whose negated path condition has not been executed before. If ϕ'_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 is no further new path to try, the algorithm terminates.

6. *Selecting the next input values*
   A constraint solver such as a Satisfiability Modulo Theory (SMT) solver [41] generates a model that satisfies ϕ'_i. This model decides next concrete input values and the whole concolic testing procedure iterates from Step 3 again with these input values.

Let us illustrate this process through a simple example of Figure 4, which returns the largest number from given three integer numbers.

1. *Declaration of symbolic variables*
   A user declares `x`, `y`, and `z` as symbolic integer variables by using `CREST_int()` (see line 4).
2. *Instrumentation*
   A concolic testing tool (i.e., CREST) inserts a probe to record a corresponding path condition at each `then` branch in an automated manner. Similarly, at each `else` branch, a probe is inserted to record a corresponding path condition. Note that probes inserted through instrumentation are shown as comments. For example, at line 6, `CREST_sym_cond(y,z,">=")` is inserted to record a path condition $y >= z$. Similarly, `CREST_sym_cond(y,z,"<")` is inserted at line 8 to record a path condition $y < z$.
3. *Concrete execution*
   Initial input values to the symbolic variables are randomly given. We assume that `x`, `y`, and `z` are given 1, 1, and 0 as initial random values, respectively. Then, the instrumented target program executes lines 2-7 and line 16-18.
4. *Obtaining a symbolic path formula ϕ_i*
   During the concrete execution of lines 2-7, the probes record two symbolic conditions `x >= y` and `y >= z` through `CREST_sym_cond(x,y, ">=")` (line 5) and `CREST_sym_cond(y,z, ">=")` (line 6). Thus, obtained symbolic formula ϕ_0 for the first iteration is $x >= y \wedge y >= z$.
5. *Generating a symbolic path formula ϕ'_i for the next input values*
   Since DFS algorithm is used for concolic testing, ϕ'_0 is $x >= y \wedge \neg(y >= z)$.
6. *Selecting the next input values*
   At line 17, the target program finishes its first iteration and invokes a constraint solver to solve ϕ'_0. Suppose that a SMT solver solves ϕ'_0 and generates 1, 1, and 2 for `x`, `y`, and `z` as a solution. Then, the target program starts the second iteration with these values and so forth.

The above algorithm does not raise false alarms, since symbolic path formulas are obtained by executing concrete paths. However, there is a limitation in Step 6. A constraint solver cannot solve complex path formulas to decide next input values; most constraint solvers cannot handle statements containing pointers and non-linear arithmetic. In addition, if a target program updates symbolic variables through external binary library functions, the corresponding symbolic formulas cannot be solved correctly. Therefore, concolic testing may not achieve a full path coverage in practice, which is an ideal coverage for concolic testing. [4]

5. Empirical Study on Concolic Testing MSR

In this section, we describe two series of experiments for concolic testing MSR. Both series of experiments target the same MSR code, but with different environment models - a *constraint-based model* and an *explicit model*. MSR assumes that SAMs and PUs which are given as a test case/input satisfy constraint rules (see Section 3.1 and Section 5.2.1) without checking whether a given input actually satisfies those constraints. Therefore, if an invalid input (which does not satisfy those contraints) is given to MSR, the testing results cannot be trusted. In other words, given an invalid input, MSR might violate the requirement property (see Section 3.2), even when MSR is correct, or vice versa. Therefore, we have to build a testing environment to feed only valid test cases to MSR.

In our experiments, we have the following two hypotheses:

- H_1: Concolic testing is effective for analyzing MSR
- H_2: Concolic testing is more efficient than model checking for analyzing MSR

Regarding H_1, we expect that concolic testing can detect bugs effectively, since it tries to explore all feasible execution paths (we describe how we show this "effectiveness" result in Section 5.1). For H_2, considering that (state) model checking analyzes all possible value combinations of variables, concolic testing (as an explicit path model checker) may analyze MSR faster.

5.1. Testbed for the Experiments

All experiments were performed on 64 bit Fedora Linux 9 equipped with a 3.6 GHz Core2Duo processor and 16 gigabytes of memory. We used CREST [7] as a concolic testing tool for our experiments, since it was the only open source tool based on the static instrumentation at the time of the experiments (see Section 2) and we could obtain more detailed experimental results (especially, regarding the generated symbolic path formulas (see Section 6) by modifying the CREST source code. However, since the CREST project is in its early stage, CREST has several limitations, such as lack of support for dereferencing of pointers and array index variables in the symbolic analysis. Consequently, the target MSR code was modified to use an array representation of the SAMs and PUs. [5] We used CREST 0.1.1 (with DFS search option), gcc 4.3.0, Yices 1.0.19 [13], which is an SMT solver used as an internal constraint solver by CREST. For model checking experiments, CBMC 2.6 [11] and MiniSAT 1.14 [14] were used. This is because CBMC can analyze C programs with arrays in a precise manner if the loop bounds were given [26], which is the case for the MSR experiments. The environment model used for the concolic testing experiments and the model for model checking were almost identical. Model checking experiments were performed on the same testbed as that of concolic testing experiments.

To evaluate the effectiveness of concolic testing, we applied mutation analysis [36, 3] by injecting the following three types of bugs (i.e., mutation operators), each of which has three instances: [6]

1. *Off-by-1 bugs*

[4] To alleviate this limitation, [40] suggests that symbolic constraints are simplified by replacing some of the symbolic values with concrete values. However, this heuristics still does not guarantee to achieve a full path coverage.

[5] The original data structure of MSR code is as follows (see Figure 1). Sectors in a unit are implemented as an array. LUs are implemented as an array. PUs are implemented as a linked list. A LU has a link to the head node of the linked list of PUs which have corresponding data. A PU has a link to its corresponding SAM. In our experiments, due to the difficulty of handling pointers, we modified the environment model to implement PUs as an array and SAMs as an array with the same indexes of the corresponding PUs. Then, MSR code contains only linear integer arithmetic.

