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

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

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

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

***Abstract*—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.

***Keywords*-silent failure; fail-stop; fault injection; LLVM; system call tracing**

I. 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 [1]. 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 programming [2], 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 [3]–[6] 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 [7]–[15], 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 [16]–[18], 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 [19] 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 [20].

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 [21], [22]. 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 [22]. 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 [23], [24], 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 [19], 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.

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

TABLE I
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

A. Fault injection

We use the EDFI [25] framework to perform fault injection. This system is based on the ability of the LLVM compiler framework [26] 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 [27]–[29]. The fault types injected by our program have been selected based on these papers and are listed in table I.

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 [30], [31]. 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.

B. 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 [32].

C. 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 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 [33] is not sufficient here. Figure 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.

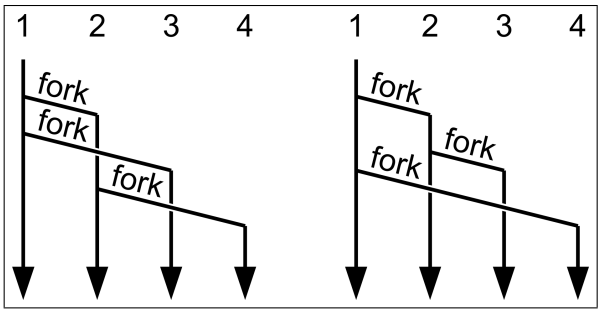

Fig. 1. Scheduling of fork resulting in different pids

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 [33], 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 [34].

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 [35]–[40] 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.

D. 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 IV, which discusses the analysis of our measurements.

E. 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.

III. 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 [41] (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 nonsilent 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.

IV. 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 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

TABLE II
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%

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 II indicates the levels of coverage we achieved. With regard to the workloads, rtest is the official regression test, doc means constructed based on documentation and ab means ApacheBench [41] 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 [42]. 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.

A. 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 95th 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 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 III 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

TABLE III
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%

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 III 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 [30], [31]. 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 [11], [43]) 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 [19]. 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 [4], [5]) 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.

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.

B. 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 IV). It is reasonable to expect that different types of faults result in different program behavior. Table IV 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

TABLE IV
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%

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.

C. 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 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 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 V 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

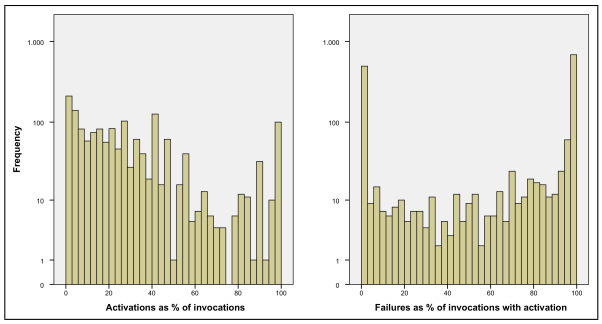

Fig. 2. Histograms of fault activation and failure ratios; frequency refers to the total number of runs for all programs/benchmarks in that bracket

TABLE V
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%
	≥ 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%
	≥ 80%	97.7%	36.1%	42.2%	25.6%

that would otherwise remain hidden. However, it is known that especially those code locations that are rarely used are more likely to contain bugs [21], [22]. 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 V 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 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.

V. 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 IV), 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 [35]–[40] to address this issue.

VI. 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 [7] and local [8]–[10] user programs to operating systems [11], [12], file caches [15], and device drivers [13], [14].

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 [22], [44], [45], at the binary level [15], [27], [46], or, more recently, at the intermediate code level [25], [47], [48]. An alternative is to introduce program mutations at runtime, using software and hardware traps [7], [46], [49], [50] or library interposition mechanisms [9], [10].

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* [27], [51] *where to inject* [21], [22], and *how to inject* [52]. 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 [30], [31]. 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 [8], [10], 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 [16], [17] or high-level properties directly exposed to the user view of the system [18]. More recent studies investigate how faults progressively propagate throughout the system [43], [53]. 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 [53], 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 [11], [54], fault propagation [11], or bug reproducibility [55], [56]. A notable exception is represented by the work of Fonseca et al. [19], 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 [15], [57]. 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 [23], [24]. 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 [58], [59] compare semantically equivalent execution runs with different memory layout (to detect security attacks) and multi-version execution [60], [61], 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 [34], [62], 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 [34].

VII. 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 [63]. 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 programs 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 [35]–[40]. 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.

Acknowledgment

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

References

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

[2] A. Avizienis, "The n-version approach to fault-tolerant software," *IEEE Trans. Softw. Eng.*, vol. 11, no. 12, pp. 1491–1501, Dec. 1985.

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

[4] S. Sidiroglou, O. Laadan, C. Perez, N. Viennot, J. Nieh, and A. 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*, 2009, pp. 37–48.

[5] G. Portokalidis and A. 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*, 2011, pp. 16–32.

[6] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "GenProg: A generic method for automatic software repair," *IEEE Trans. Softw. Eng.*, vol. 38, no. 1, pp. 54–72, 2012.

[7] W.-L. Kao and R. Iyer, "DEFINE: A distributed fault injection and monitoring environment," in *Proc. of the IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems*, Jun. 1994, pp. 252–259.

