Karlsruhe Institute of Technology

# Crash Consistency Testing for Block Based File Systems on NVMe Drives

Bachelor's Thesis
submitted by

Daniel Ritz

to the KIT Department of Informatics

| | |
|---|---|
| Reviewer: | Prof. Dr. Frank Bellosa |
| Second Reviewer: | Jun.-Prof. Dr. Christian Wressnegger |
| Advisor: | Lukas Werling, M.Sc. |

16. May 2022 – 16. September 2022

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any source or auxiliary means other than these referenced. This thesis was carried out in accordance with the Rules for Safeguarding Good Scientific Practice at Karlsruhe Institute of Technology (KIT).

Karlsruhe, September 16, 2022

# Abstract

File system consistency in case of crashes is important to prevent data loss or file system corruption. Unfortunately, common file systems give only few and inconsistent guarantees regarding the effects of crashes. To help application developers as well as file system developers to find and understand issues caused by crashes, tools for automated crash consistency testing are needed.

We provide a tool for automated crash consistency testing of unmodified full systems with block based file systems on NVMe drives. Our approach uses trace-and-replay to test for different guarantees, such as atomicity or writeability after a crash. We analyse a virtualised system by tracing NVMe commands with an emulated NVMe drive. Afterwards, a Crash Image Generator creates a set of file system images that can originate from a system crash. These crash images are then examined for their semantic file system state. These states can then be used to validate file system guarantees.

We could verify the practicability of this approach with tests on `vfat` and `ext4` file systems. Our analysis of `ext4` could reproduce a known bug and found an issue with `mkdir`.

# Contents

**Abstract** **v**

**Contents** **1**

**1 Introduction** **3**

**2 Background and Related Work** **5**
2.1 File Systems . . . . . 5
2.2 NVMe . . . . . 6
2.3 Crash Consistency . . . . . 6
2.4 Virtualisation and Emulation . . . . . 8
2.5 Crash Consistency Testing . . . . . 8

**3 Design** **11**
3.1 Overview . . . . . 11
3.2 Tracer . . . . . 12
3.3 Crash Image Generator . . . . . 14
3.4 Tester . . . . . 16

**4 Implementation** **17**
4.1 Disk Images . . . . . 17
4.2 Virtualised System . . . . . 18
4.3 Tracer . . . . . 18

**5 Evaluation** **21**
5.1 File Systems . . . . . 21
5.2 Reproduced Bugs . . . . . 22
5.3 Test Cases . . . . . 23
5.4 Observations . . . . . 26
5.5 Crash Image Generation Algorithms . . . . . 28
5.6 Performance . . . . . 30

**6 Discussion** **33**
6.1 Tracing and Crash Images . . . . . 33
6.2 Evaluation . . . . . 34
6.3 Extendability . . . . . 35

**7 Conclusion** **37**
7.1 Future Work . . . . . 38

**Bibliography** **39**

# Chapter 1

# Introduction

File systems are the foundation of persistent data storage in modern operating systems. Their reliability is therefore of outright importance. Unfortunately, file systems are prone to inconsistencies resulting from system crashes [3]. Crash consistency testing is relevant to reason about the consequences of crashes. Especially, because different file systems give incoherent guarantees regarding their persistence behaviour [4].

There have been different works examining crash consistencies in file systems. These works range from symbolic validations [4] to trace-and-replay solutions [8, 13]. Our work focuses on full system tests without source code modifications. We base our work on the Vinter test framework for NVM file systems [8], but apply its techniques to block based files systems on NVMe drives.

Our tool Revin uses a trace-and-replay approach. We use QEMU to trace NVMe commands issued by a virtualised system. This trace is used to generate images of possible file system states that can originate from a crash. These crash images are then mounted to a virtual system to examine their semantic file system state as seen by applications. Revin groups the resulting semantic states together to form a minimal set of post-crash states. These states are then used to infer file system guarantees, such as atomicity or retained writeability.

The number of crash images can increase quickly. It is therefore relevant to find strategies to limit the generated crash images without missing relevant states. We therefore assess three different algorithms to generate them.

We test Revin with `vfat` and `ext4` with and without data journaling. The tests reproduce a known bug and show issues with `mkdir` in `ext4`. Furthermore, the semantic states clearly show the impact of journaling. We could additionally confirm the practicability of using this trace-and-replay approach if we reasonably limit the number of crash images.

Due to our approach being similar to the one used by Vinter, we see a possibility of combining both tools. This could enable crash consistency testing for

hybrid file systems relying on NVM and block based storage. Nevertheless, such a merge requires adjustment of the tracing methods used.

We structure this work as follows: Chapter 2 gives an overview of the crash consistency problem, the tools used and related work. We then present our design in Chapter 3 and discuss the implementation in Chapter 4. Chapter 5 presents the results of our tests, discusses observations and concludes with performance tests. Issues with the design, as well as differences to Vinter are discussed in Chapter 6. Finally, we summarise our work in Chapter 7.

# Chapter 2

# Background and Related Work

The subject of this work are block based file systems on NVMe drives. We therefore start with a quick overview over file systems and the NVMe protocol for communication with NVMe storage devices. Afterwards, we introduce crash consistency and present journaling as a solution to prevent data loss. We discuss general file system guarantees as well. The next topic, virtualisation and emulation prepares for the technologies used in crash consistency testing. Finally, we present previous work in this field.

## 2.1 File Systems

File systems are an essential part of the storage abstraction in a modern operating system. A file system organises data in named containers called “files”. These files are catalogued in “directories”, which form a hierarchical file system tree. Multiple file systems can be part of this tree. The intersections of these file systems are called “mount points”. UNIX expands upon this idea by integrating hardware abstractions into this file system tree, for example by providing access to serial devices with pseudo-files like “/dev/serial0”. We ignore these special files for this work and focus on regular files, directories and symbolic links. [3]

Every file (or directory) consists of metadata and content data. The metadata is usually organised in a structure called “inode”. An inode can be referenced by multiple directories with different names. These names are called hard links. Therefore, hard links can only exist within the same file system. Not all file systems support multiple hard links to a file. Some file systems even embed the inode in the directory entries. Nevertheless, we keep this nomenclature regardless of implementation. Symbolic links are special files which point to other files or directories by providing an address to them (called a “path”). Symbolic links can reference files in other file systems or even non-existing files. [3]

## 2.2 NVMe

One standard for communication with non-volatile memory storage is NVMe [14]. NVMe utilises PCI Express or fabric as underlying connection. The communication is designed around submission and completion queues. The driver puts IO commands in the submission queue and signals the NVMe device that a new entry is available. The device then executes the command. When command execution is finished, the device adds a completion entry to the completion queue, which can then be read by the driver [14, Section 1.4]. A pair of submission and completion queues is called a command queue..

There are two types of command queues: The admin command queue and the IO command queue. The former one is used to allocate IO command queues and to manage the device. NVMe supports multiple IO command queues in parallel but only one admin command queue. We will focus on the IO command queues. These queues can accept different commands; the most important ones being `flush`, `read`, `write` and `write zeroes` [14, Section 6]. Although being ordered in a submission queue, there is no guarantee in which order the commands are executed [14, Section 6.2]. There are atomicity guarantees for commands up to a specific size (dependent on the hardware), but in general, commands can be interleaved [14, Section 6.4]. Especially, no power failure guarantees are given [14, Section 6.4.2.1].

A NVMe device can use a write cache. This cache is used to serve `read` operations before the corresponding data is persistently stored. The cache can be forced to be written to storage with the `flush` command. This command ensures that any operation marked as complete before the submission of the `flush` command is stored non-volatile [14, Section 6.8]. Due to this condition, `flush` can be safely reordered on device.

For this work, we make simplifications. First, we treat NVMe commands as being atomic. Any command is either run in full or not run at all. Secondly, we assume that `flush` commands are reordering barriers. As flushes are used to ensure the persistence of data, we assume that the driver software takes measures to issue `flush` commands only after all previous commands are completed.

