Fault Injection for Software Certification

Domenico Cotroneo and Roberto Natella | Università degli Studi di Napoli Federico II

Recent safety standards recommend software fault injection as a best practice, earning it some interest among practitioners, but it's still unclear how to effectively use it for certification purposes.

With the increased importance of software in safety-critical applications, we're witnessing a growing demand for techniques and tools that ensure this software fulfills its functions. At the same time, the occurrence of severe software-related accidents emphasizes the point that engineering safe, software-intensive systems is still difficult.[1] Unfortunately, our ability to deliver reliable software—via rigorous development practices for preventing defects, and validation and verification activities for removing them—is falling behind due to software's expanding complexity coupled with shrinking development budgets. Consequently, we should expect that assuring low-defect software will become increasingly infeasible in the near future.

The recent "unintended acceleration" issue in Toyota's car fleet is a good example of how tough it can be to both prevent and deal with software faults, defects, or bugs. Toyotas equipped with a new electronic throttle control system, which has several thousands of LOC, had a significantly high rate of unintended acceleration due to faults in the software, leading Toyota to recall almost half a million new cars. The US National Highway Traffic Safety Association scrutinized the system's design and implementation with the assistance of a team from NASA highly experienced in the application of formal methods for verifying mission-critical systems. But even though the team adopted a range of verification techniques, including static code analysis, model checking, and simulations, the exact cause of the unintended acceleration remains unknown (http://embeddedgurus.com/barr-code/2011/03/unintended-acceleration-and-other-embedded-software-bugs).

To avoid catastrophic failures like this, we must ensure that the system can gracefully handle hidden residual software faults—experience shows that they can't be avoided. In this article, we consider a strategy based on software fault injection (SFI), the process of deliberately introducing faults in a system to analyze its behavior and response in faulty conditions.[2] In particular, SFI can emulate the effects of residual software faults[3,4] to evaluate fault-tolerance mechanisms such as assertions and exception handlers.[5–7]

We also discuss how to use SFI in the context of safety certification. Although recent standards recommend fault injection, it's still unclear how to effectively use this approach because those standards don't provide detailed information. Moreover, the techniques and tools have matured in the past decade, and most practitioners are unaware of SFI's full potential for safety certification. To demonstrate what it can do, we describe an approach and a related tool for injecting

