POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Fault Injection and selective Hardening of Real Time Operating Systems

Supervisors

Prof. Alessandro SAVINO

Prof. Maurizio REBAUDENGO

Candidate

Dimitri SCHIAVONE

July, 2025

Acknowledgements

I would like to thank my dear parents Romana and Umberto for their constant support and encouragement throughout my academic journey. Their unwavering belief in me has been a constant source of strength.
A special thanks goes to my beloved girlfriend, Silvia, whose steadfast support, even in moments when I struggled to find motivation, has meant the world to me.
I would like to extend my heartfelt appreciation to my Uncle Guido for his invaluable support, which has been instrumental in helping me through challenging times.
I am also thankful to my best friend, Marco. His insightful advice and the moments of shared free time have provided both motivation and much needed balance during this endeavor.
Lastly, I would like to thank Professor Alessandro Savino for providing me with the opportunity to work on this experimental thesis. His patience and technical advice throughout our email correspondence have been essential to the progress of this work.

"In the midst of winter, I found there was, within me, an invincible summer. And that makes me happy. For it says that no matter how hard the world pushes against me, within me, there's something stronger – something better, pushing right back."
Albert Camus

Abstract

Real-Time operating systems provide deterministic behavior for systems operating under strict deadline requirements. Beyond managing tasks and resources, operating systems offer a hardware abstraction layer that enables developers to concentrate on application logic instead of low-level hardware details. However, both systems with and without operating system support are vulnerable to ionizing radiation from sources such as cosmic rays, solar wind, electromagnetic interference, and other deep space phenomena. These radiation effects can lead to unintended, often catastrophic, failures ranging from system malfunctions to complete device breakdowns.

Real-Time operating systems are commonly deployed in safety-critical applications including railway systems, military drones, medical devices and transportation systems in cars and airplanes; where even a minor failure could have dire consequences for human life. Thus, it is crucial to investigate mitigation techniques at both the hardware and software levels. This thesis focuses exclusively on software-based mitigation strategies within an RTOS (Real Time Operating System) environment, specifically using FreeRTOS as the platform under study. More precisely, the work targets the FreeRTOS port for POSIX (Portable Operating System Interface for Unix) [1] and Windows [2], which allows a complete FreeRTOS operating system to run as a hosted user-space application on general-purpose operating systems such as GNU (GNU Is Not Unix)/Linux, FreeBSD, Solaris, Mac OS or Windows. For the purposes of this thesis, the system of choice is GNU/Linux.

The study begins with a fault injection investigation designed to identify critical components of the RTOS that may compromise its expected behavior under radiation-induced disturbances. This research employs a software fault injection technique to emulate the effects of radiation on electronic devices, focusing on two primary error types: the SEU (Single Event Upset), which causes random bit flips in memory; and the SEHE (Single Event Hard Error), which results in permanent "stuck-at" memory states. Software fault injection offers a cost-effective alternative to specialized radiation testing facilities while still providing meaningful insights. Based on the data collected from this investigation, the objective of this thesis is to implement a targeted, selective hardening strategy to reinforce the identified vulnerable areas of the RTOS under study.

Table of Contents

List of Figures

List of Tables

Listings

Acronyms

OS
Operating System

RTOS
Real Time Operating System

POSIX
Portable Operating System Interface for Unix

BSD
Berkeley Software Distribution

GNU
GNU Is Not Unix

SEE
Single Event Effect

SET
Single Event Transient

SEU
Single Event Upset

MBU
Multiple Bit Upset

SEFI
Single Event Functional Interrupt

SEL

Single Event Latchup

SEB

Single Event Burnout

SEGR

Single Event Gate Rupture

SEHE

Single Event Hard Error

RAM

Random Access Memory

SRAM

Static Random Access Memory

DRAM

Dynamic Random Access Memory

FI

Fault Injection

GCR

Galactic Cosmic Ray

EMI

Electromagnetic Interference

CMOS

Complementary Metal Oxide Semiconductor

MMU

Memory Management Unit

MPU

Memory Protection Unit

NMOS

N-Type Metal Oxide Semiconductor

PMOS

P-Type Metal Oxide Semiconductor

SCR

Silicon Controlled Rectifier

ECC

Error Correcting Code

CPU

Central Processing Unit

IPC

Inter Process Communication

LTE

Long Term Evolution

ASIC

Application Specific Integrated Circuit

DSP

Digital Signal Processor

ISR

Interrupt Service Routine

IRQ

Iterrupt Request

MISRA

Motor Industry Software Reliability Association

RISC

Reduced Instruction Set Computer

AVR

Advanced Virtual RISC

ARM

Avanced Risc Machines

API

Application Programming Interface

FIFO

First In First Out

TCB

Task Control Block

COTS

Commercial Off The Shelf

SPI

Serial Peripheral Interface

CAN

Controller Area Network

I/O

Input Output

CLI

Command Line Interface

CSV

Comma Separated Value

SWIFI

Software Based Fault Injection

MAFALDA

Microkernel Assessment by Fault Injection Analysis and Design Aid

GDB

GNU Debugger

KGDB

Kernel GNU Debugger

RTOSG

RTOS Guardian

IEC

International Electrotechnical Commission

FIAT

Fault Injection Based Automated Testing

FIFA

Kernel-Level Fault Injection Framework for ARM-Based Embedded Linux System

SIM

Subscriber Identity Module

PCRE2

Perl Compatible Regular Expressions (version 2)

BCH

Bose Chaudhuri Hocquenghem codes

AMD

Advanced Micro Devices

LSB

Least Significant Bit

IP

Intellectual Property

PCB

Printed Circuit Board

RTL

Register Transfer Level

Chapter 1

Introduction

1.1 Rationale

Modern society's pervasive reliance on computing is a direct outcome of significant advancements in electronics and computer systems. Breakthroughs in these fields, bolstered by economies of scale, have seamlessly integrated computing into daily life. Today, a wide array of devices, including smartphones, smartwatches, smart televisions, and fitness trackers, have become ubiquitous. Moreover, nearly every household appliance has evolved into a "smart" device, ranging from refrigerators and washing machines to autonomous vacuum cleaners.

While many of these devices are categorized as embedded systems, smartphones stand apart. Initially developed as embedded devices, they have since evolved, offering computational capabilities comparable to those of mid-range computers. Beyond consumer electronics, embedded systems play a critical yet often understated role across various sectors. They are integral to the functionality of vehicles, medical devices, financial trading platforms, railway systems, military equipment, satellites, and more.

This widespread incorporation of embedded systems underscores the extent to which our contemporary world is intertwined with computing technology, a trend that is only set to intensify in the foreseeable future.

Embedded devices incorporate specialized software tailored to their specific functions. In some instances, this software is highly application-specific firmware designed exclusively for the device, directly interfacing with the underlying hardware. In other cases, these devices run specialized operating systems, such as real-time operating systems (RTOS), which are engineered to meet the stringent timing and reliability requirements of embedded applications.

What emerges from the above is that human reliance on the correct behavior of electronic devices and their supporting software is growing. This critical property,

known as dependability, is defined as the system's ability to deliver reliable, safe, and secure services continuously over time, even when faced with faults or adverse conditions.
Dependability in a system is influenced by numerous factors. Internally, elements such as human errors during hardware design, software bugs introduced during development, intentional tampering by malicious actors, and manufacturing defects can compromise system integrity. Externally, the operational environment in which the device functions also plays a significant role. In essence, internal factors pertain to the system's design, development, and fabrication processes, while external factors arise from the conditions under which the device ultimately operates.

1.2 High energy particles

Among the external factors examined, ionizing radiation, specifically through high-energy particles, emerges as one of the most damaging influences on electronic devices. This susceptibility is especially pronounced in systems operating in high-radiation environments such as outer space (e.g., satellites) and at lower altitudes including airplanes and drones. Furthermore, as transistors and electronic gates continue to shrink, the vulnerability of devices, even those functioning at sea level where radiation is less prevalent, is increasingly amplified. When these energetic particles interact with a device, they can induce computational errors under certain conditions or, in more severe cases, cause irreversible damage.
The subsequent chapters will delve deeper into the phenomenon commonly referred to as SEE (Single Event Effect), discussing its impacts on electronic systems in greater detail.

1.3 Fault injection

Therefore it is essential to employ robust testing techniques that can uncover vulnerabilities within a system, as early detection is key to improving overall system resilience. These techniques are designed not only to identify potential weaknesses but also to gather comprehensive data regarding which parts of the system are the most sensitive to perturbations. The information collected through this process provides a valuable basis for the development of specific mitigation strategies, ultimately enhancing the dependability and performance of the system as a whole.
Among the various methods available for dependability testing, fault injection (FI) stands out as one of the most extensively utilized techniques [3]. Fault injection involves deliberately introducing faults into a system in order to study its behavior under abnormal conditions. Such an approach is critical for understanding how faults propagate through system components and for verifying that appropriate

error-handling mechanisms are in place.
While the fundamental principles and practices of fault injection are briefly mentioned here, a more detailed discussion covering its various approaches and practical applications will be provided in a subsequent chapter.

1.4 Tools Used

This thesis focuses on studying the effects of faults and developing a selective hardening strategy within the context of a specific RTOS, namely *FreeRTOS*. The work utilizes *FreeRTOS* version 10.4.6, specifically employing its POSIX port [1], although a functional Windows version was developed in parallel which makes use of the *Windows FreeRTOS* port [2].
The project is compiled using the default system compiler available on most *GNU/Linux* distributions, which in this work is GNU's *GCC* version 15.1.1 [4].
To facilitate cross-platform compilation between the *POSIX* and *Windows* ports, a *CMake* based build system is used, specifically version 4.0.2 [5].
Development and testing of the project were carried out on a *GNU/Linux* platform, more precisely on *Arch Linux* [6].

Chapter 2

Single Event Effects

2.1 Introduction

In the introductory chapter, the growing importance of dependability in modern embedded systems was emphasized, and fault injection testing was identified as a critical tool for uncovering system vulnerabilities. One significant external factor that challenges system reliability is ionizing radiation from high-energy particles. As these particles interact with ever-more miniaturized electronic components, a variety of Single Event Effects (SEE) can occur, ranging from *transient* errors to *permanent* damage in severe cases.
This chapter examines the phenomenon of SEE by defining the concept, classifying the various types, and exploring their origins. The discussion is then extended to analyze the practical implications on electronic devices, particularly those running an RTOS. By establishing this comprehensive background, the inherent challenges in designing resilient systems are highlighted, underscoring the necessity of appropriate hardening strategies. These strategies can be broadly divided into hardware-based and software-based approaches. Hardware-based methods are briefly introduced here for completeness, while software-based selective hardening measures are briefly introduced and discussed in greater detail in later chapters.

2.2 Single Event Effects

Single Event Effects are disruptions in the normal operation of electronic devices triggered when a single ionizing particle interacts with a circuit. Such interactions occur as the particle deposits charge into the sensitive regions of a device, thereby inducing abrupt and significant voltage or current spikes. These transient surges can exceed the device's design tolerances and may result in temporary malfunctions, permanent damage, or erroneous output signals.

2.2.1 Sources of SEE

The ionizing particles responsible for SEE come from diverse sources:

Extraterrestrial sources

In space and high-altitude environments, devices are continuously exposed to a complex radiation environment that includes Galactic Cosmic Rays (GCR), the solar wind, and various solar phenomena that fluctuate with solar activity. In addition to GCR, other extraterrestrial sources contribute significantly to the radiation exposure:

- The Van Allen belts [7], comprised of the inner belt (mainly protons and high-energy particles between about 1,000 and 5,000 kilometers altitude) and the outer belt (dominated by high-energy electrons between roughly 15,000 and 25,000 kilometers), pose significant risks to satellites and astronauts.
- The Earth's magnetosphere traps heavy ions such as iron or carbon, which due to their mass and penetration capability can have pronounced effects on sensitive equipment.
- High-energy protons and heavy ions originating from supernova events and energetic solar flares further add to the radiation hazards.

Moreover, when GCR interact with the Earth's atmosphere, they generate cascades of secondary particles that can reach and impact devices at sea level. Although these secondary effects are less likely than direct radiation encountered in orbit, they still represent a potential risk for devices operating in lower orbits and near-Earth environments.

Terrestrial sources

On Earth, man-made sources such as nuclear power plants and materials undergoing radioactive decay contribute to the incidence of SEE. In addition to these, there are several other factors that can lead to SEE:

- **Industrial and Medical Equipment:** Industrial accelerators and certain medical imaging devices, like linear accelerators used in radiation therapy or diagnostic equipment, can emit ionizing radiation. Although these sources are typically well-shielded, accidental exposure or equipment malfunction can result in localized increases in radiation levels, potentially triggering SEE.
- **High-Voltage Equipment:** Some high-voltage power equipment and transmission systems may generate transient electromagnetic fields or emit stray

radiation during arcing or switching operations. This interference can sometimes create conditions conductive to SEE in nearby sensitive electronic components.

- **Testing and Research Facilities:** Particle accelerators, fusion experiments, and other high-energy physics research facilities often produce fluxes of high-energy particles. While these environments are usually confined and controlled, they can be hotspots for SEE if sensitive devices are exposed without proper shielding.

- **Consumer Electronics in Specific Environments:** Although typical consumer electronics operate in low-radiation environments, certain contexts, such as situated near industrial sources or within research facilities, can expose these devices to higher-than-normal levels of radiation, inadvertently increasing the risk of SEE.

- **Emerging Technologies and Unintentional Exposures:** As technology continues to evolve, new sources of radiation and high-energy emissions might arise through unanticipated couplings in complex systems. This includes phenomena like EMI (Electromagnetic Interference) from nearby electronic devices, which may simulate conditions similar to those caused by radiative SEE.

2.2.2 Mechanisms of SEE Occurrence

Ionizing particles impacting electronic circuits carry sufficient energy to ionize atoms or molecules in semiconductor materials by dislodging electrons, resulting in localized charge deposition. This sudden imbalance of charge can induce transient currents, voltage spikes, or even permanent state changes in memory cells and logic elements. While early studies of SEE focused on heavy ions and alpha particles, thanks to their especially high linear energy transfer, it is now well established that a broad spectrum of radiation (including muons, pions, neutrons, protons, electrons, and even high-energy photons) can likewise provoke these effects when they traverse or strike circuit structures.

2.2.3 SEE Classification

SEE are broadly categorized into two classes (as can be seen from Figure 2.2) based on the severity and permanence of the damage:

Non-Destructive SEE (Soft Errors)

These effects result in temporary malfunctions or transient disruptions that can usually be remedied by a system reset or power cycle. Examples include bit flips in memory cells where data temporarily changes but can be corrected.

Destructive SEE (Hard Errors)

These represent irreversible damage where critical components are compromised, rendering the device permanently inoperable or severely degraded.

2.2.4 Non-Destructive SEE (Soft Errors)

Soft errors, which include the following types, generally produce transient faults:

Single Event Transient (SET)

An SET is characterized by brief bursts of voltage or current that can propagate through circuitry. Such disturbances may be observable only downstream in the circuit, making them hard to detect and mitigate.

Single Event Upset (SEU)

In an SEU, the charge deposition changes the binary state of a device's component (commonly a bit toggling from 0 to 1 or vice versa). When several bits are affected simultaneously, the phenomenon is known as an MBU (Multiple Bit Upset). These effects are mostly observed in memory devices, such as SRAM (Static Random Access Memory) and DRAM (Dynamic Random Access Memory), caches, as well as in internal registers.

Single Event Functional Interrupt (SEFI)

SEFI are typically viewed as a subset of SEU or MBU that specifically impact internal registers (e.g., control registers). Such disruptions may cause a device to freeze, reset, or become unresponsive until a power cycle is performed.

2.2.5 Destructive SEE (Hard Errors)

Hard errors involve irreversible damage and include the following manifestations:

Single Event Latchup (SEL)

SEL occurs when a high-energy particle triggers a parasitic thyristor structure inadvertently formed in CMOS (Complementary Metal Oxide Semiconductor) devices. Once activated, this structure creates a low-impedance current path between the power supply and ground, leading to excessive current flow, thermal damage, and potentially catastrophic failure. Figure 2.1 illustrates the n-well structure in CMOS where the parasitic bipolar junction transistors can be activated by such an event.

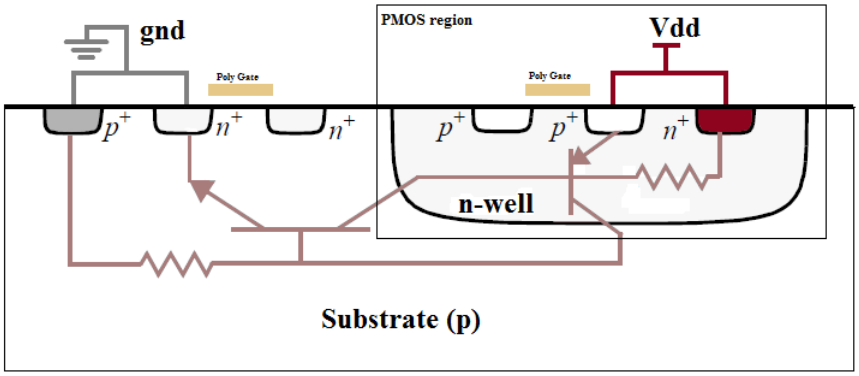

Figure 2.1: SEL Intrinsic bipolar junction transistors in the CMOS technology

Single Event Burnout (SEB)

SEB is induced by an abrupt surge in voltage or current that exceeds the rated specifications of the device, resulting in permanent damage or breakage.

Single Event Gate Rupture (SEGR)

In SEGR, the charge deposition damages the gate oxide of a transistor by creating an excessive electric field across the thin dielectric layer. This causes the gate oxide to rupture and establishes an unwanted conductive path, permanently compromising the transistor. SEGR is particularly critical in advanced semiconductor technologies with extremely thin oxide layers.

Single Event Hard Error (SEHE)

SEHE refers to unrecoverable faults where a single particle impact leads to permanent changes in the semiconductor structure. In memory devices like SRAM and

DRAM, this results in bits permanently fixed at a certain value, and when internal registers are affected, the device may ultimately become inoperable.

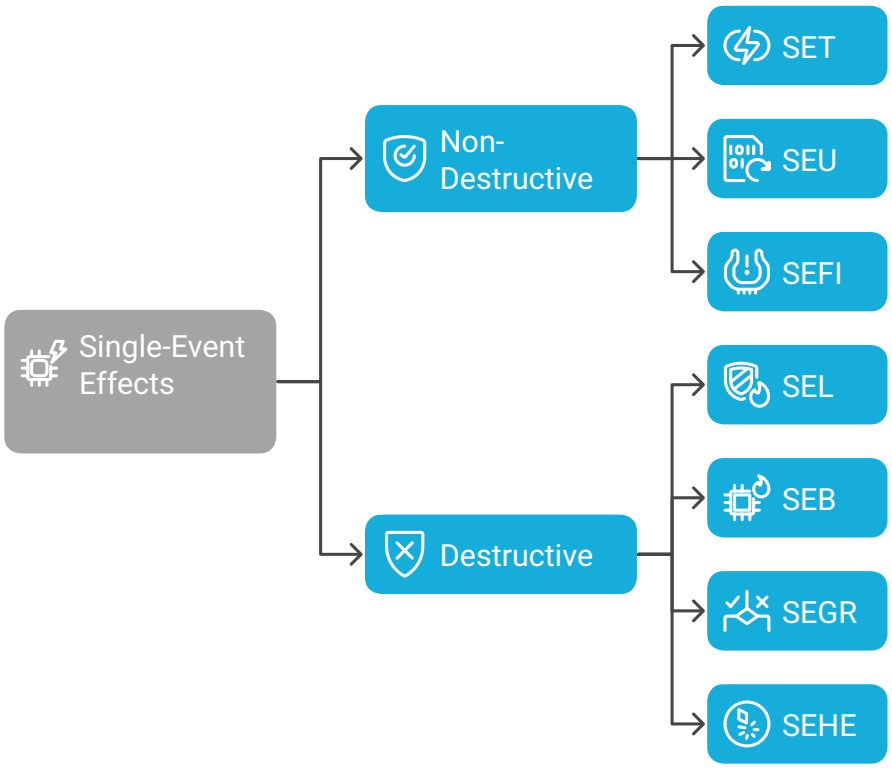

Figure 2.2: SEE Classification

2.2.6 Effects on Real-Time Operating Systems

This work primarily focuses on SEU and SEHE, as their effects are visible at the software level. The software fault injector, designed specifically for the FreeRTOS RTOS, simulates both transient SEU and permanent bit flips induced by SEHE within kernel data structures.
The impact of such bit flips varies with the location of the fault. For example:

Pointer Variables

A single bit flip in a pointer can change its target address, possibly causing it to reference an invalid or out-of-bound location, which may result in system crashes. Systems equipped with an MMU (Memory Management Unit) that supports virtual memory and physical memory segmentation might handle these issues by

terminating the affected task, while many embedded systems without an MMU could experience total system failure.

Task Scheduling Structures

Bit flips affecting task scheduling data structures may disrupt computations, cause missed deadlines, and, consequently, lead to catastrophic outcomes in real-time systems where timely task execution is crucial.
A more detailed exploration of these impacts on FreeRTOS, including a comprehensive analysis of its most sensitive kernel data structures, along with potential mitigation strategies, will be presented in a later chapter.

2.2.7 Hardware Mitigations of SEE

Mitigation of SEE can be implemented at the hardware level. For instance, to counter SEL in CMOS circuits, designers can introduce an insulating oxide layer, often referred to as a trench, around the NMOS (N-Type Metal Oxide Semiconductor) and PMOS (P-Type Metal Oxide Semiconductor) transistors (See Figure 2.3). This physical barrier disrupts the formation of the parasitic SCR (Silicon Controlled Rectifier) responsible for SEL. Similarly, SEU can be addressed through the use of ECC (Error Correcting Code) memory.
Hardware-based mitigation methods are highly effective; however, they tend to be costlier and cannot be retrofitted to devices already in the field, which may still operate reliably using appropriate software mitigation techniques. While these hardware hardening approaches play a crucial role, they will not be further discussed here, as the focus of this thesis is on software-based solutions.

2.2.8 Software Mitigations of SEE

Software mitigation primarily addresses transient SEU and MBU through mechanisms such as error correction, data redundancy, and check-pointing. This thesis adopts an ECC approach, which will be detailed in a subsequent chapter. By implementing these strategies, systems can detect and correct transient faults, thereby improving system resilience even in the presence of ionizing radiation-induced SEE.

2.3 Conclusion

This chapter has introduced and categorized SEE, outlined their fundamental mechanisms, and reviewed both hardware and software mitigation techniques. Although hardware-based protections were described for completeness, the emphasis was placed on software strategies, which form the central contribution of this thesis.

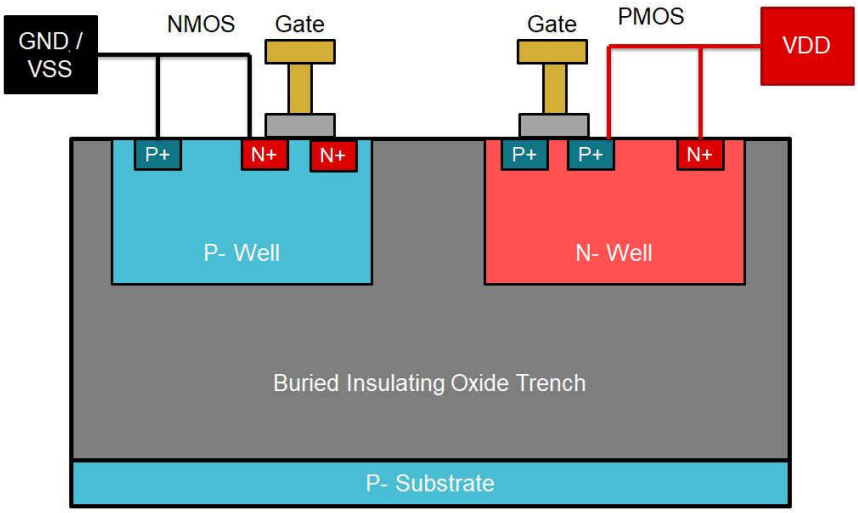

Figure 2.3: SEL Mitigation with an insulating oxide layer

In the following chapter, real-time operating systems will be examined to establish the context for assessing SEE on devices running an RTOS. This overview will pave the way for an in-depth study of FreeRTOS, the specific RTOS selected for analysis.

Chapter 3

Real-Time Operating Systems

3.1 Introduction

In the previous chapter, SEE were introduced and classified, their impact on real-time systems was briefly discussed, and an overview of both hardware-based and software-based solutions was provided. This chapter begins by presenting a general definition of operating systems, before narrowing the focus to real-time operating systems and comparing them to general-purpose operating systems. The discussion establishes the key characteristics, design requirements, and performance considerations that differentiate RTOS from their general-purpose counterparts.
A dedicated section is included to examine the critical mechanisms through which real-time performance is achieved, with particular emphasis on the contrast between poll-driven systems, commonly used in application firmware, and interrupt-driven architectures. Interrupt-driven designs, in particular, are showcased as the enabling mechanism for operating systems to regain control from applications, facilitate task switching, and enforce preemption.

3.2 Real-Time Systems

A real-time system is any computer system designed to process data and deliver responses within strictly defined time constraints, where meeting deadlines is as crucial as executing correct computations. These systems are typically employed in environments where delays can have severe consequences, such as aerospace, medical devices, automotive control systems, and industrial automation. They are characterized by deterministic processing, predictable behavior, and guaranteed

response times to ensure tasks complete within the specified time frames.
For example, consider a system controlling the spinning of drone propellers: if it fails to compute the necessary speed adjustments in time, the drone could lose stability and crash. Similarly, in civilian vehicles, the airbag system must activate immediately after sensor detection; any delay could lead to fatal outcomes.
Real-Time systems are further divided into hard real-time systems and soft real-time systems. Hard real-time systems, as illustrated by the above examples, require that deadlines be met without exception. Soft real-time systems, on the other hand, tolerate occasional missed deadlines, which might manifest as minor issues such as dropped frames in multimedia applications, rather than catastrophic failures.
In contrast, general-purpose systems can afford to miss deadlines without significant consequences, with any resulting delays causing only minor inconveniences for the end-user.

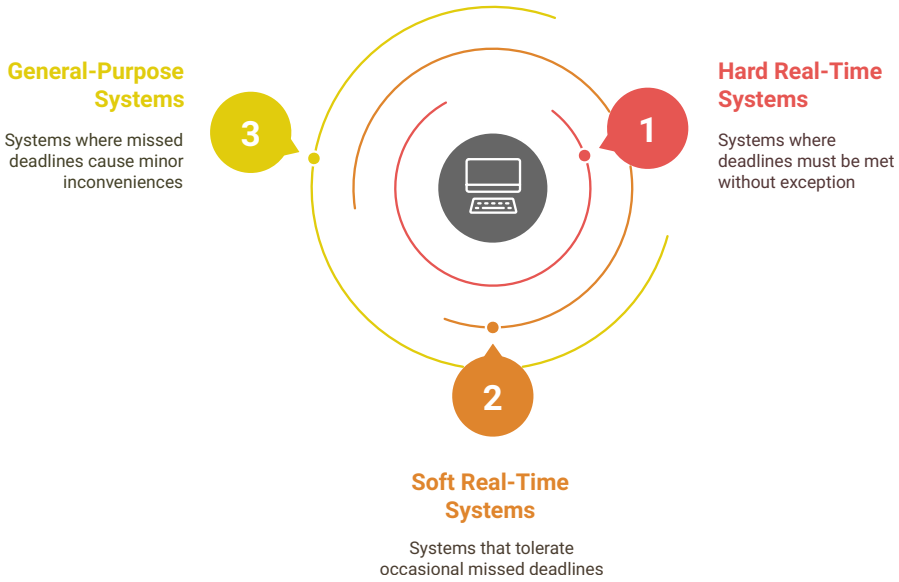

Figure 3.1: Real-Time hierarchy

3.3 Operating Systems

Before delving into real-time operating systems, it is necessary to first provide a general characterization of what an operating system is. Operating systems can be defined from multiple perspectives. One common view is as a resource arbiter, a core software program with the highest privileges that manages access to shared resources and multiplexes them among various computing entities. Alternatively, operating systems can be seen as hardware abstraction layers that mediate access

to the physical hardware on behalf of applications, ensuring fair resource allocation and secure interaction with the underlying system components.
Operating systems provide essential services to applications. At their core, they offer a common foundation that includes managing computing entities, typically in the form of tasks, processes, and threads, which are schedulable units of execution allocated CPU (Central Processing Unit) time. In addition, operating systems provide crucial services such as:

- **Memory Management:** Efficient allocation and deallocation of memory resources, ensuring that processes have access to the memory they need while preventing conflicts.
- **Timing Services:** Mechanisms for timekeeping, clock management, and scheduling, which allow the system to schedule tasks and enforce timing constraints.
- **Process Synchronization:** Tools and primitives (like semaphores, mutexes, and monitors) that facilitate the coordination and safe data sharing among concurrent processes or threads, thereby preventing race conditions and ensuring data integrity.
- **Inter-Process Communication:** Mechanisms for processes and threads to exchange data and signals, which can include message queues, pipes, shared memory, and sockets.

These services collectively create an environment where applications can run reliably and efficiently while abstracting the complexity of the underlying hardware.

3.4 General Purpose Operating Systems

General-Purpose operating systems are the ones most people use and interact with on a daily basis. Common examples include Windows, macOS, Linux, and various BSD (Berkeley Software Distribution) variants such as FreeBSD, OpenBSD, and NetBSD. In the mobile arena, popular operating systems include Android, built on a customized Linux kernel with a unique user space distinct from GNU, and iOS, which is essentially the mobile incarnation of Apple's macOS.
As the term "general-purpose" suggests, these operating systems are designed with versatility in mind; they are built to accommodate a wide array of applications and use cases rather than excelling at a single specialized task. This broad functionality means that while they may not be optimized for any particular function, they provide sufficient performance and flexibility to run diverse software, from productivity suites and multimedia applications, to complex networking and

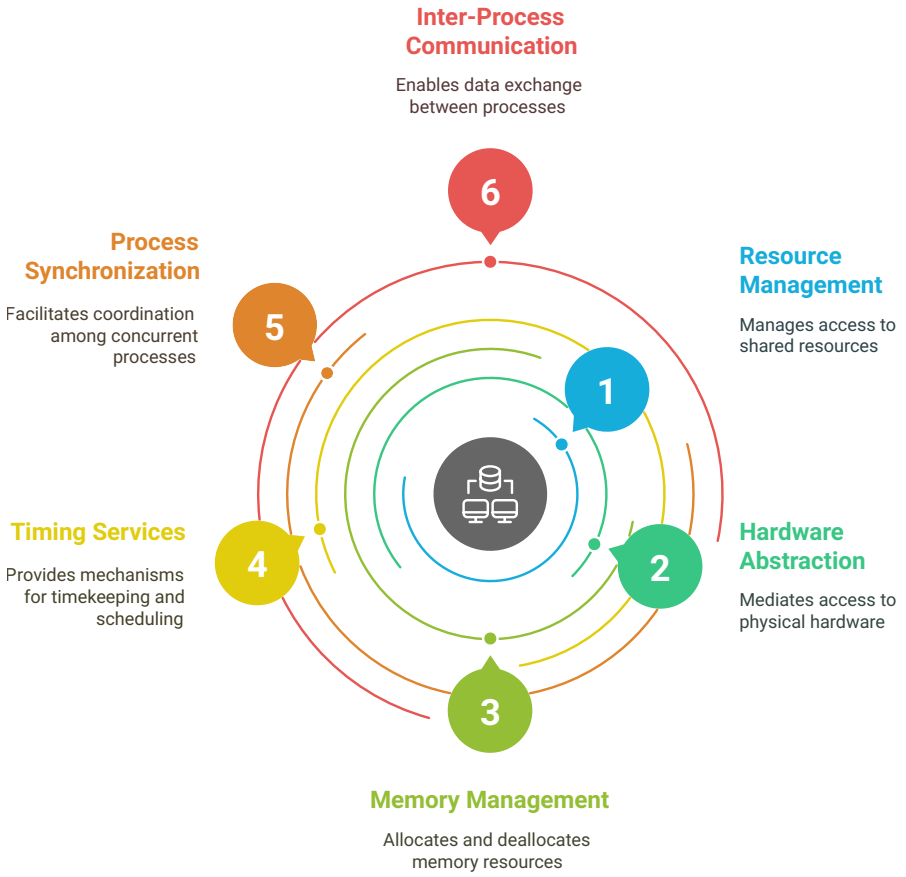

Figure 3.2: Operating system essential services

gaming software.
Additionally, general-purpose operating systems often offer extensive support for hardware, a rich ecosystem of development tools, user-friendly interfaces, and robust security features. This combination of traits makes them highly adaptable to various environments, ensuring that users and developers have the tools needed to meet everyday computing demands across personal, professional, and mobile contexts.

3.5 Real-Time Operating Systems

Real-Time operating systems [8] are specialized operating systems designed to execute tasks under strict timing constraints. In these systems, fulfilling predetermined deadlines is as critical as ensuring the correctness of the computations. Failure to meet a deadline in an RTOS can lead to system malfunctions or catastrophic outcomes, depending on the application.

RTOS are engineered for environments where the timeliness of task execution directly impacts safety, performance, or functionality. For instance, in aerospace, automotive, military, and medical applications, any delay in responding to sensor inputs or system events can compromise the mission or jeopardize lives. Unlike general-purpose operating systems that prioritize a broad set of functionalities, RTOS are optimized for deterministic behavior. This means that they guarantee that specific tasks will complete within predefined temporal windows, regardless of system load or complexity.
Key features of RTOS include:

- **Deterministic Scheduling:** RTOS employ advanced scheduling algorithms that ensure tasks are executed in a predictable and timely manner. This may involve fixed-priority scheduling, rate-monotonic scheduling, or earliest-deadline-first scheduling, among others.
- **Minimal Latency:** They are carefully designed to minimize interrupt and context switch latencies, ensuring swift responses to external events and encouraging quick task transitions.
- **Resource Management:** While providing comprehensive support for hardware resources, RTOS often use streamlined memory management and IPC (Inter Process Communication) mechanisms to avoid unpredictable delays caused by resource contention.
- **Reliability** and robustness: Given their use in mission-critical applications, RTOS are built with high reliability and robustness in mind. They often feature fault-tolerance, real-time monitoring, and error-handling mechanisms to maintain system integrity under challenging operational conditions.

In summary, RTOS serve not only as the backbone for time-sensitive tasks but also as enablers of systems where adherence to strict deadlines is paramount. Their design prioritizes both timely execution and accuracy, ensuring that real-time applications perform as expected within the rigorous constraints of their operating environments.

3.6 Heterogeneous OS Design: When General-Purpose and Real-Time Systems Coexist

After exploring the core definition of operating systems and distinguishing between general-purpose and real-time operating systems, it becomes evident that modern computing has evolved into a more complex landscape. Today's environments often incorporate multiple operating systems working in tandem to meet diverse

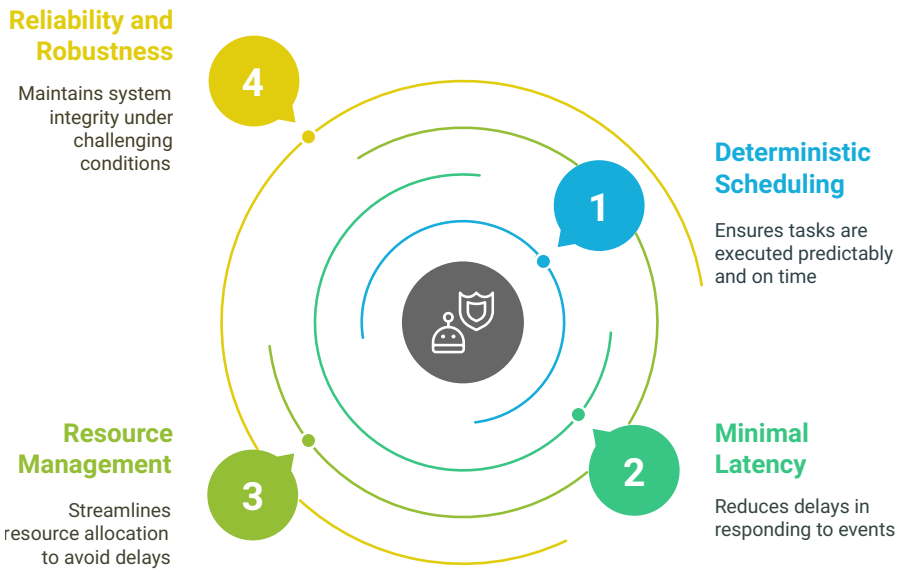

Figure 3.3: RTOS Characteristics

requirements.
For example, in the mobile arena, smartphones are not limited to a single operating system. While the primary user interface is typically managed by a general-purpose OS (such as Android or iOS), many smartphones also run additional specialized operating systems concurrently. A dedicated processor (Baseband processor) in these devices manages telephony services, including SIM (Subscriber Identity Module) card access and cellular network connectivity (spanning technologies like 4G, LTE (Long Term Evolution), and 5G), by running either a specialized RTOS or equivalent ASIC (Application Specific Integrated Circuit) firmware. Similarly, multimedia functionalities in both smartphones and general-purpose computers are often handled by DSP (Digital Signal Processor) that operate under an RTOS or firmware to ensure real-time processing of audio and video data.
In contemporary designs, specialized tasks are increasingly offloaded to dedicated processors. These processors, running either an RTOS or specifically tailored firmware, interact with the primary operating system to optimize performance, enhance responsiveness, and ensure that critical functions meet strict timing constraints. This segregation allows the main operating system to focus on a broad range of applications and user interactions, while specialized subsystems deliver deterministic performance for mission-critical or time-sensitive tasks.
This multi-OS approach reflects the growing sophistication of modern hardware architectures and the need for robust, efficient, and responsive systems in an increasingly connected and dynamic technological landscape.

3.7 Achieving Real-Time

Real-time performance depends not only on scheduling algorithms and system architecture but also on the mechanism used to interact with hardware. In essence, real time is achieved by how quickly the system can detect and handle events. There are two primary approaches: the poll-driven approach and the interrupt-driven approach.

