POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Permanent Fault Injection and Selective Hardening of Real Time Operating Systems

Supervisors

Prof. Alessandro Savino

Prof. Maurizio Rebaudengo

Candidate

Giovanni De Florio

July, 2023

Acknowledgements

I would like to thank Professor Alessandro Savino for allowing me to work on this research thesis and especially for the wonderful and important in-depth technical discussions I was able to have with him, who was available throughout the course of the thesis so that I could grow professionally along with the project.

The most important thanks go to my family, my parents, my sister, and my grandmother for always supporting me and allowing me to sustain this course of study, for living with me the happy moments and the difficult ones, for always believing in me unconditionally; the same thanks go to my girlfriend Rita, for being the beacon that has illuminated my path, teaching me day after day what commitment, sacrifice, to do what is right means.

I would also like to thank my friend Claudio with whom I have shared the challenges and satisfactions of this course of study and especially for the laughter.

Abstract

Real-Time Operating Systems (RTOS) are designed to provide deterministic response within predefined time limits. This implies the ability to reliably and predeterminedly manage the flow of events and operations in an electronic system.

Real-time Operating Systems are widely used in applications that require strict control of execution time, such as industrial control systems, medical equipment, autonomous vehicles, safety-critical systems and more. However, when the hardware on which the operating system is based is affected by ionizing radiation, such as during cosmic radiation events or failures in high-radiation environments, undesirable, even severe, effects may occur. Therefore, it becomes necessary to adopt a set of mitigation techniques at both the hardware and software levels.

Before these mitigation techniques can be applied, it is essential to conduct a thorough analysis of the behavior of real-time operating systems affected by radiation, also using empirical data.

For the thesis project, a permanent fault injection model was implemented to simulate the effects that irreversibly damage the electronic system. This model is completely software-based and requires no additional hardware other than that on which the injection environment is run. This offers the advantage of creating a test environment without the need for testing on physical systems.

Specifically, the goal of the research is to simulate permanent injections on the data structures of the FreeRTOS real-time operating system (regardless of version) and to develop fault mitigation techniques for that operating system.

Contents

Chapter 1

Introduction

For animal species, including humans, the surrounding environment has always presented a significant challenge. When we think of progress, we refer to the paradigms and means created by humans through intellect to overcome the challenges and limitations imposed by nature. One of the most current and representative examples of human progress is the computer, or more precisely, electronics and information technology, which find applications in various fields and significantly support each of our lives, directly or indirectly. Thus, progress in its various forms aims to address natural limits and mitigate their extremes. For example, consider human mortality: increasing the average lifespan of humans has always been a goal of medicine, and electronic systems play a leading role in supporting this objective, as seen in diagnostic machines (e.g., imaging diagnostics). Another significant example is the exploration of environments such as space, which were otherwise inaccessible but have been reached through the development of means in which electronic systems play a fundamental role.

1.1 Ionizing Radiation

Therefore, it is clear that electronic devices are bastions of progress, but their ability to withstand the hostile universe is an ongoing and progressive challenge between human intellect and the laws and composition of nature. A significant challenge for electronic systems, especially in space where it is particularly affected, is represented by ionizing radiation, which refers to radiation with sufficient energy to remove electrons from atoms or molecules, thus creating electrically charged ions. This type of radiation can be particularly harmful to electronic systems, even rendering them completely inoperable [1], a topic that will be further discussed in the next chapter.

1.1.1 Environment Sources

The main sources of ionizing radiation are:

- **Trapped protons and electrons in the Van Allen belts**
  [2] The Van Allen belts are two regions of charged particles that surround the Earth, forming a magnetic field around our planet. These belts are named after the physicist James Van Allen, who discovered them in 1958 during the early space flights with the Explorer satellites. The two main belts are the inner Van Allen belt and the outer Van Allen belt.

  The inner Van Allen belt extends approximately 1,000-5,000 kilometers above the Earth's surface. It mainly contains protons and high-energy particles called cosmic rays. This region is particularly hazardous for satellites and astronauts due to its high density of energetic particles.

  The outer Van Allen belt extends approximately 15,000-25,000 kilometers above the Earth and is primarily composed of high-energy electrons. This belt is influenced by interactions between the solar wind (a stream of particles emitted by the Sun) and the Earth's magnetic field. The electrons in the outer Van Allen belt can be dangerous for low Earth orbit satellites.

- **Heavy ions trapped in the magnetosphere**
  The Earth has a magnetic field generated by the molten metallic core of the planet. This magnetic field extends into space and forms the Earth's magnetosphere, which extends up to several tens of thousands of kilometers into space from the planet's surface.

  The magnetosphere traps heavy ions, such as iron or carbon. These ions can have a significant impact on space equipment due to their greater mass and penetration potential.

- **Cosmic protons and heavy ions**
  These particles come from sources such as supernova explosions. They can reach very high energies and pose a challenge for the protection of

space missions.

- **Protons and heavy ions originating from solar flares**
  Solar flares are violent eruptions on the surface of the Sun that release tremendous amounts of energy. During a solar flare, protons and heavy ions can be emitted at very high speeds.

1.1.2 Solar cycle

The levels of major natural sources of ionizing radiation are influenced by solar activity, and therefore by the solar cycle [3]. The solar cycle, also known as the magnetic solar activity cycle, sunspot cycle, or Schwabe cycle, is an almost periodic change of 11 years in the Sun's activity, measured in terms of variations in the number of sunspots observed on the surface of the Sun.
During the span of a solar cycle, the levels of solar radiation and solar material ejections, the number and size of sunspots, solar flares, and coronal loops all exhibit a synchronized fluctuation from a period of minimum activity to a period of maximum activity and back to a period of minimum activity.
The Sun's magnetic field undergoes a reversal during each solar cycle, with the reversal occurring when the solar cycle is near its maximum. After two solar cycles, the Sun's magnetic field returns to its original state, completing what is known as the Hale cycle.

Chapter 2

Single Event Effects

2.1 Introduction

Single event effects (SEE) are individual events that occur when a single incident ionizing particle deposits enough energy to cause an effect in a device. SEE are one of two phenomena that cause damage to electronic devices, the other is the total ionizing dose (TID) which instead of being a single event due to a single particle, is a long-term cumulative degradation of a device exposed to ionizing radiation[4].
The conditions of the devices affected by SEE depend on the incident particle and the specific device therefore different types of SEE can occur, which may not generate any observable consequence, or a simple transient bit flip as well as they could lead to the permanent fault of the device.
Therefore, the treatment of SEE and the analysis of the consequences arising from these plays a fundamental role when dealing with electronic systems, especially when dealing with ”mission critical” systems, i.e. systems that play a key role in the operation of a complex system or a vital mission.
”Mission critical” systems can involve a wide range of sectors such as aviation, defense, medicine, public safety and aerospace, and therefore for these systems reliability and safety are essential and any malfunction or interruption could have serious consequences, such as loss of human lives or significant damage.

2.2 Classification

It is essential to understand the differences between the different categories of SEE in order to correctly assess the level of risk and adopt appropriate measures to mitigate their effects.

In general, we can classify SEE into two categories: non-destructive and destructive[5].

2.2.1 Non-destructive

Single Event Upset (SEU) Single Event Upset (SEU) is a phenomenon that occurs in electronic circuits when a high-energy charged particle hits a memory cell or a logic component, causing a temporary disturbance in the state of the device. This disturbance can lead to an error or a change in the value of the data stored or processed by the device[6]. The effect of SEUs depends on the sensitivity of the hit device. Modern semiconductor technologies, such as CMOS (Complementary Metal-Oxide-Semiconductor) technology, are particularly vulnerable to SEUs. In CMOS devices, an SEU can cause the generation of free charge pairs that can induce a disturbance in the normal operation of the circuit. An example of a disturbance due to an SEU is a bit-flip or bit inversion. A bit flip occurs when an SEU causes a change in the value of a single bit within a data stored or processed by the device. For example, a "0" bit could be flipped to a "1" bit or vice versa. Bitflips can be problematic because they can cause errors in calculations or in data storage. If a bit needed for an operation is flipped, the final result might be wrong.

Single Hard Error (SHE) is an SEU which causes a permanent change to the operation of a device. An example is a permanent stuck bit in a memory device[7].

Single Event Functional Interrupt (SEFI) is a condition where the device stops normal functions, and usually requires a power reset to resume normal operations. It is a special case of SEU changing an internal control signal [8].

2.2.2 Destructive

Single Event Latch-up (SEL) When an energetic particle hits a critical point of the device, such as a transistor, it can cause a transient voltage and current. This transient can lead to the formation of an unwanted short circuit between the power supply and the system ground. The short circuit causes a very high current flow through the device, as the transistor behaves as an effective connection between the power supply and ground. This high current can cause the device to overheat, damaging or destroying internal components[9].

The components most affected by SEL are generally high-voltage bipolar transistors, such as bipolar junction transistors (BJT) and isolation bipolar transistors (BIT). These transistors are sensitive to the effects of high-energy particles and can undergo a short circuit between the emitter and base or between the collector and base.

Other components that can be affected by SEL include high-voltage MOSFETs (Metal-Oxide-Semiconductor Field-Effect Transistor), CMOS (Complementary Metal Oxide Semiconductor) devices, and high-power bipolar devices.

Latch-up, which is the undesired state in which the device remains locked at a high current even after the energetic particle has left the system, can occur due to positive feedback within the device. Positive feedback is a phenomenon in which the output signal of a system is fed back and amplified, contributing to further increasing the input signal. In other words, when positive feedback is present, a change in the input signal causes a response that further amplifies that change, creating a cycle of continuous amplification. Once latch-up occurs, the device may remain in a malfunctioning state until the latch-up current is interrupted. During latch-up, the system can experience a series of unwanted effects. The high current can cause the malfunction of circuit components, such as switches, amplifiers, microprocessors, or other integrated circuits. In some cases, the device may also be permanently damaged, making it unusable. It is important to note that the SEL phenomenon is often more frequent in high radiation environments, such as in space or in aeronautical applications. However, it can also occur in terrestrial environments, although with a lower probability.

Single Event Burnout (SEB) The fault of the Single Event Burnout (SEB) is caused by the high voltage or current that occurs in the device. This type of fault mainly affects high voltage transistors and power devices used in high power applications such as DC-DC converters, inverters, motor controllers, power supplies and other similar devices. When a radiation energy pulse hits the device, a sudden increase in the collector current can occur that exceeds the device's dissipation capacity. This can lead to local thermal fault, resulting in permanent damage or even destruction of the device itself. High-voltage and power devices are designed to handle high levels of voltage and current. However, in high radiation environments such as those in space or nuclear research, SEB represents a significant threat. It is important to emphasize that SEB can cause a rapid and destructive collapse of the device, with serious consequences. Furthermore, it is useful to distinguish SEB from Single Event Latchup (SEL). While SEB often leads to a rapid and destructive collapse of the device, SEL can cause damage to electronic circuits without completely destroying the device[10].

Single-Event Gate Rupture (SEGR) SEGR is caused by the interaction between ionizing particles and the gate oxide of a MOSFET. When an ionizing particle crosses the gate oxide, it creates electron-hole pairs, generating localized charges. These localized charges can accumulate near the gate oxide and create an electric field strong enough to break the oxide layer. The gate oxide is a critical component of a MOSFET as it serves as an insulating layer between the gate electrode and the channel region. When the oxide

layer breaks, the gate loses the ability to control the current flow through the channel, leading to malfunction or permanent damage to the device. SEGR can cause various fault mechanisms, including gate rupture (physical rupture of the gate oxide), gate leakage (decrease in gate control effectiveness due to the accumulation or dispersion of localized charges) or even catastrophic collapse.

Single Event Snapback (SESB) It is a phenomenon that occurs in some semiconductor devices, such as p-n junction diodes and MOSFETs (Metal Oxide Semiconductor Field Effect Transistor), when they are subject to a high-voltage or high-current transient event. During a transient event, such as an overvoltage or overcurrent, the voltage or current through the device exceeds nominal values. In response to this overload, the device enters a high-current conduction state. However, once the transient event ends and normal operating conditions are restored, the device should return to the off state, where the current through it is very low. The Single Event Snapback phenomenon occurs when, instead of returning to the off state, the device remains in a high-current conduction state. Single Event Snapback can lead to device malfunction, overheating or even its destruction.

Single Event Dielectric Rupture (SEDR) During a SEDR, the dielectric material is unable to handle the amount of energy applied and a rupture in its structure occurs. Dielectric materials are electrical insulators that are used to separate conductive components within an electronic device, providing electrical insulation and protection from short circuits. The rupture can manifest itself through a variety of mechanisms, such as the formation of an electric arc or the creation of a conductive channel through the dielectric. SEDR can lead to a system fault where the dielectric material is present.

Chapter 3

Real-Time Systems

3.1 Introduction

Any system that has a deterministic response to a given event can be considered ”real-time.” If a system is considered to fail when it doesn’t meet a timing requirement, it must be real-time. How failure is defined (and the consequences of a failed system) can vary widely. It is extremely important to realize that real-time requirements can vary widely, both in the speed of the timing requirement and also the severity of consequences if the required real-time deadlines are not met[11]. There are many different ways of achieving real-time behavior[12].

3.2 Approaches to Achieving Real Time

3.2.1 Hardware

Building specific hardware systems remains an important solution when there are precise requirements or needs for quick timings. This type of system can be implemented using discrete digital logic, analog components, or Application-Specific Integrated Circuits (ASICs). Options for programmable logic devices that can be used include Programmable Logic Devices (PLDs), Complex Programmable Logic Devices (CPLDs), and Field-Programmable Gate Arrays (FPGAs). A hardware-based real-time system can cover a wide range of applications, such as analog filters, closed-loop control, and simple state machines up to complex video codecs. When designed with a focus on power conservation, an ASIC can consume less power compared to a solution based on a microprogrammed control unit (MCU). However, there are also several disadvantages in developing real-time hardware systems. These include the lack of flexibility of non-programmable devices, difficulty in finding experts with specific hardware skills (compared to software/firmware developers), and the high cost of devices such as developing a custom ASIC[13].

3.2.2 Bare-Metal Firmware

Bare-metal firmware refers to software that runs directly on the hardware without a complete operating system. In this approach, the programmer writes code directly to control hardware resources, such as registers, timers, interrupts, etc. This gives complete and deterministic control over the system. Bare-metal firmware is particularly suited when there are a few relatively simple tasks to perform or a single, monolithic task. If the firmware is designed in a focused manner and best practices are followed, deterministic performance can be easily measured and guaranteed thanks to the reduced number of interrupt interactions. In some extreme cases, when working with microcontrollers heavily constrained in terms of memory space, bare-metal firmware might be the most viable solution. However, when bare-metal firmware becomes more complex and needs to handle events that occur asynchronously, it can become challenging to ensure proper synchronization and management of such events. In these cases, it starts to be advantageous to consider the use of a Real-Time Operating System (RTOS) to ensure the correctness and reliability of the system.

3.2.3 Operating Systems

Operating Systems (OS) provide a level of abstraction between the hardware and user programs, offering services such as memory management, process management, I/O device management, and implementation of security policies. A Real-Time Operating System (RTOS) is a special kind of operating system designed for executing applications with stringent timing requirements. Unlike general-purpose operating systems, an RTOS is able to guarantee that certain processes are run within a specific time, making them particularly useful for time-critical applications. RTOSs typically run on boards with a Memory Management Unit (MMU) that manages the access to physical memory by running programs. This allows the operating system to isolate the execution of various processes, preventing one process from interfering with the memory used by another process, thus increasing the robustness of the system. The use of an RTOS has various advantages over the use of bare-metal firmware, especially when dealing with complex systems with multiple asynchronous processes. However, using an RTOS also introduces an overhead, both in terms of memory resources needed to run the operating system and in terms of CPU time needed to handle the operating system services. It is important to note that the choice of using an RTOS or bare-metal firmware largely depends on the specifics of the project, timing, performance requirements, and the complexity of the system to be implemented.

3.3 FreeRTOS

FreeRTOS is an open-source real-time operating system designed for embedded systems.
There are several advantages to using FreeRTOS, including:

1. **Reliability**: FreeRTOS is known for its stability and reliability. It has been widely used in a range of real-time critical applications, such as medical devices, automobiles, industrial control systems, and many more. Its rigorous design and lightweight architecture allow reliable execution of real-time tasks and processes without compromising the system's overall performance.

2. **Compact Size**: FreeRTOS is designed to be highly resource-efficient. The binary image of FreeRTOS is usually very small and requires a minimal amount of system memory. This makes it suitable for resource-limited devices, like microcontrollers and microprocessors with little memory space.

3. **Scalability**: FreeRTOS is highly scalable and can be used in systems with varying levels of complexity. It can be configured to execute priority tasks deterministically, allowing efficient management of real-time constraints. It also supports both preemptive and cooperative scheduling, allowing designers to adapt the system's behavior to the specific needs of the application.

4. **Wide Range of Features**: FreeRTOS offers a range of useful features, such as task and queue management, event management, semaphores, mutexes, and more. These features simplify the development of complex applications and allow for better code organization.

5. **Active Community and Support**: FreeRTOS has a very active developer and user community that provides support, answers questions, and contributes to system improvement. There are discussion forums, detailed documentation, and a wide range of examples available to help developers quickly start and resolve any issues they might encounter during development.

In summary, using FreeRTOS offers reliability, resource efficiency, scalability, and a wide range of features, making it a popular choice for developing real-time embedded systems.

3.3.1 Characteristics of the Operating System

1. **Task Scheduler**

   The task scheduler is a fundamental component of the FreeRTOS kernel. It is responsible for scheduling and executing tasks based on their priority and system needs. FreeRTOS supports both preemptive and cooperative scheduling, allowing for great flexibility in task management. In preemptive mode, the highest priority task that is ready for execution is always run. This means that if a higher priority task becomes ready for execution, it will immediately interrupt the execution of the current task. In cooperative mode, on the other hand, a task continues to run until it explicitly yields control. This mode requires careful task design to avoid conditions where a lower priority task might block a higher priority one[14].

2. **Tick Interrupt**

   FreeRTOS uses a tick interrupt to track the passage of time. The tick interrupt is a hardware timer that generates an interrupt at regular intervals, typically every 1 millisecond. This tick interrupt is used to update task lists (like the Delayed Task List), track task execution time, and trigger processor reallocation by the scheduler.

3. **Memory Management**

   FreeRTOS offers several options for memory management, including dynamic and static memory management. In addition, FreeRTOS includes a memory allocator that supports fragmentation and allows developers to implement their own memory management logic if necessary. Developers can also choose between different allocation schemes, like First Fit, Best Fit, or

Worst Fit, depending on their specific needs.

4. **Task Control Block (TCB)**

   Each task in FreeRTOS is associated with a Task Control Block (TCB). The TCB contains important information about the task's state, such as priority, status (ready, blocked, or suspended), the stack pointer, and other control information. The TCB is crucial for the task scheduler to determine which task should be run next.

5. **Ready List and Delayed Task List**

   FreeRTOS maintains two main lists for task management: the Ready List and the Delayed Task List. The Ready List contains tasks that are ready for execution, while the Delayed Task List contains tasks that are waiting or blocked. These lists allow the scheduler to plan task execution efficiently.

6. **Queue Management**

   FreeRTOS provides queue management functionality for passing data and signals between tasks. Queues are used to implement inter-task communication mechanisms and to synchronize task execution. Each queue has an associated buffer for data storage and a set of pointers for controlling access to the data.

7. **Synchronization Primitives**

   Synchronization primitives, such as semaphores and mutexes, are provided by FreeRTOS to coordinate task execution. These mechanisms are crucial for avoiding race conditions and ensuring the correct execution of tasks in multi-thread environments.

8. **Interrupt Management**

   FreeRTOS generally has good support for interrupt management, allowing developers to leverage the power of hardware interrupts to respond promptly to external events. Interrupts can be used to awaken waiting tasks and to trigger the execution of specific functions.

9. **System Calls**

   FreeRTOS provides a set of system calls to interact with the kernel and manage tasks, queues, semaphores, interrupts, and other system features. These system calls offer a high-level programming interface that simplifies the development of complex applications[15].

3.3.2 Simulator

Within the FreeRTOS archive, you can find a port for POSIX and Windows, but for our purposes, we will focus on the POSIX one as we will be dealing with running FreeRTOS in a Linux environment.
Through this porting, also known as a simulator, it will be possible to experiment with FreeRTOS and develop applications, provided an approximation of real-time behaviors, because the hardware interrupts will be simulated and imprecise and rely on the underlying operating system (Linux in our case).
The POSIX porting layer can be found in the folder:
`FreeRTOS/Source/portable/ThirdParty/GCC/Posix`
It consists of **port.c**, which is the heart of the simulator, **portmacro.h**, and the **utils** folder, which in turn contains **wait_for_event.c\\.h**. One of the main points to consider in the porting system is task management. FreeRTOS is a real-time operating system based on tasks. These tasks need to be executed in some way in the POSIX environment. This is done by mapping each FreeRTOS task onto a POSIX thread. Furthermore, in the porting system, interrupt management is very important. When porting FreeRTOS to Linux, these interrupts are mapped onto POSIX signals. For instance, the timer interrupt might be mapped onto a SIGALRM signal in Linux. Another important aspect of porting is managing interaction with the standard C library. Some functions from this library can cause synchronization issues or deadlock if not handled correctly. This is a particular problem in the context of a multitasking system like FreeRTOS, where multiple tasks might attempt to access the same resource simultaneously. To handle this, the porting must ensure that functions like printf are used by only one task at a time, hence access to these functions is serialized.

Chapter 4

Fault Injection

In previous chapters, we understood that one of the most significant challenges concerning electronic systems exposed to ionizing radiation is single event effects. Furthermore, we also learned that Real Time Operating Systems (RTOS) are widely used when dealing with electronic systems. Therefore, it becomes necessary to mitigate the effects and, most importantly, the damages caused by these single events.

Before thinking about effective mitigation, however, we need to take an additional step, i.e., conducting an analysis supported by experimental verifications on the effects caused by SEE on electronic systems. This allows us to intervene both on the hardware and software side to strengthen the systems, making them fault-tolerant.

The most effective way is undoubtedly to expose a certain number of test electronic systems in the same hostile natural environments in which the systems we are interested in protecting are employed. Clearly, this implies enormous costs, as it would involve sending several systems into orbit for testing purposes, which would be prohibitively expensive. A less costly, but still expensive alternative, is to conduct tests in experimental environments with machines that artificially produce ionizing radiation.

There are several cost limitations to these approaches, not only in terms of economic cost but also in terms of the required energy, etc.

Therefore, what we explore in this thesis is a purely software solution that simulates the behavior of a real-time operating system working on a device hit by SEE. This allows us to perform analyses supported by various experimental tests without the aid of any additional device other than the computer on which we run the simulation environment.

4.1 Preliminary Concepts

4.1.1 Definitions

- **Fault:** A fault represents a defect or a failure in the system. A fault can be caused by human errors, hardware failures, external interferences, or other factors[16]. For example, a fault can be a defective hardware component.

- **Failure:** A failure represents the observable effect produced by a fault in the system[16]. A failure can manifest as a system crash, unexpected behavior, error output, or malfunction detectable by the end-user. For example, a failure can be an application that suddenly crashes or a system that no longer responds to user requests.

- **Error:** An error represents the human action or process that leads to the introduction of a fault in the system[16]. Errors can be made during the design, implementation, testing, or maintenance of the system. Errors can include wrong design choices, programming errors, omissions, entering incorrect data, or any other human action that leads to a fault. For example, an error can be a programmer who introduces a bug in the code or an operator who enters incorrect data into the system.

4.2 Environment

The following discusses the starting project which was the basis on which I developed this thesis.

4.2.1 The Orchestrator Process

The project I started from is based on the FreeRTOS simulator in a Linux environment running parallel to an injector thread that simulates the effects on the operating system of an SEU, particularly on the FreeRTOS data structures.
The project allows for the creation of multiple simulations, each starting from a process (and this aspect can be used to parallelize the injections).
The simulation starts from a main process, called the orchestrator, which will give rise to as many children as there are injections to manage within an injection campaign.

Each child process, in turn, will give rise to: threads for the FreeRTOS instance (the FreeRTOS simulator uses multiple concurrent threads but not in parallel, because each POSIX thread is used to manage the context of a FreeRTOS task) and an injector thread for the bit flip on the data structures. The use of multiple threads within the same process means that they share the addressing space and consequently that the injector thread has access to the FreeRTOS instance data structures launched by that process.

4.2.2 The Golden Execution

To determine whether a fault injection has caused a failure in the execution of a task launched on simulated FreeRTOS, it could be sufficient to observe the output produced at the end of the execution of the FreeRTOS instance and see if it corresponds to the desired output. But if we want to do an analysis on the errors and failures induced by the injection then it will be necessary to compare data (such as execution time) between the system execution with injection and a clean execution i.e., free of injections and therefore of faults, the Golden execution.
This is because, most likely, since the injections are launched on the data structures of the FreeRTOS kernel and not on the local data of the tasks, the errors usually do not cause variations in the output of the task as much as, for example, in the times to be respected (remember that this is a fundamental issue for a real-time operating system).
Therefore, the base project contains a command to start a golden execution, and at the end of this, create a file in which the execution time in nanoseconds is written in the first line (i.e., the time between the launch of the FreeRTOS instance and the moment it ends following the production of the output by the tasks) and in the following lines, the output generated by the tasks is reported.

4.2.3 Target extraction

The recovery of the targets to inject takes place during the execution of the simulator through a set of functions that have been added to the FreeRTOS kernel; it is important to stress the fact that the base project was therefore based on a modified FreeRTOS kernel, not a virgin one precisely because one of the objectives achieved in my thesis is to provide a tool where one does not have to worry about manually modifying the FreeRTOS kernel on which one

wants to run the tests, but on the contrary, a user of the injection environment should simply load a FreeRTOS kernel of choice into the project folder.
Returning to the functioning of target extraction, functions of the modified kernel are used to fill a globally accessible list (and therefore to the injector thread) with information about the targets (through a target_t structure).
For each target, the target_t structure stores details such as the name, address, size, and type of the target itself. These data are used during the injection phase to calculate the memory address of the target.
It is important to note that most of FreeRTOS's data structures are declared as static, which means they cannot be referenced directly. Therefore, through the developed functions, it is possible to read and retrieve the injection targets using the globally accessible list. This approach allows access to information about the injection targets at the start of the execution, allowing them to be listed and the final injection addresses for each target to be calculated.

Chapter 5

Permanent Fault Injection Model

5.1 Introduction

The objective of this thesis focuses on complementing the simulator discussed in the previous sections with a model of permanent fault injection, i.e., simulating a Single Event Effect (SEE) with permanent effects such as Single Hard Error (SHE), which ”burns” memory bits used by kernel data structures, preventing further modifications of these throughout the execution of a FreeRTOS instance. Clearly, this involves additional considerations compared to the implementation of a transient fault model, as in the transient fault model, once a bit of a data structure is modified, it is legitimate within the context of an injection simulation for the bit to be modified again by the operating system. The same cannot be said if we want to simulate the behavior of a bit subject to permanent failure and therefore should not change its value for an entire execution of a FreeRTOS instance.

