Experimental Analysis of Binary-Level Software Fault Injection in Complex Software

Domenico Cotroneo, Anna Lanzaro, Roberto Natella
Dipartimento di Informatica e Sistemistica
Università degli Studi di Napoli Federico II
Via Claudio 21, 80125, Naples, Italy
{cotroneo, anna.lanzaro, roberto.natella}@unina.it

Ricardo Barbosa
Critical Software S.A.
Parque Industrial de Taveiro, Lote 48
3045-504, Coimbra, Portugal
rbarbosa@criticalsoftware.com

***Abstract*—The injection of software faults (i.e., bugs) by mutating the binary executable code of a program enables the experimental dependability evaluation of systems for which the source code is not available. This approach requires that programming constructs used in the source code should be identified by looking only at the binary code, since the injection is performed at this level. Unfortunately, it is a difficult task to inject faults in the binary code that correctly emulate software defects in the source code. The *accuracy* of binary-level software fault injection techniques is therefore a major concern for their adoption in real-world scenarios. In this work, we propose a method for assessing the accuracy of binary-level fault injection, and provide an extensive experimental evaluation of a binary-level technique, G-SWFIT, in order to assess its limitations in a real-world complex software system. We injected more than 12 thousand binary-level faults in the OS and application code of the system, and we compared them with faults injected in the source code by using the same fault types of G-SWFIT. The method was effective at highlighting the pitfalls that can occur in the implementation of G-SWFIT. Our analysis shows that G-SWFIT can achieve an improved degree of accuracy if these pitfalls are avoided.**

***Keywords*-Software Fault Injection, G-SWFIT, Experimental Dependability Assessment, Off-The-Shelf Software**

I. Introduction

The injection of software faults (Software Fault Injection, SFI) for the assessment of fault-tolerant software is relatively new if compared to decades of research on fault injection being focused on hardware-induced faults. Existing fault injection techniques can emulate hardware faults using simple bit-flip or stuck-at fault models [1]–[5], and modern fault injection tools can inject this kind of faults through software (Software Implemented Fault Injection, SWIFI [6], [7]). Software Fault Injection, instead, aims at the realistic emulation of *software faults* (i.e., bugs[1]) in a software component to assess the impact of these faults on the system behavior. SFI is assuming an increasing relevance since software faults have been recognized as one of the major causes of system failures [10], [11]. It is used for the experimental validation and improvement of fault tolerance mechanisms and algorithms [12], [13]; it makes possible to analyze worst-case scenarios and the effects of faulty components [14], [15]; it is used in conjunction with dependability forecasting techniques, in order to populate dependability models with measures obtained from experiments [16]–[18]; and to benchmark alternative systems or design choices [19]. The realistic emulation of software faults is a key objective to achieve accurate dependability measures and to investigate faulty scenarios that the system could face during operation.

One of the most popular SFI technique is G-SWFIT (Generic Software Fault Injection Technique), proposed by Durães and Madeira [9]. G-SWFIT injects software faults by mutating the binary executable code of a program. This technique is attractive for practitioners, since it allows to perform Software Fault Injection when the source code is not available, which is often the case when third-party software is adopted. G-SWFIT defines which types of software defects have to be introduced in order to realistically emulate a faulty software, based on recent field data studies that characterized residual software faults in complex systems [9], [20], [21].

An important issue concerning the injection of software faults at binary level is the *accuracy* of the injection campaign, that is, the degree of confidence that a fault injected in the binary code correctly emulates a software defect in the source code. For instance, if we aim to emulate the absence of a variable assignment in the source code, we could remove a "move" instruction at binary level. But, if we consider the emulation of a bug in a C preprocessor macro (i.e., a piece of source code that is replicated several times in the binary code), the problem cannot be resolved by simply looking at the binary code. Therefore, it is important to assess the accuracy of binary-level SFI in order to be effectively adopted in real-world scenarios. Unfortunately, only a few studies evaluated the accuracy of binary-level SFI, which were limited to small programs or to a small number of faults [9], [22], [23], and no previous work analyzed this problem comprehensively.

In this work, we propose a method for assessing the accuracy of binary-level fault injection in complex software, and perform an extensive experimental campaign in order to assess the accuracy of G-SWFIT. To this aim, two fault

[1] In this work, we follow the notion that a software fault is a development fault originated during the coding phase [8], [9].

injection campaigns are conducted respectively on binary code and source code, where the latter is used as a term of comparison. During these campaigns we keep track of code locations targeted by fault injection. We then compare for each fault type the locations affected by source-level injection with the ones affected by binary-level injection. In this way, we are able to identify: (i) binary-level faults which correctly emulate software faults (this happens when we experience the same fault type in the same location both from binary-level injection and from source-level injection); (ii) binary-level faults that do not emulate any software fault (this happens when a binary-level fault is injected in a location in which the fault could not exist in the source code); and (iii) binary-level faults that have not been injected in a location where they could have been injected.

Experiments consist of the injection of about 30 thousand faults, 12 thousand binary-level faults and 18 thousand source-level faults, in a real world system from the space domain, i.e., a satellite data handling system. The proposed method was effective at highlighting the pitfalls that can occur in the implementation of G-SWFIT and affect the accuracy of fault injection. In particular, issues were found in the identification of code blocks and control structures, and in enforcing fault constraints. Moreover, our analysis shows that if identified pitfalls are avoided, the accuracy of G-SWFIT can be significantly improved.

The following section describes the state of the art on Software Fault Injection and provides more details about G-SWFIT. Section III describes the proposed method. Section V discusses the obtained results by applying the method on a complex system described in Section IV. Conclusions are summarized in Section VI.

II. Background and related work

A. Software Fault Injection Techniques and Tools

In order to emulate software faults in fault injection experiments, a model of software faults that can realistically occur in the system under test is required. This property, which is referred to as *representativeness*, is desirable when dependability measures have to be quantitatively assessed, such as coverage factors of fault-tolerant systems [16], [17], which depend on the probability distribution of faults and workloads [18]. Fault representativeness is also important to stimulate the complex failure modes that can be exhibited by a software system or component, which are potentially more subtle than simple process hangs or crashes and are not necessarily known a priori [13], [14].

Field data studies analyzed software faults in complex software systems, and can be used to define software fault models. Sullivan and Chillarege [20] analyzed a large set of software-related failure reports collected from the MVS OS, and proposed a classification scheme for software faults, which are described in a level of detail close to the programming level. That work was later extended in [24] where the Orthogonal Defect Classification (ODC) and the notion of defect type are introduced. This notion points to a high-level classification of faults including Function, Checking, Assignment, Algorithm and Interface faults. ODC was aimed at providing feedback during development; the work presented in [9] extends this level of description and proposes a classification scheme that was precise enough for automated fault emulation (e.g., for the "assignment" class of faults, it specifies if the assignment is an initialization, and if an expression or constant is involved). It also presents a field data study where it is pointed out that most of the software faults found in the field belong to the set of fault types shown in Table I, and that they tend to follow a generic fault distribution.

