usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications

Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau, *University of Wisconsin–Madison*

https://www.usenix.org/conference/osdi14/technical-sessions/presentation/pillai

This paper is included in the Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation.

October 6–8, 2014 • Broomfield, CO

978-1-931971-16-4

Open access to the Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation is sponsored by USENIX.

All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications

Thanumalayan Sankaranarayana Pillai   Vijay Chidambaram   Ramnatthan Alagappan
Samer Al-Kiswany   Andrea C. Arpaci-Dusseau   Remzi H. Arpaci-Dusseau
University of Wisconsin-Madison

Abstract

We present the first comprehensive study of application-level crash-consistency protocols built atop modern file systems. We find that applications use complex update protocols to persist state, and that the correctness of these protocols is highly dependent on subtle behaviors of the underlying file system, which we term *persistence properties*. We develop a tool named BOB that empirically tests persistence properties, and use it to demonstrate that these properties vary widely among six popular Linux file systems. We build a framework named ALICE that analyzes application update protocols and finds *crash vulnerabilities*, i.e., update protocol code that requires specific persistence properties to hold for correctness. Using ALICE, we analyze eleven widely-used systems (including databases, key-value stores, version control systems, distributed systems, and virtualization software) and find a total of 60 vulnerabilities, many of which lead to severe consequences. We also show that ALICE can be used to evaluate the effect of new file-system designs on application-level consistency.

1 Introduction

Crash recovery is a fundamental problem in systems research [8, 21, 34, 38], particularly in database management systems, key-value stores, and file systems. Crash recovery is hard to get right; as evidence, consider the ten-year gap between the release of commercial database products (e.g., System R [7, 8] and DB2 [34]) and the development of a working crash recovery algorithm (ARIES [33]). Even after ARIES was invented, another five years passed before the algorithm was proven correct [24, 29].

The file-systems community has developed a standard set of techniques to provide file-system metadata consistency in the face of crashes [4]: logging [5, 9, 21, 37, 45, 51], copy-on-write [22, 30, 38, 44], soft updates [18], and other similar approaches [10, 16]. While bugs remain in the file systems that implement these methods [28], the core techniques are heavily tested and well understood.

Many important applications, including databases such as SQLite [43] and key-value stores such as LevelDB [20], are currently implemented on top of these file systems instead of directly on raw disks. Such data-management applications must also be crash consistent, but achieving this goal atop modern file systems is challenging for two fundamental reasons.

The first challenge is that the exact guarantees provided by file systems are unclear and underspecified. Applications communicate with file systems through the POSIX system-call interface [48], and ideally, a well-written application using this interface would be crash-consistent on any file system that implements POSIX. Unfortunately, while the POSIX standard specifies the effect of a system call in memory, specifications of how disk state is mutated in the event of a crash are widely misunderstood and debated [1]. As a result, each file system persists application data slightly differently, leaving developers guessing.

To add to this complexity, most file systems provide a multitude of configuration options that subtly affect their behavior; for example, Linux ext3 provides numerous journaling modes, each with different performance and robustness properties [51]. While these configurations are useful, they complicate reasoning about exact file system behavior in the presence of crashes.

The second challenge is that building a high-performance application-level crash-consistency protocol is not straightforward. Maintaining application consistency would be relatively simple (though not trivial) if all state were mutated synchronously. However, such an approach is prohibitively slow, and thus most applications implement complex *update protocols* to remain crash-consistent while still achieving high performance. Similar to early file system and database schemes, it is difficult to ensure that applications recover correctly after a crash [41, 47]. The protocols must handle a wide range of corner cases, which are executed rarely, relatively untested, and (perhaps unsurprisingly) error-prone.

In this paper, we address these two challenges directly, by answering two important questions. The first question is: what are the behaviors exhibited by modern file systems that are relevant to building crash-consistent applications? We label these behaviors *persistence properties* (§2). They break down into two global categories: the *atomicity* of operations (e.g., does the file system ensure that `rename()` is atomic [32]?), and the *ordering* of operations (e.g., does the file system ensure that file creations are persisted in the same order they were issued?).

To analyze file system persistence properties, we de-

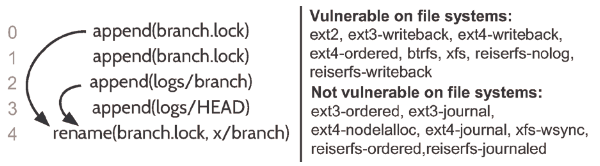
0 append(branch.lock)
1 append(branch.lock)
2 append(logs/branch)
3 append(logs/HEAD)
4 rename(branch.lock, x/branch)
Vulnerable on file systems:
ext2, ext3-writeback, ext4-writeback, ext4-ordered, btrfs, xfs, reiserfs-nolog, reiserfs-writeback
Not vulnerable on file systems:
ext3-ordered, ext3-journal, ext4-nodelalloc, ext4-journal, xfs-wsync, reiserfs-ordered,reiserfs-journaled

Figure 1: **Git Crash Vulnerability.** *The figure shows part of the Git update protocol. The arrows represent ordering dependencies: if the appends are not persisted before the rename, any further commits to the repository fail. We find that, whether the protocol is vulnerable or not varies even between configurations of the same file system.*

velop a simple tool, known as the *Block Order Breaker* (BOB). BOB collects block-level traces underneath a file system and re-orders them to explore possible on-disk crash states that may arise. With this simple approach, BOB can find which persistence properties do *not* hold for a given system. We use BOB to study six Linux file systems (ext2, ext3, ext4, reiserfs, btrfs, and xfs) in various configurations. We find that persistence properties vary widely among the tested file systems. For example, appends to file A are persisted before a later rename of file B in the ordered journaling mode of ext3, but not in the same mode of ext4, unless a special option is enabled.

