Journal of Computer and Knowledge Engineering, Vol. 2, No. 1, 2019.
DOI: 10.22067/cke.v2i1.74607

A State-aware Approach for Robustness Testing of Embedded Real-Time Operating Systems

Raheleh Shahpasand[1] Samad Paydar[2*] Yasser Sedaghat[3] Reza Ramezani[4]

Abstract: The Operating System (OS) is a major part of embedded software systems and its robustness has considerable influence on the robustness of the entire system. Thus, its robustness testing is critical for assessing the dependability of the system. In this paper, a state-aware approach is proposed to evaluate the robustness of components of embedded real-time OSs in the presence of different types of faulty inputs. This approach leads to identifying critical OS states, their criticality level, and the maximum and minimum level of the OS robustness. It also facilitates comparing the robustness level of OS's components and helps the system developers to select the most appropriate fault tolerance techniques by considering the robustness level and timing limitations. The experimental results demonstrate the ability of the proposed approach in providing more information about the robustness vulnerabilities in the states of the system.

Keywords: Robustness testing, Embedded operating system, Robustness level assessment, Safety-critical systems, Fault injection.

1. Introduction

With the increasing growth in the use of embedded systems in different applications, the importance of verifying the correct behavior of these systems under different possible conditions has increased. The software part of these systems has the responsibility of controlling the functionality of the system. Operating System (OS) is an important part of an embedded system that manages the operations of the embedded system and has significant impacts on its functionality. Thus, the guaranteed correct functionality of an embedded system highly depends on the correct behavior of its underlying OS [1]. This issue is more crucial in safety-critical applications, since their failure results in destroying lives and significant properties or environmental damage [2].

The principal role of embedded software systems is interaction with the physical world. Thus, they are reactive and should respond within a predefined time period specified by their real-time constraints [3]. The increasing complexity of embedded systems leads to the increase of the OS's functional complexity, which increases the size of the OS's source code in terms of Lines of Code (LOC). By increasing the source code size, the residual software defects raises as well [4]. The increase of software defects has become a major concern in software systems, especially those employed in safety-critical applications. Since these OSs are usually used outside of their original context and interact with an external environment, they are prone to more faults [4]. Consequently, fault tolerance techniques are necessary to assure the correct behavior of the OS's functionalities [1].

Robustness testing is used to evaluate the systems' fault tolerance [5]. In particular, OS robustness testing assesses the OS behavior in the presence of faulty inputs to detect the vulnerabilities that affect the correct behavior of the OS [6]. Faulty inputs fall into four categories [7-9]: 1) Invalid and unexpected value, 2) invalid timing of an input, 3) invalid input sequence, and 4) incorrect input format. In robustness testing of an OS, the OS's interfaces are deliberately exposed to faulty inputs through software-implemented fault injection techniques (SWIFI) which are widely used for robustness testing [4, 5]. The OS interfaces for robustness testing include the application programming interface (API) and device drivers [10].

Since the interactions of embedded OSs with the execution environment are not fully predictable at the development phase, the robustness testing of such OSs is challenging [5]. Embedded OSs encounter all the aforementioned faulty input types. The system response depends on the features of the faulty inputs and the state of the OS components, which is determined by their properties. A component is a part of the OS that is responsible for managing specific resources or providing a set of services, such as memory management and process scheduling [11]. The OS components interact with each other and their properties may change during their interaction. The OS state is obtained by analyzing the interactions between OS components. The robustness testing of OSs can be improved by taking both faulty inputs and the OS state into account [10].

This paper proposes a state-aware approach for robustness testing of embedded OSs. The proposed approach takes different types of faulty inputs and the OS state into account and thus it has the potential to reveal how critical these states are for system dependability. In this approach, first, important states of the OS components are identified in the form of a behavioral model and then, different types of faulty inputs are injected into these states. Using the fault injection results, the criticality value of each state and then the robustness level of the components are determined. These values are used to get more information about the robust

Manuscript received August, 5, 2018; accepted. February , 22, 2019.
[1] M.Sc. Department of Computer Engineering, Ferdowsi University of Mashhad, Mashhad, Iran.
[2*] Assistant Professor, Department of Computer Engineering, Ferdowsi University of Mashhad. Email: s-paydar@um.ac.ir
[3] Assistant Professor, Department of Computer Engineering, Ferdowsi University of Mashhad.
[4] Assistant Professor, Faculty of Computer Engineering, University of Isfahan, Isfahan, Iran

behavior of the OS. The proposed approach gives helpful information about the OS robustness to developers, including the critical OS states, the criticality level of the OS components, and the maximum and minimum level of the OS robustness. With this information, developers can accurately select an appropriate fault tolerance technique to improve the system dependability. Furthermore, the proposed approach can be effectively used to compare the robustness of different OS components.

The organized of the rest of the paper is as follow: In Section 2, the most important related studies are presented. The proposed approach is elaborated in Section 3. In Section 4, the results of applying the proposed approach to a case study are presented and discussed. Finally, the paper concludes in Section 5 and offers some future work.

2. Related Studies

Due to the fundamental role of OSs in computer systems, OS robustness testing has attracted the interest of researchers for many years [5]. Fuzz [12] is one of the first studies in this area that has used random inputs for robustness testing of OSs. These inputs are injected into system's user interface which stochastically activate the robustness faults [6]. Four series of Fuzz experiments [12-15], that have been conducted in the years 1990-2006 on UNIX, Windows NT and MacOS operating systems, have shown the effectiveness of random inputs in OS robustness testing. These experiments have revealed that OSs, as a mature software, are still vulnerable to faulty inputs even against random ones. Using random inputs in robustness testing has some drawbacks. For example, fault activation relies on chance and the test space is extremely large. Robustness testing based on the type of interface parameters is an attempt to overcome such drawbacks [6].

