POLITECNICO DI TORINO
Repository ISTITUZIONALE

Experimental Analysis of FreeRTOS Dependability Through Targeted Fault Injection Campaigns

Original
Experimental Analysis of FreeRTOS Dependability Through Targeted Fault Injection Campaigns / Mannella, L., Carlo, S.D., Savino, A.. - ELETTRONICO. - (2026), pp. 1-6. (29th IEEE International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS 2026) Bratislava (Slovacchia) April 27–29, 2026) [10.1109/ddecs69233.2026.11520990].

Availability:
This version is available at: 11583/3011141 since: 2026-05-20T12:26:22Z

Publisher:
Institute of Electrical and Electronics Engineers (IEEE)

Published
DOI:10.1109/ddecs69233.2026.11520990

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

©2026 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)

18 November 2026

Experimental Analysis of FreeRTOS Dependability through Targeted Fault Injection Campaigns

Luca Mannella, Stefano Di Carlo, Alessandro Savino,
Department of Control and Computer Engineering, Politecnico di Torino, Turin, Italy
{name.surname@polito.it}

***Abstract*—Real-Time Operating Systems (RTOSes) play a crucial role in safety-critical domains, where deterministic and predictable task execution is essential. Yet they are increasingly exposed to ionizing radiation, which can compromise system dependability. To assess FreeRTOS under such conditions, we introduce KRONOS, a software-based, non-intrusive post-propagation Fault Injection (FI) framework that injects transient and permanent faults into Operating System (OS)-visible kernel data structures without specialized hardware or debug interfaces. Using KRONOS, we conduct an extensive FI campaign on core FreeRTOS kernel components, including scheduler-related variables and Task Control Blocks (TCBs), characterizing the impact of kernel-level corruptions on functional correctness, timing behavior, and availability. The results show that corruption of pointer and key scheduler-related variables frequently leads to crashes, whereas many TCB fields have only a limited impact on system availability.**

***Index Terms*—Embedded Systems, Real-Time Operating System, Fault Injection, Reliability Assessment, Single Event Upset, Harsh Environment**

I. Introduction

Real-time embedded systems have become the backbone of modern critical infrastructures, extending far beyond consumer electronics to underpin applications in automotive control, aerospace and satellite systems, and other platforms [1]. Their pervasive deployment in safety and mission-critical domains demands stringent guarantees of dependability, encompassing reliability, safety, and fault tolerance [2], [3]. To meet the diverse and demanding requirements of these applications, embedded devices typically rely on specialized software stacks. In some cases, this software is implemented as tightly coupled, application-specific firmware, while in others, it is built atop Real-Time Operating Systems (RTOSes) that orchestrate complex scheduling, resource management, and timing constraints.

Despite advances in design, embedded systems remain vulnerable to both internal faults, such as hardware defects, software bugs, and malicious tampering, and external threats from their operational environment [2]. Notably, transient and permanent faults, especially in space and high-altitude applications, have a significant impact, and their occurrence is increasing even at ground level as device geometries shrink [3]–[7]. In this context, radiation-induced effects such as Single Event Upsets (SEUs) and persistent hardware faults may propagate to kernel-visible state and compromise correct execution [4]. The growing vulnerability of modern embedded platforms to these phenomena has stimulated the development of robust reliability assessment and mitigation techniques [3].

Among the available methodologies, Fault Injection (FI) is considered a cornerstone for evaluating system resilience. By deliberately introducing faults (either in hardware, software, or at the system level), researchers can systematically study fault propagation, error detection, and recovery mechanisms under controlled conditions [8], [9]. While many FI studies focus on the effects of faults at the application level, some works also emphasize the importance of considering the RTOS kernel and its data structures, e.g., scheduler queues, timers, and Task Control Blocks (TCBs), since faults in this layer can significantly influence system behavior and safety [2], [9]. Nonetheless, a systematic analysis of RTOSes under transient and permanent faults, and of their implications for system dependability, remains an open research direction.

