Improving Software Fault Injection

Ph.D. Thesis

Erik van der Kouwe

Vrije Universiteit Amsterdam, 2016

vrije Universiteit *amsterdam*

This work was funded in part by European Research Council under ERC Advanced Grant 227874.

ISBN XXX-XX-XXXX-XXX-X

Printed by XXX.

VRIJE UNIVERSITEIT

IMPROVING SOFTWARE FAULT INJECTION

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad Doctor aan
de Vrije Universiteit Amsterdam,
op gezag van de rector magnificus
prof. dr. Vinod Subramaniam,
in het openbaar te verdedigen
ten overstaan van de promotiecommissie
van de Faculteit der Exacte Wetenschappen
op X XXX 2016 om X.XX uur
in de aula van de universiteit,
De Boelelaan 1105

door

ERIK VAN DER KOUWE

geboren te Leidschendam, Nederland

promotor: prof.dr. A.S. Tanenbaum

“TODO add quote.”

TODO add quote source.

Contents

List of Figures

List of Tables

Publications

This dissertation consists of the following research papers, published in peer-reviewed journals and conferences (or submitted for review to such):

Erik van der Kouwe, Cristiano Giuffrida, and Andrew S. Tanenbaum. Finding fault with fault injection: an empirical exploration of distortion in fault injection experiments[1]. In *Software Quality Journal*. Pages 1–30, 2014.

Erik van der Kouwe, Cristiano Giuffrida, and Andrew S. Tanenbaum. On the Soundness of Silence: Investigating Silent Failures Using Fault Injection Experiments[2]. In *Proceedings of the Tenth European Dependable Computing Conference (EDCC '14)*. May 13-16, 2014, Newcastle upon Tyne, UK.

Erik van der Kouwe, Cristiano Giuffrida, Razvan Ghitulete, and Andrew S. Tanenbaum. A Methodology to Efficiently Compare Operating System Stability[3]. In *Proceedings of the 16th IEEE International Symposium on High-Assurance Systems Engineering (HASE '15)*. January 8-10, 2015, Daytona Beach, FL, USA.

Erik van der Kouwe and Andrew S. Tanenbaum. HSFI: representative fault injection scalable to large code bases[4]. *Under review*.

[1] Appears in Chapter 2.
[2] Appears in Chapter 3.
[3] Appears in Chapter 4.
[4] Appears in Chapter 5.

The following publications have not been included in the thesis:

Erik van der Kouwe, Cristiano Giuffrida, and Andrew S. Tanenbaum. Evaluating Distortion in Fault Injection Experiments. In *Proceedings of the 15th IEEE Symposium on High-Assurance Systems Engineering (HASE '14)*, January 9-11, 2014, Miami, FL, USA.
Awarded Best Paper.

Koustuba Bhat, Ben Gras, Erik van der Kouwe, Dirk Vogt, and Cristiano Giuffrida. Taking the 'distributed' out of distributed recovery. *Under review*.

1 Introduction

Fault injection

It is unlikely that anyone reading this thesis has never experienced a computer system failing. Almost certainly, you have experienced the situation where you lost a document you were working on when your operating system suddenly stopped and required a reboot to be able to continue. It is very likely that at some point you were trying to use an online service but it was unavailable and would not respond to your computer's requests. There is a fair chance that at some point your hard drive just stopped working and you had to recover all your important files from a backup, assuming you had one. In all these cases, the computer system you were using was not behaving as it was designed to do because some *fault* caused something unexpected to happen that the system was not able to deal with transparently. There are many different types of faults possible. In case of the operating system crash, the most likely possibility is that a programmer made a mistake while writing the program code of the operating system. Due to this mistake, the operating system ended up in a state where it was no longer able to provide the services required by applications, causing the system to hang and require a reboot. In case of the online service, some external system failed to provide the proper response to your system, presumably because either the server providing the service was unavailable or overloaded or the network between the computers was not delivering the messages between them properly. In case of the hard drive, wear may have caused the disk head to crash into the disk platter, damaging the magnetic surface and making it unreadable. In all these cases, users typically have no way of knowing which faults exist and when and why they might be triggered. Still, because the system was not able to deal with the consequences of the faults, the system failed and the user is left to deal with the consequences of a hardware or software fault they had no part in.

Computers have been failing for such a long time that users have come to accept this and started to live with it. In the best case they make backups and have redundant systems available to reduce loss of data and to speed up recovery after failure. In fact, if they do not, others would blame them and hold them responsible for the consequences. If buildings were like computer systems, the present situation would be one where buildings are expected to collapse every few weeks. Moreover, the people inside would be blamed for not wearing protective gear inside if they get hurt as a consequence because they should have known that buildings fail from time to time. This situation is highly undesirable not just for buildings, but also for computer systems.

The question, then, is what we can do about those failures. In an ideal world, we would simply prevent the faults causing those failures from being introduced in the first place. In case of the failing hard disk, this can be achieved by improving the design and manufacturing process and by replacing the hard disk in time, before wear introduces faults. In case of the Internet service, redundant servers and networks could be added that prevent your system from ever getting an error response when trying to reach the service. Software, however, is different. To err is human and computer programs are written by humans. Even the best programmers make mistakes when writing code, introducing faults in the software as a consequence. Safe programming languages [97] have been developed that can reduce the impact of faults. Although this approach is promising, it cannot eliminate all faults and has not yet caught on in widely-used operating systems for both legacy and performance reasons. This is unfortunate, as a working operating system is essential for any applications program to work properly and the operating system runs at the highest privilege level, allowing it to do most damage in case of faults. As a consequence, there is currently no viable solution to prevent faults from being introduced in software.

The second best option is to ensure that all faults are discovered and eliminated before the product is shipped to the consumer. In case of hard drives, we expect extensive quality assurance to detect defective drives and prevent them from leaving the factory, especially for the high-end drives used in critical systems. For software faults this means that we would need to test systems thoroughly to find these faults and fix them before the software ever reaches the consumer. Unfortunately it turns out that testing software is a very hard problem. Research has shown that even in mature software, faults tend to persist over a large numbers of releases [100]. There has been considerable work on automatic bug detection tools, but even those cannot prevent software bugs from being a major source of system crashes [83]. The seL4 [72] system has shown that formal methods can be effective in proving that a system is bug-free, but they require a heroic effort to be used in practical systems. seL4's correctness proof alone, for example, required around 20 person years for 9,300 lines of code. For comparison, the latest version of Linux (4.2.4) contains 13,598,692 lines of code (as counted by Cloc[3]). This projects out to 29,000 person years to prove Linux correct even if we ignore the fact that Linux was not built with correctness proofs in mind. This gets even worse due to the fact that operating systems like Linux are constantly changing. We must conclude that, even though

there are some ways to reduce the number of software faults, it is not viable to eliminate software faults from computer systems entirely.

Now that it has been established that we must accept the presence of software faults as a fact of nature, we must shift our focus to building *fault-tolerant* systems that are able to withstand the impact of faults without resulting in failure. Let us once again revisit our examples to show what this would mean in practice. In case of the hard drive, it is possible to build a RAID system that uses several disks, each of which has a copy of all that data that is stored on the system. If one of the disks fails, the other disks can continue to provide the data that is requested. Assuming that disk faults are independent and the failed disk can be replaced on the fly, the system as a whole continues working and users are never presented with a failure. In case of the online service, the system can use local storage to allow users to continue to work on the data they already downloaded. When the system becomes available once more, the local copies can be updated and changes made by users uploaded to the server. In case of software faults, this means that the system must be able to contain to impact of faults to prevent them from bringing down the system, to detect that something went wrong, and to restore the system to a proper working state without user intervention.

The good news is that many techniques have been developed in the research literature to prevent software faults from resulting in failures and make computer systems more fault-tolerant (for example [107; 55; 64; 108; 112; 22; 103; 81; 127; 94; 44; 126]). The bad news is that they have not been widely adopted in practice. The reasoning behind this is easy to guess. Failure containment, detection and recovery systems cost time, both in terms of development time and run-time performance. Time is easy to measure and compare, in the simplest case simply by using a stopwatch while making the system perform a particular task. Time is money. To make a good case for improving reliability through fault-tolerance is is important to be able to compare solutions and quantify how much time or money can be saved by implementing techniques to improve system reliability.

Unfortunately, measuring fault-tolerance is much harder than measuring time. To be able to measure fault-tolerance, one needs to evaluate how well a system works in the presence of faults. Although, as we discussed before, there are plenty of software faults present in production software, they are unfortunately very hard to use for measuring reliability. They are often hard to trigger, as is shown by the fact that it occasionally takes years of production use for them to be discovered. Of course, plenty of more obvious faults are found in the testing phase, but such faults are not representative of the residual faults that end up in production systems [96]. Moreover, it is not sufficient to determine how the system reacts to faults that have already been found; it is more important to be able to predict how the system will behave in response to the faults that have not been found yet. To be able to make meaningful predictions, one needs to use statistical methods, which requires a large number of experiments. Given the fact that each fault requires manual effort to find and to create a workload that triggers it, this is not viable when using only real-world faults.

Given that we need faults to be able to measure reliability and real faults are insufficient to reach strong conclusions, it becomes clear that *fault injection* is necessary. Fault injection is the introduction of deliberate faults into a system to determine how the systems behaves in the face of faults. This important technique has is widely used in reliability research. Even when considering just the injection of software faults in operating systems, it has been used to evaluate operating system stability in case of faults in the operating system itself [125; 71], to test isolation properties to protect against faulty device drivers [39; 54], to certify safety properties [33], and to improve fault tolerance by finding the most likely situations in which data can be lost [98]. In the next section, we compare the various types of fault injection that are used and explain why we focus on software fault injection in particular.

Software fault injection

The first step in a fault injection campaign is to select a *fault model*. The fault model specifies what kinds of faults the fault tolerance mechanism that is being evaluated should be able to deal with. In broad categories, fault models could specify external faults, hardware faults or software faults.

The category of external faults refers to faults that are external to the (component of the) system itself and observed on the interfaces of the (component of the) system that is being evaluated. Using the examples from the previous section, this would be the online service that is unavailable. Although there are many possible reasons why the external server could have failed, the only thing that matters for the purpose of fault tolerance is which message is received at the system's interfaces. Hence, the fault model essentially consists of all the possible responses on the external interfaces or a subset thereof. Such fault models have the benefit of allowing for a fairly simple specification based on a well-defined API or protocol. They are also relatively easy to implement in practice as the system component being tested does not need to be changed; it is sufficient to change the libraries implementing the API or interpose in the message exchanges with external systems. Examples of systems using this fault model are LFI [90] and FATE [50].

The second type of fault model, hardware faults, corresponds with the example of the failed hard drive (assuming at least that it is considered part of the system being tested, it could be considered an external failure from other perspectives). Hardware faults often include fault types such as soft errors, for example bit flips in the CPU or RAM that corrupt some code or data. Hardware faults can be transient (for example due to a cosmic ray) or permanent (for example due to a defective component, as in the hard drive example). The fault model specifies what kind of transient errors or defects are likely and the impact on the running system can be derived from the design of the hardware and physical models of their interaction with the environment. This allows the faults to be realistically emulated using either hardware (such as for example MESSALINE [12]) or software (like for example

FINE [68] and Xception [23]).

Finally, there are software fault models where the fault is caused by a programmer making a mistake while writing program code. Fault injection using such fault models is also known as software mutation testing. This corresponds with our earlier example of the computer requiring a reboot due to a bug in the operating system. Software fault models stand out due to the fact that these are mistakes made by humans; they do not follow a predefined specification nor any physical model of an electronic component. As such, they cannot be predicted theoretically but must rather be derived empirically from mistakes humans make in practice. Such fault models have been derived in the literature [27; 41; 70]. The fault injector must then find locations where a human programmer would have been likely to make the selected mistakes and modify the code there according to the way a human would have gotten it wrong. For example, the termination condition of a loop is changed to use a less than comparison operator while it should have been less than or equal to. A second major difference with the fault models discussed before is the fact that the program itself must be modified. To faithfully emulate the selected fault model, the introduced faults must carefully mimic those specified by the fault model. This is more difficult than for the other types of fault models, where a suitable fault injection is often simply an error code or a bit flip. These factors distinguish software fault models from the other types of fault models and makes it particularly challenging to get software fault injection right.

Even though software fault injection is widely used, it is rapidly developing and still far from being a mature technology. As such, there remains much to be improved before we can reach a situation where fault injection will be routinely applied in a systematic and methodologically sound way. This dissertation provides some steps in this direction, as will be explained in the next section.

Improving software fault injection

Given that software fault injection is both particularly important and difficult to perform in a methodologically sound way, this dissertation specifically focuses on ways to advance the state of the art of software fault injection. This dissertation consists of papers that have been published in or submitted to peer reviewed conferences and journals, each identifying potential pitfalls that occur while using software fault injection to measure fault-tolerance and suggesting solutions. The following papers are included as chapters:

- Chapter 2 explores the relationship between faults injected and faults actually activated to determine how the workload causes the activated faults to be distorted with respect to the defined fault model. It provides a definition of fault injection fidelity and shows the relevance of distortion problems in real-world fault injection experiments by performing a large-scale evaluation of fidelity on a number of programs and workloads. It also analyzes the key factors that can help pre-

dict and control distortion problems in fault injection experiments. Chapter 2 appeared in the *Software Quality Journal*[118]. This is an extended version of an earlier paper that was presented at the IEEE Symposium on High-Assurance Systems Engineering (HASE '14)[117] and won the best paper award there. Cristiano Giuffrida contributed to the related work section of this paper.

- Chapter 3 discusses the importance of silent failures for fault injection experiments. It describes a new automated methodology to identify silent failures in fault injection experiments that is generally applicable and can automatically identify several classes of failures from the outcome of a given experiment. This approach has been implemented and used to evaluate the impact of silent failures across several different programs for various fault types and fault locations. Our results demonstrate that the impact of silent failures is relevant and should be carefully considered in dependability benchmarking scenarios. Chapter 3 was presented at the *European Dependable Computing Conference (EDCC '14)*[119]. Cristiano Giuffrida contributed to the related work section of this paper.

- Chapter 4 provides a new measure of operating system stability in the face of software faults that is, at the same time, comparable, representative, and scalable. The concept of stability is important because it is unreasonable to expect a system to work according to specifications in case a fault is injected. We address this by using pre-tests and post-tests to distinguish the impact of the fault itself from the system becoming unstable after dealing with the fault. Our approach uses statistical testing, allowing the researcher to determine whether enough tests have been conducted to draw conclusions from the results. In addition, we have implemented our approach in a way that is easily portable and we have performed experiments to demonstrate the usefulness of our approach. Chapter 4 was presented at the *Symposium on High-Assurance Systems Engineering (HASE '15)*[119]. Cristiano Giuffrida contributed to the related work section of this paper and Razvan Ghitulete helped performing the experiments.

- Chapter 5 provides a new design point in software fault injection that combines availability of source-level information to the fault injector with scalability to large code bases. It also provides a practical implementation of this approach, which is shown to deliver considerably better performance compared to other source-level approaches. It comes close to the performance of binary-level approaches that lack source-level information. This paper is currently under review for publication.

2 Finding fault with fault injection An Empirical Exploration of Distortion in Fault Injection Experiments

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.

2.1 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 [100] even in mature software. Formal methods proposed to address such bugs, such as used by seL4 [72], require a heroic effort. seL4's correctness proof alone, for exam-

ple, 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) [65; 16], faults at the component interfaces (e.g., unexpected error conditions generated by the libraries) [90; 89; 74], or real-world software faults introduced by programmers [99; 127; 41]. 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 [90; 89]. 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 [41] and at which locations faults are most likely to occur in production [95; 96].

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 be-

```
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  }
```

Figure 2.1: Example function to demonstrate distortion

cause 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 [41; 95; 96]. We will provide a formal definition of fidelity and the other relevant terms in Section 2.3.

To demonstrate the concepts we introduced and indicate why fidelity is important, we have included a small code example in Fig. 2.1. The first step is to identify fault candidates. In working out the example we will use the same fault types used

Table 2.1: 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

in our experiment. These are listed in Table 2.1. 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 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 [117], 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.

2.2 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 [21]), fault injection is rel-

atively 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 [112; 127; 55], file caches [99], operating systems [49; 74], user programs [15; 90; 89], and distributed systems [50; 63]. Typical evaluation scenarios entail analyzing the behavior of a system under faults[74; 49], conducting high-coverage testing experiments for existing error recovery code paths [15; 63; 50], or evaluating the effectiveness and containment properties of fault-tolerance techniques [112; 127; 55].

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 [50; 63; 15] 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 [51; 12; 87; 69] or software [65; 16], (ii) software faults [99; 127; 41], (iii) interface faults at the library level [90; 89] or (iv) at the system call level [74]. Injection techniques range from static program mutations—using simulation-based strategies [26; 59; 111], compiler-based strategies [56; 127], or binary rewriting [65; 16; 99; 41]—to run time strategies that periodically interrupt the execution—using timers [114; 65; 23] or predetermined hardware or software traps [114; 65; 23; 67].

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 [99; 40; 127; 41]. The G-SWFIT tool [41], 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. [31] consider the accuracy problems of binary-level injection strategies when compared to source-level program mutations. Christmansson et al. [28] compare location-based injection strategies with timer-based approaches. Madeira et al. [88] investigate general limitations of traditional fault injection strategies when compared to real faults found in the field. In another direction, Natella et al. [95; 96] 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 [115; 62; 61], but their focus is limited to ensuring reasonable fault ac-

tivation. 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 [36]. 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.

2.3 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 [115; 90; 89; 96]. 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 subsequent 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 [96] 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 caus-

ing 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.

2.4 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 [79] (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 [31; 47].

For our investigation, we chose the standard software fault types commonly used in the literature [41; 27; 110]. The selected fault types are listed in Table 2.1. 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.

2.5 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 [62].

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.

2.6 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.

2.6.1 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.

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. 2.2 shows the total coverage as a function of the number of times the workload 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

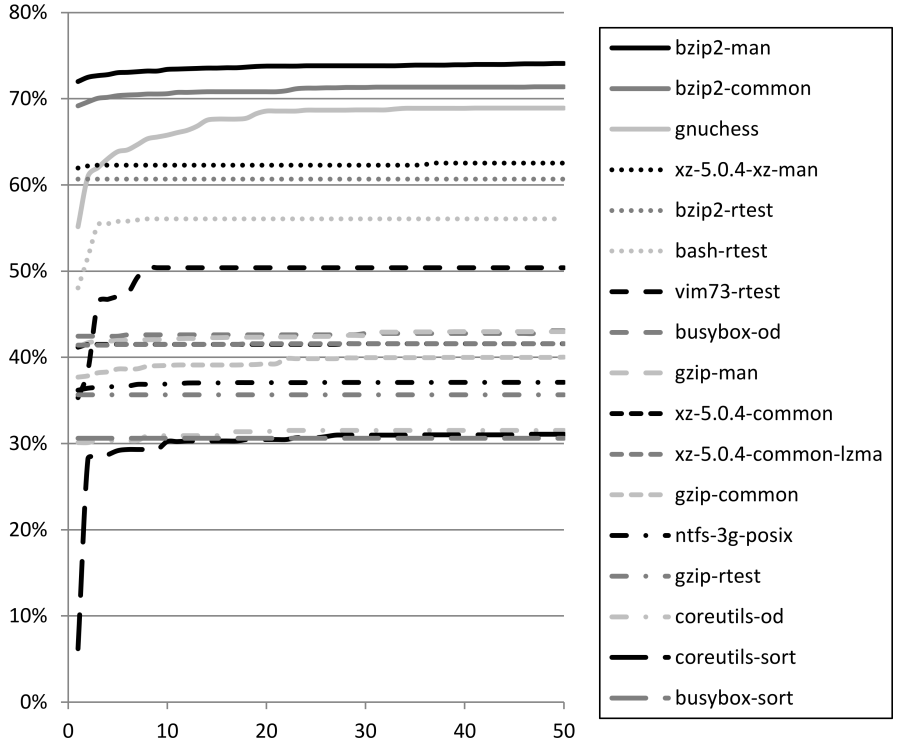

Figure 2.2: Coverage in basic blocks as a function of the number of runs with `-O4` optimization

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.

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 considera-

tion 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. 2.3 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. 2.3 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, 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

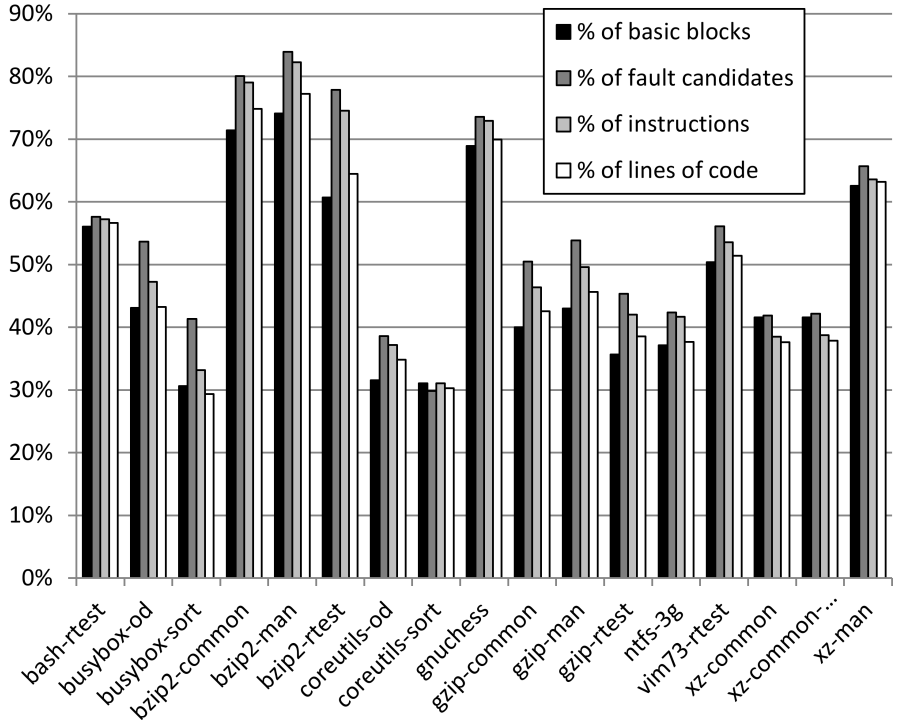

Figure 2.3: Coverage per program and workload generator with `-O4` optimization

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 2.6.1.

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

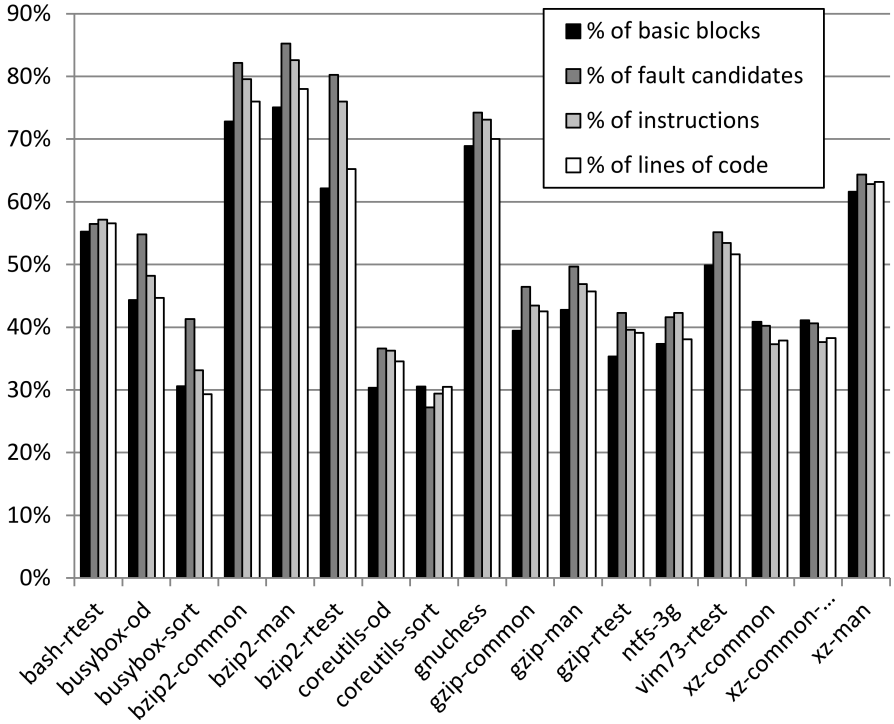

Figure 2.4: Coverage per program and workload generator without optimization

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. 2.4 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 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.

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 2.2. Basic blocks that are reachable without error conditions are classified as ‘normal execution.’ ‘Data errors’ refers to code

Table 2.2: 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%

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 2.2 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.

2.6.2 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 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 particular execution counts. These graphs are shown in Fig. 2.5. 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 [7], 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. 2.5. 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 [7].

