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

Yunho Kim\textsuperscript{a}, Dongju Lee\textsuperscript{b}, Junki Baek\textsuperscript{b}, Moonzoo Kim\textsuperscript{a,}\textsuperscript{*}

\textsuperscript{a}KAIST, 291 Daehak-ro, Yuseong-gu, Daejeon, South Korea 34141
\textsuperscript{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) \cite{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) \cite{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 \cite{5, 6, 7, 8} to achieve high code quality with reduced human effort.

Concolic testing \cite{9} \cite{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 \cite{11}, mobile phone software \cite{12, 13}, and large-scale embedded software \cite{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-
ating test cases corresponding to the executions that are not feasible at the system-level. Otherwise (e.g., generating naïve 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.

---

¹A 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.
²Several 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 ultrasonic 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. 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 te 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 te 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:

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.

```
01: int rpm_FL, rpm_FR, rpm_RL, rpm_RR;
02: int angle_FL, angle_FR;
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<E_PRESSURE) &&
15: (L_PRESSURE<press_FR && press_FR<E_PRESSURE) &&
16: (L_PRESSURE<press_RL && press_RL<E_PRESSURE) &&
17: (L_PRESSURE<press_RR && press_RR<E_PRESSURE)) { ...
```

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

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

4We exclude Linux Kernel because our Clang-based analysis tool fails to analyse GCC-specific code of Linux Kernel.
Table 1: The code statistics of the target modules

<table>
<thead>
<tr>
<th>Module</th>
<th>#files</th>
<th>#functions</th>
<th>LoC</th>
<th>#branches</th>
<th>Cyclomatic comp.</th>
</tr>
</thead>
<tbody>
<tr>
<td>APRK</td>
<td>47</td>
<td>142</td>
<td>451</td>
<td>593</td>
<td>34292 4655 1~18 4.9</td>
</tr>
<tr>
<td>BCM</td>
<td>27</td>
<td>145</td>
<td>511</td>
<td>656</td>
<td>52690 2873 1~15 4.3</td>
</tr>
<tr>
<td>SMK</td>
<td>198</td>
<td>554</td>
<td>1967</td>
<td>2521</td>
<td>134877 13768 1~24 5.8</td>
</tr>
<tr>
<td>TPMS</td>
<td>29</td>
<td>68</td>
<td>234</td>
<td>302</td>
<td>15951 1217 1~12 4.1</td>
</tr>
<tr>
<td>Total</td>
<td>301</td>
<td>909</td>
<td>3163</td>
<td>4072</td>
<td>237810 22513 1~24 5.3</td>
</tr>
</tbody>
</table>

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.

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.

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

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.

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

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.

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-

5A 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.

6The target projects often use this code pattern to update multiple bit-fields at once.
concolic 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 [RamewOrk]), 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 \( a_1 \) to \( a_4 \). \( t_1 \) consists of the entry function \( f \) and its callee functions \( a_1 \), \( a_2 \) and \( a_3 \) defined in the same file (i.e., file1.c). The function \( h \) invoked by \( a_3 \) 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 \( a_4 \). 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. \( a_3 \) 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 \( a_3 \) 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 \( \text{SYM}_{<T>}(x) \) 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 \( \text{sizeof}(T) \times n \) 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 a 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 \( \text{SYM}_{\text{bitfield}}(b) \).
Figure 4: MAESTRO framework

Figure 5 shows an example of unit test driver code generated. 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(char 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.

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. Then, the symbolic stub...

---

7The most part of the example is excerpted from Kim et al. [14].
8These baseline symbolic stubs were used by MAIST [18] (the predecessor of MAESTRO).

9MAESTRO ignores the variable updates through pointers due to the low precision of the static pointer analysis.
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 $f_1$ used in a target task. First, MAESTRO statically examines all target source files and identifies functions $f_1, ..., f_n$ that are assigned to $f_1$. Then, it adds code $c_{p_{f_1}}$ to a test driver such that $c_{p_{f_1}}$ assigns each of $f_1, ..., f_n$ to $f_1$ using a symbolic variable choice which selects each of $f_1, ..., f_n$ to assign to $f_1$.

