Toward the hardening of real-time operating systems

Alberto Bosio[1], Stefano Di Carlo[2], Maurizio Rebaudengo[2] Alessandro Savino[2]
[1]*Univ Lyon, ECL, INSA Lyon, CNRS, UCBL, CPE Lyon, INL, UMR5270, 69130 Ecully, France*
[2]*Politecnico di Torino, Dip. di Automatica e Informatica, Torino, Italy*
Email: [1]alberto.bosio@ec-lyon.fr, [2]{name.surname}@polito.it

***Abstract*—Safety and Mission-critical systems are evolving daily, requiring increasing levels of complexity in their design. While bare-metal single CPU systems were dedicated to such systems in the past, nowadays, multicore CPUs, GPUs, and other accelerators require more complex software management, with the need for an operating system controlling everything. The presence of the operating system opens more challenges to securing the final system's full dependability. This paper analyses the hardening scenarios based on the evidence gathered by selective fault injection analysis of Real-Time Operating systems. While solutions might be delivered in different fashions, the emphasis on the paper is on the right approach to spot the sensitive part of the Operating system, saving the design from massive overheads.**

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

I. Introduction

Safety- and Mission-critical systems, such as aerospace and avionics, deal with either human lives or (and) high-cost equipment; therefore, they require high reliability, where the term "high" is usually quantified according to standards, such as the ISO 26262 for automotive [1]. Said systems are often exposed to space radiation. Moreover, even systems operating at ground level, like automotive, can be subject to space radiation [2]. The study of radiation effects in semiconductor devices is a well-known field of research, as reflected by its vast literature. To resume what happens when a high-energy particle impacts a MOS transistor, we can assume that it may result in a status change: from "on" to "off" or *viceversa* [2]. When the impacted MOS is part of a memory element (e.g., SRAM, register, etc.), this turns into a single-event upset (SEU), also known as a single-event error (SEE): the logical value stored in the memory element flips. The occurred SEE may be propagated through the different system layers as depicted in Figure 1. When the error reaches the application and impacts system outputs, it becomes a "failure".

The failure can be categorized into two major classes: SDCs (silent data corruption) and FIs (functional interruption) [4], [5]. An SDC occurs when the application properly finishes (i.e., without any crash or delay), but its outputs differ from the expected ones. FIs are considered when the application hangs or terminates unexpectedly. In the context of Safety/Mission-critical applications, SDC is the most critical class of failure since the end-user has no means to understand that something went wrong.

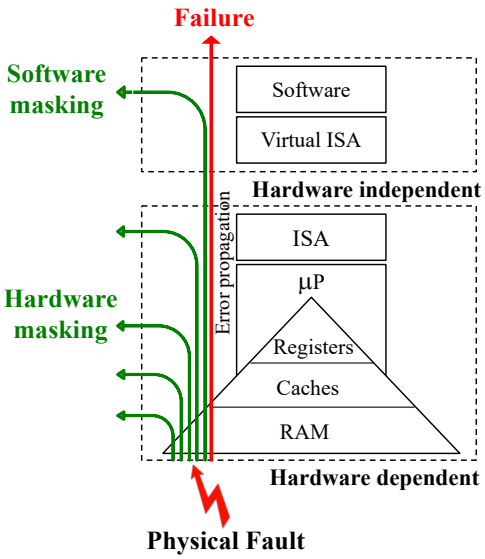

Fig. 1. Fault - Error - Failure [3]

Usually, to guarantee high dependability, hardening techniques have been applied to electronic devices [3]. Such methods were mainly used at the hardware level to add structural redundancy, meaning that specific designs must be manufactured. Despite the efficiency of hardening techniques, the design and manufacturing cost is usually very high. The industry has turned to commercial off-the-shelf (COTS) devices because of their relatively low cost and high performance. On the other hand, COTS does not provide hardening; therefore, they require a further implementation of fault tolerance techniques at the SW level (i.e., without modifying the hardware).