The Ballista [16, 17] uses a type-specific approach to test and benchmark the OS. Each robustness testing scenario consists of a system call invocation with a combination of both valid and invalid values for an input parameter. These values are predefined for each data type that, compared to the random faulty inputs, lead to a smaller test space. Ballista has succeeded in finding severe robustness vulnerabilities in several commercial OSs, but the number of test cases is still high.

Some studies have focused on the OS robustness testing with respect to device driver interface. In case of robustness testing of device drivers, a profiling framework has been proposed by [18] that assists in finding possible error propagation paths from device drivers to the applications. Similarly, the presented work by [19] concerns OS robustness testing regarding device driver interface and focuses on testing the Driver Programming Interface (DPI). DPI is a set of kernel core functions that implements the interactions way between device drivers and the kernel. In order to characterize the robustness of OSs, the faults are injected into the parameters of these core kernel functions. The results show the negative impacts of faulty drivers on the responsiveness of the kernel, safety of the workload and availability of the kernel.

In recent years, the OS robustness testing using OS states as well as invalid inputs have attracted the attention of many researchers. The OS state has a considerable influence on the OS robustness testing. The execution of a given robustness test case in different OS states would generate rare execution patterns which, as a result, increases the final coverage of robustness testing [10]. In this regard, Sârbu and others [20] have proposed a state model for testing device drivers. This state model has been derived from run-time communications among device driver interfaces. This study reports that the use of a state model reduces the number of test cases.

Johansson and others [21] have introduced the concept of call blocks to take into account the state of the OS in robustness testing. In this approach, the usage profile of a device driver is split into disjoint call blocks. Call blocks, that are recurring sequences of function calls, lead to injecting faults into different system states. The results have shown that controlling the time of fault injection has a significant impact on the robustness evaluation.

Similarly, the approach presented in [10] has the goal of enhancing the traditional approaches by considering the OS state in test case definition. By this approach, a test plan is expressed through two dimensions: the exceptional inputs and the OS states. Exceptional inputs are selected from a set of predefined invalid values. The states vary in $S = \{s_1, s_2 \ldots s_n\}$, where s_i is a set of component attribute values. In order to execute a test case, state setter takes component to one of the predefined states in S. Then, test driver injects invalid inputs to the component interface. This study has demonstrated that the robustness tests are able to reach corner cases with complex interactions with other subsystems, which cannot be covered by traditional robustness testing methods [10].

SABRINE [11] is an extension of the approach proposed by [10]. The authors have claimed that SABRINE is the first approach that applies behavioral model mining techniques in order to test the robustness of the OS. In the first phase of SABRINE, behavioral data about OS is collected, in terms of interactions between OS components at run-time. At the next phase, the behavioral data are preprocessed and are divided into disjoint sequences. Identical sequences represent a pattern. To have an efficient set of test cases, in the third phase, the patterns are further grouped using a clustering algorithm. In the fourth phase, a behavioral model is generated for every cluster in the form of finite state automata (FSA) in which the states are interconnected by events. Furthermore, injectable transitions are identified. An injectable transition is an invocation of a function in which a fault can be injected. Only one test case is generated for every injectable transition. In order to execute test cases, in the fifth phase, the system is transitioned to the initial state of the behavioral model and a fault is injected when the OS reaches the intended state.

SABRINE approach has overcome the limitation of requiring knowledge about OS internals. Nevertheless, this approach suffers from some drawbacks like neglecting different types of faulty inputs. In an attempt to emulate realistic scenarios, the proposed methodology in [22], deals with four different types of programmable faults, including: data, protocol, time-related, and state-related faults. This methodology has employed model-based robustness testing for embedded software. The results of applying this methodology have indicated that the robustness testing

method is very effective in finding vulnerabilities. The key problem of this approach is that the state model of the system is not based on the system behavioral model, but it is extracted using explicit abstractions of the system and its environment.

The presented approach in [23], TrEKer, infers error propagation from a faulty kernel component to other parts of the kernel by tracing memory accesses using compile-time instrumentation. This approach infers error propagation from deviations in the injection target's state and behavior that are visible to other parts of the kernel. The data that the system under test (SUT) operates on, has defined as SUT state. The evaluations have demonstrated that conventional oracles would misclassify up to 10 % of seemingly successful runs.

Whilst numerous researches have been carried out on embedded OS robustness testing, none of them has adequately covered the robustness testing with respect to different states and different types of faulty inputs. Furthermore, they have not considered specific characteristics of these OSs such as timeliness and reactiveness. In our previous work [24], a state-based approach for testing the robustness of embedded real-time OSs has been proposed which investigates the impact of inputs with invalid timings. In [24] the behavioral model has revealed the critical states in respect of timing delays and has not considered other types of faulty input. Some studies [22, 25] have attempted to address robustness testing with respect to different types of faulty inputs, but similarly they have not taken into account the impact of OS state in robustness testing.

To the best of our knowledge, there is no study that has employed the OS behavioral model after fault injection to build the system's behavioral profile in confronting with faulty inputs in different OS states. The aim of this study is to overcome the aforementioned limitations by improving the OS behavioral model in order to handle different types of faulty inputs and to enrich as well the model based on the fault injection results.

3. The Proposed Approach

The proposed approach consists of three main steps: behavioral modeling, fault injection, and robustness level assessment. In the first step, the Component under Test (CuT) is monitored to obtain its behavioral model. For this purpose, the SABRINE approach is enriched with some improved features to deal with different types of faulty inputs. The extracted behavioral model is used in the second step to produce and apply the fault injection test cases. Finally, the test cases are executed, and the results of fault injection experiments are exploited to augment the behavioral model and provide further information about the component's robustness. In following, the steps of the proposed approach are described.

3.1. Behavioral Modeling

In this step, the behavior of the CuT is modeled. An OS is composed of a set of components, each of which is responsible for performing one of the OS functionalities. For example, the memory management component is responsible for handling the access of different processes to the physical memory. An OS provides the services through its interfaces and the processes request these services using system calls. When a system call is invoked, one or more OS components interact with each other to provide the requested service [11]. Each component has an interface to be used by other components to invoke the component services through function calls.

