MAESTRO: Automated Test Generation Framework for High Test Coverage and Reduced Human Effort in Automotive Industry

Yunho Kim[a], Dongju Lee[b], Junki Baek[b], Moonzoo Kim[a,*]

[a]*KAIST, 291 Daehak-ro, Yuseong-gu, Daejeon, South Korea 34141*
[b]*Hyundai Mobis, 17-2 Mabuk-ro, Giheung-gu, Yongin-si, Gyeonggi-do, South Korea 16891*

Abstract

Context: The importance of automotive software has been rapidly increasing because software controls many components of motor vehicles such as smart-key system, tire pressure monitoring system, and advanced driver assistance system. Consequently, the automotive industry spends a large amount of human effort to test automotive software and is interested in automated testing techniques to ensure high-quality automotive software with reduced human effort.
Objective: Applying automated test generation techniques to automotive software is technically challenging because of false alarms caused by imprecise test drivers/stubs and lack of tool supports for symbolic analysis of bit-fields and function pointers in C. To address such challenges, we have developed an automated testing framework MAESTRO.
Method: MAESTRO automatically builds a test driver and stubs for a target task (i.e., a software unit consisting of target functions). Then, it generates test inputs to a target task with the test driver and stubs by applying concolic testing and fuzzing together in an adaptive way. In addition, MAESTRO transforms a target program that uses bit-fields into a semantically equivalent one that does not use bit-fields. Also, MAESTRO supports symbolic function pointers by identifying the candidate functions of a symbolic function pointer through static analysis.
Results: MAESTRO achieved 94.2% branch coverage and 82.3% MC/DC coverage on the four target modules (238 KLOC) developed by Hyundai Mobis. Furthermore, it significantly reduced the cost of coverage testing by reducing the manual effort for coverage testing by 58.8%.
Conclusion: By applying automated testing techniques, MAESTRO can achieve high test coverage for automotive software with significantly reduced manual testing effort.

Keywords: Automated test generation; concolic testing; fuzzing; automotive software; coverage testing

1. Introduction

The automotive industry has developed automotive software to control various components in the motor vehicle, such as the body control module (BCM), smart-key system (SMK), and tire pressure monitoring system (TPMS) [1, 2]. As automotive software becomes larger and more complex with the addition of newly introduced automated features (e.g., automatic parking system (APRK) of advanced driver assistance system (ADAS)) and more sophisticated functionality (e.g., driving mode systems) [3, 4], the cost of testing automotive software is rapidly increasing. Also, it is difficult for human engineers to develop test inputs that can ensure high-quality automotive software within tight software development schedules and budgets. To resolve these problems, the automotive industry is trying to apply automated software testing/verification techniques [5, 6, 7, 8] to achieve high code quality with reduced human effort.

Concolic testing [9] [10] has been applied to automatically generate test inputs for software in various industries. Concolic testing combines dynamic concrete execution and static symbolic execution to explore all possible execution paths of a target program, which can achieve high code coverage. Concolic testing has been applied to various industrial projects (e.g., flash memory device driver [11], mobile phone software [12, 13], and large-scale embedded software [14]) and has effectively improved the quality of industrial software by increasing test coverage and detecting corner-case bugs with modest human effort. Also, fuzzing is starting to show its potential as a general automated test input generation technique, like concolic testing, although it had been originally developed to reveal security vulnerabilities of target systems.

While we were working to apply automated test generation techniques to automotive software developed by Mobis, we observed the following technical challenges that need to be resolved to successfully apply automated test generation techniques:

1. We need to generate test drivers and stubs carefully to achieve high unit test coverage while avoiding gener-

*Corresponding author
Email addresses: yunho.kim03@gmail.com (Yunho Kim), dongju.lee@mobis.co.kr (Dongju Lee), jk.baek@mobis.co.kr (Junki Baek), moonzoo.kim@gmail.com (Moonzoo Kim)

ating test cases corresponding to the executions that are not feasible at the system-level. Otherwise (e.g., generating naive test drivers and stubs that provide unconstrained symbolic inputs to every function in a target program), we will waste human effort to manually filter out infeasible tests that lead to misleading high coverage and false alarms.

2. Current concolic testing tools do not support symbolic bit-fields in C which are frequently used for automotive software.[1] For example, automotive software uses bit-fields in message packets in the controller area network (CAN) bus to save memory and bus bandwidth. However, most concolic testing tools do not support symbolic bit-fields since a bit-field does not have a memory address (Section 3.4) and most programs running on PCs rarely use bit-fields.

3. Although automotive software uses function pointers to simplify code to dynamically select a function to execute, current automatic test generation techniques and tools do not support symbolic setting for function pointers due to the limitation of SMT (Satisfiability Modulo Theories) solvers of concolic testing and input mutation technique of fuzzing.

To address the above challenges, we have developed an automated testing framework MAESTRO (Mobis Automated tESTing fRamewOrk). MAESTRO automatically generates the test driver, stubs, and test inputs for a target unit using concolic testing and fuzzing. MAESTRO achieved 94.2% branch coverage and 82.3% MC/DC coverage of the modules of the ADAS (advanced driver assistance system) and IBU (integrated body unit) software (Section 5.1). Also, MAESTRO reduced the manual testing effort by 58.8% for coverage testing of the target modules of ADAS and IBU (Section 5.3). [2]

The main contributions of this paper are as follows:

1. We have developed MAESTRO which automatically generates the test driver, stubs, and test inputs achieving high coverage for automotive software. MAESTRO applies concolic testing and fuzzing together in an adaptive way (Section 3.5.4) to achieve high coverage.

2. We have identified the technical challenges in applications of automated test input generation to automotive software and describe how MAESTRO resolves them (i.e., *task*-oriented driver/stub generation (Section 3.3.1), symbolic bit-field support (Section 3.4), and symbolic setting for function pointers (Section 3.3.4)). Thus, this paper can support field engineers in the automotive industry to adopt automated test generation with less trial-and-error.

3. To the authors' best knowledge, this is the first industrial study that concretely demonstrates reduced human effort (i.e., human effort reduced by 58.8%) by applying concolic testing and fuzzing together in the automotive industry (Section 5.3). Thus, this study can promote the adoption of concolic testing and fuzzing in the automotive industry.

4. This paper shares lessons learned and valuable information for both field engineers in the automotive industry and researchers who develop automated testing techniques (Section 6). For example, we have found that the generation of precise test drivers and stubs is important to increase test coverage (Section 6.3) and concolic testing and fuzzing have different characteristics to achieve test coverage (Section 6.4).

This journal article is an extended version of our prior automated testing framework MAIST [18] as follows:

1. We have extended MAIST [18] to MAESTRO to achieve higher test coverage as follows:

   (a) MAESTRO uses a hybrid technique of concolic testing and fuzzing as test input generators (Section 3.5.3 and Section 3.5.4). The experiment results show that the hybrid approach achieved higher branch and MC/DC coverage than concolic testing or fuzzing alone (Section 5.7). Also, we have discussed the different characteristics of concolic testing and fuzzing (Section 6.4).

   (b) MAESTRO extended MAIST by generating symbolic stubs that provide more realistic contexts to a target code unit (Section 3.3.3). The experiment results show that MAESTRO's new symbolic stub increases branch and MC/DC coverage (Section 5.8).

2. We have targeted a new module (advanced driver assistance system's automatic parking (APRK)) as well as the ones in the prior work [18]. APRK is a crucial component for safety and it is highly complex, handling multiple sensors and actuators (Section 2.1 and Section 2.2).

3. With the new adaptive hybrid test input generator and the precise symbolic stub generation, MAESTRO achieved 94.2% branch coverage and 82.3% MC/DC coverage for the four target modules. Compared to MAIST, MAESTRO improved 4.1% branch and 5.8% MC/DC coverage.

4. We have added new sections, Section 3.4.1 to Section 3.4.3, to describe MAESTRO's bit-field transformation algorithms and related examples.

The rest of the paper is organized as follows. Section 2 explains the target project. Section 3 describes the MAESTRO framework. Section 4 explains how we have applied MAESTRO to the target modules. Section 5 describes the experiment results. Section 6 presents lessons learned from this industrial study. Section 7 discusses related work. Finally, Section 8 concludes this paper with future work.

[1]A bit-field `x:m` is an integer in a `struct` variable which has only `m` bits. For example, `unsigned int x:3` can represent only 0 to 7.

[2]Several newspapers reported these successful results [15, 16, 17] (MAIST is the predecessor of MAESTRO).

2. Target Project: Controller Software for Advanced Driver Assistance System and Integrated Body Unit

2.1. Overview

Advanced Driver Assistance System (ADAS) is a vehicle monitoring and control system that prevents or reduces damage of car accidents. ADAS developed by Hyundai Mobis consists of automatic parking system, driver monitoring system which warns a driver in drowsiness, and so on. We target the automatic parking system (APRK) because APRK is one of the most complex features in ADAS and, thus, needs intensive testing. APRK takes information from 12 ultra sonic sensors and one camera, and controls the electric stability control system and motor-driven power steering system to safely park a car.

Integrated body unit (IBU) is the first AUTOSAR-compliant electronic control unit (ECU) developed by Mobis. IBU consists of body control module (BCM), smart key system (SMK), and tire pressure monitoring system (TPMS). Mobis has developed more than 10 different versions of IBUs targeting various motor vehicle models. We chose the IBU software as our target project because IBU is essential in driving motor vehicles safely. For example, the automotive safety integrity level (ASIL) of the handle controller in SMK is D. Mobis spends a considerable amount of test engineer resource to test the IBU software.

2.2. Target Project Statistics

Table 1 shows the code statistics of APRK, BCM, SMK, and TPMS modules. [3] The target modules consist of total 301 source files and 4,072 functions having 22,513 branches. Maximum and average cyclomatic complexity of the functions are 24 and 5.3, respectively. Each target module consists of *tasks* which are mostly minimal independent units. A task t consists of

1. an entry function t_e which is defined as a non-static function in a target source file s, and
2. the callee functions that are directly or transitively invoked by the entry function t_e and defined in the same source file s.

The target modules have a total of 909 tasks each of which has a non-static function as an entry function of a task. Each source file contains 3.0 tasks (=909/301) on average. Each task consists of 5.8 functions on average.

2.3. Challenges for Manual Testing

Manual derivation of test inputs for the target projects has the following obstacles:

```
01:int rpm_FL,rpm_FR,rpm_RL,rpm_RR;
02:int angle_FL,angle_FR;
03:int press_FL,press_FR,press_RL,press_RR;
04:int mode,dir,speed;
05:void drive_mode_check(){
06:  ...
07:  if (mode==DRIVE && speed>0 &&
08:  ((dir==LEFT && angle_FL<0 && angle_FR<0 &&
09:  rpm_FL<=rpm_FR && rpm_RL<=rpm_RR) ||
10:  (dir==RIGHT && angle_FL>0 && angle_FR>0 &&
11:  rpm_FL>=rpm_FR && rpm_RL>=rpm_RR) ||
12:  (dir==STRAIGHT && angle_FL==0 && angle_FR==0 &&
13:  rpm_FL==rpm_FR && rpm_RL==rpm_RR)) &&
14:  (L_PRESSURE<press_FL && press_FL<H_PRESSURE) &&
15:  (L_PRESSURE<press_FR && press_FR<H_PRESSURE) &&
16:  (L_PRESSURE<press_RL && press_RL<H_PRESSURE) &&
17:  (L_PRESSURE<press_RR && press_RR<H_PRESSURE))
18:  {...}}
```

Figure 1: A code example of BCM that reads a large number of input variables and evaluates a complex branch condition

1. *A large number of input variables*: Most functions of the target projects take a large number of inputs through parameters and global variables because the target projects check a various status of a motor vehicle such as speed, wheel angles, tire pressure, etc.
2. *Complex branch conditions*: The branch conditions of the target projects are complex in terms of the number of the logical operators (e.g., `&&`, `||`) used in a branch condition. This is because the target projects check complex conditions on complicated status data obtained from various components of a motor vehicle.

Figure 1 is a code example of BCM that reads a large number of input variables and evaluates a complex branch condition. The function `drive_mode_check` (Lines 5–18) reads a total of 13 input variables (Lines 1–4) to evaluate the branch condition which has 24 logical operators (Lines 7–17).

Also, we compare the number of the input variables and the complexity of the branch conditions of the target projects with the five most popular open-source C projects in OpenHub [19]: Apache, MySQL, Subversion, PHP and Bash. [4] Each function of the target projects has 44.1% larger number of input variables (i.e., 9.8) than the five open-source programs (i.e., 6.8) on average. Similarly, the target projects' branch conditions have 2.1 times more number (i.e., 2.7) of `&&` and `||` than the open-source programs (i.e., 1.3).

2.4. Challenges for Concolic Testing and Fuzzing for the Target Projects

Achieving high coverage of the target modules is still challenging for concolic testing and fuzzing for the following reasons:

[3] We have accessed the target modules in the Mobis development server via approved VPN connection.

[4] We exclude Linux Kernel because our Clang-based analysis tool fails to analyze GCC-specific code of Linux Kernel.