The second question is: do modern applications implement crash consistency protocols correctly? Answering this question requires understanding update protocols, no easy task since update protocols are complex [47] and spread across multiple files in the source code. To analyze applications, we develop ALICE, a novel framework that enables us to systematically study application-level crash consistency (§3). ALICE takes advantage of the fact that, no matter how complex the application source code, the update protocol boils down to a sequence of file-system related system calls. By analyzing permutations of the system-call trace of workloads, ALICE produces *protocol diagrams*: rich annotated graphs of update protocols that abstract away low-level details to clearly present the underlying logic. ALICE determines the exact persistence properties assumed by applications as well as flaws in their design.

Figure 1 shows an example of ALICE in action. The figure shows a part of the update protocol of Git [26]. ALICE detected that the appends need to be persisted before the rename; if not, any future commits to the repository fail. This behavior varies widely among file systems: a number of file-system features such as delayed allocation and journaling mode determine whether file systems exhibit this behavior. Some common configurations like ext3 ordered mode persist these operations in order, providing a false sense of security to the developer.

We use ALICE to study and analyze the update protocols of eleven important applications: LevelDB [20], GDBM [19], LMDB [46], SQLite [43], PostgreSQL [49], HSQLDB [23], Git [26], Mercurial [31]), HDFS [40], ZooKeeper [3], and VMWare Player [52]. These applications represent software from different domains and at varying levels of maturity. The study focuses on file-system behavior that affects users, rather than on strictly verifying application correctness. We hence consider typical usage scenarios, sometimes checking for additional consistency guarantees beyond those promised in the application documentation. Our study takes a pessimistic view of file-system behavior; for example, we even consider the case where renames are not atomic on a system crash.

Overall, we find that application-level consistency in these applications is highly sensitive to the specific persistence properties of the underlying file system. In general, if application correctness depends on a specific file-system persistence property, we say the application contains a *crash vulnerability*; running the application on a different file system could result in incorrect behavior. We find a total of 60 vulnerabilities across the applications we studied; several vulnerabilities have severe consequences such as data loss or application unavailability. Using ALICE, we also show that many of these vulnerabilities (roughly half) manifest on current file systems such as Linux ext3, ext4, and btrfs.

We find that many applications implicitly expect *ordering* among system calls (e.g., that writes, even to different files, are persisted in order); when such ordering is not maintained, 7 of the 11 tested applications have trouble properly recovering from a crash. We also find that 10 of the 11 applications expect *atomicity* of file-system updates. In some cases, such a requirement is reasonable (e.g., a single 512-byte write or file rename operation are guaranteed to be atomic by many current file systems when running on a hard-disk drive); in other situations (e.g., with file appends), it is less so. We also note that some of these atomicity assumptions are not future proof; for example, new storage technology may be atomic only at a smaller granularity than 512-bytes (e.g., eight-byte PCM [12]). Finally, for 7 of the 11 applications, *durability* guarantees users likely expect are not met, often due to directory operations not being flushed.

ALICE also enables us to determine whether new file-system designs will help or harm application protocols. We use ALICE to show the benefits of an ext3 variant we propose (ext3-fast), which retains much of the positive ordering and atomicity properties of ext3 in data journaling mode, without the high cost. Such verification would have been useful in the past; when delayed allocation was introduced in Linux ext4, it broke several applications, resulting in bug reports, extensive mailing-list discussions, widespread data loss, and finally, file-system changes [14]. With ALICE, testing the impact of changing persistence properties can become part of the file-system design process.

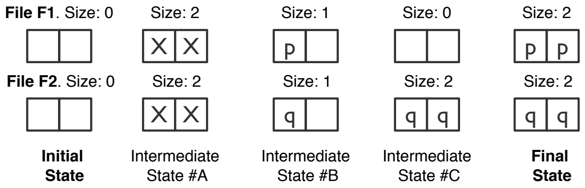

Figure 2: **Crash States.** *The figure shows the initial, final, and some of the intermediate crash states possible for the workload described in Section 2.1 . X represents garbage data in the files. Intermediate states #A and #B represent different kinds of atomicity violations, while intermediate state #C represents an ordering violation.*

2 Persistence Properties

In this section, we study the *persistence properties* of modern file systems. These properties determine which possible post-crash file system states are possible for a given file system; as we will see, different file systems provide subtly different guarantees, making the challenge of building correct application protocols atop such systems more vexing.

We begin with an example, and then describe our methodology: to explore possible on-disk states by reordering the I/O block stream, and then examine possible resulting states. Our testing is not complete, but finds persistence properties that do *not* hold for a file-system implementation. We then discuss our findings for six widely-used Linux file systems: ext2 [6], ext3 [51], ext4 [50], btrfs [30], xfs [45], and reiserfs [37].

Application-level crash consistency depends strongly upon these persistence properties, yet there are currently no standards. We believe that defining and studying persistence properties is the first step towards standardizing them across file systems.

2.1 An Example

All application update protocols boil down to a sequence of I/O-related system calls which modify on-disk state. Two broad properties of system calls affect how they are persisted. The first is *atomicity*: does the update from the call happen all at once, or are there possible intermediate states that might arise due to an untimely crash? The second is *ordering*: can this system call be persisted *after* a later system call? We now explain these properties with an example.

We consider the following pseudo-code snippet:

```
write(f1, "pp");
write(f2, "qq");
```

In this example, the application first appends the string `pp` to file descriptor `f1` and then appends the string `qq` to file descriptor `f2`. Note that we will sometimes refer to such a `write()` as an `append()` for simplicity.

