POLITECNICO DI TORINO
Repository ISTITUZIONALE

On the Analysis of Real-time Operating System Reliability in Embedded Systems

Original
On the Analysis of Real-time Operating System Reliability in Embedded Systems / Mamone, D., Bosio, A., Savino, A., Hamdioui, S., Rebaudengo, M.. - STAMPA. - (2020), pp. 1-6. (33rd IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems Frascati, Italy, Italy 19-21 Oct. 2020) [10.1109/DFT50435.2020.9250861].

Availability:
This version is available at: 11583/2853433 since: 2020-11-20T14:21:45Z

Publisher:
Institute of Electrical and Electronics Engineers Inc.

Published
DOI:10.1109/DFT50435.2020.9250861

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright
IEEE postprint/Author's Accepted Manuscript

©2020 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collecting works, for resale or lists, or reuse of any copyrighted component of this work in other works.

(Article begins on next page)

11 November 2026

On the Analysis of Real-time Operating System Reliability in Embedded Systems

Dario Mamone[1], Alberto Bosio[2], Alessandro Savino[1], Said Hamdioui[3] and Maurizio Rebaudengo[1]
[1] Politecnico di Torino, Italy, *{maurizio.rebaudengo, alessandro.savino}@polito.it
[2] Lyon Institute of Nanotechnology, École Centrale de Lyon, France, *alberto.bosio@ec-lyon.fr
[3] Computer Engineering Lab, Delft University of Technology, The Netherlands, *S.Hamdioui@tudelft.nl

***Abstract*—Nowadays, the reliability has become one of the main issues for safety-critical embedded systems, like automotive, aerospace and avionic. In an embedded system, the full system stack usually includes, between the hardware layer and the software/application layer, a middle layer composed by the Operating System (OS) and the middleware. Most of the time, in the literature only the application-layer is considered during the reliability analysis. This is due to the fact that middle layer short execution time makes the probability of a fault affecting it much lower compared to the application-level. Nevertheless, middle layer data structures lifespan is equivalent to the application layer ones. Moreover, all the times a hardware fault propagates to the middle-layer as an error, and especially to the OS, its impact can be expected to be potentially catastrophic. The aim of this work is to study the reliability of a Real-Time Operating System (RTOS) affected by Single Event Upset (SEU) faults. The methodology targets the most relevant variables and data structures of FreeRTOS analyzed through a software-based fault injection. Results show the ability to highlight the criticality in the OS fault tolerance, in terms of system integrity, data integrity and the overall inherent resiliency to faults, potentially leading to selective hardening of the OS.**

***Index Terms*—Embedded Systems, Real-Time Operating System, Fault Injection, Reliability**

I. Introduction

Embedded systems are commonly employed in several fields, spanning from consumer electronics, e.g., mobile devices, IoT, to safety-critical applications, such as automotive, aerospace and avionic. The computational power of embedded systems increased over the years in order to meet the escalation of application constraints, like object detection on autonomous driving [1]. For these reasons, embedded systems are growing in complexity, including multi-core systems, characterized by the integration of more than one Central Processing Unit (CPU) and Graphical Processing Unit (GPU) on the same chip. As a direct consequence, parallel programming paradigms are enabling a great increase in computational throughput. Moreover, the full system stack usually requires, between the hardware layer and the software/application layer, a middle layer composed by the Operating System (OS) and the middleware (e.g., peripheral drivers) to properly deploy the final application.

Embedded systems must meet both required and desirable features chosen at design time and/or imposed by standards, according to its *mission*. In any case, a desirable property for every system still remains the *dependability*. Indeed, dependability can be reduced in many ways: without taking into consideration errors due to design, hardware or software bugs, issues occurred during fabrication, intentional tampering, or many other external events that can affect this property during the lifetime of the application. Due to the interaction of the circuit with the surrounding environment memory bit-flips, signal degradation, data loss, and permanent damage to the physical circuit may happen [2].