## 2.3 Crash Consistency

System crashes can occur due to power loss or unrecoverable system errors. Those crashes require a system restart, which usually leads to a reset of volatile memory. Any data on persistent memory is retained. This leads to issues when this data is incomplete, as it can happen with partial writes or interrupted consecutive writes. This is especially relevant for file systems, as they usually store data permanently.

Crash consistency describes the ability for recovery and its outcome after a system crash. In general, file system operations consist of multiple read or write events on the underlying storage. These writes can be reordered for performance reasons, leading to different possible states on the storage after a crash. This on-disk state does not necessarily lead to a different file system state as it is perceived by an application. Journaled file systems, for example, are resistant to incomplete journal writes, because they only enter a new state after a journal entry is committed. For our definition, crash consistency covers both application data loss, as well as file system corruption. [3, 15]

**Mitigations** File system corruption can render the whole file system unusable. It can therefore affect other files than the one leading to the corruption as well. It is therefore necessary to recover the file system to a usable state. This is done with the "file system checker (fsck)" on UNIX systems. This checker needs to traverse the whole storage device to find data and inode regions for reconstructing an usable state. Data blocks not assignable to a file are collected in "lost and found" files. The user then needs to assemble this data manually. [3]

There are multiple mitigation strategies for retaining a sound file system. The most common one is "journaling" or "write-ahead logging". This approach uses a circular log where all write operations are recorded prior to their proper execution. Entries in this log are finalised or "committed" by an atomic write at the end of an entry. Only then then proper data is written to inode and data blocks. In case of a recovery, each journal entry can be replayed if it was committed before the crash. This allows for atomic file system updates. Journaling can comprise metadata and data ("data journaling") or solely metadata for better performance. There are other approaches, such as "log-structured file systems" or "copy-on-write", as well. [3]

**File system guarantees** The POSIX specification defines an API for file system interactions. This API specifies very little guarantees in case of crashes. Notable exceptions are `rename` and `fsync`. The former is required to be atomic while the latter guarantees that all data from the buffer cache is persistently written upon completion. [4] File systems may provide implicit or explicit guarantees beyond the POSIX specification. To allow software developers to make better assumptions about implicit guarantees by specific file systems, Bornholt et al. [4] envisioned more granular crash consistency models. These models are evaluated by litmus tests. As these models are inferred through observation, they are only valid for a specific file system and version. Therefore, portable software can only safely rely on the guarantees given by the POSIX API.

Unfortunately, Rebello et al. [25] show that `fsync` is not as resistant as guaranteed. Pages in the file system cache can be marked clean after a `fsync` failure.

This prevents subsequent `fsync` invocations to persist these pages, since they are perceived to be persisted already. This shows that we cannot rely on API calls when evaluating crash consistency. Instead, we need to evaluate based on commands sent to storage devices.

## 2.4 Virtualisation and Emulation

Testing of interactions between software and hardware can be enhanced by using virtualisation or emulation. Thereby, full systems run isolated on a host system, enabling tracing of specific events. Important frameworks for this cause on Linux are QEMU and PANDA.re.

**QEMU** QEMU has the ability to virtualise user mode software or full systems. Virtualisation is done by using KVM. Additionally, QEMU can utilise dynamic binary translation to automatically translate foreign platform code to native code. Nevertheless, peripheral device are emulated completely by QEMU. These peripheral emulators provide trace points for various events. It is therefore possible to test and trace interactions with devices not present on the host system. [22]

**PANDA.re** An extension of QEMU is PANDA.re. This framework specialises on trace-and-replay testing, leveraging QEMU's dynamic binary translation. In contrast to QEMU, tracing is done by injecting tracing hooks in the binary translation stage. This enables PANDA.re plugins to execute arbitrary code at specific points. These plugins can access guest software data, enabling more comprehensive insights into the running program. [16]

## 2.5 Crash Consistency Testing

Multiple tools exist to aid developers with testing file systems for crash consistency. These tools can be categorized into formal testing and trace-and-replay testing. The former verifies a formal file system model with logical solvers, while the latter analyses a test run and enumerates all possible file system states after command reordering.

**Ferrite** The work by Bornholt et al. [4] provides a framework to reason about possible outcomes of a crash by specifying crash consistency models. Those models consist of formal behaviour specifications and litmus tests to demonstrate and verify this behaviour. Ferrite can validate these formal specifications against the

litmus tests with logical checkers. Additionally, Ferrite leverages QEMU to test existing file systems by generating all possible post-crash states.

**CrashMonkey and ACE** This work by Mohan et al. [13] uses a trace-and-replay approach. A test case is executed on a file system and thereby traced. This trace is then replayed with simulated crashes at different points during execution. The resulting file system state is used for crash consistency evaluation. Therefore, CrashMonkey can test black box systems. Notably, CrashMonkey uses synthetic test cases generated by ACE. To generate these test cases, ACE enumerates permutations of a set of file system operations. To limit the number of test cases, ACE bounds the total number of operations, as well as the set of possible operations. The results indicate that a limited number of two to three file system operations per test case is sufficient to find bugs.

**Vinter** Kalbfleisch et al. [7] adapt the trace-and-replay approach to NVM file systems. Those file systems pose new problems because of their direct addressable storage. This makes the possibility of insufficient cache flushes more likely. The authors therefore expand on the concept of atomicity and propose a new guarantee called “single final state”. Atomicity guarantees that a file system is always in one of two possible states during an operation, essentially prohibiting intermediate states. Single final state, on the other hand, allows these intermediate states, but requires that an operation only has one final state. This does not automatically hold true because of more selective flushes.

Vinter uses PANDA.re to trace unmodified full systems. It then generates possible post-crash file system images, called “crash images”, and extracts their semantic state visible to applications. To reduce the number of crash images, the authors propose a heuristic approach. This heuristic excludes all permutations of write commands which are never read.

# Chapter 3

# Design

In this work, we apply crash consistency testing strategies for NVM to block based storage. We especially extend upon the design of Vinter [8] and Witcher [6]. Using a similar design for NVM and block based storage testing allows for a subsequent integration of hybrid file system testing. Contrary to other work [13], this work's focus lies on the test pipeline and not on generating test cases.

Our primary objective is to test unmodified full systems. Especially, our approach does not rely on source code annotations for test evaluation. Furthermore, our approach does not even require access to the source code, but rather relies on virtualisation and emulation. While we focus on block based file systems, we want to enable an easy integration of NVM storage testing, in order to possibly support hybrid file systems in the future.

Our test system, Revin, supports the evaluation of common file system guarantees: atomicity or single final state behaviour of file system operations in case of crashes. Generally, we want to evaluate possible data loss and file system corruption. The latter might render a file system partially or completely unusable or might prevent writes to this file system, but does not necessarily come with loss of user data. To find evidence for these guarantees and behaviours, we need to simulate all relevant states the file system can adopt after a crash. Since we can not simulate all possible states – which can differ further depending on configuration and hardware – we can only determine breaches of these guarantees but not conformance to them.

## 3.1 Overview

The test pipeline of Revin, as seen in Figure 3.1, consists of four major steps: First, the Analyser, which executes the test case on a virtual system. This step yields a tace of NVMe commands issued and processed, as well as checkpoints

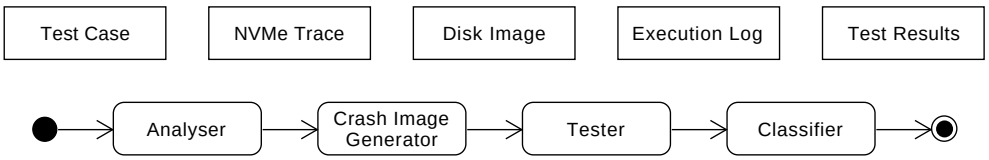


Figure 3.1: Test pipeline design: Based on the analysis of NVMe commands issued during runtime, crash images are generated and tested for possible semantic differences.

reached. Second, the Crash Image Generator creates crash images using subsets of the traced NVMe commands. This step creates a post-success image as well. This image contains the changes from all NVMe commands and is used to check whether we traced all relevant NVMe commands. In the third step, the Tester executes recovery and analysis commands on the crash images. This yields an execution log containing information about the file system recovery, the current file system state and whether the file system is still writeable. The last step classifies the execution logs for all crash images and creates a minimal set of distinct semantic states. These semantic states can then be checked for atomicity or single final state semantics.

