Software Quality Journal manuscript No.
(will be inserted by the editor)

Finding fault with fault injection

An Empirical Exploration of Distortion in Fault Injection Experiments

Erik van der Kouwe · Cristiano Giuffrida · Andrew S. Tanenbaum

Received: date / Accepted: date

Abstract It has become well-established that software will never become bug-free, which has spurred research in mechanisms to contain faults and recover from them. Since such mechanisms deal with faults, fault injection is necessary to evaluate their effectiveness. However, little thought has been put into the question whether fault injection experiments faithfully represent the fault model designed by the user. Correspondence with the fault model is crucial to be able to draw strong and general conclusions from experimental results. The aim of this paper is twofold: to make a case for carefully evaluating whether activated faults match the fault model and to gain a better understanding of which parameters affect the deviation of the activated faults from the fault model. To achieve the latter, we instrumented a number of programs with our LLVM-based fault injection framework. We investigated the biases introduced by limited coverage, parts of the program executed more often than others and the nature of the workload. We evaluated the key factors that cause activated

This research was supported in part by European Research Council grant 227874

Erik van der Kouwe
Computer Science Department
Faculty of Sciences, VU University
Amsterdam, The Netherlands
E-mail: erik@minix3.org

Cristiano Giuffrida
Computer Science Department
Faculty of Sciences, VU University
Amsterdam, The Netherlands
E-mail: giuffrida@cs.vu.nl

Andrew S. Tanenbaum
Computer Science Department
Faculty of Sciences, VU University
Amsterdam, The Netherlands
E-mail: ast@computer.org

faults to deviate from the model and from these results provide recommendations on how to reduce such deviations.

1 Introduction

Despite decades of advances in software engineering and program verification tools, many software systems are still plagued by critical software bugs. Several studies have shown that the number of bugs is roughly linear with the program size [38] even in mature software. Formal methods proposed to address such bugs, such as used by seL4 [27], require a heroic effort. seL4's correctness proof alone, for example, required around 20 person years for 9,300 lines of code. To scale to software that is hundreds to thousands of times larger would not currently be realistic. Furthermore, formal specifications can contain bugs or in turn rely on the correctness of other components (i.e., compilers, hardware, documentation, etc.). As a result, fault containment and recovery mechanisms still play a pivotal role in the design of highly reliable systems.

To validate such mechanisms, it is often necessary to evaluate the behavior of a system under faults. Identifying a sufficiently large number of real software faults is normally not an option because it requires considerable manual work. Therefore, fault injection techniques have been devised to artificially inject faults and compare the run time behavior of the system during fault-free and faulty execution.

Several fault injection tools are described in the literature, with injection strategies emulating simple hardware faults (e.g., bit flips or intermittent errors) [4, 24], faults at the component interfaces (e.g., unexpected error conditions generated by the libraries) [28, 33, 34], or real-world software faults introduced by programmers [11, 37, 44]. Each injection strategy reflects a particular fault scenario and serves a unique purpose in the reliability testing process.

Although the principles outlined here are more general, our focus is specifically on injection of realistic software bugs. Such injections are particularly critical to evaluate the effectiveness of fault containment mechanisms (i.e., preventing faults in one component from affecting other components), fault detection techniques (i.e., identifying the occurrence of faults during execution), and fault recovery mechanisms (i.e., mitigating service disruption after the occurrence of faults).

To rigorously conduct fault injection experiments, an important step is to define an appropriate fault model. The fault model specifies what kinds of faults should be tested. This model includes at least the types of faults to be injected and the locations selected for injection, but possibly also other factors such as fault triggers [33, 34]. Prior work has investigated how to accurately construct a representative fault model, for example by considering which fault types occur in which frequencies in real software [11] and at which locations faults are most likely to occur in production [35, 36].

Nevertheless, defining a representative fault model and configuring a fault injection tool to follow that model is not sufficient to thoroughly assess the quality of fault injection results. To show why, we must first understand how the fault model is instantiated by the fault injection tool into an input and output fault load. The input fault load consists of the faults that the tool inserts into the code of the program. Generally, an effort is made to configure the fault injection tool such that the input fault load carefully reflects the original fault model. Suppose, for example, that the fault model specifies that 10% of faults should be branch condition flips in a location that is executed by no more than two different tests. In this case, in an input fault load consisting of 100 faults one would expect approximately 10 such faults.

The output fault load consists of the subset of faults activated during the experiment, accounting for multiple activations. Multiple activations are important because some faults only have an impact in particular circumstances, such as a memory leak only affecting the results when already low on memory. Some faults may be activated only once, others very often, yet others may not be activated at all.

It should not be assumed that all faults in the output fault load cause actual failures, as it is possible for an activated fault not to affect any relevant state. For example, a buffer overflow might corrupt only data that is not read before being overwritten again or a memory leak might not be severe enough to cause later allocations to fail. Whether faults cause failures is important, but strongly depends on what types of failures one is interested in. For example, one might consider only crashes or one might go as far as to consider even spelling errors in displayed messages or differences in timing. In this paper we only look at distortion introduced by nonactivation and multiple activations, which is an important factor regardless of the exact types of failures being considered.

The output fault load may differ considerably from the input fault load. Even if the fault model is representative of real-world faults and the input fault load accurately instantiates the original fault model, it is possible for the output fault load to not represent a realistic fault model at all. We will refer to the difference between input and output fault load as *distortion*. If the distortion is biased towards particular fault types or locations, activated faults do not faithfully reflect the original fault model even if many experiments are carefully run. We will use the term *fidelity* to refer to the degree to which the output fault load reflects the original fault model with no distortion. The introduction of the new terminology is justified by our focus on the quality of the output fault load generated by the fault injection tool. This is in stark contrast with prior approaches described in the literature, which are solely focused on representativeness and accuracy of the input fault load [11, 35, 36]. We will provide a formal definition of fidelity and the other relevant terms in Section 3.

To demonstrate the concepts we introduced and indicate why fidelity is important, we have included a small code example in Fig. 1. The first step is to identify fault candidates. In working out the example we will use the same

```
1  char *clear(char *buf, size_t size) {
2    if (!buf) {
3      buf = malloc(size);
4    }
5    while (size > 0) {
6      buf[--size] = 0;
7    }
8    return buf;
9  }
```

Fig. 1 Example function to demonstrate distortion

fault types used in our experiment. These are listed in Table 1. For illustration, we will point out the fault candidates on the first line. Both the 'flip-branch' and the 'stuck-at-branch' fault types can be injected in the `if` statement. If the former is injected, the branch is taken when it should not be taken and vice versa, a common programmer mistake. If the latter is injected, the branch is either always or never taken. This corresponds with omission of either the entire `if` statement or just the part that makes it conditional. As this example shows, it is possible for multiple fault types to be applicable to a single code location. Both of these are considered individual fault candidates, although only one of them can be injected at a time. The controlling expression of the `if` statement is also subject to several fault types. For example, a programmer could accidentally invert the boolean operation ('flip-bool'), use the wrong unary operator ('corrupt-operator') or use the wrong variable ('no-load' and 'random-load'). In total, there are as many as 46 fault candidates of 9 different fault types in this small code snippet.

The control flow of the code example can serve as an example for two elements of distortion that we will consider. First, the memory allocation on line 2 (the only fault candidate of the 'dangling-pointer' in the example) is executed only in case `buf` equals `NULL`. Depending on the context in which this function is called that might happen on every run, it might never happen or it may depend on the workload that is used to test the program. In the last case, the distribution of fault types of accessible fault candidates is different between workloads that do reach this statement and workloads that do not. Now let us assume that fault model specifies that a certain fraction of the faults tested must be of the 'dangling-pointer' type and this fault candidate was selected for injection of such a fault. This means that the fault candidate is part of the input fault load. If, however, the workload never activates this part of the code, it is not part of the output fault load. This does not necessarily mean that the output fault load is skewed with regard to the fault model and the input fault load. This depends on the question whether specific fault types or locations are significantly more or less likely to be activated by the workload when considering the program as a whole. Without testing, it is hard

Table 1 Fault types

name	description	applicability
buffer-overflow	size too large in memory operation	call to `memcpy`, `memmove`, `memset`, `strcpy` or `strncpy`
corrupt-index	off-by-one error in array index	array element access
corrupt-integer	off-by-one error in integer operand	operation with integer arguments
corrupt-operator	replace binary operator with random operator	binary operator
corrupt-pointer	replace pointer operand with random value	pointer operation
dangling-pointer	size too small in memory allocation	call to `malloc`
flip-bool	negate result of boolean operation	boolean operation
flip-branch	negate controlling value for conditional branch	conditional branch
mem-leak	remove memory de-allocation	call to `free` or `munmap`
no-load	load zero instead of intended value	memory load
no-store	remove store operation	memory store
random-load	load random number instead of intended value	memory load
stuck-at-branch	fixed controlling value for conditional branch	conditional branch
stuck-at-loop	fixed controlling value for loop	conditional branch part of loop construct
swap	swap operands of binary operation	binary operator

to tell whether the numbers will average out at the larger scale or there is a systematic bias.

Another issue that can be illustrated with this code example is the impact of execution count. The assignment on line 6 is executed on every iteration of the `while` loop. There are various faults that could be injected here. One example target for fault injection is the unary pre-decrement operator, which provides a fault candidate for the 'corrupt-operator' fault type. If the operator is mistakenly changed by the programmer into a post-decrement operator, the result is an off-by-one error that overwrites a single byte past the end of the buffer. If, on the other hand, it is changed to the unary minus operator, it overwrites `size` bytes before the start of the buffer. As a result, the extent of the damage it can do depends on the number of loop iterations. Since the number of loop iterations might depend on the workload (if, for example, it is the size of a user-provided file), the potential for the fault to do damage also depends on the workload. In this case it has more potential to do damage than other instances of the same fault type that are executed only once. Again, we cannot a priori make any assumptions on the question whether this averages out or introduces bias that gives some fault type more potential of doing harm because they are in loops more often. Therefore it is important to consider the issue of multiple activation with an empirical test.

These examples demonstrate that it is important to determine whether these factors have an impact in practice. If they do, the output fault load cannot be assumed to always accurately instantiate the fault model. This then

needs to be taken into account when designing fault injection experiments and workloads to compensate for the biases identified. Our experiment is designed to determine whether this is indeed necessary.