Table I
Fault operators (see also [9]).

Fault Type	Description
MFC	Missing function call
MVIV	Missing variable initialization using a value
MVAV	Missing variable assignment using a value
MVAE	Missing variable assignment with an expression
MIA	Missing IF construct around statements
MIFS	Missing IF construct + statements
MIEB	Missing IF construct + statements + ELSE construct
MLAC	Missing AND in expression used as branch condition
MLOC	Missing OR in expression used as branch condition
MLPA	Missing small and localized part of the algorithm
WVAV	Wrong value assigned to variable
WPFV	Wrong variable used in parameter of function call
WAEP	Wrong arithmetic expression in function call parameter

Another aspect affecting the effectiveness of Software Fault Injection is represented by the method adopted to introduce software faults into a system. In fact, SFI requires more complex modifications of the program code/state than simply a bit-flip/stuck-at: the comparison between real software faults and faults injected by SWIFI tools [6], [7] revealed that hardware fault models cannot accurately emulate software faults. The emulation of software faults requires that what it is injected reproduces the intended fault model (we refer to this property as *accuracy*), in order to correctly evaluate the effects of software faults on the system. Several methods have been devised for emulating software faults, most of them based on rather indirect approaches (i.e., emulating the possible effects of software faults instead of injecting actual faults in the software code).

Past work on software fault injection can be divided in three categories, according to what is actually injected: data errors, interface errors, and code changes (summarized in Table II). We include tools and approaches that were adopted in past work in the context of dependability assessment of fault-tolerant systems, and do not consider tools for mutation testing (where faults are used to define test cases and not for dependability assessment) since they are out of the scope of this paper.

Data errors. This approach consists of injecting errors in the data of the target program (i.e., a deviation from the correct system state [8]). This is an indirect form of fault injection, as what is being injected is not the fault itself but only a possible effect of the fault. The representativeness of this type of injection is difficult to assert, as the relationship between data corruption and its possible root-cause (i.e., faults) is difficult to establish. However, data errors are an useful and practical means for inducing software failures and debugging of fault-tolerance mechanisms [14].

Interface errors. This approach is in fact another form of error injection where the error is specifically injected at the interface between modules (e.g., system components, or functional units within a program). This usually translates to parameter corruption in functions and API, and it is considered a form of robustness testing. The errors injected can take many forms: from simple data corruption to syntactically valid but semantically incorrect information. As with data errors, the representativeness of the errors injected at the interfaces is not clear and there is some empirical evidence that supports the idea that injecting interface errors and changing the target code produces different effects in the target [25]. This approach is complementary to the injection of actual software faults, and it has proven to be useful to find interface weaknesses [26].

Code changes. Changing the code of the target component to introduce a fault is naturally the closest thing to having the fault there in the first place. However, this is not easily achieved as it requires to know exactly where in the target code one might apply such change, and what instructions should be placed in the target code. Several works followed this notion, although with some limitations: Ng and Chen [13] and the FINE [17] and DEFINE [27] tools use code changes (e.g., changing the destination address of an assignment), although their fault model is very simple and its representativeness is not assured. Madeira et al. [7] showed that SWIFI can be used to inject simple code changes in running processes but cannot emulate more complex software faults. The G-SWFIT technique [9] was developed to address software fault representativeness, by injecting software faults according to the set of most common fault types (Table I) observed in field data.

Table II
CLASSIFICATION OF FAULT INJECTION TOOLS.

Category	Tools
Data errors	FIAT [1], FERRARI [2], PSN [14], csXception [5], NFTAPE [3], GOOFI [4]
Interface errors	BALLISTA [26], RIDDLE [28], MAFALDA [29], Jaca [30], csXception [19]
Code changes	Ng and Chen [13], FINE [17], DEFINE [27], G-SWFIT [9]

B. G-SWFIT

G-SWFIT injects code changes at the executable (binary) level (Figure 1a). It consists of a set of *fault operators* that define *code patterns* (i.e., a sequence of opcodes) in which faults can be injected (e.g., an MIA fault can be injected wherever an IF construct is found), and *code changes* to be introduced (e.g., the removal of instructions related to an IF construct) to emulate software faults[2]. The proposed fault operators inject valid faults in terms of programming language (i.e., mutated code is syntactically correct) and provide a set of constraints to exclude fault locations that are not realistic (e.g., to inject an MIA fault, the IF construct must not be associated to an ELSE construct, and it must not include more than five statements or loops). The description of a fault operator is provided in Table III. As discussed in the rest of this paper, it is not trivial to assure the accuracy of software fault injection at the binary level, due to the gap between software faults at source code level (e.g., defects in a program) and their conversion to binary level (i.e., translation of the faulty code in machine code). The implementation of G-SWFIT and the definition of fault operators are dependent on the hardware architecture, the compiler of the target application, and compiler optimizations, since the binary translation of a programming construct (e.g., an IF construct) varies with the compiler and the hardware platform in which the software can be executed. G-SWFIT was originally implemented and applied on the i386 hardware architecture and the Microsoft Windows environment [31]. The technique has then been ported to inject faults in the bytecode of Java programs [32]. In this work, we analyze G-SWFIT for the C language with respect to the PowerPC hardware architecture and the GCC compiler, which has been implemented in a R&D tool by Critical Software.

Table III
DESCRIPTION OF THE OMFC FAULT OPERATOR.

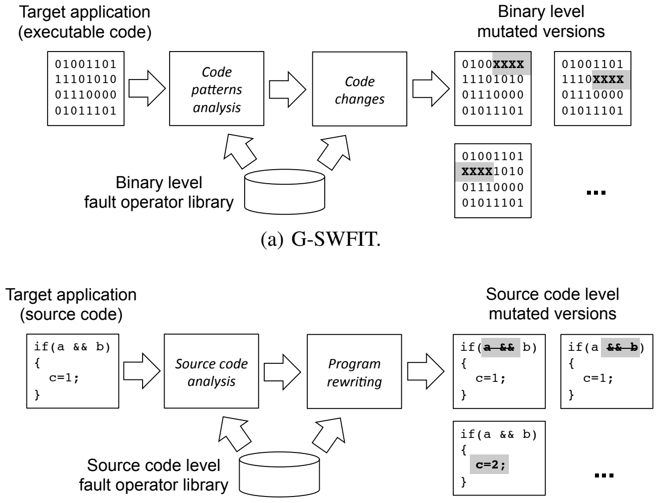
Example	*function(...);*
Example with faults	~~*function(...);*~~
Code pattern	*CALL target-address*
Code change	*CALL* instruction removed
Contraints	Return value of the function must not be used (C01) Call must not be the only statement in the block (C02)

An alternative approach to change the code of a program consists in mutating its source code, and then to compile the faulty source code to obtain a faulty version (Figure 1b). This approach has been implemented in a fault injection tool developed by our research group, namely SAFE[3]. The tool adopts the same fault types of G-SWFIT (Table I), including code patterns and constraints, although faults are introduced in the source code instead of the binary code. This tool has different objectives than G-SWFIT, since it cannot perform fault injection when the source code is not available; it is considered in this work as a support to evaluate the accuracy of G-SWFIT. In order to use the SAFE tool, a C preprocessor translates C macros in a source code file (e.g., inclusion of

[2] Each fault operator is related to a specific fault type and is denoted with the "O" prefix (e.g., the OMIA fault operator is related to the MIA fault type).

