Representativeness Analysis of Injected Software Faults in Complex Software

Roberto Natella†, Domenico Cotroneo†, João Durães‡, Henrique Madeira‡
†DIS, Università degli Studi di Napoli Federico II, Via Claudio 21, 80125, Naples, Italy
‡CISUC, University of Coimbra, 3030-290, Coimbra, Portugal
{roberto.natella, cotroneo}@unina.it, {jduraes, henrique}@dei.uc.pt

Abstract

Despite of the existence of several techniques for emulating software faults, there are still open issues regarding representativeness of the faults being injected. An important aspect, not considered by existing techniques, is the non-trivial activation condition (trigger) of real faults, which causes them to elude testing and remain hidden until operation.

In this paper, we investigate how the representativeness of injected software faults can be improved regarding the representativeness of triggers, by proposing a set of generic criteria to select representative faults from a faultload.

We used the G-SWFIT technique to inject software faults in a DBMS, resulting in over 40 thousands faults and 2 million runs of a real test suite. We analyzed faults with respect to their triggers, and concluded that a non-negligible share (15%) would not realistically elude testing. Our proposed criteria decreased the percentage of non-elusive faults in the faultload, improving its representativeness.

1 Introduction

Software Fault Injection (SFI) has emerged in the last decades as a valuable mean for assessing the impact of software faults on a system and for evaluating fault tolerance mechanisms [2]. A key property that SFI need to satisfy is the *representativeness* of injected faults, that is, the faultload should represent realistic faults experienced in the field [9, 21]. Only the appropriate choice of faults to be injected can assure an accurate evaluation of dependability metrics in the presence of faults (as in the case of *dependability benchmarking*), and an efficient testing of fault tolerance with respect to the extremely large space of faults. Unfortunately, representativeness is a very hard property to attain for *software faults*. They are difficult to characterize and to emulate since their root causes are related to human design errors [3]; this issue is exacerbated by the strong interaction of the faults with the surrounding code, and with the system as a whole. Nevertheless, they play an important role in the field of fault injection, since they are a well-known major cause of system outages [12].

Past efforts in this research area led to significant steps towards a representative model of software faults [7, 10]; they specify fault patterns that can be used to perturb the target system in a controlled way. Nevertheless, the use of representative fault patterns may not be sufficient; it can be argued that they alone do not guarantee realistic injected faults, because many other factors in the program affect representativeness. For instance, an injected fault could cause the immediate crash of the system, regardless of the submitted inputs. Although causing software failures is the goal of SFI (i.e., *fault acceleration*), a too simple activation condition (i.e., *trigger*) is not representative of real faults and thus undesirable in SFI, since a programmer would trivially find and fix the fault before the system release. In the dependability evaluation scenario, this can lead to a distorted picture of the perceived system dependability and to wrong design or selection decisions (e.g., selecting one system which is apparently more dependable than other, when in fact it might not be the case). Thus, it is important to select representative faults in a faultload.

It is worth mentioning that works such as [7, 10], although based on field studies on real software faults, could not capture the activation conditions of the software faults. In fact, these studies were based on the analysis of how real faults have been corrected, in order to infer the most typical bugs found in the field and understand what (in term of code pattern) makes a typical fault. Nevertheless, the reproduction of most frequent types of software faults based solely on its defining code patterns cannot guarantee the realism of the triggering conditions, thus cannot guarantee whether the injected faults are representative or not.

In order to put some light on this problem, this paper investigates how the representativeness of injected faults can be improved, by means of a set of criteria for selecting representative faults from a faultload. We evaluate the extent of the issue and the effectiveness of the proposed criteria in a real-world case study, using an existing SFI

technique (G-SWFIT, *Generic SoftWare Fault Injection Technique* [10]), and a complex and widely used open-source system (the MySQL DBMS). We analyze injected faults according to their ability to elude testing activities and remain hidden (we call this property *elusiveness*), as faults experienced in the field, which SFI aims to emulate, elude testing and go with the deployed product [20].

We performed an extensive SFI campaign, based on over 40 thousands injected faults and 2 million runs. We took advantage of the test suite used by MySQL developers to analyze fault elusiveness *under realistic conditions*. From the experimental results, we observed a non-negligible share of faults (15%) that are easily detected by the test suite. Such faults cannot be viewed as representative or interesting to dependability evaluation as they would most surely be identified by test cases. Since including these faults in the faultload biases the results of an experimental campaign, G-SWFIT (or any other SFI technique) can be improved by discarding these faults. In this respect, we deeply analyze the injected faults, in order to draw some general features to characterize them. The analysis reveals that the choice of target components actually affects the percentage of non-elusive faults in the faultload, because the extent of these faults varies among components. Therefore, we defined a set of criteria, based on software complexity metrics, to guide the selection of target components in which representative faults can be injected; our results confirm that using these criteria decreases the percentage of non-elusive faults in the faultload.

In section 2, we discuss related work. In section 3, we describe the research problem and the methodological approach adopted in this paper. Section 4 discusses the case study and the results from the SFI campaign. Section 5 discusses the use of complexity metrics to guide SFI. Section 6 concludes the paper.

2 Related work

The relationship between testing and fault injection has been explored in the past, but from an opposite perspective than this work. Here, we adopt testing to evaluate a fault injection technique in terms of fault representativeness. Past work focused on fault injection for generating test cases (in the hypothesis that, if a test case is effective against mutants, it will be able to detect real faults), i.e., *mutation testing* [13], and comparing testing strategies, i.e., *mutation analysis* [1]. These techniques do not aim at fault forecasting or fault removal in fault tolerant systems, thus injected faults, not necessarily representative, are simple program mutations to assess the effectiveness of tests.

