An Empirical Study of Injected versus Actual Interface Errors

Anna Lanzaro
DIETI, Federico II University of Naples, Italy
anna.lanzaro@unina.it

Roberto Natella
DIETI, Federico II University of Naples, Italy
roberto.natella@unina.it

Stefan Winter
DEEDS Group, Dept. of CS, TU Darmstadt, Germany
sw@cs.tu-darmstadt.de

Domenico Cotroneo
DIETI, Federico II University of Naples, Italy
cotroneo@unina.it

Neeraj Suri
DEEDS Group, Dept. of CS, TU Darmstadt, Germany
suri@cs.tu-darmstadt.de

ABSTRACT

The reuse of software components is a common practice in commercial applications and increasingly appearing in safety critical systems as driven also by cost considerations. This practice puts dependability at risk, as differing operating conditions in different reuse scenarios may expose residual software faults in the components. Consequently, *software fault injection* techniques are used to assess how residual faults of reused software components may affect the system, and to identify appropriate counter-measures.

As *fault injection* in components' code suffers from a number of practical disadvantages, it is often replaced by *error injection* at the component interface level. However, it is still an open issue, whether such injected errors are actually representative of the effects of residual faults. To this end, we propose a method for analyzing how software faults turn into interface errors, with the ultimate aim of supporting more representative interface error injection experiments. Our analysis in the context of widely used software libraries reveals that existing interface error models are not suitable for emulating software faults, and provides useful insights for improving the representativeness of interface error injection.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Fault tolerance; D.2.5 [**Testing and Debugging**]: Error handling and recovery

General Terms

Reliability, Verification, Performance

Keywords

Software Fault/Error Injection, Experimental Dependability Assessment, Software Faults/Errors, FMECA, Off-The-Shelf Software, Software Components

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ISSTA '14 July 21-25, 2014, San Jose, USA
Copyright 2014 ACM X-XXXXX-XX-X/XX/XX ...$15.00.

1. INTRODUCTION

The reuse of legacy or off-the-shelf software components plays a key role in the development of software systems. Unfortunately, component-based software development imposes significant risks for dependability [14, 41, 43]: When a component is reused in a new context, the system may use parts of the component that were previously seldom used and only lightly tested, or may interact with the component in unforeseen ways, thus exposing residual software faults in the component that had not been discovered before.

Software fault injection (SFI) is a valuable and advocated approach to assess the dependability of component-based systems in the presence of faulty software components, by deliberately introducing faults in its components[1] [10,13,16,41]. SFI enables several strategies for the dependability assessment of complex and component-based software, including:

- **Validating fault-tolerance mechanisms**: SFI can evaluate error detection and handling mechanisms (such as assertions and exception handlers) against component faults, and to improve such mechanisms if necessary (e.g., by introducing error checks) [2, 24, 32].
- **Aiding FMECAs (Failure Mode, Effects, and Criticality Analysis)**: Developers can quantify the impact of a faulty component on the overall system (e.g., in terms of catastrophic system failures), and mitigate risks by comprehensively testing the most critical components and revising the system design [16, 29, 42].
- **Dependability benchmarking**: SFI helps developers to choose among alternative systems or components the one that provides the best dependability and/or performance in the presence of other, faulty, components [22].

Despite the extensive development of various approaches, SFI remains a complex process, and technical limitations affect the feasibility and the quality of SFI experiments. A well-known approach is to introduce small code changes in the target component through *code mutations* (CM) [13, 30, 32], in a similar way to mutation testing but with different goals and approaches [20, 30]. This approach is able to emulate software faults in a *representative* way, i.e., the errors

[1] According to the definitions of [3], a *fault* is a software defect, and an *error* is an incorrect state caused by a fault.

generated by injected faults match the effects of real residual software faults in a component, as proved in [1, 11]. In turn, fault representativeness is a requirement for a trustworthy assessment of dependability properties, as discussed in Section 2 and in [26, 30]. Unfortunately, the mutation of components' code requires either the availability of source code for the target component, which may be impossible to obtain in the case of proprietary third-party components, or the ability to mutate the binary code. The latter has proven very difficult: In some cases it is impossible to correctly recognize and mutate high-level programming constructs in binary code [9]. Another issue with CM is *efficiency*, in terms of number of experiments that actually exhibit a component error, since injected faults may be difficult to activate and not produce any perceived error during the experiments [7].

Interface error injection (IEI) is an alternative SFI approach that overcomes these limitations of CM. IEI mimics the effects (i.e., errors) produced by faults in a component, by *injecting exceptional or invalid values* at the component's interface [15, 27, 44]. IEI is a practical and popular approach since it does not change the code of components, and assures the efficiency of experiments by definition since every injection produces an error. IEI approaches include:

- The injection of incorrect return values provided by external library functions, using the FST [15] and LFI [27] tools, to assess the effectiveness of error handling mechanisms of UNIX and Windows applications;
- The injection of incorrect system call parameters using pre-defined invalid data values, such as NULL pointers and out-of-bounds values with the BALLISTA tool [24], and random "flipping" of individual bits or bytes of parameters using the MAFALDA tool [2], to assess OSs against faulty user applications and device drivers [44];
- The injection of incorrect service invocation parameters with pre-defined invalid inputs, to assess SOA systems against faulty and malicious users [25];
- The random corruption of data words by bit-flipping heap, global, and stack areas, using the FIAT [4] and FERRARI [21] tools, to assess the effectiveness of error detection mechanisms within a program.

The representativeness of interface errors is less of an issue for traditional testing, where invalid values are useful at exposing inputs that lead to software failures. Nevertheless, the use of IEI for the representative emulation of component faults (as required by dependability assessment strategies [22, 29, 41]) is questionable, as there is a lack of evidence that IEI can realistically emulate software faults.

Paper Contributions. This work aims at analyzing how software faults in components' code result in errors at components' interfaces, in order to provide some constructive evidence towards more representative IEI techniques. Our paper makes the following contributions:

1. **A method for analyzing error propagation at the interfaces of software components.** Our method identifies how faults in software components manifest as interface errors. The method first injects faults in the software component under analysis by performing code mutations using a set of *representative* fault types that are based on *field failure data* on real faults from deployed software systems [7, 13]. Then, it instruments and executes the software component and identifies the effects of injected faults on the program that uses the component, including the corruption of data structures shared between the program and the component and erroneous return values from function calls. We have implemented a tool to perform the analysis in a fully automated manner.
2. **Experimental identification of representative errors.** We experimentally analyze interface error propagation for a set of three software components, distributed as libraries and widely adopted in real-world software applications. The analysis provides useful insights for injecting *representative* interface errors.

Paper Results. The key findings of the analysis are:

- Faults within components corrupt larger amounts of data than what is usually assumed by previous IEI techniques [44]. This suggests that the corruption of individual bits or bytes in interface parameter values cannot be considered representative of software faults.
- Erroneous return values from component invocations are accompanied by heap/stack data corruption: In almost all cases, when an error code is returned, data corruption also occurs. Therefore, both erroneous return values and data corruptions should be injected at the same time to achieve representativeness.
- Corrupted memory areas are correlated with the amount of memory accesses performed on that area: The more frequently the library accesses to a byte, the more likely that the byte will be corrupted by library faults.
- A considerable number of CMs do not produce any noticeable effect on the experiment, thus demonstrating the aforementioned efficiency issue with CM.

