This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2018.2850755, IEEE Transactions on Software Engineering

Analyzing the Effects of Bugs on Software Interfaces

Roberto Natella, Stefan Winter, Domenico Cotroneo, Neeraj Suri

Abstract—Critical systems that integrate software components (e.g., from third-parties) need to address the risk of residual software defects in these components. Software fault injection is an experimental solution to gauge such risk. Many error models have been proposed for emulating faulty components, such as by injecting error codes and exceptions, or by corrupting data with bit-flips, boundary values, and random values. Even if these error models have been able to find breaches in fragile systems, it is unclear whether these errors are in fact representative of software faults. To pursue this open question, we propose a methodology to analyze how software faults in C/C++ software components turn into errors at components' interfaces (*interface error propagation*), and present an experimental analysis on what, where, and when to inject interface errors. The results point out that the traditional error models, as used so far, do not accurately emulate software faults, but that richer interface errors need to be injected, by: injecting both fail-stop behaviors and data corruptions; targeting larger amounts of corrupted data structures; emulating silent data corruptions not signaled by the component; combining bit-flips, boundary values, and data perturbations.

Index Terms—Dependability; Fault Injection; Software Fault Tolerance; Error Propagation; Software Components; Error Models

1 Introduction

The reuse of software components, either legacy or *off-the-shelf* (OTS) from third-parties, is a fundamental practice for cost-efficient software development, even in critical systems [1], [2], [3]. However, reused components also bring the risk of introducing more *software faults* (i.e., bugs), since their internal quality and reliability are unknown [4], [5], and since they are used in a new context unanticipated by their developers [6], [7]. Software fault injection is a solution to gauge the risk of unreliable components, by emulating software faults in the components to assess the fault-tolerant behavior of the overall system [8], [9], [10].

The injection of realistic component faults is a key, yet open problem to obtain meaningful results from software fault injection. Many existing techniques are limited to simple forms of injections, such as by forcing process crashes or API call failures signaled with an error code or exception [10]. However, these approaches only address a limited part of the problem: in fact, empirical studies in open-source and commercial software suggest that software faults are more subtle and virulent than assumed by these fault injection techniques. Tan et al. [11] analyzed 2k+ bugs in three large, popular open-source projects (the Linux kernel, the Mozilla suite, and the Apache web server) and found that the dominant share was represented by *semantic* bugs, i.e., "*Inconsistencies with the requirements or the programmers' intention*" (excluding synchronization and memory management bugs), such as missing corner cases and features in an algorithm: in most of the cases, these faults do not simply lead to crashes but impact on the functional correctness of the software [11, sec. 5]. Other empirical studies, based on failure data collected from operational systems [9], [12], emphasize that the impact of software faults is not limited to unavailability (e.g., the system stops and explicitly signals the failure), but also affects semantic correctness (e.g., data errors, undefined protocol states, etc.).

Therefore, to inject a richer set of realistic software faults, the current state-of-the-art techniques adopt *code mutation* [13], [14]. This approach has been traditionally used for mutation testing [15], whose goal is to assess that test cases are able to cover and trigger the buggy statements of a program. However, code mutation is inefficient for the purposes of software fault injection: mutants are often *dormant* (i.e., they are not triggered and do not change the behavior of the injected component), and thus do not exercise fault-tolerance mechanisms [9], [16]. Moreover, OTS components (which are the main target of risk assessment experiments using fault injection) are usually only distributed as binary code, which imposes technical limitations to code mutation (e.g., at precisely identifying statement boundaries in the presence of compiler optimizations) [17], [18].

To overcome these limitations, this paper addresses the problem from the perspective of the *interface data* that are used by software components to interact with each other. Examples are data returned from API calls and shared global variables, and any other data pointed by them. Interface data represent the component "*surface*" that is exposed to users: as far as a software system and its fault-tolerance mechanisms are concerned, a fault inside a component propagates its effects through the unavailability or corruption of interface data. We name these effects as *interface errors*. Our driving idea is that these data represent a favorable target for injection: the direct injection of interface errors (i.e., replacing the original interface data with incorrect values) can emulate software faults in a more efficient way than code mutations, which, instead, generate interface errors as

- *R. Natella and D. Cotroneo are with the DIETI dept., Federico II University of Naples, Italy. E-mail: {roberto.natella,cotroneo}@unina.it*
- *S. Winter and N. Suri are with the DEEDS Group, TU Darmstadt, Germany. Email: {sw,suri}@cs.tu-darmstadt.de*

0098-5589 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2018.2850755, IEEE Transactions on Software Engineering

an indirect by-product of mutants. Injecting interface errors avoids wasting experiments' time on dormant mutations, and the technical limitations of binary code mutation when the source code is lacking.

This paper presents a new experimental methodology and techniques for analyzing, in quantitative terms, how software faults turn into interface errors in C/C++ components. The methodology injects software faults in a component, traces the evolution of interface data at run-time, and identifies interface errors by looking for deviations of the fault-injected execution from the fault-free one (i.e., interface errors). Then, the methodology characterizes the interface errors in terms of extent, signaling by detection mechanisms, and recurring patterns. Our focus is on C/C++ software, as these languages are predominant in safety-critical control systems and systems software. Flight control software, for instance, is commonly implemented in these languages [19], [20], [21], [22], as is automotive software, for which fault injection is among the test techniques recommended by the safety standard ISO 26262 [23], [24].

We applied this methodology on C and C++ components from a set of 10 popular benchmark programs from different application domains, in order to get insights on actual interface errors and how to inject them. To this aim, we compared interface errors from these components with respect to existing software error models (such as, injecting boundary and out-of-range values, or bit-flipping them), which previous research derived either from conventional wisdom [4] or from hardware fault models [25], [26]. From the experiments, we found that these error models are not suitable for software faults, and we identified guidelines for defining more accurate interface error models. Relevant findings include:

1) Interface errors are a frequent effect of software faults in software components, as they occurred in 31.8% of all experiments (55.6% of experiments not including dormant faults). Thus, injecting simple fail-stop behaviors (such as component crashes) is not sufficient to emulate the effects caused by software faults;
2) Interface errors corrupt a significant share of interface data (up to several MBs of interface data in our experiments). Thus, software faults cannot be adequately emulated with traditional, hardware-oriented error models, which are limited to corrupting few bits or bytes. Instead, error injection should corrupt more extensive areas of interface data;
3) Interface errors are often *silent* (45.8% of experiments) as they are not explicitly signaled by the faulty component (e.g., through exceptions or error codes). Moreover, interface errors also occur even when the component had raised exceptions/return codes (in 16.0% of our experiments). Thus, the simple injection of exceptions or error codes does not suffice to represent the impact of faulty software components, but it must be complemented with the injection of interface errors;
4) Interface errors are not random, but tend to follow regular *patterns*. We found that the interface errors follow some of the error types traditionally adopted by error injection techniques (e.g., boundary values); however, other traditional error types (e.g., bit-flips) seldom occur. Moreover, these types do not suffice to represent interface errors, but need to be complemented with the injection of data perturbations, as most of the corrupted data deviated from the correct ones by a small offset.

The paper is structured as follows. Section 2 reviews related work on error propagation and fault injection. Section 3 poses the research questions for this study. Section 4 describes the proposed methodology. Section 5 presents the experimental analysis. Section 6 discusses the threats to validity. Section 7 concludes the paper with a discussion of the results and perspectives on future work.

2 Related work

Previous work on the reliability of component-based systems can be divided into *modelistic* and *experimental* approaches. The modelistic approaches analyze the *architecture* of component-based systems, based on the probability of failures of the individual components [27]. They are aimed to identify the components with the highest impact on reliability, in order to focus on them more engineering and testing efforts, such as by introducing executable assertions and wrapper (i.e., sanity checks to detect and/or discard problematic components' inputs/outputs) and other error detection and recovery mechanisms. Typically, modelistic approaches use *Markov models* (e.g., where the states represent the modules or stages that can be reached by the control flow), or reuse existing design specifications in UML [28] and AADL [29] diagrams (annotated with failure probabilities) and automatically convert them to stochastic models.

A critical restriction of early models has been the assumption of *independence* of software components' failures: thus, the latest reliability models have been incorporating probabilities of *error propagation* across components [30], [31]. Popic et al. [32] derived error propagation probabilities from UML diagrams (sequence, use case, deployment). Cortellessa and Grassi [33] and Mohamed and Zulkernine [34] further refined the model, by considering that the errors can be "masked" along propagation paths and not lead to a system failure; and evaluate the sensitivity of reliability with respect to the error propagation probability. Later work incorporated omission and performance failures [35], and generalized to components with multiple failure modes [36].

All these studies remarked that experimental approaches, based on *fault injection*, have a critical role for applying architecture-based models in practice, such as to derive error propagation probabilities and the coverage of error detection and recovery [37], and to assess the accuracy of reliability models [38]. Hiller et al. [39], [40] developed an experimental fault injection framework to analyze error propagation across components, by applying corruptions on component inputs and computing metrics for guiding design trade-offs, including: the probability of propagation *from system inputs to component inputs*; *from component inputs to its outputs*; and *from component outputs to system outputs*. Voas et al. [8] proposed a *certification scheme* for OTS components, which evaluates the ability of a system to survive to residual faults in an OTS component by injecting errors at component interfaces.

However, a fundamental problem for applying these approaches is to inject corruptions that are representative

0098-5589 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2018.2850755, IEEE Transactions on Software Engineering

of software faults. The early work focused on corrupting program code and its internal data (rather than injecting at component interfaces), by *bit-flipping* the contents of individual bits or bytes, with the assumption that these errors were representative of both hardware and software faults [9], [25], [26]. However, the experimental analysis of Madeira et al. [41] showed that bit-flipping program internals is only suitable for emulating the simplest classes of software faults, since more elaborate corruptions are needed to emulate faults that span over several statements. Thus, they developed *G-SWFIT* (Generic Software Fault Injection Technique) [14] to inject *code mutations*, based on the most common programming bugs found in hundreds of bug-fixes from open-source software projects. Unfortunately, the utility of code mutation is limited by the problems of inefficiency (i.e., dormant injections) and of the accuracy of mutations if the source code of the target is not available [18].

The injection of *interface errors* is often considered a more viable alternative for emulating components' faults. The existing approaches differ with respect to the *error models* (i.e., the error patterns that are injected), which can find different robustness problems and have different cost in terms of efficiency (e.g., number of fault injection experiments generated by the error model) and implementation complexity (e.g., human effort to develop an injector) [42], [43]. The error models, and related tools, include:

Bit-flipping and fuzzing. Examples are the studies by Barton et al. [44] and Arlat et al. [5], which respectively injected random values (*fuzzing*) and bit-flips on the input parameters of OS system utilities and system calls, to evaluate their robustness against faulty users and applications.

Data-type based. Similarly, *BALLISTA* [4] aimed at evaluating the error handling of POSIX system calls with respect to invalid parameters, by adopting a *data-type based* approach: for each of the 20 data types from the C language and the POSIX standard (such as `size_t`, `mode_t`, etc.) used for the input parameters of 233 POSIX system calls, BALLISTA provides a pool of invalid values to generate test cases. The invalid values were derived from the testing literature [45] or based on personal experience, including boundary and special values such as zero, negative one, maximum/minimum representable values, pointers to nonexistent memory, lengths near virtual memory page size, and pointers to heap-allocated memory.

Data perturbations. The *perturbation analysis* technique by Voas et al. [46], [47] corrupts the internal variables of a program statement, to identify which statements are most likely to propagate errors to the rest of the program. The variable is overwritten with a corrupted value that is close to the original one (e.g., following a uniform distribution whose mean is the original value).

Erroneous return values. The *FIG* and *LFI* tools [48], [49] inject *error codes* that may be returned from system libraries to user applications, for testing error handling code. For example, these error codes can be returned in case of excessive load (e.g., failed memory allocation), hardware faults (e.g., disk I/O errors) and software faults (e.g., the system call detects that it has been invoked with incorrect parameters).

These studies recognized that defining generically-applicable error models for interface error injection as an important, yet open research problem [4], [5]. Thus, Moraes et al. [50] and Jarboui et al. [51] investigated the representativeness of these interface error models, in the context of three complex components (the Linux kernel, an embedded software, and an object-oriented DBMS). Each of them performed two distinct experimental campaigns, by injecting (i) error injections at component's interfaces (bit-flips, and boundary and invalid values, such as NULL pointers), and (ii) faults injected inside the component (respectively, using code mutations [50] and known bugs [51]). These studies found a gap between interface injections and component-internal injections, in terms of failure modes exhibited by the target (e.g., the system exhibits different percentage of crashes or invalid application results). For example, interface injections on Linux system calls (i.e., at a high-level layer of the OS architecture) were not representative of faults inside the Linux device drivers (i.e., at a low-level layer of the OS architecture) [51]. These previous studies [50], [51] motivate us to reduce this gap, by establishing a relationship between component-internal faults and the interface errors caused by them, in order to provide guidance for more representative interface error injections.

In this work, we develop a new methodology for analyzing error propagation in software components. Previous tools of this kind were presented by Kao et al. [52] and Chandra and Chen [53]. *FINE* [52] inserted probes within SunOS to keep track of key kernel variables and function calls, in order to detect symptoms of error propagation inside the OS, to be used in a Markov reward model of OS performance under faults. Chandra and Chen [53] used a virtual machine environment (*SimOS*) to analyze the effectiveness of the transaction mechanisms of the Postgres DBMS at preventing error propagation to stable storage, by evaluating the amount of corrupted memory words, the fault latency (i.e., how long the process runs after the fault is activated), and occurrences of fail-stop violations (i.e., the DBMS writes wrong data on storage before stopping). Compared to these studies, this paper presents a methodology that: (i) does not rely on experts' knowledge to insert probes in "important" variables and functions, as in FINE [52]; (ii) pinpoints corruptions that specifically affect interface data, while SimOS [53] analyzes the target as a "black-box" and does not distinguish between interface data and internal component data (where only the former is visible from outside the component). Moreover, compared to our previous work [54], the methodology leverages debugging mechanisms to obtain more information about corruptions, allowing us to investigate more in depth our initial research questions (e.g., about the extent and the signaling of corruptions) and to address new ones (e.g., recurring patterns in interface corruptions). Finally, this work analyzes a larger amount of programs (10) and faults (10k+) than any previous work on error propagation.

3 Problem Statement

In a broad sense, this paper investigates the relationship between faults in software components, and the corresponding errors at the interfaces of the component. An understanding of this relationship allows the definition of more efficient strategies for software fault injection by replacing code

0098-5589 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2018.2850755, IEEE Transactions on Software Engineering

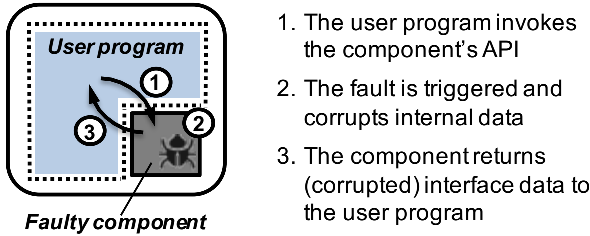


Fig. 1: Error propagation through interface errors.