[6] Offutt [35] empirically supports a basic premise of mutation testing that a test data set that detects simple faults introduced by mutation, will detect complex bugs (i.e., the combination of several simple faults). In addition, [3] demonstrates that the use of mutation operators is yielding trustworthy results (i.e., generated mutants are similar to real faults) in the eight standard C benchmark programs [12] by using the four classes of mutation operators. In our experiments, we adopted the three classes of them.

$$\forall i, j, k\ (LS[i] = PU[j].sect[k] \Rightarrow (SAM[j].valid[i \bmod m] = true \\ \& SAM[j].offset[i \bmod m] = k \\ \& \forall p.(SAM[p].valid[i \bmod m] = false) \\ \text{where } p \neq j \text{ and } PU[p] \text{ is mapped to} \lfloor \frac{i}{m} \rfloor_{th} LU))$$

Fig. 5. Environment constraints for MSR

- b_{11}: `while(numScts>`**0**`)` of the outermost loop (line 8 of Figure 2) to `while(numScts>`**1**`)`
- b_{12}: `while(readScts>`**0**`)` of the second outermost loop (line 11 of Figure 2) to `while(readScts>`**1**`)`
- b_{13}: `for(i=0;i<conScts; i++)` of `BML_READ()` (line 20 of Figure 2) to `for(i=0;i<conScts-`**1**`;i++)`

2. *Invalid condition bugs*

- b_{21}: `if(SAM[i].offset[j]`**!=**`0xFF)` in the third outermost loop to `if(SAM[i].offset[j]`**==**`0xFF)`
- b_{22}: `readScts=((4-j)>numScts)?numScts:4-j` in the outermost loop to `readScts = ((4-j)<numScts)?numScts:4-j`
- b_{23}: `if((firstOffset+nScts)==SAM[i].offset[j])` in the innermost loop to `if((firstOffset +nScts)!=SAM[i].offset[j])`

3. *Missing statement bugs*

- b_{31}: missing `nScts=1` in the second outermost loop
- b_{32}: missing `readScts--` in the second outermost loop
- b_{33}: missing `curLUN++` corresponding the line 26 of Figure 2

In addition, we injected an artificial corner case bug b_c by changing line 21 of Figure 2 (i.e., `readScts = readScts - conScts`) as follows:

```
readScts = readScts - conScts -
           (PU[1].sect[3]=='A' && PU[0].sect[0]=='B' && PU[2].sect[3]=='C' &&
            PU[1].sect[1]=='D' && PU[4].sect[3]=='E' && PU[3].sect[2]=='F')
```

b_c causes an error only when the configuration of the PUs and SAMs is equal to Figure 3(b). b_c is hard to detect, since the probability of detecting b_c through random test cases is extremely low (e.g., $7 \times 10^{-8} = 1/1416960$ when 6 logical sectors are distributed over 5 PUs (see Table 1)).

5.2. Experiments with Constraint-based Environment Model

5.2.1. Constraint-based Environment Model

MSR assumes that logical data are randomly written on PUs and the corresponding SAMs record the actual location of each LS. The writing is, however, subject to several constraint rules; the following are some of the representative rules. For example, the last two rules can be enforced by the constraints in Figure 5.

1. One PU is mapped to at most one LU.
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 (4 in our experiments).
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.

For example, Figure 3(a) represents the following distribution case:

- LS[0]='A', LS[1]='B', LS[2]='C', LS[3]='D', LS[4]='E', and LS[5]='F'.

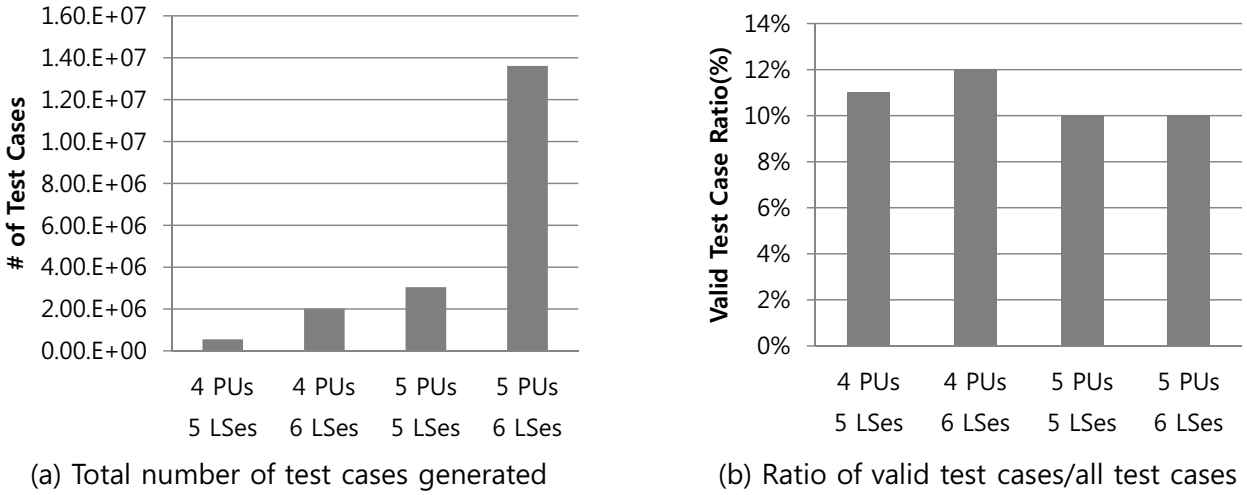


Fig. 6. Generated test cases with constraint-based environment model

- PU[0].sect[1]='A', PU[1].sect[1]='B', PU[1].sect[2]='C', PU[2].sect[3]='D', PU[3].sect[0]='E', and PU[4].sect[1] = 'F'.
- SAM[0].valid[0]=true, SAM[1].valid[1]=true, SAM[1].valid[2]=true, SAM[2].valid[3]=true, SAM[3].valid[0]=true, and SAM[4].valid[1]=true (all other validity flags of the SAMs are false).
- SAM[0].offset[0]=1, SAM[1].offset[1]=1, SAM[1].offset[2]=2, SAM[2].offset[3] =3, SAM[3].offset[0]=0, and SAM[4].offset[1]=1.

Thus, the environment constraints for $i = 2, j = 1$, and $k = 2$ are satisfied as follows:

$$\begin{aligned} LS[2] = PU[1].sect[2] \Rightarrow (&SAM[1].valid[2 \bmod 4] = true \\ \& \; &SAM[1].offset[2 \bmod 4] = 2 \\ \& \; &SAM[0].valid[2 \bmod 4] = false \\ \& \; &SAM[2].valid[2 \bmod 4] = false \\ \& \; &SAM[3].valid[2 \bmod 4] = false) \end{aligned}$$

All elements of the SAMs and PUs are declared as symbolic variables through `CREST_unsigned_char(PU[i].sect[j])` and `CREST_unsigned_char(SAM[i].offset[j])` statements for all valid `i` and `j`. Then, a test driver/environment model checks whether concrete values assigned by CREST to those variables satisfy the constraints in Figure 5. If not, the execution terminates immediately without testing MSR. Note that it is easy to build a constraint-based environment model, since we can apply the constraints in a straight-forward manner. These constraints are encoded as `if` statements in nested loops that handle universally quantified i, j, k, and p.