3.7.1 Poll-Driven Approach

In a poll-driven system, the firmware or driver continuously checks hardware registers or flags to detect events. This method is especially common in application firmware where the operating environment is predictable. Key points include:

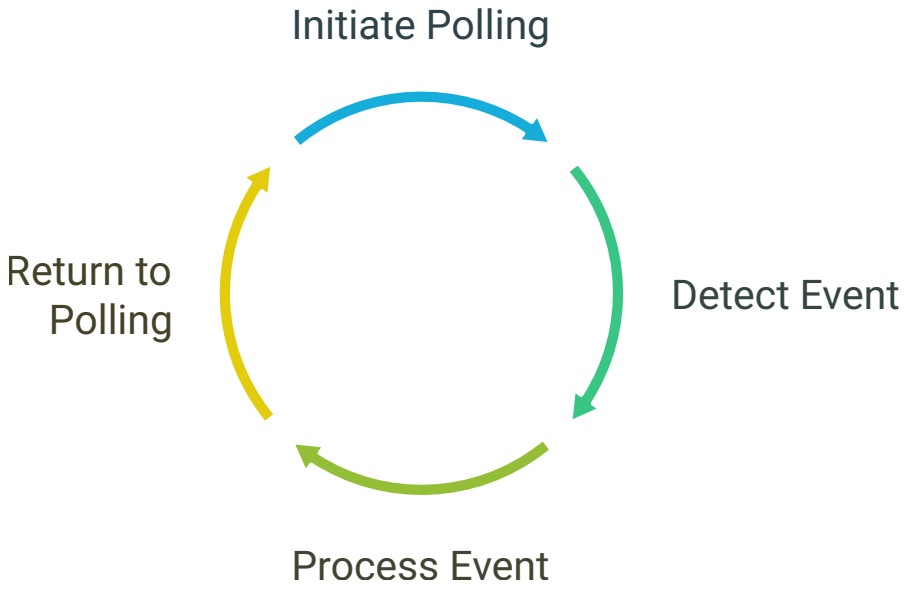

Figure 3.4: Polling cycle in firmware

- The driver continuously polls or "watches" for changes in hardware status.
- Events are detected during the polling cycle, which can introduce some latency.
- Polling works well when all events are known ahead of time and deadlines can be computed easily.

This approach is simple to implement and is often used in dedicated device routines where the timing requirements are predictable. However, as system complexity increases or when very tight deadlines are required, the continual overhead of polling can become a bottleneck.

3.7.2 The Interrupt-Driven Approach

Interrupts provide an alternative mechanism by allowing hardware to signal the CPU as soon as an event occurs. Rather than continuously monitoring hardware, the CPU can focus on other tasks until an interrupt notifies it of a required action. Consider the following:

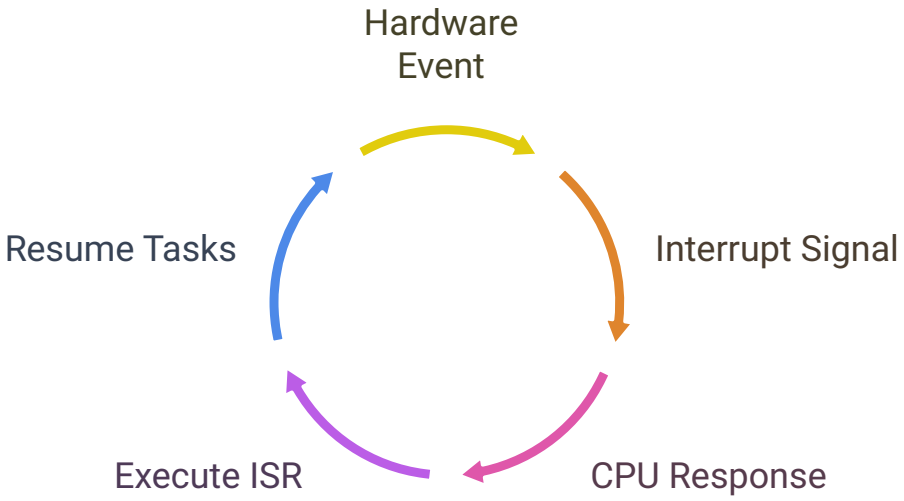

Figure 3.5: Interrupt cycle

- **Immediate Response:** Hardware generates an interrupt, causing the processor to stop its current work and run a dedicated ISR (Interrupt Service Routine), thereby minimizing latency.
- **Regaining Control:** Interrupts enable the operating system to interrupt running application code, ensuring that high-priority tasks get immediate attention.
- **Efficient CPU Utilization:** Since there is no need for constant polling, the CPU can execute other tasks until an event requires immediate response.

While interrupts aren't strictly required for scheduling (as cooperative scheduling lets tasks voluntarily yield the CPU), they are vital for preemptive schedulers and kernels. In a preemptive kernel, higher-priority interrupts can preempt tasks, even during system calls or critical kernel code, enabling multiple execution contexts within the kernel and significantly enhancing system responsiveness and multitasking capabilities. Although this approach increases the burden on system developers, it offers greater flexibility in implementing a responsive real-time operating system with a better chance of meeting its deadlines.

3.7.3 The Need for Interrupts in Complex Embedded Applications

As embedded applications scale up in complexity and the number of asynchronous events increases, the poll-driven approach becomes insufficient. Only an interrupt-driven design can efficiently manage high loads and an high multitude of asynchronous events. This approach forms the backbone of modern RTOS, which rely on sophisticated interrupt handling techniques such as:

- **Deterministic Interrupt Latency:** Keeping delays between an event and its corresponding ISR execution consistent.
- **Efficient Context Switching:** Minimizing the overhead involved when switching between tasks and interrupts.
- **Priority-Based Scheduling:** Ensuring that high-priority tasks interrupt lower-priority ones to maintain system responsiveness.

In complex systems, the interrupt-driven approach not only provides the necessary real-time response but also enables robust multitasking and dynamic resource allocation, which are essential for modern RTOS environments.
To summarize, while poll-driven systems are suited for application firmware with predictable event timing, interrupt-driven designs become fundamental as system complexity grows. Interrupts allow the operating system to regain control from running processes, perform task switching, and ensure that the system meets its real-time requirements. Although cooperative scheduling exists and can be effective in simpler systems, interrupts, by enabling preemptive scheduling and even preemption within the kernel, are the only viable option for handling many asynchronous events. This approach is central to the development of real-time operating systems.

3.8 Conclusion

This chapter presented an overview of operating systems, emphasizing the characteristics that set real-time systems apart from general-purpose ones. In the next chapter, we turn our attention to FreeRTOS, detailing its key features to establish a basis for later fault injection experiments. These experiments will help identify FreeRTOS vulnerabilities, which, in turn, drive the development and evaluation of a selective hardening strategy later in the thesis.

Chapter 4

FreeRTOS

4.1 Introduction

In the previous chapter, operating systems were defined, real-time operating systems were characterized, and their distinctions from general-purpose systems were highlighted. This chapter focuses on a specific RTOS implementation: FreeRTOS [9, 10]. An overview of FreeRTOS, including its key properties and advantages, is presented.

4.2 FreeRTOS

FreeRTOS is an open-source, MISRA (Motor Industry Software Reliability Association) compliant RTOS implemented in C and maintained by Real Time Engineers Ltd. Designed for both micro-controller and microprocessor based embedded platforms, it offers a lightweight, deterministic scheduler plus inter-task communication primitives (queues, semaphores, mutexes) and essential resource management (memory allocation, timers, interrupt handling). Its modular design can meet hard or soft real-time requirements while keeping the code footprint small enough for deeply embedded applications.

4.3 FreeRTOS Strengths

Using FreeRTOS as the RTOS of choice offers several notable advantages, including:

- **Open-Source and Professionally Supported:** FreeRTOS enjoys remarkable global success due to its compelling value proposition. It is developed by professional teams and undergoes rigorous quality control, ensuring robustness and reliable support. Its open-source nature permits unrestricted commercial

use without licensing fees or mandatory disclosure of proprietary source code, a significant benefit for companies bringing products to market. Additionally, for organizations requiring enhanced legal assurances, written guarantees, or indemnification, an affordable commercial upgrade path is available. This option provides further peace of mind and the flexibility to adopt a commercial support arrangement as needed.

- **Reliability:** FreeRTOS is celebrated for its stability and dependable performance. It has been extensively implemented in real-time critical applications, ranging from medical devices and automotive systems to industrial control solutions. Its carefully engineered, lightweight architecture facilitates the consistent execution of real-time tasks without compromising overall system performance.

- **Modularity:** FreeRTOS is built with a highly modular structure, just five source files (plus headers) are needed for a basic application. This lightweight design lets developers include only the components they require. Over time, a wide variety of software components and libraries have been successfully ported to FreeRTOS, highlighting its versatility and ease of integration.

- **Hardware support:** Thanks to its clean, modular architecture, FreeRTOS is highly portable across a wide range of embedded processors. It has been deployed on simple 8-bit micro-controllers (for example AVR (Advanced Virtual RISC) and PIC families), 16-bit devices (such as MSP430), and an extensive array of 32-bit cores, including ARM (Avanced Risc Machines) Cortex-M, RISC-V, Tensilica Xtensa, and Renesas RX. On more advanced platforms, FreeRTOS can leverage hardware features like MPU (Memory Protection Unit) to enforce secure separation between kernel and application code or between high and low-privilege tasks.

- **Code size:** The core of a FreeRTOS-based embedded application requires only five source files, demonstrating its extremely compact code footprint. This minimalistic design makes FreeRTOS particularly well-suited for resource-constrained environments where more feature-rich RTOS options might be too demanding.

- **Developer Community:** As an open-source project, FreeRTOS has fostered a vibrant and active community over the years. Developers and users alike contribute to its improvement through discussion forums, detailed documentation, and a wealth of examples. This collaborative atmosphere not only facilitates prompt support and insightful answers to technical questions but also helps developers get started swiftly and resolve issues efficiently during development.

4.4 FreeRTOS Features

As briefly outlined above, FreeRTOS can be viewed both as a real-time kernel or, more simply, as a library providing scheduling facilities for embedded applications.

4.4.1 Task Scheduling

In FreeRTOS, execution units are called *tasks* and more closely resemble processes in other operating systems rather than threads, since each task provides its own execution context and cannot contain multiple threads of execution. Although some ports offer an MPU for memory segregation, the default FreeRTOS model places both all tasks and the kernel itself into a single, shared address space.
Two fundamental scheduling schemes are provided: (see Figure 4.1)

- **Cooperative Scheduling:** Tasks voluntarily relinquish control of the CPU by calling `taskYIELD()` or by blocking on synchronization primitives. The scheduler only performs a context switch when the running task explicitly yields or enters a blocked state.
- **Preemptive Scheduling:** At any instant, the highest-priority task in the Ready state is selected for execution. A context switch occurs immediately when a task of higher priority becomes ready (for example, due to an interrupt unblocking it), or when the current task blocks or yields.

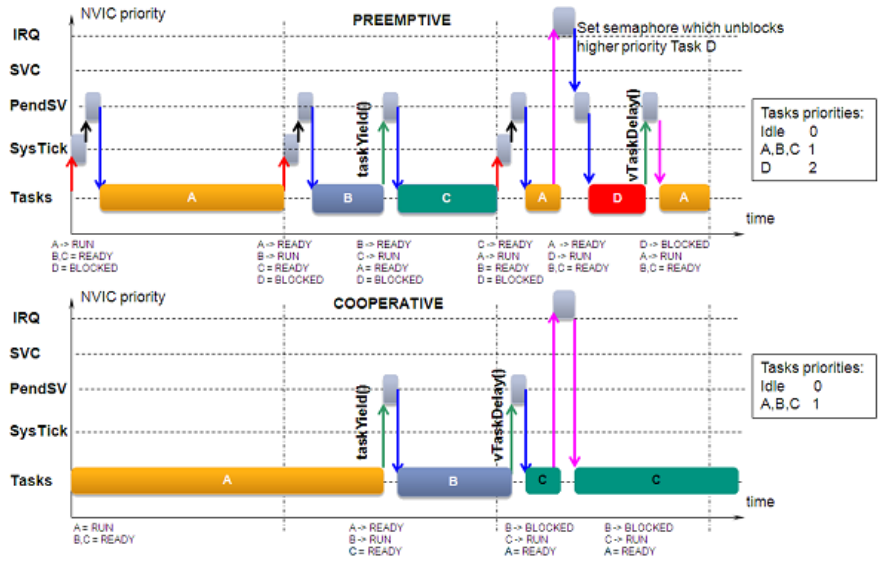

Figure 4.1: Scheduling types in FreeRTOS

4.4.2 Timing Services

FreeRTOS relies on a hardware timer, commonly referred to as the system tick, to provide its fundamental timing services. When the scheduler is launched via a call to `vTaskStartScheduler()`, the port layer configures this timer to generate periodic interrupts. Each tick interrupt increments the RTOS tick counter, which underpins task delays, timeouts, and the internal software timer mechanism.

4.4.3 Memory Management

FreeRTOS provides flexible memory management through both dynamic and static allocation schemes. Its built-in heap allocator supports configurable strategies, such as First Fit, Best Fit, and Worst Fit, to help control fragmentation. The desired allocation policy is selected at compile time by including the corresponding heap implementation file.
In addition, many FreeRTOS services offer two variants of their API (Application Programming Interface) calls, one that uses dynamic allocation and one that works with statically allocated memory buffers. This dual-API design makes it possible to build fully deterministic, mission-critical systems in which dynamic memory allocation is prohibited. By relying exclusively on static buffers and compile-time configuration, developers can avoid the unpredictability and potential fragmentation associated with heap usage, ensuring the system's timing and behavior remain fully predictable.

4.4.4 IPC Services

FreeRTOS provides several inter-task communication and synchronization primitives, enabling tasks and interrupt service routines to exchange data and coordinate execution:

- **Queues:** Typed, thread-safe FIFO structures for passing fixed-size data items between tasks or between ISR and tasks. Queues handle mutual exclusion internally, can block callers when full or empty with a user-specified maximum wait time to avoid unpredictability, and include API variants that are safe to call from interrupt (ISR) context.
- **Stream and Message Buffers:** Byte-oriented, FIFO buffers that allow variable-length messages to be sent and received. Stream buffers are optimized for a single sender/receiver pair, while message buffers support multiple readers and writers.
- **Semaphores:** Counting and binary semaphores for signaling and resource counting. Counting semaphores maintain a count of available resources; binary

semaphores act as simple event flags. Both can be used for task-to-task or ISR-to-task notifications.

- **Mutexes:** Priority-inheritance mutexes that protect shared resources while avoiding priority inversion. Mutexes can be recursively taken by the owning task and support priority inheritance. If a higher-priority task attempts to take a mutex already owned by a lower-priority task, the owner temporarily inherits the higher priority until it releases the mutex.
- **Event Groups:** Bit-masked event flags that allow tasks to wait for combinations of events to be set or cleared. Event groups support waiting for any or all bits, making them suitable for complex synchronization patterns.
- **Task Notifications:** Lightweight, per-task 32-bit values that can be used instead of semaphores or queues for fast, low-overhead signaling. Notifications support direct-to-task data passing, counting, and bit-based event flags.

Beyond these core services, FreeRTOS includes a number of other optional capabilities.

4.5 Scheduling in FreeRTOS

4.5.1 Task Control Block

In FreeRTOS, each task is represented by a *Task Control Block* TCB, defined in `tasks.c` as the `tskTCB` structure. This structure aggregates all kernel state required for task management, context switching, and debugging. A canonical layout of the TCB follows:

Listing 4.1: FreeRTOS struct tskTaskControlBlock description

```
typedef struct tskTaskControlBlock
{
    volatile StackType_t * pxTopOfStack;
    #if ( portUSING_MPU_WRAPPERS == 1 )
        xMPU_SETTINGS xMPUSettings;
    #endif
    ListItem_t             xStateListItem;
    ListItem_t             xEventListItem;
    UBaseType_t            uxPriority;
    StackType_t           *pxStack;
    char                   pcTaskName[ configMAX_TASK_NAME_LEN ];
    #if ( ( portSTACK_GROWTH > 0 ) || (
   configRECORD_STACK_HIGH_ADDRESS == 1 ) )
        StackType_t       *pxEndOfStack;
    #endif
```

```
15      #if ( portCRITICAL_NESTING_IN_TCB == 1 )
16          UBaseType_t         uxCriticalNesting;
17      #endif
18      #if ( configUSE_TRACE_FACILITY == 1 )
19          UBaseType_t         uxTCBNumber;
20          UBaseType_t         uxTaskNumber;
21      #endif
22      #if ( configUSE_MUTEXES == 1 )
23          UBaseType_t         uxBasePriority;
24          UBaseType_t         uxMutexesHeld;
25      #endif
26      #if ( configUSE_APPLICATION_TASK_TAG == 1 )
27          TaskHookFunction_t  pxTaskTag;
28      #endif
29      #if ( configNUM_THREAD_LOCAL_STORAGE_POINTERS > 0 )
30          void *              pvThreadLocalStoragePointers[
    configNUM_THREAD_LOCAL_STORAGE_POINTERS ];
31      #endif
32      #if ( configGENERATE_RUN_TIME_STATS == 1 )
33          configRUN_TIME_COUNTER_TYPE ulRunTimeCounter;
34      #endif
35      #if ( configUSE_NEWLIB_REENTRANT == 1 )
36          struct _reent       xNewLib_reent;
37      #endif
38      #if ( configUSE_TASK_NOTIFICATIONS == 1 )
39          volatile uint32_t ulNotifiedValue[
    configTASK_NOTIFICATION_ARRAY_ENTRIES ];
40          volatile uint8_t  ucNotifyState[
    configTASK_NOTIFICATION_ARRAY_ENTRIES ];
41      #endif
42      #if ( tskSTATIC_AND_DYNAMIC_ALLOCATION_POSSIBLE != 0 )
43          uint8_t             ucStaticallyAllocated;
44      #endif
45      #if ( INCLUDE_xTaskAbortDelay == 1 )
46          uint8_t             ucDelayAborted;
47      #endif
48      #if ( configUSE_POSIX_ERRNO == 1 )
49          int                 iTaskErrno;
50      #endif
51  } tskTCB;
```

The principal members of `tskTCB` are:

pxTopOfStack Pointer to the current top of the task's stack. On a context switch, CPU registers are saved here. This must be the first member.

xMPUSettings (optional) MPU configuration, if memory-protection wrappers are enabled.

xStateListItem List node linking the TCB into a state list (Ready, Blocked, Suspended).

xEventListItem List node used when the task is blocked on an event (queue, semaphore, timer).

uxPriority Current (possibly inherited) task priority; lower numerical value corresponds to lower priority.

pxStack Base address of the task's stack memory, for overflow checking and deallocation.

pcTaskName Null-terminated name string, used for debugging and trace output.

pxEndOfStack Highest valid stack address, when stack reporting is enabled.

uxCriticalNesting Depth of nested critical sections, for ports that track this in the TCB.

uxTCBNumber, uxTaskNumber Unique identifiers for tracing and debugger awareness.

uxBasePriority, uxMutexesHeld Priority-inheritance bookkeeping for mutexes.

pxTaskTag (optional) User-defined tag for application-specific task hooks.

pvThreadLocalStoragePointers (optional) Array of pointers reserved for thread-local storage.

ulRunTimeCounter (optional) Accumulated run time, for profiling if enabled.

`xNewLib_reent` (optional) Newlib reentrancy structure, if Newlib C library support is requested.

ulNotifiedValue, ucNotifyState (optional) Arrays holding values and states for task notifications.

ucStaticallyAllocated Indicates whether the task was created using static allocation.

ucDelayAborted Flag set when a blocking delay is prematurely aborted.

iTaskErrno (optional) Per-task `errno` for POSIX-style error reporting.

Together, these fields encapsulate the complete execution context, scheduling metadata, and optional tracing or protection settings for each FreeRTOS task. During a context switch, the port layer saves the CPU state to the task's stack and updates `pxTopOfStack`; it then loads the new task's stack pointer from its TCB to resume execution.

4.5.2 Task States

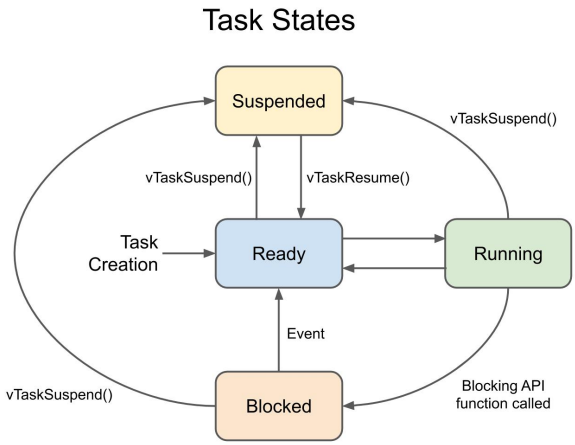

Figure 4.2: State transitions of FreeRTOS tasks

FreeRTOS tasks exist in one of four primary states. Figure 4.1 illustrates these states and the events or API calls that trigger transitions between them.

- **Ready** A task is Ready when it can run but is not currently executing. All Ready tasks are kept in priority-ordered queues.
- **Running** The Running state is occupied by exactly one task at any time: the highest-priority task in the Ready state that the scheduler has dispatched.
- **Blocked** (or *Waiting*) A task moves to Blocked when it invokes a blocking operation, such as `vTaskDelay()`, `xQueueReceive()`, or `xSemaphoreTake()`, or waits on a timeout or event. Blocked tasks consume no CPU time until the blocking condition clears.
- **Suspended** A task is Suspended when the application calls `vTaskSuspend()`. Suspended tasks remain in memory but are excluded from scheduling until explicitly resumed with `vTaskResume()` or `xTaskResumeFromISR()`.

- **Deleted** When a task is deleted (`vTaskDelete()`), it moves to the Deleted state. Its resources are reclaimed either immediately (if called from the task itself) or when the idle task runs cleanup code. As this is a special state it is not shown in the above figure.

State transitions:

- Ready → Running: the scheduler dispatches the highest-priority Ready task (on startup, after a context switch, or when a higher-priority task becomes Ready).
- Running → Ready: the current task yields (`taskYIELD()`), its time slice expires (if round-robin is enabled), or a higher-priority task becomes Ready.
- Running → Blocked: the task calls a blocking API or delays itself by calling `vTaskDelay()`.
- Blocked → Ready: the blocking condition clears (timeout elapses, notification arrives, or queue/semaphore is signaled).
- Ready → Suspended: the application calls `vTaskSuspend()`.
- Suspended → Ready: the application calls `vTaskResume()` or by calling `xTaskResumeFromISR()`.

Understanding these four states and their transitions is essential for designing predictable real-time behavior under both preemptive and cooperative scheduling modes.

4.6 FreeRTOS Source Components

A standard default configuration comprises the following eleven source files, each addressing a specific functional domain:

port.c Implements the hardware-abstraction layer required by the scheduler. It contains CPU and compiler-specific routines for context switching, scheduler initialization and shutdown, and any necessary inline assembly.

heap_n.c Offers one of five selectable heap-management schemes (heap_1 through heap_5). Each variant encapsulates a different strategy for dynamic memory allocation.

FreeRTOSConfig.h Serves as the primary configuration header. All compile-time parameters, such as task priorities, tick rate definitions, enabled kernel

features, and hook function overrides, are specified within this file. Additional headers that introduce custom hooks or extend the kernel's capabilities must be included at its end.

FreeRTOS.h Acts as a facade, exposing or concealing kernel API in accordance with the definitions provided in `FreeRTOSConfig.h`. It also declares opaque "dummy" structures that mask the kernel's internal data types, thereby preventing inadvertent access from application code. For example, the TCB (Task Control Block) type is defined privately in `tasks.c` and is never exposed in a public header. Instead, a dummy TCB structure is declared elsewhere so application code can only hold pointers to it (which are called **Task Handles**) without ever seeing its contents. This design enforces that all interactions with the TCB must go through the official FreeRTOS API rather than by direct field access, aligning with encapsulation principles and proper software engineering practices.

task.h / tasks.c Constitute the core task-management subsystem. These files implement task creation, deletion, scheduling, and context-switch policy (the mechanism is implemented by the portable layer), forming the central scheduling engine of the RTOS.

queue.h / queue.c Define the queue abstraction and its associated enqueue/dequeue operations. Semaphores and mutexes are implemented as specialized instances of this queue mechanism, allowing for efficient reuse of the underlying code.

semphr.h Provides a set of macros that map semaphore and mutex API onto the queue primitives. These macros configure the queue functions with predefined parameters to realize mutual-exclusion and signaling constructs.

list.h / list.c Deliver a generic doubly linked-list implementation. This module underpins various RTOS data structures, such as ready lists, timer lists, and delay lists, by offering insertion, removal, and traversal operations.

4.7 Conclusion

This chapter has provided an overview of FreeRTOS, emphasizing its key advantages and core functionalities, including the details of its scheduling mechanism. These concepts form the basis for the software fault-injection framework developed in this thesis, which leverages the POSIX-based FreeRTOS port. Additional FreeRTOS features used and supporting source files will be discussed in the relevant chapter.

Chapter 5

Fault Injection

5.1 Introduction

In the preceding chapters, a clear progression was established: from defining single event effects and their impact on the reliability of electronic systems, particularly embedded devices running an RTOS, to an overview of real-time operating systems and a detailed introduction to the specific RTOS under study, *FreeRTOS*. The introductory chapter also highlighted the need for fault-injection testing as a foundation for effective SEE mitigation strategies. This chapter now turns to the fundamentals of fault injection, preparing the way for the presentation of the *FreeRTOS* based software fault injection simulator developed for this research.

5.2 Dependability

As noted in the introductory chapter, dependability [11] [12] refers to a system's ability to carry out its intended function correctly and reliably whenever it is called upon. In other words, a dependable system is one you can trust to respond appropriately under all anticipated conditions, be they routine or exceptional. Dependability is made of attributes, threats and means.

5.2.1 Attributes

Attributes are the desirable qualities that together characterize a dependable system.

Core Attributes

- **Availability:** The system is ready for correct service when requested.

Figure 5.1: Attributes of dependability

- **Reliability:** The system performs its intended function without failure over a specified period.
- **Safety:** The system operates without causing unacceptable risk of harm to people or the environment.
- **Integrity:** The system prevents unauthorized or unintentional alteration of data or functionality.
- **Maintainability:** The system can be repaired, modified, or enhanced efficiently after a failure or as requirements evolve.

Additional Attributes

These supplementary attributes have emerged alongside our rapidly evolving technological landscape. For example, the rise of computer networks, most notably the Internet, has brought new concerns such as security to the forefront.

- **Security:** Security is the attribute that ensures a system's assets, data, services, and resources, are protected against unauthorized access, use, disclosure, disruption, modification, or destruction. A secure system provides:
  - **Confidentiality:** only authorized parties can read or view sensitive information.
  - **Integrity:** data and operations cannot be altered by unauthorized actors, either accidentally or maliciously.
  - **Availability:** security controls themselves do not unduly prevent legitimate users from accessing system functions when needed.

- **Authentication and Authorization:** Users and components are reliably identified, and their permitted actions are strictly enforced.
- **Accountability (Auditability):** Security-relevant events are logged so that actions can be traced back to responsible entities.

Together, these facets ensure that the system resists attacks, detects breaches, and recovers gracefully, preserving trust in its correct and intended operation.

5.2.2 Threats

Threats are the causes that can impair or violate one or more of the dependability attributes. They can be classified as:

- **Fault:** An underlying defect or flaw, physical or logical, within a system that can disrupt correct operation when activated. Examples include a manufacturing defect in hardware or the observable effect of an SEE, such as an SEU-induced bit-flip in device memory.

- **Error:** An error is an incorrect decision or action, typically by a human or automated process, that leads to the introduction of a fault into the system. Errors may occur at any phase of the life-cycle of a system (design, implementation, testing, deployment, or maintenance) and include:

  - Flawed design choices or architectural omissions,
  - Programming bugs or logic mistakes,
  - Incomplete or missing requirements,
  - Invalid data entry or configuration by an operator.

  For instance, a developer's typo in source code or an administrator's misconfigured parameter both constitute errors that can give rise to faults.

- **Failure:** A *failure* is the externally observable incorrect behavior resulting from an activated fault. For instance, an SEU-induced bit-flip in an RTOS pointer variable may lead to a system crash, this crash is the failure. Not all faults cause failures: if a fault occurs in a component that does not affect user-visible outputs or system functionality, it remains latent and does not manifest as a failure.

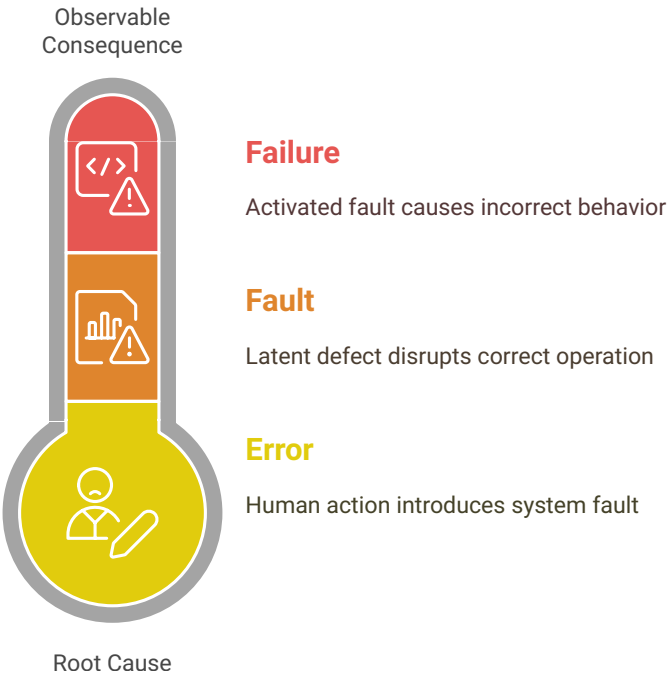

Figure 5.2: Threats of dependability

5.2.3 Means

A system's dependability can be enhanced through techniques that either mitigate the impact of faults or, ideally, render the application entirely immune to them. These methods fall into four main categories: **fault prevention**, **fault forecasting**, **fault tolerance**, and **fault removal**.

Fault Prevention

Fault prevention encompasses techniques that reduce the likelihood of faults occurring. For instance, as discussed in Chapter 2, mission-critical hardware often uses a hardened fabrication process that adds an insulating layer to prevent single event gate ruptures (SEGR) in CMOS based electronics. Commercial off-the-shelf (COTS) devices typically omit such measures because their dependability requirements are lower and cost pressures higher. In contrast, systems with stringent reliability needs, medical equipment, military hardware, and electronics for space or low-Earth orbit, rely on these more expensive, hardened manufacturing processes to ensure continuous, fault-free operation.

Fault Forecasting

Fault forecasting aims to anticipate where and how faults might affect a system. This involves a thorough analysis of the design and implementation, often employing techniques such as fault-injection testing, to uncover weaknesses before they manifest. In this study, fault forecasting is used to pinpoint the system's most vulnerable components and guide targeted hardening efforts in those high-risk areas.

Fault Tolerance

Fault tolerance is a technique that aims at equipping a system with the capacity to continue doing its work even in the adverse cases of faults happening. As mentioned in fault forecasting in this work a fault forecasting technique such as fault injection is used to identify weak areas of the system and then a targeted hardening approach enables the system to keep going even in the presence of faults.
Fault-tolerance techniques fall into two broad categories: hardware-based and software-based.

Hardware-Based Techniques

- Advanced fabrication processes and component selection
- Redundancy at the hardware level, which can be implemented as:
  - Gate-level redundancy (duplicate or triplicate critical logic gates)
  - Module or system-level redundancy (extra processors, power supplies, or I/O paths)

Software-Based Techniques

- Data redundancy (variable duplication or triplication)
- Error-correcting codes (ECC) in memory and storage
- Check-pointing and rollback recovery (periodic state snapshots with the ability to revert after a fault)

By combining both hardware and software approaches, systems can detect, mask, or recover from faults with minimal interruption to service.

Fault Removal

Fault removal encompasses techniques for detecting and eliminating defects as early as possible, ideally before deployment or at the onset of operation. By catching faults during development and testing, their cost and impact are minimized.
Key practices include:

- **Static analysis and code inspection**
  - Automated linters, type checkers, and static analyzers
  - Manual peer reviews of requirements, design documents, and source code
- **Dynamic testing**
  - Unit tests, integration tests, system tests, and acceptance tests under normal, boundary, and stress conditions
  - Test harnesses, simulators, and emulators to exercise corner cases
- **Formal verification**
  - Model checking or theorem proving for safety-critical components
  - Proofs of algorithmic correctness and adherence to specifications
- **Continuous integration and deployment (CI/CD)**
  - Automated build pipelines with regression testing
  - Early feedback loops to catch and fix faults immediately

Together, these methods drive fault density down, bolstering the overall dependability of the system.

5.3 Fault injection testing

Fault injection testing [3] is a standard industry practice for assessing a system's dependability. By deliberately injecting faults, it evaluates robustness under adverse conditions and uncovers latent vulnerabilities. Various techniques, ranging from physical stressors to logical and hardware, or software-based injections, can be employed to match different testing objectives.

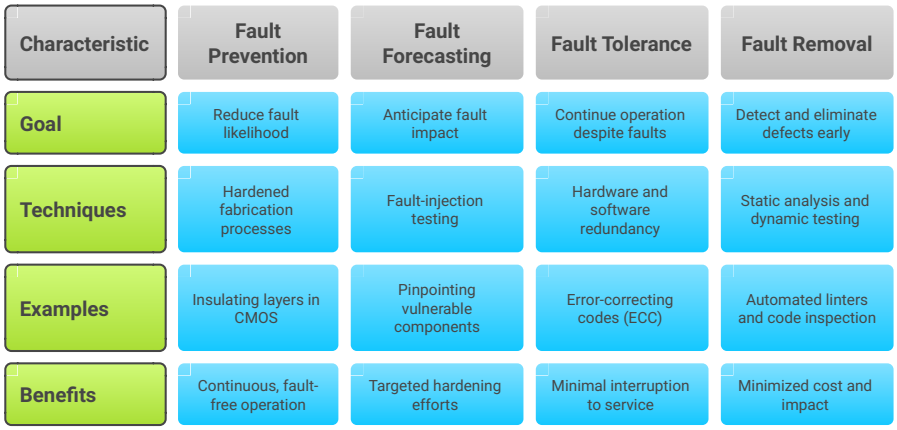

Figure 5.3: Means of dependability

5.3.1 Fault Injection Techniques

Fault injection methods fall into three broad classes, selected according to desired fidelity, flexibility, and cost:

1. **Physical Testing:** reproduces real-world stressors with high fidelity
2. **Hardware and Software Based Logical Testing:** injects faults via added circuitry or runtime perturbation
3. **Simulation Based Injection:** employs virtual platforms and cycle-accurate models

Physical Testing

Physical testing recreates the actual environmental or radiation conditions responsible for faults (e.g. single event effects). Equipment such as a linear accelerator, vacuum chamber, or EMI generator is used to induce errors in situ. Despite its accuracy, this approach is costly, inflexible, and generally incapable of targeting internal registers or specific memory locations, especially on closed IP (Intellectual Property) hardware that must first be reverse engineered.

Logical Testing

Logical fault injection may rely on either dedicated hardware circuitry or purely software driven techniques.

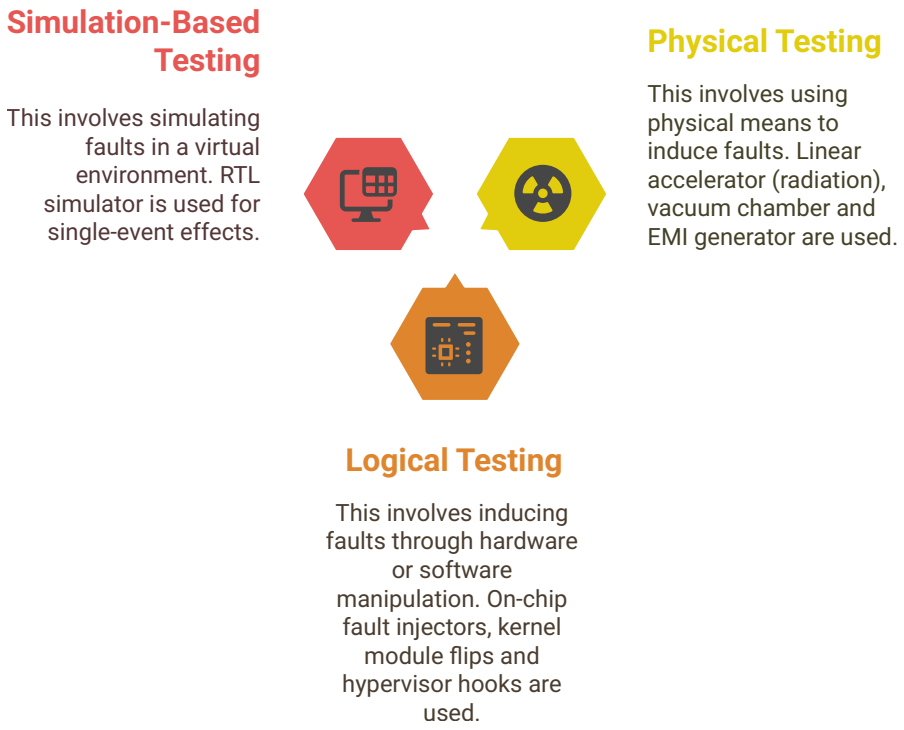

Figure 5.4: Types of Fault Injection Testing Techniques

Hardware Based Logical Injection In this mode, the target device incorporates an on-chip fault-injection module that can be activated during test runs. A control interface, often implemented as a simple serial or memory-mapped protocol, allows an external test controller to specify when and where faults are introduced. An alternative laboratory setup employs probe fixtures or "nail" adapters that physically drive chosen PCB (Printed Circuit Board) traces to known logic levels, thereby emulating signal corruption or stuck-at conditions. The embedded module approach typically incurs silicon area overhead, while probe based methods can require expensive instrumentation and precise fixture alignment.

Software Based Logical Injection (SWIFI) Software driven injection exploits routines with privileges or low-level access kernel modules, bootloader patches, or hypervisor hooks, that deliberately modify data structures, registers, or memory cells at runtime. Although this approach yields genuine bit flips or control-flow faults, the insertion and execution of the injection code itself can alter timing characteristics, potentially masking or amplifying certain failure modes.
The following chapter describes a SWIFI based injector designed to target real time operating system kernel objects with fine granularity, without recourse to specialized hardware.

Simulation Based Testing

In simulation based fault injection, error events are emulated within a virtual model of the hardware rather than being applied to a physical device. A hardware description (e.g., VHDL or Verilog) is exercised under a simulator that supports the insertion of single event effects (SEE), timing glitches, or stuck-at faults at chosen points in the design. Since the actual silicon is not required, this method offers rapid iteration and fine-grained control over injection timing and location in the RTL (Register Transfer Level) code.
When the focus shifts to software robustness, such as examining the impact of random bit flips on application data, simulation alone can be limiting. Although it remains possible to introduce faults at the RTL, pinpointing and corrupting a specific high-level memory cell or data structure in a full-system simulator is often impractical. Address remapping, dynamic allocation, and compiler optimizations obscure how source-level variables map onto physical memory, preventing precise insertion of faults into particular bytes or bits. In the context of an RTOS, this challenge becomes especially pronounced when attempting to target individual kernel data structures directly.

