End-to-end Integrity for File-System Data

Jorrit N. Herder, David C. van Moolenbroek, Raja Appuswamy, and Andrew S. Tanenbaum
Dept. Computer Science, Vrije Universiteit Amsterdam, The Netherlands
{jnherder,dcvmoole,raja,ast}@cs.vu.nl

Abstract

The MINIX 3 operating system is designed to restart misbehaving or crashed drivers, but currently cannot protect the user's file-system data. Because the block-device stack lacks end-to-end integrity, it is impossible to tell whether data corruption occurred—and, even if we could, there is no means to recover the data lost. Therefore, we have extended MINIX 3's failure-resilience mechanisms with guarantees for (1) detecting data corruption and (2) recovering lost data in the event of single block-device driver failures. Our approach is based on a flexible filter driver that transparently interposes upon all file system requests. Different protection strategies based on checksumming and mirroring of data are supported.

1 Introduction

MINIX 3 is a multiserver operating system designed to survive misbehaving and crashing drivers [4, 5]. Drivers are monitored at run-time and restarted if a failure is detected. Because many failures tend to be transient, such as hardware timing issues or aging bugs, a restart takes away the root cause of the failure and the system can continue working normally. The effectiveness of such recovery depends on whether the I/O operations are idempotent and end-to-end integrity [10] 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.

MINIX 3 can also restart failed block device drivers transparent to the file server, but currently cannot give any guarantees about how the file system is affected after a driver crash. Even though block I/O is idempotent and can be retried, the lack of end-to-end integrity for file system data means that the user's data may be corrupted silently. In addition, a buggy driver that does not crash but returns bogus data also may go unnoticed for a long time. Finally, even if we could detect data corruption, it is currently not possible to recover data once it has been lost. Any data corrupted after making the last back-up image thus is lost forever. Because of these problems, the user is not likely to trust the file system after a driver crash, even though the disk driver can be successfully recovered.

Therefore, 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 protect the user's file-system data by providing hard guarantees for (1) detecting data corruption and (2) recovering lost data in the event of single-driver failures.

1.1 Filter-based Design

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 transparently interpose upon traffic between the file server and device driver and implement different kinds of functionality, such as safe-

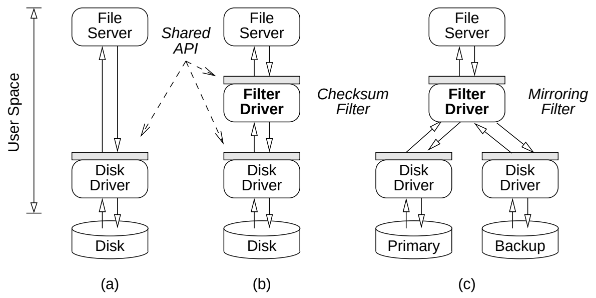

Figure 1: High-level design: (a) original setup with only the disk driver; (b) filter driver with checksumming functionality; (c) filter driver with mirroring functionality.

guarding the user's data. Although not the focus of this project, examples of other kinds of filters that are made possible by our changes include encryption of data on disk, data compression, and increased performance using RAID techniques.

In order to protect the user's file-system data, the filter changes the way in which the block-device stack works in two ways, as shown in Fig. 1. First, in order to provide end-to-end integrity and enable detection of data corruption, it modifies the low-level file system layout to use per-sector checksumming. Given that more than 50% of the files accessed are smaller than 2 KB [16], we believe that the overhead of computing checksums would not significantly degrade performance. Second, in order to support recovery of corrupted data, it uses a variant of software RAID that mirrors data written to disk. If a read request cannot be fulfilled from the primary copy, the backup copy can still be used. In order to facilitate experimenting with different configurations, our prototype implementation allows to enable or disable independently checksumming and mirroring, change the checksumming algorithm, and so on.

Our techniques are meant to protect individual block operations and cannot help to recover from file server failures if relations between disk operations are 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 file systems could be used to mitigate such problems.

We also do not examine integrity issues posed by defective hardware, that is, we assume that disks are fail-stop. Nevertheless, 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 for data corruption both by device driver and hard disk.

The proposed design affects performance in both space and time. However, we do not consider disk space a scarce resource and are willing to sacrifice some space in order to introduce checksums for error detecting. In our view, it is far more important to protect the user against data corruption. In a similar vein, a user can add a spare disk for redundant data storage and augment the system with recovery support. With hard disks costing at most a few-hundred dollars and sizes on the order of 1 TB these assumptions seem realistic.