[3] SoftwAre Fault Emulation tool: http://www.mobilab.unina.it/SFI.htm

header files) to produce a self-contained compilation unit. A C/C++ front-end then processes the compilation unit, in order to produce an internal representation of the program (Abstract Syntax Tree, AST). The tool searches for suitable fault locations in the AST and applies a fault operator if all constraints are met, e.g., to inject a MIFS fault, an IF construct should not contain more than 5 statements. The tool produces a set of faulty source code files, each containing a different software fault. The faulty version is obtained by replacing a source code file with a faulty file and recompiling the program.



(a) G-SWFIT.

(b) SAFE.

Figure 1. Software Fault Injection techniques.

Compared to the binary level approach followed by the original G-SWFIT, the source code level approach assures the accurate emulation of fault types, since full information about programming constructs and variables is available (this information is missing and has to be reconstructed when injecting faults at the binary level). Moreover, injection in the source code is portable among all platforms in which the target program can be compiled, without any additional efforts to adapt the fault injection tool to different hardware or compilers. The drawbacks of this approach are that it increases experiment time, since the program needs to be compiled after the injection of a fault, and that the approach cannot be adopted when the source code is missing.

III. Proposed method

As previously discussed, the evaluation is motivated by the fact that the accuracy of binary-level fault injection is limited by the impossibility to correctly recognize some programming constructs in a binary program. The evaluation of binary-level fault injection in a real-world system contributes to understand the limitations and the accuracy of the results that can be obtained by a fault injection campaign.

An example of a wrongly injected fault is represented by a C program containing a SWITCH construct with two branches; in some architectures and compilers (this is the case of GNU GCC compiler for PowerPC architectures), the SWITCH may be translated in binary code using the same opcode sequence of an IF-ELSE construct, since they both consist of a logical condition (which is translated using an opcode that compares two values) and two branches (which are translated using branch opcodes). Therefore, a MIEB (see Table I) fault could erroneously be injected in a code location in which there is not an IF-ELSE construct. It may also happen that a code location suitable for fault injection cannot be recognized in the binary code. For instance, a compiler may translate a function call as inline code (i.e., the function call is replaced with the body of the called function); in this case, a fault injection tool would not be able to recognize the function call, thus omitting to inject an MFC fault in that location. The experimental validation in this work aims to assess the relative occurrence of this kind of problems in real-world complex software, in order to evaluate whether G-SWFIT can achieve an acceptable degree of accuracy even in the presence of these problems. Although some of these problems are already known, their extent in large and complex software has not been investigated in previous studies.

This work also aims to point out issues that may arise when implementing G-SWFIT, by highlighting cases in which faults are not correctly injected. Binary-level fault injection tools are difficult to implement, since they have to encompass all potential ways in which programming constructs are translated. This problem is further exacerbated if we consider the complexity of modern CPUs, programming languages and compilers (whose inner working is usually unknown). Thus it is likely that developers may neglect some code patterns, thus leading to design errors in the fault injection tool.

The proposed method evaluates the accuracy of G-SWFIT by comparing the faults it generates with the ones injected in the source code. Indeed, since a software fault is a defect in the code of a program, it is clear that fault injection at source code level is more accurate. Based on this consideration, we compare the faults injected by the two techniques and we classify faults in the following three categories:

1) *Correctly Injected faults*: correct faults generated by both techniques. The larger is the set of common faults, the higher is the accuracy of G-SWFIT.
2) *Omitted faults*: faults injected only at source-code level. They correspond to programming constructs in which a fault could exist, but which have not been identified in the binary code.
3) *Spurious faults*: faults injected only by G-SWFIT at binary level that do not match any fault at source-code level. Therefore, they are not considered as representative software faults.

It is important to note that source-level faults can be used as a term of comparison for binary-level faults because (i) *the same fault types are adopted for both binary- and source-level fault injection* (shown in Table I), and (ii) *binary- and source-*

level faults are injected in every potential location (i.e., fault injection campaigns are exhaustive). The method (depicted in Figure 2) consists of two phases, namely (i) automatic matching of binary-level and source-level faults (Section III-A), in order to identify Correctly Injected faults, and (ii) fault sampling and manual analysis (Section III-B), in order to identify which issues affect the accuracy of G-SWFIT. As a real-world case study, we consider CDMS (Command and Data Management System), a real-time embedded system developed by Critical Software for the space domain (Section IV).

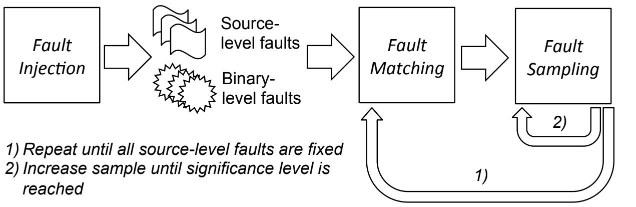
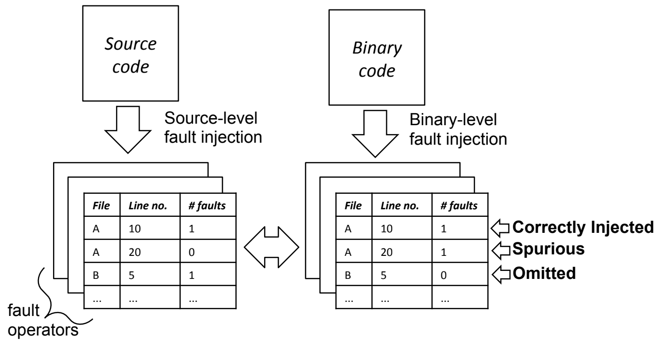

Fault Injection
Source-level faults
Binary-level faults
Fault Matching
Fault Sampling
2)
1)
1) Repeat until all source-level faults are fixed
2) Increase sample until significance level is reached

Figure 2. Overview of the method adopted for the evaluation of G-SWFIT.

A. Fault Matching

Fault Matching is based on the assumption that if both techniques inject the *same fault type* in the *same location* (e.g., an assignment or function call is removed both in the source code and in the corresponding location in the machine code), then they are injecting the same fault. It is reasonable to make this assumption since if a fault location is identified both at the binary and source levels, then that fault location is valid and correctly handled. In order to be sure that this assumption holds (and therefore the results are valid), we manually analyzed a sample of Correctly Injected faults using the Fault Sampling procedure (explained in the next subsection). Following this observation, binary-level and source-level faults are compared with respect to their fault types and their locations in the source code (i.e., the source file, the function and the line of code in which a fault is injected). A binary-level fault matches a source-level fault *if they have the same fault type and they are injected in the same code location* (compared using debug symbols in binary code).

