Assessing Dependability with Software Fault Injection: A Survey

ROBERTO NATELLA and DOMENICO COTRONEO, DIETI, Federico II University of Naples
HENRIQUE S. MADEIRA, CISUC, University of Coimbra

With the rise of software complexity, software-related accidents represent a significant threat for computer-based systems. Software Fault Injection is a method to anticipate worst-case scenarios caused by faulty software through the deliberate injection of software faults. This survey provides a comprehensive overview of the state of the art on Software Fault Injection to support researchers and practitioners in the selection of the approach that best fits their dependability assessment goals, and it discusses how these approaches have evolved to achieve fault representativeness, efficiency, and usability. The survey includes a description of relevant applications of Software Fault Injection in the context of fault-tolerant systems.

Categories and Subject Descriptors: C.4 [**Performance of Systems**]: Fault Tolerance; D.2.5 [**Testing and Debugging**]: Error Handling and Recovery

General Terms: Reliability, Verification, Performance

Additional Key Words and Phrases: Software faults, dependability assessment, software fault tolerance

ACM Reference Format:
Roberto Natella, Domenico Cotroneo, and Henrique S. Madeira. 2016. Assessing dependability with software fault injection: A survey. ACM Comput. Surv. 48, 3, Article 44 (February 2016), 55 pages.
DOI: http://dx.doi.org/10.1145/2841425

1. INTRODUCTION

Software is almost everywhere in our everyday life: from embedded systems to large critical infrastructures, such as telecommunication, health, transportation, financial, and power supply systems. Software is also a major concern for the reliability of computer-based systems. In fact, *software faults* (also called *software defects* or *bugs*) have been consistently recognized as one of the most frequent sources of computer outages, as shown by several research studies [Gray 1990; Lee and Iyer 1995; Oppenheimer et al. 2003]. In many cases, software faults were the root cause of incidents that had significant economic costs and even caused the loss of human lives [Leveson 2004; McQuaid 2012].

The issue of software faults is worsened by several factors. As systems become more complex, the number and the complexity of software functions tend to grow, making software verification more difficult, thus increasing the likelihood of residual software defects. The elimination of software defects is also made difficult by time-to-market constraints and by the adoption of reused, legacy, and third-party software components, such as Commercial Off-the-Shelf (COTS) software, even in mission-critical systems.

This work was supported by the CECRIS FP7 project (grant no. 324334) and by the COSMIC public-private laboratory (grant no. PON02 00669) funded by the Italian Ministry of Education, University and Research.
Authors' addresses: R. Natella and D. Cotroneo, DIETI, Federico II University of Naples, Via Claudio 21, 80125 Naples, Italy; email: {roberto.natella, cotroneo}@unina.it; H. S. Madeira, CISUC, University of Coimbra, Polo II-Pinhal de Marrocos, 3030-290 Coimbra, Portugal; email: henrique@dei.uc.pt.

DOI: http://dx.doi.org/10.1145/2841425

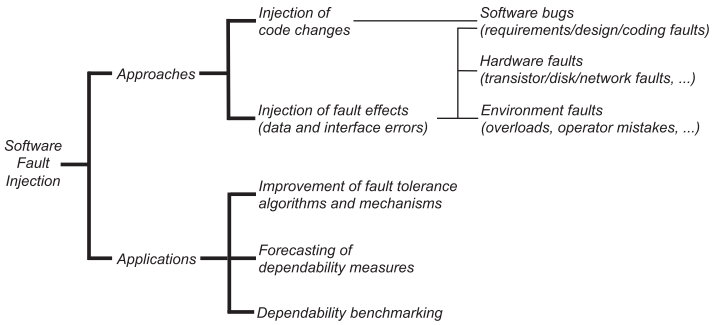


Fig. 1. Scope of the survey: approaches and applications for Software Fault Injection.

Even though COTS components avoid the costs of new software, their use outside their original context is prone to faults, and they are difficult to test and debug due to the lack of source code and/or expertise on their internals [Weyuker 1998]. As a result, the presence of residual defects in the software cannot be avoided in practice.

Therefore, it becomes important, and even recommended by several industry standards [NASA 2004; ISO 2011; Microsoft Corp. 2014] and research initiatives [Patterson et al. 2002; AMBER project 2009], to ensure that systems are robust to unforeseen faulty conditions in software components, in order to be confident that software faults cannot cause serious service failures. Fault injection is a practical approach for achieving this confidence, by deliberately introducing faults in a system. This approach can assess fault tolerance properties, such as robustness, in the presence of faults. It is thus useful to test the effectiveness of fault tolerance algorithms and mechanisms, which are adopted to mask faults or to switch to a degraded mode of service, such as assertions and exception handlers [Laprie et al. 1990; Hiller et al. 2001; Fetzer et al. 2004; Candea et al. 2004].

The injection of hardware faults (such as memory and CPU faults) has been studied for a long time, and several techniques and tools have been developed for this purpose, ranging from the physical injection of faults (e.g., using radiation and power supply disturbances) to the emulation of hardware faults through software, which are surveyed in Clark and Pradhan [1995], Hsueh et al. [1997], and Carreira et al. [1999]. However, given the increase of software-related accidents, the focus of researchers has been shifting from faulty hardware toward faulty software.

The focus of this survey is on the *injection of software faults* (known as *Software Fault Injection*, SFI). We present a comprehensive analysis of the several techniques and tools that have been proposed in the last decades to guide researchers and practitioners toward the most suitable approach for their goals (e.g., robustness testing, dependability benchmarking). The survey discusses SFI with respect to its key problems: the representativeness of injected faults, the efficiency of experiments, and the usability of techniques and tools, which represent essential concepts to better understand and map the different SFI techniques. The discussion is complemented with examples of relevant applications of Software Fault Injection in the evaluation and the improvement of fault-tolerant systems. The survey ends with a discussion of directions for future research on SFI, considering emergent research trends and recent technological developments that have worsened classic open issues on SFI and created new ones.

Scope and Structure of the Survey. The assessment of dependability through fault injection is a broad field. Recent research on SFI has been focused on the emulation of software faults (Figure 1), that is, bugs that plague software components and that originate during software development. SFI approaches mimic software faults through

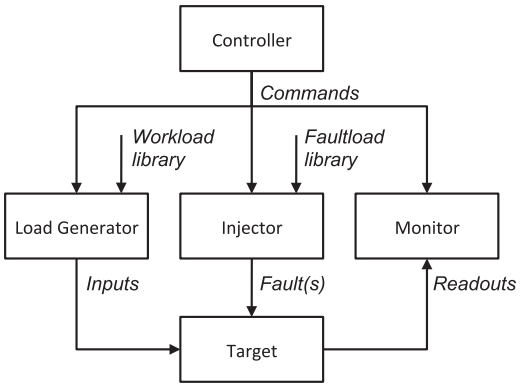


Fig. 2. Conceptual schema of fault injection [Hsueh et al. 1997].

code changes injected within the component's code. Other SFI studies have been focused on the injection of software errors, that is, the *effects* on software induced by faults, which may originate from software, hardware, or the environment (e.g., users and other systems interacting with the computer system). Software errors are injected by corrupting components' data, either internally or at its interfaces. Overall, these SFI approaches experimentally evaluate how a faulty software component (i.e., the component where faults are injected) affects the system that includes the component. As discussed in this survey, this methodology can be applied for the testing of fault tolerance algorithms and mechanisms, for the forecasting of dependability measures, and for dependability benchmarking.

This survey first provides in Section 2 general concepts about fault injection. Relevant applications are described in Section 3 to highlight how SFI can support the assessment of fault-tolerant systems. Then, in Section 4, the survey analyzes SFI techniques and tools and how they evolved to achieve fault representativeness, efficiency, and usability. These approaches and applications are compared in Section 8. In Section 9, we discuss the overlaps between Software Fault Injection and Mutation Testing, which is another approach that involves the injection of software faults. In Section 10, we discuss possible future work in the area of Software Fault Injection.

2. BASIC CONCEPTS AND DEFINITIONS

According to Avizienis et al. [2004], a *fault* is the adjudged or hypothesized cause of an incorrect system state, which is referred to as *error*. A *failure* is an event that occurs when an incorrect service is delivered; that is, an error state is perceived by users or external systems. Of course, the definition of fault has great relevance for fault injection. A *fault model* describes the faults that the system is expected to experience during operation, which depend on the nature of the system and on its operational environment, development process, and technologies. As discussed in Christmansson and Chillarege [1996], Madeira et al. [2000], and Johansson and Suri [2005], the fault model requires the definition of three important aspects, namely, *what* to inject (i.e., which kind of fault), *when* to inject (i.e., the timing of the injection), and *where* to inject (i.e., the part of the system targeted by the injection).

Even if there are several fault injection approaches and fault models, it is possible to recognize a common conceptual schema of fault injection [Hsueh et al. 1997], shown in Figure 2. The system under analysis is usually referred to as *target*. There are two entities that stimulate the system, the *load generator* and the *injector*. The first one exercises the target with inputs that will be processed during a fault injection

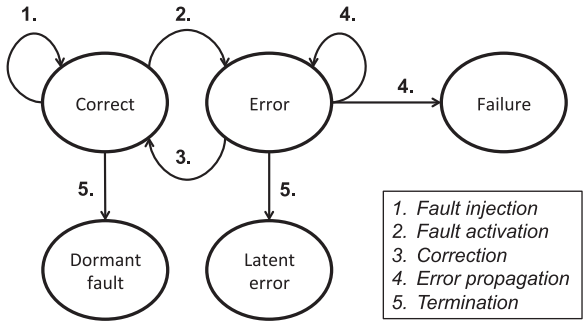


Fig. 3. States of a fault injection experiment.

experiment, whereas the other introduces a fault in the system. The inputs and faults submitted to the system are respectively referred to as *workload* and *faultload*, which are specified by the tester through a *library* by enumerating inputs and faults, or by specifying the rules for generating them (e.g., using probability distributions). A fault is injected by tampering with the structure or the state of the system or with the environment in which it executes. Fault injection involves the execution of several *experiments* (or *runs*), which form a *fault injection campaign*. Typically, only one fault from the faultload is injected during each experiment; some recent studies also investigated the injection of multiple faults during the same experiment.

The *monitor* entity collects raw data (*readouts* or *measurements*) from the target. The choice of readouts depends on the kind of system and on the properties that have to be evaluated. They may include the outputs of the target (e.g., messages sent to users or to other systems) and of specific components of the target system (e.g., error detection mechanisms), the internal state of the target (e.g., the contents of a critical variable), and information about covered code. Readouts are used to compute relevant fault tolerance measures: for example, the tester can assess whether the injected fault has been tolerated or the system has failed. In order to assess the outcome of an experiment, readouts are usually compared to the ones obtained from fault-free experiments (referred to as *golden runs* or *fault-free runs*). All the entities in Figure 2 are orchestrated by the *controller*, which is also responsible for iterating experiments of a fault injection campaign, as well as for storing the results for later analysis.

A fault injection experiment is characterized by different states, shown in Figure 3. Initially, the system is assumed to work in the "correct" state. As soon as a fault is injected and a workload is applied, two behaviors can be observed. First, the fault is not activated and it remains *dormant*. In this case, the experiment does not produce a failure. Second, the fault is activated and it causes an error. At this stage, an error (1) may *propagate*, by corrupting other parts of the system state until the system exhibits a failure; (2) can remain *latent* in the system; or (3) can be *masked* or *corrected* by the presence of *protective* or *unintentional* redundancy (e.g., erroneous data are overwritten by correct data or reinitialized) [Avizienis et al. 2004].

In many experiments, only a fraction of faults are activated and produce errors. In the other cases, fault tolerance cannot be evaluated since no error occurs. In order to accelerate the occurrence of errors, a particular form of fault injection, namely, *error injection*, is often adopted. Here, the *effects* of faults are introduced in place of the actual faults. Error injection can be cheaper and more feasible to perform, and it can avoid the problem of inactive faults. A well-known error injection approach is *Software-Implemented Fault Injection* (SWIFI), in which hardware faults (e.g., a gate-level stuck-at) are emulated by injecting the effect of a fault on the software (e.g.,

by corrupting a memory location or register that would be affected by a faulty gate), instead of physically tampering with the hardware.

A very important aspect of fault injection is the set of *measures* adopted to characterize the target system in the presence of faults. The goal of these measures is to represent the *ability to tolerate faults* of one or more *fault tolerance algorithms and mechanisms* (*FTAMs*) in the system. Given an FTAM and a set of faults F, the effectiveness of the FTAM is represented by a *coverage factor* [Arlat et al. 1990; Powell et al. 1995], which is defined as

$$c = \Pr\{H = 1 \mid F \times A\}, \tag{1}$$

that is, the *conditional probability* that a fault is correctly handled ($H = 1$), given the occurrence of any fault from F and of any sequence of inputs (i.e., a workload) from A. In other terms, the coverage factor c can be viewed as $E\{H|G\}$, the expected value of the discrete random variable H for the population of fault/inputs pairs in $G = F \times A$. This definition emphasizes that fault tolerance depends on the faults and inputs that are faced during operation. Moreover, different fault/inputs pairs $(f, a) \in G$ may have a different *relative probability* of occurrence (i.e., the probability that, given that a fault has occurred, the fault/inputs pair is actually (f, a)), and the coverage factor reflects these relative probabilities (denoted as $p(f, a)$). A coverage factor can therefore be expressed as a weighted sum,

$$c = \sum_{(f,a)\in G} h(f, a) \cdot p(f, a), \tag{2}$$

where $h(f, a) = 1$ if (f, a) is correctly handled, and $h(f, a) = 0$ otherwise. Another measure of fault tolerance effectiveness is the *mean coverage time* (also referred to as *latency*) [Arlat et al. 1990], that is, the expected value τ of a random variable T representing the time required to handle a fault. Fault injection estimates the probability density function f_T of this random variable.

3. APPLICATIONS OF SOFTWARE FAULT INJECTION

This section reviews a selection of past studies that adopted SFI to highlight its applications and usage scenarios in the assessment of fault-tolerant software systems.

3.1. Fault Forecasting

Fault forecasting through fault injection [Arlat et al. 1990] is aimed at quantitatively evaluating, usually in probabilistic terms, the fault tolerance properties of a system. Examples are the coverage factors and latency of FTAMs (i.e., the probability and the delay of correct fault handling), which can be used in analytical models to obtain dependability measures (e.g., availability, performability). Other measures of interest for fault forecasting are the probability of specific failure modes, such as the probability of a *fail-stop* behavior, and the probability of catastrophic failures from the point of view of *system safety* [RTCA 1992; ISO 2011].

An early study on software fault tolerance [Hudak et al. 1993] compared several techniques by injecting both hardware and software faults. Different teams implemented a program following the same specifications; each team adopted a different fault tolerance technique: N-version programming, recovery blocks, concurrent error detection, and algorithmic fault tolerance. The different implementations were executed using several test vectors and injecting a fault at each execution. The following coverage factors were evaluated:

—**Detection**: Pr\{detecting an error | a fault is active\},
—**Recovery**: Pr\{recovering successfully | an error is detected\},
—**Aborting**: Pr\{aborting | successful recovery is not possible\}.

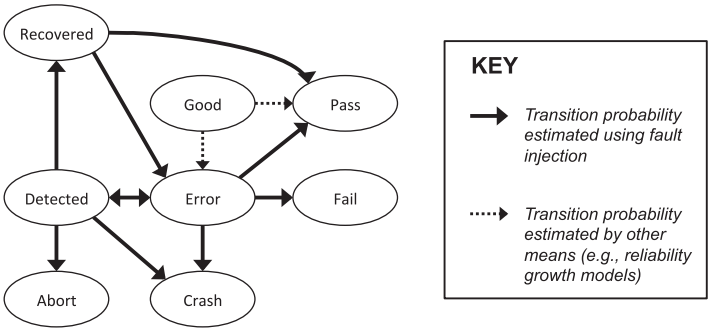


Fig. 4. Model for comparing software fault tolerance techniques [Hudak et al. 1993].

In turn, these probabilities were used as parameters of a discrete Markov model (Figure 4) to derive other figures of merit, such as the mean probability of correct execution and the mean probability that the software will behave in a fail-stop manner. These measures were used to compare fault tolerance techniques in terms of reliability and cost.

In Kao et al. [1993], both hardware and software faults were injected to analyze the propagation of faults in SunOS and their impact on performance and availability. This work defined a Markov model, whose transition probabilities and mean state holding times were populated with experimental measurements. They assigned a *reward* to each state, that is, a numeric evaluation of the performance level provided by the system in that state. They then performed a transient Markov reward analysis to estimate the average performance level in the presence of faults. Bondavalli et al. [2004] reported a *performability* analysis (i.e., evaluation of joint metrics accounting for both performance and availability) of a fault-tolerant architecture based on legacy and COTS software. The architecture was modeled using Stochastic Activity Networks (SANs), whose parameters are populated through fault injection experiments. They performed model-based simulations to assess the effectiveness of fault tolerance (in terms of long-term performability of the system) and for tuning system parameters (e.g., thresholds for error detection mechanisms).

Ng and Chen [2001] presented an example of how to use SFI to provide useful feedback to the development process of fault-tolerant systems. They designed a write-back file cache (i.e., data are flushed to the disk asynchronously) with the requirement to be as reliable as a write-through file cache (i.e., data are synchronously written on the disk) in spite of OS crashes. They adopted an iterative design process to achieve this requirement. At each iteration, software faults are injected in the OS in order to cause OS crashes and to measure the reliability of the file cache in terms of *corruption rate* (i.e., percentage of data corruptions caused by the OS crash). If the reliability of the write-back file cache is still lower than the write-through file cache, then the design of the file cache is revised in order to tolerate more crashes. Table I shows the number of injections and of data corruptions at each iteration: it is interesting to observe how the feedback provided by fault injection allowed us to incrementally improve the reliability of the file cache, by reducing the corruption rate from 39.3% to 1.9%. Another example of reliability measure is the percentage of *fail-stop* violations, where the system does not immediately halt, but it has an erratic behavior. The absence of fail-stop violations is an underlying assumption of several fault-tolerant techniques [Strom and Yemini 1985; Schiper et al. 1991]. Chandra and Chen [1998] evaluated the validity of this property for a DBMS using Software Fault Injection: they observed that the percentage of

Table I. Corruption Rate of File Cache Designs [Ng and Chen 2001]

		Write-Back File Caches			
Fault Type	**Write-Through File Cache**	*Default FreeBSD Sync*	*Basic Safe Sync*	*Enhanced Safe Sync*	*BIOS Safe Sync*
Bit-flips text area	3	51	7	5	2
Bit-flips heap area	0	3	3	2	0
Bit-flips stack area	5	28	8	3	1
Initialization	10	45	9	7	4
Missing rand. instr.	4	43	8	2	4
Incorrect dest. reg.	4	42	9	5	2
Incorrect source reg.	4	43	10	3	1
Incorrect branch	4	51	14	4	5
Corrupt pointer	3	38	5	4	2
Alloc. mgmt.	0	100	5	0	0
Copy overrun	4	36	1	3	2
Synchronization	0	3	1	0	0
Off-by-one	4	59	16	9	3
Memory leak	0	0	0	0	0
Interface	1	47	8	3	2
Corruption rate (95% conf.)	46/1500 ($3.1\% \pm 0.8\%$)	589/1500 ($39.3\% \pm 2.5\%$)	104/1500 ($6.9\% \pm 1.3\%$)	50/1500 ($3.3\% \pm 1.0\%$)	28/1500 ($1.9\% \pm 0.7\%$)

fail-stop violations (7%) can be reduced by a factor of 3 using a transaction mechanism, which undoes recent changes made right before a DBMS crash.

Quantitative evaluations of fault tolerance, such as the ones described earlier, require a representative fault model: if the injected faults do not reflect the real faults, then the estimated coverage factors can be misleading. Ng et al. [1996] evaluated the sensitivity of the (estimated) corruption rate of a file cache by taking into account the likelihood of injected faults (i.e., the relative probability of occurrence of each fault). These probabilities are represented by the weights $p(f, a)$ in Equation (2). They investigated the use of equal weights and of fault probabilities that were obtained from field failure data studies on the MVS [Sullivan and Chillarege 1991] and the Tandem NonStop-UX [Thakur et al. 1995] OSs. The study pointed out how different fault probabilities can impact on estimates of the corruption rate.

Another useful application of fault injection is to gain insights into the *failure modes* that a system can exhibit. Indeed, in most cases, it is difficult for developers to anticipate the behavior of their software in all possible worst cases. This issue is worsened by the use of COTS software, since the system integrator does not have full visibility or control on the development process of the component. Components' failure modes can be broadly grouped into [Arlat et al. 2002; Walter and Suri 2003]:

—*crash failures*: the component stops or is not responsive;
—*detected failures*: reported to the user of the component (e.g., using an exception); and
—*undetected failures*: an incorrect state that is neither detected nor confined to the component, and that propagates to other components.

Fault injection into a component allows one to (1) evaluate the extent of the most severe failure modes, (2) discover failure modes not yet known by developers, and (3) evaluate the effectiveness of FTAMs surrounding the component. For instance, MAFALDA [Arlat et al. 2002] was developed to analyze error propagation through COTS microkernel components, such as task synchronization, interprocess communication, and memory management (Figure 5), and to validate the coverage of *component wrappers*

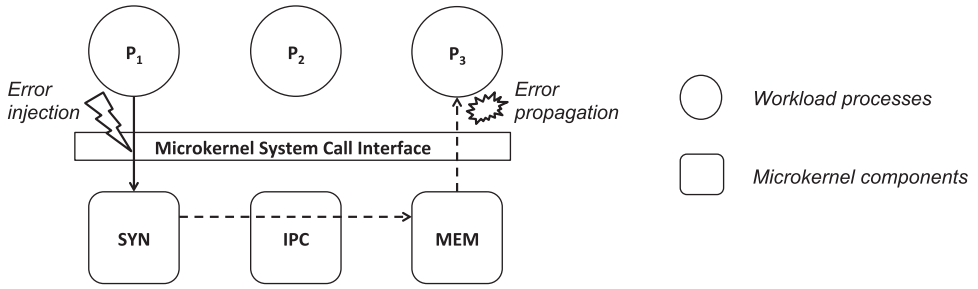


Fig. 5. Analysis of error propagation using MAFALDA [Arlat et al. 2002].

[Salles et al. 1999] (i.e., consistency checks on inputs and outputs of microkernel APIs) in preventing error propagation.