 Copublished by the IEEE Computer and Reliability Societies

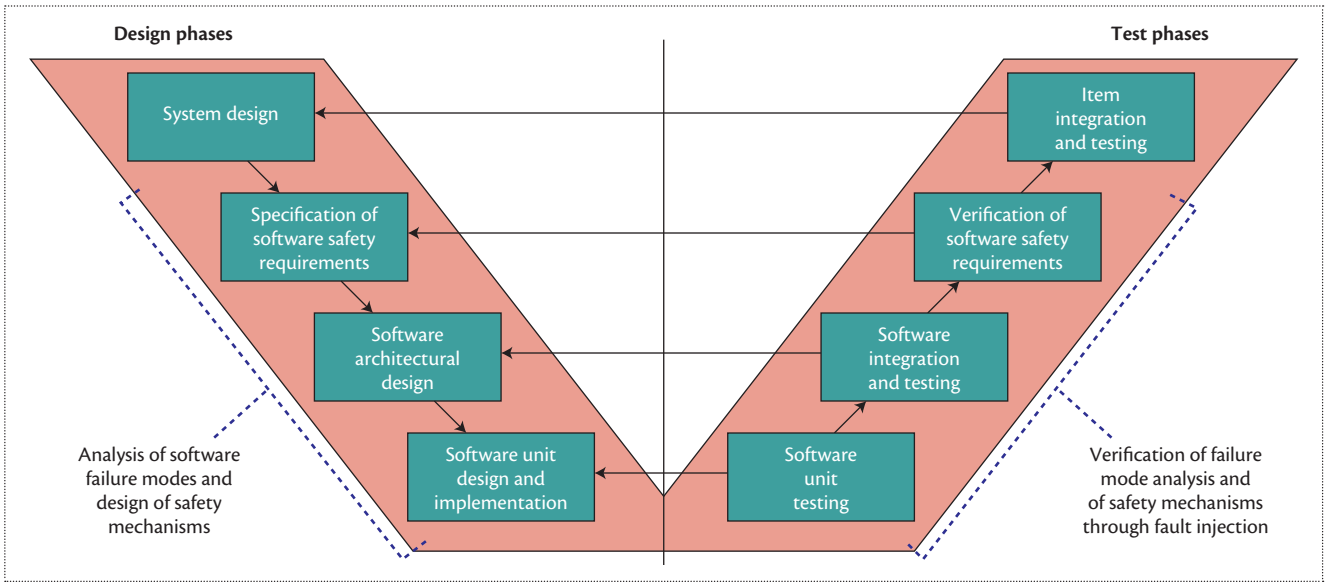

Figure 1. Software fault injection. This technique lets developers verify that safety mechanisms meet safety requirements.

realistic software faults—SAFE (*Soft*ware *F*ault *E*mulator)—that offer some guidance.

SFI and Safety Certification

Safety standards emphasize that software should be validated with respect to abnormal and faulty conditions, and many of them indicate fault injection as a suitable approach. ISO 26262, for example, explicitly mentions fault injection as a method for unit and integration testing of software items, 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. Even in safety standards that don't explicitly mention fault injection—such as DO-178B and DO-178C, which refer more generally to "robustness test cases"—it's a suitable approach for verifying software's robustness, for instance, by provoking "transitions that are not allowed by the software requirements."

Figure 1 shows how fault injection serves as a complementary activity for verifying robustness and fault tolerance during the development of safety-critical software. In particular, fault injection is appropriate, advisable, and useful for verifying systems that adopt measures (*safety mechanisms*) for detecting the effects of faults and achieving and maintaining a safe state. Such safety mechanisms are developed to fulfill *safety requirements* identified by failure modes and effects analysis (FMEA) activities at design time, which identify potential failures of components and subsystems and strategies to mitigate them; software-intensive systems require software FMEA (SW-FMEA) activities specifically focused on software components.[8] If an SW-FMEA points out that a given software component's failure might cause severe consequences, software designers must introduce a safety mechanism to let the system tolerate such failures. In this case, SFI can force a software component to fail during tests and provide evidence that the safety mechanism is effective. Multiple safety standards recommend a combination of rigorous verification and safety mechanisms to meet safety requirements.

Let's take a closer look at software safety mechanisms that can be verified through SFI. In general, a safety mechanism detects and mitigates at runtime faulty software components by introducing a fault-tolerance algorithm or mechanism (FTAM), that is, redundant hardware/software to compensate for faults.[6] A safety mechanism can detect the effects of faults by

- comparing the outputs of two or more redundant and diverse software components that perform equivalent functions and detecting an error if those outputs don't match by adopting some voting schema, as in the case of the *N*-version programming approach;[5] and
- performing plausibility checks on the values produced by a component or on the execution paths the component followed. Examples include assertions in the program that point out obvious inconsistencies between program variables or output values that are outside a range. Another example is a watchdog timer checking that the software is actually making progress.[7]

SFI supports the validation of these mechanisms. For instance, software testers can inject faults in a component to evaluate whether they can propagate to its

outputs and whether checks at its interface can detect them. Detection triggers the following recovery mechanisms for mitigating the effects of faults:

- masking the fault by performing multiple computations through multiple diverse replicas, either sequentially (recovery blocks) or concurrently (*N*-version programming),[5,6] although some faults can still be masked even if replicas aren't diverse; and
- bringing the system to a safe state, for instance, by switching to a degraded mode of service, giving priority to a subset of software functions, or forcing a fail-stop behavior.

SFI can provide evidence that recovery is effective against faulty behaviors and point out situations in which it isn't successful. Software testers can then assess, for example, whether the system is able to properly provide a degraded mode of service after a software failure.