5.2 Clauses

A concept that has accompanied the development of this project from its foundations is versatility. Indeed, creating a model that works on a modified operating system could be an end in itself. We must not forget that the goal is to have a simulated fault injection that produces effects on the simulated operating system, similar to those that would be obtained in the presence of hardware failure. Therefore, the environment must be able to work smoothly

on different versions of the kernel in order to cover a broader range of electronic systems on board different versions of the operating system.
This leads to other considerations, namely the number and degree of modifications that can be tolerated in the design phase for the project to be appreciable. To be more explicit, it might be more interesting to have a larger number of additional lines of code in the kernel that involve a low overhead cost, rather than adding a smaller number of lines of code but having a higher overhead cost. However, an opposite approach might also be preferred.
The advantages and disadvantages of the two policies are as follows:
In the case of the insertion of a substantial number of lines but with lower overhead, we would have a simulation closer to a real fault situation (which affects an original system) precisely because the changes would not be very costly in performance terms. This would not make, from a performance point of view, a custom system that implements policies to simulate the permanence of a bit-stuck, very different from an original one.
On the other hand, adding a substantial number of lines would limit the possibility of the fault simulator being exported for other kernel versions that are not those for which it was developed. This is due to the fact that it would mean applying from scratch all the necessary modifications for that system to simulate faults.
The problem expressed does not arise with the permanent fault model. Indeed, even the Single Event Upsets (SEUs) already implemented in the base project require, as explained in the previous paragraphs, a certain number of changes that result in a difficulty in exporting the test environment for different kernel versions.
The best solution would therefore be to have the right trade-off between the number of changes and the computational cost that these entail, trying to limit both as much as possible.

5.3 Implementation strategies

The following are the thought processes that led me to choose one strategy for the implementation of the permanent fault model over others:

1. The first reasoning centered on the timer for ticks. The tick timer in the FreeRTOS port for POSIX is managed by the function vPortSystemTickHandler (whose body is located in port.c) which is the function that calls xTaskIncrementTick which serves to increment the tick

counter.
Therefore, the idea to implement a permanent fault model could have been to also call a function that, at each tick, modifies the variable of the kernel on which the permanent injection has been carried out, taking into account precisely the bit that should have remained at a given value. In this way, it does not matter if a variable is modified at a certain instant by any kernel function (in particular it does not matter if the bit of interest for the injection is modified), since at the next tick that bit will be reset to the value decided at the injection phase.
This approach, as simple and apparently resolving as it may seem, actually hides numerous pitfalls. The first concerns the problem of overhead, clearly making a modification to a variable at each tick increases the overhead. The main problem, however, is due to the fact that the instructions of the FreeRTOS kernel are not atomic within a single tick, and this therefore means that there can be multiple writes and reads within the same tick. Therefore, a possible scenario (reasoning supported by experimental checks) is that a variable is injected, then modified by a kernel function and before the tick counter is triggered and therefore we arrive at the next tick, that this variable is read effectively with the possibility that the injected bit has undergone a variation, thus nullifying the permanence of the fault.
Therefore, a way to ensure that, following an injection on a variable, the modification of a variable by a kernel function and a subsequent reading do not fall within the same tick (so as to allow the rewriting of the bit of interest from the injection before a reading) can be to increase the frequency of the tick rate. By increasing the frequency of the tick, we would also reduce the time between one injection restoration (modifying the bit to the value of the injection) and another. So the result we would have is that it would be much less likely that, given a kernel modification to a variable, it would then be read within the same tick, improbable but not impossible.
So we would fall into a situation of non-determinism and this "solution" would not be cost-effective either, this due to the disadvantages that a high tick rate entails. The tick rate is set in the FreeRTOS configuration file: FreeRTOSConfig.h changing the value to the configTICK_RATE_HZ macro and usually this value is kept at 1 kHz or at most 10 KHz. Increasing it on the simulator to obtain a desirable effect would mean taking it to at least 100 KHz; this has a significant disad-

vantage, i.e., the simulation would only apply to systems that work with the tick rate at 100 KHz and therefore we would not have a versatile solution, i.e., a free simulation for different systems.

In general, as mentioned previously, embedded systems with FreeRTOS onboard do not operate with a high tick rate due to these reasons:

- Higher energy consumption: A higher tick rate requires a higher frequency of interrupts and task execution, which can result in higher energy consumption. This can be a critical problem in battery-powered or energy-saving embedded systems, where energy efficiency is a priority.
- Reduced available resources: A higher tick rate means that the processor has to dedicate more time to running the operating system, reducing the time available for other tasks or for the main application computation. This can limit the computational resources available for performance-critical applications.
- Increased latency: With a higher tick rate, tasks are executed at a higher frequency, but there may be additional delays due to interrupts and context switching operations. This can increase the latency of critical operations and affect the real-time performance of the system.
- Design complexity: A higher tick rate requires greater attention to system design. It is necessary to ensure that tasks are designed efficiently to meet the required execution times and that resources are managed correctly. This may require more design effort and testing to ensure that the system works correctly.

Therefore, the non-determinism of this solution and the fact that it restricts the discussion to little-used systems (with high tick rates) leads us to discard this solution.

2. The first solution centered on the use of a timer, i.e., the tick timer.

   A second possible approach would be to use an injector thread that continuously rewrites FreeRTOS's memory (shared) and thus continuously applies the same injection on the same kernel variable in a sort of polling, a mirror view, therefore, to that of the interrupts.
   The negative implications here are easy to imagine, we would fall into a

concurrency problem with the implications that this brings, for example, we can imagine interference problems due to the hierarchical models of multi-core processors. Therefore, it would be necessary to use synchronization mechanisms that provide that after each kernel modification to an injected variable, control is given from the kernel thread to the injector one so that it reapplies the changes and releases control to the kernel and so on; this type of behavior would lead to a substantial modification of the kernel, moreover, the execution (as the kernel would have to yield control to the injector) would be disturbed.

3. The third possibility is very interesting and would be very appropriate if it did not clash with the limits of C.

   The third possibility concerns operator overloading, i.e., the possibility of assigning programmer-pleasing behaviors to operators such as assignment operators. This idea might prove interesting from the point of view of realization in more modern languages like C++ and Rust but C unfortunately does not provide the possibility to change behavior to operators and being the FreeRTOS kernel programmed in C it is not possible to try to apply this idea.
   So the solution would be based on the fact of modifying the behavior of certain operators such as assignment operators of the real-time operating system kernel; in this way, the normal assignment operator '=' besides assigning normally to the operand on its left the result of the expression on its right, would immediately reapply the injection modification.
   So it would be possible through operator overloading to modify only the behavior of certain operators without the need to change much of the rest of the kernel code.

4. A fourth possibility, like the first, considers the use of timers, but this time not the tick timer.
   Therefore, it would be possible to use a specific timer different from the one for ticks and make this work at a high frequency, leaving the tick timer unchanged.
   Here too, we would have heavy overhead and therefore we would impact on a normal simulated execution, heavily burdening the FreeRTOS kernel.
   We should also consider that the timers of the FreeRTOS simulator for Linux (as well as that for Windows) are not based on a hardware-based

interrupt but on a high-priority thread based on the Clock of the operating system on which FreeRTOS is simulated, so we do not have a true real-time behavior on simulated FreeRTOS, and this introduces a certain uncertainty and inaccuracy in the use of timers.

5. The fifth idea for the realization of the permanent fault model was the road that I actually considered as winning and therefore undertook for the realization of the model.

   This is based on the realization of a software infrastructure that automatically modifies the original FreeRTOS kernel by applying the necessary changes to obtain the desired behavior of permanent faults in the presence of a fault injection; this approach ensures that we do not have to worry about the number of extra lines that will be added to the original kernel but only about the computational costs that derive from these. Moreover, this approach is designed to work on the different versions of the FreeRTOS kernel. More specifically, the behavior of permanence will be obtained by adding a function call after each modification of a variable (within the kernel). In this way, it is possible to obtain a deterministic behavior since it is ensured that after every write operation that occurs on the kernel side of a variable, the operation following the rewrite will take into account the injection and therefore simply reapply the fault to the bit.
   So after each clean write operated by the kernel and only after a clean write has actually occurred, there will be a subsequent rewrite to reapply the bit fault modification. In this way, it will not be possible for a clean write not taking into account the fault and a read not to have first re-applied the permanence and nor will it be possible for there to be more injection modifications (to reapply the permanence) between a clean write and another and this clearly makes the overhead minimal.

5.4 Project structure

The thesis project basically consists of 3 folders:

- The Source folder which contains the original kernel of FreeRTOS which will be automatically modified to allow fault injections

- The FreeRTOS folder which is the folder that will contain the modified FreeRTOS kernel

- And the simulator folder that contains the folder with the modified porting files of FreeRTOS for linux (which will replace the original one) and thus allows the simulation of FreeRTOS; moreover, the simulator folder contains all the files needed for fault injections that are not attributable to the kernel and are therefore external to the kernel.

In the main folder beyond these 3 folders, we have the script file for the compilation compile_posix.sh, the CMakeLists.txt file and some useful csv for the injections, these will be explained in more detail later.

Going into detail about the `simulator` folder, the most important files and folders are:

- **main.c**
  It is the heart of the project, in fact, here is the code that the main process (i.e., Orchestrator) starts executing, so the code for the commands accepted by the fault injection simulator is written here.
- **main_blinky.c**
  This file retains the name of the example code of the FreeRTOS simulator for Linux, in fact, it is used to run the benchmark code which in our case runs a qsort on words taken from an external file; therefore, it is nothing but the activity that FreeRTOS will execute whether it is a golden execution or an execution with injection.
- The injection folder
  - **injection.c**
    From the name itself, it can be inferred that it is a fundamental file regarding injections on the data structures of the FreeRTOS kernel, in fact, it is the file where the code is specified on how to reconstruct the memory address (starting from the target in the global list) on which to perform the injection. In particular, intercepting the memory address on which to carry out the injection changes depending on whether the target of interest on which you want to carry out the injection is a list, a pointer or a variable of another type.
- The posix folder

  - **fork.c**
    file where the code regarding the creation and management of child processes concerning POSIX is written.
  - **thread.c**
    This file contains the code to launch the injector thread.

- **mod_kern.c/mod_kern.h**
  These files are the heart of the permanent fault model and are used to modify the kernel appropriately to have the behavior of fault persistence but are also fundamental for transient faults as they generally deal with preparing the kernel for injections (modifying it accordingly).

- **perma_fault.c/perma_fault.h**
  These files contain the body and the heading of the ReinstatePermanentF function which is used to reapply the fault after a modification to the injected target, by the kernel. The ReinstatePermanentF function and its behavior will be explained in more detail in the next paragraphs.

- **targets.csv**
  file that contains the targets for which to call the ReinstatePermanentF function after each modification before the kernel compilation. Potentially, therefore, all possible targets should be there to have the widest range of possible runtime injections.

5.5 Design and Implementation of the Kernel Modification Infrastructure

As we have said, the behavior of permanent fault injections within this thesis project is inevitably linked to changes made to the kernel that precede the kernel compilation phase. The changes, as mentioned in previous paragraphs, are automated for the purpose of allowing to hide the implementation details on the kernel side of the injection infrastructure and therefore make it immediately usable for any user, starting from an original kernel, the injection simulation on FreeRTOS. To pursue these goals, the idea was to modify the scripting file that started the compilation of the project creating the sim.out executable. The scripting file that started the compilation in question is compile_posix.sh and with appropriate modifications now serves the purpose of recursively deleting the contents of the FreeRTOS folder (if present) which contains the kernel already modified previously then within FreeRTOS it moves a copy of the kernel in the Source folder (which is the folder containing an intact FreeRTOS kernel). The steps just mentioned are useful for not causing conflict between modifications already applied to the kernel with new modifications and this is useful when some kernel modification file is modified. Therefore at this point we find ourselves in the condition of having a copy of the original kernel in the FreeRTOS folder but clearly the source has not been removed because it will be useful if we want to recompile everything. At this point, compile_posix.sh begins to make the first modifications to the kernel, keeping in mind that only an initial part of the modifications (i.e., those that involve moving files and folders) will be operated by compile_posix.sh. The first folder to insert into the original kernel is the porting folder, this is already present but modifications have been applied that allow the execution of the injector threads (for each instance the injector is one, the plural refers to the execution of multiple instances such as for example in the campaign or in general in the execution of multiple runs). The modified porting folder is located inside the simulator folder, so compile_posix.sh takes care of removing the porting folder from the FreeRTOS kernel to be modified and in its place puts a copy of the folder with modifications. The last files that we will need to move are perma_fault.c and perma_fault.h which contain the injection function that will interest the FreeRTOS files where the structures that are possible targets are contained. At this point, we need to intervene on the kernel code, for this we will use a modification process produced by an executable that we will produce from the files mod_kern.c and mod_kern.h. So through this executable, the files for the injections will be modified for example tasks.c and timers.c adding the files to import for injections for example perma_fault.h, the code with the information on the data structures for the production of the global list of targets and of course the calls to the ReinstatePermanentF function. Only after the kernel modification process has finished will the fault simulator compilation process start so that the Orchestrator process can be started through the sim.out executable.

5.6 ReinstatePermanentF function

The ReinstatePermanentF function, where the name stands for (Reinstate Permanent Fault), is the function that takes care of reintroducing into a kernel variable the modification suffered due to a permanent fault injection, and is used to simulate the persistent effect of a permanent fault. In other

words, every time the kernel variable is genuinely modified by the kernel but subsequently to a permanent injection, the effect will be that the kernel will call the ReinstatePermanentF function which takesinto account the injection and modifies the bit marked by the injection to the value that this has induced, thus reintroducing the injection.
From a logical point of view, what needs to happen in order to reapply a permanent fault modification will depend on the value we intend to restore for the bit hit by the injection.

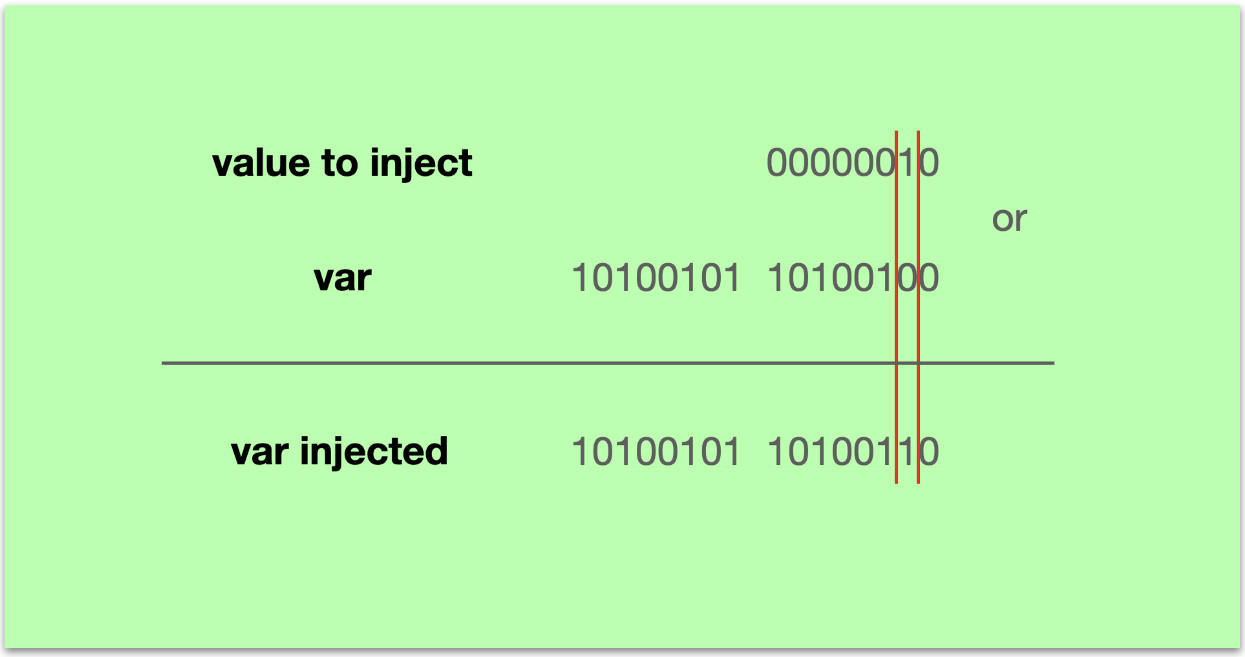


Figure 5.1: Example of action of the ReinstatePermanentF function that re-applies the modification of the injected bit by setting it to 1.

As can be seen in the image above, to reintroduce a fault to a bit in a way that maintains the value 1, we will have to do a bitwise "or" between a value that is likely to have a size of 1 Byte (in which only the bit in the position of interest by the fault will appear as 1) and the byte of the injected variable (for which we are reintroducing the fault).
Please note that for the bit flip of an SEU, we do not use bitwise or but bitwise xor because in the case where the bit at 1 of the mask corresponds to a bit at 1 in the variable to be injected, then in that case the bit should invert and therefore become 0, an effect that we do not want when we decide to inject 1 in a given bit even if 1 is already present.

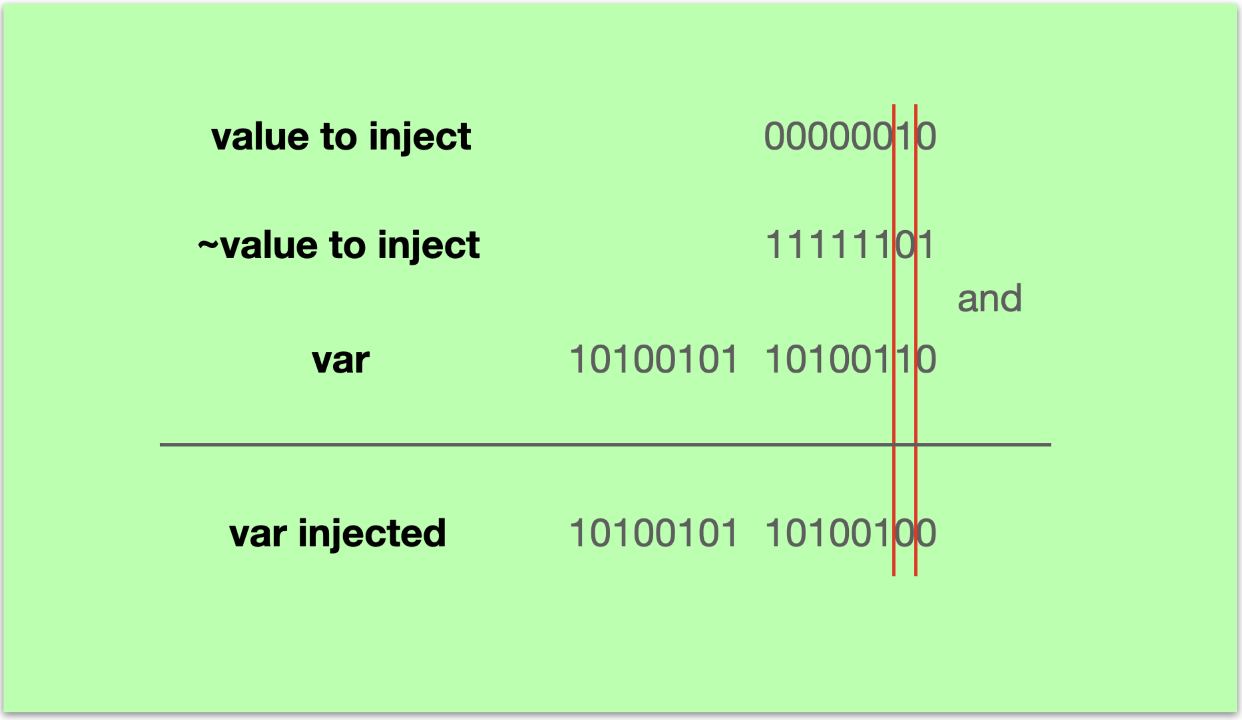


Figure 5.2: Example of action of the ReinstatePermanentF function that re-applies the modification of the injected bit by setting it to 0.

To introduce persistence with a value set to 0, we will instead need an extra step, in fact we will need to do a bitwise not on the value to be injected and then we will do a bitwise and between the obtained value and the byte where to reintroduce the modification, of the desired kernel variable.
To reintroduce the injection, we will therefore need to use auxiliary kernel variables.
One variable will therefore be the mask, i.e., it will contain the bit to be injected set to 1 and the other bits to 0. Another auxiliary kernel variable that we are interested in using will be the one that contains the memory address of the kernel variable on which we are interested in carrying out the injection. In fact, what happens in the code is that in the injection phase, the injector thread, in addition to injecting the kernel variable on which we want to carry out a permanent injection, will make two other injections: it will set the mask (which is a new kernel variable introduced in the automatic kernel modification phase) and will set in another auxiliary variable (also introduced in the automatic kernel modification phase) the memory address starting from the byte that we are interested in considering, of the injected variable.

5.7 Regular expression and pattern matching

To identify the assignment patterns or modification of a FreeRTOS kernel variable, to which it is necessary to follow the call to the f function, I chose to use the POSIX library: regex.h.
The regex or regular expressions in formal language theory are a representation of regular languages (type 3 languages according to Noam Chomsky's classification).
In general, a language is nothing more than a set of strings that we are interested in considering, the strings of this set (the language of interest) can be described by a grammar i.e., by an alphabet (set of symbols) and by rules (also called productions);

Regular languages will be described by grammars with productions in the form:

$$P = \{ \Psi \rightarrow a\,\Omega,\ \Omega \rightarrow a \mid \Psi, \Omega \in N\ ;\ a \in T \}$$

Figure 5.3: Representation of the productions of regular language grammars

Regular languages are the simplest languages to deal with in Chomsky's classification and are the languages that require a worst-case computational cost lower to solve the membership problem; the pattern-matching algorithm of our interest is largely reducible to the membership problem, i.e., the problem in which given a string we want to know if this is part or not of a predetermined language.
In our case, in fact, the language of interest will be that given by all strings in which an assignment appears or in general a modification to a variable through assignment operators or modification operators such as increment, decrement etc., and we want to recognize whether or not a substring that is accepted by this language appears within a line of FreeRTOS kernel code.

A grammar that can exemplify this language will be in the form:

$$G = (\{\Psi, \Omega\}, \{a, b, \ldots, +, -, *, =, \ldots\}, P, \Psi)$$

$$\begin{aligned} P = \{\, & \Psi \rightarrow {+}{+}\,\Omega \mid {-}{-}\,\Omega \mid \Omega\,{+}{+} \mid \Omega\,{-}{-} \mid \Omega\, {*}{=} \mid \ldots \\ & \Omega \rightarrow a\,\Omega \mid b\,\Omega \mid \ldots \mid a \mid b \mid \ldots \\ \} & \end{aligned}$$

Figure 5.4: Representation of the grammar of interest

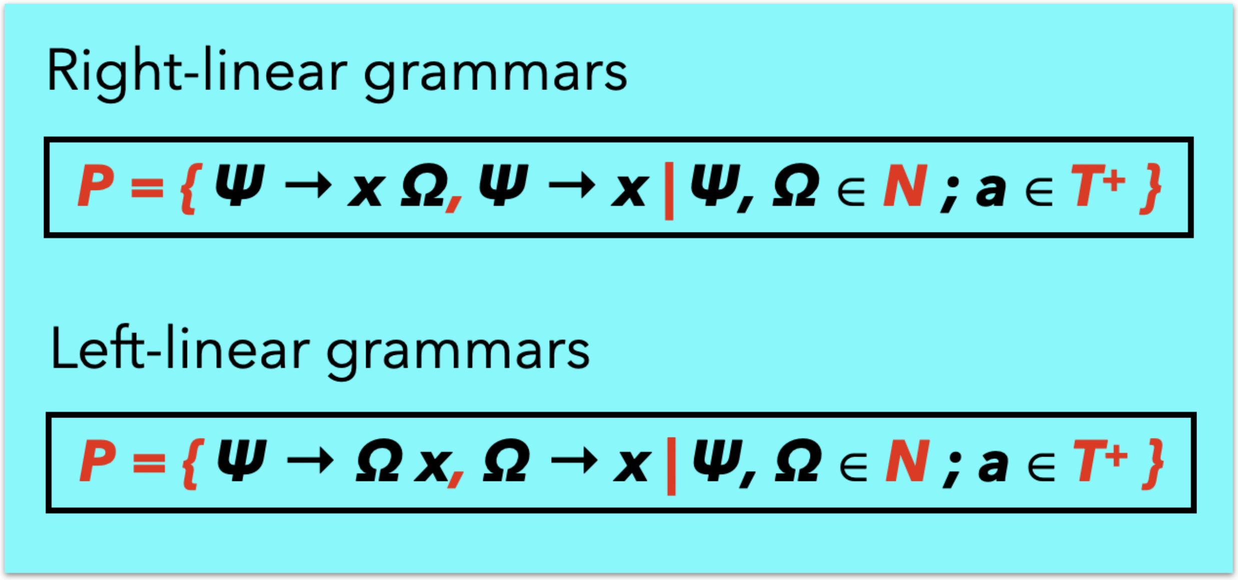
In which G represents the tuple of the grammar given by the set of non-terminal symbols, the set of terminal symbols, the set of productions, and the start symbol; P contains all the productions of the grammar.
It can be inferred that such a grammar does not correspond to a type 3 grammar i.e., a regular language. It is easy to see that more than one terminal symbol appears alongside a non-terminal (on the right side of the productions) but this is not necessarily determining because it could be a linear grammar reducible to a regular language, but this is not the case.

Remembering that a linear grammar has a form:

$$P = \{\Psi \rightarrow x\,\Omega\,y, \Psi \rightarrow x, \Psi \rightarrow y\,\Omega\,x \mid \Psi, \Omega \in N;\; x \in T^{+};\; y \in T^{*}\}$$

Figure 5.5: Representation of the productions of linear language grammars

It is possible to prove that a linear grammar can be reduced to a regular language grammar only if it is a right linear or left linear (i.e., the non-terminal in the right part of a production is only preceded or followed by terminals).