3.2. Fault Removal from Fault Tolerance Algorithms and Mechanisms

In the case of fault removal from fault tolerance [Avresky et al. 1996], fault injection is adopted to check the adequacy of fault tolerance with respect to its specifications. In this case, the results of fault injection are qualitative, and the readouts collected during the experiment are analyzed to correct specific issues of fault handling (e.g., how a specific error propagated in the system, and how a failure can be prevented). FTAMs include assertions and redundant logic that detect an erroneous state, and exception handlers and checkpoint/rollback mechanisms that correct the program state or switch to a degraded mode of service [Avizienis et al. 2004].

The *Extended Propagation Analysis* (*EPA*) technique [Voas et al. 1997] is a relevant example of fault removal from fault tolerance. EPA uses error injection in program data to assess the possibility of unsafe outcomes and to point out where the software can be extended for preventing error propagation. The EPA injects errors in the data accessed by a statement in order to evaluate the sensitivity of the software to faults in that statement. The most sensitive statements are candidates for an assertion that prevents the propagation of wrong data.

To illustrate the EPA, we discuss a case study on control software adopted for human neurosurgery [Voas et al. 1997]. The software adjusts the current level in a coil in order to move a seed in a specific location in the brain. The control software communicates with the device by reading and setting coil parameters, and must ensure that the current level falls within a specified range in order to prevent patient injury. By using error injection, EPA computes a *failure tolerance* score for each source code statement (i.e., percentage of error injection experiments leading to an acceptable outcome). The score of a function or file is defined as the lowest score among its statements. Table II shows the failure tolerance score of a subset of locations in the neurosurgery software. The low failure tolerance score ($\approx$0.48) is due to a vulnerable location at line 90. After inspecting that code, developers found that the variable representing the current value is checked by an assertion before line 90, but the variable is further processed and then sent to an actuator without any assertion to check its validity. Fault tolerance was improved by moving the assertion after the processing of the variable.

It is worth noting that the results from error injection (instead of injecting the actual faults) do not necessarily translate into probabilistic reliability measures, since the representativeness of injected errors is difficult to assert (as discussed later). However, error injection is useful to force “corner cases” in the software that are not easily produced by injecting the actual faults, and that can identify vulnerabilities.

Table II. Analysis of Error Sensitivity in Safety-Critical Software [Voas et al. 1997]

Source Code Statements	Failure Tolerance Score
test_servo1.sfr	0.48008386
...	
servoamp.cc	0.48008386
servoamp::servoamp	1
servoamp::~servoamp	1
servoamp::set_current	0.48008386
Line 89	1
Line 90	**0.48008386**
Line 94	0.83857442
Line 95	1
Line 96	0.95387841
servoamp::amps_to_dac	0.83857442
Line 350	0.83857442
servoamp::dac_to_amps	1
...	

The *Propagation Analysis Environment* (*PROPANE*) [Hiller et al. 2001] has been proposed for analyzing error propagation in component-based software. PROPANE injects errors at the inputs of a component (e.g., by corrupting messages or function parameters), logs its outputs, and identifies error propagation by comparing outputs to a golden run. It uses experimental results to evaluate the *error permeability* of components; in turn, permeability can be used to identify vulnerable or critical components that should be improved (e.g., to select suitable locations for error detection mechanisms).

The injection of errors at component interfaces has also been applied for designing protective *wrappers*, that is, additional software layers interposed between a component (e.g., COTS software) and the rest of the system. Error injection can be used to identify inputs that are not gracefully handled by the component (to be rejected using *input wrappers*) and component outputs that are not tolerated by the system (to be discarded using *output wrappers*) [Voas 1998]. In Salles et al. [1999], a COTS microkernel was extended with protective wrappers in order to prevent error propagation. In this case, wrappers are manually derived from specifications and validated through error injection.

An automated approach for deriving protective wrappers (*HEALERS*) was proposed by Fetzer and Xiao [2002] for C and C++ libraries: error injection experiments identify inputs that are not handled in a robust manner to automatically generate a library wrapper that serves as an input filter (Figure 6). In a similar way, the approach proposed in Susskraut and Fetzer [2006] (*AutoPatch*) enhances the robustness of applications with respect to *error codes* returned by external libraries: it injects error codes at library calls and patches the user's code if errors are not gracefully tolerated. Fetzer et al. [2004] improved error handling by injecting *exceptions* in C++ and Java software to identify exception handlers that leave an object in an inconsistent state (e.g., files left opened after an exception).

3.3. Dependability Benchmarking

A recent field of application for fault injection is represented by *dependability benchmarking* [Kanoun and Spainhower 2008]. A dependability benchmark compares the dependability of computer components or systems. A key aspect of dependability benchmarking, which makes it different from classical dependability evaluation techniques, is that it takes into account the interests (such as fairness and ease of applicability) of several stakeholders, including product manufacturers and users. Moreover, a

```
char* asctime (const struct tm* a1) {
        char* ret;

        if (in_flag)                            /* detects recursion */
                return (*libc_asctime)(a1);

        in_flag = 1;

        if (!check_R_ARRAY_NULL(a1,44)) {       /*
                errno = EINVAL;                  * checks that a1 is a string pointer to correctly
                ret = (char*) NULL;              * allocated memory; returns an error if this is
                goto PostProcessing;             * not the case
        }                                        */

        ret = (*libc_asctime)(a1);              /* invokes the real function */

        PostProcessing:;
                in_flag = 0;
                return ret;
}
```

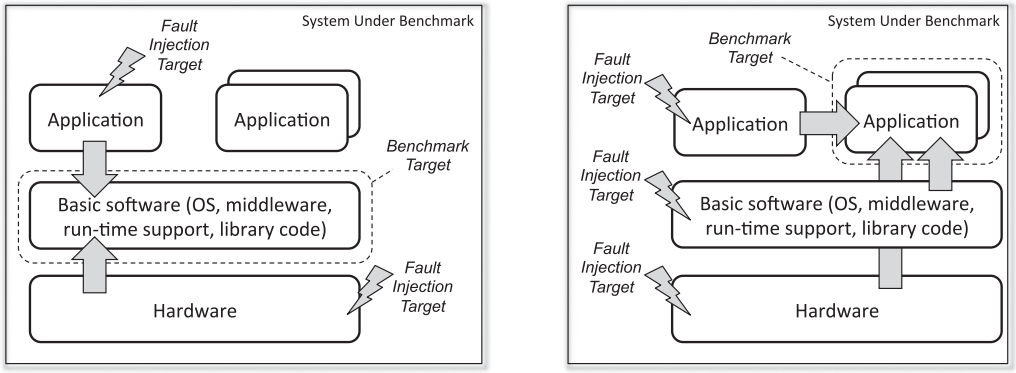
Fig. 6. Wrapper code for the *asctime* UNIX function [Fetzer and Xiao 2002].

dependability benchmark provides precise procedures and rules in order to support its users and to give meaningful and reproducible results.

A general framework for dependability benchmarking has been defined in the context of the DBench European project [2004]. This effort was followed by several studies that defined dependability benchmarks for many kinds of systems (e.g., OLTP systems, general-purpose and real-time OSs, engine control applications) [Kanoun and Spainhower 2008]. A dependability benchmark must precisely define (1) the context of the benchmark (e.g., the *domain* in which the system is adopted, the *operating environment* in which the system will work, the *life cycle phase* of the BT in which the benchmark is executed, the *benchmark performer* and the *benchmark user*), (2) the benchmark measures (e.g., end-user measures that characterize a system at the service delivery level, such as the number of transactions per minute, or measures associated to a specific aspect of the system, such as the coverage of an FTAM), and (3) the faultloads and workloads, which should be representative of faults and of the inputs that are experienced by the system in its operating environment. It is important to note that even if fault injection is an important aspect of dependability benchmarking, it is not the only evaluation approach that can be adopted. In the general case, benchmark measures are obtained from the combination of fault injection experiments and of the analysis of models.

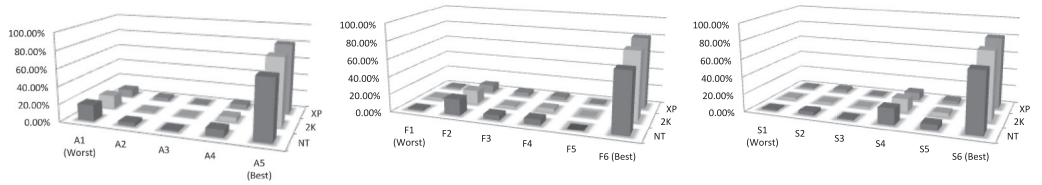
A dependability benchmark distinguishes between the *Benchmark Target* (BT), which is the benchmarked component or system, and the *System Under Benchmark* (SUB), which includes the BT along with the hardware and software resources needed by it (Figure 7). In the case of benchmarking through Software Fault Injection, a fundamental aspect is the separation between the BT and the *Fault Injection Target* (FIT), that is, the component subject to the injection of faults. This separation is required to avoid changing the BT: in order to get the benchmark accepted, especially by the vendor of the BT, it is more advisable not to modify the BT when applying the faultload. Moreover, injecting faults outside the BT allows one to compare several BTs with respect to the same set of faults, since injected faults are independent from the BT.

Figure 7 shows two typical scenarios in which, respectively, basic software and application-level software are benchmarked. In the first scenario, basic software is evaluated with respect to faults from both the lower layers (e.g., the hardware) and the upper layers (e.g., the applications). In this scenario, it is also possible to consider the system stack at a finer grain and to test one basic software component (e.g., a middleware)



(a) Benchmarking of basic software. (b) Benchmarking of application-level software.

Fig. 7. Typical dependability benchmarking scenarios and relationship between SUB, BT, and FIT.



(a) *Availability* perspective. (b) *Feedback* perspective. (c) *Stability* perspective.

Fig. 8. Comparison between Microsoft Windows OSs [Durães et al. 2003].

against faults in another basic component (e.g., the OS). In the second scenario, applications are the target of the benchmark, which must deal with faults from both software and hardware layers below them. The selection of benchmark components can result from both the software architecture and business considerations. A reasonable choice is to designate a COTS component as FIT (e.g., basic software such as an OS), since this software comes with residual faults that need to be tolerated by the rest of the system [Weyuker 1998]. Another choice is to consider a COTS component as a BT (e.g., a DBMS) in order to assess its relative value compared to alternative products (e.g., DBMSs from other vendors).

Early work used Software Fault Injection for OS dependability benchmarking [Durães et al. 2003]. This study compared three COTS OSs, namely, Windows NT4, Windows 2000, and Windows XP (which represent the BTs), against software faults in device drivers (which represent the FIT). This dependability benchmark ranked the target OSs with respect to the severity of their failure modes. In order to rank failures, the benchmark defined three *perspectives*:

—*Availability*: The failure modes that cause the unavailability of system functionalities or of the whole system are considered the worst ones.
—*Feedback*: The failure modes that provide little or no information to the user about the system state are considered the worst ones.
—*Stability*: The failure modes in which the system is working but provides an incorrect service are considered the worst ones.

Figure 8 shows the distribution of failure modes according to the aforementioned perspectives. The low number of severe failure modes denotes the ability of the OS to gracefully react to faulty device drivers in most cases. The Windows XP exhibited the

Table III. Measures of the Web-DB Dependability Benchmark [Durães et al. 2004]

Measure	Description
SPEC	Number of simultaneous conforming connections as defined in the SPECweb99 benchmark (i.e., an average bit rate of at least 320kbps and less than 1% of errors)
Throughput (THR)	Number of operations per second
Responsivity (RTM)	Average response time
Autonomy (AUT)	Percentage of cases in which an administration intervention would not be needed to restore the service (100 − (No. interventions/No. faults) · 100)
Accuracy (ACR)	Error rate (100 − (No. requests with errors/No. requests) · 100)
Availability (AVL)	The percentage of time in which the system is available to execute the workload (Amount of time the system is available during a run/the total duration of that run)

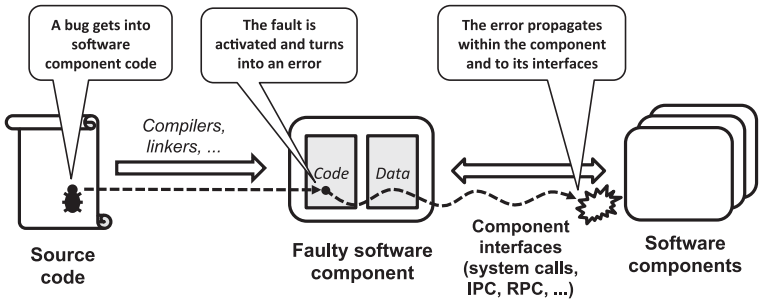
best behavior for all three perspectives. The same approach can be adopted to compare the target systems with respect to other perspectives, such as performance.

In Kalakech et al. [2004], dependability benchmarking was further developed by carefully taking into account the role of the workload. Windows NT, Windows 2000, and Windows XP were benchmarked in the presence of faults in user applications. Those systems were compared with respect to both their failure modes and their performance, in terms of their *reaction time* (i.e., time to execute a system call) and *restart time* (i.e., duration of OS restart). Experiments were conducted using a realistic workload rather than a synthetic test driver. Workload representativeness is an important aspect since it affects the propagation of errors and the reaction of the system in terms of performance and failure modes. Thus, the workload is a key aspect to extend the validity of results to real scenarios. The first definition of the dependability benchmark [Kalakech et al. 2004] used the TPC-C performance benchmark for transactional systems as workload [TPCC 2010], in order to take advantage of a well-established and agreed-upon workload. The benchmark was later extended by including the PostMark workload [Katcher 1997], which is representative of large Internet mail servers.

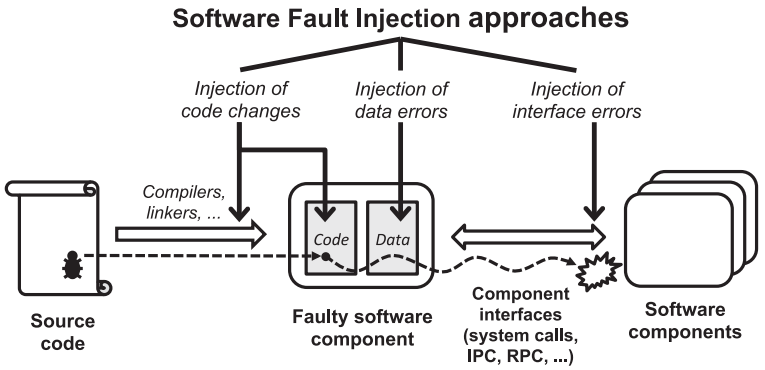
Durães et al. [2004] presented a dependability benchmark (*Web-DB*) for the evaluation of web servers with respect to dependability and performance. The faultload includes software faults in OS code and operator faults affecting connectivity, hardware, and system processes. The SPECweb99 performance benchmark for web servers [SPEC 2000] is used as workload. In a similar way to SPECweb99, the Web-DB benchmark includes (1) an initial ramp-up phase, in which the system is exercised to reach its maximum processing throughput; (2) a sequence of injection slots in which one fault at a time is injected; and (3) a ramp-down phase, in which the workload is terminated. This approach has been applied to OLTP systems too [Vieira and Madeira 2003; Kanoun and Spainhower 2008]. Benchmark measures (Table III) are derived from SPECweb99, and they characterize the performance and dependability of a web server in the presence of faults. Results obtained from fault injection are averaged and compared among them and to *baseline* results obtained in fault-free experiments.

4. OVERVIEW OF SOFTWARE FAULT INJECTION TECHNIQUES AND TOOLS

The emulation of software faults has been pursued in several ways, with many techniques and tools that have been developed over the course of two decades. We can identify three classes of fault injection approaches. The earliest class is the *injection of data errors* in the software state, by corrupting memory or registers in a similar way to hardware fault injection (Section 5). Several studies on this kind of fault injection focused on reducing intrusiveness and achieving efficiency and usability through, for example, the use of advanced debugging facilities, and by proposing tools that can be extended for new fault models and target systems. The second class, namely, *interface error injection* (Section 6), is a form of error injection that corrupts input and output



(a) Relationship between software faults and errors.



(b) Software Fault Injection approaches.

Fig. 9. Overview of Software Fault Injection.

values at component interfaces. It is adopted to evaluate and to improve the robustness of a component or of a system to invalid inputs and outputs. The third class, namely, the *injection of code changes* (Section 7), was developed to closely emulate software faults by introducing wrong code that mimics the most typical programming bugs.

To understand the relationship among those three types of approaches, we need to look at the failure process of a software system (see Figure 9(a)). In the beginning, the system is shipped with software faults, in the form of defective software components. These faults turn into an error state at runtime, which propagates through components' data. Ultimately, the errors surface at components' interfaces and impact on the rest of the system. The three approaches (injection of data errors, interface errors, and code changes) emulate the fault-to-error process in different points of the process, by injecting different types of corruptions (Figure 9(b)).

The fault models underlying these approaches make different assumptions about software faults and lead to different tradeoffs among (1) *representativeness*, (2) *usability*, and (3) *efficiency*, as we will discuss in the following. In some cases, fault models can cover different types of faults (e.g., both software and hardware faults). We clarify overlaps where necessary, paying specific attention on the emulation of software faults. Moreover, these approaches can be applied on different types of software components and on different levels of the system's stack, including basic software components (OS, middleware, runtime support, library code) and application-level software. For instance, in the case of dependability benchmarks (discussed in Section 3.3), Software Fault Injection has been adopted to inject faults at a specific level of the system's stack

(e.g., by injecting faults at the OS level) with the purpose of ensuring a fair comparison of the target systems or components.

The following sections survey these three classes of fault injection approaches, presenting for each class the most relevant techniques and tools. To give a structured presentation, we first present the general approach and the adopted fault models; then, we discuss more specific issues and solutions related to the following three *key properties* of Software Fault Injection:

—**Representativeness** is the ability of the faultload and of the workload to represent the real faults and inputs that the system will experience during operation. In the case of fault removal, this property is important to focus testing on faults that are most likely in practice. This property is also important for fault forecasting, since it is a necessary condition to ensure that dependability measures will reflect the actual behavior of the system [Arlat et al. 1990; Arlat and Moraes 2011]. The representativeness of faultloads is achieved by defining a realistic fault model, with respect to the following:
  —*Types and distribution of injected corruptions:* A fault/error model defines the corruptions to inject (in terms of *what*, *where*, and *when*; see Section 2) in program code (e.g., wrong or omitted instructions) and data (e.g., invalid pointers or indexes). These aspects are tuned to emulate the *types* of faults/errors that actually occur in practice. The relative amount of experiments for each fault type should reflect the *relative frequency* of that fault type, in order to avoid bias toward/against specific fault types. To achieve a match between the model and real faults, the definition of the model is often supported by a *field failure data analysis*, which derives fault/error types and their relative frequency from a postmortem analysis of failures, either in the system under evaluation or in similar systems. If field failure data are not available, fault/error types can be identified from a systematic analysis of the system's components and their potential faults based on expert judgment; unfortunately, this form of analysis cannot ensure the completeness of fault/error types and provides little or no information about the relative frequency of faults/errors.
  —*Types and distribution of failure modes:* Given that the goal of fault injection is to generate realistic failure modes to evaluate fault tolerance (see Section 3), the representativeness of a fault injection technique or tool can be *cross-evaluated* by comparing the failure modes from experiments against a reference distribution of failure modes (e.g., a distribution obtained from field failure data or from another reference fault injection technique/tool).
—**Usability** is the ability to use fault injection in a new target system. To this aim, the following aspects should be taken into account:
  —*Portability:* It is desirable to reuse a fault injection tool (with few or no modifications) across different target systems by supporting different software (e.g., OSs, middleware) and hardware (e.g., CPUs, networks) technologies. Portability is especially important to compare different systems or components (e.g., for dependability benchmarking). Moreover, in the case of COTS software, the fault injector should not require the availability of source code or detailed information about the internals of the target.
  —*Intrusiveness:* In order to be actually usable in practical applications, a fault injector needs to be nonintrusive; that is, its presence in the target system (e.g., for inserting faults and for collecting data) should not introduce significant perturbations that would distort the results of the experiments (e.g., performance measures and distribution of failure modes). This problem is especially important for performance-sensitive, real-time, and resource-constrained systems.

—*Flexibility:* A fault injection tool should allow the use of different fault models, and it should provide support for customizing fault models, and for adding ones, through, for example, configuration files, small programs, or graphical interfaces.

—**Efficiency** is the ability to achieve useful results with a reasonable experimental effort. In particular, the following factors reflect on efficiency:

—*Number of experiments:* The number of experiments impacts on the time and on the resources required to conduct a fault injection campaign. Therefore, this aspect significantly affects the experimental effort of fault injection. An efficient fault injection approach should keep as low as possible the number of experiments to discover FTAM deficiencies (in the case of fault removal) and to obtain a statistically significant evaluation of the system (in the case of fault forecasting). For example, this objective is pursued by an efficient selection of faults/errors to inject from the (potentially large) space of injections.

—*Activation and propagation of injected faults / errors:* Another factor that negatively affects efficiency is the injection of faults/errors that do not produce effects on the target system (see Figure 3). To reduce the number of ineffective injections, a fault injection approach needs to inject faults/errors that are likely to be activated and propagated, by tuning the type, location, and timing of the injections according to the characteristics of the target system and of the workload.

5. INJECTION OF DATA ERRORS

5.1. General Approach

The first approaches for Software Fault Injection grew in the context of studies on Software-Implemented Fault Injection (SWIFI) for hardware faults. SWIFI aimed to reproduce the effects (i.e., errors) of hardware faults (such as CPU, bus, and memory faults) by perturbing the state of memory or hardware registers through software. SWIFI overcame several problems associated with physical fault injection techniques (e.g., pin-level injection and heavy-ion radiation), such as controllability and repeatability of experiments [Arlat et al. 2003]. SWIFI approaches were also adopted for injecting data errors to emulate software faults [Barton et al. 1990].

In SWIFI, the contents of a memory location or register are replaced with a corrupted value, according to the following criteria [Christmansson and Chillarege 1996]:

—**What to inject.** A corruption is injected in the contents of an individual bit, byte, or word in a memory location or register. Error types have been defined from the analysis of errors generated by faults at the electrical or gate level [Ries et al. 1994]. Common error types are the replacement of a bit with a fixed value (*stuck-at-0* and *stuck-at-1* faults) or with its opposite value (*bit flips*).