Due to the two-step of analysing the uninterrupted program execution and evaluating the crash image state later, the test cases consist of two parts. The first part comprises the test case set-up, such as creating initial files and directories, as well as the actual test commands. The second part is made up by optional commands run after file system recovery and file system dump. Those trailing commands are used to check whether the file system is still writable, but gives the test case the ability to define, how writeability is checked.

## 3.2 Tracer

Tracing is needed in two different places: when analysing the uninterrupted program execution and when testing the crash image. Both uses require the production of NVMe command traces and execution logs as artifacts. Additionally, both uses need the ability to execute arbitrary commands on the system.

There are two possible approaches to these challenges. In the first case, Revin provides a modified NVMe driver for the tested system. This driver can then write to actual hardware or to disk images and trace all NVMe commands. With this scenario, Revin runs inside the tested system and can therefore issue commands by using the standard system calls. Case two uses a virtualisation approach. The tested system is run with a hypervisor. The hypervisor emulates a NVMe de-

vice, which can trace NVMe commands. Because of this emulation, the use of disk images instead of actual hardware is necessary. Since Revin runs outside the virtualised system, commands need to be sent over a virtual console. This can happen with ssh if the virtualised system has network connection, or by connecting with the guest's first serial input device. We decided in favour of the second approach, because then we do not need to modify drivers, which would interfere with the objective of testing unmodified systems.

**Trace** To reconstruct crash images from the trace, this trace needs to contain all relevant information about the NVMe commands issued and completed. NVMe uses command queues. The first queue is filled by the driver and contains command requests. These requests are handled by the NVMe device and might involve a DMA device for copying data to and from the main memory. Finally, the NVMe device sends a completion event to the completion queue to indicate a processed command. Requests and completions do not need to be in the same order for actual devices. [14] Since we use emulated NVMe devices, we decided to trace NVMe commands when their corresponding request is submitted, but ignore them later if no corresponding completion event is recorded. The traces need to include

- `command type`,
- `offset` (reads and writes),
- `length` (reads and writes) and
- `data` (writes only).

The command types are `read`, `write` and `flush`.

Additionally, the trace needs to include checkpoints. Those checkpoints are used to distinguish between different stages in test execution; namely, file system mounting, test set-up, test execution and file system shut down. We issue the checkpoints by writing to a separate serial device. Thus, we can alter the serial device emulator to issue traces for these writes. A sample trace with all three NVMe command traces and checkpoints can be seen in Listing 3.1.

**Communication** To allow testing of arbitrary commands, we need a way to automatically send commands to and receive responses from the virtual system. For this purpose, we emulate a serial device to which the virtualised system can connect the first virtual console (`TTY0`). Commands are then issued by writing through this connection to the running shell. Incoming data is collected as responses. The responses are used in two different ways: First, they need to be logged to the execution log, and second, the tracer needs to wait for special sequences – mostly checkpoints – to determine when the system is ready or finished with test execution.

```
R 0, 4096
R 4096, 4096
R 8192, 4096
R 12288, 4096
R 16384, 4096
C 0
W 9702400, 1024, "74␣65␣73␣74␣61␣70␣70␣65␣6E␣[...]"
W 51393536, 1024, "00␣00␣00␣00␣00␣00␣00␣00␣00␣[...]"
F
W 51394560, 1024, "C0␣3B␣39␣98␣00␣00␣00␣02␣00␣[...]"
F
C 1
```

Listing 3.1: Sample trace containing five reads, three writes in two transactions separated by flushes and two checkpoints. Fields are `command type`, `offset`, `length` and `data` in hexadecimal representation.

## 3.3 Crash Image Generator

The previous Analyser stage produces a trace of NVMe commands and checkpoints. This trace contains all relevant information to reproduce the current storage state on the block device when applied to the same base image in the same order. Read commands can be ignored. We create such a post-success image to verify that all relevant commands were logged. Primarily, we are interested in crash images. These are images, which result from an incomplete execution of write commands. To systematically generate them, we need to apply a subset of the trace to the base image. Unfortunately, the trace contains commands which were used to set up the test case or to shut down the file system. We are not interested in variations of these commands. Therefore, we always apply all commands used for set-up and do not apply any commands for shut down, because the file system will never reach shut down after a crash.

To distinguish between these three parts – the set-up, test run and shut down – we use checkpoints. We automatically insert a checkpoint before mounting, after set-up and before unmounting the file system. These checkpoints are respectively called `m`, `0` and `u`. For crash image generation, we are only interested in variations of NVMe commands issued between checkpoints `0` and `u`. We henceforth refer to this part of the trace as "test trace". All NVMe commands issued between checkpoints `m` and `0` are always applied first in unchanged order. This is called the "set-up trace" from now on. We discard all NVMe commands after checkpoint `u`. NVMe commands can be reordered. We assume that flushes are barriers for

reordering (Section 2.2). Therefore, we split the test trace at flushes and call the resulting parts "transactions".

To generate all possible crash images, we need to permute every transaction and then generate crash images from subsets of these permutations. Since we assume that transactions are sequential, we only need to permute the transaction before the crash. Previous transactions are applied in full. This naïve algorithm generates $\sum_{t \in T} n_t! \cdot n_t$ crash images for transactions $T$ ($n_t$ being the number of commands in transaction $t$). Because of the factorial, the number of crash images increases fast. A transaction with 10 commands would yield more than 36 million crash images. It is therefore important to find algorithms that yield most relevant crash images but reduce the overall number.

**Sequential algorithm** The easiest algorithm applies the test trace in sequential order starting at the beginning. We do this with the sequential algorithm and apply only the first $n$ write commands. Flush commands are ignored. We increase $n$ for each crash image and stop after we applied the whole test trace. Thus, we simulate crashes at any point during execution with the assumption that NVMe commands are not reordered.

**Heuristic algorithm** The sequential algorithm is limited, because of its assumption of sequential order. This might hold true for emulated devices but does not need to be true for real devices. As generating all possible crash images is not feasible, we examine the heuristic proposed by Liu et al. [12] and Kalbfleisch et al. [7].

The heuristic approach uses a post-success image. This image is generated by applying all write commands to a clean base image. Then we run the same post-failure tests on a virtualised system with this image, as we would do later with the crash images. All read commands are traced during these tests. Thus, we get a trace of all regions on the block device which were actually read. The heuristic assumes that only alterations of writes which touch these regions can lead to different semantic states. We could therefore ignore permutations which only affect writes to different regions, reducing the number of commands per transaction as a result.

Relying on a post-success image is a major drawback of this approach. A file system might not read the same blocks after a crash than after a clean shut down. An important example are journals: Journal entries are likely not read if there was no crash. This approach could therefore miss important crash images that lead to a different semantic state.

Revin does not implement this heuristic completely, because tests show only

a minor impact on the overall number of crash images compared to the random algorithm.

**Random algorithm** Nevertheless, we use a nondeterministic approach to test out-of-order command sequences. Transactions are always applied in order. We can only permute inside a transaction. We start with the first transaction and create $k$ random permutations of it. Since this transaction contains $n_t$ commands, we need to generate $k \cdot n_t$ crash images. Then we move to the next transaction. We do the same for all transactions, but always apply the previous transactions with sequential ordering, because all of their commands are guaranteed to be persisted beforehand. With transactions $T$, we generate $\sum_{t \in T} k \cdot n_t$ crash images.

## 3.4 Tester

The crash images represent the internal file system state after a crash. We are interested in the semantic state of the file system as it is visible to applications. To evaluate this semantic state, the Tester mounts every crash image on a fresh virtualised system and runs the file system checker on it. Then it dumps the file system structure and file contents to the execution log. Afterwards, it tests whether the file system is still writeable by executing a write operation given by the test case. To simplify analysis of the execution logs, crash images, which yield the same execution logs with the same contents are grouped together by the Classifier.