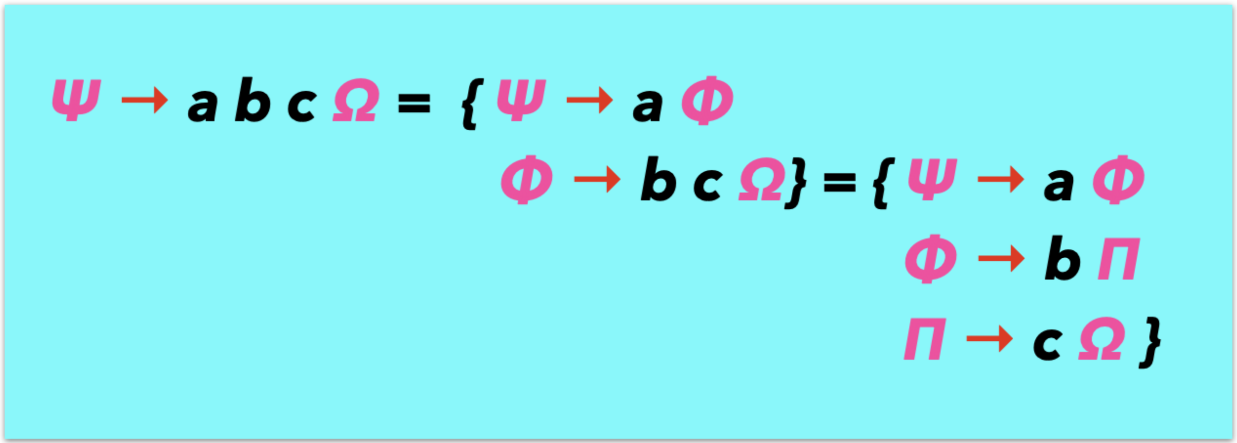

Right-linear grammars

$$\boxed{P = \{\Psi \rightarrow x\,\Omega,\ \Psi \rightarrow x \mid \Psi, \Omega \in N\ ;\ a \in T^{+}\}}$$

Left-linear grammars

$$\boxed{P = \{\Psi \rightarrow \Omega\, x,\ \Omega \rightarrow x \mid \Psi, \Omega \in N\ ;\ a \in T^{+}\}}$$

Figure 5.6: Representation of the productions of right-linear and left-linear grammars



$$\Psi \rightarrow a\,b\,c\,\Omega = \{\Psi \rightarrow a\,\Phi \quad \Phi \rightarrow b\,c\,\Omega\} = \{\Psi \rightarrow a\,\Phi \quad \Phi \rightarrow b\,\Pi \quad \Pi \rightarrow c\,\Omega\}$$

Figure 5.7: Example of equivalence between a right-linear grammar and a right-regular grammar

In our case, however, it is immediately noticed that this grammar has terminal symbols both to the right and to the left of non-terminal symbols so it is linear but not regular. As much as in formal language theory, regexes are representations of regular languages, in reality the POSIX regex.h library as well as various libraries for programming languages are more powerful tools capable of doing pattern-matching also outside of regular languages and thus the term regex loses a bit of its rigorous meaning. It should be considered that a regular language is representable through deterministic finite state automata DFA, for which it is possible to solve the membership problem with complexity O(n) in which n represents the size of the string; differently from type 3 of the formal languages (the regular languages) type 2 i.e., the context-free-languages have complexity to solve the membership problem O(n$\hat{3}$) while only in the case of deterministic CFLs (content-free languages) it is possible to solve the membership problem with complexity O(n). Considering that linearlanguages are a subset of CFLs (halfway between regular ones), we can consider O(n$\hat{3}$) as the worst-case complexity.

The regular expression of our interest, therefore, has the form:

```
(-{2}|\+{2})( *)([a-zA-Z_][a-zA-Z0-9_]*)|([a-zA-Z_]
[a-zA-Z0-9_]*)( *)-{2}|\+{2}|=[^=]|\+=|-=|\*=|\/=|<<=
|>>=)
```

- `(-{2}|\+{2})( *)`

  This indicates that the expression may or may not start with two consecutive pluses or minuses. This is important to us because `++` and `--` are clearly not atomic operations, but they hide an assignment as well as an increment.

- `([a-zA-Z_][a-zA-Z0-9_]*)`

  We are seeing this as a separate piece to be explained for simplicity, but for all intents and purposes, it was part of the previous case. This piece is used to describe a string of characters that can make up a variable name in C. Therefore, this piece is preceded by

  `(-{2}|\+{2})( *)`

  Exactly because after `++` and `--` there must be the name of a target (as we were saying from the implementation point of view, this piece in the code is replaced by the names of the targets).

- `([a-zA-Z_][a-zA-Z0-9_]*)( *)(-{2}|\+{2})`

So this part is found after a "—" (or) to indicate that it can be in place of the previously explained possibility, that is, in place of the prefix increment or decrement. It represents the postfix increment and decrement.

- `=[^=]`

  This part concerns the case where there are no postfix increments or decrements (in fact it is found after a "— (or)") and it is used to say that there must not be two consecutive equals because that would be a case of comparison and not assignment.

- `\+=|-=|\*=|\/=|<<=|>>=)`

  This part handles the remaining types of assignments that might be found in FreeRTOS, remembering that the backslash is used to not recognize the next character as a regex rule but only as a character of the strings to be identified.

The regular expression just presented may appear daunting at first glance, but the advantage of being able to represent the patterns of our interest in a single line of code allows us to avoid writing a significantly more complex algorithm, below we see an example:

```c
char input[100]; // We assume the input string has
    a maximum length of 100 characters
  printf("Enter a string: ");
  fgets(input, sizeof(input), stdin);

  // Remove the final newline character generated
     by fgets
  if (input[strlen(input) - 1] == '\n') {
      input[strlen(input) - 1] = '\0';
  }
  char *ptr = input;

  while (*ptr != '\0') {
      // Check the first pattern: (-- or ++)
      if ((*ptr == '-' && *(ptr + 1) == '-') ||
         (*ptr == '+' && *(ptr + 1) == '+')) {
            ptr += 2; // Skip the two characters --
               or ++
            while (isspace(*ptr)) {
                ptr++; // Skip the subsequent
                   spaces, if any
            }
            // Print or execute action for the
               pattern (-- or ++)
            printf("Pattern (-- or ++): ");
        }
         // Check the second pattern: (variable or
           function followed by --, ++, =, +=, -=,
           *=, /=, <<=, >>=)
        else if (isalpha(*ptr) || *ptr == '_') {
            int validName = 1; // Flag to check the
               validity of the variable or function
               name
            while (isalnum(*ptr) || *ptr == '_') {
                ptr++; // Skip the valid characters
                   for the variable or function name
            }
            while (isspace(*ptr)) {
                ptr++; // Skip the subsequent
                   spaces, if any
            }
            // Check the third pattern: (-- or ++ or
               =)
            if (*ptr == '-' || *ptr == '+' || *ptr
               == '=') {
                ptr++; // Skip the character --, ++
                   or =
                if (*ptr == '=') {
                    ptr++; // Skip the character =
                       for cases +=, -=, *=, /=,
                       <<=, >>=
                }
                // Print or execute action for the
                   pattern (variable or function
```

```c
                followed by --, ++, =, +=, -=, *=, /=, <<=, >>=)
                printf("Pattern (variable or function followed by --, ++, =, +=, -=, *=, /=, <<=, >>=): ");
            } else {
                validName = 0; // The name is not followed by one of the valid patterns
            }
            // Print or execute action for the variable or function name
            if (validName) {
                printf("Name: ");
            }
        } else {
            ptr++; // Skip the current character and move to the next one
        }
    }
    return 0;
```

Another advantage offered by the use of regular expressions is the fact that most programming languages offer valid ways to use Regex. Therefore, exporting this type of solution for a project with a different programming language can be much simpler.

5.8 Probability Distribution

To select the time instants at which an injection will be executed on a FreeRTOS variable, within an injection campaign, the idea was to use random instants given by probability distributions; in this way there will be a realistic simulation precisely because it reflects reality where the exact moment in time when a SEE will hit our system cannot be known even when carrying out statistical analyses.
The simplest example of a probability distribution that comes to mind is the

uniform distribution, in which every instant has an equal probability of being the candidate for the injection.
The probability density formula for the uniform distribution is given by:
f(x) = 1/(b-a)

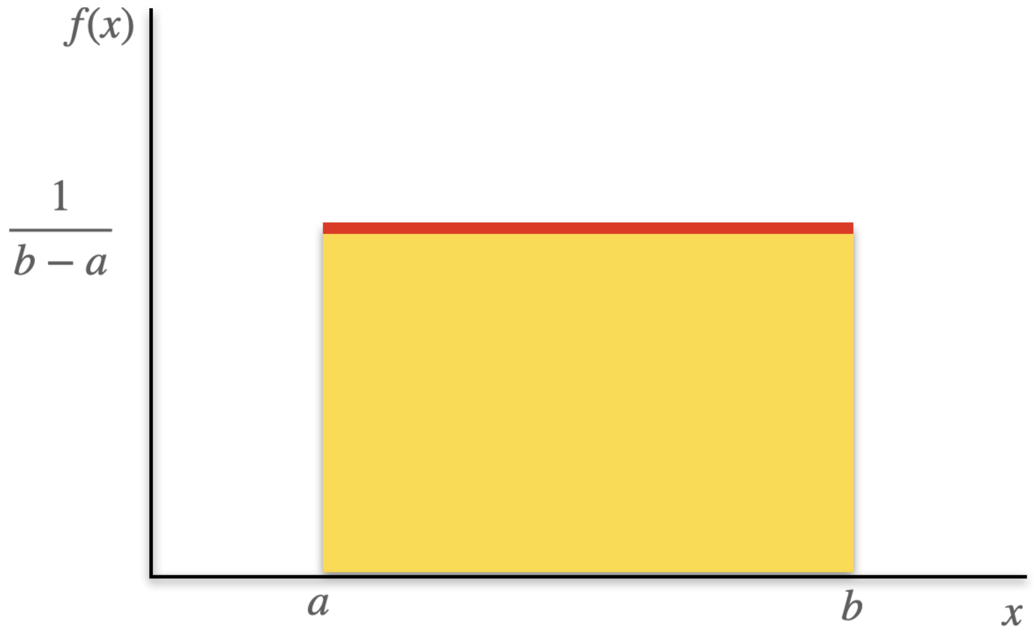


Figure 5.8: Uniform distribution

To establish the ends of the time interval to be considered and therefore the ends of the distribution, we will use the execution time of the golden execution reported in the first line of golden.txt.
The left end will be instant 0, i.e. the start of execution, while the right end will correspond to the execution time of the golden.
Therefore, with this type of distribution, the variability of the data extends over the entire interval considered.

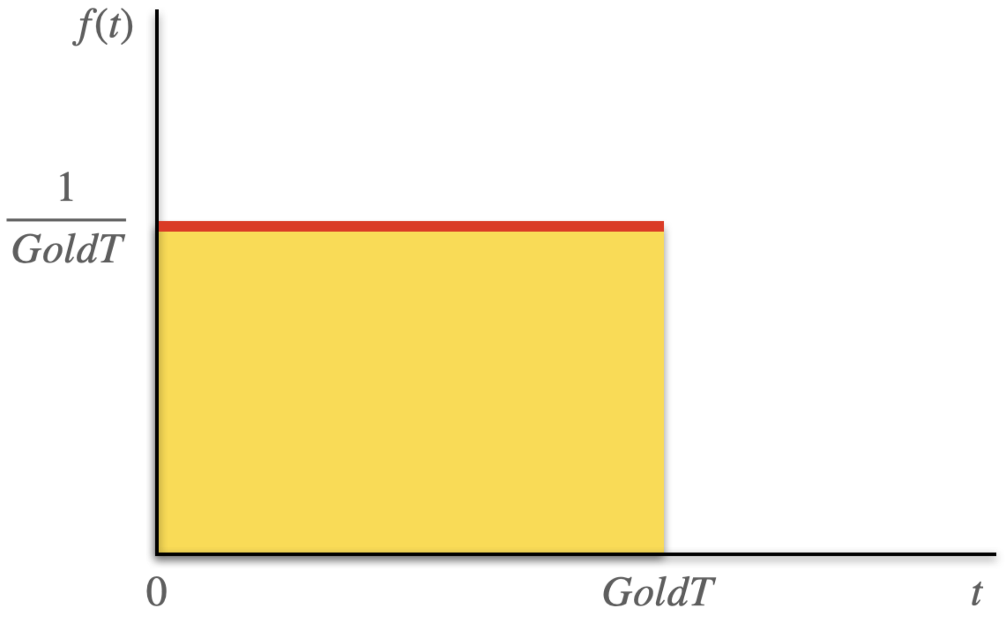


Figure 5.9: Representation of the uniform distribution used for injections, simplified (In the algorithm, the rightmost end is represented by GoldTime, from which a user-chosen percentage is subtracted)

But if we wanted to observe the behavior of the operating system when the injections are concentrated around a certain time instant, we would have to use a probability distribution different from the uniform one, and therefore we would have to think about a distribution that guarantees us a low dispersion of data compared to a central value.
We might think, for example, of using a Normal (otherwise called Gaussian) probability distribution.
Using the Normal probability distribution for the random extraction of the time instant of an injection, we will have a higher probability that this will occur in a given range of the average of the distribution; where the range is given by the standard deviation.

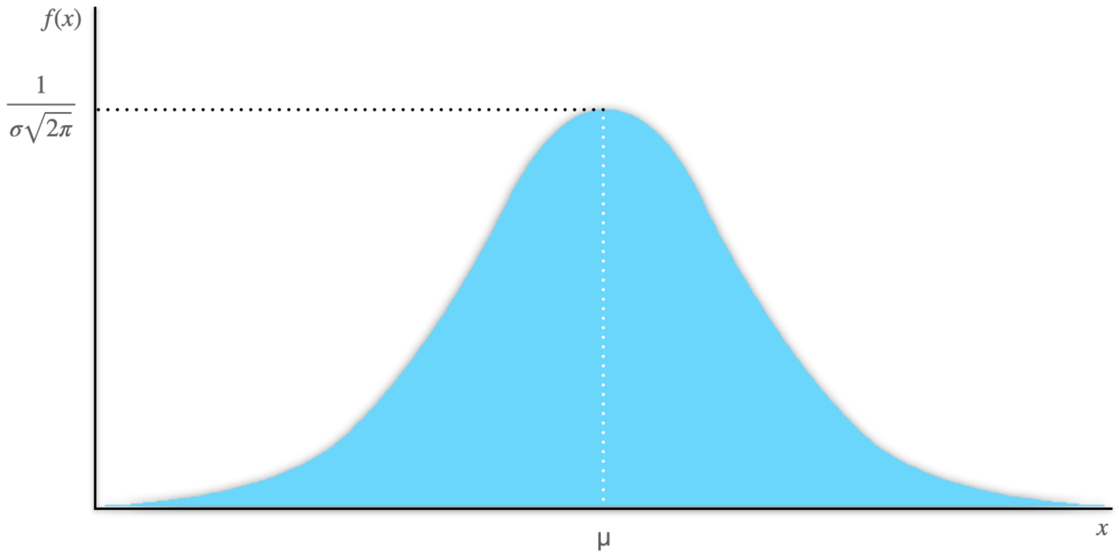


Figure 5.10: Gaussian (Normal) distribution

In particular, considering a standard normal probability distribution, i.e. with zero mean ($\mu = 0$) and standard deviation one ($\sigma = 1$), we can consider 3 standard deviation intervals:

- Interval $\pm\delta$: This interval extends one standard deviation (σ) from both sides of the mean (μ). In a standard normal distribution, this interval includes about 68% of the data. Thus, about 68% of the data will be in the interval from $-\delta$ to $+\delta$.

- Interval $\pm 2\delta$: This interval extends two standard deviations (2σ) from both sides of the mean (μ). In a standard normal distribution, this interval includes about 95% of the data. Thus, about 95% of the data will be in the interval from -2δ to $+2\delta$.

- Interval $\pm 3\delta$: This interval extends three standard deviations (3σ) from both sides of the mean (μ). In a standard normal distribution, this interval includes about 99.7% of the data. Thus, about 99.7% of the data will be in the interval from -3δ to $+3\delta$.

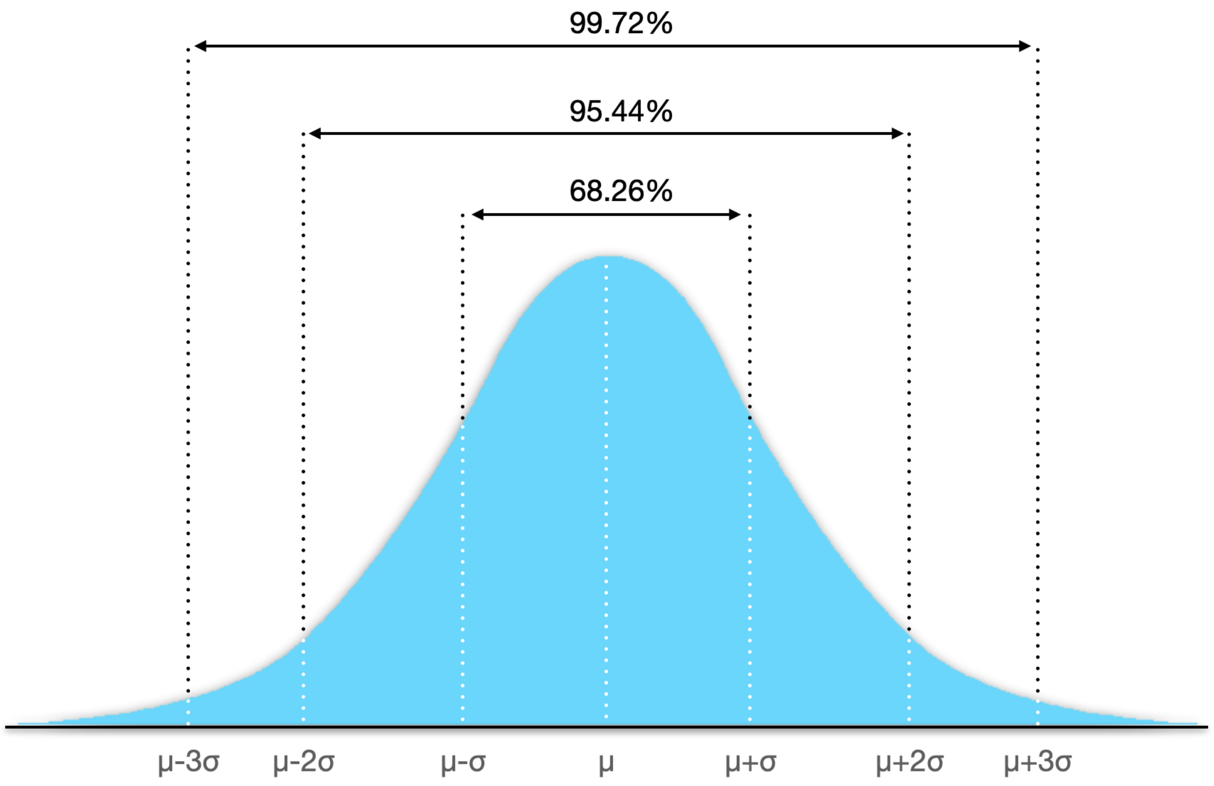


Figure 5.11: Gaussian standard distribution

So in short, the value of the extraction will fall with a 68% probability "close" to the average, i.e. in the interval $[\mu - \delta, \mu + \delta]$ and this value reaches 95% deviating a little more from the average, i.e. in the interval $[\mu - 2\delta, \mu + 2\delta]$ up to touch 99.7% in the interval $[\mu - 3\delta, \mu + 3\delta]$.
We can consider these standard deviation intervals valid with a certain approximation even for non-standard normal distributions.
The Normal distribution that interests us is not standard and in general we will not have a constant value of the standard deviation and the mean among the various injections and this is because we start with the start time of execution and the end time, i.e. the duration of the golden. So, given the considerations made earlier, we need to adapt the graph of the distribution to our interval; as we know, however, a probability distribution is not limited to an interval but has an approximate 0.3% chance that the measurement falls in the interval $[-\infty, \mu - 3\delta] \cup [\mu - 3\delta, +\infty]$.
Given that the 99.7% probability falls in the interval $[\mu - 3\delta, \mu + 3\delta]$ then we will make sure that this standard deviation interval coincides with our execution interval [0,goldenTime], by putting $\mu - 3\delta = 0$ and goldenTime=$\mu + 3\delta$;

in this way we will have the value of the mean μ and the measurement of the deviation δ for a distribution of our interest.
The knot of the 0.3% that hits outside the interval [0,goldenTime] still remains, for this we will operate a truncation to make sure that the algorithm does not draw numbers outside the interval.

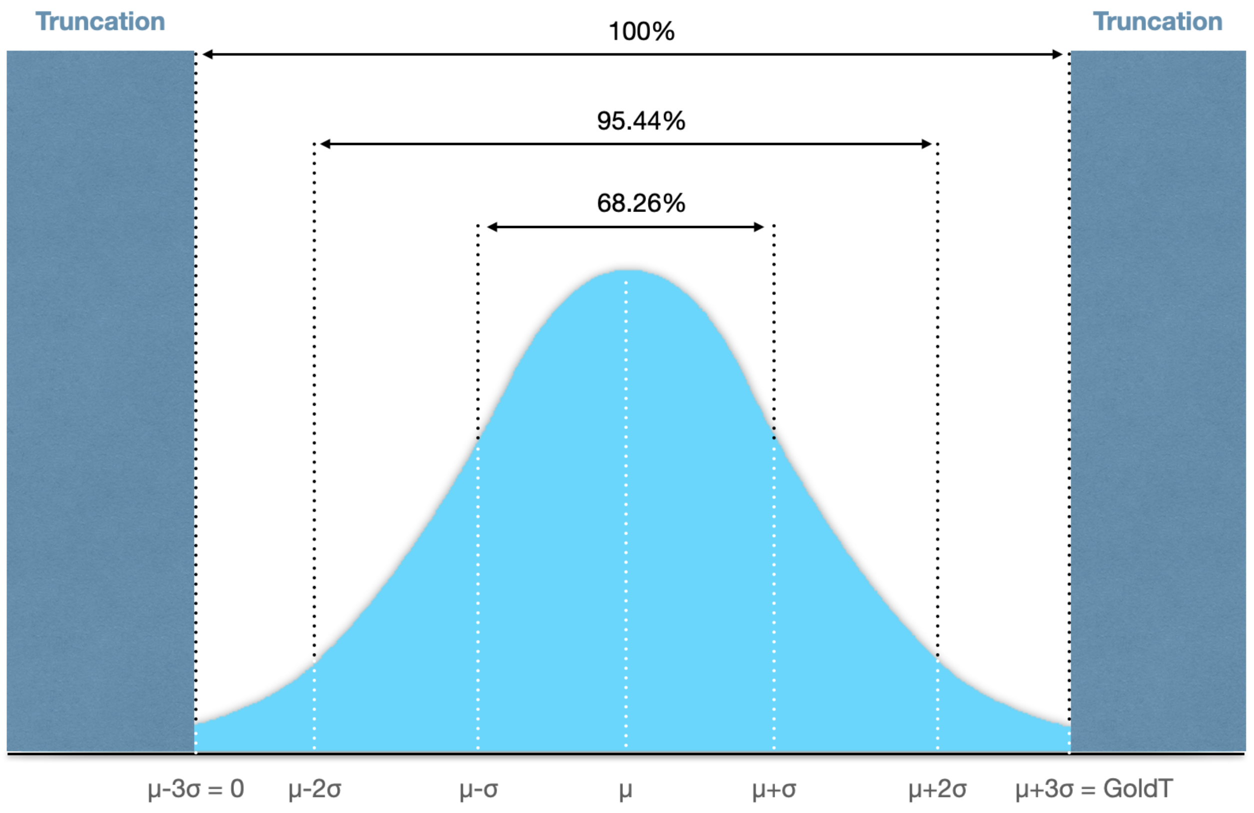


Figure 5.12: Representation of the truncated Gaussian distribution used for injections

5.9 Function rand & Box-Muller transform

In the previous paragraph, the motivations and statistical foundations used to choose the instants of the injections within a campaign were explained; what was not explained are the implementation details of the distribution algorithms.
Both algorithms are based on the rand function.

In C, the rand() function is used to generate pseudo-random numbers. This function returns a random integer in the range from 0 to RAND_MAX, a predefined constant that represents the maximum value that can be generated by the rand() function.

To properly use the rand() function, you need to initialize the random number generator using the srand() function. It is common to use the value of the system clock as a seed for initialization, so that the generated numbers are different at each run of the program.

Therefore, in itself, the randfunction is already a way to extract pseudo-random numbers using the uniform distribution, you just need to narrow down the interval of extractable values to the one of interest.

It should be noted that the interval chosen for the values to be extracted is not: [0, goldenTime] but [0, goldenTime -x%*goldenTime] where x represents a percentage value that we will subtract from the golden time to narrow the interval by decreasing the right end; in this way we can increase the probabilities that the injection falls within the execution in progress, and this is because the execution times are not always the same and therefore the value of the execution time of the golden execution might not be reached by the other executions but be close, narrowing the interval by decreasing the right end ensures that the maximum value of this interval falls within the lifespan of the execution in progress.

To narrow the range of uniform values to an interval of interest:

```
int randomNumber = min + (rand() % (max - min + 1));
```

Where in our case min is 0 and max is goldenTime -x%*goldenTime

As for the normal distribution, the discussion is more complex, precisely because rand uses a uniform and not normal distribution, we need a way that allows us to switch from the uniform to the normal.

The Box-Muller transformation is an algorithm used to generate random numbers from a normal (Gaussian) distribution from uniformly distributed random numbers. The algorithm requires two uniformly distributed random numbers (often referred to as z1 and z2) to generate two normally distributed random numbers.

The reason why two random numbers are used is that the Box-Muller transformation exploits the properties of the polar coordinate system. The method converts the two-dimensional Cartesian coordinates of the random numbers into polar coordinates. The random numbers are then transformed so that the polar distances are proportional to the square root of the opposite of the natural logarithm of the random numbers.

The process of generating a single random number from a normal distribution through the Box-Muller method therefore requires two uniformly distributed random numbers, which are used to calculate two normally distributed random numbers, but only one of these is used as the final result.

This method is based on the following idea:

Generate two independent and uniformly distributed random numbers in the range $[0, 1]$. Let's call them U_1 and U_2. Calculate the following variables:

$$Z_1 = \sqrt{-2\log(U_1)}\cos(2\pi U_2)$$

$$Z_2 = \sqrt{-2\log(U_1)}\sin(2\pi U_2)$$

The numbers Z_1 and Z_2 obtained in this way will be independent and will follow a normal distribution with mean 0 and variance 1.

To adapt the normal distribution generated by the inverse Box-Muller transformation to a desired normal distribution with a specified mean (*mean*) and standard deviation (*stddev*), you can use the following formula:

$$X = mean + stddev \cdot Z$$

Where X is the random number generated with the desired normal distribution.