2. RELATED WORK

The representativeness of faults is a key property for the *quantitative assessment* of dependability properties through fault injection. In [33], Ng and Chen designed a write-back file cache with the requirement to be as reliable as a write-through file cache. To validate this requirement, software faults are injected in the OS to estimate the probability of data loss. Using fault injection experiments, the authors identified weak points of their file cache and iteratively improved its design until its reliability was comparable to a write-through cache. In [5], fault injection was adopted to evaluate whether the PostgreSQL DBMS exhibits fail-stop behavior in the presence of software faults, and to measure its fault detection latency. The study found that the transaction mechanism is effective at preventing fail-stop violations, reducing them from 7% to 2%. Kao et al. [23] performed a Markov reward analysis, based on fault injection experiments, to quantify the expected impact of faults on performance and availability. Tang and Hetch [39] proposed an approach for accelerating the probabilistic evaluation of high-reliability systems (e.g., with a failure rate in the order of 10^{-6}) that adopts fault injection to force the occurrence of *rare events*. In [42], Voas *et al.* inject errors within a program to identify where to place assertions and to avoid error propagation. The accuracy of these measures and the confidence

on fault tolerance mechanisms is based on the assumption that the injected faults are representative of real software faults. In [40], Vieira and Madeira proposed a dependability benchmark to evaluate different DBMS configurations with respect to operator and software faults in order to aid system administrators; in this case, a representative set of faults is required to make systems comparable and to identify the best configuration.

The representativeness of error injection techniques with respect to software faults was investigated in many studies. In order to accelerate the consequences of software fault injection experiments through error injection, Christmansson and Chillarege [7] proposed a methodology to derive a set of representative errors that match the effects of residual software faults of a system, by analyzing failure data at the users' site. They proposed to inject errors through bit-flipping, which corrupts program data at run-time by changing the contents of individual bits or bytes on heap, global, and stack areas, and mechanisms that were originally developed for emulating the effects of hardware faults [4,21]. The error types were derived as the *immediate effect* of fault activations on *internal program data* and classified according to the type of data corrupted by the fault (e.g., corruption of address vs. data words). Christmansson et al. [8] observed the benefits of such error injections over fault injections for evaluating the fault-tolerance of an embedded real-time system in terms of experiment setup and execution time. Their experimental analysis also showed that the lack of error representativeness has a noticeable impact on experimental results.

It must be noted that the approach of [7] can emulate the effects of software faults only to a limited extent, as Madeira et al. [26] showed that bit-flipping is not suitable for mimicking faults that involve several statements and complex data structures. Instead, Daran and Thévenod-Fosse [11] showed that code mutations are effective at emulating software faults, by observing an overlap of the error propagation of 12 known real faults and 24 mutations in a small safety-critical program. Nevertheless, their analysis focused on *internal errors* rather than *interface errors*.

As we are interested in how faulty components can affect other components, our focus is on *error manifestations at component interfaces*, rather than immediate effects on internal data of the targeted component as in [7,11]. Moraes et al. [28] and Jarboui et al. [19] investigated the representativeness of error injections at component interfaces, by comparing the failure distributions obtained from IEI and from CM, respectively. From a series of comparative experiments between fault injection based on representative code changes [13] and data-type-based interface errors commonly adopted in robustness testing (encompassing parameter corruptions through bit-flipping, boundary values such as -2^{31}, and invalid values such as NULL pointers [24,44]), they concluded that IEI and CM produce failures.

A limitation of previous analyses on error propagation [11,19,28] was that they were manually performed on a very small number of faults and on single programs, due to the lack of an automated tool for analyzing interface error propagation. Our study thus proposes an automated approach able to analyze arbitrary memory corruptions of component interface data, focusing on data exchanged via inter-component interfaces. Unlike previous tools for error propagation analysis by Kao et al. [23] and by Chandra and Chen [5], our method is able to precisely distinguish between the corruption of internal component data and of interface data.

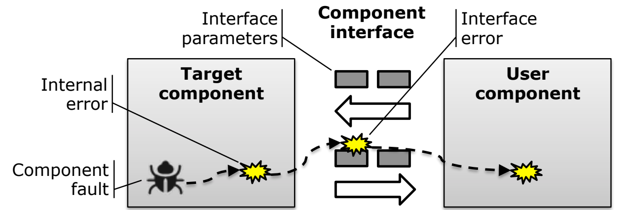

Figure 1: Relationship between component faults and interface errors.

3. PROPAGATION OF ERRORS AT COMPONENT INTERFACES

Our experimental analysis aims at identifying how *software faults* in a software component turn into *interface errors* that affect other components and the system as a whole. Figure 1 depicts the relationship between faults and errors: When a *component service* of the *target* component is requested through the *component interface* (e.g., through an API function call) by a *user* component, the target processes input data from the user, and provides results, by manipulating *interface parameters* provided by and returned to the user (e.g., data structures exchanged through input/output parameters and through the return value of a function invocation). During the execution of a component service, the activation of residual software faults in the component results in an *internal error* (e.g., corruption of internal data). When the component service terminates, the interface parameters exchanged between the target and the user components can be corrupted as an effect of such internal errors, thus producing *interface errors*. In such cases, we say that errors *propagate* from the target component to other components.

We consider software components in the form of *libraries* (i.e., collections of functions and classes) linked to a C/C++ main program at compile- or at run-time, as these languages are predominant in safety-critical control systems and systems software. However, the general approach applies for any type of software composition where components exchange data through shared data structures. Fig. 2 to 5 show the resulting error propagation paths for data errors in the case of library functions invoked from a (*main*) program.

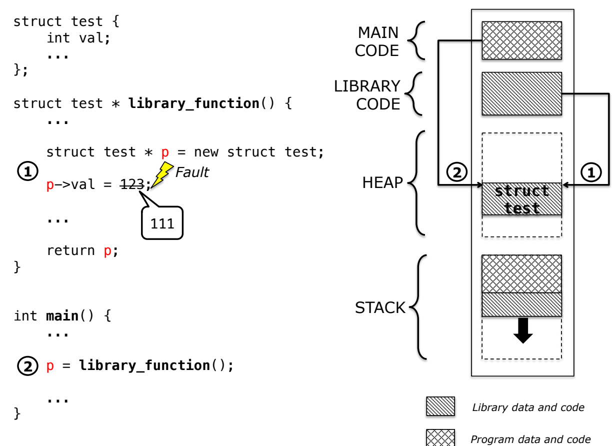

Figure 2: Propagation through a library-allocated heap area.

The first scenario (Figure 2) consists in the corruption of a data structure that is dynamically allocated on the heap by the library (①), where the corrupted data structure survives the component invocation (②) and is returned to the main program through a pointer return value (either on the stack or in a register, depending on calling conventions), which represents an erroneous interface parameter.