Indeed, performance has become a more critical design metric to meet the escalation of application constraints, like object detection on autonomous driving [6]. For this reason, COTS are growing in complexity, including multi-core systems that integrate more than one Central Processing Unit (CPU) and Graphical Processing Unit (GPU) on the same chip. As a direct consequence, parallel programming paradigms can be deployed, enabling a significant increase in computational throughput. Moreover, the whole system stack usually requires to include, between the hardware layer and the software/application layer, a middle layer composed of the Operating System (OS) and middleware (e.g., peripheral drivers) to deploy the

final application properly.

In the literature [3], the application layer is commonly the target of the reliability analysis. The assumption behind bare-metal system scenarios is that the probability of a fault affecting the middle layer is much lower than the application level due to its short execution time (i.e., system calls execution time is generally faster than the mission-related application). However, 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 crashes, missed deadlines, application freeze as well as tasks synchronization errors.

The aim of this work is thus to review the literature investigating the reliability of Real-Time Operating systems (RTOS) affected by external perturbations. The paper will then draw perspectives on improving reliability using selective fault tolerance techniques applied at the software level.

II. Background

This section presents the basics of reliability assessment through fault injection (FI) campaign and its use to assess operating system reliability. FI is a methodology where the system is forced to behave affected by faults. The outcome of FI is the Failure in Time (FIT) rate. The following characteristics can define FI:

- System definition: it can be a simulable model or a hardware prototype. Based on the system definition, the FI technique can be simulation-based or hardware-based;
- Type of fault model: it could be a bit-flip in a memory location, a resistive short, or a specific source of perturbation like radiation;
- Type of injection mechanism: it can be a modification of the system model allowing to modify the behavior accordingly to the fault model or the facility used to create the perturbation (i.e., neutron beams);
- Controllability/Observability: it defines how precisely the fault model can be injected into the system and what can be observed (i.e., only the final output or internal system state).

Further details about FI are out of the scope of this paper. The reader can refer to [3] for more information.

Not all the fault injectors in the literature address the operating system layer similarly. Some of them, like [7], mimic an error in the OS by intercepting and disrupting the API level, i.e., the interaction between the application and the OS (e.g., a system call). The result leads to the investigation of a small set of faults that affect only the data provided by the OS to the application.

One of the first effective but dated contributions that can be found is [8]. While the main contribution is a set of fault tolerance recovery schemes, the authors developed a fault injection methodology within the operating system. Since the work does not intend to provide fault coverage, it is unclear whether the kind of error can be injected or the affected part of the operating system itself.

Recent works, like [9], [10], [11], [12] target specific OS data structure to evaluate the impact of a fault to the full system. Authors in [9] and [12] seem to cover most of OS data structure but both papers do not provide any experimental results. [10] addresses the synchronization capabilities via mutex OS structures, while in [11] a completely different scenario is presented: to evaluate the capability of tampering with the OS, using a Differential Fault Attack (DFA), a specific fault injector is proposed. Due to the scope of the work, the only user data memory is interested in the fault injector, missing all the system data.

In [13] authors built a fault injector based on the remote features of kernel debugger (KGBD). They addressed an ARM-based system running a Linux OS. The main limitation is that the injection can be done only when the operating system reaches a specific part of the code, thus limiting the time selection capability of the whole fault injector.

On the side of reliability analysis of OS, to the best of our knowledge, only one paper addressed the matter. In [14], authors resort to a Bayesian network model of the internal states of a real-time operating system (CRTOS II) to predict its reliability. They support their claim by providing an experimental setup where several failures are applied. Unfortunately, the paper does not explain the fault model behind the investigation and how authors produce such failures, limiting the analysis capability.

