Fast Kernel Error Propagation Analysis in Virtualized Environments

Nicolas Coppik
TU Darmstadt
Darmstadt, Germany
nc@cs.tu-darmstadt.de

Oliver Schwahn
TU Darmstadt
Darmstadt, Germany
os@cs.tu-darmstadt.de

Neeraj Suri
Lancaster University
Lancaster, UK
neeraj.suri@lancaster.ac.uk

***Abstract*—Assessing operating system dependability remains a challenging problem, particularly in monolithic systems. Component interfaces are not well-defined and boundaries are not enforced at runtime. This allows faults in individual components to arbitrarily affect other parts of the system. Software fault injection (SFI) can be used to experimentally assess the resilience of such systems in the presence of faulty components. However, applying SFI to complex, monolithic operating systems poses challenges due to long test latencies and the difficulty of detecting corruptions in the internal state of the operating system.**

In this paper, we present a novel approach that leverages static and dynamic analysis alongside modern operating system and virtual machine features to reduce SFI test latencies for operating system kernel components while enabling efficient and accurate detection of internal state corruptions.

We demonstrate the feasibility of our approach by applying it to multiple widely used Linux file systems.

I. Introduction

Monolithic operating system (OS) kernels are large, complex, and widely used software systems. Although, unlike microkernel architectures, they do not enforce isolation between components at runtime, they are nonetheless commonly developed as collections of separate subsystems and modules, which may differ in size and complexity and implement different types of functionality. Moreover, modules, such as device drivers or file systems, may be developed and maintained by separate teams of developers, resulting in substantial differences in code quality and, consequently, the number and density of faults in different modules. To assess the dependability of an OS kernel in the presence of faulty modules, we need to analyze how such faulty modules can affect other parts of the kernel, such as other modules or the core system. A well-established technique for this purpose is software fault injection (SFI), in which software faults are deliberately injected into a target module. In this way, many faulty module versions can be created. By executing these faulty versions, the system is exposed to their faulty behavior to test how resilient it is to the type of faults that were injected.

The common workflow for SFI tests is: (1) faulty versions are generated, (2) each faulty version is loaded into the kernel, (3) the system is subjected to a test workload, and (4) the behavior of the system is monitored. With conventional monitoring, the possible test outcomes are limited to externally observable effects: if the system fails (e.g. kernel panic), reports an error, or its behavior otherwise deviates from a fault-free execution (e.g. different output), the faulty module has affected the overall system behavior; otherwise, the faulty version is assumed to have no effect. The latter case poses a particular challenge as there are different possible causes for the lack of an externally observable effect: (1) the injected fault was not activated by the test workload, (2) the fault is benign, i.e., it was activated but does not affect the system behavior or state under the test workload, or (3) the fault was activated and affects or corrupts the system state, but the SFI test ends prior to any effects becoming externally observable. The third case is problematic since executing the system for longer may have led to an observable deviation that was missed because of a short test run. We follow the Laprie taxonomy [3] and refer to instances in which faults in a module or component affect other parts of the system as *error propagation*, which is what occurs in case (3). Identifying instances of potential error propagation requires error propagation analysis (EPA) and is particularly important in the context of OS kernels as these are long-running systems, and the limited test durations typically used in SFI testing do not (and, due to the impact on test latency, cannot) reflect that fact.

Prior work has attempted to tackle this issue by introducing additional instrumentation to the faulty module for tracing the modifications made by the faulty module either to the system state or to externally visible parts of its own state [10]. These modifications can then be compared to those made by the non-faulty version of the same module, with the assumption that instances in which the behavior of the faulty version diverges from the non-faulty version constitute potential error propagation. This approach yields promising results but it also introduces further overhead, increasing the already substantial SFI test latency, especially for complex modules or long workloads. False positives are also an issue with such a detector since the system may exhibit non-deterministic behavior.

In this work, we investigate how to mitigate the impact of aforementioned instrumentation on SFI test latency without affecting the validity of SFI test results. Long test latencies are a known problem with SFI tests, for which various mitigation approaches have been proposed. Our proposed approach enhances the state of the art to be applicable to kernel code.

This paper is structured as follows: We cover related work in Sec. II. We then present our proposed approach and prototype implementation in Sec. III. Our experimental evaluation is

discussed in Sec. IV. We discuss insights and limitations in Sec. V and provide concluding remarks in Sec. VI.

II. Related Work

In this work we aim to reduce SFI test latencies for kernel code to improve the practical applicability of EPA, which is often hindered by long test latencies. Related work includes approaches for reducing SFI test latencies (Sec. II-A), more general work on test parallelization that tackles several related issues (Sec. II-B), and work on EPA (Sec. II-C).

A. SFI Test Latencies

Test latencies are a well known problem in fault injection, and numerous approaches to reduce them have been proposed. Most of these focus on parallelizing the execution of fault injection experiments using different isolation mechanisms to prevent interference between concurrent executions.

D-Cloud [5, 14] is a cloud system that enables fault injection testing of distributed systems using VMs to isolate the systems under test. We also use VMs to isolate different SFI experiments but our work targets kernel code rather than distributed systems.

Other techniques rely on more lightweight process isolation to avoid the overhead of strong VM isolation. This includes AFEX [4] as well as FastFI [25]. FastFI accelerates SFI experiments through parallelization and by avoiding redundant executions of code that is common to multiple faults and of faulty versions in which the fault location is never reached under a given workload. Unlike FastFI, we make use of the stronger isolation guarantees provided by VMs, which allows us to apply our technique to more targets, including kernel code, and lets us avoid the error-prone handling of potential resource conflicts on file descriptors that FastFI requires.

While VMs provide strong isolation, executing multiple fault injection experiments in parallel can still affect results due to the impact of a higher system load on test latencies, as shown in [28]. We take care to choose appropriate timeout values for our workload to avoid such effects and compare results across different degrees of parallelism on the same hardware.

B. Test Acceleration

Outside of the context of SFI testing, numerous ways of speeding up software testing in general have been investigated.

Many such approaches use parallel test execution and rely on the assumption that test cases are independent and can be executed in any order or concurrently without altering results (e.g., [11, 21, 23]). Recent work has shown that this assumption may not hold in practice [19, 26]. A related problem arises in SFI tests, where executing tests in parallel may also lead to interferences. Recent work investigating the static detection of conflicts between software tests [26] is not applicable to SFI testing since it relies on each test having a distinct entry point. In a typical SFI setup, all tests use the same workload and hence the same entry point.

Besides parallelization, tests can also be sped up by avoiding redundantly executing code multiple times and not executing unnecessary code. VmVm [6] avoids unnecessarily resetting the entire system state between test executions by determining which parts of the system state each test affects, and O!Snap [13] uses disk snapshots to reduce test setup and execution times as well as cost in a cloud setting. Both FastFI and our work avoid repeatedly re-executing the entire workload by cloning system state at appropriate points, but do so using fundamentally different methods: While FastFI relies on forking a process, we clone VM instances as described in Sec. III-C. This VM cloning resembles the VM fork primitive described in SnowFlock [18], however, our implementation does not require modifications to the VMM and is intended for cloning VM instances on a single host machine rather than in the distributed or cloud scenarios targeted by SnowFlock.

C. Error Propagation Analysis

Our work makes use of TrEKer [10], a technique for tracing error propagation in OS kernels using execution traces at the granularity of individual memory accesses.

Execution traces are commonly used to assess the outcome of fault injection tests [2, 27]. The execution of an unmodified system is traced one or more times. These executions, called *golden runs*, are used as an oracle that executions in which faults have been injected can be compared against.