These considerations yield the following research question: "*When performing fault injection experiments, how faithful is the output fault load observed with respect to the specified fault model and which factors affect its fidelity?*" This question is important for a number of reasons. First, if there is substantial distortion, the experiment is no longer consistent with what the user intended to measure. Suppose, for example, that one wants to measure the probability of a recovery solution being able to successfully recover state. If the output fault load is biased towards bugs that are easier to recover from, the solution appears to be more effective than it would be in reality. Second, fault injection experiments can be performed more efficiently if they follow the fault model. As the output fault load differs more from the fault model, more injections are needed to achieve the same rigor with regard to testing those faults specified by the model. Third, it is harder to compare experiments when there is distortion. If two experiments inject the same number of faults but it is not known how faithful they are to the fault model, it is possible that they differ greatly in their effectiveness in finding faults even though they both inject the same number of faults. Fourth, as fidelity is coupled with the behavior of the test workload, a high level of distortion may indicate that the workload is not properly designed for the experiment and may have to be reconsidered. These issues show that it is crucial to consider the distortion between input and output fault loads and identify the originating factors.

The main contributions of this paper are (1) providing a definition of fault injection fidelity and showing its relevance in fault injection campaigns, (2) performing the first large-scale evaluation of fidelity on a number of programs and workloads to evaluate the impact of distortion problems in real-world fault injection experiments, and (3) analyzing the key factors that can help predict and control distortion problems in fault injection experiments. Please note that this is an extended version of the earlier conference paper [29], which shares these contributions. Amongst others, this version has a more complete empirical evaluation and discusses the methodology and its limitations in more depth.

After this introduction, we continue with a description of relevant related work. Then, we define the term 'fidelity' and introduce a number of factors that influence the fidelity of a fault injection experiment. In the approach section we elaborate on the experiments we performed to determine whether distortion is an important factor to consider. In the next section, we describe the programs and workloads used in these experiments and explain how we have selected and constructed them. In the results section, we present the outcomes of our experiments. Since we investigate a number of different factors that play a role in distortion, it is split in several subsections, each dealing with one of these factors. The subsections provide the relevant results and an analysis of their impact, as well as a consideration of the factors that may threaten the validity

and generalizability of these results. Finally, the conclusion summarizes our main findings.

2 Related work

Fault injection is a popular technique to evaluate the impact of unforeseen faults on a running software system. When compared to alternative strategies that aim to uncover real software bugs (e.g., symbolic execution [5]), fault injection is relatively inexpensive, scales efficiently to large and complex programs, and allows users to emulate special conditions not necessarily present in the original program code. Fault injection is used to benchmark the dependability of several classes of software, such as: device drivers [17,41,44], file caches [37], operating systems [14,28], user programs [3,33,34], and distributed systems [15, 23]. Typical evaluation scenarios entail analyzing the behavior of a system under faults [14, 28], conducting high-coverage testing experiments for existing error recovery code paths [3,15,23], or evaluating the effectiveness and containment properties of fault-tolerance techniques [17, 41, 44].

While fault injection can theoretically be used to explore all the possible combinations of faults in a given piece of software, in practice this strategy is computationally infeasible for any nontrivial program. To address this problem, prior work has proposed either using efficient fault space exploration strategies [3, 15, 23] or relying on a well-defined fault model tailored to the particular fault scenario of interest.

Several possible fault models are described in the literature, with fault injection strategies emulating (i) hardware faults in hardware [2, 16, 26, 32] or software [4,24], (ii) software faults [11,37,44], (iii) interface faults at the library level [33, 34] or (iv) at the system call level [28]. Injection techniques range from static program mutations—using simulation-based strategies [7, 20, 40], compiler-based strategies [18,44], or binary rewriting [4,11,24,37]—to run time strategies that periodically interrupt the execution—using timers [6, 24, 43] or predetermined hardware or software traps [6, 24, 25, 43].

When selecting a fault model, an important question prior work has sought to address is whether the model is *representative* for the fault scenario of interest. Representativity is important for the validity and comparability of the final results. In particular, much research on fault model representativeness is devoted to emulating realistic software faults found in the field. In this context, a number of studies consider the problem of how accurately artificially injected fault types represent real-world fault types introduced by programmers [11,12, 37,44]. The G-SWFIT tool [11], for instance, injects fault types based on real-world bugs found in existing software. Other studies focus on the accuracy of the different injection strategies. For example, Cotroneo et al. [9] consider the accuracy problems of binary-level injection strategies when compared to source-level program mutations. Christmansson et al. [19] compare location-based injection strategies with timer-based approaches. Madeira et al. [31] investigate general limitations of traditional fault injection strategies when

compared to real faults found in the field. In another direction, Natella et al. [35, 36] consider the problem of fault location representativeness, arguing that so-called residual faults are most representative of real-world bugs that escape software testing and can be found in production systems in the field.

Unlike fault model representativeness, research on fidelity of fault injection to the original fault model has received much less attention in the literature. A number of prior approaches have considered the impact of code coverage on fault injection experiments [21, 22, 42], but their focus is limited to ensuring reasonable fault activation. Unfortunately, fault activation itself is a poor metric to evaluate how the nature of the program or workload can degrade the quality of the final fault injection results. Our notion of fault injection *fidelity*, in contrast, is much more rigorous and able to capture the full dynamics of both the test program and the workload. Our investigation, in particular, provides a thorough analysis of the impact of code coverage on fault injection experiments, while determining how low coverage distorts the original fault model. This analysis is particularly crucial to quantify the validity and comparability of fault injection results.

There is some literature on the estimation of coverage of the fault space in the context of hardware fault injection, such as for example [10]. This work has similar goals as ours—determining how thorough fault injection experiments are—but the different context means that they focus on different issues than the ones considered here, which are specific to software fault injection.

3 Fidelity

The first step in our research is to formally define the concept of fidelity. *Fidelity* is defined as the extent to which the output fault load reflects the original fault model. Here, the *fault model* is a model defined in advance of the experiment that specifies which faults would be expected in a realistic setting. It specifies fault types and their relative frequencies as well as the relative frequency of faults occurring per code location. The *output fault load* is defined as the set of faults actually activated, considering multiple activation. It is important to stress that this research does not consider whether the faults actually have an impact, which would require a different methodology involving the injection of actual faults. Investigating the impact of this effect is left for future research.

Although not directly referenced in the definition of fidelity, the input fault load is also important in the analysis of fidelity. We define the *input fault load* as the set of faults actually injected in the target program. One of the main goals of this paper is to make clear that there can be meaningful differences between the input and output fault loads. We refer to this difference as *distortion*. To the extent that distortion introduces bias in the faults actually activated, it threatens the fidelity of the experiment if nothing is done to compensate for it.

It should be noted that our current definition of fidelity is qualitative, not quantitative. This paper aims to identify the issue and empirically show that distortion occurs in commonly used fault injection settings. Our investigation of circumstances that may influence whether distortion is an issue should be considered preliminary. The definition of a quantitative metric to measure fidelity and using that definition to reach stronger conclusions on the circumstances causing distortion is out of scope for this paper and is left for future work.

To research fault injection fidelity, we investigate how the input fault load (faults injected in the program) relates to the output fault load (faults actually executed). The factors that influence the transformation from input fault load to output fault load make up the dependent variable of our research. Independent variables we investigate include program types, program implementations, workloads, and compiler settings (in particular, optimization level). Although more factors could influence fidelity, we selected those that are intuitively important and easily controlled by the researcher. To find the impact of the program and the workload independently, we include some programs with multiple workload generators as well as workloads that can be used across multiple programs.

The first factor we consider is *coverage*, defined as the fraction of the program that gets executed when the test workloads are run. It can be measured in several units, commonly lines of code, but alternatively in terms of machine instructions or basic blocks (a basic block being a part of the code which has a single entry point and a single exit point). In the context of fault injection, an alternative is to consider which fraction of the *fault candidates*—that is, program locations that are suitable to inject a fault of a particular type—is covered. This way of measuring coverage has a clear relationship with distortion because the more fault candidates are left out, the more difference is to be expected between the input and output fault loads (both of which are also expressed in terms of fault candidates). Unfortunately, coverage is rarely reported when performing fault injection experiments in research papers–with some notable exceptions [33–35, 42]. In general, higher coverage is better as it allows a larger part of the program to be tested. We address a new concern, namely whether lack of coverage introduces bias that threatens the fidelity of the experiment. Uncovered locations are not a random subset of all locations but rather those that are hard to reach, like for example code that deals with error conditions. Not just fault locations, but also fault types may be biased as uncovered code often performs a different role than covered code.

The second factor is the distribution of the execution count per basic block. It is expected that most of the run time of a program is spent executing only a small part of the code. Faults injected in this part of the code get activated over and over, whereas some other fault locations are activated only once per run. Execution count is relevant in cases where the impact of the fault depends on the context. For example, it might corrupt the state only in some particular context, it might affect different parts of the state on each activation, or prior deviations in the local context may affect the global state only after a subse-

quent activation. A typical example is a memory leak, which does not have a visible impact on the initial execution. However, as it is executed over and over again it might eventually deplete available memory completely, resulting in a crash. Our question is to what extent differences in execution count introduce bias, affecting the fidelity of the experiment. Code executed multiple times is not a random subset of the program. Most likely, it is the functional core of the program, which has been tested extensively. In particular, it seems likely that when injecting residual faults [35] the locations less likely to be triggered are also triggered less often. Assuming that activated faults may or may not propagate depending on the context, faults activated more often have a higher chance of causing anomalous behavior in excess of the impact of being activated by more of the workloads. This introduces distortion with regard to the intended fault model.

It cannot be assumed that coverage and execution count of a basic block are independent from the number and types of fault candidates present in the basic block. Fault candidate types occurring more commonly in blocks likely to be executed are another source of bias. In the case that some fault types are over- or underrepresented in the part of the code covered by the workloads, it is still possible to make the output fault load faithfully reflect the fault model. However, the effect must be measured to allow the input fault load to be altered to compensate for the bias introduced. A second issue is whether the execution count is the same between fault types. Again, this is expected not to be the case. Backward branches, for example, are almost always part of a loop hence more likely to get executed often than other types of instructions. A fault type that is specific to branches is likely to have candidates in such places, again introducing bias. In this case it is harder to compensate because the impact of multiple activations depends not just on the fault type and location, but also on the context. Still, it is important to know that multiple activations introduce distortion into the experiment.

4 Approach

We aim to find and explain differences between the input and output fault loads. To gather information on the behavior of the test program, we use compiler-based instrumentation implemented using the LLVM (Low-Level Virtual Machine) compiler framework [30] (version 3.2). LLVM is a modular compiler that can write object files in its intermediate format (also referred to as LLVM bitcode) and allows for linking at that level. It provides an API that can be used to implement new compiler passes directly operating at the bitcode level. Such passes are independent of both the front-end and the back-end of the compiler and hence portable between all supported programming languages and target architectures. Because linking can be performed at the bitcode level, compiler passes can also get an overall view of the program while still having access to the extensive source-level information present in the bitcode format. Our analysis operates at the LLVM bitcode level because the

availability of source-level information (in contrast to approaches using the binary where this information is lost) is required for fault type representativeness [9, 13].

