electronics

MDPI

Article

Virtualized Fault Injection Framework for ISO 26262-Compliant Digital Component Hardware Faults

Rui Almeida *, Vitor Silva and Jorge Cabral

Department of Industrial Electronics, Centro Algoritmi, University of Minho, 4800-058 Guimarães, Portugal; vsilva@dei.uminho.pt (V.S.); jcabral@dei.uminho.pt (J.C.)
* Correspondence: id10042@alunos.uminho.pt

Abstract: Simulation-based Fault Injection (FI) is crucial for validating system behaviour in safety-critical applications, such as the automotive industry. The ISO 26262 standard's Part 11 extension provides failure modes for digital components, driving the development of new fault models to assess software-implemented mechanisms against random hardware failures (RHF). This paper proposes a Fault Injection framework, QEFIRA, and shows its ability to achieve the failure modes proposed by Part 11 of the ISO 26262 standard and estimate relevant metrics for safety mechanisms. QEFIRA uses QEMU to inject permanent and transient faults during runtime, whilst logging the system state and providing automatic post-execution analysis. Complemented with a confusion matrix, it allows us to gather standard compliant metrics to characterise and evaluate different designs in the early stages of development. Comparatively to the native QEMU implementation, the tool only shows a slowdown of 1.4× for real-time microcontroller-based applications.

Keywords: virtual platforms; QEMU; fault injection; ISO 26262; fault modelling

check for updates

Citation: Almeida, R.; Silva, V.; Cabral, J. Virtualized Fault Injection Framework for ISO 26262-Compliant Digital Component Hardware Faults. *Electronics* **2024**, *13*, 2787. https://doi.org/10.3390/electronics13142787

Academic Editors: Heena Rathore and Henry Griffith

Received: 3 June 2024
Revised: 11 July 2024
Accepted: 14 July 2024
Published: 16 July 2024

Copyright: © 2024 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

Electronic computing is evolving at a pace where chip technology reaches high densities whilst incorporating more functionalities and new technologies into their software, such as kernels, device drivers and large data processing applications. Consequently, the higher densities increase chip susceptibility to ionizing radiation [1], increasing soft error rates. Propagation of such errors into complex software stacks may cause system instability, compromising safety, security, reliability and performance.

Concerns for these metrics are most important on safety or mission-critical applications, for instance the aerospace, automotive, and defence industries, leading them to adhere to strict safety and reliability requirements, defined by specific standards such as the ISO 26262 Road vehicles Functional Safety for the automotive sector [2]. Since theses systems directly relate to safety, it is vital to prove that they implement the correct functionality in-time and with sufficient level of reliability even in the presence of soft errors and/or faults.

Fault mitigation comes in the form of redundancy, either hardware- or software-implemented [3], by replicating components that mask faults and prevent them from propagating. From a safety standpoint, either technique can be employed, but from a cost point-of-view, software-implemented techniques are considered preferable, since high safety oriented hardware components are typically more expensive than software development costs [4]. On high volume production areas with Size, Weight, Power and Cost (SWaP-C) constraints, such as automotive, Software-Implemented Hardware Fault Tolerance (SIHFT) techniques may be preferred over replicating hardware components of the Electronic Control Units (ECUs).

However, assessing the correctness of these techniques may become a difficult task, as hardware-based tests become destructive and lack repeatability, and software-only based tests lack observability and traceability. Hence, researchers and market leaders have

adopted alternatives in the form of virtual simulation platforms to avoid the consequences of destroying hardware at the cost of modelling the system in a virtual environment.

In the automotive industry, the ISO 26262 standard stipulates that the role of simulation is critical in validating system behaviour, and recommends simulation at all development phases. Furthermore, it advises the use of Fault Injection (FI) testing to not only evaluate the hardware architectural metrics, but also fault metrics, such as diagnostic coverage (DC) of the safety mechanisms (SM) [2]. To aid with this assessment, the safety standard released part 11, which provides failure modes that support the assessment of the safety mechanisms.

With that in mind, we propose an open-source tool, namely QEFIRA, that helps developers to assess fault mitigation techniques with failure modes supported by the ISO 26262 standard, part 11. It is based on the open-source QEMU emulator with modifications for executing runtime Fault Injection campaigns. The test bench performs architectural emulation of platform, Fault Injection during runtime, result logging and classification of fault runs.

The paper is organised as follows. Section 2 presents the state-of-the-art and related work regarding virtual platforms within the scope of Fault Injection and safety standards. Section 3 describes the main features and benchmarks of the proposed QEFIRA tool. Section 4 exemplifies how QEFIRA can be used to apply ISO 26262-compliant Fault Injection to digital components. Lastly, Section 5 presents the final remarks and future work.

2. State of the Art

This section presents basic concepts and terminologies related to virtual platforms and Fault Injection. Furthermore, it briefly introduces the ISO 26262 standard and contextualizes Fault Injection within the standard.

2.1. Basic Concepts of Fault Injection

Following the nomenclature provided by Dubrova et al. [3], a *fault* is physical defect, imperfection, or flaw that occurs in some hardware or software component. Resulting from a fault, an *error* is a deviation from the expected computational value. A single error or multiple ones can lead to a system *failure*, which translates into severe system degradation. Fault injection is a technique that aims to apply faults directly on hardware, software, or architectural models, to test and assess the effectiveness fault tolerance or safety mechanisms. Contrarily to the analytical methods, this aims to experimentally observe the system behaviour when deliberately injecting faults. In safety-critical applications, such as the automotive industry, FI has become a de facto practice to improve safe design and avoid the costs associated with untested safety-critical software [2].

Depending on the FI method, injection strategies can be mainly classified into *hardware-based*, *software-based* and *simulation-based* [5]. A hardware-based strategy is performed at physical level, disturbing the physical components with parameters of the environment (heavy ion radiation, electromagnetic interference, etc.), voltage glitching on power rails, or modifying the value of the pins of the circuit. This type of technique requires specialised hardware setups, and only FI via test access ports can be achieved with COTS hardware while retaining the repeatability and controllability necessary for detailed post-injection analysis. Works based on this approach include injection frameworks such as RIFLE [6], FIST [7], and MESSALINE [8].

Software-implemented FI (SWIFI) uses the actual target running the application software with additional injection procedures that modify the contents of registers and memory elements to emulate the effect of real-world hardware faults. It offers less destructiveness than hardware FI, but high intrusiveness and low reachability, as it can only reach internal processor states. Notable SWIFI tools include FERRARI [9], XCEPTION [10], and DOCTOR [11].

Lastly, Simulation-based Fault Injection (SFI) applies faults into a system or hardware model. The injection of faults can be performed by modifying either the state of the hardware components (e.g., flip-flops), the state of the architectural resources (e.g., register

file), or the state of the software structures (e.g., variables). This technique offers virtually maximum reachability and traceability, but the model details should be accurate enough for meaningful simulations. Some SFI approaches base the Fault Injection strategy on cycle-accurate models implemented by means of Hardware Description Languages (HDLs) at the Register Transfer Level (RTL), while others use instruction-accurate models of processors and software modules to emulate hardware behaviour.

FI techniques based on HDLs, such as the MEFISTO [12] and VERIFY [13] tools, use faulty signals connected to VHDL models to provoke system failures, while the authors of [14,15] use Verilog for the same purpose. This approach provide a high degree of controllability and reachability, but the main drawbacks in this case are represented by the large development effort required to model the simulator, and by poor simulation performance which drops proportionally by increasing the accuracy of the target model. Also, traditional event-driven (gate-level) or cycle accurate (RTL) simulation is typically orders of magnitude slower than real hardware [16].