Execution traces can be collected in several ways, including using debuggers [1, 9] or full-system simulation [24]. These techniques allow for fine-grained tracing and control over the SUT, but they also induce substantial execution overhead, especially full system simulation. If this overhead is not acceptable, dynamic binary instrumentation (DBI) can be used to collect traces (e.g., [20, 29]). However, DBI is not straightforward to apply to kernel code, and while several tools and frameworks exist [8, 12, 16, 17, 15], they have not been used to collect execution traces for fault injection tests. Like TrEKer, we use compile-time instrumentation.

III. Approach

We propose an approach to quickly identify how faults in a component of a monolithic operating system affect other parts of that system. We provide a brief overview of the systems we are focusing on and how their components interact in Sec. III-A. Then, we explain how EPA can be conducted in such systems in Sec. III-B, and how we address the limitations of prior work. This leads to an overview of our proposed approach in Sec. III-C. We describe our implementation in Sec. III-D.

A. System Model

In general, we follow the Laprie taxonomy [3]. We assume a component-based system in which each component implements a function according to some specification, realized through the externally observable part of its state, i.e., its *external state*. If that external state deviates from its specification, a component *failure* has occurred, which may have been caused by a prior deviation (*error*) in the component's internal state. The cause of an error is a *fault*, and the process by which a fault causes an error is called *fault activation*. Alterations in subsequent states caused by an error are instances of *error propagation*.

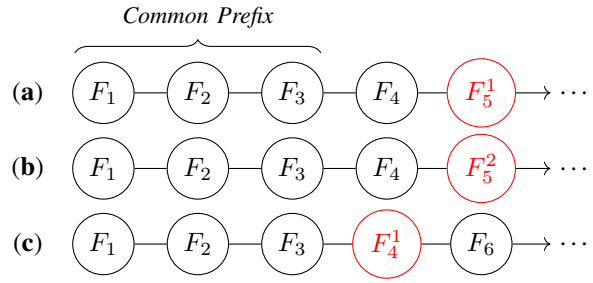


Fig. 1: The *common prefix problem*. Each F_i is a function and F_i^j is the j[th] faulty version of that function after fault injection.

These abstract notions map onto the monolithic OS kernels we are studying as follows:

- The system is the operating system kernel, and system components are individual kernel modules. Kernel modules are conceptually separate entities implementing distinct functionality, but no runtime isolation is enforced. We do not assume the existence of an explicit specification for the function they are intended to implement.
- A component's external state or interface consists of data passed between the component and the rest of the system through function call arguments, return values (in either direction), and shared memory communication.
- We focus on permanent faults in the form of software bugs and do not consider transient or hardware faults.

B. Kernel Error Propagation Analysis

We base our approach on the analysis proposed in TrEKer [10]. This approach suffers from some key limitations that we address in this work, most notably the instrumentation overhead, which further exacerbates the problem of long SFI test latencies. Moreover, TrEKer requires a large number of golden runs to obtain stable results, which results in long execution and analysis times to obtain and process these golden runs. With enough golden runs, TrEKer achieves a false positive rate below 1 %, but this can still amount to hundreds of misclassified executions given the large number of faulty versions that SFI in complex kernel modules may yield.

We propose an approach to reduce both overall SFI test latency and false positive rates with fewer golden runs. We propose integrating all faulty versions of a module into a single binary and using fast VM cloning to avoid redundant executions of common execution prefixes for different faulty versions, thereby also reducing non-determinism between executions.

C. Improving Kernel EPA with Fast VM Cloning

As our approach is conceptually related to that of FastFI [25], we first briefly summarize the FastFI execution model in Sec. III-C1 and discuss why it is not directly applicable to OS kernels. We then describe our own approach and its integration with error propagation analysis in Sec. III-C2.

1) FastFI Summarized: FastFI is a technique for reducing SFI test latencies by (1) avoiding redundant, repeated executions of code paths that are shared by different SFI tests

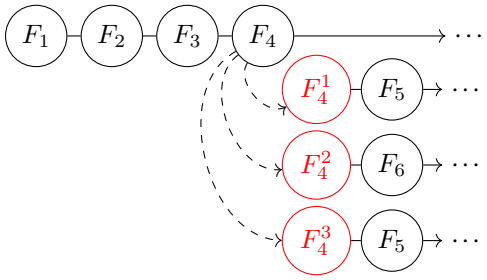


Fig. 2: The FastFI approach. The redundant re-execution of functions F_i in the common prefix is avoided. Each faulty version is executed in a new process once the faulty function F_i^j is reached.

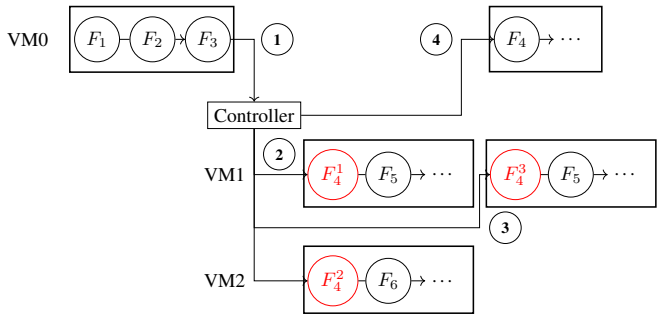


Fig. 3: Our enhanced VM-based approach. Each faulty version is executed in a VM that resumes execution from a snapshot.

(and therefore by different faulty versions), (2) avoiding the execution of faulty versions in which the given workload cannot activate the fault, and (3) facilitating parallelization of SFI testing. FastFI is based on the insight that, in SFI testing, the same workload is executed for each version, and prior to the execution reaching the part of the code that a fault has been injected in, the same code is executed for each version. We refer to this as the *common prefix* problem and show a simplified example in Figure 1. In this example, three SFI test executions (a)-(c) are depicted by illustrating the executed functions F_i. Faulty versions of a function are illustrated with F_i^j. The code in functions F_1 through F_3 is executed during each SFI test even though it contains no injected faults. The repeated execution is therefore redundant. The solution proposed in FastFI is shown in Figure 2. Here, F_1 through F_3 are only executed once, and new processes are forked for each faulty version when a function for which such versions exist is reached. While this approach can achieve substantial speedups as shown in [25], it is not applicable to kernel code for several reasons.

First, FastFI uses forking to duplicate the state of the system under test at appropriate points during the execution. As we are studying kernel code, we cannot rely on OS-provided abstractions, such as processes. Due to its reliance on the process abstraction, FastFI cannot handle multi-threaded code, SUTs consisting of multiple processes, and SUTs that rely on external resources besides the file system. While these are not insurmountable challenges for the systems FastFI has been

applied to, they render it inherently unsuitable for kernel code.

Secondly, FastFI integrates the entire control logic (which performs the forking, monitoring, and logging) in the system under test, which is not a desirable solution for kernel code as it could impose substantial delays in timing-critical portions of the system and impose additional technical challenges due to the lack of the availability of a process abstraction.

Next, we describe how we address these limitations, and how we integrate EPA in our enhanced approach.

2) Fast VM Cloning: To overcome the limitations that render FastFI unsuitable for our target systems, we propose a conceptually related approach that uses VM snapshots rather than processes, reduces the amount of control logic embedded in the SUT, and facilitates integration with trace-based EPA.

The resulting approach is illustrated in Figure 3, which continues the example from Figure 2. The execution of the target system starts in a virtual machine, VM0, where functions F_1 through F_3 are executed. When function F_4 is reached, the system notifies the controller (①), a separate process running outside the VM. The controller is where most of the control logic, which was embedded in the SUT in the FastFI model, resides in our approach. At this point, the controller suspends the execution of VM0 and instructs the VMM (we use QEMU [7]) to take a snapshot of the system. The controller decides how many VMs to create based on the number of available faulty versions of the current function and the desired degree of parallelism, spawns the new VMs, and resumes execution from the snapshot (②). Here, two parallel instances are used, VM1 and VM2, and they start executing F_4^1 and F_4^2. Execution is monitored by the controller, and once a VM finishes executing the workload, it is suspended, the snapshot is loaded and the next faulty version is executed. In our example, VM1 finishes executing F_4^1, is restored to the snapshot by the controller (③) and then executes F_4^3. One key advantage our approach of not embedding control logic in the SUT offers is that we can resume execution in VM0 before all faulty versions have finished executing. Therefore, in this example, VM0 can resume executing the unmodified system as soon as VM2 finishes and a CPU core becomes available (④). This is particularly valuable in cases where a faulty version causes the SUT to hang until a timeout detector is triggered as such cases no longer completely halt experiment progress, giving our approach a performance advantage over the FastFI approach. Experiment outcomes are logged in the controller. By the time VM0 and all other VM instances that have been spawned have finished, every faulty version that is reachable by the given workload will have been executed.

