electronics

MDPI

Article

ASFIT: AUTOSAR-Based Software Fault Injection Test for Vehicles

Jihyun Park and Byoungju Choi *

Department of Computer Science and Engineering, Ewha Womans University, Seoul 03760, Korea; pola0527@ewhain.net

* Correspondence: bjchoi@ewha.ac.kr; Tel.: +82-2-3277-2593

Received: 29 April 2020; Accepted: 18 May 2020; Published: 20 May 2020

Abstract: With recent increases in the amount of software installed in vehicles, the probability of automotive software faults that lead to accidents has also increased. Because automotive software faults can lead to serious accidents or even mortalities, vehicle software design and testing must consider safety a top priority. ISO 26262 recommends fault injection testing as a measure to verify the functional safety of vehicles. However, the standard does not clearly specify when and where faults should be injected, and the tools to support fault injection testing for automotive software are also insufficient. In the present study, we define faults that may occur in Automotive Open System Architecture (AUTOSAR)-based automotive software and propose a fault injection method to be applied during the software development process. The proposed method can inject different types of faults that may occur in AUTOSAR-based automotive software, such as access, asymmetric, and timing errors, while minimizing performance degradation due to fault injection, and without using any separate hardware devices. The superior performance of the proposed method is demonstrated through empirical studies applied to fault injection testing of a range of vehicle electronic control unit software.

Keywords: software fault injection test; fault injection automation; AUTOSAR

1. Introduction

Recently, automobiles have been embedded with many electronic control systems, which are connected to and interact with a network to exchange data. As the amount of software in the electronic control unit (ECU) is rising, the frequency of software faults is also increasing [1]. Because automotive software faults can lead to serious accidents or even mortalities, vehicle software design and testing must consider safety the top priority. ISO 26262, the international standard for the functional safety of road vehicles, recommends fault injection testing as a measure to verify functional safety [2]. However, the standard does not clearly specify which faults should be injected, or when and where.

In the past, fault injection testing has been used to verify the fault tolerance of hardware or software [3]. It attempted to verify fault tolerance by checking fault detection, fault isolation, reconfiguration, and recovery after injecting faults. However, past fault injection tests have mainly focused on hardware faults. Even software fault injection tests, in most cases, only imitate hardware faults, such as memory, CPU, and communication faults [4]. However, as the amount of software installed in vehicles and the importance of safety continue to increase, a practical software fault injection test method has become necessary to verify the functional safety of software during the development process of automotive electronics-embedded software (hereinafter referred to as "automotive software").

In this paper, we propose a fault injection method to be applied during the ECU software development process, based on Automotive Open System Architecture (AUTOSAR) [5]. AUTOSAR is a standard platform for automotive software, created to improve automotive software development productivity. AUTOSAR-based software is generally divided into four layers; calls often occur between adjacent layers, whereas direct calls are rarely generated between non-adjacent layers. We define types of software faults that can occur in the call relationship within a software component (hereinafter referred to as "SWC") of AUTOSAR as well as between different layers. We propose a method to inject software faults in the basic software (hereinafter referred to as "BSW") layer to enable the injection of all software faults that we define. The main contribution of our method is that it enables the injection of faults that can occur in automotive software without using any separate hardware devices while minimizing performance degradation due to fault injection. We demonstrate the high quality of the proposed method by applying it to a range of actual automotive software.

The paper is organized as follows. Section 2 discusses studies related to fault injection testing. Section 3 defines AUTOSAR-based automotive software faults. Section 4 proposes a method for software fault injection testing. Sections 5 and 6 analyze the effects of the proposed method. Finally, Section 7 presents conclusions and considers future research directions.

2. Background and Related Work

This section introduces the existing research on fault injection testing and software fault types applicable to AUTOSAR-based automotive software.

2.1. Fault Injection in the Development Process

ISO 26,262 suggests the V-model for automotive system development, as shown in Figure 1. It recommends using fault injection testing in the entire development process, including the system development (Hardware(HW)/Software(SW) integration test, system integration test, and vehicle integration test), the hardware development (HW integration test), and the software development (SW unit test and SW integration test). In this study, we focus on the fault injection test in SW unit/integration tests during the software development process.

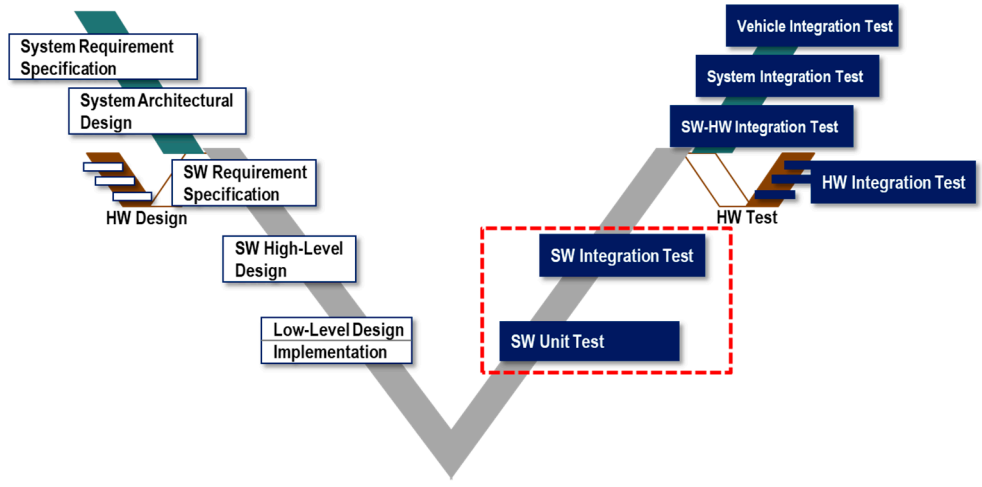


Figure 1. ISO 26, 262 V-model.

2.2. AUTOSAR-Based Automotive Software Fault Injection Method

The fault injection methods that can be used during the software development steps are largely divided into fault injection implemented by hardware and fault injection implemented by software (software-implemented fault injection, SWIFI).

The hardware-implemented fault injection methods use hardware characteristics to generate faults [6]. Principal hardware fault injection tools (FITs) include GOOFI-2 [7] and RIFLE [8]. Fault injection is also possible through a hardware probe such as TRACE32. However, these tools are expensive because they inject faults using separate hardware devices and are more appropriate for hardware or system development steps than for software development.

SWIFI is a cost-effective method of running software codes injected with faults, because unlike hardware-implemented fault injection, it does not require a separate device. SWIFI methods in AUTOSAR-based automotive software can largely be classified into three types: directly modifying the software source code, using AUTOSAR's hook function, and using a wrapper for the fault injection target function.

Directly modifying the source code entails directly modifying the execution of the binary code such that faults will be triggered when the corresponding code is executed [9]. This method accurately injects faults at the position desired by the tester, but the execution flow may become different than that of the original code because the source code has been changed. As automotive software is sensitive to execution timing, the code may operate differently from the original intention when the timing is changed by the modified code.

The hook function of AUTOSAR [10] acts by inserting tracking codes such as pre-task, post-task, and error hooks in the functions defined in the AUTOSAR standard for the purposes of execution monitoring or fault injection. This method works well if the functions defined in the standard are used, but applying it in varied contexts is challenging because the hook function can vary depending on the degree of implementation of the OS in accordance with the AUTOSAR standard.

Using a wrapper in the fault injection target function [11,12] creates wrapper components in line with various AUTOSAR hierarchical structures and injects faults while monitoring the execution of the original components. Most SWIFIs inject faults using a wrapper [13]. This method can inject various faults, such as in the parameters, return values, and function calls of the fault injection target function, but it is difficult to implement because it only uses software without the help of a hardware debugger.

2.3. Fault Types in Automotive Software

Before performing a fault injection test, the faults that should be injected must be defined. AUTOSAR defines five error types that can occur in automotive software: data, program flow, access, timing, and asymmetric errors [6]. A data error is caused by invalid values of messages exchanged between function parameters, variables, and/or components. For instance, the sudden unintended acceleration of Toyota vehicles, which was caused by a change in the data stored in the shared memory to an invalid value, corresponds to a data error [14]. A program flow error involves the execution of a program differently than what is expected due to omitted, invalid, or unnecessary operations. An access error occurs when software attempts to accesses resources for which it does not have permission. A timing error involves early or late delivery or omission of messages between components. An asymmetric error is a fault that delivers conflicting messages when multiple software components simultaneously receive a message. Tools that support the injection of these automotive software faults include Kayotee [9], CaNoe [10], G-SWFIT [11], and GRINDER [12].