For our investigation, we chose the standard software fault types commonly used in the literature [8, 11, 39]. The selected fault types are listed in Table 1. While compiling each program, we identify all fault candidates and register in which basic block they occur. It is convenient to do this at the basic block level, because normally a basic block is either executed in its entirety or not at all. The exception here are signals and exceptions, which should be uncommon enough not to interfere with the research as long as only a single fault is injected on each run. Without injecting any actual faults, we apply our instrumentation to measure execution counts for each basic block while running one or more workload generator scripts for each test program. This allows us to efficiently compute the output fault load for any input fault load. The main disadvantage is that it is not possible to consider interactions between faults. However, it should be noted that it is not possible to draw general conclusions about the impact of interactions between faults regardless because the interactions depend not just on the fault types and locations but also on the context. Interactions may introduce additional distortions, such as faults activated early being more likely to occur than faults activated late. However, these distortions are mostly a problem if many faults are injected per run, which is quite unrealistic to begin with. Our approach allows us to use a few real runs (multiple to capture random workload variations) for each program/workload generator and use the statistics collected to efficiently consider all possible single injections. Interactions would however be a good topic for future research.

Although we have conducted all our experiments on x86 Linux, testing programs written in the C programming language, our approach is more generally applicable. Our tools are built on top of LLVM and make no assumptions about the programming language or back-end used. This means that they should work with any programming language for which an LLVM front-end is available, on any operating system that can be targeted by LLVM and on any CPU architecture for which an LLVM back-end is available. In other cases, our tools cannot be used but it is still possible to apply our approach. To perform an analysis like ours, the main requirements are a profiler that provides information at the basic block level and a fault injector that can identify a relevant set of fault candidates in the programs. Depending on the programming language, it may be necessary to select a different set of fault types. For example, the memory leak fault type included in our tests would not apply for garbage-collected languages such as Java. For future work, it would be interesting to apply our approach in such different environments to determine whether our findings also hold there.

5 Programs and workloads

We selected a number of programs that is reasonably diverse, while also containing several sets of programs that are functionally similar. The latter can be used to compare different programs running the same workload. We preferentially chose programs that offer their own regression test suite to have a 'neutral' workload, but wrote our own workload generators for programs that do not offer regression tests. We selected three compression programs (`bzip2`, `gzip` and `xz`), two implementations of `sort` (GNU Coreutils and Busybox) and two implementations of `od` (same sources). Busybox is normally compiled into a single binary containing all tools, but we configured it to provide each tool as a separate binary. In addition we selected the `bash` shell because it does a lot of parsing and hence may encounter error conditions in the input, `gnuchess` because the control flow of its artificial intelligence is expected to be relatively complex and the `vim` editor to have an interactive program that has a good regression test suite. Because we also wanted to have a systems-related program, we included `ntfs-3g`, which is a user-space implementation of the NTFS file system.

Having a good workload with high coverage is desirable for fault injection experiments. For this reason, regression tests are generally more suitable than performance benchmarks and we have used these wherever they were available. We have not attempted to increase the coverage in these cases as our aim is not to perform the best fault injection experiments possible, but rather get an impression of the biases present in commonly performed experiments. However, we did randomly select a subset of the tests to ensure some variation. We ran enough runs to prevent this from negatively impacting coverage.

Where regression tests were not available or where we wanted comparability between different programs with the same benchmark, we generated workloads that randomly combine the available commands and options as specified by the documentation and randomly generate input files where needed. Some erroneous inputs are also generated, but no attempt is made to test all anomalous conditions so as to keep the results comparable with other experiments.

For the compression programs `bzip2`, `gzip` and `xz` we used the manual pages to generate a random combination of supported flags on each run. The input file is randomly generated using a Markov chain approach, randomly picking a transition matrix from several types of files, including binaries as well as text in several languages. We test compression, testing, decompression and if supported also listing. After compression, the result is sometimes corrupted by changing a byte, zeroing out an aligned 512-byte block or truncating the file. Files are corrupted to trigger some of the error handling code. These workloads are listed as bzip2-man, gzip-man, and xz-man. For purposes of comparison, a common test script using only features supported by all tools was also made (listed as bzip2-common, gzip-common, and xz-common). For `xz` we also included a variant that uses the LZMA compression method rather than the default (listed as xz-common-lzma). In addition, to allow for comparison with the same program and a different workload generator, we have included

the regression tests included with the **bzip2** and **gzip** programs (listed as bzip2-rtest and gzip-rtest).

For **sort** (from both GNU Coreutils and Busybox) and **od** (from the same sources) a similar approach was used. Here, flags were taken from the POSIX specification rather than the man-pages. Some parts had to be left out because Busybox **od** did not implement them.

bash and **vim** come with extensive regression test suites. To introduce variation, on each run we executed a random subset of these regression tests. Each test has a probability of 0.5 of being selected. Introducing variation is crucial to mimic the behavior of real-world workloads. This strategy is in stark contrast with prior approaches, which often resort to the assumption of deterministic workload behavior [21].

For **gnuchess** and **ntfs-3g**, we made a list of operations and selected a random one on each iteration with random (but generally sensible) parameters. This list consists of commands from the manual for **gnuchess** and operations from the POSIX specification operating on the subtree where the file system was mounted for **ntfs-3g**. For **ntfs-3g**, eight processes simultaneously performed operations to trigger any code dealing with concurrency.

6 Results

The results section has been split in subsections, each representing a factor causing distortion that our experiments can shed some light on. Each subsection provides the relevant results and an analysis of their impact.

6.1 Coverage

Coverage is a major concern for fidelity because parts of the code that are never executed when a test workload is run can never activate any faults. Any fault model in which these particular locations are important requires substantial effort to maximize coverage if any degree of fidelity is to be achieved. For this research, the goal is not to maximize coverage but rather to evaluate a range of coverage levels that might realistically occur in fault injection research.

6.1.1 Number of runs needed to measure coverage

Whenever the workload generator contains a degree of randomness, the coverage increases as more runs are tested. Each random input has some chance to trigger basic blocks that have not been triggered yet in previous runs. However, there are strongly diminishing returns because the higher the level coverage was before, the lower the chance of reaching a block that has not been reached previously. Hence, as more runs are performed the coverage will eventually grow to some maximum coverage for that particular workload generator. Fig. 2 shows the total coverage as a function of the number of times the workload

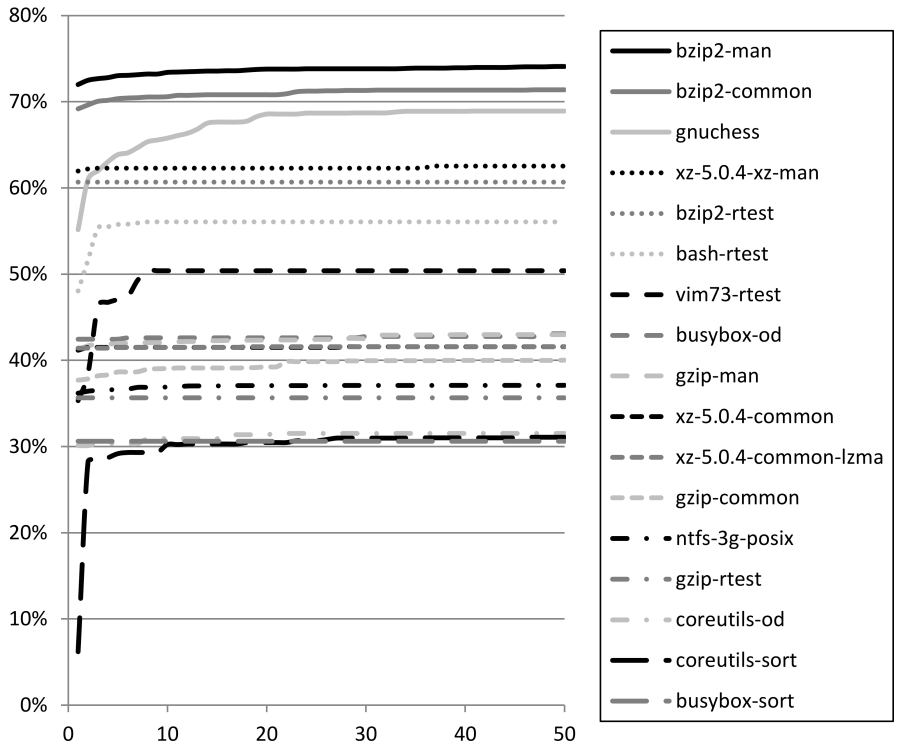

Fig. 2 Coverage in basic blocks as a function of the number of runs with `-O4` optimization

generator is executed with optimization enabled. The number shown for run n is the percentage of basic blocks that has been executed in any of the first n runs of the workload generator script. It should be noted that each invocation of the workload generator consists of multiple executions of the tested program. This number is chosen in such a way that the per-run execution time is comparable between the programs. For example, the compression utilities are used to compress, test and decompress 500 times in each run, while the **`sort`** utility sorts 50 files on each run.

The graph confirms the idea that, while it is important to have a sufficient number of runs, there are strongly diminishing returns. After only 18 runs, all programs and workloads are within 1% of the final coverage at 50 runs, **`gnuchess`** being the last to reach that point. We have performed the same analysis without optimization and the shapes of the graphs are very similar, with 21 runs needed for each workload to be within 1% of the final coverage (again, **`gnuchess`** being the last). The graph suggests that 50 runs is easily enough to get a good impression of the maximum coverage that the the workload generators can achieve for all programs and workloads we test. Given that up to some point increasing the number of runs is effective in increasing coverage, we recommend fault injection experiments to systematically consider this

factor by measuring the number of runs needed before the maximum coverage is reached.

