Fault Injection for Software Certification

Domenico Cotroneo and Roberto Natella

Università degli Studi di Napoli Federico II and *Critiware s.r.l.*

`http://www.critiware.com`

Abstract

As software becomes more and more pervasive and complex, it is increasingly important to assure that a system will be safe even in the presence of residual software faults ("bugs"). Software Fault Injection consists in the deliberate introduction of software faults for assessing the impact of faulty software on the system and improving fault-tolerance. Software Fault Injection has been included as a recommended practice in recent safety standards, and it has therefore gained interest among practitioners, but it is still unclear how it can be effectively used for certification purposes. In this paper, we discuss the adoption of Software Fault Injection in the context of safety certification, present a tool for the injection of realistic software faults, namely SAFE (SoftwAre Fault Emulator), and show the usage of the tool in the evaluation and improvement of robustness of a RTOS adopted in the avionic domain.

Keywords: Software Fault-Tolerance; Fault Injection; Software Dependability Assessment; Software Faults; Safety Certification; SW-FMEA; Software RAMS; SAFE tool

Contact info:
Email: roberto.natella@unina.it, cotroneo@unina.it
Postal address: Università degli Studi di Napoli Federico II, dipartimento DIETI, Via Claudio 21, ed. 3, 80125, Naples, Italy
Phone: +39 081 676770
Fax: +39 081 676574

IEEE Security & Privacy, special issue on *Safety-Critical Systems: The Next Generation*, vol. 11, no. 4, pp. 38-45, July/August 2013, `http://dx.doi.org/10.1109/MSP.2013.54`

1 Introduction

We are witnessing the increasing importance of software in safety-critical applications, and the increasing demand for techniques and tools for assuring that software can fulfill its functions in a safe manner. At the same time, the occurrence of severe software-related accidents emphasizes that engineering safe software-intensive systems is still a hard task [1]. Unfortunately, our ability to deliver reliable software, through rigorous development practices and processes for preventing defects and V&V activities for removing them, is behind the growing complexity of software and the shrink of development budgets. As a result, we should expect that assuring low-defect software will become more and more unfeasible in the near future.

The recent "unintended acceleration" issue in Toyota cars is an example of how difficult can be to prevent and to deal with software faults (also referred to as defects or "bugs"). Toyota cars equipped with a new electronic throttle control system (*ETCS-i*), made up of several thousands of lines of code, had a significantly high rate of "unintended acceleration" events, leading Toyota to recall almost half a million new cars. The U.S. NHTSA scrutinized the design and implementation of the system with the assistance of a team from NASA highly experienced in the application of formal methods for the verification of mission-critical systems. Even if they adopted a range of verification techniques, including static code analysis, model checking, and simulations, the cause of unintended acceleration remained unknown. Unfortunately, verification techniques cannot support conclusive statements about the safety of software.

In addition to perform rigorous development and verification to reduce the number of defects, we need to assure that the system can gracefully handle *residual software faults* that are hidden in the system, since experience showed that they cannot be avoided. In this paper, we consider a strategy, based on the *injection of software faults*, for gaining confidence that residual software faults cannot cause severe system failures, like the unintended acceleration in the throttle control system, and for improving the tolerance of the system to faulty software. Fault injection is the process of deliberately introducing faults in a system in order to analyze its behavior under faults, and to measure the efficiency of safety mechanisms [2]. In particular, *Software Fault Injection* (SFI) emulates the effects of residual software faults [3, 4], to evaluate the effectiveness of software fault tolerance mechanisms, such as assertions and exception handlers [5, 6, 7].

We discuss in this paper how Software Fault Injection can be adopted in the context of safety certification, by complementing other design and verification activities. Although safety standards suggest or recommend the adoption of fault injection, it is still unclear how Software Fault Injection can be effectively used for certification purposes, as safety standards do not provide detailed information about how to perform fault injection. Moreover, techniques and tools for Software Fault Injection have matured in the last decade, and practitioners are still unaware of the potential applications of Software Fault Injection for safety certification, as we are experiencing in joint projects with industry. We illustrate an approach and a related tool for the injection of realistic software faults, namely SAFE (SoftwAre Fault Emulator), that provides some key advantages for certification purposes since it takes into account fault representativeness. The usage of the tool is shown in a real-world case study, in which it is applied for evaluating and improving the robustness of a RTOS adopted in the avionic domain.

2 Software Fault Injection in the context of safety certification

Safety standards emphasize that software should be validated with respect to abnormal and faulty conditions. In the case of the recent ISO 26262, fault injection is explicitly mentioned among methods for unit and integration testing of software items, for instance by "*corrupting values of variables*" or by "*corrupting software or hardware components*". The NASA Software Safety Guidebook (NASA-GB-8719.13) recommends fault injection for assessing the system behavior against faulty third-party software (e.g., COTS). Even in safety standards that do not explicitly mention fault injection, such as the DO-178B and DO-178C, which refer more generally to "*robustness test cases*", fault injection is a suitable approach for verifying the robustness of software, for instance by provoking "*transitions that are not allowed by the software requirements*".