5.2.2. Experimental Results

Within a time limit of 24 hours, we could performed 4 experiments with 4 to 5 PUs with 5 to 6 LSes. The total numbers of test cases generated and the ratios of the valid test cases over the total test cases are depicted in Figure 6. For example, CREST generated 5.6×10^5 test cases in total for 4 PUs with 5 LSes, and only 61248 test cases (around 11% of the total number of the test cases) among them were valid. The reason for this low efficiency of generating valid test cases was as follows. The constraint-based model generated all possible configurations of PUs and SAMs in an exhaustive manner first. Then, it filtered out invalid test cases by using the environmental constraints. If the constraints are strong/strict, only small portion of the generated test cases are considered as valid ones, which is the case for the MSR experiments. Note that the numbers of the valid test cases for these 4 experiments are equal to the numbers of all possible configurations of the SAMs and PUs (see Table 1). This means that the concolic testing covered all possible execution paths of MSR. Consequently, all injected bugs b_{11} to b_{33} as well as b_c were detected.

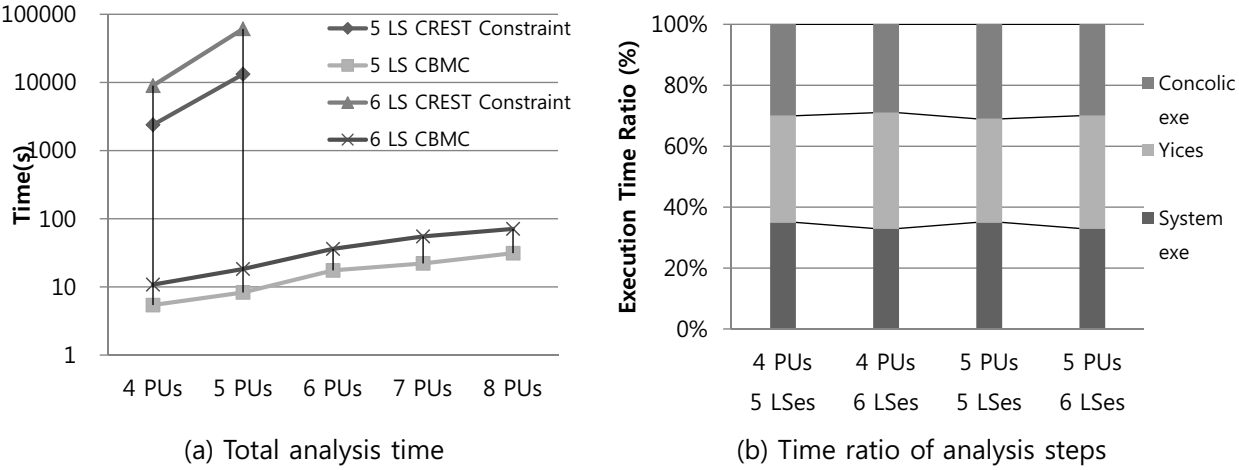


Fig. 7. Analysis time with constraint-based environment model

The performance of the concolic testing is shown in Figure 7. For example, CREST took 2394 seconds for the experiments with 4 PUs and 5 LSes. The amount of time to analyze MSR increased exponentially in terms of the number of PUs and LSes. Figure 7(a) shows that CREST is several hundred to several thousand times slower than CBMC. Figure 7(b) shows that symbolic execution, Yices, and system execution (e.g. launching a target program) take around 30%, 35%, and 35% of the total execution time, respectively. However, all experiments used around 10 megabytes of memory only, since the DFS search in CREST needed only a small amount of information regarding the previous execution path, not the whole execution tree. In comparison, CBMC consumed 40 megabytes and 89 megabytes for 4 PUs with 5 LSes and 5 PUs with 6 LSes, respectively.

5.3. Experiments with Explicit Environment Model

5.3.1. Explicit Environment Model

As we have seen from Figure 6(b), the constraint-based environment model generated too many invalid test cases. Thus, we decided to use an explicit environment model that generates valid test cases *explicitly* by selecting a PU and its sector to contain the l th logical sector (`PU[i].sect[j]=LS[l]`) and setting the corresponding SAM accordingly (`SAM[i].offset[l]=j`). Therefore, most of the generated test cases satisfy the constraints between SAMs and PUs.

However, since CREST does not support symbolic array index variables, we have to modify assignments of SAMs and PUs in the environment model so that these assignments access array elements through constants, not index variables. This workaround solution is depicted in Figure 8. `idxPU` and `idxSect`, which indicate the physical location of the ith logical sector data (`LS[i]`), are declared to be handled symbolically (lines 3 and 4). In the explicit environment model, the `switch` statements starting at line 9 and line 10/17 respectively handle `idxPU` and `idxSect` case by case.

5.3.2. Experimental Results

We performed the 4 experiments with 4 to 5 PUs with 5 to 6 LSes with the explicit environment model, as we did the experiments with the constraint-based environment model. The total numbers of test cases generated and the ratios of the valid test cases over the total test cases are depicted in Figure 9. For example, CREST generated 10^5 test cases in total for 4 PUs with 5 LSes, 61248 test cases (around 60% of the total number of the test cases) among them being valid. In other words, the explicit environment model generated test cases more efficiently compared to the constraint-

```
01:for (i=0; i< NUM_LS; i++){
02:  unsigned char idxPU, idxSect;
03:  CREST_unsigned_char(idxPU);
04:  CREST_unsigned_char(idxSect);
05:  ...
06:  //The switch statements encode the following two statements:
07:  //  PU[idxPu].sect[idxSect]= LS[i];
08:  //  SAM[idxPu].sect[i]= idxSect;
09:  switch(idxPU){
10:    case 0: switch(idxSect) {
11:            case 0: PU[0].sect[0] = LS[i];
12:                    SAM[0].offset[i] = idxSect; break;
13:            case 1: PU[0].sect[1] = LS[i];
14:                    SAM[0].offset[i] = idxSect; break;
15:            ...  }
16:            break;
17:    case 1: switch(idxSect) {
18:             ...
```