Our approach also allows for the integration of TrEKer-style EPA. We integrate the existing TrEKer implementation with our new approach by adjusting the TrEKer runtime to send trace entries to our controller. This integration offers benefits besides performance improvements (which are particularly important due to the overhead of memory access tracing). In particular, since our approach avoids re-executing common prefixes for different faulty versions as well as the original version of the system, the execution traces that have to be stored for later offline analysis get smaller and there is no opportunity for traces to diverge prior to fault activation due to non-deterministic behavior of the SUT. We study the impact this has on the false positive rate of the EPA in Sec. IV.

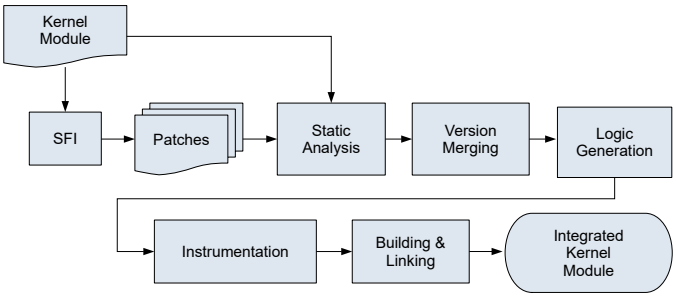


Fig. 4: An overview of our implementation.

D. Implementation

Our implementation consists of two parts. The first consists of the analyses and tools required to obtain a single, integrated kernel module containing multiple faulty versions. An overview of this process is shown in Figure 4. The second part is the runtime logic, consisting of the controller and runtime kernel module, which we describe in Sec. III-D2.

1) Compile Time: We start by applying an SFI tool[1] to the target kernel module, resulting in a number of patch files, each corresponding to a single faulty version. Next, we determine for each such patch what function it modifies and where that function is located in the original, unpreprocessed source code. We use a custom GCC plugin to efficiently obtain function location information and parse the patch files to create a mapping. With this information, we then perform the version merging and logic generation steps as shown in Figure 4. Instead of creating a copy of the kernel module for each faulty version, we merge all faulty versions into a single module by creating a copy of the modified function for each such version. We also include a separate copy of the original function implementation. Then, in the logic generation step, we replace the original function implementation with a small amount of runtime logic to determine which version should be executed. Unlike FastFI, this does not include any logic for orchestrating or monitoring the execution of different faulty versions. Rather, our support logic calls out to a runtime support module (Sec. III-D2), which in turn communicates with the controller as shown in Figure 3. We then instrument, build, and link the resulting code. By instrumenting only after merging all faulty versions, we avoid redundant instrumentation overhead resulting from instrumenting the same code multiple times. This yields a single, integrated kernel module which is fully instrumented and contains all faulty versions.

2) Runtime: The runtime of our implementation consists of two parts: The first is the controller, which runs on the host and is responsible for spawning VMs, controlling

[1]We use SAFE [22] to simulate representative residual software faults.

parallelism, monitoring experiment outcomes, and logging tracing information. The second is the runtime module, which is a kernel module that is loaded in the experiment VMs and enables communication between the controller and the experiment VMs as well as providing logging interfaces to the instrumented kernel modules and supporting functionality to the version selection logic described in Sec. III-D1.

We implemented the controller as a Rust program which manages the VMs for SFI experiments and performs result detection and logging. While we aim to achieve fast VM cloning (a crucial factor in realizing the speedups we are aiming for), we choose not to use a custom VMM to keep the applicability of our approach as broad as possible. Instead, we use QEMU and rely on its existing snapshotting functionality, along with a file system providing copy-on-write (CoW) functionality on the host, to quickly clone VMs. This lets us take advantage of the mature implementation, numerous features and supported (emulated) devices of QEMU while still achieving high performance when cloning VMs.

The runtime module is a Linux kernel module written in C. It provides an interface to the integrated kernel module to aid in version selection and handles communication with the controller over a VirtIO serial device. It also passes the log messages required for trace-based EPA to the controller over the same interface. Using a VirtIO device for logging, rather than, for instance, SSH, as in TrEKer, lets us keep overhead as well as noise on the system to a minimum.

At runtime, when the original VM reaches a function for which faulty versions exist, the following events occur:

1) The version selection logic in the integrated module calls the runtime module and provides the number of faulty versions that need to be executed.
2) The runtime module notifies the controller and blocks until a response is received.
3) The controller stops VM0 and takes a snapshot.
4) Depending on the desired degree of parallelism P, the controller creates up to P copies of the VM snapshot. To ensure that this step is fast, the snapshots are kept on a file system supporting lightweight CoW copies. In our implementation, we use a RAM-backed XFS file system.
5) The controller creates a workqueue for the faulty versions of the current function and spawns controller threads for each new VM, which start QEMU processes, load the snapshots, and respond to the runtime module indicating which faulty version to execute. These threads also monitor the execution to determine the SFI experiment outcome and log the tracing data relayed by the runtime module.
6) Depending on P, the controller either resumes VM0 or waits until one of the spawned VMs has finished executing.

When VM0 reaches another function for which a faulty version exists, this process is repeated. It is not repeated if VM0 reaches the same function again or if one of the spawned VMs reaches a function with faulty versions. This ensures that at most one faulty version is active in any VM, and none are active in VM0. When VM0 reaches the end of the workload and the controller detects that the experiment is complete, it waits for all spawned VMs to complete and terminates.

IV. Evaluation

We evaluate our approach by applying it to several real world Linux file systems. We provide a description of our experiment setup, target file systems, and workloads in Sec. IV-A. The research questions we address in this evaluation are detailed in Sec. IV-B. Experimental results are reported in Sec. IV-C

A. Experiment Setup

We first describe the execution environment we use for our experiments in Sec. IV-A1. We then cover our evaluation targets in Sec. IV-A2 and the workloads we use for our experiments in Sec. IV-A3.

1) Execution Environment: We conduct our experiments on the following two machines:

S_1: This system is equipped with an AMD Threadripper 2990WX CPU with 32 physical and 64 logical cores, 128 GiB of RAM and a 1 TB NVMe SSD.

S_2: This system is equipped with an AMD Threadripper 2970WX CPU with 24 physical and 48 logical cores, 64 GiB of RAM and a 1 TB NVMe SSD.

Both systems run Ubuntu 19.10. We use QEMU 4.0.0 as the VMM for our experiments, with KVM acceleration enabled. All SFI experiments are conducted on S_1, while S_2 is used to generate and build faulty versions.

2) Evaluation Targets: We apply our technique to 7 Linux file systems, of which an overview is given in Table I. All file systems are extracted from Linux 5.0.

The VMs we use in our SFI experiments are configured with 1 vCPU, 2 GiB of RAM and a `qcow2` disk, which is used by QEMU for snapshots but not visible to the guest system. All files required for our experiments with integrated kernel modules are placed in the `initramfs`, along with BusyBox 1.28.1. The VMs run Linux 5.0. We use a custom kernel configuration that supports the required VirtIO functionality used in our logging and controller implementation. To reduce scheduling and timing non-determinism, and noise in general, we disable preemption and run a tickless kernel. Since our approach assumes that VMs only have a single CPU core, we also disable SMP support in the kernel.