During the development of safety-critical software (Figure 1), fault injection serves as a complementary activity for verifying robustness and fault-tolerance. In particular, fault injection is recommended when the system adopts measures for detecting the effects of faults and achieving/maintaining a safe state, referred to as *safety mechanisms* in the ISO 26262. These mechanisms are defined by analyzing potential failures of components and sub-systems through a *Failure Modes and Effects Analysis* (*FMEA*) at design time. In particular, software-intensive systems require *Software FMEA* (*SW-FMEA*) activities specifically focused on software components [8]. SW-FMEA translates into *software safety requirements*, to be fulfilled through rigorous development and verification activities, and by using safety mechanisms to mitigate failures due to residual software faults. A combination of both these approaches is recommended by safety standards to achieve safety requirements.

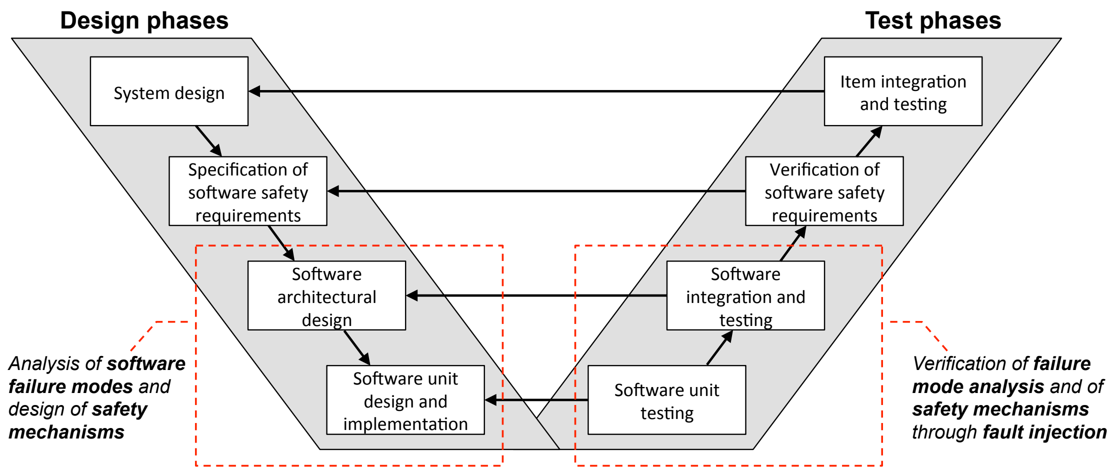

Design phases
Test phases
System design
Specification of software safety requirements
Software architectural design
Software unit design and implementation
Item integration and testing
Verification of software safety requirements
Software integration and testing
Software unit testing
Analysis of software failure modes and design of safety mechanisms
Verification of failure mode analysis and of safety mechanisms through fault injection

Figure 1: Fault injection during software development phases.

If a SW-FMEA points out that a given failure of a software component may cause severe consequences, a safety mechanism is introduced during the design phases to let the system tolerate such failures. In this case, Software Fault Injection can be adopted to force a software component to fail during tests, and to gain confidence on the effectiveness of the safety mechanism. Safety mechanisms detect and mitigate at run-time faulty software components by introducing "redundant logic" [6]. The effects of faults can be detected by:

- Comparing the outputs of two or more redundant and "diverse" (e.g., implemented by different means) software components that perform equivalent functions. An error is detected if outputs do not match, by adopting some voting schema as in the case of the *N-version programming* approach [5].

- Performing "plausibility" checks on the values produced by a component, or on the execution paths followed by the component. Examples are *assertions* in the program that point out obvious inconsistencies between program variables or output values that are outside a range, and a *watchdog timer* checking that the software is actually making progress [7].

Software Fault Injection supports the validation of these mechanisms. For instance, faults can be injected in a component to evaluate whether they can propagate to its outputs, and whether checks at its interface can detect them. Detection triggers recovery mechanisms for mitigating the effects of faults, such as:

- Mask the fault by performing multiple computations through multiple "diverse" replicas, either sequentially (e.g., Recovery Blocks) or concurrently (e.g., N-Version Programming) [5, 6]. Some faults can still be masked even if replicas are not diverse.

- Bring the system to a safe state, for instance a switch to a degraded mode of service, by giving priority to a subset of software functions, or forcing a fail-stop behavior.

Software Fault Injection can provide evidences that recovery is effective against faulty behaviors, or it can point out situations in which it is not successful. For instance, developers can assess whether the system is able to properly provide a degraded mode of service once a software failure has occurred.