This paper investigates how transient and permanent faults affecting key RTOS kernel data structures influence system dependability once these faults propagate to the RTOS-visible state. The main contributions are: (1) design of **K**ernel-level Free**R**TOS **O**bject-oriented **N**on-intrusive **O**pen fault injection **S**ystem (KRONOS), a novel, software-based, FI framework for FreeRTOS [10]; (2) extensive kernel-level FI campaign on FreeRTOS, for both transient and permanent fault models in RTOS-visible kernel state. (3) detailed analysis of outcome distributions and identification of critical kernel data structures, facilitating early-stage vulnerability assessment and identifying the most sensitive FreeRTOS components.

The remainder of this paper is organized as follows: Section II reviews relevant related work on FI. Section III details the proposed FI methodology and its integration with FreeRTOS. Section IV describes the experimental setup, while Section V presents and discusses the results of our FI campaigns. Finally, Section VI concludes the paper.

II. Related Work

FI is a widely used technique for assessing system dependability by purposefully introducing faults into the system and observing its behavior under these engineered fault scenarios [11]–[15]. FI can be classified (by injection mechanism) into four categories: (i) physical, (ii) hardware-based, (iii) software-based, and (iv) model-based. *Physical* FI involves

This work was supported by project COLTRANE-V funded by the Ministero dell'Università e della Ricerca within the PRIN 2022 program (D.D.104 - 02/02/2022) and carried out within the Space It Up project funded by the Italian Space Agency, ASI, and the Ministry of University and Research, MUR, under contract n. 2024-5-E.0 - CUP n. I53D24000060005.

subjecting the system implementation to external phenomena, such as radiation. *Hardware-based* techniques exploit hardware interfaces, e.g., the debug interface, to modify register contents or memory. *Software-based* methods alter the software layer, targeting variables or memory buffers. Finally, *model-based* FI operates on abstract models, such as Hardware Description Languages (HDLs) or instruction-set simulators, enabling FI campaigns in simulation environments. More details on those techniques can be found in [14], [15].

Targeting the Operating System (OS) layer for FI is particularly challenging [16], [17], since faults affecting core kernel data can lead to unrecoverable errors and system hang. While similar effects may also occur when injecting faults at the application level, they tend to appear less frequently, and managing hangs is typically more time-consuming, adding complexity to FI campaigns [9].

Recent advances in model-based FI frameworks include gem5-MARVEL [18]. Built on the gem5 microarchitectural simulator [19], it enables FIs at a detailed hardware level across multiple Instruction Set Architectures (ISAs) and system components, including OS kernels within simulation environments. While it offers fine-grained microarchitectural fault modeling and OS-level FI capability, the approach entails long simulation times and complexity. More broadly, microarchitectural FI comes at the expense of long simulation times.

Prior work has extensively investigated FI at the hardware and microarchitectural levels, often highlighting the limitations of simplified high-level simulation approaches. Papadimitriou and Gizopoulos [7] performed FI within a processor's microarchitecture and demonstrated that purely software-level FIs techniques that flip bits in program variables can overlook essential microarchitectural effects, sometimes even reversing vulnerability rankings when compared to full-stack analyses. Similarly, Cho et al. [20] examined architecture- and memory-level FI methods, showing that while such techniques offer high execution speed, their error-rate estimates may deviate by an order of magnitude. Collectively, these studies underscore the inherent limitations of high-abstraction FIs in capturing realistic hardware error behavior and fail to consider fault-handling mechanisms at the OS level.

In contrast, our work targets the OS layer by injecting faults directly into RTOS kernel memory structures to evaluate the dependability of system software itself. By deliberately abstracting from microarchitectural modeling, the proposed approach focuses on analyzing the OS response to corrupted internal states—for instance, the behavior of task schedulers, kernel queues, and memory managers under fault conditions. Rather than estimating hardware fault rates, our objective is to uncover OS-level failure modes and robustness limitations that remain unobservable through hardware-centric FI. This perspective complements existing cross-layer fault analyses by elucidating how faults propagate and manifest within the OS.

However, other studies began targeting OS structures. Fault-Injection-based Automated Testing (FIAT) [21] was one of the earliest software-based FI tools targeting the kernel. It applied abstract data-corruption models to running tasks but lacked the resolution to selectively corrupt internal OS structures, such as the TCBs. Another microkernel-based framework is MAFALDA [22]. Even if it can inject faults into API arguments or microkernel memory segments, its architecture, which separates injection and execution across machines, offers limited insight into internal fault propagation. RTOS Guardian [23] introduced a hardware-based fault monitoring architecture for embedded RTOS systems. Its components non-intrusively monitor the system bus to validate RTOS scheduling integrity. Although effective in electromagnetic-induced faults detection per IEC 61000-4-29 [24], its analysis is limited to temporal anomalies and does not extend to corruption of memory-resident kernel structures.