To provide a more comprehensive analysis of the reliability of the whole system in safety-critical scenarios, we merge the contribution of two works to expand the depth of the OS analysis. In [15] a reliability comparison between executing a given application as bare metal or on the top of an operating system has been made. This paper proved the OS's importance in the target system's reliability. The goal of the evaluation was to determine the impact of the OS on application reliability by measuring the amount of time a failure was observed at the application outputs. Fault Injection was carried out using OVPSim-FIM (OVP Fault Injection Module) [16]. Faults were bitflips injected on the processor registers. Two main sets of benchmarks provide support to the analysis: (i) Successive Approximation (SA) algorithms and (ii) General Purpose (GP) algorithms. The two sets have different intrinsic resilience to SEE. SAs are more resilient than GPs because of their inherent redundancy [17]. More in detail, we have three other SAs: Newton-Raphson and Trapezoid, both useful to compute the integral of a function, and the QSolver for root computation of quadratic equations. The GPs set comprises four benchmarks: Matrix Multiplication, Vector Sum, and Tower of Hanoi puzzle solver benchmarks. Each benchmark was implemented as bare metal, a task executed through FreeRTOS, and a process executed on top of Linux.

Figure 2 reports the average results for all the above-described benchmarks. For each implementation type (i.e., bare metal, FreeRTOS, and Linux), we computed the average number of observed failures and masked (i.e., when the observed outputs were correct w.r.t. the golden). The chart indicates two main things. The first one is that the OS matter

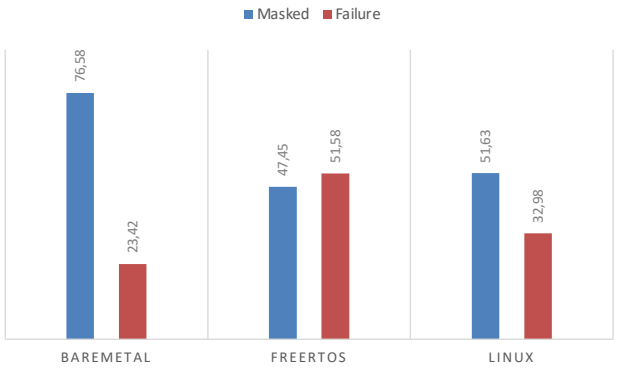

Fig. 2. OS impact of failures

when reliability is considered. The number of failures almost doubled from bare metal to OS implementations. The second consideration is that FreeRTOS is less reliable.

Starting from these results, we performed a deep analysis of the reliability of FreeRTOS alone [18]. The obtained results will be discussed in the next section, and they will be used to propose research directions to increase the reliability of FreeRTOS.

III. FreeRTOS reliability analysis

In [18] we presented the results obtained by injecting SEE in memory locations storing FreeRTOS data structure. For the sake of readability, we summarise the injection targets as follows:

- **FreeRTOS global kernel variables**: Global variables are used by the kernel to share data among the various kernel functions and to save the system's state.
- **Task control block (TCB) structure**: The OS's classical TCB structure is required to support the multitasking activity. Experiments have been performed in the TCB of running tasks and the TCB of ready tasks.
- **Tasks list**: In the FreeRTOS, the state of a task is determined based on which state list the task belongs to. Injections have been made in the ready task list and the delayed task list.
- **Mutex and Queue structure**: Queue is a structure that can be used as a semaphore or mutex by the kernel.

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

Every injection could lead to different types of (mis)behaviors in the system; nevertheless, only some of them can be identified and classified because of some limitations in the tracing capabilities of the injection environment. Results have been divided into four categories: *crash*, *freeze*, *degradation* and *silent misbehavior*.

- **Crash**: When a critical error occurs on the device under test, the internal reset handler is called 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 does not respond to any regular input or event. In this event, only interrupts are executed; when the ISR returns, the system returns to the frozen state.
- **Degradation**: A misbehavior is classified as *degradation* when only a part of the system shows a behavior substantially different from the expected one: this means that only a part of the system is blocked or acts incorrectly during the execution. At the same time, the rest is still able to run properly.
- **Silent misbehavior**: This type of misbehavior happens when, after the injection, the system continues working expectedly, without showing any appreciable difference concerning the golden run.

For the Freeze and Degradation classes, the decision is made by resorting to a threshold (called *tolerance*), representing the allowed percentage of the difference between the timing of the golden and the fault injected run. This percentage is a parameter of the fault injection campaign.

