Politecnico di Torino
Computer engineering
Department of Computer Engineering,
Cinema and Mechatronics

FreeRTOS Fault Injector Simulator for Single Event Upsets

Course of System and Device Programming

Authors
Alessandro Franco
Simone Alberto Peirone
Marzio Vallero

Professors
Alessandro Savino
Stefano Quer
Giampiero Cabodi
Antonio Vetrò

a.y. 2020/2021

Contents

1 Abstract

Developing a software is an extensive task, comprising of complexity analysis and feature inclusion, bug fixes and many more. Unfortunately the real world is not an ideal place for computer systems and software, as random errors due to electromagnetic interference interacting with hardware, static electric charges being inserted in the circuit or just vibrations and movements producing slight displacements of internal components of a machine are likely to happen and should be addressed appropriately to ensure correct execution of both hardware and software.
In the field of space exploration this issue becomes more and more relevant given its mission criticality and as such software must be able to compensate or recover in real time from random bit flips with high incidence.
In this paper we will analyse how a real-time operating system reacts to sudden alterations of a bit state during its execution, possibly identifying weaknesses in the code and spotting critical variables during runtime. The project will be based on a portable simulator that handles both fault injection (SEU, Single Event Upset) and creation of instances of FreeRTOS, used as benchmark for the analysis.

2 Introduction

Modern software is a reference point for our daily life, not only for the most common needs, such as weather forecast or news, but also mission critical applications, for example autonomous driving cars or medical support equipment. Errors in software can happen either as a consequence of a mistake during programming or as a result of random physical factors that cannot always be expected during development.
While the former can be fixed via thorough testing and debugging, the latter can only be addressed a posteriori at the end of development, after a deep analysis on how the software responds to errors linked to external interaction; given the peculiar nature of this matter, several studies have been conducted on different pieces of software. It will be the focus of this paper as well, addressing in particular how a simulated instance of FreeRTOS reacts to Single Event Upsets (SEU) in a pure software environment. The project at its core revolves around the generation of multiple threads, for the simulation of the OS and for the injector, in a modular way: after each generation the analysis is conducted based on the results for the single execution and then both threads are terminated once the result has been measured. Said analysis is carried out by means of an execution trace, a short memory log which gets updated at timed checkpoints, to verify the impact of an injection on the execution. Once the execution terminates, its results are compared to a previously ran golden execution: depending on timing and execution differences a classification for the injection's effect on execution is collected for statistics to be later displayed at the end of the injection campaigns.

3 Project Structure

The project is a C based application, boasting a modified version of FreeRTOS's kernel. The modifications we introduced allow for target extraction, access and injection, custom logging of the events performed by the scheduler and most importantly a clean way to terminate the FreeRTOS instance.

3.1 The Orchestrator Process

The project relies on the emulated execution of a FreeRTOS instance, launched alongside an injector thread to which produces SEUs in the data structures used by the OS and later classifies the effect of such events. The simulation is executed starting from a main process that generates all the required threads and manages the initialization process. The orchestrator process instantiates an array of children processes, who in turn create a thread. The child process is used to run the FreeRTOS instance (which actually is made of multiple threads itself), while the thread executes the injection function. Having separate threads for injection and FreeRTOS execution ensures that the injector has access to memory addresses used by the RTOS, but it cannot be stopped by it when trying to access memory allocated to the kernel since it is not a task internal to

the RTOS, providing fast integration without the need for further control over the interaction onto protected memory segments. The exact number of processes created is defined by the input.csv file, used to define the injection campaign, select the target variables and data structures and define timings for each injection. Since the time of injection can present a certain variance, an additional parameter is associated to each entry in the file, which will define the distribution of random numbers for the given time interval. The default distribution will handle the generation of random number as a gaussian normal zero mean. The rand() function generates a pseudo random number given a range and is used as means to apply a different distribution to the bits selected for injection. The generation is based on Irwin-Hall's distribution 8, which approximates the mean of a set of random variables to a gaussian distributed or triangular distributed random variable. The injector function also implements a timer to ensure that the orchestrator process does not get stuck waiting for FreeRTOS instances that end up in a crash or a hang.

