6th International Conference on Computer and Knowledge Engineering (ICCKE 2016), October 20-21 2016, Ferdowsi University of Mashhad

Improving the Stateful Robustness Testing of Embedded Real-Time Operating Systems

Raheleh Shahpasand, Yasser Sedaghat, Samad Paydar
Dependable Distributed Embedded Systems (DDEmS) Laboratory
Computer Engineering Department
Ferdowsi University of Mashhad
Mashhad, Iran
ra.shahpasand@stu.um.ac.ir, y_sedaghat@um.ac.ir, s-paydar@um.ac.ir

***Abstract*— Software fault tolerance is an important issue when using software systems in safety-critical applications. In such systems, software robustness is an essential requirement for improving software fault tolerance. Since an operating system (OS) is a major part of a safety-critical system, its robustness has considerable influence on the system's overall robustness. In recent years, researchers have emphasized the importance of considering the OS state in robustness testing. OS state is determined by analysis of the interactions between OS components. In this paper, an approach, named TIMEOUT, is proposed for robustness testing of embedded real-time OSs. This approach reveals the impact of time delays, i.e. inputs with invalid timing delay, on the OS kernel functionality. TIMEOUT takes the OS state into account and improves the existing robustness testing methods. The proposed approach has been implemented and the experiments have been performed on Linux PREEMPT-RT, which is an embedded real-time implementation of Linux operating system. The results show that OS state can influence the OS behavior with respect to fault tolerance, in the presence of time delays. Based on the results of this approach, system developers can identify criticality of OS states and improve robustness of OS in those states.**

Keywords—Robustness Testing; Operating System; Fault Injection; Time Delay; Safety-Critical Systems.

I. INTRODUCTION

Software is responsible for overall functionality of a computer system. Correct and dependable behavior is a critical required property for software systems. Because producing an error-free software, theoretically and empirically, is still not possible [1], software fault tolerance is being applied as an acceptable assurance of computer system's functionality. Fault tolerance is especially important when it comes to computer systems applied in safety-critical applications. Components of a safety-critical system, including software and hardware, are expected to be fault-tolerant. Since the OS is the interface between software and hardware, it has a major role in the system, and application level software relies on its correct behavior [1]. This means that a failure in OS may lead to the failure of the whole system. As a result, improving OS fault-tolerance has positive impact on the whole system fault-tolerance [2].

Since using robust software systems is a fault tolerance technique [3], for improving OS fault tolerance, robustness testing is worthwhile. In robustness testing, robust behavior of a software module in the presence of exceptional inputs is assessed. The goal of robustness testing is to activate those faults or vulnerabilities in the system that result in incorrect operation [4]. Exceptional inputs fall into four categories [1, 5, 6]: 1) Invalid and unexpected **value** or 2) **timing** of an input, 3) invalid input **sequence** and 4) unexpected input **format**. Invalid inputs are given to the system interface using fault injection, which is a common technique in robustness testing and highly recommended in safety standards like DO-178C [7].

In the literature, the majority of researches like [8-10] have injected an invalid value as an exceptional input to OS interface (e.g. API or device driver). A proposed tool in [11] has been employed to inject inputs with invalid timing into the OS kernel. These approaches are not state-aware and as a result, number of required test cases is considerable. Subsequently, due to considerable number of test cases, it is possible that not all of the system states are covered during the test. The SABRINE approach [12] evaluates the robust behavior of the OS under different states. But this approach has not considered any of the exceptional inputs that mentioned earlier, and has tested OS robustness against service failure of a component. Furthermore, SABRINE is based on a running workload that could affect its results.

In this paper, an approach for robustness testing of embedded real-time OS, called TIMEOUT, is proposed that reveals the impact of time delays on the OS kernel functionality. Concerning importance of timing in such OSs, TIMEOUT injects inputs with invalid timing into the OS kernel. In this approach, fault injections are state-aware that results in improving the effectiveness of fault injection process. We apply SABRINE approach for extracting state models, but unlike SABRINE, TIMEOUT focuses on OS kernel operation and does not depend on a running workload. To evaluate the approach, Mibench [13] as a popular benchmark in safety-critical applications is used as the workload. The results of executing the workload are processed to extract the OS model. We evaluate TIMEOUT on an embedded real-time Linux-based OS. The

978-1-5090-3586-1/16/$31.00 ©2016 IEEE

results show that erroneous delays in the kernel operations greatly affect the deadline violation of application level software. Results also can be used to identify OS critical states in the presence of time delays.