We are concerned about performance, however. Although our modular design introduces additional context switching, the overhead of microsecond context switches is negligible compared to millisecond disk seeks. In order to minimize disk seeks, however, we have experimented with different on-disk checksum layouts. Furthermore, if mirroring is used, we have tried to do a lot of work of the in parallel. Sec. 5 presents actual performance measurements that show that the costs for improved data integrity are reasonable.

1.2 Contribution and Paper Outline

The contribution of this work is a new, generic framework to run filter drivers on MINIX 3, which we used to enhance MINIX 3's support for recovering from block-device driver failures. We have built on MINIX 3's unique ability to restart misbehaving or crashed drivers on the fly. We have augmented these facilities with checksumming and data redundancy, so that a wide range of bugs in block-device drivers not only can be detected, but also can be repaired on the fly with guaranteed data integrity.

The remainder of this paper is organized as follows. First, we put the work into context by surveying related work (Sec. 2). Next, we describe the threat model and general countermeasures (Sec. 3) and discuss the actual design and implementation of our filter framework (Sec. 4). Then, we present the results of the experiments we have run on a prototype implementation (Sec. 5). Finally, we outline directions for future work (Sec. 6) and conclude (Sec. 7).

2 Related Work

Data integrity is generally achieved by applying some form of redundancy in order to detect and correct corruption. Detailed background on the techniques to achieve data integrity is provided elsewhere [13, 6], but in this work, we are primarily interested in how these techniques can be applied. There are four levels at which data integrity can be applied, namely, (1) hardware level, (2) device-driver level, (3) file-system level and (4) application level. Since this work focuses on corruption and failures at the device-driver level, we cannot rely on protection implemented at the driver or lower-level hardware. Furthermore, we believe that applications should be not be made responsible for maintaining data integrity. Therefore, we focus on the levels around the file system below.

Several file-system level applications of data integrity are found in the literature. Protected File System (PFS) [15] uses hash-logging based on WAFS [11] to protect the integrity of its meta data. One of PFS's goals is to avoid modifications to the on-disk format for backward compatibility with existing installations. In our case, we had no restrictions on the on-disk format. We envision an environment where the user makes use of the filter driver for data on one partition and simply avoids using it for accessing data on another partition in the raw filesystem format. More recently, ZFS [2, 9] also included checksumming as an integral part of the file system. ZFS stores the checksums in the parent block pointers in order provide fault isolation between data and checksum and uses a hierarchical scheme for validation. In contrast, we wanted the filter to be file-system agnostic and hence we had to employ per-sector checksumming. In our current design, the checksum sectors and data sector are close to each other, but the flexibility of our design easily allows experimenting with different on-disk layouts if need be.

The work on IRON File Systems [8] stressed that reliability should be a first-class citizen in designing file systems and propose a taxonomy of techniques that can be used to quantify file-system reliability policies. Our approach is similar in the sense that we also focus on grouping reliability policies together, thereby avoiding the diffusion of policies through code, but we differ in the level at which we implement these policies. By implementing them in a filter driver, all existing integrity-challenged file systems can be made more robust to the failures of device drivers and we also provide added flexibility.

A first step in this direction are integrity and intrusion detection using stackable file systems [12, 7]. 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 [3], a layer below the file system for enforcing data integrity and implementing reliability policies. 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 file-system agnostic and makes it possible to harden file systems without requiring any modification.

We agree that the presence of semantic information at this layer helps designing more fine-grained policies, but our current focus is to provide a generic framework for filter drivers. Once the framework is in place it definitely becomes easier to implement and experiment with different recovery strategies. For instance, the API can be extended to transform the filter into a Shepherd or we could even have multiple filter drivers cooperating in order to form a reconfigurable, modular block transformation framework like GEOM [1], but much more powerful and semantically knowledgeable. With such a framework, implementing D-GRAID [14] style graceful degradation at the software RAID layer also becomes feasible.

Finally, by using MINIX 3 we can recover from a much broader class of problems than most of the above systems. Because these approaches are employed in systems with a monolithic design, even a single-driver failure may be fatal. For example, a driver that dereferences an invalid pointer can immediately take down the entire system. In contrast, MINIX 3 compartmentalizes the operating system in user space and allows replacing misbehaving or crashing drivers on the fly. Our filter builds on these features to provide improved recovery support for faulty block-device drivers.

3 Threats and Countermeasures

The exact threat model and countermeasures are presented below. This project solely focuses on buggy (as opposed to malicious) drivers.

3.1 Dependability Threats

As described in Sec. 1, MINIX 3 currently can detect failures and restart block device drivers, but lacks the means to detect file-system data corruption. We identified the following cases in which (silent) data corruption can occur. Combinations are possible.