When we perform experiments using individually built faulty versions of kernel modules, placing all required files in `initramfs` would not be feasible, and regenerating the `initramfs` for each experiment execution would entail excessive per-execution overheads. Instead, we provide a read-only `virtfs` share to the VMs which contains all required kernel modules. For experiments comparing our approach to conventional SFI test execution with separate faulty versions, we use the same kernel as in experiments with our approach, and we take a VM snapshot after boot but prior to workload start to avoid having to reboot the VM after each experiment.

TABLE I: The Linux file systems used in our evaluation.

Module	Description	LOC
`hfsplus`	General purpose journaling read/write FS	9111
`isofs`	CD-ROM read-only FS	2922
`ntfs`	General purpose journaling FS, limited read/write	17021
`overlayfs`	Union mount read/write FS	7086
`romfs`	RomFS EEPROM read-only FS	722
`squashfs`	Compressed read-only FS	2791
`vfat`	General purpose read/write FS	6328

3) Evaluation Workloads: Even though all our evaluation targets are file systems, we cannot use identical workloads across all of them. This is because some of the included file systems are read-only, whereas others differ in their supported feature set. We use the following workloads in our experiments:

- *Read-write*: This is the workload we use for full-featured file systems. It encompasses module insertion, file system mounting, a variety of common file system operations such as directory listing, file creation and deletion, reading and writing, unmounting, and module removal. It is used for `hfsplus` and `vfat`.
- *Limited read-write*: We use this workload for our experiments with the `ntfs` file system as the kernel module does not support file creation. The workload resembles the regular read-write workload apart from the omission of the file creation step.
- *Read-only*: This is the workload we use for read-only file systems. It encompasses module insertion, mounting, common file system operations for read only file systems (i.e., directory listing and reading), unmounting, and module removal. This workload is used for `isofs`, `romfs`, and `squashfs`.
- *Overlay*: This is the workload we use for `overlayfs`. Since the prepared file system images used in other workloads are not applicable for `overlayfs`, we use this specialized workload. It resembles the read-write workload but does not use a prepared image.

B. Research Questions

To evaluate the impact our proposed approach has on the performance and precision of kernel SFI and EPA, we investigate the following research questions:

RQ 1 Can our approach speed up sequential SFI test execution?

RQ 2 Can our approach speed up parallel SFI test execution?

RQ 3 How does our approach affect the number of executed faulty versions?

RQ 4 Can our approach reduce build times for SFI experiments?

RQ 5 Does our approach affect SFI result validity?

RQ 6 How does our approach affect detection rates and false positives in error propagation analysis?

TABLE II: Number of executed and activated faulty versions in each execution mode.

Module	Classic Execution				Integrated Execution			
	Executed		Activated		Executed		Activated	
	Abs	Rel %	Abs	Rel %	Abs	Rel %	Abs	Rel %
`hfsplus`	2885	100	1228	42.56	1814	62.88	1235	68.08
`isofs`	1519	100	799	52.6	1246	82.03	799	64.13
`ntfs`	7158	100	2432	33.98	4997	69.81	2438	48.79
`overlayfs`	4180	100	1527	36.53	2117	50.65	1527	72.13
`romfs`	289	100	250	86.51	272	94.12	250	91.91
`squashfs`	1107	100	654	59.01	929	83.92	654	70.4
`vfat`	3210	100	1468	45.7	1960	61.06	1468	74.9

C. Results

In the following, we report our experimental results. All reported numbers, excepting build times, are averages over three repeated executions.

1) RQ 1: Sequential Speedup: To determine whether our approach can speed up sequential SFI testing, we compare execution times between our approach and the conventional execution model using faulty versions with TrEKer instrumentation. The speedups we achieve over the conventional execution model are shown in Figure 5. For sequential execution, the relevant numbers are the speedups reported above the bars for a degree of parallelism of 1 for each target file system. The speedups we achieve range from 1.32× for `squashfs` to 2.45× for `overlayfs`. As we do not make use of parallelism here, these speedups are entirely the result of the ability of our approach to execute fewer faulty versions (which we discuss in more detail in Sec. IV-C3) and its ability to avoid the common prefix problem discussed in Sec. III-C.

Our approach beats the conventional model for all evaluated file systems with speedups from 1.32× to 2.45×. We conclude that our approach can speed up sequential SFI testing.

2) RQ 2: Parallel Speedup: We investigate whether our approach is capable of accelerating parallel SFI test execution by comparing it to the conventional execution model across different degrees of parallelism ranging from 1 to 32.

Figure 5 shows, for each file system and degree of parallelism, the speedup achieved relative to sequential execution in the same mode. The factors above the bars correspond to the speedups achieved by our approach over the conventional execution model at the same degree of parallelism. Times inside or above the bar give the absolute execution time.

We see that, except for `squashfs` at a degree of parallelism of 32, our approach always outperforms the conventional execution model at the same degree of parallelism. However, it is also apparent that speedups relative to conventional execution at the same degree of parallelism reduce with increasingly parallel execution. This is because, in some cases, our approach is not able to utilize the full computational resources available at higher degrees of parallelism throughout the entire SFI test execution. This can happen when, for example, the faulty versions for an SFI target are distributed across a large number of functions so that there are only a few faulty versions per