3.2 The Golden Execution

The golden execution represents a full and correct execution of the benchmark without the occurrence of any externally introduced fault. This special execution doesn't create an injector thread and self terminates once the benchmark is over. Right before terminating, a *golden.txt* file is saved locally, which contains on the first line the execution time of the golden execution in nanoseconds and on the subsequent lines the output of the qsort algorithm used as benchmark, with 8000 entries ordered alphabetically during execution. The statistics provided by the *golden.txt* file are used as a comparison metric for injected executions in order to spot out delays and incorrect executions. The contents of the *golden.txt* file are subject to slight differences in the recorded value of the execution time, as the simulator runs on a host operating system and as such can be preempted by it at any time. Generally a median of about 241 milliseconds with a standard deviation of 4 milliseconds is registered.

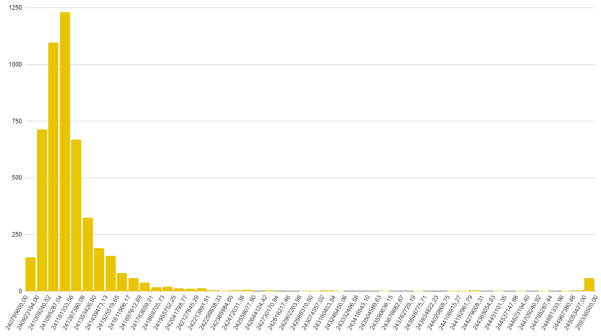

Figure 1: The distribution of 5000 golden execution times on POSIX.

On the other hand, the result of the qsort is constant and as such it's used as a measure of correctness of execution of the benchmark. Since the injection is performed on FreeRTOS kernel structures and not on local task data, generally no error is noticed in the output of the qsort; rather, an output may be not present if the qsort task failed to execute.

3.3 Target Extraction

FreeRTOS is a rather small operating system, at least regarding code size, with all its routines and data structures defined and implemented in a few files. From the beginning, we identified two viable paths that could allow us to get the list of the available injection targets.
The first approach relied on the analysis of the executable file of the target FreeRTOS simulator, whose symbol tables provide the virtual address of the data structure defined by FreeRTOS. However, this approach proved to be not ideal since it would require developing two different versions of the executable analyzer, one for the Posix environment and one for Windows. Moreover, the locations found in the executable headers may differ from the corresponding runtime addresses due to a computer security technique called *Address Space Layout Randomization* (ASLR) [1]. ASLR randomizes the virtual addresses of a process before its execution. Most operating systems, includ-

[1] On Linux, ASLR can be disabled or enabled by writing respectively 0 or 1 to `/proc/sys/kernel/randomize_va_space`.
For the sake of the current project, this approach was not considered since it requires superuser rights. Also, the same procedure in Windows requires editing the Registry.

ing Linux and Windows, implement ASLR as a measure to mitigate attacks exploiting an a priori knowledge of the positions of the data areas of a process. Therefore, reading the injection targets addresses from the executable is useless since they differ from the actual address at runtime.
The second alternative approach was to develop a mechanism capable of retrieving the injection targets during the simulator execution bearing in mind that most of the FreeRTOS data structures are declared with the static specifier and thus can not be directly referenced. We developed a set of functions defined in the kernel files that read the injection targets in a globally accessible list. For each target, a `target_t` structure stores its name, address, size and type (variable, struct, pointer, list, array or a union of the previous ones). The latter is used during the injection phase to compute the memory shape of the target. As an example, tasks.c defines `pxReadyTasksLists` as a fixed-size array of `List_t` items: its corresponding `target_t` structure contains the address of the array, its length (nmemb field), the size of each element and the type.

```
struct target_s
{
    int id;
    char name[64];
    void *address;
    unsigned int size;
    unsigned int nmemb;
    unsigned int type;
    struct target_s *parent;
    struct target_s *content;
    struct target_s *next;
};
```

All the injection targets are collected at the start of the execution. The Orchestrator uses this collection to list the targets (`list` command) and to support the execution of the golden simulation (`golden`). During the execution of an injection campaign, the orchestrator computes, for each target, the injection offset and timing using the information provided by the targets list. Due to ASLR, it is not possible to directly compute the final injection address since it is not equal to the one in the child process.