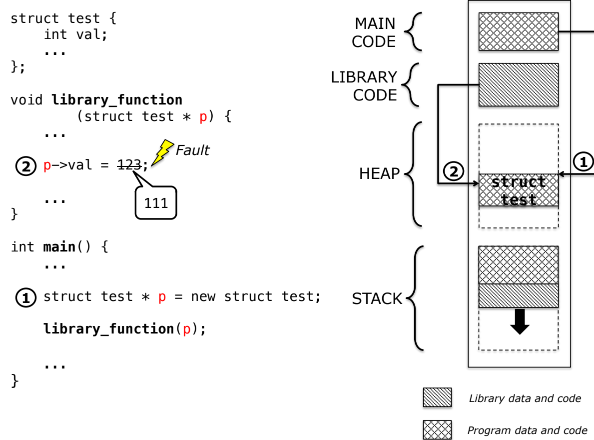

Figure 3: Propagation through a user-allocated heap area.

Figure 3 depicts a similar case, in which a data structure is allocated by the main program (①), passed to the library through a pointer interface parameter, and corrupted during the library invocation (②). The erroneous value represents an interface error, as it propagates to the main program by affecting an input-output interface parameter of the library.

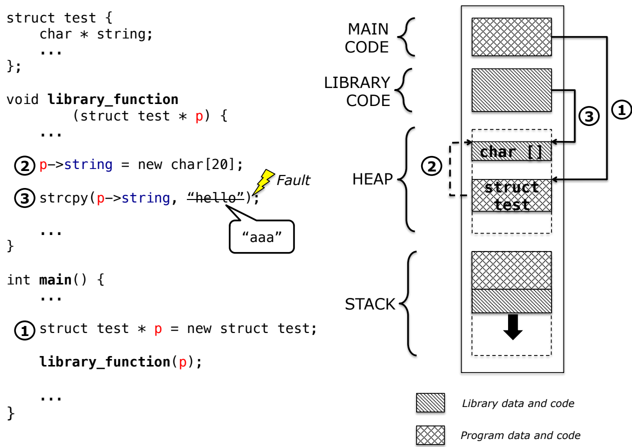

Figure 4: Propagation through a library-allocated heap area, reached through a user-allocated heap area.

Even if interface parameters are not directly corrupted, error propagation can still *indirectly* affect the main program by corrupting data that is pointed to by an interface parameter, such as in the case of complex data structures like trees and linked lists. This is the case in Figure 4, where a user-allocated data structure (①) is linked to a library-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 main program. This applies in general to any memory area reachable from an interface parameter through an arbitrary number of pointers.

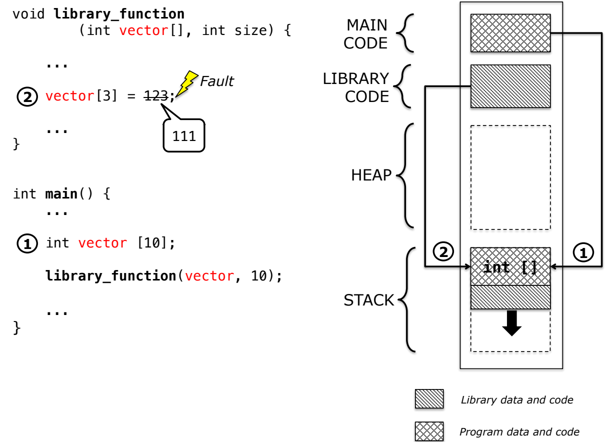

Figure 5: Propagation through a user-allocated local variable.

Finally, error propagation is not limited to heap areas, as in the case of Figure 5, in which an array is allocated as a local variable by the main program (①), a pointer to the array is passed to the library through an interface parameter, and the array's contents gets corrupted by the library (②). It is important to note that we are *not* analyzing internal errors that are not visible to the main program (i.e., memory areas not reachable outside the component). This is the case, for instance, of local variables allocated by the library, and of heap memory areas not reachable (neither directly through interface parameters, nor indirectly) by the main program.

4. PROPAGATION ANALYSIS

The proposed method enables the automated analysis of errors occurring at the interfaces of C/C++ software components, according to the workflow of Figure 6. First, the library is linked to a main program (which represents the *workload* of the experiment) and executed, collecting information about (i) memory *stores* made by the library, (ii) dynamic memory allocations of both library and main program, and (iii) library invocations performed by the program during the execution. The raw execution trace is pre-processed, in order to identify library memory *stores* that affect memory areas actually visible to the main program (such as the cases considered in Figure 2 to 5). The same steps are performed a second time, with a software fault deliberately injected into the library. Due to the injected fault, the library can generate different memory *stores* to interface parameter data, which leads to interface errors. To identify such interface errors, we compare the two execution traces and point out differences in terms of memory *stores* that write incorrect data (i.e., values differing from the fault-free execution), memory *stores* omitted by the faulty library, and superfluous memory *stores* that are only performed in the faulty execution.

To trace memory accesses performed by the target library, we perform a *dynamic binary instrumentation* (DBI) of the executable program [31]. In general, DBI techniques instrument a program during its execution by adding *analysis code* that collects data about the state of the execution. Uses of DBI range from simple analyses, such as profiling of function calls and code coverage, to more complex analyses, such as

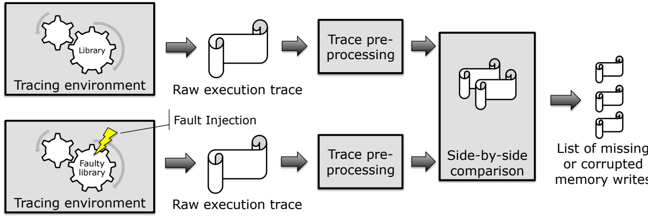

Figure 6: Overview of error propagation analysis.

undefinedness of program variables. In particular, we adopt the *disassemble-and-resynthesise* DBI approach [31], which translates the original program (native code) into an *intermediate representation* (IR), instruments the IR, and translates the IR back to native code, which is then executed on the native system. The IR code consists of architecture-independent, RISC-like instructions that perform individual operations such as memory stores (in contrast to a native CISC-like code instruction, such as x86 instructions, that can have several side effects). DBI takes advantage of conventional compiler optimizations, such as code caching, in order to accelerate the process of instrumentation. Analysis code is mixed with the original IR code to obtain an instrumented IR code: for instance, to track memory modifications, the DBI can add one or more IR instructions after each IR *store* instruction, to record the accessed address and the value written to that address. This approach allows to analyze memory accesses made by a program at a fine grain, which is the objective of the analysis of this study.

We developed a DBI analysis tool for tracing library code on top of the Valgrind program analysis framework [31]. Our tool inserts the following analysis code at run-time:

- After each instruction, we insert code to check the instruction address to detect whether the control flow moved from the main program to the code of the target library (i.e., the program enters in *library context*). In a similar way, we check whether the control flow returns from the library to the main program. We record the name of the invoked library function (which is obtained from the symbol table included in the library), and the return value of the library invocation.

- When library context is entered, we record the current value of the stack register, which marks the end of the stack frame of the main program (containing local variables of the main program) and the beginning of the stack frame of the library (containing local variables of the library). While the execution is in library context, we record changes to the stack register, in order to trace the growth of the library stack frame and, ultimately, to identify writes to local variables of the main program (which are stored on the stack) and to discard writes to local variables allocated by the library.

