Article

Fault Localization Method by Partitioning Memory Using Memory Map and the Stack for Automotive ECU Software Testing

Kwanhyo Kim, Ki-Yong Choi and Jung-Won Lee *

Department of Electrical and Computer Engineering, Ajou University, Suwon 16499, Korea;
khyo0317@gmail.com (K.K.); ki815kaisian@gmail.com (K.-Y.C.)

* Correspondence: jungwony@ajou.ac.kr; Tel.: +82-31-219-1813

Academic Editor: Antonio Maffucci

Received: 2 July 2016; Accepted: 9 September 2016; Published: 19 September 2016

Abstract: Recently, the usage of the automotive Electronic Control Unit (ECU) and its software in cars is increasing. Therefore, as the functional complexity of such software increases, so does the likelihood of software-related faults. Therefore, it is important to ensure the reliability of ECU software in order to ensure automobile safety. For this reason, systematic testing methods are required that can guarantee software quality. However, it is difficult to locate a fault during testing with the current ECU development system because a tester performs the black-box testing using a Hardware-in-the-Loop (HiL) simulator. Consequently, developers consume a large amount of money and time for debugging because they perform debugging without any information about the location of the fault. In this paper, we propose a method for localizing the fault utilizing memory information during black-box testing. This is likely to be of use to developers who debug automotive software. In order to observe whether symbols stored in the memory have been updated, the memory is partitioned by a memory map and the stack, thus the fault candidate region is reduced. A memory map method has the advantage of being able to finely partition the memory, and the stack method can partition the memory without a memory map. We validated these methods by applying these to HiL testing of the ECU for a body control system. The preliminary results indicate that a memory map and the stack reduce the possible fault locations to 22% and 19% of the updated memory, respectively.

Keywords: Electronic Control Unit (ECU); automotive software; fault localization; embedded testing; memory map; Hardware in the Loop (HiL); memory update

1. Introduction

Approximately 90% of all recent car innovations are based on the automotive Electronic Control Unit (ECU) and its software. These include active/passive safety systems (e.g., airbags and collision avoidance), driver assistance systems (e.g., lane-departure warning and night vision), and infotainment systems (e.g., telephone, internet, and TV) [1]. However, as the number of ECUs in a car increases, the functional complexity of their software also increases. Accordingly, the likelihood of faults due to systematic ECU problems is increasing [2]. A car malfunction caused by a fault of an automotive ECU can threaten driver safety as well as create potential legal difficulties. In order to solve such problems, systematic testing/debugging techniques are required that ensure software reliability while taking its complexity into account [3].

Generally, finding the location of a fault requires significant amounts of time and effort [4]. However, the fault-location information available to developers of automotive software is currently too restrictive. Therefore, the problem is that it takes a large amount of effort and time to localize the faults. One way to solve this problem is to provide the developer with fault-location information during

testing. In this paper, we propose a method whereby a tester of an automotive ECU can generate memory-location information about a potential fault during testing.

The fundamental cause of this problem is original equipment manufacturer (OEM) systems in the automotive industry, as shown schematically in Figure 1. An OEM system means a series of steps in the process where developers and a tester perform development, testing, and debugging in independent environments. Firstly, an OEM designer provides the manufacturer's requirements to the developers working for each supplier. Each developer then develops an independent ECU based on these requirements, performs unit testing on it, and then delivers it to a tester in the OEM company. This tester receives the software in a binary form with the ECU in order to protect the corporate intellectual property (IP) rights, whereupon they perform black-box testing of each supplier's ECU and send the test results back to each relevant developer. Finally, each developer performs debugging based on these test results [5].

Figure 1. The process of developing automotive Electronic Control Unit (ECU) software.

Unfortunately, there is relatively little information that is exchangeable between the OEM company and supplier companies because development and testing are performed in independent environments. The tester may find it difficult to locate the cause of a fault because he/she receives the software in a binary form (not as source code) because of IP protection. Therefore, developers find it difficult to locate faults because they do not receive location information about the faults that occurred during testing. This difficulty is shown by the dotted line of Figure 1. A tester performs black-box testing using an HiL simulator to assess whether the ECU functions correctly in accordance with the requirements [6]. Accordingly, it is impossible to find the source-code location of faults that occur during testing. Consequently, developers receive a test script and associated results that do not contain fault-location information. As shown in Figure 2a, a developer equipped with the source code can use it for purposes such as locating a fault, obtaining its description, establishing the list of called functions, and determining the pertinent value of each variable. However, as shown in Figure 2b, the test results currently give only information such as the name of the test script, the time of the test, the number of test cases, and pass/fail information. Therefore, developers who debug automotive software are at a disadvantage compared to those who perform debugging with the source code. Moreover, testers experience difficulties when debugging because they perform integration testing on independently developed ECUs with various types of interfaces.

Figure 2. Debugging information with/without source code. (a) Debugging information with source code; (b) debugging information without source code.

The memory information can be used in such an environment to provide fault-location information to developers [7]. This is because memory information is of the same type for each ECU, regardless of its interface. For example, a binary file is executed independently in each ECU in the associated environment that was developed based on AUTOSAR (AUTomotive Open System ARchitecture) or multiple ECUs. Therefore, a trace of the software that is independently executed with the platform is left in the ECU memory. Accordingly, this information can be used for each ECU in the same way. Also, memory use is independent between modules because each ECU developer determines the range of memory use before developing a module.

Meanwhile, the fundamental reason that the memory can be used for fault localization is the Def and Use (DU) pair of variables in the source code, which leave “footprints” in the memory during program execution. For example, in the statement “A = B”, which consists of variables A and B, A is defined as “Def” and B is defined as “Use” [8]. The memory update occurs in the memory address of the Def variable by combining memory interface commands such as LOAD, STORE, and MOVE when this statement occurs during program execution [9].

Thus, the most intuitive way to provide the location information of the fault to the developer using the memory update is to provide the pair of the updated addresses for fail test and fault time. By utilizing this information, developers are able to localize faults targeting the list of updated addresses at a particular time [10]. However, a disadvantage of this method is that it requires significant computational time and expense to localize a fault within an excess of memory updates. To overcome this, a method is required that can reduce the extent of potential fault locations by using the memory update information. In this paper, we propose such a method for reducing both time and expense, whose target is the data section of the memory.