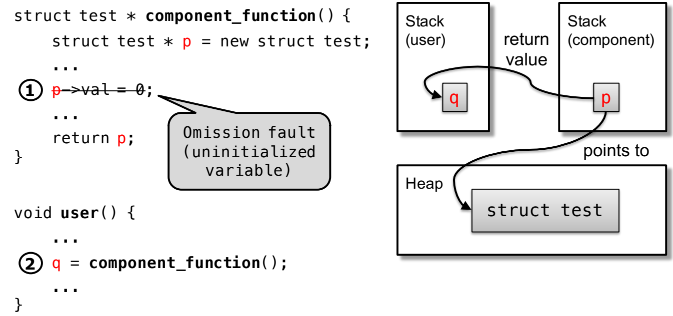


Fig. 2: Interface error on a component-allocated area.

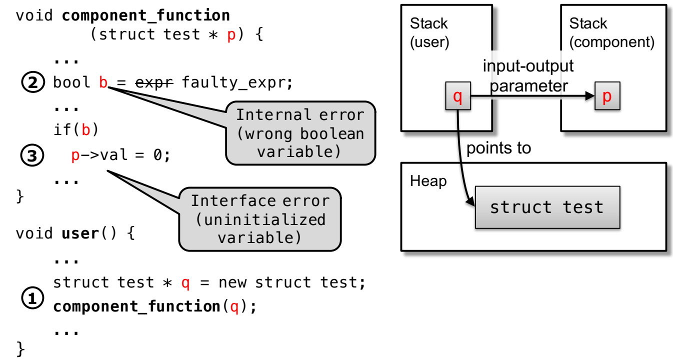


Fig. 3: Interface error on a user-allocated area.

mutations, which are difficult to trigger and to inject in OTS software, with equivalent interface error injections. In the following, we provide definitions and derive more specific research questions around this problem.

3.1 Overview of interface error propagation

The general relationship between faults, errors, and failures is showed in Figure 1. Our usage of these terms follows the taxonomy of Avizienis et al. [55]. The *target component*, which is a part of a broader system, is a software element that is assumed to be faulty in a software fault injection experiment. The target component provides an interface to an *user program*, in the form of a set of API functions and related data structures. The *interface data* consist of any data that is visible both to the target and user components when an API function is invoked (*step 1*). The API function provides both new data through the return value of the function, and updates the data structures that are passed as input/output parameters by the caller; these data include both primitive and complex types, and any other data pointed by the input/output parameters. During this invocation, software faults are triggered (*step 2*) and result in *errors* inside the component (e.g., corruption of internal component data). When the invocation terminates (*step 3*), the interface data can be in a corrupted state, as an effect of the *propagation* of such internal errors, thus producing *interface errors*. The software can then experience a *failure* (e.g., a crash or an incorrect output) as a result of the corruption.

Fig. 2 to 4 show the possible propagation paths for interface errors in the case of functions exposed by a software component (e.g., library functions and classes) and invoked by a (*user*) program. The examples are based on C/C++, which is the focus of this work, but the general approach applies for any type of software composition where components exchange data through shared data structures.

The first scenario (Figure 2) consists in the corruption of a data structure that the component (①) dynamically allocates on the heap. Due to a programmer's omission, the data are not initialized as the user program would expect. The corrupted data structure represents an interface error, as it survives the scope of the component invocation (②) and is returned to the user program through a pointer return value (e.g., allocated on the stack, depending on calling conventions).

Figure 3 depicts a similar case, in which a data structure is allocated by the user program (①), passed to the component through a pointer interface parameter, and corrupted during the component invocation due to an incorrect boolean expression (②). The figure emphasizes the difference between an internal error (i.e., the b variable internal to the component function) and an interface error (i.e., the struct pointed by p). The erroneous struct value represents an interface error, as it propagates to the user program through an input-output parameter of the component. In general, we are considering data (on the heap, stack or global memory areas) that are reachable outside the component by following pointers in the scope of the user program. Instead, we are not considering components' local variables or heap memory areas not reachable (neither directly through interface parameters, nor indirectly) by the user program.

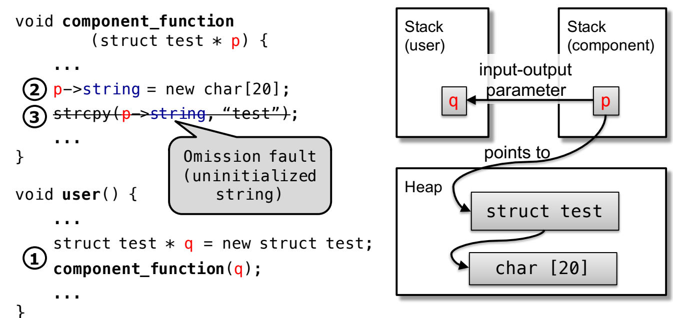


Fig. 4: Interface error on an indirectly-reachable area.

Even if interface parameters are not directly corrupted, error propagation can still *indirectly* affect the user program by corrupting data that is pointed to by an interface parameter, such as in the case of complex data structures like

0098-5589 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2018.2850755, IEEE Transactions on Software Engineering

trees and linked lists. This is the case in Figure 4, where a user-allocated data structure (①) is linked to a component-allocated string (② and ③) that can get corrupted. A corruption of the linked string can be considered an interface error, as this area is reachable by the user program. This applies in general to any memory area reachable from an interface parameter through an arbitrary number of pointers.

3.2 Research questions

From a practical perspective, the injection of errors consists in corrupting the data of a program, in order to emulate the effects that would be caused by software faults. The purpose of our the proposed methodology and experimental analysis of this paper is to guide the injection of these corruptions, by pursuing the following research questions.

The first research question is a prerequisite for motivating further investigations of interface errors. As discussed in the introduction of this paper, software errors (i.e., the effects of software faults [55]) range from simple (i.e., the errors are explicitly signaled by the faulty component, and can be managed with exception handlers) to more subtle ones (i.e., the component does not stop in the case of failure, and silently returns corrupted data to the user). In the former case, faults can be emulated with existing error injection approaches; however, we hypothesize that the latter case is quite frequent, which would require more sophisticated approaches for interface error injection. We experimentally investigate this aspect by posing the following question:

(RQ1) Are interface data corruptions a frequent effect of faulty software components?

After establishing the relevance of interface errors, the following three research questions aim to characterize these errors more precisely, in order to guide the injection of interface errors. This aim requires to define an *error model* that the injected corruptions should follow. In the fault injection literature, such models are defined in terms of *what* to inject (i.e., which corrupted data should replace the correct ones), *where* to inject (i.e., which data should be targeted by the injection), and *when* to inject (i.e., how much time the injection persists during an experiment) [9], [41], [56]. For example, the *bit-flip* (that is, the transient inversion of the content of a single bit in memory or CPU registers) is a popular fault model for hardware faults (i.e., CPU and memory faults, such as electromagnetic interferences), since simulation studies [57], [58] have shown that bit-flips are the most frequent effect of these faults. The bit-flip model dramatically relieved the cost and the complexity of hardware fault injection, since bit-flips can be injected through software-implemented fault injection techniques (SWIFI), instead of using more cumbersome physical injection techniques [16], [59]. Establishing similar patterns for software interface errors would ease the problem of emulating software faults (e.g., to avoid the dormancy of code mutations, and the lack of source code for OTS components).

The next research question is about the extent of errors, both over space (i.e., the amount the data that are affected by corruptions) and time (i.e., how the corruptions are spread over the execution period of the target software). For example, in the case of traditional hardware fault models (that have also been adopted for emulating software faults [25], [26]), single or few bit-flips with a transient timing were used to emulate electromagnetic interferences; and fixed bits (*stuck-at*s) with a persistent timing to emulate manufacturing defects of integrated circuits [37]. Similarly, knowing the extent of interface errors is the first information needed for defining how many corruptions an injector should introduce in the target system, and when it should introduce them.

(RQ2) What is the extent of interface errors, in terms of amount and timing?

Another important (but not yet investigated) aspect for modeling interface errors is represented by *error signaling* mechanisms used by software interfaces. In general, error signaling is a common pattern for architecting fault-tolerant systems, in which components at a lower-level of the architecture pass information about faults to the upper-level components, in order to handle faults at the most convenient layer in the architecture (e.g., by masking the fault, or gracefully propagating it to users as a system failure) [4], [60]. In the case of software systems, components use error signaling, such as special return values and exceptions, to notify of faults either in the hardware (such as, the unavailability of a network connection, resource exhaustion, etc.) or in the software (such as the violation of a data invariant). Since fault-tolerance mechanisms are typically designed around error signaling, a key question is whether the occurrence of interface errors can be detected through error signals. If this is true, it would suffice to test fault-tolerance mechanisms by forcing error codes and exceptions at API calls [48], [49]. Instead, if faulty components are *silent* on interface errors, then the fault-tolerance mechanisms should not rely on them, and they should also be tested by corrupting interface data without injecting any error signal.

(RQ3) Do faulty components signal interface errors, such as through error codes or exceptions?

Finally, the last aspect is about how corrupted interface data deviate from the correct interface data. Many corruption patterns have been hypothesized and proposed for modeling software errors, including: the traditional *bit-flip*s and *stuck-at*s also used for hardware [25], [26]; replacing values with boundary or special values from the same domain (e.g., the maximum value of a numeric range, or NULL pointers) [4]; or replacing the original values with entirely random ones (e.g., as in fuzz testing) [44]. However, even if it can be argued that these error patterns can originate from software faults, and that they have been useful to find breaches in fragile software interfaces [42], [43], no previous study has yet provided quantitative evidence that these errors are in fact the likely effect of software faults (and thus worth to be tested by error injection). Therefore, we investigate if, and how often, any of these error patterns are actually caused by software faults, and whether they could

0098-5589 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2018.2850755, IEEE Transactions on Software Engineering

be adopted for error injection purposes.

(RQ4) Which interface error patterns are produced by faulty software components?

4 METHODOLOGY

Our experimental methodology analyzes interface errors in software components, by executing the software with a fault injected in the component, and by comparing this execution with a fault-free one. Fig. 5 summarizes the steps of the experimental methodology, which are discussed in the following four subsections. The target component is linked to a user program that exercises the component's API, and the resulting system is executed with a realistic workload (§ 4.1). We then generate "faulty" versions of the component under analysis, by applying code mutations on components' source code (§ 4.2). We execute the mutated versions (a *fault injection experiment*), and trace the API interactions between the user program and the component, by recording the state of all interface data after every API call (§ 4.3). Finally, we compare the interface data of the fault-free execution with all the faulty executions, and identify interface errors by looking for differences, in order to address the research questions (§ 4.4).

4.1 Setup of the target software component, software system, and workload

The proposed methodology analyzes a target software component by means of experimental runs of the component, in the context of a software system. We use the term *component* to generally refer to a reusable software module, such as a library of general-purpose algorithms and data structures, that provides a clear programming interface (e.g., API functions) for use by a *software system*, that is, a larger software that includes the component to provide richer functionalities. We refer as the *user program* to the part of the software system that uses the component.

The software system used for the analysis of the target component is fixed before applying the proposed methodology, and provides context for running the component, and for analyzing interface errors produced by the target component. Similarly, the methodology fixes in advance the *workload*, that is, the set of inputs to stimulate the software system and, indirectly, the target component used by the system (e.g., by issuing API calls). This approach reflects the typical workflow of fault injection experiments, where the experimental setup reflects the expected operational conditions that the system will experience when it will be deployed in the operation [37], [61]. For example, in our study we consider components from benchmark applications (§ 5.1), where: the component provides algorithms and data structures (e.g., for graph analysis and data compression); the user program calls the component (e.g., API functions for manipulating and querying graphs, and for reading and writing data to be compressed with various options) using a workload (e.g., a dataset containing the list of nodes and edges of the graph, or an image file to be compressed) that was defined by the benchmark proposers according to feedback from the industry [62].

Once the software system and the workload have been fixed, the methodology injects faults in the target component, such that the faults can have an impact during the execution of the selected workload and software system (i.e., the fault is activated, and its effects are propagated from the injected component to the rest of the system). This implies that faults are only injected in those parts of the component that are used by the system and stimulated by the workload. As for the unused parts of the component, it is important to note that the methodology is not meant to experiment with every possible component fault, since that would be not feasible as the amount of time available for testing is limited in practice. Instead, the goal for our methodology, and in general for fault injection experiments, is to analyze the *most likely* errors of a component, by focusing on the subset of faults that can be triggered in the context of a representative system and workload of interest [63].

The proposed methodology, as for fault injection studies in general [49], [64], performs a preliminary analysis of the coverage of the component by the workload, and only injects in areas that are actually covered, in order to avoid injecting faults that would not be triggered. Despite the preliminary analysis, it is still possible that the injected faults can still be dormant, either because a fault does not infect the state of the component, or the effects of the fault remain confined inside the boundaries of the injected component [9], [50], [51]. The methodology is not expected to avoid these cases, since this behavior is a fundamental limitation of fault injection based on code mutation, and also one of our motivations for studying how software faults propagate across components. We aim to overcome this limitation through the direct injection of interface errors, which does not suffer from the dormancy problem. The purpose of the proposed methodology is to support at obviating the lack of realistic models for injecting interface errors.

4.2 Fault Injection

We inject faults in software components by using an automated technique and tool (SAFE) [63], [65]. The tool mutates the source code of the target component according to a set of *fault classes* defined by the Orthogonal Defect Classification (ODC) scheme [66], which includes *Assignment* faults (values assigned incorrectly, or not assigned at all), *Checking* faults (missing or incorrect validation of data, or incorrect loop or conditional statements), *Interface* faults (incorrect call statements, parameter lists, and interactions with external components), and *Algorithm* faults (incorrect or missing implementation that can be fixed by re-implementing an algorithm or data structure). The SAFE tool injects 12 fault types (TABLE 1) that cover the ODC fault classes. These fault types were defined in previous studies [14] by analyzing the bug-fixes of post-release software faults in several open-source projects, and by identifying the bug-fix patterns that appear most frequently and consistently across the projects. The fault types also provide several detailed rules ("constraints"), not shown for brevity, to describe the *code context* in which the fault types should be injected to reflect post-release software faults (for example, the removal of an

0098-5589 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2018.2850755, IEEE Transactions on Software Engineering

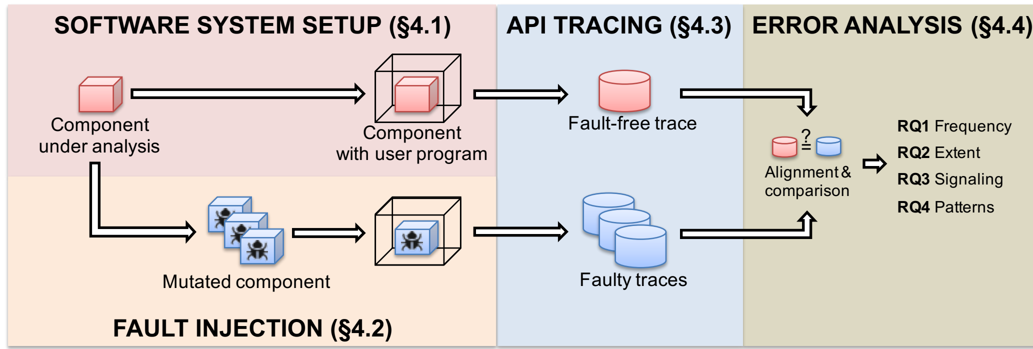


Fig. 5: Overview of the experimental methodology.