In this paper, we extended the analysis of the fault injection results we obtained by using the approach presented in [18]; final injection classes are evaluated using the following metrics:

1) *Consistency*:

$$C = \frac{Number\ of\ misbehavior}{Number\ of\ injections} \tag{1}$$

It allows understanding of how much the target location is sensitive to an injection, i.e., how many times it produces misbehavior when the fault is activated. If this value is 0, the system is not susceptible to the injection in that location.
All the other metrics must always be compared to the Consistency, as it allows us to understand the overall impact on the system.

2) For each class, excluding the Silent Misbehavior, we compute a specific rate as follows:

$$CR = \frac{Number\ of\ crashes}{Number\ of\ misbehavior} \tag{2}$$

$$FR = \frac{Number\ of\ freezes}{Number\ of\ misbehavior} \tag{3}$$

$$DR = \frac{Number\ of\ degradation}{Number\ of\ misbehavior} \tag{4}$$

Each rate covers a different analysis aspect of the OS behavior in the presence of a fault. The *Crash ratio* (CR) weights the number of crashes over the total number of misbehavior. Thus the higher this value is, the more the fault location is critical for the RTOS execution. Similarly, the *Freeze ratio* (FR) and the *Degradation ratio* (DR) show a decreasing sensitivity of the fault location since they a linked to less critical OS behaviors.

A. Benchmarks

To perform experiments using standard programs and ensure the injections' repeatability, we resort to the EEMBC®

Automotive suite [19] performance suite that reproduces some very common calculations in the automotive field. They have been used in a single core micro-controller because the performance analysis was irrelevant for this work. The benchmarks comprise 16 applications: 13 have been ported successfully to the embedded environment. The remaining three have been discarded because they require a large memory space that the existing system does not support.

Three elements define each benchmark:

- **Workload** It is the algorithm executed, which can be single-thread or multi-thread.
- **Dataset** The input data is provided in the form of files. Each one comes with a different number of input values. Almost all benchmarks can run with a large dataset (about 4MB of data) and a small dataset (about 4kB of input data). In some cases, we were forced to reduce the number of input values.
- **CRC** To check if the computation is correct, we compute pre-calculated CRCs, one for each different dataset. These CRCs are embedded in the code and are used by a checking function, which performs the CRC calculation on the fly on the output values and compares it with the given one. Such function has to be called when the main algorithm ends.

IV. Experimental Results

The actual Fault Injection Environment has been deployed on a STM32F3DISCOVERY (STMicroelectronics®) board running FreeRTOS. Three benchmarks (see Section III-A) have been extensively tested and reported here:

1) *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 a tooth: this type of mechanism is used to control the injection of fuel in the various cylinders and the subsequent spark.
2) *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.
3) *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 systems when memory resources are small and only portions of data can be stored.

All injections have been done at random times. Table I contains the total number of injected faults per location list. In the table header, each list identifies a different set of target locations: *GKVARS* includes the global kernel variables, while *MTXQVARS* is the list of the synchronization structures. The other lists are related to task management: *CURTCB* identifies the current task TCB, while *RDYTCB* represents the ready task TCB, *DLDLST* the delayed task list, and *RDYLST* the ready tasks list. All injections have been computed using the approach presented in [20] to obtain statistically significant results with an error margin of 1% and a confidence level of 95%.

TABLE I
Number of injected faults per target location list

List	#Injections
GKVARS	5000
CURTCB	9500
RDYTCB	9500
DLDLST	2500
RDYLST	2500
MTXQVARS	11000

Based on the complete set of fault injections, among all benchmarks, Table II reports a summary of the most sensitive locations to the fault injection campaign, i.e., the ones showing the highest consistency 1. All entries marked with a $*$ correspond to pointers variables.