Fault localization can be divided into two parts. First, suspicious code is identified that may contain a fault. Next, the developer determines whether it actually contains a fault [11]. In this paper, fault localization refers to the task of finding candidates for faulty symbols (symbol name and address) by tracking the memory based on the suit of failed tests (binary source file, test case, and memory) from the HiL simulation.

In order to reduce the extent of potential fault locations in the data section, we partition it using a memory map and the stack. Unlike a general memory map, the one used in automotive software includes information about the memory range to be used for each object file. This makes it possible to use the memory map to divide the data sections into object-file units. It is also possible to use the stack to divide the data section by predicting the information in a memory map. This is because the stack works by being divided into each functional unit while the automotive software is running. It is then possible to divide the data section by subjecting it to the unit information defined by the stack. The stack information is useful for dividing the data section because it is possible to apply the call relationships to the partitioned stack section, in much the same way as the call relationships that exist between functions in the source code. Meanwhile, there is the constraint that a developer has to transmit a memory map to a tester in the method by memory map. Therefore, we propose memory-map and stack methods that use only the memory information.

In this way, we adopt the standpoint of the tester when locating faults that occurred during the integration test. The tester executes the HiL test in a black-box environment and acquires the information related to a fault by utilizing the memory update information. However, the size of this information is considerable, and it is therefore possible to waste a significant amount of resources and time when localizing the fault. Hence, it is necessary to reduce the amount of target information. We partition the memory by using the memory map and the stack. We also reduce the amount of memory update information by finding those memory regions with high fault suspiciousness in the partitioned memory. The fault suspiciousness and reduced memory update information can be provided to developers; therefore, it is possible to reduce the time and resources required for debugging. In other words, it is possible to help the developer to perform debugging by adopting the standpoint of the tester. The contents of this paper are as follows. In Section 2, we analyze previous studies related to the method proposed here. In Section 3, we describe the preparation for applying the fault localization method. In Section 4, we explain the memory-map and stack methods for partitioning the memory and calculating the fault suspiciousness of each partition. In Section 5, we validate our method by applying it to the results of an HiL simulation. Finally, in Section 6, we conclude the paper and present possible future work.

2. Related Work

In general, testing and debugging are essential for ensuring reliability; they account for a significant fraction of any software development process. Debugging consists of two steps. The first is to locate the fault in order to find where the bug exists. The second is to correct the fault in order to fix the located bug [11]. Much time is spent finding the locations of faults in such a process. Therefore, it is important to reduce this time in order to save costs used in software development and maintenance [12]. Fault localization has mainly been developed based on the source code written by the developer [4]. However, the tester executes black-box tests without the source code. Research described below has been conducted in this regard to localize the faults.

2.1. Testing and Debugging in a Black-Box Environment

Research has been carried out on testing and debugging in a black-box environment. Various methods have been studied for finding functional problems, given that source code is unavailable in a black-box environment [6]. However, these methods are mainly for determining/classifying the test input in order to efficiently test the program's functionality. Therefore, they are generally not helpful for locating faults that have occurred in a program. Previous research [13] on black-box testing proposed an online fault localization system for finding the faulty component when a program is functioning abnormally. In other words, the proposed method identifies the component causing the failure in a multicomponent system, rather than finding the exact fault location in the program. However, determining the cause of faults in terms of component units is not helpful for debugging because a different supplier generally develops each component of an automotive electronic system. Therefore, a method is required that can find the accurate location of a fault in a component.

A fault-locating method exists that works by connecting a debugger externally to the ECU [14]. This involves connecting the debugger to debugging pins on the outside of the ECU and monitoring the memory state during program execution, as in the joint test action group (JTAG) background debug mode (BDM). However, this method is simply a tool for observing the memory state. A suitable algorithm is necessary in order to use this method to generate information that can help to localize faults in the source code.

2.2. Testing and Debugging Method for Automotive ECUs

A variety of research has been carried out with respect to testing and debugging automotive ECUs. In [15,16], testing methods were studied in a black-box environment using an HiL simulator at the integration step of an automotive ECU. The integration testing method proposed in [15] determines

the placement of the ECU by using the Universal Measurement and Calibration Protocol (XCP); it detects an abnormal signal through the signal value. The ECU testing method proposed by [16] models the ECU connection structure using the electric and electronic (EE) architecture in an HiL-test-system (HiL-TS) environment. However, these methods are merely for checking the functionality of an ECU, and do not address fault location.

Alternatively, debugging methods have been studied in Software-in-the-Loop (SiL) environments in which the source code is available [17,18]. A development process has been proposed for system integration in an SiL environment that can save system-development time and cost [17]. Another method is to model automotive software using the Modelica tool, and to debug by connecting an external debugger to an SiL environment running the simulation [18]. However, since these methods cannot be applied to automotive ECU debugging in an environment where the source code is unavailable, a method that can generate fault-location information in a black-box environment is still required.

2.3. Fault Localization Using Memory Information

Fault locations in memory represent indirect fault-location information for developers with source-code access. It is possible to find the fault-location information in the memory by capturing it during program execution by using an external controller area network (CAN) terminal for communication between the automotive ECUs [19,20]. Methods for fault localization using memory information have been studied. In [21], research for fault localization was performed by capturing memory information in a case of malfunctioning software distributed to several clients. However, this method is not suitable for the automotive ECU environment because developers receive memory information from multiple clients in a distributed system. Also, studies have been conducted on a fault-locating method in which the memory state is visualized [22], and on a method for finding fault locations by observing the memory while the program is executed repeatedly [23]. However, these methods cannot be applied to an automotive ECU testing environment in which the source code cannot be accessed because they do not preclude its use.

A fault-localization method has been researched that uses the memory-update information of automotive software [7]. However, this is limited by the number of memory updates that can occur in the entire memory. Therefore, in this paper, we propose a method to reduce the fault candidate region in the entire memory.

3. Preparation for Fault Localization

This section describes a fault-localization method for software installed in an automotive ECU. For this purpose, we first describe the memory-map and stack methods utilizing the memory, and then the overall process of this method.

3.1. Target Memory for Automotive ECU Software

In general, memory is composed of a data section, a text section, and a stack section, depending on its intended use. The fault-localization method proposed in this paper is aimed at the data section. The definition of each memory section is as follows.

- **Text section:** this is where the program's source code is stored; it is used only for 'read' during program execution. Therefore, this section cannot be used for fault localization.
- **Stack section:** this is assigned to each function called during program execution. Therefore, considering regions in this section as potential fault locations is not helpful to developers because this section is allocated dynamically. However, it does contain information such as the Return Address (RA) for each function, the Stack Pointer (SP), and local variables. Therefore, it is possible to partition the stack section into functional units by using this information. A partitioned stack section can be used to partition the data section.

