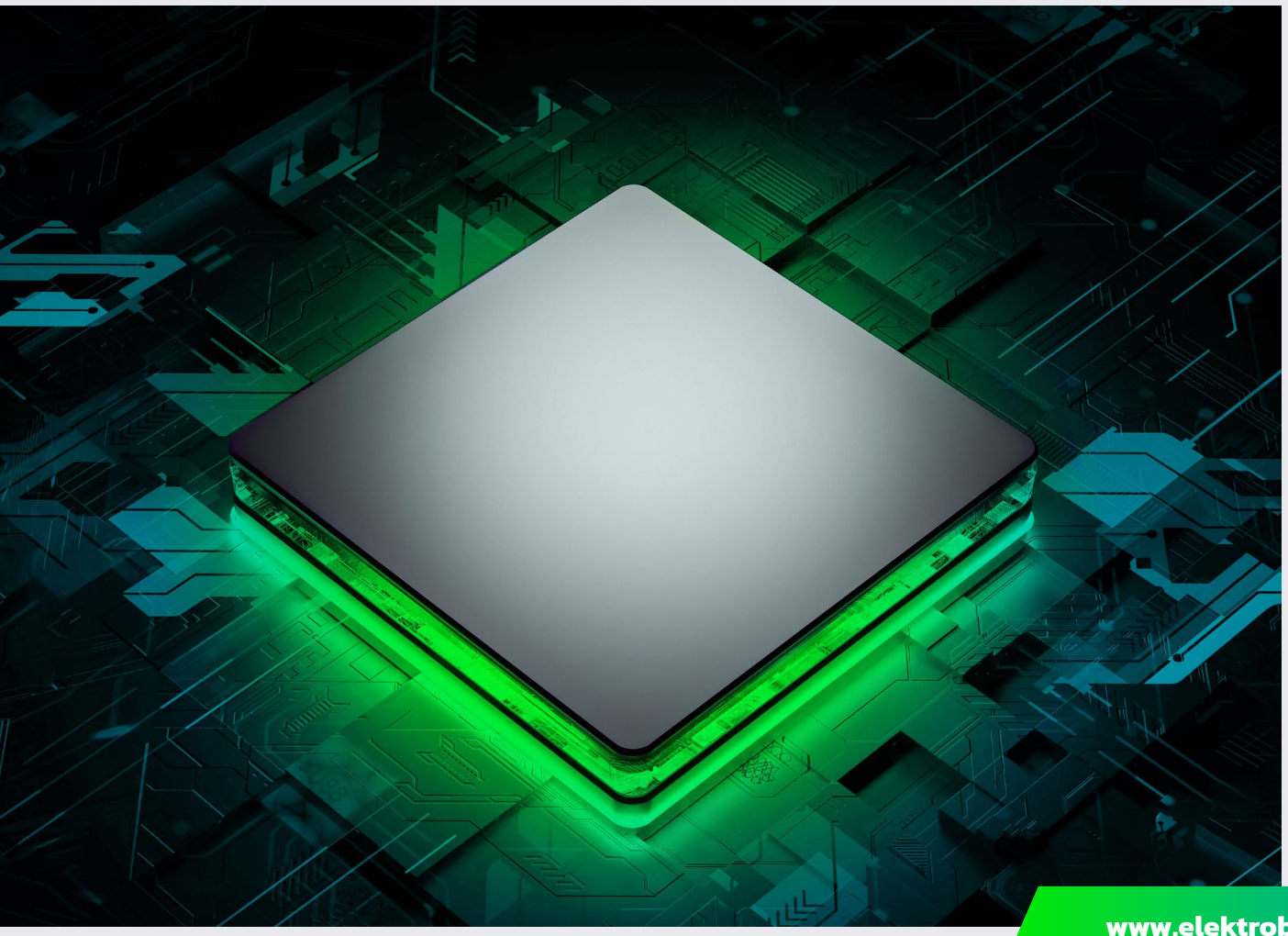
emlix
embedded linux systems

Elektrobit

Tech paper

Deterministic black box testing of file system power-fail reliability for automotive Linux



Summary of Tech paper contents

1. The problem of file system power-fail reliability

"Everything is a file", is one of the most often cited rules of thumb in the Linux ecosystem. Among other aspects, this illustrates how important files and thus reliability of file systems are for a Linux operating system.