There are several examples of dependability research works in the literature in which the representativeness of software faults is crucial. In [9], a dependability benchmark for web servers is proposed, to assess a set of metrics related to both performance (e.g., throughput, response time) and dependability (e.g., percentage of erroneous operations), with respect to software faults in the OS. Representative faults are needed to obtain a realistic estimate of measures (e.g., the expected performance degradation that may occur in operation), and to make systems comparable. In [9] faults are injected at run-time in the binary program, in order to reproduce the activation of faults during operation; although that work used a representative workload to exercise faults in a representative manner, this approach is not fully representative, because real faults are in the system before and during the entire execution of the target. In [21], a dependability benchmark is proposed to evaluate different DBMS configurations with respect to operator and software faults, in order to aid system administrators; in this case, a representative faultload is essential to identify the best configuration. The benchmark in [21] requires that faults are injected at given times after the system reaches a "steady state", in order to reproduce realistic conditions; however, since fault triggers are neglected by existing SFI techniques, it is still an open issue to reproduce software fault activation in a fully representative way. In [8], SFI in device drivers is adopted to classify 3 OSs and their failure modes with respect to different scales (i.e., availability, feedback to the user, stability), to enable the user to choose the best OS to be included into a system, according to his specific requirements. However, several faults (45.16%) are activated during the OS boot phase in [8], which are in contrast to field data studies [7,20], in which failures during startup are a small part (3.4%); therefore, not taking into account the fault trigger can have a significant impact on dependability assessment. In [17], complexity metrics were adopted for risk assessment, and in particular to estimate the distribution of faults across components; nevertheless, the representativeness of triggers is still neglected, thus affecting the confidence in risk estimates.

The problem of software fault representativeness was addressed for the first time in [7]. It proposed a set of rules for the injection of errors that emulate software faults, based on field data. However, these emulation rules are dependent on the availability of field data on real faults of the target system, which is normally not the case. This makes the technique very difficult to apply in practice, if not totally impossible.

Following studies evaluated existing techniques for emulating software faults, such as SWIFI and robustness testing [14–16], and demonstrated that existing techniques are not suitable for this purpose. To tackle these limitations, the study in [10] demonstrated that a unique fault distribution is common among several programs and thus generally applicable, and proposed G-SWFIT for precisely emulating faults at the executable code. This technique emulates the

Table 1. Most common fault types (from [10]).

Fault types	Description
MIFS	Missing "If(cond) { statements }"
MFC	Missing function call
MLC	Missing "AND EXPR" / "OR EXPR" in expression used as branch condition
MIA	Missing "If(cond)" surrounding statements
MLPA	Missing small and localized part of the algorithm
MVAE	Missing variable assignment using an expression
WLEC	Wrong logical expression used as branch condition
WVAV	Wrong value assigned to variable
MVI	Missing variable initialization
MVAV	Missing variable assignment using a value
WAEP	Wrong arithmetic expression used in parameter of function call
WPFV	Wrong variable used in parameter of function call

software fault classes most frequently observed in the field (Table 1) through a small set of *fault emulation operators*; they consist of a code change representing the class of fault and a code pattern identifying where that change can be realistically made in a program. Our work is built on top of these efforts to evaluate and improve the representativeness of injected faults.

3 Problem statement and research methodology

The problem of adequate *fault emulation* (described in section 1) is related to the two aspects that characterize software faults, namely the *fault type* and the *fault trigger* [6]. The fault type relates to the kind of change in the code to fix the fault. The fault trigger is a condition that allows the fault to surface; it can encompass user inputs, the internal system state, and the external environment such as the OS. An injected fault is deemed representative when its fault type and fault trigger are similar to the fault types and triggers most frequently occurring in the field. Although past work on fault injection focused on representativeness with respect to fault types, there is a lack of a way to assure the representativeness of the fault trigger. In G-SWFIT, fault types are well emulated, since its fault operators encompass more than 50% of fault types found in the field (Table 1), as demonstrated by a large field data study on 668 faults over 12 real-world applications [10]. Instead, fault triggers are not considered in G-SWFIT, since the available field data did not include any information about them. Therefore, G-SWFIT applies fault operators regardless of the way the faults will manifest themselves.

The first research problem arising from the above observations is whether emulated faults are triggered by representative activation conditions or not, since we aim to inject faults whose trigger is similar to triggers of field faults. Field data studies [11, 18] observed that pre-release defects (i.e., those found during system and function testing) are distributed in complex software differently than post-release defects, since post-release defects are more prone to be hidden where testing is less effective. This fact is relevant to our problem, since fault injection is commonly adopted during the late phases of the system's lifecycle, namely after or at the same time of function and system testing, for dependability evaluation purposes (section 2). Therefore, we investigate if injected faults are representative of faults encountered in this context, by analyzing if they can be triggered by testing activities.

In order to perform a rigorous analysis and to obtain meaningful results, it is necessary to ensure the realism of testing procedures. In fact, test effectiveness is influenced by many factors related to the testing process (e.g., which functionalities should be tested more thoroughly). In order to include these factors into the analysis, we adopted the real suite for function testing of a complex system. Although function testing is not the only mean to address software faults, it is the most significant contributor to detect faults in complex systems [5]. We systematically analyze injected faults, by executing the system under individual test cases from the test suite; each test case encompasses a different set of inputs. If the fault is activated only in a small subset of test cases, then the activation condition is a combination of events specific to that subset. In this case, the fault can be considered elusive, since its activation condition is subtle and testing could have missed it. We discriminate between elusive and non-elusive faults by counting the number of test cases that detected each fault; if more than 50% of test cases detect the fault, it is a non-elusive fault. Although the 50% threshold may appear arbitrary, it is the fairest choice, and we will demonstrate that this choice does not significantly affect results.