—**Where to inject.** Error injection in memory can target a randomly selected location in a specific memory area (e.g., stack, heap, global data) or a user-selected location (e.g., a specific variable in memory). Error injection in a register can be targeted at registers that are accessible through software (e.g., data and address registers).

—**When to inject.** The error injection can be *time* or *event* dependent. In the first case, an error is injected after a given experiment time has elapsed (selected by the user or according to a probability distribution). In the second one, an error is injected when a specific event occurs during execution, such as at the first access or at every access to the target location. In these ways, three types of hardware faults can be emulated: *transient* (i.e., occasional), *intermittent* (i.e., recurring several times), and *permanent* faults.

It should be noted that SWIFI tools can inject errors both in the program state (e.g., data and address registers, stack and heap memory) and in the program code (e.g., in memory areas where code is stored, before or during program execution). This is

Table IV. SWIFI Error Models Used for Emulating CPU, Bus, and Memory Faults

What to inject	*Set, reset, or toggle of a bit/byte, AND/OR/XOR with a user-defined bitmap*
Where to inject	Memory: *stack*, *heap*, *global data area*, *user code*, *kernel code*, *user-defined memory location*; Registers: *data*, *address*, *stack*, *program counter*, *status register*
When to inject	*First access to memory location or register*, *Every access to memory location or register*, *Random time*, *User-defined time*

Table V. SWIFI Error Models Used for Communication Faults

What to inject	*Lose message*, *Duplicate message*, *Alter message header or body*, *Delay message*
Where to inject	*Faulty link*, *Faulty direction*, *Single message*
When to inject	*Random time*, *User-defined time*, *User-defined message*

an important aspect for Software Fault Injection: errors injected with SWIFI emulate the *effects* of software faults (i.e., an error), such as a wrong pointer, flag, or control flow, produced by the execution of a faulty program. This differs from the emulation of *actual* software faults by corrupting the code, which is discussed later in the context of "code changes" approaches (Section 7).

5.1.1. FIAT: Basic SWIFI Error Models. The *Fault-Injection-based Automated Testing environment* (*FIAT*) [Barton et al. 1990], one of the first SWIFI tools, was developed for the emulation of errors caused by both hardware and software faults (see Table IV). Errors are injected by a *fault injection library* linked with the target process, which corrupts memory on request from an external program. Library code is also used to observe the process behavior and to collect data. This solution overcomes OS protection mechanisms that prevent external processes from modifying the state of a target process. To improve usability, FIAT provided a user interface and a hardware/software architecture for assisting in the definition of workloads and faultloads, for controlling remote experiments, and for analyzing data (e.g., computing the coverage of error detection). One of the major limitations of FIAT was the need for linking the fault injection library at compile time (due to the lack of dynamic library linking at that time) using the object or the source code of the target. This approach is, unfortunately, not suitable for third-party and COTS software, for which object and source code is often not available. FIAT, as well as other early SWIFI tools, was developed for a specific real-time hardware/software platform, and fault injection was still adopted on an ad hoc basis for few, very critical systems. Thus, it was not meant to be portable across different target systems. Finally, as FIAT was one of the earliest software-implemented tools, the authors of FIAT hypothesized that SWIFI is representative of both hardware and software faults (e.g., by injecting a wrong register destination to emulate a wrong index variable), but this aspect was overlooked and only addressed in subsequent studies, as discussed in this survey.

5.1.2. DOCTOR and ORCHESTRA: Error Models for Distributed Systems. The *integrateD sOftware fault injeCTiOn enviRonment* (*DOCTOR*) [Han et al. 1995] tool extended SWIFI to emulate communication faults (its error model is summarized in Table V). This injector is implemented on the *x-kernel*, an OS kernel that allows one to introduce a layer in the protocol stack, in order to perform additional processing on network messages. A fault injection layer is inserted below the protocol or user program to be tested, and it intercepts operations between the target and lower protocol layers. For instance, in order to cause a delayed outgoing message, the fault injection layer stops a message and schedules a future message with the same contents to be sent later. This approach was also adopted by the *ORCHESTRA* fault injection environment [Dawson et al. 1996], which introduced a programming support (*script-driven probing*) for specifying message filters able to inject faults in complex protocols (e.g., to inject in a specific protocol state). Memory and CPU faults are also injected, both using the software trap mechanism to emulate transient faults and replacing binary instructions in the executable to

emulate permanent CPU faults. DOCTOR and ORCHESTRA addressed the important problem of communication faults. With the instrumentation of the intermediate network layer, these tools became flexible and portable across different target protocols and distributed systems, but they were intrusive and OS dependent and relied on user-provided scripts to efficiently inject faults. The error model reflects physical faults of the CPU, memory, and communication medium: however, this model does not encompass faults in application-level software.

5.2. Efficiency of Data Error Injection

An important problem for data error injection is the low efficiency of experiments due to latent injected errors that do not produce effects on the target system and do not test fault tolerance mechanisms. For instance, this occurs when the injected location is not actively used by the program or when it is overwritten before usage.

5.2.1. FTAPE: Stress- and Path-Based Injection. In the *Fault Tolerance And Performance Evaluator* (*FTAPE*) [Tsai et al. 1999], the workload is recognized as a key factor to improve the efficiency of SWIFI. The idea behind this tool is that stressful workloads cause a greater propagation of errors. Two fault injection approaches have been proposed in this work.

The first approach (*stress-based injection*) combines SWIFI with a workload generator and a workload activity monitoring tool (*MEASURE*). It generates stressful workloads for fault injection in order to increase the likelihood that injected errors propagate and produce a failure behavior. FTAPE randomly generates synthetic CPU, memory, and disk operations, and it monitors their stress level to determine the timing and location of an injection (CPU, memory, or disks). The stress-based approach is very simple to apply and to adapt to different target systems, since it does not require one to analyze and to instrument system internals. Even if, in practice, stressful random workloads seem to be an effective catalyst for error propagation, the experimental results can be more difficult to replicate and to debug, and there are no guarantees that injections will be actually exercised.

The second fault injection approach (*path-based injection*) is based on the preliminary analysis of resource usage (e.g., CPU registers and memory locations). It injects errors in intervals during which a resource (e.g., a register or a memory location) is "live" and being actively used. An input set is first defined to cover as many program paths as possible. Then, for each input, the code path associated to that input is determined. Finally, for each instruction in the path, it identifies CPU registers that can affect the instruction. FTAPE injected in CPU registers that control branch and jump instructions in the trace; this approach was later adopted in GOOFI (Section 5.3.3) by injecting in register and memory operands accessed by an instruction [Barbosa et al. 2005]. Path-based injection is thus able to focus error injection on resources that are not going to be reinitialized or left unused, thus avoiding injections that produce no effect on the system. Therefore, this approach can be very precise and effective. Nevertheless, it requires a preliminary dynamic analysis of all the executed instructions, which can be both storage and time consuming: the target software has to be executed in CPU debug mode, which interrupts the CPU at each instruction and significantly slows down its execution, requiring hours to analyze even small, embedded software. Another approach is to trace execution using a hardware debugger: however, even though it reduces the slowdown, it still has a high storage cost and increases the cost and complexity of the experimental setup.

5.2.2. SymPLFIED: Data Error Injection Using Symbolic Execution. This recent fault injection approach is based on the use of *symbolic execution* by the *Symbolic Program-Level Fault Injection and Error Detection Framework* (*SymPLFIED*) [Pattabiraman et al. 2008], which is aimed at the optimal placement of error detectors in a program.

Symbolic execution is adopted to increase the efficiency of error injection by systematically exploring "corner cases" that can be missed by traditional fault injection because of its inherent "statistical" nature.

In general, symbolic execution [King 1976] explores all states that can be reached by a program by replacing *concrete* variables (i.e., variables with specific values) with *symbolic* variables (i.e., variables that can assume a set of values at a given time, depending on which conditional statements have been executed until that point of the execution). Then, when symbolic execution encounters a conditional statement, it splits the current state into two states and separately explores two flows of execution starting from these two states. When generating new states from a conditional statement, symbolic execution narrows the set of valid values for each symbolic variable involved in the conditional statement (e.g., if a program block can only be reached when an integer variable x is negative, then the set of valid values for the symbolic variable is [INT_MIN, 0)). We refer the reader to Păsăreanu and Visser [2009] for a broader introduction to symbolic execution techniques.

SymPLFIED extended symbolic execution by taking into account the possibility of data corruptions in each state covered by symbolic execution. SymPLFIED symbolically executes the assembly code of a program by considering hardware faults that may occur at the hardware level and the resulting corruption of values in CPU registers. To model data corruptions, SymPLFIED generates new "corrupted" states from correct states, where register or memory locations are replaced with an erroneous symbolic value. Then, SymPLFIED symbolically executes the program from a corrupted state to analyze the propagation of the injected error in that state. SymPLFIED can prove the effectiveness of error detection and can guide their placement by pointing out errors that elude detection and affect the program outputs. This approach provides an exhaustive analysis of error detection, but symbolic execution tends to be computationally costly; thus, it is best suited for verifying error detectors in small systems. It is worth noting that the erroneous states introduced by SymPLFIED are focused on hardware faults, and they are not necessarily representative of software faults. Moreover, this approach has still not been applied for verifying more complex FTAMs.

5.3. Usability of Data Error Injection

Two important requirements of SWIFI in the context of embedded and real-time systems are flexibility and nonintrusiveness. To this aim, SWIFI adopted advanced features provided by the hardware and software environment.

5.3.1. FERRARI: Use of OS Debugging Mechanisms. The *Fault and ERRor Automatic Real-time Injector* (*FERRARI*) [Kanawati et al. 1995] SWIFI tool adopts debugging mechanisms provided by POSIX OSs to be portable across this family of systems and to overcome the need for the object/source code, like FIAT. FERRARI injects errors by corrupting the binary executable of the target program before execution or by corrupting the in-memory image of a process during execution. The latter mode is achieved through the *ptrace* POSIX system call of UNIX OSs, which allows a process to read and to write the memory of another process. This system call is used by an "error injection process" that, in a similar way to a debugging tool, interrupts the target process when a target instruction is executed (by replacing the instruction with a "software trap" instruction) or a given time elapsed, injecting an error. The target is then executed and monitored in order to analyze the impact of the error. Compared to other SWIFI tools such as FIAT, this approach does not require the object/source code of the target, and it is more suitable for COTS software. Moreover, it achieves a fair degree of portability across different target systems, since it is based on standard POSIX APIs. Nevertheless, it is more intrusive due to the additional error injection process and the software

trap mechanism, which cause process context switching and delays, and make this approach not well suitable for testing real-time systems.

5.3.2. Xception: Use of CPU Debugging Mechanisms. The *Xception* fault injection tool [Carreira et al. 1998] exploits the debugging and performance monitoring features of modern CPUs to reduce intrusiveness. Xception uses hardware features to improve the software trap approach: a *breakpoint register* is set up to trigger an *exception handler* (the fault injector) when a given instruction is executed, which performs fault injection. This approach improves nonintrusiveness by avoiding context switches, since the handler can be implemented and executed as a kernel routine. Moreover, breakpoint registers also allow one to inject faults when a specific data address is loaded or stored, which was not possible using the software trap mechanism adopted by previous tools. Finally, *performance counters* in the CPU are used for time-based fault triggering by injecting after a given number of processor cycles and for obtaining time measures such as error latency. Performance counters are more accurate than the system clock and can reduce uncertainties of time measures. Xception is more suitable for real-time systems than pure-software tools, such as FIAT, and tools based on elementary debugging mechanisms, such as FERRARI. The intrusiveness, in terms of extra code needed to inject faults and of execution overhead, is reduced to the minimum possible (interrupt handling routines with just a few machine instructions). However, since this strategy depends on specific hardware support, it is less portable and it may not be applicable for older or simpler CPUs that do not provide such features, which are still adopted in very critical systems.

5.3.3. GOOFI: Use of Scan-Chain Testing Mechanisms, and Object-Oriented Architecture. The *Generic Object-Oriented Fault Injection* (*GOOFI*) [Aidemark et al. 2001] tool is a SWIFI tool based on hardware features available in modern embedded boards. GOOFI makes use of *boundary* and *internal scan chains*, that is, built-in test logic of modern VLSI circuits that allow one to send and to read signals within a hardware board. This approach is referred to as *Scan-Chain Implemented Fault Injection* (*SCIFI*).

GOOFI adopts a generic software architecture to achieve portability across systems and across fault models. In particular, it is written in a platform-independent language (Java), and it defines internal object-oriented abstractions to assist the user in extending the tool. Specifically, it provides a *FaultInjectionAlgorithm* abstraction with generic methods (e.g., *loadWorkload()*, *injectFault()*, *readMemory()*, etc.) that can be specialized for a specific target system and fault model. To store and to analyze experimental data in a generic and reusable way, GOOFI defines a relational schema for an SQL DBMS to store high-level information about fault injection experiments (e.g., *experiment identifier*, *logged system state*, etc.), which are populated by specialized methods for the target system. Thanks to its extensible architecture, GOOFI is adaptable to systems and fault models not foreseen in the original design of the tool. As in the case of Xception, the adoption of hardware debugging mechanisms is useful for real-time systems, at the cost of an increased cost for tailoring the tool to a specific hardware platform.

5.3.4. NFTAPE: Lightweight Fault Injectors for Distributed Systems. The FTAPE tool was rewritten in later studies, with the aim to provide greater flexibility and portability (i.e., to let the user add new fault models, and to reuse most of the code of the tool when a new platform is targeted), and to support fault injection in distributed systems. These efforts led to the *Networked Fault Tolerance And Performance Evaluator* (*NFTAPE*) [Stott et al. 2000], which introduced the concept of *LightWeight Fault Injector* (LWFI), that is, a small and specialized program running on the target system for injecting faults that embeds the implementation of a fault model for a given target platform. To inject faults, the LWFI is invoked by the NFTAPE core tool, which runs

Table VI. ODC Defect Types [Chillarege et al. 1992]

Defect Type	Definition
Assignment	Value(s) assigned incorrectly or not assigned at all.
Checking	Missing or incorrect validation of parameters or data in conditional statements.
Algorithm	Efficiency or correctness problems that affect the task and can be fixed by (re)implementing an algorithm or local data structure without the need for requesting a design change.
Timing/Serialization	Necessary serialization of shared resources is missing, wrong resource has been serialized or wrong serialization technique employed.
Interface	Communication problems between users, modules, components, or device drivers and software.
Function	Affects a sizeable amount of code and refers to capability that is either implemented incorrectly or not implemented at all.

on a remote computer, and which implements most of the fault injection tasks (e.g., logging, configuration, communication) in a reusable framework. Due to its modular design, NFTAPE was successfully ported on several platforms, ranging from FPGA boards to UNIX systems [Stott et al. 2000], and it was extended to use different fault models [Xu et al. 2001; Bondavalli et al. 2004], which were not foreseen in the initial design. It should be noted that, to keep intrusiveness at a minimum, NFTAPE makes little effort to trace and to control the global state of the distributed system. Unfortunately, the exact timing of an injection is not precisely controllable, due to the lack of a global clock and to communication delays (e.g., between the core tool and the LWFI): this uncertainty makes it difficult to accurately trigger an injection in a specific global state of the system. Later studies addressed this issue by trading off the control over the experiment with runtime overhead and duration of the experimental campaign [Chandra et al. 2004; Cotroneo et al. 2013]: after the execution of each experiment, event traces (collected at each node in the system) are analyzed using an offline clock synchronization algorithm in order to identify injections that were triggered in an incorrect global state and to repeat them if needed.

5.4. Representativeness of Data Error Injection

As discussed earlier, data error injection was not originally meant to emulate software faults in a representative way. SWIFI tools adopted the same error model for emulating both hardware and software faults by assuming that injected data corruptions (e.g., bit-flips) also match the errors generated by software faults. However, the error states of a faulty program can be much different than those caused by random physical faults, such as wear-out and electromagnetic interferences. In turn, the lack of representativeness negatively affects the assessment of FTAMs since, as stated previously in Section 3.1, injecting unrealistic types of faults can lead to misleading coverage and latency measures.

Christmansson and Chillarege [1996] proposed a procedure for defining error models that emulate software faults instead of hardware faults. This property is achieved by defining a *mapping between injected errors and the most common types of software faults*, using field failure data of the system under analysis. In particular, software faults were analyzed using the *Orthogonal Defect Classification* (ODC) [Chillarege et al. 1992], which is widely used among researchers and industries. ODC is a framework for classifying software faults found during the software life cycle (including development and field usage) and for analyzing their defect distributions across the different phases to get quantitative feedback about the software development process. It generalizes previous classification schemas adopted for OS and DBMS products from IBM [Sullivan and Chillarege 1991; Chillarege et al. 1991]. ODC classifies defects into *defect types* according to the *fix* made by programmers to correct them (Table VI). ODC defect types

Table VII. Joint Distribution of Faults and Errors Found in the Field in the IBM MVS Operating System [Christmansson and Chillarege 1996]

		ODC Defect Types					
Error Types	**#**	**Chk.**	**Ass.**	**Alg.**	**Tim.**	**Int.**	**Fun.**
Single Address (A)	**78 (19.1%)**	**13**	**27**	**26**	**4**	**5**	**3**
Control block addr.	48	9	16	17	3	2	1
Storage pointer	14	1	5	3	1	2	2
Module addr.	9	3	3	3	0	0	0
Linkage of data structure	4	0	0	3	0	1	0
Register	3	0	3	0	0	0	0
Single Nonaddress (N)	**155 (38.0%)**	**30**	**38**	**53**	**12**	**15**	**7**
Value	38	10	6	12	3	2	5
Parameter	38	8	11	10	1	6	2
Flag	37	7	10	17	0	3	0
Length	15	4	4	4	0	3	0
Lock	11	0	1	2	8	0	0
Index	8	1	3	4	0	0	0
Name	8	0	3	4	0	1	0
Multiple (M)	**69 (16.9%)**	**9**	**6**	**32**	**6**	**4**	**12**
Values	4	0	1	2	0	0	1
Parameters	3	0	0	0	0	3	0
Address +	7	1	1	3	0	0	2
Flag +	10	2	1	4	2	0	1
Data structure	37	6	3	20	3	1	4
Random	8	0	0	3	1	0	4
Control error (C)	**106 (26.0%)**	**10**	**7**	**43**	**32**	**5**	**9**
Program management	35	4	0	19	8	2	2
Storage management	33	3	7	12	5	1	5
Serialization	16	0	0	2	14	0	0
Device management	11	2	0	7	2	0	0
User I/O	6	0	0	2	0	2	2
Complex	5	1	0	1	3	0	0
Total	**408**	**62**	**78**	**154**	**54**	**29**	**31**
	100%	**15.2%**	**19.1%**	**37.8%**	**13.2%**	**7.1%**	**7.6%**

are *orthogonal* (i.e., mutually exclusive) and are close to the programmer's perspective, since they are based on the fix of the defect.

The procedure proposed in Christmansson and Chillarege [1996] defines an error model in terms of *what error types should be injected* and *where the errors should be injected*. The procedure is based on field failure data, which provide the joint distribution of ODC defect types, and of errors that they produced. Table VII shows the joint distribution of fault and errors for the IBM MVS OS. Error types are grouped in *Single address* (i.e., the software fault caused an incorrect address word), *Single Nonaddress* (i.e., a nonaddress word is affected), *Multiple* (i.e., a combination of single errors, or a data structure is affected), and *Control* (i.e., errors affecting memory in a very subtle and nondeterministic way or not affecting memory at all, such as wrong interaction with a user or terminal communication). As the authors pointed out, many error types are product specific (a mature OS), and their distribution does not necessarily apply to other systems.

The procedure uses the joint distribution of faults and errors to establish a *connection between injected errors and software faults they intend to emulate*. Where to inject errors (e.g., variables or data structures) is established by scanning the program code (e.g., using a parser) to identify, for each statement, (1) the ODC defect type that applies to the statement (e.g., the "Assignment" defect type in the case of an assignment statement) and (2) the error type that applies to that statement (e.g., the "Storage

pointer" error type in the case that the assignment involves a storage pointer). This processing provides a list of triplets *(location, defect type, error type)*. The errors to inject are obtained by sampling this list. Error representativeness is achieved by sampling the list according to the relative frequency of defect and error types in the joint distribution (Table VII). As for *when to inject errors*, it is argued in Christmansson and Chillarege [1996] that error injection must be synchronous with the execution of the emulated faulty statement (i.e., an error is injected every time the location associated to the injected error is executed) to emulate the permanent nature of software faults. The injection of errors through a SWIFI tool is proposed by using a software trap in the code location selected for error injection.

An experimental comparison between the injection of representative errors and generic time-triggered errors [Christmansson et al. 1998] showed that there can be noticeable differences in the distributions of failure modes obtained by these two techniques, and that error representativeness has a significant impact on the results. A procedure similar to Christmansson and Chillarege [1996] was defined later in Christmansson and Santhanam [1996] for *fault removal from fault tolerance*. In this case, errors to be injected are selected by giving priority to the ones that (1) have a high failure severity as perceived by the customers and (2) are correlated with fault tolerance deficiencies (i.e., an exception handler or a recovery procedure was triggered but was not able to tolerate them).

These procedures used SWIFI tools for the emulation of software faults. However, some limitations prevented their adoption in practice. The need for field failure data of the system under test is one of the main limitations, since they are often not available in the early phases of the software life cycle. Moreover, field failure data are typically not available for third-party and COTS software. Another issue is that SWIFI tools can accurately inject only a limited number of error types, since the errors caused by software faults are often not limited to individual memory words (e.g., as in the case of software faults involving several program statements), and error types tend to be specific for the system under analysis. These limitations have been faced by approaches for the injection of *code changes* in the target program, which will be discussed in Section 7.