The file system dump comprises two separate programs: `fs-dump` walks through the file system tree and dumps the metadata of all entries. Additionally, it dumps the contents of a regular file or the target of a symbolic link. The second program, `dir-dump`, is used to check the correctness of inode entries. Every directory holds an entry to itself and an entry to its parent directory. We want to check whether these entries are still correct after a crash. `dir-dump` therefore walks through each directory and prints the inode ID for each directory entry (especially including the `.` and `..` entries). This ID is the one stored in the directory. A second inode ID is obtained by calling `lstat`. This ID is found by walking through the file system tree from its root. Both inode IDs must be the same in order to be correct. Different inode IDs indicate update problems for these entries.

# Chapter 4

# Implementation

In this chapter, we apply the design outlined in the previous chapter in a prototype implementation. This Revin prototype is able to run test cases on 64-Bit Linux systems with traditional file systems. We chose the 64-Bit architecture to be consistent with the implementation of Vinter [7] and to allow future improvements for hybrid file systems. The Revin prototye is written in Rust. Internally, we use QEMU with minor adjustments to the NVMe and serial emulators for guest system execution and tracing.

## 4.1 Disk Images

The tested file systems need to reside on virtual disks. The contents of these disks is stored in disk image files. QEMU provides the "QEMU Copy on Write Version 2 (QCOW2)" file format for disk images. While this image format is destined for incremental data writes and would result in less space usage, it is a complex format as well. As of now, no Rust libraries for writing QCOW2 images exist. We therefore settled on the simple RAW format. This format stores binary data as it would be stored on disk. On supported file systems, sparse files are used to improve storage usage.

To allow customisation of the initial file system and to avoid the need of having all tools for partitioning in the virtualised system, we use base images. These base images are disk images with an initialised file system. We use these base images for tracing in the analysis stage. We also build the crash images out of these base images. For our implementation, we reserve 100 MB per base image. The base image contains a GPT partition table with one primary partition containing the file system.

Crash images are generated out of a subset of the recorded NVMe command trace. Every write command needs to be replayed on top of a base image. To write

to RAW images, we need an offset and the binary data. The offset in the RAW image corresponds to the offset recorded by the NVMe command trace. Thus, we can easily apply the write commands to a RAW image.

## 4.2 Virtualised System

For this work, we focus on Linux with a x86_64 architecture as a base system. To allow for fast start-up times, we use a stripped-down configuration of the kernel. Nevertheless, we need PCI and NVMe block driver support compiled in. As file systems, we include `ext4` and `vfat` with codepage `437` and `iso8859-1` charset; this requires `native language support` for `United States` as well. We run the kernel with `console=ttyS0,115200n8` and `loglevel=1` options to enable communication via `serial0`.

Since we want the system to remain small, we provide our own `initramfs` with `busybox` and our own `init` script. In this `init`, we mount the `proc`, `dev` and `sysfs`, run every program in `/init.d` and then start `busybox`. Our custom `initramfs` includes the `fs-dump` and `dir-dump` tools, as mentioned in Section 3.4. Those tools are built for the `x86_64-unknown-linux-musl` architecture.

## 4.3 Tracer

We use QEMU to virtualise the system described in Section 4.2. To generate the relevant trace entries for the NVMe commands, we need to hook into the NVMe device emulator. Similarly, we alter the serial emulator to trace checkpoints. The `serial0` interface is used for communication with the Tracer. The complete setup can be seen in Figure 4.1.

**Trace** QEMU allows to attach file-backed storage via the `drive` and `device` directives. We use this approach to attach a base image. The `drive` directive defines the storage backend – in this case a file. Furthermore, the `device` directive creates an emulated NVMe device, which uses the previously defined `drive` as storage backend. [21] The emulated NVMe device is where we hook into QEMU for NVMe command tracing.

QEMU's `trace` directive can be used to track various trace points built into the QEMU source. [24] Since we need more information than the default trace points for the NVMe emulation provide [19], we extend the source code. The required values for our use case are offset and length of reads and writes, as well as the actual written data for writes. Additionally, we need to trace flush commands.

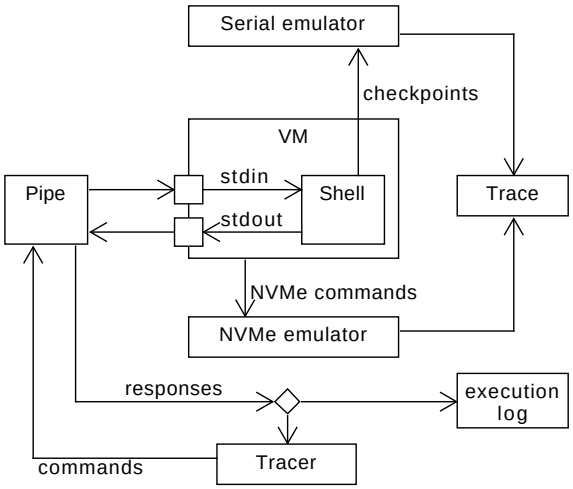


Figure 4.1: Tracing design: The virtualised system generates a trace of NVMe commands and checkpoints. Communication with the tracer is done through a UNIX pipe connected to the standard input and output.

Since reads and writes are passed through the PCI NVMe emulator and the DMA emulator [18,23], we need four trace points. The first trace point traces the NVMe request, the second trace point wraps the NVMe request (or any other DMA request) in a new object. We need this to make sure that this DMA event originated as a NVMe request. The third trace point links the written data, offset and length to the DMA event. We need the NVMe completion event as fourth trace point to ensure that the read or write was successful. The flush command only requires tracing of the flush request and completion.

Secondary, we need to communicate checkpoints to the trace. Checkpoints are used to separate command sequences from each other, e.g. to separate set-up commands from test commands. To make use of QEMU's tracing, we need to use an invocation on an emulated device. In this work, we use the emulated `serial1` device. A trace entry is created each time the virtual system writes to this device. To distinguish between different checkpoints, we trace the value of the first character written.

To make better use of the trace afterwards, we consolidate it into the simpler format described in Section 3.2. The four trace points for NVMe reads and writes are reduced to a single entry consisting of command type, offset, length and data in hexadecimal representation for writes. Checkpoint traces (writes to `serial1`) and NVMe flushes are stripped of their accompanying boilerplate text.

**Communication** To allow testing of arbitrary commands, we need a way to automatically send commands to and receive responses from the virtual system. QEMU allows to connect `serial0` to a UNIX pipe via the `serial` directive. This directive requires two pipes with suffixes `.in` and `.out` for input and output respectively [20]. To use `serial0` as standard input and output, we need to use the `console=ttyS0,115200n8` option when launching the kernel. To make sure that everything is written to standard output, we set `loglevel=1`. Commands are sent to the virtual system by writing to the input pipe.

The responses are consumed in two different ways: First, they need to be logged to the execution log, and second, the tracer needs to wait for special sequences – mostly checkpoints – to determine when the system is ready or finished with execution.

# Chapter 5

# Evaluation

In this chapter, we evaluate our design and implementation. We test our prototype with `vfat` and `ext4` as representatives of a simple and a journaling file system respectively. For `ext4` we test both with and without data journaling (`ext4` uses only metadata journaling by default). First, we test two test cases from CrashMonkey [13] to evaluate whether Revin is able to find the same bugs. Then we use modified versions of the test cases used with Vinter [7]. During these tests, we gather information about the number of crash images generated or expected by the different algorithms. Finally, we evaluate the overall performance of the Revin prototype.

## 5.1 File Systems

**vfat** The `vfat` file system driver is an implementation of the FAT file system for Linux. Contrary to the `msdos` and `umsdos` file system drivers, `vfat` uses the same data structures as the Windows and DOS drivers use and supports long file names. We therefore decided in favour of `vfat` to represent FAT in this work. Due to the size of 100 MB per crash image, we use the FAT16 variant.

