

Integrated End-to-End Dependability in the Loris Storage Stack

David C. van Moolenbroek, Raja Appuswamy, Andrew S. Tanenbaum
Dept. of Computer Science, Vrije Universiteit, Amsterdam, Netherlands
{dcvmoole, raja, ast}@cs.vu.nl

Abstract—The storage stack in an operating system faces a number of dependability threats. The importance of keeping users’ data safe makes this area particularly worth investigating. We briefly describe the main threats (disk device failures, whole-system failures, software bugs, and memory corruption), and outline the Loris storage stack that we developed previously. We then present an integrated approach that combines several techniques to protect the Loris storage stack against these dependability threats, all the way from the disk driver layer to the virtual file system (VFS) layer.

I. INTRODUCTION

Failures in both hardware and software are inevitable. Any operating system should deal with such failures the best it can. The storage stack is the part of the operating system that deals with files and disks. It is of particular interest in this regard—after all, it is responsible for keeping the user’s data safe.

The storage stack is subject to a number of important threats: disk device failures, whole-system failures, software bugs, and memory corruption. Failure to deal with any of these threats has the potential to destroy data, and seriously affect all running applications. Even though there has been a considerable amount of research on subsets of these threats in subsets of the storage stack (e.g., [1], [3], [11], [14], [15], [17], [18], [19]), thus far, no single integrated approach to protecting the entire storage stack from all these threats has been presented.

In this paper, we describe how we combine various improvements that together protect better against all mentioned dependability threats. We have previously introduced Loris, a rearrangement of the traditional storage stack with fundamental improvements in the areas of reliability, heterogeneity, and flexibility [2], and we have implemented Loris in a microkernel system. We build on this work to develop techniques that result in stronger dependability guarantees for the entire storage stack than any previous approach.

Sec. 2 summarizes the four main dependability threats faced by any operating system storage stack. Sec. 3 describes our Loris storage stack, and shows how it already deals with disk device failures by design. Sec. 4 outlines the changes we propose to protect the Loris stack against the remaining threats. Sec. 5 describes related work. Sec. 6 concludes with future plans.

II. STORAGE STACK DEPENDABILITY THREATS

Any storage stack is subject to various dependability threats, at both the hardware and the software level. We briefly outline the four main threats.

Disk device failures come in many forms. Disks may not only stop working altogether (fail-stop), but also behave in various erroneous ways (fail-partial) [14], [11]. The latter type of failures may result in *silent data corruption*, where the device ends up containing different data than intended by the storage stack.

Whole-system failures, typically caused by operating system kernel crashes or power outages, may result an inconsistent on-disk state after a subsequent reboot.

As a typical storage stack consists of tens to hundreds of thousands lines of code, the presence of many **software bugs** is likely. Such bugs may result in arbitrary behavior. Their scope can be limited by compartmentalizing the various parts of the storage stack, and providing minimal, well-defined interfaces between these parts. In this paper, we focus on fail-stop failures (“crashes”) within such compartments, possibly preceded by arbitrary behavior as long as this behavior does not propagate beyond the affected compartment.

Finally, hardware-level **memory corruption** [20], [12] may result in memory pages containing arbitrary data. The storage stack is affected in particular, as modern operating systems typically use large parts of main memory to cache file data. We focus exclusively on such cached pages here, as the storage stack can play a role in limiting the effects of memory corruption on those pages in particular [19]. Other effects of memory corruption are generally indistinguishable from those of software bugs.

III. BACKGROUND: THE LORIS STORAGE STACK

Figure 1 shows the traditional storage stack as found in most operating systems, as well as our new Loris storage stack. The Loris stack has been constructed by first splitting the traditional file system into three layers: a naming, a cache, and a physical layer. The traditional software RAID layer has subsequently been swapped with the physical layer in the stack, and renamed the logical layer. These new middle four layers (naming, cache, logical, physical) communicate in terms of *files* (not blocks). Each file has a unique file identifier and can have a number of associated *attributes*. The four layers support eight so-called Loris operations: create, delete, read, write, truncate, getattr, setattr, and sync.