Another potential application of SFI is in validating the SW-FMEA itself. SFI can reveal two kinds of FTAM failures:[2,6] faults in the implementation of FTAMs (lack of error and fault-handling coverage) and incorrect or incomplete assumptions about failure modes actually occurring in operation (lack of fault assumption coverage). FTAM failures of the second kind are due to the FTAM designer's assumptions about how software components can fail based on a potentially incorrect SW-FMEA.

In general, FMEAs can't be exhaustive: they can miss some failure modes or effects. An SW-FMEA is a difficult process that requires an expert analyst and a detailed knowledge of the system—and as with any human activity, it's prone to mistakes. Moreover, software functions are usually more complex than hardware ones, and the analyst can't typically obtain software failure modes from datasheets or field data.[8] After performing SFI, testers can look in detail at FTAM failures, identify those caused by incorrect assumptions, and revise the SW-FMEA and the system design accordingly. If software components fail according to the assumptions, software testers can trust the SW-FMEA's validity.

For assessing both safety mechanisms and the validity of SW-FMEAs, the representativeness of injected faults is important in supporting claims about the system's fault-tolerance properties. That is, fault injection should generate the same kinds of software errors that are likely to occur during operation. To evaluate an assertion's effectiveness, for example, the data checked by that assertion should be corrupted; if the injected corruption is arbitrary and not representative of reality, we can't know how the system will react to faults during operation.

From Hardware to Software Fault Injection

The use of SFI is relatively recent compared to traditional fault injection approaches. Early SFI tools corrupted software code or data by random bit flipping, which was a common practice to emulate the effects of hardware faults, with the aim of emulating the effects of software faults.[3,9] Subsequent approaches corrupted data at component interfaces 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 for pointing out corner cases in which invalid data isn't correctly handled, but they aren't suitable for emulating software faults in a representative way because the injected corruptions—such as bit flips and boundary values—don't necessarily match the effects of faults hidden in the system under evaluation. The realism of fault injection is an important condition for reproducing the faulty behaviors that are likely to occur in operation.

A more recent development is to inject realistic software faults by introducing artificial bugs in the target software. This technique produces a faulty version of a software component and generates erroneous behavior when it's executed. Empirical studies support the use of code changes for emulating software faults, finding that the injection of changed code produces erroneous behaviors similar to the effects of real software faults.[12] This method resembles mutation testing, but with a different goal: while mutation testing uses code changes to identify an adequate test suite, SFI is meant to validate FTAMs and analyze system behavior in realistic faulty scenarios. To encompass the many kinds of faults that can occur during development, mutation testing studies propose a large number of mutation operators, but not all mutation operators are necessarily representative of the residual software faults that escape testing, ship with the product, and affect the system during operation.

Several studies have focused on the definition of realistic fault models based on the rigorous analysis of failure data from both open source and commercial software systems.[4,13] These studies observed the same trend in the distribution of faults: "algorithm" faults dominate; "assignment" and "checking" faults have approximately the same weight; and "interface" and "function" faults are less frequent. The data encompasses both operating system code and user programs, with varying degrees of maturity and numbers of users.

The researchers in one study further classified software faults in terms of programming language constructs that were missing, wrong, or extraneous in the faulty code and found that most faults belonged to a few common fault types (see Table 1).[13] This set of fault types forms a fault model that's reasonably program

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

	Fault type	Number 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 of 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

independent and suitable for automated fault injection because it describes how to manipulate a program to introduce faults in terms of programming constructs to be removed or modified. The model also provides several detailed rules (not shown here, for brevity) that describe 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 five statements because it's unlikely that an "`if`" construct is lacking for larger groups of statements.

A limitation is that some field faults aren't covered because 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's very difficult to obtain such data because it requires putting the system in operation for several years. Thus, this model focuses on fault types that are generic enough that they can be used even if field failure data isn't available. Considering that code changes can generate errors in a way similar to real faults,[12] the use of representative fault types can achieve a good degree of realism even if the fault types don't account for every possible fault.