Kayotee is a FIT that directly modifies source codes and can inject a data error fault that changes the value of a variable. However, other types of fault injection are not supported. A CaNoe-based FIT uses the hook function provided by AUTOSAR and can inject data errors by intercepting and changing the signals or messages in communication with external ECUs. However, the fault types that can be injected are limited because the FIT cannot directly change the parameter or return value of the function that is currently tracking for fault injection.

G-SWFIT and GRINDER are tools that inject faults using a wrapper. Unlike CaNoe, which uses the AUTOSAR hook function, a wrapper can be used to inject faults in the parameter or return value of the fault injection target function or the function call itself. Depending on how the wrapper is implemented, wrapper-based methods can perform fault injection for all five types of errors that can occur in automotive software. G-SWFIT and GRINDER can perform fault injection for some types of data and program flow errors. They can inject invalid values in the variables used in software or inject faults in the related parameter or function itself when a function call occurs.

As described above, most automotive software FITs focus on data errors. The method we propose can perform fault injection not just for data errors but for all five error types that can occur in AUTOSAR-based applications.

3. AUTOSAR-Based Automotive Software Faults

In this section, we describe where and what type of faults can be injected in automotive software. The fault injection position is derived from the SWCs of AUTOSAR-based automotive software and the calling relationship between SWCs and other layers, while the error type that can occur at each position in the AUTOSAR software is reflected in the final definition of AUTOSAR-based automotive software faults.

3.1. Calling Relationships between AUTOSAR Software Layers

To inject faults into AUTOSAR-based automotive software, we first need to understand the AUTOSAR platform. AUTOSAR is defined by the standard architecture for productivity improvement in automotive electronics-embedded software development, as shown in Figure 2.

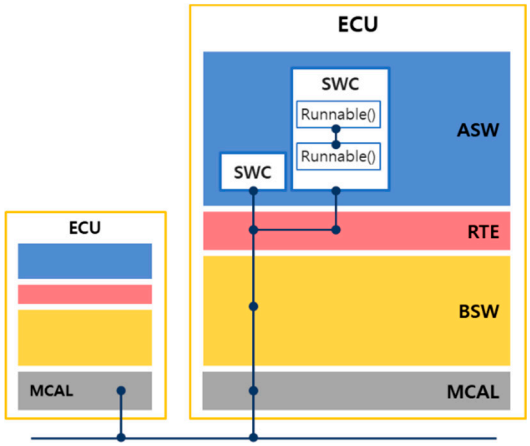


Figure 2. AUTOSAR software architecture.

The software architecture defined in AUTOSAR is essentially composed of application software (ASW), a runtime environment (RTE), BSW, and a microcontroller abstraction layer (MCAL). The RTE layer of the virtual functional bus concept exists for development productivity improvement, and various SWCs are installed on top of this layer, in the ASW layer. Runnables are functions used to implement functions inside SWCs and to call each other. However, SWCs are implemented using different methods. For this reason, direct calls between different SWCs do not occur and SWCs instead communicate with other SWCs through the RTE layer. The OS and the software that provides services such as memory communication are installed in the BSW layer. Drivers related to the hardware device control are installed in the MCAL layer. In general, the calls between the layers of AUTOSAR occur between adjacent layers, but in order to maximize software performance, an SWC may call the BSW directly without passing through the RTE layer.

The calling relationships that can occur in the AUTOSAR-based automotive software can be classified as follows:

- SW–SW calling relationships:
    - Call between runnables within an SWC;
    - Call between SWC and RTE;
    - Call between SWC and BSW;
    - Call between RTE and BSW.
- SW–HW calling relationships:
    - Call between BSW and MCAL.
- HW–HW calling relationships:
    - Call between drivers within MCALs.

3.2. Fault Injection Position

Fault injection positions must be identified in accordance with the unit and integration test steps for the automotive software. The ISO 26,262 standard specifies that in the unit test, random faults such as damaging the values of variables, changing the code, or damaging the value of the CPU register should be injected in the software unit. The standard also specifies that in the integration test, the safety mechanism should be verified by damaging the software or hardware components [2]. Accordingly, we derived the positions for injecting faults and finally selected the "position where fault injection using a wrapper is possible" as the fault injection position, which is our proposed method.

The positions at which fault injection for software unit testing are possible are statements inside software units, such as a keyword, brace, or sequence; unary/binary operators; variables, such as a scalar variable, global variable, array, pointer, or structure; and constants [15]. However, fault injection in all units except the global variable is possible only via a direct code modification. Therefore, in our proposed fault injection test using a wrapper, the unit test fault injection position is limited to global variables.

In the software integration test, the position of integration between software components must be the target of fault injection. As components can be divided by layer in AUTOSAR, the calling relationship between layers presented in Section 3.1 becomes the fault injection position. A calling process between layers involves a caller and callee. The fault injection positions were derived by applying the interface mutation operator to each of the two functions [16,17].

The fault injection positions according to the unit and integration test steps of the automotive software are as follows:

- SW unit test step:
    - Global variables used in the SWC runnable.
- SW integration test step:
    - Calling function (caller) in the calling relationship between layers: function call parameter and call statement itself;
    - Called function (callee): parameter received from the caller, variables used inside callee, and return statement.

3.3. Fault Types

As mentioned in Section 2.3, there are five error types that can occur in AUTOSAR-based automotive software: data, program flow, access, timing, and asymmetric errors [18]. Accordingly, we analyzed the causes of errors, derived the fault types, and listed them in Table 1.

3.4. AUTOSAR-Based Automotive Software Faults

We redefined automotive software faults as based on the AUTOSAR interface by applying the fault types presented in Section 3.3 to each of the fault injection positions derived in Section 3.2. The data error type is applied to the global variable, function parameter, and return value. Program flow and timing errors are applied to the calling statement of the caller function and the return statement of the callee function. Access errors are only applied to the global variables shared in the calling relationships. Asymmetric errors are applied to return values when the design considered redundancy for fault tolerance. The details are outlined in Table 2.

Table 1. Fault types.

AUTOSAR Application Error Type		**SWFIT Faults**	
		Fault Type	**Description**
Data error	Fault caused by invalid values of a parameter, variable, or message	Invalid value	Data error
Program flow error	Program running differently from what is expected due to omitted/invalid/unnecessary operations	Uncalled function	Uncalled function
		Bypass function call	Bypass function call
		Illegal instruction	Running an invalid command
Access error	Fault generated when an SW component attempts to accesses resources for which it has no permission	Invalid address	Accessing an invalid address
		Invalid register	Accessing an invalid register
Timing error	Fault caused by early or late delivery or omission of a message	Data delay	Delay of data transmission
		Data loss	Data loss during transmission
		No response	No response
		CPU clock corruption	CPU clock error
Asymmetric error	Fault generated by delivery of different values when multiple components receive messages simultaneously	Asymmetric value	Receiving asymmetric data

Table 2. Automotive software faults based on the AUTOSAR interface.

Test Step		Fault Injection Position			Software Fault		Fault ID
SW unit test		Runnable	Variables	Global variable	Data error	Invalid value	1
SW integration test	SW–SW integration test	Runnable-Runnable	Caller	Parameter	Data error	Invalid value	2
				Call statement itself	Program flow error	Uncalled function	3
						Bypass function call	4
						Illegal instruction	5
			Callee	Global variable	Data error	Invalid value	6
					Access error	Invalid address	7
				Return statement	Data error	Invalid value	8
					Program flow error	Illegal instruction	9
					Timing error	Data error	10
						CPU clock corruption	11
		SWC–RTE	Caller	Parameter	Data error	Invalid value	12
					Timing error	Data delay	13
						Data loss	14
				Call statement itself	Program flow error	Uncalled function	15
						Bypass function call	16
						Illegal instruction	17
					Timing error	CPU clock corruption	18
			Callee	Global variable	Data error	Invalid value	19
					Access error	Invalid address	20
				Return statement	Data error	Invalid value	21
					Program flow error	Illegal instruction	22
					Timing error	Data loss	23
						No response	24
						CPU clock corruption	25
					Asymmetric error	Asymmetric value	26
		SWC–BSW	Caller	Parameter	Data error	Invalid value	27
					Timing error	Data delay	28
				Call statement itself	Program flow error	Uncalled function	29
						Bypass function call	30
						Illegal instruction	31
					Timing error	CPU clock corruption	32
			Callee	Global variable	Data error	Invalid value	33
					Access error	Invalid address	34
				Return statement	Data error	Invalid value	35
					Program flow error	Illegal instruction	36
					Timing error	Data loss	37
						No response	38
						CPU clock corruption	39
					Asymmetric error	Asymmetric value	40
	SW–HW integration test	SWC–ECUHW	Caller	Parameter	Data error	Invalid value	41
					Timing error	Data delay	42
				Call statement itself	Program flow error	Uncalled function	43
						Bypass function call	44
						Illegal instruction	45
					Timing error	CPU clock corruption	46
			Callee	Global variable	Data error	Invalid value	47
					Access error	Invalid address	48
				Return statement	Data error	Invalid value	49
					Program flow error	Illegal instruction	50
					Timing error	Data loss	51
						No response	52
						CPU clock corruption	53
					Asymmetric error	Asymmetric value	54

4. ASFIT: AUTOSAR-Based Automotive Software Fault Injection Method

To inject the faults defined in Section 3 in the automotive software, a fault is generated through an analysis of the binary code execution of the automotive software, as shown in Figure 3; fault injection is performed by integrating the fault into the automotive software. Fault generation is divided into a module for extracting the position where fault injection is possible and a module for generating the "fault injection code" into which the fault has been injected. Integrating the fault into the automotive

software is divided into a module for monitoring the fault injection target and a fault injection module that actually runs the fault.