Figure 2 shows a few possible crash states that can result. If the append is not *atomic*, for example, it would be possible for the *size* of the file to be updated without the new data reflected to disk; in this case, the files could contain garbage, as shown in State A in the diagram. We refer to this as *size-atomicity*. A lack of atomicity could also be realized with only part of a write reaching disk, as shown in State B. We refer to this as *content-atomicity*.

If the file system persists the calls out of order, another outcome is possible (State C). In this case, the second write reaches the disk first, and as a result only the second file is updated. Various combinations of these states are also possible.

As we will see when we study application update protocols, modern applications expect different atomicity and ordering properties from underlying file systems. We now study such properties in detail.

2.2 Study and Results

We study the persistence properties of six Linux file systems: ext2, ext3, ext4, btrfs, xfs, and reiserfs. A large number of applications have been written targeting these file systems. Many of these file systems also provide multiple configurations that make different trade-offs between performance and consistency: for instance, the data journaling mode of ext3 provides the highest level of consistency, but often results in poor performance [35]. Between file systems and their various configurations, it is challenging to know or reason about which persistence properties are provided. Therefore, we examine different configurations of the file systems we study (a total of 16).

To study persistence properties, we built a tool, known as the *Block Order Breaker* (Bob), to empirically find cases where various persistence properties do *not* hold for a given file system. Bob first runs a simple user-supplied workload designed to stress the persistence property tested (e.g., a number of writes of a specific size to test overwrite atomicity). Bob collects the block I/O generated by the workload, and then re-orders the collected blocks, selectively writing some of them to disk to generate a new *legal* disk state (disk barriers are obeyed). In this manner, Bob generates a number of unique disk images corresponding to possible on-disk states after a system crash. Bob then runs file-system recovery on each resulting disk image, and checks whether various persistence properties hold (e.g., if writes were atomic). If Bob finds even a single disk image where the checker fails, then we know that the property does not hold on the file system. Proving the converse (that a property holds in all situations) is not possible using Bob; currently, only simple block re-orderings and all prefixes of the block trace are tested.

Note that different system calls (e.g., `writev()`, `write()`) lead to the same file-system output. We group such calls together into a generic file-system update we term an *operation*. We have found that grouping all operations into three major categories is sufficient for our purposes here: file overwrite, file append, and directory operations (including rename, link, unlink, mkdir, etc.).

Persistence Property	**File system**															
	ext2	ext2-sync	ext3-writeback	ext3-ordered	ext3-datajournal	ext4-writeback	ext4-ordered	ext4-nodelalloc	ext4-datajournal	btrfs	xfs	xfs-wsync	reiserfs-nolog	reiserfs-writeback	reiserfs-ordered	reiserfs-datajournal
Atomicity																
Single sector overwrite																
Single sector append	×		×			×								×		
Single block overwrite	×	×	×	×		×	×	×			×	×	×	×	×	
Single block append	×	×	×			×							×	×		
Multi-block append/writes	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
Multi-block prefix append	×	×	×			×							×	×		
Directory op	×	×											×			
Ordering																
Overwrite → Any op	×		×	×		×	×	×			×	×	×	×	×	
[Append, rename]→ Any op	×		×			×							×	×		
O_TRUNC Append → Any op	×		×			×							×	×		
Append → Append (same file)	×		×			×							×	×		
Append → Any op	×		×			×	×			×	×		×	×		
Dir op → Any op	×									×			×			

Table 1: **Persistence Properties.** *The table shows atomicity and ordering persistence properties that we empirically determined for different configurations of file systems. X → Y indicates that X is persisted before Y. [X,Y] → Z indicates that Y follows X in program order, and both become durable before Z. A × indicates that we have a reproducible test case where the property fails in that file system.*

Table 1 lists the results of our study. The table shows, for each file system (and specific configuration) whether a particular persistence property has been found to *not* hold; such cases are marked with an ×.

The size and alignment of an overwrite or append affects its atomicity. Hence, we show results for single sector, single block, and multi-block overwrite and append operations. For ordering, we show whether given properties hold assuming different orderings of overwrite, append, and directory operations; the append operation has some interesting special cases relating to delayed allocation (as found in Linux ext4) – we show these separately.

2.2.1 Atomicity

We observe that all tested file systems seemingly provide atomic single-sector overwrites: in some cases (e.g., ext3-ordered), this property arises because the underlying disk provides atomic sector writes. Note that if such file systems are run on top of new technologies (such as PCM) that provide only byte-level atomicity [12], single-sector overwrites will not be atomic.

Providing atomic appends requires the update of two locations (file inode, data block) atomically. Doing so requires file-system machinery, and is not provided by ext2 or writeback configurations of ext3, ext4, and reiserfs.

Overwriting an entire block atomically requires data journaling or copy-on-write techniques; atomically appending an entire block can be done using ordered mode journaling, since the file system only needs to ensure the entire block is persisted before adding a pointer to it.

Current file systems do not provide atomic multi-block appends; appends can be broken down into multiple operations. However, most file systems seemingly guarantee that some prefix of the data written (e.g., the first 10 blocks of a larger append) will be appended atomically.

Directory operations such as `rename()` and `link()` are seemingly atomic on all file systems that use techniques like journaling or copy-on-write for consistency.

2.2.2 Ordering

We observe that ext3, ext4, and reiserfs in data journaling mode, and ext2 in sync mode, persist all tested operations in order. Note that these modes often result in poor performance on many workloads [35].

The append operation has interesting special cases. On file systems with the delayed allocation feature, it may be persisted after other operations. A special exception to this rule is when a file is appended, and then renamed. Since this idiom is commonly used to atomically update files [14], many file systems recognize it and allocate blocks immediately. A similar special case is appending to files that have been opened with O_TRUNC. Even with delayed allocation, successive appends to the same file are persisted in order. Linux ext2 and btrfs freely reorder directory operations (especially operations on different directories [11]) to increase performance.