Fig. 8. Explicit environment model for MSR

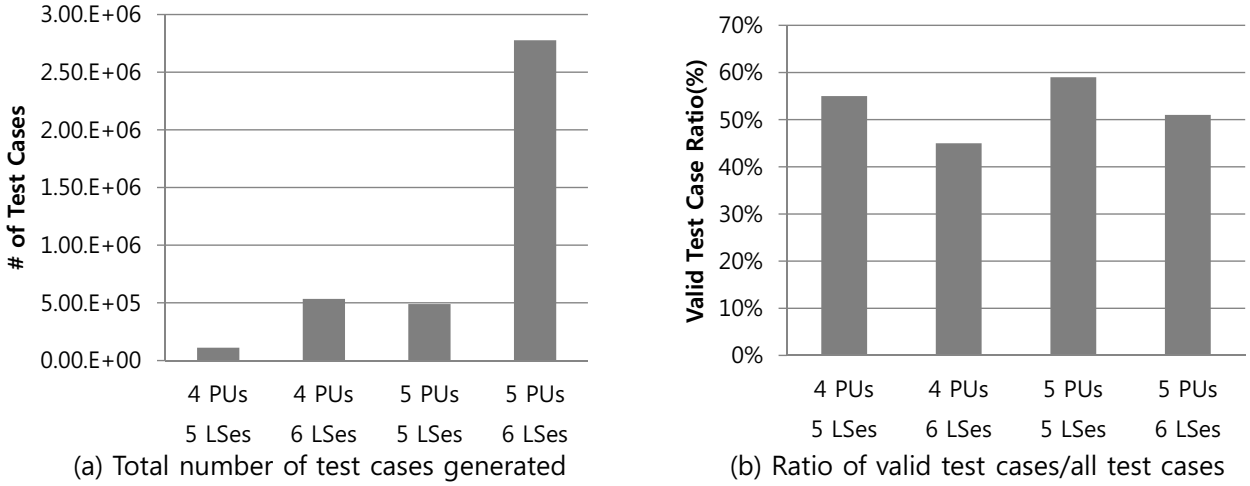
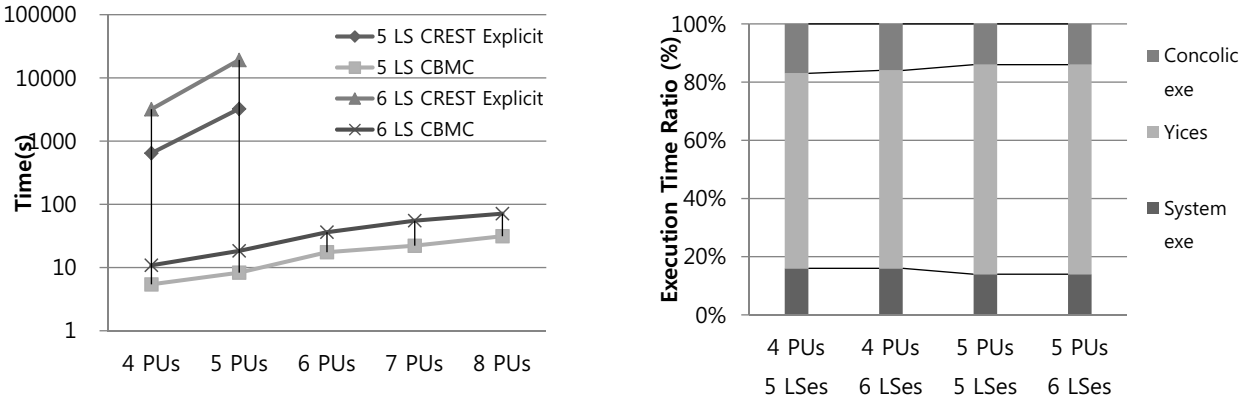


Fig. 9. Statistics on the generated test cases with explicit environment model

based model. [7] Similar to the experiments with the constraint-based model, the numbers of valid test cases for these 4 set of experiments are equal to the numbers of all possible configurations of the SAMs and PUs (see Table 1).

The performance of the concolic testing approach with the explicit environment model is depicted in Figure 10. For example, CREST took 643 seconds for the experiment with 4 PUs and 5 LSes. Although the concolic testing with the explicit model is three times faster than the testing with the constraint-based model, it is still a hundred times slower compared to CBMC (see Figure 10(a)). Yices took around 70% of the total execution time, since invalid test cases are significantly reduced, which thus decreased the portion of symbolic execution time and system execution time.

[7] In general, it is hard to claim that the explicit environment model is always superior than the constraint-based one for concolic testing. This is because the efficiency of the constraint-based one for generating valid test cases depends on the constraints. In addition, explicitly generating inputs which satisfy given constraints/preconditions/grammars is a non-trivial task [17, 15].



(a) Total analysis time (b) Time ratio of analysis steps

Fig. 10. Analysis time with explicit environment model

6. Analysis of the Symbolic Path Formulas

As shown in Figure 10, the major performance bottleneck of conconlic testing of MSR was Yices that solved millions of the symbolic path formulas generated to get test cases. Therefore, it is necessary to analyze the generated path formulas in order to identify advantages and weaknesses of the concolic testing approach which utilizes a SMT solver, and improve the performance of the concolic testing approach. In addition, we show the effectiveness of the reduction techniques of CREST on the path formulas empirically.

6.1. Background on the SMT Path Formulas Generated by CREST

A symbolic path formula ϕ' generated by CREST (see Step 5 of Section 4) is a conjunction of atomic clauses c_1, c_2, ... c_n (i.e., path conditions without boolean connectives), since CREST transforms a target C program P into a canonical form P' so that the condition statements of P' contain atomic conditions, not compound conditions.

An example of a symbolic path formula ϕ' of MSR to get the next test case is shown in Figure 11. [8] ϕ' is a conjuction of 8 path conditions each of which is described in lines 2-9 of Figure 11, respectively. `x3` at line 1 is a symbolic variable name for `idxSect` which indicates an offset of a physical sector containing the first logical sector (i.e., 'A') (see line 4 of Figure 8). Line 2 and line 3 specify that `idxSect` is an 8 bit unsigned integer (see line 4 of Figure 8). Line 4 (i.e., `x3<4`) indicates `idxSect` should be less than a number of sectors per unit (4 in our experiments). Line 5 to line 8 (i.e., `x3≠0`, `x3≠1`, `x3≠2`, and `x3=3`, respectively) correspond to the `switch` statements (see line 10 to line 15 in Figure 8) which test the value of `idxSect`. Finally, line 9 is a negated path condition and it indicates that `idxSect` contains an invalid value (i.e., `x3=255`), which is clearly not true (see the contradiction between line 8 and line 9). Since Yices detects that ϕ' is unsatisfiable, CREST generates another path formula by negating a different path condition of ϕ to create the next test case.

6.2. Path Formula Reduction Techniques of CREST

CREST applies two reduction techniques on the generated path formulas to improve the performance of solving these path formulas.