For the sake of this project, only targets defined in tasks.c and timers.c are considered. Additional targets may be included in the targets list by providing proper `read_`*filename*`_targets()` functions.

3.4 The FreeRTOS Instance

The FreeRTOS instance and the injections orchestrator share the same code. The behaviour of the simulator is controlled by its command-line arguments.

The –run command performs a single injection with the provided parameters. Users are not usually expected to run one simulation at a time but rather to use the –campaign command. The latter allows specifying an injection campaign which is a list of injections to be performed with variable parameters. More information on how to define an injection campaign are reported in the Usage section of the report. For each request in the campaign, the simulator forks a child process responsible for performing the actual injection and reporting the outcome to its father. Since the fork procedure uses platform-specific API, it is abstracted behind a common header and then implemented using fork or CreateProcess depending on the compilation target. This concept is used regularly throughout the project in order to reduce as much as possible code duplication.
Afterwards, the simulator waits for the child to terminate and classifies the injection outcome according to the returned exit code. Finally, it prints out the collected statistics. At the start of its execution, the child process determines the address of the passed injection target and it spawns the thread responsible for the injection. Then, it starts the FreeRTOS scheduler which blocks the calling thread. The injector wakes up at the specified injection time and fulfils its task.

An injection may result in a broken state of the FreeRTOS kernel. In order to recover from such an event, the simulator has two ways of terminating its execution. The first and standard procedure is provided by the Idle task hook. The Idle task is a special task that is scheduled whenever the ready queue of the kernel is empty. The function vApplicationIdleHook is triggered each time the Idle task is executed and it checks if there are other pending tasks in any of the scheduler queues. If this is not the case, the hook generates a simulated interrupt

and stops the scheduler. The simulated interrupt is used to detect hangs as described in a later section.
If the injection breaks the scheduler this path is not viable since the idle task may never be scheduled again. For this reason, after performing the injection, the injector thread waits a timeout period computed as three times the golden execution time. This timeout expiring indicates that the idle task was not able to gracefully stop the execution of the kernel. Therefore, the injector thread generates a simulated interrupt and stops the scheduler.

Once the scheduler is stopped, the control is returned to the main thread which is now responsible for checking the trace of the execution and computing the injection outcome. The outcomes are defined using a set of exit codes. Any hard fault, i.e. a memory access violation following an injection, results in a failure of the simulator with an unknown exit code which is classified as a crash.

3.5 The Injector Thread

The injector thread on paper is a very simple tool used by this project to insert random SEUs in specified points in memory: its job includes acquiring the correct injection address and inverting the value of a bit depending on the parameters passed by means of a thData_t data structure, which contains all details on the requested injection.
Regardless of its simplicity, the work at the foundation is critical to ensure relevant data and a correct execution of the simulation: from the selection of the targets to the actual insertion, a non negligible number of operations must be executed in preparation for the actual injection. First of all the address of the variable to be injected must be extracted and a bit position must be randomized: only then the injection location is set up and ready. The setup phase is not limited to target extraction, as in the input file a time for the injection has to be specified. This sparks a new set of problems: the injector must not be a FreeRTOS task, since a task cannot access kernel memory space, and moreover the injector cannot be under the control of FreeRTOS's scheduler, as it would render the injection an unrealistic approximation of real SEUs behaviour. To this end a timer function has been set up to trigger the injection thread only after the requested simulation time has been reached. Multiple versions of this function have been tested both in POSIX and Windows. At first we put the thread in a waiting state only to be triggered by a signal or an event respectively: this however caused the delay for the injection to be larger than expected due to the host OS's rescheduling time. To solve the issue, we eventually opted for a busy waiting technique, resulting in a much more precise 4 digits nanoseconds of delay with respect to the expected time.