3.1.1 Threats when Reading Data from Disk

- *Data on disk is OK.*

R.1. Driver does not read the data from disk or copy it to the file system, but 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.

- *Data on disk is stale due to a previous driver failure.*

R.4. Driver returns old data after previously missing update. Also see threat W.1 below.

R.5. Driver faithfully returns corrupted data found on disk. Also see threats W.2–W.4 below.

3.1.2 Threats when Writing Data to Disk

W.1. Driver does not write the data from the file system to disk, but responds OK.

W.2. Driver writes the block to the wrong position on disk.

W.3. Driver corrupts on disk the block requested to be written.

W.4. An arbitrary part of the disk gets corrupted. In fact, this threat is a generalized form of the above threats for writing data to disk.

3.2 Detection of Data Corruption

Below we outline the principle countermeasures to give hard guarantees for detection of data corruption with a single disk configuration. In general, read failures should be detected by writing both the data and its checksum to disk and comparing the checksum upon reading the data. An important observation is that write failures cannot be detected by the layer above without reading back the data written.

3.2.1 Approach for Reading Data from Disk

- *Data on disk is OK.*

R.1. If no data is returned the checksumming protocol should detect the problem. Primarily, we need to be careful about accidentally correct checksums. We decided to base our checksums on the 128-bit MD5 hash function that is publicly available.

R.2. The problem with misdirected reads [15] is avoided by including the block identity in checksum computation. In particular, we decided to calculate the MD5 hash of the data appended with the block number. The checksum protocol used thus is: $AUTH_{blockN} = md5sum(data||N)$. The block number, N, should be the 64-bit disk address of the start of the block.

R.3. The checksum will be wrong and detect the problem if just the data part is garbled, because accidental collisions are virtually impossible using the MD5 hash function. If both the data and checksum are garbled, the problem also will be detected because of the same reason.

- *Data on disk is stale due to a previous driver failure.*

R.4. Caused by a missing write. See the solution for threat W.1 below.

R.5. Caused by an invalid write (or a disk problem). The checksumming protocol can detect these cases, as discussed at the solutions for threats W.2 and W.3 below.

3.2.2 Approach for Writing Data to Disk

W.1. A key observation is that future reads may return old, but otherwise valid data and checksums. Therefore, we must verify that the data was written. This can be done by reading back either all data (based on a paranoid flag) or just the checksum written. Reading back all data is more costly, but gives immediate detection of data corruption and potentially leads to better, albeit not perfect, recovery with a single disk. Just verifying the checksum is enough to be able to detect data corruption at a later time.

W.2. In fact, this is a problem only for nonconsistent use of the wrong position. If the driver would consistently permutate (but not duplicately assign) all blocks, performance degradation may be noticeable (since the reordering may interfere with the file system's optimal layout), but the data integrity is guaranteed. If block N is accidentally written to position X, the error would be detected eventually because we include the block number, *N*, in the checksumming protocol.

W.3. Data corruption can only happen if the write partially fails. The checksum on disk is always correct, however, because this was verified for threat W.1. A future read returns the corrupted data, which will be detected when its checksum is compared with the checksum on disk.

W.4. Arbitrary corruption can happen due to writing a block to the wrong position, writing the wrong (corrupted) data, or any other illegal write actions. Detection will be based on a combination of the previous cases.

3.3 Recovery Strategies

With a single-driver and single-disk configuration, the best we can do is hard guarantees for detection of data corruption—because if a driver is allowed writing to the controller, it can simply wipe the entire disk and destroy all data with no backup to recover from. Although recovery may be possible in case of certain transient driver failures, only with two drivers and two disks, run in complete isolation, we can give hard guarantees for both detection of data corruption and recovery.

3.3.1 Single-driver Configuration

With a single-driver configuration, there are two possible best-effort recovery strategies if a problem is detected. First, the filter driver can retry issuing the failed operation to the block-device driver up to N times. Second, the filter driver can send to the driver manager a complaint about the block-device driver's faulty behavior in order to have it replaced with a fresh copy. When the newly restarted disk driver is available, the filter driver can reissue the file system request. The restart procedure can be retried up to M times, leading to a total of M restarts $\times$ N retries before giving up and returning an error to the client file system.

3.3.2 Mirroring Configuration

If a mirroring configuration with two drivers and two disks is used, we can provide hard guarantees for the recovery of data in the event of either a single-disk failure or a single-driver failure. In the mirroring setup, the filter relays all write request from the file system to both partitions, as shown in Fig. 1, whereas read requests are serviced from the primary partition. If a nonrecoverable failure occurs we switch to the backup partition.