- **Data section:** this is the region assigned to global variables. Treating regions in this section as potential fault locations means identifying a specific object file or symbol as a fault candidate, since this section is fixed and allocated by object file/symbol unit during program execution.

For these reasons, we propose a method that can reduce the fault candidate region in the data section. In order to do this, it is necessary firstly to partition the data section. This section is used in accordance with the object file/function unit determined at source-code level. The memory map and the stack also include information about object-file unit or function unit that is determined at a high level. Therefore, we partition the data section based on the memory map and stack.

3.1.1. Memory Usage Based on a Memory Map

A memory map is a file generated at the time of building the software that contains the memory-usage information. However, there is a difference between the information in a general memory map and that in one for automotive software. As shown in Figure 3a, a general memory map includes section information depending on their intended uses. However, a data section in the memory for automotive software is specified for each object file due to the issue of having to integrate each supplier's software. Therefore, the memory map for automotive software also contains information in its data section based on the unit of each object file (Figure 3b). Thus, automotive software memory information can be partitioned into object-file units based on its memory map.

Figure 3. Memory maps for general-purpose and automotive ECU software. (a) General-purpose memory map; (b) an example of memory map for automotive ECU software.

3.1.2. Memory Usage Based on the Stack

The reasons for being able to partition the data section based on the stack are as follows. Firstly, there is a function unit in the source code. Calling a function requires a return address and the memory region of the associated variables. The stack section is used for these purposes, and hence there is a unit in the stack that matches the function unit. This unit is the control section (CS), based upon which the stack can be partitioned [24]. In addition, there are call relationships between CSs, as there are between functions. Therefore, it is possible to partition the data section by CS units and the call relationships between them.

Thus, a CS unit is equal to a function unit, and also to an object-file unit in the case where an individual object file contains only one function. Here, however, we determine CS units by acquiring the stack information during HiL testing, in accordance with the operational cycle of the system installed on the ECU. Therefore, the stack information used to determine a CS unit does not include all of the stack information that the ECU operating system actually uses. That is, the data section is partitioned by units of overlapping regions of multiple object files. This is because the data section is partitioned by using the remaining information in the stack at each operational cycle.

in an environment that is dependent on the operational cycle of HiL testing. Therefore, memory partitioning based on the stack is less precise than that based on a memory map.

3.1.3. Example of Memory Usage

An example of memory-map and stack partitioning of the data section is shown in Figure 4. The pattern displayed in Figure 4 is the data section partitioned by the memory map. This involves 15 regions, including 2 duplicate ones. The information partitioned by the stack is displayed in color; it is partitioned into 5 regions. Thus, we confirm that the memory-map method can partition the memory more finely than the stack one. This is because the stack is allocated and released dynamically during program execution, so only the stack information remaining at the time of memory capture can be used. However, the memory map contains the data-section allocation information for each object file. Therefore, as shown in Figure 4, the stack-partitioned memory unit contains the memory-map partitioned one. A tester requires the memory map in order to access the memory that is available to developers. However, since the memory map contains confidential matters regarding how automotive-software object files use memory, a case can be made based on security grounds for denying testers access to the memory map. Therefore, we propose the stack method, which is capable of providing fault-location information without disclosing memory-map information. As shown in Figure 4, it is possible to determine the fault candidate region in terms of object-file or CS units by partitioning the memory.

Figure 4. Example of memory partitioning by memory map and stack.

3.2. Process of Fault Localization

As shown in Figure 5, the overall objective of this study is to provide developers with test results that include the potential memory location of each fault. This is provided with the symbol list in each partition in the form of a ranking according to fault suspiciousness. Figure 5 relates to the section displayed within the dotted line in Figure 1; it describes a method by which a tester can indicate the fault candidate region to developers. There are two memory-partition methods in the overall process of fault localization: ① the memory-map method, and ② the stack method without a memory map (Figure 5). We firstly explain memory-map partitioning, and then that by the stack.

Figure 5. Fault localization by partitioning memory in two ways for automotive software.

The fault-localization method comprises four steps (Figure 6). As mentioned previously, the third of the four steps (highlighted by a dashed line) is the memory-partitioning step, by either the memory map or the stack. As shown by the solid line in the fourth step in Figure 6, the final result is a fault-suspiciousness ranking based on that of each partition. The individual steps are described below.

Figure 6. Process of fault localization.

- **Capturing memory information:** this is done by CAN communication during program execution in the black-box testing environment using the HiL simulator; memory information refers to the set of values at each address [20]. In addition, files are generated related to the configuration information of the memory section (section file), the allocation information of each symbol in the memory (symbol file), and the function call relationship (call file) from the automotive software via the binary utility [25]. These files are used in the memory-partitioning step.
- **Identifying the set of updated addresses:** a list of addresses updated each time is extracted based on the acquired memory information.
- **Partitioning the data section:** this step can be performed in parallel with the second stage; the memory map and the stack are used to partition the memory.
- **Fault localization by FS:** fault suspiciousness (FS) is calculated by applying the partitioned memory obtained in the second step to the memory-update information obtained in the third step.

The FS ranking of partitions is then determined based on the calculated FS. The number of fault candidates is reduced because the final result is provided in terms of the FS ranking of the top 10 partitions, is provided with the symbol list of fault candidates.

3.3. Identifying the Set of Updated Addresses

This section describes the final preparation step for fault localization. In the second step of Figure 6, we identified the set of updated addresses based on the acquired memory information. This information is captured N times with P cycles targeting the program with execution time T , and is composed of M addresses; P refers to a period of the main task in the system. A frame (F_k) is defined as the set of memory information captured each time. Furthermore, whether each address is updated is determined by the change of the value at that address. That is, an address is updated only if its value changes from the previous frame to the present one. The definition of terms related to this is as follows:

$$\text{Memory Address Set (A)} \equiv \{a \mid 0 \leq a < M; a \text{ is an address}\}; \quad (1)$$

$$\begin{aligned} \text{Frame (F}_k) &\equiv \{V_{a,k} \mid V_{a,k} \text{ is the value of the address } a \text{ at time } P \times k, \\ &a \in A, k \text{ is a frame number, } P \text{ is a period of the system main task}\}; \end{aligned} \quad (2)$$

$$\text{Memory Update (U}_{a,k}) = \begin{cases} 1, & V_{a,k} \neq V_{a,k-1} \\ 0, & V_{a,k} = V_{a,k-1} \end{cases} \text{ for } a \in A, k \text{ is a frame number.} \quad (3)$$

The update information calculated in this way can be used for fault localization in a failed test. This is because test failure is closely related to the test-script inputs, and it is possible to find the address estimated as being at fault by processing the update frequency approximately simultaneously with an input being set. In a previous study [7], a range with a relatively high input quota based on the update frequency of each address (i.e., a range in which the number of updated addresses increases and remains above average) was defined as an input-driven update range (IDUR). The updated addresses in this range can be regarded as being related to the faults generated in the test cases. This is because IDUR is a range in which the number of update addresses increases while the output changes in response to the input. In this paper, we use the IDUR method to calculate the FS.