The easiest way to improve reliability is to configure the file system as read-only. This prohibits any modification of the content inside. Hence the file system cannot be harmed by failures during write operations. Of course, this is not always an option. Configuration data, log files, temporary information, and others need a file system that is reliable and writable.

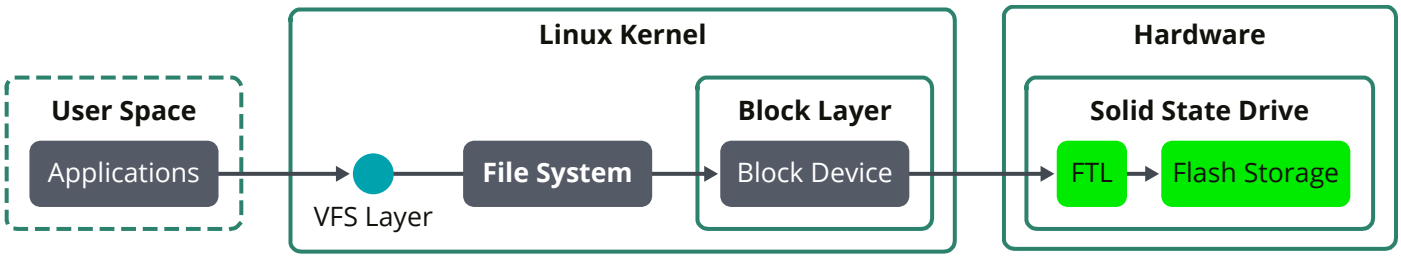


Figure 1: The file system in the Linux kernel

Given the unique requirements of automotive applications, it is clear that reliability during power-fail conditions is of crucial importance. Automotive embedded systems are both at high risk of being subjected to unexpected power loss and causing a critical loss of function if such an event is not handled robustly. A car needing to be towed to the shop because of a start attempt with a weak battery is a scenario that obviously should never happen.

1.1 Background – What exactly is a "file system"?

In any Linux setup, embedded or desktop, the file system layer is the beating heart of both persistent and volatile storage. As illustrated in figure 1, when an application wants to interact with data on the system, it can do so via the virtual file system or VFS.

The VFS layer abstracts away differences in implementation of the underlying file system implementation. The upward presentation is essentially what we regularly see while we interact with files, e.g. "/home/user/myfile.txt".

Below the VFS sits the actual file system implementation. Its job is to keep track of where and how the data on the medium below is organized. One may think of it as a kind of librarian with their very own organization scheme to store and retrieve annotated data. Well-known and commonly used examples of such file system implementations are ext4, btrfs, XFS, F2FS, and ZFS. Still, a multitude of others exist.

The layer below is called the block layer. It is an abstraction of the actual storage hardware. Block devices are representations of a storage device or a partition contained on it. The representation always looks like a large contiguous area of random-accessible raw data, organized in blocks to typically 512 bytes or 4 kilobytes each. Blocks are accessed by block numbers counting from 0 to multi-millions, depending on the overall device size.

Different hardware drivers implement this abstraction. For flash-based storage with a hardware flash translation layer (FTL, always found in eMMC, SSDs, and NVMe storage devices), such a driver may be rather lean. The FTL already does most of the abstraction work. For other forms of storage, the driver may be more complex.

Now, as the sheer amount of different file system implementations suggests, they differ in performance metrics such as speed, reliability, and resource usage. This means some may be more suited to a particular use case than others—which leaves the problem of choosing the right one. Generally, things such as speed and resource usage can be rather easily tested, but reliability in the face of real-world problems such as power failure is another matter. The test setup outlined here can help solve that issue.

1.2 Possible failure modes

Susceptibility to damage from unexpected power loss is a well-known problem for writable file systems. Generally, a change to a file results in the file system issuing one or more write commands to the underlying storage medium (through the block layer). The operation as a whole is only completed when all commands have finished.

If a power loss occurs at the wrong time, the data on disk may have only been partially written. Depending on characteristics of the medium, even single operations may be interrupted and result in an inconsistent state.

