# Dealing with Driver Failures in the Storage Stack

Jorrit N. Herder[†], David C. van Moolenbroek[†], Raja Appuswamy[†], Bingzheng Wu[§], Ben Gras[†], and Andrew S. Tanenbaum[†]

† Dept. Computer Science, Vrije Universiteit
De Boelelaan 1081, 1081 HV Amsterdam, The Netherlands
{jnherder,dcvmoole,raja,beng,ast}@cs.vu.nl

§ Institute of Software, Chinese Academy of Sciences
ZhongGuanCun NanSiJie 4th, Dept. HaiDian, Beijing, China
wubingzheng@gmail.com

## Abstract

*This work augments MINIX 3's failure-resilience mechanisms with novel disk-driver recovery strategies and guaranteed file-system data integrity. We propose a flexible filter-driver framework that operates transparently to both the file system and the disk driver and enforces different protection strategies. The filter uses checksumming and mirroring in order to achieve end-to-end integrity and provide hard guarantees for detection of silent data corruption and recovery of lost data. In addition, the filter uses semantic information about the driver's working in order to verify correct operation and proactively replace the driver if an anomaly is detected. We evaluated our design through a series of experiments on a prototype implementation: application-level benchmarks show modest performance overhead of 0–28% and software-implemented fault-injection (SWIFI) testing demonstrates the filter's ability to detect and transparently recover from both data-integrity problems and driver-protocol violations.*

## 1 DEALING WITH DRIVERS FAILURES

MINIX 3 is a multiserver operating system designed to survive misbehaving and crashing drivers [5, 6]. Because many failures tend to be transient, such as hardware timing issues or aging bugs, a restart takes away the proximate cause of the failure and the system can continue working normally. However, the effectiveness of such recovery depends on whether the I/O operations are idempotent and end-to-end integrity [16] is provided. For example, transparent recovery is possible for network drivers because the TCP protocol can detect garbled and lost packets and retransmit the data. In contrast, partial recovery is supported for character-device drivers because the I/O is not idempotent and I/O stream interruption is likely to cause data loss, for example, for streaming audio and video.

MINIX 3 can also restart crashed block-device drivers transparently to the file server, but the lack of end-to-end integrity and semantic information about the driver's correct operation currently make it impossible to pinpoint failures. On the one hand, even though block I/O is idempotent and can be retried, the lack of end-to-end integrity for file-system data means that user data may be corrupted silently. Likewise, a buggy driver that does not crash but returns bogus data also may go unnoticed for a long time. On the other hand, since the MINIX 3 driver manager does not know about the driver's internal working and cannot monitor individual driver requests, it cannot detect, for example, a buggy driver that causes legitimate I/O requests to fail.

Driver crashes are not just hypothetical, but actually responsible for the majority of OS crashes. For example, Windows XP crash dumps showed that 65–83% of all crashes can be attributed to drivers [3, 10]. Driver code also was found to contain of 3–7 times more bugs than other OS code [1]. Because of continuously changing driver configurations [10] and changes to the kernel code base [11], we believe that it is infeasible to fix all problems. Several ways to improve driver quality have been proposed [2, 9, 15], but as long as formally verified driver implementations are not common, unexpected failures seem inevitable.

### 1.1 Goal and Approach

This work aims to extend MINIX 3's failure-resilience mechanisms and improve dependability in the face of buggy (as opposed to malicious) block-device drivers. In particular, we want to (1) monitor the driver's correct operation and perform proactive recovery in case of anomalies and (2) protect the end user's file-system data by providing hard guarantees for both detecting data corruption and recovering lost data in the event of single-driver failures.

For flexibility reasons, our approach is based on a generic framework that allows installing a *filter driver* between the file server and block-device driver. The filter driver implements the same interface as the block-device driver so that it can be inserted between the file server and device driver—without having to modify either of them or even having them to be aware of the filter—and implement different protection strategies to safeguard the user's data. Although not the focus of this project, other filter functionality could include data encryption, compression, and increased performance or reliability using RAID techniques.

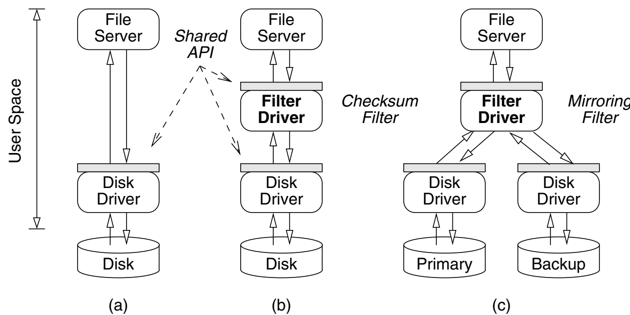


**Figure 1:** Achieving end-to-end integrity in the storage stack: a filter driver can transparently checksum and mirror data.