The second research problem is how to remove non-elusive faults from a faultload, to improve its representativeness. To this aim, we need to discriminate between elusive and non-elusive faults *a priori*, without requiring the availability of a test suite to evaluate the elusiveness of faults. A way to achieve this goal is to collect a set of features related to each fault, and to use statistical analysis techniques to characterize elusive (representative) faults. We consider software complexity metrics of the component in which the fault is injected, due to the relationship between complexity and software faults [4].

The approach adopted in this paper to answer to these two problems can be summarized as follows:

1. We generate a large set of faulty versions of the target system. Each version contains one fault;
2. We select a set of test cases from the original test suite of the system;
3. Each faulty version is executed several times, one for each test case;
4. Data about system failures are collected for each experiment; moreover, to gain additional insights, we collect information about the statement coverage of test cases;
5. We analyze the failure distribution to find non-elusive faults and evaluate their amount (**first research problem**);
6. We analyze software complexity metrics related to the location of each fault, in order to define criteria for identifying non-elusive faults *a priori* (**second research problem**).

4 Evaluation of fault representativeness

In this section, we first provide details about the case study and the experimental setup, and then we discuss the results we obtained.

4.1 Case study and experimental setup

MySQL is one of most used among all DBMSes, accounting for about 30% of installations among IT organizations[1]. The popularity of MySQL is also due to the overall adoption of Off-The-Shelf applications based on it for running services over the web. The wide use of MySQL in business-critical contexts makes it a representative target of a dependability benchmark [21].

The MySQL DBMS is also representative of complex software systems; it is made up of more than 700K Lines of Code (LoC) distributed among 2K files. Moreover, the test suite is provided along with the source code, which is actually used for regression and functional testing. Test cases are grouped based on the set of functionalities to test. In particular, we targeted the MySQL core, since it is the largest and most fundamental part of the DBMS. The core is in charge of managing threads and connections, SQL query parsing, optimization, and execution. The test suite provides 473 test cases for core functionalities. We carefully selected 50 test cases; this number is a trade-off between the time required for the experimental analysis, and the need of a sample large enough to be representative. Test cases were randomly sampled, and those cases too similar between them were discarded.

A fault injection tool[2] automated the fault injection process of this study. The tool analyzes and modifies the target program to inject software faults in its source code. It first analyzes a source code file to identify suitable locations where fault types in Table 1 can be applied. It then generates a set of faulty versions of the source file, each containing a single fault (Figure 1). The tool identified 384,650 fault locations; we injected a sample of 40,402 faults in those locations tested by at least one test case. It is worth mentioning that the injection of more than 40 thousand faults represents a quite extensive fault injection campaign.

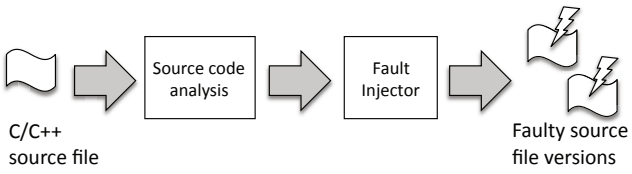


Figure 1. Software fault injection tool.

Figure 2 shows the experimental setup. Several faulty versions of the DBMS (each containing an individual fault) are preliminarily compiled. A program, namely Test Controller, was developed to execute every faulty version for each test case; in total, 2,020,100 experiments were executed. For each experiment, the results of the test case were collected, i.e., the crash of the DBMS, an incorrect answer to an SQL query, and the timeout of the test. Since we are only interested whether the test case is able to detect a given fault (i.e., to cause a failure), we did not take into account the type of failure mode. The experiments were distributed among 4 workstations equipped with an Intel Core2Duo 2.4GHz CPU, 4 Gb RAM, and a SATA 3 Gb/s NCQ disk.

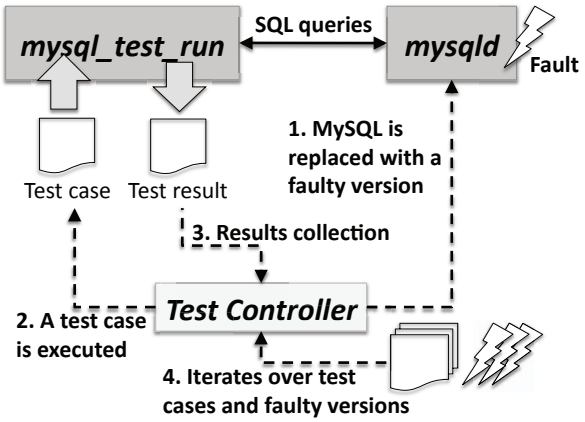


Figure 2. Experimental setup.

4.2 Result discussion

During the experimental campaign, 282,739 failures out of 2 million of experiments were observed. Since some test

[1] http://www.mysql.com/why-mysql/marketshare/
[2] Available at http://www.mobilab.unina.it/SFI.htm

cases proved not to be effective (they detect less than 1% of the injected faults) and would skew the results, we removed them from the test suite. After this operation, 40 test cases were kept, accounting for 281,027 failures. Failure occurrences for each fault and test case are shown in Figure 3.