`if` construct is injected in those `if` constructs that enclose at most 5 statements, since it is unlikely that an `if` construct is omitted for larger groups of statements).

The SAFE tool parses the C/C++ source code of the target component, and automatically identifies injectable locations for the fault types. Since the number of injectable locations is typically very large for complex software, we apply the following criteria to tune the selection of faults to be injected:

Coverage. We first execute the fault-free target component by running it with the user program and workload that will be used in fault injection experiments. We analyze the statement coverage and exclude the injectable locations that are not covered, since the faults injected there would not be triggered and would not contribute to the analysis of interface errors.

Fault proneness. We exclude injectable locations that do not fall in "*fault-prone*" parts of the target component. In our previous work [63], we found that many injectable locations are not representative of post-release faults, as faults injected there are easily discovered by test suites (this criterion is close to the notions of "semantic size" and of "sensitivity" of software faults [46]). Moreover, we found that fault-prone files and functions can be identified using software complexity metrics (in particular, lines of code and fan-out) and classification algorithms, in a similar way to defect prediction approaches [67]. We adopt the same approach in this study to identify the functions of the target component in which to inject faults.

ODC proportions. We sample the injectable locations, such that the proportion of ODC classes in the injected faults matches the proportion of ODC classes found both in commercial products [9] and open-source software [14]. These studies showed that the distribution of faults across ODC classes consistently follow a common trend, where the *Algorithm* class is the largest one, amounting to $\approx 40\%$ of faults, while the *Assignment* and *Checking* classes have approximately the same weight of $\approx 20\%$. We follow this proportion in our experimental analysis, by sampling the injectable locations to avoid that ODC classes are over- or under-represented.

TABLE 1: Injected fault types [14].

Type	ODC	Description
MFC	ALG	Missing function call
MIA	CHK	Missing IF construct around statements
MIEB	ALG	Missing IF construct plus statements plus ELSE before statements
MIFS	ALG	Missing IF construct plus statements
MLC	CHK	Missing AND / OR clause in expression used as branch condition
MLPA	ALG	Missing small and localized part of algorithm
MVAE	ASG	Missing variable assignment using an expression
MVAV	ASG	Missing variable assignment using a value
MVIV	ASG	Missing variable initialization using a value
WAEP	INT	Wrong arithmetic expression used in parameter of function call
WPFV	INT	Wrong variable used in parameter of function call
WVAV	ASG	Wrong value assigned to variable

4.3 Execution and API tracing

Both the fault-injected versions of the component under analysis (§ 4.2) and the original version of the component are executed in the context of the same user program and workload (§ 4.1). In order to identify interface errors and to characterize them according to the research questions (as will be discussed in § 4.4), we trace all interface data that are exchanged between the component and the user program. To this purpose, we developed a *tracer* tool for keeping track of interface data in C and C++ components. From a general point of view, the tracer intercepts every API function call from the user program to the component under analysis; then, it lets the API function call to execute, and records the contents of the interface data at the end of the call. Moreover, the tracer assures the determinism of the executions, in order to make the traces comparable.

The tracer takes in input an executable program (which includes the fault-injected component to be analyzed), along with command-line parameters, configuration files, and other files to be processed by the program, and the corresponding fault-free outputs (such as, the expected output messages or files to be produced by the program) that will be used to determine whether the faulty program execution failed. Moreover, the tracer takes in input the list of API functions of the target component; these functions can be identified before the experiments by inspecting the docu-

0098-5589 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2018.2850755, IEEE Transactions on Software Engineering

mentation of the component and the source code of the component's user program. For example, in the example of the graph analysis component mentioned in the previous section, the tracer automatically scans the program executable and looks for functions for manipulating and querying graphs that are listed in the API; then, it executes the program by feeding to the program the dataset of edges and nodes, and by probing the calls to these functions, in order to inspect the interface data that are exchanged between these functions and the user program (e.g., data structures representing paths in the graph, or numeric data computed from the graph).

For each API call, we analyze the interface data structures that are in the scope of the user program at the time of the API call, by automatically looking at variables in the *root set* (global and stack variables of the caller program) and any variable that can be accessed by following pointers in the root set (the *reachability graph*). For example, in the case of a graph algorithm, the root set includes the handle of the graph data structure, such as a pointer variable returned by the API, and the tracer follows the pointer to identify the nodes and edges of the graph data structure. The sequence of all API calls of an execution, and the set of interface data at each call, form a *trace*.

The tracer is built on top of the GDB debugger, using Python bindings to control and to inspect the state of program executions, and runs on the Linux OS. The tracer installs a *breakpoint* for every API function of the target component. When the user program invokes an API function, a breakpoint handler is triggered. The handler performs the following steps before returning the control flow to the component's user.

1) It gets the list of variables that are visible in the scope of the component's user, including global and local variables, and checks whether a return value is expected from the function call;
2) It logs the name of the called API function, and the location of the user program that calls the API function (*call site*).
3) It temporarily disables further breakpoints, which might be triggered again during the execution of the API function (e.g., the function indirectly calls another API function), since our analysis does not focus on error propagation inside the component, but on errors propagated to the component's users.
4) It poisons the stack frame of the current API function call, which will be used by the called function to store local variables. The stack is overwritten with a fixed, known bit pattern, in order to detect local variables that are not initialized during the course of the API function call.
5) It resumes the API function execution, and suspends again the execution once the function call has been completed. It then re-enables the breakpoints.
6) It dumps into a log the full contents of interface data, by recursively inspecting the root-set variables and other structures linked to the root-set. If the interface data contain a pointer, it will dereference the pointer if valid, and dumps the pointed data structure; if the interface data include an array, it will dump the individual elements of the array, dereferencing any pointer if necessary; and it will dump the content of any primitive type (e.g., `int`, `char`, `float`).

The comparison of traces in faulty and fault-free conditions requires that differences between traces are actually due to faults, and not due to random variations caused by *non-determinism*. In our experimental setup, we avoid such variations by addressing the following sources of non-determinism.

Memory management. Heap areas (that are dynamically allocated) and the stack area (that grows and shrinks over an execution) may be assigned to different memory addresses across executions, depending on the state of the OS and of physical memory at the time of execution. In order to allow the comparison of the addresses within pointer variables, the tracer rewrites these addresses by replacing them with a *symbolic representation*, which is composed by a pair <*area id*, *offset*>. The tracer assigns to each memory area an unique identifier (i.e., the *area id*) when the area is initially allocated; this identifier is consistent across executions, since it is computed by hashing the call stack at the time of the memory allocation, and an incremental counter for distinguishing between allocations with the same call stack. Afterwards, when the tracer meets an address inside the interface data, it identifies the memory area that includes the address, and it computes the relative distance (i.e., the *offset*) between the address and the beginning of its memory area. The <*area id*, *offset*> symbolic representation is then saved in the trace in place of the original address. If, during a fault injection experiment, the injected fault does not corrupt a pointer (i.e., the pointer still contains the address of the intended memory area), then the symbolic representation of that pointer will match the symbolic representation of the pointer in the fault-free execution, even if the memory area is assigned a different memory address.

To efficiently keep track of dynamic memory areas allocated on the heap, we adopt an *interval tree* data structure, i.e., a tree whose objects are numeric intervals, and which can be queried to find intervals that overlap with any given value [68]. In our case, the intervals represent memory areas (denoted by the start and end addresses), which are queried by looking for the address to be rewritten. At each heap memory allocation, the tracer updates the interval tree; moreover, the tracer also poisons the heap area when it is allocated, before returning its pointer to the program. Memory allocations are traced by intercepting memory allocation functions, such as `malloc()` and `free()`, using the function wrapping mechanism provided by Linux and Unix systems [69].

Memory management can also lead to non-deterministic program behaviors when fault injection results in memory management bugs. For example, in the case that the injected fault causes a *buffer overflow*, the outcome of the execution is non-deterministic since it will overwrite whatever stack, heap or global areas adjacent to the overflown buffer (depending on the memory layout, which is determined by the compiler and by memory allocation algorithms). Moreover, if the overflow corrupts a pointer variable (such as the return address on the call stack), the control flow will depend on whatever has been allocated at the (wrong) address written on the pointer variable. To have a deterministic and

0098-5589 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2018.2850755, IEEE Transactions on Software Engineering

consistent behavior across fault injection experiments, we instrument the target program at compile-time, by using the *AddressSanitizer* memory error detector [70], to insert "redzones" around buffers and lightweight sanity checks before memory access operations. We let the sanitizer to terminate the program, in order to provide a consistent behavior in the case of buffer overflows and other memory-related bugs.

Thread scheduling. Thread scheduling can influence an experiment, by causing differences on the interface data not due to the injected fault, but due to the different interleaving of threads (i.e., the component behavior changes regardless of the presence of the fault). To avoid such misleading differences, we need to assure that if the fault does not have any impact on the behavior of the component, then the interface data will match the interface data of the fault-free execution. In this study, we focus on sequential programs, as they represent the base case for analyzing error propagation, and cannot be affected by misleading differences due to thread scheduling. However, we here discuss how to apply the approach in the general case of multi-threaded software, and why multi-threading is an orthogonal problem to the research questions of this paper and should be addressed separately.

To ensure the deterministic execution of a multi-threaded program, it should be executed using *record-and-replay* techniques: these techniques record the points of the execution at which thread switches happen (*preemption points*) and, when the program is executed again, they force thread switches at the same preemption points. Record-and-replay techniques have matured enough to be integrated in debugging tools, and it is supported in our tracer by using the *rr* tool in conjunction with the GDB debugger [71].

If the injected fault does not impact on the component behavior, then the faulty program will be able to reach the same preemption points of the fault-free run: in this case, record-and-replay can enforce the same thread schedule, thus reproducing the same interface data and avoiding spurious differences caused by non-determinism. Instead, if the fault impacts on the behavior of the component by changing its control flow, then the execution deviates from the preemption points in the fault-free run, and there is no reference schedule that could be enforced: in this case, the tracer lets the component to execute with an arbitrary thread schedule, and we study the interface errors under the schedule that occurs during the experiment. On the one hand, this approach makes the analysis tractable, since it requires to only execute the program once for each injected fault; on the other hand, we (intentionally) do not analyze the possible variability of interface errors with respect to thread scheduling, which could amplify or dampen error propagation. Analyzing this variability would require to actively explore the space of thread interleavings (by varying thread scheduling, and evaluating the resulting interface errors), but this approach would hit the infeasibility of exhaustive search for complex software and for a high number of fault injection experiments. For this reason, we leave the exploration of thread interleavings out of the scope of the tracer, and focus our approach on sequential and individual multi-threaded executions.

I/O operations. The timing and the contents of I/O operations can also affect the execution flow and the interface data of a component. Non-determinism due to I/O timing can be avoided if the effects of thread scheduling are avoided, either by focusing on single-threaded applications, or through record-and-replay. In the former case, the execution of single-threaded applications is insensitive to I/O timing, since there is no thread preemption that may be delayed or anticipated due to variations of I/O waits. In the latter case, record-and-replay makes the thread schedule deterministic, despite variations of I/O waits. Moreover, we avoid non-determinism of I/O contents by executing our target applications in a controlled experimental environment, in which the target is fed with the same I/O data (e.g., the same input files) at each execution.

Random number generators. The use of (pseudo) random numbers in a program can lead to random values being written to memory and to variations of the execution flow. We avoid the effects of random numbers by wrapping random number generators, such as `rand_r`, and forcing them to return the same sequence of numbers at each execution.

4.4 Analysis of interface errors

The methodology identifies interface errors through a comparative analysis between a fault-free execution of the target software, which serves as reference, and several faulty executions, each with a different software fault.

Given a pair of executions (faulty and fault-free), our methodology compares the two sequences of API calls made by the user program to the component under analysis. The comparison points out the following parts of the faulty execution:

Initiation. At the beginning, the injected fault has not been triggered yet. During this phase, the faulty execution performs the same API calls of the fault-free one, and exchanges the same interface data, thus there are no interface errors to analyze.

Corruption. When an API call triggers the injected fault, it propagates errors on the interface data; the errors will then be processed by the user program. From this API call onwards, the comparative analysis identifies which parts of the faulty interface data are different from their fault-free counterparts.

Termination. The run ends with any of these outcomes:

1) The errors cause the premature termination of the program (a *crash* failure).
2) The program is unable to finish the execution within a period of time comparable to the fault-free run (a *hang* failure), e.g., due to an infinite loop caused by errors; in this case, we force the termination of the experiment.
3) The program terminates the execution by its own, by producing results in output that either differ (*wrong termination*) or are the same of the fault-free execution (*correct termination*).

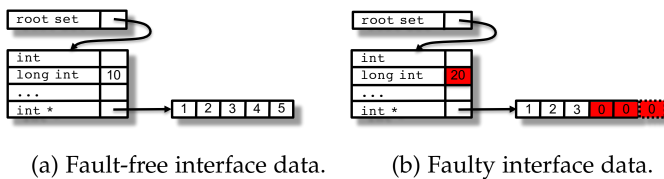
Before terminating, the user program may perform the same sequence of API calls of the fault-free execution (even if interface data are corrupted); or the faulty execution may diverge from the fault-free one, and exhibit different API calls. In the case of divergence, the comparative analysis is performed up to the last API call that matches the fault-free execution, since the subsequent API calls (and related

0098-5589 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2018.2850755, IEEE Transactions on Software Engineering

interface data) must be considered different than the correct ones, and thus corrupted.

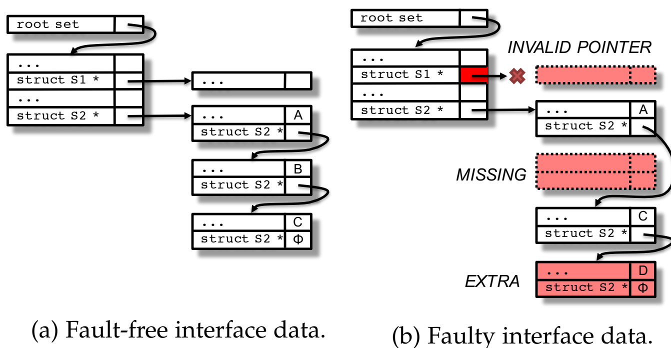
We perform a member-by-member comparison between each data structure in the faulty trace and its fault-free counterpart. The comparison is performed recursively for each complex data type within the data structure (such as *structs* and *arrays* within a larger C struct). For example, Fig. 6 shows a hypothetical C struct with `int`, `long int`, and `int *` members. In this example, the `long int` member of the faulty structure differs from the fault-free one, and it is considered an interface error. The same comparison is performed recursively on the array on integers pointed by the `int *` member, which contains 3 interface errors: the two elements at the end of the fault-free array are different from the faulty counterparts; and the faulty array is oversized by one element that is not present in the fault-free array.