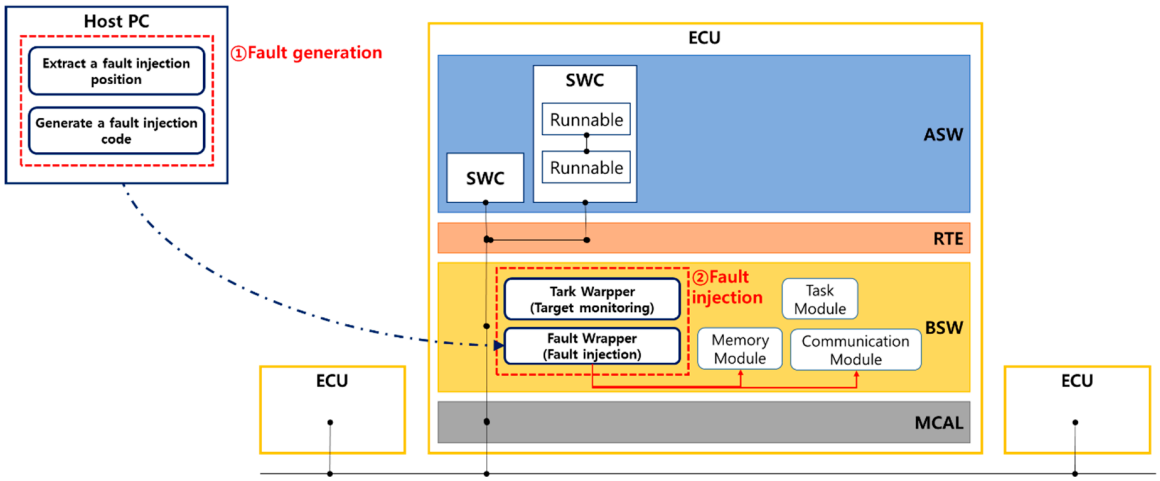


Figure 3. ASFIT: AUTOSAR-based automotive software fault injection method.

4.1. Extraction of a Fault Injection Position

Table 2 defines the fault injection positions for each test step. Fault injection positions are derived through static analysis of the executable binary file of the fault injection test target software. The most important factor in deriving fault injection positions is to extract runnables and tasks. A runnable is a unit for running the commands inside an SWC together with a C function. A task is a set of runnables; one runnable can be included in multiple tasks [18]. Because runnables form the core of the SWC operation, we use the runnable as the minimum unit of fault injection and also as the unit for monitoring tasks that call and manage runnables for execution of fault injection. On the basis of the tasks and runnables, we extract the global variables used in the runnables, other runnables called by runnables, and/or the functions of the RTE/BSW layer.

Fault injection positions extracted from binary automotive software:

- Runnables inside SWC;
- Tasks allocated to each runnable;
- Global variables used in the runnables;
- Runnables in the calling relationships (runnable–runnable, runnable (SWC)–RTE, runnable (SWC)–BSW) or prototypes of functions (parameter, return value).

4.2. Generating the Fault Injection Code

A fault injection code is a wrapper function that injects a fault when the correct time for fault injection arrives, while monitoring whether to inject the fault in a function identified as a fault injection position. The wrapper function is composed of a task wrapper "T_wrapper" for monitoring and an "F_wrapper," a fault wrapper that actually injects the fault. The T_wrapper monitors whether the currently running task is included among the fault injection positions. If so, then the F_wrapper is called to inject the fault and the original task is continued.

Figure 4 shows an example of injecting an uncalled function fault (fault type 15 in Table 2) when a function is called between the SWC–RTE layers in the task FuncOSTask_ASW_FW1_100ms. Figure 4a shows a T_wrapper, which checks whether the fault injection is enabled and then checks whether the function called by the current task is the fault injection target function. If so, then the F_wrapper, as shown in Figure 4b, is called as described above.

```
extern void FuncOsTask_ASW_FG1_100ms(void);

void __wrap_FuncOSTask_ASW_FG1_100ms(void) {
//Check whether the fault injection is enabled
If(fEnable [0] == 0) {
__real_FuncOsTask_ASW_FG1_100ms();
return 0;
}

//Check whether the function called by the task is the fault
injection target
If(!isTargetFunc(callerFunc, size)) {
      __real_FuncOsTask_ASW_FG1_100ms();
      return 0;
}

   //Call the fault injection wrapper
AE_invalid_addr(&targetAddress);

return __real_FuncOsTask_ASW_FG1_100ms();
}
```

(**a**) T_wrapper

```
//fault injection wrapper
//Write in assembly code
void AE_invalid_addr(void* addr) {
#pragma asm
//Set the address of addr as an OS protected area
e_lis r3, <value1>
  e_addl6i r3, r3, <value2>
//return
se_blr
#pragma endasm
}
```

(**b**) F_wrapper

Figure 4. Example of fault injection code.

4.3. T_Wrapper

In Section 2.2, we compared directly modifying the code, using an AUTOSAR hook function, and using a wrapper as methods for injecting faults in AUTOSAR-based automotive software. Among these, we adopt the wrapper-based fault injection method. Using the wrapper function has a significant advantage: it generates the various faults defined in Table 2 by calling the wrapper function rather than the original function to inject faults.

When injecting faults using a wrapper, caution should be exercised regarding how much effect the fault injection has on the original execution due to the wrapper. As AUTOSAR-based automotive software is a hard, real-time embedded system with a large time constraint, the fault injection must be performed with a minimum time overhead.

To minimize the overhead due to fault injection, we monitor the task that includes the position where the fault is injected instead of monitoring the position directly. If the monitored task includes a fault injection position, then the fault is injected by calling the wrapper. In other words, we can reduce the overhead by checking whether the position is a fault injection target by monitoring only the task that consists of multiple runnables, instead of monitoring all the calls between runnables inside the SWC, the runnables of the SWC, and all function calls of the RTE/BSW/MCAL layers that may constitute the fault injection position.

In Figure 5, the fault injection position is the call between SWC_runnable_1() and SWC_runnable_2(). To monitor all positions selected for fault injection for Task_1, we would need to monitor all the runnable calls of Task_1 and all the calling relationships inside SWC_runnable_1() and SWC_runnable_3(). In other words, at least six calls must be monitored. However, this method has a large overhead because whether a position is a fault injection target must be checked whenever a call occurs.

The method we propose only monitors the task itself and performs monitoring for Task_1 only once. When Task_1 is run, whether the fault injection position is included in the runnable called by Task_1 is checked, and in this case the fault is injected immediately. This method saves overhead and enables lightweight fault injection by not checking every time a call between layers occurs.

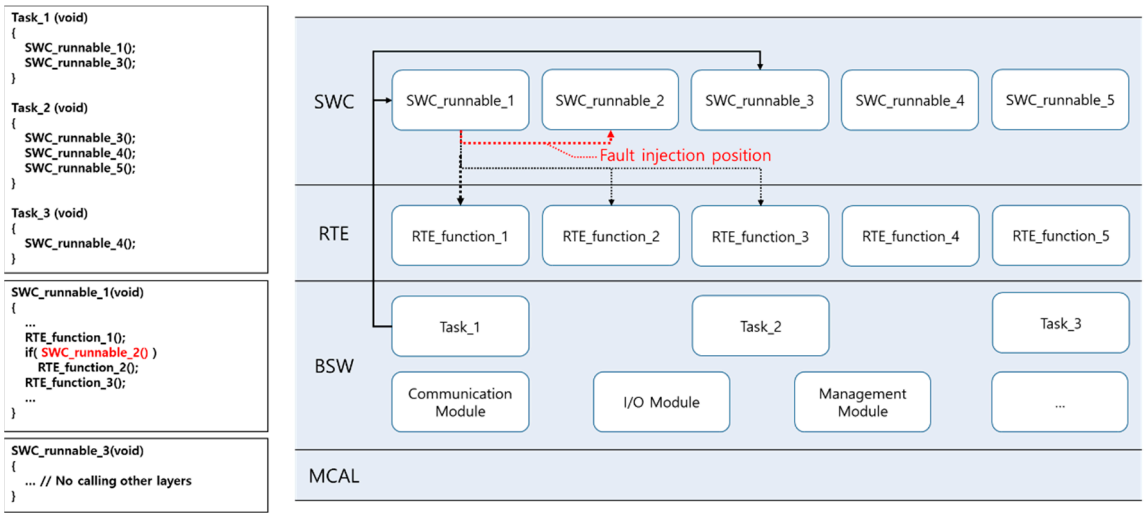


Figure 5. Example of task monitoring and fault injection.