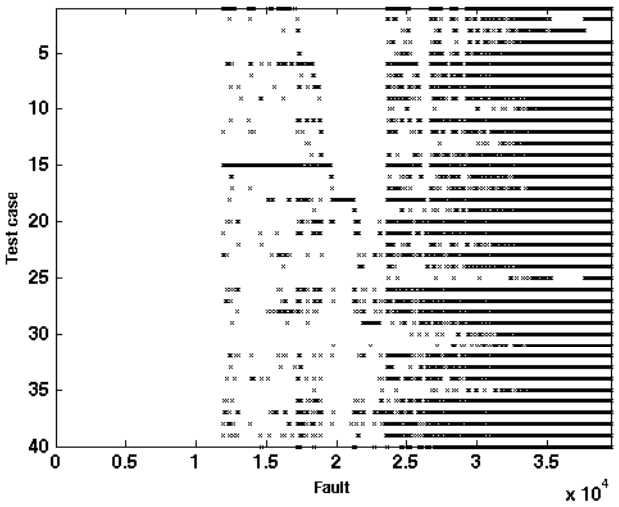
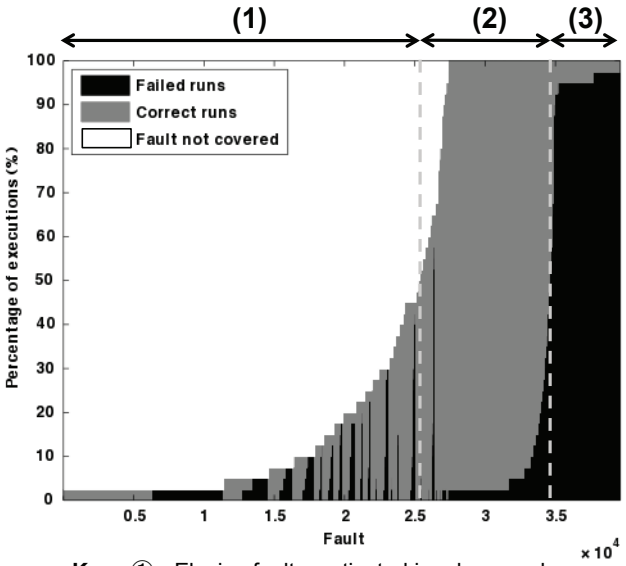


Figure 3. Experiments in which a failure occurred. Points represent experiments in which a given fault was activated by a given test case, causing a failure. Faults are ordered by number of failures observed.

In the figure, points represent an experiment in which a given fault was activated by a given test case, causing a failure; white areas represent experiments in which the test case was not able to detect the fault. Part of the faults are detected by most of test cases (right side of the figure); these faults should be considered as non-elusive, since they are easily detected by tests.

The histogram in Figure 4 provides a perspective from the point of view of faults, in which each vertical bar represents the percentage of failed executions for a given fault (on the horizontal axis). For ease of reading and for evaluating the amount of elusive and non-elusive faults, faults are ordered by the percentage of failures and correct executions, respectively. This enabled us to analyze the two reasons why several injected faults are elusive. A first subset of faults (63.60%) is detected by a small number of test cases since their location in the source code is not executed in the remaining test cases. This fact is representative of real faults that elude testing since the portions of code where they reside are hard to cover and exercise; the problem of generating test cases to achieve a high coverage in a limited amount of time often occurs in testing of complex systems [1].

Moreover, there is a second subset of faults (21.83%) that seldom manifest themselves even if their source code location is executed many times. They are also representative of



Key: ① Elusive faults, activated in a low number of executions (21.83%)
② Elusive faults, covered in a low number of cases (63.60%)
③ Non-elusive faults (14.57%)

Figure 4. Percentages of failures and correct executions for each fault. Faults are ordered by the percentage of failures and correct executions, respectively.

real faults, since they cause a failure only when the faulty location is executed under specific activation conditions, which can be missed during testing. For instance, the activation condition can be related to specific values took by input and state variables. This problem is related to the unfeasibility of exhaustive testing in complex systems [1].

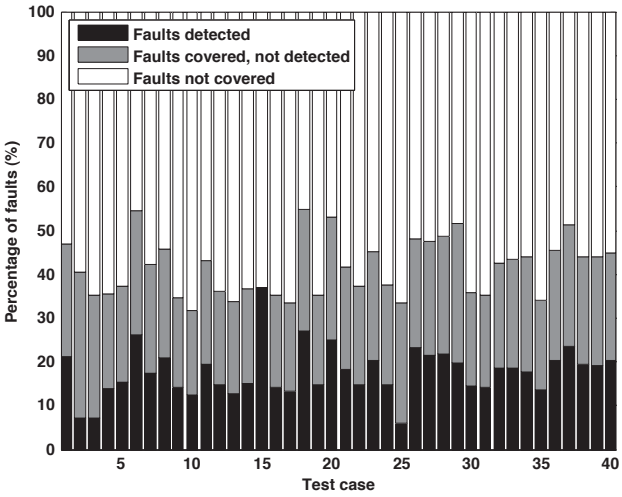


Figure 5. Percentages of failures and correct executions for each test case.

The percentage of faults detected and covered by each test case is shown in Figure 5. Because the percentage of failures is low for non-elusive faults, there are only are small differences in the number of detected faults between test cases (with few exceptions). This also means that the activation triggers of elusive faults are distributed among test cases; therefore, an SFI campaign requires a set of several inputs to trigger elusive faults. This is still an open issue that should be analyzed in future research.

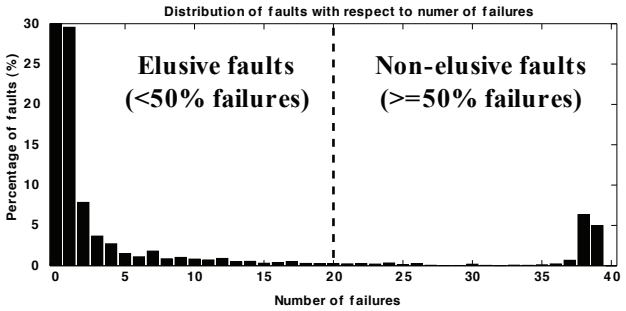


Figure 6. Distribution of faults with respect to the number of failures.

The last set of faults in Figure 4 contains non-elusive faults, which are easy to detect. Their extent can be observed in Figure 6, which shows the fault distribution with respect to the number of times they caused a failure. The majority of faults cause a small number of failures, with the predominance of faults causing 0 (30%) or 1 failure (29.56%). This result seems to support the thesis that faults injected by G-SWFIT are elusive and therefore representative; nevertheless, the percentage of non-elusive faults is non-negligible, and they cause a large number of failures (more than 35). In fact, 14.57% of faults caused a failure at least in the 50% of cases. It should be noted that choosing a threshold different than 50% does not significantly affect this result (see Figure 6), ranging from 12.73% (at least 75% of failures) to 19.86% (at least 25% of failures) of non-elusive faults.