2.3 Summary

From Table 1, we observe that persistence properties vary *widely* among file systems, and even among different configurations of the same file system. The order of persistence of system calls depends upon small details like whether the calls are to the same file or whether the file was renamed. From the viewpoint of an application developer, it is risky to assume that any particular property will be supported by all file systems.

3 The Application-Level Intelligent Crash Explorer (ALICE)

We have now seen that file systems provide different persistence properties. However, some important questions remain: How do current applications update their on-disk structures? What do they assume about the underlying file systems? Do such update protocols have vulnerabilities? To address these questions, we developed ALICE *(Application-Level Intelligent Crash Explorer).* ALICE constructs different on-disk file states that may result due to a crash, and then verifies application correctness on each created state.

Unlike other approaches [53, 54] that simply test an application atop a given storage stack, ALICE finds the generic persistence properties required for application correctness, without being restricted to only a specified

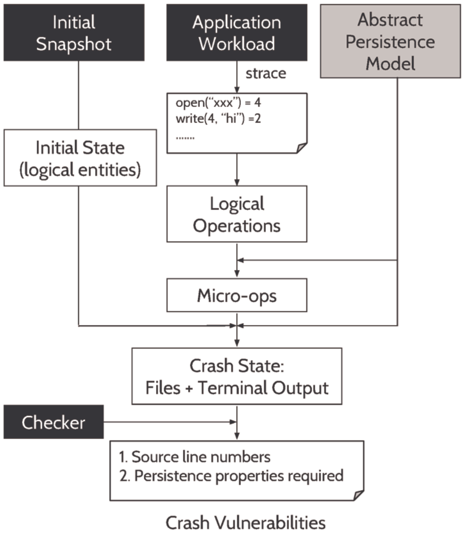

Figure 3: **ALICE Overview.** *The figure shows how ALICE converts user inputs into crash states and finally into crash vulnerabilities. Black boxes are user inputs. Grey boxes are optional inputs.*

file system. ALICE associates discovered vulnerabilities directly with source lines, and targets specific states that are prone to reveal crash vulnerabilities in different source lines. ALICE achieves this by constructing file states *directly* from the system-call trace of an application workload. The states to be explored and verified can be described purely in terms of system calls: the actual storage stack is not involved. ALICE can also be used to abstractly test the safety of new file systems.

We first describe how ALICE is used (§3.1). We then describe how ALICE calculates states possible during a system crash, using an Abstract Persistence Model (APM) (§3.2). Next, we describe how these states are selectively explored so as to discover application requirements in terms of persistence properties (§3.3), and how discovered vulnerabilities are reported associated with source code lines (§3.4). Finally, we describe our implementation (§3.5) and its limitations (§3.6).

3.1 Usage

ALICE is simple to use. The user first supplies ALICE with an initial snapshot of the files used by the application (typically an entire directory), and a workload script that exercises the application (such as performing a transaction). The user also supplies a checker script corresponding to the workload that verifies whether invariants of the workload are maintained (such as atomicity of the transaction). ALICE runs the checker atop different *crash states*, i.e., the state of files after rebooting from a system crash that can occur during the workload. ALICE then produces a logical representation of the update protocol executed during the workload, vulnerabilities in the protocol and their associated source lines, and persistence properties required for correctness.

```
# Workload

# Opening database
db = gdbm.open('mydb')

# Inserting key-value
db['x'] = 'foo'
db.sync()

print 'Done'
```

```
# Checker

db = gdbm.open('mydb')
c = len(db.keys())
if alice.printed('Done'):
  assert c == 1
else:
  assert c == 0 or c == 1
if c == 1:
  assert db['x'] == 'foo'
```

Listing 1: **Workload and Checker.** *Simplified form of python workload and checker for GDBM (a key-value store).*

The exact crash states possible for a workload varies with the file system. For example, depending on the file system, appending to a file can result in the file containing either a prefix of the data persisted, with random data intermixed with file data, or various combinations thereof. ALICE uses file-system *Abstract Persistence Models (APMs)* to define the exact crash states possible in a given file system. By default, ALICE uses an APM with few restrictions on the possible crash states, so as to find generic persistence properties required for application correctness. However, ALICE can be restricted to find vulnerabilities occurring only on a specific file system, by supplying the APM of that file system.

Listing 1 shows example workload and checker scripts for GDBM, a key-value store, written in Python. We discuss how APMs are specified in the next subsection.

3.2 Crash States and APMs

Figure 3 shows an overview of the steps ALICE follows to find crash vulnerabilities. The user-supplied workload is first run, and a system-call trace obtained; the trace represents an execution of the application's update protocol. The trace is converted into a sequence of *logical operations* by ALICE. The sequence of logical operations, along with an APM, is used to calculate the different crash states that are possible from the initial state. These steps are now explained in detail.

3.2.1 Logical Operations

ALICE first converts the trace of system calls in the application workload to *logical operations*. Logical operations abstract away details such as current read and write offsets, file descriptors, and transform a large set of system calls and other I/O producing behavior into a small set of file-system operations. For example, `write()`, `pwrite()`, `writev()`, `pwritev()`, and `mmap()`-writes are all translated into `overwrite` or `append` logical operations. Logical operations also associate a conceptual inode to each file or directory involved.

3.2.2 Abstract Persistence Models

An APM specifies *all* constraints on the atomicity and ordering of logical operations for a given file system, thus defining which crash states are possible.

APMs represent crash states as consisting of two logical entities: *file inodes* containing data and a file size,

Logical Operation	Micro-operations
overwrite	$N\times$ write_block(data)

append	$N\times$ { change_file_size
	write_block(random)
	write_block(data)

truncate	$N\times$ { change_file_size
	write_block(random)
	write_block(zeroes)

link	create_dir_entry
unlink	delete_dir_entry + *truncate if last link*

rename	delete_dir_entry(dest) + *truncate if last link*
	create_dir_entry(dest)
	delete_dir_entry(source)

print	stdout

(a) Atomicity Constraints.