Therefore, higher levels of abstraction are preferred if the underlying simulators are fast and do not compromise the accuracy of the simulation. The fastest solutions are represented by purely functional simulators that can almost reach the speed of the simulated hardware. However, simulating low-level faults could be very misleading when the simulation is only functional. Following these considerations, approaches based on instruction-accurate simulators, which rely on fast virtual platform systems that can perform simulation at an higher level of abstraction, such as at micro-architectural level, seem to be preferable to RTL-based simulators. On this level of abstraction, several studies were made based on simulators such as GEM5 [17,18] and QEMU [16,19–23]. Since this type of simulator is a significant part of this manuscript, the next section provides a more in-depth research regarding the current state of the art of this type of simulation.

2.2. Fault Injection in Virtual Platforms

Micro-architectural FI tools aim to allow designers to emulate faults at processor state level and verify the efficiency of fault tolerance solutions with low overhead, high repeatability and reachability, and low intrusiveness. Although these are desirable characteristics, these are heavily dependent on the simulator used and on how much the tool internals need to be modified to achieve meaningful injections. The following section provides an overview of tools that were extended with these characteristics in mind, focusing on the GEM5 and QEMU simulators for Fault Injection campaigns.

The authors of [17] propose a framework, supported by GEM5 and M*DEV, that allows the assessment, flaw identification, hardening, and profiling of software architectures resilience against soft errors. The Fault Injection module supports single-bit upset (SBU) and multiple-bit upset (MBU) faults injected in registers and memory addresses during runtime. The fault occurrence, location and injection time are assigned by a random uniform function. The framework also provides a analysis module, that retrieves fault campaign results and provides analytics about code execution and classifies fault run results.

The authors of [18] propose a simulator-agnostic framework, validated using GEM5 and QEMU, to assess the efficacy of SIHFT mechanisms. The fault models are user-defined by a runtime abstraction and need to be implemented for each simulator. Reachability and fault models depend on the verboseness of the controlled back-end system. Injection is made directly during the simulation execution loop, but can also be performed by test-port, such as JTAG probes. Furthermore, the framework performs post-injection analysis in the form of mapping failure to code.

Using QEMU, the authors of [19] presented QEFI, a framework which aims to assess system behaviour, focusing on simulating hardware faults and testing software reactions to them. It was designed for supporting the ARM architecture for both system-wide and kernel-based FI. Fault models include permanent faults in CPU, RAM, system peripherals and *bit-flips* in memory. Injection triggers are made by Program Counter (PC) value, injection by user-defined probability or by external application triggering. The internal

source code was modified, mainly the Tiny Code Generator (TCG), to inject faults during runtime. In addition to runtime injection, faults can also be injected while debugging with GDB protocol. The protocol has been enhanced to support not only breakpoints and watch points, but also injection points. Deeper into QEMU, the authors of [20] proposed FIES, a framework focusing on the implementation of the IEC 61508 standard [24] fault models. These include register faults in cells and address decoding, faults in CPU instructions and condition flags, and faults in DRAM memory cells and address decoding. Fault models include transient *bit-flips* to simulate SEU's and permanent *stuck-at* faults. Similarly to [19], QEMU version 2.1 TCG was modified to allow injection within the simulation execution loop. Architecture support is limited to the ARM architecture, and faults are user-defined through an XML file, passed as argument upon simulation start. Extending the fault triggering capabilities of previous works, the authors of [16] used QEMU to implement FI at register level. The focus was made for both the register operands and the register file status bits, e.g., CPSR in ARM architecture. The fault models include permanent *stuck-at*, permanent transition, and both transient and intermittent *bit-flips*. Comparing with other works, it provides the most complete regarding fault models, at the cost of multiple fault locations. The work focused on the ARM and the x86 architectures.

Contrarily to both works previously mentioned, the authors of [21], did not change the internal QEMU source code. Instead of modifying the TCG, the authors used the TCG plugin interface for code instrumentation released with QEMU 4.2 to monitor execution and perform injection. This solution method allows the injection to be more architecture-agnostic than solutions that required TCG modifications. Support fault models include transient and permanent faults in CPU instructions, RAM and registers. Injection trigger is made on user-defined CPU instructions. Execution results including the register contents throughout simulation, fault specification, memory dump and target-to-host translation info, are logged in raw format for post-analysis, if necessary. To validate the work, the authors executed a physical experiment using laser Fault Injection to cross-check the results from the experiment with the simulation. They have concluded that the modifications made for FI can be used to predict the timing and location of the fault candidates for a fault attack. Similar work was performed in FIG-QEMU [22], where the authors adopted a GDB-based approach to evaluate the robustness of application software. It focused on emulating single-event upsets on CPU registers and is heavily dependent on the program debug symbols.

More recently, the authors of [23] proposed a test bench, based on QEMU, to assess the efficacy of SIHFT methods for fault models proposed on the ISO 26262 standard. The test bench primary modules are a GDB-based fault injector and a classifier entity for result analysis. Injection is made by starting and stopping the simulation using the debugger stub, meaning that no changes in the simulator source code are needed. This approach grants high versatility, since it is not bound to any architecture. By monitoring execution, faults can be injected in into the program counter (PC), the register file, and system variables and memory. Furthermore, the authors also propose a classification according to the standard, made automatically by the classifier, which receives log data runtimes watches, that monitor variables and memory locations. One of the main takeways from this work is that the fault models provided in the standard can be correctly emulated using QEMU. The authors proposed that fault models supported by the test bench can be mapped into the failure modes reported in Table 30 of the standard, part 11, concerning the central processing unit (CPU), the interrupt handling (INTH), and the interrupt control unit (ICU). Regarding performance, the tests shown an average execution time of about 31 s per injection (comprising the time needed for logging and classification).

To summarise the mentioned works, Table 1 presents a simplified description of each tool. The table lists the works by year and provides the simulator used and the injection methodology, either by changing the internal simulator software or by using a debug probe attached to the running program. Furthermore, it shows the fault locations and models supported by the tools.

Table 1. Summary of related works in Fault Injection developed on virtual platforms.

Works	Year	Simulator	Method		Fault Location	Fault Model
			Extension	Debug Probe		
Chyłek et al. [19]	2012	QEMU	✓	✓	CPU, RAM, MMIO	Permanent *stuck-at*, bit-flips
Höller et al. [20]	2015	QEMU	✓		CPU, RAM	Permanent *stuck-at*, transient, bit-flips
Schirmeier et al. [18]	2016	gem5 + QEMU	✓	✓	Register, RAM	Transient, bit-flips
Ferraretto et al. [16]	2016	QEMU	✓		Register	Transient, permanent *stuck-at*, intermittent, bit-flips
An et al. [22]	2020	QEMU		✓	Register	Transient (SEU)
Hauschild et al. [21]	2021	QEMU	✓		CPU, RAM, Register	Permanent *stuck-at*, transient
Gava et al. [17]	2022	gem5	✓		Register, RAM	Bit-flips
Sini et al. [23]	2022	QEMU		✓	CPU, Register, RAM	Transient
Our work	2024	QEMU	✓		CPU, Register, RAM, MMIO, FLASH, IRQ	Permanent *stuck-at*, transient, bit-flips

2.3. ISO 26262

The ISO 26262 standard, released in 2011, focuses on the functional safety of electrical and electronic systems in road vehicles. It defines development guidelines to minimise the risk of accidents and ensure that automotive components perform correct functionality and a correct time. It is divided into twelve parts, but this work focuses only applies to a subset of them composed of: (i) part 5, *product development at hardware level*, which focuses how to prepare the hardware to prevent errors and how to retrieve architectural metrics; (ii) part 6, *product development at software level*, which focuses on the design and implementation of processing software modules; and part 11, *guidelines on applying ISO 26262 to semiconductors*, which focuses on guaranteeing safety levels on digital components.