- While in library context, after each IR *store* instruction, we insert code for recording the address of the instruction that writes to memory, the address and the size of the area being written, and the new contents of the memory area. The DBI tool records memory accesses to heap and global data (e.g., Figures 2 to 4), and to data in the stack frame of the main program (e.g., Figure 5).

Moreover, the tool wraps and intercepts the invocation of the following functions of the C library:

- Invocations of `mmap()`, which is invoked at run-time by the loader to link a shared library to the address space of the process: We record the addresses of memory areas in which library code and data are mapped.

- Invocations of memory allocation functions (e.g., `new`, `malloc()`), both in library context and in the main program: We record the address and the size of each allocated and freed memory area, and the code location that allocated that memory area. This information is used later in the analysis for identifying memory areas reachable by the main program.

As a result, the execution trace obtained from the DBI tool provides (i) all invocations of and returns from library functions (LIB_INVOCATION and LIB_RETURN events), and their return value, (ii) all memory writes made by the library outside its local variables (STORE events), and (iii) all memory allocations and deallocations (ALLOCATION and FREE events).

The trace is then processed (Figure 6) to identify memory *stores* that *write data accessible by the main program*, that is, interface parameter data. These data are identified by building a *graph*, where nodes represent *memory areas* (i.e., a range of contiguous memory addresses, such as an array of bytes allocated on the heap), and edges represent *pointer-pointee* relationship between memory areas (i.e., a memory area contains a pointer variable, pointing to another memory area). A memory area is *reachable* by a program using the library if there is a path in the graph between that memory area and any variable of the user program, i.e., a variable in the user heap (represented by the *UH* node), in the user stack (*US* node) or an output value from the library function call (*O* node). Figure 7 shows an example.

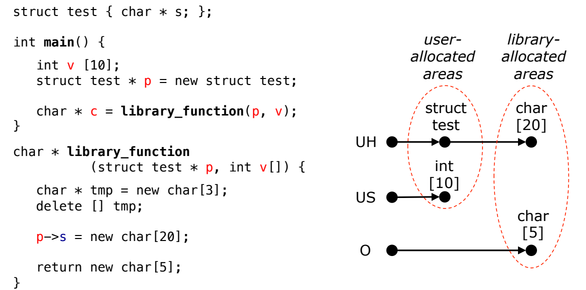

Figure 7: Example of reachability graph.

Figure 8 provides the detailed algorithm for building and analyzing the graph. The trace is analyzed in three passes. Each pass processes events of the trace in sequential order, by invoking for each event a function according to the type of event (e.g., when a STORE event is encountered while scanning the trace, it is processed by invoking HANDLE_STORE).

Pass 1. This pass (Fig. 8a) identifies heap memory areas that are allocated and de-allocated within the same library invocation (i.e., "temporary" memory areas used during an individual library invocation, such as "tmp" in Figure 7), and removes them from the analysis (Fig. 8a, line 20), since these areas cannot be accessed by the user program (they do not "survive" a library invocation). The remaining heap areas can still potentially be accessed by the main program.

```
1: Allocs ← LibraryGlobalAreas
2: AllocsLib ← ∅

3: function HANDLE_LIB_INVOCATION
4:     library_context ← 1
5: end function

6: function HANDLE_LIB_RETURN
7:     library_context ← 0
8:     Allocs ← Allocs ∪ AllocsLib
9:     AllocsLib ← ∅
10: end function

11: function HANDLE_ALLOCATION(new_alloc)
12:     if library_context = 1 then
13:         AllocsLib ← AllocsLib ∪ {new_alloc}        ▷ library allocation
14:     else
15:         Allocs ← Allocs ∪ {new_alloc}              ▷ user allocation
16:     end if
17: end function

18: function HANDLE_FREE(alloc)
19:     if library_context = 1 then
20:         AllocsLib ← AllocsLib \ {alloc}
21:     end if
22: end function
```

(a) Pass 1: Collection of memory allocations.

```
1: E ← GET_LIBRARY_ALLOCATED_AREAS(Allocs) ∪ {UH, US, O}
2: V ← ∅

3: function HANDLE_STORE(store)
4:     pointed_area ← INTERVALSEARCH(Allocs, store.value)

5:     if pointed_area ≠ ∅ ∧ (IS_LIB_HEAP_AREA(pointed_area) ∨
   IS_LIB_GLOBAL_AREA(pointed_area)) then

6:         accessed_area ← INTERVALSEARCH(Allocs, store.address)
7:         if accessed_area ≠ ∅ then
8:             if IS_USER_HEAP_AREA(accessed_area) then
9:                 V ← V ∪ (pointed_area, UH)
10:            else
11:                V ← V ∪ (pointed_area, accessed_area)
12:            end if
13:        else if IS_USER_STACK_DATA(store.address) then
14:            V ← V ∪ (pointed_area, US)
15:        end if
16:    end if
17: end function

18: function HANDLE_LIB_RETURN(returned_value)
19:     pointed_area ← INTERVALSEARCH(Allocs, returned_value)
20:     if pointed_area ≠ ∅ then
21:         V ← V ∪ (pointed_area, O)
22:     end if
23: end function
```

(b) Pass 2: Generation of the reachability graph.

```
1: Trace ← ∅

2: function HANDLE_STORE(store)
3:     area ← INTERVALSEARCH(Allocs, store.address)
4:     address ← store.address
5:     if IS_LIB_HEAP_AREA(area)∨IS_LIB_GLOBAL_AREA(area) then
6:         if IS_REACHABLE_BY_USER(V, E, address) then
7:             Trace ← Trace ∪ {store}
8:         end if
9:     else if IS_USER_HEAP_AREA(area) ∨
   IS_USER_STACK_DATA(address) then
10:        Trace ← Trace ∪ {store}
11:    end if
12: end function
```

(c) Pass 3: Event filtering.

Figure 8: Trace pre-processing.

The *address ranges* $[start, end)$ of remaining heap areas are arranged in an *interval tree*, the *Allocs* set (Fig.8a, line 8), which is a data structure that allows to search for ranges containing a given value: We use this feature in subsequent passes to find, for a given address in the trace, the heap area to which that address belongs. Address ranges of global data structures of the library (obtained from the library symbol table) are also inserted in the interval tree at the beginning of the pass (Fig. 8a, line 1).

Pass 2. This pass (Fig. 8b) constructs a directed graph (E, V) representing pointer-pointee relationships between library-allocated memory areas, and between these areas and memory areas of the user program. Node A of the graph (representing a memory area A) is connected to node B if the area A contains a pointer with an address to the memory area B (i.e., B is "reachable" by A). The graph includes a node for each library-allocated memory area. Moreover, we introduce in the graph the UH, US, and O nodes (Fig. 8b, line 1): if a node A is connected to any of these nodes, then the memory area A is directly reachable through user-allocated heap memory, user stack memory, or an output value of a function call, respectively. To identify pointer-pointee relationships, we check the value written by *store* operations (*store.value*) and see whether that value represents an address within one of the memory areas in *Allocs*: if this is the case, then the written value represents a pointer, and the two areas (i.e., the one containing the pointer, and the one with the pointed address) are connected in the graph (Fig. 8b, line 11). If a library-allocated heap/global area is pointed to by user heap areas, the user stack, or values returned by library invocations, that library-allocated area is connected to UH, US, or O, respectively (Fig. 8b, lines 9, 14, 21). This pass uses an interval tree search in *Allocs* (Fig. 8b, line 4) to identify pointers and the areas they point to.