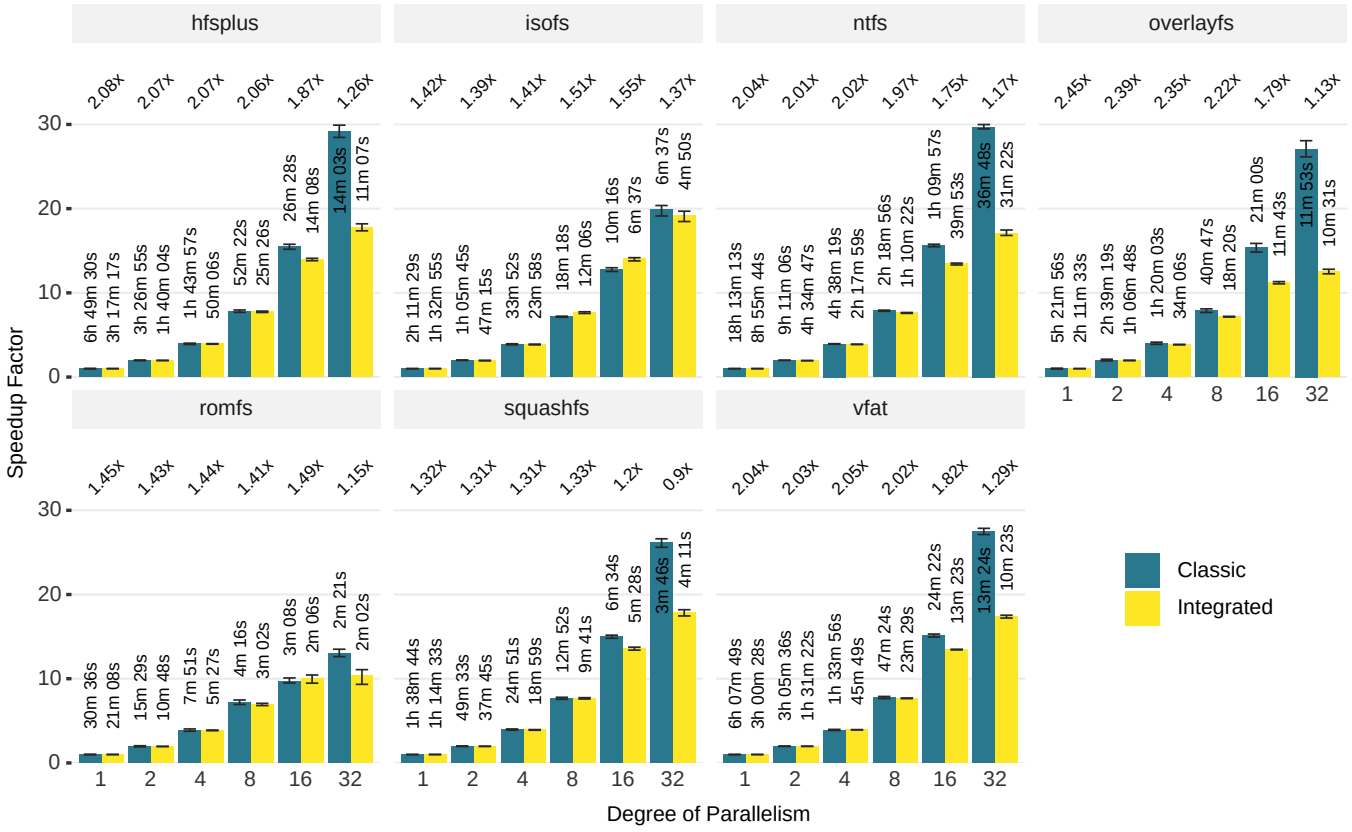


Fig. 5: Execution times and relative speedups for SFI experiments (based on wall time). Speedups compared to sequential execution are shown on the Y-axis. Speedups of integrated compared to conventional execution are shown at the top. Absolute times are reported above/inside bars. Average results of three repetitions are shown, with error bars indicating min/max values.

function. In that case, our approach will not be able to fully utilize all available CPU cores with the faulty versions of a single function, and some cores will remain idle until the root VM (VM0) has reached the next function for which faulty versions exist. The conventional execution model, on the other hand, can simply run as many VMs in parallel as it has cores available. We note that, while wall time speedups for our approach are lower at higher degrees of parallelism, user time speedups remain high since our approach reduces the overall code to be executed, with `squashfs` at a degree of parallelism of 32 achieving a user time speedup of 15.2×.

Looking at speedups relative to sequential execution in the same mode as shown in Figure 5, it appears that our approach scales less well than the conventional execution model. This is for the reasons related to CPU utilization described above. However, our approach does achieve increasing speedups with increasing degree of parallelism for all file systems, with further optimization potential in our prototype implementation.

We conclude that our approach is capable of accelerating parallel SFI test execution, although the benefits in execution time diminish for very high degrees of parallelism.

3) RQ 3: Executed Versions: We investigate how our approach affects the number of executed faulty versions and activated faults by comparing the number of executed and activated faulty versions in the conventional execution model and in our approach as reported in Table II. The table lists executed faulty versions, both absolute and as a percentage of all faulty versions, and activated faulty versions, absolute and as a percentage of executed faulty versions in the same mode. The conventional execution model requires execution of every faulty version. Therefore, 100 % of faulty versions are executed in this mode. This includes faulty versions with faults in functions that the workload does not trigger, resulting in activation rates below 60 % for all file systems except `romfs`, which, due to its fairly simple structure, achieves quite high code coverage with our workload and a fault activation rate of 86.51 %. Our approach requires the execution of fewer faulty versions than the conventional execution model, ranging from 50.65 % to 94.12 %. This corresponds to a reduction in the number of executed faulty versions of 250 to 2438. As this reduction is the result of not executing inactive faulty versions, our approach also reaches higher activation rates ranging from 48.79 % to 91.91 %. While, in the conventional execution model, we only saw an activation rate above 60 % for a single file system, with our approach, all file systems but one achieve such an activation rate.

We conclude that our approach can effectively reduce the number of faulty versions that need to be executed in SFI

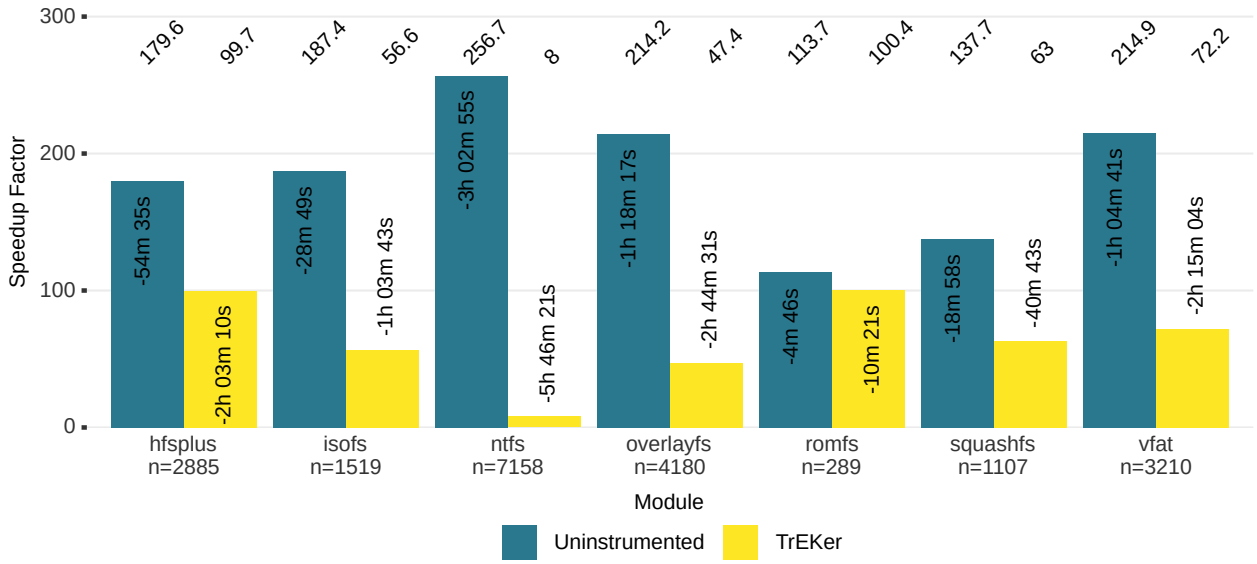


Fig. 6: Build time speedups with and without instrumentation

testing. We achieved fewer executed faulty versions and higher activation rates for all evaluated file systems.

4) RQ 4: Build Times: We investigate the impact of our approach has on build times for SFI experiments, both with the instrumentation required for kernel error propagation analysis using TrEKer and without it. User build time speedups for all 7 file systems used in our evaluation, both instrumented and uninstrumented, are shown in Figure 6. Absolute time savings are listed vertically inside or above the bars. Speedup factors are shown on the y axis and above the bars. For all modules and in both modes, building an integrated module containing all faulty versions is substantially faster than building each faulty version separately. Note that we use incremental compilation in the latter case to minimize the work required for each faulty version. Speedups range from 8× to over 250×, with the instrumented case achieving lower speedups across all modules. This is particularly noticeable for `ntfs`, which is also the file system with the most faulty versions in our evaluation. We attribute this to inefficiencies in the instrumentation code and note that, although it achieves the lowest speedup, the instrumented `ntfs` build is also the one with the largest absolute time reduction.

With speedups ranging from 8× to 256×, we conclude that our approach can substantially reduce build times for SFI experiments.

5) RQ 5: Result Validity: We investigate the impact of our approach on SFI results when traditional failure mode detectors are used and whether result validity is adversely affected. To that end, we report on the observed results for integrated executions at different degrees of parallelism with activated fault and compare them to the results for sequential conventional executions. Figure 7 shows the observed result distributions for all evaluation targets. The sequential conventional executions are labeled *seq* on the X axis; the integrated executions are labeled with the respective degree of parallelism (1 to 32). We distinguish four common classes of SFI test results (failure modes): "No Failure": The test run finished without any error indication; "Timeout": The test run did not finish within its execution time budget and was terminated; "Workload Failure": The test run terminated with an error indication from user-mode; "Kernel Failure": The test run terminated with an error indication from the kernel (e.g. kernel panic). We set the execution time limit to 30 s for all runs since this value is considerably higher than the fault-free, but instrumented, execution time for all our target modules and should hence avoid premature timeout detections.