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 $c_{p_{f_1}}$ 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 using 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 in $P'$ (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., $ba_0$, $ba_1$, ...). 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 $x$, 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 $ba$ 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 $ba$ 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:   int y:5;
  4:   unsigned char ba0[3];
  5: };

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., $ba_0$) 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 $ba_0$).

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 $i = 0; i < \text{the number of fields in } S_b; i + +$ do
7. 1. if $field_i$ is a bit-field then
8. 2. $paddingBitSize = getPaddingBitSize(field_i, curBitSize_{ba}, curByteSize);$  
9. 3. $mapBFtoBA.add(field_i, <i_{ba}, curBitSize_{ba} + paddingBitSize, field_i.bitSize>);$  
10. 4. $bitSize_{ba} += field_i.bitSize + paddingBitSize;$  
11. 5. if $field_i$ is the last field of $S_b$ or $field_{i+1}$ is not a bit-field then
12. 6. $appendBitArray(S_{nb}, i_{ba}, bitSize_{ba}); // Add $ba_{<i_{ba}>}$ field (e.g., $ba_0, ba_1, ...$ ) to $S_{nb}$
13. 7. $bitSize_{ba} = 0;$  
14. 8. $i_{ba} += ;$
15. else
16. 9. $appendField(S_{nb}, field_i.name, field_i.type);$  
17. 10. $curByteSize += field_i.byteSize + getPaddingByteSize(field_i, S_{nb}.byteSize);$  
18. end
19. end
20. if $curByteSize < byteSize(S_b)$ then
21. 11. $// Add a new padding variable$
22. 12. $appendPaddingVar(S_{nb}, byteSize(S_b) - curByteSize);$  
23. end
24. return $S_{nb}$ and $mapBFtoBA$;
25. }
26. $getPaddingBitSize(bitField, curBitSize_{ba}, curByteSize) \{$
27. // Compute the padding size to align $bitField$ variable according to its type
28. $paddingBitSize = bitField.typeSize * 8 - ((curBitSize_{ba} + curByteSize * 8) \% (bitField.typeSize * 8));$
29. if $bitField.bitSize > paddingBitSize$ then
30. 1. $return paddingBitSize;$  
31. else
32. 2. $return 0;$  
33. end
34. }
35. $getPaddingByteSize(field, curByteSize) \{$
36. // Compute the padding size to align $field$ variable according to its type
37. $paddingByteSize = field.typeSize - (curByteSize \% field.typeSize);$  
38. if $field.typeSize > paddingByteSize$ then
39. 3. $return paddingByteSize;$  
40. else
41. 4. $return 0;$  
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 $ba_0$, the starting bit location of $field_1$ in $ba_0$ 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 in-
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 \( ba_0 \) 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).

4. Finally, the algorithm returns the transformed struct \( S_{nb} \) and \( mapBFtoBA \) as inputs and returns the transformed expression on \( v_b \) whose value is equal to \( S.z \) at Line 25.

3.4.2. Bit-field Read Expression Transformation

Algorithm 2 takes the bit-field \( v_b \) (e.g., \( S.z \)) to read and a map \( mapBFtoBA \) (e.g., \( \{ y \rightarrow <0,0,5>, z \rightarrow <0,5,18>, \} \)) as inputs and returns the transformed expression on \( v_b \) whose value is the same as \( v_b \). 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 \( v_b \) is included (i.e., \( ba <i_{ba}> \)). \( i_{ba} \) is obtained from \( mapBFtoBA \).get(\( v_b \)). For \( v_b = S.z \), \( i_{ba} \) is 0.

  - \( pos_l \); the starting bit position of \( v_b \) in \( ba <i_{ba}> \). \( pos_l \) is obtained from \( mapBFtoBA \).get(\( v_b \)). For \( v_b = S.z \), \( pos_l \) is 5.