The procedure shown in Figure 3 has been adopted to identify Correctly Injected faults. If a binary-level fault matches a source-level fault, and only one binary-level fault and only source-level fault exist for the code location under analysis, then the binary-level fault is considered as Correctly Injected. In some cases (e.g., when there are more than one statement in the same line of code), more than one binary-level fault (N), or more than one source-level fault (M) may occur in the same code location. If there are more binary-level faults than source-level faults in the same location ($N > M$), then there are M Correctly Injected faults, and $N - M$ Spurious faults. Similarly, if source-level faults are more than binary level faults ($M > N$), then there are $M - N$ Omitted faults. It follows that if a binary-level fault does not match any source-level fault, then it is considered a Spurious fault, and that if a source-level fault does not match any binary-level fault, then it is considered an Omitted fault. In the examples of Figure 3, the proposed procedure identifies one Correctly Injected fault (location *A-10*), one Spurious fault (location *A-20*), and one Omitted fault (location *B-5*).



File	Line no.	# faults
A	10	1
A	20	0
B	5	1
...	...	...

File	Line no.	# faults
A	10	1
A	20	1
B	5	0
...	...	...

```
for each fault operator
  for each fault location
    M = Source-level Faults
    N = Binary-level Faults
    if M<N then
      Correctly Injected Faults += M
      Spurious Faults += N-M
    if M>N then
      Correctly Injected Faults += N
      Omitted Faults += M-N
    if M==N then
      Correctly Injected Faults += M
```

Figure 3. Fault matching procedure.

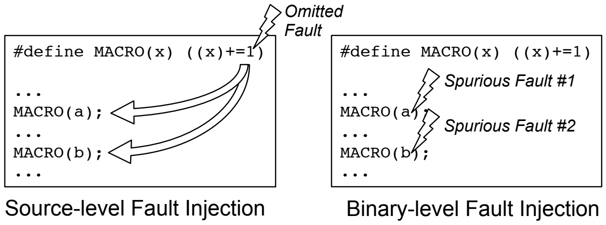
B. Fault Sampling

After the Fault Matching procedure, we perform a detailed analysis of faults in order to investigate the causes of Spurious and Omitted faults, and to verify that Correctly Injected faults are actually correct. Moreover, we aim to understand whether Omitted and Spurious faults are due to inherent limitations of G-SWFIT or not. Indeed, these faults may occur due to design issues in G-SWFIT as previously discussed; the identification of these issues is useful to provide guidelines for improving G-SWFIT, and to obtain a more precise figure of merit of the G-SWFIT technique. For these reasons, we manually analyze a random sample of Omitted and Spurious faults, and classify them into the following categories:

1) *C preprocessor macros*. When the G-SWFIT technique was proposed, preprocessor macros have been recognized as a frequent cause of Omitted and Spurious faults [9]. A preprocessor macro consists of a piece of code that is replicated for each time the macro is referred within the program. Therefore, when a preprocessor macro has a software fault, the faulty code is replicated several times in the binary code. Since the binary code lacks information about macros, G-SWFIT cannot recognize that macro code is replicated elsewhere within the program: therefore, a Spurious fault is injected for each replica of

the macro, and source-level faults that could be injected into macro represent Omitted faults since G-SWFIT cannot correctly injected them (see also Figure 4).

2) *Inline functions*. In a similar way to preprocessor macros, inline functions are replicated each time the function is called within the program. Since G-SWFIT does not recognize inline functions within binary code, they lead to Spurious and Omitted faults as well.
3) *Various causes*. In this category, we include all the other causes of Spurious and Omitted faults that are not related to macros or inline functions.
4) *Issues in the SAFE tool*. Even if source-level fault injection can be considered accurate, we did not exclude this possibility that the source-level fault injection tool we adopt could inject faults incorrectly. Therefore, during the manual analysis, we also look for issues in the SAFE tool that caused faults to erroneously appear as Spurious or Omitted faults. Since we need to assure that source-level faults are correctly injected, we fix the SAFE tool when an issue is found and repeat the whole analysis (including both Fault Matching and Fault Sampling) until this category becomes empty.



```
#define MACRO(x) ((x)+=1)
...
MACRO(a);
...
MACRO(b);
...
```

```
#define MACRO(x) ((x)+=1)
...
MACRO(a);
...
MACRO(b);
...
```

Figure 4. Examples of Spurious and Omitted faults due to the occurrence of a C preprocessor macro within a program.

Because of the high number of the generated faults, the manual analysis is conducted on a sample of faults and then conclusions are drawn about the whole set of faults. In order to generalize the results from the sample, we have to address the problem of choosing a sample of appropriate size, such that it could be considered representative of a population with more than two categories (i.e., a multinomial distribution, where we define π_i as the proportion of the ith category). The sample should be large enough to assure that all of the estimated proportions π_i are within a given confidence interval with significance level $1 - \alpha$.

Assuming that the population and the sample are large enough to use the normal approximation, the probability α_i that the proportion π_i lies outside an interval of width $2d_i$ is given by (see [33] for more details about sampling)

$$\alpha_i = Pr\left\{|Z_i| \geq d_i\sqrt{n}/\sqrt{\pi_i(1-\pi_i)}\right\} \qquad (1)$$

where $1 \leq i \leq k$ and Z_i is a standard normal random variable. By Bonferroni's inequality [33], the probability that one or more of the k estimates will fall outside its interval will be less than or equal to $\sum_i^k \alpha_i$. Equation (1) allows to assess if the sample size is large enough to achieve accurate results. If $\sum_i^k \alpha_i > \alpha$, then a larger sample size is required, otherwise the estimated proportions are considered accurate.

This method was applied to the populations of Omitted and Spurious faults by considering $k = 4$ categories (C preprocessor macros, inline functions, various causes, issues in the SAFE tool), assuming a confidence interval of half-width $d_i = 0.05$ and a significance level $1 - \alpha = 0.9$. This method was also applied to the population of Correctly Injected faults, in order to analyze whether they are truly correct or not ($k = 2$ categories are considered). For each population, we extract a sample of 5% of faults and then we manually analyze each fault in order to obtain an initial estimate of the proportions; the sample size is gradually increased and analyzed until the required significance level is reached. The results are described in the Section V.