Any such error can manifest itself in different ways on the file system layer. Depending on purpose and location of the interrupted write operation, we may observe either data (content of files) or metadata corruption (file names, file sizes, directories, etc.). The latter is generally regarded as the more serious error. If metadata gets corrupted, the result can be anything ranging from errors in file names up to an unusable file system as a whole. Although, as the size of metadata is comparatively small, it is also easier to mitigate damage, for example, via redundancy. Data corruption is usually costlier to prevent, but the file system as a whole remains usable, even if a single file somewhere contains false data.

1.3 Mitigation tactics used by modern file systems

As mentioned, methods exist to mitigate or prevent the kinds of damage described above. Three of the more common approaches are journaling, copy-on-write (CoW), and a log-structured file system.

All of them rely on some form of data redundancy. Journaling essentially writes everything into a placeholder area first, and then copies it over to the final location. That way there is always an intact copy of the prior data or the newly written data after the interruption, and a consistent state can be recovered.

Copy-on-write is a similar idea where new data is always written to an unused location and “overwritten” data is just marked as such. As an improvement to journaling, this saves the write-back action. The downside is fragmentation of often-modified files.

The log-structured approach goes a step further by committing everything to what is essentially a circular buffer. This is meant to improve redundancy and write performance at the cost of necessary periodic garbage collection.

Other hardware-based tactics such as backup batteries are not taken into account.

1.4 Use-case dependent severity

The failure modes mentioned in 1.2 may be more or less critical depending on the use case.

If the damaged file system is the so-called root file system, the ECU cannot be operated any longer and would need a replacement or at the very least a hands-on intervention by qualified personnel.

In case the file system was used for secondary data logging or less important configurations, an automatic repair strategy by reformatting the file system could be imagined. In the worst case, this would result in a system factory reset. Whether that is an acceptable occurrence needs to be decided during the system design phase.

As an example, we can think of a system regulating the cabin A/C and entertainment functions versus a system directly involved with vehicle driver assistance. We can categorize a short-time loss of function of the former – possibly with the loss of some personal settings – as an annoyance but not a danger to the vehicle driver. The same is not true for the latter.

Other factors to consider are present hardware redundancies or automatic back-ups. Also there can be trade-offs between reliability, speed, and flash wear depending on file system configuration.

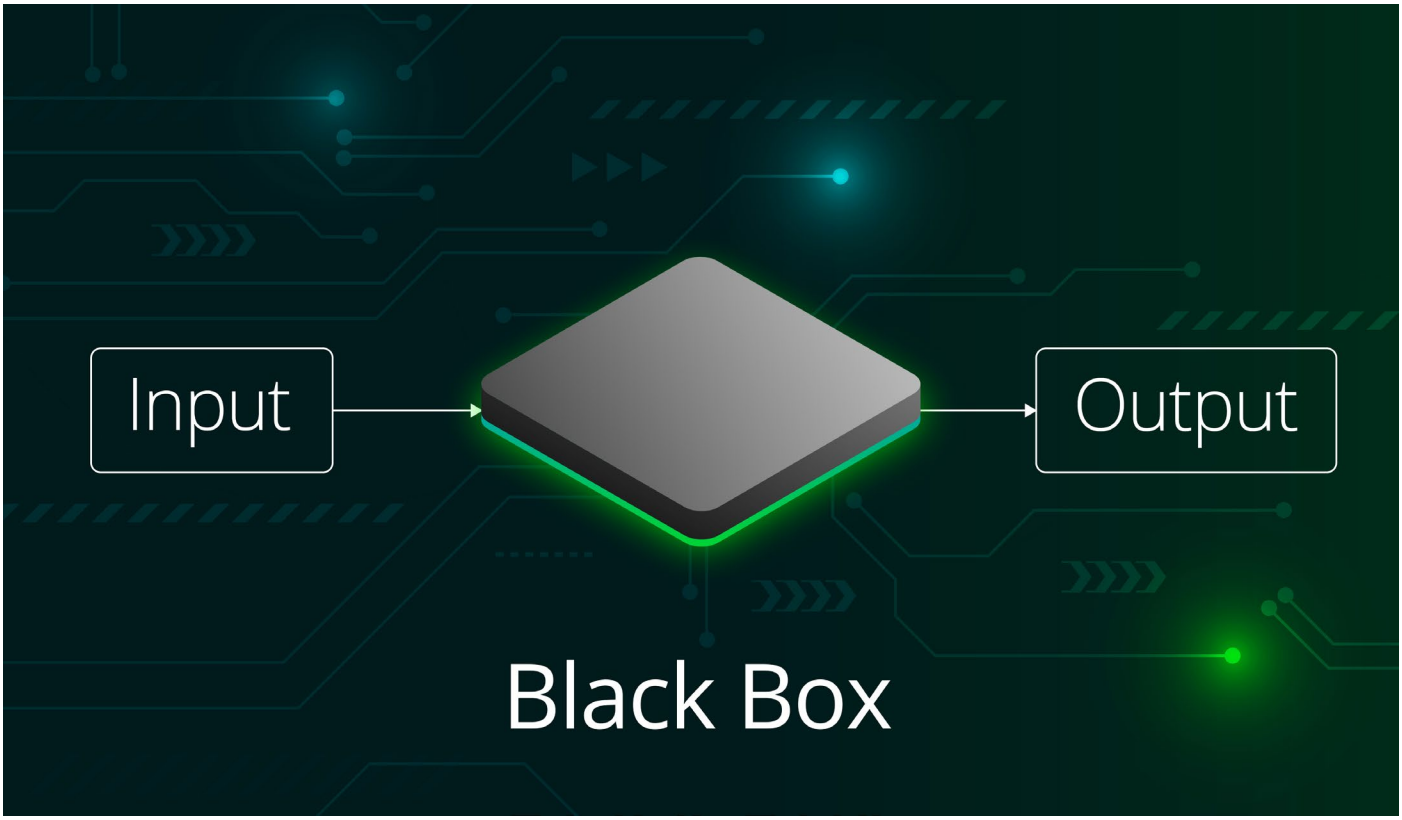
2. Systematic testing of file system power-fail immunity