As `vfat` does not use journaling for metadata or data, we expect to find intermediate states for crashes. Specifically, we assume no atomic or single final state behaviour. Additionally, we do not expect to find pre-execution states, because there is no recorded previous state the file system checker can roll back to. Some test cases cannot be run on `vfat` as there is no support for UNIX file permissions or soft links.

**ext4** The `ext4` file system was specifically developed for Linux and uses journaling to be able to roll back changes after crashes. By default, journaling is only

```
fd = open("foo")
data = generate(8k)
write(fd, data, size: 8k)
fsync(fd)
fallocate(fd, KEEP_SIZE, offset: 8k, size: 8k)
fdatasync(fd)
close(fd)
```

Listing 5.1: Pseudocode of test case `CM2`

```
fd = open("foo")
lseek(fd, offset: 16k, SEEK_SET)
data = generate(4k)
write(fd, data, size: 4k)
close(fd)

fd = open("foo", O_DIRECT)
blocksize = stat("foo")->blocksize
buffer = generate(4k, align: blocksize)
write(fd, data, 4k)
close(fd)
```

Listing 5.2: Pseudocode of test case `CM4`

used for metadata but can be enabled for data with a mount directive as well. We test `ext4` with both configurations.

Due to journaling, we expect to only find pre-test and post-test states for single file system operations. Intermediate states should not be visible, because any intermediate state would be rolled back with the journal. We therefore assume atomicity guarantees for file system operations that only involve the metadata of one file. Any other operation is expected to have single final state characteristics.

## 5.2 Reproduced Bugs

For their work with CrashMonkey, Mohan et al. [13] reproduced two bugs which affect `ext4`. We call them `CM2` and `CM4` to be consistent with their naming. We chose to reimplement these test cases to evaluate whether Revin is able to find these bugs as well. `CM2` found a bug where a crash after `fallocate` could lead to loss of allocated blocks. The other test case, `CM4` surfaced a metadata inconsistency with the file size when crashing during writes in direct mode. The test cases can be seen in Listing 5.1 and Listing 5.2 as pseudo code. We test both test cases with Linux kernel versions 4.2 and 5.18.

Because of our system architecture (`x86_64-unknown-linux-musl`) and no available code for the test cases used by CrashMonkey (their test cases are generated automatically), we have to resort to reimplementing the tests with Rust. To minimize the impact of modifications automatically done by Rust, we use the foreign function interface for C. This was especially important, because direct writes require a specific data alignment. Nevertheless, we cannot rule out the possibility of minimal differences to the CrashMonkey tests that may cause our tests to not behave in such a way required to trigger the bugs.

In our tests, Revin is able to reproduce the bug for `CM2` mentioned by Mohan et al. [13] with kernel version 4.2 and 5.18. In both tests, an intermediate state with file size 8192 B and block count 16 (compared to the correct file size 8192 B and block count 32) is visible.

On the other hand, we are unable to reproduce the bug caused by `CM4` with either kernel version. This test case uses direct file access, which requires buffers to be aligned in a specific fashion. Our reimplementation might be different to the CrashMonkey tests, causing our tests to elude this bug.

## 5.3 Test Cases

We base our test cases for evaluating `vfat` and `ext4` on the test cases provided by Vinter [7]. These test cases were intended for use with NVM file systems and focus on metadata integrity. We modify the test cases to fit our format and test only one specific operation per test case. Our test cases always include a `fsync` after the tested operation. The following list includes the test cases used.

| | |
|---|---|
| **append** | append text to a preexisting file |
| initial state | non-empty file |
| expectation | text is appended |
| **chmod** | change mode to `0o666` |
| initial state | default mode |
| expectation | mode `0o666` |
| **chown** | change owner to `321:789` |
| initial state | default owner |
| expectation | owner `321:789` |
| **hardlink** | create hard link |
| initial state | `myfile` exists |
| expectation | `hardlink` and `myfile` share the same inode |

| | |
|---|---|
| **hardlink2** | remove hard link |
| initial state | `myfile` and `hardlink` share the same inode |
| expectation | `myfile` is removed; `hardlink` still exists |
| **symlink** | create symbolic link |
| initial state | `myfile` exists |
| expectation | `symlink` points to `myfile` |
| **symlink2** | create dangling symbolic link |
| initial state | `symlink` points to `myfile` |
| expectation | `myfile` is removed;<br>`symlink` is dangling link to `myfile` |
| **mkdir** | create directory |
| initial state | no directory exists |
| expectation | directory `mydir` exists |
| **rmdir** | remove directory |
| initial state | directory `mydir` exists |
| expectation | `mydir` does not exist anymore |
| **rename** | rename file |
| initial state | `myfile` and `myfile2` exist |
| expectation | `myfile2` is renamed to `myfile`;<br>original contents of `myfile` do not exist anymore |
| **move** | move file to other directory |
| initial state | `myfile` and `dir` exist |
| expectation | `myfile` is moved to `dir` and renamed to `myfile2` |
| **unlink** | unlink file |
| initial state | `myfile` exists |
| expectation | `myfile` is removed |
| **update** | update file contents in the middle |
| initial state | non-empty file |
| expectation | file contains new string in the middle |
| **touch** | touch file |
| initial state | `ctime`, `mtime` and `atime` have old time |
| expectation | `mtime` is updated; `ctime` and `atime` keep old time |

| | vfat | | | | ext4 | | | | ext4-dj | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | A | S | W | E | A | S | W | E | A | S | W | E |
| **append** | 1 | ✓ | ✓ | - | ✓ | ✓ | ✓ | - | ✓ | ✓ | ✓ | - |
| **chmod** | – not supported – | | | | ✓ | ✓ | ✓ | - | ✓ | ✓ | ✓ | - |
| **chown** | – not supported – | | | | ✓ | ✓ | ✓ | - | ✓ | ✓ | ✓ | - |
| **hardlink** | – not supported – | | | | T | ✓ | ✓ | - | T | ✓ | ✓ | - |
| **hardlink2** | – not supported – | | | | T | ✓ | ✓ | - | T | ✓ | ✓ | - |
| **symlink** | – not supported – | | | | ✓ | ✓ | ✓ | - | ✓ | ✓ | ✓ | - |
| **symlink2** | – not supported – | | | | ✓ | ✓ | ✓ | - | ✓ | ✓ | ✓ | - |
| **mkdir** | R | ✓ | X | MF | 2 | ✓ | X | MCL | L | ✓ | ✓ | - |
| **rmdir** | R | ✓ | X | MF | ✓ | ✓ | ✓ | - | ✓ | ✓ | ✓ | - |
| **rename** | R | ✓ | ✓ | F | ✓ | ✓ | ✓ | - | ✓ | ✓ | ✓ | - |
| **move** | R | ✓ | ✓ | - | ✓ | ✓ | ✓ | - | ✓ | ✓ | ✓ | - |
| **unlink** | RD | ✓ | ✓ | F | ✓ | ✓ | ✓ | - | ✓ | ✓ | ✓ | - |
| **update** | ✓ | ✓ | ✓ | - | T | ✓ | ✓ | - | ✓ | ✓ | ✓ | - |
| **touch** | ✓ | ✓ | ✓ | - | ✓ | ✓ | ✓ | - | ✓ | ✓ | ✓ | - |

Table 5.2: Crash consistency test results for `vfat`, `ext4` and `ext4` with data journaling (`ext4-dj`). The columns indicate atomicity, single final state behaviour, writeability and file system errors.

- ✓ no violations found
- **1** file size increased but contents not written yet
- **2** `mydir` listed with type "other"
- **R** file is recovered
- **D** data in file lost
- **T** multiple combinations of timestamps
- **X** file system not writeable
- **M** metadata error
- **F** FATs are different
- **C** directory entry corrupt
- **L** wrong number of hard links

These test cases are evaluated with Kernel version 5.18 on the system described in Section 4.2. We check the resulting semantic states for atomicity and single final state semantics and file system corruptions, such as ceased writeability or access errors. The results can be seen in Table 5.2.