IV. Case study

The case study considered in this work is a satellite data handling system named Command and Data Management System (CDMS). A satellite data handling system is responsible for managing all data transactions (both scientific and satellite control) between ground system and a spacecraft (Figure 5), based on the ECSS-E-70-41A standard [34] adopted by the European Space Agency. In this system, a space telescope is being controlled and the data collected is sent to a ground system. As shown in the Figure, the CDMS, which executes on the spacecraft (*on-board system*), is composed by several subsystems: the TC Manager receives a series of commands from the ground control requesting telemetry information; the TM Manager sends back telemetry information for each command sent; the other modules (PC, PL, OBS, RM, DHS) perform tasks for the data management and the telescope handling. The importance of the *accuracy* of SFI in mission-critical systems like CDMS has been demonstrated in [15], in which two OSs (RTLinux and RTEMS) were compared with respect to the risk of failures of the CDMS due to OS faults, in order to select the most reliable OS for this scenario.

The CDMS application was developed in C and runs on top of an open-source, real-time operating system, namely RTEMS[4]. The CDMS makes use of the RTEMS API for task management, communication and synchronization, and for time management. This software system is compiled to run on a PowerPC hardware board by using the GCC compiler and disabling compiler optimization settings, which is the setup currently supported by the G-SWFIT tool.

In this work, we analyze faults injected in both the OS (i.e., RTEMS) and application (i.e., CDMS) code. *We only consider the code which is actually compiled and linked in the executable running on the on-board system.* A small part of the code (1.90%), which is written in assembly language

[4] http://www.rtems.org

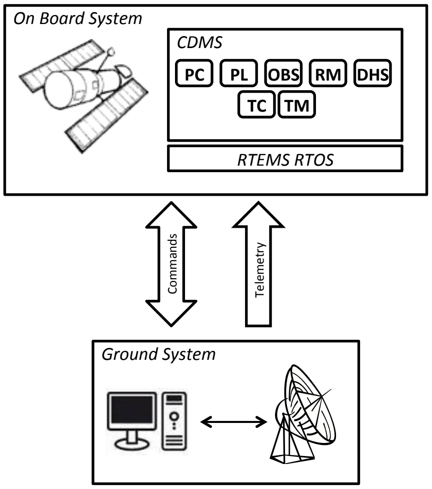


Figure 5. Architecture of the case study.

to provide board-specific support, is not targeted by our source-level fault injection tool, but its influence on the results can be considered negligible.

V. Results

In this section, software faults injected at the binary and source-level in a complex case study are analyzed using the method proposed in Section III. Faults at the binary level were generated with the G-SWFIT technique, by using a R&D prototype tool provided by Critical Software company. Faults at the source code level were generated using the SAFE fault injection tool (described in Section II-B). In total, 18,183 source-level faults and 12,380 binary-level faults were generated, respectively. Their distribution across fault types is shown in Figure 6. The two distributions exhibit noticeable differences: more source-level faults are injected with respect to some fault operators (such as OMLPA, OWVAV, OWPFV, and OWAEP), whereas in other cases more binary-level faults are injected (such as OMIEB and OMVA, where the latter groups together the OMVAV, OMVIV, and OMVAE operators).

The Fault Matching procedure (Section III-A) identified the subset of Correctly Injected faults (i.e., common to both techniques) that we further analyzed in order to assure the correctness of our method. Correctly Injected faults have been sampled (see Section III-B), and then compared by looking at i) the faulty binary-code generated by G-SWFIT, and ii) the one produced by faults injected in the corresponding source-code locations. This analysis revealed that the binary-level faults match the source-level faults for each fault types and for each sampled faults, except the OWPFV operator. We found that 40.69% of OWPFV faults at the binary level do not match OWPFV faults at the source-code level even if they affect the same locations, since there are several functions parameters and possible replacements for a given location. In order to take into account this aspect, results shown in Figure 7 have been updated by reducing the number of Correctly Injected faults for the OWPFV operator and increasing the number of Omitted and Spurious faults by the same amount.

Correctly Injected faults turned out to be 5,927 (Figure 7). They represent 47.88% of faults injected by G-SWFIT. The remaining faults injected by G-SWFIT (52.12%) in the binary code do not match to a software fault in the source code, therefore most of G-SWFIT faults are Spurious. Correctly injected faults represent 32.60% of faults injected in the source code, so the remaining faults at the source level (67.40%) are not emulated by G-SWFIT and they result as Omitted faults. The experimental campaign confirms that the accurate injection at the binary level is a challenging task, at least when a complex software system is considered.

The distribution of the causes of inaccuracies (for both Omitted and Spurious faults) are presented in Figure 8. These distributions have been obtained by applying the sampling procedure described in Section III-B. Most of spurious faults (Figure 8b) are caused by C macros (56%) and inline functions (17%). In these cases, every time that a macro or inline function has been replicated in the binary code, G-SWFIT generated an individual binary-level fault; this led to a large number of Spurious faults (i.e., Spurious faults are repeated for each replica of a macro or inline function). In a similar way, macros and inline functions are a noticeable part of Omitted faults (27% and 1%, respectively); this percentage is low when compared to Spurious faults, since one Omitted fault in a macro or inline function leads to several Spurious faults, one for each replica of the code (see also Figure 4).

In order to gain more insights into the results, we separately analyzed the faults injected in the OS and application code, respectively. Figures 9 and 10 show from a different perspective the data of Figures 7 and 8, by dividing the results between faults in RTEMS (i.e., OS code) and in CDMS (i.e., application code). It can be noted that faults follow a similar trend in OS and application code, since in both cases the number of spurious faults is close to the number of correctly injected faults, and the number of omitted faults is predominant. Nevertheless, omitted faults seem to be much more in the case of CDMS (Figure 9b).

Figure 10 shows that omitted and spurious faults due to various causes (i.e., not related to macro or inline functions) are more frequent in CDMS than in RTEMS. The constructs not correctly recognized at the binary level (e.g., see the examples in Figures 12 and 13 discussed later in this section) likely occur more often in application code due to higher complexity of that code, thus causing an higher number of omitted faults. Moreover, macros and inline functions are more frequent for RTEMS; this is due to the fact that several RTEMS functions are exported as macros and inline functions in order to be used by external code (i.e., user and library code that is compiled and linked with RTEMS code).

Software complexity metrics collected from the case study code (see Table IV) confirm that functions in the application code tend to be more complex than those in the OS code (in

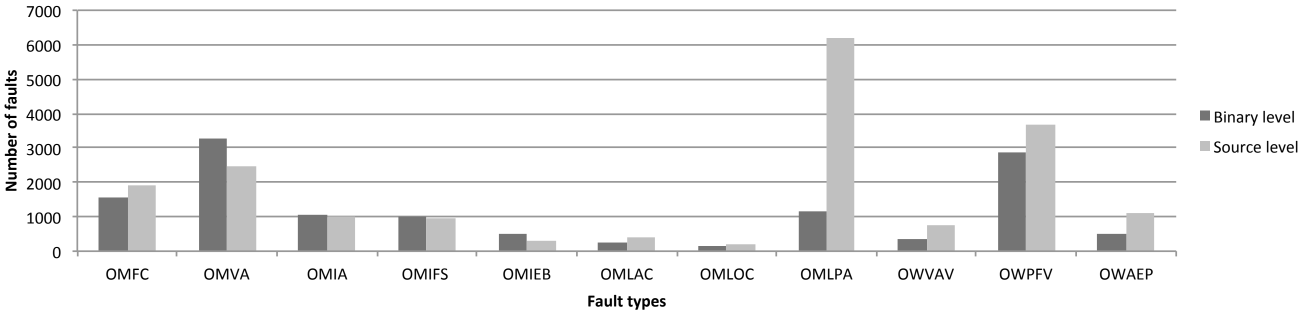


Figure 6. Distributions of software faults injected at the binary and source code level, respectively.