Table 1: The code statistics of the target modules

Module	#files	#functions			LoC	#branches	Cyclomatic comp.		
		non-static	static	total			min	max	avg
APRK	47	142	451	593	34292	4655	1	18	4.9
BCM	27	145	511	656	52690	2873	1	15	4.3
SMK	198	554	1967	2521	134877	13768	1	24	5.8
TPMS	29	68	234	302	15951	1217	1	12	4.1
Total	301	909	3163	4072	237810	22513	1	24	5.3

```
01:#define M_MAX 10
02:int model[M_MAX];
03:void f(int x){
04:  ...
05:  g(x%M_MAX);}
06:#define TM9 9
07:static void g(int idx){
08:  ...
09:  for (int i=0;i<idx;i++){
10:    // FALSE ALARM
11:    if (model[i]==TM9){ ... }
12:  ...}}
13:void driver_g(){
14:  int param1;
15:  SYM_int(param1);
16:  g(param1);}
```

Figure 2: An example of false alarms raised by a naive test driver

2.4.1. Infeasible Unit Test Executions Generated

Concolic testing and fuzzing may generate infeasible unit test executions (i.e., tests that are not feasible at system-level) which can report misleading coverage results and waste human engineers' effort in filtering out false alarms (e.g., crashes caused by infeasible test executions). This is because a test driver or stubs may *violate* the assumptions/preconditions on the inputs of the target unit which are always satisfied in the entire system. This problem of infeasible unit test executions has been discussed in several papers [20, 21, 22].

Figure 2 is a code example that shows that a naive function-oriented test driver raises a false alarm. Suppose that only `f` invokes `g`. `f` calls `g` with an argument `(x % M_MAX)` which is always less than 10 (i.e., Line 11 is always safe since `model` has 10 elements (Lines 1–2)). However, a naive test driver `driver_g` (Lines 13–16) directly calls `g` with a symbolic argument value and raises access out-of-bound alarms (i.e., false alarms) at Line 11 because `idx` can be larger than 10.

2.4.2. No Support for Symbolic Bit-fields

The target modules use bit-fields to save memory space and the controller area network (CAN) bus bandwidth. The existing concolic testing tools do not support symbolic bit-field [5] and may not achieve high coverage of the target projects because they cannot guide symbolic executions to cover branches whose conditions depend on bit-fields.

```
01: struct ST{
02:   int b0:1;
03:   ...
04:   int b7:1;};
05: union U{
06:   struct ST s;
07:   char c;};
08: void f(char c){
09:   union U u;
10:   u.c=c;
11:   if(u.s.b7==1){ // concretized
12:     /* not covered */...}}
13: void driver_f(){
14:   char param1;
15:   SYM_char(param1);
16:   f(param1);}
```

Figure 3: An example showing that the branch at Line 12 is not covered due to the lack of symbolic bit-field support

Figure 3 shows a code example where concolic testing may not cover the branch at Line 12. This is because the branching condition depends on a bit-field `u.s.b7` (Line 11) which cannot have a symbolic value due to the lack of symbolic bit-field support.

One naive solution can be to transform all bit-fields into integer variables. However, this approach changes the semantics of a target program when an integer overflow occurs to a bit-field or `union` is used with bit-fields. In Figure 3, `union U` has two fields, `struct ST s` and `char c` which are located in the same memory space. Since `u.s` and `u.c` share the same memory space, the assignment of a value to `u.c` (Line 10) also updates all bit-fields `b0`, ..., `b7` in `u.s` at the same time. [6] Suppose that we transform the bit-fields in `struct ST` into integer variables. Then, the assignment of a (symbolic) value to `u.c` (Line 10) does not update `u.s.b7` because `u.s.b7` is not located in the same memory space of `u.c` anymore. Consequently, con-

[5]A concolic testing tool maintains a symbolic memory which maps a *memory address* to a corresponding symbolic variable. A concolic testing tool cannot get a memory address of a bit-field because C does not support to get the address of a bit-field.

[6]The target projects often use this code pattern to update multiple bit-fields at once.

colic testing may fail to reach Line 12 because `u.s.b7` is not symbolic.

2.4.3. No support for symbolic function pointers

The target projects use function pointers to make compact code. Current concolic testing and fuzzing tools, however, do not support symbolic function pointers due to the limitation of SMT solvers of concolic testing and input mutation technique of fuzzing. Thus, they fail to cover branches whose conditions depend on a function pointed by a function pointer p_f (i.e., concolic testing and fuzzing fail to generate various execution scenarios enforced by assigning different functions to p_f).

3. MAESTRO: Mobis Automated Testing Framework

3.1. Overview

Figure 4(a) overviews MAESTRO (Mobis Automated tESTing fRamewOrk), which takes C source code files as inputs. MAESTRO consists of the three components: *test harness generator*, *converter*, and *test input generator*. First MAESTRO harness generator analyzes the input C source files and generates test driver and stub functions for every task in the source files (Section 3.3). MAESTRO converter transforms the C code that uses bit-fields into semantically equivalent one that does not use bit-fields (Section 3.4). Finally, MAESTRO test input generator applies both concolic testing and fuzzing together in an adaptive way (Section 3.5). MAESTRO uses CROWN (Concolic testing for Real-wOrld softWare aNalysis) [23] and AFL (American Fuzzy Lop) [24] as concolic testing and fuzzing engines to generate test inputs, respectively.

3.2. MAESTRO Implementation

The development team of MAESTRO consists of two Mobis engineers (one senior and one junior engineer) and three researchers of KAIST. The team spent five months to implement MAESTRO. MAESTRO test harness generator is implemented in 4,100 lines of code (LoC) in C++ using Clang/LLVM 4.0 [25]. MAESTRO converter is implemented in 1,100 LoC in OCaml using CIL 1.7.3 [26]. We chose CIL for MAESTRO bit-field transformer because the canonical C code generated by CIL makes the algorithm and implementation of the bit-field transformer easy. MAESTRO test input generator is implemented in 350 LoC in Bash shell script. MAESTRO test input generator uses CROWN [23] and AFL [24] as concolic input generator and fuzzing input generator, respectively. We use CROWN as the concolic input generator because CROWN (and its predecessor CREST) has been successfully applied to various industrial projects (Section 7.3) and two of the authors are involved in developing CROWN. We use AFL as the fuzzing input generator because AFL is widely known to be effective for detecting crash bugs.

3.3. MAESTRO Test Harness Generator

3.3.1. Task-Oriented Driver and Stub

Figure 4(b) shows an example of generating test drivers and stubs for two tasks t_1 and t_2. Suppose that a target program p consists of `file1.c` and `file2.c`. `file1.c` has two non-static functions `f` and `g` and four static functions `s1` to `s4`. t_1 consists of the entry function `f` and its callee functions `s1`, `s2` and `s3` defined in the same file (i.e., `file1.c`). The function `h` invoked by `s3` is not included in t_1 because `h` is defined in another source file (i.e., `file2.c`). Similarly, t_2 consists of the entry function `g` and its callee function `s4`. MAESTRO targets a task t as a testing target unit and generates a test driver (Section 3.3.2) to invoke a task entry function t_e and symbolic stubs (Section 3.3.3) to replace the callee functions of the task t located in other source files.

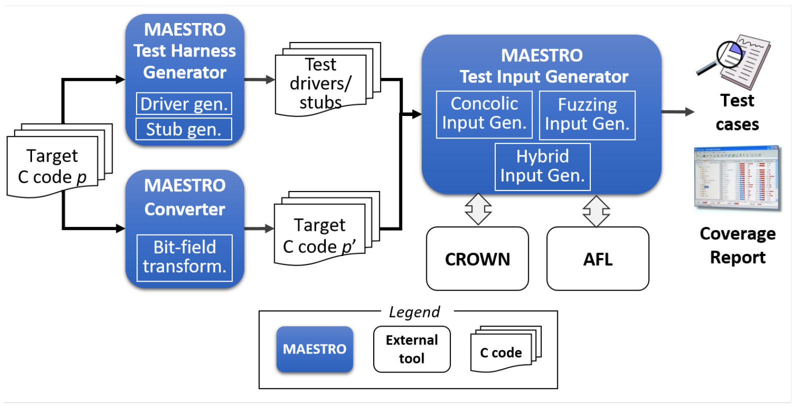
3.3.2. Automated Generation of Test Drivers

MAESTRO automatically generates a test driver that invokes the entry function t_e of each task. The test driver assigns symbolic values to the arguments of t_e and to the global variables used in the task t for concolic testing and fuzzing, and invokes t_e.

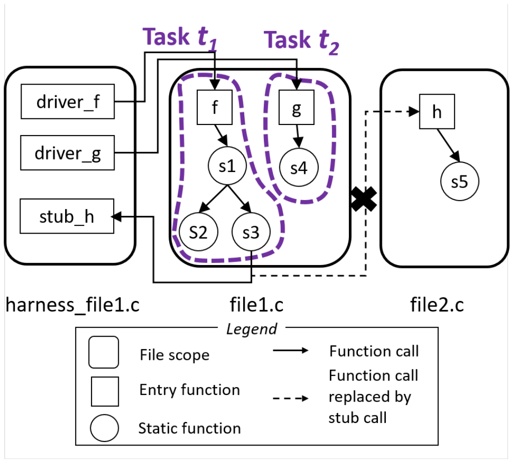
In Figure 4(b), MAESTRO generates `harness_file1.c` that contains test drivers and a stub function for t_1 and t_2 in `file1.c`. `driver_f` and `driver_g` are test driver functions that call the entry functions `f` and `g` of t_1 and t_2, respectively. `s3` in t_1 invokes `h` which is not defined in `file1.c`. Thus, MAESTRO generates a stub function `stub_h` in `harness_file1.c` and modifies `s3` to call `stub_h` instead of `h`.

MAESTRO specifies a variable as a symbolic input according to its type as follows:

- *Primitive types:* MAESTRO specifies a primitive variable `x` as a symbolic input by using `SYM_<T>(x)` where `<T>` is a type of `x`.
- *Array types:* MAESTRO specifies each array element as a symbolic input according to the type of the element.
- *Pointer types:* For a pointer `p` pointing to a memory of a type `T`, MAESTRO allocates memory in `sizeof(T)*n` bytes where `n` is a user-given bound (i.e., MAESTRO considers `p` to point to an array which has `n` elements and whose element type is `T`).
- *Structure types:* For a `struct` variable `s`, MAESTRO specifies each field of `s` as a symbolic input according to the field type recursively. To prevent infinite recursive dereferences (e.g., a linked list forming a cycle), MAESTRO follows a pointer to `s` within a user-given bound k and assigns `NULL` to a pointer that is not reachable within the bound.
- *Bit-field types:* MAESTRO specifies a bit-field `b` as a symbolic input by using `SYM_bitfield(b)`.



(a) The overview of MAESTRO



(b) Test driver/stubs for tasks t_1 and t_2 in `file1.c`

Figure 4: MAESTRO framework

Figure 5 shows an example of unit test driver code generated. [7] `Node` (lines 1-4) represents a node of a linked list which contains a character value and a pointer to its adjacent node. The target unit function is `add_last(v)`(lines 7-10) which takes a character `v` as an input parameter and adds a new node containing `v` to the end of the global linked list whose head is `head` (line 5). `test_add_last()`(lines 12-20) is the test driver code generated for `add_last()`.

The generated driver sets all global variables used by the target unit as symbolic inputs. Since the target unit `add_last()` uses `head`, the driver allocates appropriate memory space to `head` (line 14). Next, the driver declares all fields of the structure pointed by `head` as symbolic variables. Suppose that a user-given bound k for a pointer to a structure variable is 1. The driver sets `head->c` as a symbolic character variable (line 15) and `head->next` as an address of memory obtained by `malloc(_Node)` (Line 16). Then, it sets `head->next->c` as symbolic (Line 17) and `head->next->next` as Null (since $k = 1$) (Line 18).

After the driver finishes setting symbolic global variables, the driver declares function parameters symbolically (line 19). After the driver finishes symbolic input setting, it invokes the target unit function with the symbolic parameters (line 20).

3.3.3. *Symbolic Stub Generation*

A baseline symbolic stub simply returns a symbolic value according to the return type of the symbolic stub. Thus, baseline symbolic stubs do not represent a target task's context accurately because the stub functions do not set global variables nor output parameter variables as symbolic inputs. Thus, such imprecise stubs may prohibit concolic testing from achieving high test coverage [8]

```
01:typedef struct _Node {
02:    char c;
03:    struct _Node *next;
04:} Node;
05:Node *head;
06:// Target unit-under-test
07:void add_last(char v){
08:    // add a new node containing v
09:    // to the end of the linked list
10:   ...}
11:// Test driver for the target unit
12:void test_add_last(){
13:    char v1;
14:    head = malloc(sizeof(Node));
15:    SYM_char(head->c);
16:    head->next = malloc(sizeof(_Node));
17:    SYM_char(head->next->c);
18:    head->next->next=NULL;
19:    SYM_char(v1);
20:    add_last(v1); }
```

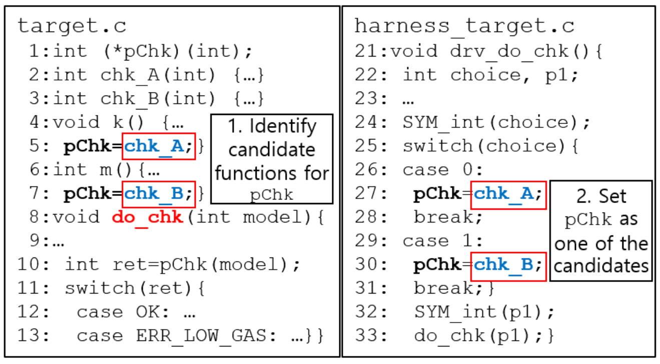
Figure 5: An example of an automatically generated unit test driver

To increase test coverage, we have developed a new symbolic stub generation strategy for MAESTRO. The symbolic stubs generated by MAESTRO set not only the return value of the stub, but also the global variables and output parameters (e.g., pointer parameters that point to the memory locations modified/updated by the function) that can be updated by the function replaced by the symbolic stub as symbolic inputs. Suppose that the function g invoked by the function f in a target task is replaced by a symbolic stub. MAESTRO identifies all global variables and output parameters of g which are modified/updated by g and its callee functions [9]. Then, the symbolic stub

[7]The most part of the example is excerpted from Kim et al. [14].