Mamone et al. [25] presented one of the first experimental analyses targeting the reliability of FreeRTOS kernel structures through FI. That work demonstrated the relevance of FIs directly into RTOS 's internal data structures to identify critical components, using a hardware-based experimental setup and focusing on transient-fault effects. Our work aims at extending this analysis along several dimensions, as detailed in Sections III, IV, and V. In particular, KRONOS enables automated, repeatable kernel-level FI without dedicated hardware, and supports both transient and permanent fault models. Moreover, the proposed approach extends the experimental scope to a broader set of kernel objects and execution conditions, enabling a more comprehensive characterization of RTOS-level failure modes.

III. Proposed Approach

KRONOS is a post-propagation FI framework that operates on OS-visible memory, assuming faults have already reached kernel-accessible state. This abstraction allows controlled analysis of RTOS behavior under corrupted kernel data without modeling lower-level propagation. It systematically reveals the most vulnerable kernel structures and failure modes, providing a conditional vulnerability assessment of FreeRTOS components rather than absolute hardware fault rates.

KRONOS is built on the FreeRTOS port for Linux and Windows, allowing us to run a complete FreeRTOS system as a set of user-space processes and threads on a hosted operating system without hardware. Leveraging the capability of FreeRTOS to run as a system process, KRONOS is fully software-based and requires no architectural modifications or external debug interfaces. Its portability and direct access to kernel structures make it well-suited for systematic dependability evaluation of FreeRTOS-based systems. Specifically, KRONOS is composed of: (i) a *command* module, which specifies injection commands; (ii) a *target* module, responsible for extracting and acquiring the injection targets; (iii) an *OS-abstraction layer*, allowing the injector to execute on different FreeRTOS ports; (iv) an injector module; and (v) a logging module, which hooks into FreeRTOS at key events (e.g., task switches) to record runtime behavior. Figure 1 shows a high-level overview of the main components of the FI application.

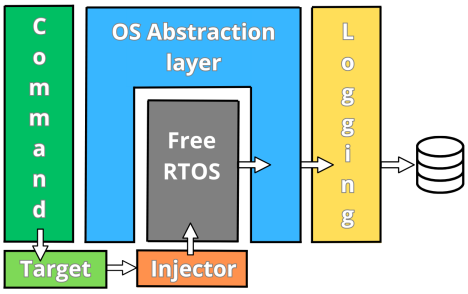


Fig. 1. Overview of the main components of KRONOS and their interactions.

A. Fault Injection and Result Collection Workflow

In its simulation ports, FreeRTOS runs as a process (and the port implementation guarantees the tick reference). Hence, KRONOS is designed to run in a dedicated thread within the same process. This design allows the injector thread to share the same process address space as the simulated FreeRTOS, without altering the FreeRTOS implementation and configuration. This shared-memory model enables precise access to FI within key FreeRTOS components, with minimal impact on the system and without requiring special hardware or architectural modifications.

The FI workflow comprises four key steps:

(**I**) *Injection Target Specification*: Qualified kernel objects (variables, pointers, lists, structures) are identified at runtime via specialized gatherer functions through name, base memory address, size, and structural hierarchy within kernel data.

(**II**) *Golden Run Profiling*: FreeRTOS tasks execute uninterrupted to establish timing baselines and expected outputs for functional correctness comparison.

(**III**) *Run Initialization*: The injector loads target parameters (injection time offset from scheduler start, byte/bit offsets, fault type) alongside the golden run profile, starts the FreeRTOS scheduler (allowing normal task/timer execution), and sleeps until the specified injection time.

(**IV**) *Fault Injection*: The injector wakes and applies the configured fault.

Post-injection, KRONOS monitors the scheduler state: if the scheduler is functional (only the *Idle task* is ready), it lets the Idle task hook gracefully terminate the system; otherwise, it forces a shutdown after a configurable timeout. Logs and outputs are then analyzed against golden-run references for task correctness and execution timing, in line with broader software test monitoring approaches already proposed [20].