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

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

[10] P. D. Marinescu, R. Banabic, and G. Candea, "An extensible technique for high-precision testing of recovery code," in *Proc. of the USENIX Annual Tech. Conf.*, 2010, p. 23.

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

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

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

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

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

[16] A. Albinet, J. Arlat, and J.-C. 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*, 2004, p. 867.

[17] T. Jarboui, J. Arlat, Y. Crouzet, and K. 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*, 2002, pp. 331–336.

[18] J. Duraes and H. 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*, 2002, p. 201.

[19] P. Fonseca, C. 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*, 2010, pp. 221–230.

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

[21] 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*, Jul. 2010, pp. 437–446.

[22] ——, "On fault representativeness of software fault injection," *IEEE Trans. Softw. Eng.*, vol. PP, no. 99, p. 1, 2012.

[23] D. Mutz, F. Valeur, G. Vigna, and C. Kruegel, "Anomalous system call detection," *ACM Trans. Inf. Syst. Secur.*, vol. 9, no. 1, pp. 61–93, Feb. 2006.

[24] D. Mutz, W. Robertson, G. Vigna, and R. 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*, 2007, pp. 1–20.

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

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

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

[28] J. Christmansson and R. Chillarege, "Generation of an error set that emulates software faults based on field data," in *Proc. of the 26th Int'l Symp. on Fault-Tolerant Computing*, 1996, p. 304.

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

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

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

[32] "External visibility of system calls." [Online]. Available: http://www.cs.vu.nl/%7Evdkouwe/edcc2014/syscalls.html

[33] E. W. Biederman, "Multiple instances of the global Linux namespaces," in *Proc. of the Linux Symposium*, 2006.

[34] N. Viennot, S. Nair, and J. 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*, Mar. 2013, pp. 127–138.

[35] S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. Zhou, "Flashback: A lightweight extension for rollback and deterministic replay for software debugging," in *Proc. of the USENIX Annual Tech. Conf.*, 2004, p. 3.

[36] G. Altekar and I. Stoica, "ODR: Output-deterministic replay for multi-core debugging," in *Proc. of the 22nd ACM Symp. on Operating Systems Principles*, 2009, pp. 193–206.

[37] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu, "PRES: Probabilistic replay with execution sketching on multiprocessors," in *Proc. of the 22nd ACM Symp. on Operating Systems Principles*, 2009, pp. 177–192.

[38] Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, M. Wu, M. F. Kaashoek, and Z. Zhang, "R2: An application-level kernel for record and replay," in *Proc. of the Eighth USENIX Symp. on Operating Systems Design and Implementation*, 2008, pp. 193–208.

[39] D. Subhraveti and J. 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*, 2011, pp. 109–120.

[40] O. Laadan, N. Viennot, and J. 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*, Jun. 2010, pp. 155–166.

[41] "Apache benchmark (AB)," http://httpd.apache.org/docs/2.0/programs/ab.html.

[42] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proc. of the Eighth USENIX Symp. on Operating Systems Design and Implementation*, 2008, pp. 209–224.

[43] T. Yoshimura, H. Yamada, and K. Kono, "Is linux kernel oops useful or not?" in *Proc. of the Eighth Workshop on Hot Topics in System Dependability*, 2012, p. 2.

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

[45] F. Zhou, J. Condit, Z. Anderson, I. Bagrak, R. Ennals, M. Harren, G. Necula, and E. Brewer, "SafeDrive: Safe and recoverable extensions using language-based techniques," in *Proc. of the Seventh Symp. on Operating Systems Design and Implementation*, 2006, pp. 45–60.

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

[47] 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.*, 2010, pp. 13–22.

[48] A. 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.

[49] T. K. Tsai and R. K. Iyer, "Measuring fault tolerance with the FTAPE fault injection tool," in *Proc. of the Eighth Int'l Conf. on Modelling Techniques and Tools for Computer Performance Evaluation*, 1995, pp. 26–40.

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

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

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

[53] V. Nagarajan, D. Jeffrey, and R. Gupta, "Self-recovery in server programs," in *Proc. of the Int'l Symp. on Memory management*, 2009, pp. 49–58.

[54] S. Lu, S. Park, E. Seo, and Y. 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*, Mar. 2008, pp. 329–339.

[55] S. K. Sahoo, J. Criswell, and V. 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*, 2010, pp. 485–494.

[56] S. Chandra and P. 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*, 2000, pp. 97–106.

[57] 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.

[58] B. Salamat, A. Gal, T. Jackson, K. Manivannan, G. Wagner, and M. 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*, 2008, pp. 843–848.

[59] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser, "N-variant systems: A secretless framework for security through diversity," in *Proc. of the 15th USENIX Security Symp.*, 2006, pp. 105–120.

[60] J. Tucek, W. Xiong, and Y. Zhou, "Efficient online validation with delta execution," in *Proc. of the 14th Int'l Conf. on Architectural support for programming languages and operating systems*, Mar. 2009, pp. 193–204.

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

[62] I. Kravets and D. Tsafrir, "Feasibility of mutable replay for automated regression testing of security updates," in *Workshop on Runtime Environments, Systems, Layering and Virtualized Environments*, 2012.

[63] 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*, 2011, pp. 215–228.