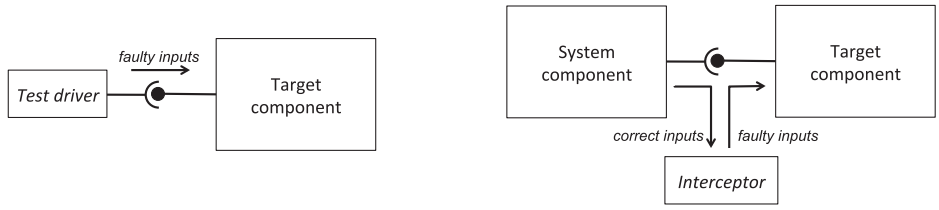
6. INJECTION OF INTERFACE ERRORS

6.1. General Approach

Interface error injection is a particular form of error injection that corrupts the *input* and *output values* that a target component exchanges with other software components, the hardware, and the environment (Figure 9). Error injection at input parameters emulates the effects of faults outside the target (e.g., software faults affecting software components that use the target) and evaluates the ability of the target to detect and to handle corrupted inputs (e.g., input sanitization). In a similar way, the corruption of output values is adopted to emulate the outputs of faulty components and can be used to assess their impact on the rest of the system.[1]

Interface error injection is commonly adopted for *robustness testing*, which evaluates "the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions" [IEEE 1990]. The injection of interface errors is especially useful for assessing software components (e.g., COTS) that provide a generic *Application Programming Interface* (API). In such components, which are generic and not developed for a specific system, interface errors can arise

[1]This form of fault injection is referred to as "[interface] error injection" or "failure injection" in some studies. Other studies refer to it as "fault injection," since corrupted values can be seen as "external faults," according to the definitions of Avizienis et al. [2004].



(a) Test driver. (b) Interceptor.

Fig. 10. Approaches for interface error injection.

when they are combined with other software, due to unforeseen interactions among components and to incorrect usage of the API. An example is an OS that exposes an API for device drivers, which is provided to third-party suppliers in order to support new devices. In order to mitigate interface errors from other components, API implementations typically provide error detection mechanisms to identify erroneous inputs and to prevent them from further propagating through the component.

Interface error injection can be performed in two ways (Figure 10). The first approach is based on a *test driver* program that is linked with the target component (e.g., a program that uses the API of the target) and that exercises it with invalid inputs. This approach resembles unit testing, but it focuses on FTAMs (e.g., error and exception handlers) instead of functional verification. The second approach consists of intercepting and corrupting the interactions between the target and the rest of the system: an *interceptor* is triggered when the target component is invoked, and it modifies the original inputs in order to introduce a corrupted input. In this scenario, the target component is tested in the context of the whole system that integrates the component. This approach resembles SWIFI, since the original data (in this case, interface inputs) flowing through the system is replaced with corrupted data.

The criteria adopted for interface error injection are as follows [Johansson and Suri 2005]:

—**What to inject.** An invalid value to inject is generated by:
  —**Fuzzing**: replacing a correct value with a randomly generated one.
  —**Bit-flipping**: corrupting a value by inverting one of its bits.
  —**Data-type based injection**: replacing a value with an invalid one, selected on the basis of the *type* of the parameter being corrupted. The types are derived from the analysis of services exported by the target interface (e.g., APIs). For each type, this approach defines a pool of invalid values from the type domain by considering those values that tend to be more difficult to handle (e.g., "NULL" in the case of C pointers, or error codes in the case of integer return values).
—**Where to inject.** Errors are injected at the input or output parameters of a software interface, which can be identified from the architecture of the target system. An injection can be performed on any call site that invokes a service of the target component (e.g., a function exported by an API).
—**When to inject.** The injection can be triggered when a service of the target component is invoked. For instance, the injection can be determined by counting the number of invocations of a specific service (e.g., an error is injected at the first or at the *n*th invocation) or by waiting a given amount of time (e.g., the first invocation after a timer elapsed).

6.1.1. Fuzz and Ptyjig: Robustness Testing of UNIX Applications Through Interface Error Injection. One of the first studies on interface error injection [Miller et al. 1990] evaluated the robustness of UNIX utilities with respect to inputs from external programs and users.

Two tools, *Fuzz* and *ptyjig*, were developed to submit a random stream of data to the target through the standard input and through the terminal device. The study found that a significant number (between 24% and 33%) of utility programs on three UNIX systems are vulnerable to interface errors, causing a process crash or stall. Moreover, error injection pointed out several bugs, such as buffer overruns and unchecked return codes. A subsequent experiment [Miller et al. 1998] found that the same utilities were still affected by a significant part of faults found in Miller et al. [1990], and that similar issues affected network and GUI applications. The injection of corrupted interface data (even if randomly generated) is a cheap but effective way to discover severe bugs even in commercial and mature software. Being one of the first studies in this field, the approach was purely random and fostered further research on the problem of generating corrupted data in a systematic way, in order to enhance the efficiency of interface error injection.

6.1.2. FIG: Robustness Testing of UNIX Applications Against the C Library. The *Fault Injection in Glibc* tool (*FIG*) [Broadwell et al. 2002] aimed at testing the robustness of desktop and server applications in the UNIX environment by injecting errors from the C library. This tool emulates errors caused by excessive load (e.g., failed memory allocation), hardware faults (e.g., a failed disk), and software faults (e.g., an OS system call that has been invoked with incorrect parameters). This kind of error injection is useful to test error handling code in user applications, which should deal with error conditions signaled by the C library (e.g., by gracefully terminating the program) and would be difficult to exercise with traditional testing approaches.

FIG focused on commonly used library functions (*malloc()*, *read()*, *write()*, *select()*, *open()*) by injecting their most common erroneous return codes (*ENOMEM*: insufficient memory, *EINTR*: system call was interrupted, *EIO*: general I/O error, *ENOSPC*: insufficient disk space). To inject these error codes, FIG uses a *wrapper library* with the same API interface as the original C library, which is dynamically linked to the user program through the LD_PRELOAD mechanism provided by the UNIX loader. This wrapper returns an error code when an injection is triggered; otherwise, it forwards function calls to the original C library. This tool had limited portability, since great effort was required to adapt it to new target systems and libraries, as error codes to be injected had to be manually specified by the user for each target library function.

The FIG tool has been applied on desktop (e.g., Emacs, Netscape) and server (e.g., Berkeley DB, MySQL, Apache HTTPD) UNIX applications to identify library errors that are not gracefully handled by the application; for example, the application abruptly crashes, corrupts critical data, or lacks informative messages to the user about the error. We emphasize that this form of interface error injection focuses on *faults detected by basic software (e.g., libraries, OS) and signaled to user applications (e.g., using special return values or exceptions)* by deliberately injecting "signaled errors" at library interfaces (e.g., by forcing a special return value). But, as already pointed out in Section 6.4, these errors represent only a subset of errors that can be caused by faulty software, which also include "unsignaled" and mixed errors. For this reason, this approach is better suited for *fault removal* purposes (Section 3.2), by systematically detecting application code that lacks proper handling of errors signaled from library invocations. It is less suitable for *fault forecasting* and *dependability benchmarking* purposes (Sections 3.1 and 3.3), since the faultload is biased toward signaled errors and it can provide only a partial evaluation of dependability.

6.1.3. Ballista: Data-Type-Based Robustness Testing and Benchmarking of Operating Systems. Dingman and Marshall [1995] first, and then Koopman and DeVale [2000], adopted the data-type-based error injection approach in order to test OS robustness, respectively, in the context of a Real-Time OS (RTOS) and of POSIX-compliant OSs [IEEE 1994].

Table VIII. Data Types for Testing the *Write* System Call
[Koopman and DeVale 2000]

File Descriptor	Memory Buffer	Size
FD_CLOSED	BUF_SMALL_1	SIZE_1
FD_OPEN_READ	BUF_MED_PAGESIZE	SIZE_16
FD_OPEN_WRITE	BUF_LARGE_512MB	SIZE_PAGE
FD_DELETED	BUF_XLARGE_1GB	SIZE_PAGEx16
FD_EMPTY_FILE	BUF_NULL	SIZE_PAGEx16plus1
...	...	...

They tested OS robustness against invalid inputs passed to the *system call interface* in order to assess the ability to handle errors generated by faulty user-space programs. A robustness test case consists of a system call invocation by a test driver program, which passes a combination of both valid and invalid values to the system call. Invalid values for an input parameter are selected from a set of predefined values for the data type of that parameter.

In particular, the *Ballista* approach proposed in Koopman and DeVale [2000] tests the robustness of the POSIX system call interface [IEEE 1994] in a scalable way, by defining 20 data types for testing 233 system calls of the POSIX standard. Invalid values were taken from the testing literature or selected according to the experience of developers and represent exceptional situations induced by faulty applications. Table VIII provides examples of three data types used in the *write* system call [Koopman and DeVale 2000]. This system call takes three input parameters, namely, a file descriptor, a pointer to a memory buffer, and an integer representing the buffer size. Exceptional values for a file descriptor parameter include a file that has been opened and deleted from the file system, or an empty file. In a similar way, exceptional values for a memory buffer include extremely large buffers, or a buffer that has been previously deallocated. It is important to note that the definition of data types is not tied to the semantic of a specific system call (e.g., the "memory buffer" data type can be used for any system call involving memory buffers).

The outcome of a robustness test case is determined by recording the error code returned by the system call (if any) and by monitoring system processes using a watchdog process (e.g., a failure occurs if a process unexpectedly terminates during the experiment, or it is stalled). Ballista uses a *failure severity scale*, which is referred to as *C.R.A.S.H.*, for classifying test results:

—***C****atastrophic*. The OS state becomes corrupted or the machine crashes and reboots.
—***R****estart*. The OS never returns control to the caller of a system call, and the calling process is stalled and needs to be restarted.
—***A****bort*. The OS terminates a process in an abnormal way.
—***S****ilent*. The OS system call does not return an error code.
—***H****indering*. The OS system call returns a misleading error code.

Ballista was adopted to compare COTS UNIX OSs [Koopman and DeVale 2000] and, in later studies, to evaluate the robustness of Microsoft Windows OSs, of CORBA ORB implementations, and of web services [Kanoun and Spainhower 2008; Laranjeiro et al. 2014]. In Koopman and DeVale [2000], a total of 1,082,541 tests on 15 UNIX OSs were automatically generated and executed. The inputs causing most of the robustness failures were invalid file and buffer pointers, and out-of-range integer values.

These tests were used to compare alternative OSs, based on the distributions of failure modes. We remark that comparisons are made on a *qualitative* basis, since the perceived severity of failure modes may vary among different users. Many users consider *Catastrophic* failures a severe robustness gap of an OS, since the isolation

between a faulty user process (such as the test driver) and the OS is often a critical requirement. However, in other contexts, such as distributed systems, *Silent* failures (i.e., faulty system call invocations that lack error signaling) are considered more severe failures, since they can go undetected for a long period of time and are more difficult to recover than a fast, easily detected crash of a node. Moreover, the severity of failure modes depends on the approach adopted by developers for exception handling. Some OSs provide fine-grain error codes to point out invalid system call invocations. Other ones immediately kill processes that appear to be misbehaving, resulting in a higher percentage of *Restart* failures. The second approach is useful to force developers to notice an error but may be inappropriate in some safety-critical contexts.

We also point out that Ballista and, more in general, robustness testing cannot be adopted alone for a comprehensive evaluation of dependability. Robustness testing does not encompass OS failures in the presence of other types of faults, such as hardware faults, resource leaks, and high-volume workloads. Moreover, the lack of information on the relative frequencies of invalid system call inputs (as required in Equation (2)) hampers quantitative evaluations (e.g., in probabilistic terms). The parameters of Equation (2) depend on user applications and other factors outside the OS under test. Nevertheless, robustness testing proved to be useful at finding severe OS bugs (e.g., inputs-of-death causing *Catastrophic* failures) and at pointing out and comparing error handling in different OSs, such as in the case of "forced" *Restart* failures (which cause large differences in the distributions of failure modes).

6.1.4. MAFALDA: Interface Error Injection in Microkernels Through Library Interposition. The *Microkernel Assessment by Fault injection AnaLysis and Design Aid* (*MAFALDA*) tool [Arlat et al. 2002] was proposed for assessing microkernel OSs through interface error injection. MAFALDA injects both errors within a microkernel component and interface errors at the interfaces of microkernel components. Interface data are corrupted through the *bit-flipping* approach to emulate both hardware and software faults. The representativeness of bit-flips for the emulation of software faults, as even acknowledged by its authors, was still not established at that time and was a largely open research issue. Interface error injection in MAFALDA is based on the interception of system call invocations (Figure 10): (1) it intercepts system calls issued through *software interrupts* by inserting a trap in interrupt service routines; (2) it intercepts system calls implemented in library code by linking the user-space application with a *fault injection library* that wraps the microkernel library and that invokes it by corrupting the system call inputs.

6.2. Efficiency of Interface Error Injection

The following techniques enhanced the efficiency of interface error injection, in terms of the number of experiments and of fault tolerance issues found, by tuning the type, location, and time of error injection.

6.2.1. Grammar-Based Interface Error Injection. The fuzzing approach is easy to implement and can reveal deficiencies in error handling and recovery, but its efficiency has been questioned, since it relies on many trials and "luck." In Ghosh et al. [1998], it is pointed out that unstructured random tests mostly cover the input parsing code of the program, but that they are not efficient at stressing other software functions. In fact, the target software may immediately discard syntactically incorrect inputs, or inputs that do not mix valid and invalid values, without even starting to process them. Therefore, the *Random and Intelligent Data Design Library Environment* (*RIDDLE*) [Ghosh et al. 1998] extended the fuzzing approach for robustness testing of Windows NT utilities by generating syntactically correct inputs able to test more thoroughly the target program. To generate erroneous inputs, RIDDLE adopts a grammar to describe the format of

inputs in a similar way to the Backus-Naur form; random and boundary values are generated for tokens representing input parameters. This approach can increase the efficiency of robustness testing, at the cost of an increased effort to set a grammar up, in order to describe the space of program inputs. Typically, this approach requires more knowledge about the target than purely random ones. It is most effective when it is applied to components with "standard" interfaces (e.g., the POSIX interface for UNIX), since the definition of the input space can be reused across different targets.

6.2.2. Combining Error Models. Since the interface of complex software components consists of several methods and parameters, there is a very high number of potential injections that can be performed. Therefore, several studies were focused on improving the effectiveness of interface error injection at revealing robustness issues by carefully selecting the most advantageous error model [Johansson et al. 2007b; Winter et al. 2011]. Several studies of this kind were conducted on robustness testing of the device driver interface of modern OSs, such Linux [Albinet et al. 2004] and Windows [Johansson and Suri 2005], since faulty third-party device drivers represent a major cause of OS failures, and since OSs tend to be vulnerable against faulty device drivers: developers tend to omit checks at the device driver interface to improve performance, and they trust device drivers more than user applications.

Johansson et al. [2007b] and Winter et al. [2011] compared the bit-flipping, fuzzing, and data-type-based approaches with respect to their effectiveness in detecting vulnerabilities and the efforts required to set up and execute experiments. On the one hand, they found that bit-flipping is the most effective approach (in terms of number of vulnerabilities found), although it incurs a high execution cost due to the large number of experiments. On the other hand, data-type-based injection and fuzzing are more efficient in terms of fraction of experiments able to find vulnerabilities, although they incur a higher implementation cost (e.g., in the case of the data-type-based approach, the user has to define exceptional values for each data type). An important conclusion of these studies is that the best tradeoff between effectiveness and cost can be achieved by performing experiments with several error models: in the case of Johansson et al. [2007b], performing two campaigns with fuzzing and selective bit-flipping (i.e., focused on a subset of bits) achieved a higher efficiency, since these two error models were complementary and found different vulnerabilities. These studies further compared different error models with respect to the efforts required to implement them in a fault injection tool (*implementation complexity*). In Winter et al. [2011], this effort has been quantified using two syntactic code metrics, *delivered source instructions* and *cyclomatic complexity*, computed from the code of fault injection tools. They found noticeable differences between implementations of error models, where the bit-flip model was the cheapest model to implement and the data-type-based model was the most expensive one. However, the implementation cost of a model does not seem to be correlated with the execution time of that model, since data-type-based injections required more effort to set up and led to a lower execution time.

While the methodologies adopted in these studies have a broad application (e.g., to different target systems and to different fault/error models), their experimental results are not necessarily valid for other contexts. Thus, we remark that researchers and practitioners, before making assumptions or decisions based on an experimental study, must pay attention to whether their own context is similar to the one of the experimental study. For instance, experimental studies on OS robustness testing can be useful, to some degree, to other low-level software in C that tightly interacts with hardware, as in the case of embedded systems, but it would be risky to assume that results from these experiments are also valid for, say, web services using J2EE technologies. Moreover, researchers that report experimental results should consider generalization issues when

planning an experimental study: this practice is referred to as *validity evaluation* in the field of software engineering [Wohlin et al. 2012]. To foster generalization of experimental results, researchers should (1) identify the *population* of interest, that is, the broader set of systems and faults to which the experiment is aimed (in particular, fault injection considers systems with dependability requirements); (2) perform experiments on systems and faults that are representative of this population; and (3) when reporting results, clearly discuss how generalization issues (often referred to as *threats to validity*) have been managed, and warn about any residual limitation that has not been addressed.

6.2.3. Tuning the Triggering of Error Injection With State Models. A complex system performs its functions by interacting many times with its software components through APIs. The same API failure (e.g., an exception or error code raised by an API function) can be injected whenever that API function is invoked. The timing of injection is a critical parameter, since the effect of the injection depends on the *state* of the system at that time. In fact, systems' failures are most probable if an error occurs during the execution of critical tasks, or when holding shared resources, or under some other subtle condition. Uncovering these vulnerable states using "black-box" approaches is inefficient [Suri and Sinha 1998], and thus, formal techniques were proposed to support experiments.

Johansson et al. [2007a] analyzed the efficiency of interface error injection with respect to the timing of error injection in the context of OS robustness testing. They evaluated the approaches typically adopted for interface error injection, namely, *first occurrence* (i.e., at the first time in which a code location is executed) and *time-triggered* injection (i.e., after a timer has been elapsed). They also proposed a novel approach. First, the usage of OS API functions is profiled. The sequence of function calls is then divided into subsequences, namely, *call blocks*: each call block represents a recurring sequence of calls, such as "reading data" or "setting connection parameters," and denotes a distinct state of the OS. To improve the efficiency of experiments, errors are injected at every first occurrence of a function call *in each distinct call block*. This approach improved robustness testing experiments on Windows CE by detecting vulnerabilities that could be triggered only by injecting in specific call blocks, thus highlighting how the system state can impact on the results.

Prabhakaran et al. [2005] developed a model-based approach that takes advantage of a similar idea to perform robustness testing of journaling file systems. A file system is tested by injecting I/O errors at the device driver interface: such errors can be caused, for instance, by damaged disk sectors, transient or permanent faults in the system bus, and bugs in the device driver. The timing of error injection is tuned according to a state model of the journaling file system. This model describes the sequences of disk writes that can be performed by a file system, including data, journal, commit, checkpoint, and superblock writes. Using this model, the fault injector is able to inject errors for different types of writes and at different times during a sequence of several writes. Experiments on three file systems in Linux showed that file systems react differently (ranging from data corruptions to unmountable file systems) depending on the type and time of write failures, which can be efficiently injected by this approach. Moreover, having high-level semantic information about what the file system is doing at the time of the error makes it easier for developers to identify and fix file system vulnerabilities.

A critical aspect of model-based approaches is that they require detailed knowledge about the target system and a manual analysis to define state models. To overcome this limitation, the *SABRINE* approach [Cotroneo et al. 2013] automatically infers state models of an OS component under test. The basic idea of this approach is that the internal state of an OS component is determined by the history of interactions with

other components. Thus, SABRINE collects traces of interactions between the target component and other OS components and extracts a *behavioral model* from them using clustering and FSM learning techniques. Then, it performs distinct error injections at each state of the behavioral model, thus efficiently covering different states of the target. The approach was experimented on two filesystems and a device driver of a Linux-based OS for safety-critical applications by comparing SABRINE to random-time robustness testing. These experiments showed that SABRINE can efficiently find robustness issues while significantly reducing the number of tests.

The approaches presented in this section have been developed and validated in the context of OS and system code (e.g., server applications). As discussed in Section 6.2.2, it is important to consider that these approaches are not necessarily valid for other systems. Other aspects that these studies did not consider, and that could be investigated to improve efficiency, are (1) the applicability of the approaches to other error models (e.g., multiple errors), (2) the representativeness of errors injected using a state model (as some types of errors can occur only in specific states, or some states can be more prone to errors than others), and (3) the evaluation of the quality of state models and related formalisms, along with the tradeoff between coverage and efficiency.

6.2.4. Heuristic-Driven Exploration of the Error Space. The previous approaches tune error models by considering different definitions of the error space (e.g., in terms of type and timing of errors). However, *sampling* the space of injectable errors can further increase the efficiency of error injection. In fact, when considering complex software systems, the number of combinations of *what*, *where,* and *when* to inject typically reaches millions of possibilities. In these cases, exhaustive exploration is not feasible, and the space of experiments needs to be sampled.

The *Library Fault Injector* (*LFI*, further discussed in Section 6.3.2) is a tool that emulates errors at the interface between applications and system libraries. It injects erroneous return codes from library functions (e.g., in output parameters and global variables), extending the approach of FIG (Section 6.1.2). LFI introduced a number of techniques to make library error injection more efficient and usable for modern server and user applications. To improve efficiency, LFI embeds heuristics for sampling error injection experiments by giving priority to errors that are more likely to find robustness issues. In particular, it performs a profiling of program locations (*callsites*) that invoke library functions, by inspecting whether the caller program checks for error codes that can be returned by the library. If return values are not checked by the code near the callsite, then the callsite is a suspicious location that needs to be tested. LFI performs a dataflow static analysis of the caller code by following the propagation of return values and by looking for instructions that compare these return values with some constant. Then, priority is given to callsites that do not check return values at all and to callsites that only check for a subset of potential error values from the library.

The *Automated Fault Explorer* (*AFEX*) [Banabic and Candea 2012] is a general, black-box approach to drive the selection of fault injection experiments. AFEX uses a fitness-guided algorithm to efficiently sample the space of experiments. It is based on the observation that this space has often an inherent *structure*; that is, there are attributes of experiments that are more likely to discover robustness issues: for instance, programmers may often misuse a specific API function or neglect a specific error code.

AFEX models an experiment as a tuple $<\alpha_1, \ldots, \alpha_n>$, where each α_i represents an attribute of the experiment (e.g., an API function or error code). It generates injection experiments dynamically, by mutating individual attributes of experiments that have already been performed. To exploit structures in the experiment space, AFEX records the experiments and the attribute mutations that had the highest *fitness* (e.g., in terms of number and severity of failures discovered). It assigns to each past experiment and

to each attribute a probability based on their impact and selects which experiment and which attribute to mutate according to their relative probabilities, thus giving priority to the most rewarding attributes. Moreover, AFEX can take advantage of domain knowledge of developers, which can tune the selection process, by excluding or, on the contrary, focusing on specific API functions and error codes.