The outline of this paper is as follows: Section II provides an overview of the most important related works. The proposed approach (TIMEOUT) is introduced in Section III, and we evaluate the proposed approach on a case study in Section IV. Finally, Section V concludes the paper.

II. Related Work

Due to the importance of OS in overall system functionality, OS robustness testing has been of interest to researchers for many years, such that in the literature, robustness testing is mainly focused on operating systems [14]. In [15] an approach is proposed for testing the robustness of a real-time operating system. In order to evaluate the ability of commercial OSs to handle errors generated by user-space applications, the invalid input values have been presented to system call interface. This approach has improved the efficiency of robustness testing by using data-type based error injection.

Proposed tool in [16] has used a grammar-based description of the system's input to generate random and syntactically valid but anomalous input. The tests have been performed on the Windows NT platform, and the robustness failures observed are mainly memory access violation exceptions, privileged instruction exceptions and illegal instruction exceptions.

BALLISTA [8, 17] has extended the approach of [15] with the goal of testing and benchmarking of commercial OSs. Each robustness test consists of a system call invocation with a combination of both valid and invalid parameters. Despite the large number of test cases, these studies have found severe robustness vulnerabilities in several commercial OSs.

A profiling framework is proposed in [18] that assists in finding possible error propagation paths from drivers through the OS to the applications. The aim of the profiling framework is to find effective location of OS wrappers to handle driver errors and estimate the impacts of errors on the services provided by applications. The framework has helped to enhance the OS with selective robustness hardening capabilities like wrappers.

The presented work in [19] concerns OS robustness testing with respect to device driver interface and focuses on testing the Driver Programming Interface (DPI). DPI is a set of kernel core functions that implements a way device drivers interact with the kernel. To characterize the robustness of OSs relating to faulty drivers, faults have been injected on the parameters of these kernel core functions. The results show the negative impact of faulty drivers on responsiveness of the kernel, safety of the workload and availability. Sârbu et.al. in [20] have proposed a state model for testing device drivers, using communications on device driver interface, therefore this work has characterized run-time behavior of device driver. They have found that the use of a state model can reduce number of test cases.

Johansson et al. [21] have introduced the concept of call blocks to take into account the state of the OS in robustness testing. The usage profile of a device driver is split into disjoint call blocks. Call blocks, i.e. recurring sequences of function calls, guide injections into different system states. The results have shown that controlling the time of injection has significant impact on the robustness evaluation.

In [2] the goal is also to enhance the traditional approaches by considering the OS state in test case definition. Since state of the OS components has a significant influence on the OS correct behavior, it is necessary to take the states of the component under test into account. A component is a subsystem of the OS that is responsible for managing a resource or for providing a set of services, such as memory management and process scheduling. Since OSs are complex and stateful systems, executing a given robustness test case in different states increases the probability to explore those parts of the code most rarely reached during the execution, i.e., it increases the final coverage [12].

Based on this view, test plan has been expressed through two dimensions: the exceptional inputs and the states. Inputs are selected as usual (e.g. through boundary value analysis), while the state varies in $S = \{s_1, s_2 \dots s_n\}$, 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 component interface [2]. This approach is extended by SABRINE in 5 phases:

1) Behavioral Data Collection: Before performing robustness testing, the system is executed and profiled under fault-free conditions. This phase collects data about the OS behavior, in terms of interactions between OS components at run-time and records them in a log file.

2) Pattern Identification: Log file that includes target component interactions is converted to a set of sequences. Each sequence is a set of events. Sequences of a particular call are grouped together, and represented as a pattern.

3) Pattern Clustering: Identified patterns in previous phase, are further grouped together, in a cluster. To perform clustering, the similarity among all pairs of patterns is measured based on a similarity function.

4) State Model and Test Suite Generation: For each cluster, a behavioral model in form of a Finite State Automata (FSA) is generated. Injectable interactions are identified and robustness test cases are generated for each of them.

5) Test Execution: Robustness test cases are translated in test programs that are then executed to inject faults in the different states of the OS. Each test executes the system under the same working condition of the first phase. During execution, behavioral data is collected and analyzed at run-time, and a fault is injected when the OS reaches a given state of the behavioral model.