The average estimated exponents (over 50 experiments) and the standard errors are shown in Fig. 2.6. 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, 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 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

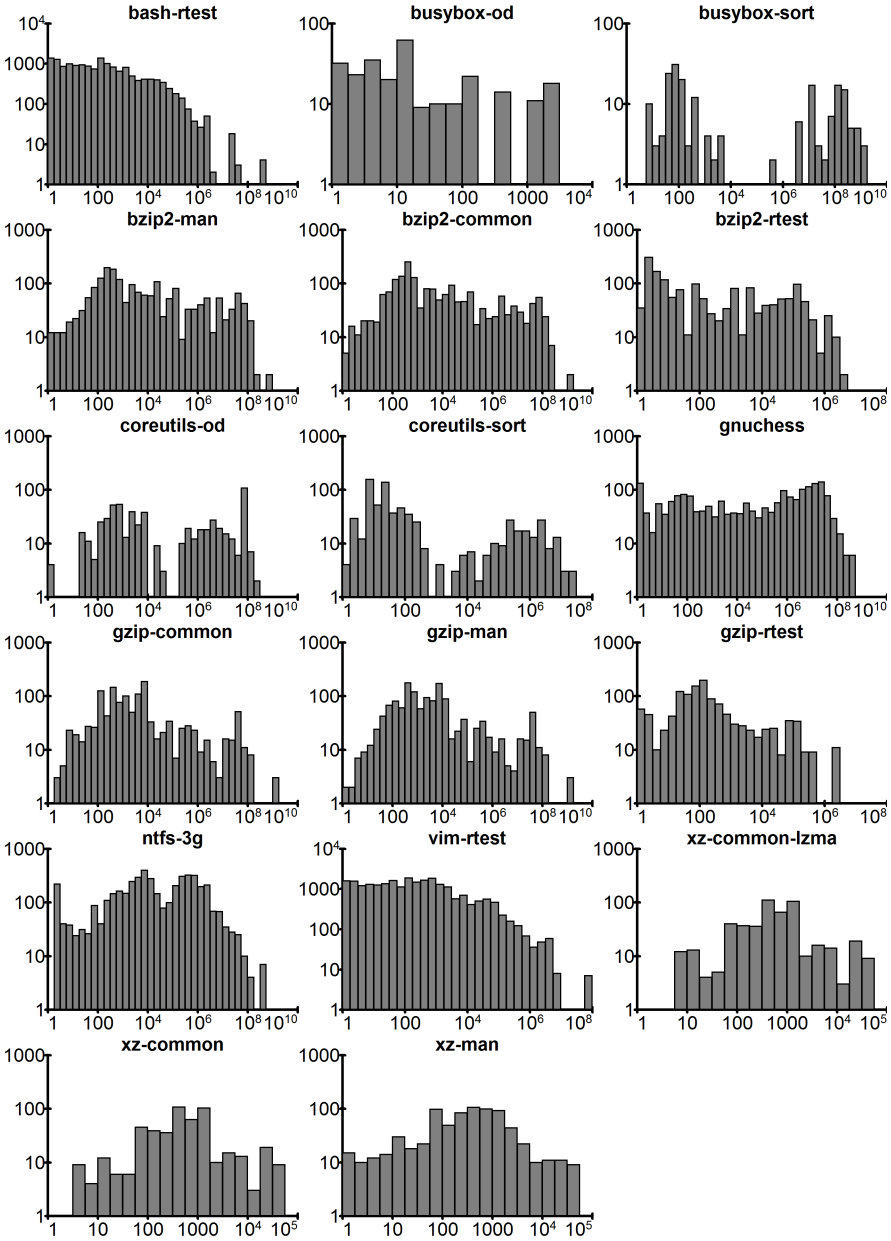

Figure 2.5: Log-log histograms of execution count (median over 50 runs) per basic block; the x-axis shows the number of times a block was executed and the y axis how many basic blocks have been executed that often

idea of what the program is doing.

It is noteworthy that the exponents are all close to one, which is the minimum for

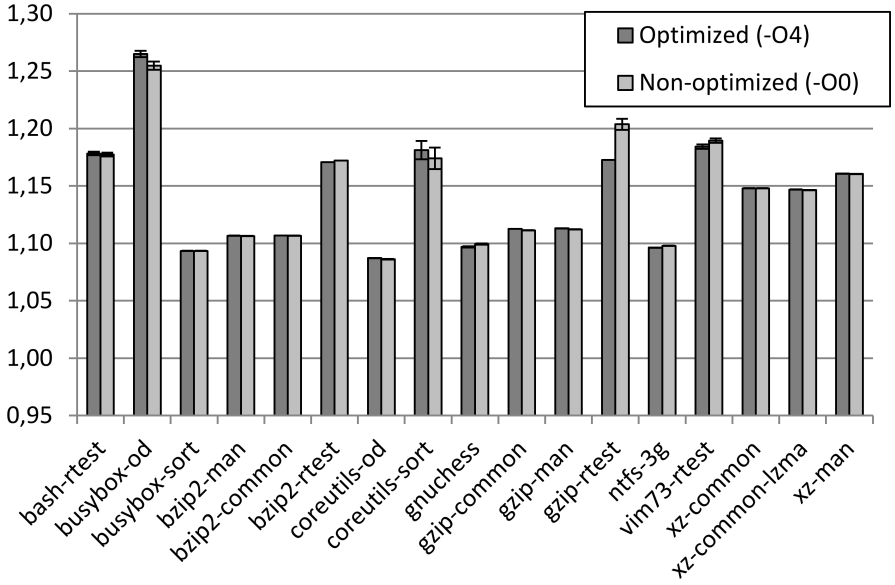

Figure 2.6: Estimation of the distribution exponent; lines indicate standard errors

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. 2.5. 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.

2.6.3 Relationship between execution count and coverage

Residual faults are activated only by a small fraction of the tests [96]. 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. 2.7 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 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. 2.7 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

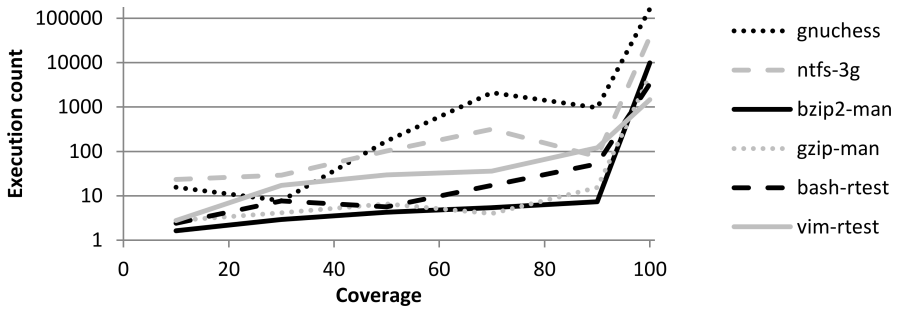

Figure 2.7: Geometric mean of maximum execution count per basic block depending on coverage

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.

2.6.4 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. 2.8 shows the average number of faults per basic block for each fault type, averaged over all 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 2.1). 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. 2.8 provide some indication of the differences in how often fault types occur between the various

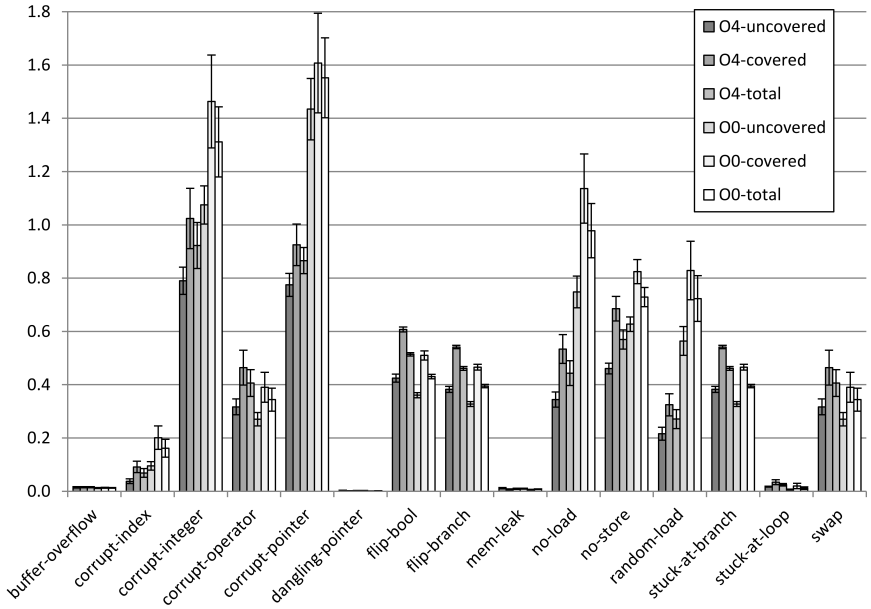

Figure 2.8: Number of faults per basic block for each fault type, distinguishing whether blocks are covered by the workload and whether the program is optimized (O4) or not (O0); the numbers are an average over all programs/workloads and the lines refer to standard errors

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. 2.8 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.

2.7 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 [41]), where this issue is far more pronounced [31; 47]. 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 2.1 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.

2.8 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 [96] 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.

2.9 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.

3

On the Soundness of Silence Investigating Silent Failures Using Fault Injection Experiments

Abstract

Fault injection campaigns have been used extensively to characterize the behavior of systems under errors. Traditional characterization studies, however, focus only on analyzing fail-stop behavior, incorrect test results, and other obvious failures observed during the experiment. More research is needed to evaluate the impact of *silent failures*—a relevant and insidious class of real-world failures—and doing so in a fully automated way in a fault injection setting.

This paper presents a new methodology to identify fault injection-induced silent failures and assess their impact in a fully automated way. Drawing inspiration from system call-based anomaly detection, we compare faulty and fault-free execution runs and pinpoint behavioral differences that result in externally visible changes—not reported to the user—to detect silent failures. Our investigation across several different programs demonstrates that the impact of silent failures is relevant, consistent with field data, and should be carefully considered to avoid compromising the soundness of fault injection results.

3.1 Introduction

Practice shows that producing software that is entirely free of bugs is not feasible. As software becomes more mature, the number of bugs approximates a linear function of software complexity [100]. Given that software complexity is, in turn, steadily increasing over time, software faults are naturally becoming more and more prevalent. To mitigate this problem, researchers have devised several different strategies to build fault-tolerant software systems. One approach is N-version pro-

gramming [14], which relies on multiple semantically equivalent versions to achieve software-implemented redundancy. Although this approach can tolerate several different classes of failures, the need to implement at least three versions of the same software is often deemed prohibitively expensive.

Other, more cost-effective approaches to deal with unreliable software rely on a predetermined failure model to implement fault detection and containment mechanisms. For example, crash recovery techniques [44; 107; 103; 80] restart individual components or computations when a fail-stop failure occurs. These techniques are based on two key assumptions: the system can detect that a failure has occurred and the underlying fault does not propagate outside the affected component before the failure is detected. If these assumptions are violated, recovery actions may fail to preserve the dependability of the system. To validate these assumptions, researchers traditionally rely on fault injection [67; 15; 90; 89; 49; 74; 112; 55; 99], a popular technique to evaluate the effectiveness of fault-tolerance mechanisms and characterize the behavior of a system under errors. Most studies, however, limit their analysis to trivially observable failures. Traditional dependability characterization studies [8; 58; 39], for instance, focus only on fail-stop behavior and other high-level properties directly exposed to the user. More research is needed to evaluate the impact of *silent failures* in fault injection experiments. We define silent failures as a situation where a fault causes externally visible behavior to deviate from normal behavior (which makes it a failure) but with no clear indication of failure such as an error exit status, a segmentation fault, or an abnormal run time (which makes it silent). This scarcity of research is surprising, given that silent failures are a relevant fraction of real-world bug manifestations [42] and also known to introduce insidious errors that can completely compromise the dependability of a system or the effectiveness of its fault containment mechanisms. As an example, a silent failure introduced by a seemingly innocuous software update has been reported as "*one of the biggest computer errors in banking history*", leading to the system mistakenly deducting about $15 million from over 100,000 customers' accounts [53].

Assessing the impact of silent failures in a fault injection setting is more than a simple academic exercise. If their impact is found to be marginal, researchers may need more sophisticated fault injection tools to accurately emulate this important class of real-world failures. If their impact is found to be significant, on the other hand, characterization studies ignoring silent failures may undermine the soundness of the results. Furthermore, the ability to identify silent failures may play an important role in the design of fault injection campaigns. For instance, prior studies argued that faults that are activated during testing but do not result in directly observable failures are more representative of *residual faults* (faults that are actually experienced in the field) and suggested a strategy to eliminate irrelevant fault injection runs [95; 96]. In this context, it is crucial to distinguish between residual faults introducing silent failures and other faults introducing nonsilent failures that are only triggered by a specific set of conditions. This distinction makes it possible to better formulate the purpose of the experiments and interpret the results correctly.

This paper presents a new automated methodology to identify and assess the impact of silent failures in fault injection experiments. Our goal is to shed some light on the relevance of these failures in an experimental setting, determine under which circumstances they are most likely to occur and improve the general understanding of what "*silent*" behavioral changes an artificially injected fault may introduce in a system. To this end, our approach is to track programs' externally visible behavior (exposed through system calls), their run time, and their exit status. To identify the relevant behavioral changes induced by fault injection, we perform both a fault-free reference run and an experimental run with faults injected in a controlled setting—with a predetermined workload and fault load [96]. The controlled setting allows us to log all the relevant events and abstract away any irrelevant differences introduced by the environment. This approach makes it possible to compare the behavior of the two runs and identify relevant deviations using a simple system call matching strategy. In contrast to prior work on anomalous system call detection [92; 93], our strategy considers only externally visible behavioral deviations between isomorphic execution runs to conservatively identify all the relevant differences. In contrast to prior work on real-world bug characterization [42], our strategy allows us to reason on the outcome of the fault injection experiment (i.e., nonfailure, silent failure, other failures) in a fully automated fashion, opening up opportunities for large-scale failure analyses.

The contribution of this paper is threefold. First, we describe a new automated methodology to identify silent failures in fault injection experiments. Our methodology is of general applicability and can automatically identify several classes of failures from the outcome of a given experiment. Second, we present an implementation of our methodology for user-space programs. Our current prototype runs on Linux, but can be easily extended to other UNIX operation systems that provide similar tracing functionality. Finally, we have applied our methodology to evaluate the impact of silent failures across several different programs and artificially injected fault types and fault locations. Our results demonstrate that the impact of silent failures is relevant, reflects field data, and should be carefully considered in dependability benchmarking scenarios.

3.2 Approach

To determine how common silent failures are, we perform fault injection experiments to introduce artificial but realistic software bugs into a number of popular open source programs and analyze the impact of the injected faults on the externally visible behavior of those programs. We have chosen to use fault injection rather than real bugs because this allows us to perform a far larger number of experiments, suitable for statistical analysis. The main steps we have taken are to: (i) inject realistic software faults into the target program, (ii) log the externally visible behavior of the program when subject to a predetermined workload, and (iii) compare the re-

Table 3.1: Fault types

corrupt-index	off-by-one error in array index
corrupt-integer	off-by-one error in integer operand
corrupt-operator	replace binary operator with random operator
corrupt-pointer	replace pointer operand with random value
flip-bool	negate result of boolean operation
flip-branch	negate controlling value for conditional branch
no-load	load zero instead of intended value
no-store	remove store operation
random-load	load random number instead of intended value
stuck-at-branch	fixed controlling value for conditional branch
swap	swap operands of binary operation

sulting logs against the behavior of a fault-free reference run while preventing false positives due to nondeterminism. This section illustrates these steps in detail.

3.2.1 Fault injection

We use the EDFI [47] framework to perform fault injection. This system is based on the ability of the LLVM compiler framework [79] to support plug-ins that manipulate intermediate (LLVM IR) compiler code. The EDFI plug-in operates on the intermediate code before any compiler optimizations are performed, so it has almost as much information as systems working directly on the source code, but is more easily portable due to its integration with LLVM. The only loss of information is the fact that preprocessor macros are already expanded in the intermediate code, which means that faults injected at the macro level cannot be directly supported.

One important step when designing a fault injection experiment is to decide which types of faults are to be injected. Ideally, these fault types should be as similar as possible to real bugs introduced by human programmers. A number of investigations on which types of faults are most commonly encountered can be found in the literature [41; 27; 110]. The fault types injected by our program have been selected based on these papers and are listed in table 3.1.

When injecting faults, for each run we inject a single fault that we know will be activated by the workload. Doing so eliminates runs known not to trigger the fault, similar to the fault acceleration strategies proposed in prior work [115; 62]. We use the EDFI framework to count how often each part of the code is executed and have it write out a map file during compilation that lists all *fault candidates*. We define a fault candidate as the combination of a code location and a fault type that can be injected at that location. Execution counts are tracked per basic block, which is a part of the code with a single entry point and a single exit point. We perform a reference run which yields execution counts and randomly select fault candidates from the set of fault candidates in the map file that are in basic blocks with nonzero

execution counts.

We only inject a single fault at a time because simultaneous injection of multiple faults is less controllable. Execution of the first fault potentially influences whether any subsequence faults are executed and potentially even changes their effects. While interactions between faults are also an interesting field of study, currently little is known about the relevance of silent failures in fault injection even for the simpler case of a single fault per run. A single-fault strategy makes it easier to analyze the results and ascribe the observed behavior to the injected fault.

3.2.2 Program behavior

To log the externally visible behavior of the program while running a workload, we use the `ptrace` system call on Linux. This call allows the interception of system calls before and after they are performed. By logging system calls rather than just comparing the expected output with the logged output, we can identify many more cases of deviant behavior. Suppose, for example, that the `gzip` program is run with the `-k` flag that specifies that the input file should not be deleted. In this case, checking the contents of the output file is not sufficient, because some faults could cause the flag to be ignored and the input file to be deleted. Since it is impossible to predict what a program will do when faults are injected, all externally visible behavior must be monitored to be able to detect any possible failure.

An important question is which calls are externally visible. Our intuition here is that, for example, a successful call to `unlink` would affect other application programs while a call to `getpid` has no impact whatsoever. We consider system calls externally visible only if they can potentially affect the values returned by system calls performed by unrelated processes. We consider two processes related if both either are the root process started by our tracer or descend from it. For example, a `write` to a file counts but a `write` to a pipe shared only with a child process does not. Information from the `/proc` file system is not considered externally visible. It is not normally used in application programs and including it would make every memory write externally visible. Writes to memory shared with unrelated programs do count, although in our test set there were no cases of this happening. We ignore timing in the sense that, for example, we do not consider a `sleep` call to be externally visible. Ordering of externally visible behavior, on the other hand, does count. For example, if an old file is deleted and a new one created, swapping the sequence of the operations would be considered an externally visible difference. For reproducibility, we have made available a full list of externally visible system calls at [4].

3.2.3 Comparing logs

The major issue when comparing the logs is nondeterminism. The main source of nondeterminism is multi-processing and multi-threading. Our log includes externally visible system calls made by all child processes and threads, tagged with the

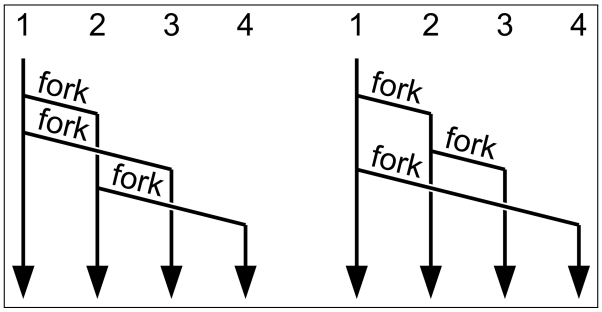

Figure 3.1: Scheduling of fork resulting in different pids

calling process or thread. As a result, system calls are interleaved randomly. To solve this issue, we compare the log for each subprocess or thread separately, ensuring that interleaving does not interfere with system call matching. As a consequence, some ordering information is lost, but it is essential in making the logs comparable between different runs. A more subtle issue is naming. To compare the processes and threads between different runs, we need to assign names to each that are consistent between runs. There is no such guarantee with the pids provided by the operating system, so we provide our own process naming scheme. A sequential scheme, such as the one provided by Linux PID namespaces [18] is not sufficient here. Figure 3.1 shows the case of a process forking twice and its first child forking once. Numbers are assigned based on the sequence in which the fork calls are scheduled. For example, the grandchild is either number 3 or 4 depending on which process forks first. To address this, we assign hierarchical process names. We use the name `r` for the root process (pid 1), `r.1` for its first child (pid 2), `r.2` for its second child (3 on the left, 4 on the right) and `r.1.1` for its grandchild through `r.1` (4 on the left, 3 on the right). Our naming scheme allows the processes to be reliably matched between runs, regardless of scheduling.

There are some other sources of nondeterminism we have to control to be able to match the logs correctly. The most obvious one is time. Many programs write the current time in their output or use it to initialize random seeds. To prevent time from introducing differences, we intercept calls that read the clock and the `rdtsc` instruction to make them return a virtualized time. This time is initialized to a fixed value when the tracer starts and only incremented when the time is read. It is inherited by child processes and threads but not shared afterwards to prevent the introduction of more scheduling-dependent behavior. The pid is also sometimes used in externally visible ways, such as for pid files and as a random seed. For this reason, we virtualize pids using a hierarchical scheme similar to the naming system described before, encoded in a `pid_t` value. We use six bits for the top level child index and four bits for each level below. The highest-order (sign) bit is left untouched because some system calls use negation of pids. The second highest-order bit is used to distinguish virtual pids from real pids, because Linux never assigns such high pid numbers. Although

`ptrace`-based pid virtualization means that we have to intercept and modify all system calls that take pids as parameters or return them, there is little performance loss as we already have to intercept all system calls. Compared to, for example, Linux PID namespaces [18], our approach has the advantage of not requiring specific privileges, which makes the approach safer to use and easier to deploy. In addition, we virtualized the `/dev/random` device to provide deterministic random data. The position in this stream is inherited by subprocesses but never shared to avoid nondeterminism. One final source of nondeterminism is the port on which connections are accepted by the `accept` call in the web servers. Because these do not affect any other behavior, they need not be virtualized but can simply be filtered out of the log.

Having dealt with the relevant sources of nondeterminism, comparing the externally visible system calls performed by the faulty program with the fault-free reference run is as easy as invoking the `diff` tool for each subprocess log. We logged each system call to a single line to make determining which calls are different as simple as parsing them from the `diff` output. Wherever relevant, any data pointed to by the parameters is included, such as the data written in case of the `write` call. To prevent logs from getting excessively large, write buffers are hashed. Pointers passed to system calls are never logged directly, because they might change between runs and are not relevant to other processes. Performing system call matching offline using a simple tool such as `diff` is much simpler than many other systems and has the advantage of allowing all behavioral differences to be revealed without heuristics because all the necessary information is available. Prior work, in contrast, typically requires far more complex approaches [121].

Overall, we identified all the sources of nondeterminism that were an issue for our test programs (and workloads) and successfully eliminated them. We verified this by running fault-free runs at least 256 times for each program/workload combination to check for unexpected differences between the logs. The 16 fault-free runs performed as a part of each injection experiment were also checked each time. We do not claim that our system would be able to tackle any possible source of nondeterminism, but we believe that our approach is effective for a broad class of programs, given the diversity in the programs we tested. To be able to deal with more difficult cases, fully deterministic record-replay techniques [108; 9; 102; 52; 109; 77] would be needed. Because we were able to deal with the nondeterminism present in our test programs, we can compare logs from faulty runs directly with those from fault-free runs, allowing us to identify all the externally visible failures.

3.2.4 Silent failures

We have described how to detect failures in terms of deviant externally visible behavior. The next step is to decide which of these failures are silent. Our approach is to consider whether the exit status and the run time are anomalous. Exit status refers to the value written into the `stat_loc` parameter when the parent which invoked the program uses the `waitpid` system call. For correct runs, this value should normally

indicate that the program exited with exit status zero. However, it also allows the program to indicate to its caller that an error occurred by specifying a nonzero value. In case the program is terminated by a signal, for example due to a segmentation fault, the status code indicates both the fact that the program was killed and the number of the signal that killed it. We compared the exit status of the faulty runs with the exit status of the fault-free reference runs, because in some cases a nonzero exit status can legitimately be returned (for example, the `diff` program returns 1 to report differences between the input files). There were no cases where the reference run was killed by a signal. Hence, an exit status indicating death by signal is always a clear sign for the caller that something went wrong, making the observed failure nonsilent.

In addition to the exit status, we also consider the run time as a method of detecting failures. Programs that exit very early or take an excessive amount of time are a clear indication that something went wrong. We defined the run time as the real time the caller would measure from the `exec` call to launch the program to the `waitpid` call that confirms that the program has exited. The time taken is standardized using only mean and standard deviation of the run time for the fault-free reference runs. The standardized time gives a good indication of the degree to which the run time is anomalous. We decided on a cut-off point of four standard deviations based on our measurements. The details of this choice are described in Section 3.4, which discusses the analysis of our measurements.

3.2.5 General applicability

We focus on legacy applications written in low-level languages and our tools can easily be used with other programs written in languages for which an LLVM front-end is available, including C and C++ as well as many others. To achieve this, the only change that needs to be made is to use the LLVM compiler and set the flags to generate bitcode. This is usually a matter of running the configure script to change the compiler settings and then recompiling. Our tools are built for Linux, but could easily be adapted to other POSIX-based platforms by changing the tracer to use the appropriate `ptrace` alternative. For non-POSIX systems such as Windows, the system calls would need to be reclassified based on their external visibility. For programs written in higher level languages, specific tools as well as a more appropriate set of fault types would be needed. However, the general approach is still equally applicable.

3.3 Programs and workloads

Our aim in selecting programs to test with has been to on the one hand have a diverse set of programs while on the other hand also having a few sets of similar programs. Diversity in terms of size, complexity and type of work done makes the results applicable to a wider set of software. Having sets of similar programs that

can run the same workloads allows us to determine whether there are any patterns in the variation of the results. This approach makes it possible to distinguish between properties that apply to a specific class of programs and whether the results are more affected by the implementation of the program itself or by the selected workload. We define workload here as a fixed sequence of invocations to the program being analyzed, which in practice is defined by the script performing these invocations. We preferentially chose programs that offer their own regression test suite to have a 'neutral' workload but constructed our own workloads for programs that do not offer regression tests.

The first set consists of the compression programs `bzip2`, `gzip` and `xz`. Although the programs use different algorithms, they essentially perform the same function and are invoked in the same way. Both `bzip2` and `gzip` provide a small regression test set, both of which we included in our testing. In addition, we used the manual pages of these tools to construct a workload that is similar for the three programs and provides more coverage than either of the regression tests. Our workload performs 500 iterations, performing zip, test and unzip operations on each iteration. Arguments are randomly combined from the available ones listed in the manual page. Input files are also randomly generated, based on a Markov chain approach. The transition matrix is randomly chosen from a number of matrices representing different types of files, including both text and binary types. The intermediate zipped files are sometimes randomly corrupted to also invoke some of the error handling code in the programs tested.

Two other sets are the `od` and `sort` utilities taken from the Busybox and Coreutils projects. Busybox is normally compiled into a single binary containing all tools, but we configured it to provide each tool as a separate binary. We constructed workloads for these tools using a similar approach as described for the compression utilities. However, the arguments to provide are based on the POSIX specification (which covers both tools) and are therefore exactly identical between the two implementations of both programs. For `sort`, in addition to the randomly selected command line arguments, we also specify a random language and include input files in several languages to exercise more of its capabilities.

We selected `bash` and `vim` because they are considerably more complex than the previously mentioned programs and both include extensive regression tests. In addition, we expect the control flow to differ from the other programs because `bash` performs complex parsing and `vim` is more interactive than the other programs. In the case of `bash`, we selected the fastest half of the subtests to be able to perform a sufficient number of fault injection experiments to be statistically meaningful in the time available.

