A Methodology to Efficiently Compare Operating System Stability

Erik van der Kouwe*, Cristiano Giuffrida†, Razvan Ghitulete‡ and Andrew S. Tanenbaum§
Computer Systems Section
Faculty of Sciences, VU University
Amsterdam, The Netherlands
*Email: erik@minix3.org
†Email: giuffrida@cs.vu.nl
‡Email: razvan.ghitulete@gmail.com
§Email: ast@cs.vu.nl

***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 runtime 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.

***Keywords*-Fault injection; Stability; Operating Systems;**

I. 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 [1], 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 [2]. 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 compare two OSes (Linux and MINIX 3) to demonstrate the usefulness of our approach.

II. 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 [3], 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 [4], implementations [5], or evaluating the impact of faulty drivers on the robustness of the operating system [6]–[8]. A number of robustness testing studies have also sought to evaluate the experimental impact of different fault models [7], injection techniques [9], and injection triggers [8], [10]. 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 [4]—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 [11], [12], evaluating the OS behavior in presence of errors [9], [11], [13], [14], or comparing the dependability of different operating systems [5], [15]. The testing methodologies proposed in prior work encompass different code mutation techniques, ranging from run-time injection [12], [13], [15] to binary rewriting [5], [14] and compile-time injection [16]. 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 [12]. Prior studies, however, have found run-time injection strategies to be poorly representative of realistic software faults [17]. Similar representativeness problems have been also evidenced for binary rewriting strategies [18], 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 [11], 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 [16], 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 [2].

III. 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.

A. 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 [19]–[21]. These works form the basis for our selection of fault types, which is listed in table I. 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 can be injected at that location [22]. 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 [18], [23]. The intermediate code level is decoupled from

TABLE I: 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

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 [24].

B. 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 [7] 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 [22], 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 [1]) 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 [23],

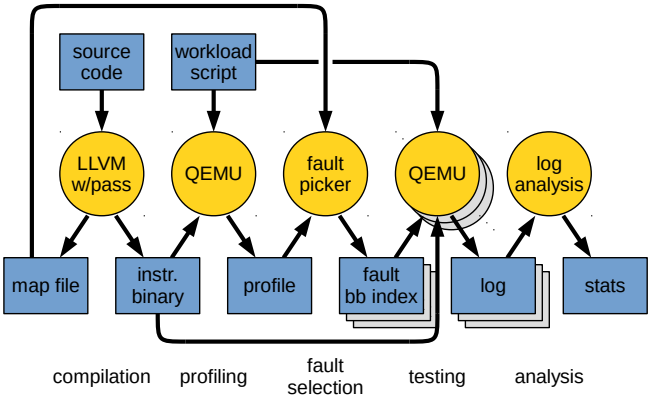

Fig. 1: Phases of our approach

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* [23] 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 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.

C. 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 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 [2]. 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 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.

D. 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 [12]. 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.

TABLE II: 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

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 II. 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.

E. 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 [25], 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.

IV. 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 IV). 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 [26]. 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 [25]) 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.

A. Coverage

Table III shows the coverage reached by our workloads. in terms of both fault candidates ("fc") and lines of code ("loc").

TABLE III: 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

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 [22], 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 III, 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.

B. 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 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.

C. Scalability

As discussed in the approach section, we have opted to accept a slowdown to select the active fault at runtime to

TABLE IV: 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

TABLE V: 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 #=$p < 0.05$, ##=$p < 0.01$, ###=$p < 0.001$

save compilation time and overall hope to accelerate the experiments. Table IV 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.

D. 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 V 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. Given that this workload also reaches the highest coverage (see Table III) 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?

E. 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

TABLE VI: 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 *=$p < 0.05$, **=$p < 0.01$, ***=$p < 0.001$

in Table VI. 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 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.

TABLE VII: 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 ***=$p < 0.001$

F. 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 VII 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 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.

V. 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 [17]. It should also be considered that we used only LLVM and other compilers may generate code that reacts to faults differently.

VI. 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.

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," vol. 27, no. 4, pp. 55–64.

[2] R. Natella, D. Cotroneo, J. Duraes, and H. Madeira, "On fault representativeness of software fault injection," vol. PP, no. 99, p. 1.

[3] P. Koopman and J. DeVale, "The exception handling effectiveness of POSIX operating systems," vol. 26, no. 9, pp. 837–848.

[4] K. Kanoun, Y. Crouzet, A. Kalakech, A.-E. Rugina, and P. Rumeau, "Benchmarking the dependability of Windows and Linux using PostMark workloads," in *Proc. of the 16th Int'l Symp. on Software Reliability Engineering*, pp. 11–20.

[5] J. Arlat, J.-C. Fabre, and M. Rodriguez, "Dependability of COTS microkernel-based systems," vol. 51, no. 2, pp. 138–163.

[6] 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*, p. 867.

[7] A. Johansson, N. Suri, and B. 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*, pp. 502–511.

[8] ——, "On the impact of injection triggers for OS robustness evaluation," in *Proc. of the 18th Int'l Symp. on Software Reliability*, p. 127.

[9] 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*, pp. 331–336.

[10] 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*, pp. 125–135.

[11] J. Dures 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*, p. 201.

[12] J. Herder, H. Bos, B. Gras, P. Homburg, and A. Tanenbaum, "Fault isolation for device drivers," in *Proc. of the Int'l Conf. on Dependable Systems and Networks*, pp. 33–42.

[13] 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*, pp. 459–468.

[14] W. Gu, Z. Kalbarczyk, and R. 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*, pp. 887–896.

[15] D. Chen, G. Jacques-Silva, Z. Kalbarczyk, R. Iyer, and B. Mealey, "Error behavior 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*, pp. 339–346.

[16] 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*, pp. 45–60.

[17] 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*, pp. 417–426.

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

[19] J. A. Duraes and H. S. Madeira, "Emulation of software faults: A field data study and a practical approach," vol. 32, no. 11, pp. 849–867.

[20] 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*, p. 304.

[21] 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*, pp. 475–484.

[22] E. Van Der Kouwe, C. Giuffrida, and A. Tanenbaum, "Evaluating distortion in fault injection experiments," in *Proc. of the 15th Int'l Symp. on High-Assurance Systems Engineering*, pp. 25–32.

[23] 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*.

[24] 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*, p. 75.

[25] LLVM Linux. http://llvm.linuxfoundation.org/index.php/Main_Page.

[26] F. Yates, "Contingency table involving small numbers and the 2 test," vol. 1, no. 2, pp. 217–235.