5.4 State of the Art

A concise overview of notable fault-injection frameworks follows [12, 13].

5.4.1 FIAT

Barton *et al.* introduced the Fault Injection and Testing framework (FIAT (Fault Injection Based Automated Testing)) for kernel-space fault injection [14]. Three abstract data fault models, decoupled from any physical fault source, are applied to the task image, perturbing both user-space applications and kernel code. Although pioneering, FIAT lacks the granularity needed to corrupt specific internal RTOS data structures.

5.4.2 MAFALDA

Arlat *et al.* presented MAFALDA (Microkernel Assessment by Fault Injection Analysis and Design Aid), a micro-kernel oriented injector [15]. Faults are inserted at the byte level into API arguments and micro-kernel segments. This unguided approach exposes resilience gaps but provides limited insight into how faults propagate through core kernel objects.

5.4.3 FIFA

Jeong *et al.* devised FIFA (Kernel-Level Fault Injection Framework for ARM-Based Embedded Linux System) to reproduce real hardware errors on live boards [16]. Two mechanisms are offered:

- **KGDB-based injection**: halts execution via the GNU debugger to flip bits in registers or memory.
- **Hardware breakpoint injection**: uses ARM breakpoints to trigger faults on instruction or data access.

Transient bit-flips, timing delays, and device failures can thus be emulated. Despite its power, FIFA does not specifically target high-level kernel data structures.

5.4.4 RTOS Guardian

Silva *et al.* developed RTOSG (RTOS Guardian), a passive bus monitoring module that verifies RTOS scheduling integrity [17]. Key components include:

1. Task Controller (TC)
2. Function Identifier (FI)
3. List Monitor & Error Generator (LMEG)
4. Content-Addressable Memories (CAM1 & CAM2)

IEC 61000-4-29 experiments show superior coverage of EMI-induced faults, but focus remains on scheduler events rather than arbitrary kernel objects.

5.4.5 Positioning of the Proposed Injector

The above frameworks expose two gaps:

- Software-only injectors (FIAT, MAFALDA) operate at task-image or API-argument levels, lacking kernel-object granularity.
- Hardware assisted methods (FIFA, RTOSG) reach registers and memory but depend on external hooks or probes, and seldom relate faults back to OS internal data structures.

The approach introduced here addresses both gaps. Runtime mapping of core RTOS structures, task control blocks, ready-queue nodes, timer lists, etc., combined with bit-level corruptions injected via a lightweight mechanism sharing the RTOS address space, delivers:

1. Precise, structure-level targeting of kernel data without compiler or silicon modifications.
2. Programmatic control over injection timing and location, enabling reproducible, systematic campaigns.
3. Direct correlation of faults in specific RTOS internal data structures with observable system level outcomes.

This methodology bridges coarse grained, bit-flip campaigns and heavyweight, hardware centric tests, offering high fidelity in OS state fault modeling alongside the flexibility of a purely software solution. The next chapter details its design and implementation.

5.5 Conclusion

This chapter has expanded the concept of dependability by defining its core attributes, identifying prevalent threats, and outlining established evaluation techniques. Fault injection was highlighted as a versatile methodology for assessing system resilience, followed by a concise survey of state-of-the-art approaches. The shortcomings and focus areas revealed by that survey motivated the design of the current software-based fault injector, which directly corrupts RTOS internal data structures to deliver more fine-grained, realistic, and reproducible resilience evaluations.
The next chapter presents the custom fault injector developed for this research. Implemented as a software-based framework for *FreeRTOS*, it supports both transient bit-flips SEU and permanent bit-flips SEHE within kernel data structures, enabling a controlled, systematic analysis of the operating system's fault-tolerance capabilities.

Chapter 6

FreeRTOS Fault Injector

6.1 Introduction

The preceding chapters established the necessary foundation by first examining the threats faced by real-time operating systems in harsh conditions, most notably ionizing radiation, and detailing the classification of resulting single-event effects (SEE). An overview of RTOS design principles followed, with *FreeRTOS* identified as the case study platform. Subsequent discussion characterized dependability and highlighted fault-injection testing as the de facto industry technique for evaluating system robustness.
This chapter concludes the introductory sequence by describing the *FreeRTOS Injector*, a software-based fault-injection framework that introduces both transient and permanent bit-flips into FreeRTOS kernel data structures to facilitate a systematic assessment of the operating system's resilience.

6.2 Background, Prior Implementations, Divergences

This thesis builds on an earlier framework [18] for fault-injection in *FreeRTOS* that supported transient SEU and SEHE faults. That prior work identified the kernel's fault-sensitive regions and provided a platform for injection campaigns. Using those insights, this project implemented a selective hardening strategy to protect only the most critical code paths.
Despite this inheritance, the current implementation diverges substantially while preserving the original semantics:

- The codebase was reorganized into clean submodules, and numerous memory-management bugs were fixed.

- Permanent fault support, originally driven by a shell script, was migrated into the CMake build system and extended to the Windows port via the PCRE2 (Perl Compatible Regular Expressions (version 2)) library [19].
- The POSIX port's portable layer now checks every POSIX system-call return value and incorporates simulated interrupts matching the *Windows* port semantics.
- A `"-dry-run"` option was added to the existing `"-campaign"` command. It outputs the selected fault-injection targets without executing them, enabling reproducible campaigns and making random selections explicit.
- The `-campaign` command can now export campaign results to a user-specified CSV file.
- If no `-j` option is supplied, `-campaign` automatically spawns as many concurrent processes as there are CPU threads (queried via the underlying OS API).
- The project now seeds `rand()` with `srand()` using a high-quality entropy source when available (on Linux, via the `getrandom()` system call).
- The selective hardening mechanism itself was designed and implemented as the core contribution of this thesis.
- On *Linux* (the platform for this work), FreeRTOS port-layer threads and tasks can, when permissions allow, be scheduled under the POSIX-defined **SCHED_RR** policy, common on many *UNIX*-like systems. This setup enables simulations to execute under conditions that more closely approximate real-time behavior. Future work could introduce Linux's **PREEMPT_RT** real-time kernel patches (merged into mainline as of version 6.12) to improve timing determinism, and separately evaluate the Linux-exclusive **SCHED_DEADLINE** policy as an alternative for hard real-time scheduling.

In addition, both the POSIX task I/O library and the *FreeRTOS Injector*'s I/O module have been designed for maximum debuggability: when the project is built with the `DEBUG` option, they emit detailed, verbose output. The primary user-facing scripts, `posix.sh` and `windows.bat`, were created from scratch to streamline common workflows. These scripts:

- Build the project on POSIX or *Windows* platforms.
- Configure and launch fault-injection campaigns.

- Automatically patch the *FreeRTOS* source code for permanent-fault emulation.
- Execute any required dependencies.

Together, these components provide a user-friendly, script-driven interface and enhance transparency throughout the injection process.

6.3 Project Structure

The implementation adheres to the official *FreeRTOS* coding conventions and style guidelines [9], ensuring seamless integration with the existing codebase and simplifying navigation and comprehension.
Within the base directory of a *FreeRTOS* project cloned from the official Git repository, there is a single top-level folder containing the entire FreeRTOS Fault Injector. This folder is named **FreeRTOS-Injector**. Its internal layout is illustrated in Figure 6.1 which presents only the project's top-level view. Below, a concise yet more detailed outline of the key files will be shown.

6.3.1 Project files

- `CMakeLists.txt`: The **CMake** configuration file for the project's build system. It defines two build targets:

  1. The *FreeRTOS* Injector executable.
  2. An auxiliary tool that implements the permanent "stuck-at" fault model used for the SEHE single-event effect.

  The auxiliary tool injects permanent faults into the *FreeRTOS* source by applying targeted patches. It scans the RTOS code with regular expressions to locate operations on selected targets, such as accesses, increments, or decrements, and inserts a function call at each site to override the variable's value with the chosen fault. A full description of this patching mechanism is omitted here, since it has been documented in earlier work.

- `posix.sh` and `windows.bat`: A POSIX-compatible shell script and a Windows batch script that act as the project's primary interface. They bundle common setup and execution steps into simple commands, enabling users to start fault-injection campaigns immediately, no deep familiarity with the project structure is needed. Advanced users, however, can bypass these wrappers and interact directly with the executables and other project components as required.

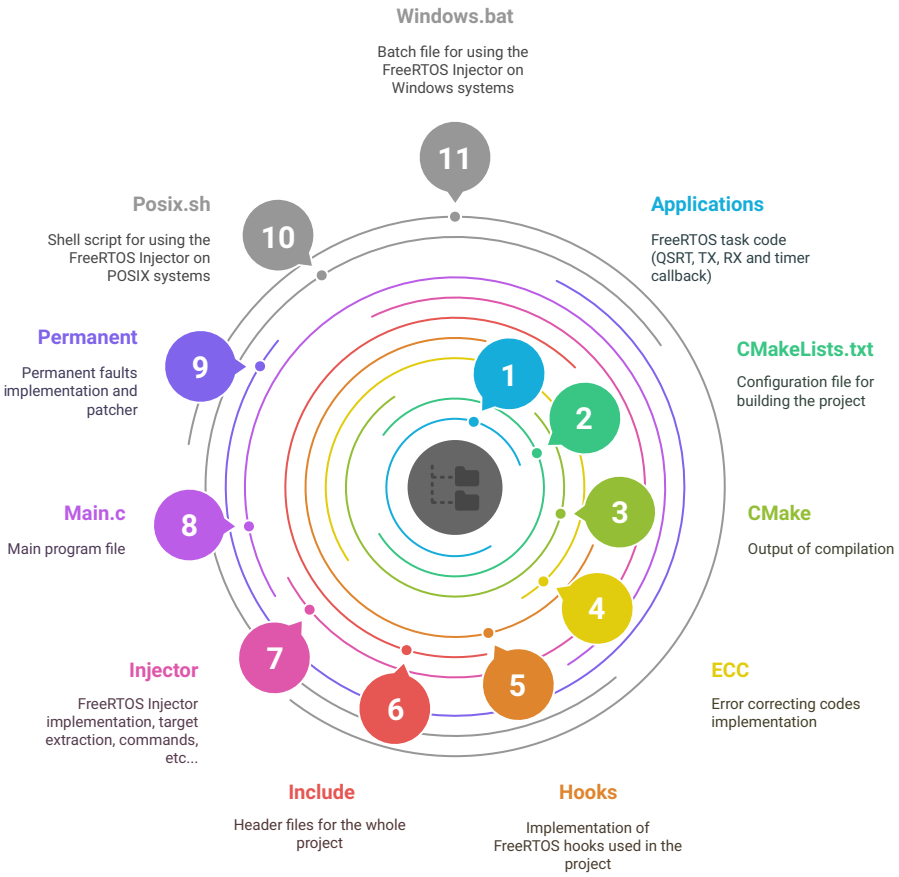

Figure 6.1: FreeRTOS Injector project structure

- `main.c`: The application's entry point on both POSIX and Windows. It initializes the corresponding *FreeRTOS* port, POSIX on Unix-like systems or the Windows port, so the full *FreeRTOS* kernel runs as a hosted user-space application. It then parses command-line arguments to dispatch the appropriate injector commands. By using these hosted ports, the *FreeRTOS Injector* operates entirely in software on any standard workstation, with no specialized hardware required.

- `task_main.c`: Located in the **Applications** directory, this file serves as the entry point for the user-space portion of the simulated *FreeRTOS* kernel. It sets up all *FreeRTOS* tasks and calls `vTaskStartScheduler()` to launch the RTOS scheduler.

- **Applications**: This directory contains the user-space components of the simulated FreeRTOS kernel. In addition to the `task_main.c` entry point, it includes:

  - An I/O library that wraps underlying OS I/O calls. It uses *FreeRTOS* mutexes to serialize access across concurrent tasks.
  - Implementations of three *FreeRTOS* tasks, Quicksort (QSRT), Receive (RX), and Send (TX), registered by those names in the *FreeRTOS* task API.
  - A timer callback function invoked periodically by a *FreeRTOS* software timer.
  - A subfolder `tacle_benchmarks` contains the ported benchmarks from the TACLe software suite.

- **Hooks**: This directory contains implementations of the *FreeRTOS* hook functions used by the injector. For example, the **"Idle"** task hook is invoked each time the *FreeRTOS* **Idle** task runs. These hooks allow custom code to execute at key points in the kernel's operation.

- **Injector** This directory houses the core *FreeRTOS Injector* implementation and is organized into four subdirectories:

  - **commands** Implements the CLI commands `-golden`, `-list`, `-run`, and `-campaign` in the files `command_golden.c`, `command_list.c`, `command_run.c`, and `command_campaign.c`, respectively. A **common** subfolder contains `common.c`, which provides utility functions shared across all commands.
  - **target** Contains `target.c`, responsible for extracting and acquiring the injection targets within the framework.
  - **tracer** Contains `tracer.c`, the logging engine for the injector. It hooks into FreeRTOS at key events, task switch-in/switch-out, queue send/receive success or failure, to record runtime behavior.
  - **os_deps** Provides an OS-abstraction layer for both POSIX and Windows:
    * `thread.c`: Wraps underlying threading primitives so the injector can create and manage threads uniformly.
    * `injection.c`: Supplies OS-independent routines used by the `-campaign` command, for example, spawning processes, waiting for termination and exit codes, creating and waiting on timers.

- **Include**: This directory holds all header files for the *FreeRTOS* Injector modules. Its subfolder layout mirrors the injector's internal structure. For example:

  `include/injector/commands/common/common.h`

corresponds to the source file `common.c`, enabling clean includes such as:

```
#include <injector/commands/common/common.h>
```

Of particular note are:

- `injector_config.h`: Modeled after FreeRTOS's `FreeRTOSConfig.h`, this header defines injector-specific configuration parameters, paths to required files, iteration counts for QSRT/TX/RX tasks, timeouts, and more.
- `injector_io.h`: Defines an I/O library for the injector framework that wraps native OS I/O calls. Unlike the I/O library used within *FreeRTOS* tasks, this layer serves the injector's own threads, those not managed by the *FreeRTOS* POSIX portable layer. Because the POSIX port uses **UNIX** signals to implement task switching, its signal setup and handling must be carefully coordinated to prevent deadlocks. A later section (see 6.4.7) will outline the POSIX portable layer's design and explain why this separate I/O abstraction is necessary.

- **Permanent**: Contains the standalone binary sources that patch *FreeRTOS* to introduce the permanent "stuck-at" fault model (as referenced in the `CMakeLists.txt`). Although this thesis centers on the POSIX injector, the permanent patcher has been ported to *Windows*. Because Windows standard C library lacks POSIX's `regex.h` interface, the PCRE2 library [19][20] was integrated to provide equivalent regular expression support. Building PCRE2 is orchestrated within `windows.bat` and the project's `CMakeLists.txt`.

- **ECC**: Contains a single source file, `hamming.c`, which implements the error-correcting code (ECC) mechanism used for selective hardening in this work. Further details are provided in a later chapter.

6.4 POSIX Port Layer Design

This section provides an overview of the POSIX port [1] layer design and implementation, which underlies the *FreeRTOS Injector* on UNIX-like systems. The POSIX port layer enables the full *FreeRTOS* kernel to execute as a hosted, user-space application and serves as the foundation for task scheduling, inter-task synchronization, and timing on POSIX platforms.

6.4.1 Overview Of the Port Layer Architecture

The POSIX port layer implements the *FreeRTOS* kernel as a purely user-space application on UNIX-like systems. At its core, each *FreeRTOS* task is represented

by a dedicated POSIX thread, while a separate scheduler thread manages timer events and shutdown signals. Task scheduling and preemption are driven by a periodic **SIGALRM** delivered via a POSIX timer, and context-switch coordination relies on per-thread condition variables.
Simulated hardware interrupts are handled by an auxiliary "interrupt" thread that invokes registered handlers. Critical-section semantics and the *FreeRTOS* API for entering/exiting critical regions are realized by masking and unmasking all signals in the calling thread. Finally, all I/O calls flow through a custom wrapper layer to avoid conflicts between the standard C library's internal locking and the signal-driven scheduler.
Together, these elements form a cohesive architecture that faithfully reproduces *FreeRTOS* behavior, task creation, scheduling, synchronization, timing, interrupts, and safe critical sections, entirely within a hosted POSIX environment.
Each component is examined in detail in the subsections that follow.

6.4.2 FreeRTOS Task Mapping

Each *FreeRTOS* task runs as a separate POSIX thread. During task creation, the *FreeRTOS* kernel invokes the port-specific initialization function:

`pxPortInitializeStack()`

This function stores a `Thread_t` descriptor at the top of the task's stack:

Listing 6.1: FreeRTOS Posix port Thread_t description

```
typedef struct THREAD {
    pthread_t pthread;  // POSIX thread handle
    pdTASK_CODE pxCode; // FreeRTOS Task function
    void *pvParams;     // FreeRTOS Task function parameters
    BaseType_t xDying;  // Port layer termination flag
    struct event *ev;   // Port layer syncrhonization event
} Thread_t;
```

The port then calls `pthread_create()` to spawn the thread. Next, a `struct event` object is allocated and initialized:

Listing 6.2: FreeRTOS Posix port struct event description

```
struct event {
    pthread_mutex_t mutex; // Protects the event state
    pthread_cond_t  cond;  // Condition variable for signaling
    bool event_triggered;  // Flag indicating signal status
};
```

Each new thread begins execution in the function:

`prvWaitForStart()`

which immediately calls:

`event_wait()`

Internally, `event_wait()` blocks on the thread's condition variable until the scheduler signals that the task may run.

6.4.3 Port Layer Initialization

When the *FreeRTOS* kernel is ready to start, it invokes

`vTaskStartScheduler()  % in tasks.c`

Inside `vTaskStartScheduler()`, the architecture specific entry point

`xPortStartScheduler()`

performs two main steps:

1. **Setup the tick interrupt.** Calls

   `vPortStartFirstTask()`

   This routine:

   - Obtains the first ready task handle via `xTaskGetCurrentTaskHandle()`.
   - Extracts its `Thread_t` object from the top-of-stack frame.
   - Signals the POSIX condition variable on which that task's pthread is blocked in `prvWaitForStart()`.
   - Upon receiving the signal, `prvWaitForStart()` resumes execution, retrieves the *FreeRTOS* task function and its parameters from the thread's `Thread_t` structure, and invokes it, thereby implementing the *FreeRTOS* task's execution.

Scheduler Initialization

Before any tasks run, the port must initialize signals and scheduling policy exactly once. To guarantee one-time setup, `pxPortInitializeStack()` uses POSIX's

`pthread_once()`

On the first call to `pxPortInitializeStack()`, which occurs when the very first *FreeRTOS* task is created via the RTOS task-creation API, the following function is invoked:

`prvSetupSignalsAndSchedulerPolicy()`

which performs:

1. **Block all signals.** Set a mask that blocks every signal, then SIGINT is explicitly unblocked. This ensures each new FreeRTOS task's pthread inherits a "block everything" policy while preserving *Ctrl-C* handling for terminating the injector or simulator and enabling *SIGINT*-driven breakpoints in GDB.

2. **Install the system tick handler.** With `sigaction()`, associate **SIGALRM** to `vPortSystemTickHandler()`.

3. **Spawn the simulated interrupt thread.** Create a pthread to emulate peripheral interrupts for simulating device interrupts.

4. **Prepare the scheduler thread.** The thread that invoked `xPortStartScheduler()` blocks all signals except **SIGUSR1** and waits in `sigwait()`. When **SIGUSR1** arrives, sent by a call to

   ```
   xPortEndScheduler()  % via vTaskEndScheduler()
   ```

   the scheduler cleanly shuts down and returns control to the code that originally called `vTaskStartScheduler()`. In this POSIX-simulated environment, this return path is essential, tools such as the *FreeRTOS Injector* rely on it to capture and analyze a complete *FreeRTOS* run. In contrast, on real embedded hardware the *FreeRTOS* scheduler typically runs indefinitely, terminating only in response to catastrophic faults or a full system reset.

6.4.4 FreeRTOS Task Switching

The core of task switching in the POSIX port is the **SIGALRM** handler:

Listing 6.3: FreeRTOS Posix port system tick handler

```
1  static void vPortSystemTickHandler(int sig)
2  {
3      Thread_t *toSuspend, *toResume;
4      UBaseType_t switchRequired;
5
6      /* Enter critical section */
7      uxCriticalNesting++;
8      switchRequired = xTaskIncrementTick();
9
10     #if (configUSE_PREEMPTION == 1)
11     if (switchRequired)
12     {
```

```
13          /* Identify current and next tasks */
14          toSuspend = prvGetThreadFromTask(xTaskGetCurrentTaskHandle())
    ;
15          vTaskSwitchContext();
16          toResume = prvGetThreadFromTask(xTaskGetCurrentTaskHandle());
17
18          /* Perform the context switch */
19          prvSwitchThread(toResume, toSuspend);
20      }
21      #endif
22
23      /* Exit critical section */
24      uxCriticalNesting--;
25  }
```

Workflow:

1. Block signals and enter critical section by incrementing `uxCriticalNesting`.
2. Tick the kernel with `xTaskIncrementTick()`, which returns whether a context switch is required.
3. If preemption is enabled *and* a switch is required:
   (a) Look up the current task's `Thread_t`.
   (b) Call `vTaskSwitchContext()` to select the next ready task.
   (c) Look up the next task's `Thread_t`.
   (d) Call `prvSwitchThread()` to block the outgoing pthread and unblock the incoming one.
4. Exit the critical section by decrementing `uxCriticalNesting`.

This signal-driven mechanism ensures each tick can preempt a running task and switch to the highest-priority ready task.

6.4.5 Simulated Interrupts

The POSIX port layer has been extended to support simulated device interrupts, functionality that was not present in the original implementation [1]. Inspired by the *Windows* port, the author added:

- A dedicated pthread that drives the interrupt simulation.
- API functions:
  - `vPortSetInterruptHandler()` Registers a custom handler for interrupt numbers 0–32.

– `vPortGenerateSimulatedInterrupt()` Schedules a previously registered handler to execute as if the hardware interrupt had occurred.

The dedicated interrupt-manager thread uses **SIGUSR2** to avoid polling. A call to `vPortGenerateSimulatedInterrupt()` sets the bit in an internal bitmask corresponding to the simulated interrupt number and then sends **SIGUSR2** to that thread. The interrupt manager thread sits in a `sigwait()` loop awaiting this signal; upon receipt, it scans the bitmask for pending interrupts and invokes any registered handler(s).
This simulated-interrupt infrastructure is a critical dependency of the *FreeRTOS Injector*, which will be explained in the 6.5.3 section.

6.4.6 Enabling and Disabling Interrupts

The portable layer exposes two API functions, `vPortDisableInterrupts()` and `vPortEnableInterrupts()`, which mask and unmask all signals within the calling pthread. Their implementations are shown below:

Listing 6.4: FreeRTOS Posix port interrupt management functions

```
1  void vPortDisableInterrupts(void)
2  {
3      int ret = pthread_sigmask(SIG_BLOCK, &xAllSignals, NULL);
4      if (ret != 0)
5      {
6          prvFatalError("pthread_sigmask", ret);
7      }
8  }
9
10 void vPortEnableInterrupts(void)
11 {
12     int ret = pthread_sigmask(SIG_UNBLOCK, &xAllSignals, NULL);
13     if (ret != 0)
14     {
15         prvFatalError("pthread_sigmask", ret);
16     }
17 }
```

These functions are not invoked directly by application code but form the backbone of the *FreeRTOS* kernel's critical-section API, namely, `vTaskEnterCritical()` and `vTaskExitCritical()`. By blocking (`SIG_BLOCK`) or unblocking (`SIG_UNBLOCK`) the complete signal set (`xAllSignals`) for the pthread that backs each *FreeRTOS* task, they effectively disable or enable "interrupts" in a POSIX environment.

6.4.7 POSIX Port and I/O

A dedicated I/O library is provided for both *FreeRTOS* tasks and the *FreeRTOS* Injector, rather than invoking the native POSIX I/O API directly. This wrapper serves two critical purposes:

1. Task switching in the POSIX port depends on a `SIG_ALRM` handler. Each *FreeRTOS* task unblocks this signal so the scheduler can preempt and perform context switches. Threads outside the POSIX port layer, such as those used by the *FreeRTOS Injector*, must keep `SIG_ALRM` blocked; otherwise, receiving the alarm signal without proper context or associated POSIX condition variables may cause deadlocks and simulator stalls.

2. Some standard C library functions employ internal pthread mutexes. If the *FreeRTOS* scheduler switches tasks while such a mutex is held, deadlocks can occur. Routing all I/O through the custom API prevents unexpected internal locking and ensures safe task-switch points.

As noted by the original POSIX port author:

> "Use of parts of the standard C library requires care, as some functions take pthread mutexes internally. If the FreeRTOS kernel switches tasks while a mutex is held, deadlocks can occur."

As a final remark, this overview focuses on the most pertinent components and omits full low-level details, complete coverage would require extensive code excerpts and commentary beyond this work's scope. The chosen level of detail should suffice to convey the core concepts.

6.5 FreeRTOS Injector System Architecture

This section presents the overall architecture of the *FreeRTOS Injector*. Before diving into the fault-injection system overview, key terminology is defined.

6.5.1 Key Definitions

- **Experiment**: A single execution instance.
- **Golden Run**: An experiment without fault injection, used to establish baseline timing and correct output.
- **Run**: Either a golden run or a fault-injection run.
- **Campaign**: An ordered sequence of experiments, starting with one golden run and followed by multiple fault-injection runs.

- **Fault List**: The set of locations where faults may be injected.
- **Fault**: A specific injection event at a given location. In this work, a fault corresponds to the memory address where a bit-flip (transient or permanent) is introduced.

6.5.2 Experiment Environment

The experimental setup consists of two scenarios:

First Scenario

The first scenario comprises a *FreeRTOS* instance with three benchmark tasks and one software timer:

- **QSRT** Executes one iteration of the Quicksort algorithm on an input data file, then self-deletes.
- **TX & RX** Two tasks sharing a FreeRTOS queue. **TX** enqueues data; **RX** dequeues it. RX runs at priority 2 (highest), while TX and QSRT share priority 1. Both tasks repeat for a configurable number of iterations (see `injector_config.h` in 6.3.1) before deleting themselves.
- **Software Timer** On each expiration, its callback enqueues a value on the same queue used by TX/RX. A static counter, also configurable via `injector_config.h` (see 6.3.1), tracks the number of callbacks; once it reaches its limit, the timer deletes itself.

In addition to the above, the *FreeRTOS* kernel creates several system tasks:

- **IDLE Task** Runs at the lowest priority (usually 0). It reclaims resources of tasks deleted by other contexts. If preemption and idle-yield are enabled in `FreeRTOSConfig.h` (see 4.6), it also calls `vTaskYield()`.
- **Timer Daemon Task** Implements the FreeRTOS software-timer API and runs at the highest kernel-task priority (as configured in `FreeRTOSConfig.h`).

Finally, the portable layer introduces two additional threads:

- **Main Thread** Begins in `main.c`, initializes the port layer, and then invokes `vTaskStartScheduler()` (in `task_main.c`). After scheduler start, it remains in the scheduler loop until receiving a termination signal (typically via `vTaskEndScheduler()`).

- **Interrupt Simulator Thread** Waits for simulated interrupts through `vPortGenerateSimulatedInterrupt()`, dispatches the appropriate handler, and is cleanly shut down by the main scheduler loop upon scheduler termination.

Second Scenario

The second scenario employs five automotive industry benchmarks from the TACLe suite, each ported to the *FreeRTOS Injector*. These tasks better represent real embedded workloads:

- **SHA** – computes a SHA hash
- **FFT** – performs a fast Fourier transform
- **CUBIC** – solves a cubic equation
- **HUFF_DEC** – carries out Huffman decoding
- **ADPCM_ENC** – executes adaptive pulse-code modulation encoding

Priorities are assigned as follows:

- SHA, FFT, and CUBIC: priority 1
- HUFF_DEC: priority 2
- ADPCM_ENC: priority 3

Additionally, as in the first scenario (see Section 6.5.2), the system tasks are present. Each task executes its computation and then self-deletes, allowing the shutdown mechanism to terminate the experiment correctly (see 6.5.3). The same queue created in scenario 1 is also instantiated here to accommodate the simulated interrupt mechanism (see 6.5.3).

6.5.3 Injection Environment

The injection environment builds on a hosted, user-space *FreeRTOS* instance by adding a dedicated POSIX thread that carries out fault injections at runtime. In this model, each *FreeRTOS* instance runs as a separate process under the host OS, with all its *FreeRTOS* "threads" sharing a single address space. By introducing the injector as a peer POSIX thread in that same process, the injector gains direct access to the target memory addresses. An injection run thus consists of the standard benchmark tasks plus this external injector thread, which triggers a fault at a specified time and memory location.

FreeRTOS Source Patches for Fault Injection Support

To integrate the fault-injection framework, several modifications were made to the original *FreeRTOS* kernel. The primary addition is a helper function in `tasks.c` that detects when only the `IDLE` task remains, allowing the clean shut down of the *FreeRTOS* scheduler:

Listing 6.5: FreeRTOS Injector helper function

```
/**
 * @brief  Returns true if the IDLE task is the only ready task.
 *         Used by the FreeRTOS Injector to determine when all
 *         benchmark tasks have completed.
 * @return true if only the IDLE task is ready; false otherwise.
 */
bool bIsIdleTheOnlyTaskInTheSystem(void)
{
    /* uxTopReadyPriority holds the highest priority level with ready
    tasks.
       If it is zero (IDLE priority) and there is exactly one task in
    that list,
       while all delayed and pending lists are empty, and no tasks
   await
       termination, then the IDLE task is the sole remaining task. */
    if ( uxTopReadyPriority == 0
      && listCURRENT_LIST_LENGTH(&pxReadyTasksLists[0]) == 1
      && listCURRENT_LIST_LENGTH(&xDelayedTaskList1)    == 0
      && listCURRENT_LIST_LENGTH(&xDelayedTaskList2)    == 0
      && listCURRENT_LIST_LENGTH(&xPendingReadyList)    == 0
      && listCURRENT_LIST_LENGTH(&xTasksWaitingTermination) == 0 )
    {
        return true;
    }
    return false;
}
```

When all benchmark tasks have deleted themselves after their configured iterations, this function returns `true`, signalling the injector's scheduler loop to perform a graceful shutdown.

FreeRTOS Scheduler Shutdown Mechanism

The *FreeRTOS* scheduler can be terminated via two complementary paths:

1. Idle-Hook Path When `configUSE_IDLE_HOOK` is set to 1 in `FreeRTOSConfig.h`, the hook `vApplicationIdleHook()` is called each time the `IDLE` task executes. Since the hook must avoid any blocking *FreeRTOS* API calls,

it serves as a safe point to check whether `IDLE` is the only remaining ready task. If so, the hook records timing (for a golden run), generates a simulated interrupt (the purpose of which is explained below), and calls `vTaskEndScheduler()` to shut down the scheduler:

Listing 6.6: FreeRTOS IDLE task hook

```
#if (configUSE_IDLE_HOOK == 1)

void vApplicationIdleHook(void)
{
    extern bool bIsIdleTheOnlyTaskInTheSystem(void);

    if (bIsIdleTheOnlyTaskInTheSystem())
    {
        if (bIsGoldenRun)
        {
            vCommonWriteGoldenExecutionTimeFile();
        }

        vPortGenerateSimulatedInterrupt(
            injector_configISR_INTERRUPT_NUMBER
        );

        vInjectorIoPrintDebug(
            "Idle Hook: calling vTaskEndScheduler()\n"
        );
        vTaskEndScheduler();
    }
}

#endif /* configUSE_IDLE_HOOK == 1 */
```

2. Injector-Thread Path Some fault scenarios can prevent the scheduler (and thus the `IDLE` task) from ever running again. To handle such hangs, the injector thread monitors execution time post, injection. If it exceeds a configurable timeout, typically set to 300% of the golden-run duration, the injector thread performs the same shutdown sequence as the `IDLE` hook, ensuring a graceful termination even when the scheduler is blocked.

3. Watchdog-Timer Path In this extreme recovery scenario, the RTOS instance has become unresponsive to all standard shutdown requests. When running in campaign mode (see 6.5.3), the *FreeRTOS Injector* arms a watchdog timer whose timeout interval is configurable via `injector_config.h` (see 6.3.1). By default this interval is set to exceed 300% of the "golden run" timeout that each injector

thread waits, ensuring the watchdog only fires after all normal recovery attempts have failed. Once the configurable interval elapses, a dedicated handler thread is invoked to issue an immediate forced termination, sending **SIGKILL** on POSIX platforms (or the equivalent forced-termination call on Windows), to forcefully terminate the stuck RTOS process.

Logging Facility

To classify outcomes of both golden and fault-injection runs, the *FreeRTOS Injector* leverages the kernel's tracing hooks. These hooks, defined in `FreeRTOSConfig.h`, invoke custom code at critical kernel events:

Listing 6.7: FreeRTOS Injector trace hook macros

```
/* Tracing macros (must remain in FreeRTOSConfig.h) */
#define traceTASK_SWITCHED_IN()                    vTracerLogEvent(0)
#define traceTASK_SWITCHED_OUT()                   vTracerLogEvent(1)
#define traceQUEUE_SEND_FAILED(pxQueue)            vTracerLogEvent(2)
#define traceQUEUE_RECEIVE_FAILED(pxQueue)         vTracerLogEvent(3)
#define traceQUEUE_SEND_FROM_ISR_FAILED(pxQueue)   vTracerLogEvent
    (4)
#define traceQUEUE_RECEIVE_FROM_ISR_FAILED(pxQueue) vTracerLogEvent
    (5)
```

Each macro calls `vTracerLogEvent()`, declared in `tracer.h` and implemented in `tracer.c`. Its logic is as follows:

Listing 6.8: FreeRTOS Injector Logging function

```
void vTracerLogEvent(uint8_t ucEventType)
{
    char     cEventBuf[injector_configTRACER_MAX_COLUMNS];
    char     cTaskName[configMAX_TASK_NAME_LEN];
    unsigned long ulTime = ulGetRunTimeCounterValue();

    /* Stop logging after a simulated\  interrupt    failure. */
    if (bISRExecuted) {
        return;
    }

    /* Get the current task's name */
    vTaskGetTaskName(cTaskName);

    /* Format and flush the log entry */
    switch (ucEventType)
    {
        case 0: /* Task switched in */
            snprintf(cEventBuf, sizeof(cEventBuf),
                     "%-16lu\t[IN]\t%s", ulTime, cTaskName);
```

```
21              break;
22          case 1: /* Task switched out */
23              snprintf(cEventBuf, sizeof(cEventBuf),
24                       "%-16lu\t[OUT]\t%s", ulTime, cTaskName);
25              break;
26          case 2: /* xQueueSend() failed */
27              snprintf(cEventBuf, sizeof(cEventBuf),
28                       "%-16lu\t[QSF]\t%s", ulTime, cTaskName);
29              break;
30          case 3: /* xQueueReceive() failed */
31              snprintf(cEventBuf, sizeof(cEventBuf),
32                       "%-16lu\t[QRF]\t%s", ulTime, cTaskName);
33              break;
34          case 4: /* xQueueSendFromISR() failed */
35              bISRExecuted = true;
36              snprintf(cEventBuf, sizeof(cEventBuf),
37                       "%-16lu\t[QSIF]\t%s", ulTime, cTaskName);
38              break;
39          case 5: /* xQueueReceiveFromISR() failed */
40              bISRExecuted = true;
41              snprintf(cEventBuf, sizeof(cEventBuf),
42                       "%-16lu\t[QRIF]\t%s", ulTime, cTaskName);
43              break;
44          default: /* Unexpected event */
45              snprintf(cEventBuf, sizeof(cEventBuf),
46                       "%-16lu\t[INV]\tNOTASK", ulTime);
47      }
48
49      prvFlushEvent(cEventBuf);
50  }
```

This tracer captures every task switch-in/out and all queue operations (in both thread and ISR contexts), prefixing each entry with the current runtime counter value. Downstream, the classification algorithm uses these timestamps to detect unexpected scheduling delays. Once the simulated interrupt handler sets the flag `bISRExecuted`, all further tracing is suppressed.

Simulated Interrupt Handler

The injector leverages the POSIX (and *Windows*-equivalent) simulated-ISR facility to trigger a controlled "interrupt" during an experiment run. Its sole purpose is to attempt a queue receive from ISR context, thereby revealing scheduler disruptions.

Listing 6.9: FreeRTOS Injector Simulated IRQ Handler

```
1 #ifdef _POSIX_SOURCE_
2 void vCommonInterruptHandler(uint32_t ulInterruptNumber)
3 #elif defined(_WINDOWS_SOURCE_)
```

```
uint32_t vCommonInterruptHandler(uint32_t ulInterruptNumber)
#endif
{
    /* Shared queue defined in applications/task_main.c */
    extern QueueHandle_t xQueue;

    /* Attempt to receive without blocking */
    uint32_t xQueueReceivedValue = 0;
    xQueueReceiveFromISR(xQueue,
                         (void *)&xQueueReceivedValue,
                         NULL);

    /*
     * If we unexpectedly succeed, the scheduler has failed
     * to dispatch the RX task in time.  Record this anomaly.
     */
    if (xQueueReceivedValue != 0) {
        vInjectorIoPrintFromISRDebug(
          "ISR received a queue value: %u\n"
          "Unexpected! Scheduler was disrupted by the injection.\n",
          xQueueReceivedValue
        );
    }

    /* Log that the interrupt handler executed */
    vInjectorIoPrintFromISRDebug(
      "[%lu]\tInterrupt %lu handler invoked.\n",
      ulGetRunTimeCounterValue(),
      ulInterruptNumber
    );

#ifdef _WINDOWS_SOURCE_
    return pdFALSE;
#endif
}
```

Operational Context All benchmark tasks of the first scenario (see 6.5.2) (TX, RX) and the software-timer callback share a single *FreeRTOS* queue. They are configured with finite iteration counts so that

$$\text{RX iterations} = \text{TX iterations} + \text{timer callback iterations}.$$

Hence, in a *correct* (golden) run, or any run where faults do not break the scheduler path, the ISR's nonblocking receive will fail, triggering one of the trace-hook macros (see Listing 6.7). The presence of that specific trace entry is used by the classification algorithm which will be described below.