Each run can be classified with one of the following standard outcome categories:

- **Benign**: The system completes its task within the expected time and produces correct results.
- **DELAY**: The result is functionally correct, but the execution exceeds the predefined deadline (which led to a performance or real-time violation).
- **Silent Data Corruption (SDC)**: The system finishes on time but yields an incorrect result without raising errors. To better capture the nuances of real-time execution, this class has two labels: if SDC is produced with a delay, it is labeled accordingly (SDC DELAY).
- **HANG**: The system never completes (no output or completion signal), typically due to deadlock, livelock, or an unhandled fault. The time limit is configurable as a percentage of the golden run time.
- **CRASH**: The system terminates abruptly (e.g., via exception, abort, or unhandled signal), preventing normal shutdown and output generation.
- **INVALID**: Since certain OS structures are valid targets only under specific use-case behaviors, we classify them into a separate category when, even though injections are attempted on the target, it is not considered valid.

B. Fault Injector Implementation

KRONOS implements two software-level fault models on RTOS-visible kernel state: transient faults and permanent faults. These models do not emulate the physical generation of radiation-induced effects at the circuit level; rather, they represent the condition in which a fault has already propagated to a kernel data structure and become architecturally visible to the RTOS. In this sense, the proposed FI approach is post-propagation and targets the software manifestation of the fault.

Transient faults model a non-persistent single-event corruption of a kernel target, represented as a one-time bit flip in the runtime memory of the selected kernel object. At the configured FI instant, the injector directly accesses the kernel process memory space, locates the target memory address plus the byte and bit offsets, and performs the flip of the targeted bit. After the injection, subsequent kernel writes to the same location are not altered. This model captures a temporary fault effect that becomes visible in the RTOS state but is not permanently retained.

Permanent faults model a persistent corruption of a kernel target, represented as a stuck-at condition on a selected bit of the targeted kernel object. Since the corrupted value must be preserved across subsequent updates, KRONOS introduces a compile-time patching mechanism that preprocesses the FreeRTOS kernel code before building the hosted executable. The patcher uses PCRE2 to locate all writes to the targeted locations and inserts a bitwise mask enforcing the stuck-at condition after each write. This strategy ensures that the corrupted bit remains permanently forced during execution, at the cost of a minimal extra execution time. It therefore models a persistent software-visible fault condition rather than the low-level physical mechanisms that originally generated it.

IV. Experimental Setup

To evaluate FreeRTOS's dependability, we conducted an extensive FI campaign across different kernel structures. Specifically, the targets of our analysis were divided into four groups: (i) *Global Variables:* variables that influence scheduling decisions and help manage FreeRTOS, described in Table I; (ii) *Pointers:* pointers to FreeRTOS global variables or data structures used to manage the RTOS (see Table II); (iii) *Lists:* structures used by FreeRTOS for managing tasks. Tasks

TABLE I
FREERTOS GLOBAL VARIABLE TARGETS

TARGET	TYPE	DESCRIPTION
uxCurrentNumberOfTasks	*UBaseType_t*	Number of active tasks.
uxDeletedTasksWaitingCleanup	*UBaseType_t*	Number of TCB pending cleanup by the IDLE task.
xPendedTicks	*TickType_t*	Ticks accumulated while scheduler was suspended.
uxTaskNumber	*UBaseType_t*	Counter for task creation (unique ID).
uxTopReadyPriority	*UBaseType_t*	Highest priority level with at least one ready task.
xNextTaskUnblockTime	*TickType_t*	Tick count when the next delayed task is due to unblock.
xTickCount	*TickType_t*	System tick counter since scheduler start.
xNumOfOverflows	*BaseType_t*	Number of times xTickCount has wrapped around.
xSchedulerRunning	*BaseType_t*	Scheduler started flag.
xTimerQueue	*QueueHandle_t*	Handle of the internal timer command queue.
xTimerTaskHandle	*TaskHandle_t*	Timer Daemon task handle.
xYieldPending	*BaseType_t*	Context switch pending flag

TABLE II
FREERTOS POINTER TARGETS.