In order to verify correct driver operation the filter driver changes the way in which the storage stack works in two ways. First, it introduces end-to-end integrity by transparently checksumming and mirroring file-system data, as shown in Fig. 1. In this way, the filter driver can detect otherwise silent data corruption and switch to the backup partition if need be. Second, it verifies correct driver semantics by monitoring requests and replies for unexpected behavior. If an anomaly is detected, the filter driver can proactively replace the block-device driver with a fresh copy. All this is done transparently to the rest of the storage stack.

Our techniques are meant to protect individual block operations and cannot help recover from file server failures if the ordering of disk operations is violated. For example, since modern disks use caching of writes for increased performance—seek times are minimized by writing adjacent blocks first—a hard power-off due to a power failure, may cause data not to be written or to be written out-of-order. While such failures may be recovered at the individual block level, higher-level file-system inconsistencies can only be detected by the file server. For example, journaling could be used to mitigate such problems.

Although we do not explicitly examine integrity issues posed by defective hardware, many problems we solve with respect to device drivers are equally applicable to buggy firmware and hardware. For example, the detection mechanisms of our filter driver are effective against data corruption by the driver and the disk controller or drive itself. This strongly contrasts to related work that assumes trusted drivers—despite numerous field studies showing the opposite [1, 3, 10]—and, therefore, can handle only a subset of the problems that our architecture addresses.

### 1.2 Contribution and Paper Outline

The contribution of this work is improved recovery support for both driver failures and hardware problems in the storage stack. We used a filter driver to implement different protection strategies without changes to either the file system or the block-device driver. First, we used checksumming and mirroring to achieve end-to-end integrity. Second, we used semantic information to verify the driver's correct operation. We also enhanced MINIX 3 such that filter driver can proactively restart the driver if an anomaly is detected. Fault-injecting tests on a prototype demonstrate that a wide range of problems now can not only be detected, but also be repaired without user intervention.

The proposed design affects performance in both space and time. With 1-TB disks costing under $200, we do not consider disk space a scarce resource and are willing to sacrifice some space for checksums and mirroring. However, the protection mechanisms needed to guarantee end-to-end integrity are on the critical path and affect execution time. Application-level benchmarks show a modest performance overhead ranging from 0% to 28%. Where other approaches weaken integrity guarantees by offloading work to a background process, we believe this to be a reasonable price for improved dependability in safety-critical applications that are too important to fail ever.

The remainder of this paper is organized as follows. First, we survey related work (Sec. 2). Next, we describe the threat model and countermeasures (Sec. 3) and discuss the design and implementation of our filter-driver framework (Sec. 4). Then, we present the results of performance measurements, fault-injection tests, and a source-code analysis (Sec. 5). Finally, we conclude (Sec. 6).

## 2 RELATED WORK

Data integrity techniques [7, 16, 17] can be applied at the (1) hardware, (2) device-driver, (3) file-system, or (4) application level. We believe that applications should not be responsible for maintaining data integrity, and considered the file-system level and below. Since this work focuses on driver failures, we cannot rely on protection implemented by the (possibly failing) driver or underlying hardware. Data-integrity checking thus *must* be done by the file system or a filter driver between the file system and driver. The implementation can potentially benefit from hardware advances such as SCSI Data Integrity Field (SCSI DIF) or SATA External Path Protection (SATA EPP), which provide integrity metadata for I/O. Unfortunately, this requires modern hardware as well as metadata awareness in the driver layer. In contrast, our design does also work with legacy hardware and existing drivers.

Several file-system level applications of data integrity exist. For example, PFS [18], Ext4 [8], and ZFS [14] all provide some form of checksumming, typically at the block level, but sometimes also at the meta-data level. Since the file system is aware of the importance and on-disk layout of data structures, efficient checksumming algorithms are possible. For instance, ZFS stores the checksums in the parent block pointers in order to provide fault isolation between

data and checksum. A downside, however, is that developing a new file system requires a huge effort, which cannot be generalized to existing systems.

The work on IRON File Systems [13] stressed that reliability should be a first-class citizen in designing file systems and proposes grouping reliability policies together. Our approach is similar in that we also avoid the diffusion of policies throughout the code, but by implementing them in a filter driver rather than the file system, we provide added flexibility for experimentation with different protection strategies. The filter driver employs file-system agnostic, per-sector checksumming and can make all existing integrity-challenged file systems more robust to driver failures.