Throughout the development phases, Fault Injection testing is recommended, either physical or through simulation, to validate the fulfilment of safety goals, i.e., requirements that are assigned to a system with the purpose of reducing the risk of one or more hazardous events. Part 5 of the standard classifies faults that may affect safety goals as *single point*, *residual*, *detected multi-point*, *perceived multi-point*, and *latent multi-point*. *Safe faults* (SF) do not impact safety critical logic either because they lack physical connection, or they are masked by a mitigating mechanism, such as a redundant system. *Single point* faults (SPF) are faults that can reach a safety critical logic, and there are no safety mechanisms, such as CRC, to detect or correct them. *Residual* faults (RF) happen in an area monitored by a safety mechanism, but might not be detected by it. *Multi-point* faults (MPF) refer to faults that the safety mechanism detects, with the implication that, for these faults to cause harm, there would need to be an additional fault. *Detected multi-point* faults are detected, within a prescribed time, by a safety mechanism, which prevents it from being latent, *latent multi-point* faults are faults whose presence is not detected by a safety mechanism nor perceived by the driver and, finally, *perceived multi-point* faults are faults that are not fully detected, but have some noticeable impact on the driving experience.

The fault classification can be used to retrieve metrics relative to the component under FI experiments. Resulting metrics are expressed in terms of the *failure rate of a item*, λ, when exposed to faults [2], as shown in Equation (1). This equation reflects the sum of different failure rates in respect to different fault types.

$$\lambda_{Item} = \lambda_{SPF} + \lambda_{RF} + \lambda_{MPF} + \lambda_{SF} \quad (1)$$

From Fault Injection results, one can retrieve the Single-Point Fault Metric (SPFM), calculated using Equation (2), which reflects the robustness of the component to single-point and residual faults either by coverage from safety mechanisms or by design (primarily safe faults). A high single-point fault metric implies that the proportion of single-point faults and residual faults in the component is low.

$$SPFM = 1 - \frac{\lambda_{MPF} + \lambda_{SF}}{\lambda_{Item}} \tag{2}$$

Another relevant metric is Latent-Fault Metric (LFM) (Equation (3)) that reflects the component robustness to latent faults either by coverage through safety mechanisms or by the driver recognising that the fault exists before the violation of the safety goal, or by design (primarily safe faults). A high latent-fault metric implies that the proportion of latent faults in the hardware is low.

$$LFM = 1 - \frac{\lambda_{MPF} + \lambda_{SF}}{\lambda_{Item} - \lambda_{SPF} - \lambda_{RF}} \tag{3}$$

Diagnostic coverage provided by safety mechanisms can also be retrieved by Fault Injection testing [2], and can be seen as the ratio, given as a percentage, between the failure rates of detected faults with respect to the failure rates of all faults, as shown in Equation (4).

$$K_{DC} = (1 - \frac{\lambda_{Undetected}}{\lambda_{Detected} + \lambda_{Undetected}}) * 100 \tag{4}$$

The diagnostic coverage expresses the effectiveness a safety mechanism. Although the standard does not provide an explicit expression, it can be inferred, theoretically, as all possible faults that lead to unsafe states that are capable of being detected by a safety mechanism. Since metrics are mathematically supported, one can claim that methods that map resulting faults into standard-compliant characterisation can automate the process of retrieving relevant metrics. This process was previously validated by the authors of [23].

3. QEFIRA: QEMU-Based Fault Injection Framework

The proposed Fault Injection framework, QEmu Fault Injection for Reliability Assessment (QEFIRA), is based on QEMU and extends its run-time environment to provide the ability to modify the target state during simulation. It monitors the execution of an emulated ARM target machine and injects faults according to a user specified fault experiment, as shown in Figure 1.

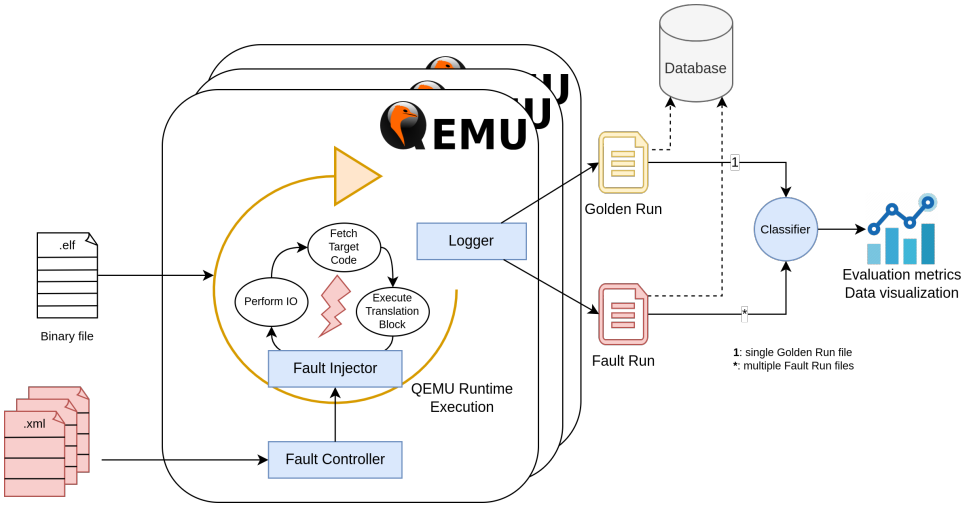


Figure 1. QEFIRA framework overview.

The framework receives a target application and user specified fault experiment files. These files contain the descriptions of the faults to be injected during the simulation, alongside variables for simulation control. These files are parsed by the Fault Controller, which creates a virtual representation of each defined fault and enqueues them for activation according to their simulation times and their respective triggers. During run-time, the Fault Injector module is aware of the enqueued faults and checks for fault triggers in the form of accesses to memory and registers, interrupt calls, and changes to the Program Counter (PC) by continuously monitoring these operations. Each one of these triggers dispatches it to verify if any fault within the list is pending activation or deactivation. Alongside this process, the internal virtual clock monitoring provides the current simulation time so that timed faults can be inserted and removed from the list according to their respective duration or start time. When an injection point activates a fault, the Logger saves the resulting system state changes. Logged data includes fault affected memory containing prior and post injection values, the PC execution flow, and user-defined memory monitor variables, all paired with the current simulation time.

A valid fault campaign is composed of a *golden run* and, at least, a *fault run*. The fault run occurs as previously described, with faults being injected throughout the simulation. The *golden run* follows the same execution, but no faults are injected. While the *golden run* executes, the system state is logged according to its non-faulty behaviour, allowing for a post-execution comparison between executions. At the end of the campaign, the resulting logs are sent to the Classifier, which compares the golden run with fault runs. It makes a suggestion about the campaign result and provides it to the Data Visualization tool which, in turn, provides a visual representation of the system state throughout the simulation. These two entities are further explained in Sections 3.4 and 3.3, respectively.