So the code will look like this:

```
U1 = random_uniform()
U2 = random_uniform()

Z1 = sqrt(-2 * log(U1)) * cos(2 * pi * U2)
Z2 = sqrt(-2 * log(U1)) * sin(2 * pi * U2)
```

```
X = mean + stddev * Z1
```

Chapter 6

Experimental results

6.1 Introduction

The injection environment presented so far has been developed with the aim of obtaining experimental results that could be consistent. For this purpose, I created a module that could automatically generate datasets with the specifications on the injections chosen by the user. The specifications present in the datasets for the injections are then used by another module, the injection campaign module, which reads from the dataset the targets on which to perform the injections and for each target sees the number of injections to be made on that target, the type of distribution to be used for the extraction of the moments in time to perform the injection on the target, the type of injection i.e. whether the fault should be transient or permanent, I also included the possibility of choosing whether the injection should be on a single bit or multiple (this for when this second possibility will be implemented), and also from the dataset for an injection campaign it is possible to also take the data relating to the percentage to subtract from GoldenTime to generate an upper right extreme of the injection (in terms of time) that could also be less than the golden time, effectively allowing us for example to shift the mean of the Gaussian with respect to GoldenTime/2 with the result of considering standard deviation intervals that potentially fall towards the initial moments. The parameters chosen to evaluate during the injection campaign for the total number of injections on the single target are (in percentage): Ok, Delay, SDC (silent data corruption), Hang, Crash.
These are all exit conditions that a single execution with fault injection may be subject to and particularly concern the execution time of a benchmark task on FreeRTOS when a fault is injected on a target and the output produced

by this execution. Indeed, the execution time can be significantly greater in the presence of an injection compared to a golden execution and in that case it will be evaluated as an execution time delay, in general the goldenTimes that use the same benchmark carried out on the same machine, never have a percentage difference greater than 5%, so to assess whether an execution has suffered slowdowns compared to the golden execution, we will see that it is more than 105% of the golden execution; moreover, within the campaign, the output produced by the task is also evaluated, in the presence of a fault, if it does not correspond to that given by the golden execution, it will be evaluated as incorrect.
Therefore, if there is no delay in the execution and the output is correct we will say that the injection has led to an Ok, if instead there is only a delay but the result is correct we will be in the presence of Delay.
If the output is different but there is no delay in the execution then it will be said that there has been an SDC, if instead there is a delay and the output is different then we will have a Hang. The crash is detected at the moment there is an incorrect termination and therefore there was a problem with the ISR (Interrupt Service Routine)
From each campaign we will therefore have the percentage of Ok, Delay, SDC, Hang, Crash given a number of injections for each single target. From a first analysis it became clear how the results showed a high variability with a number of injections on the single target less than 50 per campaign, effectively stabilizing and not showing big differences between 100 injections and a larger number of injections (in fact, up to 1000 injections per target on a single campaign were tested). For the injection campaigns that will be shown, it was chosen to use as a limit instant a value lower by 10% compared to the GoldenTime.

6.2 Operational groups

To examine the impact of the injection campaigns on the different aspects of the real-time system more effectively, it was decided to group the targets into operational groups:

- **System State Variables**: This category includes global variables or data structures that maintain the current state of the operating system. These variables might represent, for instance, the state of the system clock, the number of currently running tasks, or whether the operating system is currently in an idle or execution state.

- **Task Lists**: FreeRTOS manages tasks using various lists. For example, there might be a list for tasks that are ready for execution, one for tasks that are waiting for a resource or event, and one for tasks that are suspended. Each list is a data structure that contains pointers to Task Control Blocks (TCB) for each task in the list. Performing tests on these lists could help evaluate the robustness of the operating system functions related to task scheduling and resource management.

- **Current TCB**: This category refers to the Task Control Block (TCB) for the currently executing task. In FreeRTOS, a TCB contains all the necessary information for managing and scheduling a task. This includes details such as the task's priority, the task's state, the task's stack information, and much more. Performing tests on the current task's TCB could help evaluate how the operating system manages the current task and identify any weaknesses or issues in the system operating code or logic related to this aspect.

6.3 Benchmarks

To carry out an evaluation on the target injections, it was chosen to use the benchmarks from MiBench. MiBench is a set of free benchmarks, widely used in both academic and industrial fields to test and evaluate the performance of embedded systems. The benchmark set is divided into various categories that represent different real-world applications, including automotive, consumer, network, office, security, and telecommunications. In particular, it was chosen to use the automotive benchmarks that simulate applications typically performed in embedded systems within vehicles. These benchmarks focus on control applications, sensors, and communication systems typically used in the automotive industry.
Specifically, the following tests were used:

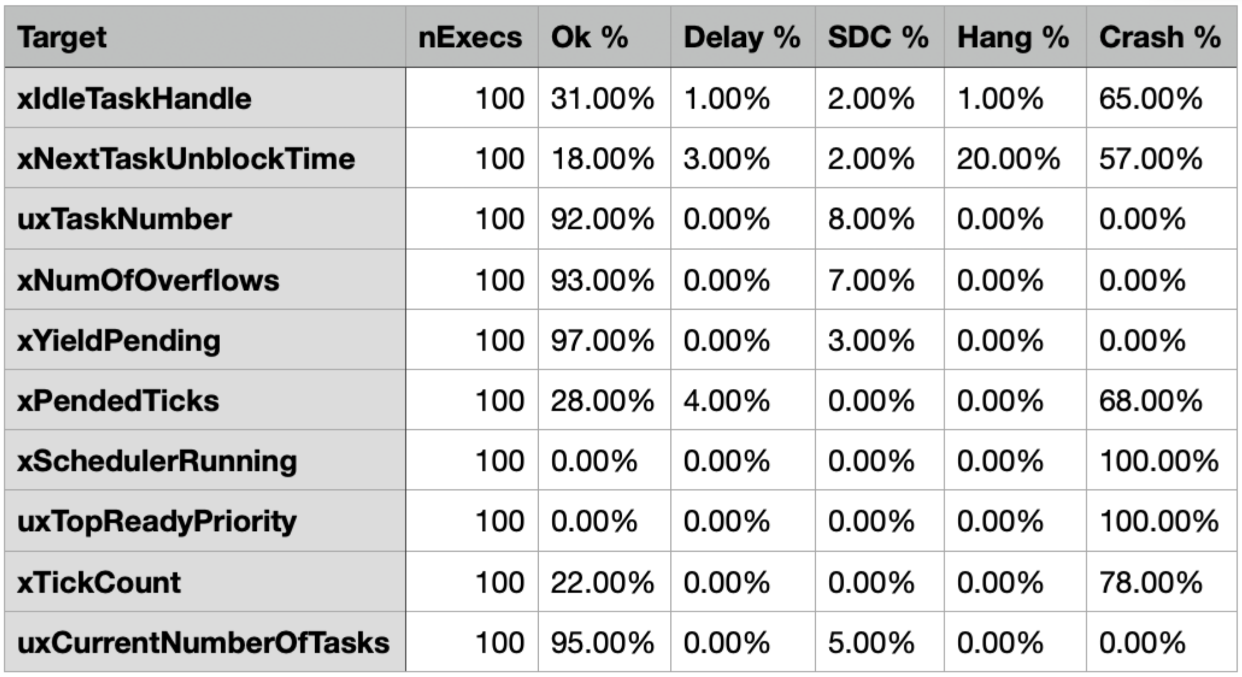
- Bitcount: Counts the number of bits set in an integer, simulating bit-level operations often used in embedded systems.

- Qsort: Implements the quicksort algorithm, useful for sorting large amounts of data, as might be needed in an automotive embedded system.

- Basicmath: Simulates the basic mathematical operations that might be required by an automotive embedded system, such as square root, division, etc.

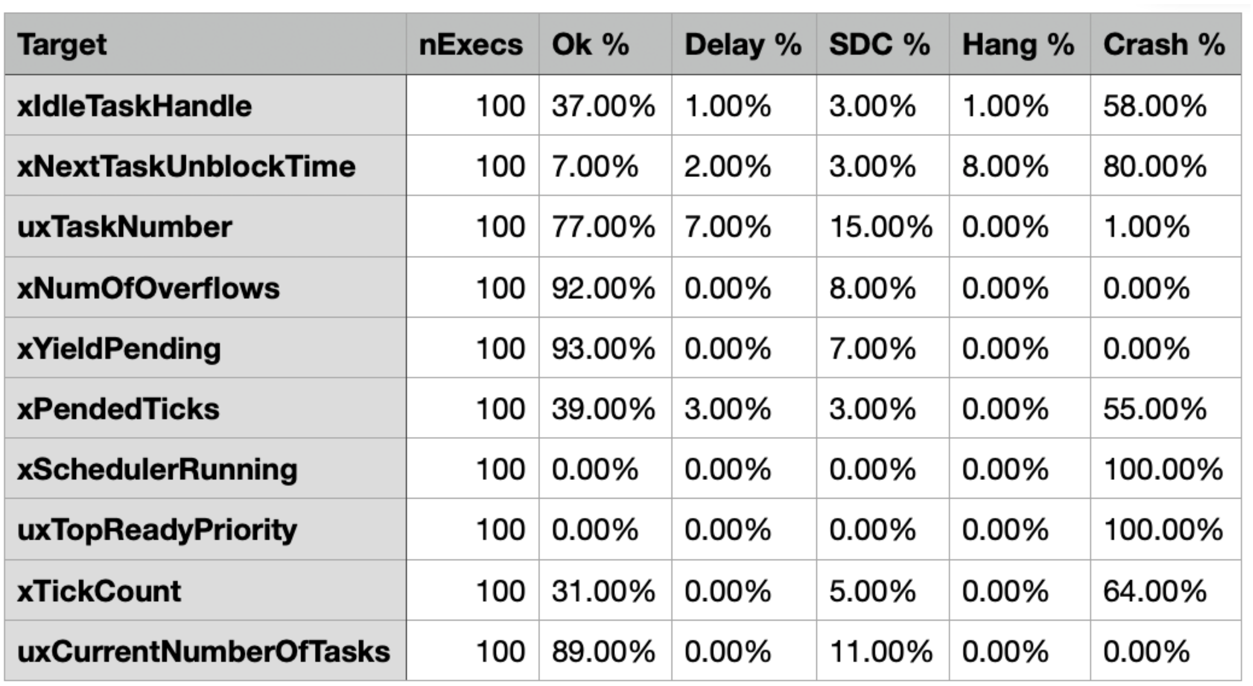
Below, several permanent fault injection campaigns will be shown, for which, for each benchmark, there will be a permanent fault injection campaign that exploits uniform distributions (for the selection of time instants) and a campaign that exploits Gaussian distributions; moreover, for each benchmark, there will be a division into operational groups.

6.4 Results of permanent fault injection campaigns



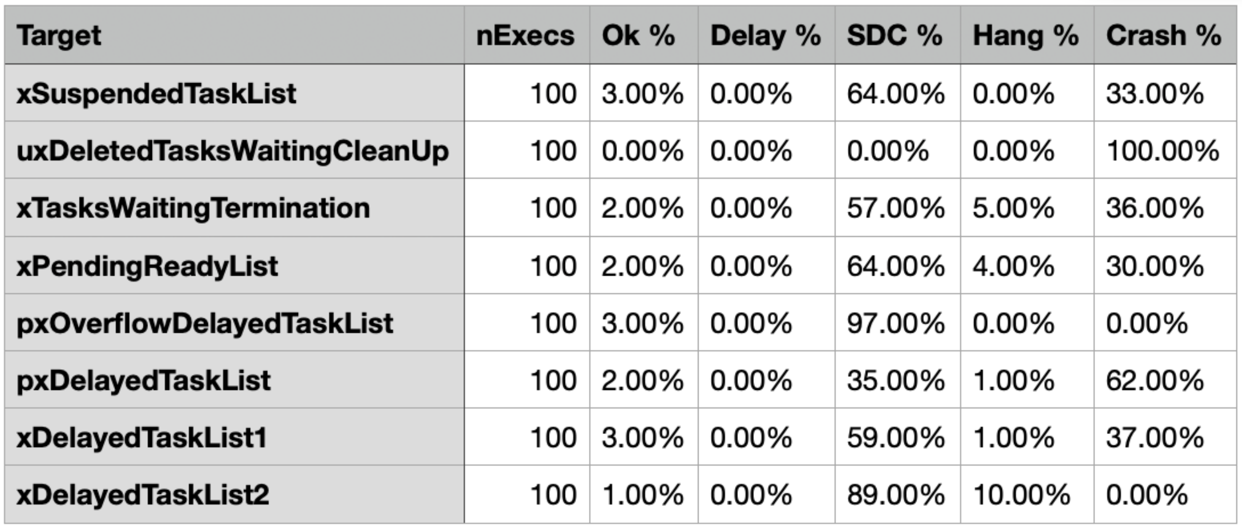
Target	nExecs	Ok %	Delay %	SDC %	Hang %	Crash %
xIdleTaskHandle	100	31.00%	1.00%	2.00%	1.00%	65.00%
xNextTaskUnblockTime	100	18.00%	3.00%	2.00%	20.00%	57.00%
uxTaskNumber	100	92.00%	0.00%	8.00%	0.00%	0.00%
xNumOfOverflows	100	93.00%	0.00%	7.00%	0.00%	0.00%
xYieldPending	100	97.00%	0.00%	3.00%	0.00%	0.00%
xPendedTicks	100	28.00%	4.00%	0.00%	0.00%	68.00%
xSchedulerRunning	100	0.00%	0.00%	0.00%	0.00%	100.00%
uxTopReadyPriority	100	0.00%	0.00%	0.00%	0.00%	100.00%
xTickCount	100	22.00%	0.00%	0.00%	0.00%	78.00%
uxCurrentNumberOfTasks	100	95.00%	0.00%	5.00%	0.00%	0.00%

Figure 6.1: System State Variables - uniform distribution - permanent injection - Bitcount



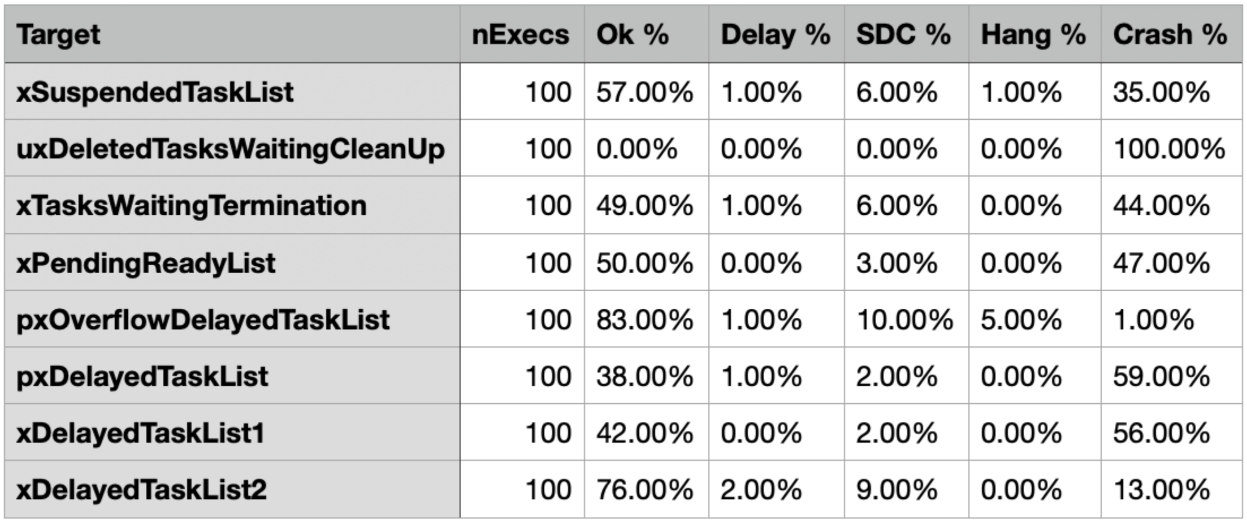
Target	nExecs	Ok %	Delay %	SDC %	Hang %	Crash %
xIdleTaskHandle	100	37.00%	1.00%	3.00%	1.00%	58.00%
xNextTaskUnblockTime	100	7.00%	2.00%	3.00%	8.00%	80.00%
uxTaskNumber	100	77.00%	7.00%	15.00%	0.00%	1.00%
xNumOfOverflows	100	92.00%	0.00%	8.00%	0.00%	0.00%
xYieldPending	100	93.00%	0.00%	7.00%	0.00%	0.00%
xPendedTicks	100	39.00%	3.00%	3.00%	0.00%	55.00%
xSchedulerRunning	100	0.00%	0.00%	0.00%	0.00%	100.00%
uxTopReadyPriority	100	0.00%	0.00%	0.00%	0.00%	100.00%
xTickCount	100	31.00%	0.00%	5.00%	0.00%	64.00%
uxCurrentNumberOfTasks	100	89.00%	0.00%	11.00%	0.00%	0.00%

Figure 6.2: System State Variables - gaussian distribution - permanent injection - Bitcount



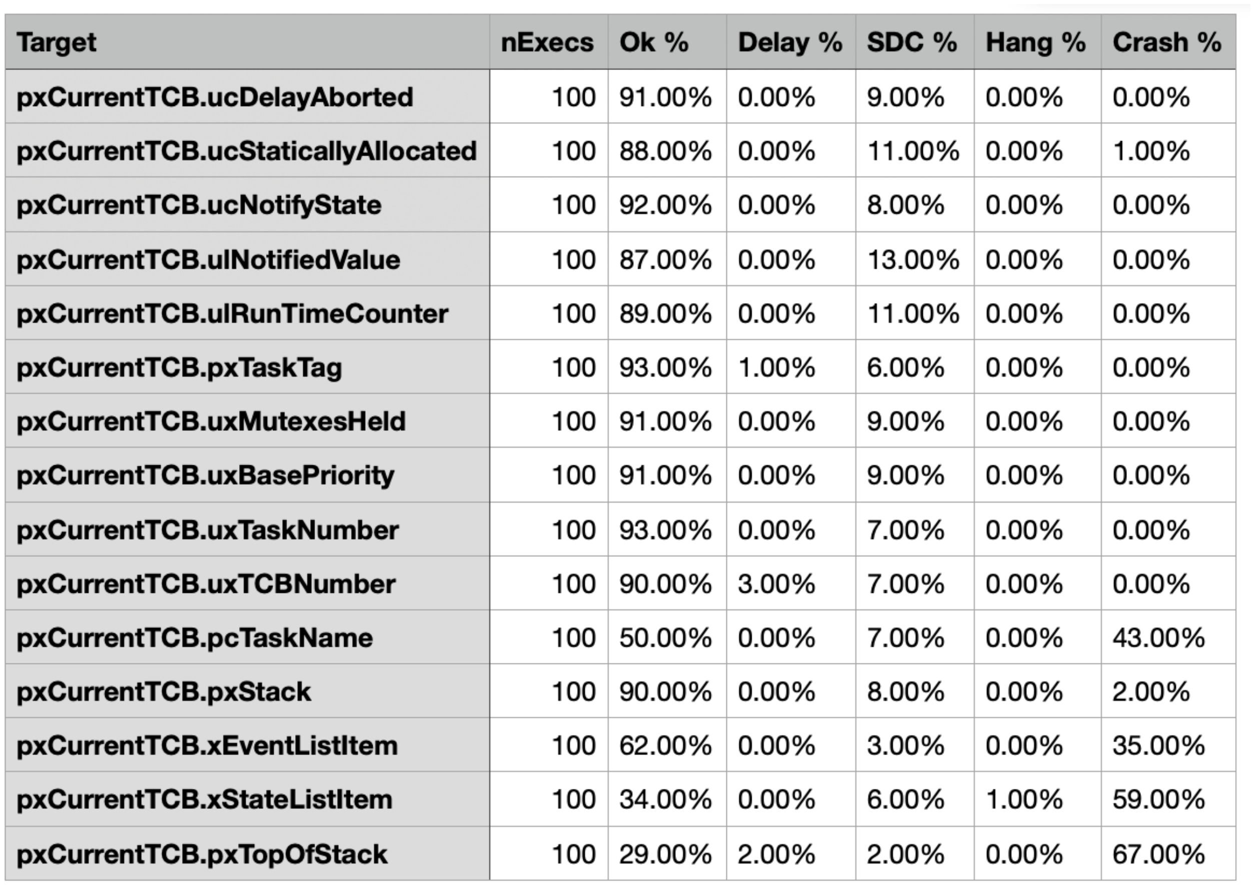
Target	nExecs	Ok %	Delay %	SDC %	Hang %	Crash %
xSuspendedTaskList	100	3.00%	0.00%	64.00%	0.00%	33.00%
uxDeletedTasksWaitingCleanUp	100	0.00%	0.00%	0.00%	0.00%	100.00%
xTasksWaitingTermination	100	2.00%	0.00%	57.00%	5.00%	36.00%
xPendingReadyList	100	2.00%	0.00%	64.00%	4.00%	30.00%
pxOverflowDelayedTaskList	100	3.00%	0.00%	97.00%	0.00%	0.00%
pxDelayedTaskList	100	2.00%	0.00%	35.00%	1.00%	62.00%
xDelayedTaskList1	100	3.00%	0.00%	59.00%	1.00%	37.00%
xDelayedTaskList2	100	1.00%	0.00%	89.00%	10.00%	0.00%

Figure 6.3: Task Lists - uniform distribution - permanent injection - Bitcount



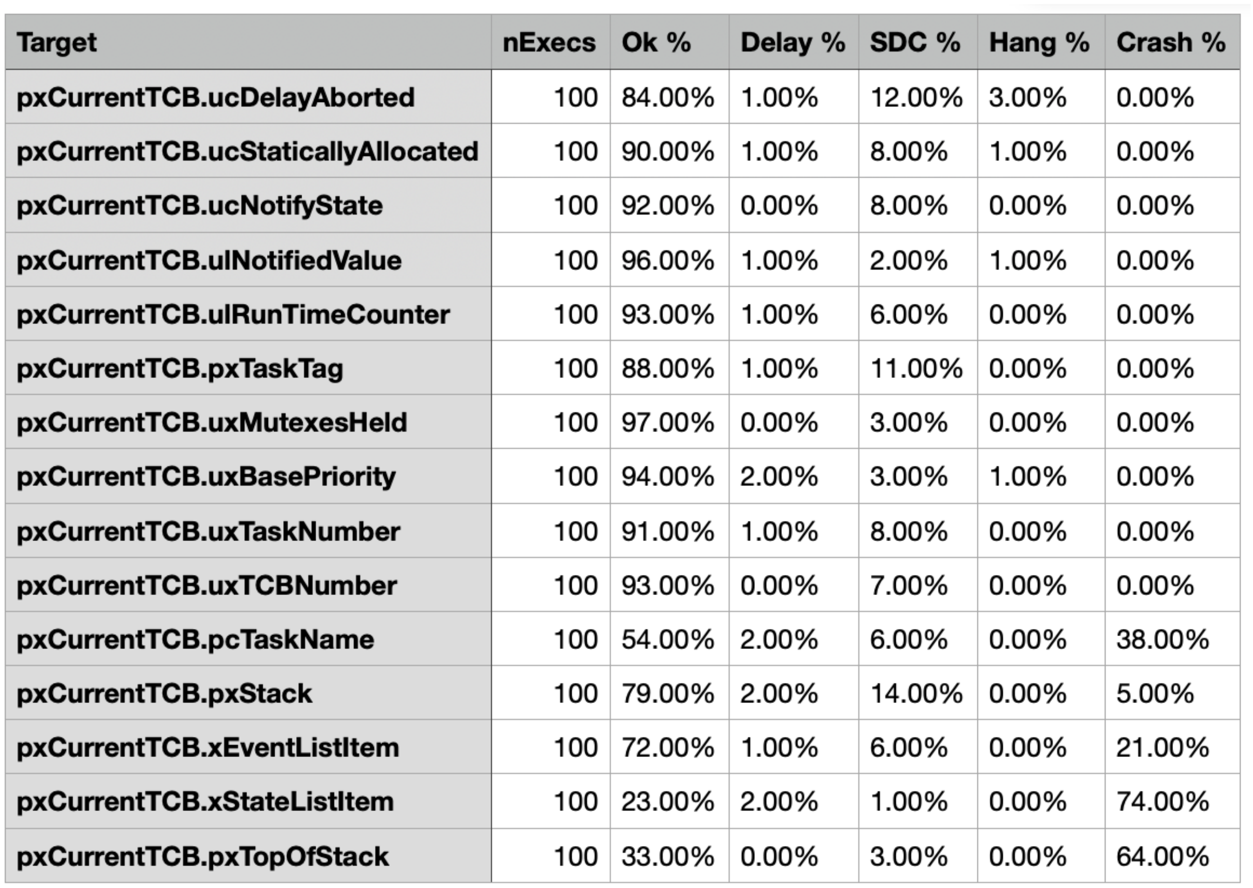
Target	nExecs	Ok %	Delay %	SDC %	Hang %	Crash %
xSuspendedTaskList	100	57.00%	1.00%	6.00%	1.00%	35.00%
uxDeletedTasksWaitingCleanUp	100	0.00%	0.00%	0.00%	0.00%	100.00%
xTasksWaitingTermination	100	49.00%	1.00%	6.00%	0.00%	44.00%
xPendingReadyList	100	50.00%	0.00%	3.00%	0.00%	47.00%
pxOverflowDelayedTaskList	100	83.00%	1.00%	10.00%	5.00%	1.00%
pxDelayedTaskList	100	38.00%	1.00%	2.00%	0.00%	59.00%
xDelayedTaskList1	100	42.00%	0.00%	2.00%	0.00%	56.00%
xDelayedTaskList2	100	76.00%	2.00%	9.00%	0.00%	13.00%