A related approach is integrity detection using stackable file systems [12]. Stackable file systems operate at the file (vnode) level and require significant changes to the working of the virtual file system (VFS). The work that is most relevant to ours is IOShepherd [4], a layer below the file system for enforcing data integrity and implementing reliability policies. However, using IOShepherd to enforce reliability requires making the file system "Shepherd-Aware", which involves changes to the system consistency management routines, layout engine, disk scheduler and buffer cache. In contrast, our approach is transparent to the file system and makes it possible to harden file systems without having to modify them.

We agree that the presence of semantic information at this layer helps designing more fine-grained policies, but rather than focusing on file-system data structures, we have attempted to provide the filter driver with a model of the working of the disk driver. This enabled us to detect and recover from a broad range of failures, such as erroneous driver replies, that other efforts cannot capture.

A final area where we differ from related work is that we can build on MINIX 3 [5, 6] in order to recover from a much broader class of problems. Because most of the above approaches are employed in systems with a monolithic design, even a single driver failure, such as dereferencing a bad pointer, may be fatal. In contrast, MINIX 3 compartmentalizes the OS in user space and allows replacing drivers on the fly without affecting running programs. Our filter-driver framework builds on these features (and extends them with the possibility to file complaints) in order to provide improved recovery support for faulty block-device drivers.

# 3 THREATS AND COUNTERMEASURES

We now discuss the threats posed by buggy (but not malicious) drivers and the countermeasures to achieve end-to-end integrity, verify driver semantics, and recover from failures. We believe these are generally applicable principles; our design and implementation is discussed in Sec. 4.

## 3.1 Achieving End-to-end Integrity

A first requirement for verifying correct driver operation is introducing end-to-end integrity in the storage stack. Since it is impossible to tell whether data corruption occurred due to a buggy driver or a problem with the controller or drive, we focus on the driver interface below. In particular, we identified the following threats:

*Threats when Reading Data from Disk:*

- R.1. Driver does not read the data from disk or copy it to the file system correctly, but nevertheless responds OK.
- R.2. Driver reads and returns a (different) block from the wrong position on disk.
- R.3. Driver somehow garbles the block requested and returns corrupted data.
- R.4. Driver returns old data after previously missing update. Also see threat W.1 below.
- R.5. Driver returns corrupted data due to an invalid write (see threats W.2 and W.3 below) or a hardware problem, such as buggy firmware or bit rot on the disk.

*Threats when Writing Data to Disk:*

- W.1. Driver does not write the data from the file system to disk, but responds OK anyway.
- W.2. Driver writes the data to the wrong position on disk, leading to arbitrary data corruption.
- W.3. Driver writes garbled data to the correct block.

The general approach for detecting data corruption is based on checksumming the data. In order to give hard guarantees the following countermeasures are needed:

*Approach for Reading Data from Disk:*

- R.1. If no data is returned the checksum computation over an empty buffer should detect the problem.
- R.2. Misdirected reads are detected by including the block identity in the checksum protocol, for example, $AUTH_{blockN} = checksum(data||N)$.
- R.3. If data, checksum or both are garbled, the checksum verification will fail and detect the problem.
- R.4. Stale data can only be caused by a missing write. See the solution for threat W.1 below.
- R.5. Caused by an invalid write or a hardware problem. The checksumming protocol detects these cases, as discussed for cases W.2 and W.3 below.

*Approach for Writing Data to Disk:*

- W.1. A key observation is that future reads may return stale, but otherwise valid data and checksums. Therefore, we must verify that the write succeeded by reading back either all data or just the checksum. An important assumption here is that the driver does not maintain an internal cache, but actually retrieves the data from disk.

W.2. If block N is written to position X, the error is detected when block X is read, because the block number is part of the checksumming algorithm.[1]

W.3. If we read back both the data and checksum, garbled writes will be immediately detected. If only the checksum were read back, the checksum verification of future reads will detect the data corruption.

In order to achieve full-fledged end-to-end integrity, detection of data corruption must be augmented with some form of redundancy that allows recovering corrupted data. Error-correcting codes are a possible solution, but stronger guarantees can be given by mirroring the data on a backup partition. Because one buggy driver should not be able to wipe the partition of the other, the drivers must be fully isolated with respect to the accessible I/O resources. In fact, only with two independent (preferably different) drivers and two drives on separate controllers, we can give hard guarantees for both detection of data corruption and recovery in the face of single-driver failures or single-disk failures.

### 3.2 Verifying Correct Semantics

A second requirement for verifying correct driver operation is monitoring for driver-protocol violations based on semantic information about the working of the block-device driver. This information is available at the layer directly above the block-device driver, where request and replies can be monitored for deviations from the driver's specified behavior, for example, if the driver sends an unexpected reply or fails to handle legitimate requests. What constitutes to normal behavior is specific to the FS–driver protocol. Therefore, we refer to Sec. 4.2 for a description of how we extracted the semantic information and programmed the filter driver to monitor all communication.