(a) Fault-free interface data. (b) Faulty interface data.

Fig. 6: Example of interface error identification.

Fig. 7 shows a more elaborated example with a linked data structure, with variable size. The first interface error is in the `struct S1 *` pointer member, which contains an invalid address (that is, the address does not belong to any heap, stack or global memory area allocated by the program). In this case, the interface error includes the wrong address in the pointer member, and the missing sub-struct that is not reachable in the faulty execution. Moreover, there are two interface errors in the linked list of `struct S2` elements. Two elements of the list (the ones containing the values `A` and `C`) are included in both the traces; instead, the faulty interface data lack the element with the value `B` (the first interface error), and contains a spurious element with the value `D` (the second interface error).

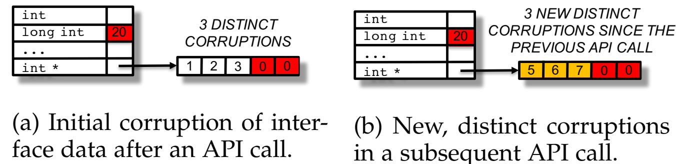


(a) Fault-free interface data. (b) Faulty interface data.

Fig. 7: Example of interface errors in linked data structures.

The comparison between variable-sized data structures is performed after a pre-processing step (*data structure alignment*), which identifies the common parts between the faulty and the fault-free data structures, and points out the elements that are missing, spurious or different in the faulty data structures. To find the common parts, the faulty and fault-free structures are flattened into sequences, by recursively traversing the structures and inserting the visited elements in the sequences; each element is traversed and inserted at most once to avoid infinite recursion. Then, we apply an algorithm on the two sequences to find their *longest common subsequence* (LCS). The LCS is a subset of elements that are present in both sequences in the same order, and that can be obtained by removing (a minimal number of) elements from the original sequences. This kind of problem is recurrent in computer science, such as in bioinformatics and in source code versioning (e.g., in the *diff* Unix tool), and can be solved with efficient algorithms [72], [73]. In our case, we compare linked data structures element-by-element, by computing the LCS and identifying their differences, which denote the corrupted elements.

At any given point in the trace, we take into account whether an interface error happened during the current API call, or it has been caused by a previous API call, in order to avoid that the same interface error is accounted for a second time. Fig. 8 shows an example with 6 interface errors, where 3 interface errors (in red) happen at the first API call, and 3 more (in green) happen at a subsequent API call. The last 3 interface errors are the result of a new execution of the software fault, or the byproduct of previous interface errors that are used for further computations. Our methodology identifies which interface errors for a data structure are new in the current API call, by retrieving the interface errors of the last API call that involved the same data structure, and comparing them with the current interface errors. This approach avoids to over-represent interface errors that persist over time (i.e., across several API calls); moreover, it allows us to analyze when and for how long interface errors are introduced during a faulty execution.



(a) Initial corruption of interface data after an API call. (b) New, distinct corruptions in a subsequent API call.

Fig. 8: Example of distinct interface errors across API calls.

After identifying interface errors in all faulty executions, we aggregate and analyze them to address the four research questions, as discussed below.

▷ ***(RQ 1) Frequency of interface errors***. This research question is addressed by classifying the outcome of experiments among the following types:

Propagation, failure. The injected fault propagated at component interfaces, and generated interface errors in interface data. Then, the interface errors led to a failure of the component's user (*crash*, *hang*, *wrong termination*).

Propagation, no failure. The injected fault propagated as interface errors, but the component's user was able to execute correctly despite the interface errors (*correct termination*).

No propagation, failure. The injected fault did not propagate as interface errors, but it still caused a software failure (*crash*, *hang*, *wrong termination*) before propagating to the component's user. For example, the software crashes

0098-5589 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

or becomes hung while executing the component's code, and the control flow never returns to the component's user.

No propagation, no failure. The injected fault neither propagated any interface error, nor it caused a failure of the software (i.e., a *correct termination* of the run). In these cases, the injected fault is *dormant*: it either has not been triggered, or its effects remained limited inside the boundaries of the injected component. For example, the fault corrupts internal component data that are never passed to the caller program, or that are overwritten with correct values before returned to the caller.

The relative number of experiments with propagation (compared to the total number of experiments with propagation and/or failure) represents the frequency of interface errors. If the fraction of these cases is low, it means that interface errors are an unlikely effect of injected software faults: thus, faulty software components could be emulated trivially by forcing the crash or stall of the component, as suggested by the most simplistic software error models. Instead, a high frequency of interface errors would motivate richer error models to also encompass interface data corruption. In the case of the experiments with "no propagation, no failure", we cannot draw definite conclusions, since the lack of failures and propagation may have been caused by *equivalent mutants* (i.e., the injected component has an equivalent behavior of the original one); in such cases, the outcome of the experiment is not a genuine (lack of) effect of the fault, but a code mutation that was ineffective at injecting a fault. Thus, these experiments do not provide information on the relative frequency of interface errors. However, for completeness, we also report and discuss about the cases of "no propagation, no failure".

▹ ***(RQ 2) Extent of interface errors***. We address this research question by analyzing interface errors both with respect to space (i.e., the quantity of interface data affected by corruptions) and time (i.e., when interface errors happen during the execution of the experiment).

As for the quantity of interface errors, we adopt two measures: the *number of corrupted bytes*, and the *number of corrupted variables*. For the first measure, we compare the binary representation of each member of interface data structures, and count the number of differing bytes. This measure is useful to study interface errors with respect to previous error models, which were hardware-oriented and represented errors in terms of byte-granularity corruptions [25], [26]. We only consider *distinct* interface errors (see also Fig. 8): thus, even if an interface error persists over the course of an experimental run, it adds to the count of corruptions only at the first API call at which the error occurred.

In addition to this measure, we also evaluate the extent of interface errors in terms of number of corrupted member variables (i.e., if a member variable differs from the faulty-free counterpart, it counts as 1 corruption regardless of its binary representation). If we would base our conclusions only on a binary-level analysis, the extent of interface errors could be over- or under-estimated due to the dependencies on the underlying compiler and hardware architecture: for example, in an x86-64 system, the same software fault (with the same apparent effects on the program) could cause a different number of corrupted bytes in an integer member variable, depending on whether the integer is represented as an `int` type (with 4 bytes) or a `long` integer type (with 8 bytes). Using an additional measure at the program-level (corrupted variables) in conjunction with a binary-level measure (corrupted bytes) provides a better understanding of the extent of errors.

To study the temporal extent of interface errors, we analyze how interface errors are spread during the course of an experimental run. It is important to note that interface errors can be introduced only when the user program invokes the APIs of the faulty component. Since interface errors cannot happen at an arbitrary time, it would be misleading to consider the duration of interface errors in terms of "physical" time (as in traditional hardware error models). Therefore, we quantify the temporal extent of interface errors in terms of *number of API calls that introduce new interface errors*. At one extreme, only one specific API call may introduce interface errors, while the remaining API calls do not corrupt any additional interface data. At the other extreme, the injected fault may add new (distinct) interface errors every time that an API is called. We analyze the distribution of this measure across all of the experiments, in order to understand how often interface errors should be injected to emulate components' faults.

▹ ***(RQ 3) Signaling of interface errors***. To address this research question, we analyze in which experiments the injected component generated an error signal. We detect the occurrence of error signals by looking for specific events raised by the injected component, including:

- The faulty API call throws an exception that is never thrown during the fault-free execution.
- The faulty API call provides to the caller a return value that is different from the fault-free execution, and that denotes an invalid value (e.g., the return value is null or negative).
- The faulty API call invokes a callback function to signal and handle the error, e.g., by printing a message or causing the program to end the execution. For example, a memory allocation procedure may invoke the callback on allocation failures; or the API may check for data inconsistencies using the `assert` primitive in the C language.

Once we determine the occurrence of these events, we measure the percentages of the following three cases:

Propagation, no signal. The experiment exhibited interface errors, but the component did not raise any error signal (i.e., the error was not noticed by the component).

Propagation, signaled. The experiment both exhibited interface errors, and the component raised an error signal (e.g., the component noticed and signaled an error only after it corrupted interface data).

No propagation, no signal. The experiment did not exhibited interface errors, but the component raised an error signal (e.g., the control flow returned to the component's user before any interface error could occur).

The relative percentages of these cases provide indications for interface error injection. If the third case is predominant, then the error injection could be limited to force an

0098-5589 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2018.2850755, IEEE Transactions on Software Engineering

error signal to the component's user; otherwise, if the first two cases are a significant percentage, then it means that error injection should focus on interface data (propagation, no signal), or that it should inject both interface errors and error signals (propagation, signaled).

To better support the injection of error signals, we also measure the *latency* of error signals. We defined the latency as the number of API calls between the occurrence of interface errors (if any) and the occurrence of an error signal; ideally, the error signal is raised in the same API call that generates interface errors (i.e., the latency is 0). The measured latency will provide suggestion on whether the injection of error signals should be delayed with respect to the injection of interface errors.

▷ ***(RQ 4) Patterns in interface errors***. In this research question, we investigate the differences between corrupted and correct interface data, in terms of their *contents*. This aspect is important since it defines what corruptions should be introduced by an error injector, and it is commonly referred as the problem of "*what to inject*" in the scientific literature. We hypothesize that software faults do not corrupt interface data in an arbitrary way but they tend to follow patterns, which error injection should also follow. To investigate this hypothesis, we define a set of patterns, based on existing error models from previous studies on error injection (§ 2), in order to evaluate whether these error models actually fit software faults.

As for the previous research questions, we analyze individual primitive members of the interface data structures. In this case, we check whether the contents of faulty interface data fit any of the patterns (summarized in TABLE 2). To analyze the individual primitive members of interface data structures, we defined patterns for four groups of primitive data types: integers (including its variants such as signed/unsigned and short/long integers), pointers, characters, and floating-point numbers. The patterns evaluate the (erroneous) value that is assigned to a corrupted variable.

TABLE 2: Error patterns for interface data types.

Type	Pattern
Integers (`int`, `long`, `short`)	• Uninitialized; • Special value (0, 1, -1, signed and unsigned `int/long/short` max); • Power-of-two (2^n) with a (small, large) offset; • Correct value with (small, large) offset; • Bit-flip (1 or 2 bits)
Pointers	• Uninitialized; • Invalid; • Special value (0, 1, -1, `void *` max); • Power-of-two (2^n) with a (small, large) offset; • Correct value with (small, large) offset; • Bit-flip (1 or 2 bits)
Characters (`char`)	• Uninitialized; • Special value (0, 1, -1, `char` max); • ASCII (printable, non-printable) code; • Correct value with (small, large) offset; • Bit-flip (1 or 2 bits)
Floating-point numbers (`float`, `double`)	• Uninitialized; • Special value (0, 1, -1, `float/double` max, NaN, Inf); • Power-of-two (2^n) with a (small, large) offset; • Correct value (exponent, mantissa) with (small, large) offset, inverted sign; • Bit-flip (1 or 2 bits in mantissa and/or exponent)
Arrays	• Missing (few, many) elements at end of array; • Surplus (few, many) elements at end of array; • Wrong (few, many, most) sparse elements within the array; • Wrong (few, many, most) contiguous elements within the array
Linked structures	• Missing (few, many, most) contiguous elements; • Missing (few, many, most) sparse elements; • Surplus (few, many, most) contiguous elements; • Surplus (few, many, most) sparse elements; • Wrong (few, many, most) contiguous elements; • Wrong (few, many, most) sparse elements

Special values. The erroneous value is a boundary or special value for the data type. For example: for integer types (such as `int`), 0, 1, -1, and the maximum representable value; for characters, the erroneous value is in the range of ASCII codes that are printable ($32 \ldots 126$) or non-printable ($0 \ldots 31, 127$); for pointers, *NULL* and *invalid* addresses (i.e., addresses that do not belong to any memory block allocated in the stack, heap or global areas); for floating-point numbers, *NaN*, *Inf*, and the maximum representable values. These special and boundary values are often used in software testing [45], and for data type-based fault injection such as in BALLISTA [4] and FIG [48]. Moreover, for integer variables, we check whether the corrupted value is (close to) a power of two, as these values have often a special meaning, such as in the case of integer variables used as bitmasks (e.g., special values for changing the behavior of a system call), where only one or few bits are set and the variable equates, or is close, to a power of two.

Bit-flips. The erroneous value differs from the correct value by 1 or 2 bit-flips, as in traditional hardware models for SWIFI [5]. For integers and pointer types, we include bit-flips in the least significant byte, in the most significant byte, or in any of the remaining bytes of the variable. For floating-point numbers, we include bit-flips located in the mantissa, in the exponent, or in both.

Offsets. The erroneous value is different, but close to the correct value (i.e., the one from the fault-free run). For example, such corruptions were adopted for perturbation analysis by Voas et al. [46], [47]. We consider "special" values as ranges for the data corruptions. We include cases in which the offset is, respectively, within a small or large range Δ: for integer types, small if $\Delta \leq 10$, or large if $\Delta \leq 1000$; for pointers, small if $\Delta \leq 64$, or large if $\Delta \leq 4096$, since addresses are typically word-aligned or page-aligned. For floating-point numbers, we check whether the sign is inverted, or there is a small offset either in the exponent or in the mantissa, or in both.

Uninitialized. *Uninitialized* data are a special case of random data in C/C++ software, where a program does not initialize data due to memory management bugs [12], and the value of the data is determined by the environment (i.e., the compiler, the OS, and the hardware). In our context, such cases occur when a variable is initialized in the fault-free run, but uninitialized in the faulty run. Uninitialized variables are found by "poisoning" memory with a known bit pattern when it allocated, as

0098-5589 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

discussed in § 4.3.

The patterns in TABLE 2 can be interpreted as classes of values over the domain of a variable. Every interface error is classified into *at most one of the patterns*. If the value does not fit within any of these classes, then it is considered an arbitrary error (i.e., not following any pattern), and it is labeled as "*other*" in our analysis. If our hypothesis is true, then the majority interface errors should fit one of the patterns. If the value fits more that one pattern, we give priority to the most specific class (following the order of TABLE 2): for example, if the erroneous value is both a special value and deviates from the correct value by a small offset, we consider it as an erroneous special value.

In addition to primitive types, we consider patterns that affect *aggregates* of elements (either primitive variables or data structures), since conventional wisdom associates software faults to corruptions of aggregate data (such as, buffer overruns [12]). We consider both *arrays*, i.e., sequences of contiguously-allocated elements, and *linked data structures*, i.e., elements are connected through pointer variables, such as linked lists, binary trees, and graphs in general. We look in detail at how corruptions are spread across aggregates, and how they should be introduced by error injection.

To define error patterns for arrays, we generalize the idea of "overruns", by considering the possibility of omitted, surplus, and wrong elements. We compare the elements of the faulty and fault-free arrays, by first analyzing whether the faulty array has still the same elements except for some missing (i.e., truncated) or superfluous elements at the end. If the faulty array has the same size but contains wrong elements, we analyze whether they are located in the initial or in the last part of the array; whether the wrong elements are sparse or contiguous; whether the corruptions affect a small group of elements, all the elements, or most of them.