**Atomicity** We regard an operation as atomic if the tests yield no more than two semantic states, where each semantic state is either the pre-test or post-test state. We include the possibility of only one semantic state, because `vfat` does not always yield a pre-test state, due to not being journaled. We regard any difference (even minor metadata differences, such as different timestamps in `ext4`) as a violation to atomic behaviour.

**Single final state** Generally, single final state guarantees that an operation may only have one final state as soon as it is completed. Due to the POSIX API having no guarantees prior to a `fsync`, this completeness is only given after a subsequent `fsync`. This means that we check for SFS after applying all operations (in any order) up to a checkpoint following a `fsync`. SFS is always satisfied in our tests.

**Writeability** A file system is deemed as writeable if the write operation following the file system dump does not exit with an error. If this is not guaranteed for every semantic state, we regard the file system as not writeable anymore.

**Errors** During our tests, we encountered errors such as corrupt FATs, wrong hard link counts or metadata errors when dumping the file system. Those errors might not interfere with the perceived file system state but rather concern the file system internals. As such, they always coincide with non-atomic behaviour.

## 5.4 Observations

**Journaling** In our tests, we could observe a different behaviour of `vfat` and `ext4` which can be attributed to journaling. While `ext4` always provided a state corresponding to the state after test set-up, `vfat` did not provide such a state. This is expected, because without a journal, `vfat` cannot roll back changes. Additionally, `vfat` usually committed one to four changes without flushes in between. `Ext4`, on the other hand, always committed two transactions separated by a flush. The final state never occured if the first transaction was incomplete. This leads to the assumption that this first transaction writes to the journal, while the second transaction writes the actual data.

**Data journaling** We tested `ext4` with and without data journaling. Except from the `mkdir` test case, both configurations behaved similarly. Without data journaling, `mkdir` experienced metadata errors as well as corrupt directory entries. This did not happen with data journaling. In Linux, directories are files as well. Therefore, directory entries belong to the file data. It is therefore valid that they are only journaled with data journaling enabled, which explains the observed behaviour.

**Time stamps** UNIX provides three different time stamps per file: access time for the last read or write on the file, change time for the last metadata changes and modify time for the last write operation [1]. `Ext4`, as a Linux file system provides all three time stamps, whereas `vfat` exchanges change time with creation time [9, 11]. In Linux, the `touch` command can be used to set the access and modification time stamps [2]. Our tests initially included a test case for time stamps. The test case `timestamps` creates a new file by writing to it and then uses `touch` to set the access and modification times to January 1st, 2020. We expect the file to have change time set to now and access and modification time to be set to 2020.

Neither file system showed this behaviour when run for analysis. All file systems returned an access time set to now. This might be possible if `readdir` updates the access times of files or if `stat` updates the access times prior to returning them. Additionally, `vfat` had modification and change time set to 2020. This should not happen, as change time (in `vfat`: creation time) is set independently from modification time [10]. Due to these issues, we did not include this test case in the final set.

Another time stamp issue was observed with hard links on `ext4`. Both `hardlink` and `hardlink2` yield multiple semantic states that only differ in their access times. This only happens as long as two hard links to the same inode exist. Except for `atime` both test cases are atomic.

**Directories** Operations involving directories can lead to multiple problems with `vfat`. The most frequent issue are diverging FATs. Internally, `vfat` uses two file allocation tables (FATs) for error correction. If they differ, `vfat` recovers the corresponding blocks. The recovered blocks are available as binary files in the file system root but their original name and place are lost. This might be the cause of the issue arising with `mkdir` and `rmdir`. Both test cases yield metadata errors when printing the directory structure. Additionally, writeability is not retained after these crashes.

Even `ext4` yields problems with `mkdir`. Aside from metadata errors when listing the file system, two other severe errors are possible: First, one crash image produced a state where the newly created directory was listed with file type “other”. Secondly, one crash image contained a corrupt directory entry that caused

`readdir` to return with an error. In both crash images, the file system refused writes. These errors do not occur with data journaling. The reason might be that directory entries are part of the file data and are therefore not covered by metadata journaling.

Nevertheless, `ext4` with and without data journaling exposed another error with `mkdir`: The number of hard links to the file system root was inconsistent. The `mkdir` test case creates a directory `/dir` in the file system root `/`. Before the operation, `/` should have two hard links (from `/.` and `/..` because it is the file system root). Afterwards, `/` should have three hard links, because `/dir/..` links to it as well. The test case yields crash images where `/` has a hard link count of two or four while `/dir` is present. Anyhow, there are no other accessible inodes that link to `/`; therefore, the correct hard link count should be three.

## 5.5 Crash Image Generation Algorithms

In Section 3.3, we discussed the relevance of reducing the number of crash images. Our Crash Image Generator therefore uses uses two different algorithms to generate crash images. The `sequential` algorithm creates crash images with the command ordering intact while the `random` algorithm uses random permutations of the commands. To generate every possible crash image, we would need to use every possible permutation. This is done by the hypothetical `full` algorithm. Because the number of crash images generated by the `full` algorithm increases quickly with the number of NVMe commands, a `heuristic` approach was discussed in Section 3.3. Revin traces the number of transactions and their commands for each test case. With this data, we can establish the number of crash images each of the four algorithms would produce. The results can be seen in Table 5.3.

We found that the `vfat` file system does not use flushes. Therefore, only one transaction exists for every operation. Additionally, `vfat` issues two to three commands on average without any redundancy. Therefore, every written block is read later. This renders the heuristic approach useless for `vfat`. The random algorithm produces around the same number of images, and sometimes even more, as needed for full coverage. In the case of `vfat`, the full algorithm can be reasonably used.

On the other hand, `ext4` usually uses one flush in our tests. This is done after writing journaling data and before writing the actual data. The number of commands issued in total is with 8 - 12 in average more than twice the number of commands issued by `vfat`. Fortunately, not all written data is read later. It is therefore possible to decrease the number of commands by around one or two

| | vfat | | | | ext4 | | | |
|---|---|---|---|---|---|---|---|---|
| | seq | rand | heur | full | seq | rand | heur | full |
| **append** | 2 | 10 | 4 | 4 | 5 | 25 | 22 | 22 |
| **chmod** | – not supported – | | | | 6 | 30 | 22 | 36 |
| **chown** | – not supported – | | | | 4 | 20 | 8 | 8 |
| **hardlink** | – not supported – | | | | 8 | 40 | 36 | 192 |
| **hardlink2** | – not supported – | | | | 8 | 40 | 36 | 192 |
| **symlink** | – not supported – | | | | 12 | 60 | 604 | 8640 |
| **symlink2** | – not supported – | | | | 16 | 80 | 35284 | 645120 |
| **mkdir** | 4 | 20 | 96 | 96 | 17 | 85 | 3588480 | 3588480 |
| **rmdir** | 3 | 15 | 18 | 18 | 17 | 85 | 35284 | 3588480 |
| **rename** | 3 | 15 | 18 | 18 | 16 | 80 | 35298 | 645120 |
| **move** | 2 | 10 | 4 | 4 | 10 | 50 | 97 | 1200 |
| **unlink** | 3 | 15 | 18 | 18 | 16 | 80 | 35284 | 645120 |
| **update** | 2 | 10 | 4 | 4 | 5 | 25 | 22 | 22 |
| **touch** | 1 | 5 | 1 | 1 | 4 | 20 | 8 | 8 |

Table 5.3: Number of crash images generated by the respective algorithm. The heuristic can decrease the number of crash images by around a half for `ext4` compared to the full algorithm. The random algorithm was configured to generate five permutations per transaction.
`seq`: sequential algorithm
`rand`: random algorithm
`heur`: heuristic algorithm

by using the proposed heuristic. This can decrease the number of crash images for tests with many commands. Nevertheless, the impact of the heuristic is minor compared to its impact for NVM file systems [7, Section 5.2]. The random algorithm produces a very limited number of possible crash images. Furthermore, for less than six commands, the random algorithm produces more images than needed. These images possibly contain duplicates.

As we can see, the random algorithm is problematic for both smaller traces as well as for larger ones. For smaller traces, more crash images are generated than necessary. But those crash images possibly contain duplicates. In this case, the full algorithm would be suited best, because it generates less crash images with full coverage. For larger traces, the heuristic approach is feasible but not as promising as expected.