During runtime, the framework provides support to inject faults in:

Instruction Execution (CPU_INSN): Changing the currently fetched instruction from the target application. Valid for both Arm and Arm Thumb instruction sets.

Registers (CPU_REG): Modifying register file values, or modifying the register address decoding by altering register operands. The current implementation supports the Arm main register bank from R0 to R12.

Memory: Modifying memory values or blocking read/write operations on memory. Valid for both program and code memory, and for Memory-Mapped IO (MMIO).

Interrupt Handling and Control (ICU): changing the processor state by either modifying the asserted interrupt index, forcing the interrupt controller to ignore specific interrupt requests or by causing spurious interrupts.

All faults can be defined either as permanent *stuck-at*, which reproduces a permanent defect on the underlying emulated hardware, or as transient single event upsets (SEUs), modelling short-lived faults such as the ones caused by cosmic radiation [25]. As previously mentioned, faults can be triggered either when a specific instruction is ready to be executed, pointed to by the PC, or by read/write operations in a specific resource, e.g., accessing a memory location or when a specific interrupt request is asserted. By monitoring operations, the injector avoids activating latent faults that do not cause change in system state, reducing logging overhead.

The faults used on fault experiments are described by an XML file containing the properties of faults to be injected and system variables that should be monitored throughout the campaign. The XML file schema is presented in Table 2. Multiple specifications can be included in a single fault campaign, inserting multiple faults for injection. Alongside the fault specification, there the additional parameters: (i) *simulation duration*, which specifies how much time the simulation should run, and (ii) *monitor*, which specifies a memory address to continuously monitor for changes, e.g., an application level state variable. Various *monitor* items can be defined for more information about the system running state.

Table 2. XML fault specification structure.

Label	Description
<component>	CPU_INSN, CPU_REG, RAM, FLASH, MMIO, or ICU
<target>	For register faults, either address decoder or register cell value. For ICU faults, either swap/ignore IRQ request, or spurious interrupt. For memory, memory cell value or block an IO operation.
<trigger>	Trigger by resource access or by PC
<trigger access>	Register number, memory address, PC value or victim IRQ index
<type>	Permanent or transient
<timer>	Start time of permanent and transient faults
<duration>	Duration of transient faults (latent)
<cpu index>	CPU core index to inject fault
<instruction>	Instruction that should be replaced for CPU faults or new triggered IRQ index
<mask>	New *bitmask* definition for memory, register, and instruction faults
<set_bit>	Mask to select if bits defined in <mask> should be set or unset

3.1. QEMU Internal Changes

Integration of the framework was made on QEMU 8.2 focusing on the ARM architecture. The native code translation and IO loops were modified to allow monitoring of target memory and instructions during the course of the simulation. This monitoring is needed, since QEMU caches blocks of executable host code, named Translation Blocks (TLB), to prevent continuously repeating translation of target code, thus accelerating the simulation. The QEMU's execution loop can be seen in Figure 2 along with a high level-representation of where the monitoring and injection points were inserted within the loop.

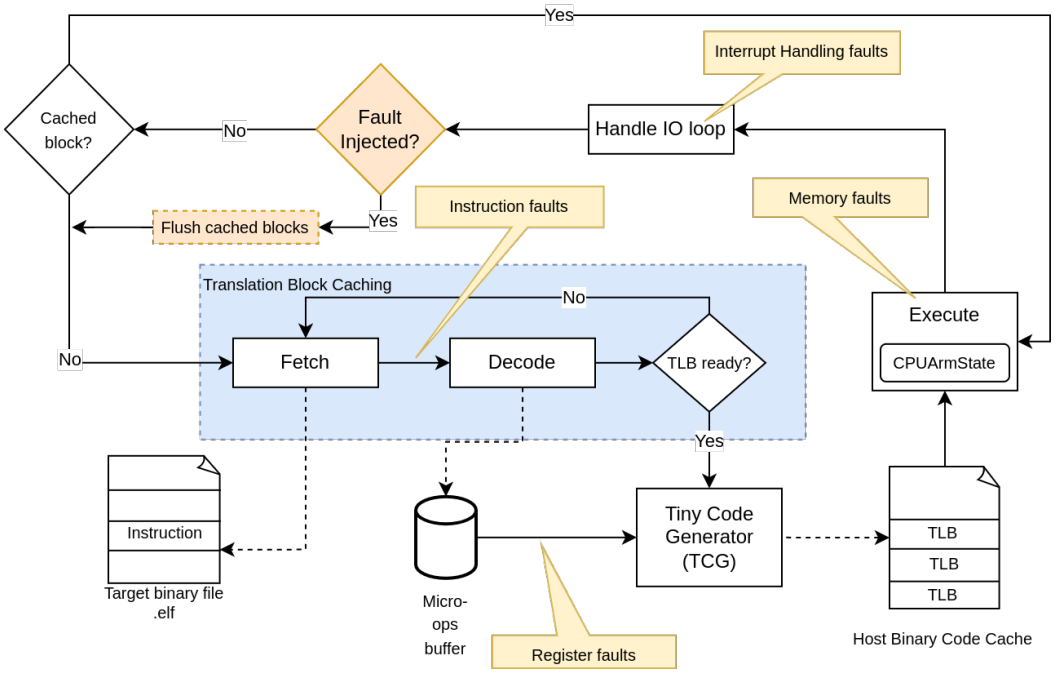


Figure 2. QEMU internal execution loop with injection points: execution loop was modified to consider monitoring of data and instructions. Dashed lines represent accesses to cached information regarding dynamic translation.

As shown in the figure, injection points are inserted either before target code is translated or within the IO loop. The instruction faults are inserted when the target code is fetched from the application binary file. This process runs at least once, with the TCG using target instructions, translating them into host code in form of TLB's and caching them for performance. Code fetching is the main entry point for PC triggered faults. After fetch, instructions are enqueued and checked for register accesses. At this point, register faults come into action, for both register cell value and register address decoding, before all code is translated into host. During the execution loop, all types of memory access through the software-emulated Memory Management Unit (MMU) are monitored for injection point triggers. Prior to entering the IO loop, the system is able to handle exceptions in the form of interrupt requests or debug handling. At this point, interrupt requests can be contaminated or ignored. After every successful injection, all translation caches are emptied to force instruction re-translation. This is important for timed faults, since cached instructions may avoid subsequent triggers during simulation. All modifications are contained within the QEMU internal translation loops, outside TCG action. Logging points are synchronized with injection points, providing a valid execution flow between the execution and IO loops.

During the simulation, the internal *virtual clock* provides near deterministic execution of instructions. This is enforced by the usage of the QEMU's *icount* parameter, which provides a simulation time that is proportional to the emulated instructions and is not impacted by the host wall-clock time. By defining this parameter, one target instruction counter tick equals 2^N nanoseconds, with N being the user specified *icount* value. With a deterministic simulation time, transient faults and delayed injections can be triggered and deactivated with increased time granularity.

3.2. Benchmarks

One important metric to gather when developing simulation extensions is how the tool performance is affected. With that in mind, two benchmarks were performed to have an overview of how the QEMU runtime is affected. The graph in Figure 3 shows the wall-clock time comparison, in seconds, between the runtime injection test with QEFIRA and a GDB-based implementation, such as the one proposed in [23]. The approaches are compared against a baseline execution of two ARMv8 target applications running ThreadX real-time operating-system that: (i) perform a Triple Modular Redundant (TMR) calculation of a block of RAM memory; and (ii) perform bubble sort of an array of 300 four byte words. Both applications run for a total simulation time of ten seconds. The injection tests performed alongside the baseline are composed of the execution of the previously mentioned application with a fault experiment with the following description:

- Three SEUs on a block of RAM which applies a bitmask of *11223344h*, from 100 ms to 200 ms simulation time.
- A spurious UART interrupt every time the processor reaches a data processing function.
- Replacement of register cell R3 contents with a random 32-bit value, triggered by PC.