In the proposed approach, in order to make the fault injection and robustness level assessment techniques more effective, first, a model of the interactions between components is created to identify the appropriate points where the faults should be injected. This step itself is divided into three phases:

Phase 1. Behavioral Data Collection: In this phase, the software system is run and, using a workload, profiled under fault-free conditions. Workload is a graph of tasks, each of which invokes an OS service. Thus, a workload causes some kernel calls and interactions between the OS components. During the execution of the workload, data about the interactions of the target component, which its robustness is supposed to be assessed, are collected as behavioral data. This data is then used to model the behavior of the system.

The interactions among components along with their details such as the ID of the operation requested by the workload, the name of the kernel functions that have been invoked, and the values of the functions' input parameters, are logged. In addition, the start and finish times of interactions are recorded in the log file. In the experiments, by giving the highest priority to the workload, the execution time of the workload will not include interrupts execution or the OS scheduling time.

The detailed information logged during the workload execution can then be used to create a sequence of interactions. However, some factors such as different execution paths in kernel functions would affect the sequence of interactions. Thus, at this phase, the execution of the workload is repeated several times. Every execution of this phase produces an individual log file. The log files are then processed to extract the recurring patterns of interactions.

Phase 2. Pattern Identification and Clustering: Since the functionality of the OS kernel should be assessed in general, independently of a particular workload, a sequence of interactions is defined as the set of interactions that have been happened during the execution of an individual kernel function call (not the system calls or interrupt services which are requested by the workload). Due to different execution paths in kernel functions, two executions of a particular kernel function call will not necessarily lead to identical sequences. Thus, the sequences of a kernel function would generate different patterns. In this phase, these patterns are identified, using a clustering technique. In this regard, the similarity of each pair of patterns is quantified using the spectral clustering algorithm [11]. This algorithm groups a set of elements based on their similarity, and hence, it can group more similar patterns in the clusters. Finally, infrequent patterns are ignored.

Phase 3. Generating the Behavioral Model: The relative start and finish times of the interactions are recorded in the first phase. Thus, in this phase, for each cluster identified in

the second phase, a behavioral model is created in the form of a Timed Automata [26]. A behavioral model is a directed graph in which the nodes represent the states of the corresponding component and the edges represent the transitions of the component to new states. Transitions are caused by the interactions with other components.

A cluster consists of one or more recurring patterns. If a cluster has more than one pattern, the behavioral model is augmented with new edges to include all patterns. Therefore, when a particular cluster has different patterns, its corresponding behavioral model will have more than one edge in some states. This process is repeated to generate a behavioral model for each cluster.

For example, the behavioral model represented in Figure 1 shows two different patterns. Each path denotes a pattern of interactions. The edges in the graph are labeled with the interaction name, which is composed of the name of the invoked service along with its relative time of occurrence. As mentioned before, the state of a component is determined by its properties. Thus, the properties of the component may change in a new state. For example, when the write system call is invoked, an interaction is occurred and the state of the memory management component (which is the amount of available and used memory) might be changed.

3.2. Fault Injection

Although the proposed approach has the potential of considering different types of faulty inputs, in this paper only the data and timing faults were taken into account. As mentioned before, the goal of robustness testing is to evaluate the impacts of faulty inputs on the function responses and to assess its robust behavior. Thus, for each faulty input type, injectable interactions in the behavioral model are identified through the analysis of the invoked function. Then, for each injectable interaction, a test or a set of test cases are generated by the procedure described below. Finally, the test cases are used in the fault injection experiments.

Without losing generality, in this paper the test cases are generated from these two perspectives:

A. Timing Faults: Injectable interactions for timing faults are those interactions whose input parameters value influence the execution path. For example, if an input parameter value affects a loop control or conditional statement, it will affect the execution path. Therefore, the lines of code that contain these statements are considered as candidate lines for injecting timing faults. An injected timing fault actually simulates the real world conditions that may appear due to unexpected change in the input parameter value. For example, a Single-Event Upset (SEU) flips a memory bit and would cause a data error [27]. If such faults appear in the injectable line, the execution path and consequently the execution time may change. It is noteworthy to remark that changing the program execution path does not necessarily increase the execution time. Thus, in this paper, just deadline misses caused by the increased execution time are paid attention. For this purpose, delays are injected in functions using the timing faults generated by the following method.

Test Case Generation for Timing Faults. In order to generate timing error test cases, a binary search-based method is used based on the function call deadline. The deadline of a function is defined as the specified time constraint that the function guarantees to response within. If x is the relative deadline value of a particular function, the range of possible delays is [1, x]. The first test case is intended to cause a delay of $\frac{x}{2}$. The fault injection method translates this test case to a statement in the injectable line of the function source code that causes a delay of $\frac{x}{2}$ time unit. Then, based on the results of this fault injection experiment, a new range is identified for timing fault injection. If the deadline is missed, a new test case is designed to impose a delay in the range of $[1, \frac{x}{2})$. Otherwise, a test case is generated for injecting a delay in the range of $(\frac{x}{2}, x]$. This process continues until the maximum deadline violation threshold that does not miss the function deadline is found. Therefore, the result of timing fault injection experiments is the deadline violation threshold of each injectable transition.

B. Value Faults. For faulty inputs with invalid value, injectable interactions are those interactions which use the input parameter. There are three error models that are usually employed for evaluating the OS behavior in the presence of invalid input parameter values: bit-flip, data type, and fuzzing [9, 21, 28]. In this paper, the data type error model is selected, because it has the shortest execution overhead compared to the bit-flip and fuzzing error models [28]. In addition, the data type error model has a fair injection efficiency [29] and it is the representative of the actual OS errors [18].

