Evaluating Distortion in Fault Injection Experiments

Erik van der Kouwe
Computer Systems Section
Faculty of Sciences, VU University
Amsterdam, The Netherlands
Email: erik@minix3.org

Cristiano Giuffrida
Computer Systems Section
Faculty of Sciences, VU University
Amsterdam, The Netherlands
Email: giuffrida@cs.vu.nl

Andrew S. Tanenbaum
Computer Systems Section
Faculty of Sciences, VU University
Amsterdam, The Netherlands
Email: ast@cs.vu.nl

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

***Index Terms*—fault injection; LLVM; reliability;**

I. 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 [1] even in mature software. Formal methods proposed to address such bugs, such as used by seL4 [2], would require a heroic effort. seL4's correctness proof alone, for example, required around 20 person years for 9,300 lines of codes. To scale to software that is hundreds to thousands of times larger would not currently be realistic. Furthermore, formal specifications can still 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. 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) [3], faults at the component interfaces (e.g., unexpected error conditions generated by the libraries) [4], [5], or real-world software faults introduced by programmers [6], [7]. 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 [4]. 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 [7] and at which locations faults are most likely to occur in production [8].

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 reflects the original fault model. 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.

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. 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 differences in timing. In this paper we only look at distortion introduced by nonactivation and multiple activation, which is an important

factor regardless of the exact types of failure 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 define *fidelity* as 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 [7], [8].

Our research question is: "*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.

II. 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 [9]), 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 [10], [11], file caches [6], operating systems [5], [12], user programs [4], [13], and distributed systems [14]. Typical evaluation scenarios entail analyzing the behavior of a system under faults [5], [12], conducting high-coverage testing experiments for existing error recovery code paths [13], [14], or evaluating the effectiveness and containment properties of fault-tolerance techniques [10], [11].

Several possible fault models are described in the literature, with fault injection strategies emulating (i) hardware faults [3], (ii) software faults [6], [7], (iii) interface faults at the library level [4] or (iv) at the system call level [5]. Injection techniques range from static program mutations—using compiler-based strategies [15] or binary rewriting [3], [6], [7]—to run time strategies that periodically interrupt the execution—using timers [3], [16], [17] or predetermined hardware or software traps [3], [16], [17].

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 [6], [7]. The G-SWFIT tool [7], 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. [18] 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. [20] investigate general limitations of traditional fault injection strategies when compared to real faults found in the field. In another direction, Natella et al. [8] 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 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], 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.

III. Fidelity

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. In the context of fault injection, it is particularly important 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. Unfortunately, coverage is rarely reported when performing fault injection experiments in research papers–with some notable exceptions [4], [8], [21]. 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 per-basic block execution count. 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. 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 [8] 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.

IV. 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 compiler framework [23] (version 3.2). Our analysis operates at the LLVM bitcode level, which (in contrast to approaches using the binary) allows us to preserve source-level information required for fault type representativeness [18], [24], while supporting a broad range of programs and platforms. For our investigation, we chose the standard software fault types commonly used in the literature [7], [25], [26].

While compiling each program, we identify all fault candidates and register in which basic block they occur. 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.

Our analysis is largely qualitative. Our aim is to show that distortion occurs in commonly used fault injection settings and under which circumstances distortion may be an issue. For this purpose, we do not need a quantitative measure of distortion. Nevertheless, future work constructing such a measure would certainly be valuable.

V. 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 AI is expected to be relatively complex and the `vim` editor to have an interactive program that has a good regression test. 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 a regression test was 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. Input files are randomly generated where needed, using a Markov chain approach. The transition matrix determining the type of file is picked randomly from several possibilities, including binaries as well as text in several languages. 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.

VI. Results

The results section has been split in sub-sections, each representing a part of the experiment. Threats to validity are different for each part, so they are not considered in a separate section but rather considered while drawing conclusions.