One problem with this approach is that it is impossible to distinguish between controller or drive failures that are faithfully reported by the driver and internal driver failures. While a failed operation may be successfully retried for temporary driver failures, the problem is likely to persist for hardware problems. The recovery strategy can acknowledge this fact by checking for similar failure characteristics and giving up after a predefined number of retries.

### 3.3 Recovering from Driver Failures

With a single-driver and single-disk configuration, the best we can do is hard guarantees for detection of data corruption—because a driver can simply wipe the entire disk with no backup to recover from. Nevertheless, two best-effort recovery strategies are possible. First, the filter driver can reissue the failed operation to the block-device driver up to N times. Second, the filter driver can complain about the driver's behavior to have it replaced with a fresh copy up to M times. After a total of M restarts $\times$ N retries, the filter has to give up and return an error to the client file server. This strategy can be attempted for both individual driver operations or the driver's entire life span.

With a mirrored setup we can give hard guarantees for recovering from single-disk or single-driver failures. Different approaches need to be distinguished for reading and writing. Recovery in case of *read failures* can be attempted by reading data from the backup partition and bringing the primary into a consistent state. The filter driver can either attempt the above best-effort recovery strategy for the primary partition or directly switch to the backup. Recovery of *write failures* poses another issue because mirroring requires all data to be written to both disks. Upon a block-device driver failure, the filter driver can first attempt best-effort recovery and, if the failure persists, gracefully degrade its service by continuing with a single partition.

## 4 DESIGN AND IMPLEMENTATION

The filter-driver framework builds on MINIX 3's support for configuring drivers at run-time. The filter driver is started using MINIX 3's *service* utility—specifying the program binary, device node, and the filter drivers's configuration parameters. The latter is used to specify the primary and backup partition to operate on, enable independently checksumming and mirroring, set the number of data sectors before each checksum sector, and configure the filter's behavior on failures. After starting the filter driver, the filter's device node can be used as if it were a normal block-device node. For example, the command sequence to create and start using a new filter-controlled file system is:

```
$ service up filter -dev /dev/filter -args <config>
$ mkfs /dev/filter
$ mount /dev/filter /mnt
```

We envision an environment where the user uses the filter driver for data on one partition and avoids it for accessing partitions with normal file systems. In this way, backward compatibility is preserved for ordinary applications, whereas safety-critical applications can, at the same time, benefit from our protection mechanisms.

### 4.1 Checksumming and Mirroring

The filter presents a *virtual disk image* smaller than the physical disk to the file server and uses the extra space for checksumming the data. Since the filter is not aware of important file-system data structures nor the file-system layout

[1] If a driver would consistently permute blocks in a nonoverlapping manner this cannot be detected, but is not a problem since data integrity is preserved. However, if the driver would crash and the new driver does not use the same permutation, all data will be detected as being corrupted.

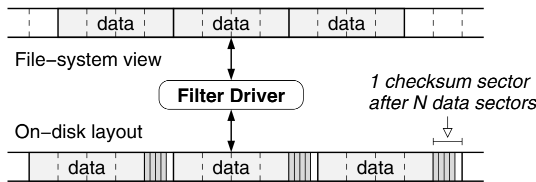


**Figure 2:** The filter driver intersperses 1 checksum sector for every N data sectors. This figure shows the on-disk layout for N=4.

on disk, we checksummed each 512-B data sector independently. In particular, we based our checksum on an XOR of all words contained in the data appended with the 64-bit disk address of the block.

Different on-disk layouts are still possible. We decided to store the checksums close to the data sectors in order to eliminate the overhead of extra disk seeks. In addition, it is important to prevent gaps in the on-disk layout of data and checksums in order to maximize the disk's bandwidth and throughput. Therefore, we interspersed data sectors and checksums sectors, as shown in Fig. 2. In principle, each checksum sector can contain SECTOR_SIZE / CHECKSUM_SIZE checksums. However, a complication for write requests is that, if the requested data's checksums do not cover a whole checksum sector, the checksum sector needs to be read before it can be written—or the checksums of the other data will be lost. Since the optimal performance depends on the file-system block size, we made the number of checksums per checksum sector a filter parameter.

If checksumming is enabled, file-server requests are handled as follows. For read operations, the filter retrieves both the data and checksum(s) from disk, calculates the checksums of the retrieved blocks, and verifies that it matches the on-disk checksums before copying the data to the file server. For write operations, the filter copies the data to its address space, calculates the checksums, and writes both the data and checksums to disk—using a single driver operation for maximum performance. Because our measurements showed a negligible performance difference in the approach for threat W.1, we decided read back both the data and checksum in the verification phase in order to gain immediate detection of data corruption.