- *Syntactic contradiction check*:
  Given a generated path formula $\phi' : c_1 \wedge ... \wedge \neg c_n$ with a negated path condition $\neg c_n$, CREST checks whether there exists c_i which is syntactically identical to c_n (i.e., ϕ' is unsatisfiable because c_i is contradictory to $\neg c_n$).

[8] Note that the path formula in Figure 11 is written in the SMT language format for Yices, not SMTLIB format [41]. Symbolic variables in a formula are defined with a keyword `define`. Each clause of a formula starts with a keyword `assert`. Also, each clause is written in prefix notation.

```
1:(define x3::int)
2:(assert (>= x3 0))
3:(assert (<= x3 255))
4:(assert (< (+ -4 (* x3 1)) 0))
5:(assert (/= (+ 0 (* x3 1)) 0))
6:(assert (/= (+ -1 (* x3 1)) 0))
7:(assert (/= (+ -2 (* x3 1)) 0))
8:(assert (= (+ -3 (* x3 1)) 0))
9:(assert (= (+ -255 (* x3 1)) 0))
```

Fig. 11. An SMT path formula ϕ' generated by CREST

Table 2. Statistics on the number of the path formulas

	With the constraint-based environment model				With the explicit environment model			
($\times 10^6$)	4 PUs w/ 5 LSes	4 PUs w/ 6 LSes	5 PUs w/ 5 LSes	5 PUs w/ 6 LSes	4 PUs w/ 5 LSes	4 PUs w/ 6 LSes	5 PUs w/ 5 LSes	5 PUs w/ 6 LSes
# of test cases	0.56	2.01	3.05	13.61	0.11	0.53	0.49	2.78
# of total path formulas	3.17	12.87	17.72	88.12	0.86	3.94	4.55	25.24
# of path formulas removed	1.52	6.07	8.88	43.41	0.10	0.37	0.74	3.31
# of remaining path formulas	1.66	6.80	8.85	44.71	0.76	3.57	3.81	21.93
Reduction ratio	48%	47%	50%	49%	12%	9%	16%	13%

For example, given a $\phi' : x = 0 \wedge ... \wedge x \neq 0$ with $x \neq 0$ as $\neg c_n$, CREST detects that ϕ' is unsatisfiable because $c_1(x = 0)$ is identical to $c_n(x = 0)$ and removes ϕ'.

- *Slicing for the negated path condition*:
  Suppose that c_j of ϕ is to be negated to generate ϕ'. Then, ϕ' consists of $\neg c_j$ and *only* path conditions of ϕ which are dependent on c_j through variables in terms of satisfiability [40]. CREST invokes Yices on this simplified ϕ' and get a solution for those variables. Thus, the next input values are the same as the previous input values except the variables in the solution. For example, given $\phi : a < b \wedge c < d \wedge d < e \wedge e < f$ with $e < f$ as a path condition to negate, CREST generates a path formula for the next test case as $\phi' : c < d \wedge d < e \wedge \neg(e < f)$ without $a < b$, since $a < b$ is not dependent on e nor f. Note that this technique utilizes the fact that path formulas share many path conditions in common.

In our experiments, these simple reduction techniques are effective to decrease the number of the path formulas and to reduce the sizes of the path formulas (see Table 2 and Table 3, respectively).

6.3. Statistics on the Path Formulas

Table 2 shows statistics on the number of the generated path formulas and the number of the path formulas removed by the syntactic contradiction check. For example, for MSR with the constraint-based model on 4 PUs and 5 LSes, 0.56×10^6 test cases are generated from 3.17×10^6 path formulas in total (the second column of Table 2). Note that the number of the path formulas is larger than the number of the test cases (approximately 6:1 for MSR with the constraint-based model and 9:1 for MSR with the explicit model), since test cases are generated from only satisfiable path formulas. Also note that the syntactic contradiction check technique removes, on average, 49% and 13% of the path formulas for MSR with the constraint-based model and MSR with the explicit model, respectively (see the last row of Table 2).

Table 3 shows statistics on the size (a number of path conditions in a path formula) of the remaining path formulas after the syntactic contradiction check is applied and the number of the path conditions removed by the slicing technique. For example, for MSR with the constraint-based model on 4 PUs and 5 LSes, an original path formula contains 194.09 path conditions on average. After applying the slicing technique, 187.52 out of 194.09 path conditions

Table 3. Statistics on the size of the path formulas

	With the constraint-based environment model				With the explicit environment model			
	4 PUs w/ 5 LSes	4 PUs w/ 6 LSes	5 PUs w/ 5 LSes	5 PUs w/ 6 LSes	4 PUs w/ 5 LSes	4 PUs w/ 6 LSes	5 PUs w/ 5 LSes	5 PUs w/ 6 LSes
Avg # of path conditions in one path formula	194.09	220.45	241.11	274.65	67.38	79.60	71.34	84.12
Avg # of reduced path conditions in one path formula	187.52	213.26	234.43	267.38	52.25	63.07	54.06	65.53
Avg # of remaining path conditions in one path formula	6.56	7.19	6.67	7.27	15.12	16.54	17.28	18.59
Reduction ratio	97%	97%	97%	97%	78%	79%	76%	78%