4.4. F_Wrapper

Our fault injection method uses a wrapper for the fault injection target function, but it does not generate a wrapper for the fault injection position. As described in Section 4.3, our method "injects a software fault using a wrapper for the functions of the BSW layer" when the task that includes the fault injection target is run.

In fact, the wrapper-based fault injection method can vary depending on which layer's function is used as the wrapper function, as shown in Table 3. In addition, the calling relationship into which a fault can be injected as well as the fault type can vary depending on the layer using the wrapper function.

Only data and program flow errors can be injected when the wrapper function is generated in the SWC layer and a fault is injected, because the SWC cannot call the BSW layer functions, which are related to communication or memory management. However, if the wrapper function is generated in the BSW layer, then all five fault types in all layers of AUTOSAR from SWC to ECUHW layers can be injected because BSW-layer functions perform execution control (task module), memory management (memory module), and communication control (communication module), as shown in Figure 2.

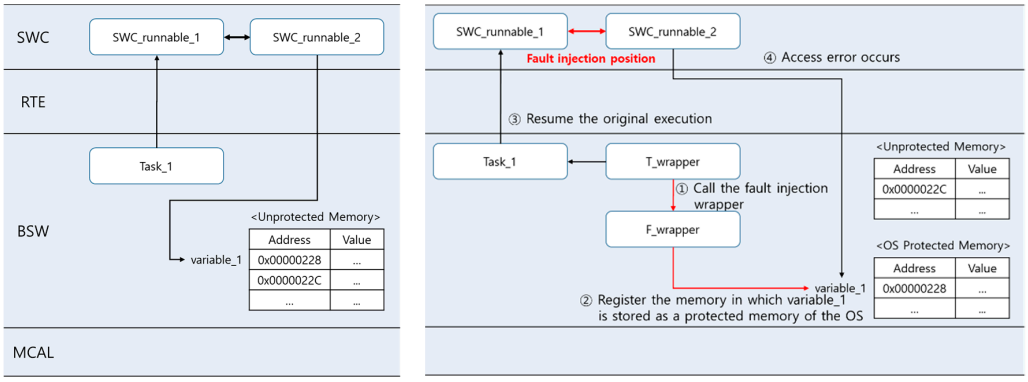
The SW faults to be injected are related to the fault injection position, which may be a call between runnables inside the SWC or a call between the SWC and the functions of the RTE/BSW/MCAL layers, as listed in Table 3. We create a wrapper for each function corresponding to these fault injection positions and inject faults by "implementing wrappers only for the functions of the BSW layer" rather than performing fault monitoring and fault injection together. Generating the fault injection wrapper only for the functions of the BSW layer can achieve the same effect as injecting faults directly to the SWC or RTE layers. As the wrapper function can minimize the added codes, the execution binary size can be also minimized.

The wrappers for fault injection are generated by the memory and communication modules of the BSW layer. Even if data are changed in the SWC or RTE, the data are stored and managed by the BSW-layer monitoring module. In this way, the corresponding wrappers are generated and the faults are injected by them. Furthermore, calls between layers or communication with another ECU are performed through the communication module of the BSW layer. Thus, even if a call is made in the SWC, it is performed internally through the BSW-layer communication module. Therefore, the core concept of our method is that fault injection for all layers is possible even if a fault injection wrapper is not generated in each layer.

Table 3. Fault injection method using a wrapper for each AUTOSAR layer.

Layer Using Wrapper	Description	Calling Relationships in Which Faults Can Be Injected	Fault Types That Can Be Injected
SWC	The SWC layer provides various services that run in automotive SW. The wrapper function injects faults in runnables or functions of the SWC layer.	• Calls inside SWC • Calls between SWC and RTE • Calls between SWC and BSW	• Data error • Program flow error
RTE	The RTE layer is an interface that connects the SWC and BSW layers. The wrapper function injects faults in functions of the RTE layer.	• Calls between SWC and RTE • Calls between RTE and BSW	• Data error • Program flow error
BSW (our method)	The BSW layer performs service execution control, memory management, and device management. The wrapper function injects faults in functions of the BSW layer.	• Calls of all layers from SWC to ECUHW layers	• Data error • Program flow error • Access error • Timing error • Asymmetric error

To illustrate further, in Figure 6 we use an "invalid address" for the global variable—the callee-side runnable in a runnable–runnable call corresponding to fault ID 7. Before the call is injected, when SWC_runnable_1() calls SWC_runnable_2() as shown in Figure 6a, the address of the global variable accessed from callee SWC_runnable_2 exists in the unprotected memory area and can be accessed from SWC_runnable_2. However, when the fault is injected by the proposed method, the T_wrapper of the BSW layer is run before Task_1(), as shown in Figure 6b, and the F_wrapper for fault injection is called (①). The F_wrapper registers the memory (②) in which the variable_1 accessed from SWC_runnable_2 is registered as a protected memory in the OS. Then, while the original Task_1 is performed (③), an access error occurs when SWC_runnable_2() accesses variable_1 because it occupies a memory area that is not allowed to access the variable.



(**a**) Before fault injection (**b**) Fault injection

Figure 6. Invalid address fault injection for variables in a runnable–runnable call.

It is worth noting that if faults are injected by generating a wrapper for SWC_runnable_2() (the fault injection position in this example) without using the F_wrapper in the BSW, then a fault of changing the value of variable_1 can be injected, but a fault that changes the address of the variable to a protected memory area of the OS to make it an inaccessible address cannot be injected.

5. Empirical Study

For the fault types that we defined, we compared the options of injecting faults in the automotive software using ASFIT versus using the existing fault injection method.

5.1. Experiment Design

The purpose of this experiment is to demonstrate the superior performance of our method by comparing how well the faults defined in Table 2 are supported by the existing fault injection methods and the proposed method. As shown in Table 4, fault injection methods using hardware [19–21] and using SWIFI [9–12], as mentioned in Section 2.2, were selected for comparison.

Table 4. Fault injection method for performing the experiment.

Fault Injection Method	
SW-based method	**ASFIT** Kayotee [9] CaNoe-based FIT [10] G-SWFIT [11] GRINDER [12]
HW-based method	TRACE32 [19] GreenHills Debug Probes (hereinafter, GHS Probe) [20] Code Warrior IDE (hereinafter, CW IDE) [21]

The software faults that can occur also differ according to the characteristics of the ECU. Therefore, various ECUs were used to inject all the automotive software fault types defined in Table 2. As shown in Figure 7, the ECUs used in this experiment were a wiper control system, which is a body control module for vehicle interior convenience facility control; an electronic steering column lock (ESCL) control system, which acts as a smart key control; and a vehicle control unit (VCU), which is a vehicle driving system based on a mobile open platform for experimental development of cyber-physical systems.

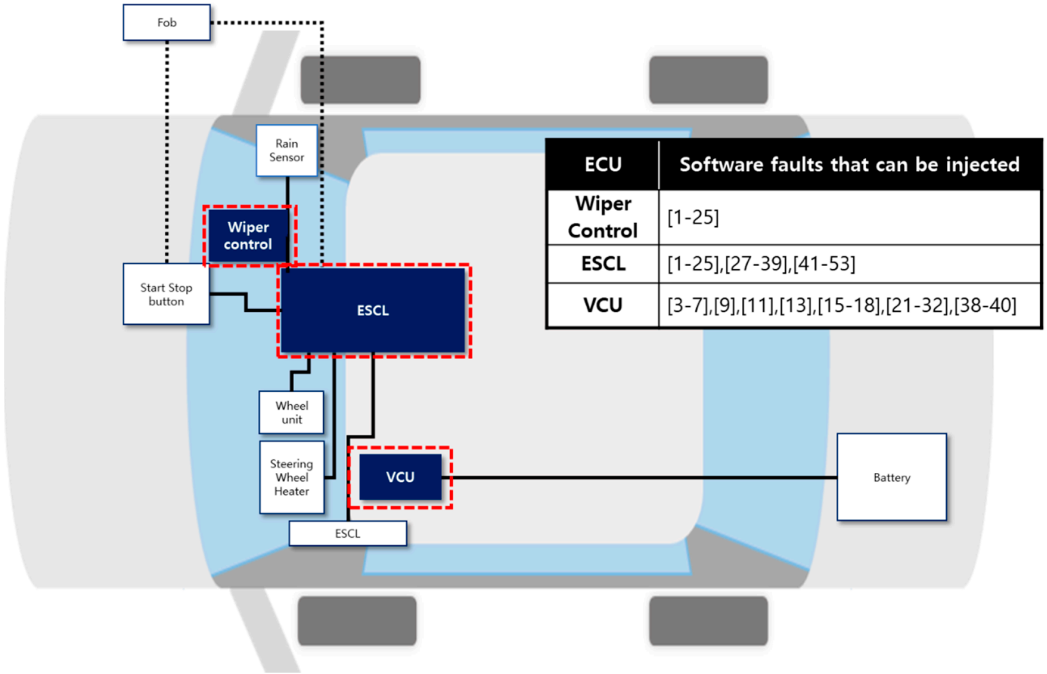


Figure 7. Target ECU for the experiment.