If mirroring is enabled, read requests are still served from the primary partition, but write requests are transparently duplicated to the backup partition. In order to maximize performance, mirrored writes are not handled sequentially, but sent to both block-device drivers simultaneously.

## 4.2 Dealing with Driver Failures

Since we assume that the block-device driver can hang or crash, an important requirement it that the filter driver never blocks on an underlying driver. Therefore, the filter is programmed as a state machine and uses asynchronous IPC to forward the I/O requests to the block-device driver(s). After sending the IPC message, the filter driver schedules an alarm associated with the request, returns to its main loop, and waits for a message from any process. If a new IPC message arrives, the filter can distinguish new requests, driver replies, and timeouts based on the caller's IPC endpoint that is reliably set by the IPC subsystem.

In order to be able to detect driver-protocol violations we provided the filter driver with a semantic model of the FS–driver interface. We did this by analyzing the set of driver operations, enumerating the expected responses, and programming the filter driver to enforce this behavior. Since the model captures the behavior of the FS–driver interface, it can be used with for different block-device drivers as well. Upon starting and stopping the filter driver the underlying device is transparently opened and closed, so that DEV_OPEN and DEV_CLOSE calls can be handled local to the filter driver. Upon starting the filter driver also looks up the partition's size, using a DEV_IOCTL call, which is assumed to work correctly at this stage. The information is used to monitor subsequent DEV_READ and DEV_WRITE calls: sector-aligned requests that span a sector-multiple and do not exceed the partition's size must succeed, that is, the driver must return OK in the reply. Other operations are ignored in the current prototype implementation, but can be checked in a similar fashion, if need be.

We also extended MINIX 3's driver manager [6] such that certain trusted components can file a complaint about other, malfunctioning components. Although the driver manager can already detect driver crashes, it cannot inspect the requests and replies and is unaware of the working of the block-device driver. Therefore, we rely on the filter driver to detect deviations from the normal behavior and report to the driver manager. In order to prevent abuse, a new field in MINIX 3's isolation policies informs the driver manager whether a newly started component is allowed to use this functionality and, if so, for which parties. If a complaint is filed, the driver manager kills the bad component and restarts a fresh copy. After the driver restart, the filter driver reopens the device and reissues the failed request.

The filter driver's recovery procedure works as described in Sec. 3.3. If a data-integrity problem or driver-protocol violation is detected, the filter driver attempts best-effort recovery using M restarts $\times$ N retries on a per-request basis. Retries are not attempted if a request is undeliverable due to a block-device driver crash. If a permanent failure is detected, the filter driver gracefully degrades by shutting down the bad partition in case of mirroring. If a permanent failure occurs on the last active partition, a POSIX EIO error is reported to the user. In all problem cases, the filter writes a warning to the system log which may be inspected by the administrator to verify the system's working.

# 5 EXPERIMENTAL EVALUATION

We have evaluated our design using a prototype implementation. Below we discuss performance measurements, fault-injection testing, and a source-code analysis.

## 5.1 Performance Measurements

The performance tests were conducted on a PC with an AMD Athlon64 X2 Dual Core 4400+, 1-GiB RAM, and two identical 500-GiB Western Digital Caviar SE16 SATA hard-disk drives (WD5000AAKS). Since the on-disk location influences performance, we allocated a 64-GiB test partition at the same location at the beginning of both disks.

We first measured the raw filter performance excluding file-system overhead by directly requesting services from either the block-device driver or filter driver. The test performed 10,000 I/O requests in units of 128 KiB (matching the MINIX 3 file-system read-ahead size) with both sequential and random access patterns. The filter was configured to have 1 checksum sector after every 8 data sectors (matching the 4-KiB MINIX 3 file-system block size). The resulting disk throughput and CPU utilization are plotted in Fig. 3.

The results for sequential access are as follows. The test with no filter driver shows that the disk's maximum throughput is 92.2 MiB/s with a 13.9% CPU utilization. The null filter that merely forwards I/O requests without checksumming or mirroring does not affect throughput, but reveals an increased CPU utilization, to 18.4%, due to context-switching and copying overhead. The mirroring setup exhibits similar performance for reads from one partition, but a higher CPU utilization for writes to both the primary and backup. The checksumming setup shows that, for reads, the CPU utilization jumps to 33.3% due to computing the checksums, whereas the throughput degrades to 82.0 MiB/s because a fraction (11%) of the disk is now used to store the checksums; for writes, the throughput degrades to 52.1 MiB/s due to the read backs needed for verification. Finally, with both checksumming and mirroring, there is an extra overhead for writing, because read backs now must be done from both the primary and backup disk. As discussed in Sec. 3.1, these costs cannot be removed from the critical path without weakening the protection.