Description	Constraint
sync-ops	$[\textit{any-op}_i(\text{A}) \ldots \text{fsync}_j(\text{A})] \rightarrow \textit{any-op}_k \ \forall\ i < j < k$
stdout	$\text{stdout}_i() \rightarrow \textit{any-op}_j \ \forall\ i < j$

(b) Ordering Constraints.

Table 2: **Default APM Constraints.** *(a) shows atomicity constraints; N indicates a logical operation being divided into many micro-ops. (b) shows ordering constraints.* X_i *is the* i^{th} *operation, and any-op(A) is an operation on the file or directory A.*

and *directories* containing directory entries. Each logical operation operates on one or more of these entities. An infinite number of instances of each logical entity exist, and they are never allocated or de-allocated, but rather simply changed. Additionally, each crash state also includes any output printed to the terminal during the time of the crash as a separate entity.

To capture intermediate crash states, APMs break logical operations into *micro-operations*, i.e., the smallest atomic modification that can be performed upon each logical entity. There are five micro-ops:

- *write_block*: A write of size *block* to a file. Two special arguments to *write_block* are *zeroes* and *random*: *zeroes* indicates the file system initializing a newly allocated block to zero; *random* indicates an uninitialized block. Writes beyond the end of a file cause data to be stored without changing file size.
- *change_file_size*: Changes the size of a file inode.
- *create_dir_entry*: Creates a directory entry in a directory, and associates a file inode or directory with it.
- *delete_dir_entry*: Deletes a directory entry.
- *stdout*: Adds messages to the terminal output.

The APM specifies atomicity constraints by defining how logical operations are translated into micro-ops. The APM specifies ordering constraints by defining which micro-ops can reach the disk before other micro-ops.

In most cases, we utilize a default APM to find the greatest number of vulnerabilities in application update protocols. The atomicity constraints followed by this default file system are shown in Table 2(a), which specifically shows how each logical operation is broken down into micro-ops. The ordering constraints imposed by the default APM are quite simple, as seen in Table 2(b): all micro-ops followed by a sync on a file A are ordered after writes to A that precede the sync. Similar ordering also applies to *stdout*, and additionally, all operations following an *stdout* must be ordered after it.

```
open(path="/x2VC") = 10
Micro-ops: None
Ordered after: None

pwrite(fd=10, offset=0, size=1024)
Micro-ops: #1 write_block(inode=8, offset=0, size=512)
           #2 write_block(inode=8, offset=512, size=512)
Ordered after: None

fsync(10)
Micro-ops: None
Ordered after: None

pwrite(fd=10, offset=1024, size=1024)
Micro-ops: #3 write_block(inode=8, offset=1024, size=512)
           #4 write_block(inode=8, offset=1536, size=512)
Ordered after: #1, #2

link(oldpath="/x2VC", newpath="/file")
Micro-ops: #5 create_dir_entry(dir=2, entry='file', inode=8)
Ordered after: #1, #2

write(fd=1, data="Writes recorded", size=15)
Micro-ops: #6 stdout("Writes recorded")
Ordered after: #1, #2
```

Listing 2: **Annotated Update Protocol Example.** *Micro-operations generated for each system call are shown along with their dependencies. The inode number of* x2VC *is 8, and for the root directory is 2. Some details of listed system calls have been omitted.*

ALICE can also model the behavior of real file systems when configured with other APMs. As an example, for the ext3 file system under the *data=journal* mode, the ordering constraint is simply that each micro-op depends on all previous micro-ops. Atomicity constraints for ext3 are mostly simple: all operations are atomic, except file writes and truncates, which are split at block-granularity. Atomic renames are imposed by a circular ordering dependency between the micro-ops of each rename.

3.2.3 Constructing crash states.

As explained, using the APM, ALICE can translate the system-call trace into micro-ops and calculate ordering dependencies amongst them. Listing 2 shows an example system-call trace, and the resulting micro-ops and ordering constraints. ALICE also represents the initial snapshot of files used by the application as logical entities.

ALICE then selects different sets of the translated micro-ops that obey the ordering constraints. A new crash state is constructed by sequentially applying the micro-ops in a selected set to the initial state (represented as logical entities). For each crash state, ALICE then converts the logical entities back into actual files, and supplies them to the checker. The user-supplied checker thus verifies the crash state.

3.3 Finding Application Requirements

By default, ALICE targets specific crash states that concern the ordering and atomicity of each individual system call. The explored states thus relate to basic persistence properties like those discussed in Section 2, making it straightforward to determine application requirements. We now briefly describe the crash states explored.

Atomicity *across* System Calls. The application update protocol may require multiple system calls to be persisted together atomically. This property is easy to check: if the protocol has N system calls, ALICE constructs one crash state for each prefix (i.e., the first X system calls, $\forall\ 1 < X < N$) applied. In the sequence of crash states generated in this manner, the first crash state to have an application invariant violated indicates the start of an atomic group. The invariant will hold once again in crash states where all the system calls in the atomic group are applied. If ALICE determines that a system call X is part of an atomic group, it does not test whether the protocol is vulnerable to X being persisted out of order, or being partially persisted.