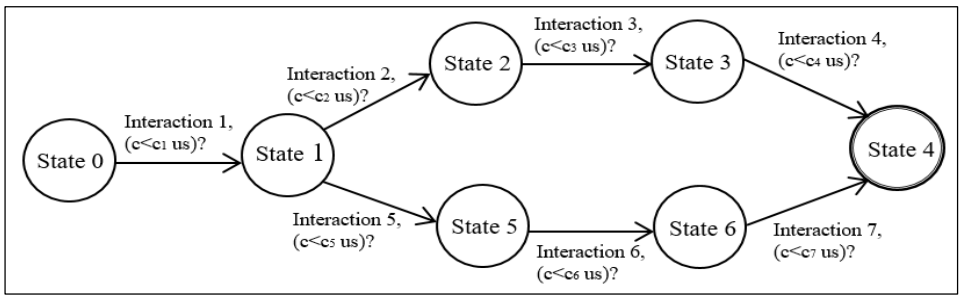

Fig. 1 Example of a behavioral model

C. Test Case Generation for Value Faults. For value faults, test cases are generated based on a test methodology like Ballista [16]. One advantage of Ballista is that the result of fault injection is highly repeatable. In this approach, the CuT is transited to a given initial state and its services are invoked with faulty parameter values. As mentioned before, faulty parameter values are defined based on the data type of the input parameter. For example, if the type of the input parameter is integer, the possible test cases will be 0, -1, INT_MAX, and INT_MIN. The CuT response to the service invocation with faulty parameter value is monitored to identify the probable robustness failures. Robustness failures are measured using CRASH scale [16].

Generating fault injection test cases, in order to perform a fault injection experiment, a workload is executed and the interactions of the target component are monitored like the first step. This monitoring is performed to track the states of the component. Thus, the faults are injected when the component reaches a specific state. The faults are then injected into the interactions between components by a fault injector. This means that the fault injector intercepts an interaction, replaces its parameter with faulty ones, and then gives the interaction back to the invoked component. When the faults are injected, the workload, the CuT and the OS status are monitored by the failure detector component to identify the type of any potential failure.

3.3. Robustness Level Assessment

The fault injection results are used to augment the behavioral model created in the first step. For this purpose, the robustness score of every transition is computed. The idea is that more robust paths should get higher scores while lower scores should be assigned to the less robust paths. In order to compute the score of each path, first the score of the transitions is computed. For each transition of the generated behavioral model, the score is computed based on the results of both timing and value fault injection experiments. The transitions are the target of fault injection, because they represent the function calls caused by a service request. Therefore, it can be investigated how a faulty transition can take the CuT to a faulty state.

In the value fault injection, the number of test cases can be different for each injectable transition and depends on the number of input parameters of the interaction. Therefore, the results of the value fault injection should be normalized to fairly compare the results. For this purpose, each class of failure is weighted, such that the lowest weight is assigned to the most severe failure, whereas the highest weight is assigned to the gentle failure. For example, the failure classes of CRASH scale are scored as shown in Table 1.

Table 1. The Score of each Failure Class in CRASH Scale

CRASH Failure Class	Weight
Catastrophic	0
Restart	0.2
Abort	0.4
Silent	0.6
Hindering	0.8
No Failure (Robust)	1

It can be seen from Table 1 that the weight of a catastrophic failure, which is the most severe one, is zero ($w_C = 0$) and the weight of a robust response is one ($w_R = 1$). The weight of other classes is increased with a fixed rate. Once the value fault injection results are weighted, the average is calculated for each injectable transition based on the results of the value fault injection.

In order to compute the scores based on the results of timing fault injection, which are deadline violation thresholds, the behavioral model's deadline is divided into six intervals (similar to the number of failure classes in the value fault injection) in such a way that each interval has one score. The largest deadline threshold gets the highest score and the smallest one gets the lowest score. Thus, the scores of transitions are obtained based on their deadline violation threshold. After obtaining the scores of timing and value fault injections, the robustness of the transition (R_t), is calculated as follows:

Definition 1: For a given transition t, let TS_t denote the robustness score of t in timing fault injection and VS_t is the robustness score of t in value fault injection. The robustness score of t, denoted by R_t, is obtained as $(TS_t + VS_t) / 2$.
Once the robustness of every transition is calculated, the robustness of each path of the behavioral model is computed.

Definition 2: Let P denotes a given path in the behavioral model and f(P) represents the probability of passing P. Thus, the robustness of P is obtained by:

$$R_P = f(P) \sum R_{t_i} \quad (1)$$

where R_{t_i} denotes the robustness scores of transition i in P. R_P quantifies the robustness score of P using the robustness scores of its transitions. Finally, the maximum and minimum robustness level of the CuT are obtained by comparing the robustness of different paths in its behavioral model. The maximum and minimum R_P among all paths in the behavioral model of CuT represent the maximum and minimum robustness level of the CuT, respectively.

4. Evaluation

In order to evaluate the proposed approach, it has been implemented to perform robustness testing on Linux PREEMPT-RT v3.14.3-rt4, which is a real-time OS used in embedded systems [30]. In the experiments, the memory management component was selected as the CuT. Furthermore, Mibench [31], which is a representative benchmark for embedded programs, was employed as the workload. In this benchmark, automotive category of Mibench is intended for safety-critical applications. Qsort, which is in the automotive category was executed on the underlying OS, as the workload.

In order to record the interactions between the memory management component and other components, SystemTap [32] is utilized. SystemTap is also responsible for storing the recorded information as a log file. It uses a dynamic method for monitoring and tracing the operations of the running Linux kernel. Thus, it makes it possible to investigate the behavior of the kernel. SystemTap has a low overhead when

it monitors or instruments the kernel operations [32]. As mentioned in [11], there are some tools with the same functionality as SystemTap for most modern OSs. Hence, the experiments discussed in this section can be ported to other OSs.