Another potential application of Software Fault Injection is the validation of SW-FMEA. Software Fault Injection can reveal two kinds of FTAM (Fault-Tolerance Algorithm or Mechanism) failures [2, 6]: (i) faults in the implementation of FTAMs (lack of "error and fault handling coverage"), or (ii) incorrect or incomplete assumptions about failure modes really occurring in operation (lack of "fault assumption coverage"). FTAM failures of the second kind are due to wrong assumptions made by the designer of FTAMs about how software components can fail (e.g., exhibiting fail-stop behavior), on the basis of a potentially incorrect SW-FMEA. In general, FMEAs cannot be exhaustive, as some failure modes or effects can be missed. SW-FMEA is a difficult process, which requires an expert analyst and a detailed knowledge about the system, and as any human activity it is prone to mistakes. Moreover, software functions are usually more complex than hardware ones, and software failure modes cannot typically be obtained from datasheets or field data [8]. After performing Software Fault Injection, developers can look in detail at FTAM failures, identify those ones caused by incorrect assumptions, and revise the SW-FMEA and the system design accordingly.

Otherwise, if software components fail according to the assumptions, developers can gain confidence on the validity of SW-FMEA.

In both scenarios, the *representativeness* of injected faults is an important concern to support claims about fault-tolerance properties of the system. That is, fault injection should generate software errors (i.e., an erroneous software state) of the same kind of errors that are likely to occur during operation. For instance, to evaluate the effectiveness of an assertion, the data that is checked by the assertion should be corrupted; if the injected corruption is arbitrary and not representative of real data corruptions, then fault injection could not provide evidence about the likelihood of detecting data corruptions during operation.

3 From hardware to software fault injection

The use of fault injection to emulate the effects of software faults, namely *Software Fault Injection* (SFI), is relatively recent when compared to traditional fault injection approaches. Early Software Fault Injection tools adopted *Software-Implemented Fault Injection* (SWIFI) techniques existing at that time, aimed at emulating the effects of *hardware faults* by changing the value corrupting software code or data (e.g., using *bit-flips*), to also emulate the effects of *software faults* [9, 3]. Subsequent approaches corrupt data at the interfaces of software components, by replacing the parameters of function invocations with invalid parameters, such as invalid pointers and boundary values [10, 11], to emulate faulty interactions between components. These techniques are useful to point out corner cases in which invalid data is not correctly handled, but they are not suitable for emulating software faults in a representative way since the injected corruptions, such as bit-flips or boundary values, do not necessarily match the effects of faults hidden in the system under evaluation. The realism of fault injection is an important condition for reproducing faulty behaviors that are likely to occur in operation, and for gaining confidence on FTAMs.

The injection of realistic software faults can be achieved by introducing *artificial bugs* in the target software. This technique produces a *faulty version* of a software component, generating an erroneous behavior when it is executed. The use of code changes for emulating software faults is supported by empirical studies, which found that the injection of code changes produces erroneous behaviors similar to the effects of real software faults [12]. This way of injecting faults resembles mutation testing, but it has completely different goals: while mutation testing uses code changes to identify an adequate test suite, Software Fault Injection is meant to validate FTAMs and to analyze the system behavior in the presence of realistic faulty scenarios. This difference of goals reflects on the approaches and fault models. Mutation testing studies proposed a large number of mutation operators, in order to encompass many kinds of faults that can occur during development, for assessing the thoroughness of test cases. However, not every mutation operator is necessarily representative of *residual software faults* that escape testing, go with the deployed product, and affect the system during operation.

Several studies have been focused on the definition of a realistic model of software faults. The fault model is based on the rigorous analysis of extensive failure data of both open-source and commercial software systems [4, 13]. These studies observed the same trend in the distribution of faults: "Algorithm" faults are the dominant ones; "Assignment" and "Checking" faults have approximately the same weight; "Interface" and "Function" faults are the less frequent ones. These data encompass both OS code (e.g., hardware-management code) and user programs (e.g., compilers, interpreters, desktop applications), with varying degrees of maturity and number of users. Software faults on the field were further classified in [13] in terms of *programming language construct* that is either *missing*, *wrong*, or *extraneous* in the faulty code. The majority of faults belonged to few fault types (Table 1), which have a much higher frequency and occur consistently in most of the considered projects. This set of fault types forms a fault model reasonably independent from the nature of the program, and is suitable for automated fault injection, as it details how to manipulate a program to introduce faults, in terms of programming constructs to be removed or modified. The fault model also provides several detailed rules, not shown for brevity, describing the *code context* in which each fault type should be injected to be realistic: for instance, the removal of an *if* construct can be injected in those *if* constructs that have at most 5 statements, since it is unlikely that an *if* construct is lacking for larger groups of statements.

A limitation of these fault types is that part of field faults are not covered, as they occur only in specific projects: to increase the percentage of covered faults, the injector would require field failure data about the specific project under evaluation. Unfortunately, it is very difficult to obtain field failure data as it requires to put the system in operation for several years. Thus, the fault model focuses on fault types that are generic and can be adopted even if field failure data are not available. However, considering that code changes are able to generate errors in a similar way to real faults [12], the use of representative fault types can achieve a good degree of realism even if the fault types do not account for every possible fault.