TARGET	DESCRIPTION
pxCurrentTCB	Pointer to the TCB of the running task.
pxCurrentTimerList	Pointer to the current timer list.
pxDelayedTaskList	Pointer to the delayed task list.
pxOverflowDelayedTaskList	Pointer to the overflowed delayed task list.
pxOverflowTimerList	Pointer to the current overflowed timer list.
xIdleTaskHandle	Pointer to the Idle task's TCB

can belong to different lists based on their state, i.e., ready, delayed, pending, or suspended (see Table III); (iv) *Current TCB:* is the data structure associated with the currently running task (see Table IV). It contains relevant information about the task, i.e., task state, priority, and stack pointer.

The experimental setup employs five benchmarks from the TACLeBench suite executed by different tasks with different priorities [26]: (i) SHA (classic hash function), (ii) FFT (Fast Fourier Transform computation), (iii) CUBIC (cubic equation solver), (iv) HUFF_DEC (Huffman decoding), and (v) ADPCM_ENC (adaptive pulse-code modulation encoding). These benchmarks cover cryptographic, signal-processing, and

TABLE III
FREERTOS LIST TARGETS.

TARGET	DESCRIPTION
pxReadyTasksLists	Array of ready-task lists indexed by task priority.
xDelayedTaskList1	Lists of tasks delayed via vTaskDelay().
xDelayedTaskList2	Lists of tasks delayed via vTaskDelay().
xPendingReadyList	Tasks that became ready while the scheduler was locked.
xActiveTimerList1	List of active software timers.
xActiveTimerList2	List of active software timers.
xSuspendedTaskList	List of suspended tasks.
xTasksWaitingTermination	List of deleted tasks pending resource cleanup by the Idle task.

TABLE IV
FREERTOS CURRENT TCB TARGETS. THEY CAN BE ACCESSED THROUGH PXCURRENTTCB.

TARGET	TYPE	DESCRIPTION
pcTaskName	*char**	Name of the current task.
pxStack	*StackType_t**	Pointer to the base of the current task stack.
pxTaskTag	*TaskHookFunction_t*	Pointer to the application task tag of the currently running task.
pxTopOfStack	*StackType_t**	Pointer to the top of the stack of the currently executing task.
ucDelayAborted	*uint8_t*	Flags whether a blocking delay was prematurely aborted for the current task.
ucNotifyState	*uint8_t**	Array of integer used by the task notification mechanism.
ulNotifiedValue	*uint32_t**	Array of integer used by the task notification mechanism.
ulRunTimeCounter	*uint32_t*	Accumulates the task's time spent in the running state.
uxBasePriority	*UBaseType_t*	Current task's base priority used by the mutex priority inheritance mechanism.
uxMutexesHeld	*UBaseType_t*	Current task's number of mutexes held.
uxPriority	*UBaseType_t*	Priority of the current task.
uxTaskNumber	*UBaseType_t*	User-defined task tag.
uxTCBNumber	*UBaseType_t*	Kernel-assigned unique TCB ID.
xEventListItem	*ListItem_t*	List node for events (queues, semaphores).
xStateListItem	*ListItem_t*	List node linking TCB into scheduler lists.

compression workloads, providing a diverse task mix. The first three tasks (SHA, FFT, and CUBIC) were configured with priority 1, HUFF_DEC with priority 2, and ADPCM_ENC with priority 3. Priorities were intentionally differentiated to exercise multiple *ready lists* and create non-trivial preemption behavior during the campaign. Each task executes its computation once and then self-deletes, ensuring the FreeRTOS scheduler can detect task completion and shut down properly.

The FreeRTOS kernel was configured with a 1 kHz tick rate, 7 priority levels, cooperative multitasking, dynamic task creation, and support for software timers, counting semaphores, recursive mutexes, and queue sets. Both transient and permanent faults were considered, with injections occurring within a 10,000 ns window after scheduler start (delay = 5% deviation, hang = 300% of golden run time). Each run executed the five TACLeBench tasks plus system/timer daemon tasks until graceful shutdown.

To achieve statistically meaningful results, we applied the binomial proportion estimation approach from [27] (32-bit reference), treating the fault space as effectively infinite. This yielded 666 FI injections per location (99% confidence, 5% margin of error), totaling 83,916 injections across all targets.