Research in the recent years has been emphasized the significance of OS state in robustness testing. In addition, the importance of time in a real-time OS illustrates the necessity of OS robustness testing in the presence of inputs with invalid timing. Moreover, timeliness is one of the vital embedded software systems' characteristic that influences the correctness of the embedded systems' behavior [14]. Nevertheless, investigating the impact of inputs with invalid timing in each OS state and resulting delays in workload runtime is an open issue, not considered in existing works. The goal of this paper is to address this issue by injecting time delays in the kernel and investigating the impact of inputs with invalid timing delay on OS kernel functionality.

III. PROPOSED APPROACH

For a system in safety-critical applications, time delays are serious. In these systems, there are timing deadlines, such that if the deadline misses, the answer will be unacceptable. In this paper, for stateful robustness testing of an embedded real-time OS, time delays are injected to the OS kernel and the impact of them on application level software is investigated. We extract the OS states based on SABRINE approach. One advantage of this approach is considering OS state in robustness testing other than exceptional inputs. The state can affect how an event impacts on the OS and the ability of the OS to robustly handle its occurrence [12]. In this section, the TIMEOUT approach is explained. It consists of 4 phases:

1) Workload Execution. This is needed to extract behavioral data of target component. This phase is independent of the running workload and focuses on OS kernel functionality. During the workload execution, we log component input (the target component may be invoked by another component) and output (the target component may invoke another component) interactions. Moreover, start and end times of every interaction is recorded in the log file based on a predefined time unit. It should be noted that some factors such as different execution paths can affect interactions. For this reason, the execution of workload is repeated several times during this phase.

2) Pattern Identification. At this point, there is a log file containing a set of sequences. Given that evaluating the functionality of the OS kernel in general (and not by running a particular workload) has been considered, we define a sequence as a set of events that have happened during the execution of an individual kernel function call. Two executions of a particular kernel function call will not necessarily lead to identical sequences. The sequences of a function call are grouped together, and represent a pattern. Due to the different execution paths in functions, sequences of a pattern will vary. For this reason, patterns are clustered.

3) Behavioral Modeling. Given that the relative start and end times of interactions are recorded, we infer a behavioral model in the form of a Timed Automata [22] from each cluster. This behavioral model consists of a set of states that are connected with events which have defined timing.

4) Test Case Generation and Test Execution. The model has more than one transition in some states, because patterns in a particular cluster are different. This process is repeated for every cluster. Each injectable transition in the model generates one or more test cases. The result of test case execution can help to identify critical states about time delays and their criticality.

IV. EVALUATION

The TIMEOUT approach described in the previous section has been implemented, and to evaluate it, different experiments have been performed. As a case study, we have selected Linux PREEMPT-RT, a real-time Linux implementation which is used in embedded applications [23]. Due to the importance of memory in real-time embedded systems [24], memory management is selected as the component under test.

The execution of workload is essential, such that the target component interacts with other components and calls kernel functions. As a workload, Mibench [13] is selected which is a representative benchmark for embedded programs. More specifically, automotive category of Mibench is intended for safety-critical applications and from this category, we have used Qsort as our workload program running on the Linux PREEMPT-RT.

To record interactions between the target and other components, and to produce a log file, SystemTap [25] tool has been utilized. SystemTap allows to investigate the behavior of the kernel. It uses a dynamic method of monitoring and tracing the operation of a running Linux kernel. Also, SystemTap has been applied in test execution phase to perform fault injection experiments.

In the TIMEOUT approach, worst execution time for a particular function call that has been recorded in log file is considered as its deadline. Since the target OS is a real-time OS, by giving priority to workload, we can be sure that execution time does not include interrupts or OS scheduling.

The four mentioned phases of the TIMEOUT approach have been performed and the timed automata model for *generic_file_aio_read* call (generic read routine to read filesystem) is shown in Figure 1. The *generic_file_aio_read* call follows by a read system call. In this figure, clock *t* has specified the execution time of each function call in microseconds. In this model, injectable transitions have been marked.