Across all modules and execution modes, "No Failure" is the most common result, with the exception of `romfs` for which "Workload Failure" is the most common. "Timeout" is the least common result with less than 10 % of runs having this outcome, which suggests that our execution time budget choice was sensible. The result distributions remain stable across repeated runs and across different degrees of parallelism. We observe the largest variability for `hfsplus` ($P = 16$) in both execution modes with a maximum difference of result class counts of 30 (less than 3 %) between repeated executions. To assess whether the result distributions are significantly affected by integrated execution when compared to conventional sequential execution, we conduct pairwise Pearson's χ^2-tests of independence between the conventional sequential execution and each integrated execution. We test the null hypothesis H_0 that "the obtained result distribution is independent from the execution mode", with the alternative hypothesis H_1 that "there is an association between result distribution and execution mode". We try to reject H_0 with a significance level of $\alpha = 0.05$. Table III reports the resulting p and Cramer's V values along with the decision whether we can reject H_0. Cramer's V is a measure of association based on the χ^2-statistic and ranges from 0 to 1, where the larger the value, the

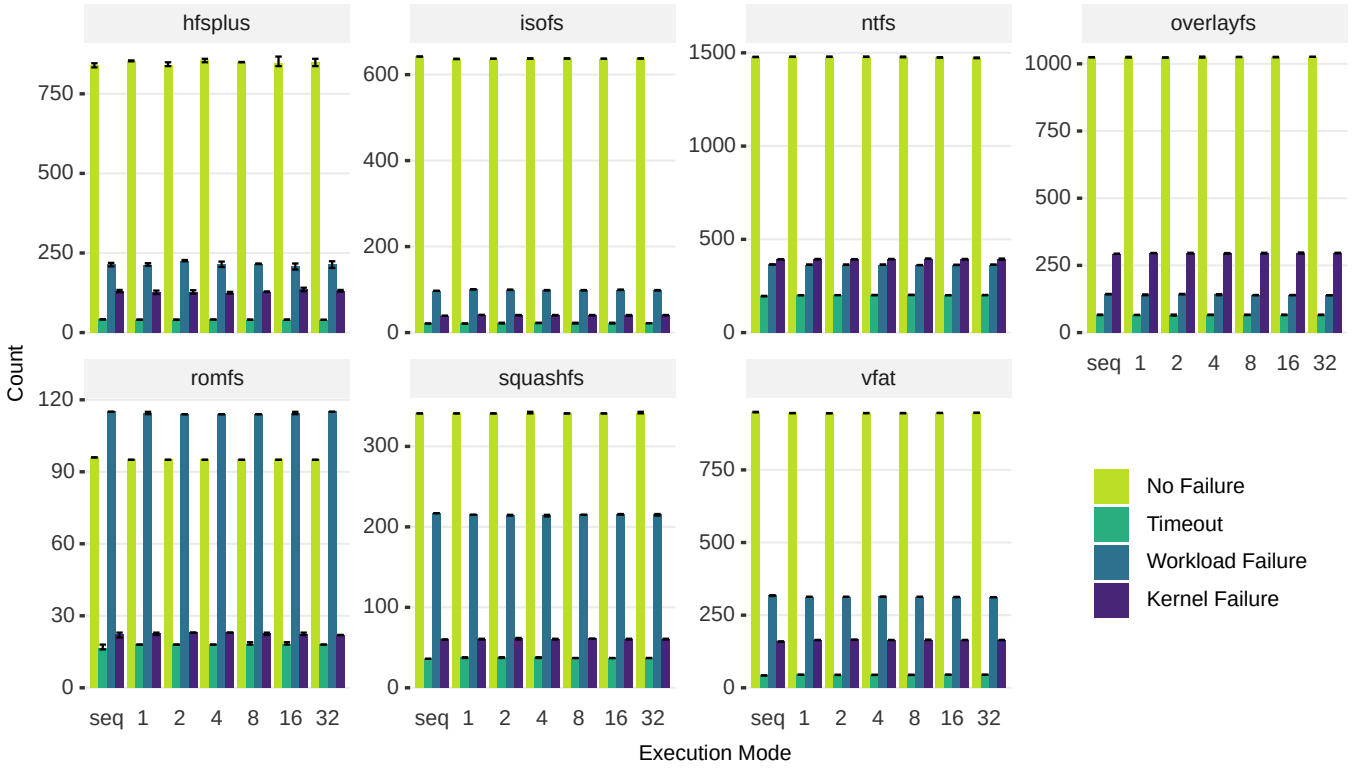


Fig. 7: Result distributions for SFI experiments with classic failure mode detection. Average results of three repetitions across sequential classic execution (seq) and six different parallelism degrees for integrated execution (1 to 32) for all runs with activated fault are shown. Error bars indicate min/max values.

TABLE III: Results of χ^2-tests. p values and Cramer's V are reported for each test. The *Rej* column indicates if H_0 (no association between SFI results and execution mode) can be rejected (✓) or not (✗).

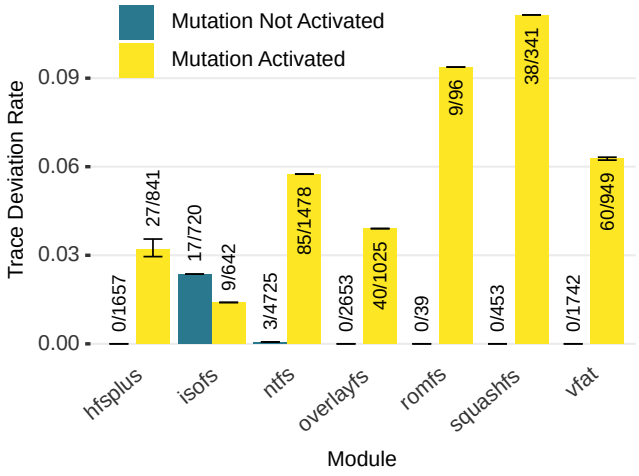
	par = 1			par = 2			par = 4			par = 8			par = 16			par = 32		
Module	p	V	Rej	p	V	Rej	p	V	Rej	p	V	Rej	p	V	Rej	p	V	Rej
`hfsplus`	0.943	0.0072	✗	0.786	0.0120	✗	0.818	0.0112	✗	0.974	0.0055	✗	0.925	0.0080	✗	0.983	0.0048	✗
`isofs`	0.943	0.0090	✗	0.957	0.0081	✗	0.963	0.0077	✗	0.967	0.0074	✗	0.955	0.0083	✗	0.967	0.0074	✗
`ntfs`	0.981	0.0035	✗	0.978	0.0037	✗	0.980	0.0036	✗	0.953	0.0048	✗	0.974	0.0039	✗	0.969	0.0042	✗
`overlayfs`	0.986	0.0040	✗	0.999	0.0018	✗	0.998	0.0021	✗	0.981	0.0044	✗	0.979	0.0045	✗	0.969	0.0052	✗
`romfs`	0.980	0.0110	✗	0.976	0.0119	✗	0.976	0.0119	✗	0.965	0.0135	✗	0.967	0.0132	✗	0.981	0.0109	✗
`squashfs`	0.990	0.0053	✗	0.987	0.0059	✗	0.983	0.0065	✗	0.993	0.0049	✗	0.995	0.0043	✗	0.995	0.0043	✗
`vfat`	0.900	0.0081	✗	0.888	0.0085	✗	0.916	0.0076	✗	0.897	0.0082	✗	0.898	0.0082	✗	0.884	0.0086	✗

stronger the association. With $p \gg \alpha$ for all modules across all execution modes, we fail to reject H_0. Hence, we cannot establish that there is an association between result distributions and execution mode. Accordingly, Cramer's V does not hint at association with $V < 0.015$ for all tests.