To fully understand the results reported in Table II, we need to detail the effects of the injected faults:

- *uxTopReadyPriority* represents the task with the highest priority, so selecting the ready task with the highest priority in the vector pxReadyTasksLists is done trying to access an utterly wrong memory address. This always leads to a crash;
- *uxPendedTicks* represents all missing ticks for the current task. Due to the injection, the stored value can become enormous, leading to the freeze of the system;
- *uxSchedulerSuspended*: this variable is always checked by an internal assertion before using it: in all cases, if its value is different from the expected one, the system is blocked by an infinite loop;
- *xStateListItem.pvOwner*: this variable contains the pointer to the TCB of the following item to be switched in. Once the injection corrupts the pointer value, it will likely generate a crash.
- Interestingly, faults locations like *xStateListItem.pxNext*, *xStateListItem.pxPrevious*, *xStateListItem.pvOwner*, *xStateListItem.pvContainer* and *xEventListItem.pvContainer*) have almost a fixed consistency and, in the most of cases they produce a crash. The reason lays that they belongs to the task in the ready queue, which is more prone to scheduler selection, thus some fault injection might lead to silent misbehavior;
- *xListEnd.pxNext* is used in delayed-to-ready state transition when a task has reached its timeout delay and is required to move back to the ready state list. Each time the value is corrupted, the change cannot be completed correctly, leading to a crash.
- *uxNumberOfItems* is an integer variable containing the number of elements in the list: injecting there leads mainly to crashes. Indeed, this variable is often accessed by the kernel, particularly during the contest switching task, which checks the length of the ready list only to exclude that it reached a 0 value. With a wrong number

TABLE II
SUMMARY OF MOST SENSITIVE LOCATIONS

Fault Location list	Fault Name	Consistency
GKVARS	uxTopReadyPriority	$C = 100\%$
	uxPendedTicks	$C = 100\%$
	uxSchedulerSuspended	$C = 100\%$
CURTCB	*xStateListItem.pxNext	$C = 100\%$
	*xStateListItem.pvOwner	$C = 100\%$
RDYTCB	*xStateListItem.pxNext	$32\% < C < 100\%$
	*xStateListItem.pxPrevious	$32\% < C < 100\%$
	*xStateListItem.pvOwner	$32\% < C < 100\%$
	*xStateListItem.pvContainer	$32\% < C < 100\%$
	*xEventListItem.pvContainer	$32\% < C < 100\%$
DLDLST	*xListEnd.pxNext	$C = 100\%$
RDYLST	uxNumberOfItems	$C = 100\%$
	*pxIndex	$C = 100\%$
MTXQVARS	xTasksWaitingToSend.uxNumberOfItems	$36\% < C < 100\%$
	xTasksWaitingToReceive.uxNumberOfItems	$36\% < C < 100\%$
	uxItemSize	$36\% < C < 100\%$

that differs from 0, the switch is done using the incorrect data.

- *pxIndex* produces crashes too: this happens because this variable is a pointer to the last item inserted in the list. When modified, the kernel tries to access the wrong memory region. Moreover, its value is used to update other pointers, making the disruption of the value even more severe;
- *xTasksWaitingToSend.uxNumberOfItems* always causes freezes because it contains the number of tasks waiting to release the mutex.
- *xTasksWaitingToReceive.uxNumberOfItems* is the opposite of *xTasksWaitingToSend.uxNumberOfItems* as it is a value which represents the number of tasks waiting to hold the mutex;
- *uxItemSize* causes always freezes. Since it holds the size, in bytes, required to hold each item in the queue, when it is altered in a way that contradicts the real size of the items, the synchronization process is not able to run properly, preventing the system from running tasks waiting for a mutex to be released.

V. DISCUSSION

Experimental results demonstrate that FreeRTOS is affected by some vulnerabilities. Most critical vulnerabilities are pointers and numerical values stored in integer variables (both signed and unsigned) used to address elements of lists or vectors. Thanks to the observed results, it is possible to identify directions for making FreeRTOS more robust through hardening techniques.