6.1.2 Units for measuring coverage

Coverage is the percentage of a program that is executed while running a workload. There are multiple ways to measure this. A commonly seen metric is the percentage of the lines of code in the program that are executed. This has the advantage of being intuitive because it considers the code written by programmers rather than the (mostly invisible) compiler output. It has the disadvantage that some lines may be much more complicated (and hence prone to bugs) than others. To address this, one could instead consider what percentage of the machine instructions generated by the compiler is executed. This way, complex code has more weight. Another consideration is how easy or hard it would be to increase coverage. If many machine instructions are on the same code path, it would be easy to exercise them by adding an input to the workload that causes this code path to run. If, on the other hand, they are all in conditional branches, many inputs might be needed to reach the same number of instructions. If this is what counts, it is more convenient to measure code coverage in terms of basic blocks, because usually each basic block is either executed or skipped over in its entirety. One final consideration is how many opportunities there are for bugs. Fault injectors identify fault candidates, code locations where the bugs of the types known to the injector can be introduced. This means it may also be a good idea to consider how many of these fault candidates can be activated by the workload as a coverage metric. Given that there are so many ways to measure coverage, it is important to find out how large the impact of the choice of metric is and, if there is a substantial difference, which one is most suitable in the context of fault injection.

Fig. 3 shows the level of coverage we reached for each program using 50 runs using all four metrics of coverage discussed in the previous paragraph. When considering the differences between programs and workloads, it is clear that there is considerable diversity. This is the case even when using the same workload on different programs. For example, bzip2-common and gzip-common are identical workloads used on the different programs, but the former reaches 71.4% of basic block coverage while the latter only activates 40.0% of basic blocks. The regression tests included by the authors of these programs show a similar difference. This suggests that program organization can have a large impact on coverage. We investigated `gzip`'s poor coverage and found that much of the uncovered code is in reimplementations of functions normally imported from the C library such as `printf`. Many features of these functions are never used, resulting in code that the compiler does not know is unreachable. Unreachable code makes it harder to get meaningful information about coverage. Ideally, such code should be removed by the authors or disabled by the researcher before performing any experiments.

Fig. 3 also shows that the way coverage is measured can make a substantial difference. In almost all cases, coverage in terms of fault candidates is highest,

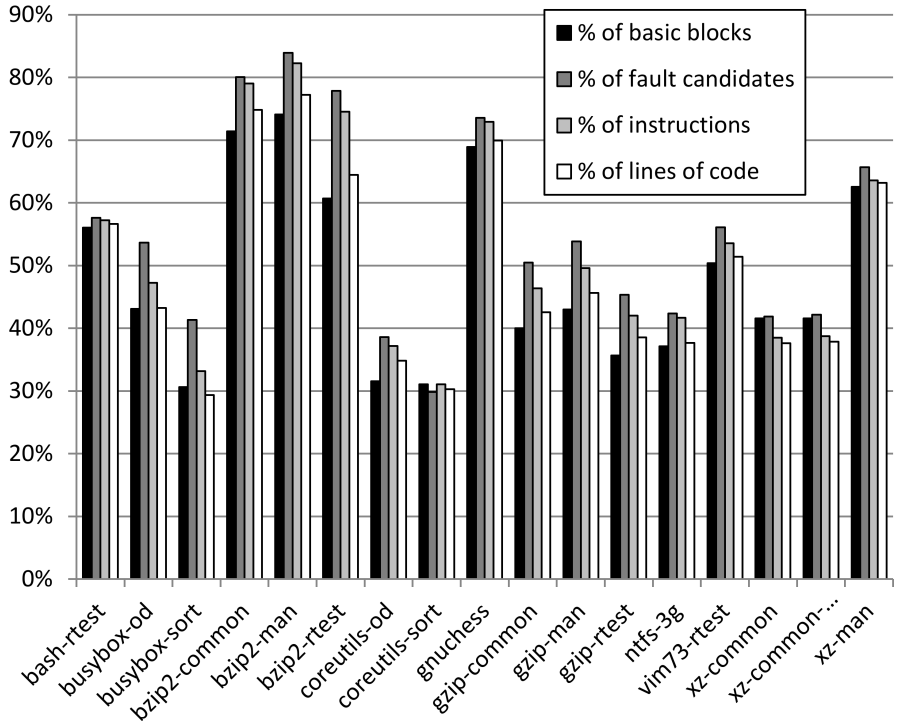

Fig. 3 Coverage per program and workload generator with `-O4` optimization

followed by coverage in terms of instructions and then lines of code. Expressing coverage in terms of basic blocks tends to yield the lowest number. The clearest deviation from this pattern is seen for xz-common and xz-common-lzma, where coverage in terms of basic blocks is relatively high.

We computed correlations between the different metrics and found that correlation was strongest between instructions and lines of code (0.991), instructions and fault candidates (0.990), and basic blocks and lines of code (also 0.990). Correlation is weakest between basic blocks and fault candidates (though still 0.954). All these correlations are highly significant, which is to be expected since they measure the same quantity even though they do so in different ways.

The fact that some coverage metrics are systematically higher than others supports our idea that uncovered code is not representative of all code. Hence, not testing this part of the code introduces a bias that makes the output fault load a more distorted view of the input fault load and hence the fault model. In particular, the fact that coverage in terms of basic blocks is lower than the other measures means that the larger basic blocks (in terms of lines, instructions and fault candidates) are more likely to get executed than smaller basic blocks. We think this may be due to error handling code being tested less thoroughly, with regression tests mostly focusing on the program working

correctly on valid inputs. It seems reasonable to assume that error handling code has smaller basic blocks when a common response to errors is simply to print a message and terminate the program. We will consider this hypothesis in more depth in Section 6.1.3.

Despite the strong correlations, the graph makes clear that there are cases where the choice of metric has a substantial impact on how the coverage numbers compare between programs and workloads. A clear example can be seen with the `sort` utility. Even though the programs implement the same specification, the same workload is used and coverage is very similar in terms of basic blocks (30.6% versus 31.1%), lines of code (29.3% versus 30.3%) and instructions (33.1% versus 31.0%), coverage of fault candidates is much higher for the Busybox implementation (41.3% versus 29.8%). Therefore, fault injection experiments using Busybox are expected to be more faithful to the fault model. A similar situation is found with gzip-common and xz-common, which have similar basic block coverage (40.0% versus 41.6%) but a large difference in fault candidate coverage (50.5% versus 41.9%). A comparison of bzip2-rtest and bzip2-common shows that such situations can exist even with the same programs when running different workloads. Although bzip2-common reaches substantially more basic blocks (71.4% versus 60.7%) and lines of code (74.8% versus 64.5%), its lead in terms of fault candidates is not nearly as large (80.0% versus 77.9%). It has become clear that the choice of metric makes a substantial difference, even when the same program or workload is used. Such differences are visible in cases with very low coverage as well as with reasonably high coverage.

Given that the coverage metrics we discussed show small but meaningful differences, care must be taken to measure coverage in the right way. In case of fault injection experiments we believe coverage in terms of fault candidates to be most appropriate because reaching fewer fault candidates increases distortion. This is especially relevant because our experiments have also shown that covered locations are not representative of all code locations.

In addition to the comparison between programs and workloads, we have investigated the impact of optimization level on coverage by comparing the optimized programs discussed before with unoptimized versions. In LLVM 3.2, `-O4` is the highest level of optimization possible, combining maximal compiler optimization (`-O3`) with link-time optimization (`-flto`), so we are comparing the two extremes. Although optimization allows for the elimination of more dead code, it may also increase code size due to inlining. Therefore, it is a priori unclear whether optimization should have an impact on coverage.

Fig. 4 shows coverage when optimization is disabled. When taking the average over all program/workload combinations, optimization only makes a minor difference. Coverage in terms of fault candidates is most affected, with coverage being slightly higher with optimization (54.9% versus 54.0%). This difference is mostly caused by `gzip`, which is the program for which optimization has most impact. For example, the `gzip-man` test has 53.8% fault candidate coverage with optimization and 49.7% without. To investigate why this program stands out, we listed the coverage per function in the program

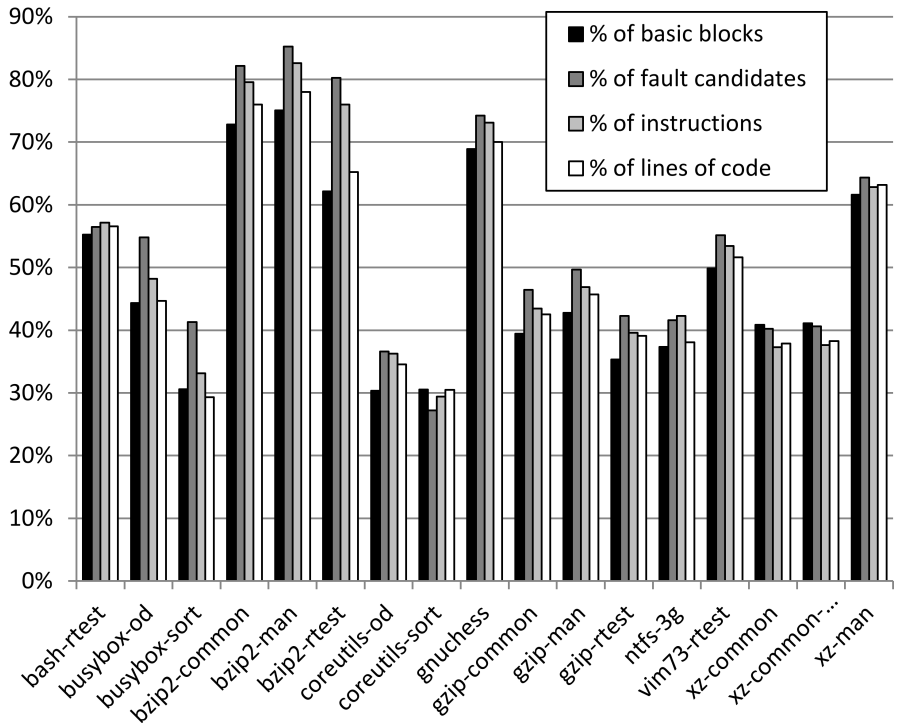

Fig. 4 Coverage per program and workload generator without optimization

to find where this impact of optimization comes from. It turns out that it is mostly due to reimplemented C library functions with poor coverage being optimized more aggressively than the rest of the code.

We have not found evidence that optimization has a meaningful impact on coverage. This does not rule out that it could be a factor in other cases and it is possible that the impact may also depend on which specific optimizations are enabled. Unfortunately we were unable to test levels between unoptimized and `-O4` because of technical limitations of the version of LLVM used for the experiment when bitcode linking is enabled. Based on these results we cannot give a general recommendation on the preferred optimization level. In addition is should be considered that the exact meaning of the optimization switches changes over time as more optimizations are added. Hence, the most prudent approach would be to measure the impact of various levels of optimization in the context where it is being used and base a decision on this. If the coverage differences are minimal as in this case, we would recommend using the same compiler settings as in production settings to bring the fault injection experiment as close as possible to the way the software is used in production environment.