  - \( bitSize \); the size of \( v_b \) in bit. \( bitSize \) is obtained from \( mapBFtoBA \).get(\( v_b \)). For \( v_b = S.z \), \( bitSize \) is 18.

- \( pos_h \); the ending bit position of \( v_b \) in \( ba <i_{ba}> \). \( pos_h \) is set as \( pos_l + bitSize - 1 \). For \( v_b = S.z \), \( pos_h \) is 22 (\( 5 + 18 \)).

- \( bit \); the variable name (i.e., string) of \( i_{th} \) bit array \( ba <i_{ba}> \) field in a string type (i.e., \( S_{nb}.ba0 \), \( S_{nb}.ba1 \), ...). For \( v_b = S.z \), \( ba \) is \( S_{nb}.ba0 \).

- \( locByte \); the lowest byte address of \( v_b \) in \( ba <i_{ba}> \). \( locByte \) is set as \( [pos_l/8] \). For \( v_b = S.z \), \( locByte \) is 0.

- \( locByte_h \); the highest byte address of \( v_b \) in \( ba <i_{ba}> \). \( locByte_h \) is set as \( [pos_h/8] \). For \( v_b = S.z \), \( locByte_h \) is 2.

- \( locBit_l \); the beginning bit offset of \( v_b \) in \( ba <i_{ba}> \). \( locBit_l \) is set as \( pos_l \% 8 \). For \( v_b = S.z \), \( locBit_l \) is 5.

- \( locBit_h \); the ending bit offset of \( v_b \) in \( ba <i_{ba}> \). \( locBit_h \) is set as \( pos_h \% 8 \). For \( v_b = S.z \), \( locBit_h \) 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 \( v_f \) at Line 15. For \( v_f = S.z \), \( exp \) 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 \( v_b \) in \( ba <i_{ba}> \).

   i. At the first iteration of the loop (i.e., \( i = 1 \)), the algorithm adds the C expressions to \( exp \) to represent the data of ith byte of \( v_b \) in \( ba <i_{ba}> \) at Line 17. For \( v_b = S.z \), \( exp \) 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 \( locBit_l \) bits of the highest byte of \( v_f \). For \( v_f = S.z \), \( bitMask \) 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 \( v_f \) at Line 20. After Line 20, \( exp \) becomes \( (S_{nb}.ba0[2] <<16) | (S_{nb}.ba0[1] <<8) | S_{nb}.ba0[0] \).

3. It adds the C expression that performs right-shift \( locBit_l \) times to \( exp \) at Line 22. For \( v_f = S.z \), \( exp \) 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 \( v_b \) (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., \( S.z = c \)) in which a bit-field is on the left-hand side (LHS) and a map \( mapBFtoBA \) (e.g., \( \{ y \rightarrow <0,0,5>, z \rightarrow <0,5,18>, \} \)) as inputs.
Algorithm 2: Bit-field read expression transformation algorithm

**Input:** va[i]: a bit-field to read (e.g., S.z[i]), 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 va[i].

```plaintext
1 transformReadExp(va[i], mapBFtoBA) {
2    // va[i] is included in ba<i ba>, pos: the starting bit position of va[i] in ba<i ba>, bitSize: the size of va[i] in bit
3    <i ba>, pos, bitSize = mapBFtoBA.get(va[i]);
4    posh = pos + bitSize * 1; // the ending bit position of va[i] 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    locByte1 = [posh/8]; // the lowest byte address of va[i] in ba<i ba>
7    locByteh = [posh/8]; // the highest byte address of va[i] in ba<i ba>
8    locBitl = pos % 8; // the beginning bit offset of va[i] in the lowest byte of ba<i ba>
9    locBith = pos % 8; // the ending bit offset of va[i] in the highest byte of ba<i ba>
10   bitMask = 0;
11   if (locByte1 == locByteh) then
12       setBits(bitMask, locBitl, locBith, 1);
13       exp = ba[locByte1] & bitMask;
14   else
15       exp = ba[locByte1];
16       for i = 1; i < locByteh - locByte1; i + + do
17           exp = (ba[i + locByte1] << 8 * i)]exp;
18       end
19       setBits(bitMask, 0, locBitl, 1);
20       exp = ((ba[locByte1] & bitMask) << 8 * (locByteh - locByte1))]exp;
21   end
22   exp = (exp) >> locBitl;
23   return exp,
24 }
25 // Set var's from th to tack bits to bitVal. The bit index starts from 0.
26 setBits(var, from, to, bitVal) {...}
```