Fig. 2 shows the behavioral model of generic_ file_aio_read kernel function, as a timed automata. This behavioral model has been created by the proposed approach during the experimental evaluations. The role of the generic_file_aio_read kernel function is to implement both synchronous and asynchronous read operations and it is invoked when a system call for reading the memory occurs. As Fig. 2 shows, clock c is reset at State 0. State 1 represents the CuT state before calling the generic_segment_checks kernel function. When the execution of generic_segment_checks finishes, the CuT is transitioned to State 2.

In this model, clock c specifies the deadline of each function call in microseconds. Since there is no predefined deadline for the Qsort benchmark, the worst execution time of every interaction, which is the execution time of the kernel function, was considered as its deadline. The deadline of each interaction was obtained in the first step of the proposed approach. Starting from State 0, clock c is increased in microsecond. The label (c<39us)? shows that the deadline of generic_segment_checks is 39us, and the label (c<86us)?, which belongs to the find_get_page kernel function, means that the deadline of this function from the initial state is 86us.

In Fig. 2, States 0 to 4 are connected by unique interactions. As this figure shows, there exist two different paths between States 4 to 7. Thus, it contains two patterns.

The paths and the injectable transitions of this behavioral model have been marked in Fig. 3. The injectable transitions of timing faults and value faults are identified based on the definition of an injectable transition in the second step of the proposed approach. For example, file_read_actor kernel function, which transitions the CuT from State 5 to State 6, has an input parameter (*desc) that affects a conditional statement and influences the execution path. Therefore, file_read_actor is an injectable interaction of timing faults. Because it has input parameters, file_read_actor is also an injectable interaction of value faults. In this experiment, injectable transitions of timing faults and value faults are the same, but they can be different for each fault type.

To the best of our knowledge, there is not a similar study which considers the stateful robustness testing in OSs in the presence of different types of faulty inputs. Therefore, as the goal of this paper is to demonstrate the effects of using fault injection results on increasing the robustness of components based on their behavioral model, the results and effects of timing and value fault injection experiments are presented and evaluated separately.

4.1. Timing Fault Injection

In order to generate timing test cases, first, the candidate lines for injecting delays are detected. Then, one of them is randomly selected and the intended delays are injected using the binary search-based method. The result of this experiment is the deadline violation threshold of injectable transitions.

Since the proposed approach considers the OS state, it is expected that the result of fault injection in put_page 1 (put_page function call in path No. 1) differs from that of put_page 2 (put_page function call in path No. 2). The impacts of applying timing fault injection to put_page 1 and put_page 2 have been depicted in Fig. 4 and Fig. 5, respectively. In these figures, the horizontal axis shows the amount of delays injected into the function source code and the vertical axis shows the execution time. Fig. 4 demonstrates the effects of timing fault injections of put_page 1 on deadline violation of generic_file_aio_read. Similarly, Fig. 5 shows the same results for put_page 2.

Based on the values recorded in the log file, the worst execution time of the generic_file_aio_read function is 266us. This time value is considered as the deadline of the generic_file_aio_read function and has been shown by a red dashed horizontal line in Fig. 4 and 5. As it can be seen from these figures, the generic_file_aio_read deadline has been missed by injecting delays more than 213us into put_page 1 and 61us into put_page 2. As a result, in some cases, the deadline violation in one of the interactions like put_page 1 or put_page 2 does not necessarily lead to deadline violation of the generic_file_aio_read function, because the deadline of this function is sufficient to tolerate some delay imposed to the functions that it interacts with. Let us bear in mind that the deadline of put_page function is 6us.

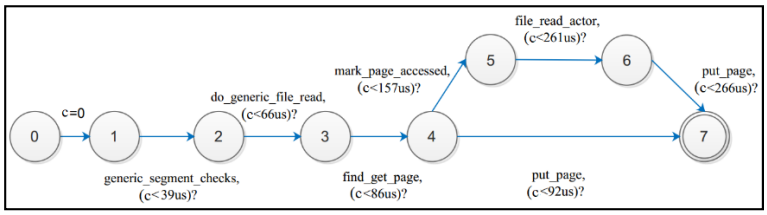

Fig. 1. *Behavioral model of generic_file_aio_read*

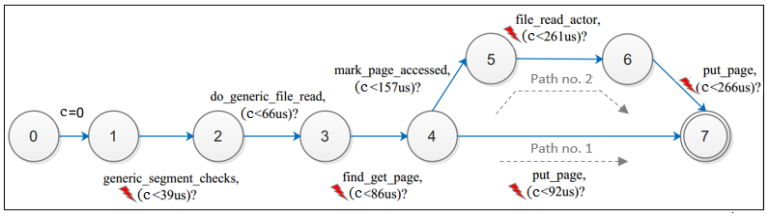

Fig. 3. The paths and injectable transitions in the behavioral model of generic_file_aio_read