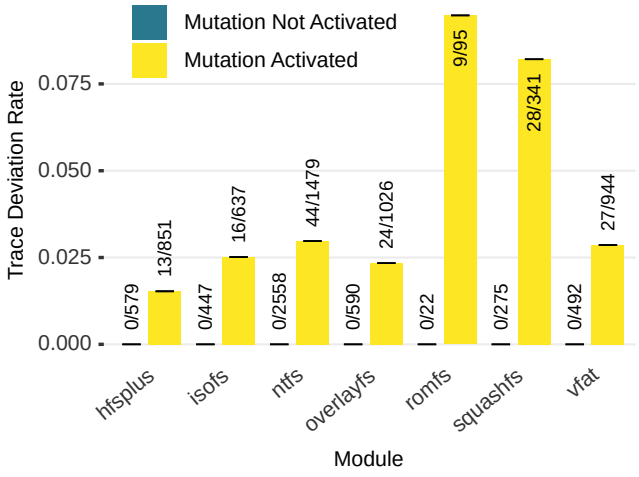
We conclude that integrated execution does not affect SFI results of traditional failure mode detectors when compared to conventional sequential execution.

6) RQ 6: Detection Rates: To investigate the effect of integrated execution on the trace deviation rates detected by the TrEKer EPA, we analyze the execution traces that we collected during the sequential runs of our SFI experiments in both conventional and integrated execution. As apparent from Figure 7, the "No Failure" class is the most common SFI outcome. More than 60 % of the runs with activated fault for all modules fall into this class, with the exceptions of `romfs` with 38 % and `squashfs` with 52 %. For EPA, this is the most interesting result class as traditional failure mode detectors cannot detect any deviation despite the fault being activated. Either the injected fault is benign and has no effect or the effects have not manifested yet in a way that can be detected by traditional detectors. Both cases can be distinguished by EPA techniques such as TrEKer since execution trace deviations can be detected in the latter case. We report the TrEKer trace deviation rates for that case in Figure 8 with the bars labeled "Mutation Activated". In addition to the rates, the plots also contain the absolute numbers inside or above the bars.

In order to assess the false positive detection rates, we



(a) Conventional Execution Mode



(b) Integrated Execution Mode

Fig. 8: Trace deviation rates for both execution modes. Averages from three repetitions are shown. Error bars indicate min/max values. Numbers inside/above bars indicate absolute counts in the format detections/total.

also perform an analysis of "No Failure" run traces without activated fault (labeled accordingly in the plots). Any deviation detected in such a run cannot occur due to an injected fault and is therefore a false positive. The TrEKer analysis detects deviations by comparing execution traces against fault-free golden run traces, i.e., traces without injected faults. Since we took special care to create a low-noise and deterministic execution environment for our experiments, we use only a single golden run as comparison basis. We also attempted comparisons against 1000 golden runs, but we did not see meaningful changes in detection rates that would justify the increased analysis effort.

Overall, we observe low false positive rates, which are at 0 % across all modules when using integrated execution. In conventional execution, we observe some false positives for `isofs` (2.4 %) and `ntfs` (below 0.1 %). The deviation rates with activated fault range from 1.4 % to 11.1 % for conventional and from 1.5 % to 9.5 % for integrated execution. Overall, fewer deviations are detected with integrated execution for all modules, except for `isofs`. We attribute this reduction to less noise in the execution traces due to the fine-grained VM snapshotting employed for integrated execution.

With a 0 % false positive rate across all our target modules when comparing against a single golden run, we conclude that integrated execution is effective and does not introduce additional noise.

V. Discussion

As the investigation of our research questions in Sec. IV shows, our approach is applicable to real world kernel code and can effectively accelerate sequential and parallel SFI testing without adversely affecting result validity. We therefore conclude that our approach enables effective SFI testing of kernel code on modern, parallel hardware.

We identify the following three main threats to the validity of our study:

1) The choice of target modules, system setup, configuration, and workload.
2) Limitations of the error propagation analysis approach used in our study.
3) Interactions between non-deterministic or timing-dependent behavior in the system under test (SUT) and our snapshot-based experiments.

We evaluate our approach on seven widely used file systems from the Linux kernel using different workloads to exercise common file system functionality. However, kernel modules implementing different, unrelated functionality may behave differently. A different choice of workload may also yield different results. We tailor the configuration of the kernel on the target system to minimize noise and facilitate our VM cloning approach. To this end, we disable SMP support and preemption and run a small, BusyBox-based userspace. Different kernel configurations may increase scheduling noise, thereby affecting results or result stability. A more full-featured userspace, for example, from a common desktop-focused Linux distribution, could also increase noise on the system and affect the experimental results. Our system configuration is a likely reason for the lower false positive rate we observed relative to the TrEKer experiments using the same error propagation analysis approach. Such a target system may, for instance, require substantially more golden runs to achieve stable results. Our results may not generalize to other target modules, workloads, or configurations.

We use an augmented version of the TrEKer error propagation analysis in our experiments. As noted in [10], this approach restricts instrumentation and trace analysis in some respects to reduce overhead and improve performance. It may therefore miss behavioral divergences in some instances. The detection rates reported in Sec. IV-C6 are subject to these restrictions. As we do not depend on the accuracy of the reported detection rates in our investigation of our other research questions, these limitations do not otherwise threaten the validity of our results.

Even though we have taken care to minimize noise and non-deterministic behavior on the system with our configuration, the SUT may still exhibit non-deterministic behavior between different executions, or timing-dependent behavior that leads to seemingly non-deterministic variations between executions. Our snapshot-based execution model may limit the first but could potentially increase the second. This can, in turn, influence the detection rates of the error propagation analysis. Since our approach does not result in any false positives for the modules used in our evaluation, this does not seem to occur in our evaluation, but, as noted above, other configurations may yield different results.

VI. Conclusion

In this paper, we introduced a novel approach for accelerating the SFI testing and error propagation analysis of operating system kernel code. Our approach speeds up SFI test execution in three ways: We avoid redundant code execution, automatically skip the execution of inactive faulty versions, and facilitate paralleziation of SFI experiment execution. Moreover, our approach substantially reduces build times by integrating all faulty versions into a single module, thereby avoiding redundant compilation effort.

In our evaluation on seven widely used Linux file system implementations, we achieve sequential speedups of up to 2.45×, parallel speedups of up to 36.8× using 16 parallel instances, and build time speedups of up to 100.4× with instrumentation and 256.7× without. SFI result validity is not affected by these speedups.

Acknowledgments

This research work has been funded by the German Federal Ministery of Education and Research and the Hessen State Ministry for Higher Education, Research and the Arts within their joint support of the National Research Center for Applied Cybersecurity ATHENE. This work was supported in part by EC H2020 CONCORDIA GA#830927 and the Lancaster Security Institute.

References

[1] J. Aidemark, J. Vinter, P. Folkesson, and J. Karlsson. "GOOFI: generic object-oriented fault injection tool." In: *Proc. of DSN*. 2001, pp. 83–88.

[2] M. R. Aliabadi, K. Pattabiraman, and N. Bidokhti. "Soft-LLFI: A Comprehensive Framework for Software Fault Injection." In: *Proc. of International Symposium on Software Reliability Engineering Workshops (ISSREW)*. 2014, pp. 1–5.

[3] A. Avizienis, J. C. Laprie, B. Randell, and C. Landwehr. "Basic concepts and taxonomy of dependable and secure computing." In: 1.1 (Jan. 2004), pp. 11–33. ISSN: 1545-5971. DOI: 10.1109/TDSC.2004.2.

[4] Radu Banabic and George Candea. "Fast black-box testing of system recovery code." In: *Proc. EuroSys*. 2012, pp. 281–294.