In our previous study, we evaluated the suitability of this model for safety-critical software.[14] Given the more rigorous testing activities such software undergoes, a different distribution of fault types could hold. We analyzed how testing affects the types of residual software faults in a real-time operating system (RTOS) used in space applications, comparing the distribution of injected faults with the distribution of faults obtained after removing those faults detected by test suites. As expected, test suites were very effective in safety-critical software, with only a minority of injected faults escaping. A key finding was that test suites didn't affect the distribution of fault types—that is, the relative proportions of fault types before and after testing were the same. Instead, testing affected the distribution of faults across code locations. Therefore, to exploit these fault types in safety-critical software, we need to fine-tune the code location in which faults are injected to achieve fault representativeness.

SAFE

The SAFE tool supports the automated analysis of FTAMs and failure modes through SFI and was originally developed in the context of academic research. The tool can inject software faults into C and C++ software according to the fault model described earlier. Differing from previous SFI tools that inject bit flips or invalid values,[3,7,10,11] SAFE emulates software faults by

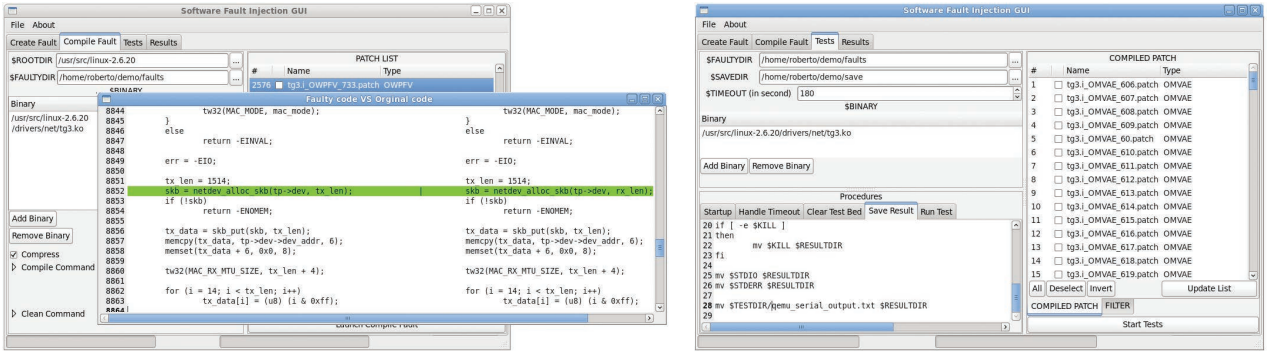

Figure 2. The SAFE tool supports the automated execution of large sets of fault injection tests. (a) Selection and preview of faults to inject and (b) setup of fault injection experiments.

adopting a representative fault model, which is required to provide sound evidence that a system will be fault tolerant during operation.

The fault injection approach closest to ours (which injects Table 1's fault types by mutating the target software's source code) is represented by the G-SWFIT technique, which mutates the target's binary code, while our approach mutates the target's source code.[13] In our previous work with the European project Critical-Step (www.critical-step.eu), we compared an industrial implementation of G-SWFIT to our technique.[15] Ultimately, we found that binary-level injection works in the absence of source code, but the mutation of binary code is often inaccurate and difficult to implement correctly. We've improved the SAFE tool to the point where it's now mature enough to handle very large fault injection efforts in complex real-world software.