4. Fault Localization Method

Faults can be localized by either the memory-map method or the stack one. The former has the advantage of being able to partition the memory more finely, while the latter has the advantage of being able to localize faults without a memory map. The remaining goal is to find the fault candidates associated with the causes of each fault by using both methods. Here, a fault candidate refers to the symbol that is closely related to an input and that is found based on the memory update information.

4.1. Partitioning by Memory Map

In Figure 7, memory map (MM) is the input of a memory-map partitioning method. As shown in Figure 3b, it contains the range of each section, as well as that allocated to each object file. Therefore, we partition the data section using the range allocated to each object file in the memory map, i.e., the start-address information and the size of the data section. Lines 1–2 in Figure 7 refer to each object file ($OBJ(n)$) in the memory map, and a partitioned data section ($PD(t)$). In lines 3–8, we define the criteria for partitioning the data section based on the address range of each object file in the memory map. As shown in line 4, an object file that is not allocated to the data section is excluded from being a fault candidate in the memory-partitioning step.

Algorithm 1. The Algorithm for Partitioning the Data Section by Memory Map**INPUT :** MM, memory map for automotive software**OUTPUT :** PD, the data section partitioned by memory map

```

1: OBJ(n)  $\equiv$  the  $n^{\text{th}}$  object file in MM
2: PD(t)  $\equiv$  the  $t^{\text{th}}$  range of address partitioning the data section

3: WHILE not the end of MM do
4:   if(OBJ(n) is allocated the data section) then
5:     PD(t)  $\leftarrow$  OBJ(n).data section
6:     t  $\leftarrow$  t + 1
7:     n  $\leftarrow$  n + 1
8:   END WHILE

9: ASCENDING_SORTING(PD)
10: RETURN PD

```

Figure 7. Algorithm for partitioning the data section by memory map.**4.2. Partitioning by the Stack**

The partitioning method by the stack involves partitioning the data section using itself and the stack data section. As shown in the flow chart of Figure 8, the files from a binary file (executable) about an ECU acquired by using the values set stored on the stack and binary utilities are utilized. For each file that is acquired, the configuration information of the memory section (section file (SEC)) and the function-call information (call file (CALL)) are acquired by the objdump program. The memory-allocation information of each symbol (symbol file (SYM)) is acquired by a gnu binary utility program [25]. The stack accesses the address values from bottom to top according to the default address-size of the system. Partitioning the data section by the stack is carried out in three stages. First, we partition the stack into control-section units using the value stored in the stack and the section file. Then, we identify each control section using the symbol and call files. Finally, we partition the data section used by each control section. An explanation of each of these steps is given by using the example of Figure 9.

Figure 8. Flowchart of partitioning the data section using the stack.

- **Step 1: Partitioning the stack section**

In this step, we partition the stack section into CS units (Figure 8①–⑤). The CS structure is in the form {(local variable) *, LVG *, RA, SP} (Here, * means that occurs over zero or more than once, LVG is

local variable sheared with global, RA is return address, and SP is stack pointer). We now turn our attention to Figure 9. If we read the value of the stack (Figure 9a) with memory allocated from the bottom address, this can be represented by (address, value) pairs (Figure 9b). If we check the values from the bottom address based on the flow chart, the address 40004028 is a local variable because it does not correspond to the address range of the SEC file (Figure 9f; cf. Figure 8①). The next address 40004024 is the LVG. In this address, 40000F20 is included in the scope of the data section of the SEC file (Figure 9f; cf. Figure 8④). In the same way, address 40004020 is the RA (Figure 8③). However, the value in address 4000401C is 00000000, and address 4000401C is the SP because the first SP in the stack points to the starting address of the entire memory. If we find the SP depending on Figure 8②, the CS index is increased. If this process is repeated until the top address is reached, it is possible to partition the stack into CS units (Figure 8⑤), as shown in Figure 9c. In this way, the task of partitioning the stack into CS units is repeated during the total frame. Each CS with the same RA is considered an identical CS because it is judged to be used by the same function.

● Step 2: Identifying the control section

We identify the information in each CS (Figure 8⑥,⑦). As we confirmed in the previous step, address 40004020 in Figure 9 is the RA, and its value is 0004A5E8. In the SYM file (Figure 9g), 0004A5E8 is included in the range of Func_A. Func_A is known to call Func_B in the CALL file (Figure 9h). In other words, Func_A is the function of the first CS and has Func_B as the caller. If this process is performed for each CS, it is possible to obtain a CS list as shown in Figure 9d.

● Step 3: Partitioning the data section

Finally, the data section is partitioned by the LVG in each CS (Figure 8⑧–⑩). We make reference points for partitioning the data section using each LVG and CS in Figure 9e. If these reference points are sorted, the data section is partitioned as shown in Figure 9i. The LVG is the boundary for partitioning the data section. Therefore, the more LVGs stored on the stack, the more finely the data section can be partitioned. This process is represented as the algorithm in Figure 10.

Figure 9. Example of partitioning the data section using the stack (a–i).