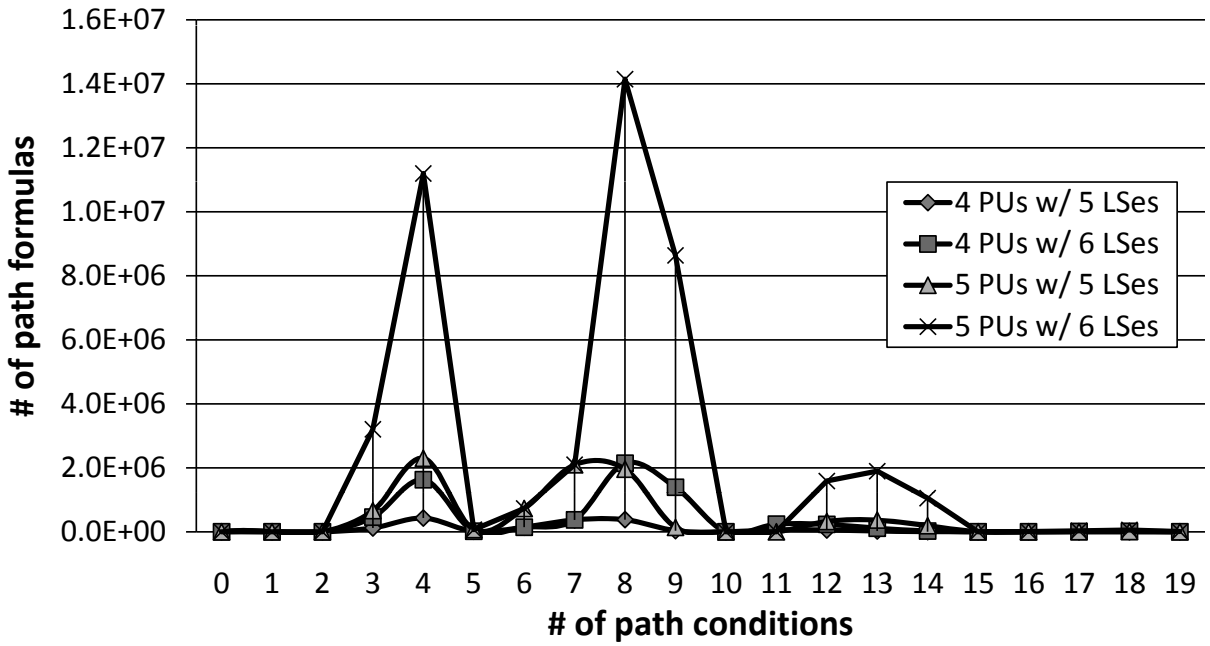


Fig. 12. Distribution of the path formula sizes for MSR with the constraint-based environment model

are removed and only 6.56 path conditions remain in the path formula (see the second column of Table 3). Note that the slicing technique was quite effective, as this technique removed roughly 97% and 78% of the path conditions on average (see the last row of Table 3) with the constraint-based model and MSR with the explicit model, respectively. As a result, the size of the path formulas after applying the slicing technique is small (less than 19 path conditions).

Figure 12 and Figure 13 illustrate the distribution of the sizes of the reduced path formulas (i.e., the remaining path formulas after the applying syntactic contradiction check and the slicing technique) for MSR with the constraint-based model and MSR with the explicit model, respectively. For example, MSR with the constraint-based model on 5 PUs with 6 LSes generates 1.41×10^7 path formulas which contain 8 path conditions. In other words, 32% of the remaining path formulas (=$1.41 \times 10^7/44.71 \times 10^6$ (see the intersection of the sixth row and the fifth column of Table 2)) for MSR with the constraint-based model have 8 path conditions each. [9]

[9] The gap between the number of the path formulas for 5 PUs with 6 LSes and the number for 5 PUs with 5 LSes is large, since a number of possible test cases increases exponentially in terms of both a number of PUs and a number of LSes (see Table 1).

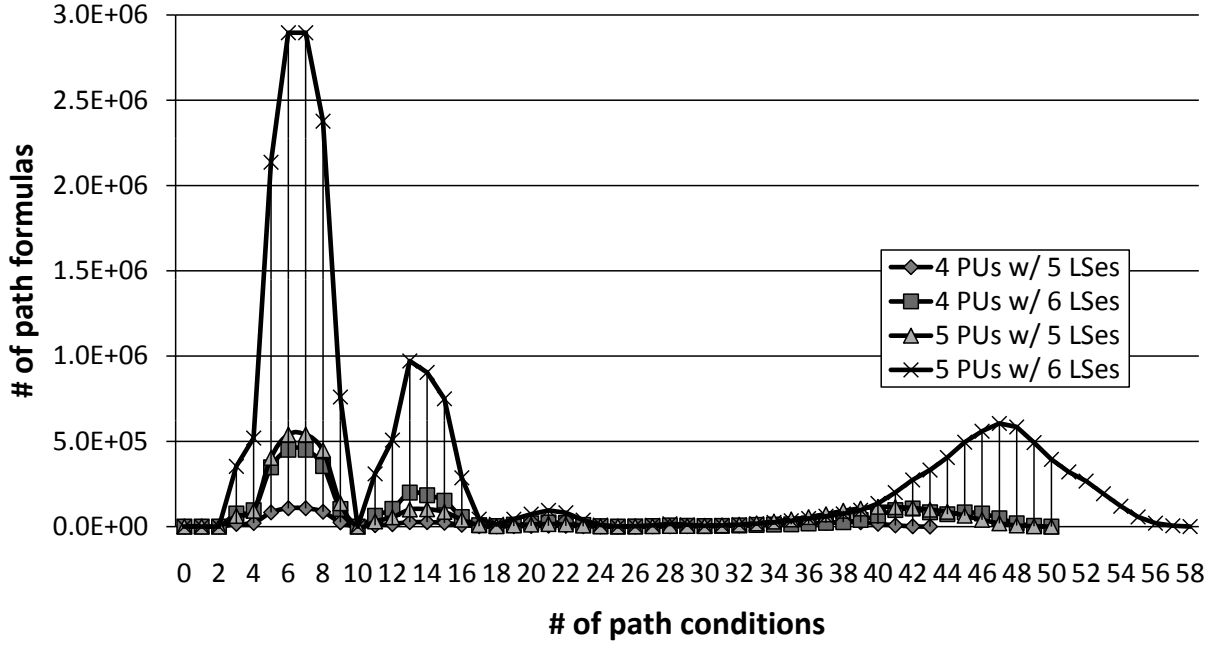


Fig. 13. Distribution of the path formula sizes for MSR with the explicit environment model

7. Discussion

In this section, several issues are discussed on the basis of our experience of applying concolic testing to MSR.

7.1. Importance of Environment Model

Through the various experiments carried out to analyze MSR, including conventional testing [25], model checking [23], and concolic testing [24], we found that it is important to build an accurate and efficient environment model for the analysis of the flash storage platform software. Also, it was found that different analysis techniques can commonly use the same environment model. For example, the constraint-based environmental model (see Section 5.2) was originally designed for model checking through CBMC and used as-is with nominal modification. Similarly, the explicit environmental model was originally designed for model checking through SPIN [20]. We used this environmental model for SPIN with slight modification due to the limitation of CREST (i.e., array index variables are not symbolically handled). Furthermore, the design of the environment model substantially affects the analysis performance (see Section 5.2.2 and Section 5.3.2).

7.2. Poor Performance of CREST

Although our hypothesis H_1 is accepted through the empirical study (i.e., the concolic testing method demonstrates capability of detecting bugs through the full path coverage), H_2 is rejected (i.e., its performance on MSR is worse than the performance of model checking MSR by CBMC (see Figure 7(a) and Figure 10(a))). This poor performance was caused by the several steps of the concolic testing process in Section 4.