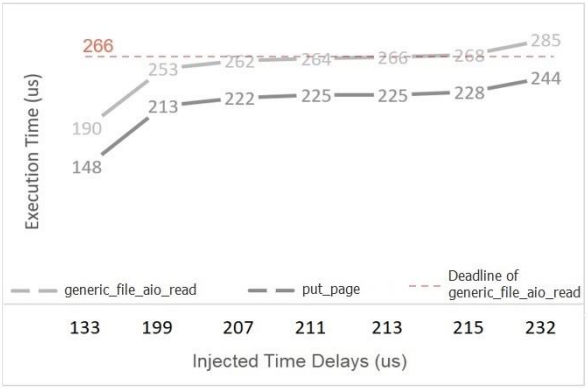

Fig.4. Impacts of timing fault injection on *put_page 1*

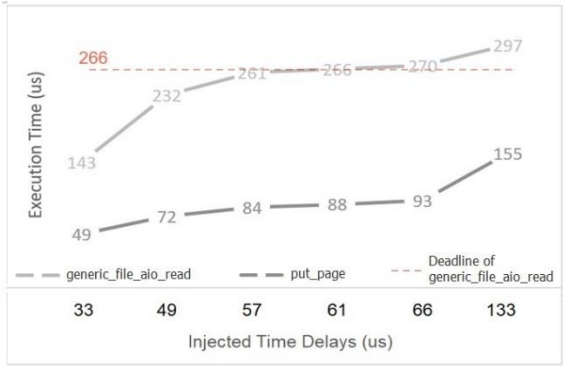

Fig.5. Impacts of timing fault injection on *put_page 2*

As shown in Fig. 5, another observation of this experiment is that, compared to put_page 1, the fault injection in put_page 2 has more impact on the execution time of the generic_file_aio_read function. Actually, the tolerable delay –an injected delay which does not lead to the deadline violation– of put_page 2 is 61us whereas it is 213us for put_page 1. More transitions in path No. 2 can increase the execution time of generic_file_aio_read. Thus, by considering the deadline of generic_file_aio_read function (which is 266us and is independent of the passed path), increasing the number of transitions in path No. 2 leads to decreasing the tolerable delay of put_page 2. It also causes fault injection in put_page 2 increases the execution time of generic_file_aio_read with a higher rate. In contrast, as Fig. 4 shows, the execution time of generic_file_aio_read increases with a constant rate with respect to the execution time of put_page 1.

The results of applying the proposed timing fault injection method to 5 out of 8 transitions of Fig. 3, which are injectable transitions, have been summarized in Table 2. This table shows the number of injectable lines in injectable transitions and their deadline violation thresholds in us. Such results help developers to identify the critical OS states in the presence of timing faults. For example, in this experiment, one can conclude that put_page 2 has a low robustness level, since it has more impact on deadline violation of generic_file_aio_read function. Thus, some low-cost fault tolerance techniques are required to increase its robustness.

Table 2. Deadline violation thresholds obtained by the proposed approachand the random approach

	Transition	Number of injectable lines	Deadline violation threshold (us)
Proposed approach	generic_segment_checks	3	76
	find_get_page	5	99
	file_read_actor	2	99
	put_page 1	2	213
	put_page 2		61
Random approach	-	18	90

To the best of our knowledge, no study has been conducted to compare it with the proposed timing fault injection method. Hence, in order to evaluate the efficiency of the proposed approach in assessing the robustness level of the OS's components, it was compared with a random timing fault injection method. In the random fault injection method, there is not any behavioral model to identify the execution paths. Thus, the timing faults are injected in timing injectable lines of the generic_file_aio_read function and the functions it calls. In the random approach, one of the identified injectable lines is selected randomly and the delays are injected into it. Moreover, in order to specify the deadline violation threshold, the binary search-based method is used. In this approach, the deadline violation threshold of generic_file_aio_read is 90us.

In the experiments, six timing injectable lines have been identified in the source code of the generic_file_aio_read function and the total number of identified injectable lines in the functions which are called by the generic_file_aio_read function is 12. Thus, the random approach identified 18 injectable lines for timing fault injection. In our proposed approach, this number depends on the existing paths of the behavioral model and varies from 10 to 12. Thus, in this experiment, the number of injectable lines effectively decreases about 33.3% to 44.4%. This leads to a reduction in the number of test cases for finding the deadline violation threshold using the binary search-based method.

As this Table 2 shows, the proposed approach has a fine-grain view and assesses the criticality of injectable interactions separately. Thus, developers can use lower-cost yet content aware fault tolerance techniques for such small components of the OS. On the other hand, the random approach does not consider interactions and just examines the function and its direct function calls. Thus, the assessment and improvement of CuT's robustness based on its behavior model is not possible. For example, according to the proposed approach, put_page 2 with 61us deadline threshold is the most critical transition in the presence of delays, whereas the random approach provides deadline violation threshold just for the entire generic_file_aio_read function.

4.2. Value Fault Injection

In the proposed approach, the test cases of value fault injection are generated for every value injectable transition by extending the Ballista test methodology which is the most well-known technique in OS robustness testing [33]. In our

approach, different states of the CuT are considered in the value fault injection whereas Ballista just considers a given initial state to execute all test cases. Thus, in order to execute the test cases in our proposed approach, the CuT is transited to the intended state and then a value fault is injected. The observed failures of both approaches are presented in Table 3.

In this experiment, the observed failures are different in our proposed approach when the CuT is in different states during the fault injections. For this experiment, a unique test case of value fault injection was used for put_page 1 and put_page 2. In the obtained results, there is one catastrophic failure in value fault injection of put_page 1 which indicates that put_page 1 is more exposed to a robustness failure. The reason is that put_page 1 and put_page 2 are in different execution paths. The put_page function checks some conditions, which depend on the state of the CuT. Thus, the response of this function to a unique input value can change based on the state of the CuT.

In this experiment, due to the repeatability of the value fault injection method, the number of robustness failures of both approaches is the same for each injectable transition. As the results show, the type and the number of robustness failures of generic_segment_checks are the same. The reason is that, according to the log file, the generic_segment_checks function is called only when the generic_file_aio_read function executes. On the other hand, other functions (which have different type and the number of robustness failures) are called by some functions other than generic_file_aio_read. In such case, Ballisa approach focuses on the target function, regardless of the CuT state, whereas the proposed approach accurately detects when the target function is called by the CuT in the intended state and then it injects the fault. For example, the proposed approach discriminates between the put_page function in different paths of the model. In contrast, the Ballista method, tests the put_page function independent of the execution path. This makes differences in the results of fault injections in the two approaches. For example, as can be seen from the Table 3, Ballista has detected a catastrophic failure for find_get_page function whereas the proposed approach has been faced with a silent one. The proposed approach indicates that the execution of find_get_page in the execution path of the model, does not lead to a catastrophic failure. Thus, it can be concluded that by considering the OS state, it is possible to accurately identify the robustness problems of the CuT, since it determines the execution paths and the transitions which robustness failures occur in.

As the experimental results show, it is possible to determine the criticality of CuT states in the presence of incoming delays using the results of timing fault injections. In addition, the value fault injection can help to identify the possible class of robustness failures for each state of the CuT. These results can be used together to assess the robustness level of the CuT.