Usually, the system dependability can be achieved with a deep analysis of the system weaknesses, and then with the implementation of mitigation techniques that allow to reduce or completely remove them. However, extensive testing phases come at cost of money and time, delaying time-to-market and increasing the final price per-unit. For those reasons an optimal trade-off must be found during the design phase so that the system can still work with the desired quality level without impacting the production too much.

In the literature, when the full system is considered, the application-layer is commonly the target of the reliability analysis [3]. Excluding bare-metal system scenarios, the assumption is the fact that the probability of a fault affecting the middle layer is much lower compared to the application-level due to its short execution time (i.e., system calls execution time is generally shorter than the overall application). On the other hand, middle layer data structures (e.g., process queues) lifespan is equivalent to the application layer ones, if not even bigger. Moreover, it is clear that, all the times a hardware fault propagates to the middle-layer as an error, and especially falling within the OS, its impact can be expected to be potentially catastrophic, causing a various range of misbehavior, such as system crash, missed deadlines, application freeze as well as tasks synchronization errors.

The aim of this work is to study the reliability of a Real-Time Operating System (RTOS) affected by Single Event Upset (SEU) faults. The applied methodology targets the most relevant variables and data structures of the FreeRTOS [4] that will be analyzed through a software-based fault injection campaign. The paper evaluates the impact on the application in terms of system integrity, data integrity and the overall resistance to faults.

The remaining of this paper is structured as follows. Section II presents the basics knowledge and the state of the art.

*This work has been partially founded by CNRS PICS07968 project.

Section III details the proposed methodology, and Section IV gives the experimental results. Finally, Section V draws the conclusions.

II. Background

This section presents the state of the art about *Fault Injection* techniques (including a discussion about related works) and the basics knowledge regarding *Real-Time Operating System*.

A. Fault Injection

Fault injection is the standard technique to evaluate the reliability of systems under faults [5]. It is based on the realization of controlled experiments, where a fault is introduced in the target system, in order to observe the performance of the system in the presence of faults. Fault injection techniques are classified as (i) hardware-based, when injecting faults directly in the target hardware, and (ii) software-based, when modeling the hardware fault at an abstract level. Hardware-based techniques [6] perform fault injection campaigns in more realistic conditions and provide therefore more accurate results, while software-based techniques [7] [8] provide cheaper solutions to evaluate the reliability with sufficient accuracy levels.

In general, while both methodologies can work with most applications, the operating system fault injection can be a less trivial subject, because the injection of a fault can easily lead to a non-responsive system, where only rebooting the system allows interacting with the system again. Probably because of that, not all the fault injectors in literature address the operating system layer in the same way. In fact, some of them, like [9], just mimic an error in the OS by wrapping around the API level (e.g., a system call). The result leads to the investigation of a small set of faults: the ones that affect only the data provided by the OS to the application.

A fist attempt to analyze the effect of faults into the kernel space have been proposed in [10], where a basic *software-implemented fault injection* (SWIFI) was developed to address a set of three data fault models not related to any physical error. Authors claim to generally target the task image, probably covering part of the user space, i.e., the program itself, as well as part of the operating system.

An important work has been proposed under the name of MAFALDA in [11]. The tool targets micro-kernels as injected systems. Micro-kernels are operating systems provided as a library to be integrated with the actual set of applications to run on small systems, and their peculiarity is that their functionalities are invoked using *trap* calls instead of common API functions. The actual fault injection is limited to the corruption of pseudo-random bytes of (i) the parameters passed to the micro-kernel primitives and (ii) of the address space of the micro-kernel functional components. Again, this approach limits the capability of the tool to provide insight on the behavior of operating system components under faults.

Authors in [12], [13] target a more complex Operating System such as Linux similarly: they resort to the debug interface to stop the execution and inject the fault into an operating system instruction. While the appealing of the work is the usage of performance counters to measure the latency of the error, the fault model is just a bit flip in the assembly instruction belonging to the kernel execution (either opcode or data), which makes the approach not totally refined to be useful in operating system reliability analysis and hardening.