The same mechanism is applied in the second scenario (see 6.5.2). Here, an empty queue, identical to the one used previously, is instantiated solely to signal the tracing mechanism to stop and to verify that the simulated interrupt (see Section 6.5.3) executes correctly.

Execution Outcome Categories in Fault Injection

Fault injection experiments typically classify each run into one of several standard outcome categories, as established in the dependability and resilience literature:

OK (Correct) The system completes its task within the expected time and produces correct results.

SDC (Silent Data Corruption) The system finishes on time but yields an incorrect result without raising any error, an undetected data corruption.

DELAY (Timing Violation) The result is functionally correct, but the execution exceeds the predefined timing threshold, indicating a performance or real-time violation.

HANG (Hang/Deadlock) The system never completes (no output or completion signal), typically due to deadlock, livelock, or unhandled fault.

CRASH (Immediate Failure) The system terminates abruptly (e.g., via exception, abort, or unhandled signal), preventing normal shutdown and output generation.

These categories help quantify the resilience of real-time and safety-critical systems by distinguishing between benign timing slips, silent corruptions, and severe failures that require recovery intervention.

Experiment Classification

After the scheduler has been cleanly shut down (see Section 6.5.3), the classifier inspects the in-memory trace and execution results to assign one of several outcome codes:

Listing 6.10: FreeRTOS Injector Classifier

```
1 int prvCheckExecutionResult(void *pvData)
2 {
3 #ifdef DEBUG
4
5     vTracerPrint();
6
7 #endif /** DEBUG */
```

```
8
9       if (bTracerTraceIsOk())
10      {
11          unsigned long ulExecTime = ulTracerGetExecutionTime();
12          vInjectorIoPrintExtThreadDebug("Execution time: %lu\n",
    ulExecTime);
13          if (bTracerExecutionResultIsOk(pvData))
14          {
15              if (ulExecTime == 0)
16              {
17                  return error_codesBAD_EXECUTION_TIME;
18              }
19              else if (ulExecTime < (injector_configDELAY_THRESHOLD *
    ulGoldenExecutionTime))
20              /** Silent execution, correct output. */
21              {
22                  return error_codesEXECUTION_OUTCOME_OK;
23              }
24              else
25              /** Delayed execution, correct output. */
26              {
27                  return error_codesEXECUTION_OUTCOME_DELAY;
28              }
29          }
30          else
31          /** Execution result is not correct. */
32          {
33              if (ulExecTime < (injector_configDELAY_THRESHOLD *
    ulGoldenExecutionTime))
34              /** Error execution, incorrect output. */
35              {
36                  return error_codesEXECUTION_OUTCOME_SDC;
37              }
38              else
39              /** SDC with Delay. */
40              {
41                  return error_codesEXECUTION_OUTCOME_SDC_DELAY;
42              }
43          }
44      }
45      else
46      /** Incorrect Trace output, ISR didn't work. */
47      {
48          return error_codesEXECUTION_OUTCOME_CRASH;
49      }
50  }
```

FreeRTOS Injector Classification Logic The outcome of each run is determined as follows:

1. If the trace is invalid (for example, logging ended before the simulated ISR), classify the run as a *crash*.

2. Otherwise, measure the total execution time and compare it against the configurable threshold.

3. Verify functional correctness by comparing:

   - In the first scenario (see 6.5.2), the output of the *QSRT* task against the golden reference using `bTracerExecutionResultIsOk()`.
   - In the second scenario (see Section 6.5.2), each TACLe benchmark's output is validated in `bTracerExecutionResultIsOk()` as well.

4. Assign one of five primary outcome codes:

   - `EXECUTION_OUTCOME_OK`: correct output, on-time
   - `EXECUTION_OUTCOME_DELAY`: correct output, but delayed
   - `EXECUTION_OUTCOME_SDC`: incorrect output (SDC), on-time
   - `EXECUTION_OUTCOME_SDC_DELAY`: incorrect output, and delayed
   - `EXECUTION_OUTCOME_CRASH`: crash

A sixth outcome code, `EXECUTION_OUTCOME_HANG`, is returned when the watchdog timer intervenes (see Section 6.5.3). In this scenario, the OS-dependent logic in os_deps.c (see 6.3.1) recognizes that the *FreeRTOS* instance was terminated by the specific *UNIX* signal sent by the watchdog handler, rather than by a normal exit, and reports the hang back to the *Orchestrator* process.

Target Extraction

Injection targets inside the *FreeRTOS* kernel are discovered at startup by module-specific "gatherer" functions. Each gatherer is registered in a global function table. During initialization, the helper macros build a global, singly-linked list of `struct target` entries. Finally, a single call to `vTargetCallGatherers()` iterates over that table, invoking each gatherer in turn, and thus populates the complete fault-injection target list.

Target Data Structure The main data structure used for the list of injection targets is shown below:

Listing 6.11: `struct target` description

```
/** Main data structure for gathering data about injection targets.
    Targets are gathered from the corresponding functions implemented
    in the relevant files (task.c, queue.c etc...) and are linked
    together in a linked list. Each target can have a child (for
    example a structure's member is a child of the structure itself).
    The first member of a struct is it's only children because other
    children are linked together. Every single child has instead a
    link back to its parent. */
struct target
{
        size_t ulId;
        char pcName[injector_configTARGET_NAME_LENGTH];
        void *pvAddress;
        bool bNeedsParentDereferenceAtRuntime;
        size_t ulSize;
        /** This takes into account the size of the memory areas
    pointed to by pointer targets. */
        size_t ulSizeOfPointedData;
        size_t ulNMembers;
        uint8_t ucType;
        struct target *pxParent;
        struct target *pxChild;
        struct target *pxNext;
};
```

Helper Macros A suite of macros wraps calls to the internal constructor `pxTargetCreateInternal()`, managing parent/child links, compile-time vs. run-time addresses, and array lengths:

Listing 6.12: Target-creation helper macros

```
/** These helper macros can be used to construct the injection target
     list. */

/** Creates a new target. */
#define pxTargetCreate(xTarget, ucType, ulSize, ulPointedSize,
    ulNMemb) \
        pxTargetCreateInternal(targetNAME_OF(xTarget), (void *)&(
    xTarget), ucType, ulSize, ulPointedSize, NULL, NULL, NULL, ulNMemb
    , 0, 0);

/** Creates a new child target. */
#define pxTargetCreateChild(xTarget, ucType, ulSize, ulPointedSize,
    ulNMemb, pxParent) \
```

```
9           pxTargetCreateInternal(targetNAME_OF(xTarget), (void *)&(
    xTarget), ucType, ulSize, ulPointedSize, pxParent, NULL, NULL,
    ulNMemb, 0, 1);
10
11  /** Creates a new child target with an unknown compile time address.
    */
12  #define pxTargetCreateChildNoAddress(xTarget, ucType, ulSize,
    ulPointedSize, ulNMemb, pxParent) \
13          pxTargetCreateInternal(targetNAME_OF(xTarget), NULL, ucType,
    ulSize, ulPointedSize, pxParent, NULL, NULL, ulNMemb, 0, 1);
14
15  /** Appends a new target to the previous one. */
16  #define pxTargetAppend(pxPrevious, xTarget, ucType, ulSize,
    ulPointedSize, ulNMemb) \
17          pxPrevious = pxTargetCreateInternal(targetNAME_OF(xTarget), (
    void *)&(xTarget), ucType, ulSize, ulPointedSize, NULL, NULL,
    pxPrevious, ulNMemb, 1, 0);
18
19  /** Appends a new child target to the previous one and links it to
    the father. */
20  #define pxTargetAppendChild(pxPrevious, xTarget, ucType, ulSize,
    ulPointedSize, ulNMemb, pxParent) \
21          pxPrevious = pxTargetCreateInternal(targetNAME_OF(xTarget), (
    void *)&(xTarget), ucType, ulSize, ulPointedSize, pxParent, NULL,
    pxPrevious, ulNMemb, 1, 0);
22
23  /** Appends a new child target to the prevous one and links it to the
     father. This version is for targets with unknown compile time
    address. */
24  #define pxTargetAppendChildNoAddress(pxPrevious, xTarget, ucType,
    ulSize, ulPointedSize, ulNMemb, pxParent) \
25          pxPrevious = pxTargetCreateInternal(targetNAME_OF(xTarget),
    NULL, ucType, ulSize, ulPointedSize, pxParent, NULL, pxPrevious,
    ulNMemb, 1, 0);
```

Module-Specific Gatherer Example Below is the timer module's gatherer (`timers.c`):

Listing 6.13: Timer-target gatherer (`pxTimerGatherTargets`)

```
1  struct target *pxTimerGatherTargets(struct target *pxPrevious) {
2
3      /** The list of injection targets is global to the whole FreeRTOS
        Injector and the order of calls to the functions implementing
       target gathering is unknown. This leads to 2 conditions: this is
       the first target gathering function called and its first member is
        the first element of the list (does not need to be appended) or
       the function needs to append its first target to an already
       existing list. */
```

```
4       if (pxPrevious)
5       {
6           // static List_t xActiveTimerList1;
7           pxTargetAppend(pxPrevious, xActiveTimerList1, targetTYPE_LIST
        , sizeof(xActiveTimerList1), 0, 1);
8       }
9       else
10      {
11          // static List_t xActiveTimerList1;
12          pxPrevious = pxTargetCreate(xActiveTimerList1,
        targetTYPE_LIST, sizeof(xActiveTimerList1), 0, 1);
13      }
14      struct target *const pxFirstTargetInList = pxPrevious;
15
16      // static List_t xActiveTimerList2;
17      pxTargetAppend(pxPrevious, xActiveTimerList2, targetTYPE_LIST,
        sizeof(xActiveTimerList2), 0, 1);
18
19      // static List_t * pxCurrentTimerList;
20      pxTargetAppend(pxPrevious, pxCurrentTimerList, targetTYPE_POINTER
         | targetTYPE_LIST, sizeof(pxCurrentTimerList), sizeof(*
        pxCurrentTimerList), 1);
21
22      // static List_t * pxOverflowTimerList;
23      pxTargetAppend(pxPrevious, pxOverflowTimerList,
        targetTYPE_POINTER | targetTYPE_LIST, sizeof(pxOverflowTimerList),
         sizeof(*pxOverflowTimerList), 1);
24
25      // static QueueHandle_t xTimerQueue;
26      // initialized in prvCheckForValidListAndQueue (timers.c)
27      pxTargetAppend(pxPrevious, xTimerQueue, targetTYPE_VARIABLE,
        sizeof(xTimerQueue), 0, 1);
28
29      // static TaskHandle_t xTimerTaskHandle;
30      // initialized in xTimerCreateTimerTask (timers.c)
31      pxTargetAppend(pxPrevious, xTimerTaskHandle, targetTYPE_VARIABLE,
         sizeof(xTimerTaskHandle), 0, 1);
32
33      return pxFirstTargetInList;
34  }
```

Each kernel module implements a similar gatherer and registers it in an initialization table. At runtime, a single invocation of `vTargetCallGatherers()` calls all registered gatherers in turn, building the complete target list for fault injection.

Golden Run Overview

A *golden run* is an execution in which no fault injection takes place. Its primary purpose is to establish a baseline performance benchmark against which all subsequent fault-injection runs can be compared. During the golden execution:

1. The *FreeRTOS* scheduler is started.
2. The benchmark tasks are allowed to run uninterrupted until they complete.
3. The shutdown mechanism described in Section 6.5.3 is invoked to stop the RTOS scheduler.
4. Two output files are generated:
    - `golden.txt` records the execution time (in nanoseconds) from scheduler start to scheduler shutdown.
    - `golden_data.txt` contains the results produced by the QSRT benchmark task.

Injection Run Overview

This run follows the same steps as the golden run, except that after a configurable delay the injector thread flips a single bit, either transiently or permanently, at a specified memory address. After the fault is injected, all tasks are left free to run to completion. Once every task has finished, the normal shutdown mechanism described in Section 6.5.3 is invoked to stop the RTOS scheduler gracefully.

- If the scheduler shuts down normally, a classification algorithm analyzes the outcome and the RTOS instance exits with one of the predefined error codes from `error_codes.h`. Several of these codes were introduced in Section 6.5.3.
- If the scheduler crashes, producing an exit code outside the range defined in `error_codes.h`, the run is classified as a crash.

Note: The "single-run" mode is intended for internal use by campaign mode. It is not designed to be invoked as a standalone test.

Campaign Mode Overview

In campaign mode, the *FreeRTOS Injector* reads a user-supplied CSV file detailing one or more injection campaigns. Each line in the CSV must follow the format:

```
Target,Execs,Time,Variance,Distribution,Fault
```

Here:

- **Target**: injection target
- **Execs**: how many times to run the injection on this target
- **Time**: nominal time in nanoseconds when to perform the injection (from the RTOS scheduler start)
- **Variance**: timing variance in nanoseconds around the nominal time
- **Distribution**: one of uniform (u), Gaussian (g), triangular (t), or fixed (f)
- **Fault**: Fault type, transient (t) or permanent (p) bit-flip

An *Orchestrator* process carries out the campaign:

1. **Parse CSV:** Load all campaign entries into an in-memory data structure.
2. **Parallelize:** Determine the worker count using the `-j` option (or default to the number of logical CPU threads).
3. **Schedule Injections:** For each CSV entry and for each of its specified number of executions:
   - Sample an actual injection time from the chosen distribution, applying the nominal time and variance.

     Note: The exact byte and bit location within the target region are chosen uniformly at random for each injection trial.
   - Compute the injection byte and bit based on the specified target type.
   - Fork a worker process that executes the *FreeRTOS Injector* `-run` command with appropriate parameters (see 6.6).
   - Attach a per-experiment watchdog timer to force terminate any hung RTOS instance.
4. **Collect Results:** As each worker exits (normally or by watchdog), record its exit code in an internal statistics table.
5. **Report:** After all injections complete, output the aggregated statistics to stdout (unless suppressed) or to a file if the `-w` option is specified.

   Note: An effectively infinite initial population of targets is assumed. For each specified target, the injection campaign selects uniformly at random one byte and one bit within its memory region. All selections are independent, so the same byte and bit location may be chosen multiple times (at different instants). If injection times are also drawn from a probability distribution, repeated injections into a single location remain independent events. Consequently, any downstream statistical analysis must treat these observations as samples with replacement.

Final Remarks

The system architecture overview and core concepts were covered thoroughly, though a deeper dive would require more listings and space than this presentation allows. Nonetheless, the fundamental ideas have been clearly explained.

6.6 FreeRTOS Injector Commands

Following the detailed description of the *FreeRTOS Injector* architecture and fault-injection mechanisms, this section provides a concise overview of the supported commands and their usage for conducting injection campaigns.

- –**golden** Execute a single golden run:

  ```
  freertos --golden
  ```

  This generates two output files (see Section 6.5.3) that serve as the reference benchmark for subsequent injection campaigns.

- –**run** Perform a single fault-injection run. Usage:

  ```
  freertos --run Target Time Byte Bit Type
  ```

  where

  Target: The name of the injection target.

  Time Injection time in nanoseconds.

  Byte Byte offset within the specified target.

  Bit Bit position (0–7) within the specified byte.

  Type Fault type: `t` for a transient bit-flip, `p` for a permanent bit-flip.

- –**campaign** Execute or prepare a full injection campaign using a CSV file. Usage:

  ```
  freertos --campaign file.csv [options]
  ```

  Options:

  -d out.csv Dry run mode: read `file.csv`, compute injection times, bytes, and bits, and write these parameters to `out.csv` without performing any injections.

Note: Unlike a standard campaign, **`out.csv`** *reports the final per-trial byte and bit locations.*

For example, given an input `file.csv` with this single entry:

```
exampleTarget,5,1000,0,f,t
```

the generated `out.csv` might look like:

```
#exampleTarget,5,1000,t
exampleTarget,10,5
exampleTarget,7,7
exampleTarget,1,5
exampleTarget,4,3
exampleTarget,1,0
```

The dry-run command generates a new CSV file in which each row of the original becomes a header. Directly after each header, it inserts one entry per specified injection, explicitly listing the target byte and bit for that run. The execution time from the original file, sampled according to its probability distribution, is also fixed and recorded in the header of the new file. Because this time has already been determined, the new file omits the original probability and variance-related columns.

Note: In the current implementation, the dry-run command does not ensure uniqueness when generating random byte and bit locations from the original CSV file. As a result, multiple injection targets may share identical locations. This limitation is addressed during experimental validation by applying a statistical formula designed for sampling with replacement (i.e., an infinite initial population), which will be described in detail in the corresponding chapter.

`-ud out.csv` Use `out.csv` (produced by the `-d` dry-run option) to drive the injection campaign directly, bypassing the runtime computation of byte-bit locations and injection times. A typical invocation is:

```
freertos --campaign out.csv -ud [other-options]
```

Here, `-ud` signals that `out.csv` was generated earlier by a dry-run command.

-y Automatically answer "yes" to all interactive prompts.

-p Display progress updates.

-s Suppress summary statistics on standard output.

-w results.csv Write aggregated campaign statistics in CSV format to results.csv.

-j n Run up to `n` parallel worker processes (default: number of logical CPU threads available on the system).

- **–list**

  ```
  freertos --list
  ```

  This command shows the available injection targets along additional information, useful to know which injection targets are supported by the *FreeRTOS Injector*. The list of injection target has been described in 6.7.

6.7 Fault List

The following sections briefly describe the supported injection target types and list the specific targets available in the *FreeRTOS Injector*.

6.7.1 Target Types

The *FreeRTOS Injector* supports several categories of injection targets. Before listing all available targets, the following recap defines each target type.

Variable Targets

Scalar kernel variables (e.g., `uxCurrentNumberOfTasks`, `xTickCount`).

Array Targets

Contiguous collections of elements, indexed like C arrays. Injection can occur at any index, for example:

```
pxCurrentTCB->pcTaskName[0]
pxReadyTasksLists[2]
pxReadyTasksLists[-1]
```

Here, an index of `-1` directs the injector to pick a random valid position at runtime.

Struct Targets

Fields within C `structs`. Any member may be selected for injection, for example:

```
pxCurrentTCB->pxTopOfStack
pxCurrentTCB->uxBasePriority
```

List Targets

FreeRTOS `List` objects (doubly linked lists). Injection can target a specific list node by index, for example:

```
xDelayedTaskList1[2]
*pxCurrentTimerList[-1]
```

As with arrays, an index of `-1` instructs the injector to choose a random valid position at runtime. If the list is empty, this may lead to no injection or an invalid access that the injector will detect and skip.
Note that when you have a pointer to a `List`, you must dereference it before applying an index. For instance,

```
pxCurrentTimerList[-1]
```

is invalid because `pxCurrentTimerList` is a pointer; you must write

```
*pxCurrentTimerList[-1]
```

to access a node. Pure `List` variables (not pointers) can be indexed directly.

Pointer Targets

These targets are variables that hold memory addresses. Fault injections can either corrupt the pointer's value or the data it references. Examples:
Inject into the pointer variable itself:

```
pxCurrentTCB
xTimerQueue
```

Inject into the memory pointed to (using a leading asterisk):

```
*pxCurrentTCB
```

The leading-asterisk notation forces the injector to target the pointee rather than the pointer. This syntax also applies when a pointer refers to a struct or a list, e.g., `*pxDelayedTaskList` injects into the memory area of the `List_t` structure.

Mixed Targets

Composite types combining two or more of the above categories. For instance, `pxReadyTasksLists` is an array of lists and supports nested indexing:

```
pxReadyTasksLists[0][0]
```

This target specification directs the injector to:

1. Select the ready-tasks list at priority level 0 (the *IDLE* task priority usually): `pxReadyTasksLists[0]`
2. Within that list, choose the first element (list node index 0): `[0]`

The injector will corrupt a field (for example, `pxNext`, `pxPrevious`, or `pvOwner`) of the first node in the priority 0 ready-tasks list.

6.7.2 FreeRTOS Injection Target Groups

The various targets available have been grouped into four major groups:

- **Global Variables:** These are global variables used within the RTOS that influence scheduling decisions and help manage the RTOS system. The targets under this group are shown in table 6.1.
- **Lists:** These are used for queuing tasks into scheduling lists, event lists, blocked lists, delayed lists or lists of tasks blocked on mutexes or semaphores. The targets under this group are shown in table 6.2.
- **Current TCB:** These are targets pertaining to the TCB of the currently running task under the RTOS. The fields of a task's TCB influence its scheduling and managing by the RTOS. The targets under this group are shown in table 6.4 and 6.5.
- **Pointers:** These are pointer targets. Targets under this group are shown in table 6.3.

Table 6.1: FreeRTOS Global Variable Targets

Target	**Type**	**Description**
uxDeletedTasksWaitingCleanup	VARIABLE	count of TCB pending cleanup by the IDLE task
uxCurrentNumberOfTasks	VARIABLE	Total number of currently active tasks.
xTickCount	VARIABLE	System tick counter since scheduler start.
uxTopReadyPriority	VARIABLE	Highest priority level with at least one ready task.
xSchedulerRunning	VARIABLE	Boolean flag: has the scheduler started?
xPendedTicks	VARIABLE	Ticks accumulated while scheduler was suspended.
xYieldPending	VARIABLE	Flag indicating a pending context switch.
xNumOfOverflows	VARIABLE	Number of times xTickCount has wrapped around.
uxTaskNumber	VARIABLE	Monotonic counter for task creation (unique ID).
xNextTaskUnblockTime	VARIABLE	Tick count when the next delayed task is due to unblock.
xTimerQueue	VARIABLE	Handle of the internal timer command queue.
xTimerTaskHandle	VARIABLE	Task handle of the Timer Daemon Task.

Table 6.2: FreeRTOS List Targets

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

Table 6.3: FreeRTOS Pointer Targets

Target	**Type**	**Description**
pxDelayedTaskList	POINTER	Pointer to the current delayed task list. The pointer switches on xTickCount overflow between xDelayedTaskList1 and xDelayed-TaskList2
pxOverflowDelayedTaskList	POINTER	Pointer to the current overflowed delayed task list. The pointer switches on xTickCount overflow between xDelayedTaskList1 and xDelayedTaskList2
xIdleTaskHandle	POINTER	Pointer to the Idle task's TCB (its task handle).
pxCurrentTCB	POINTER	Pointer to the TCB of the currently running task.
pxCurrentTCB.pxTopOfStack	POINTER	Pointer to the top of the stack of the currently running task.
pxCurrentTCB.pxTaskTag	POINTER	Pointer to the application task tag of the currently running task.
pxCurrentTimerList	POINTER	Pointer to the current timer list. Switches on xTickCount overflow between xActiveTimerList1 and xActiveTimerList2.
pxOverflowTimerList	POINTER	Pointer to the current overflowed timer list. Switches on xTickCount overflow between xActiveTimerList1 and xActive-TimerList2.

Table 6.4: FreeRTOS current TCB Targets

Target	**Type**	**Description**
pxCurrentTCB.pxTopOfStack	POINTER	Pointer to the top of the stack of the currently executing task.
pxCurrentTCB.xStateListItem	STRUCT	List node linking TCB into scheduler lists.
pxCurrentTCB.xEventListItem	STRUCT	List node for events (queues, semaphores).
pxCurrentTCB.uxPriority	VARIABLE	Priority of the current task.
pxCurrentTCB.pxStack	POINTER	Pointer to the base of the stack of the current task.
pxCurrentTCB.pcTaskName	ARRAY	Array of characters representing the name of the current task.
pxCurrentTCB.uxTCBNumber	VARIABLE	Unique identifier of the current task.
pxCurrentTCB.uxTaskNumber	VARIABLE	Unique identifier of the current task.

Table 6.5: FreeRTOS current TCB Targets (continued)

Target	**Type**	**Description**
pxCurrentTCB.uxBasePriority	VARIABLE	Current task's base priority used by the mutex priority inheritance mechanism.
pxCurrentTCB.uxMutexesHeld	VARIABLE	Current task's number of mutexes held, also used by the mutex priority inheritance mechanism.
pxCurrentTCB.pxTaskTag	POINTER	Current task's application specific tag. This is a mechanism to permit tasks to call specific functions at specific times.
pxCurrentTCB.ulRunTimeCounter	VARIABLE	Accumulates the task's time spent in the running state.
pxCurrentTCB.ulNotifiedValue	VARIABLE	Internal variable used by the task notification mechanism.
pxCurrentTCB.ucNotifyState	VARIABLE	Internal variable used by the task notification mechanism.
pxCurrentTCB.ucDelayAborted	VARIABLE	Flags whether a blocking delay was prematurely aborted for the current task.

6.8 Conclusion

This chapter has provided a detailed tour of the *FreeRTOS Injector*. After a brief survey of related work and a recap of core implementation principles, the project's directory structure was laid out and the roles of its key modules highlighted. The design of the POSIX port layer followed, establishing the groundwork for a deep dive into the injector's architecture. Next, the injector's core fault-injection mechanisms and command-line interface were described, and the complete set of supported targets was organized by type.
In the next chapter, baseline results from initial fault-injection campaigns (with no hardening enabled) will be presented. These observations will motivate the selective hardening strategy proposed in this study, which will then be described in detail. Finally, a second round of injections, this time with hardening active, will quantify the protection's effectiveness. The concluding chapter will synthesize these results and explore their broader implications.

Chapter 7

Experimental Results

7.1 Introduction

The previous chapter introduced the *FreeRTOS Injector*, described its project structure, and detailed its architecture and implementation. The threat landscape confronting embedded devices running a real-time operating system, specifically FreeRTOS, was also reviewed.
This chapter begins the experimental evaluation. First, an initial set of baseline experiments conducted with no hardening enabled is presented. These results expose *FreeRTOS*'s primary vulnerabilities and motivate the need for a selective hardening strategy. Next, the hardening approach is described in detail , outlining its rationale and implementation. Finally, a second set of experiments with hardening active is reported and compared against the unprotected baseline to quantify the effectiveness of the defenses.

7.2 Experiment Setup

7.2.1 First Scenario

The first scenario test harness (see Section 6.5.2) executes four benchmark activities under *FreeRTOS*:

- **QSRT task:** in-place sort of a data array
- **TX task:** periodic producer task
- **RX task:** periodic consumer task
- **Software timer callback:** invoked by the Timer Daemon; also acts as a producer

Fault injections are triggered at a fixed time, 10000 nanoseconds after scheduler start, to ensure the benchmarks are still active. Three batches of experiments differ only in task workload:

1. *Batch 1*
    - QSRT sorts 1000 elements
    - TX loops 5 times
    - RX loops 10 times
    - Timer callback fires 5 times
2. *Batch 2*
    - QSRT sorts 5000 elements
    - TX loops 10 times
    - RX loops 20 times
    - Timer callback fires 10 times
3. *Batch 3*
    - QSRT sorts 10000 elements
    - TX loops 20 times
    - RX loops 40 times
    - Timer callback fires 20 times

The 10000 nanoseconds injection point was chosen experimentally: on the host-based POSIX port, the scheduler and benchmarks execute rapidly, so later injections would occur after task completion but before scheduler shutdown.

7.2.2 Second Scenario

As detailed in 6.5.2, this scenario runs five TACLe benchmarks exactly once, to completion (no repeated iterations). The injection time remains 10000 ns after the *FreeRTOS* scheduler start, as in the first scenario. Because the POSIX simulated port on a high performance workstation executes these tasks extremely rapidly, finishing in a single pass, the early injection deadline is critical to ensure the faults are injected before task completion.

7.2.3 Number of Experiments

The *FreeRTOS Injector* samples with replacement, each chosen injection location remains eligible for reselection, so the space of possible faults is effectively infinite. Leveugle et al. [21] present a rigorous framework for computing error bounds and confidence intervals in fault-injection experiments, but their methods assume a finite, non-replenished set of fault sites and thus do not apply directly to a with-replacement scheme.
The required number of injections per location and the resulting confidence interval are given by two key formulas, both derived under the assumption of an effectively infinite population (sampling with replacement):

- **Sample size** n For margin of error e at confidence level $1 - \alpha$, let

  $$z = Z_{1-\alpha/2}$$

  be the critical value of the standard normal distribution. The required number of injections is

  $$n \;=\; \frac{z^2\, p\,(1-p)}{e^2}\,.$$

- **Confidence Interval (CI)** After observing a sample proportion p over n injections, the true failure probability lies within

  $$\mathrm{CI} \;=\; p \;\pm\; z\sqrt{\frac{p\,(1-p)}{n}}\,.$$

The Chosen parameters for the experimental campaigns in this work are:

- Confidence level $1 - \alpha = 0.99$, hence $z \approx 2.576$.
- Margin of error $e = 0.05$.
- Proportion estimate $p = 0.5$, a conservative assumption that each bit in a target injection location is equally likely to reveal a manifest error.

Substitution into the first formula yields

$$n \;=\; \frac{(2.576)^2 \times 0.5 \times (1-0.5)}{(0.05)^2} \;\approx\; 666.$$

Performing 666 injections per location ensures a 99% confidence level that the observed outcome distribution deviates by no more than ±5% from the true probabilities.

7.3 Experimental Results Before Hardening

In the following the results of injection campaigns before hardening are shown:

7.3.1 First Scenario

The first scenario (see subsection 7.3.1) comprises three primary tasks, **QSRT**, **TX**, and **RX**, together with a timer callback function. This callback is driven by a software timer, which is scheduled and executed by the *FreeRTOS* **Timer Daemon Task**. Within the scenario, three workloads of increasing intensity have been defined. Each workload differs solely in the number of iterations executed by the three tasks and the timer callback, as well as in the size of the dataset that the **QSRT** task must sort. Fault injection outcomes are organized into two categories, transient and permanent faults, and reported separately for each workload.

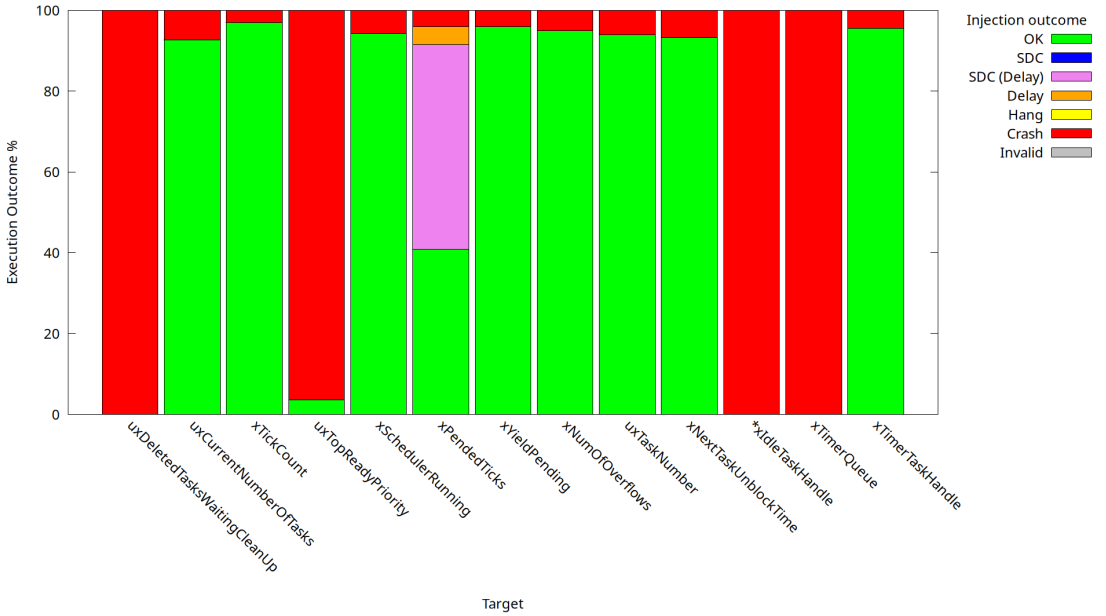

Figure 7.1: Injections on *FreeRTOS* **variables NO ECC** (**Transient Faults**) *QSRT* items **1000** *TX* iterations **5** *Timer* iterations **5** *RX* iterations **10**

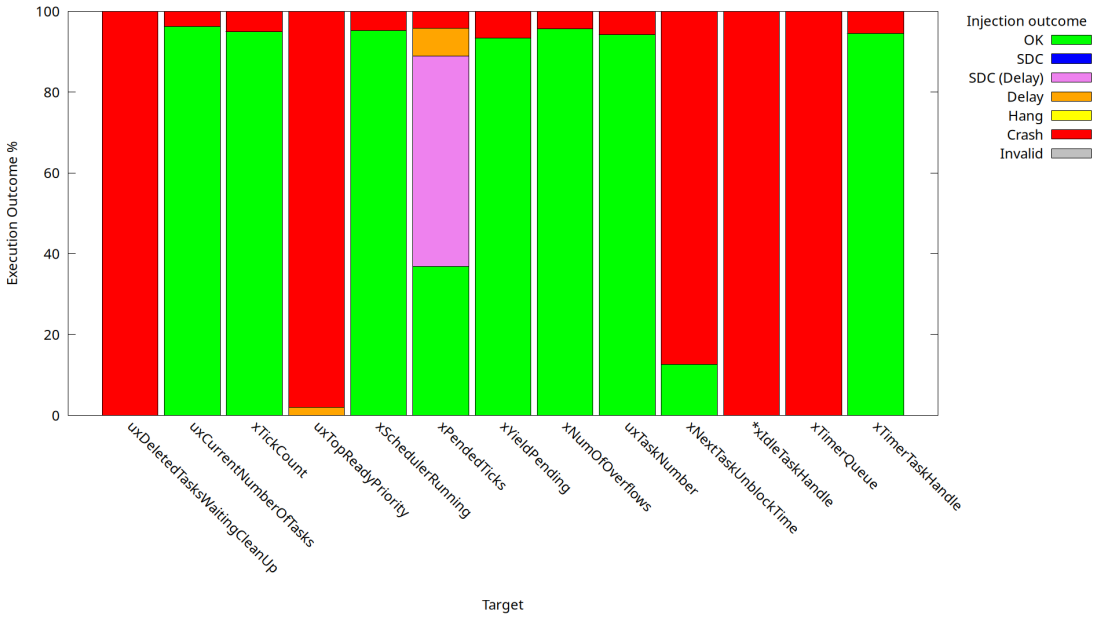

Figure 7.2: Injections on *FreeRTOS* **variables NO ECC** (**Permanent Faults**) *QSRT* items **1000** *TX* iterations **5** *Timer* iterations **5** *RX* iterations **10**

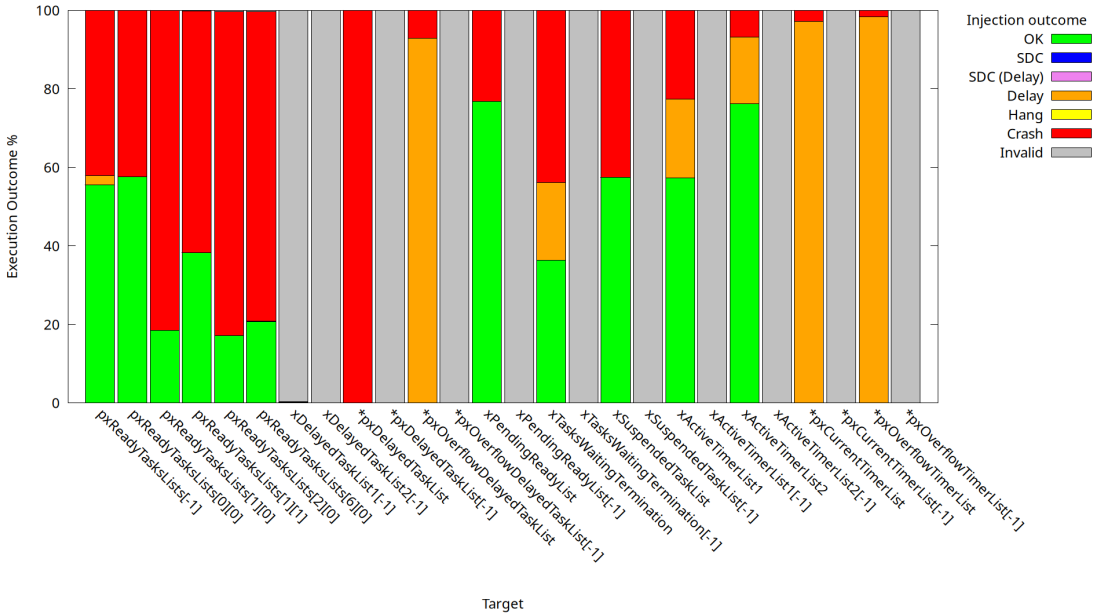

Figure 7.3: Injections on *FreeRTOS* **lists NO ECC** (**Transient Faults**) *QSRT* items **1000** *TX* iterations **5** *Timer* iterations **5** *RX* iterations **10**

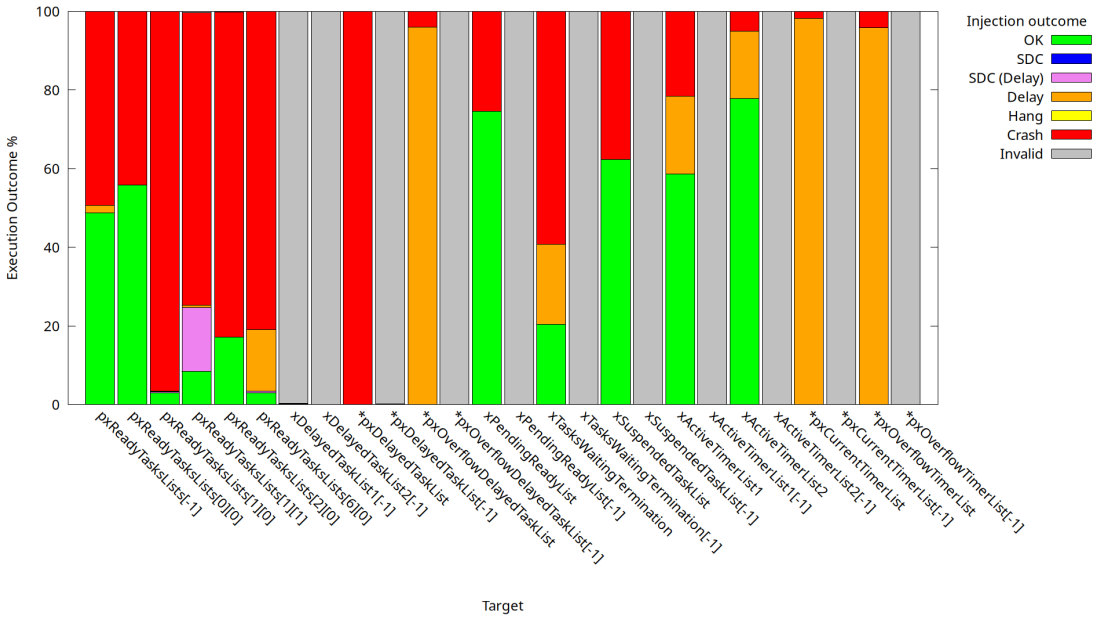

Figure 7.4: Injections on *FreeRTOS* **lists NO ECC** (**Permanent Faults**) *QSRT* items **1000** *TX* iterations **5** *Timer* iterations **5** *RX* iterations **10**