System-Call Atomicity. The protocol may require a single system call to be persisted atomically. ALICE tests this for each system call by applying all previous system calls to the crash state, and then generating crash states corresponding to different intermediate states of the system call and checking if application invariants are violated. The intermediate states for file-system operations depend upon the APM, as shown (for example) in Table 2. Some interesting cases include how ALICE handles appends and how it explores the atomicity of writes. For appends, we introduce intermediate states where blocks are filled with random data; this models the update of the size of a file reaching disk before the data has been written. We split overwrites and appends in two ways: into block-sized micro-operations, and into three parts regardless of size. Though not exhaustive, we have found our exploration of append and write atomicity useful in finding application vulnerabilities.

Ordering Dependency among System Calls. The protocol requires system call A to be persisted before B if a crash state with B applied (and not A) violates application invariants. ALICE tests this for each pair of system calls in the update protocol by applying every system call from the beginning of the protocol until B except for A.

3.4 Static Vulnerabilities

ALICE must be careful in how it associates problems found in a system-call trace with source code. For example, consider an application issuing ten writes in a loop. The update protocol would then contain ten `write()` system calls. If each write is required to be atomic for application correctness, ALICE detects that *each* system call is involved in a vulnerability; we term these as **dynamic** vulnerabilities. However, the cause of all these vulnerabilities is a single source line. ALICE uses stack trace information to correlate all 10 system calls to the line, and reports it as a single **static** vulnerability. In the rest of this paper, we only discuss static vulnerabilities.

3.5 Implementation

ALICE consists of around 4000 lines of Python code, and also traces memory-mapped writes in addition to system calls. It employs a number of optimizations.

First, ALICE caches crash states, and constructs a new crash state by incrementally applying micro-operations onto a cached crash state. We also found that the time required to check a crash state was much higher than the time required to incrementally construct a crash state. Hence, ALICE constructs crash states sequentially, but invokes checkers concurrently in multiple threads.

Different micro-op sequences can lead to the same crash state. For example, different micro-op sequences may write to different parts of a file, but if the file is unlinked at the end of sequence, the resulting disk state is the same. Therefore, ALICE hashes crash states and only checks the crash state if it is new.

We found that many applications write to debug logs and other files that do not affect application invariants. ALICE filters out system calls involved with these files.

3.6 Limitations

ALICE is not complete, in that there may be vulnerabilities that are not detected by ALICE. It also requires the user to write application workloads and checkers; we believe workload automation is orthogonal to the goal of ALICE, and various model-checking techniques can be used to augment ALICE. For workloads that use multiple threads to interact with the file system, ALICE serializes system calls in the order they were issued; in most cases, this does not affect vulnerabilities as the application uses some form of locking to synchronize between threads. ALICE currently does not handle file attributes; it would be straight-forward to extend ALICE to do so.

4 Application Vulnerabilities

We study 11 widely used applications to find whether file-system behavior significantly affects application users, which file-system behaviors are thus important, and whether testing using ALICE is worthwhile in general. One of ALICE's unique advantages, of being able to find targeted vulnerabilities under an abstract file system and reporting them in terms of a persistence property violated, is thus integral to the study. The applications each represent different domains, and range in maturity from a few years-old to decades-old. We study three key-value stores (LevelDB [20], GDBM [19], LMDB [46]), three relational databases (SQLite [43], PostgreSQL [49], HSQLDB [23]), two version control systems (Git [26], Mercurial [31]), two distributed systems (HDFS [40], ZooKeeper [3]), and a virtualization software (VMWare Player [52]). We study two versions of LevelDB (1.10, 1.15), since they vary considerably in their update-protocol implementation.

Aiming towards the stated goal of the study, we try to consider typical user expectations and deployment scenarios for applications, rather than only the guarantees listed in their documentation. Indeed, for some applications (Git, Mercurial), we could not find any documented guarantees. We also consider file-system behaviors that may not be common now, but may become prevalent in the future (especially with new classes of I/O devices). Moreover, the number of vulnerabilities we report (in each application) only relates to the number of source code lines depending on file-system behavior. Note that, due to these reasons, the study is not suitable for comparing the correctness between different applications, or towards strictly verifying application correctness.

We first describe the workloads and checkers used in detecting vulnerabilities (§4.1). We then present an overview of the protocols and vulnerabilities found in different applications (§4.2). We discuss the importance of the discovered vulnerabilities (§4.3), interesting patterns observable among the vulnerabilities (§4.4), and whether vulnerabilities are exposed on current file systems (§4.5). We also evaluate whether ALICE can validate new file-system designs (§4.6).

4.1 Workloads and Checkers

Most applications have configuration options that change the update protocol or application crash guarantees. Our workloads test a total of 34 such configuration options across the 11 applications. Our checkers are conceptually simple: they do read operations to verify workload invariants for that particular configuration, and then try writes to the datastore. However, some applications have complex invariants, and recovery procedures that they expect users to carry out (such as removing a leftover lock file). Our checkers are hence complex (e.g., about 500 LOC for Git), invoking all recovery procedures we are aware of that are expected of normal users.

We now discuss the workloads and checkers for each application class. Where applicable, we also present the guarantees we believe each application makes to users, information garnered from documentation, mailing-list discussions, interaction with developers, and other relevant sources.

Key-value Stores and Relational Databases. Each workload tests different parts of the protocol, typically opening a database, and inserting enough data to trigger checkpoints. The checkers check for atomicity, ordering, and durability of transactions. We note here that GDBM does not provide any crash guarantees, though we believe lay users will be affected by any loss of integrity. Similarly, SQLite does not provide durability under the default journal-mode (we became aware of this only after interacting with developers), but its documentation seems misleading. We enable checksums on LevelDB.

Version Control Systems. Git's crash guarantees are fuzzy; mailing-list discussions suggest that Git expects a fully-ordered file system [27]. Mercurial does not provide *any* guarantees, but does provide a plethora of manual recovery techniques. Our workloads add two files to the repository and then commit them. The checker uses commands like `git-log`, `git-fsck`, and `git-commit` to verify repository state, checking the integrity of the repository and the durability of the workload commands. The checkers remove any leftover lock files, and perform recovery techniques that do not discard committed data or require previous backups.