Table 1: Most frequent types of software faults found in the field [13].

	Fault type	# of faults	% of faults
Missing	*if* construct plus statements	71	10.63%
	AND sub-expr in expression used as branch condition	47	7.04%
	function call	46	6.89%
	if construct around statements	34	5.09%
	OR sub-expr in expression used as branch condition	32	4.79%
	small and localized part of the algorithm	23	3.44%
	variable assignment using an expression	21	3.14%
	functionality	21	3.14%
	variable assignment using a value	20	2.99%
	if construct plus statements plus *else* before statements	18	2.69%
	variable initialization	15	2.25%
Wrong	logical expression used as branch condition	22	3.29%
	algorithm - large modifications	20	2.99%
	value assigned to variable	16	2.40%
	arithmetic expression in parameter of function call	14	2.10%
	data types or conversion used	12	1.78%
	variable used in parameter of function call	11	1.65%
Extra	variable assignment using another variable	9	1.35%
Total		**452**	**67.66%**

In our previous study [14], we evaluated the suitability of this fault model for safety-critical software since, given the more rigorous testing activities it undergoes, a different distribution of fault types could hold. Therefore, we analyzed how testing affects the types of residual software faults in a Real-Time Operating System (RTOS) adopted in space applications. We compared the distribution of injected faults with the distribution of faults obtained after removing faults detected by test suites. As expected, test suites were very effective in this safety-critical software, as only a minority of injected faults escaped testing. A key finding was that the distribution of fault types is not affected by test suites, i.e., the relative proportions of fault types before and after testing are the same. Instead, testing affects the distribution of faults across code locations (e.g., files and functions). Therefore, to adopt these fault types in safety-critical software, we need to tune the code location in which faults are injected to achieve fault representativeness. These findings have driven the development of the SFI tool discussed in this paper.

4 SoftwAre Fault Emulator

The *SoftwAre Fault Emulator* (*SAFE*) is a tool for supporting the automated analysis of software FTAMs and failure modes through Software Fault Injection, which has been originally developed in the context of academic research and has then evolved into a mature fault injection suite. The tool can perform the injection of software faults into C and C++ software, according to the fault model described above. Differing from previous SFI tools that inject bit-flips or invalid values [3, 10, 7, 11], SAFE emulates software faults by adopting an representative fault model, which is required in order to provide sound evidence that a system will be fault-tolerant during operation.

The fault injection approach closest to ours, which injects the fault types of Table 1 by mutating the *source code* of the target software, is represented by the G-SWFIT technique [13], which mutates the *binary code* of the target. In our previous work [15], in the context of the European project "Critical-Step" (`www.critical-step.eu`), we compared an industrial implementation of G-SWFIT with our technique. The study pointed out strengths and limitations of these techniques: binary-level injection works in the absence of source code, but the mutation of binary code is often inaccurate, and difficult to implement and to perform correctly. The SAFE tool has been improved in the context of this project, and it is now mature enough to handle very large fault injection campaigns in complex real-world software.

The tool supervises all the phases of fault injection and allows their automated execution. The workflow consists of the following phases:

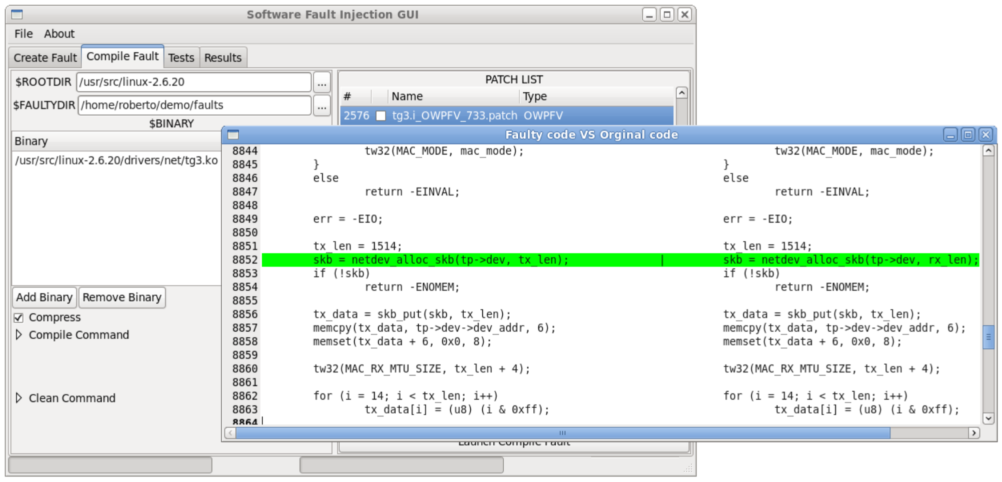
1. **Code analysis**: The tool analyzes the code of the target software, to identify code locations where faults can be injected. The code is first transformed into an abstract representation (in the form of an abstract syntax tree), which is then analyzed to identify which constructs in the program fulfill the rules of the fault model and are suitable for injecting realistic faults.