The results for random access show that the baseline disk throughput degrades to only 13.9 MiB/s, even though no filter is used. Interestingly, the worst-case overhead for random writes with both checksumming and mirroring enabled now is only 21%, because the costs are amortized over disk seeks in the 64-GiB test partition. This shows that the system's workload must be taken into account.

We also ran a series of application-level benchmarks by making a new file system on the test partition, mounting it on */mnt*, copying the MINIX 3 installation, and executing the actual test script in a chroot jail. The *mount* command

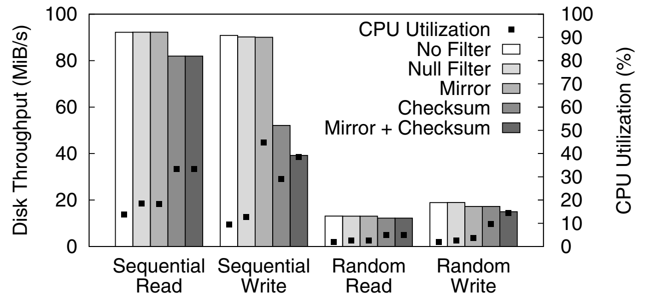


**Figure 3:** Raw disk throughput and CPU utilization for direct requests to either the block-device driver or the filter driver.

was executed on either the block-device node (*/dev/c1d0p0*) or the filter-device node (*/dev/filter*). We used a standard MINIX 3 file system with a 4-KiB block size and a 32-MiB buffer cache. After each benchmark we synchronized the cache to disk, which is included in the reported run times. The average results out of three test runs are shown in Fig. 4.

The system's workload determines the user-perceived overhead of the filter driver. First, workloads where writes dominate reads show higher overheads. Second, while the overhead is visible for I/O-bound jobs, it is negligible for CPU-intensive jobs, even with the best protection strategy. For example, with both checksumming and mirroring, the overhead compared to running without filter is 28% for copying the source tree, 13% for doing a file system check, only 4% for building the MINIX 3 OS, and 0% for building the system libraries. The overhead for normal usage without calling *sync* after each operation would be even lower.

## 5.2 Fault-injection Testing

We tested the effectiveness of our protection techniques by artificially injecting faults into the storage stack. To start with, we manually manipulated the block-device driver's code in order to mimic data-integrity violations. For example, for threat R.1 we let the driver respond OK while not doing any work, for threat R.2 we changed the disk address to be read, and so on. We also emulated protocol failures by provoking driver crashes and other erroneous behavior. These tests confirmed the filter driver's correct working with respect to detection of data corruption, repeatedly retrying failed operations, recovery of corrupted data from a mirror, and graceful degradation for permanent failures.

Next, we ran automated software-implemented fault-injection (SWIFI) tests in order to simulate more realistic errors. Our test suite uses process tracing in order to inject at run-time faults into the text segment of the block-device driver. The fault types that we used include random bit flips to emulate hardware faults as well as a range of program text transformations that mimic common C-level programming errors. For example, one fault type corrupts memory-

| Benchmark | No Filter | | Null Filter | | Mirror | | Checksum | | Mirror+Checksum | |
|---|---|---|---|---|---|---|---|---|---|---|
| Copy root FS | 14.89 | (1.00) | 15.39 | (1.03) | 15.44 | (1.04) | 17.11 | (1.15) | 18.34 | (1.23) |
| Find and touch | 2.75 | (1.00) | 2.85 | (1.04) | 2.83 | (1.03) | 2.94 | (1.07) | 2.91 | (1.06) |
| Build libraries | 28.84 | (1.00) | 28.30 | (0.98) | 29.10 | (1.01) | 28.82 | (1.00) | 28.72 | (1.00) |
| Build MINIX 3 | 14.26 | (1.00) | 14.71 | (1.03) | 14.69 | (1.03) | 14.79 | (1.04) | 14.86 | (1.04) |
| Copy source tree | 2.54 | (1.00) | 2.61 | (1.03) | 2.73 | (1.07) | 3.06 | (1.20) | 3.26 | (1.28) |
| Find and grep | 5.16 | (1.00) | 5.27 | (1.02) | 5.23 | (1.01) | 5.65 | (1.10) | 5.67 | (1.10) |
| File system check | 3.46 | (1.00) | 3.54 | (1.02) | 3.55 | (1.03) | 3.91 | (1.13) | 3.91 | (1.13) |
| Delete root FS | 10.72 | (1.00) | 10.77 | (1.00) | 11.20 | (1.05) | 12.30 | (1.15) | 13.07 | (1.22) |

**Figure 4:** Average run times in seconds and performance relative to 'No Filter' (in parentheses) for various benchmarks.

address calculations in order to emulate pointer management errors. Further background on the exact fault-injection methodology is available elsewhere [6].