Virtualization and Distributed Systems. The VMWare Player workload issues writes and flushes from within the guest; the checker repairs the virtual disk and verifies that flushed writes are durable. HDFS is configured with replicated metadata and restore enabled. HDFS and ZooKeeper workloads create a new directory hierarchy; the checker tests that files created before the crash exist. In ZooKeeper, the checker also verifies that quota and ACL modifications are consistent.

If ALICE finds a vulnerability related to a system call, it does not search for other vulnerabilities related to the same call. If the system call is involved in multiple, logically separate vulnerabilities, this has the effect of hiding some of the vulnerabilities. Most tested applications, however, have distinct, independent sets of failures (e.g., *dirstate* and *repository* corruption in Mercurial, consistency and durability violation in other applications). We use different checkers for each type of failure, and report vulnerabilities for each checker separately.

Summary. If application invariants for the tested configuration are explicitly and conspicuously documented, we consider violating those invariants as failure; otherwise, our checkers consider violating a lay user's expectations as failure. We are careful about any recovery procedures that need to be followed on a system crash. Space constraints here limit exact descriptions of the checkers; we provide more details in our webpage [2].

4.2 Overview

We now discuss the logical protocols of the applications examined. Figure 4 visually represents the update protocols, showing the logical operations in the protocol (organized as modules) and discovered vulnerabilities.

4.2.1 Databases and Key-Value Stores

Most databases use a variant of write-ahead logging. First, the new data is written to a log. Then, the log is checkpointed or compacted, i.e., the actual database is usually overwritten, and the log is deleted.

Figure 4(A) shows the protocol used by LevelDB-1.15. LevelDB adds inserted key-value pairs to the log until it reaches a threshold, and then switches to a new