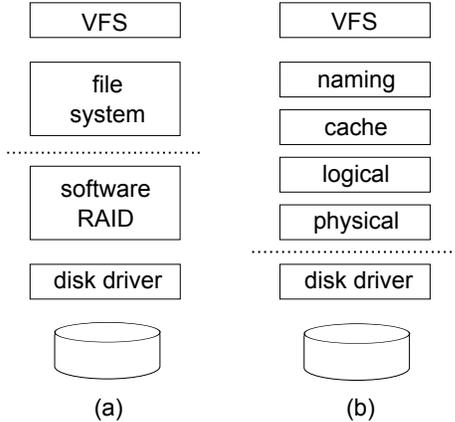


Figure 1: A conceptual view of the traditional storage stack (a) and the Loris storage stack (b). The layers above the dotted line are file aware; the layers below are only block aware.

We now briefly describe the purpose and responsibilities of each of the layers in a bottom-up fashion.

A. Layers of the stack

At the bottom, the disk drivers use device-specific commands to send block requests to hardware devices.

On top of the disk driver layer, the physical layer is responsible for the on-disk layout. It converts file operations into block operations. There may be multiple “modules” (independent instances) in the physical layer—one per device. Each module has an independent set of files, and may implement any layout scheme tailored to the underlying device. All physical modules implement parental checksumming for fully reliable detection of on-disk data corruption [2].

The logical layer implements software RAID, multiplexing file operations across physical modules. RAID policies may be assigned on a per-file basis. The logical layer keeps a *mapping* file to store each file’s RAID settings and associated physical-level files. If a physical module replies to an I/O request with a checksum failure, the logical layer uses the redundancy to restore the file when possible. This layer may also implement advanced features such as volume management and snapshotting.

The cache layer caches file data. Its internal state consists of clean and dirty pages and it uses page eviction policies that work on a per-file basis.

The naming layer handles path and file names, directories, and POSIX attributes. It maps the broad VFS vnode interface to the narrow Loris file interface. Below the naming layer, directories are simply files. POSIX attributes (e.g., user ID, file times) are stored as Loris attributes. The naming layer is also responsible for picking a unique identifier for each file. Other naming layer implementations could expose different naming schemes.

The Virtual File System (VFS) layer handles file-related application calls, and forwards calls to the lower layers as

appropriate. VFS maintains essential POSIX state for each application process, such as open file descriptors, current working and root directories, and various security properties.

We have implemented a prototype of all the layers of the Loris stack on the MINIX 3 microkernel operating system. Each Loris module is implemented as an isolated user-space process, that is, with a private address space and with access only to inter-process communication (IPC) primitives to communicate with certain other processes.

IV. INTEGRATING DEPENDABILITY INTO LORIS

One of the threats, disk device failures, is already handled in the Loris stack by design. As mentioned in Sec. 3, the stack uses parental checksumming to detect disk corruption, and redundancy to recover from such cases. The file awareness of the logical layer solves reliability problems that are inherently present in block-level RAID solutions [2], [11]. In addition, the per-file policies used by the logical layer allow us to store important files with more redundancy than less important (e.g., temporary) files.

In this section, we present our plan to deal with the remaining three threats—whole-system failures, software bugs, and memory corruption—in an integrated way. Not all parts of the stack can be covered equally well with the same techniques, so we use a combination of different techniques across the various parts of the stack: simple restart for the lowest (driver) layer, checkpoints and logging for the lower layers, execution environments for the upper layers, and a custom solution based on checksumming for the middle (cache) layer. Figure 2 shows the resulting system and the scope of each technique.