In order to gather hard and quantifiable evidence on a file system's power-fail reliability, we implement a deterministic and file-system-agnostic test setup. From the perspective of our test, the file system is a black box. Apart from that, we have a full view of the lower block and upper VFS layers.

2.1 Idea

As outlined above, any type of power-fail file system corruption is the result of a hardware-level write operation which was interrupted due to the event. We also know from flash hardware specifications that larger write operations consist of smaller increments, which are written atomically, meaning they are either written completely or not at all [1]–[3]. So given a correct implementation of atomicity on the hardware side, there is a finite number of states at which a write operation can be interrupted.

This means we can build a test which simulates a power failure at every possible moment within a set of write operations. At each of these interruptions the file systems under test are given a chance to recover. Then we check if any of the failure modes have occurred.



2.2 Methodology

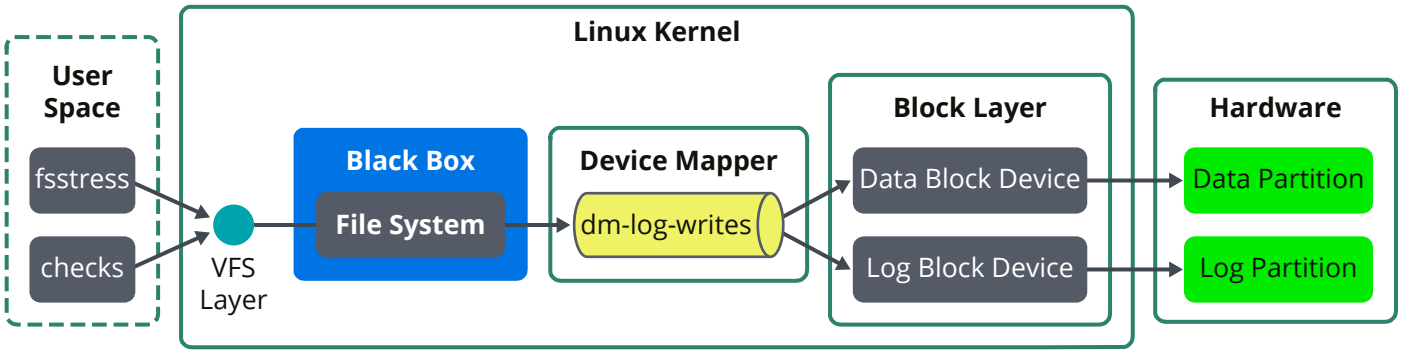


Figure 2: The dm-log-writes test setup

The Linux device mapper offers the dm-log-writes software driver to perform these kinds of tests. It wedges itself between the file system and the block device layer and will record every write that goes through it. The resulting binary write log is written to large persistent storage so it can later be replayed.