To also include some long-running programs, we added two HTTP servers, namely Apache `httpd` and `nginx`. We tested both HTTP servers with the Apachebench benchmark [1] (AB), configured to perform 1000 requests. One issue with these programs is that they consist of a parent process and a number of worker processes. Therefore, the parent process can deal with any failures occurring in the workers and

recovery strategies could be implemented at that level. For this reason, we consider not just the exit status of the root process started by our tracer, but also the exit status of any workers that die while the benchmark is running. This information makes our analysis more conservative as it results in fewer failures being considered silent. Another consideration was that error codes being logged could be considered a non-silent failure mode. We analyzed the logs and found that errors are logged in only very few cases, not enough to influence the analysis. Specifically, `httpd` logged two "403 Forbidden" errors, one "404 Not Found" and one "500 Internal Server Error", while `nginx` logged three "400 Bad Request" errors.

To achieve determinism in the externally visible behavior, we had to make some very minor changes in three of the programs described. In `bzip2`, a field written to a file was not initialized, causing the value to be different between runs. This problem was fixed by always initializing the field to zero. The `gzip` regression test randomly generates a directory name to save output files to. Since the regression test script is not run by the tracer (unlike `gzip` itself), time virtualization did not make this name deterministic. We modified the script to always use the same name. The `xz` program prints an unterminated string to the error output under certain conditions. We fixed the program to always terminate the string. With these small modifications, all programs and workloads ran sufficiently deterministically to allow for the comparison described in the previous section. These changes remove what would otherwise be spurious differences. Note that this does not affect the validity of the experiment because writing uninitialized data is always a bug that requires fixing anyways while in the `gzip` case the change in the test script does not affect program behavior.

3.4 Results

We have run each workload for each program 16 times without injecting faults and 256 times with a single injected fault. The runs without faults serve as a reference for calls performed, exit status and timing. They have also been compared with each other to ensure that there are no false positives due to nondeterminism. Faults to inject have been selected randomly from the set of candidate faults that were activated in the reference runs, ensuring that all faulty runs result in activation of the injected fault. This approach also means that the injected faults are representative of activated candidate faults. Hence, fault types that can be injected in more different executed code locations are represented proportionally more often. This choice is based on the intuition that mistakes that can be made in more places are likely to be made more often.

All experiments were performed in a Ubuntu 12.04.3 LTS virtual machine with 3GB of memory running a 32-bit x86 Linux 3.8.0-29 kernel. For virtualization, we used QEMU 1.6.0 with KVM acceleration enabled. The host machines used CentOS 6.4 with a 64-bit x86 Linux 2.6.32-358 kernel. A new virtual machine was started for each individual experiment to ensure that the context is exactly the same every

Table 3.2: Coverage of test programs

program	workld.	% of basic blocks	% of fault cand.
bash	rtest	44.3%	46.4%
bzip2	doc	71.2%	83.1%
bzip2	rtest	60.2%	77.5%
gzip	doc	41.1%	52.3%
gzip	rtest	25.3%	31.6%
httpd	ab	17.9%	19.3%
nginx	ab	22.4%	23.8%
od (bb)	doc	45.0%	55.9%
od (cu)	doc	35.9%	44.5%
sort (bb)	doc	34.4%	45.5%
sort (cu)	doc	30.3%	28.6%
vim	rtest	47.8%	53.5%
xz	doc	60.6%	63.6%

time. To reduce interference that might make the time measurements unreliable, we never ran more than one virtual machine on the same host machine at the same time.

It should be noted that the number of times we have run the workload is not the same as the number of times the program has been invoked. The number of times the tested program is invoked differs between workloads. On the low end, there are the HTTP servers `httpd` and `nginx`, both of which are started once and continue to serve requests until after the workload script has been completed. On the high end, there are the compression programs `bzip2`, `gzip` and `xz`, which are invoked by our workload many times to test different types of files and combinations of arguments. For example, each workload run of `gzip` represents 1870 invocations of the program.

It should also be noted that the number of times the program is invoked by the workload script is not the same as the number of processes. Each program may use the `fork` or `clone` calls to create new processes and threads. These are tracked by the same tracer instance so that we can consider the end result of the whole as perceived by the script that invoked the program. In most cases, we consider the exit status reported by the root process; that is, the process that was created directly from the workload script. The idea is that any other exit codes are internal to the application and the caller never learns about them. However, we made an exception for the HTTP servers `httpd` and `nginx`, where we also consider the exit codes of the worker processes. The reasoning here is that it is reasonable to expect that these servers themselves deal with failing worker processes, for example by logging errors and/or launching new ones. The results presented in this section are all per invocation rather than per process, with the HTTP servers including a summary of the behavior of the subprocesses of that invocation.

Table 3.2 indicates the levels of coverage we achieved. With regard to the workloads, rtest is the official regression test, doc means constructed based on documenta-

tion and ab means ApacheBench [1] has been used. Coverage numbers are provided both in terms of basic blocks and in terms of fault candidates. There is a clear difference between the two and generally coverage in terms of fault candidates is substantially higher. This means that basic blocks executed by the workload tend to be larger on average than basic blocks not executed. This seems reasonable if one assumes that error handling code is on the one hand relatively unlikely to be executed and on the other hand contains relatively many branches and hence smaller basic blocks.

The coverage numbers we reached are relatively low and could have been made higher by using symbolic execution to artificially create coverage-maximizing workloads [21]. However, we specifically chose these workloads to mimic the types of regression tests that would be used by software developers in practice. Artificial workloads are of little use in this case because in most cases they cannot easily be verified whether the program functions correctly for these inputs - in fact they often specifically aim to make the program fail to exercise error handling code. In addition, such workloads would not allow comparison between similar programs with the same workload as they are specifically tailored to a single program. Hence, we believe that the workloads we use are most suitable for our specific purpose despite their low coverage.

We cross-checked the coverage of basic blocks between runs and found that the only program where variation between runs is found is `httpd`. The potential impact is that faults may be selected that are not activated in all benchmark runs. Our analysis showed that this effect is minor, with more than 95% of the injected faults being activated.

In this section we will use the results of the tests described to compare the impact of a number of factors on silent failures. First, we consider to what extent there is a difference between the programs we tested and whether it is the program itself or the workload that makes a difference. Next, we consider how the different fault types behave with regard to silent failures. Finally, we consider whether the number of times a fault is activated makes a difference.

3.4.1 Differences across programs

With regard to timing, we have used the fault-free reference runs to estimate the mean and standard deviation of the run time for each individual invocation. We consider the run time to be anomalous if it is at least four standard deviations above or below the mean of the reference run times. Using two standard deviations would give 5.5% false positives and three standard deviations would give 1.7% false positives, while with four standard deviations there is not a single case in our reference runs that would violate the time constraints. The standard deviations are generally very low compared to the mean, allowing anomalies to be detected quite well. For example, in the 95^{th} percentile of the invocations (instances where program is started by the workload script) the standard deviation is only 3% of the mean runtime.

To get an impression of what happens in cases where programs show failure, we

Table 3.3: Number of failures per program/workload

program	workld.	total	silent-exit	silent-time	silent-both
bash	rtest	33.7%	14.7%	9.7%	8.0%
bzip2	doc	66.3%	32.1%	35.0%	23.1%
bzip2	rtest	57.1%	33.9%	34.6%	26.0%
gzip	doc	64.3%	24.5%	26.1%	17.5%
gzip	rtest	49.1%	7.2%	35.8%	7.0%
httpd	ab	51.5%	15.8%	14.9%	14.4%
nginx	ab	57.0%	13.3%	5.9%	4.7%
od (bb)	doc	64.6%	30.9%	23.1%	19.6%
od (cu)	doc	54.2%	22.9%	21.3%	17.8%
sort (bb)	doc	52.8%	14.0%	9.5%	5.2%
sort (cu)	doc	51.2%	10.2%	9.7%	5.0%
vim	rtest	29.7%	17.8%	24.1%	16.9%
xz	doc	46.1%	21.1%	23.0%	14.3%

have computed the frequencies of all combinations of calls that cause differences between correct and faulty behavior. By far the most common case is that the difference is only due to `write` calls. This accounts for 38.3% of the faulty invocations when weighing each program is equally. This case is also the hardest to detect, because there is no visible difference other than the incorrect output. In a further 17.6% of the cases, `open` and `write` calls make up the only visible difference. In a further 2.5%, `read` and `write` calls together make up the difference. It should be noted here that `read` calls are only externally visible when from sockets and pipes to unrelated processes. No other common combinations of just a few calls differing are particularly common; in most other instances there are many simultaneous differences.

Table 3.3 shows how many failures occurred and how many are considered silent according to different criteria. Each number represents a percentage of the program invocations in which the injected fault was activated. Although we only injected faults that are activated by the workload at least once, there are invocations in which the fault was not activated and these are excluded here. It should be noted that (as explained before) for `httpd` and `nginx` we consider the exit status of the worker processes as well as the main process.

The first interesting observation from Table 3.3 is that there are many faults that do not cause any externally visible deviations in behavior even when they are activated. The percentages suggest that the issue of nonfailure of activated faults is of a similar magnitude as the issue as nonactivation due to limited coverage. The different between programs is substantial. This means that, for example, to test a similar number of failures one would have to inject more than twice as many faults in `vim` as in Busybox `od`. Unlike coverage, this issue rarely receives any attention in traditional fault injection campaigns, which typically only strive to provide reasonable fault activation guarantees [115; 62]. When comparing between programs

and benchmarks, what stands out most is that the number of failures is very low for `bash` and `vim`. These programs perform relatively complex processing compared to the others as they implement many different functionalities. Busybox `od`, `bzip2` and `gzip`, on the other hand, have relatively high failure rates. These programs linearly process a single stream of input, always in more or less the same way. It seems plausible that there is more opportunity for a corrupted state not to be used again in case of the more complex programs, while the linear programs are using the same state over and over again. This means that studies which examine the impact of faults and recovery after faults for a single program (such as [49; 125]) should not be generalized to different classes of programs.

Having looked at the failure rates in general, we will now consider the number of silent failures. The "Silent-exit" column indicates what percentage of activated faults consists of failures that would not be detected by the exit status. "Silent-time" refers to failures that do not differ from the reference run time by more than four standard deviations. The "Silent-both" column refers to failures that cannot be detected from either exit status or run time. While performing checks on the exit status and run time allows at least half of the failures to be detected in almost all cases, it is also clear that each program has a substantial number of silent failures. The average over all programs is 13.8% of all activated faults resulting in silent failures, which, interestingly, seems to suggest high correlation with the fraction of faults introducing *latent bugs* according to findings discussed in prior work [42]. This number is high enough to say that any research involving fault injection should consider that a substantial number of faults might spread to other components (in this case through system calls) while not being easily detected. Care must be taken to detect these faults through their anomalous behavior. In addition, any research performing state recovery based on the assumption that failures are usually fail-stop (such as [107; 103]) should consider the implications of silent failures.

Which approach is more effective at detecting failures differs strongly between programs. It is clear that different detection mechanisms are effective for different programs. Many failures triggered by the `gzip` and `vim` regression tests can be detected from the exit status, while it is relatively uncommon for failures in these programs to have a large effect on the run time. For `bash` and Busybox `sort`, on the other hand, run time is a better detection mechanism. There is no obvious pattern in which programs and workloads are most like to have many silent failures. The numbers for `od` and `sort` are very similar between the Busybox and Coreutils implementations, which may be due to the fact that both implement the exact same functionality and run the same benchmark. However, when considering the compression programs using the same benchmark, it is clear that `bzip2` has more silent failures than `xz` does. This might have to do with the fact that `xz` implements a more advanced algorithm and is hence more complex, the same reasoning as for the failure rate in general. Still, it is conceivable that programming style also plays a substantial role here. A program that contains many checks and assertions, tests each return code and exits whenever anything is wrong would be much less likely to have any silent failures.

Table 3.4: Number of failures per fault type

fault type	total	silent-exit	silent-time	silent-both
corrupt-index	72.5%	48.7%	55.4%	42.7%
corrupt-integer	42.3%	24.7%	22.4%	17.8%
corrupt-operator	40.4%	21.3%	18.6%	12.7%
corrupt-pointer	92.1%	20.3%	7.7%	3.1%
flip-bool	61.9%	29.1%	25.6%	16.4%
flip-branch	54.3%	27.4%	23.7%	17.3%
no-load	48.1%	32.3%	25.4%	21.0%
no-store	44.2%	15.7%	12.9%	7.8%
random-load	55.4%	23.2%	17.0%	13.3%
stuck-at-branch	38.3%	19.5%	17.8%	11.9%
swap	12.0%	4.0%	5.6%	3.4%

The main conclusion from our experiments comparing programs is that silent failures occur in sufficient numbers to be a serious threat in fault injection experiments. We have not been able to pinpoint the source of variation between programs, but it seems credible that coding style is a big factor. Error checks and consistency checks (including assertions) should be able to make some silent failures nonsilent. In addition, we found that considering just activation of faults is not enough because many activated faults do not result in deviant behavior, especially in more complex programs.

3.4.2 Differences across fault types

Usually fault injection experiments consider a variety of mistakes commonly made by programmers (for fault types used by us, see Table 3.4). It is reasonable to expect that different types of faults result in different program behavior. Table 3.4 shows the percentage of activated faults resulting in failures per fault type, with each program weighed equally. The differences in the likelihood of failure are very large. We will discuss the fault types that stand out here.

The "corrupt-pointer" fault type stands out for almost always causing deviant behavior when activated. This is easily explained by the fact that most random pointers point into unallocated memory, causing a segmentation fault when they are dereferenced. This is consistent with the fact that many failures triggered by this fault type can be detected by exit code, either because of being killed by a segmentation fault or by returning an error exit code after a segmentation fault is caught or detected in a child process. Even more cases are detected based on run time. Hence, although an activated "corrupted-pointer" fault is very likely to result in failure, this failure is very unlikely to be silent.

The "swap" fault, on the other hand, stands out for resulting in very few failures. The most likely reason here is that some of the most commonly used operators, such as `+`, `*`, `==`, `!=`, `&` and `|` are commutative so that swapping the operands has no effect.

Potential effects for the other operators are quite diverse, including incorrect values, buffer over- or underflows, divisions by zero or `NULL` pointer dereferences. Many of these cases are caught by exit status, which suggests that the more dramatic results are quite common for the noncommutative operators.

Another interesting fault type is "corrupt-index", where activated faults are most likely by far to result in silent failures. This is caused in part by a high failure rate in general, but it also has the highest proportion of silent failures of all fault types. Since this fault type is an off-by-one error, it is likely to cause small deviations in buffers and minimal buffer overflows. Unless this happens to affect another variable used as a pointer or as an index or it overwrites a string terminator, segmentation faults are relatively unlikely to result (only 5.8% of the cases). Hence, the relatively few cases where it is detected from the exit status are most likely due to consistency checks and assertions.

Summarizing, it has been shown that different fault types differ greatly with regard to the likelihood to cause failure and the ease with which failures can be detected. In both senses, experiments injecting pointer-related faults are much easier to perform and control than experiments with data-related faults. Branch-related faults are somewhere in between, often triggering silent failures but having more potential of being uncovered by anomaly detection systems.

3.4.3 Impact of ease of reachability

We have now considered differences between programs and fault types with regard to the occurrence of silent failures. The final factor we consider is fault location. One particularly important aspect of fault location is which locations are easy to reach while testing and which locations are harder to reach. Considering for example the compression programs, we would say that the main compression loop is easy to reach as any test that does not cause the program to fail early (for example by specifying invalid arguments) would reach it. Some other parts of the code execute in some operation modes but not others. Those are considered moderately easy to reach. There are many error handlers, on the other hand, that only execute under a very specific set of conditions. These are the locations we consider to be harder to reach. In addition, there is the question of how likely a fault is to cause failure when it has been activated. This is a similar idea to being in a hard-to-reach location if we consider it hard-to-reach in an input space. A typical example to illustrate this is a buffer overflow. Even if the code location of the bug is easy to reach (for example in the main loop), an input size must also be found that is sufficiently large to overflow the buffer into some relevant state while not being so large as to fail input size checks earlier on. An overflow of a small buffer is therefore easier to reach in the input search space than an overflow of a large buffer. Because hard-to-reach bugs are more likely to escape testing, it is important to know whether they suffer as much from silent failures as other fault locations.

To determine which fault locations are hard-to-reach, we consider the fraction

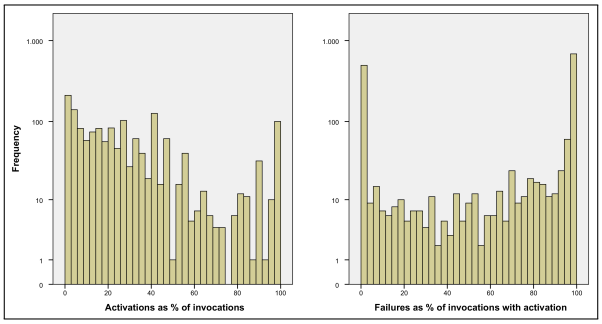

Figure 3.2: Histograms of fault activation and failure ratios; frequency refers to the total number of runs for all programs/benchmarks in that bracket

of invocations in which they are activated (reachability in code space) or in which they result in failure when activated (reachability in input space). A histogram of both variables is shown in figure 3.2. It should be noted that the HTTP servers have been excluded here because they only involve a single invocation. All other programs have been weighed equally. The "frequency" axis specifies the number of injected faults in the bins on the x-axis. It is clear that for both issues, there is great diversity between fault locations. Some faults are (almost) always activated while other faults are activated only in a few test cases. Similarly, while many faults either always cause failure or never cause failure when activated, there is also a substantial number of faults that only cause failure in a part of the cases.

For our further analysis, we classified both the activation and failure variables in three groups: <20% is considered hard to reach, $\geq$ 80% is considered easy to reach and the remainder is considered moderately easy to reach.

Table 3.5 shows the relationship between reachability and hidden failures. The most interesting result is found when considering the likelihood of activation. Faults that are activated only in a few of the test invocations, suggesting that they are relatively hard to reach in a test set, do not only result in failure more often but are also especially likely to result in hidden failures. This suggests that more extensive error checking is in place in code locations that are often used, exposing failures that would otherwise remain hidden. However, it is known that especially those code locations that are rarely used are more likely to contain bugs [95; 96]. This means that developers need to spend more effort into extending their regression tests to cover a larger part of the code and that they should perform more sanity checking especially in those regions that are not often used and hence more likely to contain bugs.

With regard to faults that are relatively unlikely to result in failure when activated, table 3.5 shows that, although (by definition) a relatively small fraction of those result in failure, a relatively large fraction of those failures remain silent. This supports our argument that considering whether or how often faults are activated is not enough to identify which faults are likely to escape testing. There is also a class

Table 3.5: Number of failures per reachability class

reachability class		total	silent-exit	silent-time	silent-both
Activation	<20%	62.1%	36.1%	33.9%	26.6%
	20-80%	56.9%	26.1%	23.2%	17.6%
	$\geq$ 80%	57.0%	14.5%	26.5%	10.5%
Failure	<20%	1.5%	0.9%	0.9%	0.7%
	20-80%	52.6%	34.0%	27.6%	21.6%
	$\geq$ 80%	97.7%	36.1%	42.2%	25.6%

of faults that can be triggered often but only results in failure under very specific conditions. The fact that these faults are also relatively likely to be silent supports the idea that fault injection experiments should not focus on just activation of faults, but also consider how many faults result in behavioral differences.

3.5 Threats to validity

In this section, we consider various factors that may have interfered with our experiments and the extent to which they influenced the results.

Although most of our data was gathered outside the faulty program by our tracer, the number of fault activations was determined from the per-basic block execution information gathered by EDFI inside the faulty program. The choice to take this approach was made because externally logging such events would be prohibitively expensive since we counted execution of every basic block. It is therefore possible that faults overwriting random memory could have interfered with activation counts read from the faulty program's memory by the tracer. The worst thing that could happen in this case would be an activated fault resetting the activation count to zero. If the fault were to result in a failure, this would be visible as a failure without fault activation. This did not happen a single time in our experiments. We think it is therefore safe to assume that if the activation count was reset in cases without failure, it would at most affect an insignificant number of runs and have no impact on the results.

Another concern is the faulty program interfering with the tracer itself. Because there is no shared memory between them, the only way to interfere would be through system calls, which are monitored. To prevent interference, we have the tracer check the arguments of risky system calls against a white list. The system call is canceled with error code `EPERM` for system calls that modify resources not on the white list, such as opening a file or sending a signal to an external (untraced) program. We manually expanded the white list to the point where no calls need to be canceled and because there are no resources on the white list that would affect the tracer, we know that it is safe from interference.

Another potential issue is the use of virtualization, which could potentially interfere with the times measured. To minimize this effect, we only run a single virtual

machine per host at a time. In addition (as described in the Section 3.4), we found that there were no extreme deviations in run time in the reference run, which suggests that we successfully mitigated this issue.

Although our approach has the advantage of allowing detection of any externally visible failure because system call logs can be compared to reference runs, it also comes with some limitations. Ordering information between processes or threads is lost. This is desirable in most cases because it prevents scheduling from introducing false positives, but it could miss failures caused by incorrect interactions between processes. Suppose, for example, that process A performs action X and process B performs action Y. It could be the case that process X has to be performed before action Y, for example if the former is the deletion of an old log file and the latter is creation of a new one. In this case, processes A and B need to use some form of interprocess communication, such as signals, semaphores, pipes or shared memory, to enforce the correct order. If a fault causes Y to be performed prematurely, it would be a failure that our system call matching approach would not be able to detect. However, if programs are to be studied that perform this kind of behavior, it would be possible to address this issue by identifying synchronization points between threads and processes. If the synchronization point is marked in the logs for both processes, reordering with respect to the synchronization point would be detected and race condition-inducing failures would also be found. Alternatively, for particularly complicated cases it would be possible to complement our analysis with general-purpose deterministic record-replay frameworks [108; 9; 102; 52; 109; 77] to address this issue.

3.6 Related work

Fault injection is the de facto standard technique for system dependability benchmarking. Its versatility and relatively low implementation costs have helped many researchers assess the dependability of several classes of systems, spanning from distributed [67] and local [15; 90; 89] user programs to operating systems [49; 74], file caches [99], and device drivers [112; 55].

Much prior work in the area focuses on the development of general-purpose fault injection tools. A common implementation strategy is to introduce program mutations that mimic realistic software or hardware faults. Mutations have been applied at the source level [56; 127; 96], at the binary level [65; 99; 41], or, more recently, at the intermediate code level [106; 113; 47]. An alternative is to introduce program mutations at runtime, using software and hardware traps [114; 65; 23; 67] or library interposition mechanisms [90; 89].

Until recently, however, there has been little attempt to investigate the general properties of fault injection and its impact on the running system. A number of studies evaluate the ability to inject *realistic* and *representative* software faults, typically focusing on *what to inject* [88; 41] *where to inject* [95; 96], and *how to inject* [28]. This is useful to reliably draw general conclusions from experimental results. Other

studies investigate how to improve fault activation guarantees, typically injecting faults into hot code spots identified by program profiling [115; 62]. This is useful to eliminate invalid runs with no faults activated and ultimately improve the efficiency of large-scale fault injection experiments. The latter is also the goal of efficient fault exploration strategies [89; 15], which rely on domain-specific heuristics to drastically reduce the number of fault injection runs to the interesting cases. While useful to analyze and improve the general properties of fault injection, all these studies reveal little insight into its impact on the system behavior.

Other studies have sought to analyze the impact of artificially injected faults on a running system, but without attempting to fully characterize its behavior—and thus unable to thoroughly investigate the impact of silent failures. Traditional characterization studies, for instance, solely focus on fail-stop behavior [8; 58] or high-level properties directly exposed to the user view of the system [39]. More recent studies investigate how faults progressively propagate throughout the system [125; 94]. The typical strategy is to rely on taint analysis techniques to identify all the corrupted portions of the internal system state. While able to expose some failures that do not normally result in fail-stop behavior during the experiment, this strategy cannot alone pinpoint behavioral changes that corrupt the external state of the system and eventually lead to subtle long-term failures. Given that prior work has demonstrated that fault propagation normally results in transient internal state corruption [94], we expect any strategy that ignores behavioral changes and external state corruption to heavily underestimate the impact of silent failures.

Relatedly, bug characterization studies have also attempted to analyze the system behavior in presence of real-world software faults. Most studies, however, do not specifically consider silent failures, but typically focus on fail-stop behavior [49; 86], fault propagation [49], or bug reproducibility [104; 24]. A notable exception is represented by the work of Fonseca et al. [42], which investigates internal and external effects of real-world concurrency bugs. Their notion of *latent bugs* is similar, in spirit, to our definition of silent failures in that they both lead to subtle long-term errors not immediately reported to the user. Their study demonstrates the substantial presence of silent failures in real-world bug manifestations and also confirms that they are most often induced by corruption of external system state—persistent on-disk state, in particular. When compared to our analysis, however, their investigation is limited to concurrency bugs, requires extensive manual analysis, and is based on a relatively small sample size. Our investigation, in contrast, is supported by automated analysis of fault injection results, which requires no manual inspection and naturally provides much more stable and general results.

To conclude, our work draws inspiration from prior work in different research areas. The general methodology used in our analysis is inspired by prior work comparing faulty and fault-free execution runs using state diffing techniques [99; 45]. Compared to our work, prior efforts differ in purpose—evaluating the effectiveness of fault-tolerance mechanisms—and scope—analyzing differences in the system state (and not in its behavior). Our system call-based behavior characterization

is inspired by prior work on anomalous system call detection [92; 93]. In contrast to prior work, our detection strategy compares semantically equivalent execution runs, naturally resulting in a simpler and more conservative behavioral analysis. Our detection strategy, in turn, is inspired by prior work comparing similar execution runs to detect deviant program behavior. In contrast to our work, N-variant systems [105; 35] compare semantically equivalent execution runs with different memory layout (to detect security attacks) and multi-version execution [116; 20], compares execution runs from multiple program versions (to perform online patch validation). Finally, our cross-execution system call matching strategy is inspired by recent mutable record-replay techniques [75; 121], which seek to deterministically replay a recorded execution on a different program version. In our work, however, mutability is solely induced by fault injection and system call matching is only operated after "replaying" the fault-free execution run. This drastically simplifies our matching strategy, which only compares completed execution runs and need not rely on the sophisticated heuristics proposed in prior work [121].