5.2. Experiment Results

The software faults that can be injected by each fault injection mechanism are listed in Table **??**. The proposed ASFIT can inject all 54 faults, but other methods can inject only 12 to 42 faults.

Table 5. Software faults that can be injected.

Fault Injecting Method		Software Fault ID																		#
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	
		19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	54
		36	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	
SW-based method	**ASFIT**	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	**54**
		o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	
		o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	
	Kayotee	o	x	x	x	x	o	x	x	x	x	x	o	x	x	x	x	x	x	5
		x	x	x	x	x	x	x	X	o	x	x	x	x	x	x	x	x	x	
		x	x	x	x	o	x	x	x	x	x	x	x	x	x	x	x	x	x	
	CaNoe	x	o	x	x	x	o	x	o	x	x	x	o	x	x	x	x	x	x	12
		o	x	o	x	x	x	x	x	o	x	x	x	x	x	o	x	o	x	
		x	x	x	x	o	x	x	x	x	x	o	x	o	x	x	x	x	x	
	G-SWFIT	x	o	x	x	x	o	x	o	x	x	x	o	x	x	x	x	x	x	12
		o	x	o	x	x	x	x	x	o	x	x	x	x	x	o	x	o	x	
		x	x	x	x	o	x	x	x	x	x	o	x	o	x	x	x	x	x	
	GRINDER	x	o	x	x	x	o	x	o	x	o	x	o	o	o	x	x	x	x	23
		o	x	o	x	o	o	x	x	o	o	x	x	x	x	o	x	o	x	
		o	o	x	x	o	o	x	x	x	x	o	x	o	x	o	o	x	x	
HW-based method	TRACE32	o	o	o	o	o	o	x	o	o	o	o	o	x	o	o	o	o	o	42
		o	x	o	o	o	x	o	x	O	x	o	o	o	o	o	x	o	o	
		o	x	o	x	o	x	o	o	o	o	o	x	o	o	o	o	o	x	
	GHS Probe	o	o	o	o	o	o	x	o	o	o	o	o	x	o	o	o	o	o	42
		o	x	o	o	o	x	o	x	o	x	o	o	o	o	o	x	o	o	
		o	x	o	x	o	x	o	o	o	o	o	x	o	o	o	o	o	x	
	Code Warrior	o	o	o	o	o	o	x	o	o	o	o	o	x	o	o	o	o	o	42
		o	x	o	o	o	x	o	x	o	x	o	o	o	o	o	x	o	o	
		o	x	o	x	o	x	o	o	o	o	o	x	o	o	o	o	o	x	

5.3. Analysis

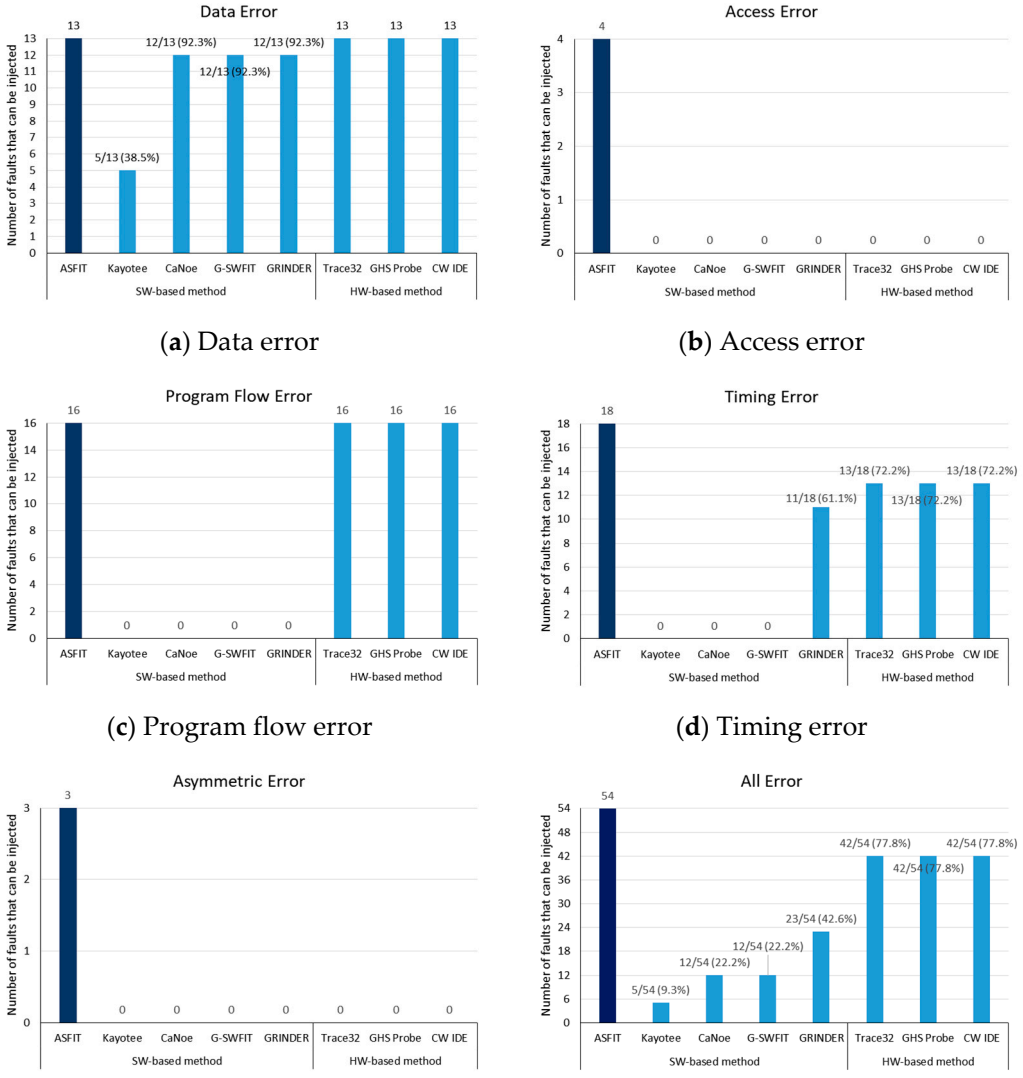
Figure 8f compares the numbers of faults that can be injected by ASFIT and other fault injection methods out of a total of 54 faults. ASFIT can inject all 54 faults, whereas the software-based FIT tools can inject 5–23 faults and the hardware-based FIT tools can inject 42 faults. Figure 8a–e shows the results for the 54 faults by error type.

The software-based FIT tools Kayotee, CaNoe, and G-SWFIT can inject 5, 12, and 12 faults, respectively, related to data error, as shown in Figure 8a. As shown in Table 4, CaNoe and G-SWFIT can only inject faults related to data errors that change the variables, change the parameters, or return values used in calling relationships between functions. Kayotee can only inject data error faults for variables used in the software. However, these three tools cannot inject software faults related to program flow, access, timing, or asymmetric errors.

According to ISO 26262, GRINDER, another software-based FIT tool, can inject bit flips, data type-based corruption, and timing faults for every layer of AUTOSAR [12]. However, bit flips and data type-based corruption faults can be injected only in the calling relationship functions; data errors for global variables cannot be injected. Moreover, CPU clock corruption, a timing error fault that can be injected by the CPU clock control corresponding to the MCAL layer, cannot be injected by GRINDER

either. In addition, fault injection for access, program flow, and asymmetric errors, excluding data or timing errors, was not supported.