First, for a target program with a complex environmental model such as MSR, concolic testing may waste a large amount of time to generate invalid test cases. In the experiments with the constraint-based environment model and the explicit environment model, around 90% and 45% of the total test cases generated were invalid, respectively (see Figure 6(b) and Figure 9(b)). Considering a unit under testing often has preconditions or constraints enforced by its interacting components, concolic testing frameworks should provide an efficient way to control the generation of concrete input values so as to generate only valid test cases [31, 17]. Second, in the concolic testing process which obtains symbolic path formulas through the inserted probes, concolic execution (see Step 3 and Step 4 of the process in

Table 4. Number of test cases generated to detect the bugs $b_{11} - b_{33}$

	With the constraint-based environment model				With the explicit environment model			
	4 PUs 5 LSes	4 PUs 6 LSes	5 PUs 5 LSes	5 PUs 6 LSes	4 PUs 5 LSes	4 PUs 6 LSes	5 PUs 5 LSes	5 PUs 6 LSes
b_{11}	84	N/A	103	N/A	30	N/A	30	N/A
b_{12}	70	102	95	126	25	37	30	34
b_{13}	78	85	95	107	25	37	35	34
b_{21}	78	85	95	121	30	37	30	42
b_{22}	78	92	87	114	34	40	25	45
b_{23}	121	99	231	131	34	33	25	34
b_{31}	78	98	103	121	34	37	30	45
b_{32}	78	89	95	112	101	496	109	345
b_{33}	70	85	109	107	25	33	30	34

Section 4) causes overhead, since each original statement is supplemented with a probe recording a concrete execution in a symbolic manner; around 30% and 15% of the total concolic testing time were spent for the concolic executions with the constraint-based model and the explicit model, respectively (see Figure 7(b) and Figure 10(b)). Lastly, for CREST, the performance of Yices was not fast enough to solve millions of path formulas in a modest amount of time, although the path formulas of MSR, which contains linear integer arithmetic only, can be solved rapidly by many efficient algorithms [5]. [10] Therefore, from our experiments, we can conclude that CREST has large room to improve its performance for practical usage.

7.3. Future Directions of Concolic Testing

Although concolic testing approach has several advantages, it shows poor performance in our experiments. We can suggest a few future directions of concolic testing and SMT solvers in this regard.

First, the main reason of the poor performance of concolic testing is that significant amount of time is spent to solve an enormous number of path formulas (see Table 2) to explore all possible execution paths on the specified input data domain (in the MSR experiments, the data domain was limited to 5 PUs and 6 LSes). However, if we use concolic testing as a bug detecting technique, not verification technique, concolic testing can produce testing results fast. For example, Table 4 shows the number of test cases generated until the inserted bugs $b_{11} - b_{33}$ (see Section 5.1) are detected. [11] Most of them were detected in a few seconds through the first few hundred test cases. For example, MSR with the constraint-based model on 4 PUs and 5 LSes, b_{11} is detected after generating 84 test cases, which takes less than 1 second (see the second column of Table 4). Considering that concolic testing can analyze a target program with binary libraries or with physical environment, concolic testing can be effectively used as an automated bug detecting technique in practice.

Secondly, for the concolic testing approaches based on SMT solvers, it is important to apply effective reduction techniques on symbolic path formulas before solving them by using an SMT solver. Since SMT solvers are designed to solve general SMT formulas, they do not explicitly utilize the characteristics of path formulas, thus losing a chance to improve the performance. In contrast, as shown in Table 2 and Table 3, two simple reduction techniques which exploit the characteristics of the path formulas improve the concolic testing performance significantly. Therefore, it can be an interesting research topic to develop effective reduction techniques, especially by exploiting the relationship between multiple path formulas. In addition, it is also important to optimize the SMT solvers for concolic testing. SMT solvers are often optimized to get high score on SMT competition [4], whose benchmarks are big instances of complex formulas. For example, the QF_LIA benchmark `nec-smt/large/checkpass/prp-19-47.smt` [1] in SMT-COMP 2009 is 17000 lines long with 62 variables. In the concolic testing using CREST, however, we have to solve a huge number of short path formulas. Therefore, the concolic testing approaches based on SMT solvers can be improved further by optimizing SMT solvers to solve many small formulas quickly.

Finally, it can be also interesting to pursue weaker but more practical coverage criteria (i.e., branch coverage,

[10] We modified CREST to use Z3 2.0 [33] instead of Yices, but no significant performance difference was observed.
[11] b_{11} does not violate the given `assert` statement when MSR is tested on 4 PUs with 6 LSes or 5 PUs with 6 LSes.

Table 5. Comparison of concolic testing and model checking

	Accuracy	Analysis speed	Memory usage	Scalability	Applicability	Manual effort
Concolic testing	High	Slower	Low	Large	High	Middle
Model checking	Highest	Slow	Highest	Small	Low	High

prime path coverage [2], k-paths coverage [45], etc.) than path coverage criteria. Although path coverage is effective to detect bugs, it requires large amount of time to achieve. In addition, branch/statement coverages are still most commonly used criteria in industries. Thus, developing new concolic algorithms for various coverage criteria can be a promising direction. For example, CREST [8] employs several concolic testing algorithms (i.e., CFG algorithm, random algorithm, and uniform random algorithm) for branch coverage.

7.4. Comparison with Model Checking

Concolic testing can be considered as a light-weight model checking method, since it generates test cases to explore all possible execution paths (i.e., explicit path model checking [40]). However, these two different analysis techniques have as many different characteristics as common characteristics. Table 5 compares these techniques briefly based on our experience, although this comparison result might not be applicable to other target applications.

In general, model checking provides more accurate verification results than concolic testing does. Also, constraint solvers used for concolic testing are not advanced enough to solve a large number of symbolic path formulas efficiently. However, in terms of memory usage, concolic testing does not store the whole execution tree, but only a single execution trace, which consumes a small amount of memory only. For the same reason, concolic testing can be applied to a large target program. In terms of applicability, concolic testing has clear advantages, since it can analyze a target program with external binary libraries and physical environment such as network and file systems. For the same reason, concolic testing requires less manual effort than model checking, since model checking needs abstract models of external binary libraries or the physical environment.

7.5. Hard Characteristics of MSR for Concolic Testing

Through the project, it was found that MSR is a hard instance for concolic testing. Concolic testing can efficiently analyze programs whose data domain can be easily abstracted. For example, concolic testing can analyze binary search programs or sort programs quickly. The data domain of MSR (especially SAMs), however, cannot be abstracted, since every different value in every single element of SAMs leads to a unique execution path. Thus, as shown in Section 5.2.2 and Section 5.3.2, the total number of valid test cases generated is exactly the same as the number of all possible configurations of the PUs and SAMs (i.e., all possible states) (see Table 1). In other words, in the analysis of MSR, concolic testing is burdened by as much complexity as state model checking.