A very interesting approach is presented in [14]. Authors aim at building a fault injector that lays between the firmware level and the operating system. While the idea does not represent an OS-specific fault injector, the basic concepts could represent an interesting approach. Unfortunately, authors did not provide any results.

More recent works, like [15]–[18] target specific OS data structure to evaluate the impact of a fault to the full system. Authors in [15] and [18] seem to cover most of OS data structure but both papers do not provide any experimental results. In [16] the synchronization capabilities via mutex OS structures are addressed, while in [17] a completely different scenario is presented: in order to evaluate the capability of tampering the OS, using a Differential Fault Attack (DFA), a specific fault injector is proposed. Due to the scope of the work, only user data memory is targeted by the fault injector, missing all system data.

On the side of reliability analysis of OS, to the best of our knowledge only one paper addressed the matter. In [19], authors resort to a Bayesian network model of the internal states of a real-time operating system (CRTOS II) to predict its reliability. In order to support their claim, they describe an experimental setup where several failures are applied. Unfortunately, the paper does not contain any explanation about the fault model behind the investigation and how authors produce the failures.

B. RTOS

A Real-Time Operating System (RTOS) is an operating system designed to perform operations in a precise amount of time, respecting well-defined deadlines. An RTOS is then used all the times the system reactivity and computation times are very crucial for the whole application. Moreover, in safety-critical applications, it is very common to being able to avoid such deadlines to be violated: if this happens, the system could injure more or less severely other systems, people and objects.

RTOSs are able to schedule concurrent operations belonging to different contexts in the form of *tasks* (or *processes*) and to switch among them in such a way that desired timing is still respected. Each task can be in a defined state in every moment of its lifetime and the programmer can partially choose how and when a task must change it; usually all operating systems (not only RTOSs) recognize three states for each task: the *ready state*, when the task is ready to be scheduled; the *running state* when the task has been switched in and it holds the core of the processor; the *waiting state* when the task is waiting for an event to happen. Sometimes two additional states are added, and they are the *new state*, used to identify tasks that have been just created and never scheduled, and the *deleted state*, when a task must be removed from the system and it

is waiting for the kernel to clear its stack and to free all the memory associated to it.

In this work, we used the FreeRTOS [4] as case study. Among the class of RTOS, it targets embedded systems where other OSs are not able to fit the memory due to its small footprint. It is a good choice also for its completeness. Moreover, FreeRTOS does not include any additional safety-oriented features. This lack makes FreeRTOS a right choice in order to investigate the impact of faults on the OS.

III. Fault Injection Environment

The aim of this work is to develop a *Fault Injection Environment* (FIE) able to reproduce the effects of SEU, and in particular Single Bit Upset(SBU), in the memory of the Device Under Test (DUT), focusing only on main data structures and variables of FreeRTOS and to trace the events so that they can be saved on a host computer and successively analyzed. The DUT must be chosen in order to be representative of a common platform used in embedded systems, thus with limited resources.

When designing a FIE, several characteristics have to be included: (i) the FIE must be able to perform automatically long fault injection campaigns after an initial configuration, (ii) it must be able to inject in given memory location, at given times and in the desired bit of the datum, (iii) experiments must be repeatable in order to be relevant and (iv) the FIE must be as less invasive as possible not to alter the system performances.

Our FIE system is composed by a board that acts as DUT, and by a Host machine working as platform, controlling the injection campaigns and evaluating the results. It is able to inject in different injection targets, i.e., the OS locations (e.g., variables or data structures), and to analyze the data collected in order to classify the observed OS behavior into four classes, as described later.

From the flow point of view, the host-side program sends to the DUT a sequence of injection parameters when the application starts. Then, the DUT is left free to run for a defined amount of time and, finally, the injection is performed on the desired target. A resume routine is used by the DUT to send back to the host results of the injection. Figure 1 shows a scheme of the whole system.