The testbench was repeated for the QEFIRA runtime injection with and without the Fault Logger. The GDB-based injection was performed by developing a script that sets breakpoints on the needed instructions, performs changes to the variables and logs the results into a text file. Tests were made on an AMD Ryzen 7 six-core PC with 24 GB of RAM, running Ubuntu 22.04.

The tests included a well-defined *icount* value, and were repeated with and without the inclusion of the *singlestep* parameter. The latter guarantees maximum granularity on TCG translated blocks, which avoids caching of big blocks of translated code. For all applications, it had no effect on the simulation results, only on the performance. Comparing with the baseline run, the changes made for QEFIRA introduced a slowdown of 1.4× for *singlestep* execution, while the GDB-based approach introduced a slowdown up to 4.62×. By discarding the aforementioned parameter, GDB-based simulations do not have meaningful changes in performance. This is due to the fact that, whilst using the GDB stub, QEMU runs automatically in *singlestep* mode. On the other hand, QEFIRA had a slowdown

of 3.8× comparing with the baseline. However, its overall speed, compared with *singlestep* execution, increased 4.8× in the bubble sort benchmark and 3.34× in TMR benchmark. On either application, logging had close to no influence on performance. Comparing both implementations, QEFIRA's runtime based Fault Injection is, at least, 3× faster than a GDB-based implementation. This value increases when the *icount* parameter is well-defined, and does not provoke changes in simulation results.

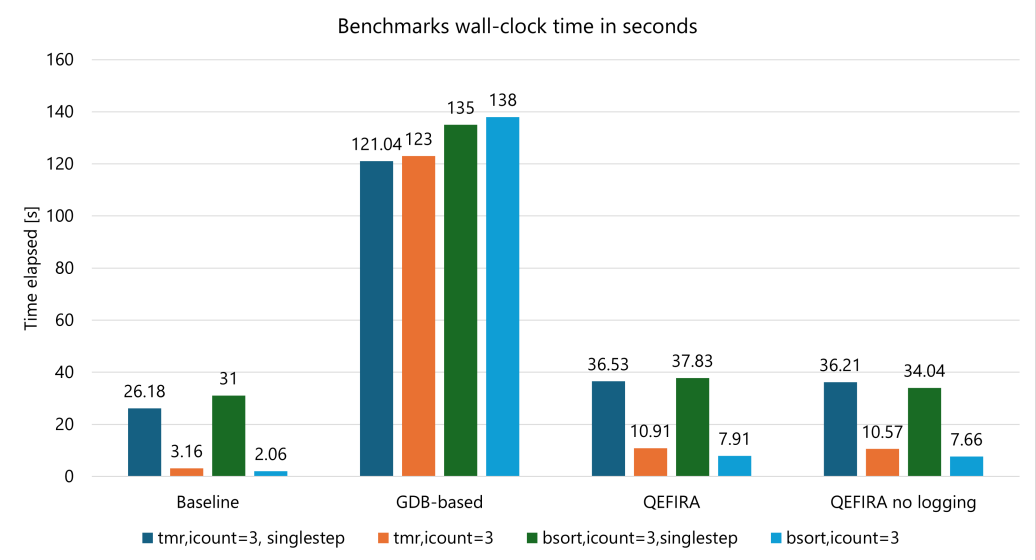


Figure 3. Results of the benchmarks for each simulation type.

3.3. Logging and Data Visualization

The Data Visualization entity retrieves all data from the Logger and provides a visual summary of the simulation in a *web-based* application, as shown in Figure 4. This extension provides an overview of the program execution flow and corresponding software instructions, fault affected memory, and the resulting fault run classification, which is better explained in Section 3.4. Execution flow is shown by charting the PC value sequence throughout the simulation of both the *golden run* and *fault run*, shown as blue and red dot graphs, respectively. Within this graph, the injection points are highlighted according to their affected PC value and simulation time, whilst, on the right side of the graph, the fault-affected instructions are also automatically highlighted. Below the execution flow graph, the log shows the changes in the memory values pre-injection and post-injection.

The example used in the figure refers to a TMR software similar to the one used in the benchmarks of the previous section, regarding both a *golden run* and *fault run*. The software loops between three states: (i) the *Input* state, which waits for three different 256-byte values to be written into a block of RAM starting in *0x200020A0* by an acquisition thread; (ii) the *Computation* state, which performs bit-wise triple modular calculation of acquired values; and (iii) the *Output* state, which posts the correct value and logs the result into a UART for debug purposes. Throughout the simulation, two faults were injected: (i) a permanent *stuck-at* fault at PC value *0x66C*, avoiding half the computation loop, and (ii) an SEU starting at 22 millisecond simulation time at the start of the RAM block. As shown in the figure, the SEU injected in the RAM is shown in the affected memory log, and is highlighted both in the binary code and in the *fault run* graph. The two charts present an offset in execution, highlighted in the figure by a red ∗, resulting from a permanent fault that exits the computation loop ahead its termination, offsetting the executed instructions.

Alongside the visual representation, the tool provides the user with a result according to how the system behaved with the executed fault experiment under the Fault Classification tab. The Classifier provides the user the proposed fault classification, by showing, within the aforementioned tab, the possible classifications and asserting the correct classification label. The automatic classification procedure is further explained in the next section.

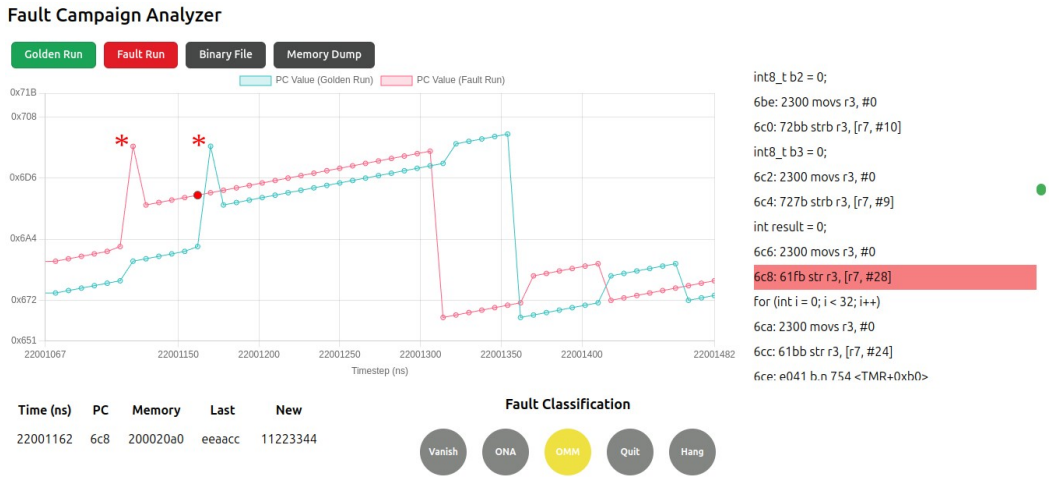


Figure 4. Snapshot of the data visualisation tool: blue and red lines show the evolution of the PC value throughout the execution, while the bright red dots show injected faults. The target application assembly code and the memory log at fault location are also shown to the user.