## 5.6 Performance

We discussed how to reduce the number of crash images in the previous section. Now we take a look at the performance of a Revin workload. First, we compare the Tracer run-time with different test cases and file systems. Then, we focus on the `rename` test case and evaluate the component run-times in detail.

Tracing is done for both analysis and test stage. We focus on the Analysis stage for this evaluation, but the results can be applied for the Test stage as well. Table 5.4 shows the Analysis run-time for each test case described in Section 5.3. As we can see, all test cases are run in approximately 5s or 7s, regardless of file system. The test cases which need 7s contain a delay of 2s. As a result, all test cases need approximately the same time on any file system. We therefore assume that the run-time difference between test cases is insignificant.

Based on this observation, we can focus on a specific test case for a more detailed run-time evaluation. We run the whole test pipeline ten times with the `rename` test case on `ext4` and Linux kernel version 5.18 and trace the run-time of the individual stages, as well as the run-time between checkpoints in the Analyser. The results can be seen in Table 5.5. As expected by the previous evaluation, the Analyser (which uses the Tracer to analyse the test case behaviour) takes around 5s per analysis. Both algorithms for the Crash Image Generator take less than a second to generate all crash images. Most of the time is needed by the Tester with around 8 minutes of run-time. This equals to around 5s per crash image as well.

The total run-time is dominated by the Tester, and therefore by the number of crash images. The time needed to generate the crash images is insignificant with the two algorithms used. Overall performance would therefore benefit from an algorithm that reduces the number of crash images significantly. This algorithm

| | Run-time in s | | |
|---|---|---|---|
| | vfat | ext4 | ext4-dj |
| **append** | 7.030 | 7.037 | 7.031 |
| **chmod** | | 7.035 | 7.034 |
| **chown** | | 7.035 | 7.033 |
| **hardlink** | | 5.034 | 5.034 |
| **hardlink2** | | 5.034 | 5.034 |
| **symlink** | | 7.035 | 7.033 |
| **symlink2** | | 5.035 | 5.034 |
| **mkdir** | 5.034 | 5.032 | 5.034 |
| **rmdir** | 5.035 | 5.034 | 5.033 |
| **rename** | 5.035 | 5.035 | 5.034 |
| **move** | 5.033 | 5.035 | 5.033 |
| **unlink** | 5.036 | 5.034 | 5.034 |
| **update** | 5.034 | 5.034 | 5.034 |
| **touch** | 7.035 | 7.034 | 7.034 |

Table 5.4: Analysis stage run-time for different test cases. Run-time does not depend on file system. Test cases with approx. 7s run-time contain a 2s delay.

| Component | per image run-time in ms | per image $\sigma$ in ms | total run-time in s | total $\sigma$ in s |
|---|---|---|---|---|
| Analyser | 5034 | 0.635 | | |
| booting | 1099 | 256 | | |
| mounting | 177 | 49 | | |
| test run | 849 | 51 | | |
| unmounting | 1030 | 10 | | |
| shutdown | 1894 | 227 | | |
| process collection | 11 | 17 | | |
| Crash Image Generator (sequential) | 4.873 | 0.085 | 0.078 | 0.001 |
| Crash Image Generator (random) | 7.116 | 0.317 | 0.569 | 0.025 |
| Tester | 5035 | 0.237 | 483.344 | 0.022 |

Table 5.5: Run-time of individual stages. Per image and total run-time are the same for the Analyser.

can be more complex and time-consuming than the algorithms currently used.

We break down the Anayser further: an analysis sequence consists of booting, file system mounting, running the test case, unmounting, shut down and collecting the QEMU process. The most time-consuming part is the system shut down, whereas file system mounting and process collection are negligible. We could save 1s by using system snapshots after the boot process. The shut down process cannot be reduced further, as we already use a kill signal to stop the QEMU process.

# Chapter 6

# Discussion

The previous chapters focused on the design, implementation and evaluation of Revin. This chapter presents difficulties and possible solutions concerning the design. First, we discuss the reliability and completeness of the recorded trace. Then, we focus on issues regarding our assumptions of flushes acting as reordering barriers. The next section deals with the significance of single final state semantics for block based file systems. Finally, we discuss the possibility of integrating Revin with Vinter to accommodate for hybrid file system testing.

## 6.1 Tracing and Crash Images

**Tracer** The Tracer is a central part of Revin's design, because all other components rely on the correctness and completeness of the recorded trace. Therefore, care must be taken to record any relevant information. Revin records only `read`, `write` and `flush` commands. Another possibly important command, `write zeroes`, is not traced. NVMe management commands are also not part of the trace. Accordingly, Revin does not distinguish between IO command queues as well. Multiple such queues can introduce new possibilities for concurrent NVMe commands [14].

To verify the integrity of the trace, Revin generates a success image out of the traced data and compares this image to the image generated by the Analysis stage. Equality of these images indicates that any relevant data was traced.

The traced information was sufficient for our test cases. We also did not find any use of `write zeroes` in the file systems included in the Linux kernel. Nevertheless, third-party file systems or future versions might use currently untraced NVMe commands. It is then necessary to integrate tracing for these commands to maintain trace integrity.

**Flushes** The most critical issue is the simplification made by our assumption that flushes act as reordering barriers (Section 2.2). As per design, a `flush` command leads to the NVMe device persisting the contents in the write caches to non-volatile memory. The command guarantees that all changes are persisted if these commands were completed before the `flush` was issued [14, Section 6.8]. This means that a `write` issued before a `flush` is not guaranteed to be persisted if it completes after the `flush` command was issued. Therefore, flushes are no reordering barriers in reality. Lifting this assumption would require us to reason about these special cases. Simply removing barrier semantics and putting all commands into the same transaction might lead to false positives. The number of crash images would increase significantly as well. Tracing both submission and completion events would not suffice either. This would yield one possible execution order, from which we cannot infer any guaranteed ordering.

On the other hand, this extended tracing can be used to determine whether such a reordering over flushes happens. While the NVMe standard only gives the aforementioned guarantees, lower operating system layers (such as the block layer) or the file systems themselves might enforce flushes as reordering barriers. This is possible by delaying flushes until all previous writes have completed. Tracing both submissions and completions, as well as delaying writes can give insights whether such a mechanism is in place. Unfortunately, this is not possible with the current NVMe emulator for QEMU, because it executes commands in order. An asynchronous reimplementation of the emulator is necessary for these tests.

## 6.2 Evaluation

**Single final state** Unsurprisingly, our tests show no violation to single final state (SFS) semantics. Any SFS operation must lead to a single semantic state after its return. This is always true for our tests, because we end every operation with a `fsync`. With block based file systems, this `fsync` persists all data in the file system cache with regard to the given file. On the other hand, NVM file systems have different semantics due to the use of cache lines. The `clflush` operation used by NVM file systems writes back the whole cache line corresponding to a memory address [5], whereas NVMe flushes write back all blocks present in the write cache [14]. The `clflush` semantics offer more possibilities for unpersisted data. Therefore, SFS semantics are more important for NVM file systems than for block based file systems.

## 6.3 Extendability

**Vinter** Revin is based on Vinter [7]. While Vinter focuses on NVM file systems, our work is focused on block based file systems with regard to NVMe as underlying technology. Both approaches share the same general design with Analyser, Crash Image Generator and Tester. The major difference is found with the Tracer. Vinter is implemented by leveraging hooks in PANDA.re. PANDA.re is a framework based on QEMU that allows to invoke code when a guest program reaches a specific state. This is done by using QEMUs dynamic binary translation to insert hooks. [16] Within these hooks, Vinter can trace NVM access and runtime metadata which can then be used to find the cause of bugs.

Revin, on the other hand, does not use dynamic binary translation. We use the tracing capabilities within QEMU by including additional trace points in QEMUs NVMe emulator. Therefore, we can only inspect the trace afterwards, whereas Vinter can run code during test execution. Thus, Revin cannot trace runtime metadata as easily. This design choice was made because PANDA.re uses an old version of QEMU which does not support NVMe yet.