Different recovery strategies need to be distinguished for read and write failures. Recovery in case of *read failures* can be attempted by retrieving data from backup partition and bring primary in consistent state. The filter driver could either attempt the above best-effort recovery strategy for the primary partition or directly switch to the backup partition. Recovery of *write failures* poses another issue, because mirroring requires that all data is written to both disks. If a block-device driver fails, the filter driver can attempt the above best-effort recovery strategy or gracefully degrade its service. If recovery is not successful, the filter driver shuts down the bad partition and continues operating with a single-driver configuration.

4 Design and Implementation

The work can roughly be divided in three orthogonal subprojects, which are discussed in turn.

4.1 Working of the Filter Driver

We started out by implementing a very basic filter that simply forwards all requests from the file system as well as the driver replies. The filter implements the exact same interface as the disk driver, so that it can transparently interpose upon and forward IPC communications between the file server and block-device driver. As far as the virtual file system (VFS) and file server (FS) are concerned, the filter *is* the block-device driver. Only the filter knows about the actual block-device driver, which is running when the filter driver starts. The block-device driver(s) to be used can be passed as an argument to the filter upon starting.

Since we assume the block-device driver to be buggy, an important requirement is that filter never blocks on it. Therefore, the filter is programmed as a state machine that accepts a message, handles it, forwards it using asynchronous IPC, saves the state, and returns to its main loop to receive a message from ANY. As described in Sec. 4.3, this design is also helpful to control simultaneously two block-device drivers in case of mirroring. If a new IPC message arrives, the filter can distinguish new requests and driver replies and determine what to do based on the caller's IPC endpoint.

In order to add or verify the checksums the filter driver should have access to the data that is written to or read from the file system. The filter can safely copy the data to and from the file server and block-device driver using MINIX 3's *memory grants* [5]. We decided not to investigate page-mapping optimizations because we expected disk seeks to be the dominating factor. The experiments presented in Sec. 5 indeed confirm that the copying overhead is negligible.

Finally, in order to attempt recovery if a problem is detected, the filter should be able to start and stop block-device drivers. Since the MINIX 3 driver manager is the only party that has the privileges to start and stop device drivers, a new call was added to file a complaint about malfunctioning components. In addition, MINIX 3's isolation policies were extended so that the driver manager is informed upon starting a component whether it is allowed to file complaints. If a complaint is received from a trusted party, such as the filter driver, the driver manager replaces the bad component with a fresh copy.

4.2 Checksumming Algorithm

The filter presents a *virtual hard disk image* smaller than the physical disk to the file server and uses the extra space to enforce the countermeasures discussed in Sec. 3.2. If the file server requests a write operation, the filter copies the data to its address space, calculates the checksum, and writes both the data and checksum to disk. The checksum is read back for verification. If the file server requests a read operation, the filter retrieves both the data and checksum from disk, calculates the checksum of the retrieved block, and compares it to the checksum stored on disk. If the two checksums match, the data part is returned to the file server. Otherwise, the filter concludes that the data is corrupted and attempts to retry to failed operation.

Since the filter is not aware of important file-system data structures nor the file-system layout on disk, we decided to use per-sector checksums. A sector is the smallest I/O unit that the disk controller operates on: 512 B. File-system block sizes are typically much larger, for example 4 KB in MINIX 3, so that typical I/O operations comprise multiple consecutive sectors. Each of these data sectors thus are independently checksummed by the filter.

Different on-disk layouts are still possible and the optimal layout depends on the disk access pattern. For random access, the checksum should be stored close to the data sectors, so that performance overhead caused by extra disk seeks is eliminated. Therefore, we interspersed data sectors and checksums sectors, as shown in Fig. 2. After every N data sectors there is 1 checksum sector with N checksums. For sequential access, however it is important not waste any disk space and ensure that all data returned is actually meaningful. Therefore, we made the number of consecutive data sectors before each checksum sector a parameter that can be passed to the filter upon starting.

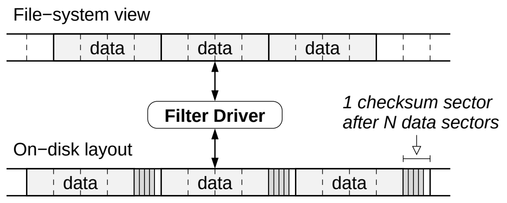

Figure 2: The filter driver intersperses 1 checksum sector for every N data sectors. This figure shows the on-disk layout for N=4.

4.3 Mirroring Approach

Finally, we want to be able to recover corrupted data by creating two isolated partitions using two drivers and two disks on separate controllers. Mirroring is enabled by specifying two instead of one partitions upon starting the filter driver. In this case the filter driver transparently duplicates write requests to both partitions. Read requests are fulfilled from the primary copy, but the system automatically uses the backup data if the primary fails. By using two physically separate disks and having a separate driver for each disk we can survive both single-drivers failures and single-disk failures.