The monitored memory addresses are used to automatically generate Finite State Automatas (FSM) from a user-selected address, such as the one shown in Figure 5. This feature is particularly helpful in low complexity software implementations, such as simple redundancy check algorithms, since they employ a well-defined finite state machine that expresses their functionality. By monitoring its internal state variables, the automatas visually demonstrate how the system behaves in each experiment, and the state changes can be used to infer about execution flow. The automata creation algorithm verifies all the different values and assigns a state to each different value. Since the FSM in Figure 5 is created from a previous example, the resulting state *q0* shows the initial state where all is initialized, and the looping states *q1*, *q2*, and *q3*, are, respectively, the *Input*, *Computation*, and *Output* states. At the current state of development, the tool expresses only the automatas in terms of their output states, not their state triggers. For this reason, state changes are always shown as '1' and '0'.

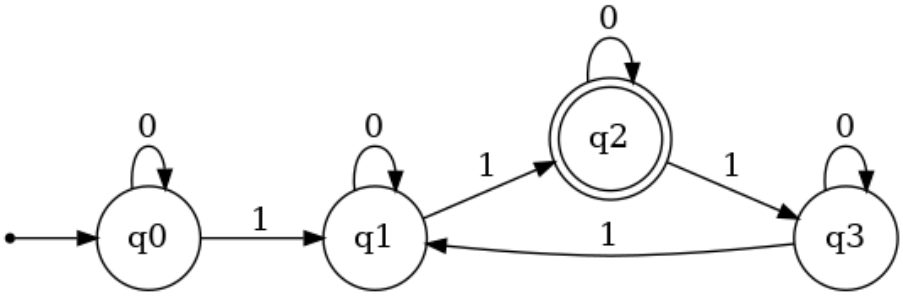


Figure 5. Finite state automata generated by the Data Visualization entity for a low complexity application.

All data used for the fault experiments and resulting log files are stored into a database, allowing for posterior analysis and more in-depth examination. For the presented example, the log files size of both *fault run* and *golden run* for a simulation time of 10 s *wall-clock time*, was approximately 9 megabytes (MB), excluding the application binary file.

3.4. Automatic Fault Classification

With the log files from the Fault Logger, the Classifier provides an automatic classification regarding the *fault runs*. Each run is classified using the proposal by the authors of [26], which categorises Fault Injection experiment outcomes into five groups: (i) Vanished, where no fault traces are left; (ii) Application Output Not Affected (ONA), where the resulting instruction flow is not modified, but one or more remaining bits of the architectural state

is incorrect; (iii) Application Output Mismatch (OMM), where the application terminates without any error indication, and the resulting memory is affected; (iv) Unexpected Termination (UT), where the application terminates abnormally with an error indication; and (v) Hang, where the application does not finish requiring a preemptive removal.

The Classifier entity, previously depicted in Figure 1, uses all Logger information and compares them with the baseline *golden run* system state. The fault classification is based on the following criteria:

- Vanished: When the *golden run* PC execution flow, and both fault-affected and monitored memory match the *fault run*, whilst the *fault-run* contains at least one active fault during simulation. Valid of for latent faults.
- ONA: the resulting monitored variables values and PC execution flow is equal, discarding fault-affected memory differences.
- OMM: the monitored state variables are different, resulting in different instruction flows.
- Unexpected termination: when the simulation ends before the expected simulation duration.
- Hang: when internal simulation watchdog triggers at 2× the expected simulation duration.

With all QEFIRA's functionalities properly presented, the next section provides an overview on how the tool can be used to reach ISO 26262 compliant classifications.

4. ISO 26262 Compliant Fault Models

The latest addition to the ISO 26262 standard, part 11, presents a proposal of failure modes for digital memory components, such as Flash and RAM, and non-memory components such as Central Processing Units (CPU), Direct Memory Access (DMA) modules, and interrupt controllers [2]. Furthermore, it also specifies how faults are characterised according to their duration. The standard specifies that a physical fault, represented by its fault model abstraction, can either be Permanent *stuck-at* or Transient:

- Permanent: (i) stuck-at fault, (ii) open-circuit fault, (iii) bridging fault, and (iv) single-event hard error.
- Transient: (i) single-event transient, (ii) single event upset, (iii) single bit upset, (iv) multiple cell upset, and (v) multiple bit upset.

These fault modes are supported by QEFIRA, as both timed permanent and transient faults can be specified for injection. Furthermore, the conjunction of the parameter *mask* and *set_bit* allows for single or multiple bit upsets. Further analysis of part 11 of the standard reveals that QEFIRA is able to emulate the faults that provoke the proposed failure modes for digital components. Targets for these failure modes include the CPU instructions, the CPU Interrupt Handler circuit (CPU_INTH), the Interrupt Controller Unit (ICU), the DMA controllers, data memory coherency (DATA), and communication peripherals (COM).

The proposed failure modes and respective fault models supported by QEFIRA were compiled into Table 3.

The first column provides the identification of the targeted failure mode according to its part/subpart, i.e., the digital component affected, followed by its description as per the standard part 11 specification. The third column proposes the fault model behaviour that achieves the correct failure mode. The standard does not provide the explicit behaviour to achieve the failure modes, meaning that the tool designer's responsibility to correctly model faults and the user responsibility to correctly specify them in fault run inputs. This means that supporting a larger fault space can tackle different ways to achieve the same failure mode, improving coverage at the cost of an higher effort on fault experiment specification. Lastly, the final column provides the target component that should be specified in the fault experiment schema to achieve the failure mode.

Table 3. Details of fault locations and fault modes according to ISO 26262, part 11.

Identification	Description	Behaviour Model	Component
CPU_FM1.1	given instruction flow not executed due to program counter (PC) hang up	PC triggered fault to force control flow to be outside the program context or trigger an exception	INSN or REG
CPU_FM1.2	given instruction flow not executed due to instruction fetch hang up	PC triggered fault to force control flow to be outside the program context or trigger an exception	INSN or REG
CPU_FM2	unintended instruction flow executed	PC triggered fault to force control flow to jump to a wrong instruction	INSN or REG
CPU_FM3	incorrect instruction flow timing	PC triggered fault to skip or omit various instructions (early or late program termination)	INSN
CPU_FM4	incorrect instruction flow result	PC triggered fault to create a control flow different from the original program, forcing wrong flow execution	INSN or REG
CPU_INTH_FM1	ISR not executed (omission/too few)	PC is not allowed to jump into the ISR handler by forcing it to jump to another instruction	INSN
CPU_INTH_FM2	unintended ISR execution (commission/too many)	spurious fault to force control flow to enter an ISR	INSN
CPU_INTH_FM3	delayed ISR execution (too early/late)	or late ISR attendance, use a timed PC triggered fault to loop the same instruction, for early, inject a PC triggered fault forcing a to jump into an ISR handler	INSN or ICU
CPU_INTH_FM4	incorrect ISR execution	PC triggered fault to create a control flow different from the original program, forcing wrong execution flow	ICU
ICU_FM1	interrupt request to CPU missing	upon an interrupt request, ignore pending requests on the NVIC	ICU
ICU_FM2	interrupt request without trigger event	fault injected in the NVIC such that the CPU will attend to the ISR	ICU
ICU_FM3	interrupt request too early/late	for early requests, inject a random NVIC request (rogue interrupts) and for late, ignore pending NVIC requests followed by a random one	ICU
ICU_FM4	interrupt request sent with incorrect data	fault injected in the NVIC such that the IRQ number does not match the correct one	ICU
DMA_FM2	data transfer without a request	fault injected as memory write at arbitrary PC	RAM
DMA_FM4	incorrect output	fault injected after request with memory contents altered with bit flips	RAM
DATA_FM1	write to memory not executed (omission)	fault injected upon store instruction ignoring it or by blocking operations	RAM or REG
DATA_FM2	unintended write to memory (commission)	fault injected randomly by writing on a memory location or during store instructions	RAM or REG
DATA_FM4	content of memory is corrupt	fault injected after request with memory contents altered with bit flips	RAM or REG
COM_FM1	no message transferred as requested	fault injected in the communication with virtualized data bus	MMIO
COM_FM3	message transferred too early/late	both a block write and timed fault injected at the peripheral memory	MMIO
COM_FM4	message transferred with incorrect value	fault injected after request with memory contents altered with bit flips	MMIO