[8]These baseline symbolic stubs were used by MAIST [18] (the predecessor of MAESTRO).

[9]MAESTRO ignores the variable updates through pointers due to the low precision of the static pointer analysis.



```
target.c
 1:int (*pChk)(int);
 2:int chk_A(int) {…}
 3:int chk_B(int) {…}
 4:void k() {…
 5: pChk=chk_A;}
 6:int m(){…
 7: pChk=chk_B;}
 8:void do_chk(int model){
 9:…
10: int ret=pChk(model);
11: switch(ret){
12:   case OK: …
13:   case ERR_LOW_GAS: …}}
```

```
harness_target.c
21:void drv_do_chk(){
22: int choice, p1;
23: …
24: SYM_int(choice);
25: switch(choice){
26: case 0:
27:   pChk=chk_A;
28:   break;
29: case 1:
30:   pChk=chk_B;
31:   break;}
32:   SYM_int(p1);
33:   do_chk(p1);}
```

1. Identify candidate functions for pChk
2. Set pChk as one of the candidates

Figure 6: Example of symbolic setting for function pointer `pChk`

of g sets the return value, the output parameters, and the global variables updated by g and its callee functions as symbolic variables. Thus, the symbolic stubs generated by MAESTRO represents a target task's context more accurately than the baseline symbolic stubs.

3.3.4. Symbolic Setting for Function Pointers

MAESTRO generates a test driver which assigns various functions to a function pointer p_f used in a target task. First, MAESTRO statically examines all target source files and identifies functions $f_1, ...f_n$ that are assigned to p_f. Then, it adds code c_{p_f} to a test driver such that c_{p_f} assigns each of $f_1, ..., f_n$ to p_f using a symbolic variable *choice* which selects each of $f_1, ..., f_n$ to assign to p_f.

For example, Figure 6 has `do_chk` (Lines 8–13) which has branches whose conditions depend on `ret` (Lines 11–13). `ret` value is determined by the return value of the function pointed to by `pChk` (Line 10). After MAESTRO identifies that `pChk` may point to `chk_A` or `chk_B` at Line 5 or Line 7 respectively, it adds code c_{p_f} to a test driver `drv_do_chk` which assigns `chk_A` and `chk_B` to `pChk` depending on a symbolic variable `choice` (Lines 25–31), as shown in the right part of Figure 6.

3.3.5. Test Driver Generation for a Task with Internal States

Several functions in the target modules use `static` local variables to keep the execution results of the previous invocation as its internal state. For a task containing such a function, MAESTRO generates a test driver to call a target task multiple times with fresh symbolic inputs.

3.4. MAESTRO Converter

Automotive software uses *bit-fields* to minimize the data size. Unlike other primitive types in C which occupy multiples of 8 bits, the size of a bit-field does not have to be in multiples of 8 bits and can be smaller than 8. Currently, concolic testing tools for C programs such as CREST [27], CUTE [9], KLEE [28], and PathCrawler [29] do not support symbolic declaration of bit-fields and fail to achieve high coverage of automotive software that uses bit-fields (Section 2.4.2).

To solve this problem, MAESTRO transforms a target program p into p' that is semantically equivalent to p but does not use bit-fields. In other words, MAESTRO replaces bit-fields with a data array of a byte type and also replaces all arithmetic expressions on the bit-fields with semantically equivalent ones without the bit-fields using the data array with bit-wise operators.

Bit-field conversion algorithm consists of the three sub-algorithms: `struct` transformation (Algorithm 1), bit-field read expression transformation (Algorithm 2), and bit-field write statement transformation (Algorithm 3).

For example, the bit-field conversion algorithm transforms the following original program P with `struct S` (in Figure 7) into P' as shown in Figure 8:

- `struct S` (Lines 1–4) in P is converted to `struct S_nb` (Lines 21–24) in P' that is equivalent to `struct S` but does not have a bit-field (see Section 3.4.1).
- A statement reading a bit-field `s.z` (Lines 11) in P is converted to Lines 31–33 in P' (see Section 3.4.2).
- A statement writing a bit-field `s.z` (Lines 13) in P is converted to Lines 35–42 iin P' (see Section 3.4.3).

3.4.1. `struct` *Transformation*

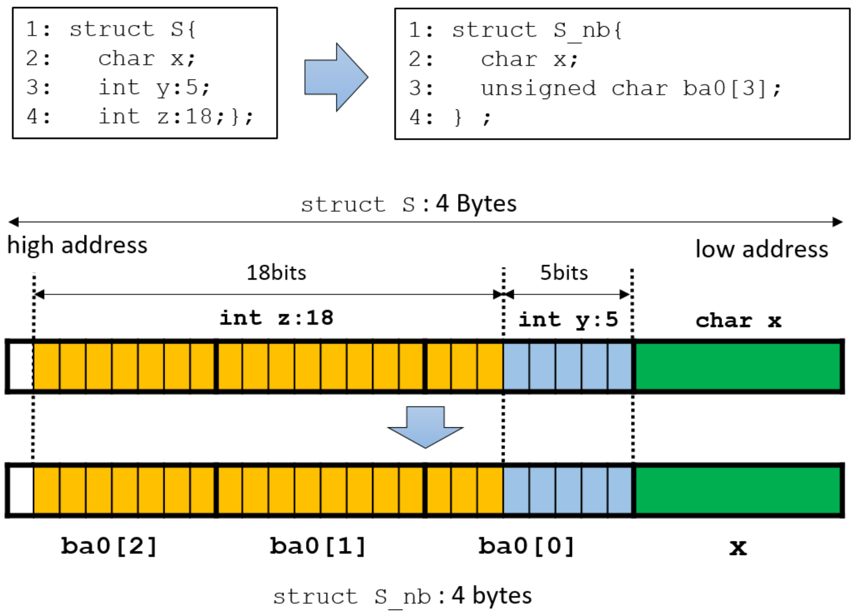
Algorithm 1 takes a `struct` definition S_b that contains bit-fields and generates

- a transformed `struct` definition S_{nb} that does not contain bit-fields, and
- *mapBFtoBA* which maps a bit-field of S_b to its bit location in the corresponding bit-array in S_{nb} which contains transformed bit-field data

S_{nb} may have multiple bit-arrays (e.g., `ba0`, `ba1`, ...). Note that each bit-array in S_{nb} has *bit-wisely identical* contents to the bit-fields in S_b. This bit-wisely identical transformation is important since, without the bit-wisely identical transformation, the transformed program P' may behave differently from the original program P (due to the complex low-level padding and align rules of C, P' can behave differently from P that uses `union` and/or `sizeof`).

We explain Algorithm 1 with an example of `struct S` in Figure 7 as follows. `struct S` has one `char` field (`char x`), and two bit-fields (`int y:5` and `int z:18`). $transformStruct(S_b)$ in Algorithm 1 takes `struct S` as an input and returns the transformed `struct S_nb` and a map from a bit-field (i.e., `int y:5` and `int z:18`) in `struct S` to its bit location in the bit-array `ba0` in `struct S_nb`.

1. The algorithm first initializes the following variables as zero and empty set at Lines 2–5:.
   - *curByteSize*:a current size of S_{nb} in byte.
   - $curBitSize_{ba}$:a current bit-size of a bit array `ba`$<i_{ba}>$ field in S_{nb}.



```
1: struct S{
2:   char x;
3:   int y:5;
4:   int z:18;};
```

```
1: struct S_nb{
2:   char x;
3:   unsigned char ba0[3];
4: } ;
```

Figure 7: An example `struct` for the bit-field transformation algorithm

```
// Original program P
01:struct S{
02:  char x;
03:  int y:5;
04:  int z:18;};
05:
06:S s;
07:
08:int main() {
09:  int n, e=10;
10:
11:  n = s.z + 1;
12:
13:  s.z = e;
14:}
```

```
// Tranformed program P'
21:struct S_nb{
22:  char x;
23:  unsinged char ba0[3];
24:};
25:
26:S_sb s;
27:
28:int main() {
29:  int n, e=10;
30:
31:  n=((((s.ba0[2]&127)<<16)
32:       |(s.ba0[1]<<8)
33:       |s.ba0[0]) >> 5) + 1;
34:
35:  s.ba0[2]=
36:     ((((e)<<5)>>(2*8))&127)
37:     |(s.ba0[2]&128);
38:  s.ba0[1]=
39:     (((e)<<5)>>(1*8))&255;
40:  s.ba0[0]=
41:     (((e)<<5)&224)
42:     |(s.ba0[0]&31); }
```

Figure 8: An example to show bit-field transformation

- $mapBFtoBA$:a map from a bit-field to the bit-location in the corresponding bit-array `ba`$<i_{ba}>$.
- i_{ba}:a number of the generated bit-arrays so far.

2. The loop at Lines 6–20 transforms field definitions in S_b into those in S_{nb} one by one.

   (a) The first iteration:$field_0$ (i.e., the first field of S_b) is `x` which is not a bit-field. If the field is not a bit-field, the algorithm simply appends the field to S_{nb} at Line 17 and increases the current byte size of S_{nb} by the added field's byte size (and padding size if necessary) at Line 18. In this example, $curByteSize$ is increased by one (=the size of `x`).

   (b) The second iteration:$field_1$ (i.e., the second field of S_b) is `y:5` and the algorithm takes `then` branch at Lines 8–15.

      i. The algorithm computes $paddingBitSize$ to align a bit-field in a bit-array (i.e., `ba0`) that will contain the bit-field (i.e., `y:5`) at Line 8. $paddingBitSize$ is zero because `y` does not break the align of its type (i.e., `int`).

      ii. Then, the mapping information from $field_1$ to its corresponding bit location in `ba`$<i_{ba}>$ is added to $mapBFtoBA$ at Line 9. $field_1$ (i.e., `y:5`) is mapped to a tuple $<i_{ba} = 0, pos_l = 0, bitSize = 5>$ (i.e., `y:5` ranges from the first (i.e., pos_l) bit to the fifth bit (i.e., $pos_l + bitSize$) of `ba0`).

      iii. $bitSize_{ba}$ is increased by the field's bit size (and the bit size of the padding if necessary) at Line 10. $bitSize_{ba}$ is increased by five because $field_1$ is 5-bit-long and the padding size is zero.

      iv. If the current field $field_i$ is the last field of S_b or $field_{i+1}$ is not a bit-field, the algorithm completes generating a corresponding bit-array and adds it to S_{nb} (the `then` branch at Lines 12–15). In this example, since $field_1$ is not the last field and $field_2$ is also a bit-field, the algorithm does not take the `then` branch.

   (c) The third iteration:$field_3$ is `z:18` and the algorithm takes `then` branch at Lines 8–15.

      i. The algorithm computes $paddingBitSize$ to compute how many bits are necessary to add as padding

Algorithm 1: `struct` transformation algorithm

Input: S_b:A `struct` that contains bit-fields.