A. Consistency dependence on evaluation tolerance

The injection classification methodology exposes all results to a potential dependence on the tolerance used to detect misbehavior classes like freezes and degradation. Such relation can impact the final evidence depending on the tolerance used to analyze the results. Different tolerance values make the analysis more or less sensitive to differences between the golden and injection runs.

We analyzed such dependency, which is visible in a restricted subset of faults injected in the FreeRTOS global variables. In contrast, other fault locations give more homogeneous results for different tolerance values. Figure 3 shows how consistency increases for lower values of tolerance applied to freezes for the *idctrn* benchmark. Even small deviations are classified as misbehavior in those cases, although they are not necessarily due to the injection.

It is interesting to notice that only fault locations related to the real-time scheduling of tasks depend on tolerance: variables like xTickCount and xNextTaskUnblockTime are used to choose when a task must be moved back from delayed to ready state. On the contrary, uxTopReadyPriority is a variable exploited to identify, at every system tick, which is the highest priority available in the OS. It does not change its impact on consistency when the tolerance is altered.

B. FreeRTOS hardening

FreeRTOS can be hardened in different ways. One of the best methods is introducing a certain level of redundancy, especially in those places where most critical problems occurred during experiments. All the most sensitive data must be duplicated or triplicated, and a voting system must be implemented, adding some computational overhead to all kernel procedures, which might contrast the real-time requirements of the OS.

Error correction mechanism would be preferable when a high level of reliability has to be reached. If the application has to show a high availability, a simple error detection system would be sufficient, forcing a software reset or isolating the affected component of the system if a critical error is detected. It is essential to notice that FreeRTOS already includes some macros that must be implemented by the programmer to perform checks on the integrity of data in the system or in parameters passed to kernel functions before using them. Moreover, the RTOS already provides an asserting macro to program the detection of critical values in system data.

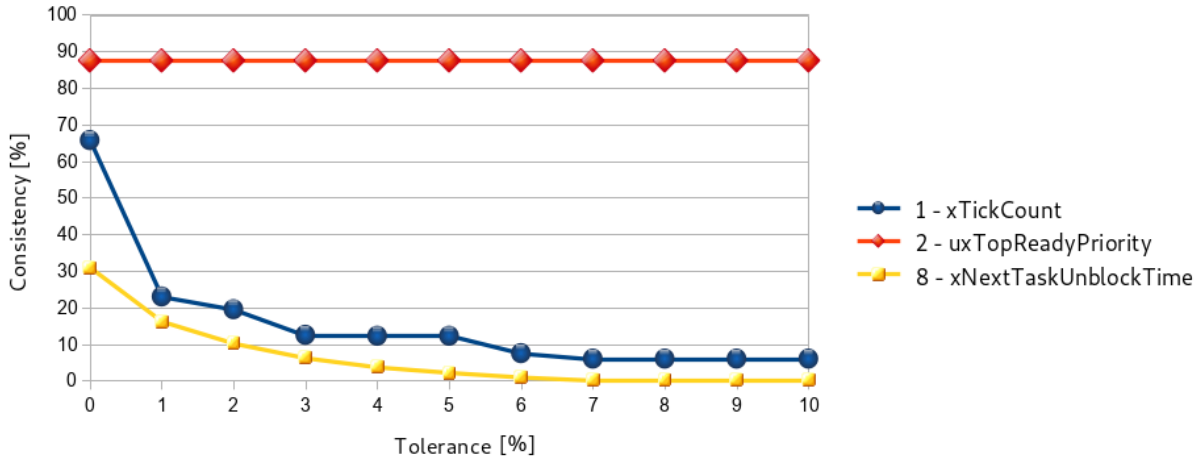

Fig. 3. Tolerance - Consistency dependency analysis for *idctrn* benchmark

VI. Conclusions

This paper presented a detailed analysis of the Operating System's impact on the safety-critical systems' reliability. The study resorted to two previous works that detailed the sensitivity to soft errors at the operating system level. Then, we introduced a further level of analysis by defining a consistency definition and consequent rates to tackle the sensitivity to errors of each part of the OS. This consistency can also support a deeper evaluation of classes of error's effects that might rely on a time-dependent definition by showing some tolerance when relaxed.