4.3 Validation of results

In order to get more confidence in our conclusions, we challenged these results by means of a second experimental campaign. Our aim was to confirm (or not) that the low number of non-elusive faults is not due to the simplicity of the test cases but is really related to the nature of the faults. Therefore, we augmented test cases with an implementation of the TPC-C benchmark[3] as an additional workload. In this setup (Figure 7), TPC-C and a test case are executed at the same time on the same database, in order to analyze the effects of a more demanding workload (in terms of amount of data and operations rate).

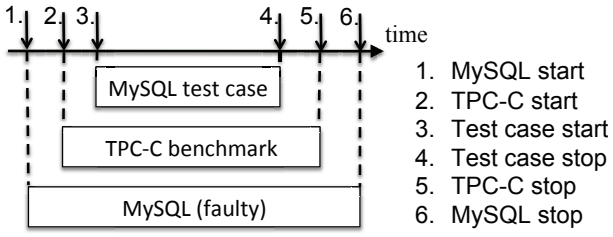


Figure 7. Experiment setup for the validation of results using TPC-C.

We selected a random sample of 668 faults from those causing 0 or 1 failure, keeping about the same proportion of Figure 6. As in the previous campaign, each faulty version has been executed several times, one for each test case. We considered only test cases (34 in total) not conflicting with TPC-C (e.g., some test cases forced a reboot of the DBMS).

Figure 8 shows the distribution of faults in the sample with respect to the number of failures, after the addition of TPC-C. During the experiments, only a small part of faults (1.35%) caused 2 or more failures, which were always triggered by TPC-C (0.45%) or by the joint execution of TPC-C and a test case (0.90%). Even with TPC-C benchmark, which represents a quite demanding workload for the MySQL DBMS, 98.65% of faults in the sample revealed an elusive nature.

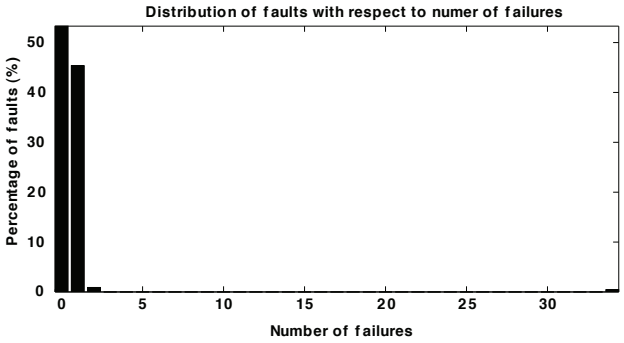


Figure 8. Distribution of a sample of elusive faults with respect to the number of failures (under test cases and TPC-C running at the same time).

5 Improving fault representativeness

In this section we address our second research problem. Our goal is to tell apart elusive from non-elusive faults before any actual fault injection. This ability will allow us to filter non-elusive faults from the faultload and thus increase its representativeness.

[3]http://jtpcc.sourceforge.net/

5.1 Analysis of fault types and components

In order to understand the key features of the injected faults, we deeply analyze them with respect to two important factors: their fault type and the component in which they are located. We hypothesize that these factors can affect the elusiveness of injected faults; if this hypothesis is true, faultload representativeness can be improved by including specific fault types or components.

In Figure 9 and Figure 10 we compare the distributions of elusive (including subsets 1 and 2 in Figure 4) and non-elusive faults across fault types (Table 1), respectively. Apparently, fault elusiveness has little influence on the shape of the distribution. To quantitatively evaluate if differences between distributions are statistically significant (i.e., they are not caused by random factors) we use the two-sample Kolmogorov-Smirnov (KS) test [19], which is a procedure for evaluating if two samples are drawn from the same underlying probability distribution. On the basis of the KS test (p-value $= 0.4333$), the null hypothesis that the two samples are from the same distribution cannot be rejected, i.e., differences are not statistically significant. Therefore, the fault type alone cannot discriminate non-elusive faults. It should be noted that this result is not obvious, since the fault type affects where faults are injected (e.g., a fault type could have occurred more frequently in code easy to test).

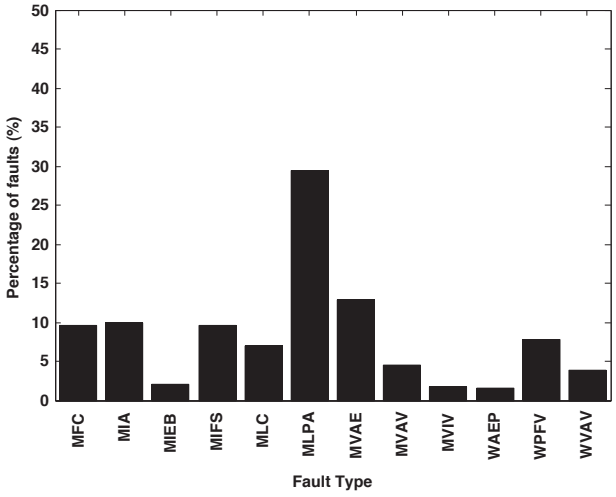


Figure 9. Elusive faults distribution across types.

In order to analyze whether the elusiveness of injected faults is related to a specific system component, we applied the same procedure on the distributions across components of elusive and non-elusive faults (Figure 11 and Figure 12). We use the term "component" to refer to a source file of MySQL core. Source files are adopted by C/C++ developers to group related procedures/classes; by defining source files as components, we take into account the influence of both individual procedures/classes and the whole source file (e.g., the use of a shared variable by related procedures). Faults were injected in 96 components.