Pass 3. It identifies memory *stores* to areas that are reachable by the main program (Fig. 8c). If the address of the *store* (*store.address*) belongs to a library-allocated area, the algorithm inspects the graph using the IS_REACHABLE_BY_USER function (Fig. 8c, line 6) to find whether the area is reachable outside the library, and only adds the *store* to the final trace if there exists a path in the graph between the memory area and one of the US, UH, or O nodes (i.e., the area is reachable by the user). *Stores* on user-allocated memory are also included in the trace (Fig. 8c, line 10).

After the execution of an experiment and of pre-processing, we obtain a trace consisting of a sequence of tuples, each representing a memory *store* performed by the library on user-reachable memory. A tuple is defined as: <*instruction address*, *memory address*, *store size*, *stored value*>. A "faulty" execution trace is then compared with a "fault-free" execution trace. Given that execution traces are always identical when the target software is executed without faults (effects of non-determinism must be factored out, as discussed below), any differences between the faulty and the fault-free traces are actually due to injected faults. Traces are compared by searching for the *longest common subsequences*, using the algorithm described in [18]: it aligns two sequences such that two tuples at the same position in the aligned sequences will have the same values, by comparing, respectively, the instruction, the address, the size and the value of memory *stores*. In the example of Figure 9, the first and the third *stores* of both sequences are aligned; the *stores* at the second position are performed by the same instruction on the same memory area (a heap area allocated at *buf.c:158*), but

Fault-free trace / **Faulty trace**

```
Fault-free trace                                              Faulty trace
Instruction  Address             Size  Value                  Instruction  Address             Size  Value

buf.c:613,   HEAP-buf.c:158+20,  8,    0000000000000004       buf.c:613,   HEAP-buf.c:158+20,  8,    0000000000000004
buf.c:614,   HEAP-buf.c:158+c,   4,    00002002          |    buf.c:614,   HEAP-buf.c:158+c,   4,    00004004
buf.c:614,   HEAP-buf.c:158+8,   4,    00000004               buf.c:614,   HEAP-buf.c:158+8,   4,    00000004
buf.c:616,   HEAP-buf.c:171+4,   1,    00                >                       missing store
```

Figure 9: Example of comparison between faulty and fault-free traces.

a wrong value is written in the faulty execution; the fourth *store* is only performed in the fault-free execution, while it is omitted in the faulty one. In this example, 4 bytes are corrupted by writing a wrong value at the second position, and another byte is corrupted since its initialization is omitted at the fourth position. We also detect corruptions due to spurious *stores* not performed in the fault-free execution. In a similar way, we compare *return values* of library invocations.

When comparing faulty and fault-free traces, we focus on memory *stores* and return values produced by *the first library function invocation that exhibits differences from fault-free executions.* The differences exhibited by subsequent invocations of library functions need to be discarded since they may not be due to the injected fault, but due to an incorrect behavior of the main program caused by the first "faulty" library invocation. Focusing on the first faulty library invocation is more precise and avoids confusion between effects. Our approach is able to catch interface errors produced both by the injected library function, and by library functions that are indirectly impacted by the injected one.

Another important aspect that we needed to take into account in the design of our DBI technique is the degree of *non-determinism* in execution traces. The comparison of traces in faulty and fault-free conditions (as depicted in Figure 6) 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 took into account the following sources of non-determinism:

Memory management. A dynamically-allocated memory area can be mapped at different addresses in different executions. To enable the comparison of *store* operations performed on the same heap area, we rewrite memory addresses in the trace by replacing *absolute addresses* of heap memory areas with *relative addresses* within that area. Relative addresses are composed by a pair <*area id*, *offset*> (e.g., Figure 9); the *offset* represents the distance between the beginning of the heap area and the address being rewritten, and the *area id* is a number that uniquely identifies the allocation, which is computed from the code location where the area was allocated, the call stack at the time of allocation, and an incrementing integer. This allows to identify two identical *stores* (i.e., *stores* performed by the same instruction, on same heap area, and with the same value) even if the heap area is mapped at different addresses. In a similar way, the trace is rewritten to replace addresses belonging to global areas with relative addresses.

Thread scheduling. The program execution flow and, therefore, the sequence of *stores* performed during the execution, can vary among executions due to thread scheduling. *Recording and replay* techniques can be adopted to mitigate this source of non-determinism [35, 38]: the reference execution (i.e., the execution without faults) can be recorded, and then replayed while executing the faulty version of the target software. Since the current implementation of our DBI tool does not support deterministic recording and replay, in the experiments of this work we focus on single-threaded workloads, and plan to extend the analysis to multi-threaded workloads in future. Previous studies on recording and replay for DBI [35,38] (unfortunately not supported by the Valgrind framework at the time of writing) makes us confident that the approach is applicable to multi-thread software.

I/O operations. Similarly to thread scheduling, the timing and the contents of I/O operations can affect the execution flow and the sequence of *stores* of a program. Non-determinism due to I/O timing can be avoided if the effects of thread scheduling are avoided, either through recording and replay or by focusing on single-threaded applications: In the case of recording and replay, the deterministic thread scheduling makes the execution tolerant to variations in the timing of I/O operations; in the case of single-threaded applications, the execution is insensitive to I/O timing. 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.

5. COMPONENT FAULT INJECTION

To inject software faults in library code, we use the approach and the automated tool (SAFE) described in [10,30]. The tool injects a set of *representative* fault types (Table 1), which were defined on the basis of field data on *real software faults* found in deployed software systems, both commercial and open-source [7,13]. The SAFE tool injects these fault types by mutating the source code instead of the binary code, which assures a high degree of accuracy of fault injection experiments [9]. The tool automatically identifies *code locations* in which faults can be injected, and *code changes* for realistically emulating the fault types of Table 1. Each injected fault produces a distinct *faulty version* of the target library code (Figure 10), which replaces the original code.

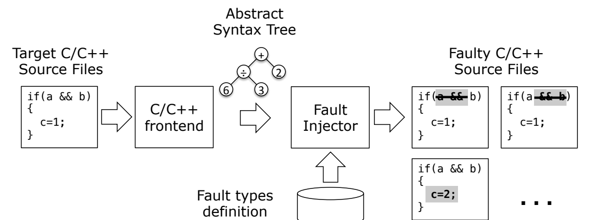

Figure 10: Software Fault Injection approach [10,30].

The representativeness of injected faults is an important requirement for obtaining a realistic profile of interface errors

Table 1: Fault types adopted in this study [13].