A. Lowest (driver) layer: simple restart

In a microkernel context, our assumptions in Sec. 2 about software bugs translate into the following. We use the term “process crash” to indicate detectable process failure, be it by incurring a CPU exception, triggering an assert, or causing a detectable IPC protocol violation. In our model, a process is allowed to misbehave arbitrarily before crashing, as long as the effects of such misbehavior do not propagate outside the process. As such, tolerated misbehavior includes arbitrary memory overwrites, function calls, and resource leaks within the process.

Previous work on MINIX 3 has introduced the ability to restart crashed system processes [9], that is, essential parts of the operating system that run in user-space. However, a restart results in loss of all internal state of the process. This technique by itself therefore works only for system processes that are *stateless*. In the entire Loris stack, this holds only for the disk drivers. Therefore, after a crash, a disk driver process can be restarted without any further effort. The other parts of the stack are all *stateful*, and therefore require a more sophisticated solution to recover from process crashes.

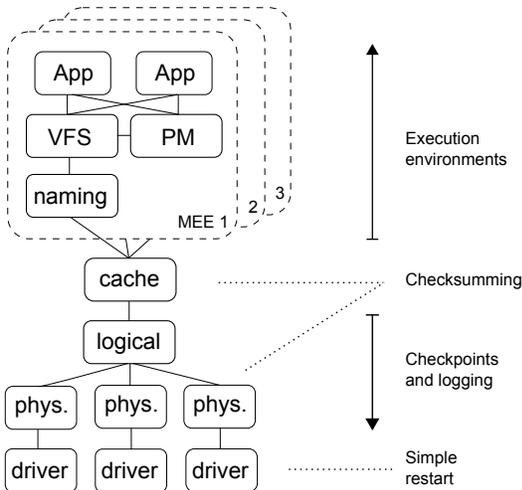


Figure 2: The processes in an example Loris/MEE setup, and the dependability protection applied to the various parts. The lines between the processes denote IPC permissions.

B. Lower layers: checkpoints and logging

The physical, logical, and cache layers are highly stateful. They contain data structures that are written to disk lazily (for performance reasons), and as a result, storage stack metadata may be inconsistent on disk at any time. Recovery from whole-system failures requires that the system return to a recent consistent state. Recovery from process crashes requires that the most recent internal state be restored in that process. This can be done by first returning to a recent consistent state, and then going forward to the latest state by replaying subsequent operations. Thus, in both cases, the recovery procedure requires the restoration of a recent consistent state.

For the logical and physical layers, we can therefore integrate whole-system and process crash recovery into a single design, by combining periodic on-disk checkpoints with an in-memory log of operations performed since each checkpoint.

Whenever a *sync* call goes through the Loris stack, all physical modules establish a consistent on-disk checkpoint. Each physical module may implement this in its own way, for example using journaling [8], [15] or copy-on-write [13], [10], [1]. The checkpoint determines what will be restored after a whole-system failure. It must contain all metadata of the storage stack, and may also include user data (in Loris, this can be configured on a per-file basis). During startup, the logical layer coordinates that all physical modules reload the same checkpoint, thus resulting in recovery to a globally consistent state.

The cache layer keeps a log of Loris operations performed after the last *sync* call. When any of the modules in the logical and physical layers crashes, all modules in these two layers are restarted using generic process restart code

(avoiding special recovery code within those modules). Just like during normal startup, they will then revert to the latest checkpoint. After this, the cache replays its log to go forward from the checkpointed state to the state right before the request causing the crash was issued. At this point, the process crash recovery is complete, and the cache may reissue the current request. If the request keeps causing a crash, the cache may give up after a few tries, and return an error.

C. Upper layers: execution environments

The VFS process directly interfaces with applications, and contains the only copy of a large set of application-related state (such as file descriptors). This makes application-transparent recovery from a VFS crash infeasible. We instead use an approach that is guaranteed to limit the effect of a VFS crash to only a subset of the running processes. We observe that even in a POSIX environment, many groups of applications do not share any state with each other, which means that it is not necessary to have a single VFS instance manage them all.