Since robustness looks at the system's response to faulty input, injectable transitions are those input interactions that their input parameter value, influences the execution path. For example, if an input parameter is used as a condition expression (in a loop or condition statement), it will be the basis for decision-making and consequently it can change the execution path (usually this information can be obtained from the function call specification and availability of OS source code is not necessary). In this case, when a fault appears and changes input value (e.g. existence of transient hardware defects that conduces bit-flip error), the execution path and then execution time can change. It is worth mentioning that changing the program execution path not necessarily increases the execution time, but what is important is not to miss deadline. In this paper, time delays are injected in functions.

Finally, a set of robustness test cases are extracted from the behavioral model. Such that every injectable transition generates one or more test cases. We have used binary search to select test cases (time delays that have injected) based on function call deadline. If x is a particular function call deadline in microsecond, the range of possible delays is [1, x], and the first test case is a delay with $\frac{x}{2}$ of timing unit.

In our fault injection method, the injectable line of function code is delayed for $\frac{x}{2}$ unit of time. According to the result of fault injection experiment, new fault injection range is identified. If the deadline is missed, the new test case is in the range $[1, \frac{x}{2})$, otherwise it is in the range $(\frac{x}{2}, x]$. This process continues to find a delay threshold that does not miss function deadline.

Injectable lines of every transition are identified and one of them is selected to perform a fault injection. To inject fault in a desired state, the target component should be in that state. Hence the component interactions are monitored to identify current state of the component and then fault is injected.

Based on the model in Figure 1, five out of the seven transitions have been detected as injectable transitions. The number of injectable lines in each injectable transition has been shown in Table I. Because OS state has been considered in the evaluations, it is expected that the result of fault injection in *put_page 1* (*put_page* function call in path no. 1) is different with the result of fault injection in *put_page 2* (*put_page* function call in path no. 2). The results have been shown in the Figure 2 and Figure 3. Horizontal axis shows injected time delays to the function source code and vertical axis determines the execution time.

According to our experiments, the worst execution time for *generic_file_aio_read* call that has been recorded in log file, has been 266us which is considered as the call execution deadline. Figure 2 shows that in all of the fault injection experiments, *put_page* deadline (according to the model it is 6us) has been missed but *generic_file_aio_read* deadline only missed when 215us and 232us delays were injected. As a result, we can state that deadline violation in one of the transitions not necessarily leads to deadline violation in the call.

Figure 3 indicates that fault injection in *put_page 2* has a major impact on *generic_file_aio_read* call compared to fault injection in *put_page 1*. So that tolerable time delay (an injected time delay which has not led to deadline violation) for *put_page 2* is 61us and it is 213us for *put_page 1*. Because there are two more transitions in path 2, execution time of *generic_file_aio_read* has increased and the time delay threshold of *put_page 2* has decreased. Furthermore, comparing Figure 2 and Figure 3 with each other it is observable that fault injection in *put_page 2* increases the execution time of *generic_file_aio_read* with a higher rate. However, in Figure 2 it can be seen that the execution time of *generic_file_aio_read* call compared to the execution time of *put_page* call increases with a constant rate.

Table I shows the deadline violation thresholds of *generic_file_aio_read* call in every injectable transition of Figure 1. Comparing the deadline violation thresholds in *generic_segment_checks* with the deadline violation thresholds of other transitions indicates that existence of more transitions in the path, not necessarily leads to decrease the deadline violation threshold.

To the best of our knowledge, there is not a similar approach in the literature to compare with TIMEOUT. Hence, to evaluate the efficiency of TIMEOUT, the TIMEOUT's results are compared with the results of a random fault injection approach. In the random approach, there is not a model to identify the execution path. Thus injectable lines in this approach are consisted of the injectable lines in the source code of the *generic_file_aio_read* function and the injectable lines in each identified injectable interaction in the TIMEOUT approach (i.e. all of the identified injectable lines). 6 injectable lines were identified in the source code of *generic_file_aio_read* and the total number of identified injectable lines in TIMEOUT is 12. Consequently, there are 18 injectable lines that in the random approach can be selected for fault injection. This amount in TIMEOUT depends on model paths varies between 10 to 12 lines. Decreasing of injectable lines in TIMEOUT efficiently leads to decrease the test space. Table I indicates the number of injectable lines for each approach.

In the random approach, we have selected one of the injectable lines randomly and have injected time delays to it. Moreover, to specify the range of a time delay, binary search method has been used.