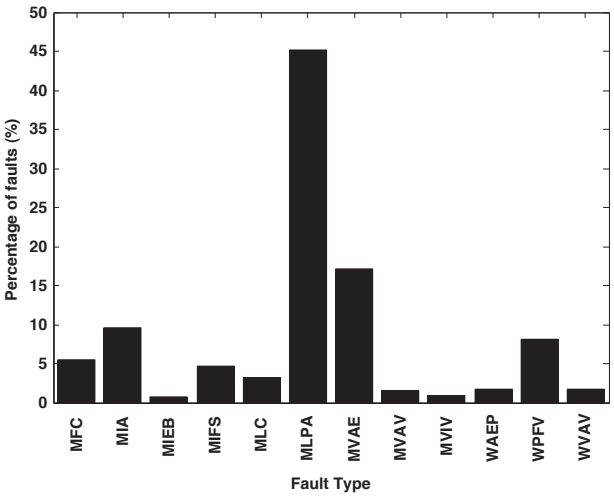


Figure 10. Non-elusive faults distribution across types.

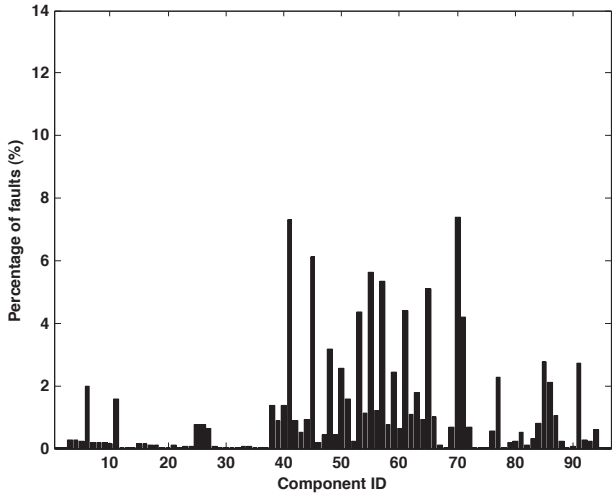


Figure 11. Elusive faults distribution across components.

The two distributions are noticeably different; the KS test confirms this observation, since the null hypothesis can be rejected with a confidence level greater than 99% (p-value $= 7.2862 \cdot 10^{-7}$). This result suggests that the elusiveness is influenced by the target component; it is probably due to the interplay between faults and the surrounding code. Therefore, we will analyze if the features of the target component can be exploited to discriminate between elusive and non-elusive faults.

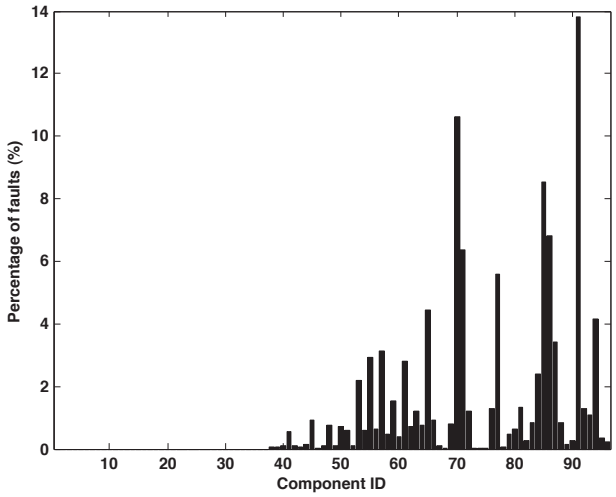


Figure 12. Non-elusive faults distribution across components.

Table 2. Software complexity metrics.

Name	Description
AvgCyc	Average cyclomatic complexity of functions in the component
MaxCyc	Maximum cyclomatic complexity of functions in the component
LoC	Number of lines of code
OutDepC	Dependencies of the component ("fan-out"), number of components
OutDepR	Dependencies of the component ("fan-out"), number of references
InDepC	Dependencies on the component ("fan-in"), number of components
InDepR	Dependencies on the component ("fan-in"), number of references

5.2 Filtering non-elusive faults

From the results of the previous section, we observe that the distributions of elusive and non-elusive faults across components are significantly different. In fact, the relative percentage of non-elusive faults tends to be higher in some components (on the right side in Figure 11 and Figure 12). Therefore, if these components are identified before fault injection, they can be discarded from the analysis in order to reduce the relative percentage of non-elusive faults. In practical terms, this means that fault which locations fall into these components will be filtered out from the fault-load. This reduces the skew of non-elusive faults on the results and the loss of time in performing misleading experiments. To characterize the components, we collected a set of complexity metrics. We investigate complexity metrics since the complexity of a component (e.g., the number of paths) and its relationships with other components affect the ability to thoroughly test the component; therefore, metrics can reveal where faults may hide from tests.

We selected a set of complexity metrics (Table 2) that can be easily collected from a software artifact (assuming source code availability), and thus are available to the tester before SFI: they take into account the cyclomatic complexity (*AvgCyc*, *MaxCyc*), the code size (*LoC*), and dependencies on symbols (e.g., a function) that are imported (*InDepR*, *InDepC*) or exported (*OutDepR*, *OutDepC*) by the component. In order to extract the hidden relationship between complexity metrics and components containing non-elusive faults, we adopted a technique commonly used in data mining problems, namely *decision trees* [22]. A decision tree is a hierarchical set of questions that are used to classify an element. In our study, questions are based on complexity metrics (for instance "Is LoC greater than 340?"), and the components are the elements to be classified. A decision tree has been preferred over other classifiers because it is simple to interpret, and it provides insights on why the percentage of non-elusive faults is higher in some components.