Algorithm 2. The Algorithm for Partitioning the Data Section by the Stack	
INPUT : STACK, the set of (address, value) in stack section SEC, the range of address allocated to each section SYM, the range of address allocated to each symbol(function) CALL, the set of call relation between symbols(functions) OUTPUT : PD, data section partitioned by the stack 1: STACK(n) $\equiv$ (address, value) in stack section 2: PD(t) $\equiv$ the t th range of address partitioning the data section 3: CS $\equiv$ the list of control sections (LVG: local variable shared with global)	Step 2. Identifying the control section 15: FOR i $\leftarrow$ 1 to SIZE(CS) do 16: if (FIND(CS(i).return address in SYM)) then 17: CS(i).func $\leftarrow$ function symbol 18: if (FIND(CS(i).func in CALL)) then 19: CS(i).callee $\leftarrow$ call relation 20: END FOR
Step 1. Partitioning the stack section 4: FOR n $\leftarrow$ last address to first address in stack section do 5: if (STACK(n) is in SEC(stack)) 6: CS(i).stack pointer $\leftarrow$ STACK(n) 7: i $\leftarrow$ i + 1 8: else if (STACK(n) is in SEC(text)) 9: CS(i).return address $\leftarrow$ STACK(n) 10: else if (STACK(n) is in SEC(data)) 11: CS(i).LVG $\leftarrow$ STACK(n) 12: else 13: CS(i).local variables $\leftarrow$ STACK(n) 14: END FOR	Step 3. Partitioning the data section 21: FOR j $\leftarrow$ 1 to SIZE(CS) do 22: if (CS(j).LVG is not null) then 23: PD(t) $\leftarrow$ CS(j).LVG 24: t $\leftarrow$ t + 1 25: END FOR 26: ASCENDING_SORTING (PD) 27: RETURN PD

Figure 10. Algorithm for partitioning the data section by the stack.

4.3. Fault Localization by Fault Suspiciousness

The FS (FS_j) of each partition (PT_j) is calculated by applying the memory update information to the partitioned memory information. FS is a numerical value obtained by determining the non-executing region of the updated partitions ($FSP_{k,j}$) in each frame, and by applying a weight to the number of determined times ($NFSP_j$) in an entire frame based on their relevance to the input. The terms used to calculate FS are as follows. Terms F_k and $U_{a,k}$ used in (5) are defined by (2) and (3) in Section 3.3. As described there, IDUR defined in (8) is the range in which the number of update addresses increases while the output changes according to the input. Like (9), FS is calculated by the ratio between the numbers of frames determined as FS partitions in all frames and in IDUR, respectively. This is because FS is calculated based on the number of times an FS partition is determined and the correlation between the input and the update information. The algorithm calculating FS on the basis of such defined terms is shown in Figure 11.

Algorithm 3. The Algorithm for Fault Localization by FS	
INPUT : UC, the number of updated address in data section for each object file F^{IDUR} , the range of frames for IDUR F , the range of the total frames OUTPUT : FS, the fault suspiciousness for each objet file 1: $UC_{k,j} \equiv$ the number of update address in data section for each object file j in frame k of UC 2: $NFSP_j \equiv$ the number of frame that partition j is determined as FSP in the total frames 3: $NFSP_j^{IDUR} \equiv$ the number of frame that partition j is determined as FSP in IDUR 4: $FS_j \equiv$ the fault suspiciousness for each object file j of FS 5: FOR k $\leftarrow$ 1 to the total number of frames do 6: Average $\leftarrow$ SUM(UC)/SIZE(UC) 7: FOR j $\leftarrow$ 1 to SIZE(UC) 8: if (0 < $UC_{k,j}$ < Average) then 9: if (k is in IDUR) then 10: $NFSP_j^{IDUR} \leftarrow NFSP_j^{IDUR} + 1$ 11: $NFSP_j \leftarrow NFSP_j + 1$ 12: END FOR 13: END FOR 14: FOR j $\leftarrow$ 1 to SIZE(UC) 15: $FS_j \leftarrow (NFSP_j^{IDUR} \div F^{IDUR}) \div (NFSP_j \div F)$ 16: END FOR 17: RETURN FS	

Figure 11. Algorithm for fault localization by fault suspiciousness (FS).

Partition (PT_j) $\equiv j^{th}$ memory region partitioned by the memory map and the stack (4)

Update Count ($UC_{k,j}$) $\equiv$ number of $U_{a,k}$ in PT_j for F_k (5)

Fault – suspicious Partition ($FSP_{k,j}$) $\equiv \{(PT_j, UC_{k,j}) | 0 < UC_{k,j} < AVG_k\}$ (6)
 $AVG_k = \frac{\sum \# \text{ of updated addresses in } F_k}{\# \text{ of partitions in } F_k}$

Number of Fault – suspicious Partitions ($NFSP_j$) $\equiv$ number of frames PT_j determined as FSP (7)

Input – driven Update Range ($IDUR$) $\equiv$ the range of the number of update addresses (8)
 while the output is changed according to the input

Fault Suspiciousness (FS_j) $\equiv \frac{NFSP_j \text{ in } IDUR}{\text{The number of frames in } IDUR} \div \frac{NFSP_j \text{ in all frames}}{\text{The total number of frames}}$ (9)

Firstly, the number of update addresses ($UC_{k,j}$) in each partition is calculated in each frame. Next, FS partitions are determined based on the number of update addresses in each partition. As in Figure 12, partitions are classified into executing regions ($UC_k \geq AVG_k$), non-executing regions ($UC_k = 0$), and FS partitions ($FSP_{k,j}$, $0 < UC_k < AVG_k$) based on the average number of update addresses (AVG_k) in each frame.

Figure 12. Three types of partition in the data section.

FS partitions are determined in each frame based on these criteria, and the number of frames ($NFSP_j$) determined as having FS partitions is calculated. Finally, FS (FS_j) is calculated by using $NFSP_j$ and the number of frames ($NFSP_j^{IDUR}$) determined as having FS partitions in IDUR based on the accumulated number of frames (Figure 11, lines 14–16). The equation for calculating the fault suspiciousness is implemented in Figure 11, line 15. In other words, by calculating a ratio determined by FS partitions in all frames and one determined by such partitions in IDUR, the ratio of the latter to the former is used as the FS. This ratio can be used for FS because the more frames that are determined as having FS partitions in IDUR (which is highly relevant to the input during the entire execution time), the more addresses that are associated with faults. In other words, this process calculates relevance to input. Finally, the FS ranking of partitions is determined on the basis of the FS of each partition.

5. Experimental Results

We applied the proposed algorithm to an actual body-control module (BCM). We tested whether the BCM operates normally depending on the input by simulating the other modules in the car by utilizing the HiL simulator used on site. This BCM is SPC5604B BOLERO; the memory sizes used by

the binary file are a 374-KB text section, a 24.6-KB data section, and a 4-KB stack section. As shown in Table 1, we obtained six data sets that captured the memory in 10 ms blocks (the main task period of the system). In this section, we present specifically the experimental results for data set #1.

Table 1. Specification of each data set (RAM size = 29.4 KB).