→ <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:

```plaintext
S_nb.ba0[2] = (((e) << 5) >> (2*8)) & 127
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.

- va[i]: the bit-field in LHS of stmt assign. For stmt assign (i.e., S.z=e), va[i] is S.z.
- expRHS is RHS of stmt assign with parentheses. For stmt assign (i.e., S.z=e), expRHS 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 locByte1 is equal to locByteh, 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 locByte1 and locByteh are 0 and 2, respectively.

(a) The algorithm sets bitMask at Line 19 to extract expRHS to the seventh bits of expRHS << locBitl. For stmt assign (i.e., S.z=e), bitMask is set as 224 (0b11100000 in a binary representation) because locBitl is 5.

(b) It sets expChangedBits that represents the updated bits of the lowest byte of ba<i ba> that corresponds to va[i]. After Line 20, expChangedBits becomes ((e) << 5) & 224.

(c) It sets expUnchangedBits that represents the remaining unchanged bits of the lowest byte of ba<i ba> that
Algorithm 3: Bit-field write statement transformation algorithm

Input: stmtassign: 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: stmttrans: A compound statement consisting of the transformed assignment statements that do not use a bit-field, which is equivalent to stmtassign.

1 transformWriteStmt(stmtassign, mapBFtoBA) {
2 \( v_{bf} \) = the bit-field in LHS of stmtassign; // e.g., $S.z$ in $S.z=e$
3 exprRHS = RHS of stmtassign with parentheses; // e.g., (e) in $S.z=e$
4 \( i_{bf} \): \( v_{bf} \) is included in \( ba_{<i_{ba}>} \), \( posi \): the starting bit position of \( v_{bf} \) in \( ba_{<i_{ba}>} \), \( bitSize \): the size of \( v_{bf} \) in bit
5 \( <i_{ba}, posi, bitSize> = mapBFtoBA.get(v_{bf}); \)
6 \( posh = posi + bitSize - 1; \) // 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 locBytei = \( \lfloor posi/8 \rfloor \); // the lowest byte address of \( v_{bf} \) in \( ba_{<i_{ba}>} \)
9 locByteh = \( \lfloor posh/8 \rfloor \); // the highest byte address of \( v_{bf} \) in \( ba_{<i_{ba}>} \)
10 locBitl = \( \lfloor posi \% 8 \rfloor \); // the beginning bit offset of \( v_{bf} \) in the lowest byte of \( ba_{<i_{ba}>} \)
11 locBith = \( \lfloor posh \% 8 \rfloor \); // the ending bit offset of \( v_{bf} \) in the highest byte of \( ba_{<i_{ba}>} \)
12 bitMask = 0;
13 if locBytei == locByteh then
14 setBits(bitMask, locBitl, locBith, 1);
15 \( exp_{changedBits} = (exprRHS << locBitl) \& bitMask; \)
16 \( exp_{unchangedBits} = ba_{locBytei} \& \neg bitMask; \)
17 stmttrans = ba_{locBytei} = (\( exp_{changedBits} \| exp_{unchangedBits} \))
18 else
19 setBits(bitMask, locBitl, 7, 1);
20 \( exp_{changedBits} = (exprRHS << locBitl) \& bitMask; \)
21 \( exp_{unchangedBits} = ba_{locBytei} \& \neg bitMask; \)
22 stmttrans = ba_{locBytei} = (\( exp_{changedBits} \| exp_{unchangedBits} \))
23 for \( i = 1; i < locByteh - locBytei; i + + \) do
24 \( expr_{dataByte} = extractBits(exprRHS << locBitl, i \* 8, i \* 8 + 7); \)
25 stmttrans = ba_{i + locBytei} = (\( expr_{dataByte} \); stmttrans;
26 end
27 bitMask = 0;
28 setBits(bitMask, 0, locBith, 1);
29 \( exp_{changedBits} = extractBits(exprRHS << locBitl, (locByteh - locBytei) \* 8, (locByteh - locBytei) \* 8 + locBith) \& bitMask;
30 \( exp_{unchangedBits} = ba_{locByteh} \& \neg bitMask; \)
31 stmttrans = ba_{locBytei} = (\( exp_{changedBits} \| exp_{unchangedBits} \))
32 end
33 return stmttrans;
34 }
35