ISO Compliant Fault Classification

Fault experiment results can help to reach an ISO 26262 compliant classification by testing critical-paths monitored by SM's against well-defined single faults. A compliant fault classification is supported by the proposed confusion matrix on Figure 6, adapted from the classification of analog circuitry in [27]. The classification is made according to: (i) the ability by the SM to detect the fault, and (ii) the SM efficacy to mitigate it. This characterisation designates the faults either Safe or Dangerous (matrix rows), whilst being Detected or Undetected (matrix columns) by the SM. The simulation results provided by QEFIRA's Logger can help determine if the SM detected the fault either by analysing the execution flow for a safe state trigger or by monitoring an output variable, such as a detection flag. The efficacy of the SM is indicated by the fault run result provided by the Classifier. Experiments classified as Vanished and ONA imply that the SM has yield favourable results, whilst other classifications imply failure.



Figure 6. ISO 26262 compliant fault classification confusion matrix adapted from [27].

Most of the ISO 26262-defined fault classifications presented in Section 2.3 can be mapped in the matrix. The exception is single point faults which, by definition, assume that no safety mechanism exists to detect and correct them. Notably, latent faults belong on the left side of the diagram, because they are not detected by the SM. Since a latent fault does not cause any function failure by itself, it belongs in the upper quadrants, i.e., Safe. Although, this could also imply a fault affecting the SM itself, rendering it Dangerous with a subsequent fault. The rationale for perceived faults to be cross-plane in the quadrants is similar to latent faults, but pertains to a failure in the SM that asserts a safe state. The fault itself is Dangerous, but tolerable by the safe state. Residual faults belong in the lower left quadrant, as they violate safety goals. On the other hand, Safe faults do not affect the system, as they are always mitigated by the safety mechanism. Lastly, detected faults belong in the lower right quadrant, since the SM will be able to detect but not mitigate them.

As mentioned in Section 2.3, fault metrics can be retrieved from fault classifications using the matrix results. From the matrix, the diagnostic coverage metric of an SM can be calculated by the weighted sum of all Dangerous Undetected (DU) faults as a percentage of the sum of all potential faults, as shown in Equation (5).

$$DC_{\%} = (1 - \frac{DU}{SU + SD + DU + DD}) * 100 \quad (5)$$

A more refined version of the previous equation tackles only faults that will, from earlier analysis, guarantee to jeopardise the system. This metric, namely DC-Residual, is calculated as the likelihood-weighted percentage of the Dangerous faults (DD and DU) that are Detected (DD), as presented in Equation (6).

$$DC_{\%Residual} = \frac{DD}{DU + DD} * 100 \quad (6)$$

Furthermore, the Single Point Fault Metric can be calculated according to Equation (7).

$$SPFM = 1 - \frac{SU + SD + DD}{SU + SD + DU + DD} \tag{7}$$

SPFM is calculated as the likelihood-weighted sum of the multi-point and safe faults, i.e., SU, SD and DD, as a percentage of the likelihood-weighted sum of all potential faults. This metric covers all faults that are out of the SM scope.

5. Conclusions and Future Work

In this paper, we introduce QEFIRA, a Fault Injection framework built on QEMU runtime, designed to aid the assessment of SIHFT mechanisms on embedded software. Furthermore, we proposed its usage for modelling faults compliant with the ISO 26262 automotive functional safety standard, enabling developers to better evaluate the efficacy and cost of software-implemented safety mechanisms. The framework generates a comprehensive log detailing execution flow and memory dumps, complemented by automatic classification of fault experiments. This, coupled with the proposed confusion matrix, allows us to gather compliant metrics to characterize and evaluate different designs in the early stages of development, avoiding hardware destructiveness and avoiding the need for physical hardware. We also provide an extensive list of proposed fault models compliant with the standard, addressing part location and fault duration. The framework presents high *reachability*, *performance* and injection *granularity* at the cost of portability and source code intrusiveness. We view this as a trade-off favouring more accurate fault behaviour and higher campaign throughput. The Fault Injection overhead introduced into the simulator resulted in a slowdown of 1.4× comparing to the native implementation. The post-campaign visual aid provides a quick overview of system behaviour, avoiding analysing verbose log files, thereby reducing generation overhead. Regarding the proposed fault matrix, three architectural metrics can be retrieved to evaluate the efficacy of the SMs.

Our next steps include extending the framework logging capabilities and injection points into Linux-based applications, and migrate monitoring features and injection hooks into plugins to reduce intrusiveness. These changes aim to improve framework portability without significantly compromising speed. Furthermore, register-level fault models can be extended, allowing injection on all register file. Regarding fault classification, we plan to refine the confusion matrix to address cases where safety mechanisms function correctly but safety functions still fail. This should avoid any mislead information regarding latent faults or faults that can occur in the safety mechanism itself.

Author Contributions: Conceptualization, R.A.; Methodology, R.A.; Software, R.A.; Validation, R.A.; Supervision, V.S. and J.C. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by European Structural and Investment Funds in the FEDER component, through the Operational Competitiveness and Internationalization Programme (COMPETE 2020) [Project nº 179491; Funding Reference: SIFN-01-9999-FN-179491].

Data Availability Statement: Data are contained within the article.

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

DC	Diagnostic Coverage
EC	Electronic Control Unit
E/E	Electrical and Electronics
FI	Fault Injection
FSM	Finite State Machine
HDL	Hardware Description Language

IRQ	Interrupt Request
ISA	Instruction Set Architecture
ISO	International Organization for Standardization
MMU	Memory Management Unit
MPF	Multi Point Fault
OMM	Application Output Mismatch
ONA	Application Output Not Affected
PC	Program Counter
RF	Residual Faults
RHF	Random Hardware Failure
RTL	Register Transfer Level
SEU	Single Event Upset
SF	Safe Faults
SFI	Simulation-based Fault Injection
SIHFT	Software-implemented Hardware Fault Tolerance
SM	Safety Mechanism
SPF	Single Point Fault
SPFM	Single Point Fault Metric
SWaP-C	Size, Weight, Power and Cost
SWIFI	Software-implemented Fault Injection
TCG	Tiny Code Generator
TLB	Translation Block
UT	Unexpected Termination