Experiments in server applications (MySQL and Apache HTTPD) confirmed that the use of search heuristics can provide guidance to error injection, improving its efficiency over random sampling of the error space. This approach paves the way for investigating whether the "structure" hypothesis holds for other error models, such as for the injection of invalid input parameters (e.g., in robustness testing, such as in Ballista).

6.2.5. Injection of Multiple Interface Errors. A recent trend in fault injection research is to consider the injection of multiple faults during the same experiment. Multiple injections are motivated by the observation that, even if multiple faults are rarer than single ones, they tend to be neglected during development and testing, and are often less tolerated by a system. Moreover, with the growing scale of computer systems and the higher degree of complexity and integration of hardware and software components, the injection of multiple faults is becoming more and more important.

The study in Winter et al. [2013] investigated how Software Fault Injection can benefit from multiple error injection. This study presented a rationale and formal definitions for multiple injections, distinguishing among (1) *discrete,* that is, single external faults, faults; (2) *spatially coincident / temporally spread* faults, that is, external faults affecting the same interface entity (e.g., same service or parameter) at different times (e.g., different service invocations or experiment runs); and (3) *temporally coincident / spatially spread* faults, that is, external faults affecting different interface entities at the same time. While previous studies mostly focused on faults of the first and second kind, Winter et al. [2013] analyzed in detail *temporally coincident* faults and performed experiments on the device driver interface of Windows CE to compare them with discrete faults. They injected multiple interface errors by simultaneously corrupting several bits or several parameters of the same service invocation through the combined use of fuzzing and of bit-flipping (obtaining the *FuzzFuzz*, *FuzzBF*, and *SimBF* error models). They found that multiple faults improve the coverage of robustness vulnerabilities. Moreover, even if the contribution of each error model varies among targets and workloads, the *SimBF* error model can achieve a significantly high efficiency in this specific case study. Moreover, *SimBF* leads to the lowest implementation complexity among multiple error models, while, as expected, combining two error models as in the case of *FuzzBF* requires a high implementation effort.

Fate [Gunawi et al. 2011] and its successor, *PreFail* [Joshi et al. 2011], are tools for testing cloud software systems against multiple faults from the environment (e.g., disk failures, network partitions, crashes of remote processes) through the injection of multiple interface errors during the same experiment. The main motivation is that single faults can trigger recovery mechanisms in these systems but do not test the fault tolerance of recovery procedures (i.e., their ability to tolerate additional faults that occur during recovery): for example, while a distributed file system is recovering from the crash of a replica, other crashes or network/disk faults could also occur.

PreFail performs experiments in several rounds by iteratively increasing the cardinality of the set of injected errors. During the kth round, PreFail performs experiments with k error injections, targeting *fault injection points* in the software (e.g., a code location in which I/O is performed, and where an I/O error could be injected). For each experiment, it records the new fault injection points that are executed after the injection of the kth error. It then generates a set of new experiments for the next round (with $(k + 1)$ injections) by including the new fault injection points.

To deal with the very high number of multiple fault injections, this tool provides policies to make fault injection more efficient. Before the $(k + 1)$-th round, PreFail applies the policies to prune the set of experiments that are redundant, according to some user-defined property. For example, policies can remove experiments that target recovery code paths that have already been covered; another example is to discard redundant experiments that target functionally equivalent components, such as identical process replicas, since it would be useless to perform several distinct experiments by only varying the injected replica. In this way, these tools were able to mitigate the issue of the combinatorial explosion of experiments for popular, complex cloud software.

As discussed in Section 6.2.2, experimental results on the efficiency of error models should be interpreted with caution, as the relative value of error models can vary across different systems. These studies represent fundamental contributions toward a better understanding of multiple faults and will contribute to the general problem of multiple fault injection. Further studies in this area should look at the nature of robustness vulnerabilities and of multiple faults (e.g., which multiple faults tend to occur in production, and which kind of software components and interfaces are most vulnerable) to support the definition of even more efficient criteria for selecting multiple injections.

6.2.6. SAGE: Interface Error Injection Using Symbolic Execution. To make fuzz testing more efficient, recent work adopted symbolic execution [King 1976] to systematically generate malformed inputs. These approaches combine *concrete* and *symbolic* execution into *concolic execution* in order to obtain a higher coverage and bug-finding power, compared to random and grammar-based fuzzing, which are only based on concrete execution. The *Scalable, Automated, Guided Execution* (*SAGE*) [Godefroid et al. 2008] is a relevant example that has been developed at Microsoft for fuzzing file- and packet-parsing applications against untrusted sources.

SAGE generates malformed inputs starting from well-formed ones. The program is first executed using some tentative concrete inputs. Branches taken during this initial execution are recorded in order to identify the set of *constraints* on inputs that must hold to cover the execution path (as in symbolic execution). Then, one of the constraints in the set is negated, and new malformed inputs are generated to satisfy the new set of constraints. In this way, SAGE forces the program to execute corner cases not covered by the initial inputs. For example, let's assume a tentative input stream (e.g., data read from a file) containing the values $x_1 = 5$ and $x_2 = 10$ (e.g., the first two bytes of the input stream). Moreover, let's assume that the program, after reading these values, encounters the following statements: `if(`$x_1 > 0$`) { if(`$x_2 < 100$`) { block1 } else { block2 } } else { block3 }`. In this example, the symbolic execution would follow the path leading to `block1` and would identify the constraints $C_1 : x_1 > 0$ and $C_2 : x_2 < 100$ along this path. To generate new inputs, SAGE negates constraints of the path (e.g., obtaining the path constraint $C_1 \wedge \neg C_2$) and uses a *constraint solver* to find inputs that satisfy the new path constraint (e.g., $x_1 = 5$ and $x_2 = 100$) and that steer the program execution toward corner cases (e.g., `block2`). Finally, the program is tested with the new inputs, and this process is repeated starting from the new inputs.

One of the issues for symbolic execution is the huge number (up to billions for large-scale software) of constraints and paths (along with the high computational cost to symbolically execute each path), along with the high computational cost to symbolically execute each path. Moreover, the lack of accuracy at executing some complex program elements (e.g., pointer manipulations, operations on strings and floats, calls to OS functions, and sources of nondeterminism) causes a divergence between the program execution path followed by symbolic execution for a given input, and the actual execution path for that input. To cope with these issues, SAGE includes a search algorithm that explores the most promising inputs first (based on block coverage gained

with respect to previous runs) and avoids getting stuck on paths that cannot be accurately analyzed with symbolic execution.

This approach was demonstrated to be effective at finding vulnerabilities that would be otherwise very difficult to find (i.e., vulnerabilities that are triggered only by few inputs within a very large input space), reaching one-third of all bugs found by fuzz testing in Microsoft projects [Bounimova et al. 2013]. It is also important to note that, even if very powerful, this technique cannot ensure that all vulnerabilities are identified, given the infeasibility of full path exploration, and due to inaccuracies in symbolic execution (up to 60% of explored paths in SAGE [Godefroid et al. 2008]). Moreover, the large computing power required by this technique may not be affordable for many users. For these reasons, symbolic execution should be applied in conjunction with other testing and verification techniques, including random and grammar-based fuzzing. In the first pass of testing, traditional techniques can find "easier" vulnerabilities (that are, at the same time, the most likely to surface during operation, and thus are very impactful). Symbolic execution could then focus on more subtle issues to make the best use of the time and resources required to apply this approach.

6.3. Usability of Interface Error Injection

The usability of interface error injection was pursued by using techniques for automatically instrumenting the interfaces of software components and by enabling developers to configure fault injection experiments (e.g., to filter fault locations and to combine fault triggers) in a user-friendly way.

6.3.1. Jaca: Interface Error Injection in Java Software Using Reflection. The *Jaca* tool for interface error injection [Martins et al. 2002] is based on *reflection*, a feature of modern programming languages that allows a program to inspect and manipulate its own structure at runtime. Jaca injects interface errors in Java programs by modifying input and output values of class methods through reflection mechanisms provided by the Java Virtual Machine (JVM). It introduces *class wrappers* that intercept and modify input and output values for a class. Moreover, Jaca avoids the need for source code by performing reflection at the bytecode level, it is portable across JVM implementations, and it adopts an extensible architecture for the introduction of new fault models. This is a flexible approach that takes advantage of the software infrastructure underlying the target system. Nevertheless, it can be challenging to apply this approach to embedded systems, which have fewer resource and more stringent performance requirements (thus requiring a very small time and storage overhead), and which provide much limited facilities to application software, thus requiring more effort to set up error injection.

6.3.2. LFI: Automated and Flexible Library-Level Fault Injection. An important issue for interface error injection tools is the need to identify API functions and their error codes. This issue is particularly important when considering modern server and end-user applications, which today make use of tens or even hundreds of components in the form of shared libraries. Early tools, such as FIG [Broadwell et al. 2002], required the user to manually specify the error model (e.g., error codes for emulating memory allocation and I/O faults) or hardcoded the model in the tool. However, these approaches cannot scale to modern systems since it takes too much time and effort to implement error injection for a new library and to maintain injectors up to date with new versions of a library. Moreover, there can be different implementations of the same library, with subtle differences across different platforms (e.g., some error codes may be returned only under a specific OS), which makes it difficult to provide an accurate and exhaustive error model. These problems are exacerbated by the lack or incorrectness of library documentation and hamper the practical adoption of error injection by developers.

The *Library Fault Injector* (*LFI*) [Marinescu and Candea 2011] tool includes several techniques that achieve high usability in an automated and flexible fashion. In particular, the tool proposes the following techniques:

—**Profile library code to automatically define error models (*what* to inject)**. LFI automatically infers error codes that are returned by a library, through a static analysis of its binary code. In this way, LFI derives the error model from the library code and avoids inaccuracies. To identify error codes, LFI first constructs a control flow graph (CFG) of a library function and of other functions on which it depends. Then, for each return instruction, LFI looks for paths on the CFG that propagate constant values (that are typically used for indicating failure conditions) to a *return location* (e.g., a memory location storing the return value) before the execution of the return instruction. Even if this static analysis has some limitations (regarding libraries that make use of indirect function calls and return values that can only be returned under specific inputs), it proved to be effective at handling real-world shared libraries.
—**Identify library callsites that are likely to be vulnerable (*where* to inject)**. As discussed in Section 6.2.4, LFI gives priority to library invocations that do not perform checks on return values, and thus are potentially more vulnerable. Failure-prone callsites are automatically and accurately identified by LFI, relieving developers from manually identifying them.
—**Enable flexible triggering of injections (*when* to inject)**. To control the timing of injections, LFI provides a set of predefined *injection triggers*, including callstack-based triggering (the injection occurs when the stack of function calls matches a user-defined pattern); coverage improvement triggering (injects when the callstack is different than callstacks of previous injections); program-semantic triggering (injects when a given relationship between program variables holds); call-count trigger (injects at the *n*th call to a given library function); and random triggering. These triggering conditions are checked at runtime by LFI to decide when to inject. The overhead of these checks is lower than 5% in DBMS and web server applications. The user can control triggering by defining *injection scenarios*, in which the user can select, configure, and combine the basic triggers provided by LFI. Moreover, LFI exposes a programming interface that allows users to implement their own triggers, based on domain knowledge.

LFI is a very flexible tool, which serves as a basis for error injection tests. In order to be efficient, this tool needs to be coupled with criteria and methodologies for defining test plans, since the number of potential injections is in most cases too large to be covered exhaustively. Search heuristics and fine-tuned error models, such as those presented in Section 6.2, can take advantage of this flexibility by focusing injections on the most representative or the most severe errors, in order to make the best use of developers' time. The design of LFI serves as a reference for injecting interface errors in other systems not based on library components, such as service-oriented and component-based applications.

6.3.3. Destini and PreFail: Programmable Injections and Specifications. The *Destini* [Gunawi et al. 2011] and *PreFail* [Joshi et al. 2011] tools (also discussed in Section 6.2.5) allowed the user to deal with a very high number of multiple-injection experiments using small, highly expressive programs. The *Destini* tool can be used by developers for customizing *test specifications* (i.e., fault tolerance properties that need to be fulfilled in the presence of faults) and for checking that the system complies with them. Concise and effective specifications facilitate the practical adoption of fault injection, since developers

```
errDataRec(B, N) :- cnpComplete(B), expectedNodes(B, N), NOT-IN actualNodes(B, N);
```

(a) Part of a test specification in *Destini*.

```
def flt(fs):

    last = FIP( fs[ len(fs) - 1 ] )

    return not explored( last, 'loc' )
```

(b) Filter policy in *PreFail*.

```
def cls(fs1, fs2):

    last1 = FIP( fs1[ len(fs1) - 1 ] )
    last2 = FIP( fs2[ len(fs2) - 1 ] )

    return (last1['loc'] == last2['loc'])
```

(c) Cluster policy in *PreFail*.

Fig. 11. Programmable injection policies and test specifications.

can write as many detailed specifications as needed, refine these specifications for debugging purposes, and reuse them on different systems and versions.

In Destini, test specifications are defined using Datalog, a declarative relational logic language. These specifications are expressed in terms of events (e.g., failures and protocol events), and relations over them representing expectations and facts (e.g., data blocks or packets that are expected, and that are actually observed). For instance, the test specification in Figure 11(a) asserts that a data transfer failure occurs when there is a node `N` that is expected to store a valid replica of `B` but actually does not (relations `expectedNodes` and `actualNodes`, whose definition is omitted here); moreover, this condition is checked at the time of a block completion (event `cnpComplete`). When the system under evaluation provides a rich set of events and information, this approach enables developers to write specifications that are more concise and expressive (five lines on average) compared to specifications in imperative languages (tenths of lines on average) [Gunawi et al. 2011].

The PreFail tool allows users to customize the policies for pruning the set of multiple-injection experiments. Policies can be customized to account for domain knowledge, for instance, by focusing on the type and location of faults that developers know to be most likely, and by considering the allocation of processes across nodes and racks to inject more realistic faults (e.g., network partitions between processes that are on remote nodes). PreFail considers two types of policies: *filter policies*, which remove experiments that do not comply with a user-defined condition, and *cluster policies*, which remove experiments that are equivalent to other ones according to a user-defined condition. Figure 11(b) and Figure 11(c) are examples of policies for increasing code coverage: the first policy filters experiments whose last fault injection point (FIP) code location has already been covered by a previous experiment, and the second policy clusters experiments whose last FIPs have the same code location. Using the libraries provided by PreFail (e.g., functions exposing fault injection points), policies are typically very small, with an average of 17 lines of Python code [Joshi et al. 2011]. It must be noted that programmable tools for fault/error injection require the user to get acquainted with new languages and frameworks, which can increase the learning curve and the efforts. Thus, it is advisable to integrate programmable tools with existing, popular languages; IDEs; and testing frameworks (e.g., the same frameworks already being used for writing and running regression tests) in order to achieve usability. Moreover, the integration with IDEs and reporting facilities can make easier to handle, interpret, and debug large amounts of tests and results.

6.4. Representativeness of Interface Error Injection

Interface error injection is a popular approach that has been adopted in mission-critical systems (Sections 6.1.3 and 6.1.4), as well as server and user applications (Sections 6.1.1, 6.1.2, and 6.3.2) and cloud computing systems (Section 6.3.3). Interface

errors injected by these tools emulate errors caused by faulty hardware and, to some degree, faulty software. Other approaches, not included in this survey, emulate other kinds of environment faults at the boundaries of a system, such as operator mistakes, by corrupting configuration files (e.g., by mutating configuration directives) [Keller et al. 2008] and by performing incorrect administration actions (e.g., deletion of database files and objects) [Vieira and Madeira 2003]. However, the representativeness of errors with respect to software bugs is still an open issue.

Recent studies provided some evidence that error models currently adopted for interface error injection do not necessarily match errors generated by software faults. Jarboui et al. [2002] compared interface errors injected at the inputs of the system call interface of the Linux kernel to errors injected at internal kernel functions and to real software faults found by static code analyzers. Faults and errors were compared with respect to (1) distributions of failure modes, (2) return codes of system calls, and (3) effects on internal assertions that were introduced in the kernel to monitor its behavior, such as the amount of memory allocations in the kernel. The analysis revealed that interface errors caused a different behavior than internal errors, which were in turn different from real software faults.

The analysis presented in Moraes et al. [2006] on two complex components (an RTOS and an object-oriented DBMS) compared errors injected at interfaces to errors produced when software faults are injected within components' code. These two forms of fault/error injection exhibited significant differences in the distribution of failure modes of the target system.

A recent study [Lanzaro et al. 2014] looked more in depth at how software faults, injected in library code, turn into interface errors (i.e., errors affecting data structures exchanged between the library and a user program, through input/output parameters and return values). The study performed a dynamic program analysis to track down, at the byte level, corruptions of interface data structures. Interface errors were compared against error models typically adopted for interface error injection (Section 6.1), such as bit-flipping (e.g., as in MAFALDA [Arlat et al. 2002]) and error code injection (e.g., as in LFI [Marinescu and Candea 2011]). The study analyzed three popular C libraries, finding noticeable differences between actual and injected interface errors: (1) the extent of interface data corruptions is, in the majority of cases, wider than individual bytes or words; (2) a significant part of faults are not signaled by an error code; and (3) when an error code is generated, interface data also get corrupted. Therefore, a representative interface error model should inject a mix of small and large data corruptions, and of signaled and nonsignaled data corruptions.

This experimental evidence suggests that existing interface error models are representative of software faults only to a limited extent. Injecting interface errors allows one to assess fault tolerance against some specific interface errors. However, it is difficult to ensure that injected errors are the most likely to occur in practice, and thus their representativeness. This issue hampers the use of interface error injection for the statistical or comparative evaluation of fault tolerance properties (e.g., in dependability benchmarking), since the lack of representativeness can distort the outcome of the evaluation. The issue of fault representativeness is further discussed in Section 7 and Section 8.

7. INJECTION OF CODE CHANGES

7.1. General Approach

To pursue the representativeness of software faults, several studies on SFI focused on the injection of faults in the program code (i.e., code changes). Empirical studies found that the injection of code changes can realistically emulate software faults [Daran and

Thévenod-Fosse 1996; Andrews et al. 2005], in the sense that *code changes produce errors and failures that are similar to the ones produced by real software faults*. Code changes are injected according to the following criteria [Madeira et al. 2000]:

—**What to inject.** A small set of program instructions are replaced by other instructions (e.g., instructions with different operands, or *nop*s). The definition of changes is based on common types of software faults found in real systems (e.g., ODC types).
—**Where to inject.** A fault is injected in the program code, either in its source code or in its binary executable. The code location is selected according to type of fault. For instance, a fault that affects variable assignments can only be injected in a code location containing a variable assignment. Other criteria for selecting target locations are possible, such as injecting faults only in fault-prone locations according to software complexity metrics.
—**When to inject.** To reproduce the permanent nature of software faults, faults should be injected in the target code before its execution [Daran and Thévenod-Fosse 1996; Hudak et al. 1993]. Christmansson and Chillarege [1996] proposed to inject synchronously with the execution of the target code region. More recent studies relaxed this constraint by injecting code changes at runtime asynchronously in order to force the occurrence of failures at a desired or random time [Ng and Chen 2001; Durães et al. 2004]. This approach was adopted to simplify the implementation and the execution of fault injection experiments, such as to perform an injection when the system is in a steady state.

7.1.1. Injection of Code Changes Through Mutation Testing Tools. Pioneer work on assessing fault tolerance through the injection of code changes was based on mutation testing tools. In Mahmood et al. [1984], faults were seeded in the source code for evaluating the error detection ability of assertions (e.g., "plausibility" checks on program variables and inputs/outputs) in flight software. A similar experiment was made by Hudak et al. [1993]: a program mutation tool (*FAUST*) was used to assess the effectiveness of several fault tolerance techniques by injecting defects in the source code of target programs. Nevertheless, the main problem to apply mutation tools in dependability assessment is the definition of realistic fault types. Existing mutation operators have been defined with the goal of guiding the selection of the test case; thus, they are not necessarily representative of residual software faults (i.e., faults that affect the system after testing and release). The relationship between mutation testing and SFI is further discussed in Section 9.

7.1.2. FINE and DEFINE: Injection of Code Changes Based on ODC. The *Fault Injection and Monitoring Environment* (*FINE*) [Kao et al. 1993] was the first tool aimed at SFI using code changes, by introducing faults in the executable (binary) code of a target program. FINE was developed to assess UNIX OSs in the presence of hardware and software faults. This tool was extended by the *DEFINE* tool [Kao and Iyer 1994] to support the execution of experiments in a distributed environment and included additional fault types related to hardware and communication. The FINE and DEFINE tools were used to study the impact and propagation of faults, respectively, in SunOS UNIX and in the NFS distributed filesystem.

The fault model adopted by FINE (Table IX) was based on an early version of the ODC fault classification schema, adopted in Chillarege et al. [1991] (see also Section 5.4). The Initialization, Assignment, and Checking types were implemented in FINE, while the definition of the Function type (for which there was no predefined fault pattern, as this type of fault varies across different target systems) was left to the user. The tool aimed at achieving fault representativeness by covering all of these fault types. Faults were injected by changing the executable code of the target.

Table IX. Fault Model for Software Faults Adopted in FINE [Kao et al. 1993]

Fault Type	Description
Initialization	The initialized value is replaced with an incorrect value, or no initial value is given (the corresponding instructions are changed to *nop*s).
Assignment	The assignment destination is assigned a wrong value, the evaluated expression is assigned to the wrong destination (and the right destination is not assigned the evaluated value), or the assignment is not executed (the corresponding instructions are changed to *nop*s).
Checking	The branch instruction is replaced with *nop* for a missing condition check, or the condition check is changed for an incorrect condition check.
Function	A user-defined sequence of instructions is replaced with another user-defined sequence of instructions (faulty instructions should fit into the space of the original instructions).