3.6 Performance Classifier

Once the child process returns from the execution of FreeRTOS, it has to analyze its behaviour and to communicate it to the orchestrator process. The simulator can classify injected executions as either Silent, Delay, Error, Hang or Crash. The classifier is executed by the child process right after the termination of the FreeRTOS instance and performs comparisons between the *golden.txt* file and the local *loggerTrace*. The *loggerTrace* is an array of strings which contains information with the format "<time> <hookMacroType> <taskName>" about the last 10 calls performed to a traceHookMacro, such as when a task is switched in, switched out or a read form queue inside an ISR fails. At first, the execution time of the whole RTOS instance is extracted from the most recent entry in the LoggerTrace data structure. At this point, the loggerTrace is analyzed, testing the presence of a successful processing of the Interrupt Service Routine which is called as a test before terminating the FreeRTOS instance. This ISR is triggered when trying to read from an empty queue during an ISR, calling a custom traceHookMacro defined in *loggingUtils.c*. If the ISR is executed correctly, it will be placed as the last entry in the loggerTrace, else it will not be found: the *traceOutputIsCorrect()* function performs this check. If the FreeRTOS instance did not respond to an ISR call, it means that it has crashed. If the ISR has been correctly performed, the classifiers then tests the output of the qsort

task, comparing it to the correct one present in the *golden.txt* file. If the two outputs match, the execution of the qsort task is considered successful: the *executionResultIsCorrect()* function performs this check. A correct output characterizes a Silent or Delayed execution, while an incorrect one characterizes an Error or Hang execution. The final step in this sub-branch is to compare the execution time of the current FreeRTOS instance with the golden execution time, with a margin of 5%. Faster executions are classified as Silent executions whether the output of the qsort is correct, else as Error executions. Slower executions are classified as Delayed executions whether the output is correct, else as Hang executions.

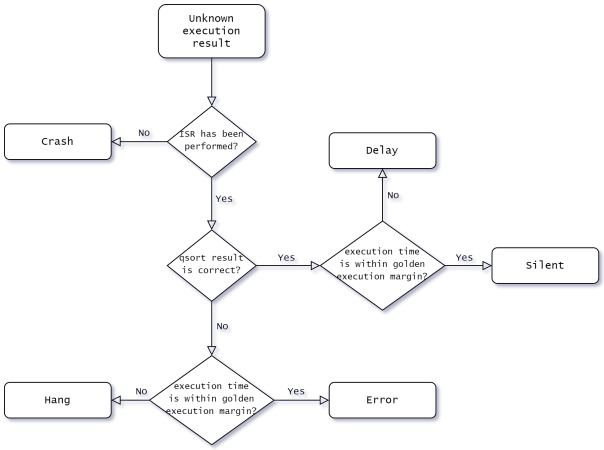

Figure 2: The performance classifier's decision model.

Once the FreeRTOS instance has been classified, the child calls *exit()* with an exit code describing its execution class. It's important to note that at times an injection will outright break the FreeRTOS instance, invoking unhandled exceptions or segmentation faults. These executions produce unhandled exit statuses and are thus considered as crashes by the orchestrator process directly.

3.7 Parallel execution

For any scientific testing procedure, results need to be statistically relevant: as such, for each injection campaign, the number of executions performed is expected to be at least on the order of 10^5. Since this requirement has a high impact on the time required to perform testing, it has been of foremost importance to be able to exploit parallelism of modern CPU architectures. Each FreeRTOS instance is executed in an isolated process, thus can be parallelized by means of a waiting queue managed by the orchestrator process. Inside a do-while loop, the function *waitFreeRTOSInjections()* is used to wait onto multiple child processes and their respective watchdog timers at once through a masked array, returning the position of the first terminating process or elapsed timer in the waiting queue. The double wait for each child process is necessary, since sometimes the child will hang without ever returning nor self terminating: in this case the child has to be terminated from the *outside* by the orchestrator process instead. The orchestrator process collects the exit status of the child, if any, then restarts the do-while loop. The internal implementation of *waitFreeRTOSInjections()* uses an array of HANDLEs to processes and timers, waiting through windows.h's *WaitForMultipleObjects()* function. On the other hand, on POSIX the waiting is performed only on the child processes' pid by means of the *waitpid()* function, whilst the timers have a hanging function that gets triggered on expiration. In both cases, whether the process or the timer get signalled, the result will be the same: the child process gets terminated and its watchdog timer gets destroyed.