It can be noticed that the FIE is composed by three parts: the first one (*FIEbrd* in Fig. 1) is written for the DUT, it is architecture-dependent, and it manages the communication with the host machine, and controls the injection in the desired location; it was developed entirely in C. The second one is a Python script called *FIEmon.py* that operates on the host-side, it manages the injection campaign and saves results of the various experiments in a file. For each injection campaign, it pilots the execution of two runs of the same algorithm: the first one is a *golden* execution to be used as reference of the fault free run, without injection, and the other one to get the outcomes of the real injection. Finally, a second Python script, called *FIEparser.py*, extracts data from the injection campaign: it takes as input the data from the two runs and

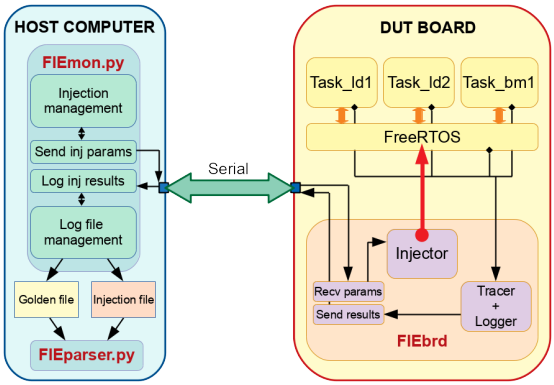

Fig. 1: Fault Injection Environment Model

performs the output comparisons to detect the effect of the injected fault; this operation can be done at any time after the injection campaign.

Figure 2 depicts the communication protocol to initiate the fault injection process and to retrieve all data. The first step requires to upload the executable file into the board. After that, the DUT setups the clock of the board's micro-controller, the USART peripheral and the GPIO port on the board because they will be used for messages and data passing purposes. For these reasons, no benchmark using those two hardware components can be implemented. After the setup of all involved peripherals, the DUT sends a synchronization message back to the Host (the *STARTITCHAR* byte in Figure 2) to inform it that the initialization is completed. This message is part of a hand-shake protocol that ends with the host sending back a further initialization message (the *INIT* command sent back to the DUT in Figure 2). Until this message is received, the DUT hangs in an endless loop, waiting for the USART interrupt to be raised. Once the handshake completes successfully, the DUT is ready to receive the fault injection parameters. When the DUT acquires all parameters, they are sent back just for channel debugging. This duplication is also used to signal the host to start the event logging. While the host starts the logging, the benchmark on the DUT initiates its run, without any knowledge about the injection. In fact, the actual injection event resorts to the timer interrupt implemented on the board, and, it is totally asynchronous with respect to the benchmark execution. To properly support this approach, before running the benchmark, the prescaler and the timer registers are set accordingly to the injection parameters to raise a timer interrupt at the chosen injection instant. Indeed, the Interrupt Service Routing (ISR) for the timer interrupt handler has then been modified to call an injection function that, given the selected location, performs the injection by bit-wise operations.

In order to ensure each run to reach an end, a resume timer is used to control the execution timing of the benchmark. Whenever it reaches its timeout or if a crash occurs, a function is called to send to the Host the results of the injection and, after that, a new message (the *STOPITCHAR* in Figure 2) to

inform the Host about the end of the injection experiment.

Eventually, the DUT performs a software reset to go back on waiting for a new injection, and the host logs the results and updates the injection parameters.

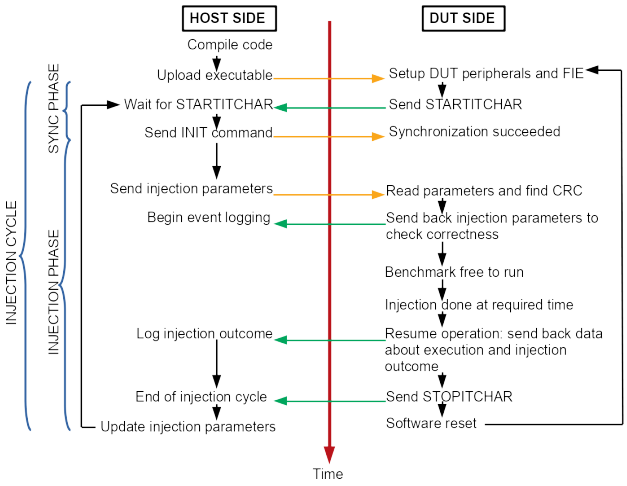

Fig. 2: Fault Injector Environment Communication Protocol

1) Injection Targets: The selection of the injection targets is a challenging job when dealing with an OS. For this reason, we provide the analysis of the impact of faults into the main components of the FreeRTOS by grouping structures and variables according to their usage. For each target, we also specify the number of fault locations.