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

generated by software faults. Compared to the mutation operators proposed in the literature for the C language, the considered fault types are more selective and only encompass faults most frequently found in the field (12 fault types against 71 mutation operators proposed in mutation testing studies [20]). This reflects the fact that mutation operators inject many kinds of faults that can occur before and during coding and are used to assess the thoroughness of test cases, while the fault types of Table 1 represent faults that tend to escape the whole development process (including testing), and are not designed for improving test suites but assessing fault tolerance properties. These fault types also provide several detailed rules ("constraints"), not shown for brevity, describing the *code context* in which fault types should be injected to be representative of field faults [13]. For instance, the removal of an `if` construct is injected in those `if` constructs that enclose at most 5 statements, since it is unlikely that an `if` construct is lacking for larger groups of statements. Moreover, the proportions of injected faults follow the distribution of fault types in the field [13]. Fault types are grouped in 4 classes, according to the Orthogonal Defect Classification (ODC) [6]: Assignment (ASG), Algorithm (ALG), Checking (CHK), and Interface (INT).

6. EXPERIMENTAL ANALYSIS

We performed a set of experiments using the proposed approach for interface error propagation analysis, in order to understand how software faults inside widely-adopted libraries surface as errors at their interfaces. Our analysis aims at answering the following research questions: *Can the interface errors injected by existing tools be considered representative? How we can improve these tools to achieve better representativeness?* Therefore, we analyze interface errors in terms of amount and distribution of corrupted bytes and of erroneous return values, and compare them with the types of errors injected by existing IEI tools (see Section 1).

6.1 Setup

We analyzed the interface errors generated by bugs in three real-world software libraries (Table 2). These libraries are complex software projects on their own, and define a set of APIs and interface data structures to be used by external applications. *SQLite* (v3.7.16.2) is an SQL database engine that implements most of the SQL-92 language, provides advanced features such as ACID transactions, and has been adopted in many well-known proprietary and open-source projects [17]. *Libxml2* (v2.9.0) is a library for parsing and generating XML documents according to W3C standards; besides several popular open-source projects, we found that it is also adopted in a mission-critical middleware for configuring and deploying CORBA services in air traffic control systems [34]. *Libbzip2* (v1.0.6) implements a compression algorithm that is used in many systems for its ability to achieve high compression rates with good performance [36].

As workload of the experiments, we adopted tests and demo programs that are distributed with each library. These programs are linked to the target libraries and exercise its functions using pre-defined inputs. We injected faults in library code exercised by the workload. For *Libbzip2*, the workload compresses and decompresses a set of sample files, covering 72.9% of library code. For *SQLite*, the workload is a set of SQL queries that create a database and update and retrieve data from it, and for *Libxml2* the workload parses a set of XML files with several types of tags, and serializes XML files to the disk. Given the large size and complexity of these two projects (they are complete implementations of the SQL and XML languages), we focused experiments on the most important functionalities, covering respectively 32.0% and 19.0% of the code; we discarded experimental, secondary or deprecated functionalities. Experiments were executed on a Fedora Linux x86-64 PC.

Table 2: Libraries analyzed in this study.

Name	Description	Size (loc)	Faults
Libxml2	XML C parser and toolkit	155k	1471
Libbzip2	Lossless data compressor	6k	463
SQLite	Transactional SQL database engine	78k	1023

6.2 Results

In our fault injection experiments, faults led to the following outcomes:

- **Crash**: the experiment is terminated by the OS due to an exception (e.g., due to an invalid memory access).
- **Hang**: the experiment is stalled, i.e., it does not terminate within a given amount of time (much larger than the duration of a fault-free execution).
- **Wrong**: the experiment produces an incorrect output, i.e., different from the output in fault-free conditions.
- **Pass, corrupted**: the experiment produces a correct output; interface errors occurred and were tolerated.
- **Pass, no corruption**: the experiment produces a correct output, and the fault did not cause interface errors.

Table 3 provides the distributions of failure types for each target library. In many cases, the output of experiments was correct even in the presence of an injected fault. By analyzing interface parameter data exchanged at component interfaces, we found that, in the 61.8% of experiments, there were neither incorrect outputs nor corruptions at component interfaces: in these experiments, the fault was not activated

Table 3: Outcomes of experiments.

Target	Outcome				
	Crash	*Hang*	*Wrong*	*Pass, corrupted*	*Pass, no corruption*
Libxml2	70 (4.8%)	20 (1.4%)	233 (15.8%)	147 (10.0%)	1001 (68.0%)
Libbzip2	6 (1.3%)	0 (0.0%)	39 (8.4%)	83 (17.9%)	335 (72.4%)
SQLite	122 (11.9%)	16 (1.6%)	182 (17.8%)	213 (20.8%)	490 (47.9%)

(even if faulty code was covered by the workload), or there was no error propagation to component interfaces. This result demonstrates that efficiency can indeed be an issue for CM approaches, as many injections do not produce effects on experiments [7]. In total, we obtained 1131 failures from fault injection experiments (38.2% of the total), on which we performed more detailed analyses on interface errors. This set of failures is much larger than previous studies [19, 28].

We first determined the extent of corruptions of interface parameters, in terms of number of bytes affected by faults. For the experiments that resulted in interface data corruptions, Fig. 11 provides the empirical cumulative distribution of the number of corrupted bytes, for each target library. The number of corrupted bytes ranges from single (10^0) bytes to thousands of bytes for all three libraries and this number depends on the types of the data structures and library functions affected by the fault. An important result, which holds for all three targets, is that 50%-60% of faults affect much more than 8 bytes, which is the size of a memory word in our target system (i.e., the maximum size of addresses or data that CPU instructions operate on). Less than 40% of faults are limited to a memory word, while the median of the number of corrupted bytes ranges between 50 and 110 bytes. This is an important finding for the design of representative interface error models, since it indicates that the traditional ones based on the corruption of individual bits or bytes on heap, global, and stack areas [4, 21], are not suitable for emulating interface errors produced by software faults. Fig. 12 provides the distribution of the number of corrupted bytes, split by ODC fault types (see Table 1). This figure shows that only the 20-30% of Algorithm, Checking, and Interface faults affect at most 10 bytes. Only in the case of Assignment faults (e.g., a missing variable initialization), about 50% of faults affect at most 10 bytes, as in these cases the incorrect assignment affects individual fields of data structures returned to the main program. Nevertheless, a significant share of Assignment faults still corrupt large memory areas.

The analysis of return values pointed out that several types of values can be returned by library invocations affected by software faults. We consider the return value of a faulty invocation as *incorrect* when it differs from the return value of the same invocation in the fault-free trace. Table 4 shows the distribution of incorrect return values, by classifying them into -1, 0 non-pointer data, *NULL* pointers, and *wrong pointers/values*, i.e., return values different from fault-free executions that do not fall into any of the other classes. Moreover, we further distinguish between the case in which both wrong return values and memory corruptions occur after the same invocation, and the case in which a wrong value is returned without the corruption of memory areas. The distributions of return values depend on the data type returned by library functions, which vary across different targets. Most importantly, we found that in most cases (75.2%) wrong return values are accompanied by memory corruptions. For instance, in the case of a library function that reads data from the disk and that behaves erroneously, both data returned through an input/output parameter (e.g., containing disk data) and the return value (e.g., representing the number of bytes read) become incorrect during the same invocation. This finding has significant implications for the injection of representative interface errors: existing tools that inject faults at library interfaces, such as FTS and LFI [15, 27], focus on the injection of wrong return values, but neglect the injection of memory corruptions. To achieve representativeness, wrong return values should be injected along with memory corruptions.