2. **Fault generation**: For each fault identified in the previous phase, the fault is actually injected and a faulty version of the software is obtained (Figure 2a). During this phase, the user can select a subset of faults to inject, by configuring a filtering criteria. Possible criteria are to inject only a subset of fault types, and to inject only in a subset of code locations, for instance to inject faults only in the parts of the software that are deemed most defect-prone according to software complexity metrics [14].

3. **Test execution**: A test is executed for each faulty version of the software generated during the previous phase. At each experiment, the tool: cleans residual errors from a previous experiment by stopping the system, starts the target system with a new fault, executes a workload, shuts down the target system after a fixed time, and collects failure data. Since these operations are system-specific, the tool allows the user to customize them using a scripting language (Figure 2b). Failure data can be collected at the end of an experiment, *after* the occurrence of a failure, such as a dump of memory and of CPU registers, and error logs generated by the system. Collecting *post-mortem* data avoids introducing excessive overheads in the system execution, which is especially important in real-time systems. If necessary, the tester can perform an additional in-depth analysis of experiments that exhibited FTAM failures by collecting and analyzing *execution traces* of the system (e.g., address and data signals produced during an experiment). To have acceptable overheads, the collection of execution traces should be performed using a *hardware debugger*, which can be integrated with SAFE if it provides interfaces to external programs.

4. **Result analysis**: Experimental data are analyzed in order to provide the user with information about failure modes and FTAMs. The tool eases the analysis of data through user-defined scripts, which evaluate specific properties of the system. For instance, the user can instruct the tool to evaluate whether the program corrupted data, by comparing experimental data with data obtained from fault-free experiments (*golden runs*), and whether safety mechanisms were able to detect and to recover from the fault.

The costs of fault injection, given the size of current software systems, are a primary concern. There are three factors that affect the cost of a fault injection campaign: the time to setup a fault injection testbed; the time to run experiments; the time to analyze the experimental data.



$ROOTDIR /usr/src/linux-2.6.20

$FAULTYDIR /home/roberto/demo/faults

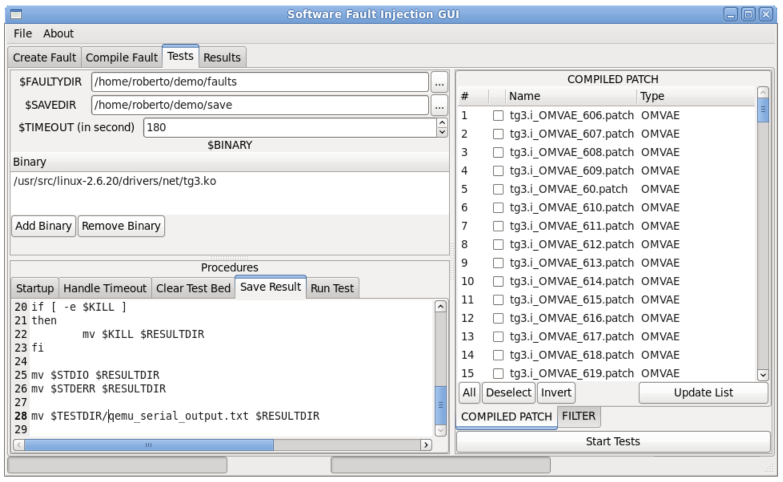
Binary
/usr/src/linux-2.6.20/drivers/net/tg3.ko

PATCH LIST

#		Name	Type
2576	☐	tg3.i_OWPFV_733.patch	OWPFV

```
        tw32(MAC_MODE, mac_mode);                              tw32(MAC_MODE, mac_mode);
    }                                                      }
    else                                                   else
        return -EINVAL;                                        return -EINVAL;

    err = -EIO;                                            err = -EIO;

    tx_len = 1514;                                         tx_len = 1514;
    skb = netdev_alloc_skb(tp->dev, tx_len);          |    skb = netdev_alloc_skb(tp->dev, rx_len);
    if (!skb)                                              if (!skb)
        return -ENOMEM;                                        return -ENOMEM;

    tx_data = skb_put(skb, tx_len);                        tx_data = skb_put(skb, tx_len);
    memcpy(tx_data, tp->dev->dev_addr, 6);                 memcpy(tx_data, tp->dev->dev_addr, 6);
    memset(tx_data + 6, 0x0, 8);                           memset(tx_data + 6, 0x0, 8);

    tw32(MAC_RX_MTU_SIZE, tx_len + 4);                     tw32(MAC_RX_MTU_SIZE, tx_len + 4);

    for (i = 14; i < tx_len; i++)                          for (i = 14; i < tx_len; i++)
        tx_data[i] = (u8) (i & 0xff);                          tx_data[i] = (u8) (i & 0xff);
```

(a) Selection and preview of faults to inject.