TRACE32, GHS Probe, and CW IDE, which inject faults using a hardware debugger, can inject faults at the desired position by adding a breakpoint in the source code [19–21]. These tools can inject most faults but cannot inject some timing errors or access and asymmetric errors. In the case of timing errors, they can inject data loss and CPU clock corruption faults through data manipulation after stopping the execution through a breakpoint, but cannot inject faults such as data delay and no response that occur during execution.



(**a**) Data error (**b**) Access error

(**c**) Program flow error (**d**) Timing error

(**e**) Asymmetric error (**f**) All errors

Figure 8. Comparison of numbers of faults that can be injected by AUTOSAR error type.

The greatest advantage of ASFIT is that, unlike other methods, it can inject all faults related to access, timing, and asymmetric errors. As automotive software is a hard, real-time system composed of various ECUs, the management of shared memory is critical and delays should be prevented when interacting with these ECUs. Furthermore, a fault that occurs during execution of the software must be accurately communicated to other interacting ECUs. Therefore, functional safety must be verified through fault injection for related access, timing, and asymmetric errors. In the following, we analyze the proposed ASFIT more concretely with particular cases of these faults.

1 **Case 1 (ESCL access error):** fault ID 7—invalid address fault injection for global variables in the callee of the runnable–runnable integration test level.

The ESCL locks or unlocks the wheel using the vehicle's smart key. The fault that we injected checks consistency for the handle lock state of the vehicle between the ESCL hardware and the control system and then changes the address of a variable used to supply power to the ESCL to an invalid value. When this fault occurs, normal power supply to the ESCL is impossible because the variable value cannot be read, and the task that contains the runnable related to the ESCL control is rebooted for safety.

This fault is injected in the step when the runnables inside the SWC are integrated. The fault that we inject changes the address of the global variable b_ESCLPowerSupplied, which is used in ESCLPowerSupply(), corresponding to the callee when the runnable EsclConsistencyCheck() calls the runnable ESCLPowerSupply(), as shown in Figure 9.

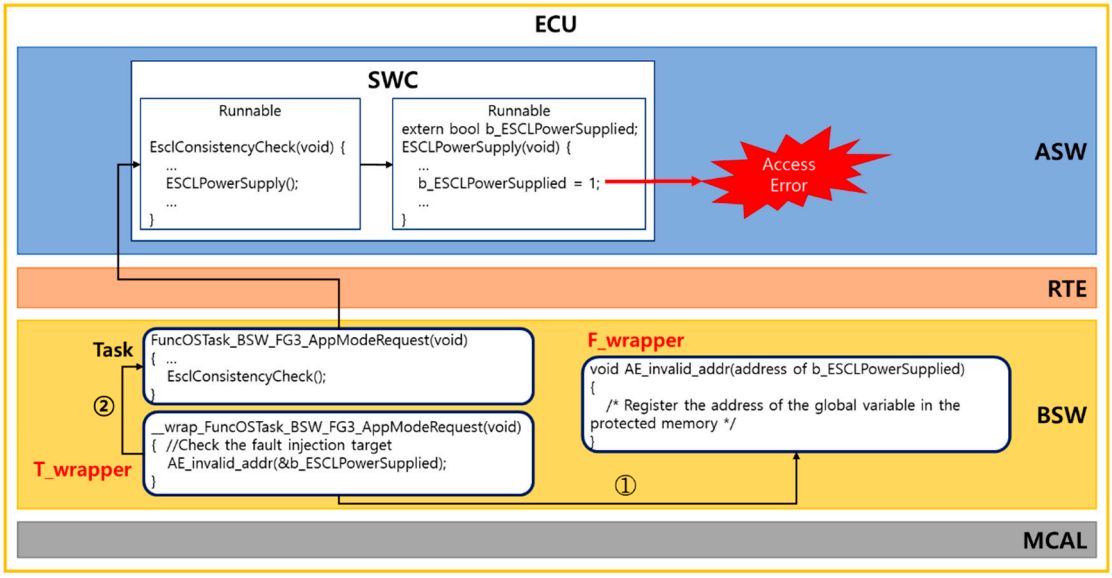


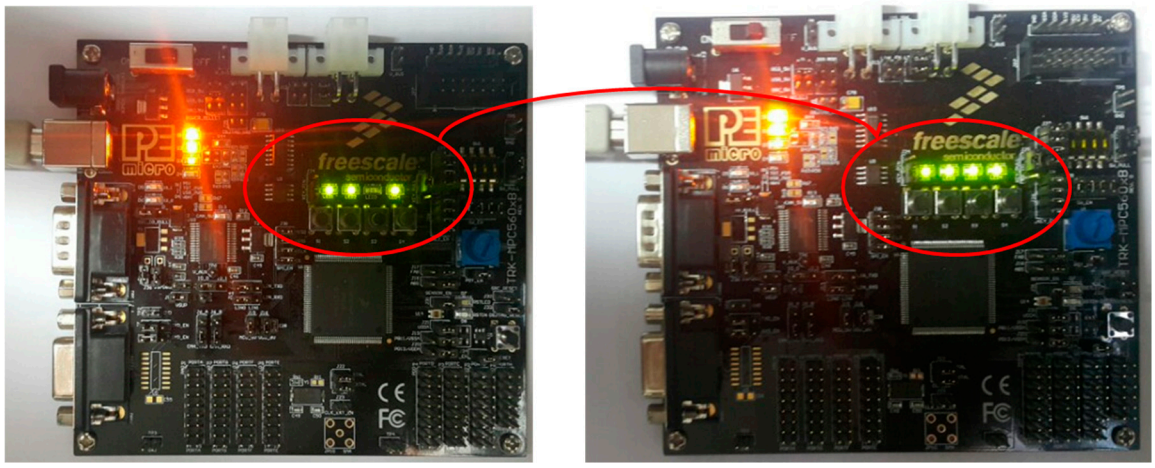
Figure 9. Case 1: access error fault injection using ASFIT.

The call between ESCLConsistencyCheck() and ESCLPowerSupply() occurs when the task FuncOSTask_BSW_FG3_AppModeRequest() is executed. Therefore, the T_wrapper of the corresponding task checks whether it is the fault injection target, then injects the fault by calling the F_wrapper (①), and finally calls the original task (②). As the access error is a fault that accesses an address for which it does not have access permission, the address in the global variable b_ESCLPowerSupplied is registered in the protected memory managed by the OS. Then, when ESCLPowerSupply() attempts to access the variable, an access error occurs because it does not have the access permission.

To check the results of fault injection using ASFIT, we operated LEDs when the fault injection code was executed in an environment wherein the automotive software operated. When a fault is not injected and the operation is normal, the initial state of the LED (all LEDs are on) is maintained, and when a fault is injected, a specific LED is turned off in the fault injection function. As with the initial state of the LED, all LEDs return to the on state when the target board is rebooted or when the LED control function is called within the safety mechanism.

Figure 10a shows that the third LED from the left is turned off to confirm the occurrence of an access error due to fault injection. After the fault was injected, the task was restarted, and all the LEDs turned on, as shown in Figure 10a. This finding shows that the safety mechanism that we defined works well.

The existing software-based fault injection method does not support changing the memory area protected by the OS, while in the hardware fault injection method, it is difficult to find the memory area protected by the OS without the total source code of the OS. In contrast, ASFIT can perform fault injection because it finds the protected memory area of the OS from the binary when the fault injection code is generated.



(**a**) Fault injection (**b**) Operation of the safety mechanism

Figure 10. Fault injection result.

2 **Case 2 (ESCL timing error): fault ID 13—**data delay fault injection for the parameter of the caller at the SWC–RTE integration test level.

The ESCL locks or unlocks the wheel using the vehicle's smart key. The fault that we injected delays the value of the data exchanged during communication to check consistency between the ESCL hardware and the control system.

This fault is injected in the step when the RTE function is called by a runnable inside the SWC, at the integration test level of the SWC and the RTE layer. This fault delays the transmission of the value of the parameter l_ESCLUnlock transmitted from the caller EsclControl() when the runnable EsclControl() calls Rte_Write_P_ConsistencyCheck_L_ESCLUnlock(). When this fault occurs, consistency between the ESCL hardware and the control system cannot be checked, and the safety mechanism that stops power to the ESCL until consistency can be confirmed is activated.

Figure 11 shows the concrete fault injection process in detail. The T_wrapper of a task that includes a SWC–RTE call, which is the fault injection position, injects a fault (①) by calling the F_wrapper, and then calls the original task (②). At this time, the F_wrapper TE_Delay() activates the fault by changing the address of the RTE function called by the SWC–RTE calling code to the address of Delay().

As in Case 1, to check the results of fault injection by ASFIT, we modified the fault injection code such that it would control the LEDs when the code was executed in an environment wherein the automotive software was operated. Furthermore, as in Case 1, the LEDs were turned off and the ESCL was rebooted after the fault injection. All the LEDs were then turned on, indicating that our proposed design for the safety mechanism works well.