In the case of linked structures, we compare the flattened interface data structures, as previously showed in Fig. 7. Similarly to arrays, we analyze whether the corrupted elements of the structure are missing (i.e., the faulty structure misses an element that is present in the correct structure), surplus (i.e., the structure includes an element that is not present in the correct structure), or wrong (i.e., deviating from the correct counterparts). We check whether the corruptions are a mix of these cases, or they consist of only omissions, only surpluses, or only wrong elements. Moreover, we check whether the corruptions affect a small or large group of elements, and whether the corrupted elements are sparse or contiguous (i.e., whether every corrupted element has at least one direct pointer to another corrupted element).

5 EXPERIMENTAL ANALYSIS

In the following, we investigate the research questions defined in § 3.2, by adopting the methodology described in § 4 on a collection of ten programs described in the following.

5.1 Case study software

In order to conduct a realistic evaluation of the presented methodology, and to derive useful insights on the propagation of interface errors, we had to choose a set of programs that are representative of software commonly targeted by fault injection tests. As previously mentioned in the introduction, fault injections are mostly applied for critical systems, which are mostly implemented in C/C++. Therefore, to be suitable for our study the targeted software needed to be (1) written in one of these languages, (2) representative of real software used in critical domains, (3) diverse in terms of size, architecture, and interactions between the component of interest and the rest of the software, and (4) commonly used for fault injection studies.

We identified the SPECint® 2006 benchmark [74] as a suitable target fulfilling all of these requirements. We found the SPEC's program suite used in a number of fault injection studies by the research community on dependable computing [61], [75], [76], [77], and by studies on error containment techniques by research communities on computer systems [70], [78], [79], [80], [81], [82]. Moreover, the suite meets the requirement of including real C/C++ software, with algorithms for compression, combinatorial optimization, artificial intelligence, simulation, path finding, gene sequence search, and XML processing [83]. These programs are representative of algorithms adopted for critical tasks, such as route planning and image processing for computer vision in autonomous systems [84], service-oriented business applications using XML-based protocols [85], cryptography [86], genome research [87], and simulation of critical infrastructures [88]. Finally, these programs exhibit a good degree of diversity, as they range from small (few kLoCs with a dozen of files) to large ones (hundreds of kLoCs, with thousands of files) with interfaces of differing size and usage frequencies (respectively, in terms of # *API Functions* and # *API Calls*, as discussed later in TABLE 3).

The SPECint® 2006 benchmark includes the source code of the programs, the workloads to execute the programs (e.g., representative inputs for the application areas of the benchmark), and support tools to build and execute the programs. We inspected the programs to identify the API interface between generic modules inside the benchmark (e.g., algorithms and abstract data types that can be reused in a larger system, and that represent the *component* targeted by our fault injections), and the *user program* that invokes them and manages the inputs and outputs of the benchmark. We identified 10 out of 12 programs of the benchmark for which this separation applies; in the other two cases, either the program does not include a component suitable to be reused in a larger software (the *gcc* program), or the main program makes a trivial use of the component API (the *perlbench* program, which embeds a Perl interpreter, but where the API is limited to pass a Perl script to the embedded interpreter without any interface data to be analyzed). In the remaining programs, the component provided a clear API to initialize data objects and to process them (such as, to find a path in a graph, or to transform blocks of data by compressing or converting them).

TABLE 3 lists the 10 programs from SPECint® 2006 that we analyze in this study. We run the programs using the same inputs from the SPECint 2006 benchmark, except for the number of iterations performed by the programs. The SPEC (as also explicitly stated in the benchmark documentation [89, quest. 17]) deliberately increased the volume of the inputs for performance evaluation purposes, in order

0098-5589 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2018.2850755, IEEE Transactions on Software Engineering

TABLE 3: Case study software from the SPECint® 2006 benchmark.

Program	Application area	Language	# LoC	# Files	# Functions	# API Functions	# API Calls	# Faults
astar	Path-finding Algorithms	C++	4,283	19	213	19	78	360
bzip2	Compression	C	5,734	12	120	6	3,981	728
gobmk	Artificial Intelligence	C	157,652	96	2,682	3	289	2,328
h264ref	Video Compression	C	36,101	81	590	37	43	3,184
hmmer	Search Gene Sequence	C	20,661	72	539	7	1,008	428
libquantum	Quantum Computing Sim.	C	2,609	31	95	13	60	105
mcf	Combinatorial Optimization	C	1,577	25	24	7	10	131
omnetpp	Discrete Event Simulation	C++	19,994	154	7,531	38	1,436	271
sjeng	Artificial Intelligence	C	10,547	23	144	10	17	372
xalan	XML Processing	C++	267,924	1,771	13,242	11	11	2,380
All								**10,287**

to perform a larger number of iterations and to inflate the execution time. In order to make our analysis feasible in terms of duration and storage occupation (as we need to perform a large number of experiments), we reduced the volume of the inputs while still covering the same code statements (with an error margin of 1% at most with respect to the original coverage) and exactly the same API functions. For *astar*, we reduce the map and region sizes; for *gobmk*, we reduce the number of moves; for *h264ref*, we reduce the number of frames; for *hmmer*, we reduce the number of sequences; for *mcf*, we reduce the number of arcs and nodes in the graph; for *omnetpp*, we reduce the duration of the simulation. TABLE 3 shows the number of API functions ("*# API Functions*") that are invoked at least once by the workload, and the total number of API function calls ("*# API Calls*"), which is higher for the benchmarks that iterate over large amount of inputs (such as file blocks in *bzip2*, events in *omnetpp*, and sequences in *hmmer*). Finally, the table shows the number of fault injections for every program, that were selected according to coverage, fault-proneness and ODC proportions as discussed in § 4.2, for a total of 10,287 faults.

5.2 Frequency of interface errors (RQ1)

We divide the fault injection experiments according to the outcome of the experiment (e.g., whether the injected program failed or not, and in which way), by applying the four failure modes defined in § 4.4, namely *crash*, *hang*, *correct termination* and *wrong termination*. The resulting distributions are shown in Fig. 9.

The distributions of failure modes vary across the target components, as they depend on the nature of the component and of its usage: for example, *crash* is a likely outcome for programs that extensively use pointer arithmetics, where a fault can turn into incorrect memory accesses, causing the OS to kill the process. For some components (*gobmk*, *h264ref*, *hmmer*, *omnetpp*, *xalan*), the most frequent outcome had been the *correct termination*, up to 60% in the most extreme cases. This is a result of the *dormancy* of code mutations: despite faulty code is injected in the component and gets executed, it did not produce any effect outside the program (e.g., the component corrupted data that were ignored or overwritten by the main program, or did not corrupt any data at all). For the remaining components, either *wrong termination* (*astar*, *bzip2*, *libquantum*, *sjeng*) or *crash* (*mcf*) were the most frequent outcomes. Almost all experiments exhibited the same sequence of API calls of the fault-free execution (either with or without interface errors), or the API call sequence was interrupted by a program failure; only in less than 1% of cases we observed divergences from the fault-free API call sequence of the user program.

With respect to the research question 1, we are specifically interested in how many of these experiments exhibited interface error propagations. These cases are showed as gray segments in Fig. 9: they are the experiments where we found any interface error propagated from the injected component to the user program. For all of the components, the fraction of experiments with interface corruption is relatively high in the cases of *wrong termination*, as 78.7% of all *wrong termination* experiments had interface corruptions; instead, interface corruptions were less frequent for the other failure modes (*crash*, *hang* and *correct termination*).

This result can be interpreted by observing that, in the case of most of the *crash* and *hang* outcomes, the component is unable to return any interface data since the crash/hang occurs while the control flow is still in the injected component. Moreover, in the case of the *correct termination* outcomes, most of the injected faults stayed dormant (87.1% of all *correct termination* experiments, and highlighted with a thicker border in Fig. 9). Conversely, most of the *wrong termination* outcomes were caused by the propagation of errors from the component to the user program, and then from the user program to the program's outputs. In the remaining cases of *wrong termination*, the program execution did not experience interface corruptions since it had been prematurely stopped due to error signaling, before interface corruptions could be returned to the component's user (this aspect is further discussed in § 5.4).

Due to the variable proportions of the four outcomes across the components, the amount of cases with interface corruptions also varies across them (e.g., interface corruptions are more frequent when there are more cases of *wrong termination* and less cases of *crash* and *hang* outcomes). Overall, the interface corruptions were numerous, regardless of the component. However, to quantify the likelihood of interface corruptions, we need to consider the uncertainty about error propagation in the case of *correct termination and no corruptions* (i.e., the segments highlighted with a thicker border). For these cases, it is not possible to determine whether the injection produced *equivalent mutants* (i.e., the injected component has an equivalent behavior of the original one, and thus are "ineffective" injections, as discussed in § 4.4 for *RQ 1*) or an actual fault that did not propagate interface errors (e.g., the fault corrupted data that were not passed to the component's user).

0098-5589 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2018.2850755, IEEE Transactions on Software Engineering

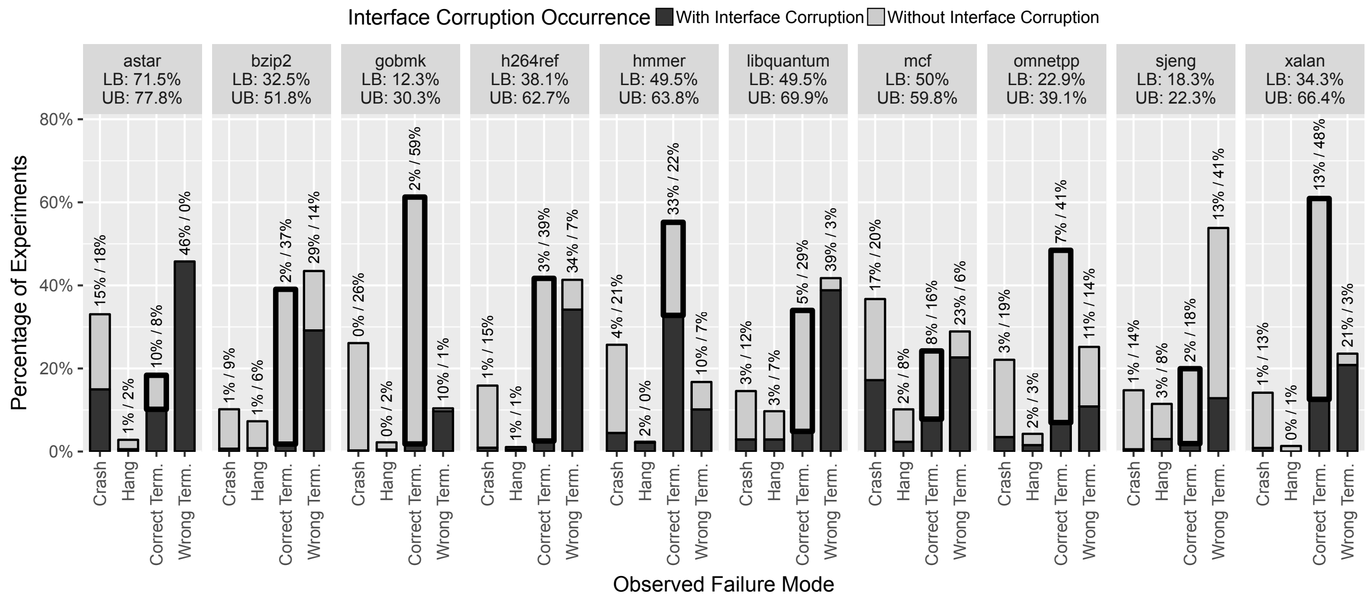


Fig. 9: Percentages of failures and of interface corruptions (thicker boxes represent dormant faults, i.e., correct terminations without interface corruptions).

Therefore, we estimate the relative frequency of interface corruptions by computing its *lower bounds* and *upper bounds*: the lower bounds (the percentages with the label *LB* in Fig. 9) represent the extreme case in which none of the injected faults was an equivalent mutant (i.e., the percentage with respect to all experiments); the upper bounds (the percentages with the label *UB* in Fig. 9) represent the other extreme in which all dormant faults were equivalent mutants (i.e., the percentage computed excluding the segments highlighted with a thicker border). When considering the lower bounds, interface corruptions occurred in 31.8% of all experiments, and ranged between 12.8% (*gobmk*) and 71.5% (*astar*) of the total. Moreover, if we consider the upper bounds, the interface corruptions represent the majority of the cases (55.6%, and ranging between 22.3% and 77.8%).

Summary and practical implications. The experimental results show that no individual failure mode is predominant, and that interface corruptions always represent a non-negligible share of component's behaviors. Therefore, error injection should not only encompass fail-stop behaviors of the component (e.g., crashes and hangs during the execution of the component's API), but should also include corruptions of its interface data.

5.3 Extent of interface errors (RQ2)

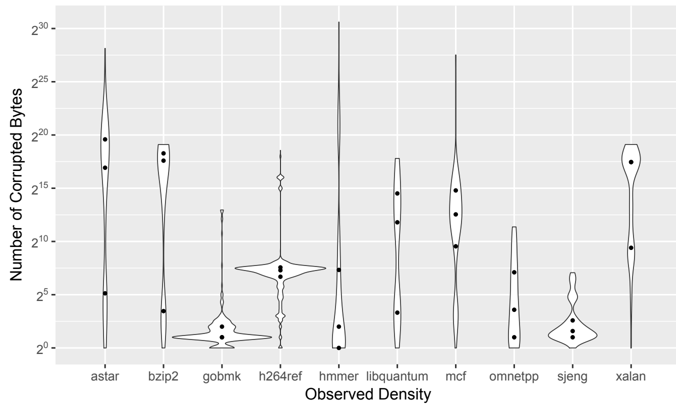
We show in Fig. 10 the distributions of the number of corruptions in interface data, respectively in terms of corrupted bytes and variables. We report the distributions for corrupted variables and corrupted bytes separately to account for the possibly large differences of bytes per variable. A `bool` variable, for instance, commonly has one byte, whereas a `long` integer variable has 8 bytes. If these two variables were accessible to the program via the component interface and all their bytes were corrupted, Figure 10a would report them as 9 corrupted bytes, whereas Figure 10b would report them as 2 corrupted variables. The dots in the violin plots represent the 25th, 50th, and 75th percentiles. In almost all target components, the distributions were bimodal or fat-tailed, with MBs of corrupted data in the most extreme cases. The target components with the largest amounts are *astar* and *mcf* (with data structures representing large graphs), *bzip2* and *libquantum* (with large arrays of data), and *xalan* (with large, nested classes to represent a document).