For each test run we attempted 40 SWIFI trials that each injected 25 faults into the running block-device driver. This fault load ensures that the faults are triggered with a high probability. The workload during the tests consisted of writing and reading back 5-MiB randomly generated data using *dd*, and comparing the *sha1* hashes afterwards. The filter driver was configured to use checksumming but no mirroring, so that the targeted partition would not be shut down due to repeated driver failures. We limited the filter driver's recovery strategy to M=3 driver restarts and N=3 retries; experiments with different parameters showed that further recovery attempts are usually pointless. The results for three test runs are shown in Fig. 5.

The results show that the filter driver is indeed effective in dealing with misbehaving and crashing block-device drivers. To start with, we observed 94 driver restarts due to panics (18%), exceptions (23%), missing heartbeats (6%), and filter-driver complaints (52%). In all these cases, the driver manager replaced the crashed or misbehaving driver with a fresh copy. Next, the breakdown of problems shows that the filter driver can detect both data-integrity problems and driver-protocol violations such as timeouts and unexpected replies. If the retries did not help, the filter driver filed a complaint with the driver manager to replace the SATA driver. Finally, even though a 100% success rate cannot be guaranteed without a backup to recover from, the logs of filter-to-driver requests showed that the filter driver's best-effort recovery is effective, especially after a restarting a bad driver. During other test runs we also encountered a small number of filter-to-driver request with unrecoverable failures, but the mere fact that we could detect these failures and warn the user is an important step forward.

While the filter driver behaved as intended, the Sitecom Serial ATA PCI RAID controller (CN-033) hardware did not, and limited the number of faults we could inject. Compared to earlier fault-injection testing with Ethernet hardware [6] we experienced a relative large number of cases where (1) the SATA controller was confused and required a BIOS reset and (2) the test PC completely froze, presumably due to a PCI bus hang. Diagnostic output showed that the controller had difficulties with the driver's deviation from normal behavior: we observed frequent warnings that the controller was not ready, controller resets failed, or commands timed out. These are hardware problems and there is nothing the OS can do when a buggy driver issues an I/O command that causes the PCI bus to hang.

| SWIFI test run | #1 | #2 | #3 | SUM |
|---|---|---|---|---|
| - Total faults injected | 1000 | 1000 | 1000 | 3000 |
| **SATA driver restarts** | 33 | 31 | 30 | 94 |
| - Driver exit due to panic | 5 | 7 | 5 | 17 |
| - Crashed due to exception | 9 | 5 | 8 | 22 |
| - Missing driver heartbeat | 1 | 4 | 1 | 6 |
| - Filter-driver complaint | 18 | 15 | 16 | 49 |
| **Problems detected by filter** | 92 | 88 | 14 | 194 |
| - Request undeliverable | 0 | 1 | 1 | 2 |
| - Timeout receiving reply | 18 | 14 | 17 | 33 |
| - Unexpected IPC reply | 24 | 33 | 33 | 60 |
| - Legitimate request failed | 35 | 40 | 38 | 78 |
| - Bad checksum detected | 15 | 0 | 6 | 21 |
| - Read-after-write failed | 0 | 0 | 0 | 0 |
| **Filter-to-driver requests** | 1648 | 1724 | 1796 | 5168 |
| - One retry needed | 18 | 15 | 17 | 50 |
| - Two retries needed | 18 | 14 | 16 | 48 |
| - One restart needed | 18 | 14 | 17 | 49 |
| - Two restarts needed | 0 | 1 | 0 | 1 |
| - Failed requests | 0 | 0 | 0 | 0 |

**Figure 5:** Results of 3 test runs with 40 SWIFI trials that each injected 25 faults of a random type into the SATA driver.

## 5.3 Source-code Analysis

Finally, in order to give some insight in the engineering effort, we analyzed the filter driver's source code (excluding header files) and contrasted it to the (unmodified) disk driver. We used McCabe's cyclomatic complexity (CC) to calculate the overall complexity: $SUM(CC) - COUNT(CC) + 1$. The results are shown in Fig. 6.

| Part | Total Lines | Code Lines | Complexity |
|---|---|---|---|
| at_wini.c | 2726 | 1923 | 223 |
| libdriver.c | 740 | 428 | 6 |
| filter.c | 1573 | 1030 | 209 |

**Figure 6:** Code statistics for the filter and block-device driver: total line count, executable code, and cyclomatic complexity.