Output: S_{nb}:A transformed `struct` definition that does not contain bit-fields. $mapBFtoBA$:A map from a bit-field of S_b to its bit location in the corresponding 'bit-array'.

```
1  $transformStruct(S_b)$ {
2  $curByteSize$=0;//a current size of $S_{nb}$ in byte
3  $curBitSize_{ba}$=0;//a current bit-size of a bit array ba<$i_{ba}$> field in $S_{nb}$.
4  $mapBFtoBA = \emptyset$;//a map from a bit-field to the bit-location in the corresponding bit-array
5  $i_{ba} = 0$;//the number of generated 'bit-array' so far.
6  for $i = 0; i <$ the number of fields in $S_b; i++$ do
7      if $field_i$ is a bit-field then
8          $paddingBitSize$=getPaddingBitSize($field_i$, $curBitSize_{ba}$, $curByteSize$);
9          $mapBFtoBA$.add($field_i$, <$i_{ba}$, $curBitSize_{ba}$+$paddingBitSize$, $field_i.bitSize$>);
10         $bitSize_{ba}$ += $field_i.bitSize$ + $paddingBitSize$;
11         if $field_i$ is the last field of $S_b$ or $field_{i+1}$ is not a bit-field then
12             appendBitArray($S_{nb}$, $i_{ba}$, $bitSize_{ba}$);//Add ba<$i_{ba}$> field (e.g., ba0, ba1, ... ) to $S_{nb}$
13             $curBitSize_{ba} = 0$;
14             $i_{ba}$ ++;
15         end
16     else
17         appendField($S_{nb}$, $field_i.name$ , $field_i.type$);
18         $curByteSize$+=$field_i.byteSize$+getPaddingByteSize($field_i$, $S_{nb}.byteSize$);
19     end
20 end
21 if $curByteSize < byteSize(S_b)$ then
22     // Add a new padding variable
23     appendPaddingVar($S_{nb}$, $byteSize(S_b)$ - $curByteSize$);
24 end
25 return $S_{nb}$ and $mapBFtoBA$;
26 }
27 $getPaddingBitSize(bitField, curBitSize_{ba}, curByteSize)$ {
28 //Compute the padding size to align $bitField$ variable according to its type
29 $paddingBitSize = bitField.typeSize * 8 - ((curBitSize_{ba} + curByteSize * 8) \% (bitField.typeSize * 8))$;
30 if $bitField.bitSize > paddingBitSize$ then
31     return $paddingBitSize$;
32 else
33     return 0;
34 end
35 }
36 $getPaddingByteSize(field, curByteSize)$ {
37 $paddingByteSize = field.typeSize$ - ($curByteSize$ % $field.typeSize$);
38 if $field.typeSize > paddingByteSize$ then
39     return $paddingByteSize$;
40 else
41     return 0;
42 end
43 }
```

in bit-array at Line 8. $paddingBitSize$ is zero because `z` does not break the align of its type (`int`) (i.e., the three fields `x`, `y` , and `z` can be stored within the four bytes (i.e., the size of `int` type)).

ii. Then, the mapping information from $field_2$ to its corresponding bit location in `ba`<i_{ba}> is added to $mapBFtoBA$ at Line 9. $field_2$ is mapped to a tuple <0,5,18>. This means that $field_2$ is located in a bit-array `ba0`, the starting bit location of $field_1$ in `ba0` is 5, and the size of $field_2$ is 18 bits.

iii. $bitSize_{ba}$ is increased by the sum of field's bit size and padding's bit size at Line 10. $bitSize_{ba}$ is increased by 18 because $field_2$ (`z`) is 18-bit-long and padding size is zero.

iv. Since $field_2$ is the last field in S_b, the algorithm takes `then` branch at Lines 12–15.

  A. The algorithm appends the bit-array `ba0` that contains bit-fields `y:5` and `z:18` into S_{nb} at Line 12.

  B. It initializes $curBitSize_{ba}$ to zero to prepare for the next bit-array at Line 13.

  C. It increases i_{ba} (i.e., the number of generated bit-array so far) by one.

(d) The loop terminates because all three fields in S_b are transformed.

3. After the loop at Lines 6–20 terminates, if the current size of S_{nb} (i.e., *curByteSize*) is less than the size of S_b (i.e., *byteSize*(S_b)) (Line 21), the algorithm adds padding to S_{nb} (taking the `then` branch at Lines 22–24). Since the size of `struct S_nb` is equal to that of `struct S`, the algorithm does not take the `then` branch at Lines 22-24.

4. Finally, the algorithm returns the transformed `struct` S_{nb} and *mapBFtoBA* ({`y`→<0,0,5>,`z`→<0,5,18>}) at Line 25.

3.4.2. Bit-field Read Expression Transformation

Algorithm 2 takes the bit-field v_{bf} (e.g., `S.z`) to read and a map *mapBFtoBA* (e.g., {`y`→<0,0,5>,`z`→<0,5,18>}) as inputs and returns the transformed expression on v_{bf} whose value is the same as v_{bf}. Suppose that a target program has `n = S.z + 1` (where `S.z` is 18 bits long bit-field in Figure 7). The algorithm transforms `S.z` of `n = S.z + 1` into `(((S_nb.ba0[2]&127)<<16)|(S_nb.ba0[1]<<8)|S_nb.ba0[0])>>5` as follows.

1. The algorithm initializes the variables at Lines 3–10.

   - i_{ba}, pos_l and *bitSize* are set using the input *mapBFtoBA*.
     - i_{ba}:an index variable to the bit-array in which v_{bf} is included (i.e., `ba`<i_{ba}>). i_{ba} is obtained from *mapBFtoBA*.get(v_{bf}). For v_{bf}=`S.z`, i_{ba} is 0.
     - pos_l:the starting bit position of v_{bf} in `ba`<i_{ba}>. pos_l is obtained from *mapBFtoBA*.get(v_{bf}). For v_{bf}=`S.z`, pos_l is 5.
     - *bitSize*:the size of v_{bf} in bit. *bitSize* is obtained from *mapBFtoBA*.get(v_{bf}). For v_{bf}=`S.z`, *bitSize* is 18.
   - pos_h:the ending bit position of v_{bf} in `ba`<i_{ba}>. pos_h is set as pos_l + *bitSize* - 1. For v_{bf}=`S.z`, pos_h is 22 (5+18-1).
   - *ba*:the variable name (i.e., string) of i_{ba}th bit array `ba`<i_{ba}> field in a string type (i.e., `S_nb.ba0`, `S_nb.ba1`, ...). For v_{bf}=`S.z`, *ba* is `S_nb.ba0`.
   - $locByte_l$:the lowest byte address of v_{bf} in `ba`<i_{ba}>. $locByte_l$ is set as $\lfloor pos_l/8 \rfloor$. For v_{bf}=`S.z`, $locByte_l$ is 0.
   - $locByte_h$:the highest byte address of v_{bf} in `ba`<i_{ba}>. $locByte_h$ is set as $\lfloor pos_h/8 \rfloor$. For v_{bf}=`S.z`, $locByte_h$ is 2.
   - $locBit_l$:the beginning bit offset of v_{bf} in `ba`<i_{ba}>`[`$locByte_l$`]`. $locBit_l$ is set as pos_l % 8. For v_{bf}=`S.z`, $locBit_l$ is 5.
   - $locBit_h$:the ending bit offset of v_{bf} in `ba`<i_{ba}>`[`$locByte_h$`]`. $locBit_h$ is set as pos_h % 8. For v_{bf}=`S.z`, $locBit_h$ is 6.
   - *bitMask*:a bit mask used to extract specific bits from data. *bitMask* is initialized as zero.

2. If $locByte_l$ is the same as $locByte_h$, the algorithm takes the `then` branch at Lines 12–14. Otherwise, it takes the `else` branch at Lines 15–21. For `S.z`, it takes the `else` branch because $locByte_l$ and $locByte_h$ are 0 and 2, respectively.

   (a) The algorithm sets *exp* to represent data in the lowest byte of v_{bf} at Line 15. For v_{bf}=`S.z`, *exp* becomes `S_nb.ba0[0]`

   (b) The loop at Lines 16–18 generates the C expression that represents data between the lowest byte and the highest byte of v_{bf} in `ba`<i_{ba}>.

      i. At the first iteration of the loop (i.e., i is 1), the algorithm adds the C expressions to *exp* to represent the data of ith byte of v_{bf} in `ba`<i_{ba}> at Line 17. For v_{bf}=`S.z`, *exp* becomes `(S_nb.ba0[1]<<8)|S_nb.ba0[0]`.

      ii. After the first iteration, the loop terminates.

   (c) The algorithm sets *bitMask* at Line 19 to extract 0th to $locBit_h$ bits of the highest byte of v_{bf}. For v_{bf}=`S.z`, *bitMask* is set as 127 (`0b01111111` in a binary representation) to extract the lower seven bits of the highest byte (i.e., `S_nb.ba0[2]`).

   (d) It adds the C expression that represents data in the highest byte of v_{bf} at Line 20. After Line 20, *exp* becomes `((S_nb.ba0[2] & 127)<<16) | (S_nb.ba0[1]<<8)|S_nb.ba0[0]`

3. It adds the C expression that performs right-shift $locBit_l$ times to *exp* at Line 22. For v_{bf}=`S.z`, *exp* becomes `(((S_nb.ba0[2] & 127)<<16)|(S_nb.ba0[1]<<8)|S_nb.ba0[0])>>5`.

4. It returns *exp*, which is the transformed expression whose value is equal to v_{bf} (i.e., `S.z`) at Line 22.

3.4.3. Bit-field Write Statement Transformation

Algorithm 3 takes the assignment statement $stmt_{assign}$ (e.g., `S.z=`e) in which a bit-field is on the left-hand side (LHS) and a map *mapBFtoBA* (e.g., {`y` → <0,0,5>,`z`

Algorithm 2: Bit-field read expression transformation algorithm

Input: v_{bf}: a bit-field to read (e.g., `S.f1`), *mapBFtoBA*: a map from a bit-field to its bit location in the corresponding bit-array.

Output: *exp*: a transformed expression of the bit-field which is evaluated to the same value of v_{bf}.

```
1  $transformReadExp(v_{bf}, mapBFtoBA)$ {
2  // $i_{ba}$:$v_{bf}$ is included in ba<$i_{ba}$>, $pos_l$:the starting bit position of $v_{bf}$ in ba<$i_{ba}$>, $bitSize$:the size of $v_{bf}$ in bit
3  <$i_{ba}$, $pos_l$, $bitSize$> = $mapBFtoBA$.get($v_{bf}$);
4  $pos_h = pos_l + bitSize$ - 1;//the ending bit position of $v_{bf}$ in ba<$i_{ba}$>
5  $ba$ = the variable name (i.e., string) of $i_{ba}$th bit array ba<$i_{ba}$> field in a string type; // e.g., "ba0", "ba1", ...
6  $locByte_l = \lfloor pos_l/8 \rfloor$;//the lowest byte address of $v_{bf}$ in ba<$i_{ba}$>
7  $locByte_h = \lfloor pos_h/8 \rfloor$;//the highest byte address of $v_{bf}$ in ba<$i_{ba}$>
8  $locBit_l = pos_l$ % 8;//the beginning bit offset of $v_{bf}$ in the lowest byte of ba<$i_{ba}$>
9  $locBit_h = pos_h$ % 8;//the ending bit offset of $v_{bf}$ in the highest byte of ba<$i_{ba}$>
10 $bitMask = 0$ ;
11 if $locByte_l == locByte_h$ then
12     $setBits(bitMask, locBit_l, locBit_h, 1)$;
13     $exp = ba[locByte_l]$ & $bitMask$;
14 else
15     $exp = ba[locByte_l]$ ;
16     for $i = 1; i < locByte_h - locByte_l; i++$ do
17         $exp = (ba[i + locByte_l] << 8 * i)|exp$ ;
18     end
19     $setBits(bitMask, 0, locBit_h, 1)$;
20     $exp = ((ba[locByte_h]$ & $bitMask) << 8 * (locByte_h - locByte_l))|exp$ ;
21 end
22 $exp = (exp) >> locBit_l$;
23 return $exp$;
24 }
25 //Set $var$'s $from$th to $to$th bits to $bitVal$. The bit index starts from 0.
26 $setBits(var, from, to, bitVal)$ {...}
```

$\rightarrow$ <0,5,18>}) as inputs. Then, it returns the compound statement consisting of the transformed assignment statements that do not use a bit-field, which is equivalent to $stmt_{assign}$.

We assume that all bit-field writes are canonicalized to the assignment statements that do not have a side-effect. MAESTRO uses CIL [26] to obtain the canonicalized C code without side-effect. Suppose that a target program has `S.z=`e (where `S.z` is 18 bits long bit-field in Figure 7 and e is a C expression (e.g., `S.x+3`)). The algorithm transforms `S.z=`e into the below code as follows:

```
S_nb.ba0[2]=((((e)<<5)>>(2*8))&127)
              |(S_nb.ba0[2]&128);
S_nb.ba0[1]=(((e)<<5)>>(1*8))&255;
S_nb.ba0[0]=(((e)<<5)&224)|(S_nb.ba0
    [0]&31);
```

1. The algorithm first sets the following variables at Lines 2–12.

   - v_{bf}:the bit-field in LHS of $stmt_{assign}$. For $stmt_{assign}$ (i.e., `S.z=`e`;`), v_{bf} is `S.z`.
   - exp_{RHS} is RHS of $stmt_{assign}$ with parentheses. For $stmt_{assign}$ (i.e., `S.z=`e`;`), exp_{RHS} is (e).
   - The algorithm sets the variables from i_{ba} at Line 5 to *bitMask* at Line 12 in the same way as Algorithm 2 does.

2. If $locByte_l$ is equal to $locByte_h$, the algorithm takes the `then` branch at Lines 14-18. Otherwise, the algorithm takes the `else` branch at Lines 19-32. For $stmt_{assign}$ (i.e., `S.z=`e`;`), the algorithm takes the `else` branch because $locByte_l$ and $locByte_h$ are 0 and 2, respectively.

   (a) The algorithm sets *bitMask* at Line 19 to extract $locBit_l$th to the seventh bits of $exp_{RHS} << locBit_l$. For $stmt_{assign}$ (i.e., `S.z=`e`;`), *bitMask* is set as 224 (`0b11100000` in a binary representation) because $locBit_l$ is 5.

   (b) It sets $exp_{changedBits}$ that represents the updated bits of the lowest byte of ba<i_{ba}> that corresponds to v_{bf}. After Line 20, $exp_{changedBits}$ becomes `((`e`)<<5) & 224`.