**Hybrid file systems** It would be beneficial to combine Revin and Vinter in order to provide a common testing interface for both block based and NVM file systems. This would enable testing hybrid file systems as well. Unfortunately, this is not easily possible as of now, due to the different tracing methods.

There are two possible solutions to this problem: Either Vinter is ported to QEMU tracing by leveraging the hardware emulators, or Revin is ported to PANDA.re hooks. The former solution might be possible by including traces in the `nvdimm` emulator [17]. Unfortunately, runtime metadata tracing would become complicated with this approach because this data is not easily available without dynamic binary translation. The second approach, on the other hand, would require porting PANDA.re to a newer version of QEMU or porting the NVMe emulator to the older QEMU version.

Crash consistency testing for hybrid file systems is only possible by combining NVMe tracing and NVM tracing. This is necessary to capture the interaction with both storage backends in the same trace. Testing the block based parts and NVM parts separately would not find bugs caused by the interaction with both backends or could lead to false positives because data is persisted in the untraced storage backend.

# Chapter 7

# Conclusion

Crash consistency testing is important to improve the reliability of existing file systems and to aid the development of future file systems. There are different works in this field, that focus on varying aspects, such as formal verification or trace-and-replay assessments, as well as different kinds of file systems.

Our work applies techniques established by tests on NVM file systems to block based file systems. We use virtualisation and emulation with QEMU to create a trace-and-replay pipeline: We trace the storage device usage during the execution of test cases and apply the variations of the resulting trace to a base image, thereby creating crash images. The resulting set of crash images is examined for their semantic file system state. We use this process to generate a set of distinctive semantic states possible after a crash.

The total number of possible crash images increases quickly with each additional NVMe command. Anyhow, not all of those crash images yield a new semantic state. Therefore, we examined different algorithms that limit the total number of crash images but retain relevant crash images. None of the examined algorithms is suitable for every test case.

We further evaluated our test pipeline and could reproduce one known bug, whereas we could not reproduce another one. We used `vfat` and `ext4` to test simple and journaled file systems. Our results show that journaling lead to more atomic operations. Nevertheless, we found a significant bug with `mkdir` when using `ext4` that leads to corrupt directory entries.

Finally, we discussed the possibility of integrating Revin with Vinter to provide a unified tool for crash consistency testing of NVM and block based file systems. This would benefit hybrid file systems as well. We established that changes to the tracing system are necessary to realise this integration.

## 7.1 Future Work

**NVMe flushes** An important aspect of our design is the assumption that NVMe flushes are reordering barriers. In Section 6.1, we examined the impact of lifting this assumption. This led to the question how NVMe flushes are handled by layers below the file system layer. This behaviour could be tested by tracing both submission and completion events, and by delaying writes. Such a test could give insights whether the operating system holds back flushes until all writes were processed. Unfortunately, this test is not possible with the current NVMe emulator in QEMU, because it processes commands sequentially. The test though, needs flushes to overtake writes.

**Crash Image Generation algorithms** Another unsolved problem is the algorithm used to reduce the total number of crash images, while retaining relevant ones. Our evaluation shows that the heuristic algorithm proposed by Kalbfleisch et al. does not work well with block based file systems. Our random algorithm is problematic as well, because it generates too many crash images for short traces, but generates not enough crash images for larger traces. There are possible duplicates among these crash images as well.

A possible solution could comprise different algorithms for differently sized test cases. An algorithm creating all crash images may be well-suited for small traces, whereas a heuristic approach might be used for traces with many redundant writes. Further work is necessary to envision, evaluate and classify algorithms for this purpose.

**Automatic Test Generation** The focus of our work was the pipeline for crash consistency tests. We provided 14 handwritten tests for the evaluation; all of them consisting of only one file system operation. It is necessary to test the interaction of different file system operations to comprehensively test file systems. These tests can be crafted manually, but Mohan et al. [13] presented a better solution: a tool that automatically generates test cases. Their tool ACE generates test cases in an intermediate text representation. This representation is then used to generate C programs. We could use this text representation as well to generate Rust test cases for Revin. Essentially, an adapter between ACE and Revin would make it possible to use these automatically generated tests cases.

**Hybrid file systems** We discussed the possibility of integrating Revin with Vinter in Section 6.3. Further work is necessary to turn both tools into a unified test solution that would enable testing of hybrid file systems as well.

# Bibliography


[1] *stat (2) - Linux man page*.

[2] *touch (1) - Linux man page*.

[3] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 1.00 edition, August 2018.

[4] James Bornholt, Antoine Kaufmann, Jialin Li, Arvid Krishnamurthy, Emina Torlak, and Xi Wang. Specifying and checking file system crash-consistency models. *ASPLOS '16: Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 83 – 98, March 2016.

[5] c9x and Intel. Clflush - x86 instruction set reference. `https://c9x.me/x86/html/file_module_x86_id_30.html`.

[6] Xinwei Fu, Wook-Hee Kim, Ajay Paddayuru Shreepathi, Mohannad Ismail, Sunny Wadkar, Dongyoon Lee, and Changwoo Min. Witcher: Systematic crash consistency testing for non-volatile memory key-value stores. In *SOSP '21: Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 100–115. ACM Press, October 2021.

[7] Samuel Kalbfleisch. Automatic non-volatile memory crash consistency testing for full systems. Master's thesis, KIT Department of Informatics, 2021.

[8] Samuel Kalbfleisch, Lukas Werling, and Frank Bellosa. Vinter: Automatic Non-Volatile memory crash consistency testing for full systems. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 933–950, Carlsbad, CA, July 2022. USENIX Association.

[9] Linux. fs/fat/inode.c. `https://github.com/torvalds/linux/blob/master/fs/fat/inode.c`.

[10] Linux. fs/fat/misc.c. `https://github.com/torvalds/linux/blob/master/fs/fat/misc.c`.

[11] Linux. include/uapi/linux/msdos_fs.h. https://github.com/torvalds/linux/blob/master/include/uapi/linux/msdos_fs.h.

[12] Sihang Liu, Korakit Seemakhupt, Yizhou Wei, Thomas Wenisch, Aasheesh Kolli, and Samira Khan. Cross-failure bug detection in persistent memory programs. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1187–1202, 2020.

[13] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. Crashmonkey and ace: Systematically testing file-system crash consistency. *ACM Transactions on Storage (TOS)*, 15(2):1–34, 2019.

[14] Inc. NVM Express. Nvm express (tm) base specification. Specification, NVM Exprerss, Inc., March 2021.

[15] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Crash Consistency. *Communications of the ACM*, 58(10):46–51, October 2015.

[16] PANDA.re Project. Panda.re. https://panda.re.

[17] QEMU Project. hw/mem/nvdimm.c. https://gitlab.com/qemu-project/qemu/-/blob/master/hw/mem/nvdimm.c.

[18] QEMU Project. hw/nvme/ctrl.c. https://gitlab.com/qemu-project/qemu/-/blob/master/hw/nvme/ctrl.c.

[19] QEMU Project. hw/nvme/trace-events. https://gitlab.com/qemu-project/qemu/-/blob/master/hw/nvme/trace-events.

[20] QEMU Project. Invocation - qemu documentation. https://qemu-project.gitlab.io/qemu/system/invocation.html.

[21] QEMU Project. Nvme emulation - qemu documentation. https://qemu-project.gitlab.io/qemu/system/devices/nvme.html.

[22] QEMU Project. Qemu. https://wiki.qemu.org/Main_Page.

[23] QEMU Project. softmmu/dma-helpers.c. https://gitlab.com/qemu-project/qemu/-/blob/master/softmmu/dma-helpers.c.

[24] QEMU Project. Tracing - qemu documentation. https://qemu-project.gitlab.io/qemu/devel/tracing.html.

[25] Anthony Rebello, Yuvraj Patel, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Can applications recover from fsync failures? *ACM Trans. Storage*, 17(2), June 2021. https://doi.org/10.1145/3450338.