References

1. Borkar, S. Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation. *IEEE Micro* **2005**, *5*, 10–16. [CrossRef]
2. *ISO 26262:2018*; Road Vehicles—Functional Safety. ISO: Geneva, Switzerland, 2018. Available online: https://www.iso.org/standard/68383.html (accessed on 15 March 2024).
3. Dubrova, E. *Fault-Tolerant Design*, 1st ed.; Springer: New York, NY, USA, 2013; pp. 55–58, ISBN 978-1-4614-2113-9.
4. Reghenzani, F.; Guo, Z.; Fornaciari, W. Software Fault Tolerance in Real-Time Systems: Identifying the Future Research Questions. *ACM Comput. Surv.* **2023**, *55*, 1–30. [CrossRef]
5. Arlat, J.; Crouzet, Y.; Karlsson, J.; Folkesson, P.; Fuchs, E.; Leber, G.H. Comparison of physical and software-implemented fault injection techniques. *IEEE Trans. Comput.* **2003**, *52*, 1115–1133. [CrossRef]
6. Madeira, H.; Rela, M.; Moreira, F.; Silva, J.G. RIFLE: A general purpose pin-level fault injector. In Proceedings of the Dependable Computing—EDCC-1: First European Dependable Computing Conference, Berlin, Germany, 4–6 October 1994; Proceedings 1; Springer: Berlin/Heidelberg, Germany, 1994.
7. Gunneflo, U.; Karlsson, J.; Torin, J. Evaluation of error detection schemes using fault injection by heavy-ion radiation. In Proceedings of the Nineteenth International Symposium on Fault-Tolerant Computing, Digest of Papers, Chicago, IL, USA, 21–23 June 1989; pp. 340–347.
8. Arlat, J.; Aguera, M.; Amat, L.; Crouzet, Y.; Fabre, J.-C.; Laprie, J.-C.; Martins, E.; Powell, D. Fault injection for dependability validation: A methodology and some applications. *IEEE Trans. Softw. Eng.* **1990**, *16*, 166–182. [CrossRef]
9. Kanawati, G.A.; Kanawati, N.A.; Abraham, J.A. FERRARI: A flexible software-based fault and error injection system. *IEEE Trans. Comput.* **1995**, *44*, 248–260. [CrossRef]
10. Costa, D.; Madeira, H.; Carreira, J.; Silva, J.G. Xception™: A Software Implemented Fault Injection Tool. In *Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation*; Frontiers in Electronic Testing; Benso, A., Prinetto, P., Eds.; Springer: Boston, MA, USA, 2003; Volume 23.
11. Han, S.; Shin, K.G.; Rosenberg, H.A. DOCTOR: An integrated software fault injection environment for distributed real-time systems. In Proceedings of the 1995 IEEE International Computer Performance and Dependability Symposium, Erlangen, Germany, 24–26 April 1995; pp. 204–213.
12. Baraza, J.C.; Gracia, J.; Gil, D.; Gil, P.J. A prototype of a VHDL-based fault injection tool. In Proceedings of the IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, Yamanashi, Japan, 25–27 October 2000; pp. 396–404.
13. Sieh, V.; Tschache, O.; Balbach, F. VERIFY: Evaluation of reliability using VHDL-models with embedded fault descriptions. In Proceedings of the IEEE 27th International Symposium on Fault Tolerant Computing, Seattle, WA, USA, 25–27 June 1997; pp. 32–36.
14. Kammler, D.; Guan, J.; Ascheid, G.; Leupers, R.; Meyr, H. A fast and flexible platform for fault injection and evaluation in Verilog-based simulations. In Proceedings of the SSIRI 2009—3rd IEEE International Conference on Secure Software Integration Reliability Improvement, Shanghai, China, 8–10 July 2009; pp. 309–314. [CrossRef]

15. Kaja, E.; Gerlin, N.; Bora, M.; Devarajegowda, K.; Stoffel, D.; Kunz, W.; Ecker, W. MetaFS: Model-driven Fault Simulation Framework. In Proceedings of the IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems, DFT, Austin, TX, USA, 19–21 October 2022. [CrossRef]
16. Ferraretto, D.; Pravadelli, G. Simulation-based Fault Injection with QEMU for Speeding-up Dependability Analysis of Embedded Software. *J. Electron. Test. Theory Appl. (JETTA)* **2016**, *32*, 43–57. [CrossRef]
17. Gava, J.; Bandeira, V.; Rosa, F.; Garibotti, R.; Reis, R.; Ost, L. SOFIA: An automated framework for early soft error assessment, identification, and mitigation. *J. Syst. Archit.* **2022**, *131*, 102710. [CrossRef]
18. Schirmeier, H.; Hoffmann, M.; Dietrich, C.; Lenz, M.; Lohmann, D.; Spinczyk, O. FAIL*: An Open and Versatile Fault-Injection Framework for the Assessment of Software-Implemented Hardware Fault Tolerance. In Proceedings of the 2015 11th European Dependable Computing Conference, EDCC 2015, Paris, France, 7–11 September 2015; pp. 245–255.
19. Chyłek, S.; Goliszewski, M. QEMU-based fault injection framework. *Stud. Inform.* **2012**, *33*, 25–42.
20. Höller, A.; Schönfelder, G.; Kajtazovic, N.; Rauter, T.; Kreiner, C. FIES: A fault injection framework for the evaluation of self-tests for COTS-based safety-critical systems. In Proceedings of the International Workshop on Microprocessor Test and Verification, Austin, TX, USA, 15–16 December 2014; pp. 105–110.
21. Hauschild, F.; Garb, K.; Auer, L.; Selmke, B.; Obermaier, J. ARCHIE: A QEMU-Based Framework for Architecture-Independent Evaluation of Faults. In Proceedings of the 2021 Workshop on Fault Detection and Tolerance in Cryptography, FDTC 2021, Milan, Italy, 17 September 2021; pp. 20–30.
22. An, J.; You, H.; Xie, F.; Yang, Y.; Sun, J. FIG-QEMU: A Fault Inject Platform Supporting Full System Simulation. In Proceedings of the 2020 7th International Conference on Dependable Systems and Their Applications (DSA), Xi'an, China, 28–29 November 2020; pp. 275–278. [CrossRef]
23. Sini, J.; Violante, M.; Tronci, F. A Novel ISO 26262-Compliant Test Bench to Assess the Diagnostic Coverage of Software Hardening Techniques against Digital Components Random Hardware Failures. *Electronics* **2022**, *11*, 901. [CrossRef]
24. *IEC 61508:2010*; Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems. 2010. Available online: https://webstore.iec.ch/en/publication/5515 (accessed on 15 July 2024).
25. Wang, F.; Agrawal, V.D. Single Event Upset: An Embedded Tutorial. In Proceedings of the 21st International Conference on VLSI Design (VLSID 2008), Hyderabad, India, 4–8 January 2008; pp. 429–434.
26. Cho, H.; Mirkhani, S.; Cher, C.Y.; Abraham, J.A.; Mitra, S. Quantitative evaluation of soft error injection techniques for robust system design. In Proceedings of the 50th Annual Design Automation Conference (DAC '13), Austin, TX, USA, 29 May–7 June 2013; Association for Computing Machinery: New York, NY, USA, 2013; Article 101, pp. 1–10.
27. Sunter, S. How to Measure ISO 26262 Metrics of Analog Circuitry. In 2018 Siemens Digital Industries Software Blog Post. Available online: https://blogs.sw.siemens.com/tessent/2018/04/17/how-to-measure-iso-26262-metrics-of-analog-circuitry (accessed on 2 June 2024).

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.