Table 2 Classification of basic blocks in bzip2

Basic block type	% of Total	% of LoC	Lines/bb average	Coverage average	at least once
Normal execution	78.1%	83.0%	1.7	39.7%	83.7%
Data errors	5.6%	4.2%	1.2	3.7%	34.0%
User errors	1.8%	2.6%	2.3	1.8%	15.2%
OS errors	3.2%	2.4%	1.2	0.0%	0.0%
Unreachable	4.2%	4.9%	1.8	0.0%	0.0%
Panic	4.2%	2.9%	1.1	0.0%	0.0%
No line info	2.9%	0.0%	0.0	43.3%	90.8%
Total	100.0%	100.0%	1.6	32.5%	70.1%

6.1.3 Relationship between code location and coverage

The claims that low coverage is caused in part by unreachable code and that error handling code has different characteristics than other code need to be verified. To check whether these are the plausible we analyzed `bzip2` program, classifying each basic block. This program has high coverage compared to the others (74.1% of basic blocks) so it should give a good impression of the nature of the hard-to-reach parts. Also, its control flow is relatively simple, making mistakes less likely. It should not be taken as a representative sample, but rather as a proof of concept that our ideas are plausible.

We classified basic blocks in the `bzip2` program based on the circumstances under which they are invoked. We then used out bzip2-man workload script to invoke the program 600 times and used our instrumentation to keep track of how often each basic block was executed. This allows us to determine coverage for each class of block. The result is shown in Table 2. Basic blocks that are reachable without error conditions are classified as 'normal execution.' 'Data errors' refers to code dealing with corrupted input files. 'User errors' refers to code run due to invalid user input. The 'OS errors' class deals with unexpected error conditions, including error codes returned from system calls as well as signal handling. 'Unreachable' code can never be executed. In `bzip2`, most unreachable code consists of functions in a library that may be used from other programs but that `bzip2` itself does not use. Note that our classification of basic blocks is based on the binary, which means that any code the compiler eliminates because it can prove it to be unreachable is not included. The 'panic' category refers to error conditions that should never occur, such as assertion failures. A few basic blocks did not include line number information, so we could not classify them.

Two of the columns in Table 2 specify coverage. The 'average' column specifies the percentage of basic blocks in this class executed per run, averaged over all the runs. The 'at least once' column specifies the percentage of basic blocks executed in at least one of the runs. This means that the 'normal execution' class of basic blocks, the basic blocks that can be reached by some input are on average reached in half the runs. The 'data errors' and 'user errors' classes that are exercised by the workload, on the other hand, are activated in only one in ten of the runs on average. These findings are consistent with

expectations, as it means that a relatively small fraction of the test cases in the workload attempt to trigger error handling code.

When considering what percentage of the code is in each class, it is noteworthy that 10.6% of the basic blocks can only be reached by triggering error conditions in the workload while 8.4% cannot or should not be reached at all. It is important when constructing high-coverage workloads as well regression tests that triggering error conditions is essential and also that 100% coverage is not realistic. When considering the size of each class in terms of lines of code rather than basic blocks, these numbers are 9.2% and 7.8% respectively. This further supports our previous finding that using the de facto standard measure of lines of code tends to underrepresent parts of the code that are particularly relevant in reliability research.

Previously, we have argued that the differences between the various coverage metrics we discussed may be caused by error handling code having relatively low coverage as well as relatively small basic blocks. This hypothesis is clearly supported by our data for `bzip2`, with the 'at least once' coverage per basic block averaging only 20.5% over all classes of error handling code, while it is more than four times as high for the 'normal execution' code class. The idea that error handling code consists of smaller basic blocks is also supported overall, but there are differences between the classes of error handling code. While code handling data and operating system errors does indeed consist of small basic blocks, code handling user errors tends to have larger basic blocks. We have looked into the code to find an explanation for this unexpected result and found that this is because user errors tend to give more verbose error messages so that more code is needed to print them. Because user errors make up a relatively small part of the error handling code, our hypothesis is still firmly supported. More generally speaking, our results show that code structure differs between the classes we identified. Although we do not claim these results are representative of other programs, it is clearly shown that this issue should not be ignored when evaluating coverage.

6.2 Execution count

The degree to which some parts of the code execute more often than others is rarely considered in fault injection experiments. Given that the impact of an activated fault may depend on the context, it is reasonable to expect that a fault being activated over and over again is more likely to have an impact than a fault activated only once each run. Although this would probably not make a difference when dereferencing an invalid pointer (which would lead to a segmentation fault on the first attempt), it is very relevant in the case of for example a memory leak (where available memory gradually runs out with subsequent activations). Therefore, it is prudent to consider whether repeated activation could introduce bias in fault injection experiments.

Which parts of a program are executed often is mostly determined by the control flow. Loops and recursion allow sections of the code to be executed ar-

bitrary numbers of times. However, the workload often determines the bounds of loop counters and the depth of recursion. We aim to determine whether the distribution of execution counts is affected mostly by the program or mostly by other factors such as the workload. In the former case there is no difference between fault injection experiments and production, so no bias is introduced. In the latter case this factor must be carefully considered.

To investigate the distribution of execution counts, we have plotted histograms showing the number of basic blocks with particular execution counts. These graphs are shown in Fig. 5. Both axes are logarithmic because of the extreme ranges of values they take. Both `bash` and `vim` (the programs with the most extensive regression test suites) show a more-or-less linear decline in frequencies as the execution count goes up. The regression test suite for `bzip2` is similar, though more ragged because of the smaller workload. This shape in a log-log histogram is typical of power law distributions [1], where the probability of each value k is proportional to $k^{-\alpha}$. Some other plots show a graph that increases, reaches a peak and then decreases linearly. This is the case for `bzip2`, `gzip` and `xz` with the workloads we constructed, as well as the `gzip` regression test suite, `ntfs-3g` and to some extent `gnuchess`. These are still good candidates for a power law-like distribution as the tail (higher values) is most important. Finally, both implementations of `sort` and `od` have distributions with two peaks. Such a graph suggests that part of the program is independent of input size, whereas another part runs a fixed number of times for each byte/line of input. This graph as a whole does not fit any commonly used probability distribution, but the behavior of the tail is still similar to a power law distribution.

The fact that execution counts of basic blocks are distributed roughly according to a power law and have fat tails means that the differences in execution counts between basic blocks are huge. This effect can readily be seen from the ranges of values in Fig. 5. Faults injected in the most executed locations get activated incomparably more often than those injected in other places.

Our aim is to find which factors influence this behavior. It is hard to find out directly from the graphs and an attempt to do so would be highly inaccurate. Since the execution count is generally either power law distributed or the tail can be approximated by such a distribution, we estimate the exponent of the distribution. Higher values of the exponent indicate that the frequencies go to zero faster, the tail is less fat and the distortion introduced less extreme. Hence, the exponent is a suitable way to characterize the execution count distributions. We assume the execution count follows a zeta distribution, a discrete power law distribution where the probability of the execution count being k is $k^{-\alpha}/\zeta(\alpha)$. Here α is the exponent we want to estimate and ζ is the Riemann zeta function. The exponent can be estimated by performing a maximum likelihood fit [1].

The average estimated exponents (over 50 experiments) and the standard errors are shown in Fig. 6. The standard errors are all very low, which is an indication that 50 experiments are enough to get an accurate estimate of the value of the exponent. We have also considered the standard deviation,

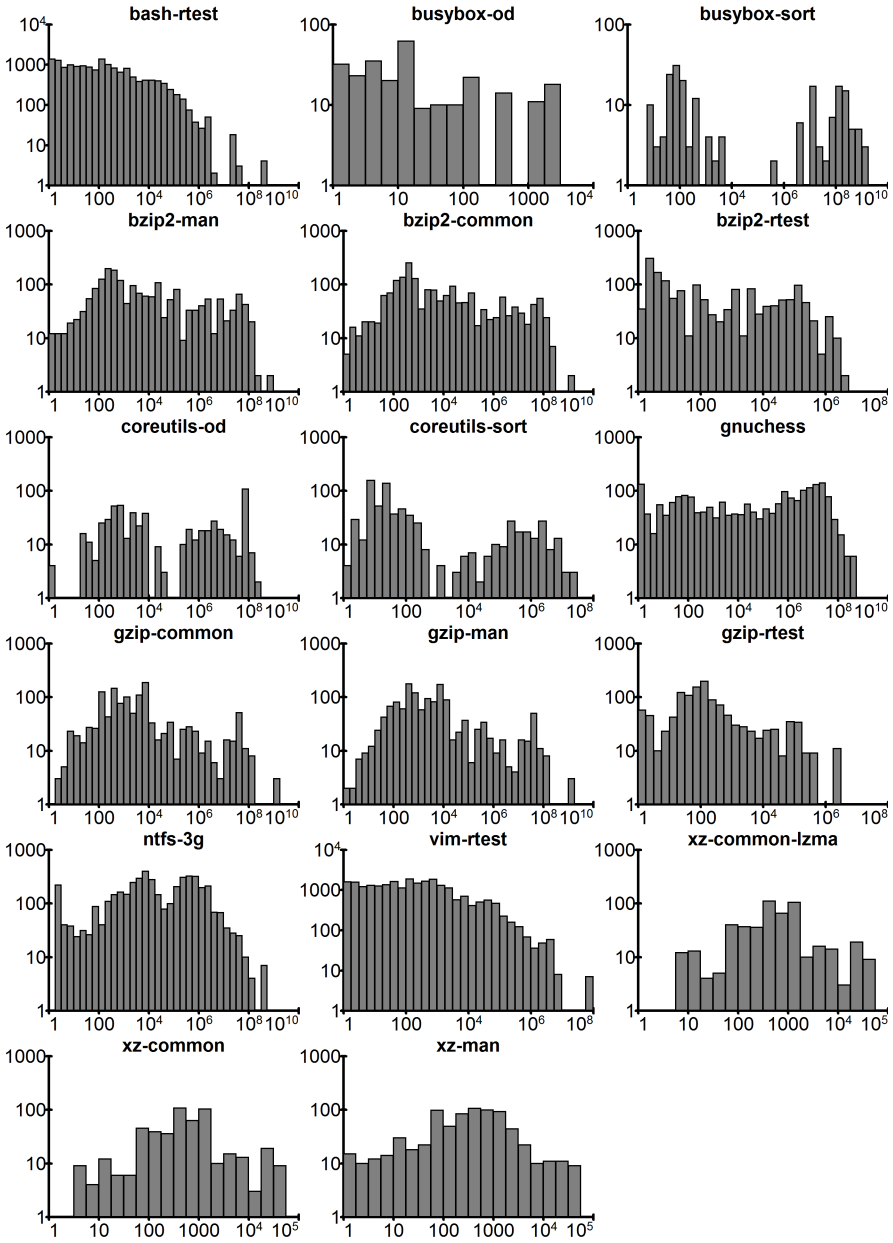

Fig. 5 Log-log histograms of execution count (median over 50 runs) per basic block; the x-axis shows the number of times a block was executed and the y axis how many basic blocks have been executed that often

which is slightly over seven times the standard deviation ($\sqrt{50}$ to be exact). In almost all cases, even the standard deviations are very low. This indicates