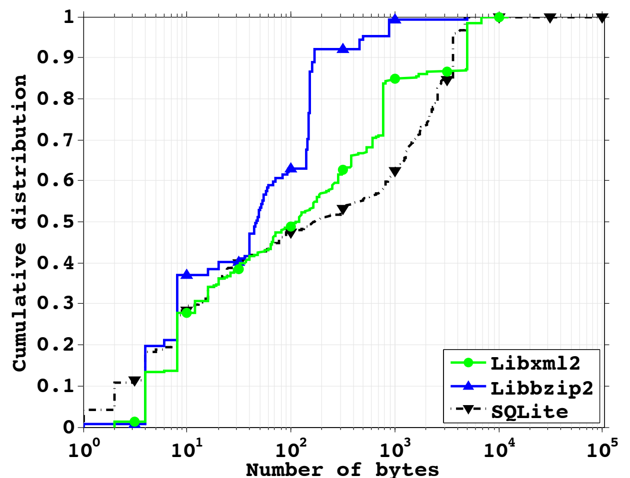

Figure 11: Cumulative distribution (per library) of the number of corrupted bytes of interface data.

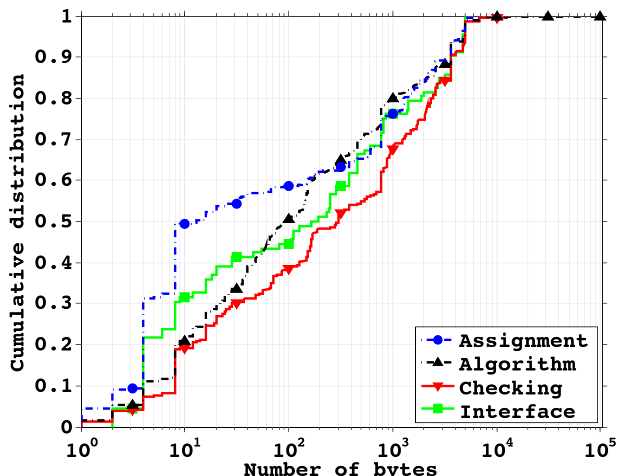

Figure 12: Cumulative distribution (per fault type) of the number of corrupted bytes of interface data.

Furthermore, by comparing the number of failures with error codes (Table 4) and the total number of experiments that lead to failures (Table 3), we found that wrong return values only occur for a fraction of cases (40.9% for *Libxml2*, 75.6% for *Libbzip2*, 22.5% for *SQLite*). This indicates that "plausibility" checks that operate solely on return values are insufficient for detecting library failures, as several failures occur despite correct return values from library functions.

As pointed out in the previous analysis of memory corruptions, the amount and location of corrupted data varies with the target library, and in particular with the type of

Table 4: Distributions of return values in fault injection experiments.

Target	Return value, without memory corruption					Return value, with memory corruption				
	−1	*NULL ptr*	0	*Wrong ptr*	*Wrong value*	−1	*NULL ptr*	0	*Wrong ptr*	*Wrong value*
Libxml2	5 (3.8%)	0 (0%)	4 (3.0%)	0 (0%)	50 (37.9%)	32 (24.2%)	33 (25.0%)	1 (0.8%)	5 (3.8%)	2 (1.5%)
Libbzip2	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	4 (11.8%)	0 (0%)	28 (82.3%)	2 (5.9%)
SQLite	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	2 (2.8%)	0 (0%)	1 (1.4%)	69 (95.8%)

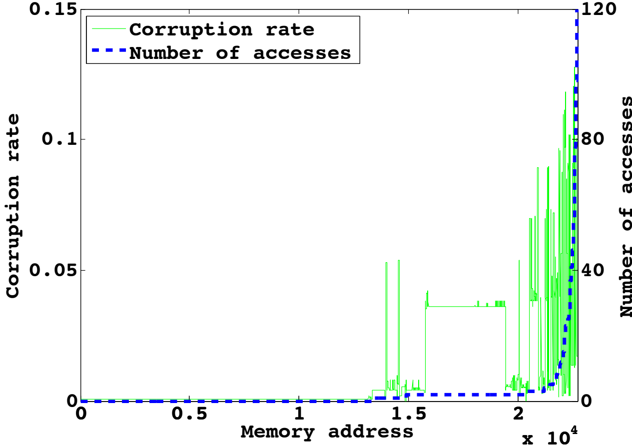

Figure 13: Byte corruption rate and number of accesses for *Libxml2*.

Table 5: Correlation between corruption rate and number of accesses.

Target	Spearman's ρ	p-value
Libxml2	0.953	~ 0
Libbzip2	0.572	~ 0
SQLite	0.883	~ 0

interface parameters and of library functions. To support the tuning of representative IEI experiments for a specific target library, we investigated a heuristic rule for selecting which memory areas of interface parameter data to corrupt. In particular, the heuristic should allow to identify which memory addresses have the highest *corruption rate*, that is, memory addresses most likely corrupted by software faults, in order to focus error injections on them.

To identify such a heuristic, we analyzed memory accesses of the target libraries during fault-free executions, which can be easily obtained through a DBI analysis. We obtained the corruption rate of each memory address by computing the percentage of our fault injection experiments that led to the corruption of that memory address. We found that *the corruption rate of a memory address is strongly correlated with the number of accesses on that memory address in fault-free executions.* Fig. 13 shows the growth of the corruption rate with the number of accesses in fault-free conditions for *Libxml2*. The quantitative analysis of correlation (Table 5), using the Spearman correlation coefficient for ordinal data (which can be applied even when the association between elements is non-linear) and a statistical hypothesis test (with the null hypothesis that there is a zero correlation) [37], confirm this observation: We found that the corruption rate and the number of accesses have a statistically significant correlation (the null hypothesis can be rejected at any reasonable type I error level, as the *p-value* of the test is lower than the smallest representable float number on our machine). From this result, we conclude that a heuristic rule for obtaining a realistic error model is to corrupt those memory addresses that are most often accessed in fault-free executions.

7. THREATS TO VALIDITY

We identified the following threats to validity: (i) the effects of non-determinism, that can lead to variations of memory *stores* that are not actually due to faults, and thus could mislead our the analysis of memory corruptions; (ii) the use of fault injection in components' code, in place of real software faults, to generate and analyze interface errors; and (iii) the selection of the target libraries.

As for non-determinism, we carefully designed our approach to factor out its effects from execution traces, and validated its ability to avoid non-deterministic interferences by verifying that execution traces are exactly reproducible when no fault is injected. The limitation of our analysis is that, in this initial phase of our research, we chose to first focus on single-threaded executions, before extending our approach to multi-threaded executions using recording-and-replay techniques as discussed in Section 4.

The use of real faults for obtaining real interface errors is unfortunately hampered by the shortage of faults to analyze for specific library versions and configurations, especially in the case of very mature and highly reliable software such as SQLite. We therefore adopted fault injection, which allows to perform a high number of experiments and, at the same time, is able to generate representative errors, as demonstrated by several empirical studies on the use of code mutation for software engineering experimentation [1,11,12]. We are thus confident that the validity of our findings is not significantly affected by the use of fault injection.