Table 3. Results of Value Fault Injection

Transition/ Function Name	Number of Test Cases	Number and Type of Failures						Approach
		Catastrophic	Restart	Abort	Silent	Hindering	Total	
generic_segment_checks	19	2	0	6	4	0	12	Proposed
		2	0	6	4	0		Ballista
find_get_page	10	0	0	4	4	0	8	Proposed
		1	0	4	3	0		Ballista
find_read_actor	19	2	0	4	8	0	14	Proposed
		1	0	4	9	0		Ballista
put_page 1	4	1	0	3	0	0	4	Proposed
put_page 2		0	0	3	1	0		
put_page		2	0	2	0	0		Ballista

Table 4. The value of R_t for Injectable Transitions

Transition Name	The Score in Value Fault Injection (VS_t)	The Score in Timing Fault Injection (TS_t)	R_t
generic_segment_checks	0.62	0.20	0.41
find_get_page	0.60	0.40	0.50
file_read_actor	0.60	0.40	0.50
put_page 1	0.30	0.80	0.55
put_page 2	0.45	0.20	0.32

4.3. Robustness Level Assessment

In this step, the results of timing and value fault injections are used to augment the behavioral model generated in the first step. The augmented behavioral model will show the CuT states and execution paths which suffer from robustness vulnerabilities. As a result, the system developer can appropriately use this behavioral model to make the system more robust using some fault tolerance techniques. In this regard, the robustness of every injectable transition t, denoted as R_t, is needed to augment the behavioral model. R_t, can be calculated using the robustness score of t in timing fault injection (denoted by TS_t) and the robustness score of t in value fault injection (denoted by VS_t). Table 4 presents the results of this calculation based on Definition 1 in Section 3.3.

As illustrated in Fig. 3, there are two paths in the behavioral model of the case study. By analyzing the log file, it was observed that the probability of passing path No. 1 (denoted by *f(P1)*) is 26.6%. Similarly, the probability of passing path No. 2 (denoted by *f(P2)*), is 73.4%. Consequently, based on the Definition 2 in Section 3.3, R_{P1}, the robustness of path No. 1, is obtained by multiplying the *f(P1)* by the average of R_t for transitions of path No. 1 (i.e. *generic_segamet_checks*, *find_get_page*, and *put_page1* transitions) which results in 0.1294.

According to the proposed approach, the minimum and maximum robustness level of a CuT is acquired based on the minimum and maximum values of R_{P_j} where P_j is a path in the behavioral model. In the behavioral model of the case study (Fig. 3), R_{P1} equals to 0.1294 and R_{P2} equals to 0.3174. Hence, the maximum robustness level of the CuT is 31.7% and its minimum robustness level is 12.9%, with respect to the fault free execution. In other words, if the CuT is called with a faulty input value, it is at least 12.9% and at most 31.7% probable that the CuT will not fail. Indeed, the augmented behavioral model can also be used to select a suitable fault tolerance technique and the precise location to apply it. Moreover, timing fault injection results determine the acceptable time overhead of the selected fault tolerance technique. For example, the augmented behavioral model can specify the states that require redundancy and determine whether their timing limitation allows using redundancy.

5. Conclusion and Future Work

The aim of this paper was to propose a state-aware approach for assessing the robustness of components of embedded real-time OSs. This approach can be used to evaluate the OS behavior in the presence of different types of faulty inputs in different OS states. In addition to effective decrease of test cases, the proposed approach can specify the precise location of robustness vulnerabilities.

This is the first study that suggests augmenting the behavioral model of the OS by using the results of fault injection experiments. The proposed approach employs the augmented behavioral model in order to extract valuable information about the robust behavior of the OS. Using this approach, it is possible to identify the critical OS states, their criticality level, and the maximum and minimum level of the OS robustness. For example, it can be shown what classes of robustness failure are probable to happen in specific system states. The proposed approach also facilitates comparing the robustness level of OS's components and helps the system developer to assess the effectiveness of different fault tolerance techniques. Therefore, the OS components are comparable with respect to their robustness level.

For further work it is suggested to consider the stressful environmental conditions in state-based robustness testing and investigating their effects in different OS states. Another possible area of future research would be to extend this approach to be used on the fly. Hence, the possible robustness failures can be forecasted based on the execution path of the system to apply appropriate fault tolerance techniques while the system is running.

References

[1] R. Ramezani, and Y. Sedaghat, "An overview of fault tolerance techniques for real-time operating systems". Proceedings of the 3rd International Conference on Computing and Knowledge Engineering, IEEE, Mashhad, Iran, October 31 - November 01, 2013, pp. 1-6.

[2] J. C. Knight, "Safety critical systems: Challenges and directions". Proceedings of the 24rd International Conference on Software Engineering, IEEE, Orlando, Florida, USA, May 19-25, 2002, pp. 547-550.

[3] E. A. Lee, "Embedded software". *Advances in Computers*, vol. 56, pp. 55-95, 2002.

[4] R. Natella, D. and H. S. Cotroneo, Madeira, "Assessing dependability with software fault injection: A survey", *ACM Computing Surveys (CSUR)*, vol. 48, no. 3, pp. 44:1-44:55, 2016.

[5] S. M. A. Shah, D. Sundmark, B. Lindström, and S. F. Andler, Robustness testing of embedded software systems: An industrial interview study. *IEEE Access*, vol. 4, pp. 1859-1871, 2016.

[6] Z. Micskei, H. Madeira, A. Avritzer, I. Majzik, M. Vieira, and N. Antunes, "Robustness Testing Techniques and Tools," In *Resilience Assessment and Evaluation of Computing Systems*, Springer, 2012; 323-339.

[7] W. Torres-Pomales, "Software fault tolerance: A tutorial", *Technical Report*, NASA Langley Research Center, Hampton, VA United States, NASA-2000-tm210616, 2000.