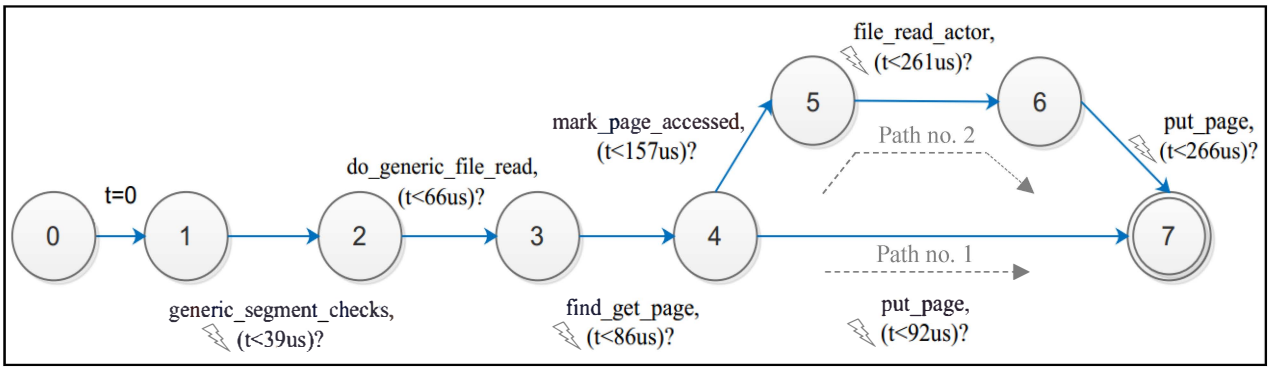

Fig. 1. Behavioral model of generic_file_aio_read

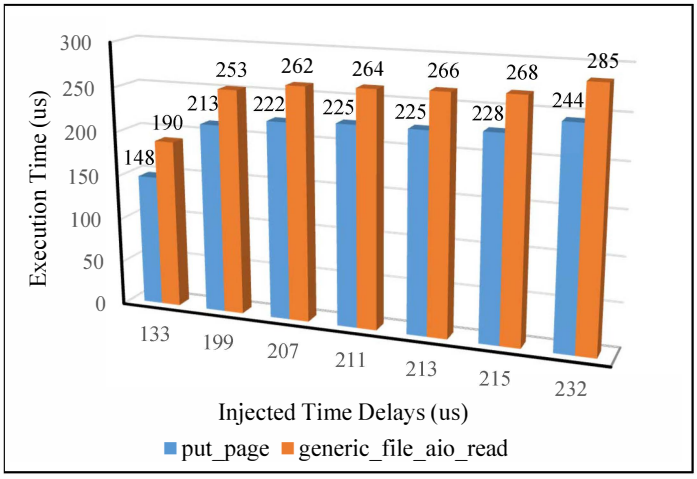

Fig. 2. Impact of fault injection on *put_page 1*

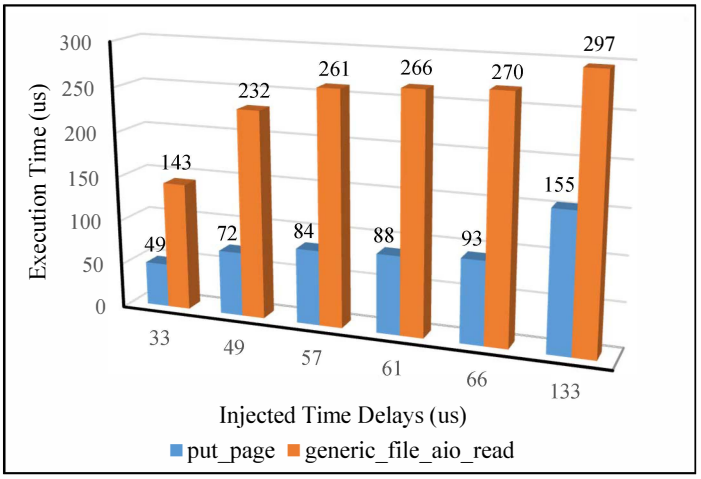

Fig. 3. Impact of fault injection on *put_page 2*

TABLE I. DEADLINE VIOLATION THRESHOLDS OF GENERIC_FILE_AIO_READ CALL

	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

Table I also shows the deadline violation thresholds of random approach. Based on Table I, TIMEOUT identifies critical OS states along with estimating criticality of each state. For example, according to TIMEOUT, *put_page 2* with 61us deadline threshold is the most critical transitions in the presence of time delays. Using this information, a developer can apply an appropriate fault tolerance technique to improve system's reliability in such states. The technique should not impose more than 61us time delay as its time overhead.