   (c) It sets $exp_{unchangedBitgs}$ that represents the remaining unchanged bits of the lowest byte of ba<i_{ba}> that

Algorithm 3: Bit-field write statement transformation algorithm

```
Input: $stmt_{assign}$: A bit-field assignment statement (a bit-field on the left-hand side (LHS) and an expression on
       the right-hand side (RHS) (e.g., S.z=$e$;). $mapBFtoBA$: a map from a bit-field to its bit location in the
       corresponding bit-array.
Output:
$stmt_{trans}$: A compound statement consisting of the transformed assignment statements that do not use a
bit-field, which is equivalent to $stmt_{assign}$.
1  $transformWriteStmt(stmt_{assign}, mapBFtoBA)$ {
2  $v_{bf}$ = the bit-field in LHS of $stmt_{assign}$; // e.g., S.z in S.z=$e$;
3  $exp_{RHS}$ = RHS of $stmt_{assign}$ with parentheses; // e.g., ($e$) in S.z=$e$;
4  // $i_{ba}$: $v_{bf}$ is included in ba<$i_{ba}$>, $pos_l$: the starting bit position of $v_{bf}$ in ba<$i_{ba}$>, $bitSize$: the size of $v_{bf}$ in bit
5  <$i_{ba}$, $pos_l$, $bitSize$> = $mapBFtoBA$.get($v_{bf}$);
6  $pos_h = pos_l + bitSize$ - 1; // the ending bit position of $v_{bf}$ in ba<$i_{ba}$>
7  $ba$ = the variable name (i.e., string) of $i_{ba}$th bit array ba<$i_{ba}$> field in a string type; // e.g., "ba0", "ba1", ...
8  $locByte_l = \lfloor pos_l/8 \rfloor$; // the lowest byte address of $v_{bf}$ in ba<$i_{ba}$>
9  $locByte_h = \lfloor pos_h/8 \rfloor$; // the highest byte address of $v_{bf}$ in ba<$i_{ba}$>
10 $locBit_l = pos_l$ % 8; // the beginning bit offset of $v_{bf}$ in the lowest byte of ba<$i_{ba}$>
11 $locBit_h = pos_h$ % 8; // the ending bit offset of $v_{bf}$ in the highest byte of ba<$i_{ba}$>
12 $bitMask = 0$ ;
13 if $locByte_l == locByte_h$ then
14     $setBits(bitMask, locBit_l, locBit_h, 1)$;
15     $exp_{changedBits} = (exp_{RHS} << locBit_l)$ & $bitMask$ ;
16     $exp_{unchangedBits} = ba[locByte_l]$ & ~$bitMask$;
17     $stmt_{trans} = ba[locByte_l] = (exp_{changedBits})|(exp_{unchangedBits})$ ;
18 else
19     $setBits(bitMask, locBit_l, 7, 1)$;
20     $exp_{changedBits} = (exp_{RHS} << locBit_l)$ & $bitMask$;
21     $exp_{unchangedBits} = ba[locByte_l]$ & ~$bitMask$);
22     $stmt_{trans} = ba[locByte_l] = (exp_{changedBits})|(exp_{unchangedBits})$;
23     for $i = 1; i < locByte_h - locByte_l; i++$ do
24         $exp_{dataByte} = extractBits(exp_{RHS} << locBit_l, i*8, i*8+7)$;
25         $stmt_{trans} = ba[i + locByte_l] = (exp_{dataByte})\,;stmt_{trans}$;
26     end
27     $bitMask = 0$ ;
28     setBits($bitMask, 0, locBit_h, 1$);
29     $exp_{changedBits} = extractBits(exp_{RHS} << locBit_l, (locByte_h - locByte_l)*8, (locByte_h - locByte_l)*8 + locBit_h)$
        & $bitMask$;
30     $exp_{unchangedBits} = ba[locByte_h$ & ~$bitMask$;
31     $stmt_{trans} = ba[locByte_h] = (exp_{changedBits})|(exp_{unchangedBits})\,;stmt_{trans}$;
32 end
33 return $stmt_{trans}$;
34 }
35 // Return a string converted from an integer value $num$
36 $str(num)$ {...}
37 //Set $var$'s $from$th to $to$th bits to $bitVal$. The bit index starts from 0.
38 $setBits(var, from, to, bitVal)$ {...}
39 // Return $var$'s $from$th to $to$th bits. The bit index starts from 0.
40 $extractBits(num, from, to)$ {...}
```

corresponds to v_{bf}. After Line 21, $exp_{unchangedBits}$ becomes `S_nb.ba0[0] & 31` (31 is `0b00011111`).

(d) It sets $stmt_{results}$ to assigns the RHS to the lowest byte of ba<i_{ba}> that corresponds to v_{bf} using $exp_{changedBits}$ and $exp_{unchangedBits}$. After Line 22, $stmt_{trans}$ becomes a string `S_nb.ba0[0] = (((`e`)<<5)&224)|(S_nb.ba0[0]&31);`

(e) The loop at Lines 23–26 generates the C statement that assigns the RHS to the byte locations between the lowest byte and the highest byte of ba<i_{ba}> that corresponds to v_{bf}. For $i = 1, ..., locByte_h - locByte_l - 1$ which is an index to the byte of ba<i_{ba}> to update , the algorithm generates the C statement that assigns ith byte data of $exp_{RHS} << locBit_l$ (i.e., `(((e)<<5)>>(i*8)) & 255` for $stmt_{assign}$ (i.e., `S.z=e;`)) to ba<i_{ba}> $[i + locByte_l]$ (i.e., `S_nb.ba0[i]` for $stmt_{assign}$ (i.e., `S.z=e;`)) at Line 24.

i. At the first iteration of the loop (i.e., i is 1), the algorithm adds the new transformed C statement to $stmt_{trans}$ where the new C statement assigns the first byte data of $exp_{RHS} << locBit_l$ to ba<i_{ba}> $[1 + locByte_l]$.

A. The algorithm extracts the first byte data of $exp_{RHS} << locBit_l$ (i.e., `(((e)<<5)>>(1*8)) & 255` for $stmt_{assign}$ (i.e., `S.z=e;`)) at Line 24. After line 24, $exp_{dataByte}$ becomes `(((e)<<5)>>(1*8)) & 255`.

B. The algorithm adds the new transformed C statement to $stmt_{trans}$ where the new C statement assigns $exp_{dataByte}$ (i.e., `(((e)<<5)>>(1*8))&255` for $stmt_{assign}$ (i.e., `S.z=e;`)) to ba<i_{ba}>$[1 + locByte_l]$ (i.e., `S_nb.ba0[1]` for $stmt_{assign}$ (i.e., `S.z=e;`)) at Line 25. After Line 25, $stmt_{trans}$ becomes the below code.

```
S_nb.ba0[1]=(((e)<<5)>>(1*8)) &
    255;
S_nb.ba0[0]=(((e)<<5) & 224)|
                (S_nb.ba0[0] & 31);
```

ii. After the first iteration, the loop terminates.

(f) The algorithm sets $bitMask$ to extract 0th to $locBit_h$th bits of the highest byte of $exp_{RHS} << locBit_l$ at Lines 27–28. For $stmt_{assign}$ (i.e., `S.z=e;`), $bitMask$ is set as 127 (`0b01111111`) because $locBit_h$ is 6.

(g) It sets $exp_{changedBits}$ that represents the updated bits of the highest byte of ba<i_{ba}> that corresponds to v_{bf} at Line 29. After Line 29 is executed, $exp_{changedBits}$ becomes `(((e)<<5)>>(2*8)) & 127`.

(h) It sets $exp_{unchangedBitgs}$ that represents the remaining unchanged bits of the highest byte of ba<i_{ba}> that corresponds to v_{bf} at Line 30. After Line 30 is executed, $exp_{unchangedBits}$ becomes `S_nb.ba0[2] & 128 (128 is 0b10000000)`.

(i) It sets $stmt_{trans}$ that assigns the RHS to the highest byte of ba<i_{ba}> that corresponds to v_{bf} using $exp_{changedBits}$ and $exp_{unchangedBits}$ at Line 31. After Line 31 is executed, $stmt_{trans}$ becomes the below code

```
S_nb.ba0[2]=((((e)<<5)>>(2*8))&127)
                |(S_nb.ba0[2]&128);
S_nb.ba0[1]=(((e)<<5)>>(1*8))&255;
S_nb.ba0[0]=(((e)<<5)&224)
                |(S_nb.ba0[0]&31);
```

3. The algorithm returns $stmt_{results}$, which is equivalent to $stmt_{assign}$ (i.e., `S.z=e;`) at Line 33.

3.5. MAESTRO Input Generator

MAESTRO input generator utilizes concolic testing (Section 3.5.2) and fuzzing (Section 3.5.3) together in an *adaptive way* (Section 3.5.4) to achieve high test coverage.

3.5.1. Input File Format of MAESTRO

The test input file of MAESTRO consists of the type and value of input variables in a text format. The odd lines and even lines of the test input file represent the type and the value of the input variables, respectively.

3.5.2. Concolic Input Generator

MAESTRO's concolic input generator utilizes various symbolic search strategies to increase test coverage. Although there are dozens of symbolic search strategies [30] to increase coverage within a given time budget, no single strategy outperforms all others because they are heuristics by their nature. MAESTRO's concolic input generator utilizes the four search strategies (i.e., depth-first-search (DFS), reverse-DFS, random negation, and control-flow-graph based search (CFG)) to increase test coverage and reduce execution time. MAESTRO's concolic input generator applies DFS as the first search strategy to explore all possible paths. This is because DFS stops concolic testing when it has explored all possible execution paths. If DFS has explored all possible execution paths, MAESTRO stops concolic testing for the target task. Otherwise, the concolic input generator applies reverse-DFS and random negation. Lastly, MAESTRO's concolic input generator applies CFG to the remaining uncovered branches where CFG tries to guide concolic testing to reach the uncovered branches [27].

3.5.3. Fuzzing Input Generator

MAESTRO has two fuzzing input generators - *baseline fuzzing* and *type-information preserving fuzzing.* The baseline fuzzing takes the input file (Section 3.5.1) as a seed input and mutates the input file without considering the input file format of MAESTRO. Thus, the baseline fuzzing not only mutates the value of the input variables, but also changes the type of input variables. Also, the baseline fuzzing can change the number of input variables by mutating the newline (`\n`) character. As a result, the baseline fuzzing can generate a large number of invalid test inputs each of which has a different number of input values from the ones accepted by the target program.

To reduce the invalid input generation, we have developed the type-information preserving fuzzing. The type-information preserving fuzzing does not change the type-information of the input variables nor the number of the input variables. To do so, type-information preserving fuzzing first separates the type-information from the values of the input variables in the input file. Then, it mutates only the values of the input variables.

3.5.4. Hybrid Input Generator

Simple Hybrid Input Generator. The simple hybrid input generator runs concolic testing for the first half of the execution time (i.e., for 50% of the timeout (Section 4.4)) and then runs the type-information preserving fuzzing (we simply call it 'fuzzing' in this sub-section) for the remaining half of the execution time. For concolic testing, MAESTRO applies the four search strategies (Section 3.5.2) each of which runs one fourth of a given time to concolic testing. After concolic testing phase finishes, MAESTRO runs fuzzing with the test inputs generated by the concolic input generator as seed tests for fuzzing. MAESTRO's simple hybrid input generator runs concolic testing first and then fuzzing because it is difficult for concolic testing to use the large number of test inputs generated by fuzzing as seed inputs within a short time (e.g., 2.5 (=10/4) minutes for each concolic testing strategy in our empirical evaluation setup (Section 4.4)). The limitation of the simple hybrid input generator is that it keeps running concolic testing (or fuzzing) even after branch coverage is saturated and, thus, fails to utilize chance to improve test coverage.

Adaptive Hybrid Input Generator. To overcome the limitation of the simple hybrid input generator, we have developed the adaptive hybrid input generator. It monitors the achieved test coverage of the current running test generation technique for every time slot and changes the test generation technique to run when the achieved test coverage does not satisfy the criterion. The hybrid input generator works as follows:

1. For the first time slot t_1, it runs concolic testing. The duration of each time slot is 1/10 of the timeout for testing a given task.

2. When the ith time slot t_i $(1 \leq i)$ finishes, it compares the increased branch coverage (i.e., the number of newly covered branches over the total number of branches) of t_i and that of t_{i-1}. If the increased branch coverage of t_i is no more than 80% of that of t_{i-1} (i.e., current testing technique reaches a coverage saturation point), it changes an input generation technique for t_{i+1} with the generated test inputs as seed inputs. When fuzzing passes multiple seed inputs to concolic testing, concolic testing runs a search strategy with each seed input for $\frac{\text{the duration of time slot } t_{i+1}}{\text{the number of seed inputs}}$. Otherwise, it runs the same input generation technique for t_{i+1}. We consider the increased branch coverage of t_0 as 0.

3. If the adaptive hybrid input generator reaches the timeout (i.e., 10 time slots), test generation finishes.

The adaptive hybrid input generator uses only CFG strategy for concolic testing. This is because using multiple search strategies with seed inputs within a short time is not effective in increasing test coverage. Suppose that the time slot for concolic testing is two minutes and three seed inputs are passed. If concolic testing runs each of the four strategies, each strategy runs for only 30 seconds (two minutes/4 strategies). Also, concolic testing runs each strategy with one seed input for only 10 seconds (30 seconds/3 seed inputs) which is too short to effectively increase test coverage. We choose CFG strategy because our exploratory study shows that using CFG with seed inputs achieves higher branch coverage than DFS, reverse-DFS, and random negation strategy.