With this in mind, we can divide the applications on the system into independent groups, each consisting of one or more processes. A group could be an office suite, a mail server, etc. Every such group then gets its own copy of system servers. The servers include most notably VFS, but also PM (the POSIX process manager server in MINIX 3), and a small number of other user-interfacing servers. Together, these servers contain all the POSIX state for the processes they manage. The combination of application processes and supporting system server processes together forms what we call a *Multiserver Execution Environment* or *MEE*.

A MEE communicates only with the base system, and is isolated completely from other MEEs. The base system contains the servers that are shared by all the MEEs; this includes all device drivers, and in particular a large part of the storage stack. The MEEs are considered untrusted by the base system, that is, the base system does not in any way depend on the MEE behaving correctly.

As a result, any parts of the MEE may crash, and even misbehave completely arbitrarily without crashing. Any failure in the MEE affects the (application and system) processes within that MEE only. In such a case, the entire MEE is shut down and restarted (losing its internal state).

Each MEE gets its own file tree namespace similar to *chroot*'ed containers [16], with a number of files that are shared read-only with other MEEs (e.g., shared libraries), and a number of files that are private to the MEE. In the base system, the cache is responsible for handling the fan-in from various MEEs. It augments each file identifier as created and used by the naming layers, with a *MEE identifier*, to provide separate file ID namespaces. To control the read-only sharing of files across MEEs, access control lists are used. The cache

implements a basic quota system to prevent any single MEE from claiming all storage space.

System administrators (or package maintainers) have to configure a MEE for each isolated application. The setup consists of determining which resources the MEE may access. We believe that in most cases, the one-time setup effort is acceptable. Not only do MEEs offer better dependability across applications, they also offer security isolation (similar to virtual machines, the applications are untrusted to the MEE servers, and the MEEs are untrusted to the base system), resource isolation, and the potential to checkpoint/restart and migrate environments [16].

1) *The naming layer*: The Loris naming layer is part of the MEE. This has two main advantages. First, this allows a MEE (and thus its applications) to choose its own, private naming scheme. The cache and lower layers essentially act as an object store for the naming modules in the MEEs. Second, like the other servers in the MEE, a failure in a naming layer only affects its own MEE.

To preserve consistency of the naming layer’s metadata (e.g., directories) across MEE restarts, the naming layer must only make atomic transitions between consistent on-disk states. The implementation of this can be simple: the naming layer could send down all changes triggered by a VFS request, to the cache using a vector of operations, at the very end of processing that request.

Such atomic transitions offer an additional advantage: if a naming layer itself crashes, it can be restarted independently from the rest of the MEE, because its external state (in the lower layers) will be the same as before the VFS call that triggered the crash, and its internal state (e.g., open files) is known by VFS, and thus can be restored from there. When a restart of only the naming layer does not suffice (e.g., due to propagation of corrupted state), the failure is still isolated to a single MEE.

D. *The middle (cache) layer*

For the same reason of preserving consistency in the light of MEE crashes, the cache can not be a part of the MEEs. Additionally, in many scenarios, several MEEs will end up sharing file data, making centralized data caching only beneficial. On the other hand, the cache can not be protected by the checkpoints-and-logging technique, because it acts as the stable point to keep the log. MEEs cannot do this, as that would violate the trust boundary set between them and the base system.

This however leaves the cache as a single point of failure in the storage stack. We argue that this is acceptable: the cache has a well-defined, relatively simple task, and uses narrow interfaces on both sides. Its implementation is therefore small and consists of simple code.

Still, in certain cases we may be able to recover from a crash of the cache. Unlike for the other layers, successful recovery can not be guaranteed—whether recovery is possible

depends on whether the cache’s crucial internal data structures can be recovered from its process image after a crash. These data structures consist of all dirty cached pages, and the structures with information about these pages.