V. CONCLUSION

This paper proposed an approach for state-based robustness testing of an OS in the presence of inputs with invalid timing delay. Due to the importance of OS state in robustness testing and significance of time in embedded real-time OSs, proposed approach evaluates the impact of time delays in different OS states. This approach can help to identify critical states, regarding time delays, and estimate their criticality. The results of our experiments highlight the influence of OS state together with time delays on the kernel execution time and emphasize the importance of the time in OS robustness testing, especially when the OS is used in a safety-critical application.

REFERENCES

[1] Torres-Pomales W., "Software fault tolerance: A tutorial," NASA Technical Report, NASA-2000-tm210616, 2000.

[2] 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," Proc. of the 30th International Conference on Computer Safety, Reliability and Security(SAFECOMP) 2011, Springer, 2011, pp. 213-227.

[3] Pullum L.L.,"Software fault tolerance techniques and implementation," Artech House, 2005.

[4] Z. Micskei, "Robustness Testing Techniques and Tools," In Resilience Assessment and Evaluation of Computing Systems, Springer, 2012, pp. 323-339.

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

[6] D. Cotroneo and H. Madeira, "Introduction to Software Fault Injection," In Innovative Technologies for Dependable OTS-Based Critical Systems, Springer, 2013, pp. 1-15.

[7] RTCA, DO-178C Software considerations in airborne systems and equipment certification, 2011.

[8] P. Koopman and J. DeVale, "The exception handling effectiveness of POSIX operating systems," IEEE Transactions on Software Engineering, 2000, 26(9): pp. 837-848.

[9] X. Ju and H. Zou,"Operating system robustness forecast and selection," Proc. of the 19th International Symposium on Software Reliability Engineerin 2008, IEEE, 2008, pp. 107-116.

[10] Z-M. Zhou, Z-R. Zhu and M. Cai, "Designing an Efficient and Extensible Robustness Benchmark of a Real-Time Operating System," Cybernetics and Information Technologies, 2015, 15(1): pp. 84-103.

[11] S. Winter, T. Piper, O. Schwahn, R. Natella, N. Suri and D. cotroneo, "GRINDER: On Reusability of Fault Injection Tools," Proc. Of the 10th International Workshop on Automation of Software Test 2015, IEEE/ACM, 2015, pp. 75-79.

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

[13] 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," IEEE International Workshop on Workload Characterization(WWC-4) 2001, IEEE, 2001, pp. 3-14.

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

[15] C.P. Dingman, J. Marshall and D.P. Siewiorek, “Measuring robustness of a fault-tolerant aerospace system,” Proc. of the 25th International Symposium on Fault-Tolerant Computing 1995, IEEE, 1995, pp. 522-527.

[16] A.K. Ghosh, M. Schmid and V. Shah, "Testing the robustness of Windows NT software,” Proc. of the 9th International Symposium on Software Reliability Engineering 1998, IEEE, 1998, pp. 231-235.

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

[18] A. Johansson, A. Sarbu, A. Jhumka and N. Suri, "On enhancing the robustness of commercial operating systems,” International Service Availability Symposium 2004, Springer, 2005, pp. 148-159.

[19] A. Albinet, J. Arlat and C. Fabre, “Characterization of the Impact of Faulty Drivers on the Robustness of the Linux Kernel,” Proc. of the 34th International Conference on Dependable Systems and Networks 2004, 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, 2010, 15(4): pp. 380-422.

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

[22] R. Alur and D.L. Dill, “A theory of timed automata,” Theoretical computer science, 1994, 126(2): pp. 183-235.

[23] H. Fayyad-Kazan, L. Perneel and M. Timmerman, “Linux PREEMPT-RT vs. commercial RTOSs: how big is the performance gap?,” GSTF Journal on Computing (JoC), 2013, 3(1): pp. 135-142.

[24] Marwedel P., “Embedded System Design,” 2nd ed., Springer, 2011.

[25] Domingo D. and Cohen W., “SystemTap 2.9 SystemTap Beginners Guide,” Red Hat, 2013.