Figure 6.4: Task Lists - gaussian distribution - permanent injection - Bitcount



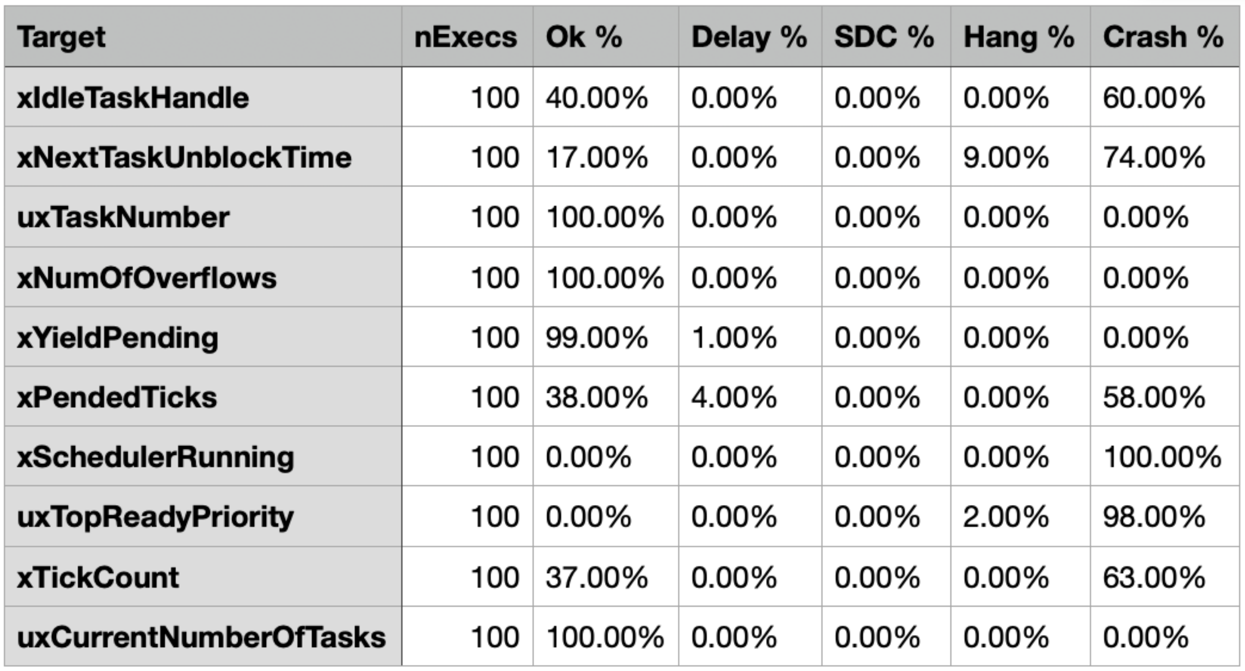
Target	nExecs	Ok %	Delay %	SDC %	Hang %	Crash %
pxCurrentTCB.ucDelayAborted	100	91.00%	0.00%	9.00%	0.00%	0.00%
pxCurrentTCB.ucStaticallyAllocated	100	88.00%	0.00%	11.00%	0.00%	1.00%
pxCurrentTCB.ucNotifyState	100	92.00%	0.00%	8.00%	0.00%	0.00%
pxCurrentTCB.ulNotifiedValue	100	87.00%	0.00%	13.00%	0.00%	0.00%
pxCurrentTCB.ulRunTimeCounter	100	89.00%	0.00%	11.00%	0.00%	0.00%
pxCurrentTCB.pxTaskTag	100	93.00%	1.00%	6.00%	0.00%	0.00%
pxCurrentTCB.uxMutexesHeld	100	91.00%	0.00%	9.00%	0.00%	0.00%
pxCurrentTCB.uxBasePriority	100	91.00%	0.00%	9.00%	0.00%	0.00%
pxCurrentTCB.uxTaskNumber	100	93.00%	0.00%	7.00%	0.00%	0.00%
pxCurrentTCB.uxTCBNumber	100	90.00%	3.00%	7.00%	0.00%	0.00%
pxCurrentTCB.pcTaskName	100	50.00%	0.00%	7.00%	0.00%	43.00%
pxCurrentTCB.pxStack	100	90.00%	0.00%	8.00%	0.00%	2.00%
pxCurrentTCB.xEventListItem	100	62.00%	0.00%	3.00%	0.00%	35.00%
pxCurrentTCB.xStateListItem	100	34.00%	0.00%	6.00%	1.00%	59.00%
pxCurrentTCB.pxTopOfStack	100	29.00%	2.00%	2.00%	0.00%	67.00%

Figure 6.5: Current TCB - uniform distribution - permanent injection - Bitcount



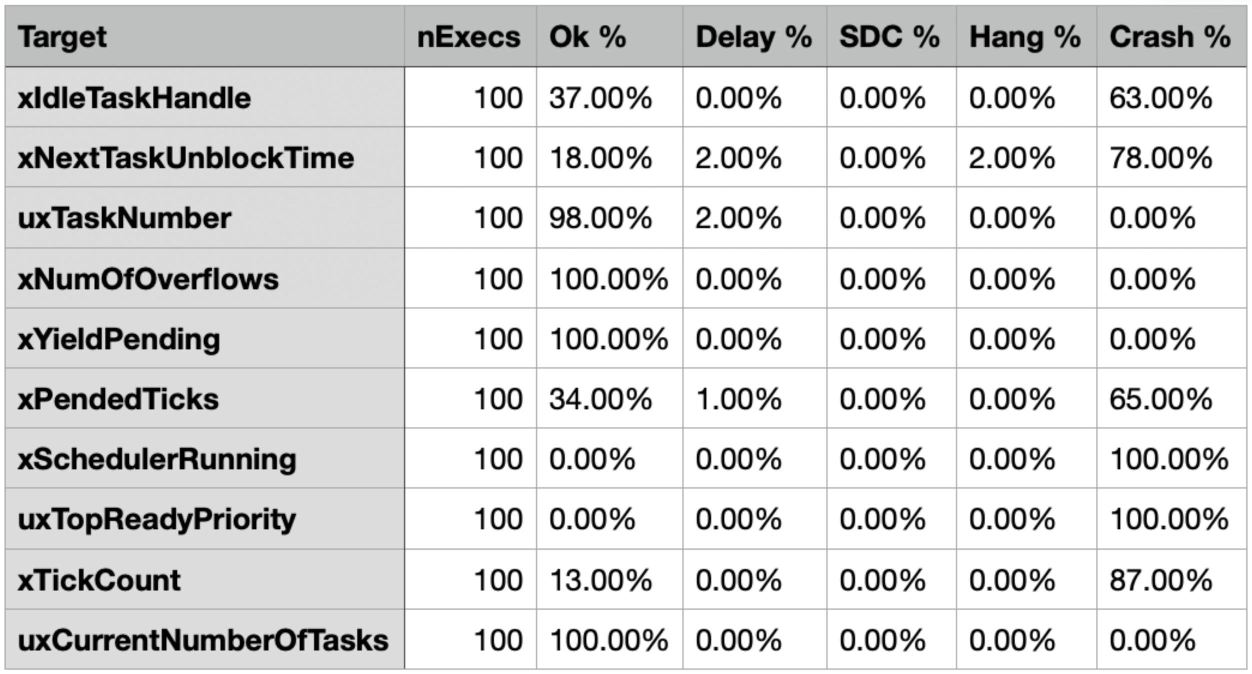
Target	nExecs	Ok %	Delay %	SDC %	Hang %	Crash %
pxCurrentTCB.ucDelayAborted	100	84.00%	1.00%	12.00%	3.00%	0.00%
pxCurrentTCB.ucStaticallyAllocated	100	90.00%	1.00%	8.00%	1.00%	0.00%
pxCurrentTCB.ucNotifyState	100	92.00%	0.00%	8.00%	0.00%	0.00%
pxCurrentTCB.ulNotifiedValue	100	96.00%	1.00%	2.00%	1.00%	0.00%
pxCurrentTCB.ulRunTimeCounter	100	93.00%	1.00%	6.00%	0.00%	0.00%
pxCurrentTCB.pxTaskTag	100	88.00%	1.00%	11.00%	0.00%	0.00%
pxCurrentTCB.uxMutexesHeld	100	97.00%	0.00%	3.00%	0.00%	0.00%
pxCurrentTCB.uxBasePriority	100	94.00%	2.00%	3.00%	1.00%	0.00%
pxCurrentTCB.uxTaskNumber	100	91.00%	1.00%	8.00%	0.00%	0.00%
pxCurrentTCB.uxTCBNumber	100	93.00%	0.00%	7.00%	0.00%	0.00%
pxCurrentTCB.pcTaskName	100	54.00%	2.00%	6.00%	0.00%	38.00%
pxCurrentTCB.pxStack	100	79.00%	2.00%	14.00%	0.00%	5.00%
pxCurrentTCB.xEventListItem	100	72.00%	1.00%	6.00%	0.00%	21.00%
pxCurrentTCB.xStateListItem	100	23.00%	2.00%	1.00%	0.00%	74.00%
pxCurrentTCB.pxTopOfStack	100	33.00%	0.00%	3.00%	0.00%	64.00%

Figure 6.6: Current TCB - gaussian distribution - permanent injection - Bitcount



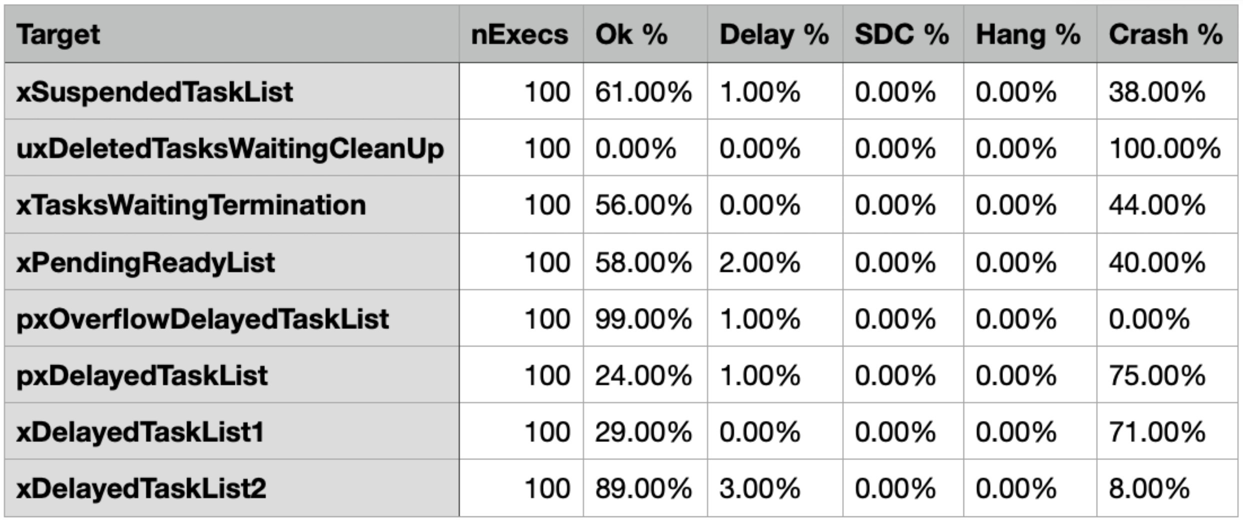
Target	nExecs	Ok %	Delay %	SDC %	Hang %	Crash %
xIdleTaskHandle	100	40.00%	0.00%	0.00%	0.00%	60.00%
xNextTaskUnblockTime	100	17.00%	0.00%	0.00%	9.00%	74.00%
uxTaskNumber	100	100.00%	0.00%	0.00%	0.00%	0.00%
xNumOfOverflows	100	100.00%	0.00%	0.00%	0.00%	0.00%
xYieldPending	100	99.00%	1.00%	0.00%	0.00%	0.00%
xPendedTicks	100	38.00%	4.00%	0.00%	0.00%	58.00%
xSchedulerRunning	100	0.00%	0.00%	0.00%	0.00%	100.00%
uxTopReadyPriority	100	0.00%	0.00%	0.00%	2.00%	98.00%
xTickCount	100	37.00%	0.00%	0.00%	0.00%	63.00%
uxCurrentNumberOfTasks	100	100.00%	0.00%	0.00%	0.00%	0.00%

Figure 6.7: System State Variables - uniform distribution - permanent injection - Qsort



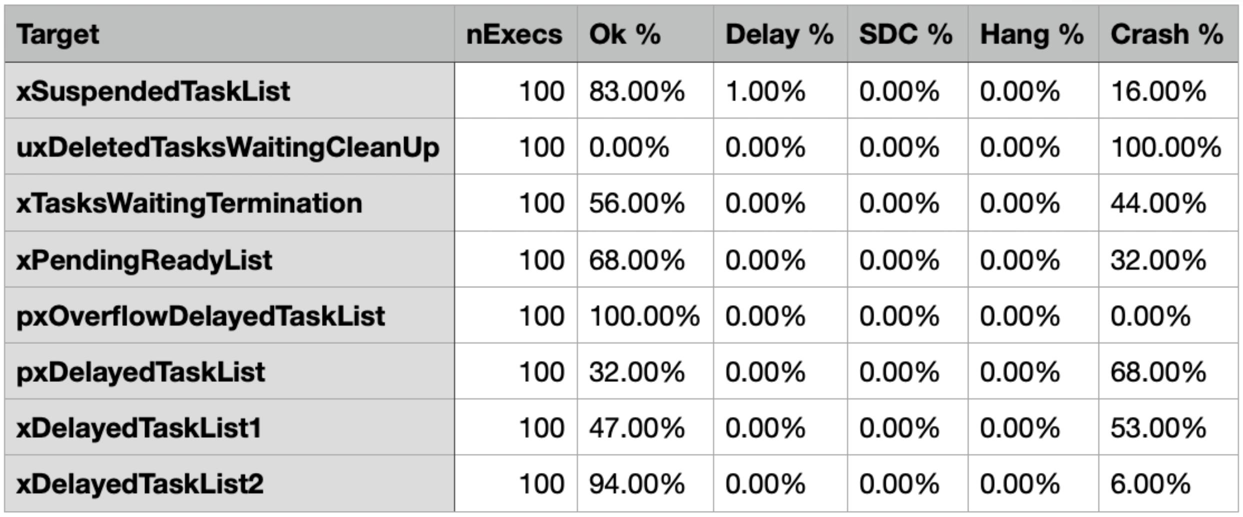
Target	nExecs	Ok %	Delay %	SDC %	Hang %	Crash %
xIdleTaskHandle	100	37.00%	0.00%	0.00%	0.00%	63.00%
xNextTaskUnblockTime	100	18.00%	2.00%	0.00%	2.00%	78.00%
uxTaskNumber	100	98.00%	2.00%	0.00%	0.00%	0.00%
xNumOfOverflows	100	100.00%	0.00%	0.00%	0.00%	0.00%
xYieldPending	100	100.00%	0.00%	0.00%	0.00%	0.00%
xPendedTicks	100	34.00%	1.00%	0.00%	0.00%	65.00%
xSchedulerRunning	100	0.00%	0.00%	0.00%	0.00%	100.00%
uxTopReadyPriority	100	0.00%	0.00%	0.00%	0.00%	100.00%
xTickCount	100	13.00%	0.00%	0.00%	0.00%	87.00%
uxCurrentNumberOfTasks	100	100.00%	0.00%	0.00%	0.00%	0.00%

Figure 6.8: System State Variables - gaussian distribution - permanent injection - Qsort



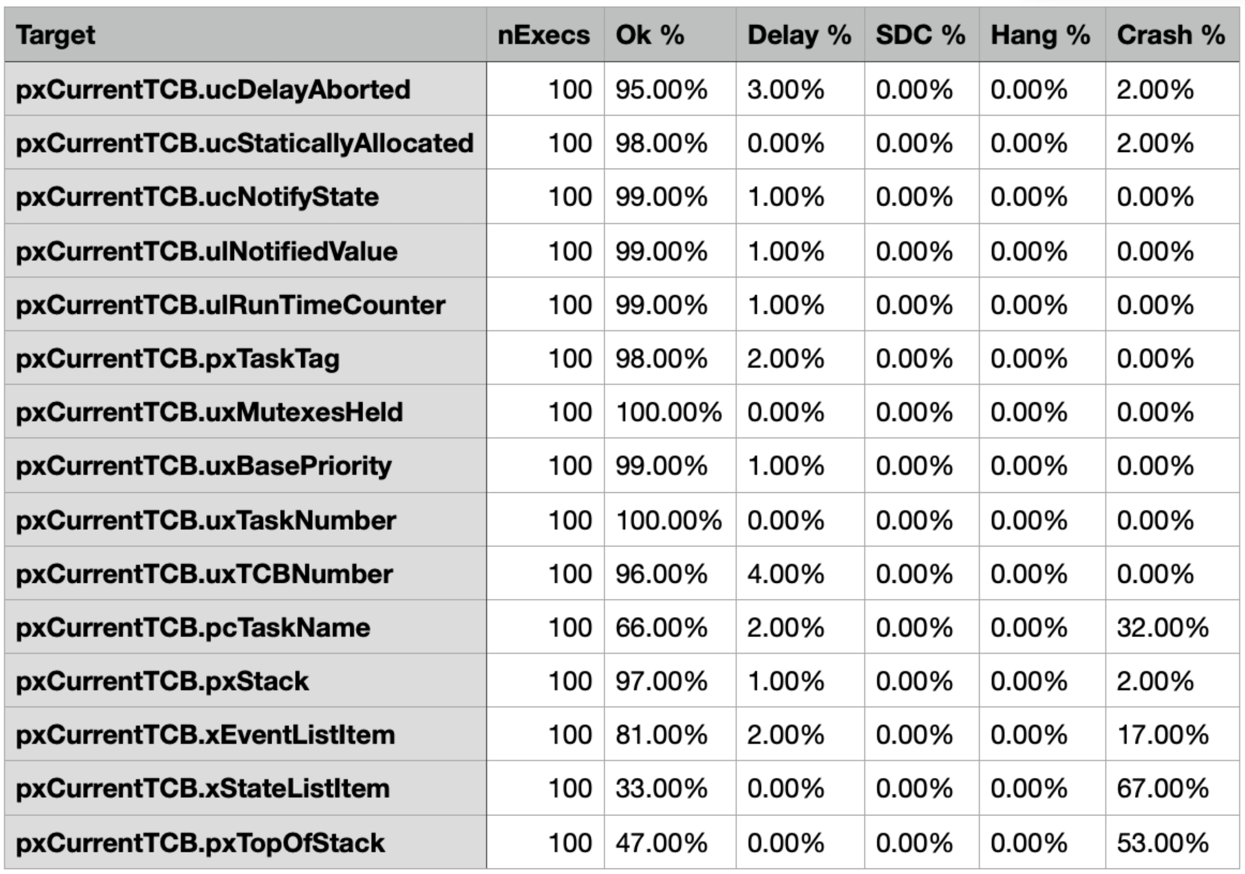
Target	nExecs	Ok %	Delay %	SDC %	Hang %	Crash %
xSuspendedTaskList	100	61.00%	1.00%	0.00%	0.00%	38.00%
uxDeletedTasksWaitingCleanUp	100	0.00%	0.00%	0.00%	0.00%	100.00%
xTasksWaitingTermination	100	56.00%	0.00%	0.00%	0.00%	44.00%
xPendingReadyList	100	58.00%	2.00%	0.00%	0.00%	40.00%
pxOverflowDelayedTaskList	100	99.00%	1.00%	0.00%	0.00%	0.00%
pxDelayedTaskList	100	24.00%	1.00%	0.00%	0.00%	75.00%
xDelayedTaskList1	100	29.00%	0.00%	0.00%	0.00%	71.00%
xDelayedTaskList2	100	89.00%	3.00%	0.00%	0.00%	8.00%

Figure 6.9: Task Lists - uniform distribution - permanent injection - Qsort



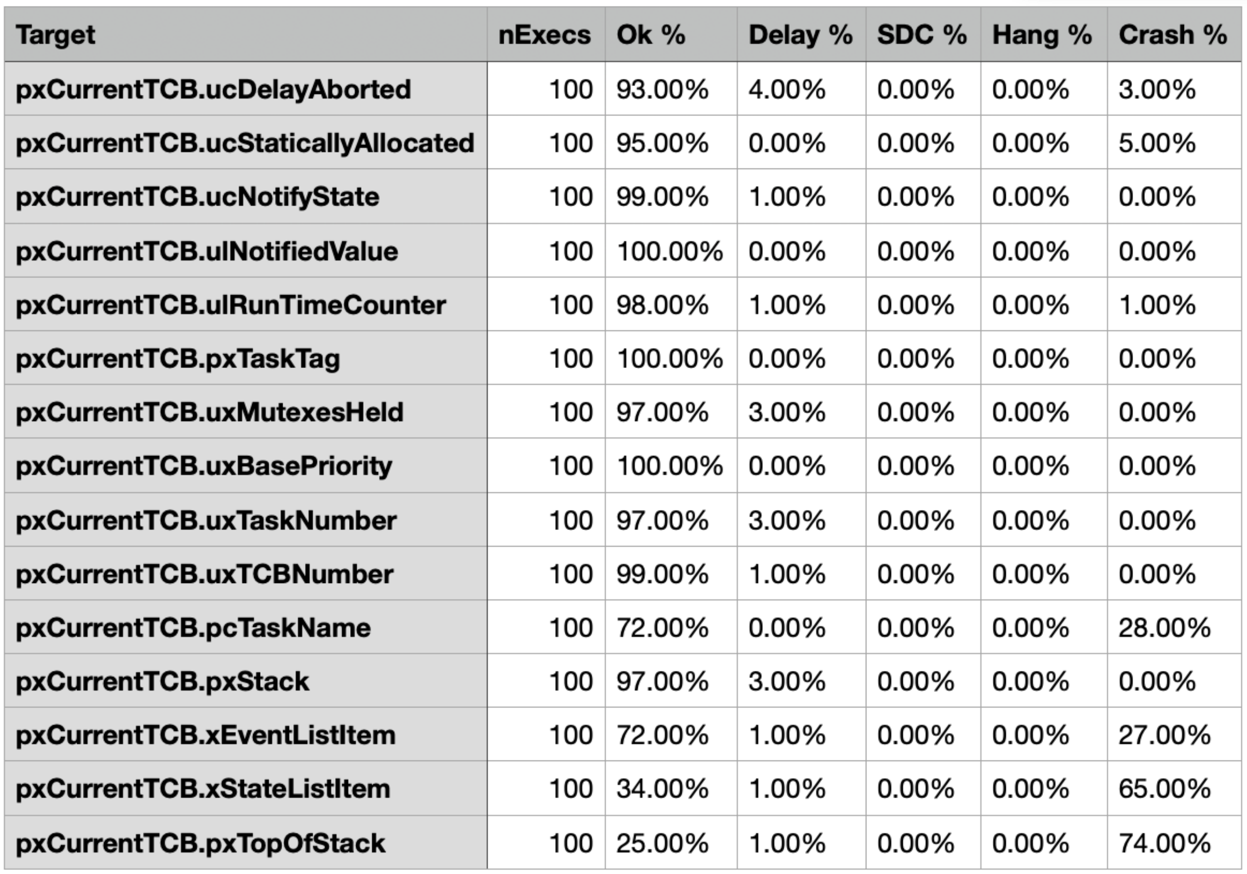
Target	nExecs	Ok %	Delay %	SDC %	Hang %	Crash %
xSuspendedTaskList	100	83.00%	1.00%	0.00%	0.00%	16.00%
uxDeletedTasksWaitingCleanUp	100	0.00%	0.00%	0.00%	0.00%	100.00%
xTasksWaitingTermination	100	56.00%	0.00%	0.00%	0.00%	44.00%
xPendingReadyList	100	68.00%	0.00%	0.00%	0.00%	32.00%
pxOverflowDelayedTaskList	100	100.00%	0.00%	0.00%	0.00%	0.00%
pxDelayedTaskList	100	32.00%	0.00%	0.00%	0.00%	68.00%
xDelayedTaskList1	100	47.00%	0.00%	0.00%	0.00%	53.00%
xDelayedTaskList2	100	94.00%	0.00%	0.00%	0.00%	6.00%

Figure 6.10: Task Lists - gaussian distribution - permanent injection - Qsort



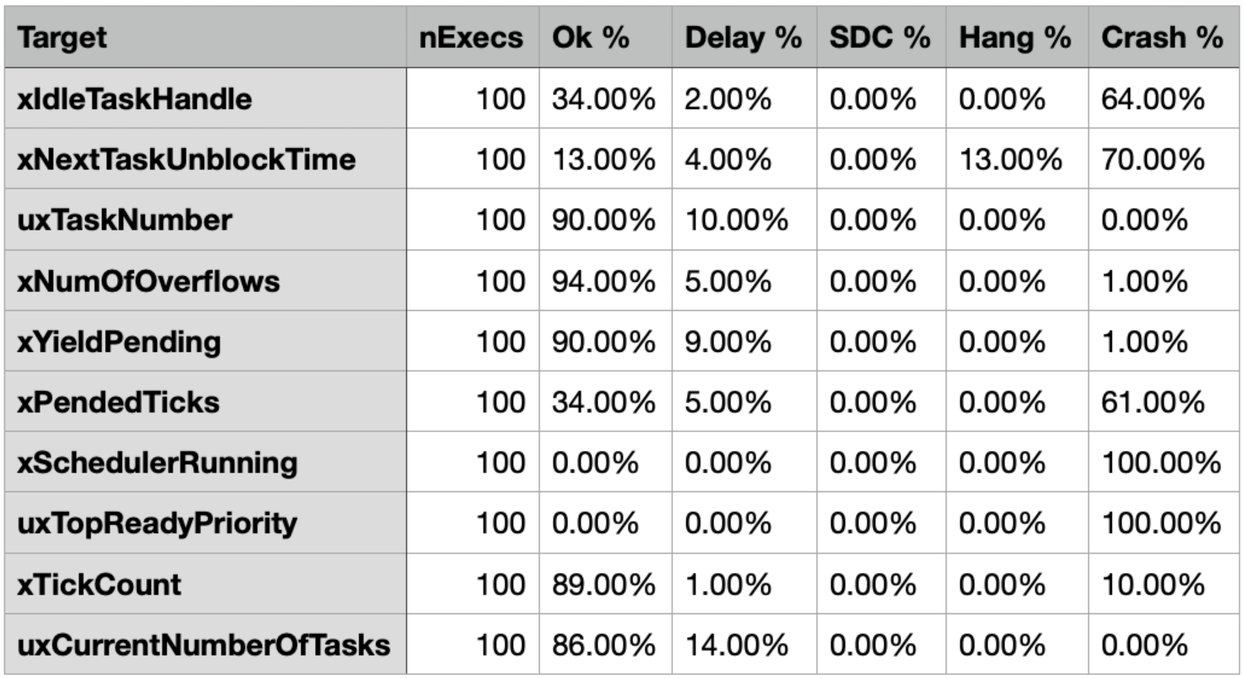
Target	nExecs	Ok %	Delay %	SDC %	Hang %	Crash %
pxCurrentTCB.ucDelayAborted	100	95.00%	3.00%	0.00%	0.00%	2.00%
pxCurrentTCB.ucStaticallyAllocated	100	98.00%	0.00%	0.00%	0.00%	2.00%
pxCurrentTCB.ucNotifyState	100	99.00%	1.00%	0.00%	0.00%	0.00%
pxCurrentTCB.ulNotifiedValue	100	99.00%	1.00%	0.00%	0.00%	0.00%
pxCurrentTCB.ulRunTimeCounter	100	99.00%	1.00%	0.00%	0.00%	0.00%
pxCurrentTCB.pxTaskTag	100	98.00%	2.00%	0.00%	0.00%	0.00%
pxCurrentTCB.uxMutexesHeld	100	100.00%	0.00%	0.00%	0.00%	0.00%
pxCurrentTCB.uxBasePriority	100	99.00%	1.00%	0.00%	0.00%	0.00%
pxCurrentTCB.uxTaskNumber	100	100.00%	0.00%	0.00%	0.00%	0.00%
pxCurrentTCB.uxTCBNumber	100	96.00%	4.00%	0.00%	0.00%	0.00%
pxCurrentTCB.pcTaskName	100	66.00%	2.00%	0.00%	0.00%	32.00%
pxCurrentTCB.pxStack	100	97.00%	1.00%	0.00%	0.00%	2.00%
pxCurrentTCB.xEventListItem	100	81.00%	2.00%	0.00%	0.00%	17.00%
pxCurrentTCB.xStateListItem	100	33.00%	0.00%	0.00%	0.00%	67.00%
pxCurrentTCB.pxTopOfStack	100	47.00%	0.00%	0.00%	0.00%	53.00%