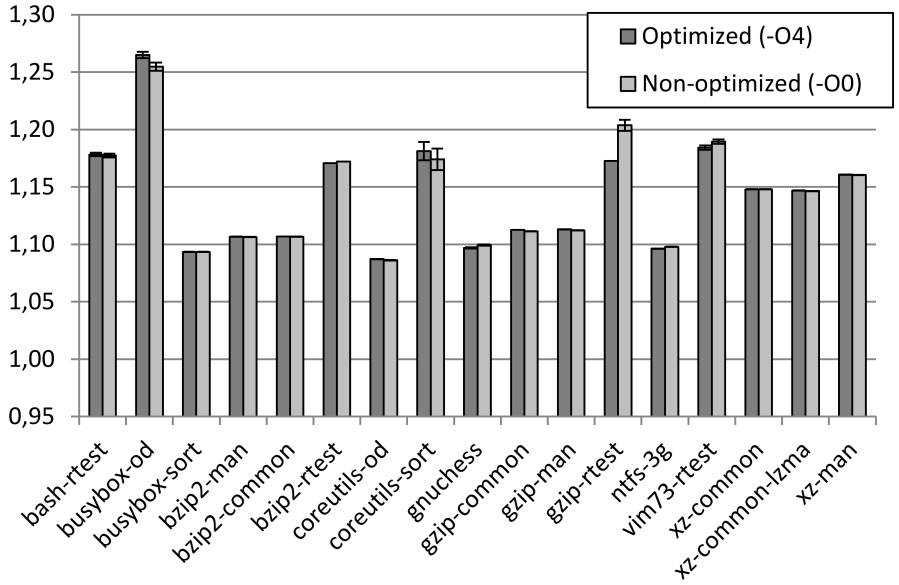

Fig. 6 Estimation of the distribution exponent; lines indicate standard errors

that the estimated parameter does not strongly depend on the random seed used to generate the workload. The main exception here is the Coreutils `sort` implementation, which has a standard deviation of 0.056 in the optimized case and 0.066 without optimization. The high standard deviations indicate that the random seed has considerable impact on distribution of execution counts for this program. Considering the numbers for the individual runs, they can be partitioned in two groups. The larger group (about 78% of the runs) has an average exponent of 1.141 (standard error 0.003, standard deviation 0.019) while the remainder averages at 1.291 (standard error 0.008, standard deviation 0.025). Considering these numbers, our workload and the Coreutils `sort` source code, it seems the difference is caused by the difference between merge operations and full sorts, with the merge operations resulting in less extreme execution counts. This suggests that the exponent of the execution count distribution provides a meaningful idea of what the program is doing.

It is noteworthy that the exponents are all close to one, which is the minimum for the exponent of the zeta distribution. These exponents suggest that all distributions are very fat-tailed and extreme execution counts are quite common.

The graph shows the impact of implementation and workload. The exponents for the two implementations of `od` and `sort` are not even close to each other, even though they run exactly the same workloads. The `bzip2` and `gzip` programs show that the workload also has a large impact. Although the difference is smaller than between programs, it is much higher than the standard deviation. This is consistent with the different shapes in Fig. 5. It is clear

that both different implementations of the same functionality and different workloads on the same program can result in different distributions.

We also tested the impact of the level of optimization on the distribution of execution counts. However, even though the programs tend to have more blocks when optimization is disabled, the distribution exponent barely changes in almost all cases. This similarity suggests that it is the structure of the original program code that matters, with the compiler having little influence. Only the regression test suite for `gzip` shows a substantial difference between the optimized and non-optimized case. The exponent is clearly higher for the non-optimized version, which means the fat tail is less extreme in this case. To find out why this is the case we investigated the raw per basic block execution counts and found that a number of basic blocks is executed approximately twice as often for the optimized version as the non-optimized version. This includes code in the cyclic redundancy check (CRC) and deflate algorithms, which are amongst the most often executed code sections. Considering the code locations where this happens it seems that the optimizer moves the place where the condition for `while` loops is checked, causing part of these loops to be executed an additional time. This should not be an issue for fault injection at the source code because the optimizer will check the code for side effects that make this optimization impossible. However, the modified code structure could have an impact on binary-level fault injection, causing injected faults to be executed a different number of times than was originally intended. For bitcode-level injection, this problem is only experienced if optimization is performed before fault injection. We used this approach here to be able to find the impact of optimization.

Our findings show that execution counts are affected by workload and have a large potential introducing distortion due to their fat-tailed distribution. High-fidelity fault injection requires execution counts similar to those in the production environment. Since the number of iterations of loops in the program is an important factor, care should be taken to select a realistic distribution of input sizes. In addition, the fact that the optimizer may modify the structure of loops in a way that affects execution counts suggests that binary-level injection may suffer from additional optimization-induced distortion.

6.3 Relationship between execution count and coverage

Residual faults are activated only by a small fraction of the tests [35]. This definition is based on the idea that such faults are likely to elude testing and are therefore more representative of real-world faults than other faults. To evaluate the impact of selection of residual faults on distortion, it is important to know whether residual fault locations are repeatedly executed to a similar degree as other locations.

Fig. 7 classifies basic blocks based on the fraction of runs triggering them (coverage) and shows geometric means of the maximum execution counts for the basic blocks in each coverage group. We use the maximum rather than the

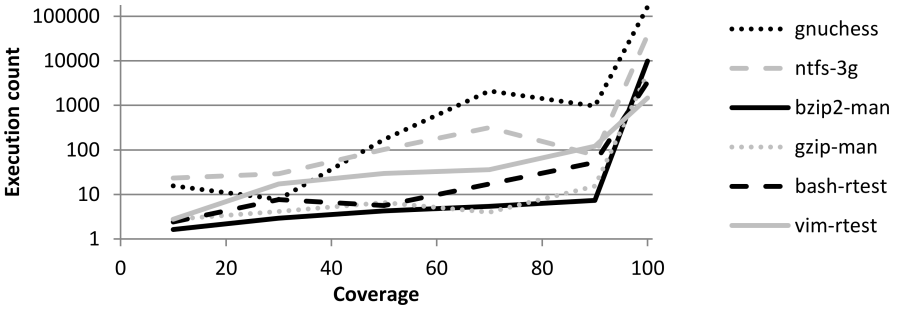

Fig. 7 Geometric mean of maximum execution count per basic block depending on coverage

mean or median for each basic block to prevent the zero execution counts from automatically introducing the effect of lower execution counts for residual locations. We use the geometric mean because we do not want to ignore extreme values (as the median would do) but we also do not want them to dominate all other values (as the arithmetic mean would do). Nevertheless, using either of these other measures the pattern is still the same. Standard errors are not directly applicable to geometric means, but we computed the standard error of the mean of the natural logarithm for each data point. This is at most 0.745, corresponding with a factor of 2.106. This shows that the effects shown are far larger than the errors. Some of the programs and workloads had either zero or very few basic blocks in some of the coverage groups. It is not possible to include these cases in the graph in a meaningful way, so unfortunately we had to leave them out.

Fig. 7 shows that basic blocks where residual faults would be injected execute far less often than other blocks, even in the workloads that activate them. Therefore, activated residual faults are expected to cause less damage compared to other activated faults. As a consequence, fault models that include both types are at risk of underestimating the impact of the residual faults. If the impact of such faults is expected to be important in production systems, they should be tested separately.

6.4 Relationship between faults and execution

We already considered impact of coverage and execution count on fault locations, but we have not considered the fault types yet. If particular fault types are more likely to execute or are executed more often, bias is introduced in the activated faults, which should be compensated by adjusting the input fault load for the experiment to be consistent with the fault model.

Our question is whether some fault types are more likely to execute than others. For each basic block and each fault type, we compute the fraction of faults in the block that is of that type. For each program, we compute the mean of these fractions for covered blocks and for uncovered blocks. Fig. 8 shows the average number of faults per basic block for each fault type, averaged over all

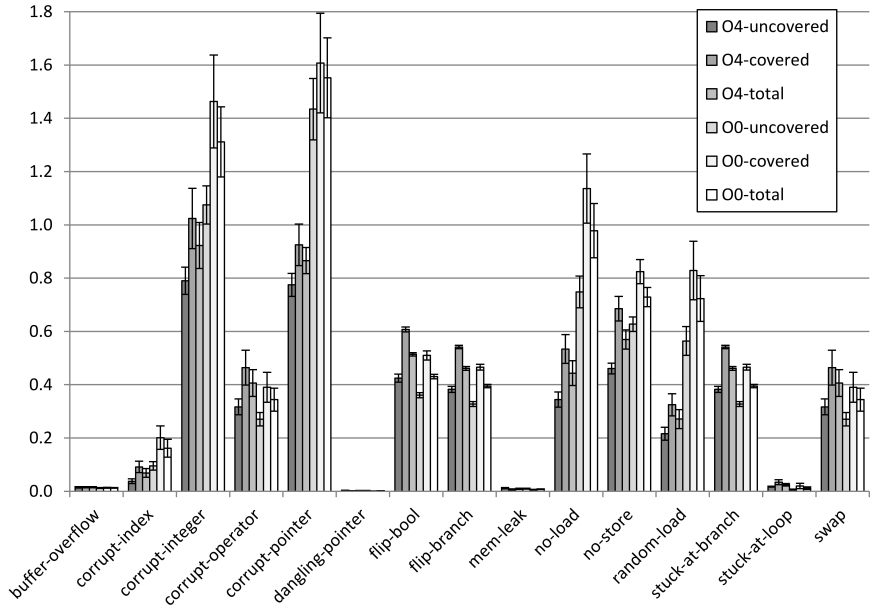

Fig. 8 Number of faults per basic block for each fault type, distinguishing whether blocks are covered by the workload and whether the program is optimized (O4) or not (O0); the numbers are an average over all programs/workloads and the lines refer to standard errors

programs (in the 'O4-total' and 'O0-total' bars), along with the standard error for this average. In optimized code, the corrupt-integer fault can be injected in most places, with an average of 0.922 per basic block (for a description of the fault types, see Table 1). For unoptimized code, the most common fault candidate is corrupt-pointer at an average of 1.552 per basic block. The difference can be explained by the fact that optimization tends to remove memory operations in favor of register operations. The least common fault candidate type is dangling-pointer, with only one in 400 basic blocks having a fault candidate of this type. The fact that there are such large differences in how often candidates for each fault type occur is relevant for fault injection. When selecting fault locations at random from the set of fault candidates, dangling-pointer faults would only be likely to be injected if a large number of faults is injected. If testing all the different fault types is important, it may be wise to preferentially select fault candidates of uncommon types.