4. Industrial Case Study: Applying MAESTRO to APRK, BCM, SMK, and TPMS

We have developed and applied MAESTRO to APRK (automatic parking system), BCM (body control module), SMK (smart-key system), and TPMS (tire pressure monitoring system) from October 2017 to March 2019 as explained in the following subsections.

4.1. Research Questions

RQ1 to RQ3 evaluate the testing effectiveness of MAESTRO in terms of test coverage and how much manual testing cost MAESTRO reduced.

RQ1. Effectiveness of the automated test generation: How much test coverage does MAESTRO achieve for the target modules in terms of branch and MC/DC coverage?

RQ2. Analysis of the uncovered branches: What are the major reasons why MAESTRO fails to reach uncovered branches?

RQ3. Benefit of MAESTRO over the manual testing: How much human effort does MAESTRO reduce in terms of the test engineer man-month spent for the target modules?

RQ4 to RQ8 evaluate how effectively MAESTRO addresses the technical challenges described in Section 2.4.

RQ4. Effect of the task-oriented automated test generation: Compared to a function-oriented technique, how much test coverage does MAESTRO achieve for the target modules in terms of branch and MC/DC coverage and how many false crash alarms does MAESTRO raise?

RQ5. Effect of the symbolic bit-field support: How much does the symbolic bit-field support of MAESTRO increase the branch and MC/DC coverage?

RQ6. Effect of the symbolic setting for function pointers: How much does the symbolic setting for function pointers provided by MAESTRO increase the branch and MC/DC coverage?

RQ7. Effect of the input generation techniques: Compared to concolic testing and fuzzing, how much does the adaptive hybrid input generation technique of MAESTRO increase the branch and MC/DC coverage?

RQ8. Effect of the symbolic stub generation strategies: Compared to the baseline symbolic stubs, how much does the symbolic stub generation strategy of MAESTRO increase the branch and MC/DC coverage?

4.2. Test Generation Techniques Used

To evaluate the strengths and weaknesses of the test generation ability of MAESTRO that applies the adaptive hybrid input generation (Section 3.5.4), we have compared MAESTRO with the following test generation techniques:

- *MAESTRO using function-oriented concolic unit testing (M^{FO})* for RQ4: This variant of MAESTRO is the same as MAESTRO but generates test drivers and stubs in a function-oriented manner. M^{FO} generates a test driver for each target function and replaces all the functions invoked by the target function with the stub functions.
- *MAESTRO without symbolic bit-field support (M^{-SBF})* for RQ5: It is the same as MAESTRO but does not support symbolic bit-fields (Section 3.4).
- *MAESTRO without symbolic setting for function pointers (M^{-SFP})* for RQ6: It is the same as MAESTRO but the generated test driver by M^{-SFP} does not provide symbolic setting for a function pointer (Section 3.3.4).
- *MAESTRO using the concolic testing technique (M^{CT})* for RQ7: It is the same as MAESTRO but the test generation technique is the concolic testing technique that uses the four search strategies (Section 3.5.2).
- *MAESTRO using the baseline fuzzing technique (M^{BF})* for RQ7: It is the same as MAESTRO but the test generation technique is the baseline fuzzing technique which mutates both of the the type-information and values of the input variables (Section 3.5.3).
- *MAESTRO using the type-information preserving fuzzing technique (M^{TPF})* for RQ7: It is the same as MAESTRO but the test generation technique is the type-information preserving fuzzing technique which preserves type-information and mutates only the values of the input variables (Section 3.5.3).
- *MAESTRO using the simple hybrid of concolic testing and fuzzing technique (M^{SH})* for RQ7: It is the same as MAESTRO but the test generation technique is the simple hybrid of M^{CT} and M^{TPF}, which runs M^{CT} for the first half of the execution time and then run M^{TPF} using the test cases generated by M^{CT} as seed test cases until it reaches timeout (Section 3.5.4).
- *MAESTRO without symbolic setting for output parameters and global variables updated by symbolic stubs (M^{BS})* for RQ8: It is the same as MAESTRO but the generated symbolic stub by M^{BS} does not update parameters and global variables, but provides symbolic setting for only return value.

4.3. Measurement

To show the test effectiveness of MAESTRO, we measure branch coverage and MC/DC coverage by using CTC++ [31]. We measure branch coverage because the manual test generation of the target modules targets 100% of branch coverage in Mobis and we need to compare the manual test generation and MAESTRO for RQ3. Also, we measure MC/DC coverage because MC/DC coverage is required for safety critical components by ISO 26262 safety requirement for automotive systems. Also, to compare the number of false crash alarms generated by MAESTRO and M^{FO}, we measure the number of crash alarms and crash locations by counting the number of test executions that cause a crash (e.g., segmentation fault) and the code lines where the crash occurs, respectively.

4.4. Test Configuration

- *Timeout*: For each target task, we set the timeout as 20 minutes for MAESTRO and its variants except M^{FO}. For M^{FO}, to make the total amounts of the testing time of M^{FO} and MAESTRO same, we set the timeout as 4.8 minutes for each target function (=(27.2 hours (total wall-clock testing time for all target tasks by MAESTRO)×3 machines×4 cores/machine)/4072 functions). Since M^{CT} applies the four search strategies (Section 3.5.2), each search strategy has five minutes as the timeout.
- *The number of repeated execution of a target task*: For the tasks that have a function with `static` local variables, MAESTRO generates a test driver that invokes the target task twice with fresh symbolic inputs. We chose the number of the repeated invocations as two because most tasks use `static` local variables to keep the immediately previous execution results.
- *Size and dereference bounds of a pointer*: We set the user-given size bound n for pointers as 10 and k for `struct` variables as 4 (Section 3.3.2).
- *Seeds for fuzzing*: For M^{BF} and M^{TPF}, we provide 10 randomly generated tests as seed inputs for each tasks.
- *Testbed*: The experiments were performed on three machines, each of which is equipped with Intel Xeon X5670 (6-cores 2.93 GHz) and 8GB RAM, running 64 bit Ubuntu 16.04. We run four test generation instances on each machine (i.e., applying MAESTRO to 12 tasks (=4 instances×3 machines) in parallel).

5. Experiment Results

5.1. RQ1. Effectiveness of the Automated Test Generation

Table 2 shows the number of the generated test inputs, execution time, and branch and MC/DC coverage of APRK, BCM, SMK, and TPMS achieved by MAESTRO.

Table 2: The number of generated tests, execution time, and branch and MC/DC coverage of the target modules achieved by MAESTRO

Targets	#tests	Time (hour)	Branch cov. (%)	#funcs achieving given branch cov. range						MC/DC cov. (%)	#funcs achieving given MC/DC cov. range					
				[0%, 20%)	[20%, 40%)	[40%, 60%)	[60%, 80%)	[80%, 100%)	100%		[0%, 20%)	[20%, 40%)	[40%, 60%)	[60%, 80%)	[80%, 100%)	100%
APRK	119347	3.8	91.2	0	13	24	41	109	406	77.9	0	14	26	123	136	294
BCM	155962	4.5	95.1	0	12	48	60	80	456	81.2	0	35	47	115	122	337
SMK	1515869	16.6	94.5	0	66	94	172	151	2038	83.5	0	74	209	246	291	1701
TPMS	154542	2.3	95.3	0	10	13	13	50	216	82.8	0	13	16	54	77	142
Total	1945720	27.2	94.2	0	101	179	286	390	3116	82.3	0	136	298	538	626	2474

MAESTRO generated 1,945,720 test inputs in 27.2 hours on three machines (i.e., on 12 cores), which achieved 94.2% branch coverage and 82.3% MC/DC coverage on the target modules.

MAESTRO achieved 100% branch and 100% MC/DC coverage of 76.5% (=3116/4072) and 60.8% (=2474/4072) of all functions in the target modules, respectively. Also, MAESTRO achieved more than 80% branch and 80% MC/DC coverage for 86.1% (=(390+3116)/4072) and 76.1% (=(626+2474)/4072) of all functions in the target modules, respectively.

5.2. RQ2. Analysis of the Uncovered Branches

We manually analyzed the uncovered branches of APRK as an example. We analyzed 37 (=0+13+24) functions in APRK whose branch coverage is less than 60%. MAESTRO did not cover the 218 uncovered branches of these 37 functions due to the following five reasons:

1. *Imprecise driver:* 68 branches (=31.2%) are uncovered because test drivers provide only limited symbolic inputs for complex data structure (i.e., a test driver provides symbolic inputs for only variables reachable from a target task within a given pointer link bound (i.e., $k = 4$) (Section 3.3.2)). To cover these branches, MAESTRO has to increase the pointer link bound (e.g., $k > 4$). But, an increased pointer link bound may not increase the coverage in given testing time due to enlarged symbolic space.

2. `static` *local variable:* 66 branches (=30.3%) are uncovered because MAESTRO fails to assign diverse values to `static` local variables through symbolic input variables of a target task (Sect. 3.3.5).

3. *Path explosion:* 36 branches (=16.5%) are uncovered due to the path explosion problem of concolic testing and fuzzing. These branches can be covered if we increase the time limit for test generation per task (i.e., larger than 20 minutes).

4. *Imprecise stub:* 24 branches (=11.0%) are uncovered because the symbolic stubs generated by MAESTRO provide only limited symbolic inputs for complex data structure for return values, global variables, and parameters similarly to the imprecise driver case.

5. *Unreachable branches:* 24 branches (=11.0%) are unreachable because APRK used in this experiment targets a specific motor vehicle model and these uncovered branches are designed to execute only for another motor vehicle model.

5.3. RQ3. Benefit of MAESTRO over the Manual Testing

5.3.1. Cost of the Manual Testing

Previously, 30 test engineers at Mobis had written test inputs for coverage testing of the target modules, respectively. The test engineers have three years of experience in testing and QA on average. A test engineer writes function test inputs targeting 100% branch coverage for 350 LoCs in one business day, on average (i.e., for one month, a test engineer writes unit test inputs for 7 KLoC (=350 LoC× 20 business days) on average). Thus, writing manual test inputs for coverage testing of the target modules (238K LoC) requires 34 man-months (MM) (= $\frac{238KLoC}{7KLoC\ per\ month}$).

5.3.2. Benefit of MAESTRO

MAESTRO reduced 58.8% of the manual testing effort as follows. After applying MAESTRO, the test engineers still have to generate test inputs to cover 5.8% (= 100-94.2) of the the target modules branches that were not covered by MAESTRO. The test engineers spent four MM to cover those branches. Also, 10 MM were spent developing MAESTRO and training the test engineers to use MAESTRO. Thus, 34 MM of the manual testing effort for coverage testing of the target modules is reduced to 14 MM, which is equivalent to reducing 58.8% of the previous manual coverage testing cost of the target modules. Note that MAESTRO will reduce the manual testing effort much further for future application since the cost of 10 MM for the development and training of MAESTRO is one time cost.

5.4. RQ4. Effect of the Task-oriented Automated Test Generation

We compare the branch and MC/DC coverage and the number of the crashes reported by M^{FO} and MAESTRO. Table 3 shows that M^{FO} achieves 3.2% (= (97.2 − 94.2)/94.2) and 8.4% higher branch and MC/DC coverage than MAESTRO, respectively. This is because M^{FO} directly controls the executions of each function f by generating test inputs to f while MAESTRO controls f indirectly through the entry function of the task that contains f.

Table 3: Branch and MC/DC coverage achieved and crash locations reported by M^{FO} and MAESTRO

Module	Branch coverage (%)		MC/DC coverage (%)		#crash alarms (and # crash lines)	
	M^{FO}	MAESTRO	M^{FO}	MAESTRO	M^{FO}	MAESTRO
APRK	94.6	91.2	88.6	77.9	9177 (15)	0 (0)
BCM	97.1	95.1	89.1	81.2	16579 (32)	0 (0)
SMK	97.8	94.5	89.3	83.5	22353 (62)	0 (0)
TPMS	97.2	95.3	90.1	82.8	13892 (21)	0 (0)
Total	97.2	94.2	89.2	82.3	62001 (130)	0 (0)

However, M^{FO} generated many infeasible test inputs and raised 62,001 false crash alarms. This is because M^{FO} directly generates inputs for every function f that violate the context of f provided by the caller and callee functions of f. We semi-automatically analyzed all 62,001 crash alarms at 130 lines in the target modules reported by M^{FO} and found that all reported crash alarms were false. [10] In contrast, MAESTRO did not raise any crash alarm because it provides valid test inputs to f indirectly through the entry function of the task of f. Thus, we can conclude that MAESTRO reports more reliable coverage information than M^{FO}.

5.5. RQ5. Effect of the Symbolic Bit-field Support

The experiment results show that MAESTRO's support of symbolic bit-fields is effective in increasing branch and MC/DC coverage. Since M^{-SBF} does not generate test inputs for bit-fields at all, it may not cover the branches whose conditions depend on bit-fields (Sect. 2.4.2). For the 143 tasks that use bit-fields, we compare the branch and MC/DC coverage achieved by M^{-SBF} and MAESTRO. M^{-SBF} achieved 51.7% branch and 41.6% MC/DC coverage while MAESTRO achieved 86.3% branch and 79.2% MC/DC coverage for the 143 tasks (i.e., MAESTRO achieved 1.7 times higher branch coverage and 1.9 times higher MC/DC coverage than M^{-SBF}). Thus, we can conclude that this support of symbolic bit-fields increases test coverage for automotive software such as the target modules.