Before training the decision tree, we split components in two classes, respectively those in which the percentage of non-elusive faults is negligible (C1), and the remaining ones (C2), in order to discriminate components with most non-elusive faults. In fact, we observed from a preliminary analysis that there are some differences in the complexity metrics when the percentage of non-elusive faults is higher than 5%. The dataset is composed by 96 components in two classes (50 and 46 components, respectively).

A decision tree (Figure 13) is obtained from the whole dataset using the C4.5 algorithm [22]. The training algorithm iteratively splits the dataset in two parts, by choosing an attribute (i.e., complexity metric) and a threshold that most effectively classifies the training data; this operation is then repeated on the subsets. The root and inner nodes represent questions about complexity metrics being over than a threshold value, and leafs represent class labels. To classify a component, a complexity metric of the component is first compared to a threshold specified in the root node, to choose one of the two children nodes; the same is repeated for each node, until a leaf is reached. Figure 13 shows the number of components of the training set classified by each leaf, and the number of components wrongly classified.

By analyzing the structure of the tree, we can notice that *InDepR* (in the root node) is the metric most relevant to non-elusive faults; it represents the number of times a reference to a symbol of the component (e.g., a function) is found in other components. In fact, *InDepR* is > 22 for most components with non-elusive faults (25 out of 46), and *InDepR* ≤ 22 for most components with a small percentage of

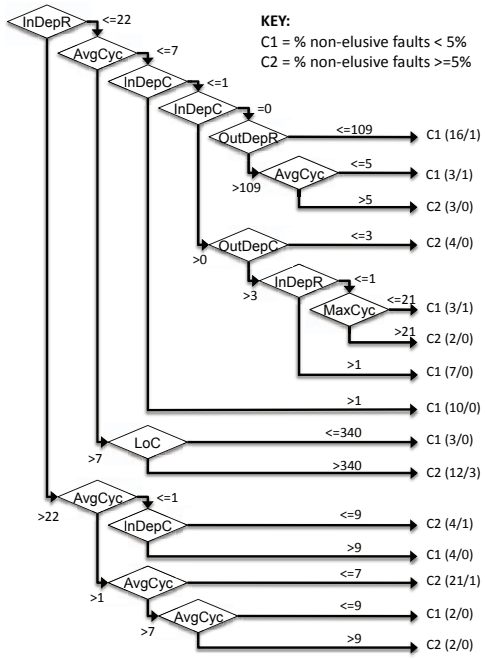


Figure 13. Decision tree based on components complexity metrics.

non-elusive faults (42 out of 50). This fact implies that components with a small *InDepR* contain a small percentage of non-elusive faults. This can be due to the higher "exposure" of faults in a component with high *InDepR* (a fault in the component can propagate to several other components), as in the case of the "sql_string.cc" component (*InDepR* $= 1168$), which provides utility functions for string handling. Another metric relevant to non-elusive faults is *AvgCyc* (in both children nodes of the root). When *InDepR* is low and *AvgCyc* is > 7, components have a noticeable percentage of non-elusive faults (9 out of 46), and when *InDepR* is high and *AvgCyc* is ≤ 1, components have a low percentage of non-elusive faults (5 out of 50). For instance, the "sql_table.cc" from class C2, which provides support for managing SQL tables, is associated with $AvgCyc = 11$. Therefore, when filtering components for fault injection, those with low *InDepR* or low *AvgCyc* should be selected, since they contain a lower percentage of non-elusive faults.

Until this point, we analyzed the decision tree to identify the key features of components containing non-elusive faults. However, we also want to investigate if the decision tree can guide the selection *a priori* of components that should be filtered out (i.e., without knowing beforehand the amount of non-elusive faults in the components). In this scenario, complexity metrics of a component are the input of the decision tree, to decide if the component should be considered (class C1) or not (class C2) in SFI.

To obtain an initial evaluation of the non-elusive faults that can be removed, we simply classified all the components in the dataset using the decision tree. This evaluation is optimistic, since the whole dataset was also used in the training phase; however, it provides an upper bound to the effectiveness that can be achieved. In this case, the tree correctly classifies 91.67% of the components. If we include in the faultload only faults in components not filtered (class C1, 48 components), we obtain 10,312 faults of which 2.55% are non-elusive faults; this percentage is significantly lower than 14.57% when no filtering is performed. However, including only components with a negligible amount of non-elusive faults comes at the cost of a reduced number of injected faults (26.02% of the whole set of faults). This does not seem to be a limitation in complex software systems, in which the number of fault locations is high enough to still obtain statistically significant results. The trade-off between number of injected faults and percentage of non-elusive faults is caused by the presence of both elusive and non-elusive faults in the same component; techniques for more fine-grained identification of non-elusive faults are needed to obtain a better trade-off.

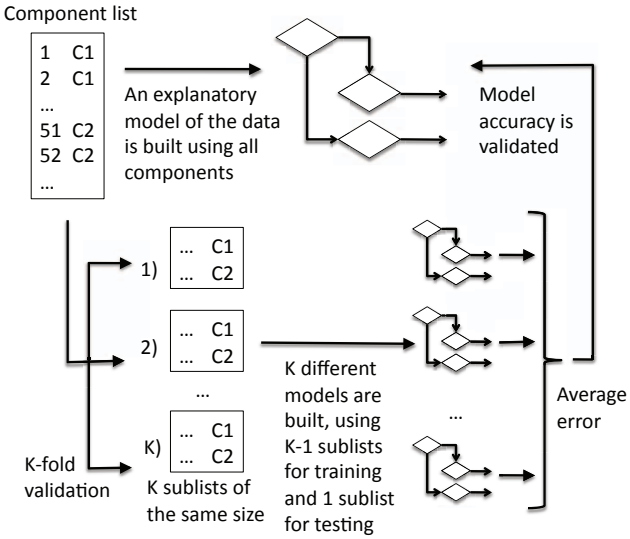


Figure 14. Model validation.