A. 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.

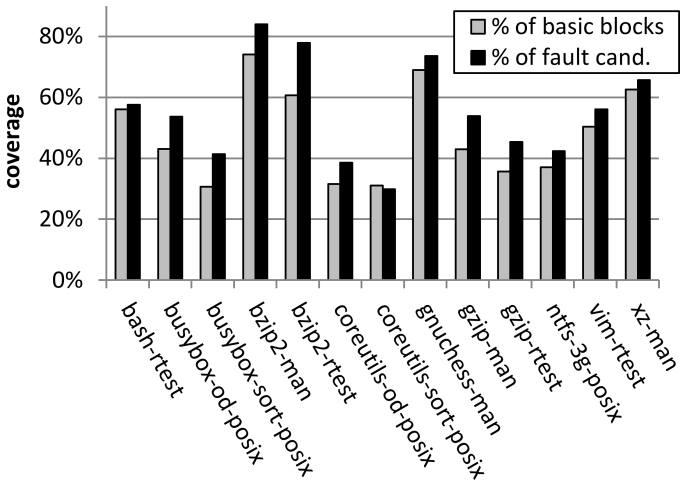

Fig. 1. Coverage per program and workload generator

Whenever the workload contains a degree of randomness, coverage increases as more runs are tested. Each input has some chance to trigger some code paths not previously reached. However, there are diminishing returns here. As more runs are performed the coverage eventually reaches the maximum for that particular workload generator. Our tests show that for all our workload generator scripts, 50 runs are easily enough to get a good impression of the maximum coverage that can be achieved. Due to our approach of counting basic block executions rather than injecting real faults, reaching the number of runs where coverage no longer increases is sufficient. There is no need for separate runs for each fault candidate to reach corner cases.

Fig. 1 shows the level of coverage we reached for each program using 50 runs. There is quite some diversity even for workloads generated in similar ways. For example, `bzip2-man` and `gzip-man` both test all combinations of flags described in their manual pages and introduce the same types of corruption, but the former reaches 74.1% coverage while the latter only gets 43.0%. 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 re-implementations of functions normally imported from libc 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. 1 also shows that the way coverage is measured can make a substantial difference. Often (but not always), coverage is higher in terms of fault candidates than in basic blocks. Though not shown in the graph, coverage in lines of code is generally slightly higher than in basic blocks. Coverage in

instructions is generally slightly lower than coverage in fault candidates. The difference 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.

The differences between the various definitions of coverage also mean that care must be taken in measuring coverage in the right way. 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 in terms of basic blocks and lines of code is very similar, coverage of fault candidates is 39% higher for the Busybox implementation. Therefore, fault injection experiments using Busybox are expected to be more faithful to the fault model. Care must be taken to report statistics in the most appropriate units, which in case of coverage in fault injection experiments we believe to be the percentage of fault candidates covered.

In addition to the comparison between programs and workloads, we have investigated the impact of optimization level on coverage. 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. Our results (omitted for brevity) show the optimization option tends to slightly increase coverage. For example, `-O4` on average results in just over one percent-point higher coverage in terms of fault candidates compared to `-O0`. In LLVM 3.2, `-O4` is the highest level of optimization possible, combining maximal compiler optimization (`-O3`) with link-time optimization (`-flto`). `gzip` is most affected, with coverage going up from 49.7% to 53.8%. This is mostly due to re-implemented libc functions with poor coverage being optimized more aggressively than the rest of the code. When ignoring `gzip`, the impact of optimization is even smaller. We recommend using the same compiler settings as in production settings, as the coverage advantage of changing them is negligible.

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 optimized (`-O4`) `bzip2` program based on the circumstances under which they are invoked. We then made 600 runs to determine coverage for each class of block. The result is shown in Table I. 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 OS error conditions. '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. 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.

TABLE I
CLASSIFICATION OF BASIC BLOCKS IN BZIP2

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

Table I provides some interesting insights. For `bzip2`, 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. In addition, it is shown that code structure differs between the classes identified. Code handling data and OS errors consists of small basic blocks. This part is underrepresented when coverage is expressed in terms of lines of code, understating its importance in testing. The opposite is seen with user errors, because more verbose output is provided in these cases. Although we do not claim these results are representative of other programs, it is clearly shown that these factors should not be ignored when evaluating coverage.