3.7 Conclusion

Based on our findings, we can now answer the research questions behind our investigation. First, it has become clear that silent failures are very common in fault injection experiments. On the one hand, this confirms that research based on fault injection experiments can be safely used to investigate the impact of real-world software bugs, where silent failures have been shown to be of similar relevance [43]. On the other hand, this also means that any research dealing with fault injection must carefully consider the impact of silent failures. Our results demonstrate that the widely adopted assumption that failures are generally fail-stop is hardly sound, as all the programs we investigated revealed a significant number of silent failures. When faults are activated but no failure is observed at the end of an experiment, it is important to assess the effectiveness of the adopted failure-detection techniques. This is important since the program might have actually failed during the experiment but the impact of the failure gone completely unnoticed. In particular, we encountered many cases of faults that only deviate from correct behavior in terms of the data read or written by the program. Such deviations could easily go unnoticed even when anomaly detection systems are used. Our approach of comparing externally visible behavior against a reference run, on the other hand, is a more sensitive tool for detecting failures and hence allows for more conservative experiments where no failure goes unnoticed.

In addition to finding that silent failures are an important concern that cannot be ignored, we also identified the circumstances in which silent failures are particularly common. In general, it can be said that complex programs with many different functionalities are more likely to show silent failure behavior than programs that linearly perform a single task. However, there is considerable variation across pro-

grams that cannot easily be explained from high-level characteristics. Instead, it seems reasonable to expect that the programs which use more consistency checks and assertions are less likely to fail silently. Given the similarity between injected faults and real-world bugs, this reinforces the idea that defensive programming is an important practice, even if the response to an unexpected condition is not more than an immediate panic.

Besides the programs tested in a fault injection experiment, the design of the experiment itself has also a substantial impact. We found large differences between fault types, with pointer corruption faults behaving most predictably (failing often, generally in highly visible ways) and faults leading to data corruption being the most difficult to address (failing infrequently, often in subtle ways). Hard-to-reach code tends to generate a relatively large number of silent failures, which, in turn, means that low-coverage benchmarks run a risk of masking the presence of silent failures. It can generally be said that a well-designed fault injection experiment using a broad range of fault types and good coverage is more likely to encounter silent failures than a poorly designed experiment with only the most obvious fault types and poor coverage.

In addition to providing the first thorough measurement of silent failures, our work also introduces a new framework to automatically identify fault-induced deviations in externally visible behavior. Our framework implements a simple and conservative system call matching strategy, without resorting to complex heuristics or missing any relevant deviations. In our future work, we are planning to extend our framework to deal with more complex forms of nondeterminism that are even more generally applicable. Detecting synchronization points across processes and threads, for instance, could be a viable option to eliminate common forms of scheduling nondeterminism. More complicated situations, such as test programs that adapt their process model to the system load, could be tackled using deterministic record-replay techniques [108; 9; 102; 52; 109; 77]. This would allow us to extend our analysis to generic systems software. Another potential application of our tool could be to study the interactions between multiple injected faults. Overall, we believe our framework could be used in dependability research to improve the soundness of fault injection experiments.

4

A Methodology to Efficiently Compare Operating System Stability

Abstract

Despite decades of advances in software engineering, operating systems (OSes) are still plagued by crashes due to software faults, calling for techniques to improve OS stability when faults occur. Evaluating such techniques requires a way to *compare* the stability of different OSes that is both *representative* of real faults and *scales* to the large code bases of modern OSes and a large (and statistically sound) number of experiments.

In this paper, we propose a widely applicable methodology meeting all such requirements. Our methodology relies on a novel fault injection strategy based on a combination of static and run-time instrumentation, which yields representative software faults while drastically reducing the instrumentation time and thus greatly enhancing scalability. To guarantee unbiased and comparable results, finally, our methodology relies on the use of pre- and posttests to isolate the direct impact of faults from the stability of the OS itself. We demonstrate our methodology by comparing the stability of Linux and MINIX 3, saving a total of 115 computer-days for the 12,000 Linux fault injection runs compared to the traditional approach of re-instrumenting for every run.

4.1 Introduction

While decades of advances in computer science have identified many ways to make software more reliable, crashes and downtime due to bugs in operating systems are still common. Even mature software contains a number of bugs proportional to the code size [100], so the key to making today's systems more stable is to deal with the presence of software faults.

While anecdotal evidence often suggests that some OSes are more stable than others, empirical measures of system stability are necessary if we wish to objectively determine the effectiveness of stability-improving mechanisms. This is crucial to uncover the trade-offs imposed on other properties, for instance, performance, code complexity, or portability.

In this paper, we present a novel way to measure OS stability that is *comparable*, *representative*, and *scalable*, allowing for a large (and statistically sound) number of measurements in a reasonable time frame. Our methodology guarantees comparability because it tests systems systematically, using the same workload, a similar fault load, and providing unbiased stability results. Our methodology guarantees representativeness because it emulates only real-world programmer-introduced faults which are most likely to remain in production software [96]. Finally, our methodology guarantees scalability for large OS code bases (up to millions of lines of code) because it operates the fault selection process entirely at runtime, eliminating the need for lengthy recompilation runs across experiments. By combining these properties, our methodology makes a fair comparison possible and allows stability-related decisions to be taken in a systematic and rational way.

To determine how robust a piece of software is in the face of bugs, one needs to confront the OS with a software fault (*software fault injection*) and determine whether its response to the fault is appropriate. An analogy can be made with crash-testing cars: to evaluate and compare the effectiveness of their safety features, crashes are deliberately induced under controlled circumstances and the impact on the passengers accurately measured. In the case of software fault injection, there are two ways to test response to faults: (i) using real-world faults that were previously identified in the software or (ii) injecting artificial faults designed to mimic real-world faults. While both approaches can be useful, our focus is on the latter because it allows us to compare systems facing with the same types of faults (*comparability*) and to obtain much larger sample sizes (*scalability*). As in the car crash example, these are not real-world faults but we attempt to mimic real-world faults as closely as possible (*representativeness*).

To develop a meaningful measure of stability, we need to define the anatomy of a legitimate OS response to a fault. We consider *stability* as the ability of an OS to remain usable in spite of the presence of faults, which may, however, still degrade the functionality of certain OS components. After a major car crash, one may expect the car can no longer be driven. In a similar vein, we expect the OS not to be fully functional after a fault has been injected, but we do hope that faulty OS components have minimal impact on the rest of the system.

Summarizing, the contribution of this paper lies in the introduction of a new measure of OS stability in the face of software faults that is, at the same time, comparable, representative, and scalable. Our approach uses statistical testing, allowing the user to determine whether enough tests have been conducted to draw conclusions from the results. In addition, we have implemented our approach in a way that is easily portable to widely used OSes and we have performed experiments to com-

pare two OSes (Linux and MINIX 3) to demonstrate the usefulness of our approach.

4.2 Related work

Many researchers have investigated methodologies to benchmark the dependability of operating systems using fault injection experiments. Two orthogonal fault injection strategies, in particular, prevail in prior work in the area: (i) supplying invalid inputs at the OS interface boundaries and (ii) operating targeted mutations in the OS code.

The former strategy is popular in *robustness testing* campaigns, which seek to evaluate the ability of the OS to function correctly in the presence of unexpected inputs or events. Pioneering work in the area of robustness testing was undertaken by researchers in the Ballista project [73], which first developed a methodology to supply invalid parameters to the system call interface for OS testing purposes. Their methodology was used to compare the robustness of several POSIX operating systems, eventually uncovering a number of critical and previously unknown bugs. Since then, robustness testing has been applied to a variety of domains, including comparing the dependability of different OS architectures [66], implementations [11], or evaluating the impact of faulty drivers on the robustness of the operating system [8; 61; 62]. A number of robustness testing studies have also sought to evaluate the experimental impact of different fault models [61], injection techniques [58], and injection triggers [62; 32]. While robustness testing methodologies share some similarities with our work—for example, our methodology relies on failure modes inspired by standard dependability benchmarking metrics which have been influential in the area [66]—they also pursue radically different objectives in that they aim to elicit erroneous behavior but do not actively inject faults *inside* the OS and evaluate their stability impact.

The latter strategy is popular in *mutation testing* campaigns, which seek to emulate realistic software or hardware faults inside the OS and analyze error propagation, thus sharing more similarities with our work. Prior mutation testing campaigns have served a number of purposes, including evaluating the effectiveness of fault-tolerance mechanisms [39; 54], evaluating the OS behavior in presence of errors [58; 39; 49; 48], or comparing the dependability of different operating systems [11; 25]. The testing methodologies proposed in prior work encompass different code mutation techniques, ranging from run-time injection [25; 49; 54] to binary rewriting [48; 11] and compile-time injection [127]. Run-time injection is popular for its scalability properties—the very same OS binary can be reused across many experiments with no relinking required—allowing, for instance, researchers to inject as many as 3,400,000 faults into the OS in [54]. Prior studies, however, have found run-time injection strategies to be poorly representative of realistic software faults [88]. Similar representativeness problems have been also evidenced for binary rewriting strategies [31], which still operate entirely at the binary level. Not

surprisingly, prior work based on all such strategies has focused on the emulation of hardware faults, with the only exception of the methodology proposed in [39], which attempts to mitigate accuracy problems by surgically reflecting source-level faults into binary-level mutations.

Compile-time injection strategies, in contrast, have been successfully used to inject realistic software faults into OS code [127], but at the cost of a less scalable experimental setting—each experiment requires recompiling the OS. Unlike prior approaches, our methodology relies on a *hybrid* instrumentation strategy, which introduces pervasive software fault mutations at compile time, but carefully selects those to actually inject only at run time. This approach results in a *both* scalable and representative fault injection strategy, which allows our methodology to efficiently and reliably compare the stability of different operating systems. Further, unlike prior approaches, our methodology sets out to discards nonresidual faults, which have been shown to potentially hinder the representativeness of fault injection campaigns [96].

4.3 Approach

This section discusses how we inject faults into the system, how we select them and how we determine the impact of the faults. We also list the workloads we used and discuss to what other systems our approach would apply.

4.3.1 Fault injection

To be able to study the impact of faults on the systems being compared, we perform software fault injection. Our goal is to inject the types of faults that programmers are likely to introduce accidentally in real programs. Common software fault types have been identified in the literature [41; 27; 110]. These works form the basis for our selection of fault types, which is listed in table 4.1. Selecting realistic fault types is an important element to achieve good representativeness.

After injecting faults in it, the OS cannot be relied upon to properly report the impact of the fault. For this reason, our methodology performs fault injection experiments inside a virtual machine (VM). Since faults are only injected in the guest OS, results of the test can safely be logged on the host. Our methodology relies on QEMU, with KVM to benefit from hardware acceleration. For each run, the VM is booted until a workload script provided by the host machine takes control. This script executes one of our workloads to stress the system and activate the injected fault. The guest reports whenever it reaches a new phase and before fault activation using a hypercall. We enabled hypercalls through simple memory accesses by implementing a new QEMU device. QEMU logs the data provided by the guest for later analysis.

The starting point for fault injection is to scan the target programs to identify *fault candidates*. A fault candidate is a pair of a code location and a fault type that

Table 4.1: 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

can be injected at that location [117]. Fault candidates can be identified at three different levels: source code, intermediate code, and binary code. The source and intermediate levels offer better representativeness because source-level information is lost during compilation [31; 47]. The intermediate code level is decoupled from both the source language and the target architecture, resulting in better portability. This makes it easier to compare systems. For these reasons, we decided to work on the intermediate code level. To gain access to the intermediate code, we have written a compiler pass for the LLVM compiler framework [79].

4.3.2 Fault selection

To accelerate the experiments and make our system more scalable, we inject faults only in locations that we expect to be executed. To determine which locations will be executed, we perform a number of profiling runs [61] before starting the experiment itself. During a profiling run, the workload is executed (just as in a faulty run) but no fault is injected. We register which basic blocks (parts of the code with a single entry point and a single exit point) are executed during the profiling run and inject only faults in those basic blocks that are executed in at least one profiling run.

In our methodology, only one fault is injected per run. Applying the principle of Occam's razor, our reasoning is that faults are rare enough that in most cases crashes experienced by users are due to a single fault. This approach also allows us to identify the circumstances of the crash better. The disadvantage of our choice is that we cannot study fault interactions.

In addition to the number of faults to inject, it is also important to decide which faults to inject. According to [117], the most representative way to select faults is to make the chance of selecting a certain location or fault type proportional to the number of fault candidates. This means that larger components have a proportionally larger chance of being selected (consistent with [100]) and fault types for which there

are many opportunities to introduce are injected more often.

The standard approach to inject a single fault per run at the intermediate code level entails relinking the program for each experiment. Unfortunately, this strategy does not scale to large code bases (like most operating systems), since linking can take a very long time. To improve scalability we opted for a radically different instrumentation strategy, which shifts overhead from compile time to run time. For this purpose, our fault injection compiler pass clones each basic block into a clean version and a faulty version. In the faulty version, a single randomly selected fault candidate is injected. This basic block cloning approach is inspired by prior techniques [47], but serves as a basis for a completely different injection strategy in our methodology. Our compiler pass always mutates *all* basic blocks in the program with faults—thus eliminating the need for recompilation across the experiments. Our compiler pass injects per-basic block *fault triggers* [47] that guarantee that only *one* faulty basic block (different for each experiment) is actually executed at runtime—thus preserving our single-fault-per-run assumption. Our compiler pass writes a map file with information on all fault candidates and injections.

Before starting the faulty runs, we randomly select basic blocks that were activated in the profiling runs for fault injection. The likelihood of a basic block being chosen is proportional to the number of fault candidates in that basic block, ensuring that each fault candidate has the same chance of being selected. The chosen basic block is passed to QEMU as an argument. The guest OS is slightly modified to perform a hypercall at the earliest opportunity to retrieve the number of the faulty basic block. In modular operating systems, each module does so individually. At run-time, each per-basic block fault trigger checks whether the stored basic block number corresponds with its own and executes the faulty version of itself if this is the case. This incurs some runtime overhead, but our measurements show that this takes far less time than the additional linking that would otherwise be needed. Hence, this strategy is effective in lifting the scalability of static bitcode-level injection close to that of run-time binary-level injection, while retaining similar representativeness guarantees to that of source level injection—the best of both worlds.

Figure 4.1 illustrates the steps that have been described here. The main consequence of the sequencing as shown in this diagram (chronologically from left to right) is that due to run-time fault selection only the “testing” phase is performed multiple times. This allows for scaling with regard to both size of the code base and number of experiments because the linking time is not multiplied by the number of experiments.

4.3.3 Classification of results

It is hard to automatically classify the state of a system after fault execution. We aim to determine the stability of the OS itself rather than the impact of the fault. We consider it acceptable for a fault to prevent the part of the system it was injected in from working properly, but we are interested in evaluating the degree to which the OS can

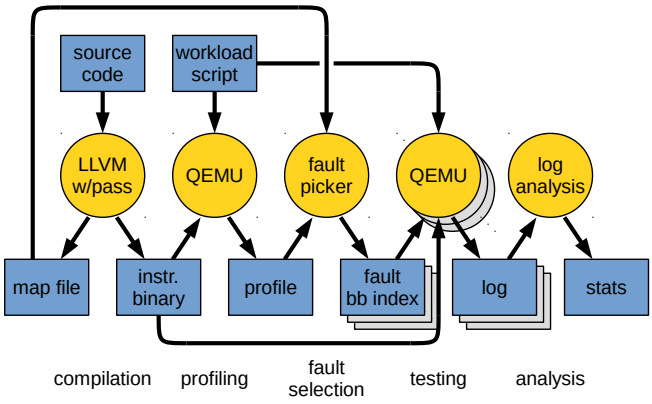

Figure 4.1: Phases of our approach

prevent arbitrary faults from bringing the system as a whole into an unstable state.

Our approach takes the problem of separating the direct impact of the fault from system stability into account by testing whether the faulty system is functional both before and after running a workload. Once the system is booted, it loads a script that will perform the tests and the workloads. Before running our workloads, a pretest is performed. If it succeeds, the run is considered valid. If it fails or is never reached, the run is marked as invalid and is not considered when determining how many faults make the system unstable. The reasoning is that a fault that prevents the system from booting or that breaks basic functionality would never go unnoticed and therefore would not end up in production software. Hence, the faults injected in valid runs are similar to what is termed "residual faults" elsewhere [96]. For valid runs, the workload is executed and afterwards a posttest (same as the pretest) is performed. If the second test also passes, we conclude that the system has retained its original functionality. This is interpreted as a sign of stability. If the pretest succeeds but the posttest fails or is never reached, the fault is considered residual and made the system unstable. This is interpreted as an indication that the OS is not very stable in the presence of faults. Such runs will be referred to as "crash" runs.

Unfortunately our methodology does not allow us to determine exactly what went wrong. We cannot distinguish different types of crashes and hangs. The aim of this paper is to provide a first demonstration of our methodology for efficient fault injection. Since the way the results are analyzed is orthogonal to this, a crude but simple approach is most suitable. More advanced approaches can be explored in future work.

The tests serve to determine whether the system would be perceived by a user as alive and functional. They test functionality that would be easily found to be broken while the system was being tested. As such, they are less rigorous than the workload, which should stress as much of the system as possible. In this work, we model the

Table 4.2: Workloads

code	description	tests
bsh	Bash regression test	shell functionality
gdb	GDB-like workload	ptrace
htp	Apache workload	networking
mnx	Reduced MINIX test set	calls provided by MINIX (incl. POSIX)
uxb	Unixbench	performance-sensitive POSIX calls
vim	Vim regression test	interactive use of the tty

OS being tested as a server system that is reachable from the outside through the network. We make the host system connect to the guest through SSH and create a file inside the VM. If the file is successfully created, the system is reachable and would be considered to be alive by a user. However, our methodology could easily use other kinds of tests for other types of systems. Test selection influences which faults are tested because faults detected by the pretest are excluded as invalid runs. We recommend adapting the test to the role of the operating systems being tested to most effectively identify residual faults and decide whether the system would be considered to be functional by a user.

4.3.4 Operating systems and workloads

Because our system operates at the intermediate code level, it requires the source code. To demonstrate our methodology we selected two open-source systems. The first is Linux as it is one of the most popular open-source operating systems. As for the second system, we opted for a system with a very different structure to test the versatility of our methodology. It seemed appropriate to select a multi-server microkernel system based on theoretical claims that this design should be more reliable than the more common monolithic design [54]. If there is indeed a difference, our methodology should be able to measure it. Based on these constraints we picked MINIX 3, an open-source multi-server microkernel system that has good POSIX support due to its use of the NetBSD C library. An additional advantage is that the latest release supports LLVM bitcode compilation out of the box.

Our aim in selecting the workloads was to find tests that work identically on both systems and use a variety of different system calls. The workloads we selected are listed in Table 4.2. The codes will be used to refer to the workloads in the results section. Unixbench and the MINIX test set were included because both use a wide range of system calls. For the MINIX test set, we had to disable a few tests because they use functionality not provided by Linux. The other workloads were all included to test specific parts of the system to determine whether this results in different stability behavior.

4.3.5 General applicability

The main requirement to be able to use our tools is that the OS can be built with the LLVM compiler in bitcode mode. The LLVM compiler has front-ends for many different languages, amongst others C and C++, which are used for most OSes. It also has a high degree of compatibility with GCC, a compiler which is very widely used for such systems. In the case of our experiment, MINIX 3 was already fully compatible with LLVM bitcode while for Linux we had to use the LLVM Linux project [2], which removes reliance on some obscure GCC extensions of the C language that LLVM chose not to implement. We expect that in time these changes will be merged into mainline Linux. Some small build system changes were required to support bitcode linking. More and more OSes are starting to provide support for LLVM, such as for example Apple and the BSDs. Also, for standards-compliant code there is no need to even make changes to be able to use it.

To be able to use our approach as described here, a few very small changes must be made to the OS. In particular, the system must be linked against the fault injection library, provide a way to perform the required hypercalls and request which fault is to be injected at boot time. Other invocations of the hypervisor are performed automatically by the compiler pass (reporting whenever the fault is activated) or the script controlling the experiment (reporting whether the pre- and posttest have succeeded). Our changes introduced 127 new lines of code in Linux and 208 in MINIX. These additions implement hooks called by compiler-generated code and add a hypercall during early boot. This means the only OS-specific knowledge needed is how to access physical memory and what is the earliest time after booting when this can be done.

4.4 Results

To compare how stable Linux and MINIX 3 are according to the methodology outlined in this paper, we have performed a total of 24,768 experiments. For each combination of the two systems and six workloads, we performed 64 profiling runs and 2,000 faulty runs. We believe the number of profiling runs is adequate because performing more runs does not increase coverage any further (for MINIX it increases up to 32 runs, for Linux up to 16) and because the standard errors on the timing are very low compared to the total runtime (see Table 4.4). As for the faulty runs, more is always better because it allows statistical tests to be performed for more uncommon events, such as faults injected in components with small code size.

To give some crude indication of the desired number of experiments we start from the common rule of thumb that the χ^2 test (used to test whether crashes are more common in some cases than in others) is only accurate if the expected value is at least five in all of the cells of the contingency table [124]. Overall, approximately 2% of the total number of runs are "crash" runs, the case that we are most interested in. Theefore, we require approximately 250 runs to be able to perform a χ^2 test. For example, the total number of fault injection experiments we ran per OS ($n =$

$12,000$) allows us to perform tests for components making up at least 2% of the code base. These numbers are not exact because it depends on the number of valid crash runs in the part used as a reference but it gives some indication of the total number of experiments required. For our purposes, the number of experiments performed turned out to be sufficient to perform tests for all relevant cases.

For the experiments, we used two VMs with 4GB RAM each. One runs Linux 3.11 rc4 (the version supported by LLVM Linux [2]) with the Ubuntu Server 12.04 LTS distribution, the other a pre-release of MINIX 3.3.0 (the first version supporting LLVM bitcode linking). Both OSes were slightly modified to report fault activations through our hypercall interface (127 lines in Linux and 208 in MINIX). As a hypervisor we used QEMU 2.0.0 with KVM acceleration. We modified QEMU by implementing an additional device to provide the hypercall interface for the guests (899 lines added). The experiments were conducted on nodes of a computer cluster, each with two Intel Xeon E5620 CPUs at 2.40GHz and 24GB of memory. These nodes were used exclusively for our experiments, with only one experiment running per node at a time to avoid interference of other jobs with the timing.

4.4.1 Coverage

Table 4.3 shows the coverage reached by our workloads. in terms of both fault candidates ("fc") and lines of code ("loc"). The units show reasonable agreement, which means our approach of making the likelihood of injection proportional to the number of fault candidates is in agreement with the most common measure of code size. Unfortunately, the combined coverage reached is quite low, especially on Linux. While it would be desirable to reach higher coverage [117], it is hard to do so because we can only use features supported by both operating systems in our workloads. It may be possible to reach higher coverage in systems that are more similar, but then there is still the issue that much of the hardware support is not used, especially when run in an emulator. For example, running the full MINIX test set (including the tests that do not run on Linux) does not give substantially better results because many hardware-related modules have very poor coverage.

When considering the individual workloads listed in Table 4.3, it is clear that the reduced MINIX test set ("mnx") is the most extensive workload, reaching the highest coverage on both systems in both units. The combined coverage, however, is still clearly better than any individual workload. This shows that the workloads test different parts of the system, so there is value in keeping them separate to find whether the response to faults is affected by the types of operations performed.

4.4.2 Fault activation

There is some nondeterminism that causes the parts of the program executed not to be exactly the same from run to run, even if no faults were injected. Although we only injected faults in basic blocks activated in the profiling runs, 1.8% of the faults

Table 4.3: Coverage as % of fault candidates (fc) and lines of code (loc)

workload	Linux		MINIX	
	fc (%)	loc (%)	fc (%)	loc (%)
bsh	15.9	16.5	38.5	38.0
gdb	15.6	16.4	37.5	36.4
htp	15.8	16.5	37.6	36.5
mnx	16.2	16.9	39.1	38.5
uxb	15.9	16.5	37.4	36.4
vim	15.9	16.5	37.4	36.4
(combined)	16.9	17.4	40.0	39.4

did not get activated on Linux and 0.3% on MINIX. Although we could re-run these experiments until the fault was activated, we opted not do do so because it would bias the results. In real-world situations faults in these locations would also be less likely to trigger than those in deterministically executed locations, so our stability conclusions should reflect this. Instead, we discarded these runs and did not consider them for the statistics. This approach seems to have the least risk of introducing bias compared to real-world faults. However, given that the number of nonactivated runs is so small, the alternative choice would not have influenced our conclusion.

4.4.3 Scalability

As discussed in the approach section, we have opted to accept a slowdown to select the active fault at runtime to save compilation time and overall hope to accelerate the experiments. Table 4.4 shows the impact of the overhead on the time each experiment takes, from starting up QEMU to the completion of the posttest. The results are averages over 64 runs, which is sufficient to obtain very reliable measurements judging from the standard errors. On our system, using LLVM with bitcode, it takes 59m10s to compile Linux, 1m4s to instrument it and 15m0s to link it. With standard bitcode-level fault injection (i.e., no runtime fault selection), the system has to be re-instrumented and re-linked before each run. This would cost 5784s for all workloads together (six runs) and would save only 811s of runtime. This means our solution is more than seven times as fast. MINIX takes 3m44s to compile, 1m16s to instrument and 3m5s to link. This is 1566s for six runs to save 629s of overhead. Here, our approach is a factor 2.5 faster than the alternative. On the whole, we save 115 computer-days on the Linux runs and 22 computer-days on the MINIX runs. This is a low estimate, as many runs crash early, resulting in even less runtime overhead. Clearly, our choice is very effective in making our approach more scalable. That said, it should be noted that the decision must be made on a case-by-case basis, as the best choice could turn out differently for smaller OSes running larger workloads.

Table 4.4: Runtime with and without instrumentation

system	workload	uninstrumented (s)		instrumented (s)		slowdown (%)
		mean	std.err.	mean	std.err.	
linux	bsh	154.6	0.0	212.0	0.1	37.1
linux	gdb	211.7	0.0	220.4	0.0	4.1
linux	htp	271.3	1.9	301.7	0.1	11.2
linux	mnx	236.3	0.2	364.7	1.2	54.3
linux	uxb	554.7	0.1	1112.1	0.1	100.5
linux	vim	176.8	1.7	205.8	1.6	16.4
linux	(total)	1605.5	4.0	2416.6	3.2	50.5
minix	bsh	168.7	0.0	224.6	0.0	33.2
minix	gdb	93.4	0.1	109.6	0.0	17.4
minix	htp	415.5	0.3	546.7	0.4	31.6
minix	mnx	377.6	0.1	682.4	0.5	80.7
minix	uxb	1097.2	0.1	1110.8	0.1	1.2
minix	vim	120.8	0.1	227.7	0.1	88.4
minix	(total)	2273.2	0.7	2901.7	1.1	27.6

4.4.4 Systems and workloads

Although it provides more information, the main aim of our methodology is to determine in a systematic way whether one system is more stable than another. Table 4.5 provides the outcome of this test. Note that the number of valid runs is not split between workloads because it is inherently unaffected by the workload, which runs after the result of the pretest has already been reported. There is a substantial and significant difference between Linux and MINIX in the number of runs that are valid but the difference in the number of valid runs where the posttest fails (further referred to as "crash" runs here) is small and not statistically significant. This is consistent between the workloads, some of them giving the benefit of the doubt to Linux and others to MINIX but not one of them showing a significant difference. Based on this result and contrary to what might have been expected theoretically, MINIX cannot claim to be more stable that Linux. Residual faults are approximately equally likely to crash both systems.