Moreover, we evaluated the accuracy of the model when we use it to discriminate components not included in the dataset. In fact, the threshold values and the tree structure may not be suited and need some tuning for each specific system. To this aim, we used the *k-fold cross validation technique* [22] (Figure 14); we split the dataset in 10 folds, and then we evaluated the classification accuracy on each fold when the remaining 9 folds are used for training a model. This approach provides an estimation of the model accuracy when it used to analyze components not included in our dataset. In this case, the tree is able to correctly classify 61.46% of components; using the decision tree to ana-

lyze components not in the dataset, the expected percentage of remaining non-elusive faults is 10.75%, which is still lower than the percentage of non-elusive faults when no filtering is performed. Although this result does not guarantee the effectiveness of the model with respect to all complex software systems, it increases our confidence in the use of complexity metrics for filtering non-elusive faults. A more throughout validation on more software systems is a future research direction. Moreover, some kind of adaptation to the specific system (e.g., by means of clustering techniques to tune thresholds in the model) can be potentially exploited to further improve the effectiveness of filtering.

6 Conclusion

This paper investigated how the representativeness of injected faults can be analyzed and improved, by means of a set of criteria for selecting representative faults. We evaluated the extent of the issue and the effectiveness of the proposed criteria in of a real-world case study, in the context of an existing SFI technique (G-SWFIT) and a complex and widely used open-source system (MySQL). Faults were analyzed with respect to their ability to elude testing and stay hidden (*elusiveness*). Experimental results show that 85.43% of injected faults elude more than 50% of test cases, thus supporting G-SWFIT as an effective way to emulate software faults. To cope with the remaining non-elusive faults (14.57%), we proposed the use of complexity metrics for filtering out components with a large extent of non-elusive faults. We observed a relationship between metrics and the occurrence of non-elusive faults (in particular, *InDepR* and *AvgCyc*). We exploited this relationship to identify components containing non-elusive faults; using a very simple classifier, the expected percentage of non-elusive faults was reduced to 10.75%, which can be potentially improved down to 2.55%. We believe that the refinement of faultloads made possible through this elimination of non-elusive faults will improve the representativeness of results obtained in dependability assessment techniques that use the injection of software faults.

Acknowledgment

This work has been partially supported by the project "CRITICAL Software Technology for an Evolutionary Partnership" (CRITICAL-STEP, http://www.critical-step.eu), Marie Curie Industry-Academia Partnerships and Pathways (IAPP) number 230672, within the context of the Seventh Framework Programme (FP7), and by the Italian Ministry for Education, University, and Research (MIUR) within the framework of the project "Dependable Off-The-Shelf based middleware systems for Large-scale Complex Critical Infrastructures" (DOTS-LCCI, http://dots-lcci.prin.dis.unina.it), DM1407.

References

[1] J. Andrews et al. Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria. *IEEE TSE*, 2006.
[2] J. Arlat et al. Fault Injection for Dependability Validation: A Methodology and Some Applications. *IEEE TSE*, 16(2), 1990.
[3] A. Avižienis et al. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE TDSC*, 2004.
[4] V. Basili and B. Perricone. Software Errors and Complexity: An Empirical Investigation. *Comm. ACM*, 27(1), 1984.
[5] R. Chillarege and K. Prasad. Test and Development Process Retrospective—a Case Study using ODC Triggers. In *Proc. DSN*, 2002.
[6] R. Chillarege et al. Orthogonal Defect Classification—A Concept for In-Process Measurements. *IEEE TSE*, 18(11), 1992.
[7] J. Christmansson and R. Chillarege. Generation of an Error Set that Emulates Software Faults based on Field Data. In *Proc. FTCS*, 1996.
[8] J. Durães and H. Madeira. Characterization of Operating Systems Behavior in the Presence of Faulty Drivers through Software Fault Emulation. In *Proc. EDCC*, 2002.
[9] J. Durães and H. Madeira. Generic Faultloads Based on Software Faults for Dependability Benchmarking. In *Proc. DSN*, 2004.
[10] J. Durães and H. Madeira. Emulation of Software Faults: A Field Data Study and a Practical Approach. *IEEE TSE*, 2006.
[11] N. Fenton and N. Ohlsson. Quantitative Analysis of Faults and Failures in a Complex Software System. *IEEE TSE*, 26(8), 2000.
[12] J. Gray. Why Do Computers Stop and What Can Be Done About It? Technical Report TANDEM TR-85.7, 1985.
[13] W. Howden. Weak Mutation Testing and Completeness of Test Sets. *IEEE TSE*, 1982.
[14] T. Jarboui et al. Analysis of the Effects of Real and Injected Software Faults: Linux as a Case Study. In *Proc. EDCC*, 2002.
[15] H. Madeira, D. Costa, and M. Vieira. On the Emulation of Software Faults by Software Fault Injection. In *Proc. DSN*, 2000.
[16] M. Moraes et al. Injection of Faults at Component Interfaces and Inside the Component Code: Are They Equivalent? In *Proc. EDCC*, 2006.
[17] M. Moraes et al. Experimental Risk Assessment and Comparison Using Software Fault Injection. In *Proc. DSN*, 2007.
[18] T. Ostrand and E. Weyuker. The Distribution of Faults in a Large Industrial Software System. In *Proc. ISSTA*, 2002.
[19] D. Sheskin. *Handbook of Parametric and Nonparametric Statistical Procedures*. 2004.
[20] M. Sullivan and R. Chillarege. Software Defects and their Impact on System Availability—A Study of Field Failures in Operating Systems. In *Proc. FTCS*, 1991.
[21] M. Vieira and H. Madeira. A Dependability Benchmark for OLTP Application Environments. In *Proc. VLDB*, 2003.
[22] I. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. 2005.