To generate input for the log, we use the well-known fsstress program from the Linux Test Project (ltp) [4]. Given a random seed, fsstress generates a set of operations on the file system via the VFS layer according to a pre-set probability distribution. This set of operations then causes the black-boxed file system under test to issue write operations to the block layer which are intercepted by dm-log-writes, logged, and relayed to the storage block devices. The path of the logged write operations is illustrated in the figure above.

A replay of a dm-log-writes binary log has the ability to reach all possible states the data block device has gone through during logging. We have created a replay implementation that is able to replay log entries partially down to a granularity of a single written block which is guaranteed to be atomic on most flash-based hardware. The resulting snapshots of block devices represent all possible states that a storage might have after a sudden power loss.

Now we can deterministically go through all possible power-fail states. At each step, we check if the file system is still usable, consistent, and free of data or metadata corruption by thoroughly inspecting the VFS layer. Also we perform further checks and additional write operations to search for “hidden” corruption which is not immediately apparent through the usual static file system checking tools.

2.3 Completeness

Our argument of completeness for the above methodology rests on the promise of simulating all possible states a power failure may lead to during a series of file system operations. Hardware documentation of relevant automotive-grade storage media indicate that at least single-block writes can be assumed to be atomic [3], [5]. It follows that if we test all possible atomic steps, i.e. written block by written block, our test is complete with respect to reachable states of the on-disk data.

Of course, that is only one half of the problem. We also need to present different sets of operations as the input data into the VFS layer. Here we make a statistical argument that using different seeds on fsstress which guarantee all possible operations to have been run in different combinations is sufficient.

3. Value for the customer

Elektrobit uses this testing method for our relevant Linux-based products to improve reliability. The results directly inform our file system-related design decisions during development. Issues arising for particular use cases can be identified early and resolved quickly.

Of course, testing does not stop with a product release but keeps going as part of the maintenance process to avoid regressions. Specifically looking at Linux kernel upgrades, this yields an added layer of protection if file system bugs are introduced in new upstream code.

Apart from that, Elektrobit provides tailored solutions for special use cases on request. They are able to deploy specialized test setups including result evaluation for different sets of file systems running on different Linux setups on different hardware, virtualized or bare-metal.

4. emlix

emlix offers industrial-grade Linux for the digitalization and secure networking of devices, machines, and plant throughout the entire product life cycle.

For more than 20 years, emlix has been transferring system knowledge, innovations from the open-source world and market knowledge into the products of our more than 350 customers in automotive, energy industry, automation technology, medical technology, safety technology, and others.

As a provider of professional open-source software, we ensure process security and transparency. The tools and development standards we use are designed for industrial requirements and certifications. We offer long-term maintenance contracts for our solutions and thus assume responsibility for the product life cycle and the investments of our customers.

References

[1] NVM Express Inc., “NVM Command Set Specification 1.0b.” Dec. 18, 2021. [Online]. Available: https://nvmexpress.org/developers/nvme-specification/

[2] JEDEC, “JESD84-B51 - Embedded Multi-Media Card (e•MMC) Electrical Standard (5.1).” Jul. 2014.

[3] Micron Technology Inc., “TN-FC-27: e·MMC Power Loss Data Integrity.” Dec. 2013.

[4] “Linux Test Project - fsstress.” Silicon Graphics Inc. [Online]. Available: https://github.com/linux-test-project/ltp/tree/master/testcases/kernel/fs/fsstress

[5] Micron Technology Inc., “e.MMC Memory - MTFC8GAM, MTFC16GAP, MTFC32GAP, MTFC64GAP, MTFC128GAP.” Oct. 2018.

About the authors

Andreas Zdziarstek
emlix GmbH

Andreas Zdziarstek is a system engineer at emlix GmbH. He works primarily on embedded Linux software, developing bespoke solutions for a range of use cases in the automotive area, focusing on safety, reliability, and availability.

Thomas Brinker
emlix GmbH

Thomas Brinker is a Senior Systems Engineer and Project Manager at emlix GmbH. He is an architect for secured embedded Linux systems in the automotive, medical, industrial, and consumer device fields, performing requirements engineering and design throughout the entire product life cycle.