Concerning the research question 2, we are especially interested in interface corruptions that go above 10^1 corrupted bytes and 10^0 corrupted variables. We recall that these limits represent traditional error models, such as *bit-flips* of single memory words and CPU registers. The experimental results show that many (and, in some components, the majority) of the injections corrupt much more than few bytes and variables, since a large part of the distributions lie above these limits. For example, in the case of programs with large graphs, arrays or classes, the corruptions affect many of the elements contained in these structures. The only two components for which we observed a small amount of corruptions were *gobmk* and *sjeng* where, in most of the cases, the injected fault caused the algorithms to compute an incorrect decision, which was represented as a pair of (incorrect) integer variables; however, the distributions exhibit long tails even for these components (e.g., large corruptions affecting the tree that represents the decision space). We conclude from these experimental results that the corruption of individual bytes/variables is too restrictive to emulate the effects of software faults, and that interface error injection should introduce extensive corruptions, especially if the target component manages large, composite data structures. Instead, the corruption of individual bytes/variables is more appropriate for components that expose a small interface to their users.

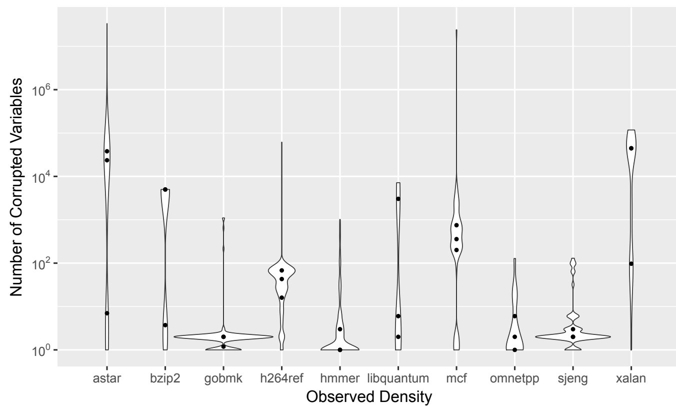
As for the extent over time of interface corruptions, we

0098-5589 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2018.2850755, IEEE Transactions on Software Engineering



(a) Corrupted bytes.



(b) Corrupted variables.

Fig. 10: Distributions of the extent of interface errors.

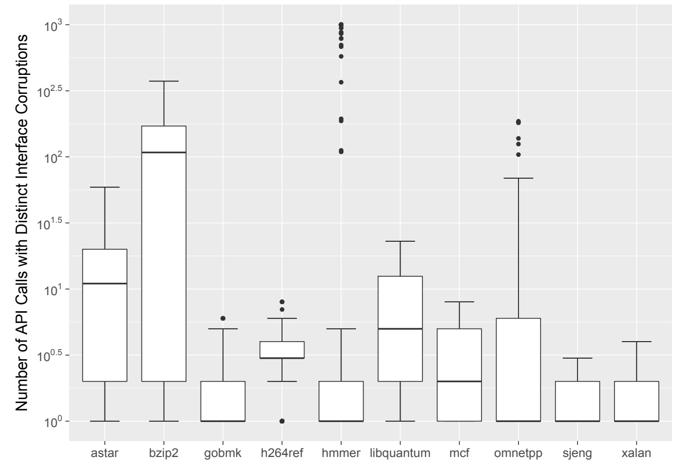


Fig. 11: Distributions of the number of API calls in which distinct interface errors occurred.

analyze how the occurrences of corruptions are spread over component API calls. Thus, we measured the number of API calls in which the target component produced new interface corruptions: the resulting distributions are showed in Fig. 11. We found that, for most of the programs, the new corruptions are concentrated in one or few API calls, while only in *bzip2*, *hmmer* and *omnetpp* the new corruptions were spread across up to hundreds of API calls.

We can notice in TABLE 3 that *bzip2*, *hmmer* and *omnetpp* have also the highest number of API calls (i.e., more than one thousand) under the considered workloads. This high number of API calls is due to the high number of loop iterations in which these APIs are called (e.g., in *bzip2*, the file to compress is split in fixed-size blocks, and the APIs are called for each block), increasing the likelihood that faults are repeatedly triggered during the same experiment. The *gobmk* program is the only target in which the new corruptions were concentrated in few API calls despite the high number of loop iterations: we found that the key data structures used by API calls are reset between loop iterations, thus making API calls independent and reducing the likelihood of spreading errors across the course of the same experiment.

Summary and practical implications. We found that the amount of interface errors caused by software faults is a fat-tailed distribution, contrarily to the assumption of few corrupted variables and bytes of SWIFI error models. Therefore, error injection should introduce a large number of corruptions, accordingly to the size of the interface data structures of the component. We also observed that the interface errors occur in one or few API calls, with the exception of APIs called within large loops and that operate over the same interface data, which can generate long sequences of API calls with interface errors. This result suggests that an interface error injector should concentrate the injections in one or few calls, except for APIs that are meant to be called within loops, where the injection should be triggered repeteadly. From a practical point of view, APIs meant for loops can be identified by performing a preliminary profiling of API uses (such as in TABLE 3).

5.4 Signaling of interface errors (RQ3)

We analyze the relationship between interface error corruptions and error signals, by considering three types of error signaling mechanisms adopted in our target programs: (i) APIs return a negative value (for integer return types) or false (for boolean return types); (ii) the component forces the termination of the program (e.g., by invoking special functions such as `assert`); (iii) the component triggers an exception from the OS (e.g., the *SIGSEGV* signal generated by a POSIX OS in the case of an incorrect memory access) that causes the termination of the program. Fig. 12 shows the relative percentages of these error signaling mechanisms for each target program; moreover, the figure further divides the cases with respect to the occurrence of interface corruptions (as in Fig. 9).

From these data, we can derive the relative frequency of three cases: (i) the fault caused interface corruptions, but the component did not raise any error signal to notify the problem; (ii) the fault caused interface corruptions, which were signaled by the component; and (iii) the component avoided interface corruptions, and just raised an error signal to notify its failure. We do not consider the cases with neither interface corruptions nor error signals, as they do not provide insights about the relationship between interface corruptions and error signals.

0098-5589 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2018.2850755, IEEE Transactions on Software Engineering

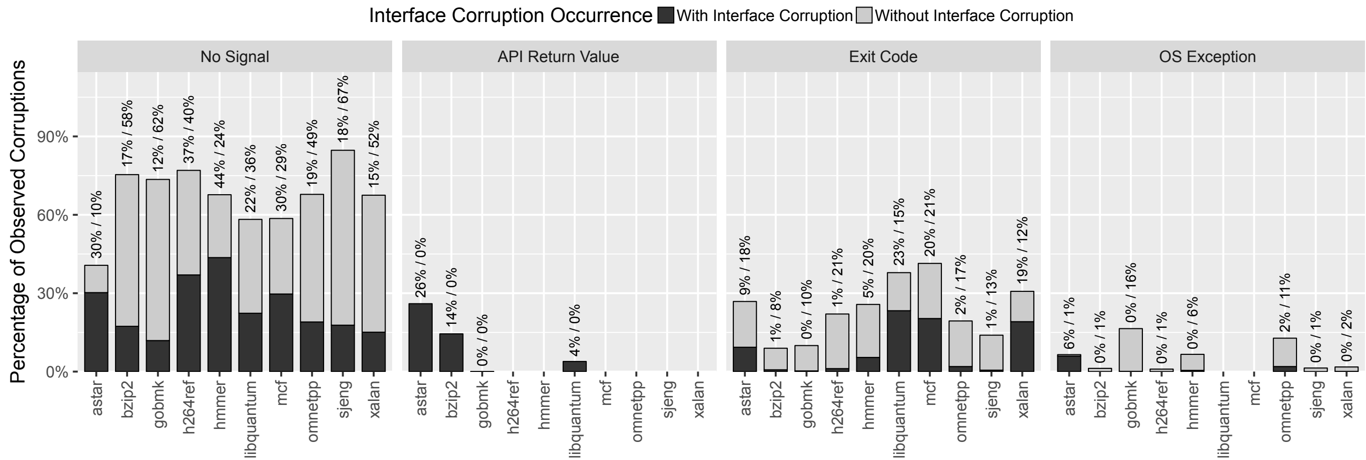


Fig. 12: Percentages of error signals.

Non-signaled ("silent") interface corruptions have been the most common case (45.8% of the experiments with either interface corruptions or error signals). The tendency towards "silent" interface corruptions highlights that it is too simplistic to emulate software faults by only injecting error signals, since the faults are often unnoticed by the component. Therefore, interface error injection should necessarily give emphasis to the corruption of interface data without error signals. However, the traditional approach of injecting error signals is still relevant, as error signals also occurred in a significant part of the experiments (38.2%); moreover, we found that interface corruptions and error signals can indeed occur together (16.0%). Thus, interface error injection should also cover these scenarios.

In the case of "signaled" interface errors, the latency of error signals (not plotted for conciseness) had been similar to the duration of the interface errors over time (previously discussed in § 5.3): in most of the cases, the interface errors were signaled by the same API call that propagates the errors for the first time (65.3% of cases), or closely after that API call (within 10 API calls in 86.4% of cases); in the remaining cases (13.6%), the interface errors were signaled only after a long sequence of API calls, e.g., at the completion of a loop as in the case of *bzip2*, *hmmer* and *omnetpp*.

Summary and practical implications. We observed that interface corruptions and errors signals do not always occur together, but may also occur independently (e.g., interface errors that are not signaled, or error signals generated before interface errors may occur). Therefore, error injection should cover all of the three cases to emulate component behaviors in a comprehensive way. Moreover, interface error injection should focus on injecting error signals that immediately follow interface corruptions; it should also inject delayed error signals for programs with a large number of loop iterations.

5.5 Patterns in interface errors (RQ4)

We further classify the interface corruptions according to the incorrect value written to interface data, by looking for recurring patterns of interface errors. The interface errors are classified with respect to their data type and the patterns defined in TABLE 2.

Fig. 13 (first plot on the left) summarizes the percentages of corruptions that fit any of the patterns for primitive types, including integer, pointers, characters, and floating point numbers, and their variants. The distributions show that the interface corruptions mostly tend to follow the expected patterns. Overall, 83.0% of corruptions belong to one of the patterns, while the remaining 17.0% did not belong to any of them. The percentage of corruptions that comply to the patterns ranged from 63.2% (*bzip2*) to 99.4% (*sjeng*).

In TABLE 4, we show the error patterns that cover most of the interface corruptions. As discussed in § 4.4 for the RQ4, we associate every interface error to exactly one pattern, and we apply a scheme of priority (i.e., from the most specific pattern to the less specific one) in case of ambiguity. The crossmarks ($\times$) point out the patterns (in the rows) that fit a significant part of the corruptions (at least 10%), and the blanks are patterns that cover less than 10% of the corruptions; we computed the percentages both with respect to interface corruptions observed for each program, and with respect to all corruptions (the "all" column). Moreover, in the left side of the table, we report the relative percentages of corruptions across the four primitive types.

In all components, the interface errors were covered by four error patterns at most. In the majority of cases, the corruptions affected integer data (56.6% of the total corruptions, and 98.8% of corruption excluding the *bzip2* program, discussed later). In particular, the majority of the components exhibited corruptions with special values (e.g., the 0 and 1 integers) and offsets from the correct value (e.g., the corrupted integer is close to the correct value). None of the programs exhibited a significant amount of "uninitialized" corruptions: even if the faulty component instantiates uninitialized data (such as in the case of the *MVIV* fault type in TABLE 1), these data do not necessarily surface as uninitialized (e.g., they instead divert the control flow, or they *crash* the component without propagating interface errors). We also observed bit-flip corruptions, but they were limited to few programs and exhibited low percentages

0098-5589 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2018.2850755, IEEE Transactions on Software Engineering

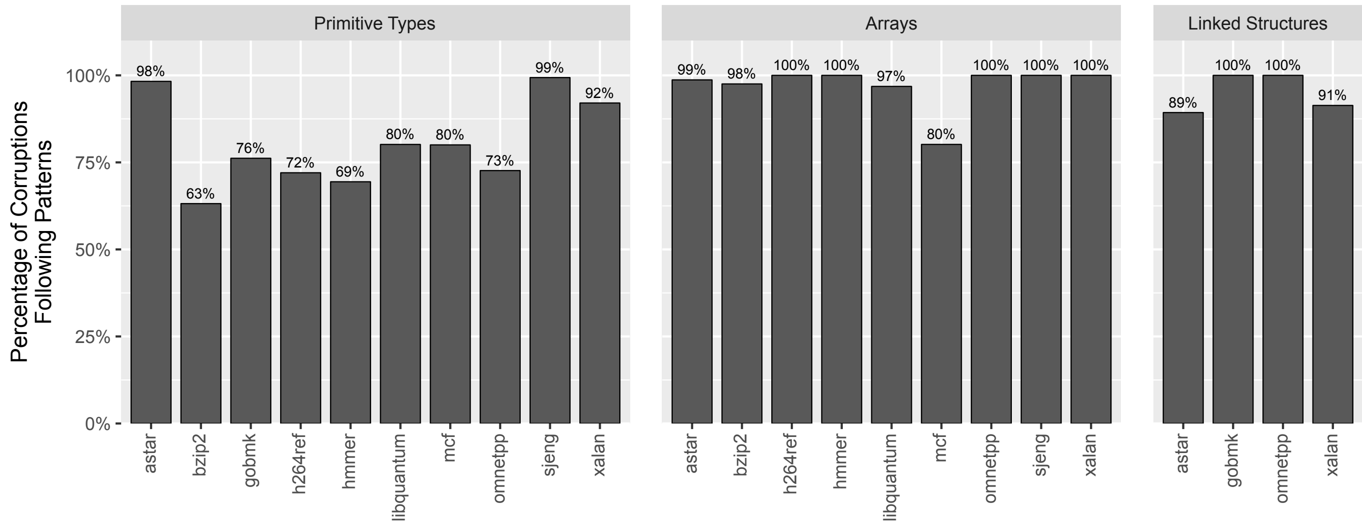


Fig. 13: Percentages of corruptions that follow any of the patterns of TABLE 2.

compared to the other error patterns.

For the remaining corruptions not fitting the patterns (i.e., the ones not included in the percentages of Fig. 13), the observed values were unrelated to the correct values or to special values: this happened most frequently for floating point numbers (where incorrect floating point computations significantly deviated from the expected result) and for characters that were used to handle binary data (e.g., compressed data in the case of *bzip2*, the only program were corruptions in characters were predominant, which account for 99% of all character corruptions in our experiments). Interestingly, pointers did not deviate in a purely random way (otherwise, we would have observed a high percentage of "invalid" pointers), but the corrupted addresses pointed to unrelated, still valid memory areas): for example, this happened for programs with very large linked data structures (such as *gobmk* and *omnetpp*), in which wrong pointers pointed out incorrect elements located in a distant part of the address space.

As for aggregate data types (arrays and linked structures), we also observed recurring corruption patterns. The percentages of corruptions covered by the patterns are summarized in Fig. 13 (respectively in the middle and rightmost barcharts). Moreover, TABLE 5 and TABLE 6 (with crossmarks on the patterns that cover more than 10% of corruptions) show that the corruptions are concentrated into few patterns.

The main observation is that corruptions did not randomly mix omissions, surplus and wrong elements. Instead, most of the faults only resulted in one kind of these corruptions (respectively labeled as "missing", "wrong" and "additional" in the tables). For arrays, in the majority of cases the fault only corrupted the existing elements (labeled as "wrong"), but did not change the number of elements with respect to the fault-free execution. Moreover, the corruptions were often concentrated on a subset of elements, either sparse or contiguous: 93.4% of array corruptions affected up to 100 elements, and 98.4% affected up to 1000 elements.