B. 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. A typical example of such a bug is a memory leak. 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 arbitrary 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

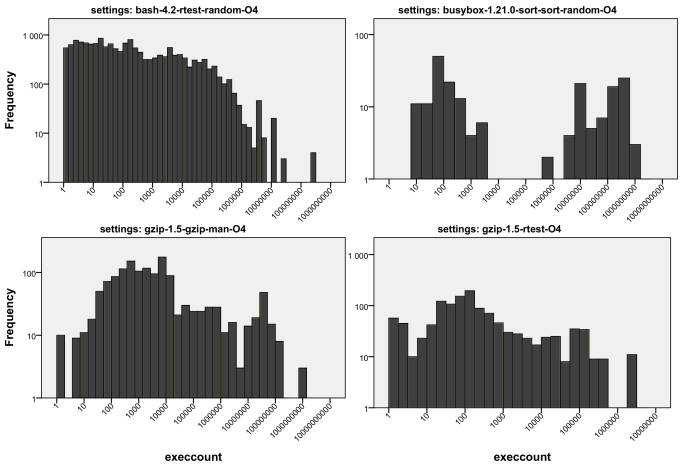

Fig. 2. Log-log histograms of execution count per basic block

particular execution counts. For brevity, we include only four programs which show all distinct shapes. These graphs are shown in Fig. 2. Both axes are logarithmic because of the extreme ranges of values they take. `bash` shows a more-or-less linear decline in frequencies as the execution count goes up. Of the programs not shown, `vim` has a very similar shape. This shape in a log-log histogram is typical of power law distributions [27], where the probability of each value x is proportional to $x^{-\alpha}$. `gzip` with our own workload generator results in a graph that increases, reaches a peak and then decreases linearly. The same applies to `bzip2` and `xz` with similar workloads, `gnuchess` and `ntfs-3g`. These are still good candidates for a power law-like distribution as the tail (higher values) is most important. The regression test for `gzip` is almost flat at the tail. The same applies to the `bzip2` regression test. These are power law distributions where the α parameter is very low. The graph is truncated because the workload was not run long enough for it to visibly decline. 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 number of times for each byte/line of input. This graph does not fit any commonly used probability distribution, but the behavior of the tail is still similar to a power law distribution.

Power law distributions are known to have fat tails, meaning that the differences in execution counts between basic blocks are huge. This effect can readily be seen from the ranges of values in Fig. 2. 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. To compare the distributions, we estimate their exponents. Higher values indicate that the frequencies decrease with count faster, the tail is less fat and the distortion introduced less extreme. We estimate the exponent by performing a maximum likelihood fit [27].

The average estimated exponents (over 50 experiments) and the standard deviations are shown in Table II. The standard deviations are low, so the estimated parameter does not strongly depend on the random seed. All exponents are close to one, which is the minimum. This suggests that the distributions are very fat-tailed and extreme execution counts are common.

TABLE II
ESTIMATION OF THE DISTRIBUTION EXPONENT

Program/workload	Exp.	S.D.	Program/workload	Exp.	S.D.
bash-rtest-random	1.18	0.01	gnuchess	1.10	0.00
busybox-od-posix	1.27	0.02	gzip-common-man	1.11	0.00
busybox-sort-posix	1.09	0.00	gzip-rtest	1.17	0.00
bzip2-common-man	1.11	0.00	ntfs-3g-posix	1.10	0.00
bzip2-rtest	1.17	0.00	vim73-rtest-random	1.18	0.01
coreutils-od-posix	1.09	0.00	xz-common-man	1.15	0.00
coreutils-sort-posix	1.18	0.06			

The table 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. 2. It is clear that both different implementations of the same functionality and different workloads on the same program can result in different distributions.

Our findings show that execution counts are affected by workload and have a large potential introducing distortion. 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.