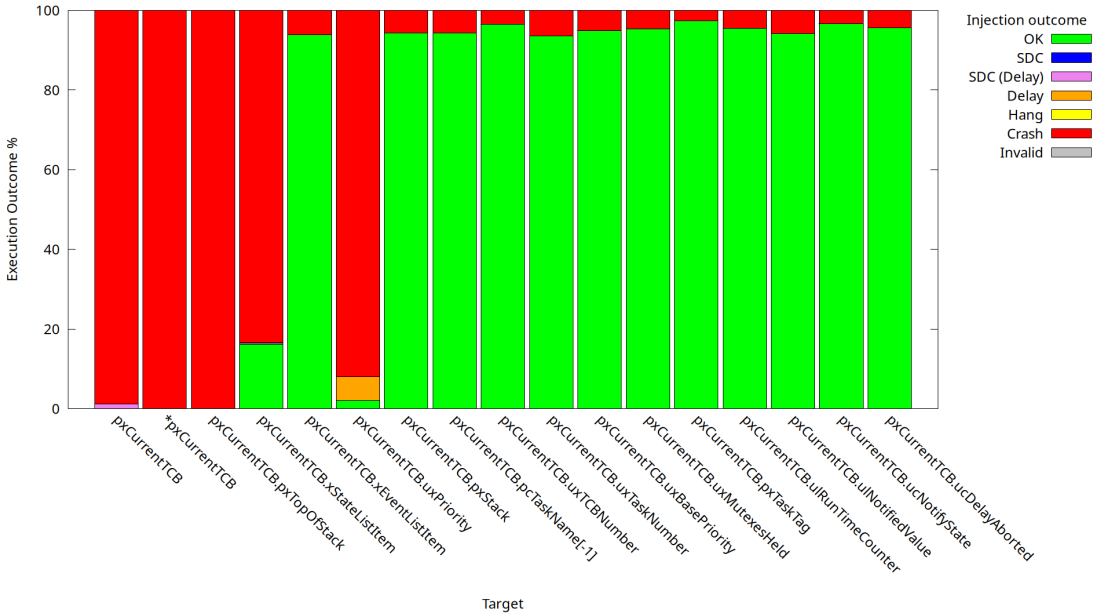

Figure 7.5: Injections on *FreeRTOS* **current TCB NO ECC** (**Transient Faults**) *QSRT* items **1000** *TX* iterations **5** *Timer* iterations **5** *RX* iterations **10**

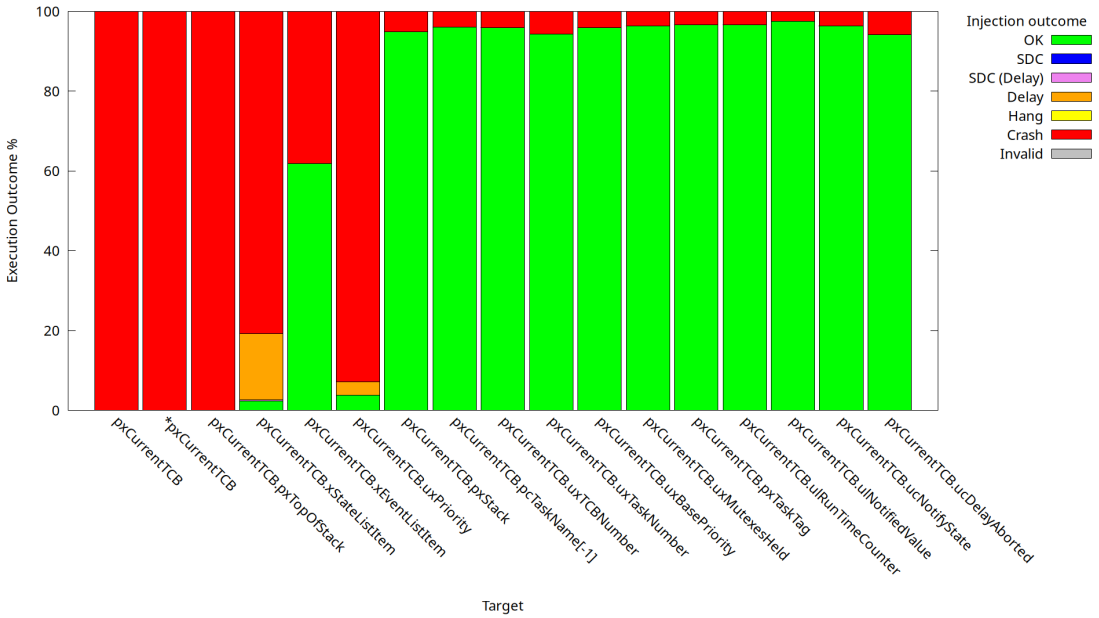

Figure 7.6: Injections on *FreeRTOS* **current TCB NO ECC** (**Permanent Faults**) *QSRT* items **1000** *TX* iterations **5** *Timer* iterations **5** *RX* iterations **10**

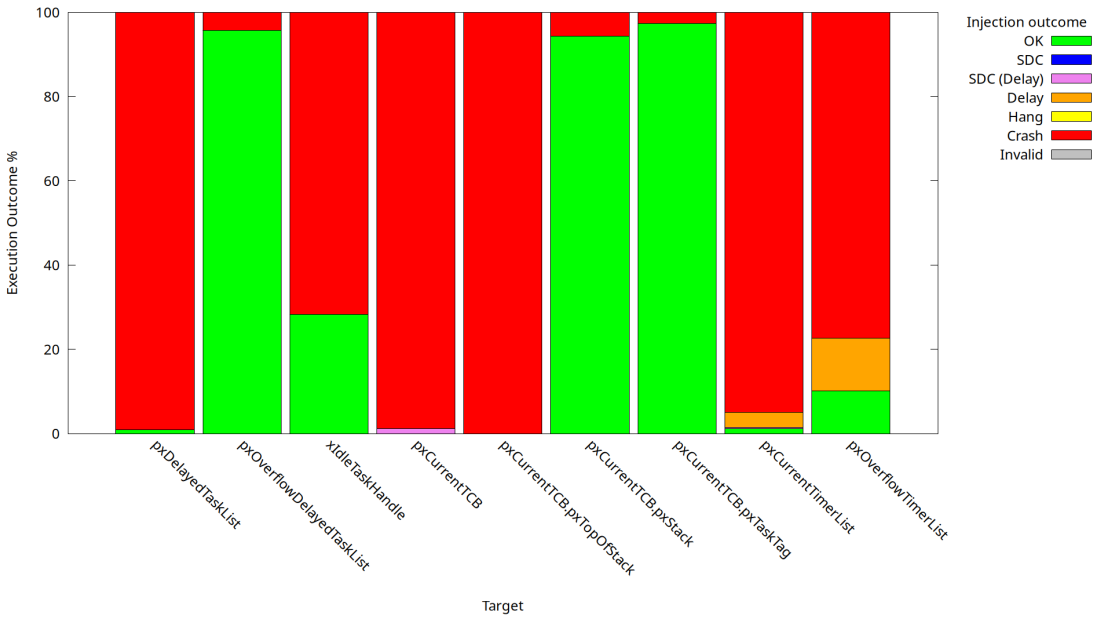

Figure 7.7: Injections on *FreeRTOS* **pointers NO ECC** (**Transient Faults**) *QSRT* items **1000** *TX* iterations **5** *Timer* iterations **5** *RX* iterations **10**

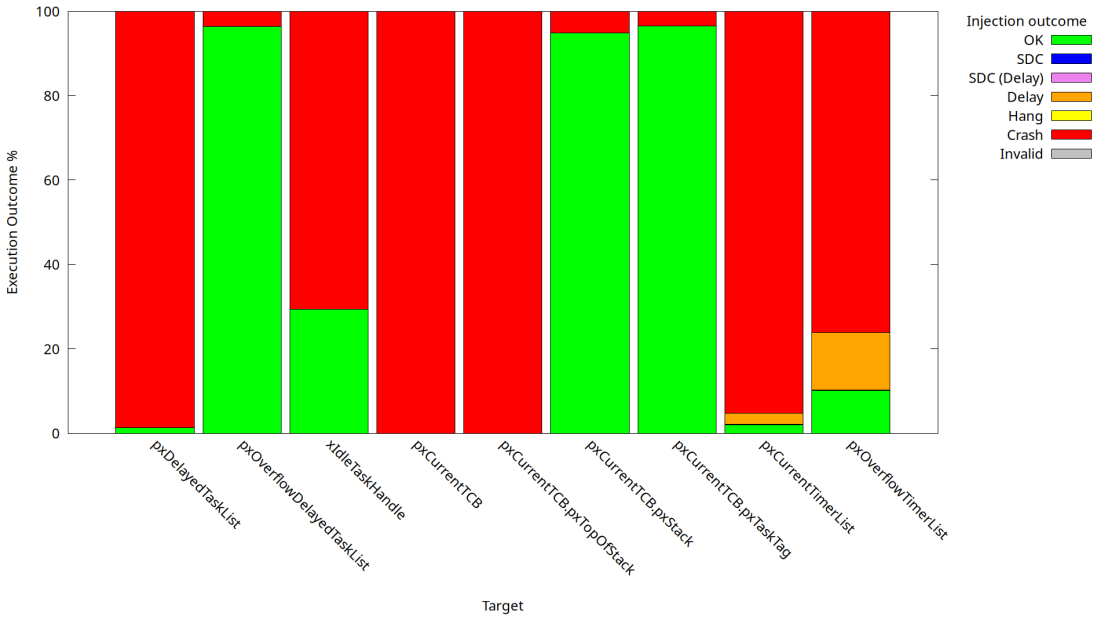

Figure 7.8: Injections on *FreeRTOS* **pointers NO ECC** (**Permanent Faults**) *QSRT* items **1000** *TX* iterations **5** *Timer* iterations **5** *RX* iterations **10**

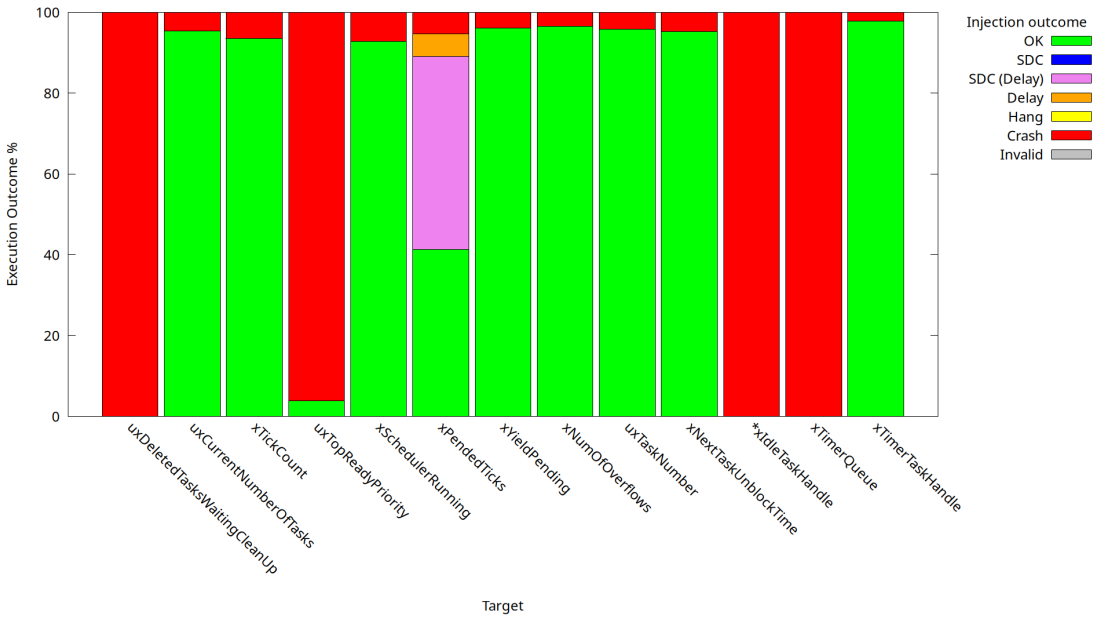

Figure 7.9: Injections on *FreeRTOS* **variables NO ECC** (**Transient Faults**) *QSRT* items **5000** *TX* iterations **10** *Timer* iterations **10** *RX* iterations **20**

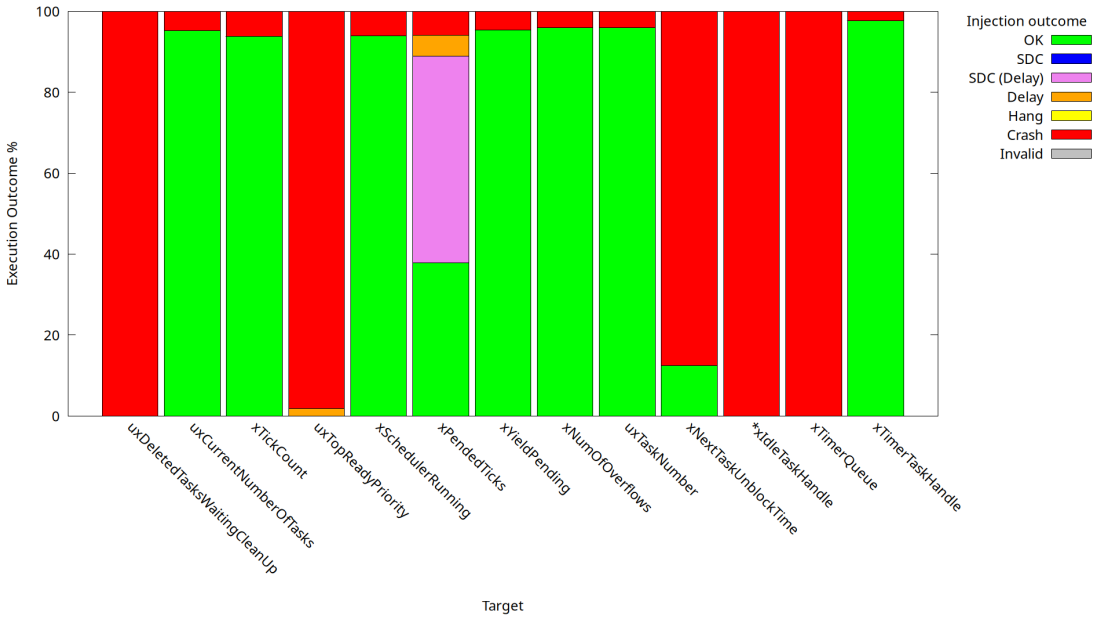

Figure 7.10: Injections on *FreeRTOS* **variables NO ECC** (**Permanent Faults**) *QSRT* items **5000** *TX* iterations **10** *Timer* iterations **10** *RX* iterations **20**

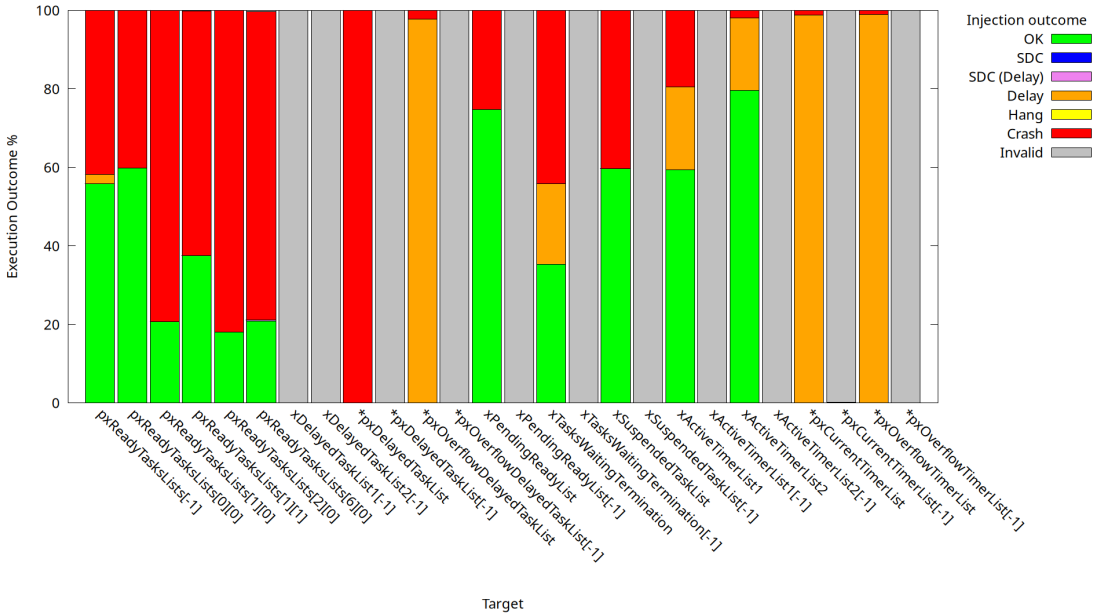

Figure 7.11: Injections on *FreeRTOS* **lists NO ECC** (**Transient Faults**) *QSRT* items **5000** *TX* iterations **10** *Timer* iterations **10** *RX* iterations **20**

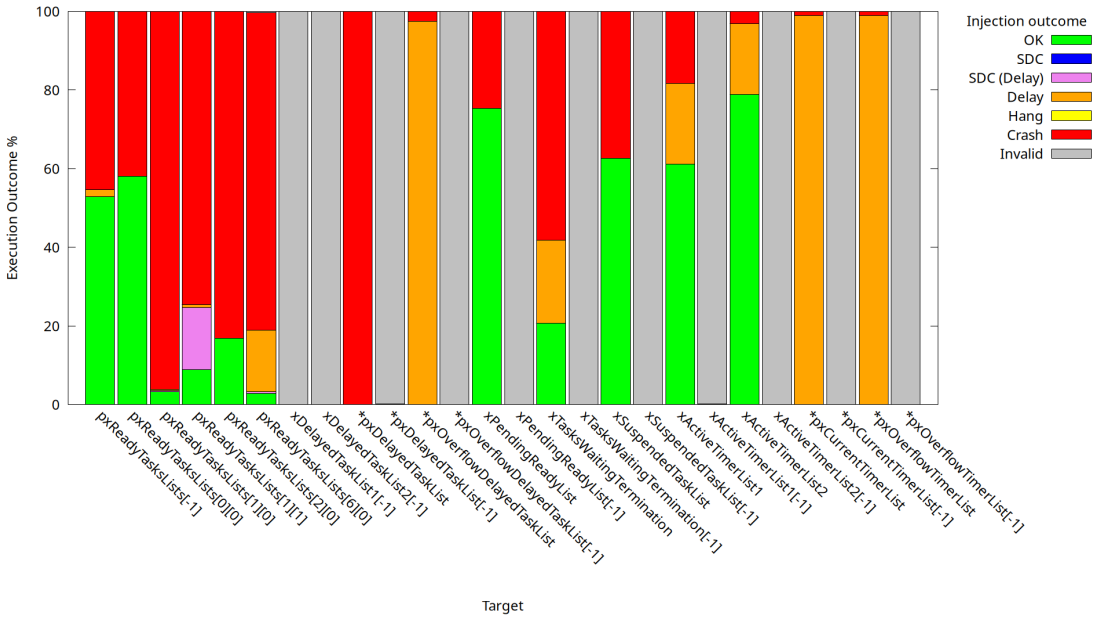

Figure 7.12: Injections on *FreeRTOS* **lists NO ECC** (**Permanent Faults**) *QSRT* items **5000** *TX* iterations **10** *Timer* iterations **10** *RX* iterations **20**

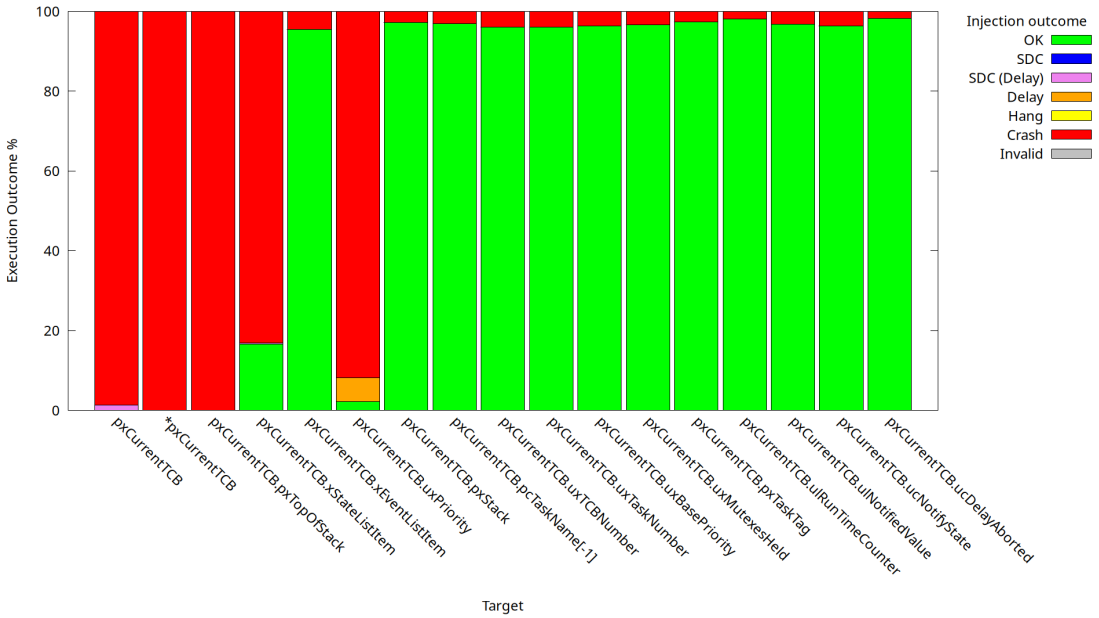

Figure 7.13: Injections on *FreeRTOS* **current TCB NO ECC** (**Transient Faults**) *QSRT* items **5000** *TX* iterations **10** *Timer* iterations **10** *RX* iterations **20**

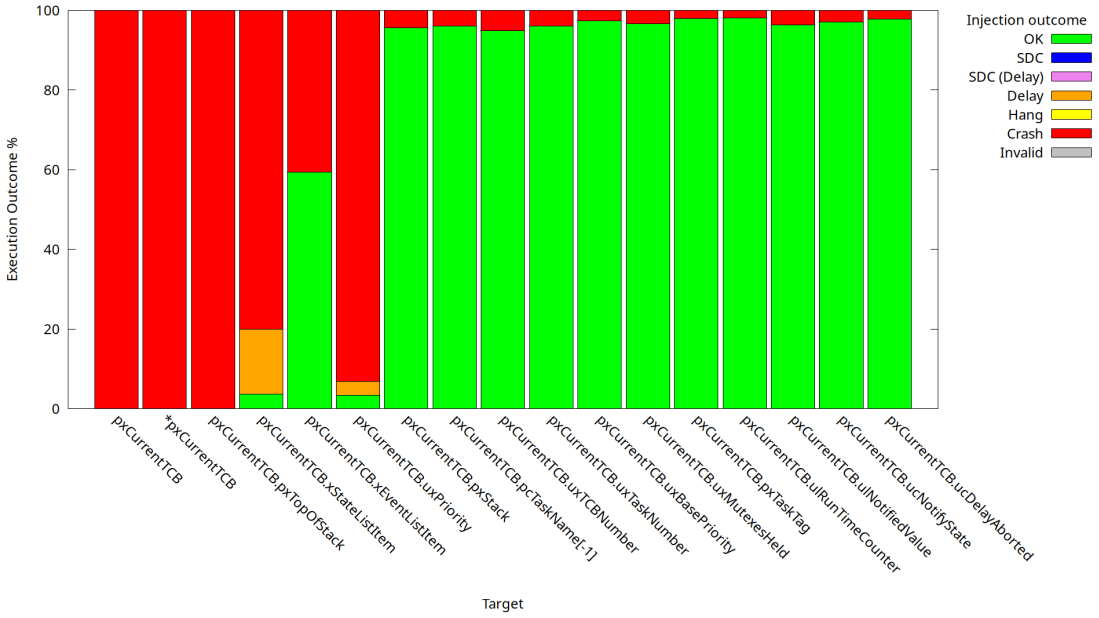

Figure 7.14: Injections on *FreeRTOS* **current TCB NO ECC** (**Permanent Faults**) *QSRT* items **5000** *TX* iterations **10** *Timer* iterations **10** *RX* iterations **20**

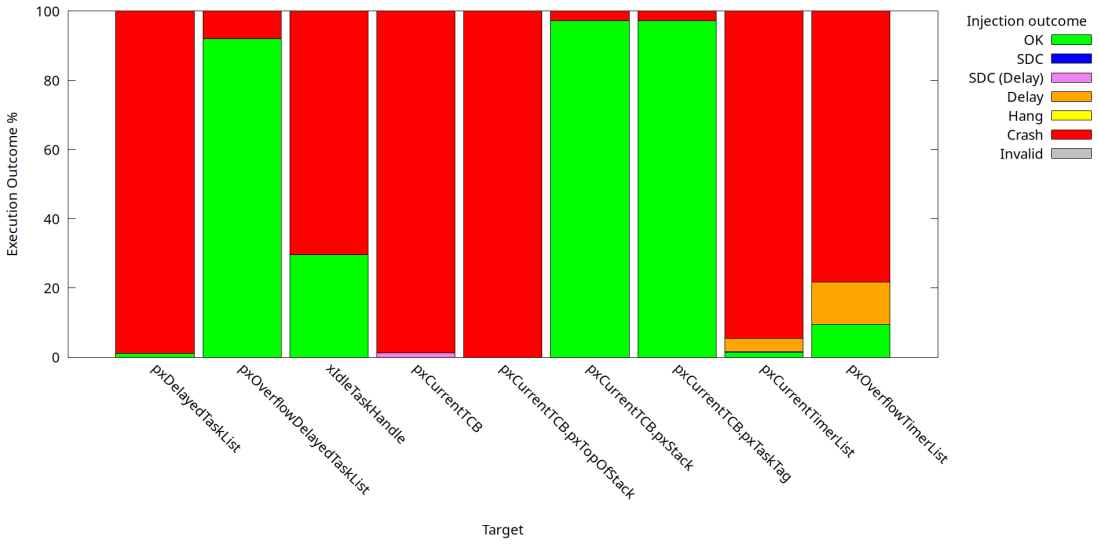

Figure 7.15: Injections on *FreeRTOS* **pointers NO ECC** (**Transient Faults**) *QSRT* items **5000** *TX* iterations **10** *Timer* iterations **10** *RX* iterations **20**

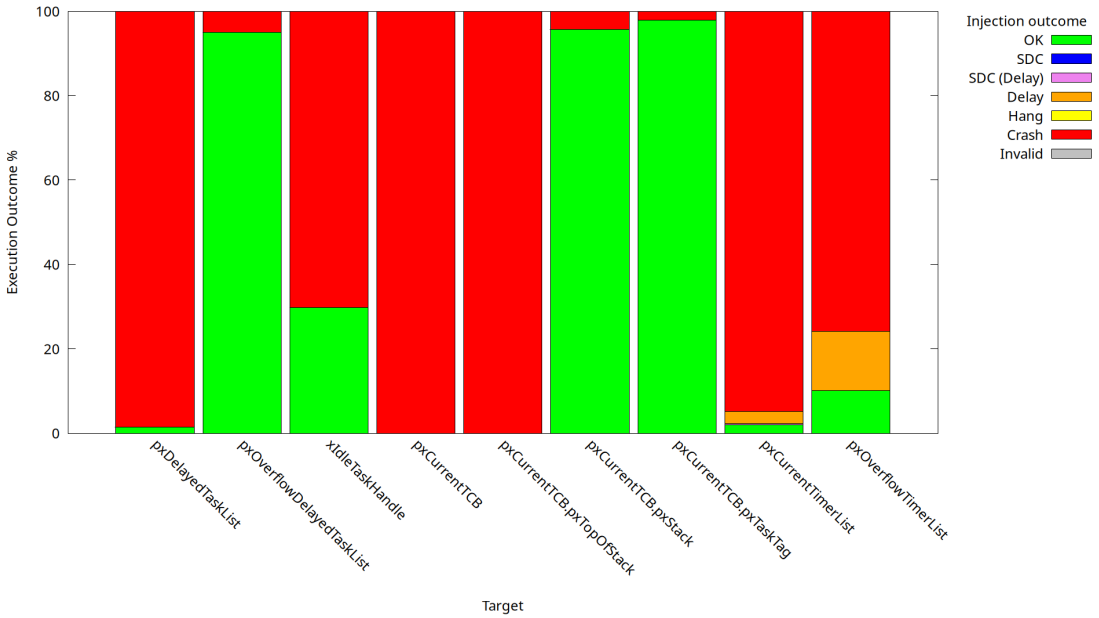

Figure 7.16: Injections on *FreeRTOS* **pointers NO ECC** (**Permanent Faults**) *QSRT* items **5000** *TX* iterations **10** *Timer* iterations **10** *RX* iterations **20**

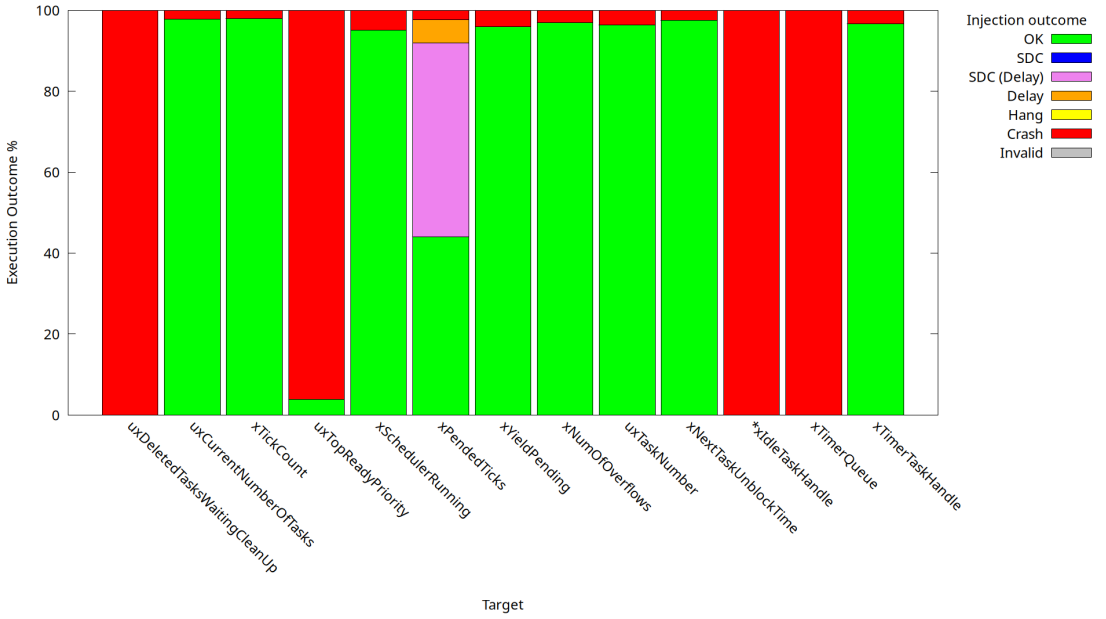

Figure 7.17: Injections on *FreeRTOS* **variables NO ECC** (**Transient Faults**) *QSRT* items **10000** *TX* iterations **20** *Timer* iterations **20** *RX* iterations **40**

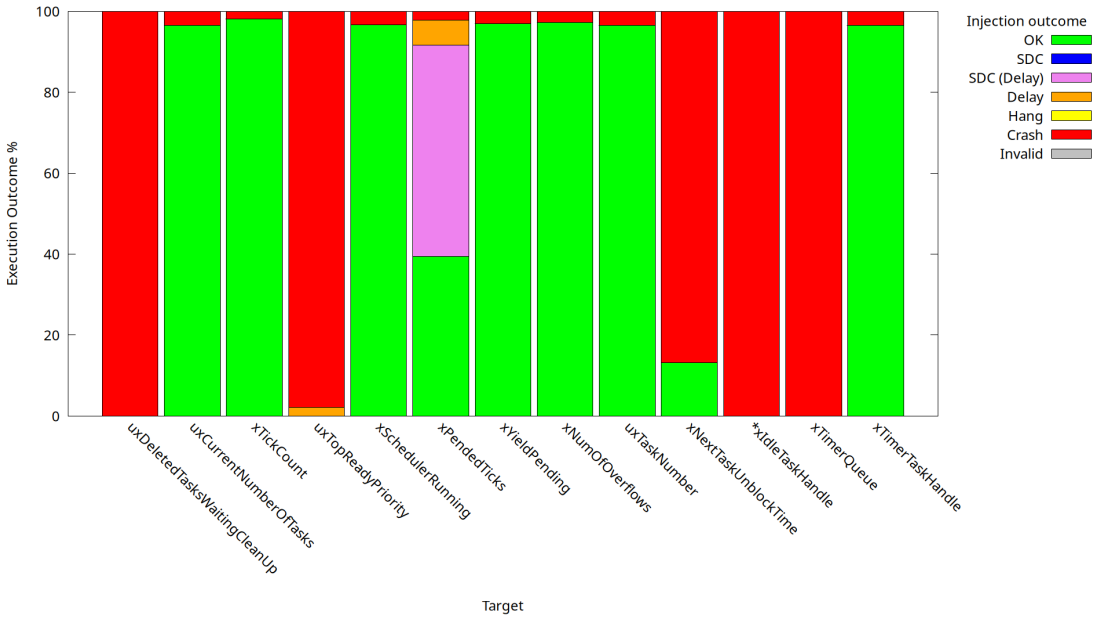

Figure 7.18: Injections on *FreeRTOS* **variables NO ECC** (**Permanent Faults**) *QSRT* items **10000** *TX* iterations **20** *Timer* iterations **20** *RX* iterations **40**

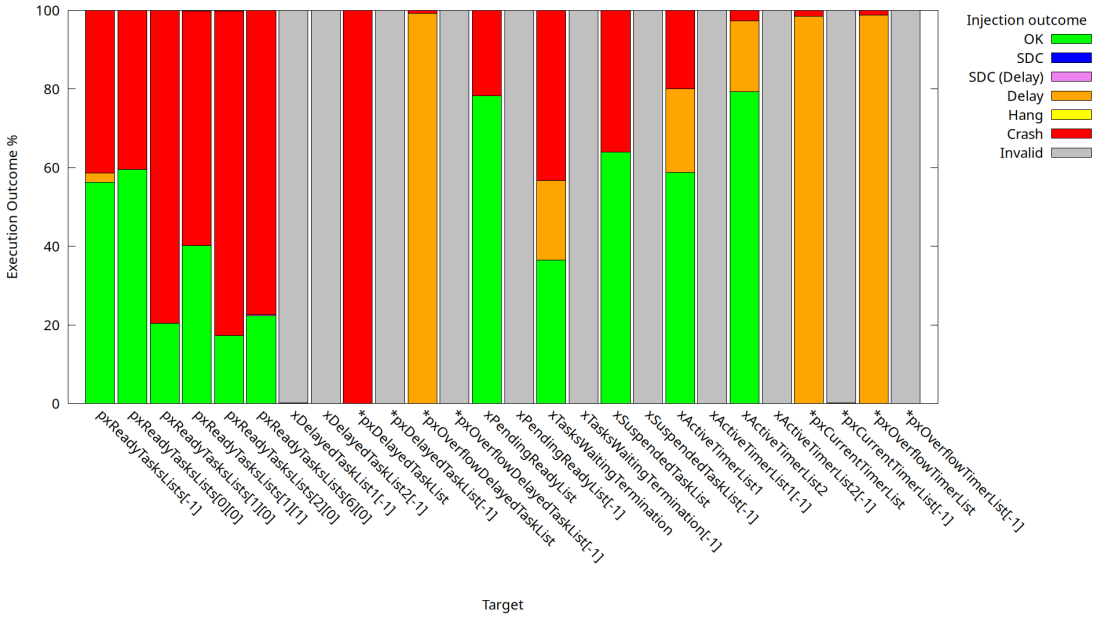

Figure 7.19: Injections on *FreeRTOS* **lists NO ECC** (**Transient Faults**) *QSRT* items **10000** *TX* iterations **20** *Timer* iterations **20** *RX* iterations **40**

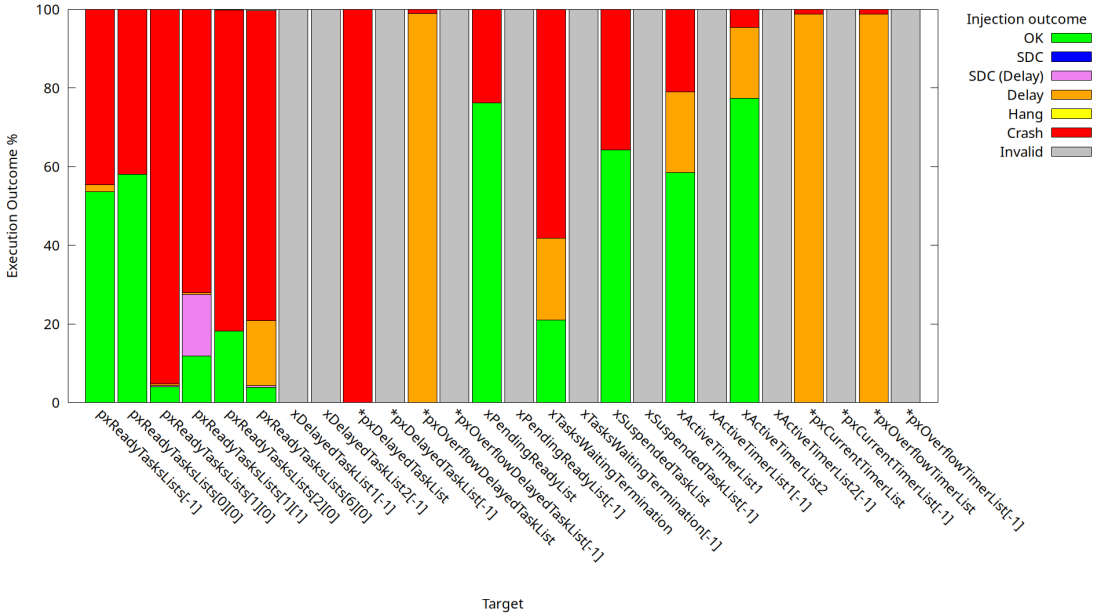

Figure 7.20: Injections on *FreeRTOS* **lists NO ECC** (**Permanent Faults**) *QSRT* items **10000** *TX* iterations **20** *Timer* iterations **20** *RX* iterations **40**