# 6 SUMMARY AND CONCLUSION

In this work, we have extended a multiserver operating system with a filter-driver framework and used it to improve MINIX 3's ability to deal with driver failures in the storage stack. The filter driver operates transparently to both the file-system server and block-device driver and does not require any changes to either component. This flexibility is typically not found in other approaches and proved to be very useful to implement quickly and experiment with different protection strategies.

In particular, we have used checksumming and mirroring in order to provided end-to-end integrity for file-system data. In addition, we have instrumented the filter with a semantic model of the driver so that it can detect driver-protocol violations and proactively recover. By building on MINIX 3's ability to dynamically start and stop drivers, our filter driver can provide recovery for many failures that would be fatal in other systems. For example, if the block-device driver exhibits a failure because of aging bugs, the filter driver can request the driver manager to replace the faulty block-device driver with a fresh copy.

We have evaluated our ideas by running several experiments on a prototype implementation. Fault-injection testing demonstrates the filter's ability to detect and recover from both data-integrity problems and driver-protocol violations. Performance measurements show that the average overhead for various benchmarks ranges from 0% to 28%, which seems an acceptable trade-off for improved dependability in safety-critical applications.

The framework's flexibility also greatly facilitates future experimentation. For example, one logical extension would be to use cryptographic hashes and data encryption in order to deal with not only buggy, but also malicious drivers. Another interesting option may be to provide the filter driver with information about key file-system data structures in order to implement more fine-grained protection strategies.

# AVAILABILITY

The changes and additions to MINIX 3 as well as the test scripts described in this paper are publicly available from the MINIX 3 source-code repository. The MINIX 3 operating system is free, open-source software, available via the Internet. The official homepage at http://www.minix3.org/ contains the MINIX 3 live CD, source code, documentation, news, contributed software packages, and more.

# ACKNOWLEDGEMENTS

Supported by Netherlands Organization for Scientific Research (NWO) under grant 612-060-420. Part of the work was done in the context of Google Summer of Code 2008.

# REFERENCES

[1] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An Empirical Study of Operating System Errors. In *Proc. 18th SOSP*, 2001.

[2] S. Dolev and R. Yagel. Self-stabilizing Device Drivers. *ACM Transactions on Autonomous Adaptive Systems*, 3(4):1–29, 2008.

[3] A. Ganapathi, V. Ganapathi, and D. Patterson. Windows XP Kernel Crash Analysis. In *Proc. 20th USENIX LISA*, 2006.

[4] H. S. Gunawi, V. Prabhakaran, S. Krishnan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Improving file system reliability with I/O shepherding. In *Proc. 21st SOSP*, 2007.

[5] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. Failure Resilience for Device Drivers. In *Proc. 37th DSN-DCCS*, 2007.

[6] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. Fault Isolation for Device Drivers. In *Proc. 39th DSN-DCCS*, 2009.

[7] A. Krioukov, L. N. Bairavasundaram, G. R.Goodson, K. Srinivasan, R. Thelen, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Parity lost and parity regained. In *Proc. 6th USENIX FAST*, 2008.

[8] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier. The New Ext4 Filesystem: Current Status and Future Plans. In *Proc. Linux Symposium*, 2007.

[9] F. Mérillon, L. Réveillère, C. Consel, R. Marlet, and G. Muller. Devil: An IDL for Hardware Programming. In *Proc. 4th OSDI*, 2000.

[10] B. Murphy. Automating Software Failure Reporting. *ACM Queue*, 2(8), 2004.

[11] Y. Padioleau, J. L. Lawall, and G. Muller. Understanding Collateral Evolution in Linux Device Drivers. In *Proc. 1st EuroSys*, 2006.

[12] S. Patil, A. Kashyap, G. Sivathanu, and E. Zadok. I3FS: An In-Kernel Integrity Checker and Intrusion Detection File System. In *Proc. 18th USENIX LISA*, 2004.

[13] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. IRON file systems. In *Proc. 20th SOSP*, 2005.

[14] R. Strobl (Sun Microsystems). ZFS: Revolution in File Systems. Sun Tech Days 2008-2009, 2008.

[15] L. Ryzhyk, P. Chubb, I. Kuz, and G. Heiser. Dingo: Taming Device Drivers. In *Proc. 4th EuroSys Conf.*, 2009.

[16] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end Arguments in System Design. *ACM Transactions on Computer Systems*, 2(4), 1984.

[17] G. Sivathanu, C. P. Wright, and E. Zadok. Ensuring Data Integrity in Storage: Techniques and Applications. In *Proc. 1st StorageSS Workshop*, 2005.

[18] C. A. Stein, J. H. Howard, and M. I. Seltzer. Unifying File System Protection. In *Proc. 2nd Conf. on Computer and Communications Security*, 2001.