TABLE 4: Error patterns for corruptions of primitive types.

		astar	bzip2	gobmk	h264ref	hmmer	libquantum	mcf	omnetpp	sjeng	xalan	**all**
Integers (56.59%)	uninitialized											
	special	×					×	×	×	×		×
	pow small	×		×	×				×			×
	pow large											
	offset small	×		×	×	×	×		×		×	×
	offset large	×		×	×	×	×			×	×	×
	bitflip 1		×				×					
	bitflip 2											
Chars (42.85%)	uninitialized											
	special	×										
	ascii		×			×						×
	offset small	×				×						
	bitflip 1											
	bitflip 2					×						
Pointers (0.49%)	uninitialized											
	invalid											
	special	×							×		×	×
	pow small		×									
	pow large											
	offset small		×				×					
	offset large		×				×	×			×	×
	bitflip 1						×	×				×
	bitflip 2											
Floats (0.05%)	uninitialized											
	special	×				×	×					
	offset small											
	offset large										×	×
	bitflip 1					×			×			
	bitflip 2			×		×			×			

For large linked structures, we only consider the programs that adopt them (*astar*, *gobmk*, *xalan*). Similarly to arrays, most of the faults resulted either in truncated structures (the cases labeled as "missing"), or in corrupted elements while still preserving the original amount of elements (the cases labeled as "wrong"). Overall, the large majority

0098-5589 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2018.2850755, IEEE Transactions on Software Engineering

TABLE 5: Error patterns for corruptions of arrays.

	astar	bzip2	h264ref	hmmer	libquantum	mcf	omnetpp	sjeng	xalan	**all**
wrong contiguous few	×		×						×	×
wrong contiguous many	×					×				
missing few										
missing many										
surplus few										
surplus many										
wrong sparse few	×		×	×	×		×	×	×	×
wrong sparse many						×				
wrong contiguous most					×					
wrong sparse most		×								

TABLE 6: Error patterns for corruptions of linked structures.

	astar	gobmk	omnetpp	xalan	**all**
missing contiguous few				×	×
missing contiguous many		×			
missing contiguous most					
missing sparse few					
missing sparse many		×		×	
missing sparse most					
surplus contiguous few		×			
surplus contiguous many					
surplus contiguous most					
surplus sparse few					
surplus sparse many					
surplus sparse most					
wrong contiguous few	×		×		×
wrong contiguous many					
wrong contiguous most					
wrong sparse few	×				
wrong sparse many				×	
wrong sparse most					
mixed few					
mixed many					

of cases (more than 80% for every target component) fit the patterns. The corruptions were localized in a subset of elements of the linked structures (up to 100 elements in 67.7% of corruptions, and up to 1000 elements in 79.6% of corruptions), either sparse or contiguous.

Summary and practical implications. Interface data corruptions are not entirely random, but tend to follow recurring patterns. In particular, for integer data (which accounted for most of the corruptions), the majority of corruptions were special values (such as 0s) and offsets from the correct values; instead, other error patterns, such as bit-flips, were infrequent or only happened in specific programs. For floating-point numbers and chars, interface errors exhibited random values. For pointer data types, the corrupted address pointed to memory areas that were still correctly allocated (e.g., pointers to wrong elements in a linked data structure). For arrays and linked data structures, the majority of faults resulted either in missing or wrong elements, but did not mix omissions with commissions. Moreover, the corruptions were often focused on small subsets of elements (less than 10), or larger subsets in few cases (up to 1000 elements).

Overall, interface error injection should follow these patterns, as they fit the majority of interface corruptions. The experimental results point out that only focusing on special values and bit-flips (which are the error models assumed by most of the error injection tools [10], [16]) is not sufficient, since offsets are the largest group of interface errors. Moreover, error injection should also adopt random corruptions when the interface data include binary or floating point data; and should encompass truncations or corruptions of elements in aggregate data types.

6 Threats to Validity and Language Implications

In our experimental analysis, we identified and addressed the following potential threats to validity: the non-determinism of program executions, that may lead to spurious differences between the faulty and the fault-free executions that are not actually due to faults; the injection of code mutations, rather than real software faults, to investigate interface errors; and the choice of the case study software.

Case study software. If the selection of programs for our study listed in Table 3 is not representative of programs typically targeted by fault injections, this poses a threat to external validity: The results of our study might not generalize. To reflect the variety and complexity of real-world software components well, we analyzed ten targets from a popular benchmark, thus benefiting from the wide acceptance of the benchmark programs as realistic, both by academics and professionals [83]. As discussed in Section 5.1, the chosen benchmarks contain programs that are commonly applied in critical contexts, and they have been adopted in fault injection studies before. Moreover, the diversity of its programs with respect to size, architecture, and application domains allows us to achieve a reasonable spectrum of error behaviors; this diversity has also been confirmed by the variations in the experimental results, e.g., in terms of different shapes of distributions and of error patterns.

The problem of generalization aside, the experiments provide sufficient evidence to conclude that traditional error models are *not suitable* for emulating software faults by corresponding counterexamples. If traditional error models had been used for interface injections in the ten chosen programs, they would have led to injections that significantly differ from the effects observed in our study.

Applicability to other languages. As stated in the introduction, the focus of our work is on C and C++, as these languages dominate the application areas that fault and error injections are most commonly applied in. As the approach has been developed for C and C++ programs, our evaluation only covers programs written in these languages. However, the presented approach is, in principle, not limited to C/C++ and we outline the necessary adjustments to target other languages in Section 7.

Fault injection. Unfortunately, to the best of our knowledge, there is a scarcity of datasets suitable for our analysis. The existing artifacts used by the software testing community

0098-5589 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2018.2850755, IEEE Transactions on Software Engineering

(such as the Siemens suite [90]) in almost all cases provide only few faults; the programs are often relatively small; and the programs often do not represent reusable components with a realistic API. Code mutation allowed us to overcome this scarcity, but introduces a potential threat to validity: If the code mutations in our study were not representative of actual bugs, the observed interface effects could not be expected to be representative, either.

The emulation of software bugs by code mutations has become an accepted practice in software testing research, as several empirical studies showed that mutants can generate representative errors [91], [92], [93]. To ensure that the mutations in our study are representative of software bugs, we chose a tool that performs these mutations according to distributions of bug patterns identified by empirical studies of software bugs in production software [9], [14]. As we consider fault injections for emulating residual software bugs in off-the-shelf software components, the tool additionally filters out mutations that would be easily detected by trivial developer tests of the component. The representativeness of the chosen bug types (listed in Table 1) is discussed in detail in [9], [14] and the selection of representative bug locations in [14], [63].

Non-determinism. We designed the experimental methodology to make the fault-free execution reproducible, such as by avoiding variations due to differences in the memory layout (e.g., by replacing raw memory addresses with symbolic ones), thread scheduling, and I/O. In our design, we chose not to investigate the sensitivity of interface errors with respect to variations of thread scheduling, as the space of possible variations is extremely large and would require a separate study. To verify that these countermeasures address all relevant aspects of "benign" execution non-determinism that does not indicate symptoms resulting from our injections, we repeatedly executed fault-free versions of the case study software. We did not find any deviation across these repeated executions.

7 Result Implications on Interface Error Models and Conclusion

In this paper, we investigated the problem of emulating faulty software components by injecting *interface errors* at their interfaces with other components. To this aim, we analyzed interface errors resulting from more than 10 000 representative code mutations in ten software components. We looked at the interface errors from four perspectives (the research questions) to critically revisit the existing error injection techniques and to understand how to focus error injection to better emulate software faults. We here summarize the main conclusions from the analysis, by framing them with respect to the three general dimensions that drive error injection techniques: *what*, *when*, and *where* to inject.

What to inject. The error injection test plan should include both "fail-stop" behaviors (*crashes* and *hangs*) and "semantic" errors (i.e., corrupted interface data) of the target component. Concerning the interface corruptions, error injections should not be limited to random corruptions or to specific error models (bit-flips, boundary values, small offsets); instead, the error injection tests should cover all of these error models. Moreover, in the case of aggregate data types (arrays and linked structures), the error injection tests should focus on truncating the data structures, or replacing subsets of existing elements with incorrect ones. Finally, the error injection test plan should include both experiments where the interface errors are signaled (e.g., by returning error codes at API calls), and experiments where the interface errors are "silent".

Where to inject. Since we are considering errors propagated at components' interfaces, the errors have to be injected into interface data exchanged at API calls between the component and its user. The error injection test plan should include both experiments where the corruptions affect small, localized areas (e.g., specific variables, as in traditional hardware error models); and experiments that corrupt large parts of the interface data (e.g., by corrupting several member variables of the data structures), where the amount of corruptions should should be calibrated with respect to the size of the interface exposed to components' users.

When to inject. In interface error injection, the errors are introduced at component API calls, right after the control flow returns from the component to its user. According to our experiments, most of the interface errors are propagated by one or few API calls, therefore the test plan can focus the injection of errors at specific API calls. However, in the case of APIs meant to be called in loops, it is necessary to also inject interface errors repeatedly over the course of a sequence of invocations. Moreover, when injecting error signals, the signal should be injected at the same API call when the interface errors are also injected; in the case of a long sequence of API calls, error signals should also be injected towards the end of the sequence.

In summary, these results point out that the traditional error injection techniques, as used so far, do not accurately emulate software faults, but that richer interface errors should be injected (e.g., in terms of extent of corruptions over the interface data structures, and of diversity of error patterns to be injected). As future research direction for this work, we foresee the development of new interface error injection tools that take these insights into consideration. Such tools must be able of injecting combinations of the existing error models, including both silent data errors and error signals. Moreover, the distribution of errors must follow the size and the characteristics of the interface data structures of the components, as we have observed different distributions across different programs. Therefore, the error distributions should be configurable, ideally in an automated way, according to a preliminary static or dynamic analysis of the software (e.g., for identifying API calls meant to be used in a loop). It remains an open research problem how to mechanize this configuration, and to validate whether it can achieve a good approximation of the error distributions observed in this study.

The source code of the tool used for this study is at:
`https://github.com/rnatella/errordumper/`

The tool can be potentially extended to support other procedural and object-oriented programming languages, by adjusting three aspects of the tool. First, the reachability graph construction depends on the scope of the interface data, which is implied by the definitions of data types and

0098-5589 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2018.2850755, IEEE Transactions on Software Engineering

by pointer values at run-time. Thus, the construction of the reachability graph would require adjustment if the scope of data is affected by other factors in other languages. Second, we have implemented the dynamic part of our analysis, the execution tracing, via GDB. If a language is not supported by GDB, a different tracing mechanism is required, possibly by leveraging existing debugging tools for the language. Third, to classify interface errors, we adopt the patterns in Table 2 from previous studies. As these rely on language-specific data types, these patterns may require a redefinition or extension for other languages.

Finally, the experimental results of this study have also indirect implications for the design of fault-tolerant software systems. A significant part of software faults are not explicitly signaled by the software component (e.g., through exceptions or special return values); thus, error handling mechanisms that simply check error signals may be not sufficient, but they may require more sophisticated approaches (e.g., consistency checks over the semantics of interface data to detect the errors). Thus, the proposed analysis serves both for defining more accurate models for interface error injections, and for supporting more advanced forms of fault tolerance based on run-time verification.

Acknowledgments

This work has been supported by UniNA and Compagnia di San Paolo in the frame of "Programma STAR" (project "FIDASTE"), and in part by BMBF TUD-CRISP.