To assess execution-time stability, the FI campaign was executed five times using four parallel worker processes. It was completed in an average wall-clock time of 76.7 s, with a standard deviation of 7.1 s (min–max: 70.3–91.3 s). These results confirm that KRONOS can execute large-scale FI campaigns within practical time bounds while maintaining repeatability across independent injection samples, making it ideal for early-stage development.

V. EXPERIMENTAL RESULTS

The experimental campaign demonstrates that, under OS-visible kernel memory corruptions, FreeRTOS exhibits moderate fault tolerance while several kernel components remain highly vulnerable (Table V). Approximately 70% of runs completed successfully, about 3% experienced delayed execution,

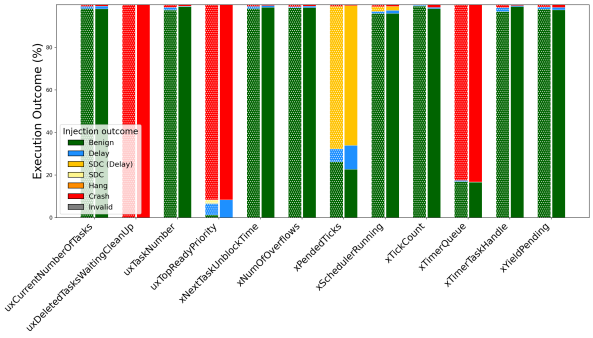


Fig. 2. Injections on *FreeRTOS* **variables**: transient (on the left) and permanent (on the right) faults.

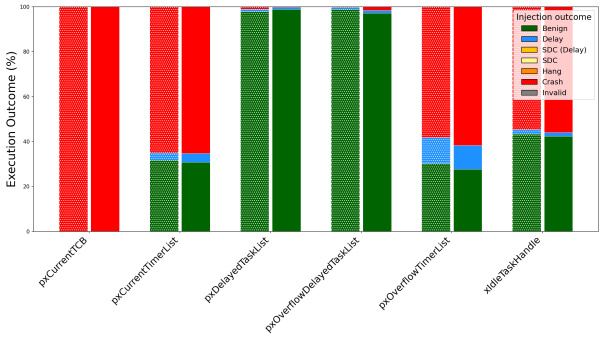


Fig. 3. Injections on *FreeRTOS* **pointers**: transient (on the left) and permanent (on the right) faults.

and more than 20% resulted in system crashes, indicating limited fault-handling capability in the default configuration. SDC occurrences were infrequent (below 2%) and generally associated with timing deviations, whereas fewer than 5% of injections targeted invalid objects. Comparable failure rates were observed for transient and permanent faults across most kernel components, as corruptions of critical data structures frequently caused immediate or irrecoverable failures, such as scheduler inconsistencies or invalid memory accesses, so fault persistence often did not materially change the observed run-level outcome within a given execution. These findings indicate that RTOS dependability is highly dependent on workload characteristics and operational context.

TABLE V
OUTCOME OF THE WHOLE FI CAMPAIGN DIVIDED BY FAULT TYPE.

FAULT TYPE	BENIGN	DELAY	SDC	SDC (DELAY)	HANG	CRASH	INVALID
Transient	70.16%	2.80%	0.04%	1.69%	0.00%	20.43%	4.88%
Permanent	69.66%	3.00%	0.00%	1.65%	0.00%	20.82%	4.88%

A category-based analysis shows that several *variables*, such as `uxDeletedTasksWaitingCleanup`, `uxTopReadyPriority`, and `xTimerQueue`, exhibit near-total failure rates when altered, almost always resulting in system crashes. This highlights their critical role in core scheduler and resource-handling decisions. Less sensitive variables, such as `xTickCount` or `uxCurrentNumberOfTasks`, typically result in correct execution but can occasionally cause crashes or delays, particularly under specific timing conditions. Notably, altering `xPendedTicks` produces a large percentage of SDC, always associated with a delay (see Figure 2).

As expected, *Pointer* variables are the most fault-sensitive elements: bit-flips almost always cause system crashes, as they can instantly invalidate objects or list references. Pointers to dynamic schedulers, delayed lists, or timer lists are the most sensitive, while `pxCurrentTCB`, whose corruption leads to crashes in all runs for both fault types, is the most critical. Other sensitive elements are `pxCurrentTimerList` and `xIdleTaskHandle`, which, when altered, account for more than 50% of system crashes. Notably, affecting this variable category does not produce any SDC (Figure 3).