However, MINIX does have the advantage that more faults are detected early by interfering with the basic functionality of the system of booting and performing the pretest. As a consequence, it is expected that faults are on average easier to detect and fewer of them will remain in production releases. This suggests that MINIX' use of memory protection between modules is effective in causing early crashes for some faults, but in the end the implemented isolation and recovery mechanisms are not sufficient to prevent faults from spreading or bringing down the system. However, to know for sure would require more in-depth analysis of what is happening on the crash runs. To do this automatically would require dropping the black box assumption. It could be a suitable topic for future research but is out of scope for this paper because it is not as widely applicable as our methodology presented here.

Comparing the workloads between each other, the reduced MINIX test ("mnx") stands out for triggering significantly more crash runs than the other workloads.

Table 4.5: Stability of systems per workload

workload	Linux			MINIX		
	valid (%)	crash (%)		valid (%)	crash (%)	
bsh		2.9	*		4.2	
gdb		4.6			3.5	
htp		4.0			4.5	
mnx		5.3	*		7.3	***
uxb		4.4			4.3	
vim		4.2			4.3	
(total)	59.2	4.2		39.9 ###	4.7	

χ^2 comparing with other workloads significant at *=$p < 0.05$, **=$p < 0.01$, ***=$p < 0.001$; χ^2 comparing with other system significant at #=$p < 0.05$, ##=$p < 0.01$, ###=$p < 0.001$

Given that this workload also reaches the highest coverage (see Table 4.3) this seems to be due to the fact that it is simply the most extensive workload, most capable of triggering crashes. The Bash regression test ("bsh") triggers significantly fewer crashes than the other workloads on Linux. This may be cause by the fact that Linux heavily relies on shell scripting to initialize the system at boot time, so the fault most affecting the Bash shell would have been spotted earlier and resulted in invalid runs.

In addition to providing a comparison between the systems and workloads tested, another interesting result is the fact that the vast majority of residual faults (more than 95% of them) do not crash the system. Despite the lack of isolation in the Linux kernel, they apparently do not cause enough corruption to interfere with the posttest. We plan to delve into this phenomenon deeper in future work by determining what kind of impact these faults do have - are they inherently harmless or do they cause some damage eventually that can only be noticed after specific triggers?

4.4.5 Operating system components

To find why MINIX is not more resilient against injected faults than Linux, we have used the code paths provided by the LLVM debug symbols to classify the locations in which we have injected faults. The results are presented in Table 4.6. We did not perform statistical tests between the systems here because, as accurate as we tried to be in classifying the code paths into components, the differences between the systems are too large to make the (groups of) components fully comparable. For example, the Linux kernel contains functionality that in MINIX is provided by the process manager ("pm"), virtual file system ("vfs") and the other system servers ("servers"). However, these different organizations do not prevent us from identifying the more and less robust parts of both systems individually.

Microkernel systems such as MINIX aim to reduce the trusted code base (TCB) of programs that have sufficient privilege to bring down the system as a whole (rather than just the parts that directly depend on its functionality). Ideally, components

Table 4.6: Fault types

component	Linux					MINIX				
	n	valid (%)		crash (%)		n	valid (%)		crash (%)	
driver	2189	61.41	*	1.06	***	1037	52.84	***	1.82	***
fs	2262	57.55		9.3	***	1057	51.18	***	4.07	
kernel	1332	65.24	***	1.99	***	1620	37.61	*	7.65	***
lib	2383	57.24	*	4.1		1738	33.24	***	4.34	
mm	730	61.1		2.03	*	1234	25.61	***	3.8	
net	1496	63.03	**	3.83		2000	41.94	*	1.79	***
pm						535	53.46	***	12.59	***
servers						352	43.75		3.9	
vfs						1896	39.7		6.12	*
other	1608	51.73	***	5.59	*	531	30.19	***	3.75	
(total)	12000	59.15		4.23		12000	39.87		4.69	

χ^2 comparing with other components significant at *=$p < 0.05$, **=$p < 0.01$, ***=$p < 0.001$

such as the drivers and networking ("net") should be outside the TCB. As expected, residual faults injected in either of these components result in significantly fewer crashes than faults injected elsewhere. Apparently, privilege reduction and isolation are effective here. The kernel, PM and VFS, on the other hand, show significantly more crashes. These three components are firmly within the TCB, with considerable privileges and the entire system depending on them. The memory manager ("mm") is also in the TCB but does not show a high number of crash runs. Given the very low number of valid runs, it seems faults in this component tend to bring down the system early and are therefore unlikely to make it into production systems. Summarizing, it seems the microkernel design is effective but the TCB is highly vulnerable, causing the average not to be better than Linux' average.

For Linux, the most vulnerable component is by far the file system. This might be due to the fact that Linux uses the EXT4 file system, which is far more complicated than MINIX' MFS. The more complex code could allow serious bugs to "hide" for a longer period of time before corrupting the experiment. In the light of arguments commonly made in favor of microkernels, it is remarkable that the drivers and the core kernel are actually Linux' least vulnerable parts. Apparently the spread of corruption in a highly privileged part of the source code is not as large an issue in practice as would be expected. To find out why this is the case, one would need to perform a more in-depth analysis, something which we plan to do in future work.

4.4.6 Activation time and fault latency

Because we log each fault activation, it is possible to determine the impact of the timing of the fault on the outcome of the experiment. Because timing in terms of seconds is hard to compare between systems and workloads, we have opted to instead consider during which step of the experiment the fault was first activated. Table 4.7 shows the results.

The first thing that stands out it the fact that the vast majority of faults (87% on Linux, 89% on MINIX) is first activated while booting. Because the likelihood of

Table 4.7: Step of first fault activation

first act.	Linux					MINIX				
	n	valid (%)		crash (%)		n	valid (%)		crash (%)	
boot	10432	56.7	***	1.6	***	10689	37.3	*	3.2	***
pretest	907	66.7	***	7.1	***	1048	53.1	*	3.2	
workload	415	100.0	***	35.9	***	226	98.2	*	34.7	***
posttest	31	100.0	***	25.8		5	100.0		0.0	
shutdown	3	100.0		0.0		0				
(never)	212					32				
(total)	12000	59.2		4.2		12000	39.9		4.7	

χ^2 comparing with other steps significant at ***=$p < 0.001$

fault injection is proportional to code size, this means there is relatively little code that is used while the OS is running but not used when initializing the system at boot time.

Considering the number of valid runs, faults first activated during boot time are most likely to cause the run to become invalid (fail or not reach the pretest). Due to the large number of faults activated at boot time and the fact that faults activated after the pretest cannot make a run invalid, this group of faults is dominant in determining the overall percentage of valid runs.

The percentage of valid runs that fails to pass the posttest (listed as "crash" in the table) also differs greatly depending on the first activation of the fault. Faults triggered during boot time and (for Linux) during the pretest are significantly less likely to be counted as crash runs than the average while faults first activated by the workload are far more likely to cause crash runs. Combining this with the previous result, it becomes clear that many of more serious early faults are weeded out because they crash the system while booting or undermine the basic functionality of the system tested in the pretest. This means that on average boot-time activated faults are less likely to go unnoticed and make it into production software and those that do are on average less dangerous that late-activation faults. However, this does not take into account that the total number of early-activation faults is much larger. When considering the total n, we find that many crash runs are caused by faults first activated at boot time (32% of them on Linux and 58% on MINIX). This means that long-latency faults cannot be ignored. It is worthy of note that MINIX seems to suffer more from long-latency faults than Linux does. Our current experiment does not allow us to identify the reason why, but we will delve deeper into latent corruption and long-latency faults in future work.

4.5 Threats to validity

Although we believe our methodology is one of the most effective ways to compare OS stability, some factors that threaten its validity must be considered when using it. For representativeness it is important to note that although we took care to select

realistic faults, the faults we test are artificial. Real-world faults are more representative, but are problematic when trying to achieve comparability and scalability. Another issue that introduces differences with real-world situations is the fact that we have to virtualize and instrument our code. While necessary to run the experiments in an automated fashion, this introduces timing differences that could change the behavior of race condition bugs. Another limitation is the fact that the choice of pre- and posttest influences the bugs that will be tested by determining which ones are classified as residual (and hence potentially harmful). This can be addressed by selecting a test that is consistent with the way the system would normally be used in practice. A related issue is the fact that it is very hard to tell whether a system is functional, especially in a black box setting. For example, one cannot tell whether a system hangs or is just being slow. It is therefore impossible to automatically classify the state of the system in all cases. Using the posttest is a workaround to bypass this issue. Finally, we cannot determine whether latent corruption is present after fault activation that could be exposed by a more thorough workload. This is something we will address in future work by determining whether the internal system state is still correct after fault activation.

With regard to our evaluation, it should be noted that we only compare against a traditional compilation-based approach. Run-time (binary-level) approaches would be faster but we do not consider them comparable due to their representativeness issues [88]. It should also be considered that we used only LLVM and other compilers may generate code that reacts to faults differently.

4.6 Conclusion

In this paper, we have presented a novel methodology to systematically compare OS stability in a way that allows for meaningful comparison using statistical methods, is representative of faults made by programmers that make it into production software and can scale to a large number of experiments even for very large code bases. Our methodology is widely applicable. We have successfully applied this approach to two structurally very different operating systems, showing that our unconventional choice to shift work from compile time to run time is highly effective in speeding up experiments without compromising on the source-level information available to the fault injector.

5

HSFI: representative fault injection scalable to large code bases

Abstract

When software fault injection is used in current research, faults are typically inserted at the binary or source level. The former is fast but provides poor fault representativeness while the latter cannot scale to large code bases because of large rebuild times. Alternatives that do not require rebuilding incur large run-time overheads by applying fault injection decisions at run-time. HSFI, our new design, allows faults to be injected with all context information from the source level and applies fault injection decisions efficiently on the final binary. It does so by placing markers in the original code that can be recognized after code generation. We implemented a tool according to the new design and evaluated the time taken per fault injection experiment when using operating systems as targets. We can perform experiments more quickly than other source-based approaches, achieving performance that come close to that of binary-level fault injection while retaining the benefits of source-level fault injection.

5.1 Introduction

Despite significant advances in bug detection tools, software bugs continue to be a major source of system crashes [83]. Moreover, the number of bugs in mature code is a linear function of the number of lines of code [100] so as software continues to grow more and more complex it is reasonable to expect that software bugs will be an even bigger threat to system reliability in the future. This means that we have no choice but to accept the presence of bugs as a given and devote our attention to mechanisms that allow the systems we build to tolerate bugs. To build such systems, we need to be able to quantify fault tolerance to verify whether we are doing a good

job. Fortunately, fault injection is an effective way to measure the impact of real faults if a sufficiently large number of experiments are performed [37; 10].

Although there has been substantial research into software fault injection and several techniques are widely used, each of the existing approaches has some important limitations. Injection of hardware faults such as bit flips is not representative of software bugs [88] and neither are faults injected at the interfaces between components [91; 78]. The main alternative is mutation testing, where software is modified to introduce deliberate bugs similar to those that a real programmer might have accidentally introduced. This can be performed at several levels. At the binary level, the machine code is analyzed to recognize coding patterns and mutate them directly in the machine code. Unfortunately, Cotroneo et al. [31] have shown that in practice this approach poorly represents true software faults. This can be explained by the fact that context information is lost when the compiler transforms a program's source code into binary code. The main alternative is to modify the source code itself. Daran et al. [37] and Andrews et al. [10] have shown this approach to be effective in mimicking real software faults. Source-level fault injection is increasingly popular [60] but unfortunately it scales poorly to large code bases due to relinking overhead [19]. Van der Kouwe et al. [120] have reduced this overhead by injecting multiple faults at a time and selecting the desired ones at run time but this incurs a severe run-time performance penalty. To the best of our knowledge, there are currently no approaches that allow high-representativeness mutation testing on large code bases without a substantial performance penalty.

In this paper, we introduce a new design called HSFI (Hybrid Software Fault Injection) that allows source-level context information to be used to inject faults while making it possible to enable and disable those faults in the final binary without rebuilding. As a consequence we do not have to rebuild the code for each experiment, which allows the approach to scale to large code bases without substantial run-time overhead. This allows for substantial savings when testing fault tolerance of large software systems such as operating systems. These savings are becoming even more important as we move towards multiple-fault models [122], which greatly increase the number of potential fault scenarios. Our technique complements other approaches to reduce the cost of fault injection, such as pruning the search space to test the most important failure scenarios first [15; 50; 63; 30] or performing multiple experiments in parallel [123]. Moreover, within the domain of software faults, our design is agnostic to the fault model and our implementation can easily be extended with new fault types or fault injection policies.

The core idea of HSFI is to inject many faults at the same time before the code generation phase and selectively enable or disable the individual faults by directly modifying the machine code in the final binary. For each injected fault, two versions of the code are generated: a pristine version that performs the original operations as specified in the source code and a faulty version where those operations are mutated according to the fault type that is applied. For each location where a fault is injected, a marker is generated that can be recognized by our binary pass. The binary pass

then uses these markers to recognize which code sections are pristine and which are faulty and uses this information to efficiently enable or disable individual faults without the need to rebuild the program.

Our evaluation shows that our approach performs well and is highly effective at marker detection. It provides performance close to that of binary-level fault injection for both operating systems we used as fault injection targets while alternative approaches that allow source-level information to be used are shown to result in substantial overhead.

5.1.1 Contributions

This paper makes the following contributions:

- A new design point in software fault injection that combines availability of source-level information with low overhead even for large code bases.
- A practical implementation of this design that outperforms both source-level fault injection and previous approaches in most cases and is close to binary-level fault injection performance in all cases.
- Source code will be made available after publication, allowing others to benefit.
- An empirical evaluation that shows in which cases it would be beneficial to use our new approach instead of traditional ones.

5.2 Background

To show why a new approach for fault injection is needed, in this section we will first discuss why fault injection itself is important. Next, we briefly go through the various types of fault injection currently used in academic research and their purposes. Finally, we go through the available techniques for software fault injection in more detail to demonstrate why it would be beneficial to obtain a new trade-off between availability of source-level information and performance on large code bases.

Fault injection is used for a wide variety of purposes, usually related to quantifying and/or improving reliability. Although fault injection is used in many different domains and our solution is also widely applicable, we will focus particularly on fault injection into operating systems. Operating systems are typically very large pieces of software and they perform a critical role in system reliability as a failing operating system causes the entire system to be unavailable. Because operating systems run at a high privilege level, bugs can do a lot of damage. Even just in the limited context of operating systems, fault injection has been used in many different ways. Examples include evaluating operating system stability in case of faults in the operating system itself [125; 71; 120], testing isolation properties to protect against faulty device drivers [39; 54], determining to what extent faults can go unnoticed and cause corruption later [119], certification of safety properties [33], and

improving fault tolerance by finding the most likely situations in which data can be lost [98]. Fault injection is a critical tool that is necessary to reach more acceptable levels of operating system reliability in the face of faults.

The first step of any fault injection campaign should be to decide on a *fault model*. The fault model specifies what types of faults will be injected and should be consistent with the types of faults the target system will be expected to tolerate. Fault models can be broadly categorized in three groups that typically require different fault emulation techniques to be used: faults external to the system that are observed at the system's interfaces, faults in the hardware the system is run on, and faults in the software of the system itself.

Fault models where interface-level faults are injected are appropriate to test system responses to external changes without changing the target system itself. An external fault could occur in a library (for example LFI [90]), another software component on the system or another system entirely (such as FATE [50] which emulates I/O failures). In all cases, such faults can be easily emulated by providing an incorrect response at the interfaces that delimit the target component, for example by replacing a system library or interposition in inter-component communications. Although these models are suitable to evaluate robustness in the face of unexpected external events, they are not appropriate for cases where we must deal with software bugs in the target component itself [91; 78].

We will refer to models where hardware faults are emulated as hardware fault injection, although it should be noted that this term is sometimes used to indicate that hardware is used as a tool to inject faults (such as for example MESSALINE [12]). However, many more recent systems emulate hardware faults in software; for example, FINE [68] and Xception [23]) inject faults in program binaries, LLFI [85] injects hardware faults in intermediate compiler representation, and GemFI [101] uses a cycle-accurate full-system emulator to inject hardware faults. Hardware fault models have been widely used for a long time and are well-researched. Typical example fault types include memory faults, CPU faults, bus faults and I/O faults, all of which can be either permanent or transient [68]. Although such fault models are suitable to research to impact of failing hardware, Madeira et al. [88] have shown that hardware-based fault models are not suitable to emulate the impact of software faults.

We refer to the final group of fault models as *software fault injection* because software faults are being injected. Again, it should be noted that the term is used inconsistently in the literature: it is sometimes used to indicate that software is used as a tool to inject faults (as for example in FERRARI [65]). Another commonly used term is mutation testing. While the other types of fault model have been used by many engineers for a long time, software fault injection has started to see widespread use mostly in recent years [60]. Compared to the other types of fault injection, injection of software faults gives rise to a number of unique challenges. Software faults, also commonly known as bugs, are introduced accidentally by human programmers. This means that fault models cannot be derived from system designs or API specifications, but should ideally be empirically derived from mistakes humans make in

practice. Several such models have been proposed in the literature, most notably G-SWFIT [41], which provides both such an analysis and a tool to apply the fault model and has seen considerable use in the literature. Another key difference with the other models is that software faults are always persistent and the target system itself must be modified, which means that the use of hardware to emulate faults is not practical and run-time approaches like virtualization are not needed. It should be noted that, although persistent, software faults need not be deterministic, as for example race condition faults can be persistent yet nondeterministic. Since improving software fault injection is the focus of this paper, we will continue by discussing the implications of these unique properties of software fault injection.

When performing software fault injection, the next step after deciding on a specific fault model is to identify *fault candidates* in the target system. A fault candidate is the combination of a code location and a type of fault that would be likely to be introduced at that location according to the fault model. Because software faults are introduced by humans, it is necessary to take into consideration the context in which the programmer wrote the code to determine whether some code location should be considered a viable fault candidate. After all, to get a realistic impression of the impact of faults, the introduced faults must be representative of those that a real programmer would make, and that is potentially influenced by the context. After fault candidate identification, one or more fault candidates are chosen for each individual fault injection experiment. The fault injector then mutates the program to introduce the selected fault types at the selected locations. Finally, the modified target system will be run with a workload to examine the impact of the injected fault(s).

One common way to implement software fault injection is to perform these steps on the binary after it has been generated by the compiler and the linker, either in the binary file or in the executable image after it has been loaded into memory. We will refer to this approach as *binary-level fault injection*. This approach is used, for example, by the widely used G-SWFIT [41] tool and has also been used to quantify and improve the fault tolerance of the RIO file cache [98]. Binary-level fault injection can be done very efficiently because recompilation and relinking is not necessary and no run-time overhead is introduced. Moreover, this approach is applicable even if the source code is not available. To identify fault candidates in binary-level fault injection, the machine code in the binary must be disassembled and the tool has to identify programming constructs in the resulting assembly code. Mutations are also introduced at this level. Unfortunately, however, context information is lost when binary code is generated. This means that binary-level fault injectors have limited knowledge about the context. For example, when functions are inlined or loops are unrolled, the binary fault injector cannot distinguish these cases from code that was manually written multiple times. Because information is lost in this way, faults injected at the binary level do not properly represent bugs that would be introduced by a real programmer, as shown by Cotroneo et al. [31]. Another drawback of binary-level fault injection is that each processor architecture requires its own fault injector. It is likely that different fault candidates will be detected on different architectures,

which limits comparability between systems. Such differences may arise even when just comparing different compilers or compiler settings on the same architecture.

The main alternative is *source-level fault injection*. In this method, fault candidates are identified and mutations performed on the source code itself, before the compiler generates machine code. A widely used example is SAFE [96], which is based on the G-SWFIT fault model. As an alternative to modifying the source code itself, this approach can also be implemented as a compiler pass (like in SCEMIT [84]). In either case, no context information is lost; the code seen by the fault injector is what the programmers themselves wrote. This approach has been shown to result in good representativeness of real software faults [37; 10]. However, every time mutations are performed in the source code, the system must be rebuilt to obtain a modified binary to test with. Programs are usually split up in many small source files that are individually compiled if they are changed, using a system such as *make*. Recompiling a single source file usually does not take a lot of time even for large programs. However, in the end all source files need to be linked together into a single binary. This step can take a lot of time for large programs such as operating systems and must be performed every time one or more source files are changed. Rebuild time can be sufficiently long that it becomes the dominant factor in fault injection experiments on large code bases[120]. Although it is possible to bypass rebuilding by injecting multiple faults and selecting one or more at run time, this results in substantial run-time overhead[120], which takes away some of the gains. This overhead is especially important for long-running workloads and makes testing of timing-related faults such as race conditions less representative of real-world conditions. It can be concluded that, although there are many techniques available for software fault injection, none of them combines the good representativeness of source-level techniques with the good performance offered by binary-level techniques.

5.3 Overview

In this section we describe the steps performed by a fault injector tool following the HSFI design. Fig. 5.1 provides an overview of this design. We only discuss elements that are fundamental to the design in this section, while leaving elements that are specific to our implementation of the design for the next section.

When setting up a fault injection campaign, the first step is to select an appropriate fault model. Although our design is specific to the injection of software faults, it is agnostic to the exact fault types defined by the fault model and these can be chosen freely. For maximum representativeness it is recommended to select a fault model based on software bugs found in real-world software, for example using prior research such as that of Duraes et al. [41] or Kidwell et al. [70]. It is also important to ensure that the selected faults are representative of residual faults that are likely to elude testing and end up in production systems [96]. The fault model can be implemented by writing simple *fault type plug-ins* for our framework. One plug-in

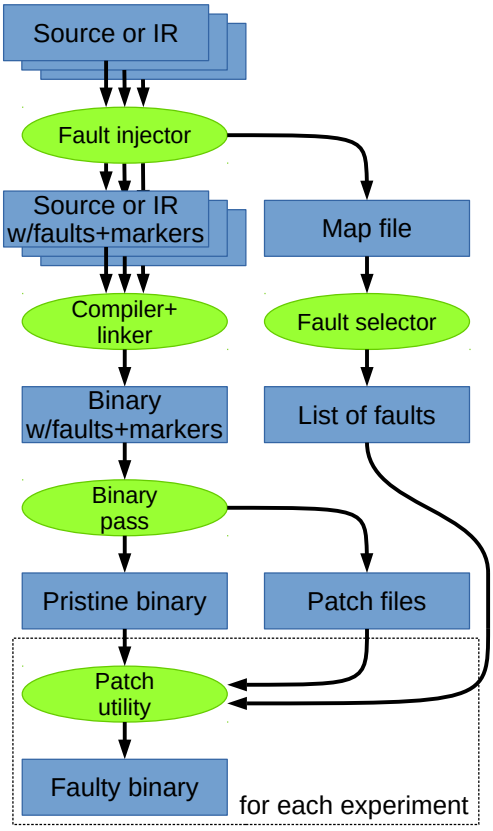

Figure 5.1: Fault injection design; IR=intermediate representation

is created for each fault type and it should implement two callbacks. The "canInject" callback receives an instruction and its context in the source code and returns a nonzero value if there is a fault candidate for the fault type at that location. The "apply" callback modifies the program at the given instruction to inject the fault. This allows our approach to be easily extensible and widely applicable.

The second step is to run the fault injector on the program code. Depending on the implementation this can be done either on the source code directly (in a language such as C) or inside the compiler on an intermediate representation (IR) used by the compiler to represent the parsed code. The fault injector iterates through the program and calls the plug-ins to identify fault candidates. It then applies as many fault candidates as possible to the code by first duplicating the target code and then applying the appropriate fault type to only one of the copies. This yields one *pristine* copy and one *faulty* copy. It then inserts a *marker* in the code that allows the two copies to be identified even after the source code or IR is converted into machine code. In addition, it writes a *map file* that contains a list of all the markers, including

context information such as the fault type and the original code location that may help in fault selection later on. It should be noted that the addresses of the markers in the final binary are not yet known at this point.

After running the fault injector pass, the program can be compiled and linked together into an executable binary. Because many faults have been injected by the fault injector, this step does not need to be repeated for every experiment like in traditional source-level fault injection.

Now that the binary has been obtained, a binary pass is run on the code. This pass scans through the binary and identifies the markers. For each marker, it modifies the binary to ensure that the pristine code is executed. It also writes a *binary patch* for each fault injected that can later be used to switch to the faulty version instead. Although this pass is architecture-specific, it only needs to recognize predefined markers and perform very simple patches. While some porting effort would be required to support multiple architectures, this does not affect the code related to recognizing the fault types or code manipulation. Moreover, our approach would identify the same fault candidates and make functionally equivalent mutations regardless of the underlying architecture.

Finally, the experiments can be performed. This is the only part that needs to be repeated for each experiment. A patch tool is invoked to enable the fault(s) that are to be tested. Because the patches usually involve changing only a few bytes, the patching is near-instant. Moreover, it can be done in-place, which avoids making a copy of the (potentially large) binary. In this case the fault patches must be reverted after the experiment. After injecting the appropriate fault(s), one can run a workload to exercise the code and observe the system's response to the fault(s).

5.4 Implementation

We have implemented a fault injector and a binary pass according to the design described in the previous section. We have opted to implement our fault injector using the intermediate compiler representation. We use LLVM [79] for this purpose as it provides an extension framework that allows the creation of passes to analyze and modify the intermediate code, during the compilation process. Using the LLVM intermediate representation has the benefit of greatly simplifying code manipulation by leveraging LLVM's parsing and rewriting mechanisms. It also provides portability to multiple source languages for free. Simplifying code manipulation is important because it makes writing fault type plug-ins easier, which allows others to more easily apply their fault models using our tool. This approach comes at a small representativeness cost compared to approaches that operate directly on the source code because the compiler expands preprocessor macros, but the intermediate code is a good and platform-independent representation of the source code otherwise [47]. If macros are considered to be critically important, it would be possible to replace them by function calls using demacrofication [76].