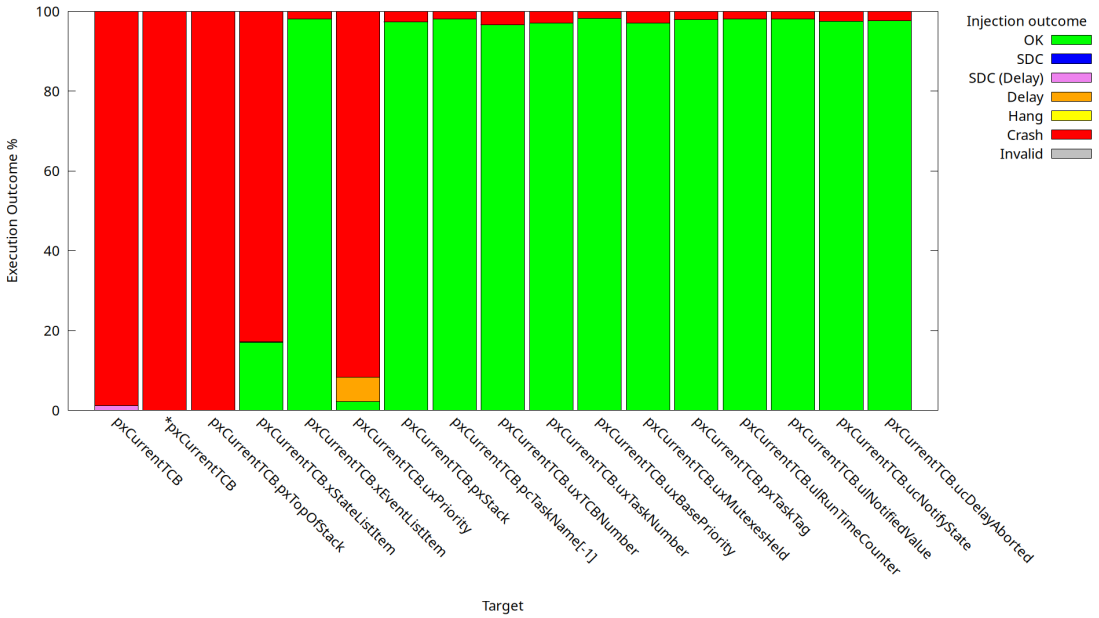

Figure 7.21: Injections on *FreeRTOS* **current TCB NO ECC** (**Transient Faults**) *QSRT* items **10000** *TX* iterations **20** *Timer* iterations **20** *RX* iterations **40**

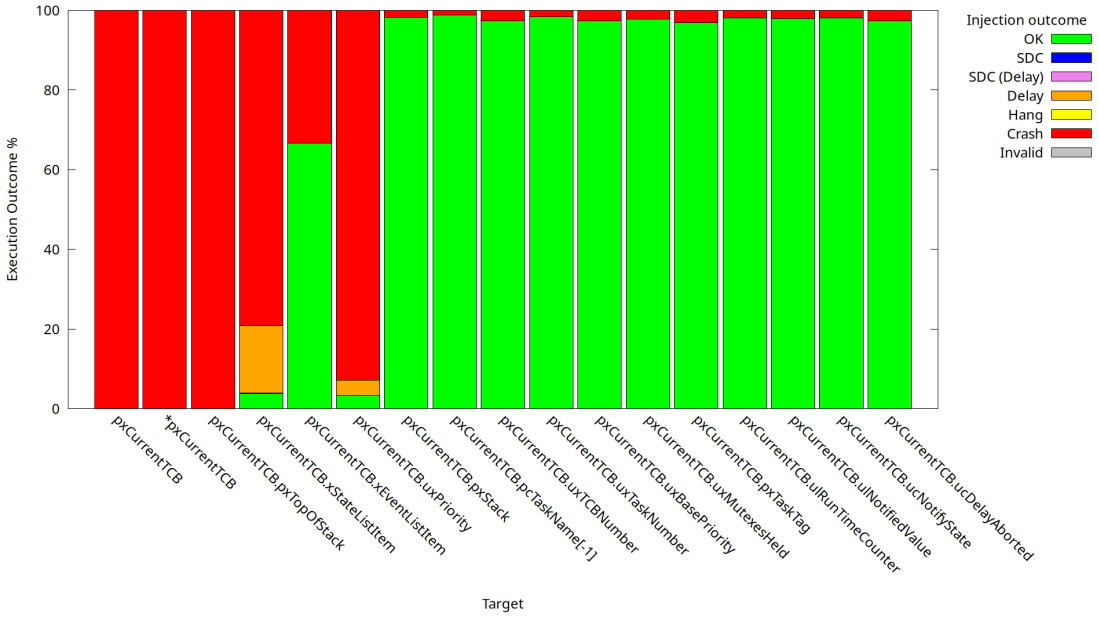

Figure 7.22: Injections on *FreeRTOS* **current TCB NO ECC** (**Permanent Faults**) *QSRT* items **10000** *TX* iterations **20** *Timer* iterations **20** *RX* iterations **40**

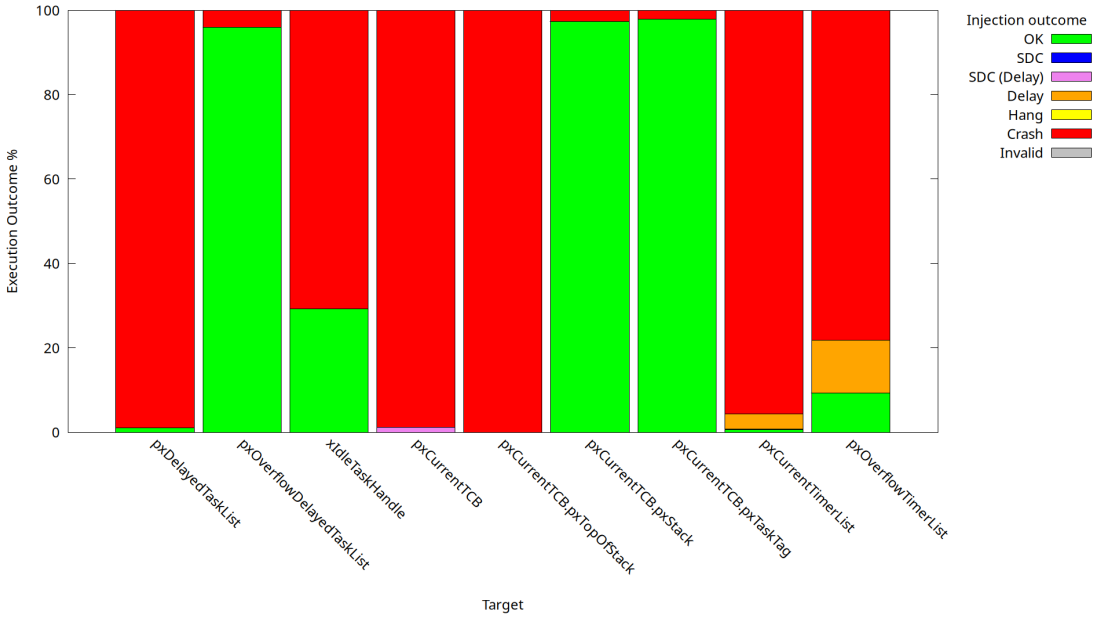

Figure 7.23: Injections on *FreeRTOS* **pointers NO ECC** (**Transient Faults**) *QSRT* items **10000** *TX* iterations **20** *Timer* iterations **20** *RX* iterations **40**

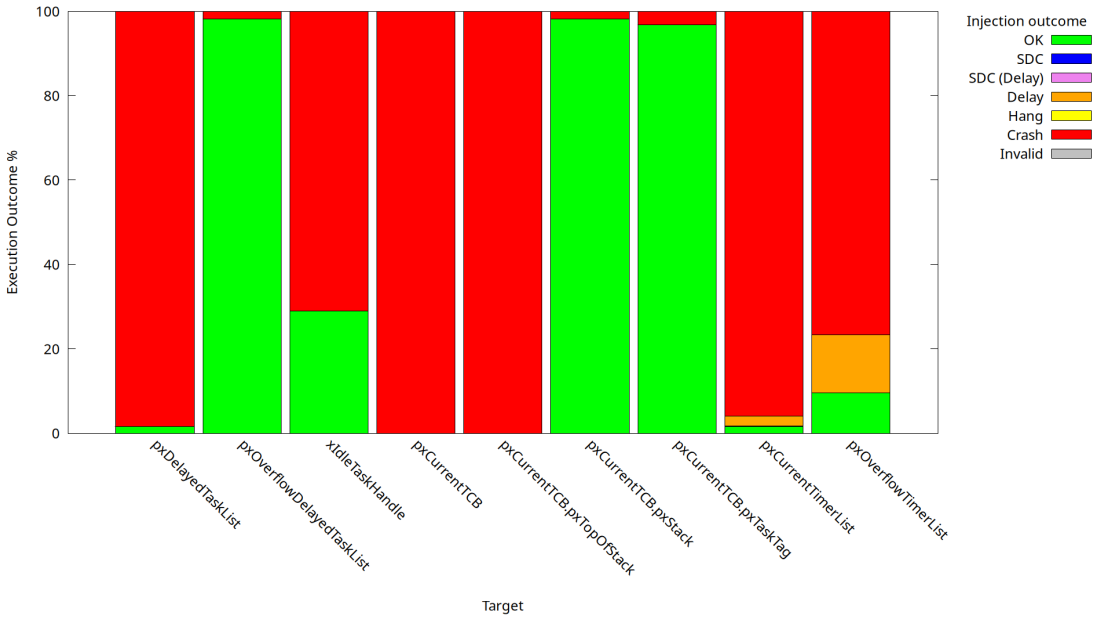

Figure 7.24: Injections on *FreeRTOS* **variables NO ECC** (**Permanent Faults**) *QSRT* items **10000** *TX* iterations **20** *Timer* iterations **20** *RX* iterations **40**

7.3.2 Second Scenario

In contrast to the first scenario, this experiment exercises five benchmarks from the TACLe suite (see subsection 7.3.2), all of which have been ported to the *FreeRTOS*–based framework. The selected benchmarks are **SHA**, **FFT**, **CUBIC**, **HUFF_DEC**, and **ADPCM_ENC**, each representing typical computation patterns found in embedded-industry applications. As in the previous scenario, fault-injection outcomes are reported separately for transient and permanent injection faults, thereby enabling an assessment of each benchmark's fault susceptibility under realistic workload conditions.

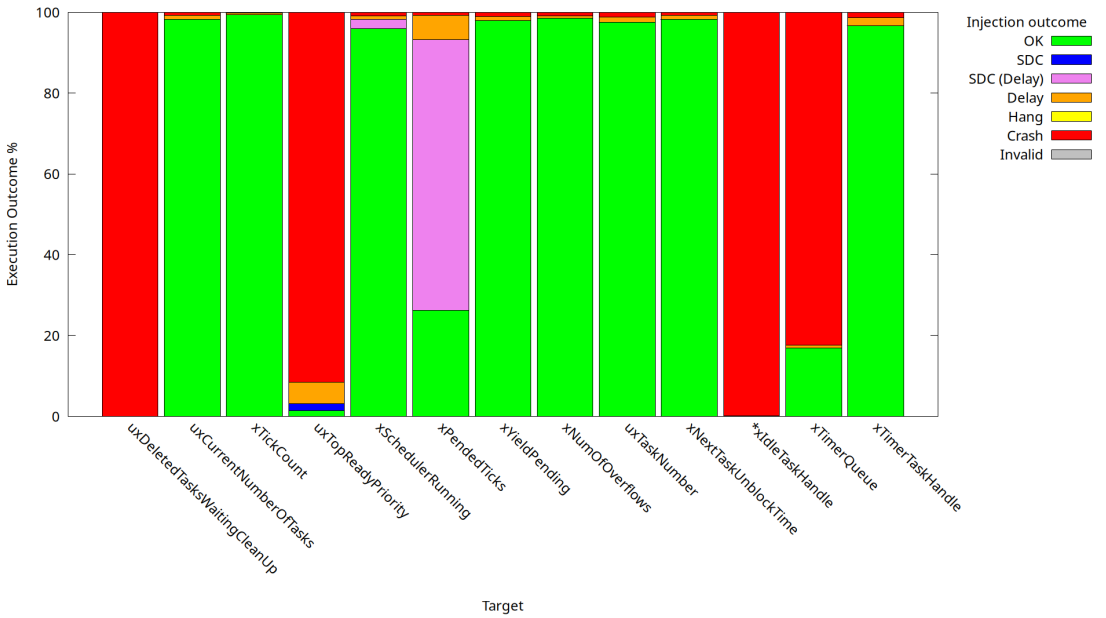

Figure 7.25: Injections on *FreeRTOS* **variables NO ECC** (**Transient Faults**) Tacle Benchmarks: *SHA*, *FFT*, *CUBIC*, *HUFF_DEC*, *ADPCM_ENC*

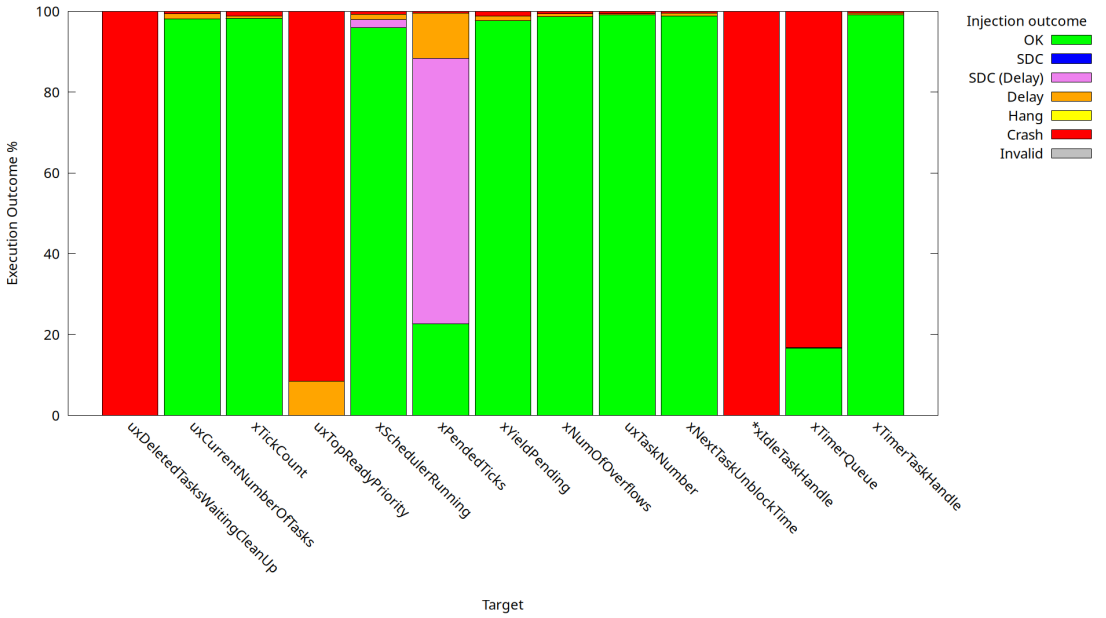

Figure 7.26: Injections on *FreeRTOS* **variables NO ECC** (**Permanent Faults**) Tacle Benchmarks: *SHA*, *FFT*, *CUBIC*, *HUFF_DEC*, *ADPCM_ENC*

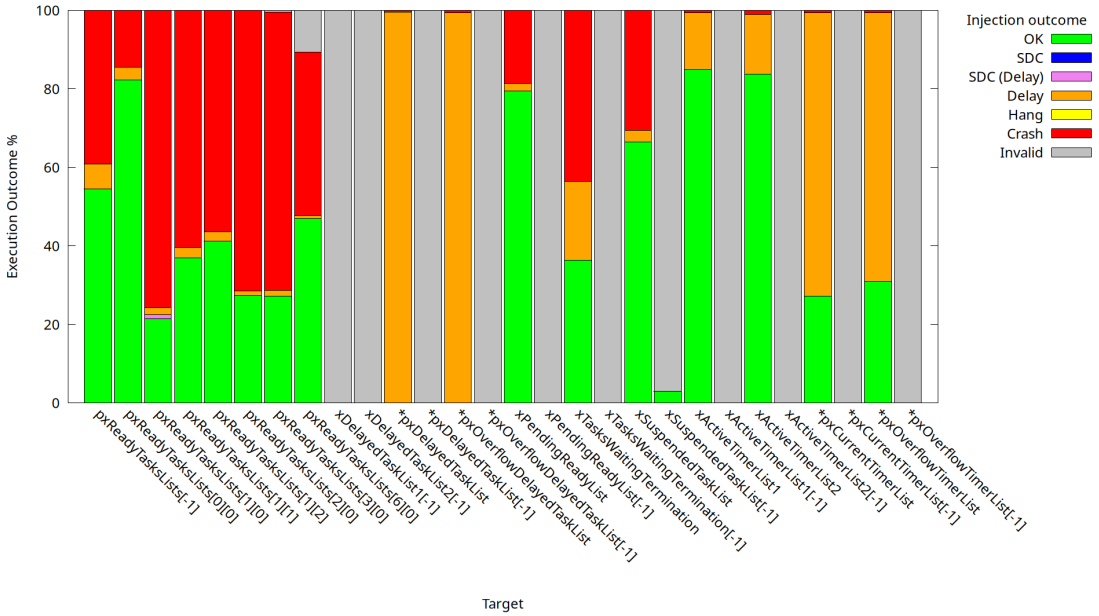

Figure 7.27: Injections on *FreeRTOS* **lists NO ECC** (**Transient Faults**) Tacle Benchmarks: *SHA*, *FFT*, *CUBIC*, *HUFF_DEC*, *ADPCM_ENC*

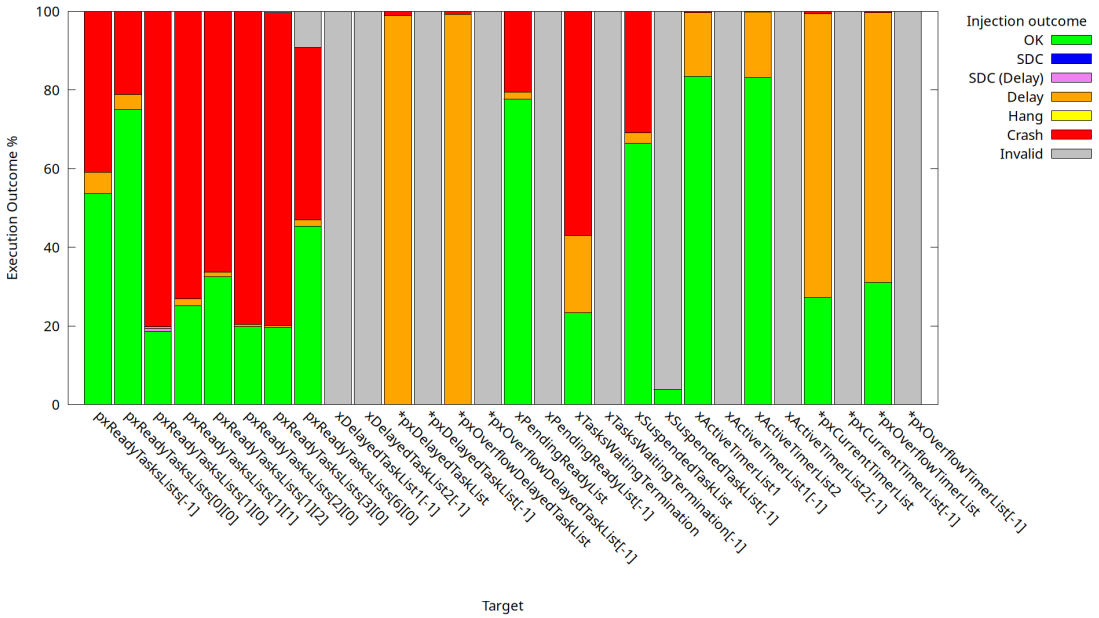

Figure 7.28: Injections on *FreeRTOS* **lists NO ECC** (**Permanent Faults**) Tacle Benchmarks: *SHA*, *FFT*, *CUBIC*, *HUFF_DEC*, *ADPCM_ENC*

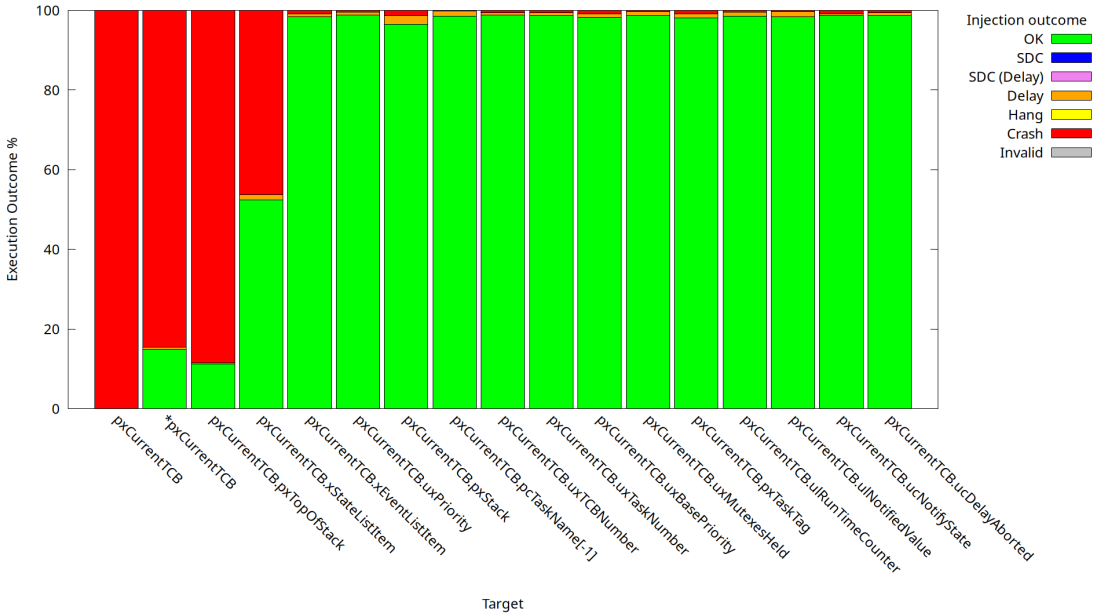

Figure 7.29: Injections on *FreeRTOS* **current TCB NO ECC** (**Transient Faults**) Tacle Benchmarks: *SHA*, *FFT*, *CUBIC*, *HUFF_DEC*, *ADPCM_ENC*

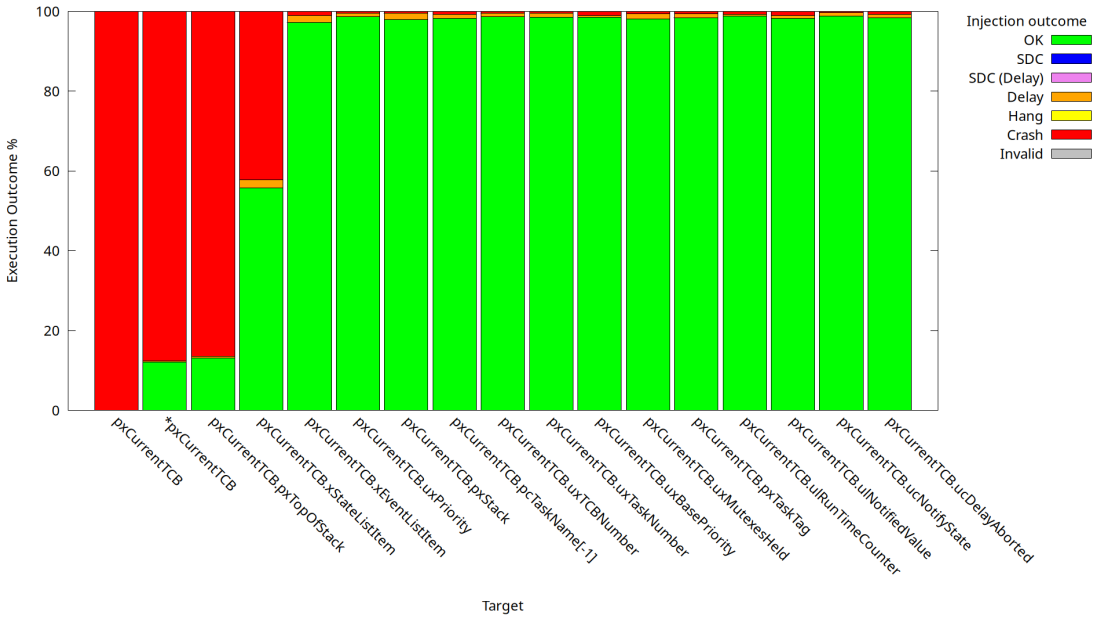

Figure 7.30: Injections on *FreeRTOS* **current TCB NO ECC** (**Permanent Faults**) Tacle Benchmarks: *SHA*, *FFT*, *CUBIC*, *HUFF_DEC*, *ADPCM_ENC*

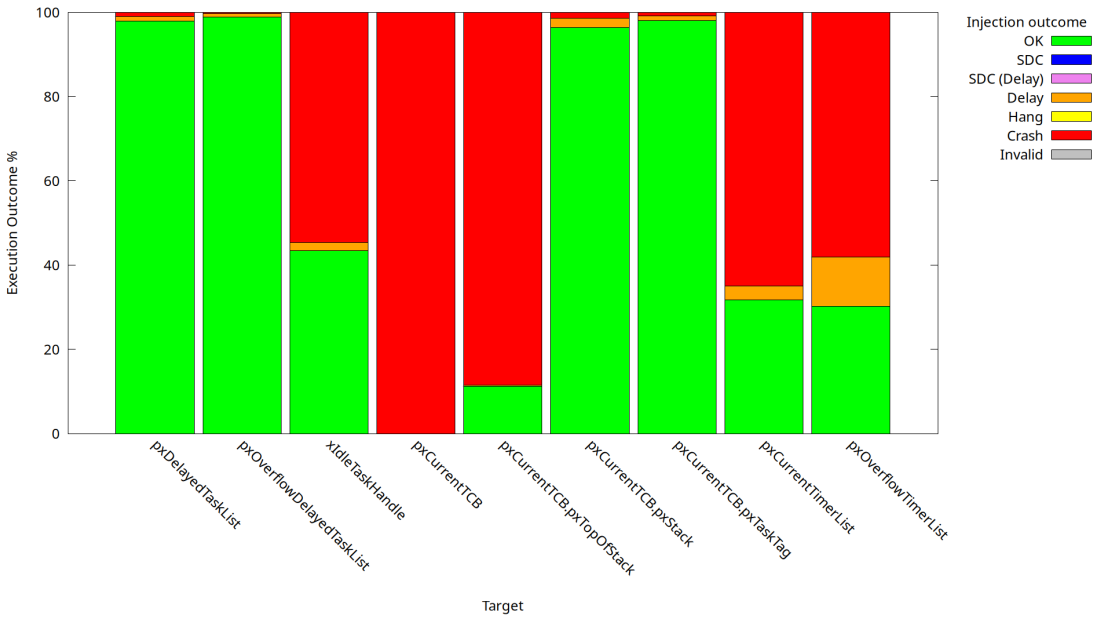

Figure 7.31: Injections on *FreeRTOS* **pointers NO ECC** (**Transient Faults**) Tacle Benchmarks: *SHA*, *FFT*, *CUBIC*, *HUFF_DEC*, *ADPCM_ENC*

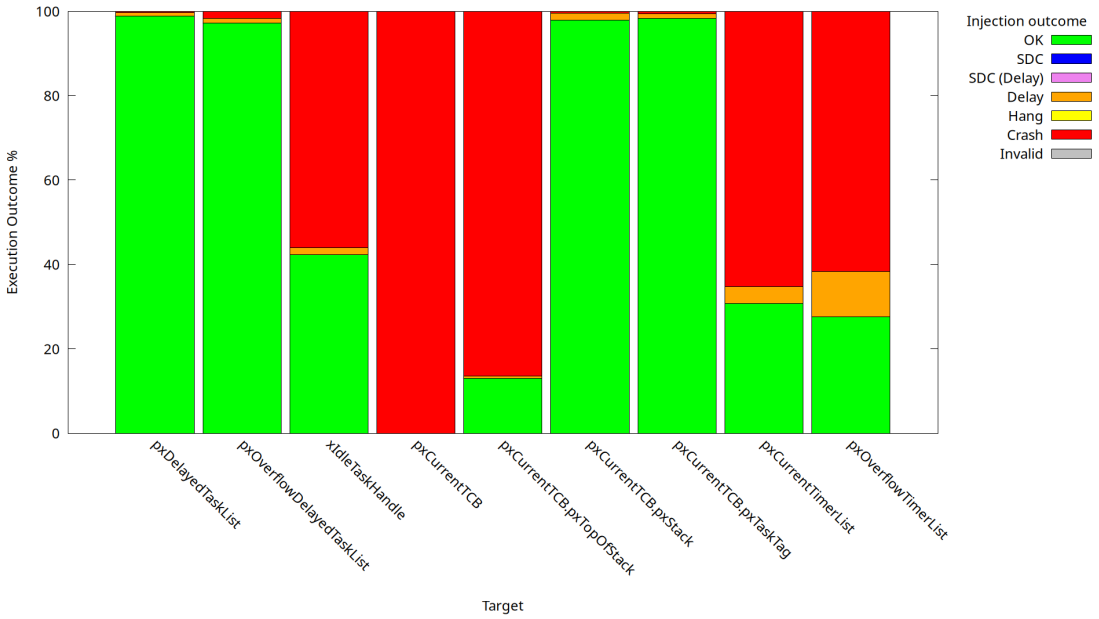

Figure 7.32: Injections on *FreeRTOS* **pointers NO ECC** (**Permanent Faults**) Tacle Benchmarks: *SHA*, *FFT*, *CUBIC*, *HUFF_DEC*, *ADPCM_ENC*

7.4 Results Summary by Target Type First Scenario

This section presents a preliminary analysis of injection outcomes, grouped by target type, based on the results shown in the preceding figures for the first scenario case.

7.4.1 Injection Results on FreeRTOS Variables (ECC Disabled)

- **uxDeletedTasksWaitingCleanUp, xIdleTaskHandle, xTimerQueue** All three targets exhibit a consistent 100% crash rate across every workload.
  - `uxDeletedTasksWaitingCleanUp` is used by the *IDLE* task to reclaim deleted task resources. Corrupting it causes the *IDLE* task to assume non-existent list entries, leading to a crash.
  - `xIdleTaskHandle` and `xTimerQueue` are pointer variables; corrupting either pointer predictably results in an immediate crash.
- **uxCurrentNumberOfTasks, xTickCount** Both variables yield mostly **OK** outcomes, with resilience improving at higher workloads. Occasional crashes indicate limited sensitivity under certain timing conditions.
- **uxTopReadyPriority** Highly critical: injections almost always produce a **CRASH** in the transient-fault scenario and a slight **DELAY** in the permanent-fault case, across all workloads.
- **xPendedTicks** Exhibits consistent behavior across workloads, with crash rates slightly decreasing as workload increases. The dominant failure mode is **SDC (Delay)**, with occasional pure **DELAY**. Explanation: after scheduler suspension, the kernel loops to process all pending ticks. Corrupting high-order bits of `xPendedTicks` inflates this count, causing the RTOS to spend excessive time handling "phantom" ticks.
- **xSchedulerRunning, xYieldPending, xNumOfOverflows, uxTaskNumber, xTimerTaskHandle** These variables show low sensitivity: most injections yield **OK**, with a small fraction of **CRASH** outcomes that further decrease at higher workloads.

7.4.2 Injection Results On FreeRTOS Lists With ECC Disabled

- **pxReadyTasksLists[-1]** Transient faults yield over 50% **OK** and a few percent **Delay**; the remainder are **Crash**. Permanent faults increase the Crash rate. Crash frequency decreases as workload grows, since corruptions of critical `List_t` fields (e.g. `pxIndex`) trigger failures.

- **pxReadyTasksLists[0][0]** Similar pattern: permanent faults produce more Crashes, with crash rates falling under heavier load. List nodes contain vulnerable pointers (`pxNext`, `pxPrevious`, `pvOwner`); corrupting any of these usually causes a Crash.

- **pxReadyTasksLists[1][0]** Permanent faults again maximize Crashes, which decline as workload increases. Occasional **SDC** and **SDC Delay** appear under permanent faults. Since this priority hosts QSRT and TX tasks, corrupting `xItemValue` (used for priority-sorting) can misplace tasks in the round-robin queue, leading to **Delay** or **SDC Delay**.

- **pxReadyTasksLists[1][1]** Follows the same trend, with a higher share of **SDC Delay** in permanent faults. Transient injections here result in fewer Crashes than at other list positions.

- **pxReadyTasksLists[2][0]** Predominantly Crashes, marginally decreasing under heavier workloads.

- **pxReadyTasksLists[6][0]** Transient faults cause mostly Crashes; permanent faults yield about 15% **Delay**. This priority runs the Timer Daemon Task, permanent faults disrupt its scheduling, resulting in **Delay**.

- **Other task lists** (`xDelayedTaskList1[-1]`, `xDelayedTaskList2[-1]`, `*pxDelayedTaskList[-1]`, `*pxOverflowDelayedTaskList[-1]`, `xPendingReadyList[-1]`, `xTasksWaitingTermination[-1]`, `xSuspendedTaskList[-1]`, `xActiveTimerList1[-1]`, `xActiveTimerList2[-1]`, `*pxCurrentTimerList[-1]`, `*pxOverflowTimeList[-1]`):
  Mostly **Invalid** (empty lists at injection); occasional **SDC** when non-empty. High timing sensitivity suggests broad-range injection timing is necessary.

- ***pxDelayedTaskList** 100% **Crash** across all workloads. Dereferencing leads to a `List_t` whose fields (`uxNumberOfItems`, `pxIndex`, `xListEnd`) are highly sensitive.

- ***pxOverflowDelayedTaskList** Predominantly Delay in both fault types, growing with workload. As an inactive delayed-list pointer, it is timing-critical but less prone to **Crash**.
- **xPendingReadyList** Mostly **OK**, with 20% **Crash** (higher in permanent faults). Scheduler lock intervals here are brief, and injections target the `List_t` structure rather than individual nodes, reducing crash likelihood.
- **xTasksWaitingTermination** High Crash rates (greater for permanent faults and under heavier load). Consistent **Delay** outcomes arise from the *IDLE* hook that polls `uxNumberOfItems` until zero (see 6.6); corruptions delay simulation termination.
- **xSuspendedTaskList** Mostly **OK** with a substantial **Crash** fraction. Transient faults cause more Crashes due to timing variability; crash rates fall as workload increases.
- **xActiveTimerList1** Primarily **OK**, significant **Delay**, and few **Crash**. Crashes decrease with load; Delays remain steady. Corrupting timer node pointers causes Crashes; corrupting `xItemValue` postpones timer expiry, causing Delay.
- **xActiveTimerList2** Similar to `xActiveTimerList1` but less crash-prone, as the two lists swap on tick-count overflow. Delays remain consistent.
- ***pxCurrentTimerList** and ***pxOverflowTimerList** Point to the active (non-overflowed) and overflowed timer lists, respectively. Both yield mostly **Delay**, since corrupting `xItemValue` defers all timer expirations far into the future.

7.4.3 Injection Results on FreeRTOS Current TCB with ECC Disabled

The most insensitive fields in the current TCB are identified as follows:

- `pxCurrentTCB.pxStack`
- `pxCurrentTCB.pcTaskName[-1]`
- `pxCurrentTCB.uxTCBNumber`
- `pxCurrentTCB.uxTaskNumber`
- `pxCurrentTCB.uxBasePriority`

- `pxCurrentTCB.uxMutexesHeld`
- `pxCurrentTCB.pxTaskTag`
- `pxCurrentTCB.ulRunTimeCounter`
- `pxCurrentTCB.ulNotifiedValue`
- `pxCurrentTCB.ucNotifyState`
- `pxCurrentTCB.ucDelayAborted`

Most of these injection targets yield **OK** outcomes, with a crash rate between approximately 2% and 7%. Interestingly, the crash frequency decreases as workload increases. In critical kernel paths, none of these fields see frequent use. For example, `pxStack` is unused in the POSIX port, and neither `uxBasePriority` nor `uxMutexesHeld` affect this benchmark suite, since no mutexes are held during the experiment (further investigation is required for mutex-related scenarios). Fields such as `pxTaskTag`, `ulNotifiedValue`, and `ucNotifyState` are not exercised by these benchmarks. The names `pcTaskName[-1]`, `uxTCBNumber`, and `uxTaskNumber` serve only for debugging and do not influence *FreeRTOS*'s computations.
Permanent injections into `pxCurrentTCB` cause a 100% **Crash** rate across all workloads. Transient injections exhibit a small but growing **SDC Delay** as workload increases. This delay likely arises when lower-order bits of the task pointer are flipped: the pointer remains valid long enough for the RTOS to overwrite it, whereas a permanent fault prevents any subsequent correction.
Note that `*pxCurrentTCB` and `pxCurrentTCB.pxTopOfStack` are semantically identical, since the first member of the `TCB_t` structure (see Section 4.5.1) is `pxTopOfStack`. In the POSIX port, this location is used to track each pthread that backs a *FreeRTOS* task (see 6.4.2), making it highly sensitive.
The `xStateListItem` field also shows a high propensity for crashes in both transient and permanent fault models, though the crash rate diminishes under heavier workloads. Permanent faults in this field induce about 16% **Delay**, whereas transient faults yield roughly 1% **SDC** and **SDC Delay**.
For `xEventListItem`, transient faults predominantly result in **OK** outcomes (with decreasing success under higher workloads), while permanent faults generate an increasing fraction of crashes, which in turn slightly declines with workload.
Finally, injections into `uxPriority` produce crashes in both fault models, with the crash rate rising slightly as workload grows. Each fault type also induces around 5% **Delay**. Such delays can occur if bit-flips lower the priority into a valid range; flips of higher-order bits push the priority outside permissible bounds, leading to out-of-range accesses in `pxReadyTasksLists[Priority]` and thus complete failure.