- **FreeRTOS global kernel variables** (*GKVARS*): Global variables are used by the kernel to share data among the various kernel functions and to save the state of the system. They comprise 10 fault locations.
- **Task control block (TCB) structure**: TCB is a structure instantiated for each task, which contains all important data to guarantee the proper execution of the task within the OS, like the context switching. Experiments have been performed in the TCB of running tasks (*CURTCB*) and in the TCB of ready tasks (*RDYTCB*). They represent the same data structure allocated in two different memory address, including 19 fault locations.
- **Tasks list**: The state of a task is determined on the basis of the state list the task belongs to. Injections have been made in the ready task list (*RDYLST*) and in the delayed task list (*DLDLST*). Both lists are generated using the same data structure, defining 5 fault locations each.
- **Mutex and Queue structure** (*MTXQVARS*): The queue is a structure that can be used as semaphore or mutex by the kernel. It is composed of 22 fault locations.

Once the fault location has been selected, the injection time is randomly generated. The bit in the location to be affected by the fault is randomly generated too.

2) Classes of misbehavior: Every injection could lead to different types of misbehavior of the system; nevertheless, only some of them can be actually identified and classified because of some limitations in the tracing capabilities of the injection environment. All times the system continues working in the expected way, without showing any appreciable difference with respect to the golden run after the injection, the effect is classified as *OK*. It usually means that the error has been masked during the execution [20]. Instead, misbehavior are classified according to the following classes:

- **Crash**: When a critical error occurs on the DUT, the internal reset handler is called in order to avoid further problems. In this case, the misbehavior is classified as *crash*.
- **Freeze**: A misbehavior is classified as *freeze* when the whole system stops working and it does not respond to any regular input or event. In this event, only interrupts are executed, then, when the ISR returns, the system goes back to the frozen state.
- **Silent Data Corruption (SDC)**: A misbehavior is classified as *SDC* when only a part of the system shows a behavior that is substantially different from the expected one: this means that only a part of the system is blocked or acts incorrectly during the execution while the rest is still able to run properly. In order to check if the computation is correct, each different benchmark is provided by a CRC based on the output values of a golden execution. The CRC is evaluated against the one calculated on the output values when the main algorithm ends and a difference allows to detect an SDC.

IV. Experimental Results

The actual version of the Fault Injection Environment has been deployed on an STM32F3DISCOVERY (STMicroelectronics®) board running FreeRTOS. The board features an STM32F303VCT6 micro-controller based on the ARM® Cortex® M4 Architecture, working at 72MHz, and an ST-LINKv2 debugger interface, while the Host machine is a Linux-based operating system. In order to perform experiments using standard programs, to ensure repeatability of the injections, we resort to the EEMBC® Automotive suite [21]: a set of benchmarks that reproduce some very common calculations in the automotive field. They are developed to be used within an enclosing environment written specifically for UNIX/Linux systems called Multi-Instance Test Harness (MITH), that allows to instantiate a chosen subset of benchmarks among the given ones in the same run and to tune the parameters of the execution. These benchmarks target multi-core processors to test the scalability of the platform used, analyzing the distribution of the workload among the different cores, and eventually helping to find bottlenecks in the schedule and in the execution. The full set of benchmarks comprises 16 applications and, among them, three benchmarks have been extensively tested:

- **a2time** (Angle to time conversion): this benchmark simulates an engine with different cylinders (4, 6 or 8, to be chosen before compilation) with a crankshaft, a toothed reluctor wheel and a sensor able to generate a pulse every time it detects the passage of a tooth: this type

of mechanism is used to control the fuel injection in the cylinders and the subsequent spark.