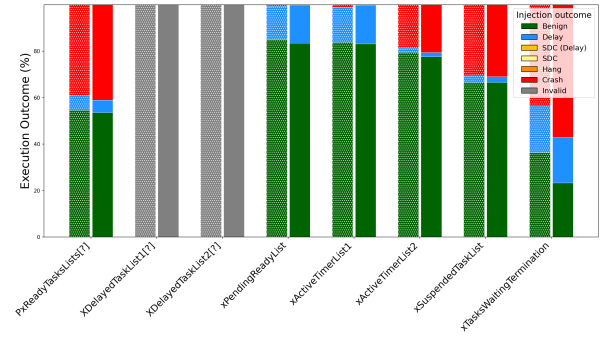


Fig. 4. Injections on *FreeRTOS* **lists**: transient (on the left) and permanent (on the right) faults.

Faults injected in list objects lead to less frequent crashes compared with the other category (approximately from 20% to 50%), but with a small subset yielding delays (see Figure 4). However, the crash rate in this category tends to increase for permanent faults. Injections into the task waiting termination lists (i.e., `xTasksWaitingTermination`) show exceptionally high sensitivity. In contrast, lists associated with less active kernel operations (e.g., `xSuspendedTaskList`) exhibit a higher rate of correct execution, even though the risk of a system crash remains non-negligible. Notably, even affecting this variable category does not produce any SDC.

As it is possible to observe in Figure 4, some targets were never valid at injection time. In particular, the delayed task lists were empty because the adopted use case is mostly computation-oriented and did not require tasks to enter delayed states at the selected injection instants.

Figure 5 reports that corrupting the fields inside the current TCB did not produce particularly sensitive effects, with a small percentage of crashes and delays in most cases. The most sensitive field is `uxPriority`, which, if altered, caused a crash in more than 80% of runs. This evidence underscores the importance of priority management for deterministic RTOS operation. Also, `ucDelayAborted` caused a crash in more than 40% of runs when altered. Notably, even affecting the fields of the TCB does not produce any SDC.

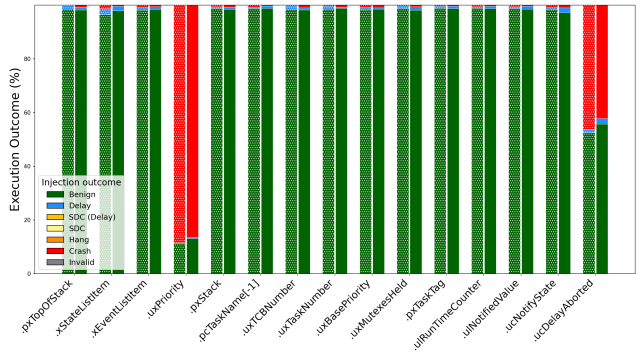


Fig. 5. Injections on *FreeRTOS* **current TCB** fields: transient (on the left) and permanent (on the right) faults.

VI. Conclusion

This work extended earlier experimental analyses of FreeRTOS reliability using KRONOS, a systematic software-based FI framework targeting kernel-level data structures. By operating on OS-visible memory state, KRONOS enables repeatable evaluation of RTOS failure modes without specialized hardware or debug interfaces.

An extensive FI campaign on critical FreeRTOS kernel objects showed that, although many corruptions are tolerated, core scheduler-related structures remain highly vulnerable, often causing crashes or deadline violations. The similar impact of transient and permanent faults indicates that corruption of critical RTOS state frequently leads to immediate, unrecoverable failures. The analysis highlights the conditional vulnerability of FreeRTOS kernel components once faults become architecturally visible and complements lower-level fault-propagation studies, helping RTOS designers identify kernel structures needing stronger protection or monitoring in safety- and mission-critical systems.

Future work includes extending KRONOS to other RTOSes and integrating cross-layer models that relate OS-visible faults to specific device-level phenomena.

Acknowledgment

The authors thank Giovanni De Florio and Dimitri Schiavone for their contributions to this research during their master's theses.