There are large differences in the frequencies with which fault candidate types occur depending on the program. The standard errors in Fig. 8 provide some indication of the differences in how often fault types occur between the various workloads and programs. The full table of fault types per program is much too large to present here, instead we summarize by discussing on a number of cases where the differences between programs are particularly large. The corrupt-index, dangling-pointer, mem-leak and stuck-at-loop fault types

stand out for having very high standard errors relative to the mean, in some cases in excess of 100% of the mean. Corrupt-index is much more common in **bzip2** (0.189 per basic block) and **gnuchess** (0.168 per basic block) compared to the other programs. This means these programs have a relatively high number of array accesses. Dangling-pointer is relatively common in Coreutils **od** (0.008 per basic block) and **gzip** (0.005 per basic block), suggesting that memory allocations are relatively common for these programs. Mem-leak especially stands out for **ntfs-3g** (0.031 per basic block) which means that allocated memory is freed in many places. Finally, stuck-at-loop is exceptionally common in Busybox **od** (0.075 per basic block), Busybox **sort** (0.059 per basic block) and **bzip2** (0.034 per basic block), indicating that these programs have relatively many loops. These examples show that the fault candidate type distribution differs considerably between programs. It seems reasonable to assume that more fault candidates for a specific fault type would also lead to more real bugs because it means there are more opportunities for a programmer to introduce a fault. The implication for fault injection is that either the program should be considered explicitly when specifying a fault model or the fault model should be specified in such a way that the frequency of fault types being injected is proportional to the frequencies of fault candidates of that type. This approach would favor a specification such as 'inject a fault for 1% of the candidates for a missing load fault' over the alternative '10% of the injected faults should be of the type missing load.'

In addition to the fact that the fault type distribution differs between programs, Fig. 8 shows that some fault types are relatively likely to get activated while others are relatively unlikely to get activated. The corrupt-index and stuck-at-loop stand out for being relatively common in code that is actually executed. This means that array accesses and loops (the locations where these faults can be injected) are relatively common in the part of the program that performs the main task of the program. Dangling-pointer and mem-leak, on the other hand, occur more frequently in parts of the code that are not executed. This suggests that relatively many memory allocations and deallocations are reached only for certain program inputs. These results make clear that some fault types are more likely than others to get activated, introducing a bias in the output fault load. This should be dealt with by considering the distortion introduced and adjusting the input fault load accordingly to compensate for overrepresentation of fault types more likely to get activated and underrepresentation of those less likely to get activated.

It has been shown now that there is a large difference in terms of fault types between covered code and uncovered code, but we have not considered yet how often faults of different types get executed. The idea here is that specific types of faults might be more likely to occur in inner loops than others. We computed correlations between the relative fault candidate counts for each fault type and the execution count for blocks that did actually get executed at least once. The conclusion is that there is a significant correlation only for a few fault types and programs. In particular, loads tend to be executed relatively often in **bzip2** while array indexing is executed more often than other types of code in

`gzip`. `ntfs-3g` executes integer arithmetic and pointer arithmetic more often. This difference is not as large a source of bias as for example coverage, but it is still important to monitor fault injection experiments to find out which faults are executed more often. In cases where strict adherence to the fault types specified in the fault model is required, it would be wise to report on any distortion caused by certain fault types being executed more often than others.

7 Threats to validity

Although we have taken care to set up our experiments in the most realistic way possible, there are several factors that may have influenced the results that we presented and made them less realistic. In this section, we identify those factors that apply to the experiment as a whole and explain the reasons for choosing an approach that suffers from the issues identified. In addition there are several factors that apply only to certain parts of the experiment presented. Those factors have already been considered in drawing the conclusions in the subsections presenting the results of those experiments and are not repeated here.

In our experiments, we inject artificial software faults using fault types modeled on classifications of real faults found in the literature. Although we have taken care to select faults that are realistic, they are not real faults introduced by human programmers. The main alternative would be to use real faults instead by working through the log of fixed bugs, but we are not aware of any work using this approach in a large-scale reliability experiment. Computer-generated faults can never perfectly mimic real faults, which is a potential source of bias in our work. However, we feel that the use of real faults is not widely applicable and hence not suitable for a paper that aims to help improve practical fault injection experiments. This is due to the fact that considerable manual work is needed for each program being tested to identify a sufficient number of faults that have been fixed in the source control system. As a consequence, the number of faults that can be used is far lower, limiting the use of statistical techniques. For relatively new software, the number of bugs identified may be too low regardless of the amount of manual work that can be invested. In addition, the use of real faults is not without bias either because it can only use faults that have already been identified and fixed. This means that the most elusive bugs would not be considered if real faults were used.

This research aims to increase understanding of the impact of distortion on the injection of software faults that resemble bugs that programmers might introduce. Because the programmer works with the source code, this is also the best level to identify fault candidates and inject such bugs. We, however, have opted to do so at the intermediate code instead. The advantages of this approach are that our tools work with all language front-ends available for the LLVM compiler and there is no need to interfere with the build systems

of target programs to intercept arguments to the preprocessor. This threatens validity because the intermediate code may not be a one-to-one mapping of the intermediate code. Fortunately, for the LLVM compiler the intermediate representation is semantically very close to the original C code, for example making pointer arithmetic explicit and modeling variables rather than stack frames. If the intermediate code were further removed from the original code, information would be lost and fault injections would be less realistic. However, even in the case of LLVM there is the issue of preprocessor macros. Because preprocessor macros are expanded before compilation, they are not represented in intermediate code. This might be an issue if a fault were injected in the code resulting from macro expansion, because the fault would be injected in only one instance of the macro being expanded. That said, many papers using fault injection actually inject faults at the binary level (for example [11]), where this issue is far more pronounced [9,13]. For example, common compiler optimizations such as inlining and common subexpression elimination could cause faults to incorrectly affect more or less of the code than intended. Hence, we expect the impact on validity to be relatively low in this work.

Another factor that should be considered is the impact of the fault model. We have seen that there are meaningful differences in the locations where the various fault types occur. As a consequence, our own selection of fault types has had an impact on the results as well. Our own fault model comes down to using the fault types presented in Table 1 with the likelihood of selection proportional to the number of fault candidates. Since the fault types used are based on the literature on faults occurring in real software we consider this choice to be reasonable, but it is not the only possibility and using a different fault model or adding additional fault types could affect the results.

8 Recommendations

Throughout our evaluation, we have identified potential sources of distortion and provided advice to either mitigate it or where that is not feasible to report it. This section summarizes the recommendations to help readers reduce the impact of distortion on their fault injection work.

We found that the regression tests included with some programs resulted in lower coverage than our own tests based on the manual pages, most likely because our tests also introduce some errors in the input data. It is important to test not just correct input, but also incorrect input because error handling code is a typical place for residual errors to hide. Although this recommendation should be well-known, regression tests included with common open source programs show that such test cases are often omitted in practice. We found that error handling code is not only less likely to be reached by the workload at all, even the part that is activated on some runs is exercised on fewer runs. It is also recommended to avoid or remove unreachable code where possible because it makes the results harder to interpret. To increase coverage efficiently in case there is random variation in the workload, it may be worthwhile to

test how many runs are needed to maximize coverage before performing the experiment itself.

It is also very important to use the most suitable definition of coverage and be explicit about which was chosen, because we have shown that there can be substantial differences between them. We have tested four different coverage metrics and found that even though they are strongly correlated, there are meaningful differences between them. In the context of fault injection, measuring coverage in terms of fault candidates is recommended. In particular, definitions based on lines of code tend to downplay the importance of error-handling code, which has relatively few lines of code per basic block but is a particularly likely place to encounter real-world faults. Another important finding is the fact that coverage is not independent from fault types. Therefore, to achieve fidelity, fault injection tools should be configured to make the output fault load rather than the input fault load match the fault model.

In addition to these findings regarding coverage, we also investigated the distribution of basic block execution counts. The main conclusion is that this distribution has a fat tail, which means that extreme execution counts are relatively common. Since we have shown that the distribution is strongly influenced by the workload, it is important to select workloads with a similar input size distribution as would be found in a production environment. Although optimization has little impact in most cases, we found that optimizations may have a substantial impact on execution counts in rare cases where they cause parts of loops to be executed more often. A solution in this case would be to inject faults before optimization, using source-level or bitcode-level fault injection techniques. Another important consideration is the fact that execution counts tend to be higher in code that is executed by many runs of the workloads. As a consequence, experiments that inject both residual faults [35] and non-residual faults will most likely execute the non-residual fault more often, causing them to be overrepresented in the output fault load.

Regarding model specification, it is important to note that different types of programs differ in the distribution of fault candidate types. Assuming that each time a programmer writes code that could be subject to one of the fault types, there is a small chance that he/she indeed makes such a mistake. Therefore, the distribution of real faults can be expected to also be affected. To deal with this elegantly, it is recommended to specify the fault model in terms of the percentage of fault candidates that will be injected rather than a percentage of the total. However, if the goal is to test all fault types it is important to consider that the number of injection opportunities differs widely between fault types. In this case, it may be necessary to increase the injection rate for fault types with few candidates.

We also investigated the impact of compiler flags, in particular optimization levels. For most metrics, we did not find a meaningful impact of the choice of optimization level, which means we cannot provide general recommendations based on those experiments. However, we did show that optimization has a non-negligible impact on the availability of fault candidates. Therefore it is

recommended that compiler flags are set as they are in a production environment, rather than compiling code in debug mode for testing.

9 Conclusion

In this paper, we defined the concept of fidelity of a fault injection experiment to mean that the activated faults faithfully represent the fault model We have shown that careless fault injection experiments threaten fidelity and may not measure what the user intended, may be less efficient, less comparable and that problems with workload construction may remain hidden if fidelity is not considered. We performed a large-scale empirical evaluation of fidelity, resulting in advice on how to improve fidelity and raising awareness of the problem of fault load distortion.