- **tblook** (Table lookup and interpolation): a table lookup algorithm to store a limited amount of data pairs, coming from one or more resources (sensors, connections, calculations) and interpolates missing pairs. It is commonly used in embedded system when the memory resources are limited and only portions of data can be stored.
- **idctrn** (Inverse Discrete Cosine Transform): the implementation of the Inverse Discrete Cosine Transform widely used in digital graphics; it is applied to an input dataset representing a matrix of 64 bits values.

Since the performance analysis was not relevant for this work, they have been used in a single core micro-controller.

All injections have been done at random times. Table I reports the total number of injected faults per location list. They are equally distributed among all the locations. All reported numbers of injections have been computed by using the approach presented in [22] to obtain statistically significant results with an error margin of 1% and a confidence level of 95%. In order to guarantee a fair analysis, all injections have been made randomly within a window of execution, different for each group of locations, also reported in Table I. The difference in the window duration is related to the usage of each location group: for GKVARS a longer period is required in order to better catch the fallout of the fault (and its propagation within the system), while for other groups, shorter periods still ensure a good analysis capability, i.e., they are used more often or their compromise is highly disruptive.

A. Results Analysis

In general, experimental results shown that FreeRTOS is affected by some vulnerabilities. As it can be expected, most critical vulnerabilities are pointers, as well as numerical indexes stored in integer variables (both signed and unsigned) used to address elements of lists or vectors.

Understanding the capabilities of the fault injector requires to carefully analyze the results from different key aspects. Figure 3 gives a first perspective of the overall response when different benchmarks are executed. It is a very important first step in understanding the value and the quality of the Fault Injector because it shows how crashes and freezes do not depend on the benchmark, while numerous faults (almost 70%) leads to a fully responsive system, with errors surfacing as SDC, depending on the benchmark. This is a significant result since the operating system interacts with all threads in the same way, while the specificity of the benchmark explains the huge difference between **tblook** and the other two.

From a different perspective, Figure 4 reports the classification of the fault injection over the six target location groups, without distinction among benchmarks. A first interesting outcome is that the disruption of the content of GKVAR group does not generate any Crash, while most of the time the system is degrading into a freeze state. Probably less surprising, the most sensible group to faults is the ready list group (RDYLST), which almost always leads to crashes, because it

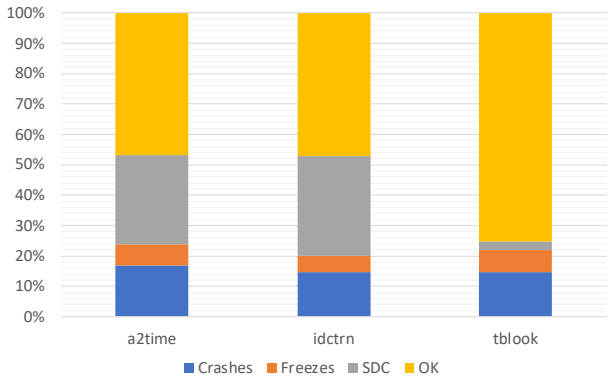

Fig. 3: Classification summary across the three benchmarks including all target locations

includes all data required to properly schedule alive threads. The other thread list (DLDLST) goes into an inoperative state, i.e., crash or freeze, in less than 50% of the injected faults, while most of the time an SDC is generated, making the OS more resilient than expected. Another peculiarity is the effect of a fault into a thread control block (either for the current task or the ready ones) where no SDC ever produced and around 40% of the time the OS never freezes or crashes, with a slightly worse scenario when the fault appears in the CURTCB. Eventually, MTXQVARS group shows a prevalence of SDCs among other classes and, less than 30% of crashes and freezes as expected since they include locations controlling mutex and semaphores.