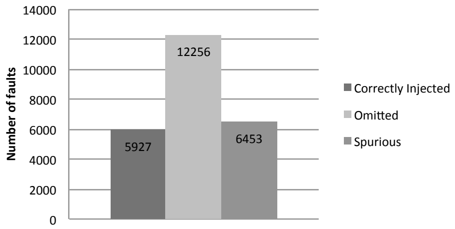
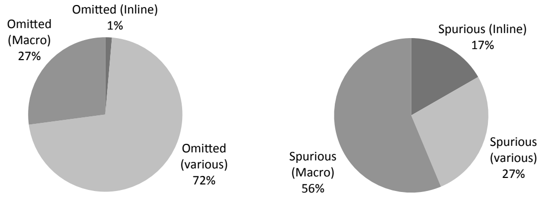


Figure 7. Number of Correctly Injected, Spurious, and Omitted faults.



(a) Omitted faults.

(b) Spurious faults.

Figure 8. Causes of incorrect fault injection in the case study.

term of size, cyclomatic complexity and input/output dependencies). This is a common trend in embedded systems, in which the OS is kept as simple as possible in order to reduce the overhead and the number of potential defects [36]. Moreover, the number of preprocessor statements per function confirms that RTEMS makes a more extensive use of macros that CDMS. Therefore, we conclude that it is even more important to fix the implementation issues mentioned above if a fault injection tool is intended to be used with complex software.

Table IV
COMPARISON OF AVERAGE SOFTWARE COMPLEXITY METRICS OF FUNCTIONS IN RTEMS AND CDMS CODE.

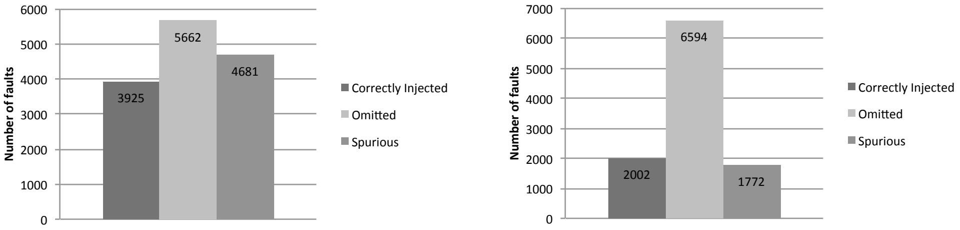
Metric	**RTEMS**	**CDMS**
Lines of Code	17.30	30.71
Preprocessor Statements	0.64	0.15
Cyclomatic number	5.63	6.61
Number of inputs	5.50	7.38
Number of outputs	4.12	6.84

The "various causes" behind spurious and omitted faults are numerous and specific to each fault operator. We cannot provide a precise estimate of the relative percentage of each cause, since it would require to manually analyze an extremely large sample of injected faults. Instead, we tried to identify which part of incorrectly injected faults are due to unavoidable limitations of G-SWFIT, and which of them can be avoided by improving the G-SWFIT fault injection tool. We do so by excluding from the sample those faults not related to macros or inline functions, and by diagnosing (with the support of Critical Software developers) the reasons why omitted faults were not injected, and why spurious faults were erroneously injected. We found that 26.02% of omitted and spurious faults were due to causes that are impossible to avoid when injecting at the binary code level, including:

- *Low-level translation of C operators*. Some C expressions (like *sizeof* and array and struct accesses using -> and *[]*) are translated by introducing arithmetic operations and constants in the binary code; these operations are recognized as arithmetic expressions by fault operators such as OMVA, OWVAV, and OWAEP.
- *Switch and goto constructs*. These constructs are translated in a similar way to IF constructs using *branches* in the binary code; therefore, IF constructs are not always correctly identified by operators such as OMIA, OMIEB, and OMIFS.
- *Forced function inlining*. Some functions (e.g., *memcpy*, *memset*) are compiled as inline functions, although they are not declared as inline.

Since the binary code lacks information about high-level constructs, the causes mentioned above cannot be avoided. In practice, these inaccuracies have to be accepted as limitations of fault injection at binary level, and should be taken into account when conclusions are drawn from fault injection experiments.

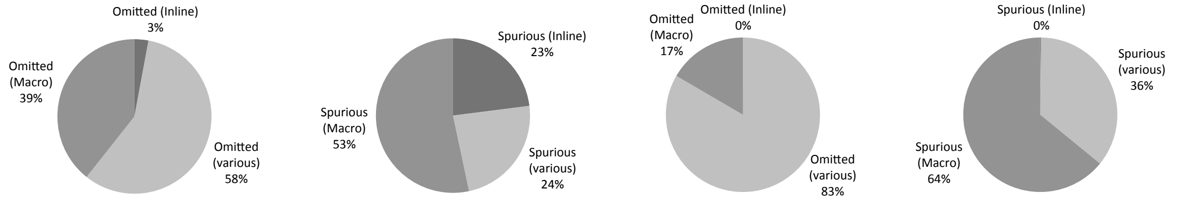
Nevertheless, during the manual analysis we observed several Omitted and Spurious faults not related to intrinsic limitations of fault injection at binary level, but were due to limitations of the fault injection tool; they represent the 73.98% of the sample that we analyzed. These inaccuracies occurred since some checks have not been implemented yet in the tool, and some fault operators diverge in some



(a) RTEMS code.

(b) CDMS code.

Figure 9. Number of faults (correctly injected, spurious, and omitted) in OS and application code.



(a) Omitted faults in RTEMS.

(b) Spurious faults in RTEMS.

(c) Omitted faults in CDMS.

(d) Spurious faults in CDMS.

Figure 10. Causes of incorrect fault injection in OS and application code.

cases from the fault types encompassed by G-SWFIT due to choices that simplify the implementation. Therefore, part of the Omitted and Spurious faults could be avoided by improving the implementation of binary-level fault injection.

An example of Spurious fault is provided in Figure 11, which shows a fault location in the source code (monospace font) along with its machine code translation (italic font). It is a spurious MFC fault in CDMS that has been injected in a wrong location. In this example, the function call should not be removed since it is the only statement within a block of code, and a fault in that location would not be realistic. The OMFC operator imposes a constraint (Table III) to avoid fault injection in this kind of location [9]. Instead, the fault has been injected by the tool since the block containing the function call is not recognized (i.e., the constraint is not enforced by the tool).