Index	Test Case ID	Test Case Name	Lab ID	Lab Name	Frame (N)
1	1	WARNING_TC2	74	1006_LAB25	400
2	1	WARNING_TC2	75	1006_LAB27	400
3	1	WARNING_TC2	95	1013_LAB39	400
4	2	TELECMATICS_TC2	96	1013_LAB40	400
5	11	TELECMATICS_TC4	127	1029_LAB1	650
6	12	TELECMATICS_TC16	131	TC16_LAB3	800

5.1. Results by Memory Map

The results of the memory-map method are as follows. The number of update addresses for each partition is shown underlined in Table 2. In addition, values for IDUR (the range of frames related to the input) are shown inside the bold borders in Table 2; the ranges are 53–74 frames and 253–276 frames, respectively.

- Total number of partitions = 308
- Number of partitions in which memory update occurred = 67
- FS partitions (FSP) in Frame #60: objs. #8 and #20 ($0 < UC < AVG = 1.84$)
- FS partitions (FSP) in Frame #261: objs. #8, #19, and #39 ($0 < UC < AVG = 1.55$)

The calculated FSs and rankings are shown in Figure 13. The FSs of the top 18 partitions are the same, so we show these. For reference, the rank 2 FS is 6.52 and the rank 3 FS is 5.95, in comparison to a rank 1 FS of 8.70. Obj files with high FS values in the graph have a common feature. These files are affected by the size of the ratio determined as a fault-suspicious region in IDUR rather than the entire frame.

Table 2. Number of update addresses for each partition based on memory map (Test Case ID = 1, Lab ID = 74, Lab Name = 1006_LAB25, Frame = 400).

Object File	Frame														
	Normal Range						IDUR			N.R.	IDUR			N.R.	
	#1	#2	#3	#4	#5	...	#60	#61	#62	...	#261	#262	#263	...	#399
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
Obj #8	<u>1</u>	<u>1</u>	<u>1</u>	<u>1</u>	<u>1</u>	...	<u>1</u>	<u>1</u>	<u>1</u>	...	<u>1</u>	<u>1</u>	<u>1</u>	...	<u>1</u>
Obj #9	5	5	5	7	5		5	5	5		5	5	5	...	5
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
Obj #19	<u>1</u>	0	0	<u>1</u>	<u>1</u>	...	0	<u>1</u>	0	...	<u>1</u>	0	<u>1</u>	...	<u>1</u>
Obj #20	0	0	0	0	0	...	<u>1</u>	0	0	...	0	0	0	...	0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
Obj #31	2	2	2	2	2	...	3	3	3	...	2	2	2	...	0
Obj #32	0	0	0	0	0		0	0	0	...	0	0	<u>1</u>	...	0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
Obj #38	17	17	17	17	17	...	20	20	20	...	19	19	19	...	17
Obj #39	0	0	0	0	0		0	0	0	...	<u>1</u>	0	0	...	0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
Obj #44	0	2	0	2	0	...	2	0	<u>2</u>	...	0	2	0	...	0
Obj #45	<u>1</u>	14	2	10	<u>1</u>		17	<u>1</u>	14	...	2	10	<u>1</u>	...	<u>1</u>
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
Obj #67	3	3	3	3	3	...	3	3	3	...	3	3	3	...	3
AVERAGE	1.23	1.55	1.39	1.57	1.27	...	1.84	1.52	2.01	...	1.55	1.93	1.94	...	1.22

"..." presents omission of meaningless information.

Figure 13. FS ranking by memory map.

5.2. Results by Stack

The results of the stack method are as follows. Figure 14 shows the ranking of fault suspiciousness in all 44 parts. The ratio size when using the stack method had no influence on fault suspiciousness, as was the case when using the memory-map method. These results can be attributed to the following. The frequency with which FS partitions are determined in IDUR is related more to the fault. Therefore, the region with high FS is selected as a candidate region because it is frequently determined as a fault-suspicious one in IDUR ($1.74 \leq FS \leq 8.70$, $1 \leq \text{Rank} \leq 3$).

- Total number of partitions = 44
- Number of partitions in which memory update occurred = 26

Figure 14. FS ranking by stack.

5.3. Result Analysis & Validation

Finally, the reduction rate of fault candidates by each method is as follows. We can confirm that both methods have relatively high rates.

- Memory-map method
 - Size of the data section in which memory updates occurred = 10,786 bytes
 - Total size of the fault candidate region (top 18) = 1859 bytes
 - Reduction rate of fault candidates = $(1859 \div 10,786) \times 100 (\%) = 17.24\%$
- Stack method
 - Size of the data section in which memory updates occurred = 24,583 bytes
 - Total size of the fault candidate region (top 10) = 6202 bytes
 - Reduction rate of fault candidates = $(6202 \div 24,583) \times 100 (\%) = 25.23\%$

The fault candidate information refers to ranges of addresses in the memory. Thus, a fault candidate region has the list of symbols allocated to that address. In order to validate the results based on the symbol names for each fault candidate region, we use the symbol information of all memory addresses from each software developer. Based on this information, we confirmed the validity of the results in terms of the correlation between the input signal of the test script and the symbols in the fault candidate region, i.e., the similarity of names in Tables 3 and 4. This is because the cause of a fault is directly related to an input when the fault occurs during a test. Therefore, it is possible to find the cause of the fault by using the symbol information if the fault-candidate symbol is relevant to the input signal. The description of each column title in Tables 3 and 4 is as follows.

- **Input signal in test script:** this is an input signal in the inspection range entering the ECU during the execution of the test using the HiL simulation.
- **Symbol in fault candidate:** this is the symbol in a fault candidate region obtained in Figures 13 and 14, displayed by matching to related input signal.
- **Description of symbol:** this is the description of each symbol in a fault candidate region, and was confirmed to be faults in the test script through verification meetings with the software developers.

Table 3. Correlation between test input and fault candidate symbol based on memory map.

Number	Input Signal in Test Script	Object File	Symbol in Fault Candidate	Description of Symbol
1	SunRoof_SW.write	Obj #39	Struct_InterBuz .SunRoof	Switching the sunroof open state
2	AssistSeatBelt_SW.write		Struct_InterBuz .AssistSeatBeet	Switching the driver-door state
3	AlternatorState_SW.write		not found	
4	EPBlockLamp_SW.write	Obj #32	Struct_EPBlockStatus .EPBlockLamp	Switching the state of the electronic parking-brake lamp
...	...	...	...	...
15	Engine2_IGN.write		not found	
16	KeyState_SW.write	Obj #20	Struct_TailGate .KeyState	Switching the key-in state
17	VehicleSpeed_SW.write	Obj #44	Struct_VecleData .VehicleSpeed	Switching the vehicle-speed state

“...” presents omission of meaningless information.