4 Results Analysis

Execution results onto kernel structures are presented in Table 1. We can notice that Error exit statuses are never registered, due to the fact that the injections are performed onto kernel structures, producing higher level faults. Error executions could be obtained by injecting onto a task's variables. Simulations executed on Windows have the tendency to fall more into the Delay and Hang statuses with respect to the simulations hosted onto POSIX. This is probably due to the scheduler's more efficient process handling, reflected also in the per-simulation execution time, which is almost double on Windows with respect to POSIX.

Injections at 5ms ($n = 50$)						
Injection target		Silent	Delay	Error	Hang	Crash
`uxTopReadyPriority`	POSIX	0.0	2.0	0.0	6.0	92.0
	Win32	8.0	6.0	0.0	2.0	84.0
`xNumberOfOverflows`	POSIX	78.0	22.0	0.0	0.0	0.0
	Win32	62.0	38.0	0.0	0.0	0.0
`pxCurrentTCB.ucNotifyState[2]`	POSIX	80.0	20.0	0.0	0.0	0.0
	Win32	42.0	58.0	0.0	0.0	0.0
`pxReadyTasksLists[1][2]`	POSIX	84.0	16.0	0.0	0.0	0.0
	Win32	66.0	34.0	0.0	0.0	0.0

Table 1: Results of the injection campaigns

5 Usage

The simulator runs on Posix and Win32 compilation targets and is tested on gcc 8 and MSVC with the Visual Studio Community 2019 toolchain. The compilation is powered by CMake and is eased by two scripts `compile_posix.sh` and `compile_win32.bat`[2].

```
Supported commands:

 --list
   Print the list of available injection targets.

 --golden
   Run a golden simulation and print its output and
   execution time in the golden.txt file

 --run <target> <time> <off_byte> <off_bit>
   Perform an injection at the specified location and
   time (in nanoseconds)

 --campaign /path/to/input.csv [-y] [--no-pg-bar] [-j=N]
   Run an injection campaign from an input file.
    -y disables interactive mode
    --no-pg-bar hides the execution progress bar
    -j=N sets the number of parallel processes executed,
    default value 1
```

--list Print the list of available targets. For each target, size, type and possibly number of elements (only array targets) are reported.

```
./build/sim --list
xTimerTaskHandle (size:  8 B, nmemb: 1, type: VARIABLE)
xTimerQueue      (size:  8 B, nmemb: 1, type: VARIABLE)
uxTaskNumber     (size:  8 B, nmemb: 1, type: VARIABLE)
...
```

--golden Run a golden simulation, i.e. an execution of the FreeRTOS instance without introducing faults. The execution time and the output trace are stored in a `golden.txt` file.

--run Run a single injection at a specific time and on a specific bit. This command is executed internally during the injection campaign process to generate the instances of the injectors.

```
./sim --run uxTaskNumber 10000000 0 1
[DEBUG] Execution timeout is 240117000
[DEBUG] Calling mainSetup...
[DEBUG] Call to mainSetup completed
[DEBUG] launchInjectorThread called...
[DEBUG] Calling mainRun...
[DEBUG] Requested injection address: 0x55c0b16ae1e0
[DEBUG] Requested injection time: 10000000
[DEBUG] Requested injection offset byte: 0
[DEBUG] Requested injection offset bit: 1
[DEBUG] Performing the injection at time 10159200...
[DEBUG] Injection delay: 159200 (10159200 - 10000000)
[DEBUG] Injection completed
[DEBUG] Waiting the execution timeout
[DEBUG] Handler has been called.
[DEBUG] Executing past vTaskStartScheduler.
[DEBUG] Call to mainRun completed
```