References

[1] J. Voas, "COTS software: The economical choice?" *IEEE Softw.*, vol. 15, no. 2, pp. 16–19, 1998.
[2] P. Bishop, R. Bloomfield, and P. Froome, *Justifying the use of software of uncertain pedigree (SOUP) in safety-related applications*. Health and Safety Executive Books, 2001.
[3] K. Petersen, D. Badampudi, S. Shah, K. Wnuk, T. Gorschek, E. Papatheocharous, J. Axelsson, S. Sentilles, I. Crnkovic, and A. Cicchetti, "Choosing Component Origins for Software Intensive Systems: In-house, COTS, OSS or Outsourcing? – A Case Survey," *IEEE Trans. Softw. Eng.*, vol. PrePrint, 2017.
[4] P. Koopman and J. DeVale, "The exception handling effectiveness of POSIX operating systems," *IEEE Trans. Softw. Eng.*, vol. 26, no. 9, pp. 837–848, 2000.
[5] J. Arlat, J. Fabre, M. Rodríguez, and F. Salles, "Dependability of COTS microkernel-based systems," *IEEE Trans. Comput.*, vol. 51, no. 2, pp. 138–163, 2002.
[6] E. J. Weyuker, "Testing component-based software: A cautionary tale," *IEEE Softw.*, vol. 15, no. 5, pp. 54–59, 1998.
[7] S. Eldh, S. Punnekkat, H. Hansson, and P. Jönsson, "Component testing is not enough–a study of software faults in telecom middleware," in *Testing of Softw. and Comm. Sys.*, 2007, pp. 74–89.
[8] J. Voas, "Certifying off-the-shelf software components," *IEEE Computer*, vol. 31, no. 6, pp. 53–59, 1998.
[9] J. Christmansson and R. Chillarege, "Generation of an error set that emulates software faults based on field data," in *FTCS*, 1996, pp. 304–313.
[10] R. Natella, D. Cotroneo, and H. S. Madeira, "Assessing dependability with software fault injection: A survey," *ACM Comput. Surv.*, vol. 48, no. 3, p. 44, 2016.
[11] L. Tan, C. Liu, Z. Li, X. Wang, Y. Zhou, and C. Zhai, "Bug characteristics in open source software," *Empirical Software Engineering*, vol. 19, no. 6, pp. 1665–1705, 2014.
[12] M. Sullivan and R. Chillarege, "Software defects and their impact on system availability: A study of field failures in operating systems," in *IEEE FTCS*, 1991.
[13] W. T. Ng and P. M. Chen, "The design and verification of the Rio file cache," *IEEE Trans. Comput.*, vol. 50, no. 4, pp. 322–337, 2001.
[14] J. Durães and H. Madeira, "Emulation of software faults: A field data study and a practical approach," *IEEE Trans. Softw. Eng.*, vol. 32, no. 11, pp. 849–867, 2006.
[15] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Trans. Softw. Eng.*, vol. 37, no. 5, pp. 649–678, 2011.
[16] M. Hsueh, T. Tsai, and R. Iyer, "Fault injection techniques and tools," *IEEE Computer*, vol. 30, no. 4, pp. 75–82, 1997.
[17] D. Cotroneo, A. Lanzaro, R. Natella, and R. Barbosa, "Experimental analysis of binary-level software fault injection in complex software," in *EDCC*, 2012, pp. 162–172.
[18] D. Cotroneo, A. Lanzaro, and R. Natella, "Faultprog: Testing the accuracy of binary-level software fault injection," *IEEE Trans. Depen. and Sec. Comput.*, vol. 15, pp. 40–53, 2018.
[19] Lockheed Martin, "Joint Strike Fighter, Air Vehicle, C++ Coding Standard," Tech. Rep., 2005.
[20] D. Dvorak, "NASA Study on Flight Software Complexity," in *Infotech@Aerospace Conf.* AIAA, 2009. [Online]. Available: https://arc.aiaa.org/doi/10.2514/6.2009-1882
[21] O. Bouissou, E. Conquet, P. Cousot, R. Cousot, J. Feret, K. Ghorbal, E. Goubault, D. Lesens, L. Mauborgne, A. Miné, S. Putot, X. Rival, and M. Turin, "Space Software Validation using Abstract Interpretation," in *DASIA*. European Space Agency, 2009, pp. 1–7.
[22] G. J. Holzmann, "Mars Code," *Commun. ACM*, vol. 57, no. 2, pp. 64–73, 2014.
[23] ISO 26262-4:2011, *Road vehicles – Functional safety – Part 4: Product development at the system level*. ISO, Geneva, Switzerland, 2011.
[24] ISO 26262-6:2011, *Road vehicles – Functional safety – Part 6: Product development at the software level*. ISO, Geneva, Switzerland, 2011.
[25] J. Barton, E. Czeck, Z. Segall, and D. Siewiorek, "Fault injection experiments using FIAT," *IEEE Trans. Comput.*, vol. 39, no. 4, pp. 575–582, 1990.
[26] G. Kanawati, N. Kanawati, and J. Abraham, "FERRARI: A flexible software-based fault and error injection system," *IEEE Trans. Comput.*, vol. 44, no. 2, pp. 248–260, 1995.
[27] K. Goševa-Popstojanova and K. S. Trivedi, "Architecture-based approach to reliability assessment of software systems," *Performance Evaluation*, vol. 45, no. 2, pp. 179–204, 2001.
[28] F. Brosch, H. Koziolek, B. Buhnova, and R. Reussner, "Architecture-based reliability prediction with the Palladio component model," *IEEE Trans. Softw. Eng.*, vol. 38, no. 6, pp. 1319–1339, 2012.
[29] L. Grunske and J. Han, "A comparative study into architecture-based safety evaluation methodologies using AADL's error annex and failure propagation models," in *IEEE HASE*, 2008.
[30] D. Nassar, W. Rabie, M. Shereshevsky, N. Gradetsky, H. Ammar, B. Yu, S. Bogazzi, and A. Mili, "Estimating error propagation probabilities in software architectures," New Jersey Institute of Technology, Tech. Rep., 2002.
[31] W. Abdelmoez, D. Nassar, M. Shereshevsky, N. Gradetsky, R. Gunnalan, H. Ammar, B. Yu, and A. Mili, "Error propagation in software architectures," in *IEEE METRICS*, 2004, pp. 384–393.
[32] P. Popic, D. Desovski, W. Abdelmoez, and B. Cukic, "Error propagation in the reliability analysis of component based systems," in *IEEE ISSRE*, 2005.
[33] V. Cortellessa and V. Grassi, "A modeling approach to analyze the impact of error propagation on reliability of component-based systems," in *ACM CBSE*, 2007.
[34] A. Mohamed and M. Zulkernine, "On failure propagation in component-based software systems," in *IEEE QSIC*, 2008.
[35] ——, "Failure type-aware reliability assessment with component failure dependency," in *IEEE SSIRI*, 2010.
[36] A. Filieri, C. Ghezzi, V. Grassi, and R. Mirandola, "Reliability analysis of component-based systems with multiple failure modes," in *ACM CBSE*, 2010.
[37] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J. Fabre, J. Laprie, E. Martins, and D. Powell, "Fault injection for dependability validation: A methodology and some applications," *IEEE Trans. Softw. Eng.*, vol. 16, no. 2, pp. 166–182, 1990.
[38] K. Goseva-Popstojanova, M. Hamill, and R. Perugupalli, "Large empirical case study of architecture-based software reliability," in *IEEE ISSRE*, 2005.
[39] M. Hiller, A. Jhumka, and N. Suri, "An approach for analysing the propagation of data errors in software," in *IEEE/IFIP DSN*, 2001.
[40] ——, "EPIC: Profiling the propagation and effect of data errors in software," *IEEE Trans. Comput.*, vol. 53, no. 5, pp. 512–530, 2004.

0098-5589 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

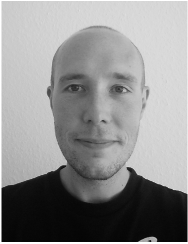
This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2018.2850755, IEEE Transactions on Software Engineering

[41] H. Madeira, D. Costa, and M. Vieira, "On the emulation of software faults by software fault injection," in *IEEE/IFIP DSN*, 2000, pp. 417–426.
[42] A. Johansson, N. Suri, and B. Murphy, "On the selection of error model(s) for OS robustness evaluation," in *IEEE/IFIP DSN*, 2007.
[43] S. Winter, C. Sârbu, N. Suri, and B. Murphy, "The impact of fault models on software robustness evaluations," in *ACM/IEEE ICSE*, 2011.
[44] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," *Comm. ACM*, vol. 33, no. 12, pp. 32–44, 1990.
[45] B. Marick, *The Craft of Software Testing*. Prentice-Hall, Inc., 1994.
[46] J. Voas, L. Morell, and K. Miller, "Predicting where faults can hide from testing," *IEEE Softw.*, vol. 8, no. 2, pp. 41–48, 1991.
[47] J. Voas, F. Charron, G. McGraw, K. Miller, and M. Friedman, "Predicting how badly "good" software can behave," *IEEE Softw.*, vol. 14, no. 4, pp. 73–83, 1997.
[48] P. Broadwell, N. Sastry, and J. Traupman, "FIG: A prototype tool for online verification of recovery mechanisms," in *Workshop on Self-Healing, Adaptive and Self-Managed Systems*, 2002.
[49] P. Marinescu and G. Candea, "Efficient testing of recovery code using fault injection," *ACM Trans. Computer Sys.*, vol. 29, no. 4, pp. 11:1–11:38, 2011.
[50] 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 *IEEE EDCC*, 2006, pp. 53–64.
[51] 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 *IEEE PRDC*, 2002.
[52] W.-I. Kao, R. Iyer, and D. Tang, "FINE: A fault injection and monitoring environment for tracing the UNIX system behavior under faults," *IEEE Trans. Softw. Eng.*, vol. 19, no. 11, pp. 1105–1118, 1993.
[53] S. Chandra and P. M. Chen, "How fail-stop are faulty programs?" in *FTCS*, 1998, pp. 240–249.
[54] A. Lanzaro, R. Natella, S. Winter, D. Cotroneo, and N. Suri, "An empirical study of injected versus actual interface errors," in *ACM ISSTA*, 2014, pp. 397–408.
[55] A. Avizienis, J. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Trans. Dependable Secure Comput.*, vol. 1, no. 1, pp. 11–33, 2004.
[56] A. Johansson and N. Suri, "Error propagation profiling of operating systems," in *IEEE/IFIP DSN*, 2005.
[57] G. Kanawati, N. Kanawati, and J. Abraham, "EMAX: An automatic extractor of high-level error models," in *AIAA ARC*, 1993.
[58] G. Ries, G. Choi, and R. Iyer, "Device-level transient fault modeling," in *IEEE FTCS*, 1994.
[59] J. Arlat, Y. Crouzet, J. Karlsson, P. Folkesson, E. Fuchs, and G. H. Leber, "Comparison of physical and software-implemented fault injection techniques," *IEEE Trans. Comp.*, vol. 52, no. 9, pp. 1115–1133, 2003.
[60] F. Cristian, "Understanding fault-tolerant distributed systems," *Comm. ACM*, vol. 34, no. 2, pp. 56–78, 1991.
[61] K. Kanoun and L. Spainhower, *Dependability Benchmarking for Computer Systems*. Wiley-IEEE Computer Society, 2008.
[62] "The SPEC Consortium: Members and Associates," https://www.spec.org/consortium/, accessed: 2018-03-20.
[63] R. Natella, D. Cotroneo, J. A. Durães, and H. S. Madeira, "On fault representativeness of software fault injection," *IEEE Trans. Softw. Eng.*, vol. 39, no. 1, pp. 80–96, 2013.
[64] R. Barbosa, J. Vinter, P. Folkesson, and J. Karlsson, "Assembly-level pre-injection analysis for improving fault injection efficiency," in *EDCC*. Springer, 2005, pp. 246–262.
[65] D. Cotroneo and R. Natella, "Fault injection for software certification," *IEEE Security & Privacy*, vol. 11, no. 4, pp. 38–45, 2013.
[66] 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 Trans. Softw. Eng.*, vol. 18, no. 11, pp. 943–956, 1992.
[67] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Predicting the location and number of faults in large software systems," *IEEE Trans. Softw. Eng.*, vol. 31, no. 4, pp. 340–355, 2005.
[68] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. MIT Press, 2009.
[69] P. Kang, "Function call interception techniques," *Software: Practice and Experience*, 2017.
[70] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A Fast Address Sanity Checker," in *USENIX ATC*, 2012.
[71] R. O'Callahan, C. Jones, N. Froyd, K. Huey, A. Noll, and N. Partush, "Engineering Record And Replay For Deployability: Extended Technical Report," *arXiv preprint arXiv:1705.05937*, 2017.
[72] J. W. Hunt and M. MacIlroy, *An algorithm for differential file comparison*. Bell Laboratories Murray Hill, 1976.
[73] E. W. Myers, "An O (ND) difference algorithm and its variations," *Algorithmica*, vol. 1, no. 1, pp. 251–266, 1986.
[74] "SPEC CINT2006 Benchmarks," https://www.spec.org/cpu2006/CINT2006/, accessed: 2017-07-31.
[75] J. Wei, A. Thomas, G. Li, and K. Pattabiraman, "Quantifying the accuracy of high-level fault injection techniques for hardware faults," in *IEEE/IFIP DSN*, 2014, pp. 375–382.
[76] L. Wang, Z. Kalbarczyk, R. K. Iyer, and A. Iyengar, "VM-μCheckpoint: Design, modeling, and assessment of lightweight in-memory VM checkpointing," *IEEE Transactions on Dependable and Secure Computing*, vol. 12, no. 2, pp. 243–255, 2015.
[77] M. Raiyat Aliabadi and K. Pattabiraman, "FIDl: A fault injection description language for compiler-based SFI tools," in *SAFECOMP*. Springer, 2016, pp. 12–23.
[78] S. S. Mukherjee, C. T. Weaver, J. Emer, S. K. Reinhardt, and T. Austin, "Measuring architectural vulnerability factors," *IEEE Micro*, vol. 23, no. 6, pp. 70–75, 2003.
[79] P. Racunas, K. Constantinides, S. Manne, and S. S. Mukherjee, "Perturbation-based Fault Screening," in *IEEE HPCA*, 2007, pp. 169–180.
[80] S. K. S. Hari, S. V. Adve, H. Naeimi, and P. Ramachandran, "Relyzer: Exploiting application-level fault equivalence to analyze application resiliency to transient faults," in *ACM SIGPLAN Notices*, vol. 47, no. 4, 2012, pp. 123–134.
[81] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum, "Enhanced operating system security through efficient and fine-grained address space randomization," in *USENIX Security Symposium*, 2012.
[82] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike, "Enforcing forward-edge control-flow integrity in GCC & LLVM," in *USENIX Security Symposium*, 2014.
[83] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.
[84] C. Goerzen, Z. Kong, and B. Mettler, "A survey of motion planning algorithms from the perspective of autonomous UAV guidance," *J. Intell. Robotic Sys.*, vol. 57, no. 1, pp. 65–100, 2009.
[85] M. Nicola and J. John, "XML parsing: A threat to database performance," in *ACM CIKM*, 2003.
[86] J. Grotendorst, D. Marx, and A. Muramatsu, "Quantum simulations of complex many-body systems: from theory to algorithms," *Publication Series John von Neumann Inst. for Comp.*, vol. 10, 2002.
[87] J. Cardoso, A. Sheth, J. Miller, J. Arnold, and K. Kochut, "Quality of service for workflows and web service processes," *Journal of Web Semantics*, vol. 1, no. 3, pp. 281–308, 2004.
[88] E. Galli, G. Cavarretta, and S. Tucci, "HLA-OMNET++: An HLA compliant network simulator," in *Proc. 12th IEEE/ACM Intl. Symp. Distributed Simulation and Real-Time Applications*, 2008, pp. 319–321.
[89] "SPEC CPU2006: Read Me First," https://www.spec.org/cpu2006/docs/readme1st.html, accessed: 2017-07-31.
[90] H. Do, S. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," *Empirical Software Engineering*, vol. 10, no. 4, pp. 405–435, 2005.
[91] M. Daran and P. Thévenod-Fosse, "Software error analysis: A real case study involving real faults and mutations," *ACM Soft. Eng. Notes*, vol. 21, no. 3, pp. 158–171, 1996.
[92] J. Andrews, L. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in *ACM/IEEE ICSE*, 2005.
[93] H. Do and G. Rothermel, "On the use of mutation faults in empirical assessments of test case prioritization techniques," *IEEE Trans. Softw. Eng.*, pp. 733–752, 2006.

0098-5589 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2018.2850755, IEEE Transactions on Software Engineering

Roberto Natella (Ph.D.) is assistant professor at the Federico II University of Naples, Italy, and co-founder of the Critiware s.r.l. spin-off company. His research interests include dependability benchmarking, software fault injection, and software aging and rejuvenation, and their application in operating systems and virtualization technologies.



Stefan Winter (Ph.D.) has obtained a doctoral degree in Computer Science from TU Darmstadt in 2015, where he is now working as a postdoctoral research fellow. His research focuses on the design and analysis of dependable software systems.

Domenico Cotroneo (Ph.D.) is associate professor at the Federico II University of Naples. His main interests include software fault injection, dependability assessment, and field-based measurements techniques. He has been member of the steering committee and general chair of the IEEE Intl. Symp. on Software Reliability Engineering (ISSRE), PC co-chair of the 46th Annual IEEE/IFIP Intl. Conf. on Dependable Systems and Networks (DSN), and PC member for several other scientific conferences on dependable computing including SRDS, EDCC, PRDC, LADC, and SafeComp.

Neeraj Suri (Ph.D.) holds the Chair Professorship in "Dependable Embedded Systems and Software" at TU Darmstadt, Germany. His research interests focus on design, analysis and assessment of trustworthy (dependable & secure) distributed systems and software. Suri currently serves as the associate Editor-in-Chief for the IEEE Trans. on Dependable and Secure Computing, and also served as editorial board member for IEEE Trans. on Software Engineering, ACM Computing Surveys, Journal of Security and Networks, and as editor for IEEE Trans. on Parallel and Distributed Systems. He is member of the IFIP WG 10.4 on Fault Tolerance and Dependability, and chaired the IEEE TC on Dependability and Fault Tolerance and the Steering Committee of the IEEE DSN conference. He has served as the PC Chair for multiple IEEE conferences such as DSN/DCCS, SRDS, HASE, ISAS, Microsoft-TUD RAF, and ICDCS.

0098-5589 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.