C. Relationship between execution count and coverage

Residual faults are activated only by a small fraction of the tests [8]. 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. 3 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 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. The

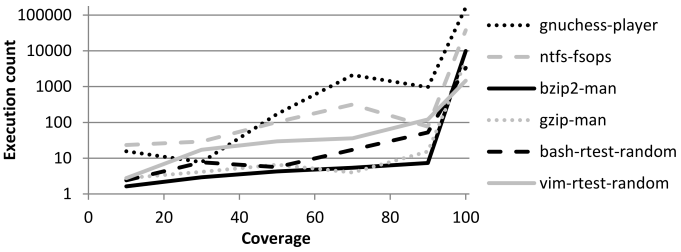

Fig. 3. Geometric mean of maximum execution count per basic block depending on coverage

programs and workloads not shown have too few basic blocks that are executed only in some of the runs to give meaningful results.

Fig. 3 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. Therefore, 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.

D. 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, 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. To find the bias introduced by incomplete coverage, we compare these means and perform a t-test to determine whether the difference is statistically significant.

There are large differences in the frequency with which fault candidate types occur depending on the program, the question whether a basic block is covered by the workloads and sometimes also the level of optimization. However, programs performing a similar task (the compression utilities as well as the implementations of `od` and `sort`) tend to be similar with regard to the distribution of fault types. It is also remarkable that the workload matters relatively little in this regard.

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 differences are large in both absolute and relative terms. The `bzip2` and `gzip` programs stand out for having a different fault candidate type distribution than most other programs. Regardless of the workload used, fault types related with integer arithmetic and memory loads are more common while branch-related types are less common. In addition, the common fault types are also more likely to actually get activated for these programs. When looking per fault type rather than per program, array index errors and load errors tend to be more common in executed code while stores and point arithmetic are less likely to get execute. Fault types related to loads and stores are clearly more common when optimization is disabled.

The fault candidate type distribution differs between types of programs, but it seems reasonable to assume that more fault candidates would also lead to more real bugs. The implication for fault injection is that either the type of program should be considered 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.’

It has been made plausible that indeed 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 the overrepresentation.

VII. 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 listed a number of factors that might be expected to introduce bias during fault injection experiments and investigated their impact. We found several factors that do indeed introduce bias and used them to provide a number of recommendations that should allow one to increase the fidelity of fault injection experiments and be more aware of biases that cannot easily be eliminated.

Regarding coverage, we found that the regression tests included with some programs performed less well 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. It is also recommended to avoid or remove unreachable code where possible because it makes the results harder to interpret. What is also very important is to use the most suitable definition of coverage and be explicit about which was chosen, because we have shown that there can be a substantial difference 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. 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 [8] 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.

We also investigated the impact of compiler flags, in particular optimization levels. We showed that these have 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.

In this paper, we defined the new concept of 'fidelity' of fault injection, which is important to ensure that the activated faults correspond with what was specified by 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.

Acknowledgment

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

References

[1] T. J. Ostrand and E. J. Weyuker, "The distribution of faults in a large industrial software system," *ACM SIGSOFT Softw. Eng. Notes*, vol. 27, no. 4, pp. 55–64, 2002.

[2] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "seL4: formal verification of an OS kernel," in *Proc. of the 22nd ACM Symp. on Oper. Systems Prin.* ACM, 2009, pp. 207–220.

[3] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham, "FERRARI: A flexible software-based fault and error injection system," *IEEE Trans. Comput.*, vol. 44, no. 2, pp. 248–260, 1995.

[4] P. Marinescu and G. Candea, "LFI: A practical and general library-level fault injector," in *Proc. of the Int'l Conf. on Dependable Systems and Networks*, 2009, pp. 379–388.

[5] P. Koopman, J. Sung, C. Dingman, D. Siewiorek, and T. Marz, "Comparing operating systems using robustness benchmarks," in *Proc. of the 16th Symp. on Reliable Distributed Systems*, 1997, p. 72.