Table X. Fault Model Adopted by Ng and Chen [2001]

	Example	
Fault Type	**Correct Code**	**Faulty Code**
Initialization	`function () { int i=0; ... }`	`function () { int i; ... }`
Missing random instruction	`for(i=0; i<10; i++,j++) { body }`	`for(i=0; i<10; i++) { body }`
Incorrect destination register	`numFreePages = count(freePageHeadPtr)`	`numPages = count(freePageHeadPtr)`
Incorrect source register	`numPages = physicalMemorySize/pageSize`	`numPages = virtualMemorySize/pageSize`
Incorrect branch	`while(flag) { body }`	`while(!flag) { body }`
Corrupt pointer	`ptr = ptr->next->next`	`ptr = ptr->next`
Allocation management	`ptr = malloc(N); use ptr; use ptr; free(ptr);`	`ptr = malloc(N); use ptr; free(ptr); use ptr again;`
Copy overrun	`for(i=0; i<sizeUsed; i++) { a[i] = b[i]; }`	`for(i=0; i<sizeTotal; i++) { a[i] = b[i]; }`
Synchronization	`getWriteLock; write(); freeWriteLock;`	`write();`
Off-by-one	`for(i=0; i<size; i++)`	`for(i=0; i<=size; i++)`
Memory leak	`ptr = malloc(N); use ptr; free(ptr); return;`	`ptr = malloc(N); use ptr; return;`
Interface	`results = strcmp(str1, str2);`	`results = strcmp(str1, str3);`

However, the fault injection procedure of FINE and DEFINE was simplistic since it overlooked the problem of defining which particular program entities should be injected (e.g., which assignment should be corrupted among the several ones in the program) and how to replace the original program entities with incorrect ones (i.e., how to select an incorrect value or variable to assign for an *Initialization* or *Assignment* fault). A closer analysis of the experimental results points out a limitation of code changes: a high number of injections are not activated during an experiment, and thus do not produce any noticeable effect on the target system. For instance, this phenomenon occurs when the fault location is difficult to exercise and very specific workloads are required to trigger the faults.

7.1.3. Fault Model for Representative Code Changes in Operating Systems. Another fault model for injecting software faults was proposed for the evaluation of a fault-tolerant write-back OS file cache [Ng and Chen 2001]. This fault model (Table X) was based on a study on OS failures that preceded the Orthogonal Defect Classification [Sullivan and Chillarege 1991]. Some of these fault types are injected by modifying instructions and their operands in the machine code, in a similar way to FINE (e.g., missing initialization, incorrect source or destination register, incorrect branch, pointer corruption). The remaining fault types are injected by modifying the behavior of kernel functions (e.g., for emulating synchronization and memory management faults, kernel functions do not perform the required operation, and overrun and corrupt memory areas according

Table XI. Comparison Between Distributions of Defect Types in Two Field Data Studies

	Distribution	
ODC Defect Type	Durães and Madeira [2006]	Christmansson and Chillarege [1996]
Assignment	21.1%	21.98%
Checking	25.0%	17.48%
Interface	7.3%	8.17%
Algorithm	40.1%	43.41%
Function	6.1%	8.74%

to the error distribution from Sullivan and Chillarege [1991]). This study was an important step toward the injection of representative faults, since its fault model was based on empirical data on real faults in an OS. The same authors in Ng et al. [1996] took into account the likelihood of faults (in quantitative terms) in order to probabilistically estimate coverage factors (see Section 3.1). However, this fault model is not general and does not necessarily apply to other systems. Moreover, some important aspects regarding the implementation of the fault model are still simplistic, such as the selection of incorrect operands to be used for replacing correct ones. If not properly implemented, this problem could lead to the injection of unrealistic faults. For instance, if a register operand is replaced with another register that does not store a program variable (e.g., a register with a temporary variable introduced by the compiler), then the fault in the machine code does not reflect a fault made by a developer.

7.1.4. G-SWFIT: Generic and Representative Fault Model. The problems of defining a representative fault model and of accurately injecting faults in the machine (binary) code were investigated more in depth by Durães and Madeira [2006], which proposed the *Generic Software Fault Injection Technique* (*G-SWFIT*). This technique initially adopted a fault model on common C programming bugs from various sources such as programming manuals, best-practice tutorials, and error reports [Duraes and Madeira 2002]. The fault model was improved in Durães and Madeira [2006] through an extensive analysis of bug fixes in several open-source and commercial software projects. A key point of the fault model is that it is *generic*, since it holds for several systems, and can thus be adopted even in the absence of field data about the specific system under analysis. As shown in Table XI, defect types follow the same trend across field data studies and systems: Algorithm defects are the largest part, Assignment and Checking defects have approximately the same weight, and Interface and Function defects are the less frequent ones. This result has an important implication on SFI, since this distribution can be adopted for defining a faultload without requiring field failure data about the specific system under analysis, which are usually not available.

The field data study in Durães and Madeira [2006] looked at bug fixes in order to achieve a more precise characterization. In particular, the ODC defect types were extended to relate the faults with the *programming language construct* (e.g., statements, expressions) that is either *missing*, *wrong*, or *extraneous*. This classification is oriented toward automated fault injection, since it gives an indication on how to modify a program in order to introduce a fault (e.g., the construct to be removed in order to inject a *missing* construct fault). The analysis revealed that the majority of the faults belong to few fault types. The study identified the most common fault types according to two criteria: (1) the number of occurrences of the fault type must be at least as high as the average, and (2) the occurrences should not be restricted to only one or two of the programs. Table XII reports the fault types fulfilling these criteria, which account for 68% of all faults found in the field. The study argues that these types should be addressed in the first place when emulating software faults, and that other types of faults are likely very rare and do not occur consistently in all software projects.

Table XII. Most Common Fault Types Found in the Field [Durães and Madeira 2006]

			ODC Defect Type				
	Fault Types	**#**	**Ass.**	**Chk.**	**Int.**	**Alg.**	**Fun.**
Missing	*if* construct plus statements	71				✓	
	AND sub-expr in expression used as branch condition	47		✓			
	function call	46				✓	
	if construct around statements	34		✓			
	OR sub-expr in expression used as branch condition	32		✓			
	small and localized part of the algorithm	23				✓	
	variable assignment using an expression	21	✓				
	functionality	21					✓
	variable assignment using a value	20	✓				
	if construct plus statements plus *else* before statements	18				✓	
	variable initialization	15	✓				
Wrong	logical expression used as branch condition	22		✓			
	algorithm - large modifications	20					✓
	value assigned to variable	16	✓				
	arithmetic expression in parameter of function call	14			✓		
	data types or conversion used	12	✓				
	variable used in parameter of function call	11			✓		
Extraneous	variable assignment using another variable	9	✓				
Total		**452**	**93**	**135**	**25**	**192**	**41**
Coverage relative to each ODC type (%)		**68%**	**65%**	**81%**	**51%**	**72%**	**100%**

The proposed fault types also provide rules ("constraints") for identifying fault locations in order to better emulate the faults found in the field. For example, the "missing function call" fault type should only affect function calls that do not use or do not get a return value, and that are not the only statement in a code block—it is unrealistic that a programmer forgets a function call but still uses its return value. Other examples are the "missing IF plus statements" fault type, which removes IF constructs containing no more than five statements; the "missing small part of algorithm" fault type, which removes between two and five consecutive statements that are not control or loop statements; and the "wrong variable used in parameter of function call" fault type, in which a function parameter is replaced with a variable that represents a local variable in the module.

It is worth noting that faults are related to syntactic constructs and, ultimately, to the programming language of a system. The fault types in G-SWFIT were derived from C programs. In principle, they could be applied to procedural programming languages similar to C, since they focus on programming constructs also available for other languages, such as control flow and assignment constructs. However, it is still necessary to evaluate whether these fault types are still representative even for those languages. Some later studies evaluated this fault model, and extended it, to Java software [Basso et al. 2009; Sanches et al. 2011], confirming that the fault types in Table XII are still the most frequent ones. Field failure studies of this kind are important to support that the generality of fault types is valid and to apply them on different programming languages.

Even if these fault types account for a significant part of faults, they do not include fault types that are system specific, since G-SWFIT is focused on generic fault types that apply across several systems. In particular, this problem affects faults of the *Function* ODC type (i.e., capabilities that are wrongly designed or omitted), which tend to be system specific as they involve functionalities of a particular system. Changing or removing a functionality goes beyond simple syntactic changes and would

require knowledge about the target system. Therefore, emulating Function faults in a generic yet representative way remains an open issue. Another issue of this kind is due to fault types that tend not to be recorded by developers and users, since they manifest only under nonreproducible conditions. This is the case, for instance, of the *Timing/Serialization* ODC type (e.g., faults involving synchronization and resource contention, such as race conditions and deadlocks). These faults are difficult to track down even when retrying the failed operation (e.g., they can be reproduced only under a specific thread interleaving), leading to a lack of data about them. Research in this area extended G-SWFIT with additional concurrency fault types by leveraging field failure data specifically focused on this kind of fault [Natella and Cotroneo 2010], but the large number of technologies and paradigms for concurrent programming makes the problem elusive.

7.2. Usability of Code Change Injection

7.2.1. Injection of Code Changes in Binary Code. COTS software represents a challenge for approaches based on code changes, as vendors typically distribute COTS software in the form of executable *binary* code without providing their source code. For this reason, code changes must necessarily be injected by changing the binary code of the component. This fact poses the problem of the *accuracy* of binary code mutations: an injection into binary code should accurately emulate a software fault as if it would be injected in the source code. The main cause of inaccuracy is the gap between source code and binary code, as it is often difficult to identify high-level programming constructs by only looking at their translation in binary code.

G-SWFIT [Durães and Madeira 2006] injects faults in the binary code of the target: it analyzes the software to recognize key programming structures at the machine code level. More specifically, G-SWFIT provides a set of *fault emulation operators* that define (1) a *search pattern*, that is, a sequence of machine code instructions that correspond to some high-level programming construct, and (2) a *code change*, that is, the mutation to be introduced in that location to emulate the intended fault type. The code patterns are tailored for a specific hardware platform and assume that a specific source code compiler has been used to compile the target binary. The proposed fault operators emulate valid faults in terms of programming language' that is, changed code reproduces high-level faults that are syntactically correct.

Cotroneo et al. [2012] conducted a throughout analysis of the accuracy that can be achieved by G-SWFIT by analyzing its use in an embedded system from the space domain. In that experiment, faults were injected in both OS and application binary code, and binary code changes were compared to changes performed on the source code by following in both cases the fault model of G-SWFIT. Several cases were found in which G-SWFIT did not correctly mutate the binary code: (1) *omitted injections*, that is, potential injections that were missed at the binary level and were only performed at the source code level, and (2) *spurious injections*, that is, binary-level injections that do not match any valid source code injection. A detailed analysis identified which incorrect injections were due to intrinsic limitations of G-SWFIT and which ones were due to implementation issues of the fault injection tool. Many omitted and spurious injections were caused by limitations of G-SWFIT that are impossible or very difficult to avoid, including the identification of some high-level C operators and constructs, and the identification of inline C functions and macros, which are replicated at each call site in the program and can mislead G-SWFIT to inject several distinct faults for each replica (instead of only one fault affecting all the replicas at the same time, as happens in the case of a real fault in an inline C function). Nevertheless, the study concluded that G-SWFIT can achieve a good degree of accuracy if its fault operators are correctly implemented, even in the presence of these intrinsic limitations. As for

the definition of fault types, the problem of injecting in binary code needs to be further explored to draw general conclusions. In particular, it is necessary to consider systems and programming languages (other than C) with a higher level of abstraction, where the gap between the binary and source code is wider.

7.3. Efficiency of Code Change Injection

The efficiency of fault injection through code changes is mainly affected by two factors. The first factor is the *high number of code changes* that can be typically injected in a software system. Injecting a fault type in every location in which it can be applied (e.g., Assignment faults can potentially be injected at every assignment in the program) dramatically increases the number of injections, and thus the cost of the campaign. If the size of software reaches thousands or millions of lines of code, tens of thousands of faults can be potentially injected, making impractical the fault injection campaign. The second factor is the injection of *code changes that do not impact on the program execution*, and thus do not test fault tolerance properties. In fact, injected faults do not influence the program execution until the software is exercised with inputs that trigger the faulty code. In some cases, even if an injected fault is activated and turns into an error state, it may not propagate to the program outputs, since the error could be not influential on them or masked by "accidental" redundancy in the program (e.g., a corrupted variable may be overwritten before use, avoiding a failure [Nagarajan et al. 2009]). These problems also affect mutation testing, as discussed in Section 9.

Fault sampling approaches have been proposed to deal with the high number of code changes and to increase the likelihood of fault activation. These approaches select a subset of faults (among all faults that could potentially be injected) in order to improve efficiency while achieving representativeness at the same time. The most basic one is sampling based on code coverage (i.e., injecting in code locations that are actually covered by the workload). Giuffrida et al. [2013] and van der Kouwe et al. [2014] observed that the amount and type of fault locations are significantly impacted by the workload, and that the selection of fault locations should be tuned according to a workload's code coverage, in order to avoid injecting code changes that will never be executed and to maintain a desired proportion between fault types (such as the one in Table XI). Additional sampling techniques that use software complexity metrics [Natella et al. 2013] are discussed in Section 7.4.2. It should be noted that considering solely code coverage cannot ensure the activation of code changes. An experiment on fault injection in C library components [Lanzaro et al. 2014] showed that a significant part of injections, even if selected after a preliminary profiling of code coverage, does not surface at components' interfaces, ranging from 47.9% to 72.4% of injections without noticeable failures or data corruptions.

7.4. Representativeness of Code Change Injection

7.4.1. The Need for Code Changes More Complex Than SWIFI. The empirical study by Madeira et al. [2000] investigated the feasibility of injecting code changes using a traditional SWIFI tool but pointed out that more sophisticated approaches were necessary for injecting representative code changes. The study applied SWIFI on the binary code of programs by changing individual bits or bytes of operands and opcodes. It analyzed a set of programs developed during an international programming contest and that were considered correct by the contest judges. The authors identified a set of residual software faults by thoroughly testing these programs. They then tried to inject these faults by using the Xception SWIFI tool on the PowerPC 601 hardware architecture. The experiment highlighted that code changes produced by SWIFI can emulate software faults only to a limited extent. SWIFI was able to correctly emulate only the Checking and Assignment ODC defect types, as the injected code was close to

the intended fault type. However, SWIFI was not suitable for injecting more complex fault types that involve several statements, including the Algorithm, Interface, Function, and Timing ODC defect types, and in some cases the Assignment type (e.g., the initialization of complex data structures). To inject these types of faults, SWIFI tools would require more breakpoint registers than those typically provided by CPUs or too many software traps, incurring high intrusiveness.

7.4.2. Injection of Code Changes Based on Software Complexity Metrics. The original proposal of G-SWFIT focused on the fault types (*what to inject*) but did not study in depth the fault locations (*where to inject*). Given a fault type, G-SWFIT injects it in every location in which it can be applied, without accounting for the realism of fault in a given context. In fact, the location of a fault depends not only on the presence of a given syntactic construct but also on several other factors: in particular, the complexity of code surrounding the fault location (i.e., its module or function) and testing efforts that have been spent on that part of code [Fenton and Ohlsson 2000; Ostrand et al. 2005]. Not considering these factors (i.e., injecting in every location) not only reduces the efficiency of SFI but also leads to injections that are "out of context" and not realistic, thus reducing the representativeness of the faultload.

This issue has been investigated in Natella et al. [2013]. G-SWFIT was applied on three real-world complex software projects (two DBMSs, MySQL and PostgreSQL, and an RTOS, RTEMS) by extensively injecting thousands of faults in every potential location. This study investigated whether these faults could be considered representative of real faults. It started from the observation that real faults are subtle enough to escape testing and to affect software after its release. Therefore, the study ran real test cases on the injected software, analyzing whether the faults were detected by test cases. The study found that, for the DBMSs (two large and less tested systems), the majority of injected faults is representative (more than 50% of test cases do not detect them) but still exhibit a remarkable share of nonrepresentative faults (15%–23%); for the RTOS (a small and well-tested system), most of the injected faults (72%) were nonrepresentative. The study also observed that most of the representative fault locations are in a subset of components (files or functions) and proposed the use of software complexity metrics (number of lines of code, fan-in and fan-out) and machine-learning algorithms (decision trees and k-means clustering) to automatically identify these components using complexity metrics. This approach was able to improve the percentage of representative faults up to 26.08% and, at the same time, significantly reduce the faultload size by filtering out up to 69.43% of faults.

8. DISCUSSION

In the previous sections, we analyzed several fault injection techniques and tools, by emphasizing their key innovations and their limitations, and relevant experimental studies on representativeness, usability, and efficiency of fault injection. On the basis of this analysis, hereafter we compare existing techniques and tools, summarize the lessons learned, and identify promising directions for future research.

Recent standardization and research initiatives [Patterson et al. 2002; NASA 2004; AMBER project 2009; ISO 2011; Microsoft Corp. 2014] show an increased interest in the practical use of fault injection in real-world systems. According to the most popular techniques and tools, it is advisable to follow these strategies in the design of fault injectors:

—Tools should take advantage of facilities provided by the underlying hardware and OS technologies for debugging and monitoring to reduce implementation efforts and to achieve low intrusiveness and portability to new target systems. SWIFI tools, such as FERRARI, Xception, and GOOFI (Section 5), leveraged, respectively, UNIX systems calls, CPU debug registers and counters, and scan-chain test circuitry to corrupt

data with high portability and low intrusiveness. Interface error injection tools, such as LFI, Jaca, and PreFail (Section 6), leverage library interposition, reflection, and aspect-oriented programming to dynamically adapt error injection to the interfaces of the software under test.

—Tools should embed realistic fault/error models, since in most cases users do not have enough information about faults affecting their systems. G-SWFIT (Section 7.1.4) represented a significant advance in the applicability of fault injection, since it provided a generic fault model that covers a large share of faults and, at the same time, is suitable for different target systems. LFI (Section 6.3.2) also provided a solution in this sense by deriving error codes that can be returned by a library through the automated analysis of its binary code, without requiring the user to define error codes.

—Tools need to be modular and to provide abstractions to their users. For instance, GOOFI and NFTAPE (Section 5.3) provide a modular architecture that decouples the fault injector, the definition of fault models and of workloads, and the target system, thus enabling the user to adapt and to extend the tool. Other examples are PreFail and Destini (Section 6.3.3): the former provides rich information about injection points (e.g., node or resource being accessed, and current protocol state of the program), thus allowing a flexible definition of fault triggering criteria; the latter provides a language to specify fault tolerance properties to be checked during experiments.

Another important requirement is the ability to deal with large-scale systems by maximizing the efficiency of fault injection. The most useful strategies to deal with this problem were:

—To identify the most vulnerable areas of a target system and focus injections there. For instance, AFEX (Section 6.2.4) gives priority to error injections on specific API calls and error codes by rewarding those ones that revealed robustness issues in previous injections. PreFail (Section 6.2.5) adopts optimizations to inject errors that increase the coverage of recovery code or to satisfy user-defined criteria (such as to focus the injection of network partitions between processes that actually reside on distinct nodes).

—To adopt a mix of several error models, since different error models tend to be complementary and to discover different types of vulnerabilities. Given a time budget for experiments, it is more efficient to sample injections from several error models than to sample from only one specific error model (Section 6.2.2).

—To leverage, where available, models and specifications of inputs and states of the system, either provided by the user or reversed from static or dynamic analysis of the system. Examples are RIDDLE (Section 6.2.1), which adopted a grammar specification to generate more thorough random inputs, and the approaches that used state models to control the timing of injections (Section 6.2.3). Tuning the type and timing of injections through abstract models of the system is a good tradeoff between black-box approaches (which perform purely random experiments, which are cheaper but less effective) and white-box ones based on full specifications and visibility of system internals (which are exhaustive but costly and, in some cases, not applicable), and proved to be able to dramatically increase the efficiency.

The experimental studies on the efficiency and on the representativeness of fault/error models are a valuable resource for both researchers and prospective users. As a side note, we point out that building a solid, comprehensive body of knowledge on faults/errors is an important goal in the field of Software Fault Injection. In fact, empirical evidence still appears to be sparse, since previous studies only apply to specific systems or specific fault/error injection approaches (see Section 6.2.5). Our recommendations for future experimental studies (including research on new fault/error models,

Table XIII. Summary of Fault Injection Techniques and Tools

Approach and Emulated Faults	Fault/Error Model	Tools	Applications: Fault Removal	Applications: Fault Forecasting	Applications: Dependability Benchmarks	Domains: End-User	Domains: Server	Domains: Embedded
Data errors *(Hardware faults)*	Bit-flips	FIAT, FTAPE, GOOFI	✓	✓	✓			✓
	Packet corruption	DOCTOR, ORCHESTRA						
Interface errors *(Hardware, software, environment faults)*	API input errors	Fuzz, Ballista, RIDDLE	✓			✓	✓	✓
	API output errors	FIG, LFI, PreFail						
Code changes *(Software faults)*	Mutation operators	FAUST		✓	✓		✓	✓
	Field failure data	FINE, G-SWFIT						

as discussed in Section 10.3) are to carefully analyze and deal with threats to external validity, to make efforts to achieve a large degree of generality by considering several target systems of a different nature and several injection approaches, and to address the gaps not explored by previous experimental research.

To select a fault injection technique or tool among the several existing ones, users need to consider the goals of fault injection (fault removal, fault forecasting, dependability benchmarking), the types of faults that are emulated (hardware, software, environmental), and the underlying fault/error models, and the applications and domains most suitable for each of them.

Table XIII outlines the three surveyed injection approaches (injection of data errors, interface errors, and code changes) with respect to the following aspects. The first is the type of fault that they can emulate. On the one hand, data errors (e.g., bit-flips) can realistically emulate corruptions that are caused by hardware faults (e.g., CPU, memory, bus faults), but using them to emulate software faults is quite difficult [Christmansson and Chillarege 1996]. On the other hand, the injection of code changes is the approach that most closely emulates software faults by mimicking the most common types of bugs on the basis of field failure data [Durães and Madeira 2006]. Interface errors are not restricted to a specific type of fault, as hardware, software, and environment faults affecting a component can turn into errors at its interfaces. Since interface error injection abstracts from the low-level causes of component failures, it is also more intuitive and easier to understand and to accept by developers. Nevertheless, interface error injection cannot be considered a replacement for data errors and code changes: as pointed out in the previous discussion, not every software fault can be mapped to error codes at the interfaces of a component [Jarboui et al. 2002; Moraes et al. 2006].