$FAULTYDIR /home/roberto/demo/faults

$SAVEDIR /home/roberto/demo/save

$TIMEOUT (in second) 180

Binary
/usr/src/linux-2.6.20/drivers/net/tg3.ko

```
if [ -e $KILL ]
then
        mv $KILL $RESULTDIR
fi

mv $STDIO $RESULTDIR
mv $STDERR $RESULTDIR

mv $TESTDIR/qemu_serial_output.txt $RESULTDIR

```

COMPILED PATCH

#		Name	Type
1	☐	tg3.i_OMVAE_606.patch	OMVAE
2	☐	tg3.i_OMVAE_607.patch	OMVAE
3	☐	tg3.i_OMVAE_608.patch	OMVAE
4	☐	tg3.i_OMVAE_609.patch	OMVAE
5	☐	tg3.i_OMVAE_60.patch	OMVAE
6	☐	tg3.i_OMVAE_610.patch	OMVAE
7	☐	tg3.i_OMVAE_611.patch	OMVAE
8	☐	tg3.i_OMVAE_612.patch	OMVAE
9	☐	tg3.i_OMVAE_613.patch	OMVAE
10	☐	tg3.i_OMVAE_614.patch	OMVAE
11	☐	tg3.i_OMVAE_615.patch	OMVAE
12	☐	tg3.i_OMVAE_616.patch	OMVAE
13	☐	tg3.i_OMVAE_617.patch	OMVAE
14	☐	tg3.i_OMVAE_618.patch	OMVAE
15	☐	tg3.i_OMVAE_619.patch	OMVAE

(b) Setup of fault injection experiments.

Figure 2: SAFE tool.

The setup of a fault injection testbed requires some manual effort. Since fault injection is supported by tools, most of the setup effort consists in integrating a fault injection tool into the system under evaluation. In the case of the SAFE tool, the setup consists in developing a set of scripts, which provide the commands for performing a specific operation on the target system. For instance, a script is devoted to start the execution of the target system, which may require to start an emulator or to send commands to a board through a serial or USB connection. These scripts are typically simple and small.

The time to run experiments is, in our experience, the largest share of a fault injection campaign. The campaign duration is mainly determined by the number of faults that are injected into the system, which depends on the size of the system and typically ranges from hundreds to hundreds of thousands of faults. In turn, the fault injection campaign requires from some hours to a few days to execute. Various approaches were proposed for speeding up test execution, by selecting a subset of faults to inject (among the many faults that can be potentially injected) to reduce the number of experiments and achieve confidence on the results. In our previous work [14], we proposed a heuristic that improves the representativeness of injected faults and reduces the number of experiments, by filtering out up to 70% percent of faults. When the evaluation is focused on a specific part of the system (e.g., a specific assertion), the tester should perform a further fault-filtering step in order to focus fault injection on code related to the specific part under evaluation. For instance, to assess a specific procedure, faults should be injected in those procedures interacting with it. The SAFE tool allows the user to customize which faults are injected.

Experimental data can be used for: (i) the quantitative analysis of FTAMs in terms of *coverage factors* and of *timing distributions*, and (ii) the analysis of root-causes behind FTAM failures, in order to provide feedback for the design and implementation of FTAMs. The former consists in summarizing, in statistical terms, the outcomes of experiments according to user-defined *predicates* (i.e., a concise specification of properties that must hold in the presence of faults, which are derived from *safety requirements* defined during design phases (Figure 1); for instance, a railway signaling system should never allow two trains to cross in the same section. The analysis of predicates over experimental data can be automated in SAFE through user-defined scripts. Quantitative results are also useful to support stochastic modeling and evaluation of the system [2]. In the latter, developers look in-depth at a subset of fault injection experiments that caused FTAM failures, in a similar way to debugging a program by looking at failed test cases.

As a case study, we present some experimental results from a pilot R&D project, in conjunction with the Finmeccanica industrial group. Goal of the project is to develop a Linux-based RTOS (FIN.X-RTOS), to be adopted in avionic applications, accompanied by evidences (e.g., test artifacts) showing the compliancy to the recommendations of the DO-178B safety standard, in order to ease the certification of systems based on FIN.X-RTOS. The original Linux kernel was enhanced in FIN.X-RTOS by providing real-timeliness and scalability for multi-core architectures and removing unessential parts. At time of writing, requirements of the level D standard were fulfilled, and additional verification activities are being performed according to the more stringent requirements of level C, which demand to test the robustness of the software against abnormal inputs and conditions.