7.4.4 Injection Results On FreeRTOS Pointers With ECC Disabled

`pxCurrentTCB`, `pxCurrentTCB.pxTopOfStack`, `pxCurrentTCB.pxStack`, `pxCurrentTCB.pxTaskTag` were already discussed in the section on current-TCB targets and won't be repeated here.

pxDelayedTaskList Yields predominantly **Crash** outcomes, with the crash rate decreasing slightly as workload increases (both in permanent and transient injection scenarios).

pxOverflowDelayedTaskList Exhibits mostly **OK** outcomes and approximately 2%–8% **Crash**, which also declines modestly under heavier workloads. This pointer is therefore not highly sensitive to injection faults.
The FreeRTOS kernel periodically swaps these pointers (`pxDelayedTaskList` and `pxOverflowDelayedTaskList` between `xDelayedTaskList1` and `xDelayedTaskList2`, automatically correcting the pointer values in the transient faults.

xIdleTaskHandle A critical pointer, particularly in the POSIX port, which calls `xTaskGetIdleTaskHandle()` to delete the *IDLE* task upon scheduler shutdown. Here, injections yield mostly **Crash**, with only 30%–35% **OK** results that remain essentially constant across all workloads.

pxCurrentTimerList Produces mainly **Crash** outcomes, with the crash frequency rising slightly as the workload grows (in both permanent and transient cases). Additionally, there are about 2%–5% **Delay** outcomes, likely due to flipping the pointer to the alternate timer list, again with marginally more delays under transient injection.

pxOverflowTimerList Also shows mostly **Crash**, but retains a consistent 10% **OK** rate compared to `pxCurrentTimerList`. Delay outcomes here range from 10% to 17%, with the permanent-injection case exhibiting slightly more delays.

7.5 Results Summary by Target Type Second Scenario

This section presents a preliminary analysis of injection outcomes, grouped by target type, based on the results shown in the preceding figures for the second scenario case.

7.5.1 Injection Results on FreeRTOS Variables (ECC Disabled)

The variables `uxDeletedTasksWaitingCleanup`, `uxTopReadyPriority`, `xIdleTaskHandle`, and `xTimerQueue` exhibit the highest crash rates in both transient and permanent fault cases. In the transient case, `uxTopReadyPriority` also yields a small fraction of SDC mis-scheduled tasks due to an incorrect top-ready priority, but these SDC disappear under permanent faults, where delays dominate instead. Both `uxDeletedTasksWaitingCleanup` and `xIdleTaskHandle` experience nearly 100% crash rates for both fault types; `xTimerQueue` shows predominantly crashes in both transient and permanent cases, with only minor delay outcomes in the transient case.
The variable `xPendedTicks` produces predominantly delays (including SDC-delays) under both transient and permanent faults. All other FreeRTOS variables in this group yield mostly OK outcomes, with occasional crashes and delays appearing only under transient faults.

7.5.2 Injection Results on FreeRTOS Lists (ECC Disabled)

Injections into `pxReadyTasksLists[-1]` and `pxReadyTasksLists[0][0]` produce consistent outcomes in both transient and permanent fault cases, with crash rates rising under permanent faults. Minor delay outcomes remain roughly constant between the two cases.
Faults injected into `pxReadyTasksLists[1][0]`, `pxReadyTasksLists[1][1]`, `pxReadyTasksLists[1][2]`, `pxReadyTasksLists[2][0]`,
and `pxReadyTasksLists[3][0]` yield predominantly crashes, supplemented by a small fraction of delays. Crash frequency increases under permanent faults.
Both `*pxDelayedTaskList` and `*pxOverflowDelayedTaskList` exhibit mostly delay outcomes, with a minor share of crashes that become slightly more frequent in permanent faults.
`xPendingReadyList` behaves consistently across transient and permanent injections, returning mostly OK results with occasional crashes and slight delays.
`xTasksWaitingTermination` shows primarily crashes, more so under permanent faults, with a small, steady fraction of delay outcomes in both cases.
`xSuspendedTaskList` yields mostly OK results, interspersed with some crashes and very few delays; behavior is comparable for both fault types.
Faults injected into `xActiveTimerList1` and `xActiveTimerList2` produce predominantly OK outcomes, a small number of delays, and an even smaller fraction of crashes, which actually decrease under permanent faults.
Finally, injections into `*pxCurrentTimerList` and `*pxOverflowTimerList` result mainly in delays, accompanied by a substantial share of OK outcomes and slight

crashes that diminish in the permanent case.
All remaining list targets exhibit invalid outcomes, reflecting their high time-sensitivity and the use of blocking features (e.g. `vTaskDelay`) in the benchmarked applications.

7.5.3 Injection Results on FreeRTOS Current TCB (ECC Disabled)

- `pxCurrentTCB`: 100% crashes under both transient and permanent faults.
- `*pxCurrentTCB`, `pxCurrentTCB.pxTopOfStack`: Predominantly crashes, with a small fraction of delays that decrease in the permanent-fault case.
- `pxCurrentTCB.xStateListItem`: Mostly OK outcomes in both scenarios, with minor delays that become more frequent under permanent faults.
- All other targets in this group behave consistently across fault types, yielding mostly OK results and a slight presence of crashes and delays.

7.5.4 Injection Results on FreeRTOS Pointers (ECC Disabled)

- `pxDelayedTaskList` and `pxOverflowDelayedTaskList` Predominantly OK outcomes under both transient and permanent faults. Compared to the first benchmark scenario, `pxDelayedTaskList` is less sensitive, likely because this scenario exercises blocking calls less thoroughly. Minor delays and occasional crashes also occur.
- `xIdleTaskHandle` Mostly crashes in both fault scenarios, with a few consistent delays.
- `pxCurrentTCB` and `pxCurrentTCB.pxTopOfStack` `pxCurrentTCB` experiences 100% crashes under both transient and permanent faults. `pxCurrentTCB.pxTopOfStack` also fails in most injections, with slight delays that increase modestly under permanent faults.
- `pxCurrentTimerList` and `pxOverflowTimerList` A mix of outcomes: crashes dominate (with higher rates in permanent faults), delays occur more often under permanent faults, and a substantial share of OK results persists in both cases.
- `pxCurrentTCB.pxStack` and `pxCurrentTCB.pxTaskTag` Mostly OK outcomes in both transient and permanent injections, with a small fraction of delays and crashes that remain consistent across fault types.

7.6 FreeRTOS Fault-Sensitive Locations

Analysis revealed numerous data-structure fields in the *FreeRTOS* kernel that are highly vulnerable to injected faults. In particular, pointer variables emerged as the most sensitive targets, surpassing even other critical fields such as `uxDeletedTasksWaitingCleanup`, `uxTopReadyPriority`, `xNextTaskUnblockTime`, and various list and list-node descriptors.
Given that pointers proved the most failure-prone locations, this thesis adopts a selective hardening strategy focused on pointer variables within the *FreeRTOS* kernel. By reinforcing only those elements with the highest risk of catastrophic failure, the approach maximizes system resilience while limiting additional overhead.

7.7 Selective Hardening

Selective hardening applies protection only to the most critical code and data regions. Analysis of the unprotected *FreeRTOS* fault-injection results revealed that pointer variables are by far the most vulnerable targets in the kernel. Although modern general-purpose architectures sometimes mitigate pointer corruption at the application level via virtual memory or memory-protection units, these protections rarely extend to the kernel itself. As a result, a corrupted kernel pointer can still cause a complete system failure.
Micro-kernel designs address this gap by minimizing the kernel's trusted computing base, but they incur significant overhead from additional context switches. In deeply resource-constrained embedded systems, where low latency and minimal overhead are paramount, such an approach is often impractical. A more lightweight alternative is selective hardening: identify the pointer-related data structures that dominate system failure and apply targeted encoding or redundancy only to those fields. This focused strategy delivers most of the reliability benefits of full fault tolerance while keeping performance and code size overhead to a minimum.

7.7.1 Chosen Approach

Error-correcting codes (ECC) were selected for their minimal intrusiveness and low overhead. In comparison, data-redundancy techniques, such as triplication with majority voting, impose significant storage and runtime costs, requiring majority-vote computation on each read and triple updates on each write.

7.7.2 Error Correcting Codes

Error Correcting Codes (ECC) are a fundamental tool for detecting and correcting faults in stored or transmitted data. By appending carefully chosen parity bits to

a block of user data, one can recover from a certain number of bit-flips, whether induced by transient disturbances (e.g. cosmic rays, power-supply glitches) or by permanent defects in memory or communication channels.

General Framework A linear block ECC transforms a data word of length k into an n-bit *codeword* by adding $r = n - k$ parity bits. On retrieval, the received codeword $c \in \{0,1\}^n$ is tested against a set of parity-check equations. Any inconsistency yields a nonzero *syndrome* vector, from which up to

$$t = \left\lfloor \frac{d_{\min} - 1}{2} \right\rfloor$$

errors can be located and corrected. Here $d_{\min}$ denotes the minimum Hamming distance of the code.

Key Performance Metrics

- **Error Detection vs. Correction.** A code of distance $d_{\min}$ can detect up to $d_{\min} - 1$ simultaneous bit-flips, and correct up to $t = \lfloor (d_{\min} - 1)/2 \rfloor$.
- **Storage Overhead.** The ratio $\frac{n}{k} = 1 + \frac{r}{k}$ quantifies the extra memory or bandwidth required.
- **Latency and Complexity.** Encoding/decoding typically reduce to XOR operations over $\{0,1\}$-vectors of length at most n, inducing additional read-/write latency proportional to r and a hardware or software cost proportional to $O(n \cdot r)$.

ECC are pervasive in embedded systems, storage devices (DRAM, flash), network protocols, and any safety-critical application where undetected data corruption is unacceptable.

7.7.3 Hamming Codes

Hamming codes form the canonical family of single-error–correcting, double-error–detecting linear block codes. An (n, k) Hamming code is characterized by

$$n = 2^r - 1, \quad k = n - r,$$

where r is the number of parity bits.

Codeword Layout Place parity bits at positions whose indices are powers of two, i.e. positions

$$1,2,4,\ldots,2^{r-1}$$

in the n-bit codeword. The remaining k positions carry the data bits in left-to-right order.

Parity-Check Matrix Define the $r \times n$ parity-check matrix H so that its columns enumerate all nonzero binary vectors of length r. For the (7,4) code ($r = 3$), one has

$$H = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix},$$

with columns labelled by positions $1,\ldots,7$.

Encoding Given a data vector $d \in \{0,1\}^k$, solve for the parity bits $p \in \{0,1\}^r$ such that

$$H \begin{pmatrix} p \\ d \end{pmatrix} = \mathbf{0} \pmod 2.$$

The full codeword is then $c = (\, p \,|\, d \,) \in \{0,1\}^n$.

Decoding Upon receiving c', compute the syndrome

$$s = H\, c'^{\mathsf{T}} \bmod 2.$$

- If $s = \mathbf{0}$, no errors are detected.
- If $s \neq \mathbf{0}$, interpret the binary vector s as an integer in $\{1,\ldots,n\}$; this index pinpoints the single flipped bit, which is then inverted to correct the error.

Summary of Properties

- Minimum Hamming distance: $d_{\min} = 3$.
- Error capabilities: single-bit correction ($t = 1$), double-bit detection.
- Overhead: $r/(2^r - 1)$ fraction of extra bits.
- Complexity: encoding and decoding in $O(nr)$ bit-wise XOR operations.

Thanks to their extremely low decoding complexity and modest overhead, Hamming codes are ubiquitous in memory systems (e.g. ECC-protected DRAM) and serial-link controllers. In particular, they are well suited for protecting critical pointer and control-word data in real-time kernels (e.g. *FreeRTOS*), where single-bit faults dominate and high reliability is required.

7.7.4 Unused Bits In Pointer Variables

This work is inspired by Ahmed et al. [22], which embeds linear-block ECC parity directly into neural-network weights via a multi-task learning objective. Although the present application does not involve neural networks, the technique of co-locating parity bits with primary data bits serves as the foundation for the chosen pointer-hardening strategy.
All experiments employ simulated *FreeRTOS* POSIX and *Windows* ports compiled as 64-bit x86_64 binaries. Although these binaries nominally use 64-bit pointers, modern CPU and MMU do not implement the full 2^{64} address space. In practice, hardware limits both physical and virtual addresses to a smaller subset, typically 48 bits or lower on current AMD and Intel consumer processors. For example, on a *Linux* system with an AMD Ryzen 9 7950X processor, the command

```
cat /proc/cpuinfo | grep address
address sizes   : 48 bits physical, 48 bits virtual
```

reveals that only 48 of the 64 pointer bits are actually used. Consequently, the unused high-order bits can be repurposed to embed ECC parity directly within each pointer.

7.7.5 Hamming ECC Implementation

This module encodes and decodes 48-bit data values (e.g. pointers) into a 54-bit Hamming code stored in a 64-bit word. It enables detection and correction of any single-bit error before extracting the original data.

Configuration

- `DATA_BITS = 48`, `PARITY_BITS = 6`.
  The six parity bits satisfy the Hamming requirement $2^p \geq p + m + 1$ for $m = 48$.

- A compile-time check ensures `DATA_BITS` + PARITY_BITS ≤ 64.

Encoding

The entry point is:

Listing 7.1: Hamming Encode Function

```
1 uintptr_t _ullHammingEncode(uintptr_t ullData)
2 {
3     uintptr_t ullCodeword = prvAddRedundantBits(ullData);
4     ullCodeword = prvComputeParity(ullCodeword);
```

```
5       return ullCodeword;
6   }
```

prvAddRedundantBits Scans bit positions 1–(DATA_BITS+PARITY_BITS), skips the power-of-two indices reserved for parity, and packs the 48 data bits (LSB first) into the remaining slots.

Listing 7.2: prvAddRedundandBits() Function

```
1   uintptr_t prvAddRedundantBits(uintptr_t ullData)
2   {
3       uintptr_t ullEncodedData = 0;
4
5       /** Tracks which data bit we are placing (starting at LSB). */
6       size_t ulDataBitIndex = 0;
7
8       /** Bit positions are counted starting from 1 as Hamming power of
         two positions require counting from 1 and not 0. This is taken
        into account when accessing the underlying bits with the correct
        position. */
9       for (size_t ulPos = 1; ulPos <= DATA_BITS + PARITY_BITS; ulPos++)
10      {
11          /** Check if the current position is reserved for a parity
        bit. */
12          if (prvIsPowerOfTwo1Indexed(ulPos))
13          {
14              /** Parity bit slot: leave as 0 for now. */
15              continue;
16          }
17          /** Otherwise, take the next data bit from 'ulData' and place
         it. */
18          bool bBit = (ullData >> ulDataBitIndex) & 1ULL;
19          if (bBit)
20          {
21              /** Take into account 1 based indexing when accessing the
         underlying bits. */
22              ullEncodedData |= (1ULL << (ulPos - 1));
23          }
24          ulDataBitIndex++;
25      }
26      return ullEncodedData;
27  }
```

prvComputeParity Iterates over each parity position $p = 1,2,4,8,16,32$, calls `prvComputeParityForBitPosition` to XOR the appropriate pattern of bits, and writes the result with `prvSetBit1Indexed`.

Listing 7.3: prvComputeParity() Function

```
1   uintptr_t prvComputeParity(uintptr_t ullEncoded)
```

```
{
    for (size_t ulP = 1; ulP <= DATA_BITS + PARITY_BITS; ulP *= 2)
    {
        if (ulP > DATA_BITS + PARITY_BITS)
        {
            break;
        }
        bool bParityBitValue = prvComputeParityForBitPosition(ullEncoded, ulP, DATA_BITS + PARITY_BITS);
        prvSetBit1Indexed(&ullEncoded, ulP, bParityBitValue);
    }
    return ullEncoded;
}
```

prvComputeParityForBitPosition computes the parity for a specific bit position.

Listing 7.4: prvComputeParityForBitPosition() Function

```
bool prvComputeParityForBitPosition(uintptr_t ullCodeWord, size_t ulParityBitPosition, size_t ulEncodedWordBitLength)
{
    bool bParity = 0UL;

    /** Starting at ulParityBitPosition, process a number of bits equal to the value of ulParityBitPosition, skip a number of bits equal to the value of ulParityBitPosition, etc... */
    for (size_t ulPos = ulParityBitPosition; ulPos <= ulEncodedWordBitLength; ulPos += 2 * ulParityBitPosition)
    {
        /** For the next positions represented by the value of ulParityBitPosition, XOR the bits to compute the parity. */
        for (size_t ulI = 0; ulI < ulParityBitPosition && (ulPos + ulI) <= ulEncodedWordBitLength; ulI++)
        {
            bool bBitVal = prvGetBit1Indexed(ullCodeWord, ulPos + ulI);
            bParity ^= bBitVal;
        }
    }
    return bParity;
}
```

Decoding

The entry point is:

Listing 7.5: Hamming Decode Function

```
uintptr_t _ullHammingDecode(uintptr_t ullCodeWord)
{
    uintptr_t ullCorrected = prvCorrectError(ullCodeWord);
    uintptr_t ullData = prvExtractData(ullCorrected);
    return ullData;
}
```

prvCorrectError Computes a syndrome by clearing each parity bit in turn, recomputing its parity, and comparing with the stored bit. Any mismatch adds that parity position to the syndrome. If the syndrome is nonzero, flips the bit at that position to correct a single-bit error.

Listing 7.6: prvCorrectError() Function

```
uintptr_t prvCorrectError(uintptr_t ullCodeWord)
{
    size_t ulSyndrome = 0UL;

    /** Loop over the parity bits to compute the syndrome. */
    for (size_t ulP = 1UL; ulP <= DATA_BITS + PARITY_BITS; ulP *= 2)
    {
        if (ulP > DATA_BITS + PARITY_BITS)
        {
            break;
        }

        /** Temporarily clear the bit at this parity position to
    ensure the parity calculation is done exactly as during the
    original encoding, taking into account the data bits and parity
    bits other than self. */
        uintptr_t ullTemp = ullCodeWord & ~(1ULL << (ulP - 1));
        size_t ulComputedParity = prvComputeParityForBitPosition(
    ullTemp, ulP, DATA_BITS + PARITY_BITS);
        size_t ulStoredParity = prvGetBit1Indexed(ullCodeWord, ulP);
        if (ulComputedParity != ulStoredParity)
        {
            ulSyndrome += ulP;
        }
    }
    if (ulSyndrome != 0)
    {
        vInjectorIoPrintDebug("ECC: detected error at bit %lu...
    correcting error!\n", ulSyndrome);
        /** Correct the error at the position indexed by the value of
     the syndrome. Take into account 1 indexed based counting. */
        ullCodeWord ^= (1ULL << (ulSyndrome - 1));
    }
    return ullCodeWord;
}
```

prvExtractData Reverses `prvAddRedundantBits` by scanning positions 1–(DATA_BITS+PARITY_BITS), skipping parity slots, and rebuilding the original 48-bit value in LSB order.

Listing 7.7: prvExtractData() Function

```
uintptr_t prvExtractData(uintptr_t ullCodeWord)
{
    uintptr_t ullData = 0ULL;
    size_t ulDataBitIndex = 0UL;
    for (size_t ulPos = 1UL; ulPos <= DATA_BITS + PARITY_BITS; ulPos
    ++)
    {
        if (prvIsPowerOfTwo1Indexed(ulPos))
        {
            continue;
        }
        bool bBitVal = prvGetBit1Indexed(ullCodeWord, ulPos);
        ullData |= ((uintptr_t)bBitVal << ulDataBitIndex);
        ulDataBitIndex++;
    }
    return ullData;
}
```

Public API

All ECC routines are declared in `hamming.h`. Two primary macros, `ullHammingEncode(ptr)` and `ullHammingDecode(cw)`, wrap the underlying functions with optional debug instrumentation and port-specific dispatch.

Listing 7.8: hamming.h header file

```
#ifndef FREERTOS_ECC_HAMMING_H

#define FREERTOS_ECC_HAMMING_H

#include <stdint.h>

#ifdef _WINDOWS_SOURCE_

#define __FILE_NAME__ (strrchr(__FILE__, '\\') ? strrchr(__FILE__, '
    \\') + 1 : __FILE__)

#endif /** _WINDOWS_SOURCE_ */

#ifdef DEBUG

```

```
15 /** @brief This function takes a 64 bit pointer representation and
       uses the unused bits of the pointer to compute a Hamming codeword
       containing a number of parity bits interleaved with the bits of
       the orignal data. @param ullData The integer value of the pointer
       on which to apply the Hamming encoding. @param pcFile Name of the
       file that called this function. @param pcFunc Name of the function
        that called this function. @param iLine Line number of the source
        file that called this function. @param pcVarName Name of the
       pointer being encoded. @return Returns the integer representation
       of encoded pointer. This pointer value is not suitable for
       dereferencing as it needs to be decoded first. */
16 uintptr_t _ullHammingEncode(uintptr_t ullData, const char *pcFile,
       const char *pcFunc, int iLine, const char *pcVarName);
17
18 /** @brief This function undoes what was done during vHammingEncode
       by reconstructing the original pointer value from the encoded
       pointer value. @param pvCodeWord The encoded pointer previously
       encoded by vHammingEncode(). @param pcFile Name of the file that
       called this function. @param pcFunc Name of the function that
       called this function. @param iLine Line number of the source file
       that called this function. @param pcVarName Name of the pointer
       being decoded. @return Returns the original pointer integer
       representation value. */
19 uintptr_t _ullHammingDecode(uintptr_t ullCodeWord, const char *pcFile
       , const char *pcFunc, int iLine, const char *pcVarName);
20
21 #else
22
23 /** @brief This function takes a 64 bit pointer representation and
       uses the unused bits of the pointer to compute a Hamming codeword
       containing a number of parity bits interleaved with the bits of
       the orignal data. @param ullData The integer value of the pointer
       on which to apply the Hamming encoding. @return Returns the
       integer representation of encoded pointer. This pointer value is
       not suitable for dereferencing as it needs to be decoded first. */
24 uintptr_t _ullHammingEncode(uintptr_t ullData);
25
26 /** @brief This function undoes what was done during vHammingEncode
       by reconstructing the original pointer value from the encoded
       pointer value. @param pvCodeWord The encoded pointer previously
       encoded by vHammingEncode(). @return Returns the original pointer
       integer representation value. */
27 uintptr_t _ullHammingDecode(uintptr_t ullCodeWord);
28
29 #endif /** DEBUG */
30
31 #ifdef _ECC_
32
33 #ifdef _POSIX_SOURCE_
```

```
34
35 #ifdef DEBUG
36
37 #define ullHammingEncode(ullData) ((typeof(ullData))_ullHammingEncode
       ((uintptr_t)ullData, __FILE_NAME__, __func__, __LINE__, #ullData))
38 #define ullHammingDecode(ullData) ((typeof(ullData))_ullHammingDecode
       ((uintptr_t)ullData, __FILE_NAME__, __func__, __LINE__, #ullData))
39
40 #else
41
42 #define ullHammingEncode(ullData) ((typeof(ullData))_ullHammingEncode
       ((uintptr_t)ullData))
43 #define ullHammingDecode(ullData) ((typeof(ullData))_ullHammingDecode
       ((uintptr_t)ullData))
44
45 #endif /** DEBUG */
46
47 #elif defined(_WINDOWS_SOURCE_)
48
49 #ifdef DEBUG
50
51 #define ullHammingEncode(ullData) ((__typeof__(ullData))
       _ullHammingEncode((uintptr_t)ullData, __FILE_NAME__, __func__,
       __LINE__, #ullData))
52 #define ullHammingDecode(ullData) ((__typeof__(ullData))
       _ullHammingDecode((uintptr_t)ullData, __FILE_NAME__, __func__,
       __LINE__, #ullData))
53
54 #else
55
56 #define ullHammingEncode(ullData) ((__typeof__(ullData))
       _ullHammingEncode((uintptr_t)ullData))
57 #define ullHammingDecode(ullData) ((__typeof__(ullData))
       _ullHammingDecode((uintptr_t)ullData))
58
59 #endif /** DEBUG */
60
61 #else /** _POSIX_SOURCE_ _WINDOWS_SOURCE_ */
62
63 #define ullHammingEncode(ullData) (ullData)
64 #define ullHammingDecode(ullData) (ullData)
65
66 #endif /* _POSIX_SOURCE_ _WINDOWS_SOURCE_ */
67
68 #else /** _ECC_ */
69
70 #define ullHammingEncode(ullData) (ullData)
71 #define ullHammingDecode(ullData) (ullData)
72
```

```
73 #endif /** _ECC_ */
74
75 #endif /** FREERTOS_ECC_HAMMING_H */
```

Protecting a *FreeRTOS* pointer with ECC requires encoding on every write and decoding on every read. In this study, the *FreeRTOS* source was manually patched to secure the chosen pointer-type injection targets: each write site invokes the ECC encoder, and each read site invokes the ECC decoder. Because the RTOS code frequently manipulates pointers in unconventional contexts, such as within macro expressions, it proved incompatible with the automated patching approach used for permanent-fault handling. A fully automated ECC patcher for pointer protection therefore remains a subject for future research.

7.8 Experimental Results with ECC

This section presents the fault-injection outcomes when Hamming-ECC is applied to selected FreeRTOS pointers. Before discussing the measurements, the list of protected pointers is recalled. These were chosen from the full set of injection targets (see 6.7) and instrumented with the ECC macros:

- `pxDelayedTaskList`
- `pxOverflowDelayedTaskList`
- `xIdleTaskHandle`
- `pxCurrentTCB`
- `pxCurrentTCB.pxTopOfStack`
- `pxCurrentTCB.pxStack`
- `pxCurrentTCB.pxTaskTag`
- `pxCurrentTimerList`
- `pxOverflowTimerList`

The following plots compare fault injection outcomes across the three workloads of the first scenario (see 6.5.2) and the TACLE benchmarks of the second scenario (see 6.5.2), first with no ECC protection and then with Hamming-ECC enabled for the specified pointer targets.

7.8.1 First Scenario

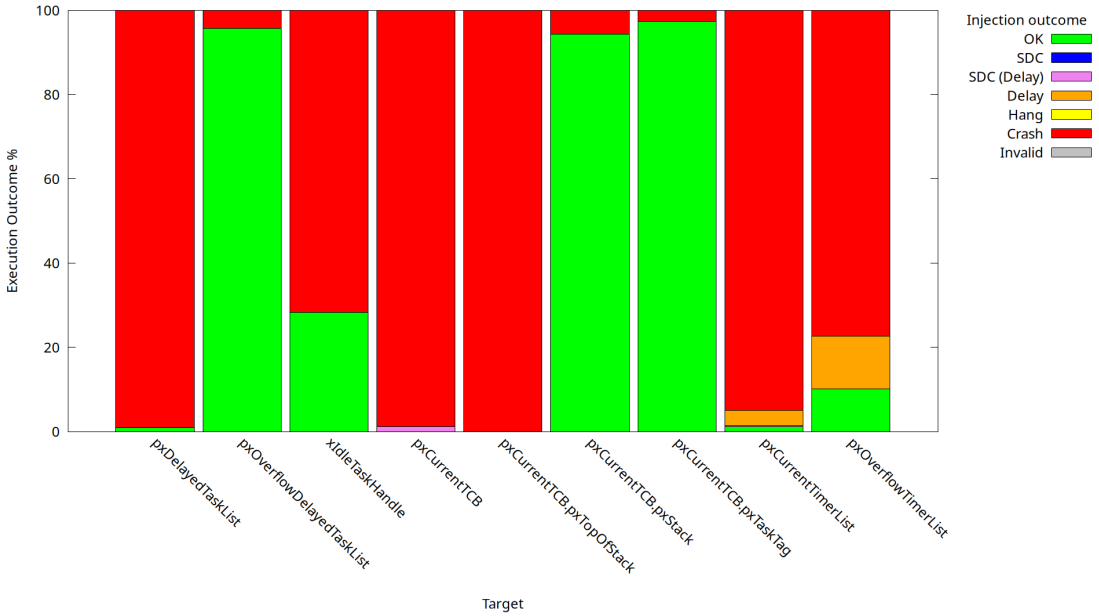

Figure 7.33: Injections on *FreeRTOS* **pointers NO ECC** (**Transient Faults**) *QSRT* items **1000** *TX* iterations **5** *Timer* iterations **5** *RX* iterations **10**

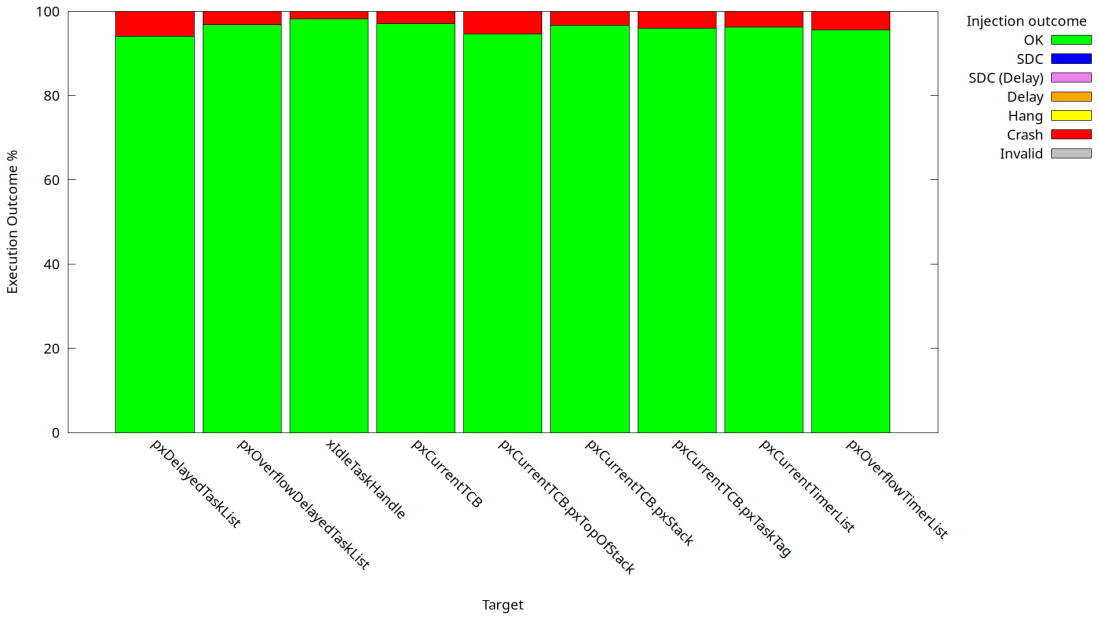

Figure 7.34: Injections on *FreeRTOS* **pointers with ECC** (**Transient Faults**) *QSRT* items **1000** *TX* iterations **5** *Timer* iterations **5** *RX* iterations **10**

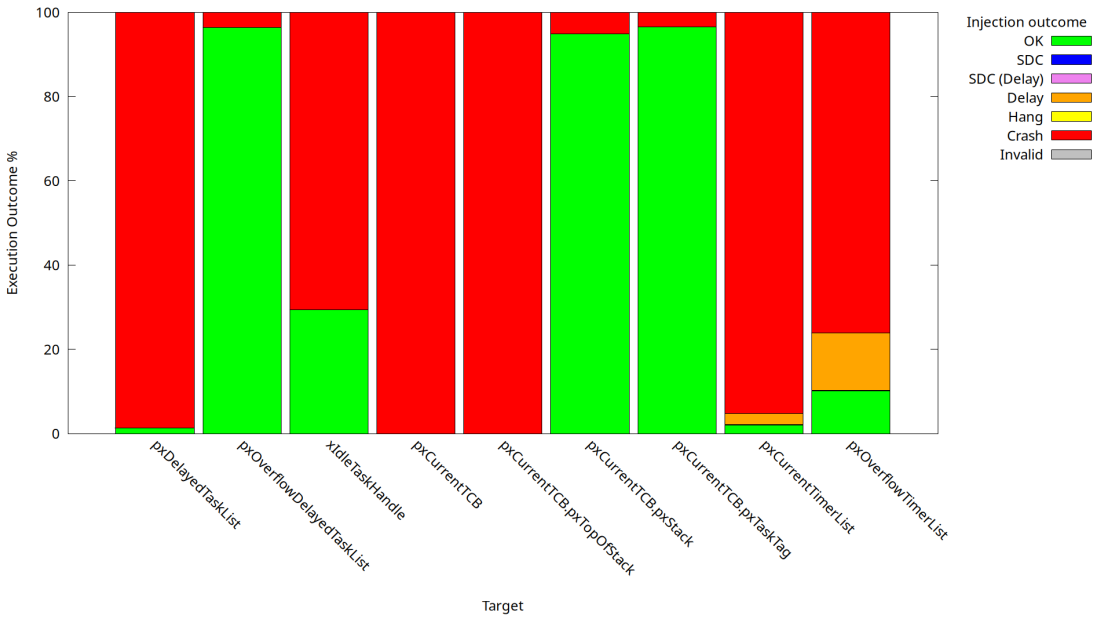

Figure 7.35: Injections on *FreeRTOS* **pointers NO ECC** (**Permanent Faults**) *QSRT* items **1000** *TX* iterations **5** *Timer* iterations **5** *RX* iterations **10**

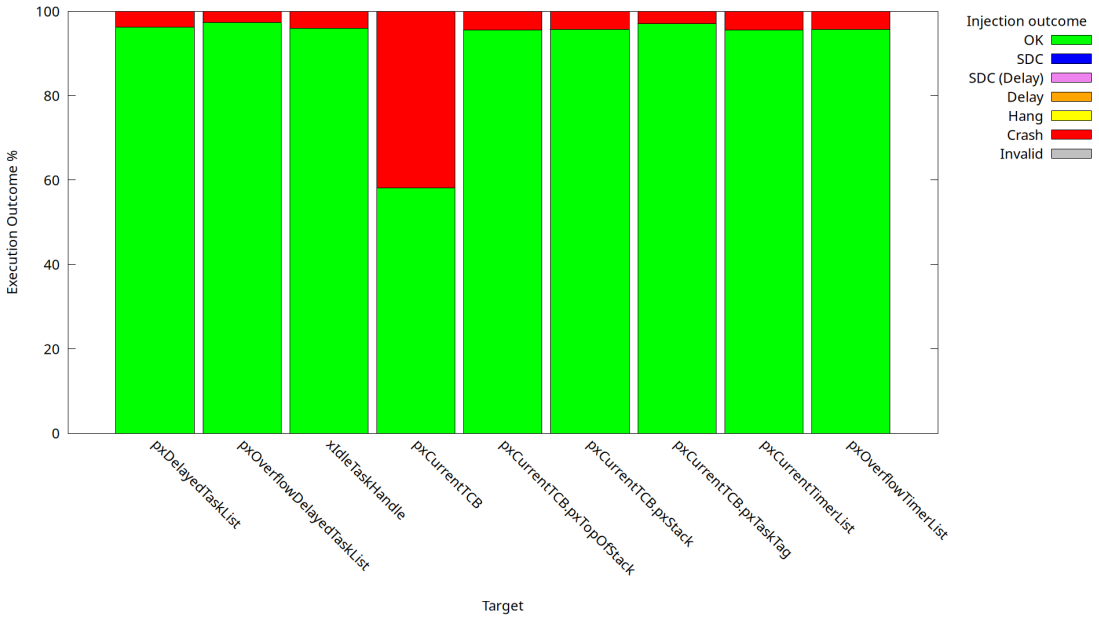

Figure 7.36: Injections on *FreeRTOS* **pointers with ECC** (**Permanent Faults**) *QSRT* items **1000** *TX* iterations **5** *Timer* iterations **5** *RX* iterations **10**

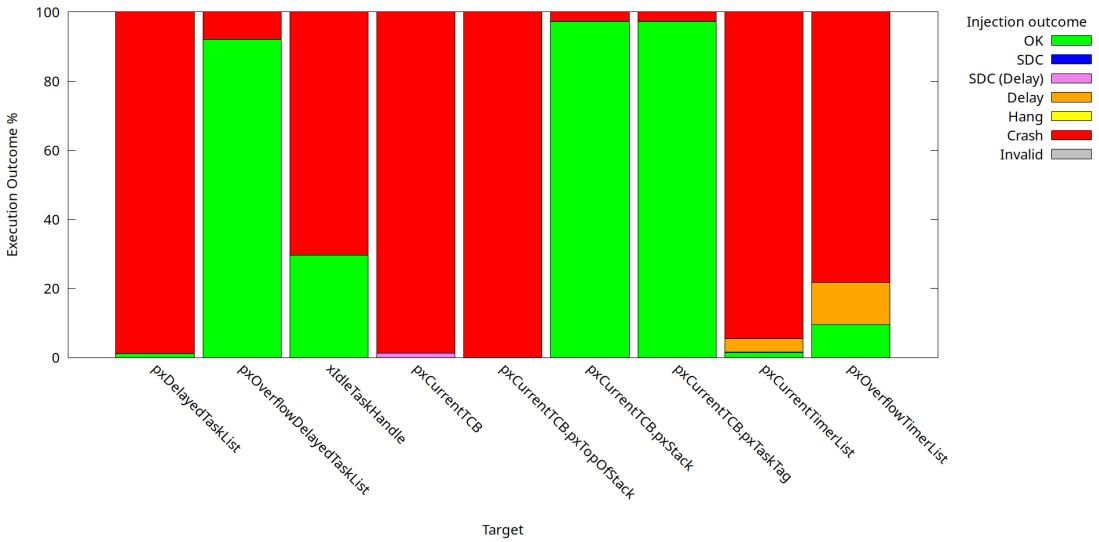

Figure 7.37: Injections on *FreeRTOS* **pointers NO ECC** (**Transient Faults**) *QSRT* items **5000** *TX* iterations **10** *Timer* iterations **10** *RX* iterations **20**

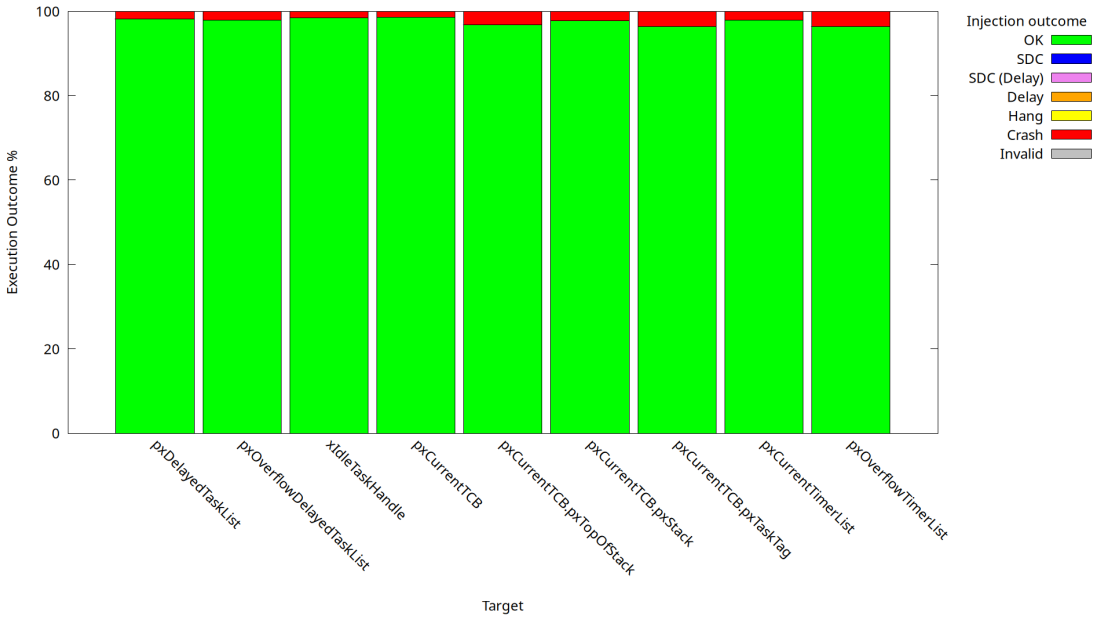

Figure 7.38: Injections on *FreeRTOS* **pointers with ECC** (**Transient Faults**) *QSRT* items **5000** *TX* iterations **10** *Timer* iterations **10** *RX* iterations **20**