Our research should be considered a first step towards improving fault injection fidelity. We identified the issue itself and performed experiments to confirm that there is indeed a potential problem and to give some preliminary guidance with regard reducing the impact on fault injection research. However, considerable future work is needed to fully understand the issue of fidelity. The most important step would be to define a quantitative metric for fidelity, which would allow for easier comparisons and hence stronger conclusions to be reached. Simple solutions such the Euclidean distance between vectors of fault candidate activation probabilities/counts are not suitable because of the fat-tailed distribution of execution counts we identified. Any work constructing a metric would need to find a meaningful way to weigh execution counts without making the extreme execution counts dominate the results. The only way to achieve this in our opinion is to look beyond activation and also quantify the extent to which activated faults cause any crashes or incorrect results. This is a direction for future work in itself and could also aid in identifying the quantitative impact of execution count. It would also allow for testing fault interactions by injecting multiple faults per run, another area for future research that is not possible within our exploratory methodology. Finally, it would be valuable to determine how robust our results are with regards to other environments, other languages and other fault types by replicating an experiment similar to this one in different settings. Our work is far from the last word on fidelity, but rather a starting point to encourage more research in this direction that could eventually allow lack of fidelity to be reported, compared and mitigated to improve the validity of software fault injection experiments.

References

1. A. Clauset C. Rohilla Shalizi, M.N.: Power-law distributions in empirical data. http://arxiv.org/abs/0706.1062 (2009)
2. Arlat, J., Crouzet, Y., Laprie, J.C.: Fault injection for dependability validation of fault-tolerant computing systems. In: Proc. of the 19th Int'l Symp. on Fault-Tolerant Computing, pp. 348–355 (1989)

3. Banabic, R., Candea, G.: Fast black-box testing of system recovery code. In: Proc. of the 7th ACM European Conf. on Computer Systems, pp. 281–294 (2012)
4. Barton, J.H., Czeck, E.W., Segall, Z.Z., Siewiorek, D.P.: Fault injection experiments using FIAT. IEEE Trans. Comput. **39**(4), 575–582 (1990)
5. Cadar, C., Dunbar, D., Engler, D.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proc. of the 8th USENIX Symp. on Operating Systems Design and Implementation, pp. 209–224 (2008)
6. Carreira, J., Madeira, H., Silva, J.G.: Xception: A technique for the experimental evaluation of dependability in modern computers. IEEE Trans. Softw. Eng. **24**(2), 125–136 (1998)
7. Choi, G., Iyer, R.: FOCUS: an experimental environment for fault sensitivity analysis. IEEE Trans. Comput. **41**(12), 1515–1526 (1992)
8. Christmansson, J., Chillarege, R.: Generation of an error set that emulates software faults based on field data. In: Proc. of the 26th Int'l Symp. on Fault-Tolerant Computing, p. 304 (1996)
9. Cotroneo, D., Lanzaro, A., Natella, R., Barbosa, R.: Experimental analysis of binary-level software fault injection in complex software. In: Proc. of the 9th European Dependable Computing Conf., pp. 162–172 (2012)
10. Cukier, M., Powell, D., Ariat, J.: Coverage estimation methods for stratified fault-injection. Computers, IEEE Transactions on **48**(7), 707–723 (1999)
11. Duraes, J.A., Madeira, H.S.: Emulation of software faults: A field data study and a practical approach. IEEE Trans. Softw. Eng. **32**(11), 849–867 (2006)
12. Dures, J., Madeira, H.: Emulation of software faults by educated mutations at machine-code level. In: Proc. of the 13th Int'l Symp. on Software Reliability Engineering, p. 329 (2002)
13. Giuffrida, C., Kuijsten, A., Tanenbaum, A.S.: EDFI: A dependable fault injection tool for dependability benchmarking experiments. In: Proc. of the Pacific Rim Int'l Symp. on Dependable Computing (2013)
14. Gu, W., Kalbarczyk, Z., Ravishankar, Iyer, K., Yang, Z.: Characterization of Linux kernel behavior under errors. In: Proc. of the Int'l Conf. on Dependable Systems and Networks, pp. 459–468 (2003)
15. Gunawi, H.S., Do, T., Joshi, P., Alvaro, P., Hellerstein, J.M., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H., Sen, K., Borthakur, D.: FATE and DESTINI: A framework for cloud recovery testing. In: Proc. of the 8th USENIX Conf. on Networked Systems Design and Implementation, pp. 18–18 (2011)
16. Gunneflo, U., Karlsson, J., Torin, J.: Evaluation of error detection schemes using fault injection by heavy-ion radiation. In: Proc. of the 19th Int'l Symp. on Fault-Tolerant Computing, pp. 340–347 (1989)
17. Herder, J.N., Bos, H., Gras, B., Homburg, P., Tanenbaum, A.S.: Failure resilience for device drivers. In: Proc. of the Int'l Conf. on Dependable Systems and Networks, pp. 41–50 (2007)
18. Hudak, J., Suh, B.H., Siewiorek, D., Segall, Z.: Evaluation and comparison of fault-tolerant software techniques. IEEE Trans. Rel. **42**(2), 190–204 (1993)
19. J. Christmansson, M.H., Rimn., M.: An experimental comparison of fault and error injection. In: Proc. of the 9th Int'l Symp. on Software Reliability Engineering, p. 369 (1998)
20. Jenn, E., Arlat, J., Rimen, M., Ohlsson, J., Karlsson, J.: Fault injection into VHDL models: the MEFISTO tool. In: Proc. of the 24th Int'l Symp. on Fault-Tolerant Computing, pp. 66–75 (1994)
21. Johansson, A., Suri, N., Murphy, B.: On the impact of injection triggers for OS robustness evaluation. In: Proc. of the 18th Int'l Symp. on Software Reliability, pp. 127–126 (2007)
22. Johansson, E., Suri, N., Murphy, B.: On the selection of error model(s) for OS robustness evaluation. In: Proc. of the 37th Int'l Conf. on Dependable Systems and Networks, pp. 502–511 (2007)
23. Joshi, P., Gunawi, H.S., Sen, K.: PREFAIL: A programmable tool for multiple-failure injection. In: Proc. of the ACM Int'l Conf. on Object Oriented Programming Systems Languages and Applications, vol. 46, pp. 171–188 (2011)

24. Kanawati, G.A., Kanawati, N.A., Abraham, J.A.: FERRARI: A flexible software-based fault and error injection system. IEEE Trans. Comput. **44**(2), 248–260 (1995)
25. Kao, W.L., Iyer, R.: DEFINE: A distributed fault injection and monitoring environment. In: Proc. of the IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems, pp. 252–259 (1994)
26. Karlsson, J., Folkesson, P.: Application of three physical fault injection techniques to the experimental assessment of the MARS architecture. In: Proc. of the 5th IFIP Working Conf. on Dependable Computing for Critical Applications, pp. 267–287 (1995)
27. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: formal verification of an OS kernel. In: Proc. of the 22nd ACM Symp. on Oper. Systems Prin., pp. 207–220. ACM (2009). DOI 10.1145/1629575.1629596
28. Koopman, P., Sung, J., Dingman, C., Siewiorek, D., Marz, T.: Comparing operating systems using robustness benchmarks. In: Proc. of the 16th Symp. on Reliable Distributed Systems, p. 72 (1997)
29. van der Kouwe, E., Giuffrida, C., Tanenbaum, A.S.: Evaluating distortion in fault injection experiments. Fifteenth IEEE International Symposium on High-Assurance Systems Engineering (HASE'14) pp. 25–32 (2014). DOI http://doi.ieeecomputersociety.org/10.1109/HASE.2014.13
30. Lattner, C., Adve, V.: LLVM: A compilation framework for lifelong program analysis & transformation. In: Proc. of the Int'l Symp. on Code Generation and Optimization, p. 75 (2004)
31. Madeira, H., Costa, D., Vieira, M.: On the emulation of software faults by software fault injection. In: Proc. of the Int'l Conf. on Dependable Systems and Networks, pp. 417–426 (2000)
32. Madeira, H., Rela, M.Z., Moreira, F., Silva, J.G.: RIFLE: a general purpose pin-level fault injector. In: Proc. of the First European Dependable Computing Conf., pp. 199–216 (1994)
33. Marinescu, P., Candea, G.: LFI: A practical and general library-level fault injector. In: Proc. of the Int'l Conf. on Dependable Systems and Networks, pp. 379–388 (2009)
34. Marinescu, P.D., Banabic, R., Candea, G.: An extensible technique for high-precision testing of recovery code. In: Proc. of the USENIX Annual Tech. Conf., pp. 23–23 (2010)
35. Natella, N., Cotroneo, D., Duraes, J., Madeira, H.: On fault representativeness of software fault injection. IEEE Trans. Softw. Eng. (99), 1 (2012)
36. Natella, R., Cotroneo, D., Duraes, J., Madeira, H.: Representativeness analysis of injected software faults in complex software. In: Proc. of the 40th Int'l Conf. on Dependable Systems and Networks, pp. 437–446 (2010)
37. Ng, W.T., Chen, P.M.: The design and verification of the Rio file cache. IEEE Trans. Comput. **50**(4), 322–337 (2001)
38. Ostrand, T.J., Weyuker, E.J.: The distribution of faults in a large industrial software system. ACM SIGSOFT Softw. Eng. Notes **27**(4), 55–64 (2002). DOI 10.1145/566171.566181
39. Sullivan, M., Chillarege, R.: A comparison of software defects in database management systems and operating systems. In: Proc. of the 22nd Int'll Symp. on Fault-Tolerant Computing, pp. 475–484 (1992)
40. Svenningsson, R., Vinter, J., Eriksson, H., Trngren, M.: MODIFI: a MODel-implemented fault injection tool. In: Proc. of the 29th Int'l Conf. on Computer Safety, Reliability, and Security, pp. 210–222 (2010)
41. Swift, M.M., Annamalai, M., Bershad, B.N., Levy, H.M.: Recovering device drivers. ACM Trans. Comput. Syst. **24**(4), 333–360 (2006)
42. Tsai, T.K., Hsueh, M.C., Zhao, H., Kalbarczyk, Z., Iyer, R.K.: Stress-based and path-based fault injection. IEEE Trans. Comput. **48**(11), 1183–1201 (1999)
43. Tsai, T.K., Iyer, R.K.: Measuring fault tolerance with the FTAPE fault injection tool. In: Proc. of the 8th Int'l Conf. on Modelling Techniques and Tools for Computer Performance Evaluation, pp. 26–40 (1995)
44. Zhou, F., Condit, J., Anderson, Z., Bagrak, I., Ennals, R., Harren, M., Necula, G., Brewer, E.: SafeDrive: Safe and recoverable extensions using language-based techniques. In: Proc. of the 7th Symp. on Operating Systems Design and Implementation, pp. 45–60 (2006)