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

- *Code analysis.* The tool analyzes the target software's code to identify where to inject the faults. It first transforms the code into an internal representation (an abstract syntax tree), which is then analyzed to determine the constructs in the program that fulfill the fault model's rules and are suitable for injecting realistic faults.
- *Fault generation.* Each fault identified in the previous phase is injected to obtain a faulty version of the target software (see Figure 2a). During this phase, users can select a subset of faults to inject by configuring filtering criteria. Possible criteria include injecting only a subset of fault types in a subset of code locations, for instance, only in the parts of the software deemed most defect prone according to software complexity metrics to achieve fault representativeness.[14]
- *Test execution.* A test is executed for each faulty version of the software generated during the previous phase. The tool cleans residual errors from previous experiments by stopping the system, starting the target system with a new fault, executing a workload, shutting down the target system after a fixed time, and collecting failure data. These operations are system-specific, so the tool lets users customize them with a scripting language (see Figure 2b). Collecting postmortem data avoids introducing excessive overhead, 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 the system's execution traces.
- *Result analysis.* SAFE analyzes experimental data to provide the users with information about failure modes and FTAMs. The tool eases data analysis through user-defined scripts that evaluate specific system properties. For instance, a 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 detected the fault.

SFI costs, given the size of current software systems, are a primary concern for software testers. Three factors affect the cost of a fault injection campaign: the time to set up a testbed, the time to run experiments, and the time to analyze the experimental data.

Setting up a fault injection testbed requires some manual effort. Because tools support the actual fault injection, most of the setup effort consists of integrating tools into the system under evaluation. With SAFE, the setup consists of developing a set of scripts for performing a specific operation on the target system. For instance, a script is used for starting the target system's

execution by starting an emulator or sending 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 bulk of a fault injection campaign. Test duration is determined by the number of faults to inject, which in turn depends on the system's size—typically, it ranges from hundreds to hundreds of thousands of faults. Researchers have proposed various approaches for speeding up test execution, which can take from hours to days, by selecting a subset of faults to inject to reduce the number of experiments and achieve confidence in the results. We developed a heuristic that improves the representativeness of injected faults and reduces the number of experiments by filtering out up to 70 percent of faults.[14] When an evaluation is focused on a specific assertion, the tester should perform a further fault-filtering step to focus SFI on code related to the part under evaluation. For instance, to assess a specific procedure, faults should be injected in those procedures interacting with it. The SAFE tool lets users customize which faults are injected.

Software testers can use experimental data to quantitatively analyze FTAMs in terms of coverage factors and timing distributions, and analyze root causes behind FTAM failures to provide feedback for future FTAM design and implementation. The former consists of summarizing, in statistical terms, the outcomes of experiments according to user-defined predicates—a concise specification of properties that must hold in the presence of faults—derived from safety requirements defined in the design phases. For instance, a railway signaling system should never allow two trains to cross in the same section. SAFE can automate the analysis of predicates over experimental data via user-defined scripts. Quantitative results are also useful for supporting the system's stochastic modeling and evaluation.[2] In the latter, developers look in depth at a subset of fault injection experiments that caused FTAM failures—a similar approach to debugging a program by looking at failed test cases.

Case Study

A pilot R&D project that we did with the Finmeccanica industrial group demonstrates SAFE in action. The project's goal was to develop a Linux-based RTOS (FIN.X-RTOS) for use in avionic applications; this RTOS also had to be tested in compliance with the DO-178B safety standard to ensure certification of future systems based on it. FIN.X-RTOS is a Linux kernel enhanced with better scalability for multicore architectures, support for hard real-time software, and stripped of unessential parts. At the time of this writing, FIN.X-RTOS developers have fulfilled the level D standard's requirements and are working on additional verification activities for the more stringent requirements of level C, which requires testing the software's robustness against abnormal inputs and conditions.