36 // Return a string converted from an integer value num
37 str(num) {...}
38 // Set var’s fromth to toth bits to bitVal. The bit index starts from 0.
39 setBits(var, from, to, bitVal) {...}
40 // Return var’s fromth to toth bits. The bit index starts from 0.
41 extractBits(num, from, to) {...}

coresponds to \( v_{bf} \). After Line 21, \( exp_{unchangedBits} \) becomes \( S_nb.ba0[0] \& 31 \) (31 is 0b00001111).

(d) It sets stmtresults to assign the RHS to the lowest byte of \( ba_{<i_{ba}>} \) that corresponds to \( v_{bf} \) using \( exp_{changedBits} \) and \( exp_{unchangedBits} \). After

(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 \( i = 1, \ldots, \text{locByte}_b - \text{locByte}_t - 1 \) which is an index to the byte of \( ba<i_ba> \) to update, the algorithm generates the C statement that assigns \( i \)th byte data of \( \text{exprHS} \) to \( ba<i_ba> \).

The algorithm sets \( \text{bitMask} \) at Line 24. After Line 24, \( \text{exp}_\text{stmt} \) becomes \(((e)<<(5)>>1)8) & 255 \) for \( \text{stmt}_\text{assign} \) (i.e., \( S.z=e; \)) at Line 24. After Line 24, \( \text{exp}_\text{dataByte} \) becomes \(((e)<<(5)>(1*8))) & 255 \).

A. The algorithm extracts the first byte data of \( \text{exprHS} \) to \( ba<i_ba> \).

B. The algorithm adds the new transformed C statement to \( \text{stmt}_\text{trans} \) where the new C statement assigns the first byte data of \( \text{exprHS} \) to \( ba<i_ba> \).

i. At the first iteration of the loop (i.e., \( i = 1 \)), the algorithm adds the new transformed C statement to \( \text{stmt}_\text{trans} \) where the new C statement assigns the first byte data of \( \text{exprHS} \) to \( ba<i_ba> \).

ii. After the first iteration, the loop terminates.

(f) The algorithm sets \( \text{bitMask} \) to extract 0th to \( \text{locBit}_b \)th bits of the highest byte of \( \text{exprHS} \) to \( \text{locBit}_t \) at Lines 27–28. For \( \text{stmt}_\text{assign} \) (i.e., \( S.z=e; \)), \( \text{bitMask} \) is set as 127 (0b01111111) because \( \text{locBit}_b \) is 6.

(g) It sets \( \text{exp}_\text{changedBits} \) 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, \( \text{exp}_\text{changedBits} \) becomes \(((e)<<(5)>>2)8) & 127.

(h) It sets \( \text{exp}_\text{unchangedBits} \) 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, \( \text{exp}_\text{unchangedBits} \) becomes \( S_nb.ba0[2] \) & 128 (128 is 0b10000000).

(i) It sets \( \text{stmt}_\text{trans} \) that assigns the RHS to the highest byte of \( ba<i_ba> \) that corresponds to \( v_{bf} \) using \( \text{exp}_\text{changedBits} \) and \( \text{exp}_\text{unchangedBits} \) at Line 31. After Line 31 is executed, \( \text{stmt}_\text{trans} \) becomes the below code:

\[
\begin{align*}
\text{S_nb.ba0}[2] &= (((e)<<(5)>>2)8) & 127) \\
&\ |(S_nb.ba0[2]\&128) \\
\text{S_nb.ba0}[1] &= (((e)<<(5)>>1)8) & 255 \\
\end{align*}
\]

3. The algorithm returns \( \text{stmt}_\text{results} \), which is equivalent to \( \text{stmt}_\text{assign} \) (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-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 subsection) 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 $i$th 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 the duration of time slot $t_{i+1}$. 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 engine 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 abilities 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 ($M^{SFF}$) for RQ6: It is the same as MAESTRO but the generated test driver by $M^{SFF}$ 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 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 runs $M^{TPF}$ 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 branch 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 task.