Table 4. Correlation between test input and fault candidate symbol based on the stack.

Number	Input Signal in Test Script	Partition	Symbol in Fault Candidate	Description of Symbol
1	SunRoof_SW.write	Part #27	Signal_SunRoof	Switching the sunroof open state
2	AssistSeatBelt_SW.write		Signal_AssistSeatBelt	Switching the seatbelt-assist state
3	AlternatorStatus_SW.write		Signal_AlternatorStatus	Switching an alternator-lamp state
4	EPBlockLamp_SW.write		Signal_EPBlockLamp	Switching the electronic parking-brake lamp
...	...	...	...	...
15	Engine2_IGN.write	Part #16	Signal_IGN	Engine ignition
16	KeyState_SW.write		Signal_KeyState	Switching the key-in state
17	VehicleSpeed_SW.write	Part #29	Signal_VehicleSpeed	Switching the vehicle-speed state

“...” presents omission of meaningless information.

Table 3 shows the correlations between input signals and the symbols in fault candidate regions obtained by the memory map. These fault candidate regions contain the symbols related to 13 input signals of a total of 17 input signals. Table 4 shows the correlations between input signals and the symbols in fault candidate regions obtained by the stack. These fault candidate regions contain the symbols related to 16 input signals out of a total of 17 input signals. Of the fault candidate regions in Table 3, those with symbols actually related to input signals are objs. #20, #32, #39, and #44. This is because the symbols in regions other than these are only indirectly affected by the input signal. Therefore, these symbols could not be confirmed through name similarity alone. For the same reason, it was not possible to confirm everything by the stack.

As a result, we could find the symbols related to 76% of the input signals by the memory map, and the symbols related to 94% of those by the stack. As can be seen from the results with the other data set in Table 5, similar results were obtained (approximately 75% and 92% on average). However, as seen by comparing Tables 3 and 4, there are differences between the symbol names obtained by the memory map and those obtained by the stack. These differences are illustrated in Figure 15. An input signal applied by the HiL simulator is transmitted internally through the virtual bus to the structure indicating the state information via software logic. In addition, a fault candidate region identified by the memory map includes the symbols of the structure indicating the actual ECU state receiving the input signal; one identified by the stack includes the symbols of the external input signals delivered through the virtual bus. This difference is due to memory information being captured by the operating cycle of the operating system, so the stack information at each time point cannot be obtained via software. Accordingly, there is a limit to directly finding the cause of faults compared to the memory-map method due to losing the stack information. Therefore, both methods have different advantages and disadvantages, as shown in Table 5. The reduction ratio of fault candidates is the ratio of the size of fault candidates divided by the size of the updated memory. There is no significant difference between the reduction ratios of the two methods (22.43% and 19.21%). In addition, a fault candidate region identified by the stack includes symbols related to external ECU communication. However, a fault candidate region identified by the memory map includes symbols related to the internal ECU state. Therefore, the memory-map method is more useful for finding the fault location directly. However, a fault candidate region identified by the stack includes more symbols related to input signals than one identified by the memory map.

Figure 15. Differences of symbol types in fault candidates (memory map and stack).

In conclusion, since only indirect clues are available for finding the cause of a fault by the stack method, whereas it is possible to find the symbol directly causing the fault by the memory-map method, additional work is required to track the exact location of the symbol causing the fault. However, although the stack method requires further work, it can be applied independently of developer even in an environment without a memory map where HiL testing is performed.

Table 5. Comparative results of the two fault-localization methods.

Feature	Fault Localization Based on Memory Map					Fault Localization Based on the Stack				
	ID #74	ID #95	ID #96	ID #131	Avg.	ID #74	ID #95	ID #96	ID #131	Avg.
Reduction rate of fault candidates (=size of fault candidate region/size of updated memory) (%)	17.24	22.10	17.20	33.18	22.43	25.23	12.32	25.53	13.76	19.21
Number of symbols related to the input signal in the fault candidate/total number of input signals in the test script (%)	76.47	76.47	75	71.88	74.96	94.12	88.24	96.88	90.63	92.47
Characteristics of the symbols related to the input signal	Structure symbols referring to an internal ECU state					Symbols of the virtual bus signals for sharing data from the external ECU				

6. Conclusions

In this paper, we proposed a method to provide developers with a reduced number of fault candidate regions in the memory targeting of an automotive ECU and its software in the course of performing black-box tests using an HiL simulator. In the proposed method, the captured memory information was partitioned by a memory map and the stack based on the memory information captured during the execution of the automotive software. A fault candidate was then calculated by applying the updated information to the partitioned memory.

The method proposed in this paper has the following advantages. Firstly, it is possible to localize faults in a black-box environment in which the source code is unavailable. Previous fault-localizing methods for a black-box environment were inadequate, but it is now possible to generate the fault candidate information by using proposed method. Secondly, it is possible to localize faults even without the use of an external ECU debugging terminal for run/stop debugging. In general, the new method is necessary for environments in which these tools do not exist. This is because debugging is performed by observing the state of the memory according to the operation period of the system; however, the method for this is not yet adequate. Nevertheless, it is possible to obtain similar information to that from a debugger by using the method proposed in this paper. Thus, the method of fault localization without using the source code can contribute significantly to saving development time that is currently consumed by finding fault locations.

In the future, we will study the effects of combining the stack and memory-map methods. From the analyses of the two separate methods in the previous section, the memory-map one is now known to have the advantage of being able to find symbols indicating the internal state of the ECUs, whereas the stack one has the advantage of being able to find symbols for the virtual bus signals related to most of the input signals. By combining the two methods, it is expected that memory methods will become capable of finding more symbols that indicate the internal ECU state. At present, we search for faults using only the information about failed test cases. However, the reliability of our method would be improved by inspecting the update patterns of passed memory. Furthermore, we plan to study how the passing information using knowledge that can be acquired from the binary code could be applied to the present research results.

Acknowledgments: This research was supported by Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT & Future Planning (NRF-2014M3C4A7030504).

Author Contributions: J.-W.L. conceived and designed the experiments; K.K. performed the experiments; K.K. and K.-Y.C. analyzed the data; K.-Y.C. contributed analysis tools; J.-W.L. and K.K. wrote the paper; K.-Y.C. contributed to the revision of the paper.