--campaign Run one or multiple injection campaigns defined in the input csv file. Each line of the file has the following format:

```
# name, n of injections, time, variance, distribution
uxTopReadyPriority,50,20000000,5000,u
xNumOfOverflows,50,20000000,5000,t
xDelayedTaskList2,50,20000000,5000,u
```

The # character indicates that the line should be ignored. Depending on the target type, different formats may be specified in the input file. Using the target name alone means that the injection has to be performed on the memory area corresponding to the label. If

[2]CMake and the MSVC compiler are only available through the *Developer Command Prompt for VS 2019*

the target is a pointer, the injection is executed on the pointer itself and not on the pointed data. The latter is achieved by specifying `*target_name`. If the target is a struct, it is possible to access member fields using the dot notation: `target_name.field_name`. List and array targets allow defining the position of the element on which the fault has to be injected using the brackets notation: `my_target[index]`. If no position is specified, the fault is injected on the first element (array target) or the list head data structure (list target). An index equal to -1 indicates that the injector should randomly select a position. This is done before the injector thread execution if the target is an array whose size is known at compile-time or before the injection. Certain structures are defined as arrays of lists: in the input.csv file it is possible to specify the index inside the array and the position of the item in the list using a double brackets notation: `array_of_lists_target[i][j]`.

6 Conclusions

The project aimed at producing an operating system independent simulator of FreeRTOS able to perform multiple injection campaigns on kernel structures and to analyze the results. Moreover, the execution on a host operating system had to not break the real-time behaviour of FreeRTOS. We are proud to state that all of these properties have been successfully achieved, despite the numerous pitfalls we incurred into during development. Modifying the RTOS's kernel code without changing its real-time behaviour proved to be a complex task, as the kernel is structured to be as much self contained as possible. One of these challenging tasks was actually to stop the FreeRTOS instance from running, returning control to the calling process, and to perform the injections, as we could not use any sychronization primitives inside the RTOS itself. We delved into advanced programming techniques for code structuring and management, deepening our knowledge and improving our problem solving skills. More than anything else, this project has been a test of knowledge and patience, forcing us to think outside of our programming "comfort zone". We are proud of our work and hope sincerely that it will aid developers looking to test and improve the robustness of FreeRTOS distributions in the future, possibly avoiding faults in critical applications and saving lifes.

7 Acknowledgments

This project has been developed by Alessandro Franco, Simone Alberto Peirone and Marzio Vallero and supervised by professor A. Savino. The project amounts as a part of the Computer Engineering Masters Degree exam System and Device Programming, taught by professors S. Quer, G. Cabodi and A. Vetrò, during the academic year 2020/2021 at the Politecnico di Torino university.

8 References

- Mamone D.: Fault injection techniques for Real-Time Operating Systems. 2018.
- Mamone D., Bosio A., Savino A., Hamdioui S., Rebaudengo M..: On the Analysis of Real-time Operating System Reliability in Embedded Systems. 2021.
- N. Alimi et al.: "An RTOS-based fault injection simulator for embedded processors". International Journal of Advanced Computer Science and Applications, vol. 8, no. 5, pp. 300–306. 2017.
- P. Troger et al.:, "Software-implemented fault injection at firmware level". Third International Conference on Dependability. pp. 13–16. 2010.
- C. Grupen: "Astroparticle physics". Springer, Germany - Siegen. 2005.
- Barry R.: Mastering the FreeRTOS™ Real Time Kernel. A Hands-On Tutorial Guide. 2016.
- FreeRTOS official forums
- MiBench: a free, commercially representative embedded benchmark suite. University of Michigan at Ann Arbor.
- Irwin-Hall Distribution
- Address Space Layout Randomization