[8] A. Shahrokni, and R. Feldt, "RobusTest: A framework for automated testing of software robustness", Proceedings of the 18th Asia Pacific Software Engineering Conference, IEEE, 2011, pp. 171-178.

[9] D. Cotroneo, and H. Madeira, "Introduction to software fault injection," In *Innovative Technologies for Dependable OTS-based Critical Systems*, Springer: Milan, 2013, pp. 1-15.

[10] D. Cotroneo, D. Di Leo, R. Natella, and R. Pietrantuono, "A case study on state-based robustness testing of an operating system for the avionic domain". Proceedings of the 30th International SAFECOMP Conference, Springer, Naples, Italy, September 19-22, 2011, pp. 213-227.

[11] D. Cotroneo, D. Di Leo, R. Natella, "SABRINE: State-based robustness testing of operating systems". Proceedings of the 28th International Conference on Automated Software Engineering, IEEE, 2013, pp. 125-135.

[12] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities", *Communications of the ACM*, vol. 33, no. 12, pp. 32-44, 1990.

[13] B. P. Miller, D. Koski, C. Lee, V. Maganty, R. Murthy, A. Natarajan, and J. Steidl, "Fuzz revisited: A re-examination of the reliability of UNIX utilities and services", *Technical Report*, University of Wisconsin-Madison, Technical Report #1268, 1995, pp. 1-23.

[14] J. E. Forrester, and B. P. Miller, "An empirical study of the robustness of Windows NT applications using random testing". Proceedings of the 4th USENIX Windows System Symposium, Seattle, 2000, pp. 59-68.

[15] B. P. Miller, G. Cooksey, and F. Moore, "An empirical study of the robustness of macos applications using random testing". Proceedings of the 1st international workshop on Random testing, ACM, 2006, pp. 46-54.

[16] P. Koopman, and J. DeVale, "The exception handling effectiveness of POSIX operating systems", *IEEE Transactions on Software Engineering*; vol. 26, no. 9, pp. 837-848, 2000.

[17] P. Koopman, and J. DeVale, "Comparing the robustness of POSIX Operating Systems". Proceedings of the 29th Annual International Symposium on Fault-Tolerant Computing, IEEE, 1999, pp. 30-37.

[18] A. Johansson, et al., "On enhancing the robustness of commercial operating systems". Proceedings of the 1st International Service Availability Symposium (ISAS), Munich, Germany, May 13-14 , 2004, Springer, pp. 148-159.

[19] A. Albinet, J. Arlat, and J.-C. Fabre, "Characterization of the impact of faulty drivers on the robustness of the linux kernel". Proceedings of the in International Conference on Dependable Systems and Networks, IEEE, 2004, pp. 867-876.

[20] C. Sârbu, A. Johansson, N. Suri, and N. Nagappan, "Profiling the operational behavior of OS device drivers", *Empirical Software Engineering*, vol. 15, no. 4, 380-422, 2010.

[21] A. Johansson, N. Suri, and B. Murphy, "On the impact of injection triggers for OS robustness evaluation". Proceeding of the 18th IEEE International Symposium on Software Reliability, IEEE, 2007, pp. 127-136.

[22] S. Yang, B., Liu, S. Wang, and M. Lu, "Model-based robustness testing for avionics-embedded software". *Chinese Journal of Aeronautics*, vol. 26, no. 3, pp. 730-740, 2013.

[23] N. Coppik, O. Schwahn, S. Winter, and N. Suri, "TrEKer: Tracing error propagation in operating system kernels". Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, IEEE, 2017, pp. 377-387.

[24] R. Shahpasand, Y. Sedaghat, and S. Paydar, "Improving the stateful robustness testing of embedded real-time operating systems". Proceedings of the 6th International Conference on Computing and Knowledge Engineering, IEEE, Mashhad, Iran, October 20-21, 2016, pp. 159-164.

[25] Z. Zhou, Y. Zhou, M. Cai, and L. Sun, "A workload model based approach to evaluate the robustness of real-time operating system". Proceedings of the 10th International Conference on High Performance Computing and Communications & International Conference on Embedded and Ubiquitous Computing, IEEE, 2013, pp. 2027-2033.

[26] R. Alur, and D. L. Dill, "A theory of timed automata. *Theoretical computer science*, vol.126, no. 2, pp. 183-235, 1994.

[27] L. Parra, A., Lindoso, M. Portela, L. Entrena, F. Restrepo-Calle, S. Cuenca-Asensi, and A. Martínez-Álvarez, "Efficient mitigation of data and control flow errors in microprocessors", *IEEE Transactions on Nuclear Science*, vol. 61, no. 4, pp. 1590-1596, 2014.

[28] A. Johansson, N. Suri, B. Murphy, "On the selection of error model (s) for OS robustness evaluation". Proceedings of the 37th Annual International Conference on Dependable Systems and Networks, IEEE, 2007, pp. 502-511.

[29] S. Winter, C. Sârbu, N. Suri, and B. Murphy, "The impact of fault models on software robustness evaluations". Proceedings of the 33rd International Conference on Software Engineering, ACM, 2011, pp. 51-60.

[30] H. Fayyad-Kazan, L. Perneel, and M. Timmerman, "Linux PREEMPT-RT vs. commercial RTOSs: How big is the performance gap? *GSTF Journal on Computing*, vol. 3, no. 1, pp. 135-142, 2013.

[31] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge,and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite". International Workshop on Workload Characterization, IEEE, 2001, pp. 3-14.

[32] V. Prasad, W. Cohen, F. C. Eigler, M. Hunt, J. Keniston, and J. Chen, "Locating system problems using dynamic instrumentation". Proceedings of the Linux Symposium, Ottawa, Ontario, Canada, July 20-23, 2005, pp.49-64.

[33] A. Shahrokni, and R. Feldt, "A systematic review of software robustness", *Information and Software Technology*, vol. 55, no. 1, pp. 1-17, 2013.