Figure 12 and Figure 13 provide two examples of Omitted faults that were caused by limitations in G-SWFIT implementation. In Figure 12, a function call which could be removed by the OMFC fault operator is not identified. As confirmed by Critical Software developers, the *TcMakePacket* function is not recognized as returning a value that is stored and used later in the program. Therefore, a fault is not injected due to a constraint of the OMFC operator requiring that the return value of a function should not be in use (Table III).

In Figure 13, the fault location has been omitted for an even more subtle reason. In this example, the *return* statement within the IF construct is translated with a branch to the end of function, and the tool incorrectly believes that the IF construct includes all the statements until the end of the current function. A fault is not injected since the IF construct

```
static void HousekeepingAction(TmPacket *STm) {
            stwu r1,-24(r1)
            mflr r0
            stw r31,20(r1)
            stw r0,28(r1)
            mr r31,r1
            stw r3,8(r31)

    SendTmMsg(pbtBuffer,
        TmGetPacketTotalLength(STm));  ← MFC fault location
            lwz r3,8(r31)                   (to be avoided)
            bl 00006184 <TmGetPacketTotalLength>
            mr r0,r3
            lis r9,7
            addi r3,r9,-21944
            mr r4,r0
            bl 0000a3b4 <SendTmMsg>

}
            lwz r11,0(r1)
            lwz r0,4(r11)
            mtlr r0
            lwz r31,-4(r11)
            mr r1,r11
            blr
```

Figure 11. Spurious MFC fault in CDMS.

```
     TcMakePacket(pbtBuffer, &STc);   ← MFC fault location
            addi r0,r31,24              (not identified)
            lis r9,9
            addi r3,r9,-21492
            mr r4,r0
            bl 000056b8 <TcMakePacket>

     bOk = CheckAppIdTypeSubtype(&STc);
            addi r0,r31,24
            mr r3,r0
            bl 00011a10 <CheckAppIdTypeSubtype>
            mr r0,r3
            stw r0,20(r31)
```

Figure 12. Omitted MFC fault in CDMS.

should not contain more than 5 statements [9]. Although the tool is provided with checks to avoid these mistakes, a check to avoid this specific case was not implemented. This kind of issue seems to be more relevant for Omitted faults than for spurious faults given the high number of omitted faults due to various causes, as depicted in Figures 7 and 8.

```
rtems_status_code sc;
n32Size = TcGetAppData(STc, &pbtData);
    lwz r3,120(r31)
    lis r9,7
    addi r4,r9,-23004
    bl 00005934 <TcGetAppData>
    mr r0,r3
    lis r9,7
    stw r0,-22992(r9)

sc = rtems_semaphore_obtain(rtems_mon_Mutex,
                            RTEMS_WAIT,
                            RTEMS_NO_TIMEOUT);
    lis r9,7
    lwz r0,-22948(r9)
    mr r3,r0
    li r4,0
    li r5,0
    bl 0003d504 <rtems_semaphore_obtain>
    mr r0,r3
    stw r0,64(r31)

if (sc != RTEMS_SUCCESSFUL)        ← MIA fault location
    lwz r0,64(r31)                   (not identified)
    cmpwi cr7,r0,0
    bne- cr7,0000c69c <AddMonitoringAction+0x97c>
    return;

if ( n32Size >= 10 ) {
    lis r9,7
    lwz r0,-22992(r9)
    cmplwi cr7,r0,9
    ble- cr7,0000c680 <AddMonitoringAction+0x960>
```

Figure 13. Omitted MIA fault in CDMS.

Other incorrect behaviors were also found in the prototype tool, which were due to the incomplete implementation of contraints or the identification of code blocks and control structures. In Figure 14, we provide an evaluation of the results that can be obtained by improving the mentioned aspects. The improvements prevent the occurrence of several Omitted and Spurious faults: Correctly Injected faults represent the majority of faults potentially injectable in the source code (i.e., only a minor part of faults is omitted), and they also represent the majority of faults actually injected by G-SWFIT (i.e., only a minor part of faults is spurious). We conclude that the evaluation of a binary-level fault injection tool on real-world complex software is useful to identify implementation issues, and should be adopted to assure that a tool does not omit valid fault locations, and that spurious faults are not generated.

VI. Lessons Learned and Future Work

In this paper, we evaluated the accuracy of a software fault injection technique (G-SWFIT) that injects faults in the binary code of a program. The accuracy of faults injected at binary level has been assessed by comparing the faults injected in the source code by using the same fault injection rules. The analysis pointed out improvements to both tools involved in the comparison. Results can be summarized as follows:

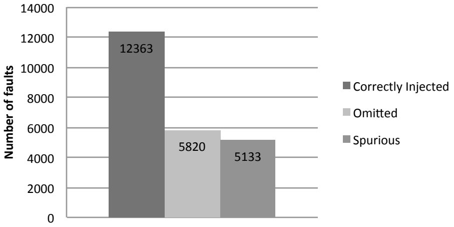


Figure 14. Number of faults (correctly injected, spurious, and omitted) when fixing implementation issues of the G-SWFIT tool.

- The accurate injection of software faults in the binary code is challenging in complex software systems. A large number of omitted and spurious faults was observed in the first analysis: for each injected fault there is about 1 omitted fault that has not been injected, and about half of the injected faults were spurious. Moreover, the problem is more significant where the code complexity is greater, as in the case of application-level code in the case study.
- Several omitted and spurious faults are due to the lack of high-level information in the binary code, and most of them are due to macros and inline functions. These inaccuracies have to be accepted as limitations of fault injection at binary level, and should be taken into account when conclusions are drawn from fault injection experiments. In some cases, such limitations can be considered acceptable: for instance, when the aim of fault injection is a coarse-grained analysis of failure modes (e.g., the relative percentage of crashes or stalls of the system), the results can be adequately estimated even in the presence of inaccurate injected faults [9], [23]. Instead, fault injection at the source level is advisable when the source code is available and a more fine-grained analysis of the effects of injected faults on the system is needed.
- Several omitted and spurious faults are not related to limitations of fault injection at binary level, but they are due to the incomplete or simplified implementation of G-SWFIT. In particular, issues are related to the implementation of fault type constraints and to the identification of code blocks and control structures. These issues are not due to the G-SWFIT technique, and they can be avoided if an experimental evaluation of the fault injection tool is performed to improve the implementation. If these aspects are improved, then omitted and spurious faults represent the minority of cases. A future research work consists in extending the proposed method in order to support the development of SFI tools at binary level, since such tools need to be re-engineered or developed from scratch when fault injection is performed in a new hardware architecture or in a system adopting a different compiler. In this context, faults injected at the source code level can be potentially exploited to understand how software faults are translated in binary code and how fault operators can be implemented.