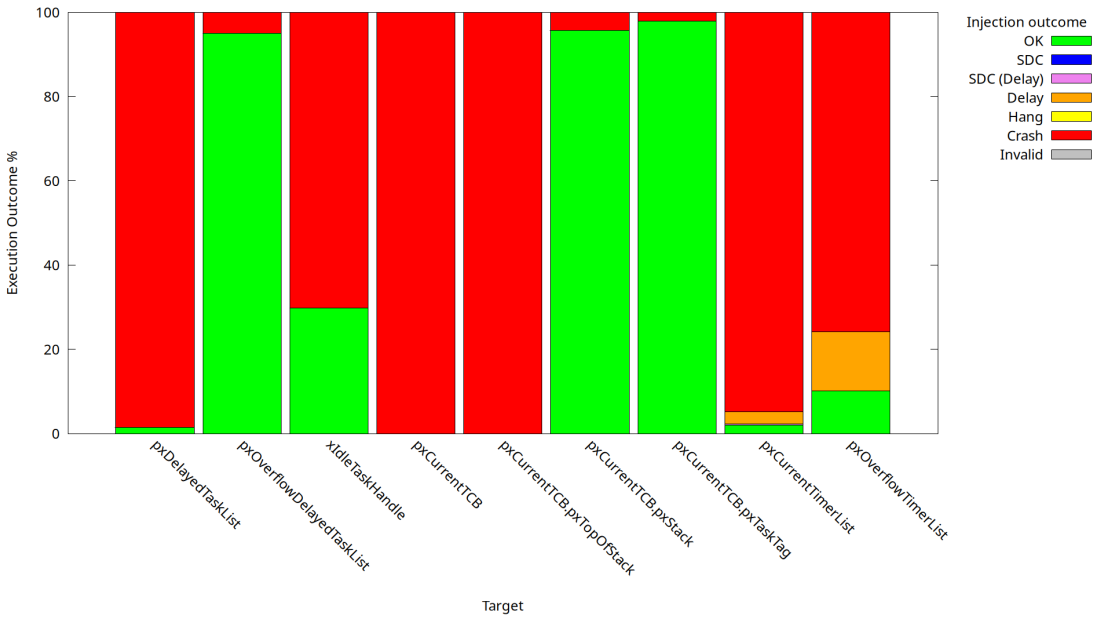

Figure 7.39: Injections on *FreeRTOS* **pointers NO ECC** (**Permanent Faults**) *QSRT* items **5000** *TX* iterations **10** *Timer* iterations **10** *RX* iterations **20**

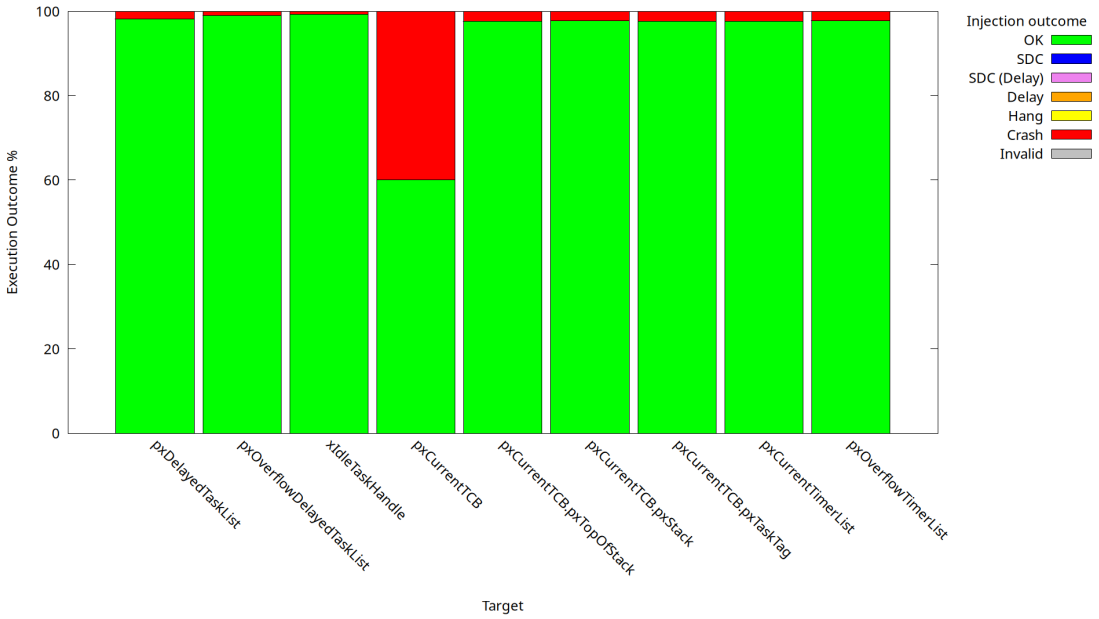

Figure 7.40: Injections on *FreeRTOS* **pointers with ECC** (**Permanent Faults**) *QSRT* items **5000** *TX* iterations **10** *Timer* iterations **10** *RX* iterations **20**

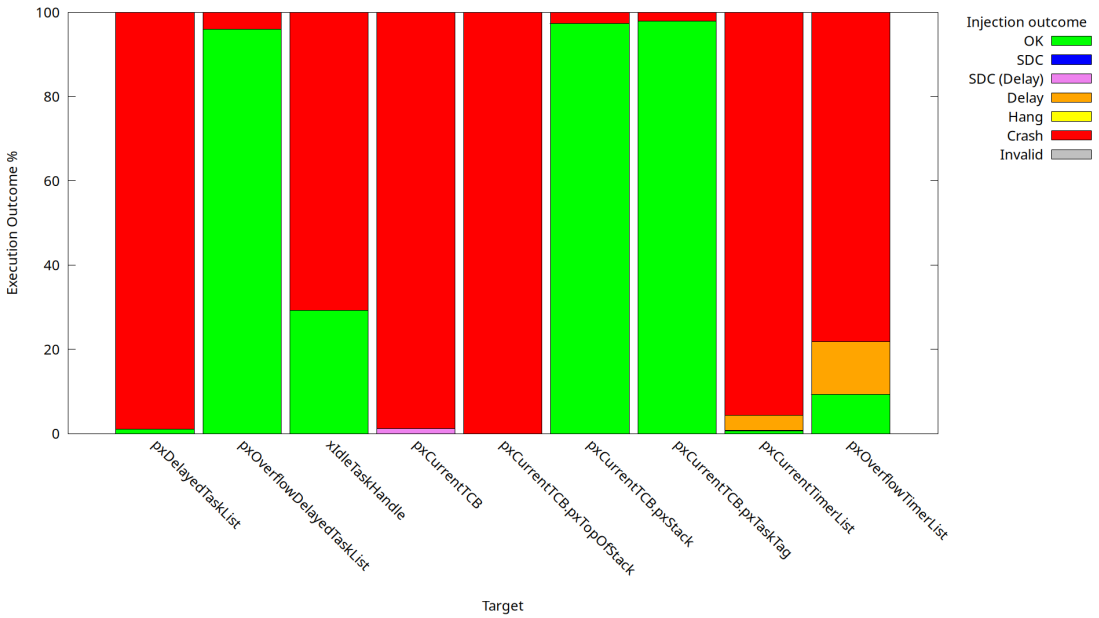

Figure 7.41: Injections on *FreeRTOS* **pointers NO ECC** (**Transient Faults**) *QSRT* items **10000** *TX* iterations **20** *Timer* iterations **20** *RX* iterations **40**

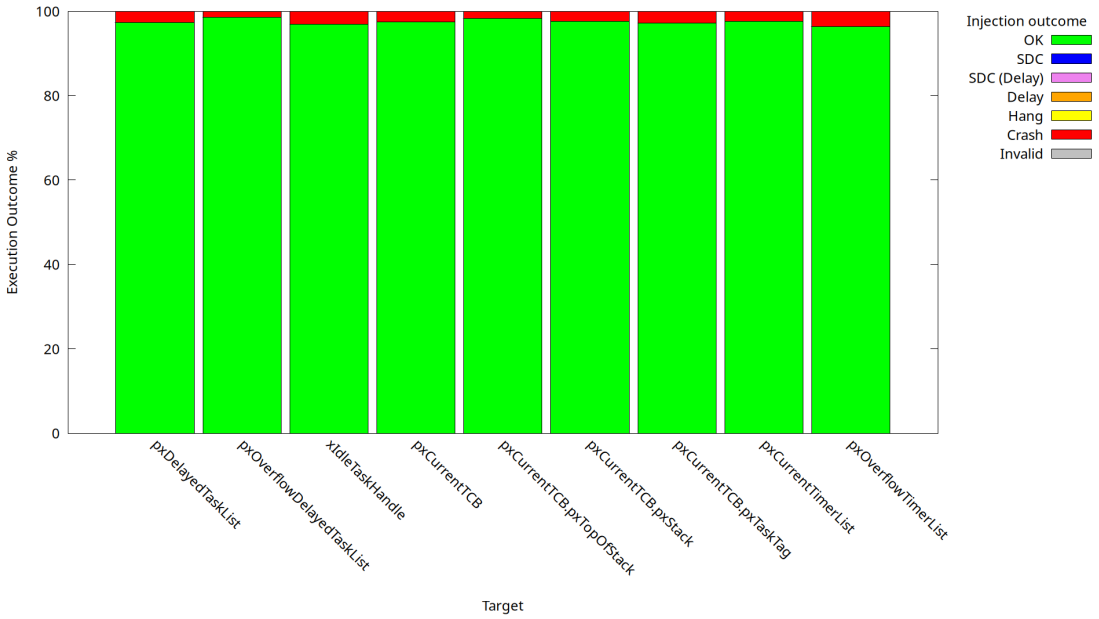

Figure 7.42: Injections on *FreeRTOS* **pointers with ECC** (**Transient Faults**) *QSRT* items **10000** *TX* iterations **20** *Timer* iterations **20** *RX* iterations **40**

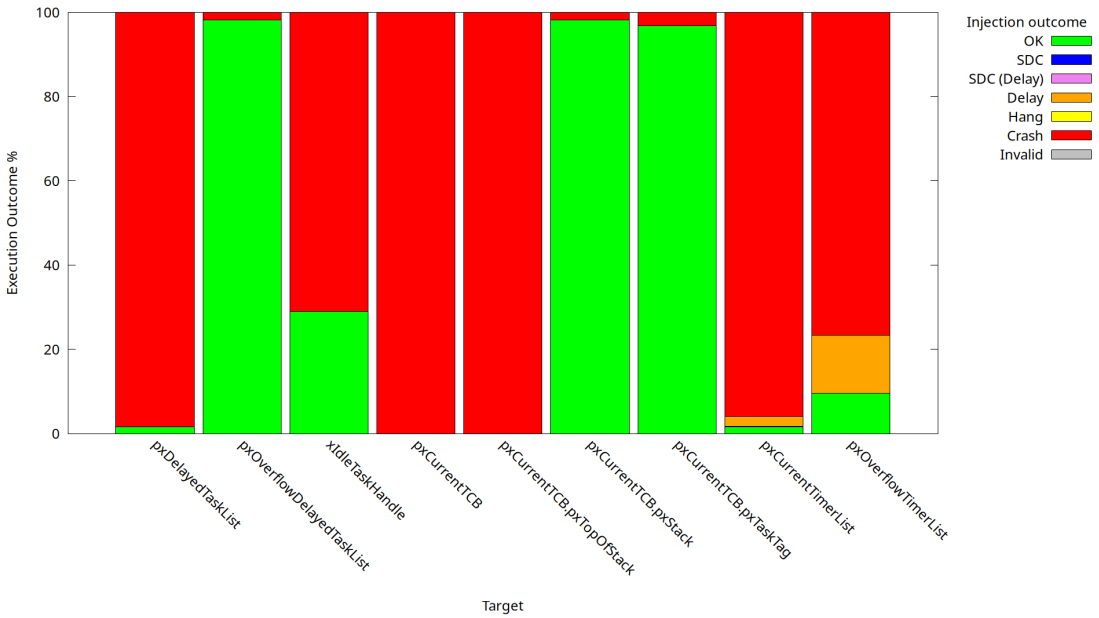

Figure 7.43: Injections on *FreeRTOS* **pointers NO ECC** (**Permanent Faults**) *QSRT* items **10000** *TX* iterations **20** *Timer* iterations **20** *RX* iterations **40**

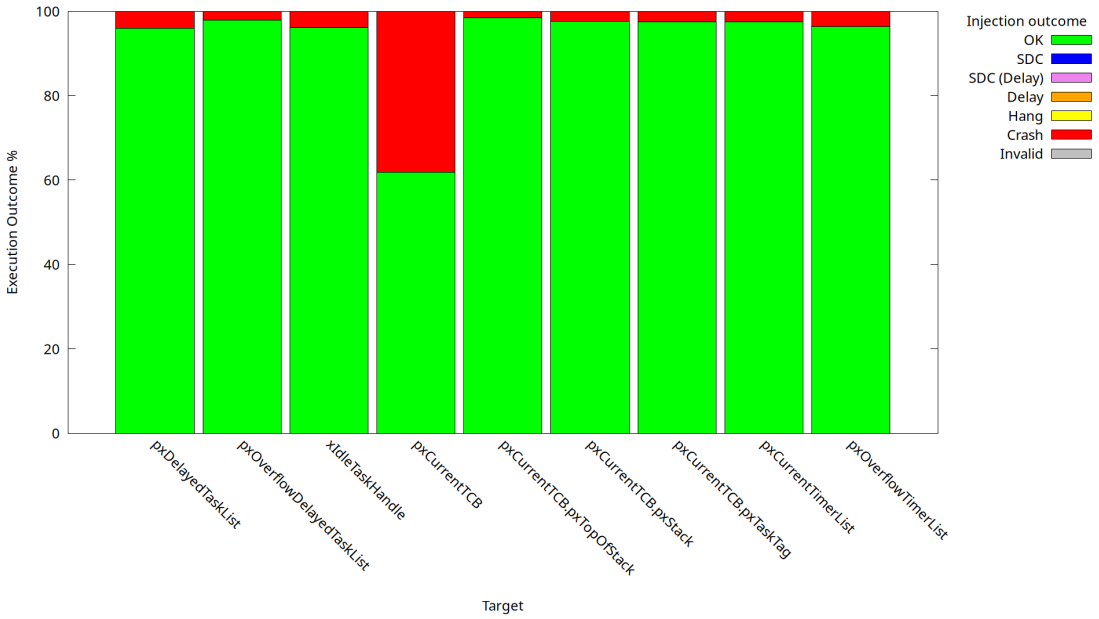

Figure 7.44: Injections on *FreeRTOS* **pointers with ECC** (**Permanent Faults**) *QSRT* items **10000** *TX* iterations **20** *Timer* iterations **20** *RX* iterations **40**

7.8.2 Second Scenario

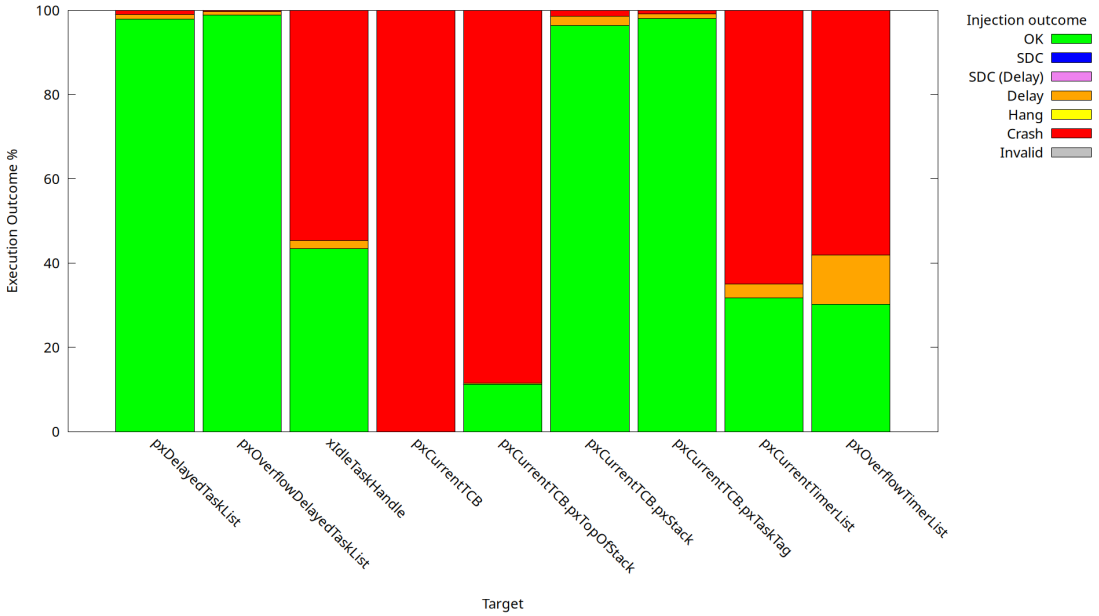

Figure 7.45: Injections on *FreeRTOS* **pointers NO ECC** (**Transient Faults**) Tacle Benchmarks: *SHA*, *FFT*, *CUBIC*, *HUFF_DEC*, *ADPCM_ENC*

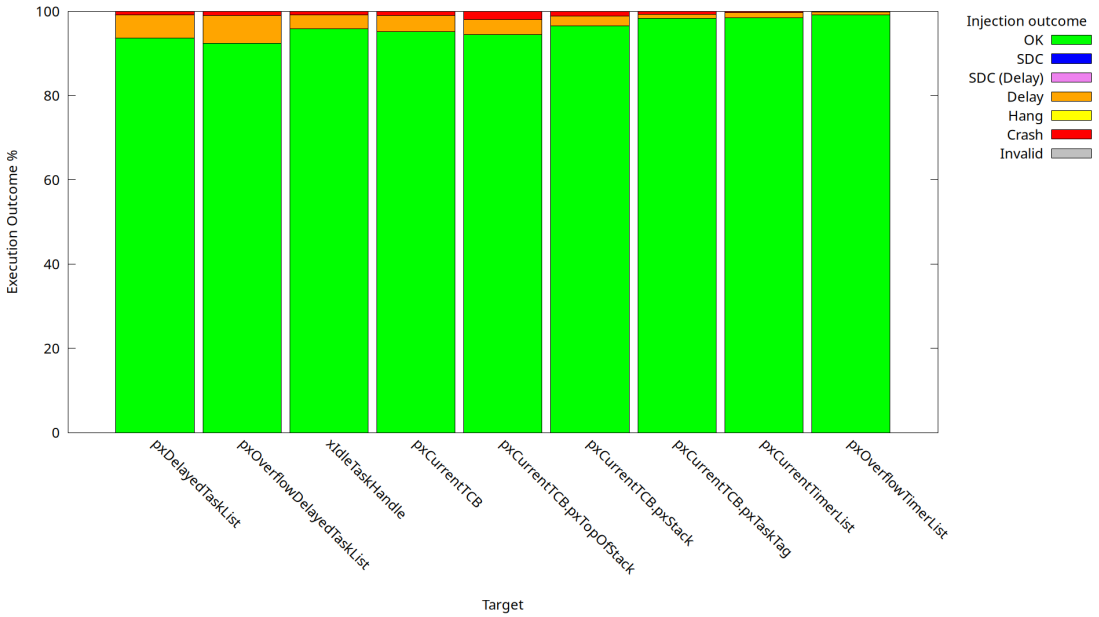

Figure 7.46: Injections on *FreeRTOS* **pointers with ECC** (**Transient Faults**) Tacle Benchmarks: *SHA*, *FFT*, *CUBIC*, *HUFF_DEC*, *ADPCM_ENC*

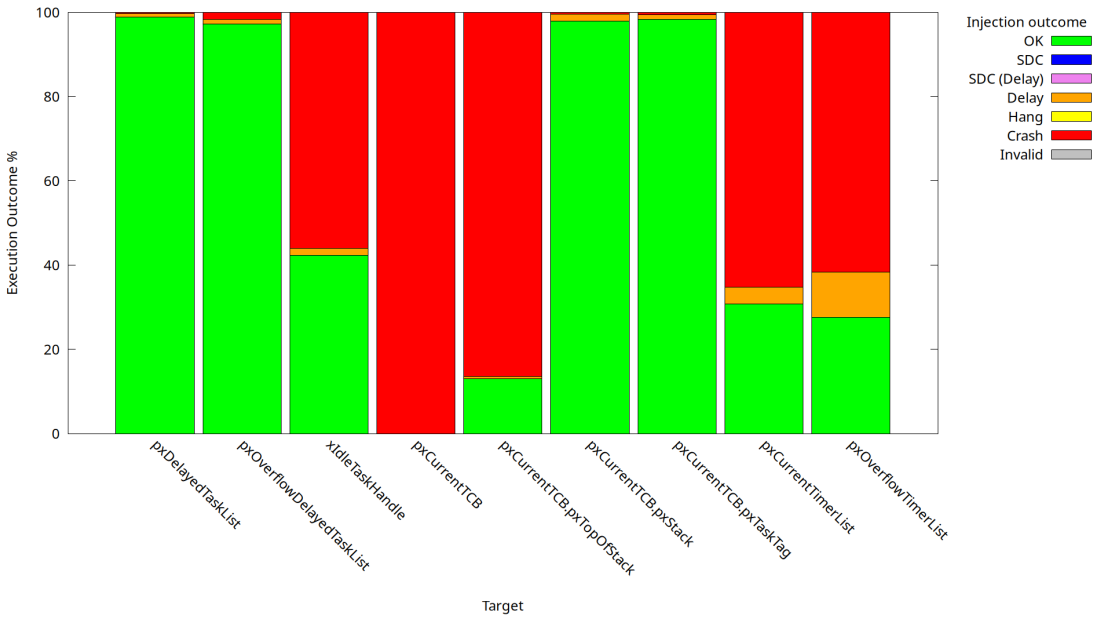

Figure 7.47: Injections on *FreeRTOS* **pointers NO ECC** (**Permanent Faults**) Tacle Benchmarks: *SHA*, *FFT*, *CUBIC*, *HUFF_DEC*, *ADPCM_ENC*

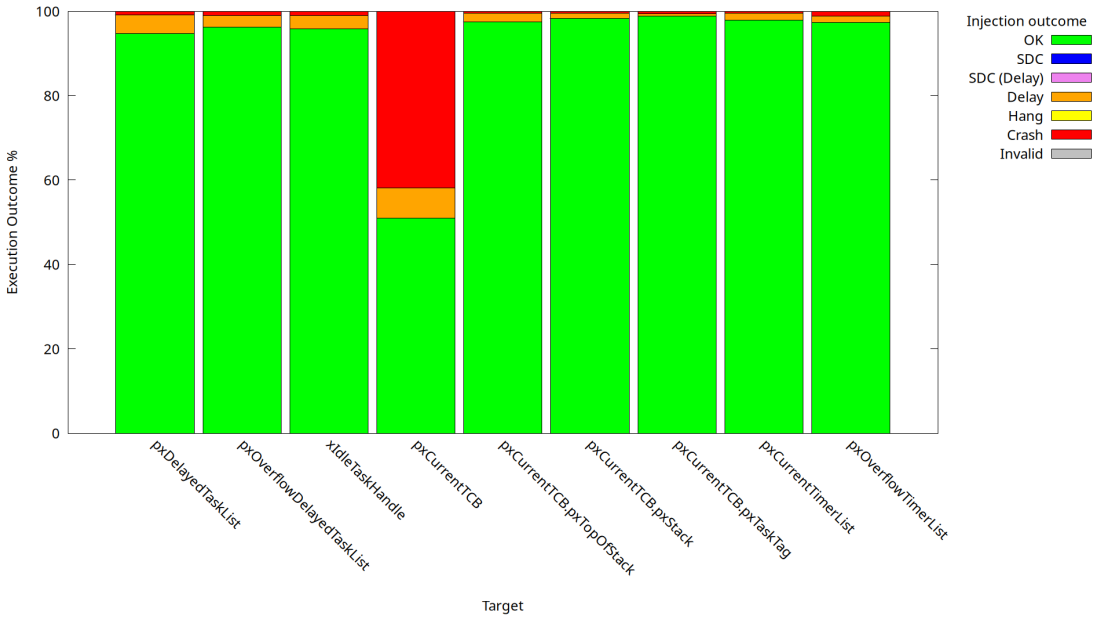

Figure 7.48: Injections on *FreeRTOS* **pointers with ECC** (**Permanent Faults**) Tacle Benchmarks: *SHA*, *FFT*, *CUBIC*, *HUFF_DEC*, *ADPCM_ENC*

7.8.3 Results Analysis

First Scenario

Across all three workloads in the first scenario, most failure modes, **SDC**, **SDC Delay**, **Delay**, and **Hang**, are eliminated by Hamming-ECC, leaving only a small residual **Crash** rate.

- `pxDelayedTaskList`: Before ECC, nearly 100% crashes. After ECC, crashes drop to about 6%, consistently across both transient and permanent faults and all workloads.
- `pxOverflowDelayedTaskList`: Originally marginally sensitive, with a few crashes; ECC reduces this already small crash fraction under both fault types.
- `xIdleTaskHandle`: Crash rate falls from 70% (pre-ECC) to 2–3% post-ECC, for both transient and permanent injections in all workloads.
- `pxCurrentTCB`:
  - Pre-ECC:
    - Transient faults: 97% crashes, 3% SDC Delay.
    - Permanent faults: 100% crashes.
  - Post-ECC:
    - SDC Delays vanish.
    - Transient crashes: 2–4%.
    - Permanent crashes: 45% (anomalous reduction due to ECC interaction with the permanent fault routine, faults injected after encoding within the first 54 codeword bits are corrected, whereas faults applied prior to encoding bypass correction and still crash).
- `pxCurrentTCB.pxTopOfStack`: Crashes fall from 100% (both fault types) to 1–4% after ECC, with a slight additional reduction at higher workloads.
- `pxCurrentTCB.pxStack`: Initial crash rate of 5% is reduced further by ECC, declining more as workload increases.
- `pxCurrentTCB.pxTaskTag`: Low sensitivity overall. Occasional slight increases in crash rate post-ECC, due to run-to-run variation in the hosted RTOS environment, are followed by consistent reductions in other workloads.
- `pxCurrentTimerList`:
  - Pre-ECC: 95% crashes plus small, workload-consistent fractions of **Delay** and **SDC Delay**.

  - Post-ECC: All Delay and SDC Delay outcomes vanish; crash rate falls to 4% under both fault types and all workloads.

- `pxOverflowTimerList`: Highly sensitive pre-ECC, with majority crashes and substantial delays. ECC eliminates all delays and reduces crashes to 5%.

Second Scenario

- `pxDelayedTaskList`: Sensitivity falls compared to the first scenario, likely because this workload invokes fewer blocking calls.
  - Pre-ECC: Mostly OK outcomes, with a small fraction of Delays. Permanent fault crashes are lower overall.
  - Post-ECC: Delay outcomes increase slightly under both fault types; crash rates remain low. This added Delay reflects the overhead of Hamming encoding/decoding in a faster-running benchmark (no long loops).
  - `pxOverflowDelayedTaskList`: Remains marginally sensitive.
    * Pre-ECC: Mostly OK, with a slight Delay fraction.
    * Post-ECC: Delay fraction rises modestly; crashes stay low.
  - `xIdleTaskHandle`: Retains high sensitivity.
    * Pre-ECC: Predominantly crashes, plus some Delays.
    * Post-ECC: Crash rate drops, but slight Delays persist under both fault types.
  - `pxCurrentTCB`: Sensitivity remains very high.
    * Pre-ECC: 100% crashes under both transient and permanent faults.
    * Post-ECC: Crashes fall to 2–3%, with 4% Delays.
  - `pxCurrentTCB.pxTopOfStack`: Continues to be highly sensitive.
    * Pre-ECC: Majority crashes, plus 10% OK and a few Delays (lower crash rate than Scenario 1).
    * Post-ECC: Crashes nearly disappear; a small Delay fraction remains.
  - `pxCurrentTCB.pxStack`: Remains low sensitivity.
    * Pre and Post-ECC: Mostly OK outcomes, with slight Delays under both fault types (absent in Scenario 1).
  - `pxCurrentTCB.pxTaskTag`: Similar to `pxStack`, but with an even smaller Delay fraction both pre and post-ECC.
  - `pxCurrentTimerList`: Sensitivity reduces from "very high" (Scenario 1) to "high."

    - Pre-ECC: Fewer crashes than in Scenario 1, with slightly more Delays.
    - Post-ECC: Crashes nearly vanish; slight Delays persist.
  - `pxOverflowTimerList`: Also down-rated from "very high" to "high" sensitivity.
    - Pre-ECC: Lower crash rate than Scenario 1; Delays remain consistent.
    - Post-ECC: Crashes mostly eliminated; minor Delays remain.

7.8.4 Overall Results Analysis

From the detailed overview of both benchmark scenarios, the Hamming code ECC proves highly effective at mitigating erroneous outcomes. In the first scenario:

- All outcome types except **Crash** vanish entirely.
- The remaining **Crash** rate is negligible, with the sole exception of modifications to `pxCurrentTCB`. This residual error stems from the interaction between permanent-fault injections and the implemented ECC.

In the second scenario:

- ECC again drives **Crash** outcomes down to very low levels.
- Compared to the first scenario, this benchmark exhibits increased sensitivity to the decoding/encoding overhead introduced by ECC.
- Most post-ECC executions complete as **OK**, with a small fraction suffering **Delay** or residual **Crash**.

In both scenarios these remaining crashes can be explained by the code protection choices. Hamming code protection was applied to only 48 data bits (the bits deemed significant on the test machine), adding 6 parity bits for a total code-word length of 54 bits. The unprotected 10 bits of each 64 bit pointer remain susceptible to random bit-flips injected by the *FreeRTOS Injector*, which can still provoke occasional **Crash** outcomes.

7.9 Conclusion

An experimental framework was established comprising two test scenarios: the first executing three tasks with a timer callback (see 7.3.1), and the second running five benchmarks from the TACLE suite (see 7.3.2). A statistical methodology determined the number of fault-injection campaigns required to achieve a 99% confidence level with a 5% error margin on the observed outcome distributions.

Baseline measurements, collected before any hardening, revealed fault-sensitive regions in the *FreeRTOS* kernel.
Multiple hardening strategies were then evaluated, with particular emphasis on a Hamming-code ECC implementation. The ECC layer was integrated into the kernel and assessed via a further series of fault-injection experiments. In every scenario, the ECC mechanism yielded a significant reduction in failure rates.
These results complete the validation cycle introduced in Chapter 1, demonstrating that a targeted, selective software hardening approach can substantially improve the fault resilience of *FreeRTOS*.

Chapter 8

Conclusion

Selective hardening of pointer variables in *FreeRTOS* via a Hamming code ECC scheme has virtually eliminated fault-induced failures at the protected addresses. The few residual errors arise from three principal sources:

1. Protection covering only the lower 54 bits of each 64-bit pointer.
2. Interference between the ECC routines and the permanent-fault injection mechanism.
3. Significant runtime overhead for ECC encoding/decoding under very high execution rates.

Each of these limitations admits a straightforward corrective action:

- Extend parity coverage to all 64 bits of each pointer.
- Improve the permanent fault implementation to avoid interfering with the ECC scheme.
- Accelerate or replace the current ECC implementation with a lower-latency alternative.

The underlying code base has also been refactored and enhanced:

- Monolithic structures were replaced by a modular sub-component architecture.
- Memory management bugs were identified and corrected.
- Inline documentation was expanded to improve maintainability.
- A "dry-run" mode now computes injection times and locations in advance of fault campaigns.

- Permanent-fault support was added to the Windows port.
- The permanent fault patcher compilation was integrated into CMake.
- Auxiliary scripts now automate compilation, patch deployment, and dependency checks.

Future work may include extending ECC protection to additional pointer-based structures (e.g. internal kernel lists) and exploring error-correction strategies for non-pointer data. Such an extension would require reserving auxiliary storage for parity bits and establishing a robust metadata-binding mechanism. Additional performance gains could arise from implementing alternative ECC schemes through the existing public API, thereby allowing multiple ECC variants to coexist within the same code base. Hybrid approaches, combining pointer ECC with data redundancy or check-pointing, also warrant investigation.
These enhancements are anticipated to further elevate the dependability of *FreeRTOS* and provide valuable insights for strengthening the resilience of embedded real-time operating systems more broadly.

Bibliography

[1] David Vrabel. *FreeRTOS port for Posix*. URL: https://www.freertos.org/Documentation/02-Kernel/03-Supported-devices/04-Demos/03-Emulation-and-simulation/Linux/FreeRTOS-simulator-for-Linux (cit. on pp. i, 3, 47, 51).

[2] FreeRTOS Community. *FreeRTOS port for Windows*. URL: https://www.freertos.org/Documentation/02-Kernel/03-Supported-devices/04-Demos/03-Emulation-and-simulation/Windows/FreeRTOS-Windows-Simulator-Emulator-for-Visual-Studio-and-Eclipse-MingW (cit. on pp. i, 3).

[3] Alfredo Benso and Stefano Di Carlo. «The Art of Fault Injection». In: *Control Engineering and Applied Informatics* 13.4 (2011), pp. 9–18. ISSN: 1454-8658 (cit. on pp. 2, 36).

[4] GNU. *GCC*. URL: https://gcc.gnu.org/ (cit. on p. 3).

[5] kitware. *CMake*. URL: https://cmake.org/ (cit. on p. 3).

[6] Arch Linux Community. *Arch Linux*. URL: https://archlinux.org/ (cit. on p. 3).

[7] Norsuzila Ya'acob, Akmarulnizam Zainudin, R. Magdugal, and Nani Fadzlina Naim. «Mitigation of space radiation effects on satellites at Low Earth Orbit (LEO)». eng. In: *2016 6th IEEE International Conference on Control System, Computing and Engineering (ICCSCE)*. IEEE, 2016, pp. 56–61. ISBN: 1509011781 (cit. on p. 5).

[8] John A. Stankovic and R. Rajkumar. «Real-Time Operating Systems». eng. In: *Real-time systems* 28.2/3 (2004), pp. 237–253. ISSN: 0922-6443 (cit. on p. 15).

[9] FreeRTOS Community. *Mastering the FreeRTOS Real Time Kernel*. URL: https://www.freertos.org/media/2018/161204_Mastering_the_FreeRTOS_Real_Time_Kernel-A_Hands-On_Tutorial_Guide.pdf (cit. on pp. 21, 44).

[10] FreeRTOS Community. *FreeRTOS Reference Manual*. URL: https://www.freertos.org/media/2018/FreeRTOS_Reference_Manual_V10.0.0.pdf (cit. on p. 21).

[11] Raj kamal Kaur, Babita Pandey, and Lalit Kumar Singh. «Dependability analysis of safety critical systems: Issues and challenges». eng. In: *Annals of nuclear energy* 120 (2018), pp. 127–154. ISSN: 0306-4549 (cit. on p. 31).

[12] Maha Kooli and Giorgio Di Natale. «A survey on simulation-based fault injection tools for complex systems». eng. In: *2014 9th IEEE International Conference on Design Technology of Integrated Systems in Nanoscale Era (DTIS)*. IEEE, 2014, pp. 1–6. ISBN: 1479949728 (cit. on pp. 31, 39).

[13] Dario Mamone, Alberto Bosio, Alessandro Savino, Said Hamdioui, and Maurizio Rebaudengo. «On the Analysis of Real-time Operating System Reliability in Embedded Systems». eng. In: *2020 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*. IEEE, 2020, pp. 1–6. ISBN: 1728194571 (cit. on p. 39).

[14] J.H. Barton, E.W. Czeck, Z.Z. Segall, and D.P. Siewiorek. «Fault injection experiments using FIAT». eng. In: *IEEE transactions on computers* 39.4 (1990), pp. 575–582. ISSN: 0018-9340 (cit. on p. 39).

[15] J. Arlat, J.-C. Fabre, and M. Rodriguez. «Dependability of COTS microkernel-based systems». eng. In: *IEEE transactions on computers* 51.2 (2002), pp. 138–163. ISSN: 0018-9340 (cit. on p. 39).

[16] Eunjin Jeong, Namgoo Lee, Jinhan Kim, Duseok Kang, and Soonhoi Ha. «FIFA: A Kernel-Level Fault Injection Framework for ARM-Based Embedded Linux System». eng. In: *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2017, pp. 23–34. ISBN: 9781509060313 (cit. on p. 40).

[17] D. Silva, K. Stangherlin, L. Bolzani, and F. Vargas. «A Hardware-Based Approach for Fault Detection in RTOS-Based Embedded Systems». eng. In: *2011 Sixteenth IEEE European Test Symposium*. IEEE, 2011, pp. 209–209. ISBN: 1457704838 (cit. on p. 40).

[18] Giovanni De Florio. *Permanent Fault Injection and Selective Hardening of Real Time Operating Systems*. Savino, Alessandro, 2023-07-28 (cit. on p. 42).

[19] PCRE2 Community. *PCRE2 Website*. URL: https://www.pcre.org/ (cit. on pp. 43, 47).

[20] PCRE2 Community. *PCRE2 Git Repository*. URL: https://github.com/PCRE2Project/pcre2/releases (cit. on p. 47).

[21] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert. «Statistical fault injection: Quantified error and confidence». eng. In: *2009 Design, Automation Test in Europe Conference Exhibition*. IEEE, 2009, pp. 502–506. ISBN: 9781424437818 (cit. on p. 82).

[22] Soyed Tuhin Ahmed, Surendra Hemaram, and Mehdi B. Tahoori. «NN-ECC: Embedding Error Correction Codes in Neural Network Weight Memories using Multi-task Learning». eng. In: *2024 IEEE 42nd VLSI Test Symposium (VTS)*. IEEE, 2024, pp. 1–7. ISBN: 9798350363784 (cit. on p. 111).