For these reasons, when a high degree of representativeness is required, data errors and/or code changes should be adopted in addition to interface errors. This requirement is relevant for *fault forecasting* and for *dependability benchmarking*: experiments need to realistically reproduce the conditions in which the system will operate, including the faultload, in order to obtain valid and meaningful evaluations in terms of performance and fault tolerance measures. This is difficult to achieve with interface error injection, since the *most likely* interface errors are often unknown. However, interface error injection is a very useful approach for testing error handling and recovery mechanisms (*fault removal from fault tolerance*), since in most systems fault tolerance is designed and implemented around error codes and exceptions that arise from components' interfaces. By injecting at the interfaces, this approach allows one to focus experiments on those error handlers, increasing the efficiency. In principle, code changes can also

Table XIV. Comparison of Mutation Testing and Software Fault Injection Concepts

Mutation Testing	Software Fault Injection
Test suite: A set of inputs and of oracles for determining whether a program complies with its specifications. In Mutation Testing, test suites are designed to "kill" (i.e., detect) as many mutants as possible in order to achieve a high fault detection ability.	**Workload:** A set of inputs aimed at exercising a target system during the injection of a software fault. Workloads can be designed either to stress specific parts of the system (e.g., *synthetic workloads*) or to exercise the whole system in a realistic way, reflecting its operational usage (e.g., *representative benchmarks*).
Equivalent mutants: Mutants that are functionally equivalent to the original software, and that cannot be killed.	**Dormant faults:** Injected faults that are not activated during an experiment.
Hard-to-kill mutants: Nonequivalent mutants that can be killed only by few, very specific test cases. These mutants are desirable, as they tend to improve the fault detection ability of test suites.	**Residual faults:** Software faults that escape testing and are shipped with a deployed system. Software Fault Injection aims at emulating these kinds of faults.
Mutation score: Represents the quality of a test suite. It is the percentage of nonequivalent mutants that are killed by tests.	**Coverage factor:** It is a measure of fault tolerance. It is the percentage of activated faults that are detected and/or tolerated at runtime by a system, or by a specific mechanism or algorithm.

be used for fault removal, but they lend themselves to statistical evaluations rather than to test error handlers at specific code locations and under specific error conditions. Finally, data error injection is a flexible approach that can serve both statistical evaluations (through extensive data error injections) and the testing of error detectors (through injections focused on them), but its scope is limited to hardware faults due to the relative simplicity of its error model.

The differences among fault injection approaches also impact on their domain of applicability. Data error injection focuses on CPU, memory, and bus faults, which are often experienced in harsh environments, such as in transportation and aerospace contexts, where the software is *embedded* in a cyber-physical system. However, the injection of data errors typically requires a high effort in terms of number and duration of experiments, which can be justified only in the case of large-scale systems such as data centers, where the high number and density of hardware components increase the likelihood of low-level hardware faults (Section 10.2.2). End-user and server applications are mostly concerned about faults from the environment and from hardware equipment, such as disks and networks. For this domain, interface error injection is thus the most useful technique to improve error handling mechanisms. Moreover, since server systems make significant use of COTS software and have stringent requirements on service continuity and quality, they also represent a relevant domain of application for dependability benchmarks based on the injection of code changes, as shown by the benchmarks for web server and DBMS systems [Kanoun and Spainhower 2008].

9. THE RELATIONSHIP BETWEEN SOFTWARE FAULT INJECTION AND MUTATION TESTING

Mutation Testing and Software Fault Injection are two V&V approaches with evident overlap, since they both involve the injection of faults in the program code. For this reason, the two approaches share similar concepts, issues, and techniques. Table XIV summarizes the key concepts of these two techniques in order to point out their similarities and differences.

The goal of Mutation Testing is to improve the ability of test cases to detect faults [Hamlet 1977; DeMillo et al. 1978; King and Offutt 1991], whereas Software Fault Injection is used after other testing and verification activities (including Mutation Testing) to evaluate the effectiveness of fault tolerance algorithms and mechanisms [Arlat et al. 1990; Voas et al. 1997; Christmansson and Chillarege 1996; Durães and

Madeira 2006]. Mutation Testing evaluates the effectiveness of test cases (namely, the *mutation adequacy score*) by executing tests on faulty versions (*mutants*) of the program. Mutants can be *hand-seeded* or generated by a set of *mutation operators* (i.e., rules followed for introducing changes in the code). The effectiveness of test cases is evaluated with respect to *killed* mutants (i.e., the output of the mutant differs from the original program for at least one test case). Thus, test cases are defined such that they kill as many of the mutants as possible. This approach is based on the *coupling hypothesis* [DeMillo et al. 1978]: test cases that are effective against mutants are also effective against real faults. The usefulness of mutants in this sense has been confirmed by empirical studies [Andrews et al. 2005; Do and Rothermel 2006].

The high number of mutants produced by mutation operators has long been recognized as a crucial problem of Mutation Testing [Jia and Harman 2011]. Ideally, Mutation Testing should focus on *hard-to-kill* mutants (Table XIV), since these mutants drive testing toward high-quality test cases and avoid "trivial" mutants that only inflate the number of test executions. To solve this problem, researchers have been looking for a *sufficient* set of mutation operators, that is, mutant operators able to produce a low the number of mutants while achieving a high test effectiveness [Wong and Mathur 1995; Offutt et al. 1996; Sridharan and Namin 2010], and at mutant sampling and clustering strategies [Jia and Harman 2011].

As mentioned, Software Fault Injection is used to evaluate the effectiveness of fault tolerance and assist in its validation (e.g., in terms of *coverage factors*). Since SFI is concerned with the fault-tolerant behavior of a system during operation, the experiments closely emulate the real operational usage of the target system. Therefore, a first difference between SFI and MT lies in the inputs used for exercising software: instead of a test suite, SFI uses a workload representative of operational usage. Moreover, to closely emulate failure scenarios that typically occur in operation, the injected faults are carefully selected to be representative of *residual faults* that affect the system in the field (i.e., faults that escape testing and are shipped with the deployed product), by tuning *what* and *where* to inject on the basis of field failure data. Compared to mutation operators proposed in the literature, the fault operators used in Software Fault Injection are more selective, focusing on fault types found in the field. For instance, G-SWFIT provides 13 fault types for the C language (Table XII) against 71 mutation operators proposed in Delamaro and Maldonado [1996], and its fault types are more specific than typical mutation operators, since they include detailed rules to select specific portions of the software in which to inject. This difference reflects the fact that mutation operators are meant to obtain thorough test cases. Conversely, fault operators in SFI are meant to produce faulty conditions for assessing fault tolerance by emulating postrelease faults that escape testing.

Another similarity between the two approaches is related to faults that do not lead to any software failure. They are referred to as *equivalent mutants* in Mutation Testing and as *dormant faults* in Software Fault Injection. The presence of these faults affects the efficiency of experiments for both approaches. But even in this case, there are important differences in dealing with this issue. Equivalent (i.e., nonkillable) mutants in MT waste developer efforts: these mutants have to be individually investigated by developers to determine whether the mutant is equivalent, and to augment the test suite only if this is not the case. Techniques for detecting equivalent mutants (based on code optimization, constraint solving, and program slicing [Madeyski et al. 2014]) are computationally expensive, and they cannot detect all equivalent mutants since proving program equivalence is an undecidable problem. Dormant faults represent a source of inefficiency for SFI experiments, since they do not produce any error for assessing dependability. To compensate for dormant faults, developers may need to improve the workload or to perform additional experiments by injecting a new set of faults. The

injection of data errors, instead of code changes, mitigates the problem of dormant faults [Christmansson and Chillarege 1996], but they are less representative of software faults. Techniques have been proposed for further increasing the propagation of data errors by exercising the system with stressful workloads [Tsai et al. 1999] and by performing a preinjection analysis of register and memory usage, in order to inject errors that are more likely to be accessed [Barbosa et al. 2005].

Finally, both Software Fault Injection and Mutation Testing researchers have been investigating the injection of multiple faults. In the field of Mutation Testing, the combined injection of several simple mutants is referred to as *higher-order mutation testing* [Jia and Harman 2009]. The potential benefits of this approach are threefold: (1) higher-order mutants can be harder to kill than their constituent mutants, since constituent mutants can partially mask each other and, in some cases, their combination requires test cases able to cover new corner cases; (2) using higher-order mutants can reduce the number of mutants and thus the testing effort, since several simple mutants can be replaced by only one higher-order mutant that combines them; and (3) higher-order mutants are much less likely to be equivalent mutants than simple mutants, as shown by several empirical studies [Offutt 1992; Madeyski et al. 2014; Papadakis and Malevris 2010]. The main issue of higher-order mutation testing is represented by the combinatorial explosion of injectable mutants [Jia and Harman 2009; Madeyski et al. 2014]. Researchers on fault-tolerant computing have also been investigating the use of multiple fault injections. However, as discussed in Section 6.2.5, the perspective of fault injection is not to improve test suites, but to assess critical systems against multiple fault patterns.

10. CONCLUDING REMARKS, OPEN ISSUES, AND DIRECTIONS FOR FUTURE RESEARCH

Software faults represent a severe problem for current and future systems, given their ever-increasing complexity of systems, the adoption of software in new domains, and the massive adoption of COTS software. In this survey, we analyzed several techniques and tools for the injection of software faults, as well as their practical applications. While these approaches have significantly evolved, several challenging issues still remain open, and recent technological developments have worsened them and created new ones. In this section, we present a condensed view on the current open issues and future research directions in Software Fault Injection.

10.1. Usability and Dependability Benchmarking

Usability remains an important point for fault injection in general, and it is even more challenging when considering the injection of software faults. Practitioners often find fault injection techniques hard and difficult to apply. The higher complexity of software fault models, when compared to the simple hardware bit-flip faults, has worsened the long-lasting issue of usability. Usability issues, and their direct impact on the effort and cost needed to perform experiments, will be increasingly important. This is due to the extreme complexity, high dynamism, and large scale of modern computer systems.

Further improvements in usability of fault injection are necessary in order to assure the feasibility of dependability benchmarking and to exploit its potential as a market tool for vendors, as a purchase decision helper, or as a tool to tune up system performance. Even if several case studies have appeared in previous studies [Kanoun and Spainhower 2008], industry hesitates to adopt dependability benchmarking. Many of the open challenges for dependability benchmarking were identified in the research roadmap [AMBER project 2009] proposed by the AMBER FP7 Coordination Action (*Assessing, Measuring, and Benchmarking Resilience*). It traced new research paths that include well-grounded foundations and methodologies for assessing dependability from a variety of diverse and uncertain evidences; the definition of holistic approaches

and measures encompassing both physical and design faults, security attacks, and human factors; and understanding the economics and overall impact of assessment and benchmarking on the life cycle of IT systems. Besides these methodological and process issues, improving the usability of fault injection is a major goal to make dependability benchmarking a viable approach for real software projects, by reducing the costs and the know-how required to perform dependability benchmarks. We believe that achieving *reusable* and *customizable* testbeds (including all software for performing and analyzing experiments) will be an important step to improve the usability by allowing users to easily reproduce dependability benchmarks. These testbeds could be tailored for specific domains (e.g., transaction processing, medical systems) and should allow users to introduce their own software and to tune the faultload and workload profiles to be adopted in the experiments in order to better reflect the foreseen conditions in which the system will operate.

10.2. The Challenge of Software Fault Tolerance

As software faults are recognized as a significant threat for modern systems and infrastructures, *tolerating software faults* remains an important but still challenging issue. While physical faults can be tolerated thanks to *redundant physical resources* that could replace failed ones, software faults (and, in general, design faults) affect the logic of a program, and thus they require some form of *logic redundancy* to be tolerated. Moreover, while physical faults can be easily anticipated and modeled, this is not the case of software faults. In fact, even if we know that software is defective, we never know exactly where defects are, when they will reveal, and which effects they will have. This uncertainty makes it difficult to devise appropriate strategies to counteract software faults at runtime. All these difficulties and challenges represent a huge opportunity for the use of Software Fault Injection. We do believe that practical and easy-to-use tools are essential for the future of software fault tolerance.

10.2.1. The Quest for New Software Fault Tolerance Methods. Several software fault tolerance approaches have been proposed, and most of them are surveyed in De Florio and Blondia [2008] and Lyu [1995]. They include masking software faults through diversity (e.g., recovery blocks, N-version programming); forcing a fail-stop behavior or a degraded mode of service when a wrong state of the system is detected (e.g., concurrent error detection, checkpointing and rollback, exception handling); and language structures that introduce fault tolerance aspects and semantics in a program (e.g., aspect-oriented programming techniques). More recent studies explored autonomous and self-healing architectures for adapting the behavior of a system in the presence of faults [Ganek and Corbi 2003; Gorla et al. 2012]. The actual application of these approaches is jeopardized by the increase in design and code complexity and by development and maintenance costs. Thus, achieving "affordable" software fault tolerance still represents a big challenge, especially for complex systems.

Exploring new ways to use SFI may contribute in this sense. Future uses of fault injection could support more thoroughly the design of fault tolerance. In fact, developers of FTAMs have to fully understand the errors that they need to tolerate, but this comprehension is very difficult to reach for software faults. Software Fault Injection, through the deliberate injection of realistic faults, provides a means to reason about the types of error caused by software faults in a given program. Then, error propagation analysis can help to develop effective error detectors and handlers. For example, fault injection can identify the most critical data and apply checkpoint-and-rollback strategies to them.

Fault injection can also be used to devise more effective *error detectors*, which are required for fault tolerance strategies such as recovery blocks (a secondary version is

executed when an *acceptance test* detects a failure of the primary version) and fail-stop software (errors must be promptly detected in order to stop a system before causing more severe errors). These strategies assume the availability of such error detectors (e.g., defined by developers from requirements and from high-level design), but there is no approach for their systematic development of software error detectors. The design and evaluation of such detectors could be supported by SFI at the early stages of software development.

10.2.2. Software Fault Tolerance Challenges Posed by the Forthcoming Hardware. The need to reduce energy consumption and to increase processing speeds are major problems for both large data centers and mobile devices. For this reason, the semiconductor manufacturing industry is committed to reduce further the transistors' size. The *International Technology Roadmap for Semiconductors* (full report available at Wilson [2013]) states that, in the near future, we will have a new generation of processors, with transistor geometries in the range of 10nm and below. But this comes with the high price of exposing the software to more and more soft faults.

In fact, the rate of so-called soft faults is destined to increase, as a result of the reduced sizes and levels of energy of future hardware. Even conservative estimations expect an increase of soft faults up to 100 times. Although soft faults can be emulated with traditional SWIFI tools, the key issue is that the forthcoming systems will rely on intensive software-implemented fault tolerance mechanisms (SIFTM) just to achieve the level of dependability of current systems. These SIFTMs need to be tested and validated, which means that SFI will play a crucial role in the development of the next generation of computer systems. This is especially true for cloud infrastructures, since they will be the first targets for the forthcoming processors with transistor sizes less than 10nm.

10.2.3. Online Failure Prediction. The prediction of failures to increase the availability of computer systems is an old and fascinating challenge. The prediction of failures, even with just a few seconds in advance, is crucial to trigger corrective actions. Failure prediction is also a key element to reduce maintenance costs, as both reactive and preventive maintenance policies can be optimized taking into account inputs from failure prediction. Unfortunately, existing failure prediction models and techniques are still very far from providing satisfactory results (see the survey of Salfner et al. [2010]). The fact that real failures are relatively rare events makes it difficult to define, validate, and tune up failure prediction models.

The use of fault injection to produce rich failure data is a great opportunity to improve failure prediction mechanisms. Even long-lasting research efforts such as the IBM model for autonomic computing (MAPE-K [Ganek and Corbi 2003]) could benefit from the use of realistic failure data artificially created by fault injection. In fact, it is commonly accepted that the big hurdle toward reliable failure prediction lies exactly in the lack of data to select and to train prediction models. SFI can solve this problem by generating large quantities of realistic failure data through experiments [Vieira et al. 2009]. This rich failure data can be used to identify the symptoms with the best predictive power and to build, train, and validate failure prediction models.

10.3. Evolution of Fault Models

The representativeness of fault models will need to keep up with the ever-changing software technologies and development paradigms. The first notable paradigm to consider is *model-driven engineering* (MDE), which today represents an important solution for safety-critical systems. MDE approaches automatically generate part of the software from design artifacts, and thus they avoid bugs that were previously caused by manual coding. However, these artifacts are exposed to new kinds of design faults arising

from other phases of development, such as requirement analysis and high-level design. Thus, understanding these faults and including them in fault injection will represent a significant research area. This will ask one to carefully characterize design faults, to understand how they turn into faults in the software, and which "coding" bugs could occur in those parts of the code that still need to be manually coded.

Fault injection has also the potential to be used in *software requirements* during the early phases of development, to analyze the impact of *requirement faults* [Leveson 2004]. The emulation of faults in requirement documents consists of introducing, removing, or modifying requirements in order to emulate the presence of incomplete, contradicting, or incorrect requirements. This procedure can be useful to provide feedback to the *requirement review process*: the effectiveness of reviews can be evaluated by asking reviewers to inspect a document with injected faults. The field data study of requirement faults in space software in Véras et al. [2012] provides a basis for the definition of realistic fault types and their automated injection.

Another concern for the future of fault injection is to keep fault models updated with technology and with the faults that will be actually experienced by the next generation of systems. We are building, in fact, more distributed and concurrent systems, which will leverage new technologies such as cloud computing, service-oriented computing, multicore systems, and machine learning. New proposals have been recently made to reflect these technological trends: they consider the *injection of multiple faults*, such as multiple data corruptions, multiple I/O errors, and multiple crashes. Nevertheless, the use of multiple injections is still at an early stage. The most important issue when testing against multiple failures is the *combinatorial explosion* of multiple failures that can be exercised, though not every combination is necessarily realistic. While these fault models are accepted for modeling hardware faults [Li et al. 2010], there is still a lack of characterization of software faults in large-scale distributed software.

Finally, future research still needs to explore possible interactions between Software Fault Injection and *software security*. In this context, SFI can be adopted to emulate *security vulnerabilities*, such as software defects that may cause unauthorized accesses to confidential data [Avizienis et al. 2004]. The injection of vulnerabilities would allow one to evaluate the effectiveness of *Intrusion Detection Systems* (IDSs) and *vulnerability scanners*, and to *train security assurance teams* that are responsible for code inspection and penetration testing. Given the high exposure of software to security attacks, the ability to emulate software vulnerability would contribute to devise better countermeasures. A preliminary result in this sense is the field data study in Fonseca and Vieira [2008], which analyzed the types of software defects that are related to two common software vulnerabilities in web applications (*SQL injection* and *Cross Site Scripting* vulnerabilities), finding that most of these software defects consist of missing code for validating and sanitizing input variables; this finding was later adopted in an automated tool for vulnerability injection in web applications [Fonseca et al. 2009]. In the future, new fault models and fault injection techniques will be needed because of the continuous rise of new security attacks.

REFERENCES

J. Aidemark, J. Vinter, P. Folkesson, and J. Karlsson. 2001. GOOFI: Generic object-oriented fault injection tool. In *Proc. IEEE/IFIP Intl. Conf. Dependable Systems and Networks*. 83–88.

A. Albinet, J. Arlat, and J. C. Fabre. 2004. Characterization of the impact of faulty drivers on the robustness of the linux kernel. In *Proc. IEEE/IFIP Intl. Conf. Dependable Systems and Networks*. 867–876.

AMBER project. 2009. *AMBER Final Research Roadmap*. Retrieved from http://www.amber-project.eu/.

J. H. Andrews, L. C. Briand, and Y. Labiche. 2005. Is mutation an appropriate tool for testing experiments? In *Proc. Intl. Conf. on Software Engineering*. 402–411.

J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J. C. Fabre, J. C. Laprie, E. Martins, and D. Powell. 1990. Fault injection for dependability validation: A methodology and some applications. *IEEE Trans. Software Eng.* 16, 2 (1990), 166–182.

J. Arlat, Y. Crouzet, J. Karlsson, P. Folkesson, E. Fuchs, and G. H. Leber. 2003. Comparison of physical and software-implemented fault injection techniques. *IEEE Trans. Comput.* 52, 9 (2003), 1115–1133.

J. Arlat, J. C. Fabre, M. Rodríguez, and F. Salles. 2002. Dependability of COTS microkernel-based systems. *IEEE Trans. Comput.* 51, 2 (2002), 138–163.

J. Arlat and R. Moraes. 2011. Collecting, analyzing and archiving results from fault injection experiments. In *Proc. Latin-American Symposium on Dependable Computing*. 100–105.

A. Avizienis, J. C. Laprie, B. Randell, and C. Landwehr. 2004. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. on Dependable and Secure Computing* 1, 1 (2004), 11–33.

D. Avresky, J. Arlat, J. C. Laprie, and Y. Crouzet. 1996. Fault injection for formal testing of fault tolerance. *IEEE Trans. on Reliability* 45, 3 (1996), 443–455.

R. Banabic and G. Candea. 2012. Fast black-box testing of system recovery code. In *Proc. ACM European Conference on Computer Systems*. 281–294.

R. Barbosa, J. Vinter, P. Folkesson, and J. Karlsson. 2005. Assembly-level pre-injection analysis for improving fault injection efficiency. In *Proc. European Dependable Computing Conf.* 246–262.

J. H. Barton, E. W. Czeck, Z. Z. Segall, and D. P. Siewiorek. 1990. Fault injection experiments using FIAT. *IEEE Trans. Comput.* 39, 4 (1990), 575–582.

T. Basso, R. Moraes, B. P. Sanches, and M. Jino. 2009. An investigation of java faults operators derived from a field data study on Java software faults. In *Workshop de Testes e Tolerância a Falhas*.

A. Bondavalli, S. Chiaradonna, D. Cotroneo, and L. Romano. 2004. Effective fault treatment for improving the dependability of COTS and legacy-based applications. *IEEE Trans. Dependable Secure Comput.* 1, 4 (2004), 223–237.

E. Bounimova, P. Godefroid, and D. Molnar. 2013. Billions and billions of constraints: Whitebox fuzz testing in production. In *Proc. Intl. Conf. on Software Engineering*. 122–131.

P. Broadwell, N. Sastry, and J. Traupman. 2002. FIG: A prototype tool for online verification of recovery mechanisms. In *Workshop on Self-Healing, Adaptive and self-MANaged Systems*.

G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. 2004. Microreboot–A technique for cheap recovery. In *Proc. Symp. on Operating Systems Design and Implementation*.

J. V. Carreira, D. Costa, and J. G. Silva. 1999. Fault injection spot-checks computer system dependability. *IEEE Spectrum* 36, 8 (1999), 50–55.

J. Carreira, H. Madeira, and J. G. Silva. 1998. Xception: A technique for the experimental evaluation of dependability in modern computers. *IEEE Trans. Software Eng.* 24, 2 (1998), 125–136.

R. Chandra, R. M. Lefever, K. R. Joshi, M. Cukier, and W. H. Sanders. 2004. A global-state-triggered fault injector for distributed system evaluation. *IEEE Trans. Parallel Distrib. Syst.* 15, 7 (2004), 593–605.