This fault can be injected by software-based fault injection but not by hardware-based fault injection. As mentioned above, the hardware-based fault injection method stops execution in order to inject faults. However, unlike the change in the value of the allocated memory or register, when a delay occurs an execution code must be added, which requires software rebuilding.

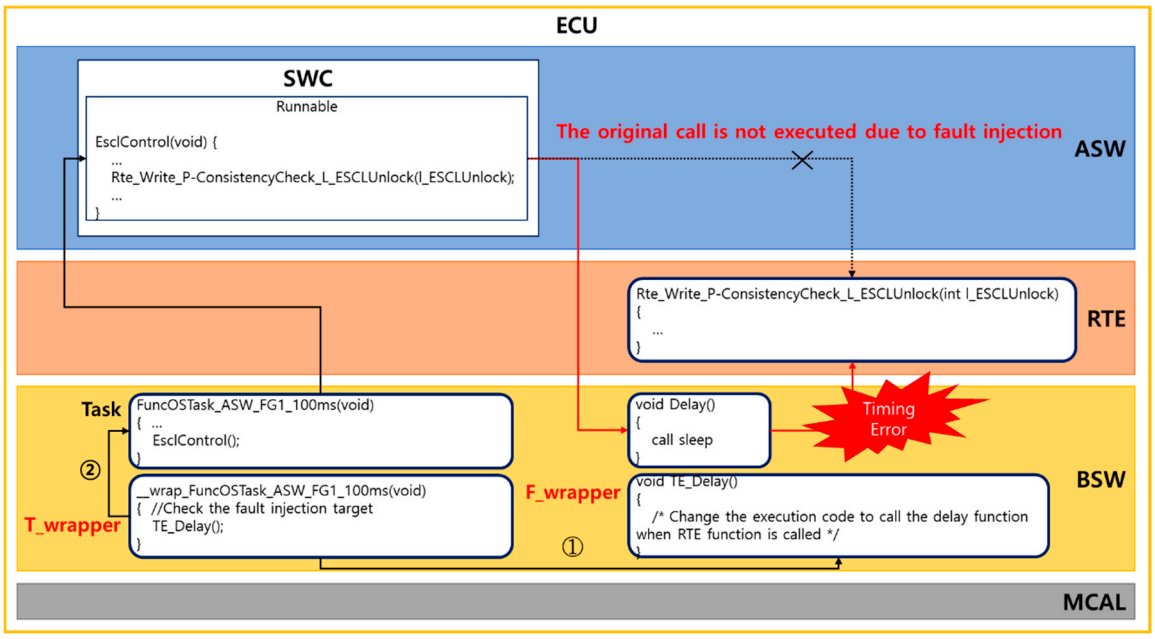


Figure 11. Case 2: timing error fault injection using ASFIT.

3 **Case 3 (asymmetric error): fault ID 40**—asymmetric value fault injection for the return statement of the callee at the SWC–BSW integration test level.

The VCU performs control related to vehicle driving, such as the motor and steering wheel. The fault that we injected calls a function that notifies an error to the entire system when an error occurs. At this time, instead of sending the same value, the error value of one call is changed. When this fault occurs, the system sequentially stops operations and shuts down.

This fault occurs in the integration test level of the SWC and the BSW layer. The fault that we injected occurs in the SWC, as shown in Figure 12. When Det_ReportError() is called, the error value transmitted from the callee Det_ReportError() to another SWC or the ECU is changed uniquely for a specific SWC. When this fault occurs, the safety mechanism that stops all the operations sequentially and shuts down the system is activated. It is difficult to confirm the operation of the safety mechanism only by the operation of the LED. Thus, in this case, to verify the operation of the safety mechanism, a serial port was connected to the target for debugging.

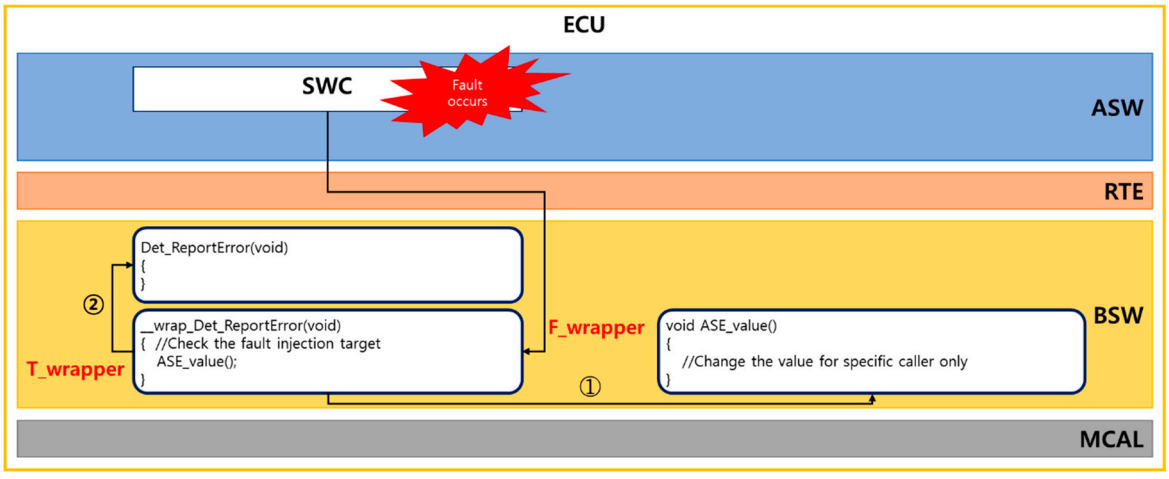


Figure 12. Asymmetric error fault injection using ASFIT.

Figure 12 shows the ASFIT fault injection process. In the case of an asymmetric error, when a fault or event occurs anywhere in the system, the T_wrapper is generated for the function of the BSDW layer, which communicates the fault or event to the entire system. In Case 3, the target for generating the T_wrapper is Det_ReportError(), which is a function that communicates a fault to other SWCs and the ECU. The T_wrapper _wrap_Det_ReportError() calls the F_wrapper (①) to inject a fault and then calls the original function (②). At this time, the fault injection function ASE_value() generates an asymmetric error by changing the error value for a specific caller.

Sending the same value to the entire system is a fault that can be injected by other fault injection methods, but sending a different value only to part of the system in a situation where the same value must be sent throughout the system cannot be performed by hardware or software-based fault injection. ASFIT makes asymmetric fault injection possible by injecting the fault uniquely to a specific calling relationship of the error report function.

6. Runtime Overhead

Real-time performance is critical for automotive software. Although injecting all the software faults listed in Table 2 is important, measuring the performance, specifically, the runtime overhead, is also crucial. As such, when the task execution is delayed by fault injection, it can also cause a side effect aside from the injected fault.

6.1. Measuring the Runtime Overhead

The HW-based FIT tools TRACE32, GHS Probe, and CW IDE were excluded from runtime overhead measurement because they are hardware debuggers and are stopped temporarily before the fault is injected. Therefore, the overhead was measured for the SW-based FIT tools CaNoe-based FIT, G-SWFIT, and GRINDER, and the results were compared with those of ASFIT. However, Kayotee was excluded from runtime overhead measurement because it injects a fault directly to the fault injection position by modifying the software.

The runtime overhead was measured only for software faults related to data error among the five AUTOSAR error types in Table 2. Timing error is not affected by the runtime because it is a fault that delays data transmission or does not transmit data. Moreover, task execution is not important for the program flow error type because this fault type does not call a function or else stops execution by accessing the register. Asymmetric and access errors were excluded because they cannot be applied to other methods; however, their operation is similar to that of data error.

The time at which the fault injection is performed can also influence the runtime overhead. For example, when a fault is injected at the time that the fault injection function is first called, the runtime overhead is always shortest for the method that directly modifies the source code. Hence, the fault was injected when the fault injection target function was called the fifth time.

The runtime overhead was determined by repeating the fault injection target task 10,000 times in the system and measuring the average times before and after applying the fault injection method using the following equation:

$$RuntimeOverhead(\%) = \frac{TaskRuntimeApplyingTheFaultInjectionMethod - TaskRuntimeNotApplyingTheFaultInjectionMethod}{TaskRuntimeNotApplyingTheFaultInjectionMethod} \times 100 \quad (1)$$

6.2. Performance Measurement Results

Figure 13 shows the runtime overhead measurement results for ASFIT and the SW-based FIT tools CaNoe-based FIT, G-SWFIT, and GRINDER. The measurement results showed that the ASFIT runtime overhead was the smallest, with runtime increasing by 1.24% compared with the original runtime. The CaNoe-based FIT, which injects faults using the hook function of AUTOSAR, showed a runtime overhead of 3.22%. G-SWFIT and GRINDER, which are wrapper-based fault injection methods, showed runtime overheads of 2.0% and 6.34%, respectively.