Acknowledgment

We would like to thank Nuno Silva, João Durães and Henrique Madeira for their useful comments and suggestions. This work has been funded by the FP7 European Project CRITICAL-STEP (http://www.critical-step.eu) IAPP no. 230672.

References

[1] J. Barton, E. Czeck, Z. Segall, and D. Siewiorek, "Fault Injection Experiments using FIAT," *IEEE Transactions on Computers*, vol. 39, no. 4, 1990.

[2] G. Kanawati, N. Kanawati, and J. Abraham, "FERRARI: A Flexible Software-Based Fault and Error Injection System," *IEEE Transactions on Computers*, vol. 44, no. 2, 1995.

[3] D. Stott, B. Floering, Z. Kalbarczyk, and R. Iyer, "A Framework for Assessing Dependability in Distributed Systems with Lightweight Fault Injectors," in *Proc. Intl. Computer Performance and Dependability Symp.*, 2000.

[4] J. Aidemark, J. Vinter, P. Folkesson, and J. Karlsson, "GOOFI: Generic Object-Oriented Fault Injection Tool," in *Proc. Intl. Conf. on Dependable Systems and Networks*, 2001.

[5] J. Carreira, H. Madeira, and J. Silva, "Xception: A technique for the experimental evaluation of dependability in modern computers," *IEEE Transactions on Software Engineering*, vol. 24, no. 2, pp. 125–136, 1998.

[6] T. Jarboui, J. Arlat, Y. Crouzet, K. Kanoun, and T. Marteau, "Analysis of the Effects of Real and Injected Software Faults: Linux as a Case Study," in *Proc. European Dependable Computing Conference*, 2002.

[7] H. Madeira, D. Costa, and M. Vieira, "On the Emulation of Software Faults by Software Fault Injection," in *Proc. Intl. Conf. on Dependable Systems and Networks*, 2000.

[8] A. Avizienis, J. Laprie, B. Randell, and C. Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, 2004.

[9] J. Durães and H. Madeira, "Emulation of Software faults: A Field Data Study and a Practical Approach," *IEEE Transactions on Software Engineering*, vol. 32, no. 11, 2006.

[10] J. Gray, "A Census of Tandem System Availability between 1985 and 1990," *IEEE Trans. on Reliability*, vol. 39, no. 4, 1990.

[11] D. Oppenheimer, A. Ganapathi, and D. Patterson, "Why Do Internet Services Fail, and What Can Be Done About It?" in *USENIX Symp. on Internet Technologies and Systems*, 2003.

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

[13] W. Ng and P. Chen, "The Systematic Improvement of Fault Tolerance in the Rio File Cache," in *Proc. 29th Intl. Symp. on Fault-Tolerant Computing*, 1999.

[14] J. Voas, F. Charron, G. McGraw, K. Miller, and M. Friedman, "Predicting How Badly "Good" Software Can Behave," *IEEE Software*, vol. 14, no. 4, 1997.

[15] R. Moraes, J. Durães, R. Barbosa, E. Martins, and H. Madeira, "Experimental Risk Assessment and Comparison using Software Fault Injection," in *Proc. Intl. Conf. on Dependable Systems and Networks*, 2007.

[16] J. Hudak, B. Suh, D. Siewiorek, and Z. Segall, "Evaluation and Comparison of Fault-Tolerant Software Techniques," *IEEE Transactions on Reliability*, vol. 42, no. 2, 1993.

[17] W.-I. Kao, R. Iyer, and D. Tang, "FINE: A Fault Injection and Monitoring Environment for Tracing the UNIX System Behavior under Faults," *IEEE Transactions on Software Engineering*, vol. 19, no. 11, 1993.

[18] P. Popov and L. Strigini, "Assessing Asymmetric Fault-Tolerant Software," in *Proc. Intl. Symp. on Software Reliability Engineering*, 2010.

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

[20] M. Sullivan and R. Chillarege, "Software Defects and their Impact on System Availability: A Study of Field Failures in Operating Systems," in *Proc. Intl. Symp. on Fault-Tolerant Computing*, 1991.

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

[22] J. Duraes and H. Madeira, "Emulation of Software Faults by Educated Mutations at Machine-Code Level," in *Proc. Intl. Symp. on Software Reliability Engineering*, 2002.

[23] A. Jin and J. Jiang, "Fault Injection Scheme for Embedded Systems at Machine Code Level and Verification," in *Proc. 15th Pacific Rim Intl. Symp. on Dependable Computing*, 2009.

[24] R. Chillarege, I. Bhandari, J. Chaar, M. Halliday, D. Moebus, B. Ray, and M. Wong, "Orthogonal Defect Classification–A Concept for In-Process Measurements," *IEEE Transactions on Software Engineering*, vol. 18, no. 11, 1992.

[25] R. Moraes, R. Barbosa, J. Durães, N. Mendes, E. Martins, and H. Madeira, "Injection of Faults at Component Interfaces and Inside the Component Code: Are They Equivalent?" in *European Dependable Computing Conference*, 2006.

[26] P. Koopman and J. DeVale, "The Exception Handling Effectiveness of POSIX Operating Systems," *IEEE Transactions on Software Engineering*, vol. 26, no. 9, 2000.

[27] W.-I. Kao and R. Iyer, "DEFINE: A Distributed Fault Injection and Monitoring Environment," in *Proc. IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems*, 1994.

[28] A. Ghosh, M. Schmid, and V. Shah, "Testing the Robustness of Windows NT Software," in *Proc. Intl. Symp. on Software Reliability Engineering*, 1998.

[29] J. Arlat, J. Fabre, M. Rodríguez, and F. Salles, "Dependability of COTS Microkernel-Based Systems," *IEEE Transactions on Computers*, 2002.

[30] E. Martins, C. Rubira, and N. Leme, "Jaca: A Reflective Fault Injection Tool based on Patterns," in *Proc. Intl. Conference on Dependable Systems and Networks*, 2002.

[31] J. Durães and H. Madeira, "Generic Faultloads based on Software Faults for Dependability Benchmarking," in *Proc. Intl. Conf. on Dependable Systems and Networks*, 2004.

[32] B. Sanches, T. Basso, and R. Moraes, "J-SWFIT: A Java Software Fault Injection Tool," in *Proc. Latin American Symp. on Dependable Computing*, 2011.

[33] S. Thompson, "Sample Size for Estimating Multinomial Proportions," *The American Statistician*, vol. 41, no. 1, 1987.

[34] European Cooperation for Space Standardization, "ECSS-E-70-41A – Ground Systems and Operations: Telemetry and Telecommand Packet Utilization," 2003.

[35] D. Costa, R. Barbosa, R. Maia, and F. Moreira, "DeBERT: Dependability Benchmarking of Embedded Real-Time Off-the-Shelf Components for Space Applications," *Dependability Benchmarking for Computer Systems*, pp. 255–283, 2008.

[36] K. Qian, D. den Haring, and L. Cao, *Embedded Software Development with C*. Springer, 2009.