5.6. RQ6. Effect of the Symbolic Setting for Function Pointers

The experiment results show that MAESTRO's symbolic setting for function pointers is effective in increasing branch and MC/DC coverage. Since M^{-SFP} does not set function pointers, the branches that have control-dependency on the function invoked through a function pointer may not be covered by M^{-SFP}. For the 111 tasks that use function pointers, we compare the branch and MC/DC coverage achieved by M^{-SFP} and MAESTRO. M^{-SFP} achieved 64.1% branch and 49.8% MC/DC coverage while MAESTRO achieved 90.2% branch and

[10]First, we classified the 62,001 crashing test inputs in 253 groups by filtering out the input values irrelevant to the crashes at the 130 crash lines. Then, we manually analyzed 253 test inputs, each of which represents a group of the crashing test inputs.

Table 4: Branch and MC/DC coverage achieved by M^{CT}, M^{BF}, M^{TPF}, M^{SH}, and MAESTRO (M represents MAESTRO)

Module	Branch coverage (%)					MC/DC coverage (%)				
	M^{CT}	M^{BF}	M^{TPF}	M^{SH}	M	M^{CT}	M^{BF}	M^{TPF}	M^{SH}	M
APRK	89.6	54.9	78.5	88.3	91.2	78.1	51.6	73.7	73.2	77.9
BCM	94.2	63.1	88.7	92.3	95.1	80.7	53.3	76.2	78.5	81.2
SMK	93.6	61.3	90.6	91.3	94.5	81.9	59.0	76.5	81.1	83.5
TPMS	94.7	63.2	90.8	93.5	95.3	80.9	59.3	78.9	80.5	82.8
Total	93.2	60.8	88.6	91.2	94.2	81.1	57.0	76.2	79.5	82.3

79.6% MC/DC coverage for the 111 tasks (i.e., MAESTRO achieved 40.7% and 59.8% relatively higher branch and MC/DC coverage than M^{-SFP}, respectively). Thus, we can conclude that symbolic setting for function pointers increases test coverage for automotive software.

5.7. RQ7. Effect of the Input Generation Techniques

The experiment results show that the adaptive hybrid approach of concolic testing and fuzzing input generation technique achieves higher branch and MC/DC coverage than the other compared techniques, respectively. Table 4 shows the branch and MC/DC coverage achieved by M^{CT}, M^{BF}, M^{TPF}, M^{SH}, and MAESTRO. MAESTRO achieved the highest branch coverage (94.2%). Similarly, MAESTRO achieved the highest MC/DC coverage (82.3%) which is followed by M^{CT}, M^{SH}, M^{TPF}, and M^{BF}. For example of APRK (see the second row of Table 4), MAESTRO achieved 91.2% of branch coverage which is relatively 66.1% and 16.2% higher than M^{BF} and M^{TPF}, respectively. Compared to M^{SH} and M^{CT}, MAESTRO achieved relatively 3.3% and 1.8% higher branch coverage for APRK, respectively.

M^{BF} achieved the lowest coverage because M^{BF} mutates the type information of the input file without considering the input file format, which generates many invalid inputs that do not increase test coverage. M^{TPF}, on the other hand, preserves the type information of the input file to prevent fuzzing from generating invalid inputs. Thus, M^{TPF} achieved 45.7% and 33.7% higher branch and MC/DC coverage than M^{BF}. M^{CT} achieved higher branch coverage than M^{TPF} because fuzzing is not effective in increasing branch coverage beyond 90% level. Fuzzing technique generates inputs using random mutation so that it can generate many duplicate inputs as time goes on.

5.8. RQ8. Effect of the Symbolic Stub Generation Strategies

The experiment results show that the symbolic stubs generation strategy (Section 3.3.3) of MAESTRO increase branch and MC/DC coverage compared to the baseline symbolic stubs. M^{BS} achieved 87.8% branch and 74.8% MC/DC coverage while MAESTRO achieved 92.8% branch and 80.2% MC/DC coverage for the 768 tasks (i.e., MAESTRO achieved 5.7% and 7.2% relatively higher branch and MC/DC coverage than M^{BS}, respectively). Because the symbolic stubs generated by M^{BS} do not update the global variables and output parameters modified by the functions (and their callee functions) replaced by the symbolic stubs, the branches that have control-dependency on the global variables and local variables that are passed to the symbolic stubs as parameters may not be covered by M^{BS}. Thus, we can conclude that MAESTRO's symbolic stub generation strategy increases test coverage for automotive software.

5.9. Threats to Validity

The primary threat to external validity for our study involves the representativeness of our subject programs, since we have examined only the industrial modules developed by Mobis (i.e., APRK, BCM, SMK, and TPMS). Since those target modules are real-world automotive programs and MAESTRO is not highly dependent on the characteristics of target modules, we believe that this threat to external validity is limited. A primary threat to internal validity is the existence of possible faults in the tools that implement MAESTRO. We controlled this threat through extensive testing of our tool.

6. Lessons Learned

6.1. Practical Benefit of Automated Test Generation in the Automotive Industry

As Sect. 5.1 and 5.3 show, an automated test generation technique like MAESTRO can improve the quality of automotive software by achieving high test coverage (i.e., 94.2% branch coverage) with reduced testing cost (i.e., 58.8% man-month per year on coverage testing) in practice. Although it is not trivial to develop an automated test generation framework that resolves various technical challenges in industrial projects, we believe that the automotive industry can significantly benefit from an automated test generation framework like MAESTRO.

6.2. Necessity of Customization of Automated Test Generation Tools for Target Projects

We have found that it is essential to identify technical challenges and customize an automated test generation tool to address those challenges in a target project. For example, if MAESTRO targeted an individual function as a target unit (not a task), it would generate misleading coverage information and waste human effort to filter out false alarms due to infeasible test inputs generated (Sect. 5.4), which would reduce the benefits of MAESTRO. Additionally, the proposed task-oriented approach might not be highly effective for other projects. Also, if MAESTRO did not support bit-fields nor symbolic setting for function pointers, MAESTRO would not have achieved 94.2% branch coverage, but much less coverage.

Table 5: Branch coverage achieved by M^{CT} and M^{TPF} with 5, 10, 15 and 20 minutes timeout

Module	M^{CT}				M^{TPF}			
	5 mins	10 mins	15 mins	20 mins	5 mins	10 mins	15 mins	20 mins
APRK	53.7	62.9	78.6	89.6	58.1	64.8	71.5	78.5
BCM	61.1	76.4	78.4	94.2	68.1	75.6	78.4	88.7
SMK	61.6	77.0	88.5	93.6	67.6	76.8	84.0	90.6
TPMS	64.0	84.2	88.6	94.7	73.2	80.6	83.0	90.8
Total	60.6	75.4	85.4	93.2	66.7	75.2	81.2	88.6

6.3. Precise Driver/Stub Generation for Automated Test Generation

As shown in Section 5.2 (i.e., 42.2% of the uncovered branches were due to the imprecise drivers and stubs), we need to generate precise drivers/stubs that provide input environments close to the real ones of a target unit. A test driver/stub generated by MAESTRO sometimes does not accurately represent a real environment because the test driver/stub provides symbolic inputs only for variables reachable within a given pointer reference bound (i.e., k=4 in Section 3.3.2).

This issue is difficult to address. Simply increasing the bound value to a larger number (i.e., $k > 4$) may not increase the coverage because increasing the bound value enlarges symbolic execution space and exploring the enlarged symbolic execution space within a fixed amount of testing time may decrease the coverage. To make MAESTRO more practical for automotive SW, we need to generate more precise drivers/stubs that provide realistic input environments for a target task.

6.4. Comparison between Concolic Testing and Fuzzing for Automated Test Generation

Concolic testing and fuzzing have different characteristics for coverage testing. Table 5 shows the branch coverage and MC/DC coverage of M^{CT} and M^{TPF} for different amount of execution time. M^{TPF} achieved more branches than M^{CT} in shorter time (i.e., 5 minutes timeout for each task), because fuzzing can generate many test inputs by mutating the seed inputs and run the generated test inputs quickly. With 10 minutes timeout, concolic testing and fuzzing covers almost the same amount of branches. With 15 and 20 minutes timeout, concolic testing achieves more branches than fuzzing. This is because fuzzing may generate test inputs that follow already explored execution paths due to random mutation of the previous inputs while concolic testing tries to avoid generating test inputs that follow already explored paths.

To improve test coverage, MAESTRO hybridizes concolic testing and fuzzing in an adaptive way. The key of the adaptive hybrid input generator is that it checks the saturation of the test coverage frequently (e.g., every 2 minutes in our experiments) and changes to another input generation technique if a current one reaches the coverage saturation point. As a result, the adaptive hybrid input generator of MAESTRO achieved higher test coverage than any single compared input generation technique.

Table 6: Comparison between MAESTRO and Angora on an original target program P and a converted one P' (to which bit-field transformation and function pointer support are applied)

Targets	MAESTRO		Angora on P		Angora on P'	
	Branch	MC/DC	Branch	MC/DC	Branch	MC/DC
APRK	91.2	77.9	78.0	67.0	84.2	71.0
BCM	95.1	81.2	82.4	68.8	84.3	75.1
SMK	94.5	83.5	81.9	72.5	84.9	76.2
TPMS	95.3	82.8	80.2	70.7	85.7	73.3
Total	94.2	82.3	81.1	70.8	84.7	74.8

For further comparison, we have compared MAESTRO with another cutting-edge fuzzer Angora [32] (with default setting) using the type information-preserving fuzzing. We ran Angora for the same amount of time spent for the MAESTRO experiment. Still, Table 6 shows that MAESTRO achieves higher branch and MC/DC coverage than Angora even on a program P' convereted by MAESTRO for high test coveraege (i.e., 94.2% vs 84.7% for branch coverage and 82.3% vs. 74.8% for MC/DC coverage).

7. Related Work

7.1. Automated Test Input Generation Techniques

7.1.1. Concolic Testing

Concolic techniques can be grouped into *instrumentation-based one* and *Virtual machine (VM)-based one*.

Instrumentation-based concolic testing techniques insert probes in target source code to obtain dynamic execution information to build symbolic path formulas. This approach is lighter and easier-to-customize than the VM-based one. However, it requires complex source code parsing and instrumentation. CUTE [9], DART [10], CREST [27] (and its distributed version SCORE [33]), CROWN [23] target C programs and jCUTE [34] and CATG [35] target Java programs. MAESTRO uses CROWN as its concolic testing engine because CROWN (and its predecessor CREST) has been successfully applied to various industrial projects (Sect. 7.3).

VM-based concolic testing techniques run as a layer on top of a VM to interpret compiled IR code of a target program p and obtain symbolic path formulas from p's executions. This approach can conveniently obtain all detailed run-time execution information of p available to a VM. However, test generation speed is slow due to slow IR interpretation and customizing the tools is non-trivial due to complex VM infrastructure. PEX [36] targets C# programs that are compiled to Microsoft .Net binaries (now available as IntelliTest [36] in Visual Studio). KLEE [28] (and its distributed version Cloud9 [37]) targets LLVM [25] binaries. jFuzz [38] and Symbolic PathFinder [39] target Java bytecode programs on top of Java PathFinder [40].

7.1.2. Fuzzing

Fuzzing [41, 42] was developed to detect crash bugs and security vulnerability of target programs. These days, fuzzing is also used for detecting bugs in programs such as compilers [43, 44], web browsers [45], OS kernels [46], and network protocol implementations [47]. Fuzzing is used in not only academics, but also industries. Microsoft provides a commercial fuzzing service [48] whose engine has been used to detect bugs in Windows OS and Office, and Google has developed ClusterFuzz [49] to detect bugs in Google Chrome and open-source software.

Fuzzers first generate a large number of test inputs for a target program. Then, fuzzers run the target program with the test inputs and report bugs if crashes or assert violations are detected. Mutation-based fuzzers generate the test inputs by mutating seed test inputs [24, 50, 51]. To achieve high test coverage and high bug detection ability, mutation-based fuzzers employ coverage-guided fuzzing techniques. Coverage-guided fuzzing evolves the generated test inputs towards high code coverage. In a fuzzing loop, coverage-guided fuzzing measures test coverage for each test execution. Then, it chooses test inputs that increase coverage and insert the chosen test inputs into a pool of seed inputs for high test coverage.

7.2. Automated Test Driver/Stub Generation

DART [10] generates symbolic unit test drivers, but not symbolic stubs for concolic testing. To avoid the infeasible test generation issue, DART targets public API functions in libraries because such functions should accept all possible inputs. UC-KLEE [52] directly starts symbolic execution from a target function using lazy initialization [53] and calls all the functions directly or transitively invoked by the target function. Thus, DART and UC-KLEE target code of a function and its all callee functions, which can make concolic testing achieve low coverage within a fixed amount of testing time because the symbolic execution space can become very large. Chakrabarti and Godefroid [54] statically divide a static call graph into partitions using topological information and consider the partitions as testing targets for concolic testing. However, the proposed partitioning method does not consider semantic information on the relation between functions.