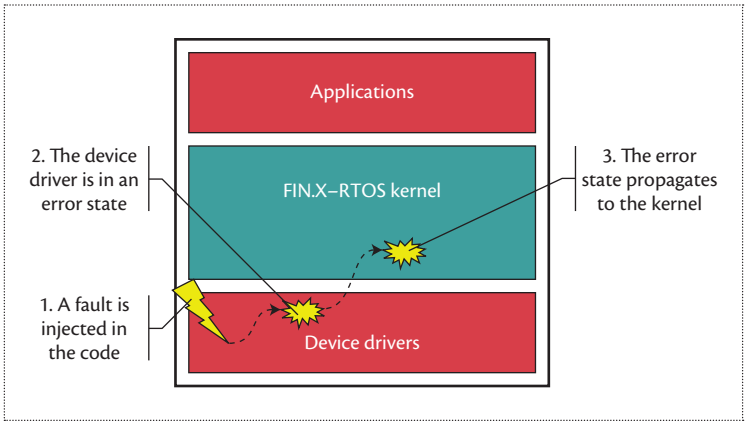

Figure 3. Software fault injection enables us to assess the ability of FIN.X-RTOS to mitigate the effects of faulty device drivers.

In particular, we focused on a robustness evaluation for the FIN.X-RTOS kernel against faulty conditions caused by device drivers (see Figure 3). Device drivers aren't part of the FIN.X-RTOS kernel: they often must be developed ad hoc or obtained from a third-party, hardware-specific board support package (BSP) after FIN.X-RTOS is integrated into a wider system. Unfortunately, kernel code tends to be vulnerable to faulty device drivers because developers often omit checks on device driver behavior to improve performance. This problem is exacerbated by the high defect rate in device drivers and by FIN.X-RTOS's monolithic architecture (inherited from the Linux kernel), in which device drivers execute in privileged mode and can affect the entire operating system.[11]

We used SFI to assess the kernel's ability to stop error propagation from device drivers to the kernel itself by injecting faults directly into the device drivers. The kernel is robust when the effects of faults are restricted to the faulty device driver; such faults can be mitigated by reinitializing the device driver or by switching to a secondary device. When the kernel can't detect or prevent error propagation, a faulty driver can affect shared kernel data or code, so it should be hardened by applying additional safety mechanisms, such as checks on function parameters.

We conducted our experiments with the SAFE tool in an emulated environment and 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 used 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 then analyzed these messages to identify whether a fault propagated to the kernel or to the workload.

In most cases (79.1 percent), the workload correctly executed even in the presence of a faulty driver: in general, this phenomenon is often observed in fault injection experiments because the fault might be accidentally masked (for example, an uninitialized or corrupted variable is overwritten later in the program) or remain latent during the experiment. In the remaining cases, faults affected either the kernel (11.3 percent) or the workload (9.6 percent). In the case of workload errors and driver crashes not propagated to the kernel, the injected fault caused network device driver unavailability, thus affecting communication between the server and the clients. These faults were successfully detected by the kernel and signaled to user applications.

> **SFI is a means to assess, before releasing the product, the impact of software faults in worst-case scenarios.**

When the fault propagated to the kernel in our experiments, it caused a kernel thread to stall or terminate, thereby affecting the entire OS and causing its failure. In general, if an exception occurs while executing OS code, the OS tries to recover from the exception by killing the current task. For instance, when a task invokes an OS system call, and the system call causes an exception or doesn't 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's a kernel thread, thus affecting OS execution.

Let's consider the case of a "missing variable initialization" fault injected in the ne2k-pci driver that killed the sirq-timer kernel thread. This kernel thread is responsible for the delayed execution of kernel functions associated with a timer, including a timer function mld_ifc_timer_expire that periodically sends network messages for discovering multicast listeners. In this experiment, this function invoked the faulty device driver, which caused an exception because it accesses an uninitialized data structure. When the sirq-timer kernel thread was killed, timer functions in the kernel couldn't be executed anymore. To avoid this situation, the kernel's exception handler should be modified 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 function fails due to a faulty driver. The use of SFI for emulating faulty device drivers can help uncover robustness vulnerabilities like this, which otherwise aren't pointed out by more traditional forms of testing.