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

References

1. Altinger, H.; Wotawa, F.; Schurius, M. Testing methods used in the automotive industry: Results from a survey. In Proceedings of the 2014 Workshop on Joining AcadeMiA and Industry Contributions to Test Automation and Model-Based Testing, San Jose, CA, USA, 21–25 July 2014; pp. 1–6.
2. Zhang, H.; Li, W.; Chen, W. Model-based hazard analysis method on automotive programmable electronic system. In Proceedings of the 3rd International Conference on Biomedical Engineering and Informatics (BMEI), Yantai, China, 16–18 October 2010; pp. 2658–2661.
3. Tatar, M.; Mauss, J. Systematic test and validation of complex embedded system. In Proceedings of the Embedded Real Time Software and Systems (ERTS) 2014, Toulouse, France, 5–7 February 2014.
4. Wong, W.E.; Gao, R.; Li, Y.; Abreu, R.; Wotawa, F. A survey of software fault localization. *IEEE Trans. Softw. Eng.* **2016**, *42*, 707–740. [[CrossRef](#)]
5. Siegl, S.; Hielscher, K.-S.; German, R. Model based requirements analysis and testing of automotive systems with timed usage models. In Proceedings of the 18th IEEE International Requirements Engineering Conference (RE), Sydney, Australia, 27 September–1 October 2010; pp. 345–350.
6. Nidhra, S.; Dondeti, J. Black box and white box testing techniques—A literature review. *Int. J. Embed. Syst. Appl.* **2012**, *2*, 29–50. [[CrossRef](#)]
7. Choi, K.-Y.; Seo, J.; Jang, S.; Lee, J.-W. HiL testing based fault localization method using memory update frequency. In *Advances in Computer Science and Ubiquitous Computing*; Springer Singapore: Singapore, 2015; pp. 765–772.
8. Wong, W.E.; Horgan, J.R.; London, S.; Mathur, A.P. Effect of test set minimization on fault detection effectiveness. In Proceedings of the 17th international conference on Software engineering (ICSE), Seattle, WA, USA, 24–28 April 1995; pp. 41–50.
9. Piguet, C.; Masgonty, J.-M.; Arm, C.; Durand, S.; Schneider, T.; Rampogna, F.; Scarnera, C.; Iseli, C.; Bardyn, J.-P.; Pache, R.; et al. Low-power design of 8-b embedded CoolRisc microcontroller cores. *IEEE J. Solid State Circuits* **1997**, *32*, 1067–1078. [[CrossRef](#)]
10. Padmanabhuni, B.; Tan, H.B.K. Techniques for defending from buffer overflow vulnerability security exploits. *IEEE Internet Comput.* **2011**. [[CrossRef](#)]
11. Wong, W.E.; Debroy, V. Software fault localization. *Encycl. Softw. Eng.* **2010**, *1*, 1147.
12. Baah, G.K.; Podgurski, A.; Harrold, M.J. Mitigating the confounding effects of program dependences for effective fault localization. In Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, Szeged, Hungary, 5–9 September 2011; pp. 146–156.
13. Nguyen, H.; Shen, Z.; Tan, Y.; Gu, X. FChain: Toward black-box online fault localization for cloud systems. In Proceedings of the 33rd International Conference on Distributed Computing Systems (ICDCS), Philadelphia, PA, USA, 8–11 July 2013.
14. Vermeulen, B. Functional debug techniques for embedded systems. *IEEE Des. Test Comput.* **2008**, *25*, 208–215. [[CrossRef](#)]
15. Caliebe, P.; Lauer, C.; German, R. Flexible integration testing of automotive ECUs by combining AUTOSAR and XCP. In Proceedings of the 2011 IEEE International Conference on Computer Applications and Industrial Electronics (ICCAIE), Penang, Malaysia, 4–7 December 2011; pp. 67–72.
16. Hillenbrand, M.; Müller-Glaser, K.D. An approach to supply simulations of the functional environment of ECUs for hardware-in-the-loop test systems based on EE-architectures conform to AUTOSAR. In Proceedings of the 20th IEEE/IFIP International Symposium on Rapid System Prototyping, Paris, France, 23–26 June 2009; pp. 188–195.
17. Bruckmann, H.; Strenkert, J.; Keller, U.; Wiesner, B.; Junghanns, A. Model-based development of a dual-clutch transmission using rapid prototyping and SiL. In Proceedings of the 2009 International VDI Congress Transmissions in Vehicles, Friedrichshafen, Germany, 30 June–1 July 2009.
18. Chrisofakis, E.; Junghanns, A.; Kehrer, C.; Rink, A. Simulation-based development of automotive control software with Modelica. In Proceedings of the 8th International Modelica Conference, Dresden, Germany, 20–22 March 2011; pp. 20–22.
19. Kim, I.; Al-Hilo, A.; Jang, H.S.; Yoo, J.-G. Conformance testing of SGSF-064-1 using CANoe. *Int. J. Appl. Sci.* **2015**, *5*, 1086–1101. [[CrossRef](#)]

20. Lee, J.-W.; Choi, K.-Y.; Jang, S.; Lee, J.-W. Data cascading method for the large automotive data acquisition beyond the CAN bandwidth in HiL testing. In *Advances in Computer Science and Ubiquitous Computing*; Springer Singapore: Singapore, 2015; pp. 773–780.
21. Wu, R.; Zhang, H.; Cheung, S.-C.; Kim, S. CrashLocator: Locating crashing faults based on crash stacks. In Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA), San Jose, CA, USA, 21–26 July 2014; pp. 204–214.
22. Egan, M.H.; McDonald, C. Program visualization and explanation for novice C programmers. In Proceedings of the 16th Australasian Computing Education Conference (ACE), Auckland, New Zealand, 20–23 January 2014; pp. 51–57.
23. Jeffrey, D.; Nagarajan, V.; Gupta, R. Execution suppression: An automated iterative technique for locating memory errors. *Trans. Program. Lang. Syst.* **2010**, *32*. [[CrossRef](#)]
24. Harris, T.L.; Fraser, K.; Pratt, I.A. A practical multi-word compare-and-swap operation. *Distrib. Comput.* **2002**, *2508*, 265–279.
25. GNU Binary Utilities. Available online: <https://sourceware.org/binutils/docs/binutils/index.html> (accessed on 2 July 2016).

© 2016 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/>).