[6] W. T. Ng and P. M. Chen, "The design and verification of the Rio file cache," *IEEE Trans. Comput.*, vol. 50, no. 4, pp. 322–337, 2001.

[7] J. A. Duraes and H. S. Madeira, "Emulation of software faults: A field data study and a practical approach," *IEEE Trans. Softw. Eng.*, vol. 32, no. 11, pp. 849–867, 2006.

[8] N. Natella, D. Cotroneo, J. Duraes, and H. Madeira, "On fault representativeness of software fault injection," *IEEE Trans. Softw. Eng.*, no. 99, p. 1, 2012.

[9] C. Cadar, D. Dunbar, and D. Engler, "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*, 2008, pp. 209–224.

[10] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy, "Recovering device drivers," *ACM Trans. Comput. Syst.*, vol. 24, no. 4, pp. 333–360, 2006.

[11] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum, "Failure resilience for device drivers," in *Proc. of the Int'l Conf. on Dependable Systems and Networks*, 2007, pp. 41–50.

[12] W. Gu, Z. Kalbarczyk, Ravishankar, K. Iyer, and Z. Yang, "Characterization of Linux kernel behavior under errors," in *Proc. of the Int'l Conf. on Dependable Systems and Networks*, 2003, pp. 459–468.

[13] R. Banabic and G. Candea, "Fast black-box testing of system recovery code," in *Proc. of the 7th ACM European Conf. on Computer Systems*, 2012, pp. 281–294.

[14] P. Joshi, H. S. Gunawi, and K. Sen, "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, 2011, pp. 171–188.

[15] J. Hudak, B.-H. Suh, D. Siewiorek, and Z. Segall, "Evaluation and comparison of fault-tolerant software techniques," *IEEE Trans. Rel.*, vol. 42, no. 2, pp. 190–204, 1993.

[16] T. K. Tsai and R. K. Iyer, "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*, 1995, pp. 26–40.

[17] J. Carreira, H. Madeira, and J. G. Silva, "Xception: A technique for the experimental evaluation of dependability in modern computers," *IEEE Trans. Softw. Eng.*, vol. 24, no. 2, pp. 125–136, 1998.

[18] D. Cotroneo, A. Lanzaro, R. Natella, and R. Barbosa, "Experimental analysis of binary-level software fault injection in complex software," in *Proc. of the 9th European Dependable Computing Conf.*, 2012, pp. 162–172.

[19] M. H. J. Christmansson and M. Rimn., "An experimental comparison of fault and error injection," in *Proc. of the 9th Int'l Symp. on Software Reliability Engineering*, 1998, p. 369.

[20] H. Madeira, D. Costa, and M. Vieira, "On the emulation of software faults by software fault injection," in *Proc. of the Int'l Conf. on Dependable Systems and Networks*, 2000, pp. 417–426.

[21] T. K. Tsai, M.-C. Hsueh, H. Zhao, Z. Kalbarczyk, and R. K. Iyer, "Stress-based and path-based fault injection," *IEEE Trans. Comput.*, vol. 48, no. 11, pp. 1183–1201, 1999.

[22] A. Johansson, N. Suri, and B. Murphy, "On the impact of injection triggers for OS robustness evaluation," in *Proc. of the 18th Int'l Symp. on Software Reliability*, 2007, pp. 127–126.

[23] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. of the Int'l Symp. on Code Generation and Optimization*, 2004, p. 75.

[24] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum, "EDFI: A dependable fault injection tool for dependability benchmarking experiments," in *Proc. of the Pacific Rim Int'l Symp. on Dependable Computing*, 2013.

[25] J. Christmansson and R. Chillarege, "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*, 1996, p. 304.

[26] M. Sullivan and R. Chillarege, "A comparison of software defects in database management systems and operating systems," in *Proc. of the 22nd Int'll Symp. on Fault-Tolerant Computing*, 1992, pp. 475–484.

[27] M. N. A. Clauset, C. Rohilla Shalizi, "Power-law distributions in empirical data," http://arxiv.org/abs/0706.1062, 2009.