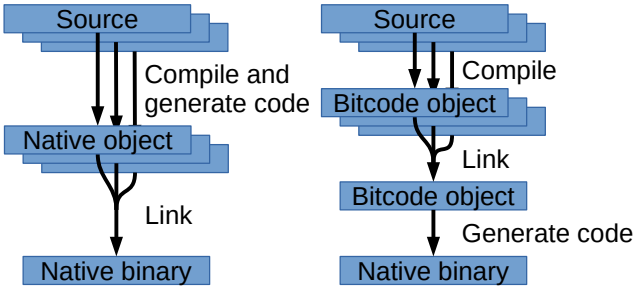

Figure 5.2: Traditional compilation (left) and LLVM with bitcode linking (right)

We will refer to the intermediate representation as *bitcode* from now on, in accordance with the terminology used by the LLVM project itself. LLVM allows linking to be performed directly on the bitcode, using the GNU Gold linker with an LLVM plug-in. Fig. 5.2 compares the traditional build process with source-level or binary-level fault injection (on the left) to our approach (on the right). The main difference in the build process is the fact that linking and code generation have been swapped. This provides the benefit of being able to run the pass over all the code at once, giving access to context information from the entire program.

In this section we will discuss implementation details of our fault injection solution based on the HSFI design. First, we explain how fault candidates are identified and the code is modified to apply them conditionally. Next we discuss how fault candidate markers are inserted in the bitcode and later recognized in the binary. Finally, we consider how binary patches are generated to enable and disable specific faults.

5.4.1 Injecting faults

We first compile the program to bitcode using the Clang compiler. This allows our tool to be applied to any language for which an LLVM front-end is available. The bitcode provides a mostly faithful representation of the original source code, with the main exception of macros being expanded. Next, all bitcode files are linked together using the Gold linker, which makes it easy to integrate bitcode compilation in existing build systems and allows our tool to operate on all code at once, providing maximal context information. Code generation is postponed until after fault injection, causing source-level information to remain available to the fault injector.

The fault injector has been implemented as an LLVM pass, which is invoked on the bitcode using the LLVM optimizer. The fault injector loops through all LLVM instructions in the code and invokes the fault type plug-ins for each instruction. The fault type plug-in then determines whether this particular instruction is a suitable target for fault injection (a fault candidate). For example, an assignment operation could be a target for injecting a “wrong value assigned to variable” fault type and the instruction performing the memory store is considered a fault candidate for that fault

```
1  void setanswer(int *p) {
2    if (p != NULL) {
3      *p = 42;
4    }
5  }
```

Figure 5.3: Code example for basic block cloning

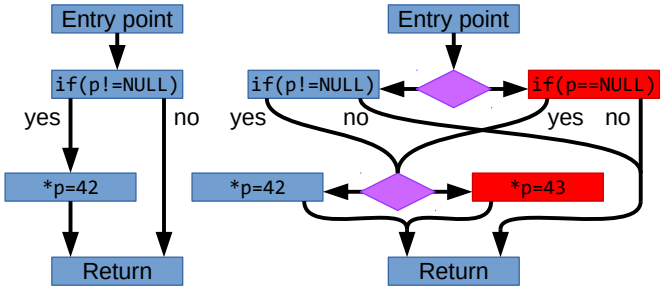

Figure 5.4: Control flow graph of the code example before (left) and after (right) fault injection

type. While scanning through the code this way, the pass writes out a map file that contains information that is potentially relevant to the fault selector later on. This file provides a list of all modules, functions, basic blocks and instructions. Where available, instructions specify which the file and line number in the source code they derive from. Each instruction also has a list of fault candidates and their fault types. Multiple fault types can apply to a single instruction.

The actual fault injection is performed using basic block cloning, which we use in a manner similar to EDFI [47]. A *basic block* is a node in the control flow graph (CFG), defined as a code sequence with exactly one entry point and exactly one exit point. Once the start of the basic block is reached, it will be executed in its entirety, assuming its execution is not interrupted by an exception, a signal or a similar deviation from the CFG that is not visible to the compiler. At the end of the basic block, a new block is selected to be executed. As an example, consider the code in Fig. 5.3. The left-hand side of Fig. 5.4 shows what the CFG looks like for this code example, with the boxes representing basic blocks (the nodes of the CFG) and the lines representing the control flow between them (the edges of the CFG).

To insert faulty code while retaining the original code, we duplicate every basic block that is subject to fault injection. One of the clones remains pristine while a single fault is injected into the other one. Our implementation only supports one fault per basic block, which means one of the fault candidates must be selected at this point. The injected fault is documented in the map file. It would be possible to create further clones to allow simultaneous injections in the same basic block, with the drawback that the code size increases exponentially with the number of

faults injected per basic block. After cloning, the incoming edges are redirected to a new basic block which we will refer to as the fault decision point (FDP). The FDP conditionally selects either of the basic blocks and is needed to embed the new basic block into the CFG. Outgoing edges are cloned, but may be changed by the injected fault. The result is a duplicated CFG with all original code as well as many injected faults (but at most one per basic block in our implementation). The right-hand side of Fig. 5.4 shows our example code after injecting two faults, with the diamonds representing the FDPs.

5.4.2 Fault candidate markers

To be able to identify pristine and faulty basic blocks in the binary, we must insert markers in the bitcode (or, alternatively, the source code itself) that can still be recognized after the code generation step. Unfortunately, debug symbols are not sufficiently accurate for this purpose and they are often even less accurate after optimization. Moreover, they have no concept of pristine or faulty basic blocks, nor of basic block identification. Although we have discussed basic block cloning, which requires the insertion of fault decision points (FDPs), we have not yet indicated what code is inserted in the FDPs. Both successor basic blocks must be reachable to prevent the compiler from optimizing them out as dead code. To solve both issues, we use the FDPs as markers. Each basic block in the original code is assigned a consecutive number. A global variable is inserted in the program and each FDP performs a test that jumps to the faulty basic block if the global variable equals the basic block number and jumps to the pristine basic block otherwise. These tests cannot be optimized away as the value of the global variable is only known at run-time and the compiler must assume it can change during execution, so that the FDP ends up as a piece of code in the binary selecting either the pristine or the faulty basic block. The name of the global variable can be looked up in the symbol table and the binary disassembled to find references to its address, both in memory pointers and immediate values. From our experiments on the x86 architecture, we have verified that every reference to this address is indeed an FDP. Moreover, we were able to find almost all inserted FDPs in the binary (we found a false negative rate of just 0.01 %, see section 5.5.3). In all cases, the basic block number is directly compared against the value of the global variable in a single instruction, which means it is very simple to determine exactly which basic block ended up where. Any deviations from this pattern (for example due to optimizations) as reported by our tool, but we have found none. We conclude that this is a reliable way to pass information across the code generation step without the need to change the compiler itself. Moreover, although the binary code is architecture-dependent, recognizing an address is very simple to implement on other architectures if a suitable disassembler is available.

5.4.3 Binary patching

As explained, the markers in the binary code are comparisons of a global variable with an immediate value, which is the number of the basic block for which the original FDP was added. On x86, such a comparison set the flags registers to control a later conditional jump. The compiler is allowed to insert instructions that do not affect the relevant flags between the comparison and the jump to optimize the code but we have found no instance of this happening, presumably because the FDP is a separate basic block and there is therefore no code available to insert. In all instances we found a compare ("cmp") opcode directly followed by either a jump-if-equal ("je") or jump-if-not-equal ("jne") opcode. To enable the pristine code and make the faulty code unreachable, our binary pass simply replaces the comparison and jump-if-equal instructions with no-operation instructions ("nop"), while jump-if-not-equal instructions are replaced with unconditional jumps ("jmp"). This can all be done in place. Because the patching is so simple, it would be easy to port to other architectures if the following assumptions hold: there exist no-operation instructions with a sizes that can be combined to the size of a comparison and a conditional jump and the unconditional jump instruction is no longer than a conditional jump. Our program issues a warning for any unrecognized code sequences, which means it is easy to debug and ensure every FDP is indeed patched.

While patching the binary to assure that only the pristine code is accessible, the binary pass also writes binary patch files that can be used to switch individual blocks from pristine to faulty and vice versa. These patches simply reverse the cases where conditional jumps are replaced with no-operation instructions and the cases where they are replaced with unconditional jumps. For most basic blocks, only up to six consecutive bytes in the binary need to be changed to switch between correct and faulty execution, which makes patching nearly instantaneous.

As a performance optimization, we assign branch weight metadata to the FDPs that indicate that the pristine block is very likely to be selected. As a consequence, LLVM places all faulty basic blocks at the ends of functions. This means that in practice no extra jumps are inserted in the common code path and cache pollution is limited. However, some slowdown is still to be expected because the compiler has to set up register usage in a way that works for both the pristine and the faulty basic block. The presence of no-operation instructions and the expansion of the total code size are also expected to have a small performance impact.

5.5 Evaluation

In this section we will evaluate our system in terms of performance, both run-time performance and the total time it takes per experiment, and marker recognition accuracy. We have chosen to evaluate HSFI with operating systems, because they are large pieces of software (making scalability to large code bases important) for which reliability is of particular importance. This is demonstrated by the fact that there

is much prior work evaluating operating system stability in the face of faults (for example [39; 54; 125; 71; 120; 119; 33; 98]). We have chosen to use the Linux and MINIX3 operating systems to evaluate our system. Linux is widely used, particularly in situations where reliability is especially important such as servers. We use LLVMLinux [2], which allows Linux to be built with LLVM bitcode linking. We use MINIX3 because it has a modular design, making it structurally different from Linux and demonstrating that our approach is applicable to a wide range of operating systems. Moreover, it is designed specifically for reliability and comes with support for LLVM bitcode linking out-of-the-box. As workloads, we use Unixbench [6] and the MINIX3 POSIX test suite. Unixbench is widely used for performance measurements of Unix-like operating systems The MINIX3 POSIX test set is an example of a regression test workload that aims for high coverage, which is important to perform representative fault injection experiments [118]. Unfortunately it contains some tests that are specific to MINIX3. We have omitted those tests on both systems to keep the results comparable. We test all combinations of operating systems and workloads, with the operating system itself instrumented for fault injection.

We performed the compilation of the target systems as well as the tests themselves on Intel Xeon E5-2630 machines with 16 cores at 2.40 GHz and 128 GB of memory, running 64-bit CentOS Linux 7. We used the lto-3.11 branch of LLVMLinux and the llvm_squashed branch of MINIX3. The operating systems were virtualized using QEMU 2.3.0 [5] with KVM enabled. We virtualized the operating systems because this is the most convenient way to perform fault injection experiments on operating systems. We modified QEMU to provide a hypercall interface that allows the guest to report its progress by writing to a fixed physical memory address. This allows times to be measured and logged on the host, with the benefit that the measurements are more accurate and the logs are retained even if the guest system crashes (which is likely to happen when faults are injected into the operating system).

We compare our new approach against binary-level fault injection, source-level fault injection and EDFI [47] with reduced linking overhead as proposed by Van der Kouwe et al. [120]. To achieve a fair comparison, we have taken an optimistic approach of estimating the overhead of competing techniques. To evaluate binary fault injection, we use an uninstrumented binary and consider the build time only once for all experiments. This means we assume run-time overhead and instrumentation time to be zero. We achieve the same for source-level fault injection by measuring run-time performance on the uninstrumented binary and measuring the rebuild time by changing the last modified date of a single small source file and calling *make*. This means we assume run-time overhead and instrumentation time to be zero and recompile time to be minimal. These decisions allow us to compare the performance of our solution against the fastest possible alternatives.

The fault model we use is taken from EDFI [47], which in turn is based on G-SWFIT [41]. Table 5.1 provides an overview for the code size and number of fault candidates found for both targets. Most numbers are reported by our compiler pass. Here we included only the code that is actually visible to the compiler; that is, we ex-

Table 5.1: Code metrics for the target programs

	Linux	MINIX3
Modules loaded during execution	1	28
Functions	34578	3389
Basic blocks	313028	41884
LLVM instructions	1817389	244775
Fault candidates	2281880	343176
Lines of code	6775204	1128797
Binary code size uninstrumented (KB)	646	995
Binary code size EDFI (KB)	2230	3158
Binary code size HSFI (KB)	1553	2525

clude source files not compiled and code disabled by preprocessor conditionals and we count static libraries used by multiple modules only once. The number of lines of code is counted by the Cloc utility [3] and includes all code files that are not disabled due to configuration or architecture. The binary code size is computed as the sum of executable sections in the ELF files, so in this case static libraries are counted twice to accurately represent the memory overhead introduced by our solution.

In this section, we first discuss the run-time performance of the various approaches. Next, we present the build times and use those to compare the average time taken for each experiment. Then, we show how well our marker detection approach does in terms of false positives and false negatives. Finally, we discuss some potential threats to validity that might influence our results presented here.

5.5.1 Run-time performance

Our approach introduces a run-time overhead compared to binary-level and source-level approaches because basic block cloning is performed throughout the program. Although the branches are fixed to avoid any extra conditional jumps, this does affect code generation and the larger code size also affects caching. We have measured the time taken by the system to boot and the time taken to run the MINIX 3 test set. The results are presented in Table 5.2 and Table 5.3 respectively. The "system CPU usage" indicates how much of the time the CPU has been running system code as reported by the "time" utility (system time divides by real time). The numbers presented are medians over 32 runs and the numbers between parentheses are sample standard deviations. The high standard deviation on the binary/source MINIX 3 boot is due to an outlier where the boot time was excessively long; the boot times are consistent otherwise and the result is not affected because the median reduces the influence of outliers. The numbers show that our approach generally incurs a modest slowdown that is at most about 10% compared to binary and source approaches. EDFI show considerably larger slowdowns. It is important to note that our instrumentation only affects system code. The fact that MINIX 3 reports larger slowdowns seems to be due to the fact that more of the time is spent running system

Table 5.2: Boot time (lower is better, std. dev. in parentheses)

		Binary/source		EDFI		HSFI	
Linux	Boot time (s)	7.9	(1.1)	11.5	(1.1)	8.7	(1.1)
	Slowdown (%)			45.6		10.5	
MINIX 3	Boot time (s)	9.5	(9.5)	24.6	(6.7)	9.7	(0.8)
	Slowdown (%)			159.8		2.0	

code, which may mean that MINIX 3 is less efficient at handling system calls due to its modular design. This is consistent with the fact that system CPU usage is higher for MINIX 3. Based on the total run times and the percentage of time spent by the CPU executing system code, we made an estimate of the overhead while running system code, which is included in Table 5.3. As expected, it shows considerably larger overhead for system code than for the time spent executing the test set as a whole. Nevertheless, HSFI still has a reasonable overhead compared to source and binary approaches (below 20 % for both Linux and MINIX 3) while EDFI shows even larger overheads than before.

While the MINIX 3 test set gives an indication of the slowdown for a full high-coverage workload, it runs at low CPU usage and does not stress the system very much. To provide an indication of system performance under load, we have run Unixbench 32 times on both systems. The resulting performance numbers are shown in Fig. 5.5 and Fig. 5.6. These numbers are normalized to 100 for the uninstrumented system. They represent the number of operations that can be performed in a fixed time, so that higher numbers indicate better performance. Unixbench scores are based on averages where the 33% worst results are left out to reduce the influence of outliers. The error bars give the sample standard deviation based on all experiments. It should be noted that the *dhry2reg* and *whetstone-double* benchmarks do not use the operating system, so the lack of slowdown is to be expected. The other subtests all give similar results. To compare the overall results, we have taken the geometric means of the scores for those other tests. EDFI averages 26.6 on Linux and 8.6 on MINIX 3, showing a very large slowdown of respectively about 3.8x and 11.6x. The fact that the slowdown is higher on MINIX 3 suggests that it handles system calls less efficiently (as was also seen with the test set), executing more (instrumented) operating system code for each call. These are substantial slowdowns that cause stress-test experiments to take considerably longer. HSFI averages at 92.6 (Linux) and 80.0 (MINIX 3), so the slowdown is just 1.1x and 1.3x respectively. This means run-time overhead is just a minor factor compared to EDFI.

5.5.2 Time taken per experiment

To determine how long each experiment takes, we must measure not just the run time but also the time needed to prepare the experiment. Different experimental setups require different steps to be performed for each experiment. Table 5.4 shows

Table 5.3: Run time and overhead on MINIX 3 test set (lower is better, std. dev. in parentheses)

		Binary/source		EDFI		HSFI	
Linux	Run time (s)	369.6	(12.4)	457.6	(17.1)	372.6	(14.2)
	Slowdown (%)			23.8		0.8	
	System CPU usage (%)	7.8	(0.4)	25.1	(0.7)	8.6	(0.4)
	System overhead (%)			298.5		11.5	
MINIX 3	Run time (s)	392.9	(11.0)	1096.7	(28.4)	410.1	(5.8)
	Slowdown (%)			179.1		4.4	
	System CPU usage (%)	21.7	(0.5)	23.0	(2.9)	24.6	(0.6)
	System overhead (%)			196.9		18.5	

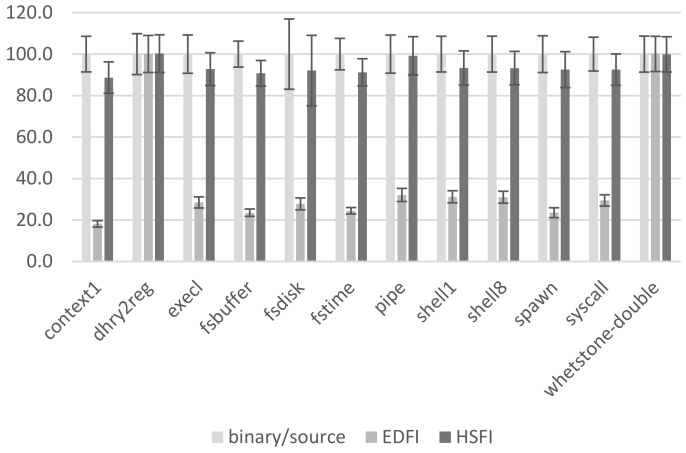

Figure 5.5: Unixbench performance on Linux (higher is better)

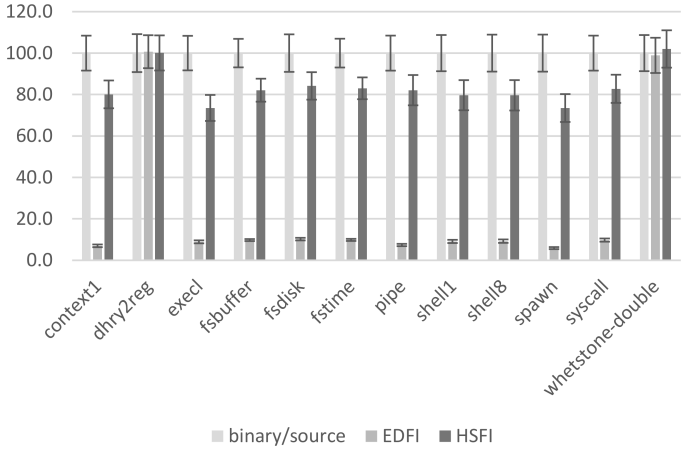

Figure 5.6: Unixbench performance on MINIX 3 (higher is better)

Table 5.4: Build time to prepare experiments in seconds (std. dev. in parentheses)

	Needed for				Time taken (s)			
	Binary	Source	EDFI	HSFI	Linux		MINIX 3	
Rebuild	once	often	rarely	rarely	3798.5	(216.3)	232.5	(13.8)
EDFI pass	never	never	rarely	never	670.0	(15.3)	174.0	(4.5)
HSFI passes	never	never	never	rarely	580.6	(12.7)	214.7	(4.9)
Patch application	never	never	never	often	17.3	(0.1)	0.0	(0.0)

an overview. For binary-level approaches, the target program only needs to be compiled once because faults are injected directly into the binary. For source-level approaches, rebuilding the system introduces very large overhead. Every time a source file is changed, the build system must go through the source tree, recompile the appropriate file(s) and eventually relink the entire program. In the case of Linux, some postprocessing on the binary is also required to perform linker tricks such as support for kernel modules. Both EDFI and HSFI require the system to be fully built only a few times for many experiments, namely when all faults injected on the previous build have been tested. In addition, they require their passes to be applied to the system to perform fault injection and (in the case of HSFI) generate patch files from the binary. For HSFI, the binary must be patched for every experiment. In principle this is near-instant as it requires only modifying a few bytes at a known file offset. However, in case of Linux it takes longer because we also have to rebuild the boot image, which includes a compressed version of the kernel. The numbers show that rebuilding introduces a large amount of overhead, especially for a large program such as Linux. Although the passes also take some time to run, this is not needed for every experiment.

To compare how efficient the approaches are overall, we need to compute the average time taken per experiment. Here, it is very important the know how often the “rarely” from Table 5.4 actually is. With our implementation, many faults can be injected with one compilation, but just one for each basic block. This means the number of largest number of selected faults in a single basic block determines how often we must recompile. It is not clear a priori how large this number will be and the probability distribution is fairly complicated. If we assume a fixed number of faults is selected for injection and equal probability of selection for each fault candidate, the number of faults selected in a basic block is binomially distributed but the number of faults selected per basic block is not independent between basic blocks; a large number selected in one leaves fewer for the others. For this reason, we took the distribution of fault candidates per basic block for both target systems and performed a Monte Carlo simulation[13] to determine the number of rebuilds needed for various numbers of faults to inject. For this simulation, we assume that each fault candidate is equally likely to be selected. The results are shown in Fig. 5.7 (Linux) and Fig. 5.8 (MINIX 3). There is clearly a linear relationship between the number of faults injected and the number of rebuilds needed, although the ratio between the two

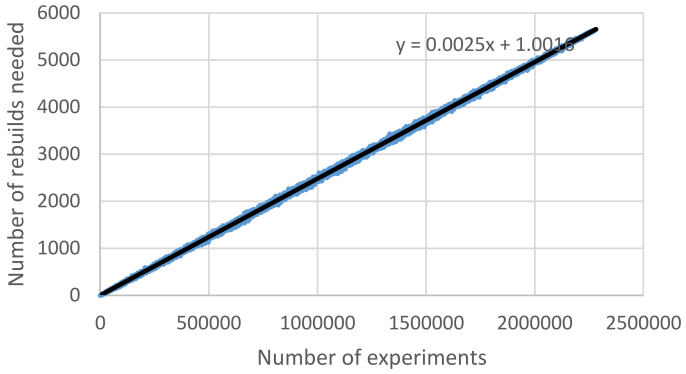

Figure 5.7: Monte Carlo simulation of rebuilds needed for Linux with HSFI

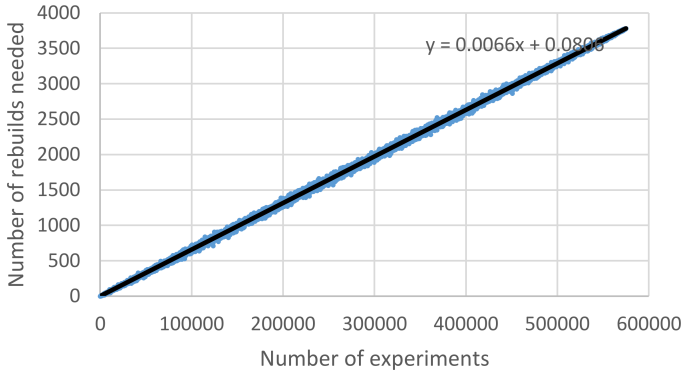

Figure 5.8: Monte Carlo simulation of rebuilds needed for MINIX 3 with HSFI

differs between the operating systems. This difference is caused by the distribution of fault candidates over basic blocks; if there are relatively more large basic blocks with many fault candidates, more recompiles should be needed on average.

Now that we know the boot time, the slowdown of the workload, the time it takes to rebuild and instrument the system and how often a rebuild is needed, we can compute the average time taken per experiment by adding up the expected values of the numbers. The results are shown in Fig. 5.9 (Linux) and Fig. 5.10 (MINIX 3). We show both the slowdowns from the MINIX 3 test set to represent a high-coverage regression test workload and Unixbench to represent a stress test. Source-level fault injection is suitable on smaller code bases with large workload durations, but introduces substantial rebuild overhead otherwise. EDFI, on the other hand, works well for large code bases but only in case the workload duration is short and the workload is not too intense. With stress tests or long-running workloads the run-time overhead becomes a problem. HSFI, on the other hand, always achieves performance close to binary-level fault injection, regardless of the target system and workload.