Figure 6.11: Current TCB - uniform distribution - permanent injection - Qsort



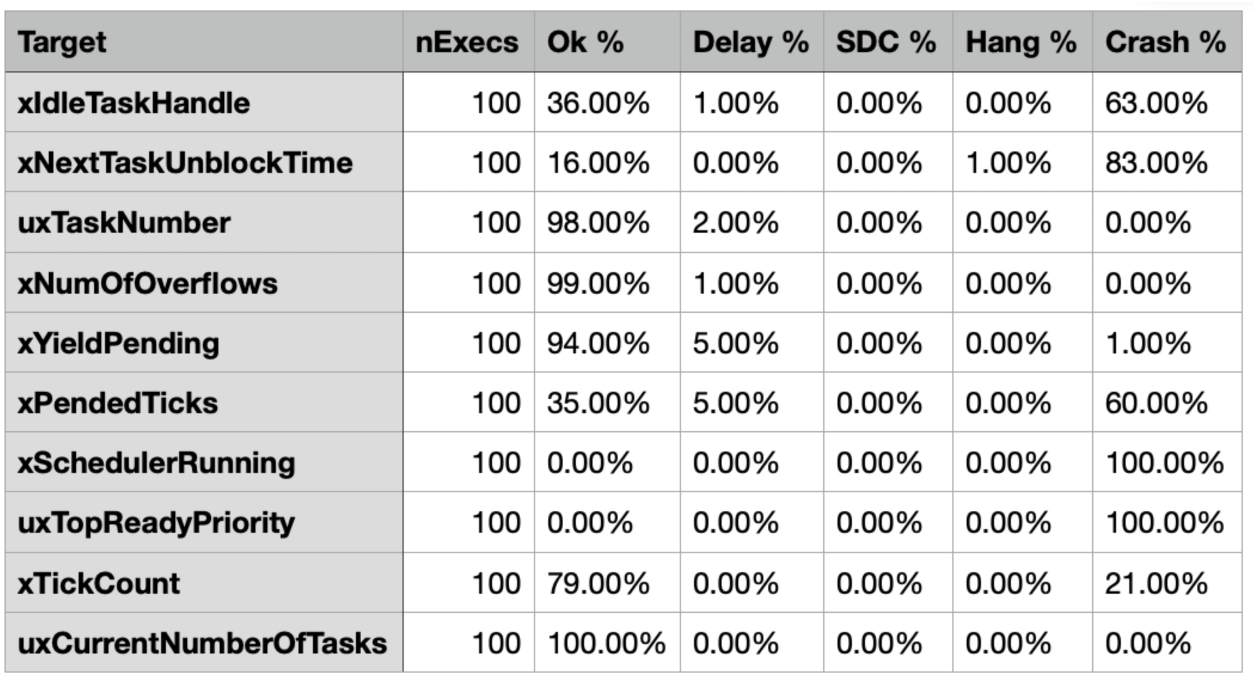
Target	nExecs	Ok %	Delay %	SDC %	Hang %	Crash %
pxCurrentTCB.ucDelayAborted	100	93.00%	4.00%	0.00%	0.00%	3.00%
pxCurrentTCB.ucStaticallyAllocated	100	95.00%	0.00%	0.00%	0.00%	5.00%
pxCurrentTCB.ucNotifyState	100	99.00%	1.00%	0.00%	0.00%	0.00%
pxCurrentTCB.ulNotifiedValue	100	100.00%	0.00%	0.00%	0.00%	0.00%
pxCurrentTCB.ulRunTimeCounter	100	98.00%	1.00%	0.00%	0.00%	1.00%
pxCurrentTCB.pxTaskTag	100	100.00%	0.00%	0.00%	0.00%	0.00%
pxCurrentTCB.uxMutexesHeld	100	97.00%	3.00%	0.00%	0.00%	0.00%
pxCurrentTCB.uxBasePriority	100	100.00%	0.00%	0.00%	0.00%	0.00%
pxCurrentTCB.uxTaskNumber	100	97.00%	3.00%	0.00%	0.00%	0.00%
pxCurrentTCB.uxTCBNumber	100	99.00%	1.00%	0.00%	0.00%	0.00%
pxCurrentTCB.pcTaskName	100	72.00%	0.00%	0.00%	0.00%	28.00%
pxCurrentTCB.pxStack	100	97.00%	3.00%	0.00%	0.00%	0.00%
pxCurrentTCB.xEventListItem	100	72.00%	1.00%	0.00%	0.00%	27.00%
pxCurrentTCB.xStateListItem	100	34.00%	1.00%	0.00%	0.00%	65.00%
pxCurrentTCB.pxTopOfStack	100	25.00%	1.00%	0.00%	0.00%	74.00%

Figure 6.12: Current TCB - gaussian distribution - permanent injection - Qsort



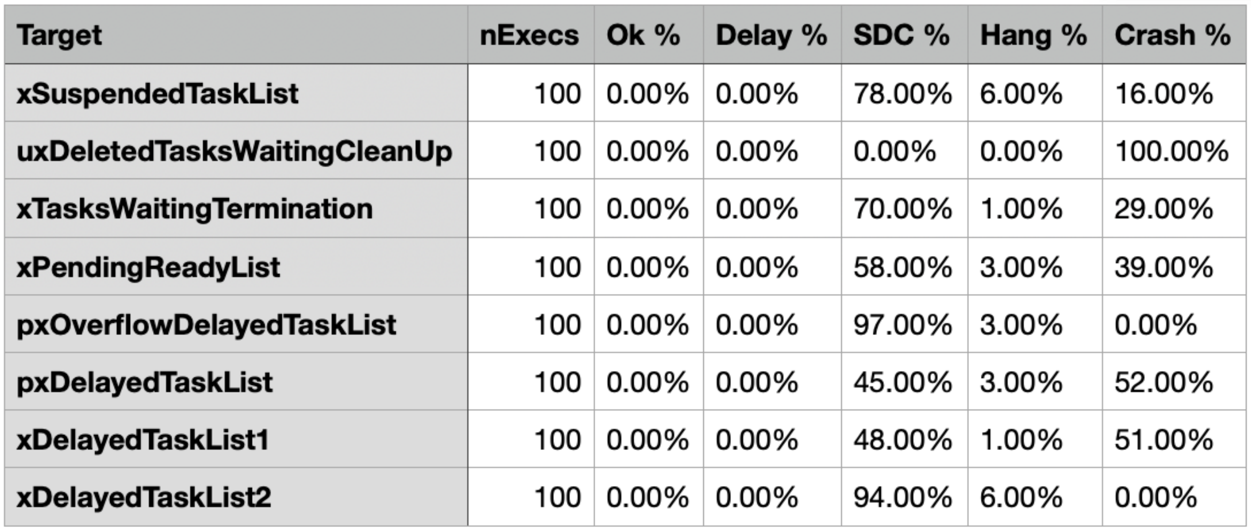
Target	nExecs	Ok %	Delay %	SDC %	Hang %	Crash %
xIdleTaskHandle	100	34.00%	2.00%	0.00%	0.00%	64.00%
xNextTaskUnblockTime	100	13.00%	4.00%	0.00%	13.00%	70.00%
uxTaskNumber	100	90.00%	10.00%	0.00%	0.00%	0.00%
xNumOfOverflows	100	94.00%	5.00%	0.00%	0.00%	1.00%
xYieldPending	100	90.00%	9.00%	0.00%	0.00%	1.00%
xPendedTicks	100	34.00%	5.00%	0.00%	0.00%	61.00%
xSchedulerRunning	100	0.00%	0.00%	0.00%	0.00%	100.00%
uxTopReadyPriority	100	0.00%	0.00%	0.00%	0.00%	100.00%
xTickCount	100	89.00%	1.00%	0.00%	0.00%	10.00%
uxCurrentNumberOfTasks	100	86.00%	14.00%	0.00%	0.00%	0.00%

Figure 6.13: System State Variables - uniform distribution - permanent injection - Basicmath



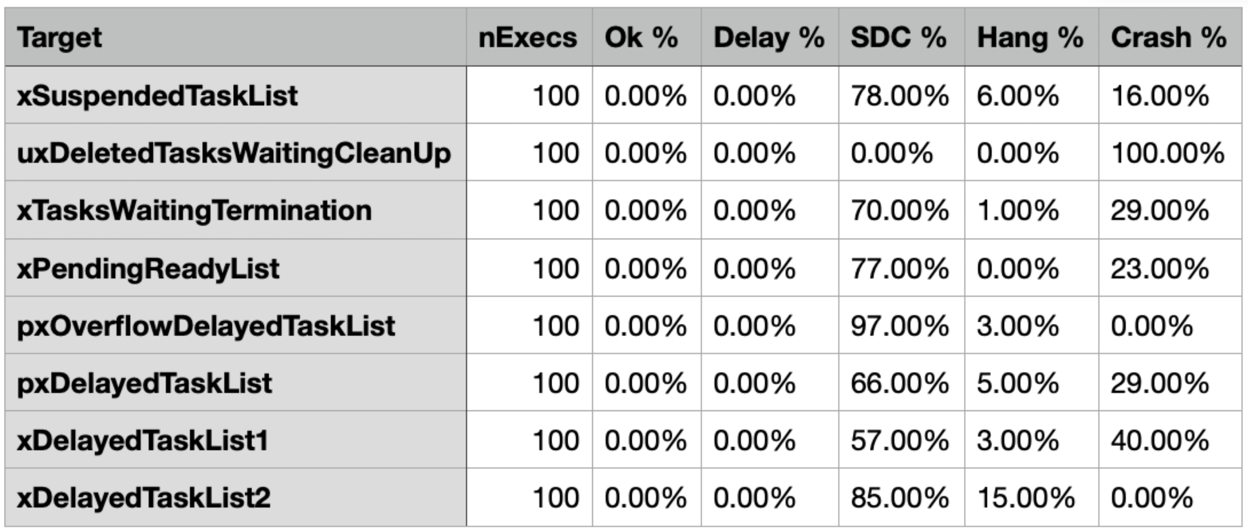
Target	nExecs	Ok %	Delay %	SDC %	Hang %	Crash %
xIdleTaskHandle	100	36.00%	1.00%	0.00%	0.00%	63.00%
xNextTaskUnblockTime	100	16.00%	0.00%	0.00%	1.00%	83.00%
uxTaskNumber	100	98.00%	2.00%	0.00%	0.00%	0.00%
xNumOfOverflows	100	99.00%	1.00%	0.00%	0.00%	0.00%
xYieldPending	100	94.00%	5.00%	0.00%	0.00%	1.00%
xPendedTicks	100	35.00%	5.00%	0.00%	0.00%	60.00%
xSchedulerRunning	100	0.00%	0.00%	0.00%	0.00%	100.00%
uxTopReadyPriority	100	0.00%	0.00%	0.00%	0.00%	100.00%
xTickCount	100	79.00%	0.00%	0.00%	0.00%	21.00%
uxCurrentNumberOfTasks	100	100.00%	0.00%	0.00%	0.00%	0.00%

Figure 6.14: System State Variables - gaussian distribution - permanent injection - Basicmath



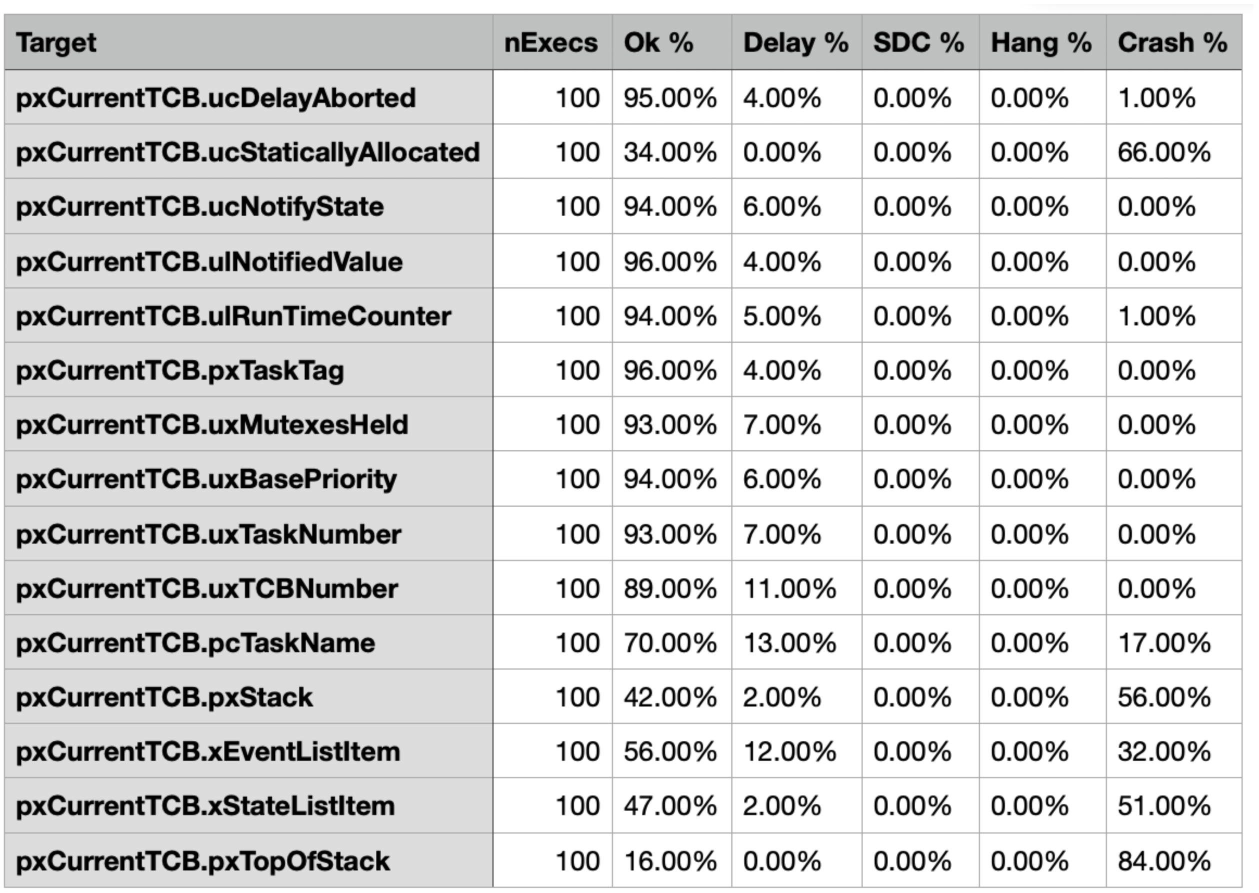
Target	nExecs	Ok %	Delay %	SDC %	Hang %	Crash %
xSuspendedTaskList	100	0.00%	0.00%	78.00%	6.00%	16.00%
uxDeletedTasksWaitingCleanUp	100	0.00%	0.00%	0.00%	0.00%	100.00%
xTasksWaitingTermination	100	0.00%	0.00%	70.00%	1.00%	29.00%
xPendingReadyList	100	0.00%	0.00%	58.00%	3.00%	39.00%
pxOverflowDelayedTaskList	100	0.00%	0.00%	97.00%	3.00%	0.00%
pxDelayedTaskList	100	0.00%	0.00%	45.00%	3.00%	52.00%
xDelayedTaskList1	100	0.00%	0.00%	48.00%	1.00%	51.00%
xDelayedTaskList2	100	0.00%	0.00%	94.00%	6.00%	0.00%

Figure 6.15: Task Lists - uniform distribution - permanent injection - Basicmath



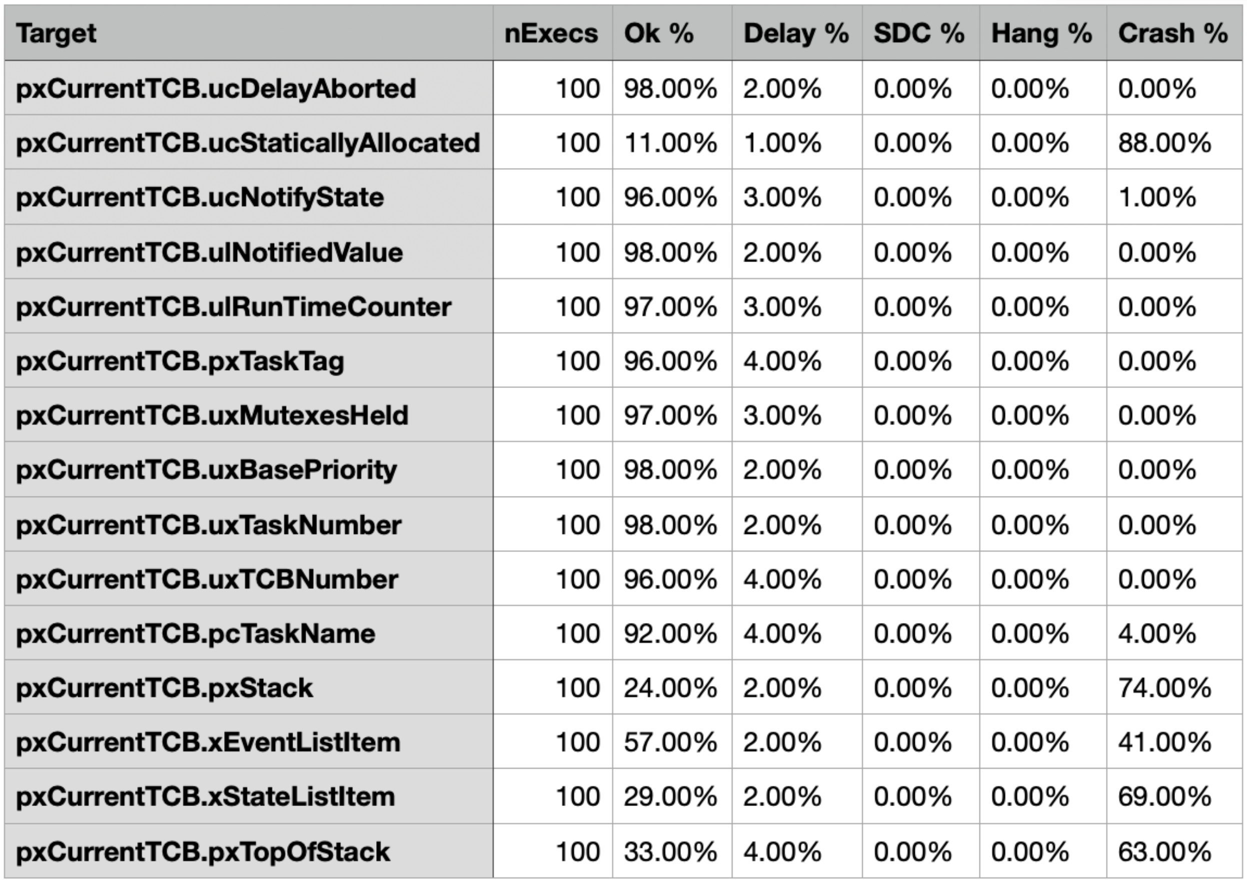
Target	nExecs	Ok %	Delay %	SDC %	Hang %	Crash %
xSuspendedTaskList	100	0.00%	0.00%	78.00%	6.00%	16.00%
uxDeletedTasksWaitingCleanUp	100	0.00%	0.00%	0.00%	0.00%	100.00%
xTasksWaitingTermination	100	0.00%	0.00%	70.00%	1.00%	29.00%
xPendingReadyList	100	0.00%	0.00%	77.00%	0.00%	23.00%
pxOverflowDelayedTaskList	100	0.00%	0.00%	97.00%	3.00%	0.00%
pxDelayedTaskList	100	0.00%	0.00%	66.00%	5.00%	29.00%
xDelayedTaskList1	100	0.00%	0.00%	57.00%	3.00%	40.00%
xDelayedTaskList2	100	0.00%	0.00%	85.00%	15.00%	0.00%

Figure 6.16: Task Lists - gaussian distribution - permanent injection - Basicmath



Target	nExecs	Ok %	Delay %	SDC %	Hang %	Crash %
pxCurrentTCB.ucDelayAborted	100	95.00%	4.00%	0.00%	0.00%	1.00%
pxCurrentTCB.ucStaticallyAllocated	100	34.00%	0.00%	0.00%	0.00%	66.00%
pxCurrentTCB.ucNotifyState	100	94.00%	6.00%	0.00%	0.00%	0.00%
pxCurrentTCB.ulNotifiedValue	100	96.00%	4.00%	0.00%	0.00%	0.00%
pxCurrentTCB.ulRunTimeCounter	100	94.00%	5.00%	0.00%	0.00%	1.00%
pxCurrentTCB.pxTaskTag	100	96.00%	4.00%	0.00%	0.00%	0.00%
pxCurrentTCB.uxMutexesHeld	100	93.00%	7.00%	0.00%	0.00%	0.00%
pxCurrentTCB.uxBasePriority	100	94.00%	6.00%	0.00%	0.00%	0.00%
pxCurrentTCB.uxTaskNumber	100	93.00%	7.00%	0.00%	0.00%	0.00%
pxCurrentTCB.uxTCBNumber	100	89.00%	11.00%	0.00%	0.00%	0.00%
pxCurrentTCB.pcTaskName	100	70.00%	13.00%	0.00%	0.00%	17.00%
pxCurrentTCB.pxStack	100	42.00%	2.00%	0.00%	0.00%	56.00%
pxCurrentTCB.xEventListItem	100	56.00%	12.00%	0.00%	0.00%	32.00%
pxCurrentTCB.xStateListItem	100	47.00%	2.00%	0.00%	0.00%	51.00%
pxCurrentTCB.pxTopOfStack	100	16.00%	0.00%	0.00%	0.00%	84.00%

Figure 6.17: Current TCB - uniform distribution - permanent injection - Basicmath



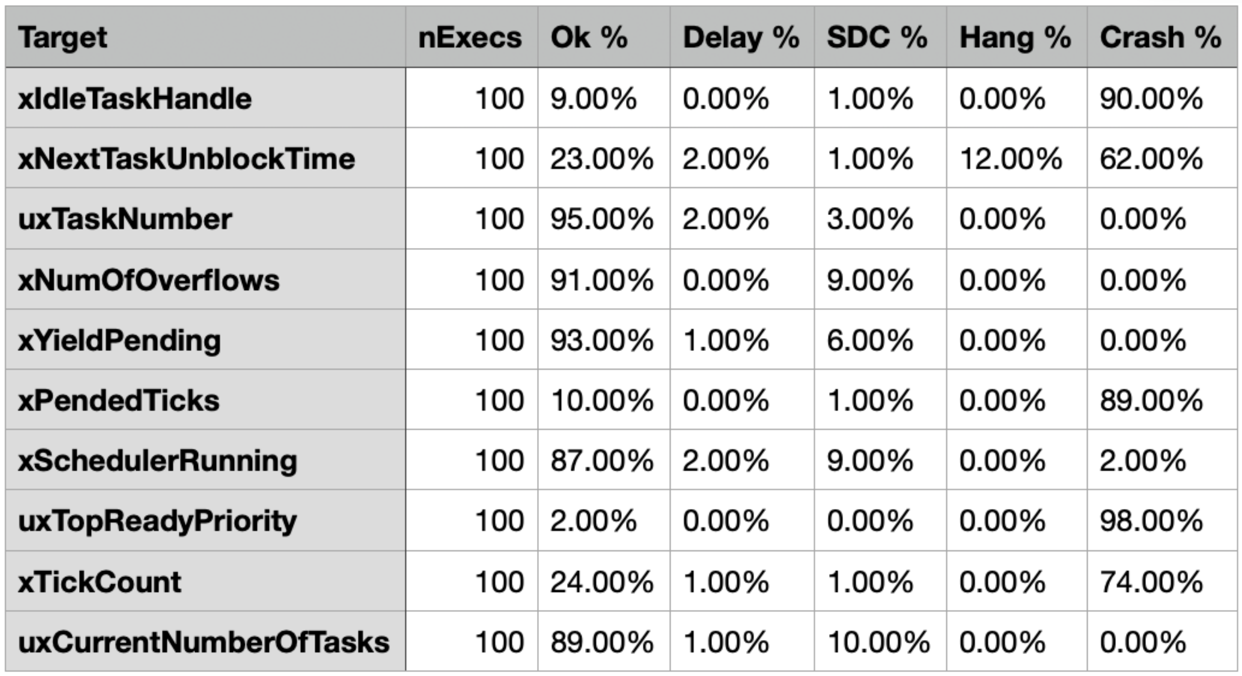
Target	nExecs	Ok %	Delay %	SDC %	Hang %	Crash %
pxCurrentTCB.ucDelayAborted	100	98.00%	2.00%	0.00%	0.00%	0.00%
pxCurrentTCB.ucStaticallyAllocated	100	11.00%	1.00%	0.00%	0.00%	88.00%
pxCurrentTCB.ucNotifyState	100	96.00%	3.00%	0.00%	0.00%	1.00%
pxCurrentTCB.ulNotifiedValue	100	98.00%	2.00%	0.00%	0.00%	0.00%
pxCurrentTCB.ulRunTimeCounter	100	97.00%	3.00%	0.00%	0.00%	0.00%
pxCurrentTCB.pxTaskTag	100	96.00%	4.00%	0.00%	0.00%	0.00%
pxCurrentTCB.uxMutexesHeld	100	97.00%	3.00%	0.00%	0.00%	0.00%
pxCurrentTCB.uxBasePriority	100	98.00%	2.00%	0.00%	0.00%	0.00%
pxCurrentTCB.uxTaskNumber	100	98.00%	2.00%	0.00%	0.00%	0.00%
pxCurrentTCB.uxTCBNumber	100	96.00%	4.00%	0.00%	0.00%	0.00%
pxCurrentTCB.pcTaskName	100	92.00%	4.00%	0.00%	0.00%	4.00%
pxCurrentTCB.pxStack	100	24.00%	2.00%	0.00%	0.00%	74.00%
pxCurrentTCB.xEventListItem	100	57.00%	2.00%	0.00%	0.00%	41.00%
pxCurrentTCB.xStateListItem	100	29.00%	2.00%	0.00%	0.00%	69.00%
pxCurrentTCB.pxTopOfStack	100	33.00%	4.00%	0.00%	0.00%	63.00%

Figure 6.18: Current TCB - gaussian distribution - permanent injection - Basicmath

It is possible from the previous tables to notice a very interesting aspect about the results obtained. Indeed, by observing the differences between similar campaigns that differ only in the distributions (used for the extraction of the injection times), we can see how the moment in which an SHE occurs in the execution interval of a certain FreeRTOS task can be decisive. Just look at the difference in SDC (Silent Data Corruption) between the uniform and the Gaussian with the Bitcount task, on the set of targets of the "Task Lists". With the uniform, we have percentages higher than 50% on all targets up to reaching 97%, while with the Gaussian, the peak is at 10%. In general, except in the case of Quicksort, it is possible to see high percentage values of SDC in the targets of the "Task Lists" and in general it is possible to see how with Quicksort there are never any SDCs and the percentage of Hang is often nil or less than 10% in those few targets that do not show a null percentage. It is also possible to notice that in general the crash percentage of the targets referable to the "current TCB" group is very low or nil, except for pcTaskName, xEventListItem, xStateListItem, pxTopOfStack. Targets from the "System State Variables" group like xSchedulerRunning and uxTopReadyPriority are very sensitive to SHEs, effectively causing the system crash with almost probability 1.