Eventually, the analysis opens to a more selective design of the hardening techniques. It inspires a new set of tools for a more comprehensive analysis of the faults in the system.

References

[1] ISO, *ISO 26262 Road Vehicles - Function Safety*, International Standardization Organization Std., Dec. 2018.

[2] R. Baumann, "Radiation-induced soft errors in advanced semiconductor technologies," *IEEE Transactions on Device and Materials Reliability*, vol. 5, no. 3, pp. 305–316, 2005.

[3] G. D. Natale, D. Gizopoulos, S. D. Carlo, A. Bosio, and R. Canal, Eds., *Cross-Layer Reliability of Computing Systems*. Institution of Engineering and Technology, Oct. 2020. [Online]. Available: https://doi.org/10.1049/pbcs057e

[4] M. Portolan, A. Savino, R. Leveugle, S. Di Carlo, A. Bosio, and G. Di Natale, "Alternatives to fault injections for early safety/security evaluations," in *2019 IEEE European Test Symposium (ETS)*, 2019, pp. 1–10.

[5] A. Vallero, A. Savino, A. Chatzidimitriou, M. Kaliorakis, M. Kooli, M. Riera, M. Anglada, G. Di Natale, A. Bosio, R. Canal, A. Gonzalez, D. Gizopoulos, R. Mariani, and S. Di Carlo, "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.

[6] C. Chen, A. Seff, A. Kornhauser, and J. Xiao, "Deepdriving: Learning affordance for direct perception in autonomous driving," in *2015 IEEE International Conference on Computer Vision (ICCV)*, Dec 2015, pp. 2722–2730.

[7] 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.

[8] H. Yoshida, H. Suzuki, and K. Okazaki, "Fault tolerance assurance methodology of the sxo operating system for continuous operation," in *[1991] Proceedings Pacific Rim International Symposium on Fault Tolerant Systems*, Sep. 1991, pp. 182–187.

[9] D. Silva, K. Stangherlin, L. Bolzani, and F. Vargas, "A hardware-based approach for fault detection in rtos-based embedded systems," in *2011 Sixteenth IEEE European Test Symposium*, May 2011, pp. 209–209.

[10] B. Montrucchio, M. Rebaudengo, and A. David Velasco, "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.

[11] N. Alimi, M. Machhout, Y. Lahbib, and R. Tourki, "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.

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

[13] E. Jeong, N. Lee, J. Kim, D. Kang, and S. Ha, "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] H. Chen and Z. Qin, "Reliability demonstration testing method for embedded operating systems," in *The 2nd International Conference on Software Engineering and Data Mining*, June 2010, pp. 272–275.

[15] E. Casseau, P. Dobiáš, O. Sinnen, G. S. Rodrigues, F. Kastensmidt, A. Savino, S. Di Carlo, M. Rebaudengo, and A. Bosio, "Special session: Operating systems under test: an overview of the significance of the operating system in the resiliency of the computing continuum," in *2021 IEEE 39th VLSI Test Symposium (VTS)*, 2021, pp. 1–10.

[16] F. Rosa, F. Kastensmidt, R. Reis, and L. Ost, "A fast and scalable fault injection framework to evaluate multi/many-core soft error reliability," in *2015 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFTS)*, 2015, pp. 211–214.

[17] Q. Xu, T. Mytkowicz, and N. S. Kim, "Approximate computing: A survey," *IEEE Design Test*, vol. 33, no. 1, pp. 8–22, 2016.

[18] D. Mamone, A. Bosio, A. Savino, S. Hamdioui, and M. Rebaudengo, "On the analysis of real-time operating system reliability in embedded systems," in *2020 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, 2020, pp. 1–6.

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

[20] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert, "Statistical fault injection: Quantified error and confidence," in *2009 Design, Automation Test in Europe Conference Exhibition*, April 2009, pp. 502–506.