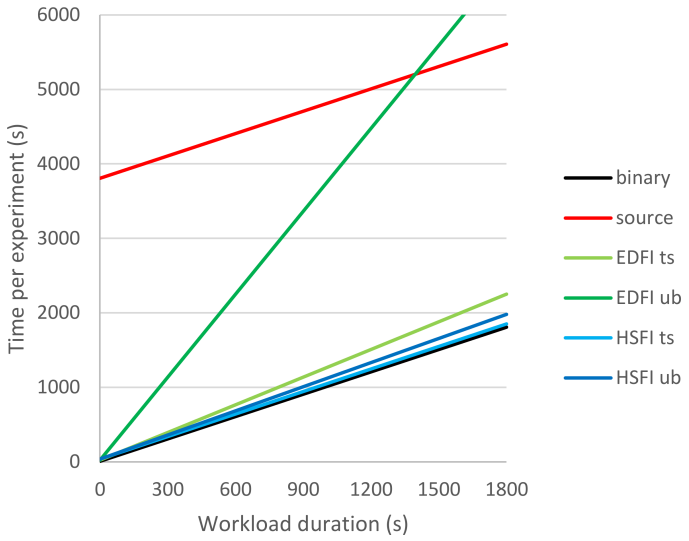

Figure 5.9: Time taken per experiment depending on the workload duration for Linux; ts=test set, ub=Unixbench

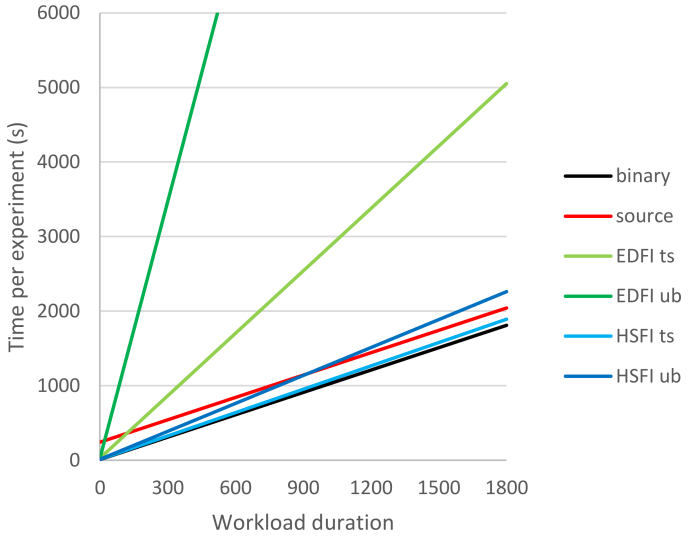

Figure 5.10: Time taken per experiment depending on the workload duration for MINIX 3; ts=test set, ub=Unixbench

5.5.3 Marker recognition

To determine whether our approach yields false positives and false negatives, our tool reports any unexpected code sequences involving any references to the marker variable. We have also cross-referenced the patches, the map files and the output

from the *objdump* utility on the final binary. We have not found any instances of false positives; every reference to the marker variable was indeed a valid marker. In case of Linux we found that out of the 313028 basic blocks for which markers were inserted, 125 were not found by our binary pass. Out of these, 93 do not seem to be present in the binary at all and were presumably eliminated by the optimizer. The remaining 32 were present but not found, so those should be considered false negatives. In all those cases, this seems to be because Dyninst did not recognize some instructions included in inline assembly, causing it to produce an incomplete control-flow graph. The result is a false negative rate of just 0.01 % for Linux. In case of MINIX 3, all compiled modules together contain 219261 basic blocks with markers (with some occurring multiple times due to static libraries). In 6602 cases, the marker was not found. We found that just 25 of these are actually false negatives, the remainder not ending up in the final binaries. All 25 false negatives were in the kernel module, which contains 5706 basic blocks. Like for Linux, the false negatives were found near assembly instructions not recognized by Dyninst. The false negative rate is 0.01 % for all of MINIX 3 and 0.38 % for the kernel. Given that we have found no false positives and very few false negatives, we conclude that our marker mechanism is very effective in mapping binary code to the original basic blocks.

5.5.4 Threats to validity

There are several factors that might influence the results presented here. First, we measured performance when running on a virtual machine, namely QEMU with KVM. This is a deliberate choice because fault injection experiments on operating systems are likely to be performed in virtual machines in practice because this makes the experiments much easier to set up. Times measured in inside the virtual machine may not be as accurate as on the host machine. For this reason we implemented a hypercall that causes the current status including timestamp to be logged on the host. We have used these timestamps to get our performance results. The only exception is CPU usage, which has been measured on the guest. Any inaccuracies here would not affect the comparison. A second threat is the fact that we only measured performance without enabling any faults. If a fault is hit, it could cause the system to crash, cutting the test short, or hang, causing the test to last until some set timeout. We decided not to consider this as the behavior depends strongly on the exact fault model, something which is out of scope for this paper. A third threat is the fact that we have not measured any actual binary-level or source-level fault injection tools. However, we made an optimistic estimate which means that they should not be able to perform any faster than what we measured. As a result, the conclusions would either remain unchanged or become even more favorable to our proposed approach. Finally, it is possible that we missed false negatives in cases where basic blocks were duplicated by the optimizer, for example due to inlining or loop unrolling, and at least one instance was detected. Given the very low number of false negatives overall, it is unlikely this has happened on a substantial scale.

5.6 Limitations

The evaluation confirms that our new approach is able to perform fault injection with the benefits of source-level fault injection at a performance close to that of binary-level fault injection. That said, some limitations of this approach still need to be considered. One fundamental limitation that it shares with other source-based approaches is that the source code of the target program must be available. If no source code is available, a binary-level approach must be used. Another limitation is the fact that our implementation requires the target program to be ported to LLVM with bitcode-based linking. Fortunately, Clang mostly provides a drop-in replacement for the more widely used GCC and the GNU Gold allows linking to be performed on bitcode files in the same way machine code object files are normally linked together. This means that often the configure script is versatile enough to allow LLVM with bitcode linking to be used, but in other cases small build system changes may be required. Moreover, this is not a fundamental limitation of our design but rather an implementation choice. If the design were implemented working directly on the source code, this step would not be needed. Another limitation is the fact that, due to the use of the intermediate representation, some information from the source code does get lost. However this mostly affects preprocessor macros and demacrofication [76] allows those to be replaced by function calls, which are accurately represented in the bitcode. Moreover, this is again just a limitation of our implementation because the design could be implemented modifying the source code directly. Finally, our implementation only allows a single fault to be selected for each basic block. This could be solved within the design by creating more clones of the basic blocks to be able to select one of several combinations of faults. The drawback would be that the code size of the basic block would grow exponentially in the number of faults injected. The number of rebuilds needed has been shown to be very low even when selecting just one fault candidate per basic block at a time, so such a change is unlikely to be worthwhile unless the fault model requires it.

5.7 Related work

In this section we will consider the use of various software fault injection implementations in research, papers assessing the representativeness of fault injection, and finally other research about improving the performance of fault injection. We do not repeat the classification of the various types of fault injection (which has been discussed in section 5.2) and will instead focus specifically on software fault injection, which is the approach used in this paper.

5.7.1 Use of software fault injection

Software fault injection is widely used in the research literature for many different purposes. For binary-level fault injection, G-SWFIT [41] stands out for being reused

for many purposes. It has been used, amongst others, to compare binary-level injection against source-level fault injection [31] and interface-based faults models [91], for failure prediction [57], and to evaluate suitable fault loads for large systems[30]. Another tool has been produced to quantify and improve fault tolerance for the RIO file cache [98], which has been reused to study impact of faults on Linux kernel [125] and the isolation properties of MINIX 3 device drivers [54]. Duraes et al. [39] have implemented this approach to evaluate the impact of faults in drivers on system. A different approach to binary-level fault injection is found in XEMU [17], which uses QEMU to perform binary rewriting to inject faults.

As for source-level software fault injection, Jia and Harman [60] provide a literature review which shows how the use of this technique has greatly increased over time. Recently SAFE [96] has gained popularity and has been reused for certification of software [33], to evaluate the relationship between fault types and software metrics [34], to compare monitoring techniques [29], for evaluating whether Linux crashes in response to faults [71], and for evaluating the impact of running multiple fault injection experiments simultaneously [123]. SCEMIT [84] is notable for being implemented as a compiler pass rather than modifying the source code directly. EDFI [47] is implemented deeper in the compiler, performing fault injection on the intermediate representation after linking as we do. This still provides most benefits of source-level injection while allowing for easy build system integration and access to the entire program. This tool has been reused to determine how common silent failures are [119], to measure the impact fault load distortion [118], and to compare operating system stability [120].

Both source-level and binary-level approaches are widely used to improve software reliability. Our design provides a hybrid of the two, offering access to the source code or intermediate representation when injecting the faults, while achieving performance close to that of binary-level approaches through a binary-level pass selecting which of the injected faults to enable.

5.7.2 Fault representativeness

There are several papers providing an overview of representative fault types based on bugs found in real software [27; 41; 70]. Natella et al. [96] consider which fault locations are most representative of residual faults that are likely to remain in the software in production systems. These works are orthogonal to our approach, which applies equally to any fault model based on software mutation testing. Our implementation allows the desired fault model to be implemented quickly, using the code rewriting abilities of the LLVM framework.

Several papers compare the representativeness of different fault models or injection techniques. Of particular importance is Cotroneo et al. [31], who show that source-level fault injection is more representative of real software faults than binary-level injection. Lanzaro et al. [78] and Moraes et al. [91] show that interface-level faults do not accurately mimic software faults. [88] shows that hardware faults

poorly represent software faults. Daran et al. [37] and Andrews et al. [10] perform source-level fault injection and show that their approach is representative of real faults. These results supports the case for preferring source-level fault injection if real software faults are to be emulated.

The influence of other factors on fault representativeness has also been considered. Winter et al. [122] argue that injection of multiple simultaneous faults is a more realistic fault model than the traditional approach of injecting just one fault at a time. Our design allows for arbitrary combinations of faults, while our implementation allows almost all combinations. Kikuchi et al. [71] and Van der Kouwe et al.[118] show that using an improper workload leads to poor fault representativeness. As our evaluation shows, our approach is suitable for both high-coverage and stress-testing workloads and beneficial for a wide range of workload durations, allowing flexibility to select the most suitable workload.

5.7.3 Fault injection performance

Many previous papers have worked on improving the performance of fault injection. One common theme is speeding up fault space exploration to try the most relevant failures first. Banabic et al. [15] use fitness-guided exploration to search for high-impact faults that impact system recovery code. FATE [50] searches the most different failure scenarios first. Prefail [63] prunes the search space by finding which (combination of) faults is most promising to uncover bugs. Li et al. [82] use static analysis to determine which faults are most likely to cause long latency crashes. Although these papers are based on different fault models than ours (interface-level and hardware faults), variations that deal with software faults would be complementary to our approach and could help to speed up fault injection experiments even further.

There has also been some work that, like ours, proposes ways to speed up the experiments themselves. Winter et al. [123] investigate parallel fault injection, performing multiple experiments simultaneously on the same machine. This approach is complementary to ours. Van der Kouwe et al. [120] use EDFI to inject multiple faults at compile time and select a single one at run-time. This approach is closest to ours, but it introduces considerable run-time overhead because the decision to inject must be made at run-time. Our proposed solution performs better in all cases.

5.8 Conclusion

Although source-level software fault injection is a good technique to inject faults that are representative of software faults, it suffers from poor scalability to large code bases. In this paper, we presented a design that solves the scalability issue by reducing the number of times the target program needs to be rebuilt. We also built an implementation of this design, which can easily be adjusted for different fault models and different programming languages, which has most source-level information

available to the fault injection logic, and which can inject software faults fully automatically. Although porting the binary pass to different architectures would require some effort, this effort is greatly reduced due to its simplicity and the fact that the binary pass is not involved in the actual fault injection. Although our approach incurs run-time overhead compared to source-level fault injection, our evaluation shows that this overhead is reasonable even for stress-testing workloads. As a consequence, the total time needed per experiment is considerably lower for large code bases, which means that statistically sound fault injection results can be reached at lower cost. Our approach also outperforms another solution for source-level injection with reduced rebuild overhead by a wide margin due to a much lower run-time overhead. We conclude that our solution retains the benefits of source-level source injection while achieving performance close to that of binary-level approaches, providing the best of both worlds.

6 Conclusion

Software fault injection is an essential technique to be able to test and improve fault tolerance and is widely used in computer science research, but there is need for improvement to allow it to be performed more systematically and to allow it to gain widespread use outside the academic community as well. This dissertation provides several improvements that help us get closer to this goal.

In Chapter 2 we raised the issue of fault load distortion and defined the concept of fidelity to formalize its impact on fault injection representativeness. Our experiments demonstrate that fault activation correlates with factors that are important in defining the fault model, such as fault types and code locations with specific properties. In particular, it shows that low coverage and extreme execution counts due to loops distort the faults that are activated with respect to the fault model. It is therefore important to design workloads carefully to maximize coverage and to mimic job sizes encountered in production. Moreover, it is important to report coverage and do so in the units that are most relevant given the goal of the fault injection experiment, because there is a meaningful difference between (for example) coverage expressed in terms of basic blocks and coverage in terms of fault candidates. The results also have implications for fault model specification. Software has different fault type distributions depending on its purpose, which means it is better to specify the model in terms of the number of fault candidates available per fault type rather than injecting a fixed number of faults for each fault type. By showing the relevance of fault load distortion, demonstrating its importance empirically and providing suggestions to minimize its impact, we hope to increase the quality of the results from future software fault injection experiments.

In Chapter 3 we presented a widely applicable methodology to identify silent failures in fault injection experiments and presented a tool that can do so in a fully automated way. Our approach consists of comparing externally visible behavior between a golden run using the original program and a faulty run where a fault has been injected in the program. The main challenge is to prevent nondeterminism from in-

troducing false positives, which we succeed in doing for the programs we tested. We performed an experiment which demonstrates that silent failures are common. The good news is that the numbers are consistent with field data about silent failures in real-world systems [43], which suggests that software fault injection is a suitable approach to investigate their impact. The bad news is that the widely relied on fail-stop assumption does not hold. Experiments based on this assumption risk producing unsound results and fault-tolerance mechanisms assuming fail-stop behavior are unable to deal with what we have shown to be an important class of failures.

In Chapter 4 we presented an approach to compare the stability of different operating systems. The main contribution is the definition of stability, a methodology to measure it and an implementation that we use to demonstrate how it can be used to compare operating systems. Rather then simply comparing the number of crashes when performing fault injection experiments, we make the point that one cannot expect the operating system to function correctly if faults are injected. Instead, we use a pre-test and a post-test to determine whether running a workload that triggers the fault causes the system to become unstable afterwards. This allows for a meaningful comparison of the effectiveness of fault-tolerance mechanisms in operating systems, even if they are structured differently. In addition, our approach improves scalability of representative fault injection to large code bases such as the Linux kernel, though at the cost of substantial run-time overhead. The idea behind this approach serves as a basis that is further improved on in Chapter 5. Another contribution is the fact that our methodology provides a way to estimate how many fault injection experiments are needed to be able to draw statistical conclusions. Together, these elements provide a major step towards the systematic evaluation of fault-tolerance mechanisms in operating systems.

Finally, Chapter 5 presents a methodology to run software fault injection experiments in such a way that source-level information is available to the fault injector while avoiding the slowdown caused by the need to recompile the (potentially very large) source code for each experiment. It offers the best of both worlds: the representativeness of source-level approaches and performance close to that of binary-level fault injection. In practice, this means high-quality fault injection experiments on large code bases can be run at lower cost. In contrast to the solution used to speed up fault injection experiments on large code bases in Chapter 4, this approach has considerably lower run-time overhead, which is necessary to achieve performance close to that of binary-level fault injection. Moreover, it can do so even for stress-testing and long-running workloads. The key is to create markers before code generation that can be recognized with very high accuracy (no false positives, only 0.01 % false negatives) in the resulting binary. Our evaluation compares this approach against alternatives and shows that it is the only one that can reach performance close to that of binary-level fault injection in all cases.

Taken together, the contributions made in this dissertation allow software fault injection to be applied to evaluate fault-tolerance mechanisms in a way that is sound, low-cost even for large code bases such as operating systems and that provides state-

of-the-art fault representativeness. By making it easier to measure fault-tolerance accurately, we hope to contribute to a future where reliability becomes a key consideration in software development and the software crashes we have all become so accustomed to will eventually become a thing of the past.

Future Directions

Although this dissertation advances the state of the art in software fault injection in various ways, there remains much room for further improvement. In this section, we discuss possibilities to expand on the presented work.

The work in Chapter 2 could be taken further by developing a quantitative metric of fidelity. Unfortunately, it is hard to do this in terms of the input fault load and output fault load directly because the fat tail of the execution count distribution would cause overrepresentation of the code locations that are executed very often. To obtain a proper metric would require a study of the exact impact of repeatedly executing a fault. How likely is it for repeated activation of the same fault to trigger new failure behavior? This depends on both the fault type and the context. It seems to us that a viable solution would be to go beyond the output fault load and consider actual failures. The work presented in Chapter 3 could serve as a basis to be able to detect these failures even if they do not cause fail-stop behavior.

Another direction to extend Chapter 2 would be to compare against systems in production use to determine how well fault injection techniques manage to approximate those situations. It would be very useful to compare the output fault load against activation of real faults on production workloads. It would be possible to record a workload on a system in production use and replay it while recording fault activation. The main difficulty would be to find a sufficient number of real faults to reach statistical conclusions.

The methodology presented in Chapter 3 to detect silent failures could be improved to make it even more widely applicable. Most importantly, although we have shown that we can successfully deal with the nondeterminism present in the programs we tested, there are some scenarios where further steps are needed. For example, in both server programs and operating systems, requests are initiated by external processes or hardware. If there are multiple sources of requests or those sources themselves issue requests in a nondeterministic order, our approach would be unable to decide which externally visible behavior is linked to which request, causing false positives because the resulting behavior differs from run to run. White-box methods could be used to link requests to responses, allowing silent failures to be detected in such scenarios. Alternatively, the entire system (including the processes that make the requests) could be emulated using record-replay techniques [108; 9; 102; 52; 109; 77] to force determinism.

Another possibility to extend Chapter 3 would be to take the analysis one step further and consider not just externally visible behavior but also the internal state.

This could be done by using either taint tracking [38] or a more lightweight approach that only instruments memory allocations [46] where the memory image is compared at the end of the experiment. This would allow us to uncover silent data corruption even in cases where the workload is unable to activate the point where the corruption affects externally visible behavior.

While the methodology proposed in Chapter 4 provides an important step towards being able to systematically evaluate operating system stability, the analysis of the system state could be improved to obtain more information about the source of instability. The approach that is used to determine whether the system crashed or not has the benefit of being black-box (and therefore easy to apply to many different systems) but the information provided is fairly crude. By using knowledge of the operating system it would be possible to distinguish various causes of crashes such as different types of hardware exceptions, assertion failures, deadlocks, and hanging components. In a multiserver system like MINIX 3 it might also be valuable to identify the module that caused the crash, which may not be the same module that the fault was injected in. In addition, introspection into the state of the operating system at the time of the post-test might reveal more information about why the system became unstable. With these additions, we could take the step from comparing stability to providing a tool that helps to improve a lack of stability.

The evaluation in Chapter 5 helps in taking an informed decision for one of the available software fault injection techniques based on performance and the availability of source-level information to the fault injector. Although Cotroneo et al. [31] have shown that source-level fault injection provides better fault representativeness than binary-level fault injection and Daran et al. [37] and Andrews et al. [10] have shown that source-level fault injection is representative of real faults, there is still no comparison of the various ways source-level fault injection can be performed: directly on the source code or in the compiler, using the compiler's intermediate representation. We have used the latter due to its superior portability and ease of code rewriting, allowing it to be extended with new fault models easily. An explicit comparison between the fault representativeness of both solutions would help decide whether this approach is indeed appropriate. Going further, it would be even better if a quantitative metric of fault representativeness were available to indicate whether a more representative approach is worthwhile in a particular situation and allow the choice in this trade-off to be made in a fully informed manner.

References

[1] Apache benchmark (AB). `http://httpd.apache.org/docs/2.0/programs/ab.html`.

[2] LLVM linux. `http://llvm.linuxfoundation.org/index.php/Main_Page`.

[3] cloc - count lines of code. `https://github.com/AlDanial/cloc`.

[4] External visibility of system calls. http://www.cs.vu.nl/

[5] Qemu - open source processor emulator. `http://wiki.qemu.org/Main_Page`.

[6] byte-unixbench. `https://github.com/kdlucas/byte-unixbench`.

[7] Clauset A., C. Rohilla Shalizi, and M.E.J. Newman. Power-law distributions in empirical data. `http://arxiv.org/abs/0706.1062`.

[8] Arnaud Albinet, Jean Arlat, and Jean-Charles Fabre. Characterization of the impact of faulty drivers on the robustness of the Linux kernel. In *Proc. of the Int'l Conf. on Dependable Systems and Networks*, page 867, 2004.

[9] Gautam Altekar and Ion Stoica. ODR: Output-deterministic replay for multicore debugging. In *Proc. of the 22nd ACM Symp. on Operating Systems Principles*, pages 193–206, 2009.

[10] James H Andrews, Lionel C Briand, and Yvan Labiche. Is mutation an appropriate tool for testing experiments? In *Proceedings of the 27th international conference on Software engineering*, pages 402–411. ACM, 2005.

[11] J. Arlat, J.-C. Fabre, and M. Rodriguez. Dependability of COTS microkernel-based systems. *IEEE Trans. Computers*, 51(2):138–163, February 2002.

[12] Jean Arlat, Yves Crouzet, and Jean-Claude Laprie. Fault injection for dependability validation of fault-tolerant computing systems. In *Proc. of the 19th Int'l Symp. on Fault-Tolerant Computing*, pages 348–355, 1989.

[13] Søren Asmussen and Peter W Glynn. *Stochastic simulation: Algorithms and analysis*, volume 57. Springer Science & Business Media, 2007.

[14] A. Avizienis. The n-version approach to fault-tolerant software. *IEEE Trans. Softw. Eng.*, 11(12):1491–1501, December 1985.

[15] Radu Banabic and George Candea. Fast black-box testing of system recovery code. In *Proc. of the 7th ACM European Conf. on Computer Systems*, pages 281–294, 2012.

[16] J. H. Barton, E. W. Czeck, Z. Z. Segall, and D. P. Siewiorek. Fault injection experiments using FIAT. *IEEE Trans. Comput.*, 39(4):575–582, 1990.

[17] Markus Becker, Daniel Baldin, Christoph Kuznik, Mabel Mary Joy, Tao Xie, and Wolfgang Mueller. Xemu: an efficient qemu based binary mutation testing framework for embedded software. In *Proceedings of the tenth ACM international conference on Embedded software*, pages 33–42. ACM, 2012.

[18] E. W. Biederman. Multiple instances of the global Linux namespaces. In *Proc. of the Linux Symposium*, 2006.

[19] Choi Byoungju and Aditya P Mathur. High-performance mutation testing. *Journal of Systems and Software*, 20(2):135–152, 1993.

[20] C. Cadar and P. Hosek. Multi-version software updates. In *Proc. of the Fourth Int'l Workshop on Hot Topics in Software Upgrades*, pages 36–40, June 2012.

[21] Cristian Cadar, Daniel Dunbar, and Dawson 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*, pages 209–224, 2008.

[22] George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. Microreboot — a technique for cheap recovery. In *Proc. of the Sixth USENIX Symp. on Operating Systems Design and Implementation*, pages 31–44, 2004.

[23] Joao Carreira, Henrique Madeira, and Joao Gabriel Silva. Xception: A technique for the experimental evaluation of dependability in modern computers. *IEEE Trans. Softw. Eng.*, 24(2):125–136, February 1998.

[24] Subhachandra Chandra and Peter M. Chen. Whither generic recovery from application faults? a fault study using open-source software. In *Proc. of the Int'l Conf. on Dependable Systems and Networks*, pages 97–106, 2000.

[25] D. Chen, G. Jacques-Silva, Z. Kalbarczyk, R.K. Iyer, and B. Mealey. Error be-

havior comparison of multiple computing systems: A case study using Linux on Pentium, Solaris on SPARC, and AIX on POWER. In *Proc. of the Pacific Rim Int'l Symp. on Dependable Computing*, pages 339–346, December 2008.

[26] G.S. Choi and R.K. Iyer. FOCUS: an experimental environment for fault sensitivity analysis. *IEEE Trans. Comput.*, 41(12):1515–1526, 1992.

[27] 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*, page 304, 1996.

[28] J. Christmansson, M. Hiller, and M. Rimén. An experimental comparison of fault and error injection. In *Proc. of the 9th Int'l Symp. on Software Reliability Engineering*, page 369, 1998.

[29] Marcello Cinque, Domenico Cotroneo, Raffaele Della Corte, and Antonio Pecchia. Assessing direct monitoring techniques to analyze failures of critical industrial systems. In *Software Reliability Engineering (ISSRE), 2014 IEEE 25th International Symposium on*, pages 212–222. IEEE, 2014.

[30] Pedro Costa, Joao Gabriel Silva, and Henrique Madeira. Practical and representative faultloads for large-scale software systems. *Journal of Systems and Software*, 103:182–197, 2015.

[31] 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.*, pages 162–172, May 2012.

[32] D. Cotroneo, D. Di Leo, F. Fucci, and R. Natella. SABRINE: State-based robustness testing of operating systems. In *Proc. of the 28th Int'l Conf. on Automated Software Engineering*, pages 125–135, November 2013.

[33] Domenico Cotroneo and Roberto Natella. Fault injection for software certification. *Security & Privacy, IEEE*, 11(4):38–45, 2013.

[34] Domenico Cotroneo, Roberto Pietrantuono, and Stefano Russo. Testing techniques selection based on odc fault types and software metrics. *Journal of Systems and Software*, 86(6):1613–1637, 2013.

[35] Benjamin Cox, David Evans, Adrian Filipi, Jonathan Rowanhill, Wei Hu, Jack Davidson, John Knight, Anh Nguyen-Tuong, and Jason Hiser. N-variant systems: A secretless framework for security through diversity. In *Proc. of the 15th USENIX Security Symp.*, pages 105–120, 2006.

[36] Michel Cukier, David Powell, and J Ariat. Coverage estimation methods for stratified fault-injection. *Computers, IEEE Transactions on*, 48(7):707–723, 1999.

[37] Murial Daran and Pascale Thévenod-Fosse. Software error analysis: a real

case study involving real faults and mutations. In *ACM SIGSOFT Software Engineering Notes*, volume 21, pages 158–171. ACM, 1996.

[38] Dorothy E Denning and Peter J Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, 1977.

[39] Joao Duraes and Henrique Madeira. Characterization of operating systems behavior in the presence of faulty drivers through software fault emulation. In *Proc. of the Pacific Rim Int'l Symp. on Dependable Computing*, page 201, 2002.

[40] Joao Duraes and Henrique Madeira. Emulation of software faults by educated mutations at machine-code level. In *Proc. of the 13th Int'l Symp. on Software Reliability Engineering*, page 329, 2002.

[41] Joao A. Duraes and Henrique S. Madeira. Emulation of software faults: A field data study and a practical approach. *IEEE Trans. Softw. Eng.*, 32(11): 849–867, 2006.

[42] P. Fonseca, Cheng Li, V. Singhal, and R. Rodrigues. A study of the internal and external effects of concurrency bugs. In *Proc. of the Int'l Conf. on Dependable Systems and Networks*, pages 221–230, 2010.

[43] P. Fonseca, C. Li, and R. Rodrigues. Finding complex concurrency bugs in large multi-threaded applications. In *Proc. of the Sixth ACM European Conf. on Computer Systems*, pages 215–228, 2011.

[44] C. Giuffrida, L. Cavallaro, and A.S. Tanenbaum. We crashed, now what? In *Proc. of the Sixth Workshop on Hot Topics in System Dependability*, pages 1–8, 2010. ACM ID: 1924912.