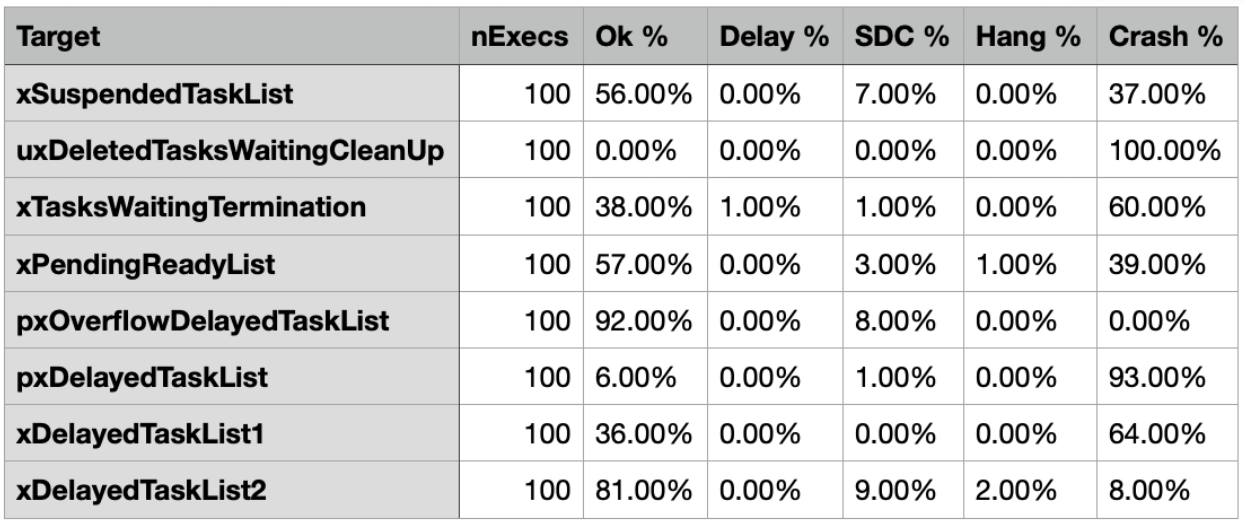
6.5 Results of transient fault injection campaigns

For completeness, transient fault injection campaigns using uniform distributions are shown below.



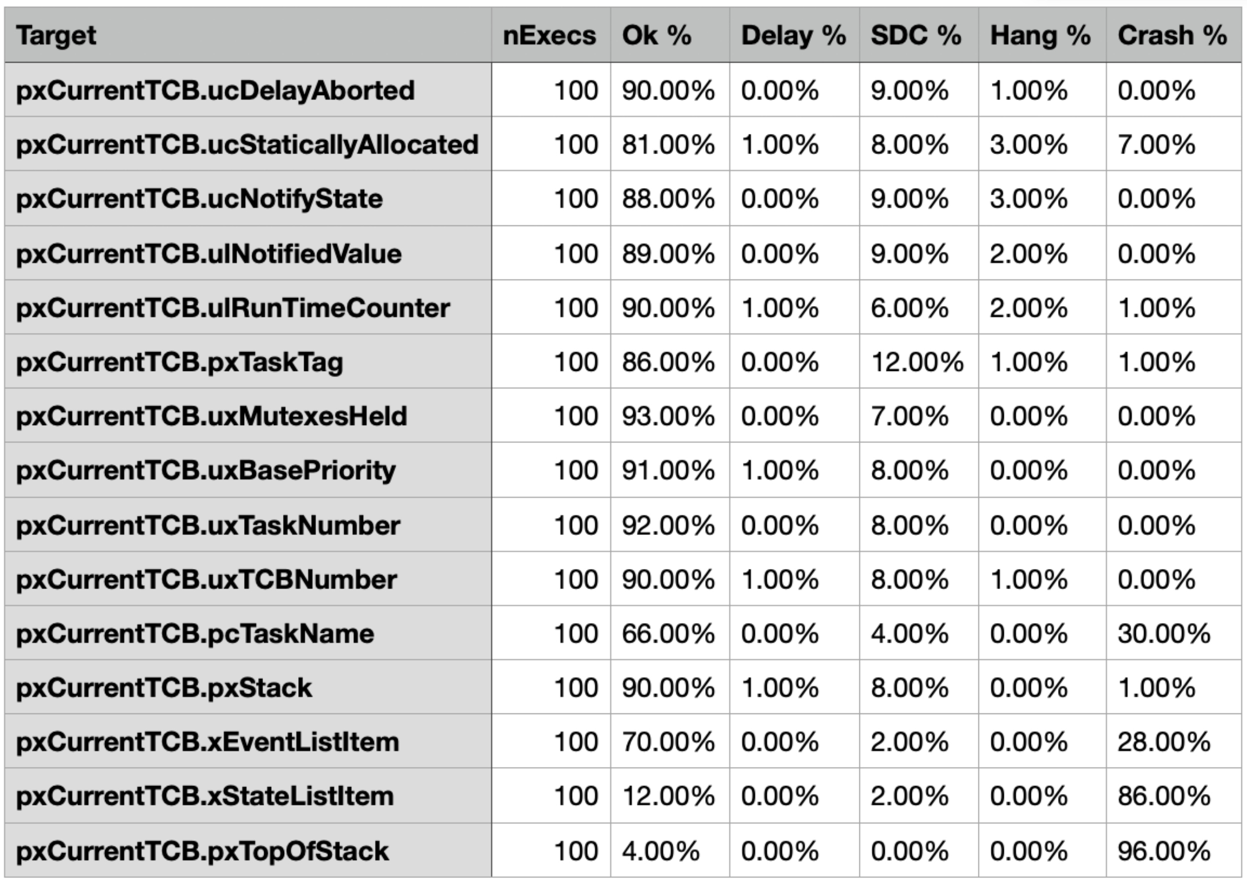
Target	nExecs	Ok %	Delay %	SDC %	Hang %	Crash %
xIdleTaskHandle	100	9.00%	0.00%	1.00%	0.00%	90.00%
xNextTaskUnblockTime	100	23.00%	2.00%	1.00%	12.00%	62.00%
uxTaskNumber	100	95.00%	2.00%	3.00%	0.00%	0.00%
xNumOfOverflows	100	91.00%	0.00%	9.00%	0.00%	0.00%
xYieldPending	100	93.00%	1.00%	6.00%	0.00%	0.00%
xPendedTicks	100	10.00%	0.00%	1.00%	0.00%	89.00%
xSchedulerRunning	100	87.00%	2.00%	9.00%	0.00%	2.00%
uxTopReadyPriority	100	2.00%	0.00%	0.00%	0.00%	98.00%
xTickCount	100	24.00%	1.00%	1.00%	0.00%	74.00%
uxCurrentNumberOfTasks	100	89.00%	1.00%	10.00%	0.00%	0.00%

Figure 6.19: System State Variables - uniform distribution - transient injection - Bitcount



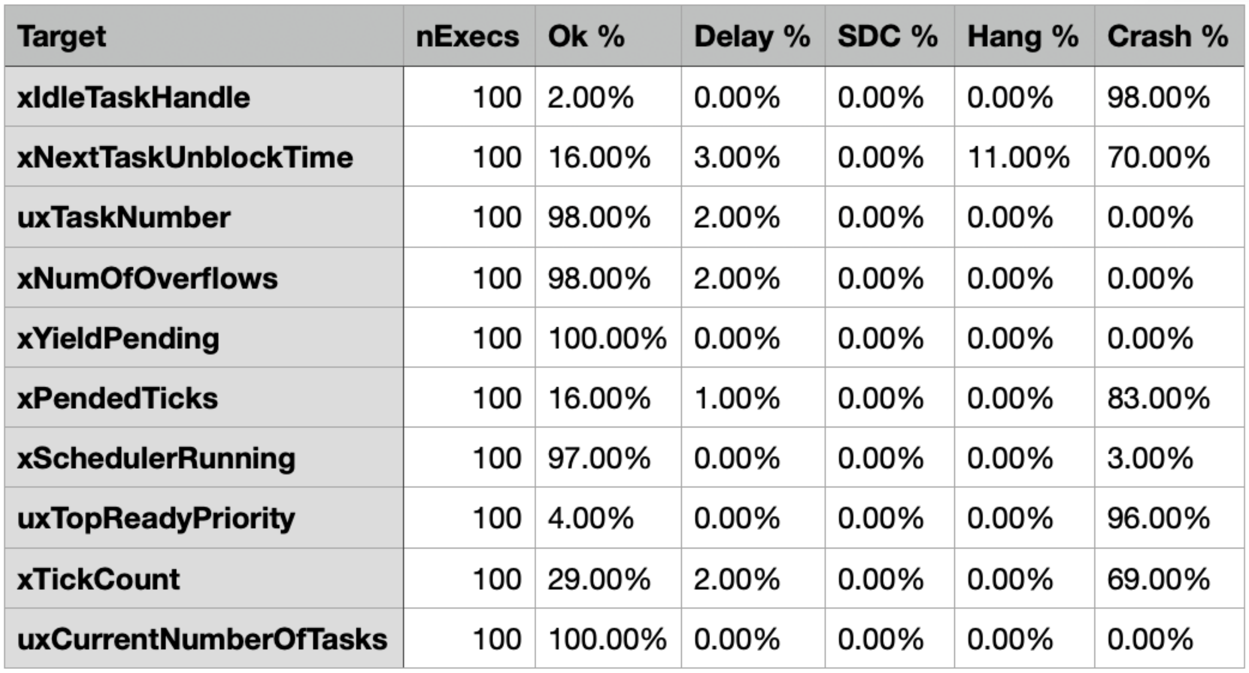
Target	nExecs	Ok %	Delay %	SDC %	Hang %	Crash %
xSuspendedTaskList	100	56.00%	0.00%	7.00%	0.00%	37.00%
uxDeletedTasksWaitingCleanUp	100	0.00%	0.00%	0.00%	0.00%	100.00%
xTasksWaitingTermination	100	38.00%	1.00%	1.00%	0.00%	60.00%
xPendingReadyList	100	57.00%	0.00%	3.00%	1.00%	39.00%
pxOverflowDelayedTaskList	100	92.00%	0.00%	8.00%	0.00%	0.00%
pxDelayedTaskList	100	6.00%	0.00%	1.00%	0.00%	93.00%
xDelayedTaskList1	100	36.00%	0.00%	0.00%	0.00%	64.00%
xDelayedTaskList2	100	81.00%	0.00%	9.00%	2.00%	8.00%

Figure 6.20: Task Lists - uniform distribution - transient injection - Bitcount



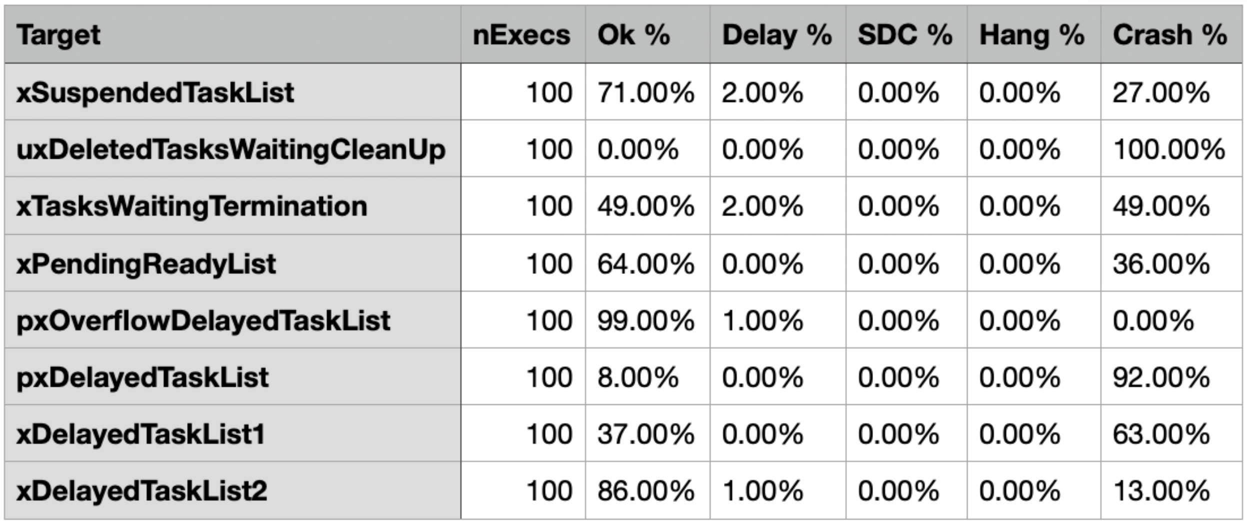
Target	nExecs	Ok %	Delay %	SDC %	Hang %	Crash %
pxCurrentTCB.ucDelayAborted	100	90.00%	0.00%	9.00%	1.00%	0.00%
pxCurrentTCB.ucStaticallyAllocated	100	81.00%	1.00%	8.00%	3.00%	7.00%
pxCurrentTCB.ucNotifyState	100	88.00%	0.00%	9.00%	3.00%	0.00%
pxCurrentTCB.ulNotifiedValue	100	89.00%	0.00%	9.00%	2.00%	0.00%
pxCurrentTCB.ulRunTimeCounter	100	90.00%	1.00%	6.00%	2.00%	1.00%
pxCurrentTCB.pxTaskTag	100	86.00%	0.00%	12.00%	1.00%	1.00%
pxCurrentTCB.uxMutexesHeld	100	93.00%	0.00%	7.00%	0.00%	0.00%
pxCurrentTCB.uxBasePriority	100	91.00%	1.00%	8.00%	0.00%	0.00%
pxCurrentTCB.uxTaskNumber	100	92.00%	0.00%	8.00%	0.00%	0.00%
pxCurrentTCB.uxTCBNumber	100	90.00%	1.00%	8.00%	1.00%	0.00%
pxCurrentTCB.pcTaskName	100	66.00%	0.00%	4.00%	0.00%	30.00%
pxCurrentTCB.pxStack	100	90.00%	1.00%	8.00%	0.00%	1.00%
pxCurrentTCB.xEventListItem	100	70.00%	0.00%	2.00%	0.00%	28.00%
pxCurrentTCB.xStateListItem	100	12.00%	0.00%	2.00%	0.00%	86.00%
pxCurrentTCB.pxTopOfStack	100	4.00%	0.00%	0.00%	0.00%	96.00%

Figure 6.21: Current TCB - uniform distribution - transient injection - Bitcount



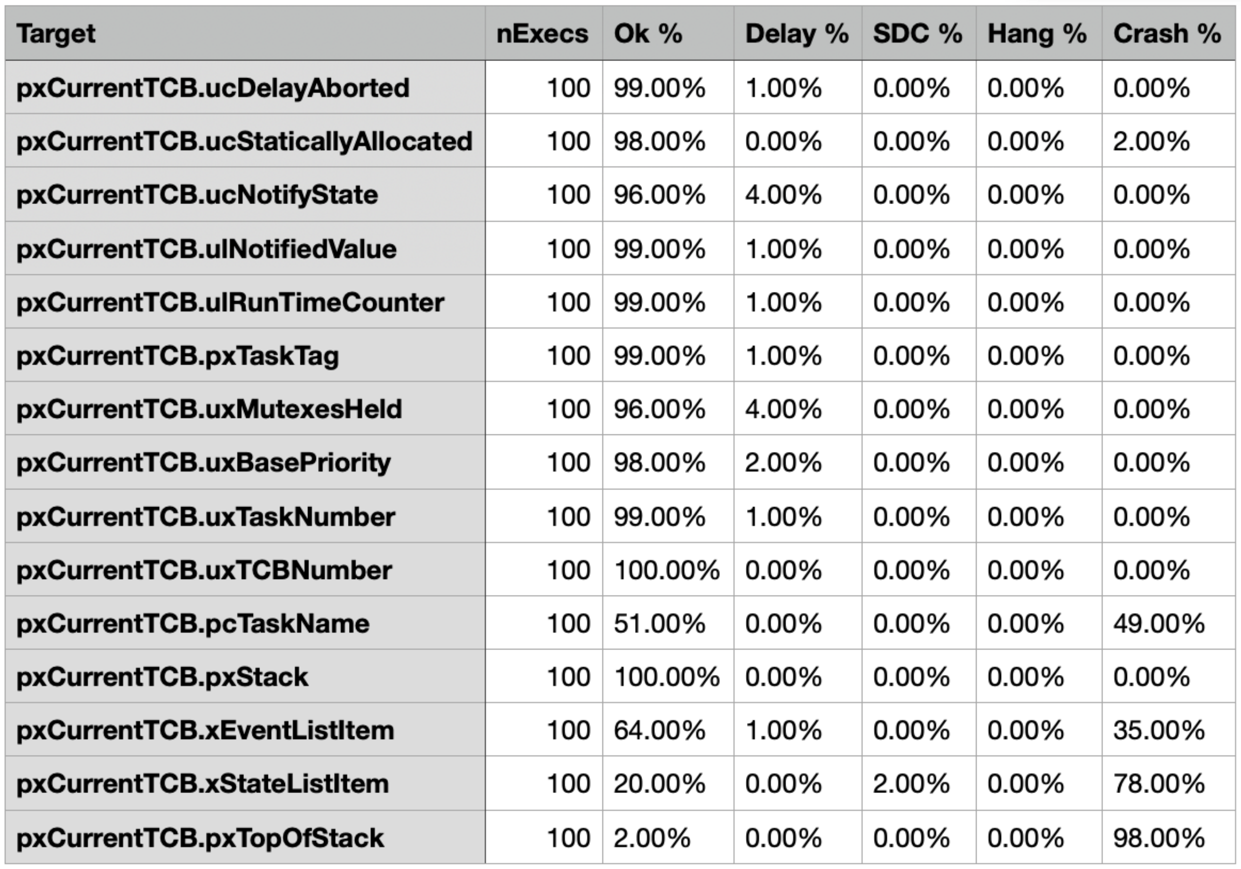
Target	nExecs	Ok %	Delay %	SDC %	Hang %	Crash %
xIdleTaskHandle	100	2.00%	0.00%	0.00%	0.00%	98.00%
xNextTaskUnblockTime	100	16.00%	3.00%	0.00%	11.00%	70.00%
uxTaskNumber	100	98.00%	2.00%	0.00%	0.00%	0.00%
xNumOfOverflows	100	98.00%	2.00%	0.00%	0.00%	0.00%
xYieldPending	100	100.00%	0.00%	0.00%	0.00%	0.00%
xPendedTicks	100	16.00%	1.00%	0.00%	0.00%	83.00%
xSchedulerRunning	100	97.00%	0.00%	0.00%	0.00%	3.00%
uxTopReadyPriority	100	4.00%	0.00%	0.00%	0.00%	96.00%
xTickCount	100	29.00%	2.00%	0.00%	0.00%	69.00%
uxCurrentNumberOfTasks	100	100.00%	0.00%	0.00%	0.00%	0.00%

Figure 6.22: System State Variables - uniform distribution - transient injection - Qsort



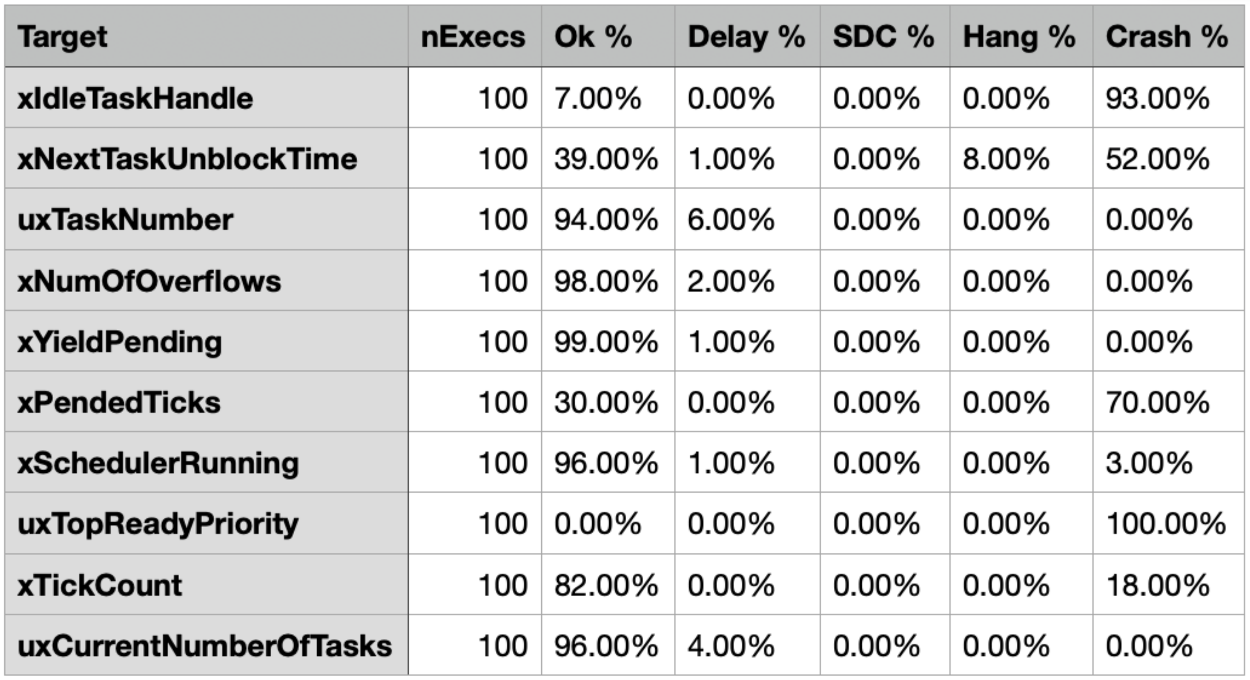
Target	nExecs	Ok %	Delay %	SDC %	Hang %	Crash %
xSuspendedTaskList	100	71.00%	2.00%	0.00%	0.00%	27.00%
uxDeletedTasksWaitingCleanUp	100	0.00%	0.00%	0.00%	0.00%	100.00%
xTasksWaitingTermination	100	49.00%	2.00%	0.00%	0.00%	49.00%
xPendingReadyList	100	64.00%	0.00%	0.00%	0.00%	36.00%
pxOverflowDelayedTaskList	100	99.00%	1.00%	0.00%	0.00%	0.00%
pxDelayedTaskList	100	8.00%	0.00%	0.00%	0.00%	92.00%
xDelayedTaskList1	100	37.00%	0.00%	0.00%	0.00%	63.00%
xDelayedTaskList2	100	86.00%	1.00%	0.00%	0.00%	13.00%

Figure 6.23: Task Lists - uniform distribution - transient injection - Qsort



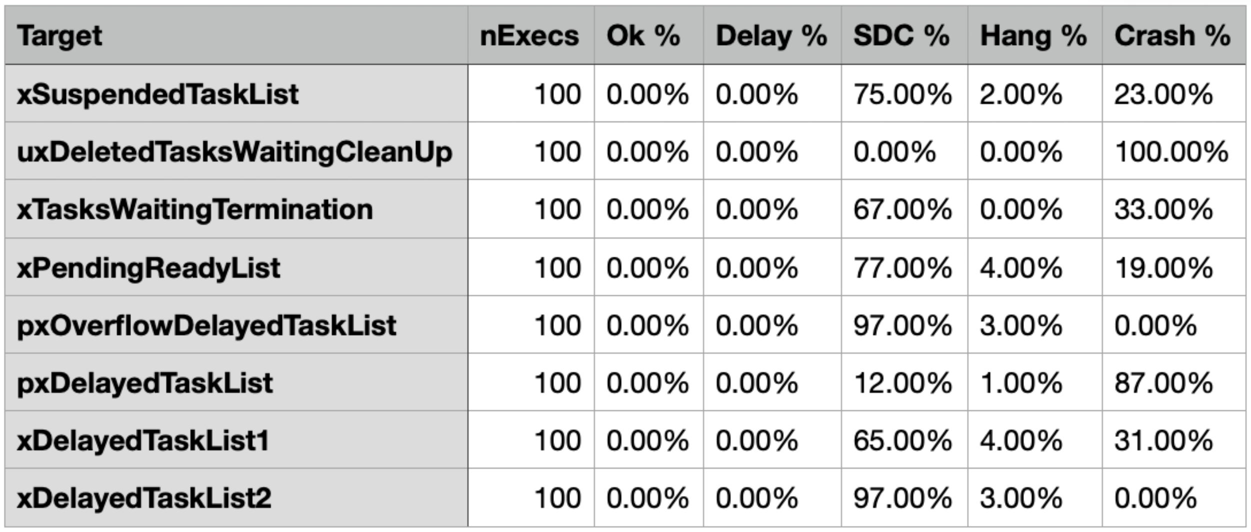
Target	nExecs	Ok %	Delay %	SDC %	Hang %	Crash %
pxCurrentTCB.ucDelayAborted	100	99.00%	1.00%	0.00%	0.00%	0.00%
pxCurrentTCB.ucStaticallyAllocated	100	98.00%	0.00%	0.00%	0.00%	2.00%
pxCurrentTCB.ucNotifyState	100	96.00%	4.00%	0.00%	0.00%	0.00%
pxCurrentTCB.ulNotifiedValue	100	99.00%	1.00%	0.00%	0.00%	0.00%
pxCurrentTCB.ulRunTimeCounter	100	99.00%	1.00%	0.00%	0.00%	0.00%
pxCurrentTCB.pxTaskTag	100	99.00%	1.00%	0.00%	0.00%	0.00%
pxCurrentTCB.uxMutexesHeld	100	96.00%	4.00%	0.00%	0.00%	0.00%
pxCurrentTCB.uxBasePriority	100	98.00%	2.00%	0.00%	0.00%	0.00%
pxCurrentTCB.uxTaskNumber	100	99.00%	1.00%	0.00%	0.00%	0.00%
pxCurrentTCB.uxTCBNumber	100	100.00%	0.00%	0.00%	0.00%	0.00%
pxCurrentTCB.pcTaskName	100	51.00%	0.00%	0.00%	0.00%	49.00%
pxCurrentTCB.pxStack	100	100.00%	0.00%	0.00%	0.00%	0.00%
pxCurrentTCB.xEventListItem	100	64.00%	1.00%	0.00%	0.00%	35.00%
pxCurrentTCB.xStateListItem	100	20.00%	0.00%	2.00%	0.00%	78.00%
pxCurrentTCB.pxTopOfStack	100	2.00%	0.00%	0.00%	0.00%	98.00%