Our conclusions are based on experimental results from three libraries, and may not generalize to all types of libraries. As the chosen libraries are widely used and functionally diverse, we believe that the results are representative for a larger set of libraries. Apart from whether a generalization is valid, we demonstrate that libraries with the discussed error manifestations exist and that existing error models do not match these manifestations. Moreover, we provide an approach that is suitable to assess any library of interest.

8. CONCLUSION

In this paper, we proposed an approach for analyzing how software faults in library code manifest as interface errors. We analyzed interface errors in three real-world libraries, obtaining guidelines for representative error injection experiments. In future work, we aim at applying these findings in IEI experiments, and investigating whether they can improve the representativeness of experiments.

Acknowledgments: Work supported by MIUR projects *SVEVIA* (PON02_00485_3487758) and *TENACE* (PRIN n.20103P34XC), and by TU-Darmstadt's projects BMBF EC-SPRIDE and LOEWE-CASED.

9. REFERENCES

[1] J. Andrews, L. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *ICSE*, pages 402–411, 2005.

[2] J. Arlat, J. Fabre, M. Rodríguez, and F. Salles. Dependability of COTS Microkernel-Based Systems. *IEEE Trans. Comput.*, 51(2):138–163, 2002.

[3] A. Avizienis, J. Laprie, B. Randell, and C. Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Trans. Dependable Secure Comput.*, 1(1):11–33, 2004.

[4] J. Barton, E. Czeck, Z. Segall, and D. Siewiorek. Fault Injection Experiments using FIAT. *IEEE Trans. Comput.*, 39(4):575–582, 1990.

[5] S. Chandra and P. M. Chen. How fail-stop are faulty programs? In *FTCS*, pages 240–249, 1998.

[6] 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.*, 18(11):943–956, 1992.

[7] J. Christmansson and R. Chillarege. Generation of an Error Set that Emulates Software Faults based on Field Data. In *FTCS*, pages 304–313, 1996.

[8] J. Christmansson, M. Hiller, and M. Rimen. An Experimental Comparison of Fault and Error Injection. In *ISSRE*, pages 369–378, 1998.

[9] D. Cotroneo, A. Lanzaro, R. Natella, and R. Barbosa. Experimental Analysis of Binary-Level Software Fault Injection in Complex Software. In *EDCC*, pages 162–172, 2012.

[10] D. Cotroneo and R. Natella. Fault Injection for Software Certification. *IEEE Security & Privacy*, 11(4):38–45, 2013.

[11] M. Daran and P. Thévenod-Fosse. Software Error Analysis: A Real Case Study Involving Real Faults and Mutations. *ACM Soft. Eng. Notes*, 21(3):158–171, 1996.

[12] H. Do and G. Rothermel. On the use of mutation faults in empirical assessments of test case prioritization techniques. *IEEE Trans. Softw. Eng.*, pages 733–752, 2006.

[13] J. Durães and H. Madeira. Emulation of Software faults: A Field Data Study and a Practical Approach. *IEEE Trans. Softw. Eng.*, 32(11):849–867, 2006.

[14] 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.*, pages 74–89. 2007.

[15] A. K. Ghosh and M. Schmid. An approach to testing COTS software for robustness to operating system exceptions and errors. In *ISSRE*, pages 166–174, 1999.

[16] M. Hiller, A. Jhumka, and N. Suri. EPIC: Profiling the propagation and effect of data errors in software. *IEEE Trans. Comput.*, 53(5):512–530, 2004.

[17] Hipp, Wyrick & Company, Inc. Well-Known Users of SQLite, 2013. `http://www.sqlite.org/famous.html`.

[18] J. W. Hunt and T. G. Szymanski. A fast algorithm for computing longest common subsequences. *Comm. ACM*, 20(5):350–353, 1977.

[19] 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 *PRDC*, pages 51–58, 2002.

[20] Y. Jia and M. Harman. An Analysis and Survey of the Development of Mutation Testing. *IEEE Trans. Softw. Eng.*, 37(5):649–678, 2011.

[21] G. Kanawati, N. Kanawati, and J. Abraham. FERRARI: A Flexible Software-Based Fault and Error Injection System. *IEEE Trans. Comput.*, 44(2):248–260, 1995.

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

[23] 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.*, 19(11):1105–1118, 1993.

[24] P. Koopman and J. DeVale. The Exception Handling Effectiveness of POSIX Operating Systems. *IEEE Trans. Softw. Eng.*, 26(9):837–848, 2000.

[25] N. Laranjeiro, M. Vieira, and H. Madeira. A Technique for Deploying Robust Web Services. *IEEE Trans. Services Comput.*, 7(1):68–81, 2014.

[26] H. Madeira, D. Costa, and M. Vieira. On the Emulation of Software Faults by Software Fault Injection. In *DSN*, pages 417–426, 2000.

[27] P. Marinescu and G. Candea. Efficient testing of recovery code using fault injection. *ACM Trans. Computer Sys.*, 29(4):11:1–11:38, 2011.

[28] 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 *EDCC*, pages 53–64, 2006.

[29] R. Moraes, J. Durães, R. Barbosa, E. Martins, and H. Madeira. Experimental Risk Assessment and Comparison using Software Fault Injection. In *DSN*, pages 512–521, 2007.

[30] R. Natella, D. Cotroneo, J. A. Durães, and H. S. Madeira. On Fault Representativeness of Software Fault Injection. *IEEE Trans. Softw. Eng.*, 39(1):80–96, 2013.

[31] N. Nethercote and J. Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. *ACM Sigplan Not.*, 42(6):89–100, 2007.

[32] W. Ng and P. Chen. The Systematic Improvement of Fault Tolerance in the Rio File Cache. In *FTCS*, pages 76–83, 1999.

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

[34] ObjectWeb Consortium. CARDAMOM - An Enterprise Middleware for Building Mission and Safety Critical Applications, 2013. `http://cardamom.ow2.org`.

[35] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie. PinPlay: a framework for deterministic replay and reproducible analysis of parallel programs. In *CGO*, pages 2–11, 2010.

[36] J. Seward. bzip2 Home Page, 2013. `http://www.bzip.org`.

[37] D. J. Sheskin. *Handbook of parametric and nonparametric statistical procedures.* Chapman and Hall/CRC, 2003.

[38] S. Tallam, C. Tian, R. Gupta, and X. Zhang. Enabling Tracing of Long-Running Multithreaded Programs Via Dynamic Execution Reduction. In *ISSTA*, pages 207–218, 2007.

[39] D. Tang and H. Hecht. An Approach to Measuring and Assessing Dependability for Critical Software Systems. In *ISSRE*, pages 192–202, 1997.

[40] M. Vieira and H. Madeira. A dependability benchmark for OLTP application environments. In *VLDB*, pages 742–753, 2003.

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

[42] J. Voas, F. Charron, G. McGraw, K. Miller, and M. Friedman. Predicting How Badly “Good” Software Can Behave. *IEEE Softw.*, 14(4):73–83, 1997.

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

[44] S. Winter, C. Sârbu, N. Suri, and B. Murphy. The impact of fault models on software robustness evaluations. In *ICSE*, pages 51–60, 2011.