All those considerations already suggest that FreeRTOS may be hardened in different ways. For example by simply duplicating or triplicating the most sensitive data, we will certainly improve the reliability but results show that a full duplication or triplication would be excessive. Moreover, the voting systems is going to add some computational overhead to all kernel procedures, which might contrast the real-time requirements of the OS. This is why the proposed fault injection framework can help in carefully selecting sensitive data, reducing the impact on the OS size and performances.

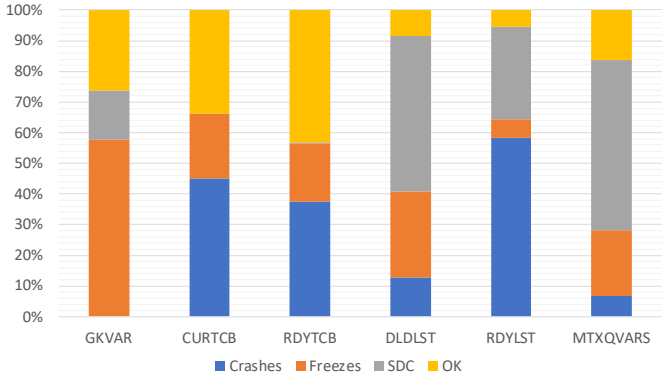

Fig. 4: Classification summary across the different target location groups

List	# Locations	# Injections	Injection Window (seconds)	Locations
GKVARS	10	4000	[0-5]	uxCurrentNumberOfTasks, xTickCount, uxTopReadyPriority, xSchedulerRunning, uxPendedTicks, xYieldPending, xNumOfOverflows, uxTaskNumber, xNextTaskUnblockTime, uxSchedulerSuspended
CURTCB	19	7600	[0-3]	pxTopOfStack, uxPriority, pxStack, uxTCBNumber, uxTaskNumber, uxBasePriority, uxMutexesHeld, ulNotifiedValue, ucNotifyState, xStateListItem, xItemValue, xStateListItem.{pxNext, pxPrevious, pvOwner, pvContainer, xItemValue} xEventListItem.{pxNext, pxPrevious, pvOwner, pvContainer}
RDYTCB	19	7600	[0-2]	*[same as previous one]*
DLDLST	5	2000	[0-3]	uxNumberOfItems, pxIndex xListEnd.{xItemValue, pxNext, pxPrevious}
RDYLST	5	2000	[0-3]	*[same as previous one]*
MTXQVARS	22	8800	[0-2]	pcHeadpcTail, pcWriteTo, u.pcReadFrom, u.uxRecursiveCallCount xTasksWaitingToSend.{uxNumberOfItems, pxIndex, xListEnd.xItemValue, xListEnd.pxNext, xListEnd.pxPrevious} xTasksWaitingToReceive.{uxNumberOfItems, pxIndex, xListEnd.xItemValue, xListEnd.pxNext, xListEnd.pxPrevious} uxMessagesWaiting, uxLength, uxItemSize, cRxLock, cTxLock, uxQueueNumber, ucQueueType

TABLE I: Injection lists summary, including the total number of fault locations (variable and structure fields), the total number of injection for each list, the window of injection and the full set of locations name as in the RTOS

V. Conclusions

This paper reports the study on the dependability of a Real-Time operating system by building a Fault Injection Environment able to target data belonging to the operating system and to trace the effects on the full system. The injections are done in a real environment using a host machine to prepare each injection and post-processing the results and a hardware board infrastructure to run the system under investigation and to make the injection. The results of several fault injection campaigns allowed, to the best of our knowledge, to investigate which data structure are more sensitive to SEUs, proving that the OS has some inherent resiliency, and thus supporting future developments such as a selective OS hardening based on FI data.

References

[1] C. Chen *et al.*, "Deepdriving: Learning affordance for direct perception in autonomous driving," in *2015 IEEE International Conference on Computer Vision (ICCV)*, Dec 2015, pp. 2722–2730.

[2] A. Vallero *et al.*, "Syra: Early system reliability analysis for cross-layer soft errors resilience in memory arrays of microprocessor systems," *IEEE Transactions on Computers*, vol. 68, no. 5, pp. 765–783, May 2019.