Other work proposed that on-disk checksums be reused to detect hardware-induced memory corruption in cached pages [19]. We intend to start off by implementing this idea in our stack. Page checksums are then passed up and down between the cache and physical layers as part of Loris read and write calls. They are rechecked before a user read/write call or a Loris write call, and updated on a user write call. When memory corruption ends up causing a checksum mismatch, affected clean pages can be reread from disk. Affected dirty pages have to be persisted such that subsequent application read calls result in an error. This requires only a small extension to the Loris write operation, as the physical layer already supports marking (corrupted) byte ranges in files as invalid.

Once the cache is aware of checksums, the same checksums can be used for a third purpose: to assess the validity of the dirty cached pages after a crash of the cache. A recovery procedure would iterate over all dirty pages in memory, checking the data against the checksums that are stored in memory as well. If all dirty pages have matching checksums, they can be transferred to the new instance of the cache server, and the new instance would continue operating without loss of service.

This effectively combines detection of on-disk corruption, detection of memory corruption, and best-effort process crash recovery. Upon a crash, other crucial structures in the cache must be verified separately. The compiler may help in automating the computation of checksums whenever data structures are changed [6].

V. RELATED WORK

In this section, we briefly discuss three different areas of related work: microkernel-based dependability, file system dependability, and execution environments.

A. *Microkernel-based dependability*

Previous research has shown how to restart stateless OS servers such as device drivers [9]. Recovery of stateful servers is not as trivial: when a server fails, crucial internal state may be inconsistent or even corrupted. Generic approaches can handle some of such cases [6], but are unaware of how redundancy outside the server can be used to restore lost state. Hence, we argue that better guarantees can be provided by looking at specific cases.

CuriOS [4] attempts to curb state corruption by making client-specific server state accessible to a server only when handling a request from that client. If the server fails, it is restarted and only the requesting client is killed; other clients and their state are unaffected. This is comparable to MEEs. While CuriOS works at the granularity of processes rather

than process groups, internal propagation of corrupted state is still possible, resulting in weaker state isolation guarantees than MEEs.

B. File system dependability

Membrane [17] can restart Linux file systems after crash-only failures by using checkpoints and logging. This work is closest to our protection of the lower layers. However, we cover a different part of the storage stack with this technique, most notably including logical layer, with its potentially complicated algorithms and data structures for e.g. volume management and snapshotting.

Re-FUSE [18] can restart FUSE file systems, but introduces more limitations on (mis)behavior so as to cover a broader spectrum of file systems.

A study of ZFS's ability to cope with disk and memory corruption [19] made a number of suggestions for protecting against memory corruption. We incorporate and further extend these ideas in our cache layer.

EnvyFS [3] uses n-version programming and deduplication to protect against software bugs in single file systems. We can partially achieve the same effect by mirroring files onto different physical module implementations.

C. Execution environments

The Fluke project built stackable execution environments on a microkernel [5]. It focused mainly on infrastructure aspects, and could be used to implement our ideas.

Virtual machines [7] provide the same isolation guarantees as MEEs, but their compartmentalization takes place at a coarser granularity. As such, isolation at the level of file namespaces does not come naturally. On the other hand, operating system containers [16] provide the same namespace isolation granularity as MEEs, but do not provide fault isolation within the operating system.

VI. CONCLUSION AND FUTURE WORK

In this paper, we have presented a combination of solutions that together address the dependability threats to the storage stack, providing overall better guarantees than any previous solution. We have implemented restartability of the lower layers in our current prototype, and we are working on implementing the other parts.

Future work focuses on two areas: 1) extending the per-file aspect of Loris to use more system resources for protecting important files better than other files (e.g. using more internal redundancy), and 2) investigating to which extent the modular structure of the Loris/MEE combination helps in achieving better overall performance on manycore systems.

ACKNOWLEDGMENT

This research was supported in part by European Research Council grant 227874.