- 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.
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% (=2474+3116)/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 with 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).

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. 3.3.3).

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 module (238K LoC) requires 34 man-months (MM) (=238K LoC/25KLoC 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 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 MFO and MAESTRO. Table 3 shows that MFO achieves 3.2% (=97.2−94.2/94.2) and 8.4% higher branch and MC/DC coverage than MAESTRO, respectively. This is because MFO 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 2: The number of generated tests, execution time, and branch and MC/DC coverage of the target modules achieved by MAESTRO

<table>
<thead>
<tr>
<th>Targets</th>
<th>#tests</th>
<th>Time (hour)</th>
<th>Branch cov. (%)</th>
<th>#funcs achieving given branch cov. range</th>
<th>MC/DC cov. (%)</th>
<th>#funcs achieving given MC/DC cov. range</th>
</tr>
</thead>
<tbody>
<tr>
<td>APRK</td>
<td>119347</td>
<td>3.8</td>
<td>91.2</td>
<td>0 [10%, 20%, 40%, 60%, 80%, 100%]</td>
<td>77.9</td>
<td>0 [10%, 20%, 40%, 60%, 80%, 100%]</td>
</tr>
<tr>
<td>BCM</td>
<td>2474</td>
<td>20%</td>
<td>95.1</td>
<td>0 [10%, 20%, 40%, 60%, 80%, 100%]</td>
<td>81.2</td>
<td>0 [10%, 20%, 40%, 60%, 80%, 100%]</td>
</tr>
<tr>
<td>SMK</td>
<td>625</td>
<td>24%</td>
<td>94.5</td>
<td>0 [10%, 20%, 40%, 60%, 80%, 100%]</td>
<td>85.5</td>
<td>0 [10%, 20%, 40%, 60%, 80%, 100%]</td>
</tr>
<tr>
<td>TPM</td>
<td>44</td>
<td>41%</td>
<td>95.3</td>
<td>0 [10%, 20%, 40%, 60%, 80%, 100%]</td>
<td>88.8</td>
<td>0 [10%, 20%, 40%, 60%, 80%, 100%]</td>
</tr>
<tr>
<td>Total</td>
<td>1945720</td>
<td>27.2</td>
<td>94.2</td>
<td>0 [10%, 20%, 40%, 60%, 80%, 100%]</td>
<td>92.1</td>
<td>0 [10%, 20%, 40%, 60%, 80%, 100%]</td>
</tr>
</tbody>
</table>
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 violates 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. 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 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

---

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

<table>
<thead>
<tr>
<th>Module</th>
<th>Branch coverage (%)</th>
<th>MC/DC coverage (%)</th>
<th># crash alarms</th>
</tr>
</thead>
<tbody>
<tr>
<td>APRK</td>
<td>94.6</td>
<td>88.6</td>
<td>779</td>
</tr>
<tr>
<td>BCM</td>
<td>97.1</td>
<td>95.1</td>
<td>812</td>
</tr>
<tr>
<td>SMK</td>
<td>97.8</td>
<td>94.5</td>
<td>835</td>
</tr>
<tr>
<td>TPMS</td>
<td>97.2</td>
<td>95.3</td>
<td>828</td>
</tr>
<tr>
<td>Total</td>
<td>97.2</td>
<td>94.2</td>
<td>62001 (130)</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Module</th>
<th>Branch coverage (%)</th>
<th>MC/DC coverage (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>APRK</td>
<td>90.6</td>
<td>84.9</td>
</tr>
<tr>
<td>BCM</td>
<td>94.2</td>
<td>93.3</td>
</tr>
<tr>
<td>SMK</td>
<td>90.6</td>
<td>91.2</td>
</tr>
<tr>
<td>TPMS</td>
<td>94.7</td>
<td>93.2</td>
</tr>
<tr>
<td>Total</td>
<td>93.2</td>
<td>92.8</td>
</tr>
</tbody>
</table>

---