CONBOL [14] generates symbolic unit test driver and stubs for C functions in large-scale embedded software. It replaces all functions invoked by a target function by symbolic stubs. CONBOL uses target project specific false alarm reduction heuristics, which may not be effective for other projects. SmartUnit [55] generates symbolic test drivers and stubs for C functions (the authors do not clearly describe how SmartUnit generates driver and stubs). The paper reports that SmartUnit achieved high coverage, but it does not report how many false alarms were raised. We could not directly compare the performance of MAESTRO with CONBOL and SmartUnit since they are not publicly available.

CONBRIO [23] constructs extended units as testing target units by using highly-relevant callee of a target function. The relevance between functions is computed based on system-level execution profiles. Targeting the extended units, CONBRIO achieves high bug detection power with low false alarm ratio. However, CONBRIO was not applicable to ADAS and IBU in this testing project because we could not obtain ADAS and IBU execution profiles by driving physical motor vehicles.

7.3. Industrial Application of Concolic Testing

Microsoft developed SAGE [56, 57] for x86/64 binaries to detect security vulnerabilities of Windows and Office products. Bardin and Herrmann [58, 59] developed and applied OSMOSE to embedded software. They translated machine code into an intermediate representation to apply concolic testing. Intel developed and applied MicroFormal [60] for Intel CPU's microcode. Fujitsu developed KLOVER [61] by extending KLEE targeting C++ programs. Zhang et al. developed and applied SmartUnit [55] to embedded software to achieve high branch and MC/DC coverage. Kim et al. applied CREST to the Samsung flash memory device driver code [62, 11]. They also compared CREST and KLEE for industrial use of concolic testing for the Samsung mobile phone software [13] and developed a systematic event-sequence generation framework using CREST for LG electric oven [63].

These industrial case studies focused on increasing the test effectiveness but did not report how much manual testing effort was saved. In contrast, this paper reports how much manual testing effort (in man-months) MAESTRO saved in the automotive company (Section 5.2). Also, we have shared the technical challenges and the solutions for the application of concolic testing to automotive software, which can promote field engineers to adopt concolic testing in their projects.

8. Conclusion and Future Work

We have presented the industrial study of applying MAESTRO to the automotive software developed by Mobis. After we identified and addressed the technical challenges of applying automated test generation to automotive software, we have developed an automated test generation framework MAESTRO. It generates a task-oriented test driver and stubs to reduce infeasible test executions and supports bit-fields input generation, input setting for function pointers that automotive software uses, and improved symbolic stub generation to increase test coverage. Also, MAESTRO applies the hybrid input generation technique that utilizes both concolic testing and fuzzing in an adaptive way. MAESTRO has achieved 94.2% branch and 82.3% MC/DC coverage on the four target modules, and reduced the manual testing effort by 58.8%.

As future work, we plan to generate more precise test driver/stubs by adopting advanced techniques (e.g., MCDC coverage improvement [64, 65, 66] and/or distributed concolic testing [33, 67, 68]). Also, we will extend MAESTRO by applying compositional concolic testing [69] and apply it to other automotive software modules.

Acknowledgment

We thank Ahcheong Lee and Hyunwoo Kim for their initial effort of applying MAESTRO to IBU. This research has been supported by Hyundai Mobis, Next-Generation Information Computing Development Program through NRF funded by MSIT (No. NRF-2017M3C4A7068177), Basic Science Research Program through NRF funded by MSIT (NRF-2019R1A2B5B01069865), and Basic Science Research Program through NRF funded by the Ministry of Education (NRF-2017R1D1A1B03035851).

References

[1] Manfred Broy. Challenges in automotive software engineering. In *ICSE*, 2006.

[2] M. Broy, I. H. Kruger, A. Pretschner, and C. Salzmann. Engineering automotive software. *Proc. IEEE*, 95(2), 2007.

[3] F. Falcini, G. Lami, and A. M. Costanza. Deep learning in automotive software. *IEEE Software*, 34(3), 2017.

[4] M. Traub, A. Maier, and K. L. Barbehön. Future automotive architecture and the impact of it trends. *IEEE Software*, 34(3):27–32, May 2017.

[5] M. W. Whalen, D. Cofer, and A. Gacek. Requirements and architectures for secure vehicles. *IEEE Software*, 33(4), 2016.

[6] K. T. Le, Y. Chibay, and T. Aoki. Formalization and verification of autosar os standard's memory protection. In *TASE*, 2018.

[7] X. Guo, H. Lin, T. Aoki, and Y. Chiba. A reusable framework for modeling and verifying in-vehicle networking systems in the presence of can and flexray. In *APSEC*, 2017.

[8] Yunja Choi. A configurable v&v framework using formal behavioral patterns for osek/vdx operating systems. *JSS*, 137, 2018.

[9] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A concolic unit testing engine for C. In *ESEC/FSE*, 2005.

[10] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. In *PLDI*, 2005.

[11] Moonzoo Kim, Yunho Kim, and Yunja Choi. Concolic testing of the multi-sector read operation for flash storage platform software. *FACJ*, 24(3), 2012.

[12] Moonzoo Kim, Yunho Kim, and Yoonkyu Jang. Industrial application of concolic testing on embedded software: Case studies. In *ICST*, 2012.

[13] Yunho Kim, Moonzoo Kim, YoungJoo Kim, and Yoonkyu Jang. Industrial application of concolic testing approach: A case study on libexif by using CREST-BV and KLEE. In *ICSE*, 2012.

[14] Yunho Kim, Youil Kim, Taeksu Kim, Gunwoo Lee, Yoonkyu Jang, and Moonzoo Kim. Automated unit testing of large industrial embedded software using concolic testing. In *ASE*, 2013.

[15] Hyundai mobis increases ai use for improved efficiency. `http://www.koreaherald.com/view.php?ud=20180722000186`.

[16] Hyundai mobis taps ai for car software. `https://www.koreatimes.co.kr/www/tech/2018/07/419_252629.html`.

[17] Hyundai mobis introduces ai-based software verification system. `http://www.businesskorea.co.kr/news/articleView.html?idxno=23830`.

[18] Yunho Kim, Dongju Lee, Junki Baek, and Moonoo Kim. Concolic testing for high test coverage and reduced human effort in automotive industry. In *ICSE*, 2019.

[19] Open hub, the open source network. `https://www.openhub.net/`.

[20] Sina Shamshiri, Rene Just, Jose Miguel Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges. In *Automated Software Engineering (ASE)*, 2015.

[21] G.Fraser and A.Arcuri. 1600 faults in 100 projects: automatically finding faults while achieving high coverage with evosuite. *Empirical Software Engineering*, 20:611–639, 2015.

[22] Florian Gross, Gordon Fraser, and Andreas Zeller. Search-based system testing: High coverage, no false alarms. In *ISSTA 2012*, pages 67–77, New York, NY, USA, 2012. ACM.
[23] Yunho Kim, Yunja Choi, and Moonzoo Kim. Precise concolic unit testing of C programs using extended units and symbolic alarm filtering. In *ICSE*, 2018.
[24] Michal Zalewski. American fuzzy lop (afl) fuzzer. `http://lcamtuf.coredump.cx/afl/`, 2017.
[25] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, 2004.
[26] G. Necula, S. McPeak, S. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *CC*, 2002.
[27] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *ASE*, 2008.
[28] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.
[29] N. Williams, B. Marre, P. Mouy, and M. Roger. Pathcrawler: automatic generation of path tests by combining static and dynamic analysis. In *EDCC*, 2005.
[30] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Comput. Surv.*, 51(3), 2018.
[31] Testwell CTC++. `https://www.verifysoft.com/en_ctcpp.html`.
[32] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *IEEE Symposium on Security and Privacy*, 2018.
[33] Moonzoo Kim, Yunho Kim, and Gregg Rothermel. A scalable distributed concolic testing approach: An empirical evaluation. In *ICST*, 2012.
[34] K. Sen and G. Agha. CUTE and jCUTE : Concolic unit testing and explicit path model-checking tools. In *CAV*, 2006.
[35] CATG: Concolic testing engine for Java. `https://github.com/ksen007/janala2`.
[36] Nikolai Tillmann and Jonathan De Halleux. Pex: White box test generation for .NET. In *TAP*, 2008.
[37] L. Ciortea, C. Zamfir, S. Bucur, V. Chipounov, and G. Candea. Cloud9: a software testing service. *OSR*, 43(4), 2010.
[38] K. Jayaraman, D. Harvison, V. Ganesh, and A. Kiezun. jFuzz: A concolic whitebox fuzzer for Java. In *NFM*, 2009.
[39] Corina S. Păsăreanu, Peter C. Mehlitz, David H. Bushnell, Karen Gundy-Burlet, Michael Lowry, Suzette Person, and Mark Pape. Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In *ISSTA*, 2008.
[40] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *ASE*, 2000.
[41] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen. CollAFL: Path sensitive fuzzing. In *IEEE SP*, 2018.
[42] Jun Li, Bodong Zhao, and Chao Zhang. Fuzzing: a survey. *Cybersecurity*, 1(1), 2018.
[43] Chris Cummins, Pavlos Petoumenos, Alastair Murray, and Hugh Leather. Compiler fuzzing through deep learning. In *ISSTA*, 2018.
[44] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F. Donaldson. Many-core compiler fuzzing. In *PLDI*, 2015.
[45] L. Li, Q. Dong, D. Liu, and L. Zhu. The application of fuzzing in web software security vulnerabilities test. In *ICITA*, 2019.
[46] Shankara Pailoor, Andrew Aday, and Suman Jana. Moonshine: Optimizing OS fuzzer seed selection with trace distillation. In *SEC*, 2018.
[47] Serge Gorbunov and Arnold Rosenbloom. Autofuzz: Automated network protocol fuzzing framework. *IJCSNS*, 10(8), 2010.
[48] Microsoft. Microsoft security risk detection. `https://www.microsoft.com/en-us/security-risk-detection/`.
[49] Google. Clusterfuzz. `https://google.github.io/clusterfuzz/`.
[50] K. Serebryany. Continuous fuzzing with libfuzzer and addresssanitizer. In *SecDev*, 2016.
[51] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *NDSS*, 2017.
[52] David A. Ramos and Dawson Engler. Under-constrained symbolic execution: Correctness checking for real code. In *SEC*, 2015.
[53] Sarfraz Khurshid, Corina S. Păsăreanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In *TACAS*, 2003.
[54] Arindam Chakrabarti and Patrice Godefroid. Software partitioning for effective automated unit testing. In *EMSOFT*, 2006.
[55] Chengyu Zhang, Yichen Yan, Hanru Zhou, Yinbo Yao, Ke Wu, Ting Su, Weikai Miao, and Geguang Pu. Smartunit: Empirical evaluations for automated unit testing of embedded software in industry. In *ICSE*, 2018.
[56] P. Godefroid, A. Kiezun, and M. Y. Levin. Grammar-based whitebox fuzzing. In *PLDI*, 2008.
[57] Microsoft security risk detection. `https://www.microsoft.com/en-us/security-risk-detection/`.
[58] S. Bardin and P. Herrmann. Structural testing of executables. In *ICST*, 2008.
[59] S. Bardin and P. Herrmann. OSMOSE: automatic structural testing of executables. *STVR*, 21(1), 2011.
[60] A. Franzén, A. Cimatti, A. Nadel, R. Sebastiani, and J. Shalev. Applying SMT in symbolic execution of microcode. In *FMCAD*, 2010.
[61] Guodong Li, Indradeep Ghosh, and Sreeranga P. Rajan. Klover: A symbolic execution and automatic test generation tool for c++ programs. In *CAV*, 2011.
[62] M. Kim and Y. Kim. Concolic testing of the multi-sector read operation for flash memory file system. In *SBMF*, 2009.
[63] Y. Park, S. Hong, M. Kim, D. Lee, and J. Cho. Systematic testing of reactive software with non-deterministic events: A case study on LG electric oven. In *ICSE*, 2015.
[64] S.Godboley, Arpita Dutta, Avijit Das, D.P.Mohapatra, and Rajib Mall. Making a concolic tester achieve increased mc/dc. *Innovations Systems and Software Engineering*, 12:319–332, 2016.
[65] S.Godboley, A.Dutta, D.P.Mohapatra, and R.Mall. J3 model: A novel framework for improved modified condition/decision coverage analysis. *Computer Standards and Interfaces*, 50:1–17, Feb 2017.
[66] S.Godboley, A.Dutta, D.P.Mohapatra, and R.Mall. GECOJAP: A novel source-code preprocessing technique to improve code coverage. *Computer Standards and Interfaces*, 55:27–46, 2018.
[67] S.Godboley, D.P.Mohapatra, A.Das, and R.Mall. An improved distributed concolic testing approach. *Software: Practice and Experience*, 47(2):311–342, 2017.
[68] S.Godboley, A.Dutta, D.P.Mohapatra, and R.Mall. Scaling modified condition / decision coverage using distributed concolic testing for Java programs. *Computer Standards and Interfaces*, 59:61–86, 2018.
[69] Y.Kim, S.Hong, and M.Kim. Target-driven compositional concolic testing with function summary refinement for effective bug detection. In *European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE)*, 2019.