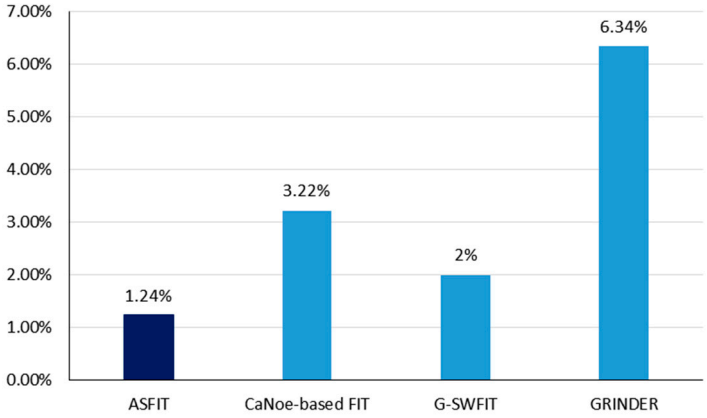


Figure 13. Performance measurement results.

7. Conclusions and Future Research

In this study, we defined types of software faults that can occur in automotive software based on the call relationships between different layers of AUTOSAR, an automotive software platform. We also proposed a method of the software fault injection test for these faults. The proposed method can inject all the automotive software faults defined in Table 2 using software-based methods without any separate hardware device while minimizing the runtime overhead due to fault injection.

We implemented the proposed method as ASFIT and conducted case studies to compare the proposed method with representative existing software and hardware-based fault injection methods. The results showed that ASFIT enabled fault injection for access and asymmetric errors, which cannot be injected by other fault injection methods. It can also perform fault injection for data and timing errors, only some of which can be injected by other methods. Furthermore, when we measured the runtime overhead of the fault injection methods, ASFIT showed the lowest runtime overhead, excluding hardware-based fault injection methods and tools that directly modify the code. Thus, ASFIT was proven to be a lightweight method.

As described in the empirical study in Section 5, a Korean motor company conducted a fault injection test for some ECUs developed based on AUTOSAR in the software development steps using the software faults and fault injection method proposed in this study. Among the faults we injected, faults such as program flow errors due to illegal instructions and timing errors due to CPU clock corruption may vary depending on the hardware. Therefore, we plan to apply our method to more types of ECUs in the future to make it general. Additionally, we will also expand our method for the SW/HW integration test phase during system development.

Author Contributions: J.P. and B.C. contributed to the design and implementation of the research, to the analysis of the results and to the writing of the manuscript. B.C. supervised the findings of this work. All authors have read and agreed to the published version of the manuscript.

Funding: This research was supported by the Hyundai-Kia Motor Company. This research was supported by the MSIT(Ministry of Science and ICT), Korea, under the ITRC(Information Technology Research Center) support program(IITP-2020-2017-0-01628) supervised by the IITP(Institute for Information & communications Technology Promotion)

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Rana, R.; Staron, M.; Berger, C.; Hansson, J.; Nilsson, M.; Törner, F. Increasing Efficiency of ISO 26262 Verification and Validation by Combining Fault Injection and Mutation Testing with Model based Development. In Proceedings of the International Conference on Software Engineering and Applications, Reykjavik, Iceland, 29–31 July 2013.
2. ISO 26262–2011. Road vehicles-Functional Safety-Part 1–10. 2011. Available online: https://www.iso.org/standard/43464.html (accessed on 1 December 2017).
3. Hsueh, M.C.; Tsai, T.K.; Iyer, R.K. Fault injection techniques and tools. *Computer* **1997**, *30*, 75–82. [CrossRef]
4. Han, S.; Shin, K.G.; Rosenberg, H.A. Doctor: An integrated software fault injection environment for distributed real-time systems. In Proceedings of the 1995 IEEE International Computer Performance and Dependability Symposium, Erlangen, Germany, 24–26 April 1995; pp. 204–213.
5. Fürst, S.; Mössinger, J.; Bunzel, S.; Weber, T.; Kirschke-Biller, F.; Heitkämper, P.; Kinkelin, G.; Nishikawa, K.; Lange, K. AUTOSAR—A Worldwide Standard is on the Road. In Proceedings of the 14th International VDI Congress Electronic Systems for Vehicles, Baden-Baden, Germany, 7–8 October 2009.
6. Meng, X.; Tan, Q.; Shao, Z.; Zhang, N.; Xu, J.; Zhang, H. SEInjector: A dynamic fault injection tool for soft errors on x86. In Proceedings of the 2017 International Conference on Computer Systems, Electronics and Control (ICCSEC), Dalian, China, 25–27 December 2017; pp. 1492–1495.
7. Skarin, D.; Barbosa, R.; Karlsson, J. GOOFI-2: A tool for experimental dependability assessment. In Proceedings of the 2010 IEEE/IFIP International Conference on Dependable Systems & Networks (DSN), Chicago, IL, USA, 28 June–1 July 2010; pp. 557–562.
8. Madeira, H.; Rela, M.; Moreira, F.; Silva, J.G. RIFLE: A general purpose pin-level fault injector. In Proceedings of the European Dependable Computing Conference, Berlin, Germany, 4–6 October 1994; pp. 197–216.
9. Jha, S.; Tsai, T.; Hari, S.; Sullivan, M.; Kalbarczyk, Z.; Keckler, S.W.; Iyer, R.K. Kayotee: A fault injection-based system to assess the safety and reliability of autonomous vehicles to faults and errors. *arXiv Prepr* **2019**, arXiv:1907.01024.
10. Lanigan, P.E.; Narasimhan, P.; Fuhrman, T.E. Experiences with a CANoe-based fault injection framework for AUTOSAR. In Proceedings of the 2010 IEEE/IFIP International Conference on Dependable Systems & Networks (DSN), Chicago, IL, USA, 28 June–1 July 2010; pp. 569–574.
11. Duraes, J.; Madeira, H. Emulation of software faults: A field data study and a practical approach. *IEEE Trans. Softw. Eng.* **2006**, *32*, 849–867. [CrossRef]
12. Winter, S.; Piper, T.; Schwahn, O.; Natella, R.; Suri, N.; Cotroneo, D. GRINDER: On reusability of fault injection tools. In Proceedings of the 2015 IEEE/ACM 10th International Workshop on Automation of Software Test, Florence, Italy, 23–24 May 2015; pp. 75–79.
13. Islam, M.M.; Karunakaran, N.M.; Haraldsson, J.; Bernin, F.; Karlsson, J. Binary-Level Fault Injection for AUTOSAR Systems (Short Paper). In Proceedings of the 2014 Tenth European Dependable Computing Conference, Newcastle, UK, 13–16 May 2014; pp. 138–141.
14. Barr, M. Bookout vs. Toyota: 2005 Camry L4 Software Analysis. District Court of Oklahoma County. 2013. Available online: http://www.safetyresearch.net/Library/BarrSlides_final_scrubbed.pdf,consultadoel,10 (accessed on 28 April 2020).
15. Agrawal, H.; DeMillo, R.A.; Hathaway, B.; Hsu, W.; Hsu, W.; Krauser, E.W.; Martin, R.J.; Mathur, A.P.; Spaord, E. *Design of Mutant Operators for C Programming Language*; Technical Report SERC-TR-41-P, SERC; Purdue University: West Lafayette, IN, USA, 1989.
16. Delamaro, M.E.; Maidonado, J.C.; Mathur, A.P. Interface mutation: An approach for integration testing. *IEEE Trans. Softw. Eng.* **2001**, *27*, 228–247. [CrossRef]
17. Ghosh, S.; Mathur, A.P. Interface mutation. *Softw. Test. Verif. Reliab.* **2001**, *11*, 227–247. [CrossRef]
18. AUTOSAR. "Explanation of Error Handling on Application Level," AUTOSAR, Munich. 2009. Available online: https://www.autosar.org/fileadmin/user_upload/standards/classic/4-0/AUTOSAR_EXP_ApplicationLevelErrorHandling.pdf (accessed on 28 April 2020).
19. MDS Technology. Trace32 Debugger. Available online: http://www.mdstec.com/ (accessed on 28 April 2020).

20. Green Hills Software. GreenHills Debug Probes. Available online: http://www.ghs.com/ (accessed on 28 April 2020).
21. Metrowerks. CodeWarrior IDE. Available online: http://www.nxp.com/ (accessed on 28 April 2020).

© 2020 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (http://creativecommons.org/licenses/by/4.0/).