S. Chandra and P. M. Chen. 1998. How fail-stop are faulty programs? In *Digest of Papers, Intl. Symp. on Fault-Tolerant Computing*. 240–249.

R. Chillarege, I. S. Bhandari, J. K. Chaar, M. J. Halliday, D. S. Moebus, B. K. Ray, and M. Y. Wong. 1992. Orthogonal defect classification–A concept for in-process measurements. *IEEE Trans. Software Eng.* 18, 11 (1992), 943–956.

R. Chillarege, W. L. Kao, and R. G. Condit. 1991. Defect type and its impact on the growth curve. In *Proc. Intl. Conf. on Software Engineering*. 246–255.

J. Christmansson and R. Chillarege. 1996. Generation of an error set that emulates software faults based on field data. In *Digest of Papers, Intl. Symp. on Fault-Tolerant Computing*. 304–313.

J. Christmansson, M. Hiller, and M. Rimen. 1998. An experimental comparison of fault and error injection. In *Proc. IEEE Intl. Symp. on Software Reliability Engineering*. 369–378.

J. Christmansson and P. Santhanam. 1996. Error injection aimed at fault removal in fault tolerance mechanisms–Criteria for error selection using field data on software faults. In *Proc. IEEE Intl. Symp. on Software Reliability Engineering*. 175–184.

J. A. Clark and D. K. Pradhan. 1995. Fault injection: A method for validating computer-system dependability. *IEEE Computer* 28, 6 (1995), 47–56.

D. Cotroneo, D. Di Leo, F. Fucci, and R. Natella. 2013. SABRINE: State-based robustness testing of operating systems. In *Proc. IEEE/ACM Intl. Conf. on Automated Software Engineering*. 125–135.

D. Cotroneo, A. Lanzaro, R. Natella, and R. Barbosa. 2012. Experimental analysis of binary-level software fault injection in complex software. In *Proc. European Dependable Computing Conf.* 162–172.

D. Cotroneo, R. Natella, S. Russo, and F. Scippacercola. 2013. State-driven testing of distributed systems. In *Proc. Intl. Conf. Principles of Distributed Systems*. 114–128.

M. Daran and P. Thévenod-Fosse. 1996. Software error analysis: A real case study involving real faults and mutations. *ACM Software Engineering Notes* 21, 3 (1996), 158–171.

S. Dawson, F. Jahanian, T. Mitton, and T. L. Tung. 1996. Testing of fault-tolerant and real-time distributed systems via protocol fault injection. In *Digest of Papers, Intl. Symp. on Fault-Tolerant Computing*. 404–414.

DBench project. 2004. *DBench Final Report*. Retrieved from http://www.laas.fr/DBench/.

V. De Florio and C. Blondia. 2008. A survey of linguistic structures for application-level fault tolerance. *Comput. Surveys* 40, 2 (2008), 6.

M. E. Delamaro and J. C. Maldonado. 1996. Proteum—A tool for the assessment of test adequacy for c programs. In *Proc. Conf. Performability in Computer Systems*. 79–95.

R. A. DeMillo, R. J. Lipton, and F. G. Sayward. 1978. Hints on test data selection: Help for the practicing programmer. *IEEE Computer* 11, 4 (1978), 34–41.

C. P. Dingman and J. Marshall. 1995. Measuring robustness of a fault-tolerant aerospace system. In *Digest of Papers, Intl. Symp. on Fault-Tolerant Computing*. 522–527.

H. Do and G. Rothermel. 2006. On the use of mutation faults in empirical assessments of test case prioritization techniques. *IEEE Trans. Software Eng.* (2006), 733–752.

J. Duraes and H. Madeira. 2002. Emulation of software faults by educated mutations at machine-code level. In *Proc. IEEE Intl. Symp. on Software Reliability Engineering*. 329–340.

J. Durães and H. Madeira. 2006. Emulation of software faults: A field data study and a practical approach. *IEEE Trans. Software Eng. 32*, 11 (2006), 849–867.

J. Durães, M. Vieira, and H. Madeira. 2003. Multidimensional characterization of the impact of faulty drivers on the operating systems behavior. *IEICE Trans. Inf. Sys.* 86, 12 (2003), 2563–2570.

J. Durães, M. Vieira, and H. Madeira. 2004. Dependability benchmarking of Web-servers. In *Proc. Intl. Conf. on Computer Safety, Reliability, and Security*. 297–310.

N. E. Fenton and N. Ohlsson. 2000. Quantitative analysis of faults and failures in a complex software system. *IEEE Trans. Software Eng.* 26, 8 (2000), 797–814.

C. Fetzer, P. Felber, and K. Högstedt. 2004. Automatic detection and masking of nonatomic exception handling. *IEEE Trans. Software Eng.* 30 (2004), 547–560. Issue 8.

C. Fetzer and Z. Xiao. 2002. An automated approach to increasing the robustness of C libraries. In *Proc. IEEE/IFIP Intl. Conf. Dependable Systems and Networks*. 155–164.

J. Fonseca and M. Vieira. 2008. Mapping software faults with web security vulnerabilities. In *Proc. IEEE/IFIP Intl. Conf. Dependable Systems and Networks*. 257–266.

J. Fonseca, M. Vieira, and H. Madeira. 2009. Vulnerability & attack injection for Web applications. In *Proc. IEEE/IFIP Intl. Conf. Dependable Systems and Networks*. 93–102.

A. G. Ganek and T. A. Corbi. 2003. The dawning of the autonomic computing era. *IBM Syst. J.* 42, 1 (2003), 5–18.

A. K. Ghosh, M. Schmid, and V. Shah. 1998. Testing the robustness of windows NT software. In *Proc. IEEE Intl. Symp. on Software Reliability Engineering*. 231–235.

C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum. 2013. EDFI: A dependable fault injection tool for dependability benchmarking experiments. In *Proc. Pacific Rim Intl. Symp. on Dependable Computing*.

P. Godefroid, M. Y. Levin, and D. A. Molnar. 2008. Automated whitebox fuzz testing. In *Proc. Network and Distributed Sys. Sec. Symp.* 151–166.

A. Gorla, M. Pezzè, J. Wuttke, L. Mariani, and F. Pastore. 2012. Achieving cost-effective software reliability through self-healing. *Comput. Inf.* 29, 1 (2012), 93–115.

J. Gray. 1990. A census of tandem system availability between 1985 and 1990. *IEEE Trans. on Reliability* 39, 4 (1990), 409–418.

H. S. Gunawi, T. Do, P. Joshi, P. Alvaro, J. M. Hellerstein, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, K. Sen, and D. Borthakur. 2011. FATE and DESTINI: A framework for cloud recovery testing. In *Proc. USENIX Symposium on Networked Systems Design and Implementation*.

R. G. Hamlet. 1977. Testing programs with the aid of a compiler. *IEEE Trans. Software Eng.* 3, 4 (1977), 279–290.

S. Han, K. G. Shin, and H. A. Rosenberg. 1995. DOCTOR: An integrated software fault injection environment. In *Proc. Intl. Computer Performance and Dependability Symp.* 204–213.

M. Hiller, A. Jhumka, and N. Suri. 2001. An approach for analysing the propagation of data errors in software. In *Proc. IEEE/IFIP Intl. Conf. Dependable Systems and Networks*. 161–170.

M. C. Hsueh, T. K. Tsai, and R. K. Iyer. 1997. Fault injection techniques and tools. *IEEE Computer* 30, 4 (1997), 75–82.

J. J. Hudak, B. H. Suh, D. P. Siewiorek, and Z. Segall. 1993. Evaluation and comparison of fault-tolerant software techniques. *IEEE Trans. Reliability* 42, 2 (1993), 190–204.

IEEE. 1990. IEEE standard glossary of software engineering terminology. *IEEE Std 610.12-1990* (1990).

IEEE. 1994. IEEE standard for information technology–Portable operating system interface (POSIX) part 1. *IEEE Std 1003.1b-1993* (1994).

ISO. 2011. Product development: software level. *ISO 26262: Road vehicles – Functional safety* 6 (2011).

T. Jarboui, J. Arlat, Y. Crouzet, K. Kanoun, and T. Marteau. 2002. Analysis of the effects of real and injected software faults: Linux as a case study. In *Proc. Pacific Rim Intl. Symp. on Dependable Computing*. 51–58.

Y. Jia and M. Harman. 2009. Higher order mutation testing. *Inf. Software Technol.* 51, 10 (2009), 1379–1393.

Y. Jia and M. Harman. 2011. An analysis and survey of the development of mutation testing. *IEEE Trans. Software Eng.* 37, 5 (2011), 649–678.

A. Johansson and N. Suri. 2005. Error propagation profiling of operating systems. In *Proc. IEEE/IFIP Intl. Conf. Dependable Systems and Networks*. 86–95.

A. Johansson, N. Suri, and B. Murphy. 2007a. On the impact of injection triggers for OS robustness evaluation. In *Proc. IEEE Intl. Symp. on Software Reliability Engineering*. 127–126.

A. Johansson, N. Suri, and B. Murphy. 2007b. On the selection of error model(s) for OS robustness evaluation. In *Proc. IEEE/IFIP Intl. Conf. Dependable Systems and Networks*. 502–511.

P. Joshi, H. S. Gunawi, and K. Sen. 2011. PREFAIL: A programmable tool for multiple-failure injection. *ACM SIGPLAN Not.* 46, 10 (2011), 171–188.

A. Kalakech, K. Kanoun, Y. Crouzet, and J. Arlat. 2004. Benchmarking the dependability of windows NT4, 2000 and XP. In *Proc. IEEE/IFIP Intl. Conf. Dependable Systems and Networks*. 681–686.

G. A. Kanawati, N. A. Kanawati, and J. A. Abraham. 1995. FERRARI: A flexible software-based fault and error injection system. *IEEE Trans. Comput.* 44, 2 (1995), 248–260.

K. Kanoun and L. Spainhower. 2008. *Dependability Benchmarking for Computer Systems*. Wiley-IEEE Computer Society.

W.-I. Kao and R. K. Iyer. 1994. DEFINE: A distributed fault injection and monitoring environment. In *Proc. Workshop on Fault-Tolerant Parallel and Distributed Systems*. 252–259.

W.-I. Kao, R. K. Iyer, and D. Tang. 1993. FINE: A fault injection and monitoring environment for tracing the UNIX system behavior under faults. *IEEE Trans. Software Eng.* 19, 11 (1993), 1105–1118.

J. Katcher. 1997. *Postmark: A New File System Benchmark*. Technical Report TR-3022.

L. Keller, P. Upadhyaya, and G. Candea. 2008. ConfErr: A tool for assessing resilience to human configuration errors. In *Proc. IEEE/IFIP Intl. Conf. Dependable Systems and Networks*. 157–166.

J. C. King. 1976. Symbolic execution and program testing. *Commun. ACM* 19, 7 (1976), 385–394.

K. N. King and A. J. Offutt. 1991. A fortran language system for mutation-based software testing. *Software: Practice Exp.* 21, 7 (1991), 685–718.

P. Koopman and J. DeVale. 2000. The exception handling effectiveness of POSIX operating systems. *IEEE Trans. Software Eng.* 26, 9 (2000), 837–848.

A. Lanzaro, R. Natella, S. Winter, D. Cotroneo, and N. Suri. 2014. An empirical study of injected versus actual interface errors. In *Proc. Intl. Symp. Soft. Testing and Analysis*. 397–408.

J.-C. Laprie, J. Arlat, C. Beounes, and K. Kanoun. 1990. Definition and analysis of hardware-and software-fault-tolerant architectures. *IEEE Computer* 23, 7 (1990), 39–51.

N. Laranjeiro, M. Vieira, and H. Madeira. 2014. A technique for deploying robust Web services. *IEEE Trans. Services Comput.* 7, 1 (2014), 68–81.

I. Lee and R. K. Iyer. 1995. Software dependability in the tandem guardian system. *IEEE Trans. Software Eng.* 21, 5 (1995), 455–467.

N. G. Leveson. 2004. Role of software in spacecraft accidents. *J. Spacecraft Rockets* 41, 4 (2004), 564–575.

X. Li, M. C. Huang, K. Shen, and L. Chu. 2010. A realistic evaluation of memory hardware errors and software system susceptibility. In *Proc. USENIX Annual Technical Conf.*

M. R. Lyu. 1995. *Software Fault Tolerance*. John Wiley & Sons.

H. Madeira, D. Costa, and M. Vieira. 2000. On the emulation of software faults by software fault injection. In *Proc. IEEE/IFIP Intl. Conf. Dependable Systems and Networks*. 417–426.

L. Madeyski, W. Orzeszyna, R. Torkar, and M. Józala. 2014. Overcoming the equivalent mutant problem: A systematic literature review and a comparative experiment of second order mutation. *IEEE Trans. Software Eng.* 40, 1 (2014), 23–42.

A. Mahmood, D. M. Andrews, and EJ McCluskey. 1984. Executable assertions and flight software. In *Proceedings of the 6th Digital Avionics Systems Conference*. 346–351.

P. D. Marinescu and G. Candea. 2011. Efficient testing of recovery code using fault injection. *ACM Trans. Comput. Syst.* 29, 4 (2011), 11:1–11:38.

E. Martins, C. M. F. Rubira, and N. G. M. Leme. 2002. Jaca: A reflective fault injection tool based on patterns. In *Proc. IEEE/IFIP Intl. Conf. Dependable Systems and Networks*. 483–487.

P. A. McQuaid. 2012. Software disasters—understanding the past, to improve the future. *J. Software: Evolution and Process* 24, 5 (2012), 459–470.

Microsoft Corp. 2014. *Resilience by Design for Cloud Services*. Retrieved from http://www.microsoft.com/en-us/download/details.aspx?id=38823.

B. P. Miller, L. Fredriksen, and B. So. 1990. An empirical study of the reliability of UNIX utilities. *Commun. ACM* 33, 12 (1990), 32–44.

B. Miller, D. Koski, C. Lee, V. Maganty, R. Murthy, A. Natarajan, and J. Steidl. 1998. *Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services*. Technical Report CSTR-95-1268.

R. Moraes, R. Barbosa, J. Durães, N. Mendes, E. Martins, and H. Madeira. 2006. Injection of faults at component interfaces and inside the component code: Are they equivalent? In *Proc. European Dependable Computing Conf.* 53–64.

V. Nagarajan, D. Jeffrey, and R. Gupta. 2009. Self-recovery in server programs. In *Proc. Intl. Symp. on Memory Management*. 49–58.

NASA. 2004. NASA software safety guidebook. *NASA-GB-8719.13* (2004).

R. Natella and D. Cotroneo. 2010. Emulation of transient software faults for dependability assessment: A case study. In *Proc. European Dependable Computing Conf.* 23–32.

R. Natella, D. Cotroneo, J. A. Duraes, and H. Madeira. 2013. On fault representativeness of software fault injection. *IEEE Trans. Software Eng.* 39, 1 (2013), 80–96.

W. T. Ng, C. M. Aycock, G. Rajamani, and P. M. Chen. 1996. Comparing disk and memory's resistance to operating system crashes. In *Proc. IEEE Intl. Symp. on Software Reliability Engineering*. 185–194.

W. T. Ng and P. M. Chen. 2001. The design and verification of the rio file cache. *IEEE Trans. Comput.* 50, 4 (2001), 322–337.

A. J. Offutt. 1992. Investigations of the software testing coupling effect. *ACM Trans. Software Eng Methodol.* 1, 1 (1992), 5–20.

A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf. 1996. An experimental determination of sufficient mutant operators. *ACM Trans. Software Eng Methodol.* 5, 2 (1996), 99–118.

D. Oppenheimer, A. Ganapathi, and D. A. Patterson. 2003. Why do internet services fail, and what can be done about it? In *USENIX Symp. on Internet Technologies and Systems*.

T. J. Ostrand, E. J. Weyuker, and R. M. Bell. 2005. Predicting the location and number of faults in large software systems. *IEEE Trans. Software Eng.* 31, 4 (2005), 340–355.

M. Papadakis and N. Malevris. 2010. An empirical evaluation of the first and second order mutation testing strategies. In *Proc. Intl. Conf. Software Testing, Verification, and Validation Workshops*. 90–99.

C. S. Păsăreanu and W. Visser. 2009. A survey of new trends in symbolic execution for software testing and analysis. *Intl. J. Software Tools Tech. Transf.* 11, 4 (2009), 339–353.

K. Pattabiraman, N. Nakka, Z. Kalbarczyk, and R. K. Iyer. 2008. SymPLFIED: symbolic program-level fault injection and error detection framework. In *Proc. IEEE/IFIP Intl. Conf. Dependable Systems and Networks*. 472–481.

D. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, and N. Treuhaft. 2002. *Recovery-Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies*. Technical Report TR-02-1175.

D. Powell, E. Martins, J. Arlat, and Y. Crouzet. 1995. Estimators for fault tolerance coverage evaluation. *IEEE Trans. Comput.* 44, 2 (1995), 261–274.

V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. 2005. Model-based failure analysis of journaling file systems. In *Proc. IEEE/IFIP Intl. Conf. Dependable Systems and Networks*. 802–811.

G. L. Ries, G. S. Choi, and R. K. Iyer. 1994. Device-level transient fault modeling. In *Digest of Papers, Intl. Symp. on Fault-Tolerant Computing*. 86–94.

RTCA. 1992. DO-178B software considerations in airborne systems and equipment certification. *Requirements and Technical Concepts for Aviation* (1992).

F. Salfner, M. Lenk, and M. Malek. 2010. A survey of online failure prediction methods. *Comput. Surveys* 42, 3 (2010), 10.

F. Salles, M. Rodriguez, J.-C. Fabre, and J. Arlat. 1999. MetaKernels and fault containment wrappers. In *Digest of Papers, Intl. Symp. on Fault-Tolerant Computing*. 22–29.

B. P. Sanches, T. Basso, and R. Moraes. 2011. J-SWFIT: A Java software fault injection tool. In *Proc. Latin American Symp. on Dependable Computing*.

A. Schiper, K. Birman, and P. Stephenson. 1991. Lightweight causal and atomic group multicast. *ACM Trans. Comput. Syst.* 9, 3 (1991), 272–314.

SPEC. 2000. *SPECweb99 v1.02*. Retrieved from http://www.spec.org/web99/.

M. Sridharan and A. S. Namin. 2010. Prioritizing mutation operators based on importance sampling. In *Proc. IEEE Intl. Symp. on Software Reliability Engineering*. 378–387.

D. T. Stott, B. Floering, Z. Kalbarczyk, and R. K. Iyer. 2000. A framework for assessing dependability in distributed systems with lightweight fault injectors. In *Proc. Intl. Computer Performance and Dependability Symp.* 91–100.

R. Strom and S. Yemini. 1985. Optimistic recovery in distributed systems. *ACM Trans. Comput. Syst.* 3, 3 (1985), 204–226.

M. Sullivan and R. Chillarege. 1991. Software defects and their impact on system availability: A study of field failures in operating systems. In *Digest of Papers, Intl. Symp. on Fault-Tolerant Computing*. 2–9.

N. Suri and P. Sinha. 1998. On the use of formal techniques for validation. In *Digest of Papers, Intl. Symp. on Fault-Tolerant Computing*. 390–399.

M. Susskraut and C. Fetzer. 2006. Automatically finding and patching bad error handling. In *Proc. European Dependable Computing Conf.* 13–22.

A. Thakur, R. K. Iyer, L. Young, and I. Lee. 1995. Analysis of failures in the tandem nonstop-UX operating system. In *Proc. IEEE Intl. Symp. on Software Reliability Engineering*. 40–50.

TPCC. 2010. *TPC Benchmark C (TPC-C) v5.11*. Retrieved from http://www.tpc.org/tpcc/.

T. K. Tsai, M. C. Hsueh, H. Zhao, Z. Kalbarczyk, and R. K. Iyer. 1999. Stress-based and path-based fault injection. *IEEE Trans. Comput.* 48, 11 (1999), 1183–1201.

E. van der Kouwe, C. Giuffrida, and A. S. Tanenbaum. 2014. Evaluating distortion in fault injection experiments. In *Proc. IEEE Intl. Symp. High-Assurance Systems Engineering*. 25–32.

P. C. Véras, E. Villani, A. M. Ambrosio, N. Silva, M. Vieira, and H. Madeira. 2012. Errors on space software requirements: A field study and application scenarios. In *Proc. IEEE Intl. Symp. on Software Reliability Engineering*. 61–70.

M. Vieira and H. Madeira. 2003. A dependability benchmark for OLTP application environments. In *Proc. Intl. Conf. on Very Large Data Bases*. 742–753.

M. Vieira, H. Madeira, I. Irrera, and M. Malek. 2009. Fault injection for failure prediction methods validation. In *Proc. Workshop on Hot Topics in System Dependability*.

J. M. Voas. 1998. Certifying off-the-shelf software components. *IEEE Computer* 31, 6 (1998), 53–59.

J. M. Voas, F. Charron, G. McGraw, K. Miller, and M. Friedman. 1997. Predicting how badly "Good" software can behave. *IEEE Software* 14, 4 (1997), 73–83.

C. J. Walter and N. Suri. 2003. The customizable fault/error model for dependable distributed systems. *Theor. Comput. Sci.* 290, 2 (2003), 1223–1251.

E. J. Weyuker. 1998. Testing component-based software: A cautionary tale. *IEEE Software* 15, 5 (1998), 54–59.

L. Wilson. 2013. *International Technology Roadmap for Semiconductors*. Retrieved from http://www.itrs.net.

S. Winter, C. Sârbu, N. Suri, and B. Murphy. 2011. The impact of fault models on software robustness evaluations. In *Proc. Intl. Conf. on Software Engineering*. 51–60.

S. Winter, M. Tretter, B. Sattler, and N. Suri. 2013. simFI: From single to simultaneous software fault injections. In *Proc. IEEE/IFIP Intl. Conf. Dependable Systems and Networks*.

C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. 2012. *Experimentation in Software Engineering*. Springer.

W. E. Wong and A. P. Mathur. 1995. Reducing the cost of mutation testing: An empirical study. *J. Syst. Software* 31, 3 (1995), 185–196.

J. Xu, S. Chen, Z. Kalbarczyk, and R. K. Iyer. 2001. An experimental study of security vulnerabilities caused by errors. In *Proc. IEEE/IFIP Intl. Conf. Dependable Systems and Networks*. 421–430.

Received June 2013; revised July 2015; accepted October 2015