To minimize overhead, it is important that mirroring requests are not handled sequentially, but are sent to the block-device drivers simultaneously. As described above, the filter uses asynchronous IPC and is programmed as a state machine. In this way, it can send the I/O requests to both partitions, returns to its main loop and receives a new message from ANY. If everything succeeds, the driver replies should come in one after the other. The drivers can be told apart based on the IPC endpoint that is reliably patched into the message by the kernel's IPC subsystem.

5 Experimental Evaluation

We have evaluated our design by running experiments on a prototype implementation. Below we present the results of performance measurements and fault simulation.

5.1 Performance Measurements

The performance measurements were run on a PC with an AMD Athlon64 X2 Dual Core 4400+, 1-GB 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 on both disks.

We 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 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

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)
File system check	3.46	(1.00)	3.54	(1.02)	3.55	(1.03)	3.91	(1.13)	3.91	(1.13)

Figure 3: Average run times in seconds and performance relative to 'No Filter' (in parentheses) for various benchmarks.

the reported run times. The average results out of three test runs are shown in Fig. 3.

These results show that the system's workload dominates the user-perceived overhead of the filter driver. While the filter's overhead is still 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 enabled, the overhead compared to running without filter is 28% for copying the source tree, 13% for running a file system check, only 4% for building the MINIX 3 operating system, and 0% for building the system libraries. The actual overhead for normal usage without calling *sync* after each operation would be even lower.

5.2 Fault Simulation

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 reprogrammed the driver to throw away some requests but respond the work has been done, for threat R.2 we manipulated the disk address to be read, and so on. We also emulated driver-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 single-partition failures.

6 Future Work

The flexibility of having a separate filter driver facilitates testing with different checksumming and mirroring approaches. For example, we could use error-correcting codes (ECC) in order to improve the best-effort recovery of single-driver configurations. In this scenario, an on-disk layout that separate the data sectors and checksums is probably more effective to protect against actual failures. Other possibilities include mirroring of data on one and the same partition.

Although this work focused on buggy drivers, a logical extension would take malicious drivers into account. If malicious behavior is taken into account, we generally need to worry about not only *data integrity*, but also its *authenticity* and *privacy*. Using simple checksums no longer is sufficient, since it is not a reasonable assumption that the checksum algorithm is secret. Instead, protection should be realized by means of secret key, so that the driver cannot forge the checksum.

7 Conclusion

In this work, we have extended MINIX 3 with a generic filter framework and used it to guarantee data integrity in the block-device stack. In particular, we have provided end-to-end integrity for file-system data based on checksumming and mirroring. The filter 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 data-integrity policies. Furthermore, by building on MINIX 3's ability to dynamically start and stop drivers, our filter driver can provide improved recovery support for certain failures that would be fatal in other systems. For example, in case of data corruption, the filter can ultimately request the driver manager to replace the faulty block-device driver with a fresh copy. Our experimental evaluation demonstrates the viability of this approach.

Availability

The changes and additions to MINIX 3 that were described in this paper are publicly available from the source-code repository at http://www.minix3.org/.

References

[1] GEOM: Modular Disk Transformation. Also see: http://en.wikipedia.org/wiki/GEOM.

[2] ZFS: The Last Word in File Systems. Also see: http://www.sun.com/2004-0914/feature/.

[3] 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.

[4] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. Construction of a Highly Dependable Operating System. In *Proc. 6th EDCC*, 2006.

[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] 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.

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

[8] 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.

[9] Roman Strobl (Sun microsystems). ZFS: Revolution in File Systems. Sun Tech Days 2008-2009, 2008.

[10] 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.

[11] M. I. Seltzer, G. R. Ganger, M. K. McKusick, K. A. Smith, C. A. Soules, and C. A. Stein. Journaling vs. Soft Updates: Asynchronous Meta-data Protection in File Systems. In *Proc. USENIX Annual Technical Conference*, 2000.

[12] G. Sivathanu, C. P. Wright, and E. Zadok. Enhancing File System Integrity Through Checksums. Technical Report FSL-04-04, Stony Brook University.

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

[14] M. Sivathanu, V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H.Arpaci-Dusseau. Improving storage system availability with D-GRAID. *ACM Transactions on Storage*, 1(2), 2005.

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

[16] A. S. Tanenbaum, J. N. Herder, and H. Bos. File size distributioin on UNIX systems: then and now. *ACM SIGOPS OSR*, 40(1), 2006.