Footnote: 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.
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 MAE-
STRO’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 de-
developed by Mobis (i.e., APRK, BCM, SMK, and TPMS).
Since those target modules are real-world automotive pro-
grams 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 genera-
tion 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 prac-
tice. Although it is not trivial to develop an automated
test generation framework that resolves various technical
challenges in industrial projects, we believe that the au-
208

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

<table>
<thead>
<tr>
<th>Module</th>
<th>$M^{CT}$</th>
<th>$M^{TPF}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>APRK</td>
<td>53.7 62.9 78.6 89.6</td>
<td>58.1 61.8 71.5 79.5</td>
</tr>
<tr>
<td>BCM</td>
<td>61.1 76.4 87.4 94.2</td>
<td>69.1 75.6 78.4 88.7</td>
</tr>
<tr>
<td>SMK</td>
<td>61.6 77.9 86.5 93.6</td>
<td>67.6 76.8 84.0 90.6</td>
</tr>
<tr>
<td>TPMS</td>
<td>64.0 84.2 89.6 94.7</td>
<td>73.2 80.6 85.0 90.8</td>
</tr>
<tr>
<td>Total</td>
<td>60.6 74.4 86.4 93.2</td>
<td>66.7 75.2 81.2 88.6</td>
</tr>
</tbody>
</table>

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 in-
crease the coverage because increasing the bound value en-
larges 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 en-
vironments for a target task.

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

Concolic testing and fuzzing have different characteris-
tics for coverage testing. Table 5 shows the branch cover-
age 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 execu-
tion 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 con-
colic 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 cover-
age saturation point. As a result, the adaptive hybrid in-
put generator of MAESTRO achieved higher test coverage
than any single compared input generation technique.
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 $P'$ converted by MAESTRO for high test coverage (i.e., 94.2% vs 84.7% for branch coverage and 82.3% vs. 74.8% for MC/DC coverage).

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)

<table>
<thead>
<tr>
<th>Targets</th>
<th>MAESTRO</th>
<th>Angora on P</th>
<th>Angora on P'</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Branch</td>
<td>MC/DC</td>
<td>Branch</td>
</tr>
<tr>
<td>APRK</td>
<td>91.2</td>
<td>77.9</td>
<td>78.0</td>
</tr>
<tr>
<td>BCM</td>
<td>95.1</td>
<td>81.2</td>
<td>82.4</td>
</tr>
<tr>
<td>SMK</td>
<td>94.5</td>
<td>83.5</td>
<td>81.9</td>
</tr>
<tr>
<td>TPMs</td>
<td>95.3</td>
<td>82.8</td>
<td>80.2</td>
</tr>
<tr>
<td>Total</td>
<td>94.2</td>
<td>82.3</td>
<td>81.1</td>
</tr>
</tbody>
</table>

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 [30] 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 assertion 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.

CONBRI [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. CONBRIIO achieves high bug detection power with low false alarm ratio. However, CONBRIIO 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 Micro Formal [60] for Intel CPU’s microcode. Fujitsu developed KLOSER [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 bi-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 Acheong 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-2019R1A2B5B01008665), and Basic Science Research Program through NRF funded by the Ministry of Education (NRF-2017R1D1A1B03035851).

References