In particular, we focus on robustness evaluation of the FIN.X-RTOS kernel against faulty conditions caused by *device drivers* (Figure 3). Device drivers are not part of the FIN.X-RTOS kernel, since they often need to be developed ad-hoc, or obtained from a third-party hardware-specific Board Support Package (BSP), when FIN.X-RTOS is integrated into a wider system. Unfortunately, kernel code tends to be vulnerable against faulty device drivers, since kernel developers often omit checks on the behavior of device drivers to improve performance, neglecting the risk of faulty drivers. This threat is exacerbated by the high defect rate in device drivers, and by the monolithic architecture of FIN.X-RTOS (inherited from the Linux kernel), where device drivers execute in privileged mode and can affect the whole OS [11].

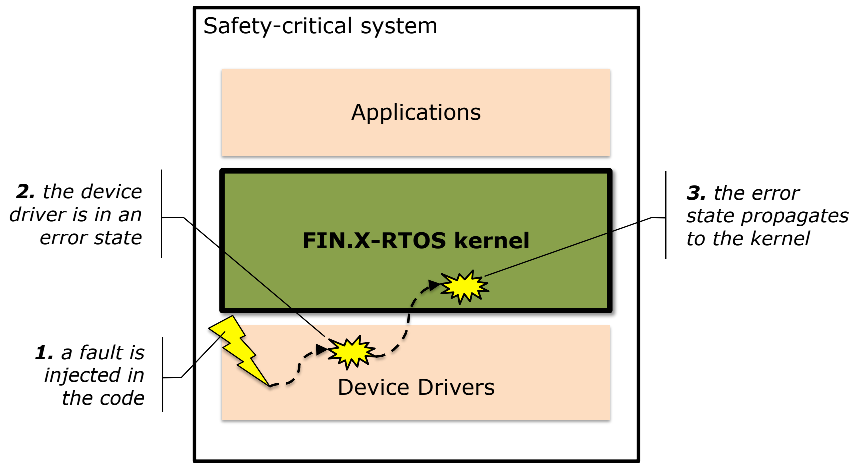

Safety-critical system
Applications
2. the device driver is in an error state
FIN.X-RTOS kernel
3. the error state propagates to the kernel
1. a fault is injected in the code
Device Drivers

Figure 3: Overview of the robustness testing case study on FIN.X-RTOS.

Software Fault Injection was adopted for assessing the ability of the kernel to prevent error propagation from device drivers to the kernel itself, by injecting faults in device drivers. The kernel is robust when the effects of faults are restricted to the faulty device driver; such faults can be tolerated by re-initializing the device driver, or by switching to a secondary device. When kernel is not able to detect and to prevent error propagation, a faulty driver can affect shared kernel data or code, and it should be hardened by applying additional safety mechanisms, such as checks on function parameters or before accessing shared data structures.

Experiments were conducted in an emulated environment, and using the SAFE tool to orchestrate experiments. We injected faults in device drivers for three Ethernet network cards (*ne2k-pci*, *rtl8139cp*, *pcnet32*), by randomly sampling 150 faults to inject for each device driver. We adopted a network-bound workload running Apache HTTPD on the target system and a request generator on the host machine. During the experiments, we collected error messages from the kernel (through a virtual serial port) and from workload applications, and analyzed these messages to identify whether a fault propagated to the kernel or to the workload.

In most cases (79.1%), the workload can correctly execute even in the presence of a faulty driver: in general, this phenomenon is often observed in fault injection experiments, since the fault may be accidentally masked (e.g., an uninitialized or corrupted variable is overwritten later in the program) or may remain latent during the experiment. However, there were several cases in which faults affected the kernel (11.3%) or the workload (9.6%). In the case of workload errors and of driver crashes not propagated to the kernel, the injected fault caused the unavailability of the network device driver, thus affecting communication between the server and the clients, and were successfully detected and signaled by the kernel through return codes of system calls.

When in our experiments the fault propagated to the kernel, it caused the stall or the termination of a kernel thread (i.e., a privileged task that runs in supervisor mode and that executes kernel code), affecting the whole OS. In general, if an exception occurs (e.g., an illegal memory access) while executing OS code, the OS tries to recover from the exception by killing the current task under execution. For instance, when a task invokes an OS system call, and the system call causes an exception or does not terminate within a fixed time period, the kernel kills the task (thus terminating the system call) to allow the execution of other tasks. We found that the exception handler can kill the current task even when it is part of the OS (i.e., a kernel thread), thus affecting the execution of the OS. For instance, a "missing variable initialization" fault injected in the *ne2k-pci* driver caused the kill of the *sirq-timer* kernel thread, which is responsible for the delayed execution of kernel functions associated to a timer. In particular, the kernel thread was executing a timer function *mld_ifc_timer_expire*, which periodically sends network messages for discovering multicast listeners. In turn, this function invokes the faulty device driver, which causes an exception since it accesses to an uninitialized data structure. When the *sirq-timer* kernel thread is killed, timer functions in the kernel cannot be executed anymore. An approach to handle this situation is to modify the exception handler in the kernel, in order to restart a kernel thread when an exception occurs instead of terminating it; in this way, the kernel could preserve the execution of other timer functions when a timer functions fails due to a faulty driver.