[3] N. J. Wang *et al.*, "Examining ace analysis reliability estimates using fault-injection," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ser. ISCA '07. New York, NY, USA: ACM, 2007, pp. 460–469. [Online]. Available: http://doi.acm.org/10.1145/1250662.1250719

[4] Freertos. [Online]. Available: https://www.freertos.org/index.html

[5] M. Kooli *et al.*, "A survey on simulation-based fault injection tools for complex systems," in *2014 9th IEEE International Conference on Design Technology of Integrated Systems in Nanoscale Era (DTIS)*, May 2014, pp. 1–6.

[6] M. Ebrahimi *et al.*, "A fast, flexible, and easy-to-develop fpga-based fault injection technique," *Microelectronics Reliability*, vol. 54, no. 5, pp. 1000–1008, 2014.

[7] J. Carreira *et al.*, "Xception: A technique for the experimental evaluation of dependability in modern computers," *IEEE Transactions on Software Engineering*, vol. 24, no. 2, pp. 125–136, February 1998.

[8] G. A. Kanawati *et al.*, "Ferrari: A flexible software-based fault and error injection system," *IEEE Trans. Comput.*, vol. 44, no. 2, pp. 248–260, Feb. 1995.

[9] F. Jianyong, "Research on the nonintrusive resource level fault injection technology for windows system," in *2016 IEEE Trustcom/BigDataSE/ISPA*, Aug 2016, pp. 1856–1860.

[10] J. H. Barton *et al.*, "Fault injection experiments using fiat," *IEEE Transactions on Computers*, vol. 39, no. 4, pp. 575–582, April 1990.

[11] J. Arlat *et al.*, "Dependability of cots microkernel-based systems," *IEEE Transactions on Computers*, vol. 51, no. 2, pp. 138–163, Feb 2002.

[12] Weining Gu *et al.*, "Characterization of linux kernel behavior under errors," in *2003 International Conference on Dependable Systems and Networks, 2003. Proceedings.*, June 2003, pp. 459–468.

[13] E. Jeong *et al.*, "Fifa: A kernel-level fault injection framework for arm-based embedded linux system," *10th IEEE International Conference on Software Testing, Verification and Validation*, pp. 23–34, 2017.

[14] P. Tröger *et al.*, "Software-implemented fault injection at firmware level," in *2010 Third International Conference on Dependability*, July 2010, pp. 13–16.

[15] D. Silva *et al.*, "A hardware-based approach for fault detection in rtos-based embedded systems," in *2011 Sixteenth IEEE European Test Symposium*, May 2011, pp. 209–209.

[16] B. Montrucchio *et al.*, "Software-implemented fault injection in operating system kernel mutex data structure," in *2014 IEEE 5th Latin American Symposium on Circuits and Systems*, Feb 2014, pp. 1–6.

[17] N. Alimi *et al.*, "An rtos-based fault injection simulator for embedded processors," *International Journal of Advanced Computer Science and Applications*, vol. 8, no. 5, pp. 300–306, 2017.

[18] S. C. Lee *et al.*, "rostest: Universal test framework for real-time operating system," in *2016 IEEE 25th Asian Test Symposium (ATS)*, Nov 2016, pp. 129–129.

[19] H. Chen *et al.*, "Reliability demonstration testing method for embedded operating systems," in *The 2nd International Conference on Software Engineering and Data Mining*, June 2010, pp. 272–275.

[20] A. Savino *et al.*, "Statistical reliability estimation of microprocessor-based systems," *IEEE Transactions on Computers*, vol. 61, no. 11, pp. 1521–1534, 2012.

[21] EEMBC, *Autobench - Software benchmark data book*. www.eembc.org: EEMBC, 2015.

[22] R. Leveugle *et al.*, "Statistical fault injection: Quantified error and confidence," in *2009 Design, Automation Test in Europe Conference Exhibition*, April 2009, pp. 502–506.