[45] C. Giuffrida, C. Iorgulescu, A. Kuijsten, and A.S. Tanenbaum. Back to the future: Fault-tolerant live update with time-traveling state transfer. In *Proc. of the 27th USENIX Systems Administration Conf.*, 2013.

[46] Cristiano Giuffrida and Andrew S Tanenbaum. Safe and automated state transfer for secure and reliable live update. In *Proceedings of the 4th International Workshop on Hot Topics in Software Upgrades*, pages 16–20. IEEE Press, 2012.

[47] Cristiano Giuffrida, Anton Kuijsten, and Andrew 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.

[48] Weining Gu, Z. Kalbarczyk, and R.K. Iyer. Error sensitivity of the Linux kernel executing on PowerPC G4 and Pentium 4 processors. In *Proc. of the Int'l Conf. on Dependable Systems and Networks*, pages 887–896.

[49] Weining Gu, Z. Kalbarczyk, Ravishankar, K. Iyer, and Zhenyu Yang. Char-

acterization of Linux kernel behavior under errors. In *Proc. of the Int'l Conf. on Dependable Systems and Networks*, pages 459–468, June 2003.

[50] Haryadi S. Gunawi, Thanh Do, Pallavi Joshi, Peter Alvaro, Joseph M. Hellerstein, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Koushik Sen, and Dhruba Borthakur. FATE and DESTINI: A framework for cloud recovery testing. In *Proc. of the 8th USENIX Conf. on Networked Systems Design and Implementation*, pages 18–18, 2011.

[51] U. Gunneflo, J. Karlsson, and J. Torin. Evaluation of error detection schemes using fault injection by heavy-ion radiation. In *Proc. of the 19th Int'l Symp. on Fault-Tolerant Computing*, pages 340–347, 1989.

[52] Zhenyu Guo, Xi Wang, Jian Tang, Xuezheng Liu, Zhilei Xu, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. R2: An application-level kernel for record and replay. In *Proc. of the Eighth USENIX Symp. on Operating Systems Design and Implementation*, pages 193–208, 2008.

[53] Saul Hansell. Glitch makes teller machines take twice what they give. *The New York Times*, 1994.

[54] J.N. Herder, H. Bos, B. Gras, P. Homburg, and AS. Tanenbaum. Fault isolation for device drivers. In *Proc. of the Int'l Conf. on Dependable Systems and Networks*, pages 33–42.

[55] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. Failure resilience for device drivers. In *Proc. of the Int'l Conf. on Dependable Systems and Networks*, pages 41–50, 2007.

[56] J.J. Hudak, B.-H. Suh, D.P. Siewiorek, and Z. Segall. Evaluation and comparison of fault-tolerant software techniques. *IEEE Trans. Rel.*, 42(2):190–204, 1993.

[57] Ivano Irrera, Marco Vieira, and Joao Duraes. Adaptive failure prediction for computer systems: A framework and a case study. In *High Assurance Systems Engineering (HASE), 2015 IEEE 16th International Symposium on*, pages 142–149. IEEE, 2015.

[58] Tahar Jarboui, Jean Arlat, Yves Crouzet, and Karama Kanoun. Experimental analysis of the errors induced into Linux by three fault injection techniques. In *Proc. of the Int'l Conf. on Dependable Systems and Networks*, pages 331–336, 2002.

[59] E. Jenn, J. Arlat, M. Rimen, J. Ohlsson, and J. Karlsson. Fault injection into VHDL models: the MEFISTO tool. In *Proc. of the 24th Int'l Symp. on Fault-Tolerant Computing*, pages 66–75, 1994.

[60] Yue Jia and Mark Harman. An analysis and survey of the development of

mutation testing. *Software Engineering, IEEE Transactions on*, 37(5):649–678, 2011.

[61] Andreas Johansson, Neeraj Suri, and Brendan Murphy. On the selection of error model(s) for OS robustness evaluation. In *Proc. of the 37th Int'l Conf. on Dependable Systems and Networks*, pages 502–511, 2007.

[62] Andréas Johansson, Neeraj Suri, and Brendan Murphy. On the impact of injection triggers for OS robustness evaluation. In *Proc. of the 18th Int'l Symp. on Software Reliability*, page 127, 2007.

[63] Pallavi Joshi, Haryadi S. Gunawi, and Koushik 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*, volume 46, pages 171–188, 2011.

[64] Asim Kadav, Matthew J. Renzelmann, and Michael M. Swift. Fine-grained fault tolerance using device checkpoints. In *Proc. of the 18th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 473–484, 2013.

[65] Ghani A. Kanawati, Nasser A. Kanawati, and Jacob A. Abraham. FERRARI: A flexible software-based fault and error injection system. *IEEE Trans. Comput.*, 44(2):248–260, 1995.

[66] Karama Kanoun, Yves Crouzet, Ali Kalakech, Ana-Elena Rugina, and Philippe Rumeau. Benchmarking the dependability of Windows and Linux using PostMark workloads. In *Proc. of the 16th Int'l Symp. on Software Reliability Engineering*, pages 11–20.

[67] Wei-Lun Kao and R.K. Iyer. DEFINE: A distributed fault injection and monitoring environment. In *Proc. of the IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems*, pages 252–259, June 1994.

[68] Wei-lun Kao, Ravishankar K. Iyer, and Dong Tang. FINE: a fault injection and monitoring environment for tracing the UNIX system behavior under faults. *IEEE Trans. Softw. Eng.*, 19(11):1105–1118, 1993.

[69] Johan Karlsson and Peter Folkesson. 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*, pages 267–287, 1995.

[70] Billy Kidwell, Jane Huffman Hayes, and Allen P Nikora. Toward extended change types for analyzing software faults. In *Quality Software (QSIC), 2014 14th International Conference on*, pages 202–211. IEEE, 2014.

[71] Naoya Kikuchi, Tetsuzo Yoshimura, Ryo Sakuma, and Kenji Kono. Do in-

jected faults cause real failures? a case study of linux. In *Software Reliability Engineering Workshops (ISSREW), 2014 IEEE International Symposium on*, pages 174–179. IEEE, 2014.

[72] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *Proc. of the 22nd ACM Symp. on Oper. Systems Prin.*, pages 207–220. ACM, 2009. ISBN 978-1-60558-752-3. doi: 10.1145/1629575.1629596.

[73] Philip Koopman and John DeVale. The exception handling effectiveness of POSIX operating systems. *IEEE Trans. Softw. Eng.*, 26(9):837–848. ISSN 0098-5589. doi: 10.1109/32.877845.

[74] Philip Koopman, John Sung, Christopher Dingman, Daniel Siewiorek, and Ted Marz. Comparing operating systems using robustness benchmarks. In *Proc. of the 16th Symp. on Reliable Distributed Systems*, page 72, 1997.

[75] Ilia Kravets and Dan Tsafrir. Feasibility of mutable replay for automated regression testing of security updates. In *Workshop on Runtime Environments, Systems, Layering and Virtualized Environments*, 2012.

[76] A. Kumar, A. Sutton, and B. Stroustrup. Rejuvenating C++ programs through demacrofication. In *Proc. of the 28th IEEE Int'l Conf. on Software Maintenance*, 2012.

[77] Oren Laadan, Nicolas Viennot, and Jason Nieh. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. In *Proc. of the Int'l Conf. on Measurement and Modeling of Computer Systems*, pages 155–166, June 2010.

[78] Anna Lanzaro, Roberto Natella, Stefan Winter, Domenico Cotroneo, and Neeraj Suri. An empirical study of injected versus actual interface errors. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 397–408. ACM, 2014.

[79] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proc. of the Int'l Symp. on Code Generation and Optimization*, page 75, 2004.

[80] C. Le Goues, ThanhVu Nguyen, S. Forrest, and W. Weimer. GenProg: A generic method for automatic software repair. *IEEE Trans. Softw. Eng.*, 38 (1):54–72, 2012.

[81] Andrew Lenharth, Vikram S. Adve, and Samuel T. King. Recovery domains: An organizing principle for recoverable operating systems. In *Proc. of the 14th Int'l Conf. on Architectural Support for Programming Languages and*

Operating Systems, pages 49–60, 2009.

[82] Guanpeng Li, Qining Lu, and Karthik Pattabiraman. Fine-grained characterization of faults causing long latency crashes in programs. 2015.

[83] Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang Zhai. Have things changed now?: an empirical study of bug characteristics in modern open source software. In *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, pages 25–33. ACM, 2006.

[84] Peter Lisherness and Kwang-Ting Cheng. Scemit: a systemc error and mutation injection tool. In *Design Automation Conference (DAC), 2010 47th ACM/IEEE*, pages 228–233. IEEE, 2010.

[85] Qining Lu, Mostafa Farahani, Jiesheng Wei, Anna Thomas, and Karthik Pattabiraman. Llfi: An intermediate code-level fault injection tool for hardware faults. In *Software Quality, Reliability and Security (QRS), 2015 IEEE International Conference on*, pages 11–16. IEEE, 2015.

[86] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *Proc. of the 13th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 329–339, March 2008.

[87] Henrique Madeira, Mario Rela, Francisco Moreira, and Joao Gabriel Silva. RIFLE: a general purpose pin-level fault injector. In *Proc. of the First European Dependable Computing Conf.*, pages 199–216, 1994.

[88] Henrique Madeira, Diamantino Costa, and Marco Vieira. On the emulation of software faults by software fault injection. In *Proc. of the Int'l Conf. on Dependable Systems and Networks*, pages 417–426, 2000.

[89] Paul D. Marinescu, Radu Banabic, and George Candea. An extensible technique for high-precision testing of recovery code. In *Proc. of the USENIX Annual Tech. Conf.*, page 23, 2010.

[90] P.D. 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*, pages 379–388, July 2009.

[91] R. Moraes, R. Barbosa, J. Duraes, N. Mendes, E. Martins, and H. Madeira. Injection of faults at component interfaces and inside the component code: are they equivalent? In *Proc. of the 6th European Dependable Computing Conf.*, pages 53–64, 2006.

[92] Darren Mutz, Fredrik Valeur, Giovanni Vigna, and Christopher Kruegel. Anomalous system call detection. *ACM Trans. Inf. Syst. Secur.*, 9(1):61–93,

February 2006.

[93] Darren Mutz, William Robertson, Giovanni Vigna, and Richard Kemmerer. Exploiting execution context for the detection of anomalous system calls. In *Proc. of the 10th Int'l Conf. on Recent Advances in Intrusion Detection*, pages 1–20, 2007.

[94] Vijay Nagarajan, Dennis Jeffrey, and Rajiv Gupta. Self-recovery in server programs. In *Proc. of the Int'l Symp. on Memory management*, pages 49–58, 2009.

[95] R. Natella, D. Cotroneo, J. Duraes, and H. Madeira. Representativeness analysis of injected software faults in complex software. In *Proc. of the 40th Int'l Conf. on Dependable Systems and Networks*, pages 437–446, July 2010.

[96] Roberto Natella, Domenico Cotroneo, Joao Duraes, Henrique S Madeira, et al. On fault representativeness of software fault injection. *Software Engineering, IEEE Transactions on*, 39(1):80–96, 2013.

[97] George C. Necula, Scott McPeak, and Westley Weimer. CCured: Type-safe retrofitting of legacy code. In *Proc. of the 29th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 128–139.

[98] Wee Teck Ng and Peter M Chen. The systematic improvement of fault tolerance in the rio file cache. In *Fault-Tolerant Computing, 1999. Digest of Papers. Twenty-Ninth Annual International Symposium on*, pages 76–83. IEEE, 1999.

[99] Wee Teck Ng and Peter M. Chen. The design and verification of the Rio file cache. *IEEE Trans. Comput.*, 50(4):322–337, April 2001.

[100] Thomas J. Ostrand and Elaine J. Weyuker. 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.

[101] Konstantinos Parasyris, Georgios Tziantzoulis, Christos D Antonopoulos, and Nikolaos Bellas. Gemfi: A fault injection tool for studying the behavior of applications on unreliable substrates. In *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*, pages 622–629. IEEE, 2014.

[102] Soyeon Park, Yuanyuan Zhou, Weiwei Xiong, Zuoning Yin, Rini Kaushik, Kyu H. Lee, and Shan Lu. PRES: Probabilistic replay with execution sketching on multiprocessors. In *Proc. of the 22nd ACM Symp. on Operating Systems Principles*, pages 177–192, 2009.

[103] Georgios Portokalidis and Angelos D. Keromytis. REASSURE: A self-contained mechanism for healing software using rescue points. In *Proc. of the*

Sixth Int'l Conf. on Advances in Information and Computer Security, pages 16–32, 2011.

[104] Swarup Kumar Sahoo, John Criswell, and Vikram Adve. An empirical study of reported bugs in server software with implications for automated bug diagnosis. In *Proc. of the 32nd Int'l Conf. on Software Engineering*, pages 485–494, 2010.

[105] Babak Salamat, Andreas Gal, Todd Jackson, Karthikeyan Manivannan, Gregor Wagner, and Michael Franz. Multi-variant program execution: Using multi-core systems to defuse buffer-overflow vulnerabilities. In *Proc. of the Int'l Conf. on Complex, Intelligent and Software Intensive Systems*, pages 843–848, 2008.

[106] U. Schiffel, A. Schmitt, M. Susskraut, and C. Fetzer. Slice your bug: Debugging error detection mechanisms using error injection slicing. In *Proc. of the European Dependable Computing Conf.*, pages 13–22, 2010.

[107] Stelios Sidiroglou, Oren Laadan, Carlos Perez, Nicolas Viennot, Jason Nieh, and Angelos D. Keromytis. ASSURE: Automatic software self-healing using rescue points. In *Proc. of the 14th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 37–48, 2009.

[108] Sudarshan M. Srinivasan, Srikanth Kandula, Christopher R. Andrews, and Yuanyuan Zhou. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *Proc. of the USENIX Annual Tech. Conf.*, page 3, 2004.

[109] Dinesh Subhraveti and Jason Nieh. Record and transplay: Partial checkpointing for replay debugging across heterogeneous systems. In *Proc. of the Int'l Conf. on Measurement and Modeling of Computer Systems*, pages 109–120, 2011.

[110] 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*, pages 475–484, 1992.

[111] Rickard Svenningsson, Jonny Vinter, Henrik Eriksson, and Martin Törngren. MODIFI: a MODel-implemented fault injection tool. In *Proc. of the 29th Int'l Conf. on Computer Safety, Reliability, and Security*, pages 210–222, 2010.

[112] Michael M. Swift, Muthukaruppan Annamalai, Brian N. Bershad, and Henry M. Levy. Recovering device drivers. *ACM Trans. Comput. Syst.*, 24 (4):333–360, November 2006.

[113] Anna Thomas. LLFI: An intermediate code level fault injector for soft computing applications. In *Proc. of the Pacific Rim Int'l Symp. on Dependable Computing*, 2012.

[114] Timothy K. Tsai and Ravishankar 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*, pages 26–40, 1995.

[115] Timothy K. Tsai, Mei-Chen Hsueh, Hong Zhao, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Stress-based and path-based fault injection. *IEEE Trans. Comput.*, 48(11):1183–1201, November 1999.

[116] Joseph Tucek, Weiwei Xiong, and Yuanyuan Zhou. Efficient online validation with delta execution. In *Proc. of the 14th Int'l Conf. on Architectural support for programming languages and operating systems*, pages 193–204, March 2009.

[117] Erik Van der Kouwe, Cristiano Giuffrida, and Andrew S. Tanenbaum. Evaluating distortion in fault injection experiments. In *Proc. of the 15th Int'l Symp. on High-Assurance Systems Engineering*, pages 25–32, Los Alamitos, CA, USA, January 2014. IEEE Computer Society. doi: http://doi.ieeecomputersociety.org/10.1109/HASE.2014.13.

[118] Erik Van der Kouwe, Cristiano Giuffrida, and Andrew S Tanenbaum. Finding fault with fault injection: an empirical exploration of distortion in fault injection experiments. *Software Quality Journal*, pages 1–30, 2014.

[119] Erik Van der Kouwe, Cristiano Giuffrida, and Andrew S Tanenbaum. On the soundness of silence: Investigating silent failures using fault injection experiments. In *Dependable Computing Conference (EDCC), 2014 Tenth European*, pages 118–129. IEEE, 2014.

[120] Erik Van der Kouwe, Cristiano Giuffrida, Razvan Ghitulete, and Andrew S Tanenbaum. A methodology to efficiently compare operating system stability. In *High Assurance Systems Engineering (HASE), 2015 IEEE 16th International Symposium on*, pages 93–100. IEEE, 2015.

[121] Nicolas Viennot, Siddharth Nair, and Jason Nieh. Transparent mutable replay for multicore debugging and patch validation. In *Proc. of the 18th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 127–138, March 2013.

[122] Stefan Winter, Michael Tretter, Benjamin Sattler, and Neeraj Suri. simfi: From single to simultaneous software fault injections. In *Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on*, pages 1–12. IEEE, 2013.

[123] Stefan Winter, Oliver Schwahn, Roberto Natella, Neeraj Suri, and Domenico Cotroneo. No pain, no gain? the utility of parallel fault injections. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages

494–505, 2015.

[124] F. Yates. Contingency table involving small numbers and the chi-square test. *Supplement to the Journal of the Royal Statistical Society*, 1(2):217—235.

[125] Takeshi Yoshimura, Hiroshi Yamada, and Kenji Kono. Is linux kernel oops useful or not? In *Proc. of the Eighth Workshop on Hot Topics in System Dependability*, page 2, 2012.

[126] Wei Zhang, Marc de Kruijf, Ang Li, Shan Lu, and Karthikeyan Sankaralingam. ConAir: Featherweight concurrency bug recovery via single-threaded idempotent execution. In *Proc. of the 18th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 113–126, 2013.

[127] Feng Zhou, Jeremy Condit, Zachary Anderson, Ilya Bagrak, Rob Ennals, Matthew Harren, George Necula, and Eric Brewer. SafeDrive: Safe and recoverable extensions using language-based techniques. In *Proc. of the 7th Symp. on Operating Systems Design and Implementation*, pages 45–60, 2006.

Summary

Despite decades of advances in computer science, computer systems are still plagued by crashes and security lapses due to software faults caused by programmer mistakes. As a consequence, there is a great need for fault-tolerance in software systems. To be able to compare such approaches and justify their use, we need better ways to be able to determine how much they contribute to fault-tolerance in practical settings. To measure the fault tolerance of a software system, one must expose it to actual software faults. Software fault injection, also known as software mutation testing, is a suitable approach to achieve this purpose. By introducing deliberate software faults that are similar to those that could have been introduced by a human programmer, it becomes possible to perform a sufficiently large number of such experiments to be able to reach statistically significant conclusions. However, compared to other types of fault injection, software fault injection faces some unique challenges that make it harder to perform such experiments in a way that is representative of real-world faults. Because human mistakes can not be predicted with purely theoretical models, they must be based on empirical analyses that investigate faults that humans have made in practice. A second difficulty is the fact that the system itself must be modified; it is not sufficient to alter the environment the system is running in. As a consequence, it is still hard to perform software fault injection in a methodologically sound way.

In this dissertation, we aim to advance the state of the art in software fault injection to allow this technique to be used for measuring fault tolerance in a way that is methodologically sound as well as efficient.

First, we raise the issue of fault load distortion. Our experiments demonstrate that fault activation correlates with factors that are important in defining the fault model, such as fault types and code locations with specific properties. We explain the implications of these results for workload selection and fault model specification.

By raising awareness of the issue of fault load distortion and providing guidelines to minimize its impact, we hope to increase the quality of the results from future software fault injection experiments.

Next, we present a widely applicable methodology to identify silent failures in fault injection experiments. Our approach consists of comparing externally visible behavior between a golden run using the original program and a faulty run where a fault has been injected in the program, while avoiding false positives due to non-determinism. We show that silent failures are common and consistent with field data, which suggests they they can be accurately emulated through fault injection. The main implication is that the fail-stop assumption does not hold, which means that fault-tolerance mechanisms depending on it need to be extended and re-evaluated using a method such as ours.

Then, we present an approach to compare the stability of operating systems. Rather then simply comparing the number of crashes when performing fault injection experiments, we make the point that one cannot expect the operating system to function correctly if faults are injected. Instead, we use a pre-test and a post-test to determine whether running a workload that triggers the fault causes the system to become unstable afterwards. This allows for a meaningful comparison of the effectiveness of fault-tolerance mechanisms in operating systems. Another contribution is the fact that our methodology provides a way to estimate how many fault injection experiments are needed to be able to draw statistical conclusions. Together, these elements provide a major step towards the systematic evaluation of fault-tolerance mechanisms in operating systems.

Finally, we present a methodology to run software fault injection experiments in such a way that source-level information is available to the fault injector while avoiding the slowdown caused by the need to recompile the source code for each experiment. The key is to create markers before code generation that can be recognized with very high accuracy in the resulting binary. Our evaluation compares this approach against alternatives and shows that it is the only one that can reach performance close to that of binary-level fault injection in all cases. This means high-quality fault injection experiments on large code bases can be run at lower cost.

Taken together, the contributions made in this dissertation allow software fault injection to be applied to evaluate fault-tolerance mechanisms in a way that is sound, low-cost even for large code bases such as operating systems and that provides state-of-the-art fault representativeness.

Samenvatting

Ondanks decennia van voortuitgang in de informatica worden computersystemen nog altijd geplaagd door crashes en beveiligingslekken door softwarefouten veroorzaakt door fouten van programmeurs. Hierdoor is er een grote behoefte aan fouttolerantie in softwaresystemen. Om dergelijke methodes te kunnen vergelijken en hun kosten te kunnen rechtvaardigen hebben we betere manieren nodig om te kunnen bepalen hoeveel ze in de praktijk bijdragen aan verbeterde fouttolerantie. Om de fouttolerantie van een softwaresysteem te meten moet het worden blootgesteld aan echte softwarefouten. Software foutinjectie, ook bekend als *software mutation testing*, is een geschikte aanpak om dit doel te bereiken. Door doelbewust fouten te introduceren die lijken op de fouten die een menselijke programmeur zou maken is het mogelijk om genoeg experimenten uit te voeren om statistisch relevante conclusies te trekken. Echter, in vergelijking met andere vormen van foutinjectie heeft software foutinjectie te maken met een aantal unieke uitdagingen die het moeilijker maken om experimenten zodanig uit te voeren dat ze representatief zijn voor fouten die in de praktijk gemaakt worden. Omdat menselijke fouten niet voorspeld kunnen worden door puur theoretische modellen moeten ze gebaseerd worden op empirische analyses waarin onderzocht wordt welke fouten mensen in de praktijk maken. Verder is het lastig dat het systeem zelf gewijzigd moet worden; het is niet voldoende om enkel de omgeving waarin het systeem wordt uitgevoerd aan te passen. Als gevolg van deze moeilijkheden is het nog altijd lastig om software foutinjectie uit te voeren op een manier die methodologisch deugdelijk is.

Met deze dissertatie streven we ernaar de kennis over software foutinjectie te vergroten zodat deze techniek gebruikt kan worden voor het meten van fouttolerantie op een manier die zowel methodologisch deugdelijk als efficiënt is.

Als eerste brengen we *fault load distortion* ter sprake. Onze experimenten tonen aan dat de activatie van fouten correleert met factoren die belangrijk zijn voor het

definiëren van het foutmodel, zoals fouttypes en locaties in de code met specifieke eigenschappen. We leggen de implicaties van deze resultaten uit met betrekking tot de keuze van een werklast en de specificatie van het foutmodel. Door de aandacht te vestigen op *fault load distortion* en richtlijnen te geven om het effect daarvan de minimaliseren hopen we de kwaliteit van de resultaten uit toekomstige foutinjectie experimenten te verbeteren.

Vervolgens presenteren we een breed toepasbare methodologie om *silent failures* in foutinjectie experimenten te kunnen identificeren. Onze aanpak bestaat uit het vergelijken van het extern zichtbare gedrag tussen een *golden run* die gebruik maakt van het oorspronkelijke programma en een *faulty run* waar een fout in het programma is geïnjecteerd, waarbij we valse positieven door nondeterminisme vermijden. We tonen aan dat *silent failures* regelmatig voorkomt en consistent is met metingen in het wild, hetgeen het idee dat het nauwkeurig door foutinjectie geëmuleerd kan worden ondersteunt. De belangrijkste implicatie is dat de *fail-stop* aanname onjuist is, hetgeen betekent dat mechanismes voor fouttolerantie die ervan afhankelijk zijn uitgebreid moeten worden en opnieuw geëvalueerd moeten worden met een methode zoals de onze.

Daarna presenteren we een aanpak om de stabiliteit van besturingssystemen te vergelijken. In plaats van het simpelweg vergelijken van het aantal crashes tijdens het uitvoeren van foutinjectie experimenten beargumenteren we dat men niet kan verwachten dat het besturingssysteem nog correct werkt als er fouten in geïnjecteerd zijn. In plaats daarvan gebruiken we een voortest en een natest om te bepalen of het uitvoeren van een werklast die de fout activeert ervoor zorgt dat het systeem achteraf instabiel wordt. Dit maakt een betekenisvolle vergelijking mogelijk van de effectiviteit van mechanismes voor fouttolerantie in besturingssystemen. Een andere bijdrage is het feit dat onze methodologie een mogelijkheid biedt om in te schatten hoeveel foutinjectie experimenten nodig zijn om statistisch relevante conclusies te kunnen trekken. Samen genomen leveren deze elementen een grote stap in het systematisch kunnen evalueren van fouttolerantie mechanismes in besturingssystemen.

Tenslotte presenteren we een methodologie om software foutinjectie experimenten zodanig uit te voeren dat informatie uit de broncode beschikbaar is voor de foutinjector terwijl we de vertraging die ontstaat door de noodzaak om de broncode te hercompileren voor elk experiment kunnen vermijden. De oplossing is om markeringen aan te maken voor de codegeneratie die met zeer hoge nauwkeurigheid herkend kunnen worden in het resulterende binaire bestand. Onze evaluatie vergelijkt deze aanpak met alternatieven en toont aan dat het de enige aanpak is die in alle gevallen bijna net zo goed presteert als foutinjectie op binair niveau. Dit betekent dat foutinjectie experimenten van hoge kwaliteit kunnen worden uitgevoerd op grote programma's tegen lagere kosten.

Bij elkaar maken de bijdragen in deze dissertatie het mogelijk om software foutinjectie toe te passen om mechanismes voor fouttolerantie te evalueren op een manier die deugdelijk is, goedkoop zelfs voor grote programma's zoals besturingssystemen en die foutrepresentativiteit biedt die state-of-the-art is.