REFERENCES

- [1] Sun Microsystems, Solaris ZFS file storage solution. Solaris 10 Data Sheets, 2004.
- [2] R. Appuswamy, D. C. van Moolenbroek, and A. S. Tanenbaum. Loris - A Dependable, Modular File-Based Storage Stack. In *Pacific Rim International Symposium on Dependable Computing*, PRDC'10, pages 165–174, 2010.
- [3] L. N. Bairavasundaram, S. Sundararaman, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Tolerating file-system mistakes with EnvyFS. In *Proc. of the USENIX Annual Technical Conf.*, USENIX'09, 2009.
- [4] F. M. David, E. M. Chan, J. C. Carlyle, and R. H. Campbell. CuriOS: Improving Reliability through Operating System Structure. In *USENIX Symposium on Operating Systems Design and Implementation*, OSDI'08, pages 59–72, 2008.
- [5] B. Ford, M. Hibler, J. Lepreau, P. Tullmann, G. Back, and S. Clawson. Microkernels meet recursive virtual machines. In *Proc. of the second USENIX symposium on Operating systems design and implementation*, OSDI'96, pages 137–151, 1996.
- [6] C. Giuffrida, L. Cavallaro, and A. S. Tanenbaum. We crashed, now what? In *Proc. of the Sixth international conf. on Hot topics in system dependability*, HotDep'10, pages 1–8, 2010.
- [7] R. P. Goldberg. Survey of Virtual Machine Research. *IEEE Computer*, 7(6):34–45, 1974.
- [8] R. Hagmann. Reimplementing the Cedar file system using logging and group commit. In *Proc. of the Eleventh ACM Symp. on Operating Systems Principles*, SOSP'87, pages 155–162, 1987.
- [9] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. Construction of a Highly Dependable Operating System. In *Proc. of the Sixth European Dependable Computing Conference*, EDCC'06, pages 3–12, 2006.
- [10] D. Hitz, J. Lau, and M. Malcolm. File system design for an NFS file server appliance. In *Proc. of the USENIX Winter 1994 Technical Conf.*, 1994.
- [11] 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. of the Sixth USENIX conf. on File and storage technologies*, FAST'08, pages 1–15, 2008.
- [12] X. Li, M. C. Huang, K. Shen, and L. Chu. A realistic evaluation of memory hardware errors and software system susceptibility. In *Proc. of the USENIX Annual Technical Conf.*, USENIX'10, 2010.
- [13] J. Ousterhout and F. Douglis. Beating the I/O bottleneck: a case for log-structured file systems. *SIGOPS Oper. Syst. Rev.*, 23:11–28, January 1989.
- [14] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. IRON file systems. In *Proc. of the twentieth ACM Symp. on Operating Systems Principles*, SOSP'05, pages 206–220, 2005.
- [15] M. I. Seltzer, G. R. Ganger, M. K. McKusick, K. A. Smith, C. A. N. Soules, and C. A. Stein. Journaling versus soft updates: asynchronous meta-data protection in file systems. In *Proc. of the USENIX Annual Technical Conf.*, USENIX'00, 2000.
- [16] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson. Container-based Operating System Virtualization: A Scalable, High-performance Alternative to Hypervisors. In *Proc. of the Second ACM SIGOPS/EuroSys European Conf. on Computer Systems*, EuroSys'07, pages 275–287, 2007.
- [17] S. Sundararaman, S. Subramanian, A. Rajimwale, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and M. M. Swift.

- Membrane: operating system support for restartable file systems. In *Proc. of the Eighth USENIX conf. on File and storage technologies*, FAST'10, 2010.
- [18] Swaminathan Sundararaman, Laxman Visampalli, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau. Refuse to Crash with Re-FUSE. In *Proceedings of the 6th European Conference on Computer Systems*, EuroSys'11, 2011.
- [19] Y. Zhang, A. Rajimwale, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. End-to-end data integrity for file systems: a ZFS case study. In *Proc. of the Eighth USENIX conf. on File and storage technologies*, FAST'10, 2010.
- [20] J. F. Ziegler, H. W. Curtis, H. P. Muhlfeld, C. J. Montrose, and B. Chin. IBM experiments in soft fails in computer electronics (19781994). *IBM J. Res. Dev.*, 40:3–18, January 1996.