8. Conclusion and Future Work

We reported our experience of applying a concolic testing method to analyze MSR, and analyzed the strengths and weaknesses of the approach empirically. Although several goals of the concolic testing method could be achieved through the experiments (e.g., automated test case generation, full path coverage, and detection of bugs), CREST suffered a few limitations including slow analysis speed and lack of support for array index variables. We expect that CREST will be able to overcome these limitations in the near future.

As future work, we plan to develop new reduction techniques on path formulas and algorithms to generate the next set of path formulas to improve concolic testing as a bug detection technique as suggested in Section 7.3. In addition, we plan to build a flash file system model that can be used by file-system-dependent applications in a concolic testing framework. One inspiring related work was carried out by Microsoft [32], where an intelligent mock object (an environment model in our terminology) for a file system was developed to test target applications in the PEX framework [42]. The mock file system automatically generates various possible test cases necessary to test applica-

tions, which can save significant effort to test file-system-dependent applications. For this project, we will work on open-source file systems to avoid limitations caused by the intellectual property agreement enforced by the company.

Acknowledgments

We would like to thank Hotae Kim at Samsung Electronics for his valuable discussion on the environment models for flash file systems. This work was supported by the Engineering Research Center of Excellence Program of Korea Ministry of Education, Science and Technology(MEST)/ National Research Foundation of Korea(NRF) (grant number 2010-0001727) and the MKE(Ministry of Knowledge Economy), Korea, under the ITRC(Information Technology Research Center) support program supervised by NIPA(National IT Industry Promotion Agency) (NIPA-2009-(C1090-0902-0032)).

References

[1] QF_LIA benchmark: nec-smt/large/checkpass/prp-19-47.smt, SMT-COMP 2009. `http://www.smtexec.org/exec/benchmarkResults.php?jobs=529\&benchmark=1087660`.

[2] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge press, 2008.

[3] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *International Conference on Software Engineering (ICSE)*, 2005.

[4] C. Barrett, L. de Moura, and A. Stump. SMT-COMP: Satisfiability Modulo Theories Competition. *Computer Aided Verification*, pages 20–23, 2005.

[5] S. Berezin, V. Ganesh, and D. L. Dill. An online proof-producing decision procedure for mixed integer linear arithmetic. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2003.

[6] B. Botella, M. Delahaye, S. Hong-Tuan-Ha, N. Kosmatov, P. Mouy, M. Roger, and N. Williams. Automating structural testing of c programs: Experience with pathcrawler. In *Automation of Software Test*, 2009.

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

[8] 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.

[9] A. Butterfield, L. Freitas, and J. Woodcock. Mechanising a formal model of flash memory. *Science of Computer Programming*, 74(4):219–237, 2009.

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

[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] H. Do, G. Rothermel, and S. Elbaum. Infrastructure support for controlled experimentation with software testing and regression testing techniques. Technical report, January 2004. Oregon State Univ. Technical report 04-06-01.

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

[14] N. Een and N. Sorensson. An extensible sat-solver. In *SAT 2003*, 2003.

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

[16] M. A. Ferreira, S. S. Silva, and J. N. Oliveira. Verifying Intel flash file system core specification. In *4th VDM-Overture Workshop*, 2008.

[17] P. Godefroid, A. Kiezun, and M. Y. Levin. Grammar-based whitebox fuzzing. In *Programming Language Design and Implementation (PLDI)*, 2008.

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

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

[20] G. Holzmann. *The Spin Model Checker*. Wiley, New York, 2003.

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

[22] E. Kang and D. Jackson. Formal modeling and analysis of a flash filesystem in Alloy. In *Abstract state machines, B and Z*, 2008.

[23] M. Kim, Y. Choi, Y. Kim, and H. Kim. Formal verification of a flash memory device driver - an experience report. In *Spin Workshop*, 2008.

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

[25] M. Kim, Y. Kim, Y. Choi, and H. Kim. Pre-testing flash device driver through model checking techniques. In *IEEE Int. Conf. on Software Testing, Verification and Validation*, 2008.

[26] 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)*. to appear.

[27] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7), 1976.

[28] K. Lakhotia, M. Harman, and P. McMinn. Handling dynamic data structures in search based testing. In *Genetic and evolutionary computation*, 2008.

[29] K. Lakhotia, P. McMinn, and M. Harman. Automated test data generation for coverage: Haven't we solved this problem yet? In *Testing: Academic & Industrial Conference Practice and Research Techniques*, 2009.
[30] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation, 2004.
[31] R. Majumdar and R. Xu. Directed test generation with symbolic grammars. In *Automated Software Engineering (ASE)*, 2007.
[32] M. Marri, T. Xie, N. Tillmann, J.de Halleux, and W. Schulte. An empirical study of testing file-system-dependent software with mock objects. In *Automation of Software Test*, 2009.
[33] L. Moura and N. Bjorner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.
[34] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Programming Language Design and Implementation (PLDI)*, 2007.
[35] A. J. Offutt. Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering and Methodology*, 1(1):3–18, 1992.
[36] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf. An experimental determination of sufficient mutation operators. *ACM Transactions on Software Engineering and Methodology*, 5(2):99–118, 1996.
[37] Samsung OneNAND fusion memory. `http://www.samsung.com/global/business/semiconductor/products/fusionmemory/Products_OneNAND.html`.
[38] C. Pasareanu and W. Visser. A survey of new trends in symbolic execution for software testing and analysis. *Software Tools for Technology Transfer (STTT)*, 11(4):339–353, 2009.
[39] K. Sen and G. Agha. CUTE and jCUTE : Concolic unit testing and explicit path model-checking tools. In *Computer Aided Verification (CAV)*, 2006.
[40] 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.
[41] SMT-LIB: The satisfiability module theories library. `http://combination.cs.uiowa.edu/smtlib/`.
[42] N. Tillmann and W. Schulte. Parameterized unit tests. In *European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE)*, 2005.
[43] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *Automated Software Engineering (ASE)*, September 2000.
[44] W. Visser, C. S. Pasareanu, and S. Khurshid. Test input generation with Java PathFinder. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2004.
[45] N. Williams, B. Marre, P. Mouy, and M. Roger. Pathcrawler: automatic generation of path tests by combining static and dynamic analysis. In *EDCC*, 2005.