5 Conclusion

Residual faults are hidden in our software, and they will eventually manifest themselves during operation. This threat will likely get worse as the complexity of software steadily rises. Software Fault Injection is a means to assess, before releasing the product, the impact of software faults, and enables the evaluation and improvement of fault-tolerance. Software Fault Injection represents a reasonably mature technology for the assessment of safety-critical software, as it is able to realistically emulate residual software faults, which is a requirement for trustworthy results, and can be fully automated, which is important to make it a feasible and cost-effective approach.

Acknowledgments

This work has been partially supported by the CECRIS FP7 project (grant agreement no. 324334) and by "Embedded Systems in Critical Domains" national project (CUP B25B09000100007).

About the authors

Roberto Natella is a postdoctoral researcher at the Federico II University of Naples, Italy, and co-founder of the Critiware S.r.L. spin-off company. He received the PhD degree in 2011 in computer engineering from the same university. His research is in the area of dependability assessment of mission-critical systems, and in particular on software fault injection and on software aging and rejuvenation. He has been involved in industrial research projects with companies of the Finmeccanica group (*Iniziativa Software*). He is a member of IEEE.

Domenico Cotroneo received his Ph.D. in 2001 from the Department of Computer Science and System Engineering at the Federico II University of Naples. He is currently associate professor at the same university. His main interests include dependability assessment techniques, software fault injection, and field-based measurement techniques. Domenico Cotroneo has served as program committee member in several scientific conferences on dependability, including DSN, EDCC, ISSRE, SRDS, and LADC; and he is involved in several national and European projects in the context of dependable systems. He is a member of IEEE.

References

[1] N. Leveson, "Role of software in spacecraft accidents," *J. Spacecraft and Rockets*, vol. 41, no. 4, pp. 564–575, 2004.

[2] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J. Fabre, J. Laprie, E. Martins, and D. Powell, "Fault Injection for Dependability Validation: A Methodology and Some Applications," *IEEE Trans. on Software Engineering*, vol. 16, no. 2, pp. 166–182, 1990.

[3] J. M. Voas and G. McGraw, *Software Fault Injection: Inoculating Programs Against Errors.* John Wiley & Sons, Inc., 1998.

[4] J. Christmansson and R. Chillarege, "Generation of an Error Set that Emulates Software Faults based on Field Data," in *Proc. Intl. Symp. on Fault-Tolerant Comp.*, 1996, pp. 304–313.

[5] J.-C. Laprie, J. Arlat, C. Beounes, and K. Kanoun, "Definition and analysis of hardware-and software-fault-tolerant architectures," *Computer*, vol. 23, no. 7, pp. 39–51, 1990.

[6] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, 2004.

[7] M. Hiller, A. Jhumka, and N. Suri, "EPIC: Profiling the propagation and effect of data errors in software," *IEEE Transactions on Computers*, vol. 53, no. 5, pp. 512–530, 2004.

[8] P. Goddard, "Software FMEA techniques," in *Proc. Annual Reliability and Maintainability Symposium*, 2000, pp. 118–123.

[9] M. Hsueh, T. Tsai, and R. Iyer, "Fault injection techniques and tools," *IEEE Computer*, vol. 30, no. 4, pp. 75–82, 1997.

[10] P. Koopman and J. DeVale, "The exception handling effectiveness of POSIX operating systems," *IEEE Transactions on Software Engineering*, vol. 26, no. 9, pp. 837–848, 2000.

[11] A. Albinet, J. Arlat, and J.-C. Fabre, "Characterization of the Impact of Faulty Drivers on the Robustness of the Linux Kernel," in *Proc. Intl. Conf. on Dependable Systems and Networks.* IEEE, 2004, pp. 867–876.

[12] M. Daran and P. Thévenod-Fosse, "Software Error Analysis: A Real Case Study Involving Real Faults and Mutations," *ACM Soft. Eng. Notes*, vol. 21, no. 3, pp. 158–171, 1996.

[13] J. Durães and H. Madeira, "Emulation of Software faults: A Field Data Study and a Practical Approach," *IEEE Trans. on Software Engineering*, vol. 32, no. 11, pp. 849–867, 2006.

[14] R. Natella, D. Cotroneo, J. Duraes, and H. Madeira, "On Fault Representativeness of Software Fault Injection," *IEEE Transactions on Software Engineering*, vol. 39, no. 1, pp. 80–96, 2013.

[15] D. Cotroneo, A. Lanzaro, R. Natella, and R. Barbosa, "Experimental Analysis of Binary-Level Software Fault Injection in Complex Software," in *Proc. Ninth European Dependable Computing Conference*, 2012, pp. 162–172.