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] M. A. Solouki *et al.*, "Dependability in Embedded Systems: A Survey of Fault Tolerance Methods and Software-Based Mitigation Techniques," *IEEE Access*, vol. 12, pp. 180 939–180 967, 2024.

[3] R. Aalund and V. Philip Paglioni, "Enhancing Reliability in Embedded Systems Hardware: A Literature Survey," *IEEE Access*, vol. 13, pp. 17 285–17 302, 2025.

[4] J. R. Srour and J. W. Palko, "Displacement damage effects in irradiated semiconductor devices," *IEEE Transactions on Nuclear Science*, vol. 60, no. 3, pp. 1740–1766, June 2013.

[5] 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.

[6] M. Portolan *et al.*, "Alternatives to fault injections for early safety/security evaluations," in *2019 IEEE European Test Symposium (ETS)*, 2019, pp. 1–10.

[7] G. Papadimitriou and D. Gizopoulos, "Demystifying the system vulnerability stack: Transient fault effects across the layers," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021, pp. 902–915.

[8] A. Benso and S. Di Carlo, "The Art of Fault Injection," *Control Engineering and Applied Informatics*, vol. 13, no. 4, pp. 9–18, 2011.

[9] C. De Sio *et al.*, "Evaluating reliability against SEE of embedded systems: A comparison of RTOS and bare-metal approaches," *Microelectronics Reliability*, vol. 150, p. 115124, 2023, Special issue of 34th European Symposium on Reliability of Electron Devices, Failure Physics and Analysis, ESREF 2023.

[10] F. Guan *et al.*, "Open source FreeRTOS as a case study in real-time operating system evolution," *Journal of Systems and Software*, vol. 118, pp. 19–35, 2016.

[11] G. A. Kanawati *et al.*, "FERRARI: a flexible software-based fault and error injection system," *IEEE Transactions on Computers*, vol. 44, no. 2, pp. 248–260, Feb 1995.

[12] 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, Feb 1998.

[13] 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.

[14] G. Di Natale *et al.*, *Cross-Layer Reliability of Computing Systems*. The Institution of Engineering and Technology (IET), 2020.

[15] P. Bodmann *et al.*, "Soft Error Effects on Arm Microprocessors: Early Estimations vs. Chip Measurements," *IEEE Transactions on Computers*, pp. 1–1, 2021.

[16] A. Bosio *et al.*, "Reliability assessment of FreeRTOS in Embedded Systems," in *52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks - Supplemental Volume (DSN-S)*, 2022, pp. 28–30.

[17] E. Casseau *et al.*, "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.

[18] O. Chatzopoulos *et al.*, "Gem5-MARVEL: Microarchitecture-Level Resilience Analysis of Heterogeneous SoC Architectures," in *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, March 2024, pp. 543–559.

[19] N. Binkert *et al.*, "The gem5 simulator," *SIGARCH Computer Architecture News*, vol. 39, no. 2, p. 1–7, Aug. 2011.

[20] H. Cho *et al.*, "Quantitative evaluation of soft error injection techniques for robust system design," in *Proceedings of the 50th Annual Design Automation Conference*, ser. DAC '13. New York, NY, USA: Association for Computing Machinery (ACM), 2013.

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

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

[23] D. Silva *et al.*, "A Hardware-Based Approach for Fault Detection in RTOS-Based Embedded Systems," in *2011 Sixteenth IEEE European Test Symposium*. IEEE, 5 2011, pp. 209–209.

[24] International Electrotechnical Commission (IEC), "IEC 61000-4-29: Electromagnetic compatibility (EMC) – Part 4-29: Testing and measurement techniques – Voltage dips, short interruptions and voltage variations on DC input power port immunity tests," 2000.

[25] D. Mamone *et al.*, "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)*. IEEE, 2020, pp. 1–6.

[26] H. Falk *et al.*, "TACLeBench: A Benchmark Collection to Support Worst-Case Execution Time Research," in *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)*, ser. OpenAccess Series in Informatics (OASIcs), M. Schoeberl, Ed., vol. 55. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2016, pp. 2:1–2:10.

[27] R. Leveugle *et al.*, "Statistical fault injection: Quantified error and confidence," in *2009 Design, Automation & Test in Europe Conference & Exhibition*. IEEE, 4 2009, pp. 502–506.