[5] Takayuki Banzai, Hitoshi Koizumi, Ryo Kanbayashi, Takayuki Imada, Toshihiro Hanawa, and Mitsuhisa Sato. "D-Cloud: Design of a Software Testing Environment for Reliable Distributed Systems Using Cloud Computing Technology." In: *Proc. CCGRID*. 2010, pp. 631–636.

[6] Jonathan Bell and Gail Kaiser. "Unit Test Virtualization with VMVM." In: *Proc. ICSE*. 2014, pp. 550–561.

[7] Fabrice Bellard. "QEMU, a Fast and Portable Dynamic Translator." In: *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. ATC '05. Anaheim, CA: USENIX Association, 2005, pp. 41–41. URL: http://dl.acm.org/citation.cfm?id=1247360.1247401.

[8] Prashanth P. Bungale and Chi-Keung Luk. "PinOS: A Programmable Framework for Whole-system Dynamic Instrumentation." In: *Proc. of VEE*. 2007, pp. 137–147. ISBN: 978-1-59593-630-1. DOI: 10.1145/1254810.1254830.

[9] J. Carreira, H. Madeira, and J. G. Silva. "Xception: a technique for the experimental evaluation of dependability in modern computers." In: 24.2 (Feb. 1998), pp. 125–136. ISSN: 0098-5589. DOI: 10.1109/32.666826.

[10] Nicolas Coppik, Oliver Schwahn, Stefan Winter, and Neeraj Suri. "TrEKer: Tracing Error Propagation in Operating System Kernels." In: *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*. ASE 2017. Urbana-Champaign, IL, USA: IEEE Press, 2017, pp. 377–387. ISBN: 978-1-5386-2684-9. URL: http://dl.acm.org/citation.cfm?id=3155562.3155612.

[11] Alexandre Duarte, Walfredo Cirne, Francisco Brasileiro, and Patricia Machado. "GridUnit: Software Testing on the Grid." In: *Proceedings of the 28th International Conference on Software Engineering*. ICSE '06. ACM, 2006, pp. 779–782. ISBN: 978-1-59593-375-1. DOI: 10.1145/1134285.1134410. URL: http://doi.acm.org/10.1145/1134285.1134410.

[12] Peter Feiner, Angela Demke Brown, and Ashvin Goel. "Comprehensive Kernel Instrumentation via Dynamic Binary Translation." In: *Proc. of ASPLOS*. 2012, pp. 135–146. ISBN: 978-1-4503-0759-8. DOI: 10.1145/2150976.2150992.

[13] A. Gambi, A. Gorla, and A. Zeller. "O!Snap: Cost-Efficient Testing in the Cloud." In: *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. Mar. 2017, pp. 454–459. DOI: 10.1109/ICST.2017.51.

[14] T. Hanawa, T. Banzai, H. Koizumi, R. Kanbayashi, T. Imada, and M. Sato. "Large-Scale Software Testing Environment Using Cloud Computing Technology for Dependable Parallel and Distributed Systems." In: *Proc. ICSTW*. 2010, pp. 428–433.

[15] A. Henderson, L. Yan, X. Hu, A. Prakash, H. Yin, and S. McCamant. "DECAF: A Platform-Neutral Whole-System Dynamic Binary Analysis Platform." In: PP.99 (2016), pp. 1–1. ISSN: 0098-5589. DOI: 10.1109/TSE.2016.2589242.

[16] Andrew Henderson, Aravind Prakash, Lok Kwong Yan, Xunchao Hu, Xujiewen Wang, Rundong Zhou, and Heng Yin. "Make It Work, Make It Right, Make It Fast: Building a Platform-neutral Whole-system Dynamic Binary Analysis Platform." In: *Proc. of ISSTA*. 2014, pp. 248–258. ISBN: 978-1-4503-2645-2. DOI: 10.1145/2610384.2610407.

[17] Piyus Kedia and Sorav Bansal. "Fast Dynamic Binary Translation for the Kernel." In: *Proc. of SOSP*. 2013, pp. 101–115. ISBN: 978-1-4503-2388-8. DOI: 10.1145/2517349.2522718.

[18] Horacio Andrés Lagar-Cavilla, Joseph Andrew Whitney, Adin Matthew Scannell, Philip Patchin, Stephen M. Rumble, Eyal de Lara, Michael Brudno, and Mahadev Satyanarayanan. "SnowFlock: Rapid Virtual Machine Cloning for Cloud Computing." In: *Proceedings of the 4th ACM European Conference on Computer Systems*. EuroSys '09. Nuremberg, Germany: ACM, 2009, pp. 1–12. ISBN: 978-1-60558-482-9. DOI: 10.1145/1519065.1519067. URL: http://doi.acm.org/10.1145/1519065.1519067.

[19] Wing Lam, Sai Zhang, and Michael D. Ernst. *When Tests Collide: Evaluating and Coping with the Impact of Test Dependence*. Tech. rep. University of Washington Department of Computer Science and Engineering, 2015.

[20] Anna Lanzaro, Roberto Natella, Stefan Winter, Domenico Cotroneo, and Neeraj Suri. "An Empirical Study of Injected Versus Actual Interface Errors." In: *Proc. of ISSTA*. 2014,

pp. 397–408. ISBN: 978-1-4503-2645-2. DOI: 10.1145/2610384.2610418.

[21] Sasa Misailovic, Aleksandar Milicevic, Nemanja Petrovic, Sarfraz Khurshid, and Darko Marinov. "Parallel Test Generation and Execution with Korat." In: *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*. ESEC-FSE '07. ACM, 2007, pp. 135–144. ISBN: 978-1-59593-811-4. DOI: 10.1145/1287624.1287645. URL: http://doi.acm.org/10.1145/1287624.1287645.

[22] R. Natella, D. Cotroneo, J.A. Duraes, and H.S. Madeira. "On Fault Representativeness of Software Fault Injection." In: 39.1 (Jan. 2013), pp. 80–96. ISSN: 0098-5589.

[23] T. Parveen, S. Tilley, N. Daley, and P. Morales. "Towards a Distributed Execution Framework for JUnit Test Cases." In: *2009 IEEE International Conference on Software Maintenance*. Sept. 2009, pp. 425–428. DOI: 10.1109/ICSM.2009.5306292.

[24] M. Sand, S. Potyra, and V. Sieh. "Deterministic high-speed simulation of complex systems including fault-injection." In: *Proc. of DSN*. 2009, pp. 211–216. DOI: 10.1109/DSN.2009.5270335.

[25] O. Schwahn, N. Coppik, S. Winter, and N. Suri. "FastFI: Accelerating Software Fault Injections." In: *2018 IEEE 23rd Pacific Rim International Symposium on Dependable Computing (PRDC)*. Dec. 2018, pp. 193–202. DOI: 10.1109/PRDC.2018.00035.

[26] Oliver Schwahn, Nicolas Coppik, Stefan Winter, and Neeraj Suri. "Assessing the State and Improving the Art of Parallel Testing for C." In: *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2019. Beijing, China: ACM, 2019, pp. 123–133. ISBN: 978-1-4503-6224-5. DOI: 10.1145/3293882.3330573. URL: http://doi.acm.org/10.1145/3293882.3330573.

[27] Anna Thomas and Karthik Pattabiraman. "LLFI: An intermediate code level fault injector for soft computing applications." In: *Proc. of SELSE*. 2013.

[28] Stefan Winter, Oliver Schwahn, Roberto Natella, Neeraj Suri, and Domenico Cotroneo. "No PAIN, No Gain?: The Utility of PArallel Fault INjections." In: *Proc. ICSE*. 2015, pp. 494–505.

[29] Bowen Zhou, Milind Kulkarni, and Saurabh Bagchi. "WuKong: Effective Diagnosis of Bugs at Large System Scales." In: *SIGPLAN Not.* 48.8 (Feb. 2013), pp. 317–318. ISSN: 0362-1340. DOI: 10.1145/2517327.2442563.