Figure 6.24: Current TCB - uniform distribution - transient injection - Qsort



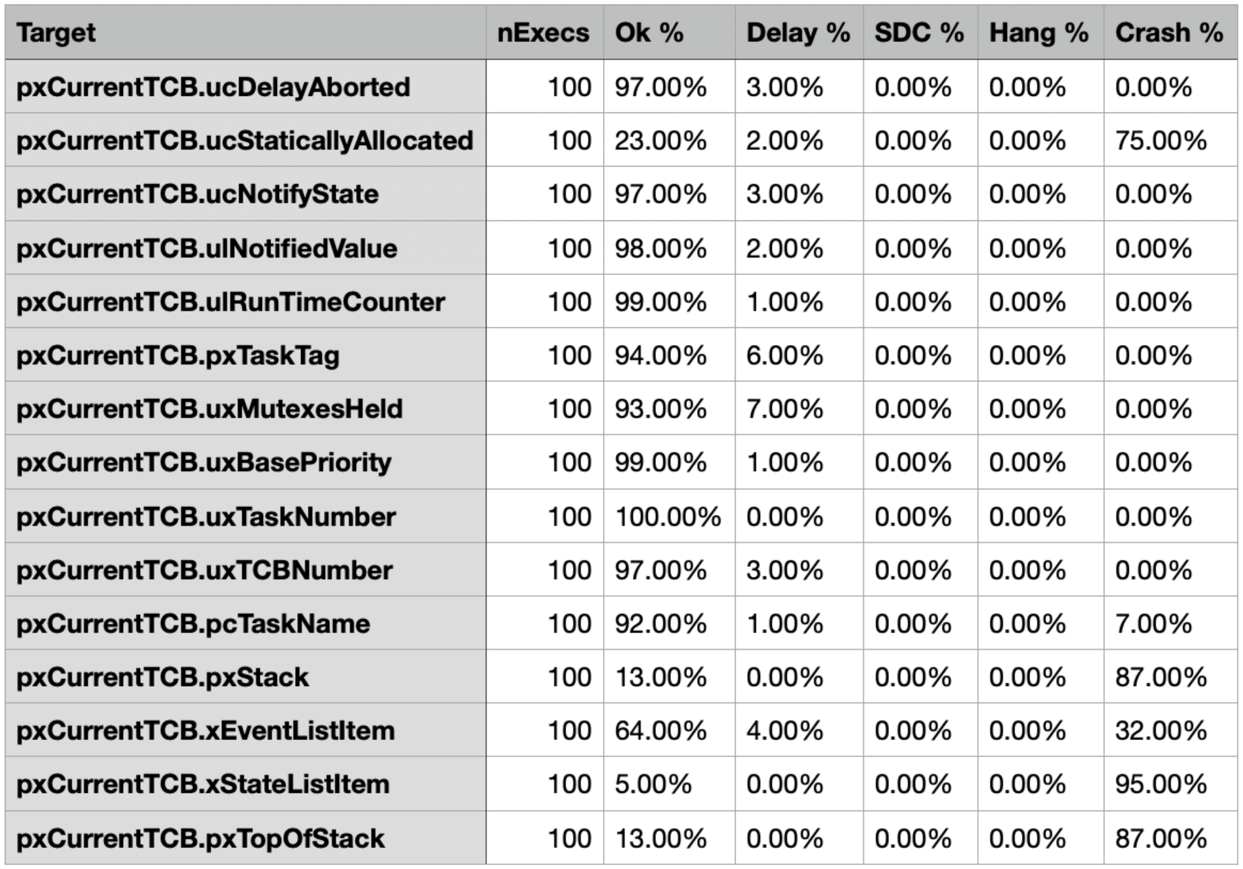
Target	nExecs	Ok %	Delay %	SDC %	Hang %	Crash %
xIdleTaskHandle	100	7.00%	0.00%	0.00%	0.00%	93.00%
xNextTaskUnblockTime	100	39.00%	1.00%	0.00%	8.00%	52.00%
uxTaskNumber	100	94.00%	6.00%	0.00%	0.00%	0.00%
xNumOfOverflows	100	98.00%	2.00%	0.00%	0.00%	0.00%
xYieldPending	100	99.00%	1.00%	0.00%	0.00%	0.00%
xPendedTicks	100	30.00%	0.00%	0.00%	0.00%	70.00%
xSchedulerRunning	100	96.00%	1.00%	0.00%	0.00%	3.00%
uxTopReadyPriority	100	0.00%	0.00%	0.00%	0.00%	100.00%
xTickCount	100	82.00%	0.00%	0.00%	0.00%	18.00%
uxCurrentNumberOfTasks	100	96.00%	4.00%	0.00%	0.00%	0.00%

Figure 6.25: System State Variables - uniform distribution - transient injection - Basicmath



Target	nExecs	Ok %	Delay %	SDC %	Hang %	Crash %
xSuspendedTaskList	100	0.00%	0.00%	75.00%	2.00%	23.00%
uxDeletedTasksWaitingCleanUp	100	0.00%	0.00%	0.00%	0.00%	100.00%
xTasksWaitingTermination	100	0.00%	0.00%	67.00%	0.00%	33.00%
xPendingReadyList	100	0.00%	0.00%	77.00%	4.00%	19.00%
pxOverflowDelayedTaskList	100	0.00%	0.00%	97.00%	3.00%	0.00%
pxDelayedTaskList	100	0.00%	0.00%	12.00%	1.00%	87.00%
xDelayedTaskList1	100	0.00%	0.00%	65.00%	4.00%	31.00%
xDelayedTaskList2	100	0.00%	0.00%	97.00%	3.00%	0.00%

Figure 6.26: Task Lists - uniform distribution - transient injection - Basicmath



Target	nExecs	Ok %	Delay %	SDC %	Hang %	Crash %
pxCurrentTCB.ucDelayAborted	100	97.00%	3.00%	0.00%	0.00%	0.00%
pxCurrentTCB.ucStaticallyAllocated	100	23.00%	2.00%	0.00%	0.00%	75.00%
pxCurrentTCB.ucNotifyState	100	97.00%	3.00%	0.00%	0.00%	0.00%
pxCurrentTCB.ulNotifiedValue	100	98.00%	2.00%	0.00%	0.00%	0.00%
pxCurrentTCB.ulRunTimeCounter	100	99.00%	1.00%	0.00%	0.00%	0.00%
pxCurrentTCB.pxTaskTag	100	94.00%	6.00%	0.00%	0.00%	0.00%
pxCurrentTCB.uxMutexesHeld	100	93.00%	7.00%	0.00%	0.00%	0.00%
pxCurrentTCB.uxBasePriority	100	99.00%	1.00%	0.00%	0.00%	0.00%
pxCurrentTCB.uxTaskNumber	100	100.00%	0.00%	0.00%	0.00%	0.00%
pxCurrentTCB.uxTCBNumber	100	97.00%	3.00%	0.00%	0.00%	0.00%
pxCurrentTCB.pcTaskName	100	92.00%	1.00%	0.00%	0.00%	7.00%
pxCurrentTCB.pxStack	100	13.00%	0.00%	0.00%	0.00%	87.00%
pxCurrentTCB.xEventListItem	100	64.00%	4.00%	0.00%	0.00%	32.00%
pxCurrentTCB.xStateListItem	100	5.00%	0.00%	0.00%	0.00%	95.00%
pxCurrentTCB.pxTopOfStack	100	13.00%	0.00%	0.00%	0.00%	87.00%

Figure 6.27: Current TCB - uniform distribution - transient injection - Basicmath

From what can be deduced from the xSchedulerRunning tables, this variable is not affected by SEUs like SHEs, in fact the SEU almost never leads this variable to a system crash, the values are almost null. Another interesting consideration that can be made concerns the group of ”task lists”. In fact, here unlike the case of SHEs the percentage of SDCs using the uniform distribution with Bitcount, does not exceed 9%.

Chapter 7

Conclusion

With the following thesis, the predefined objectives have been achieved, namely the development of a permanent fault simulation environment capable of simulating the effects on the real-time operating system FreeRTOS following a Single Hard Error (SHE) that affects the kernel data. Furthermore, the fault simulator has been enhanced and improved, particularly regarding transient faults. To benefit both types of faults, more accurate algorithms have been implemented for generating time instances using distributions. Moreover, an important aspect that has been achieved is the goal of decoupling from specific versions of the FreeRTOS kernel, by implementing adaptive code that is not bound to a specific kernel version. This was made possible by focusing on writing an environment that makes changes to the kernel automatically, avoiding manual kernel modifications. This approach helped reduce technical debt and facilitated the decoupling from a specific kernel.
Among the achieved objectives, there is also the implementation of a module that enables the automatic generation of datasets for injection campaigns based on a command with desired specifications for the injection campaign.
On the other hand, an aspect that would be interesting to explore is that of hardening, because during the development of the thesis project, I was able to address it by also developing a Hamming algorithm (15,11), which is an Error Correction Code (ECC), discussed in the appendix. However, further study should be conducted on the computational complexity of an error correction code (ECC) solution for this type of operating system. Alternatively, a hybrid solution with data redundancy could be implemented, or one that solely considers redundancy.
The hope is that this thesis work can be useful in the future and possibly further developed and enhanced for the benefit of scientific research.

Appendix A
Hardening

A.1 Introduction

Software hardening, in the context of fault tolerance, refers to the process of reinforcing a system or software application through a series of code modifications and improvements, aiming to reduce potential points of failure and increase the overall robustness of the software[17].
Hardening techniques can include implementing stricter error controls, adding error recovery procedures, using error-resistant code (such as defensive programming), improving resource management (such as memory management), creating backups or restores, and enhancing the resilience of dependent services.
Hardening may also involve intensive testing to identify and resolve bugs or vulnerabilities, using code analysis tools to identify code quality issues, and adhering to coding best practices and software quality standards.
In summary, software hardening is an essential component of software quality assurance, helping to create software that is more resistant, reliable, and less susceptible to malfunctions. Some hardening techniques include:

- **Error Correction Code (ECC)**: Under this name, a set of error correction techniques is used to prevent or correct errors that can be caused by various forms of electrical interference or hardware failures. ECC techniques can detect and correct errors, significantly improving system resilience.

- **Improved isolation between software components**: The principle of isolation implies that software components should operate independently of each other as much as possible. If one component fails, isolation can help limit the impact of the failure to that particular component, preventing error propagation to the rest of the system. This can be achieved through various techniques, such as sandboxing, containers, and virtualizers.

- **Replication and redundancy techniques**: Replication and redundancy are fundamental techniques for increasing fault tolerance. Replication involves creating multiple copies of data or functionality so that if one copy fails, another can take its place. Redundancy, on the other hand, involves using additional hardware or software components that are not necessary for normal system operation but can be utilized in case of component failure, thus increasing system resilience.

- **Implementation of stricter control protocols**: Implementing stricter control protocols can help prevent failures or limit their impact. For example, implementing regular integrity checks can quickly detect any failures or corruptions. Additionally, implementing robust recovery protocols can ensure that, in the case of transient failures, the system can quickly restore to a correct operating state.

A.2 Error Correction Code (ECC)

Error Correction Codes (ECC) are particularly useful for hardening real-time systems as these systems require high reliability and low latency. The need to reduce errors is crucial for maintaining data integrity and system operability. Below is a list of various ECCs, including Hamming code, with a detailed explanation of each:

1. **Hamming Codes**: These codes, developed by Richard Hamming, are capable of detecting and correcting single-bit errors and detecting double-bit errors. They work by adding parity bits to the data that is sent from a source to a destination. The parity bits are positioned at powers of 2 (1, 2, 4, 8, etc.), and each checks the bits in positions that are a subset of that power of 2. They are ideal for hardening real-time systems as they have low computational overhead.

2. **Reed-Solomon Codes**: These codes are particularly useful for correcting burst errors, which are errors that occur in consecutive sequences of bits. They operate on data blocks rather than individual bits, which

makes them suitable for correcting larger errors. They are widely used in communication systems, such as CDs and 2D barcodes, and can be used to enhance the robustness of real-time systems.

3. **Bose, Ray-Chaudhuri, Hocquenghem (BCH) Codes**: Similar to Reed-Solomon codes, BCH codes can correct multiple errors per symbol. They use a mathematical approach based on finite field theory, which makes them more complex but capable of correcting a higher number of errors per data block. These codes are used in various applications, including flash memory and optical disks.

4. **Low Density Parity Check (LDPC) Codes**: LDPC codes can correct multiple errors and are known for their decoding efficiency. They use a sparse parity check matrix, which makes decoding very efficient. Although they are computationally more intensive than Hamming codes, they provide more robust error correction and can be used in situations where reliability is crucial.

5. **Turbo Codes**: These codes leverage two separate convolutional codes with an interleaver in between to redistribute bits. This enables very effective error correction, albeit with higher computational complexity. Turbo codes are widely used in wireless communication applications, where error correction is essential for maintaining signal quality.

Hardening a real-time system requires a trade-off between error correction and computational overhead. While Hamming codes offer low overhead and single-bit error correction, other ECCs allow for the correction of multiple errors or burst errors at the cost of increased complexity.

A.3 Hamming Code

Hamming code is a type of error-correcting code used to detect and correct errors in a stream of binary data. Hamming code is based on adding parity bits to the original data stream.
The main idea behind Hamming code is to place the parity bits at specific power-of-2 positions within the data stream. These parity bit positions are calculated based on the positions of the data bits, which are numbered starting from 1.

The steps of the encoding process are as follows:

1. Given a binary data stream of length n, a new data stream is generated that includes both the original data bits and the parity bits. The length of the encoded data stream is given by m, where $m \geq n + r$ and r is the number of parity bits to be added.

2. Power-of-2 positions (within the encoded data stream) are reserved for the parity bits. For example, with a data stream length of 7 ($n = 7$), positions 1, 2, 4, and 8 are reserved for the parity bits. To determine the number of parity bits, a binary inequality $2^r \geq n+r+1$ is used, finding the smallest value of r for which this inequality holds. For example, trying $r = 3$: $2^3 = 8 \geq 7 + 3 + 1 = 11$ (not satisfied). Trying $r = 4$: $2^4 = 16 \geq 7+4+1 = 12$ (satisfied). Therefore, 4 parity bits are required for $n = 7$.

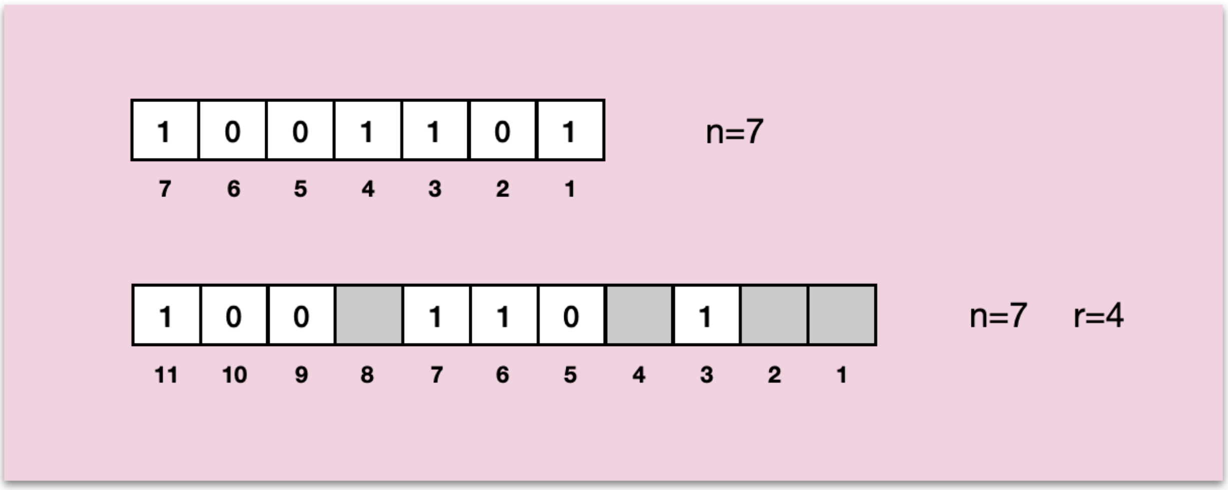


Figure A.1: Representation of sequence expansion with the addition of parity bits

3. The values of the parity bits are calculated based on the corresponding data bits. For each parity bit, the following operations are performed:

   - Consider the position of the parity bit for which we need to calculate the value, and consider the bits whose positions (in binary) have a 1 in the position of the parity bit to be calculated. For example, for the parity bit at position 1, consider the bits that have a binary index with the least significant bit equal to 1 (1, 3, 5, 7, etc., as their binary representations are 1, 11, 101, 111). For the parity bit at position 2, consider the data bits that have the second least significant bit equal to 1 in their binary representation. In other words, consider the data bits whose binary index has a 1 in the second rightmost position (2, 3, 6, 7, etc., as they correspond to 10, 11, 110, 111, which have 1 in the second bit).

- Perform an XOR operation on the values of the considered data bits (excluding the parity bit not yet set) for each parity bit, and assign the result of the XOR operation to the respective parity bit. This XOR operation checks whether the sum of the values of the considered bits is even or odd, and the parity bit will be set to 1 if the sum is odd and 0 if it is even. For example, if data bits 3, 5, and 7 are 1, the value of the parity bit at position 1 will be 1 (1 XOR 1 XOR 1), while with values 1, 0, 1, it will be 0 (1 XOR 0 XOR 1).

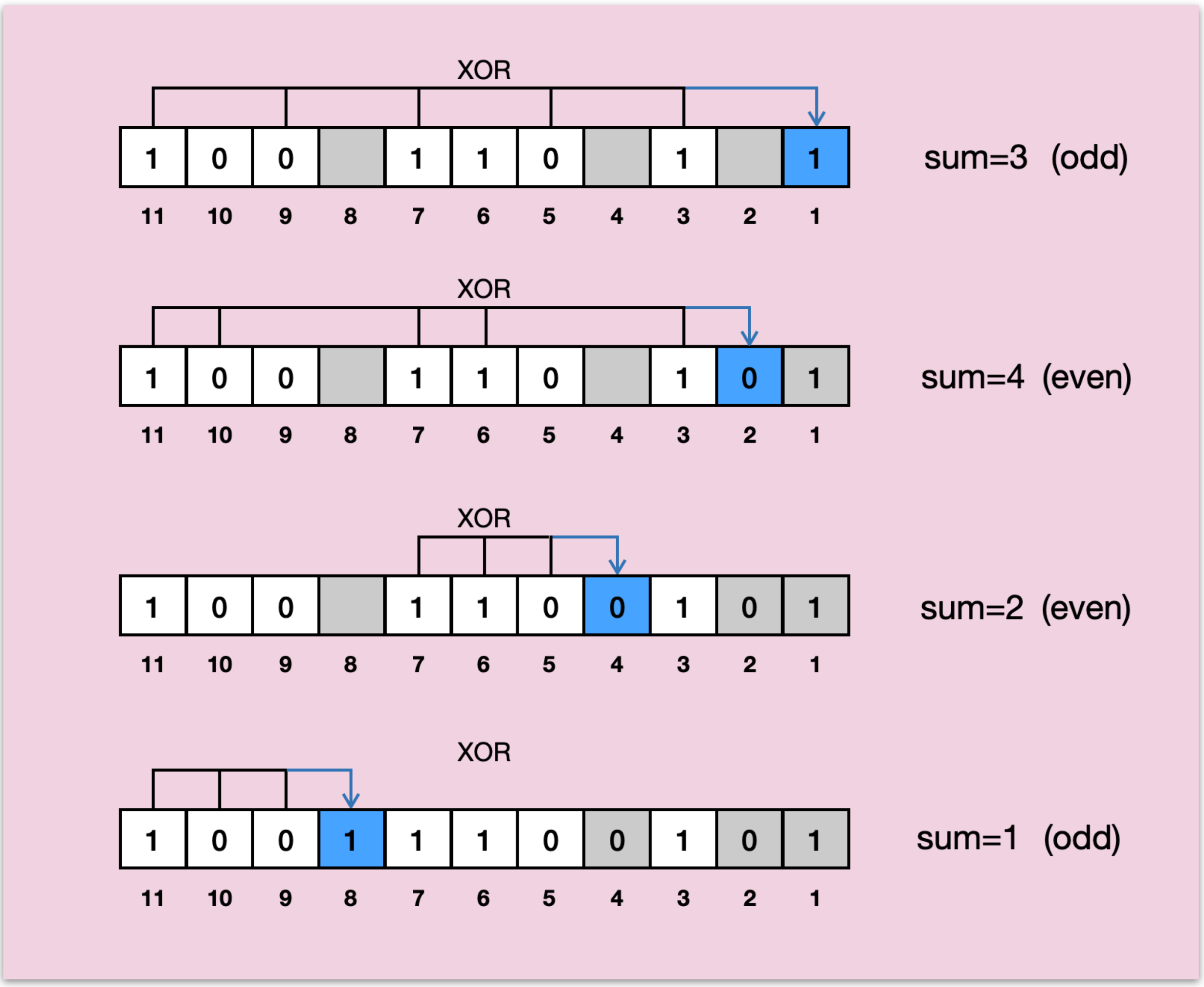


Figure A.2: Calculation of parity bit values in Hamming code

During decoding, the parity bits can be used to detect and correct any errors in the original data stream. It should be noted that Hamming code can detect and correct 1-bit errors if present or detect up to 2-bit errors without correction[19].

A.4 Data Redundancy as a Solution for Data Integrity and Data Recovery

We have seen how error correction codes (ECCs) can be extremely useful in detecting and correcting bit errors. However, when considering their implementation in a real-time operating system, they come with an inherent disadvantage: a high overhead that can easily be too burdensome for such

systems.
The excessive overhead is due to the fact that data protection mechanisms like ECC cannot distinguish between "illegitimate" modifications (such as accidental errors or attacks) and "legitimate" modifications (such as intentional data updates). For them, a modification is a modification, and if the goal is to protect data integrity, every modification would require an update of the protection mechanism, resulting in recalculating the code (including parity bits) for every legitimate modification of each individual variable to be protected.

One possible approach is to selectively choose which kernel variables need to be protected, reducing the overall computational cost by recalculating parity bits only for specific variables. However, when considering the desire to protect as many variables as possible, a specific need arises: ensuring that if a variable is modified due to an undesired alteration on a bit, that change can be detected and restored. At the same time, it is necessary that when a variable is legally modified, no further complications arise, such as continually updating parity bits. An alternative solution could be the use of hash functions for data integrity verification. While effective in detecting changes, hash functions have a significant limitation: they do not provide a mechanism for data recovery in case of alteration.
Faced with these challenges, a solution that could prove more efficient is the implementation of data redundancy. Although it requires increased memory consumption, this strategy presents significantly lower computational costs compared to ECCs.
Data redundancy not only enables error detection but also offers the ability to restore data to its original form if it is altered. In a real-time operational context, where system performance and responsiveness are crucial, implementing data redundancy can be a particularly advantageous strategy. This approach reduces the computational burden, preventing potential system overload, and ensures effective data management regardless of the presence of single-event upsets (SEUs) or single hard errors (SHEs).
It should be noted that an ECC code, in terms of error detection and correction, can certainly be effective in the case of SEUs. However, in the case of SHEs, it is ineffective on its own. Therefore, at the very least, a hybrid solution that considers data redundancy, specifically variable duplication, is necessary. Therefore, despite requiring higher memory consumption, data redundancy appears as an efficient and practical solution to maintain data

integrity and ensure data recovery in the context of real-time operating systems.

Bibliography

[1] Srivastava, A. K., & Sharma, M. (2018). Impact of Ionizing Radiation on Electronic Devices and Modeling of Displacement Damage in Silicon. In *Ionizing Radiation Effects and Applications*, InTech.

[2] Thorne, R. M. (2010). Van Allen Probes: Resolving the Radiation Belts' Mysteries. *EOS, Transactions American Geophysical Union*, 91(17), 153-154.

[3] Hathaway, D. H. (2010). The solar cycle. *Living Reviews in Solar Physics*, 7(1), 1-65.

[4] J. R. Schwank, M. R. Shaneyfelt, P. E. Dodd, J. A. Felix, P. Paillet, M. Gaillardin, E. W. Blackmore, "Radiation Effects in Microelectronics," in Fundamentals of Radiation Materials Science: Metals and Alloys, Second Edition, G. S. Was, Ed. Berlin, Germany: Springer, 2017, pp. 713-803.

[5] S. Buchner, "Single Event Effects in Avionics," IEEE Transactions on Nuclear Science, vol. 43, no. 2, pp. 461-474, Apr. 1996.

[6] K. A. LaBel et al., "Emerging radiation hardness assurance (RHA) issues: Hard errors, nondestructive effects and single-event effects (SEEs)," in IEEE Radiation Effects Data Workshop, 2015.

[7] T. Heijmen, "Single Event Effect Criticality Analysis (SEECA)," ESA/ESTEC, Noordwijk, The Netherlands, Technical Note, 2000.

[8] J. L. Titus, "Single Event Effects in Avionics," IEEE Transactions on Nuclear Science, vol. 43, no. 2, pp. 461-474, Apr. 1996.

[9] T. F. Miyahira, P. Y. K. Cheung, L. W. Massengill, A. T. Kelly, B. L. Bhuva, "Single-Event Effects in SOI Technologies and Devices," IEEE Transactions on Nuclear Science, vol. 60, no. 3, pp. 1855-1875, June 2013.

[10] J. C. Pickel, J. B. Blish II, ”Single Event Burnout,” IEEE Transactions on Nuclear Science, vol. 38, no. 6, pp. 1437-1444, Dec. 1991.

[11] J. W. S. Liu, ”Real-Time Systems,” Prentice Hall, 2000.

[12] A. Burns, A. Wellings, ”Real-Time Systems and Programming Languages: Ada, Real-Time Java and C/Real-Time POSIX,” Pearson, 4th Edition, 2009.

[13] H. Kopetz, ”Real-Time Systems: Design Principles for Distributed Embedded Applications,” Springer, 2nd Edition, 2011.

[14] R. Barry, ”Using the FreeRTOS Real Time Kernel - a Practical Guide,” Real Time Engineers Ltd, 2010.

[15] R. Barry, ”Mastering the FreeRTOS Real Time Kernel: A Hands-On Tutorial Guide,” Real Time Engineers Ltd, 2016.

[16] Avizienis, A., Laprie, J. C., Randell, B., & Landwehr, C. (2004). *Basic concepts and taxonomy of dependable and secure computing.* IEEE transactions on dependable and secure computing, 1(1), 11-33.

[17] Pomeranz, I. (2016). *Software hardening through reverse engineering.* IEEE Design & Test, 33(4), 95-102.

[18] Lin, S., & Costello, D. J. (2004). *Error control coding.* Pearson Education.

[19] Hamming, R. W. (1950). *Error detecting and error correcting codes.* The Bell system technical journal, 29(2), 147-160.