Residual faults are hidden in our software and will eventually manifest themselves during operation. This threat will likely get worse as the complexity of software steadily rises. SFI is a means to assess, before releasing the product, the impact of software faults in worst-case scenarios—it enables the evaluation and improvement of fault tolerance. It represents a reasonably mature technology for the assessment of safety-critical software because it can realistically emulate residual software faults and can be fully automated, which makes it a feasible and cost-effective approach. ■

Acknowledgments

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

References

1. N. Leveson, "Role of Software in Spacecraft Accidents," *J. Spacecraft and Rockets*, vol. 41, no. 4, 2004, pp. 564–575.
2. J. Arlat et al., "Fault Injection for Dependability Validation: A Methodology and Some Applications," *IEEE Trans. Software Eng.*, vol. 16, no. 2, 1990, pp. 166–182.
3. J.M. Voas and G. McGraw, *Software Fault Injection: Inoculating Programs against Errors*, John Wiley & Sons, 1998.
4. J. Christmansson and R. Chillarege, "Generation of an Error Set that Emulates Software Faults based on Field Data," *Proc. IEEE Int'l Symp. Fault-Tolerant Computing*, IEEE, 1996, pp. 304–313.
5. J.-C. Laprie et al., "Definition and Analysis of Hardware- and Software-Fault-Tolerant Architectures," *Computer*, vol. 23, no. 7, 1990, pp. 39–51.
6. J.-C. Avizienis et al., "Basic Concepts and Taxonomy of Dependable and Secure Computing," *IEEE Trans. Dependable and Secure Computing*, vol. 1, no. 1, 2004, pp. 11–33.
7. M. Hiller, A. Jhumka, and N. Suri, "EPIC: Profiling the Propagation and Effect of Data Errors in Software," *IEEE Trans. Computers*, vol. 53, no. 5, 2004, pp. 512–530.
8. P. Goddard, "Software FMEA Techniques," *Proc. Annual*

Reliability and Maintainability Symp., IEEE, 2000, pp. 118–123.
9. M. Hsueh, T. Tsai, and R. Iyer, "Fault Injection Techniques and Tools," *Computer*, vol. 30, no. 4, 1997, pp. 75–82.
10. P. Koopman and J. DeVale, "The Exception Handling Effectiveness of POSIX Operating Systems," *IEEE Trans. Software Eng.*, vol. 26, no. 9, 2000, pp. 837–848.
11. J. Albinet, J. Arlat, and J.-C. Fabre, "Characterization of the Impact of Faulty Drivers on the Robustness of the Linux Kernel," *Proc. IEEE/IFIP Int'l Conf. Dependable Systems and Networks*, IEEE, 2004, pp. 867–876.
12. M. Daran and P. Thevenod-Fosse, "Software Error Analysis: A Real Case Study Involving Real Faults and Mutations," *ACM Soft. Eng. Notes*, vol. 21, no. 3, 1996, pp. 158–171.
13. J. Duraes and H. Madeira, "Emulation of Software Faults: A Field Data Study and a Practical Approach," *IEEE Trans. Software Eng.*, vol. 32, no. 11, 2006, pp. 849–867.
14. R. Natella et al., "On Fault Representativeness of Software Fault Injection," *IEEE Trans. Software Eng.*, vol. 39, no. 1, 2013, pp. 80–96.
15. D. Cotroneo et al., "Experimental Analysis of Binary-Level Software Fault Injection in Complex Software," *Proc. IEEE European Dependable Computing Conf.*, IEEE, 2012, pp. 162–172.

Domenico Cotroneo is an associate professor at Università degli Studi di Napoli Federico II. His main interests include dependability assessment techniques, software fault injection, and field-based measurement techniques. Cotroneo received his PhD in computer science and system engineering from Università degli Studi di Napoli Federico II. He's a member of IEEE. Contact him at cotroneo@unina.it.

Roberto Natella is a postdoctoral researcher at Università degli Studi di Napoli Federico II, and cofounder of Critiware S.r.L. His research is in the area of dependability assessment of mission-critical systems, in particular, with software fault injection and software aging and rejuvenation. Natella received a PhD in computer engineering from Università degli Studi di Napoli Federico II. He's a member of IEEE. Contact him at roberto.natella@critiware.com.