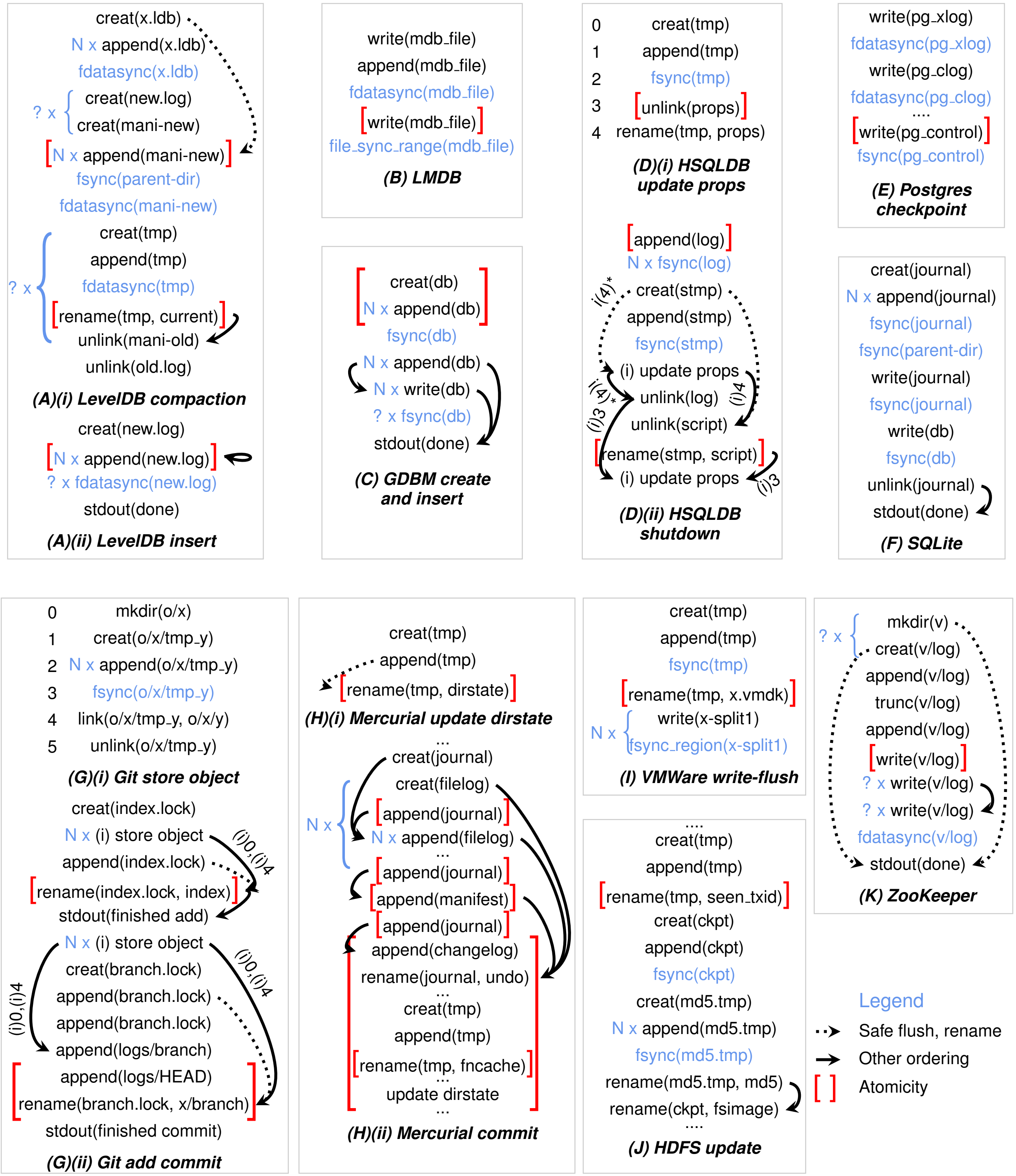

Figure 4: **Protocol Diagrams.** *The diagram shows the modularized update protocol for all applications. For applications with more than one configuration (or versions), only a single configuration is shown (SQLite: Rollback, LevelDB: 1.15). Uninteresting parts of the protocol and a few vulnerabilities (similar to those already shown) are omitted. Repeated operations in a protocol are shown as 'N ×' next to the operation, and portions of the protocol executed conditionally are shown as '? ×'. Blue-colored text simply highlights such annotations and sync calls. Ordering and durability dependencies are indicated with arrows, and dependencies between modules are indicated by the numbers on the arrows, corresponding to line numbers in modules. Durability dependency arrows end in an* stdout *micro-op; additionally, the two dependencies marked with * in HSQLDB are also durability dependencies. Dotted arrows correspond to safe* rename *or* safe file flush *vulnerabilities discussed in Section 4.4. Operations inside brackets must be persisted together atomically.*

log; during the switch, a background thread starts compacting the old log file. Figure 4(A)(i) shows the compaction; Figure 4(A)(ii) shows the appends to the log file. During compaction, LevelDB first writes data to a new *ldb* file, updates pointers to point to the new file (by appending to a *manifest*), and then deletes the old log file.

In LevelDB, we find vulnerabilities occurring while appending to the log file. A crash can result in the appended portion of the file containing garbage; LevelDB's recovery code does not properly handle this situation, and the user gets an error if trying to access the inserted key-value pair (which should not exist in the database). We also find some vulnerabilities occurring during compaction. For example, LevelDB does not explicitly persist the directory entries of *ldb* files; a crash might cause the files to vanish, resulting in unavailability.

Some databases follow protocols that are radically different from write-ahead logging. For example, LMDB uses shadow-paging (copy-on-write). LMDB requires that the final pointer update (106 bytes) in the copy-on-write tree to be atomic. HSQLDB uses a combination of write-ahead logging and update-via-rename, on the same files, to maintain consistency. The update-via-rename is performed by first separately unlinking the destination file, and then renaming; out-of-order persistence of `rename()`, `unlink()`, or log creation causes problems.

4.2.2 Version Control Systems

Git and Mercurial maintain meta-information about their repository in the form of logs. The Git protocol is illustrated in Figure 4(G). Git stores information in the form of object files, which are never modified; they are created as temporary files, and then linked to their permanent file names. Git also maintains pointers in separate files, which point to both the meta-information log and the object files, and are updated using update-via-rename. Mercurial, on the other hand, uses a journal to maintain consistency, using update-via-rename only for a few unimportant pieces of information.

We find many ordering dependencies in the Git protocol, as shown in Figure 4(G). This result is not surprising, since mailing-list discussions suggest Git developers expect total ordering from the file system. We also find a Git vulnerability involving atomicity across multiple system calls; a pointer file being updated (via an append) has to be persisted atomically with another file getting updated (via an update-via-rename). In Mercurial, we find many ordering vulnerabilities for the same reason, not being designed to tolerate out-of-order persistence.

4.2.3 Virtualization and Distributed Systems

VMWare Player's protocol is simple. VMWare maintains a static, constant mapping between blocks in the virtual disk, and in the `VMDK` file (even for dynamically allocated `VMDK` files); directly overwriting the `VMDK` file maintains consistency (though VMWare does use update-via-rename for some small files). Both HDFS and ZooKeeper use write-ahead logging. Figure 4(K) shows the ZooKeeper logging module. We find that ZooKeeper does not explicitly persist directory entries of log files, which can lead to lost data. ZooKeeper also requires some log writes to be atomic.

4.3 Vulnerabilities Found

ALICE finds 60 static vulnerabilities in total, corresponding to 156 dynamic vulnerabilities. Altogether, applications failed in more than 4000 crash states. Table 3(a) shows the vulnerabilities classified by the affected persistence property, and 3(b) shows the vulnerabilities classified by failure consequence. Table 3(b) also separates out those vulnerabilities related only to user expectations and not to documented guarantees, with an asterik (*); many of these correspond to applications for which we could not find any documentation of guarantees.

The different journal-mode configurations provided by SQLite use different protocols, and the different versions of LevelDB differ on whether their protocols are designed around the `mmap()` interface. Tables 3(a) and 3(b) hence show these configurations of SQLite and LevelDB separately. All other configurations (in all applications) do not change the basic protocol, but vary on the application invariants; among different configurations of the same update protocol, all vulnerabilities are revealed in the *safest* configuration. Table 3 and the rest of the paper only show vulnerabilities we find in the safest configuration, i.e., we do not count separately the same vulnerabilities from different configurations of the same protocol.

We find many vulnerabilities have severe consequences such as silent errors or data loss. Seven applications are affected by data loss, while two (both LevelDB versions and HSQLDB) are affected by silent errors. The *cannot open* failures include failure to start the server in HDFS and ZooKeeper, while the *failed reads and writes* include basic commands (e.g., `git-log`, `git-commit`) failing in Git and Mercurial. A few *cannot open* failures and *failed reads and writes* might be solvable by application experts, but we believe lay users would have difficulty recovering from such failures (our checkers invoke standard recovery techniques). We also checked if any discovered vulnerabilities are previously known, or considered inconsequential. The single PostgreSQL vulnerability is documented; it can be solved with non-standard (although simple) recovery techniques. The single LMDB vulnerability is discussed in a mailing list, though there is no available workaround. All these previously known vulnerabilities are separated out in Table 3(b) (†). The five *dirstate fail* vulnerabilities in Mercurial are shown separately, since they are less harmful than other vulnerabilities (though frustrating to the lay

Application	Types									Unique static vulnerabilities
	Across-syscalls atomicity	Atomicity			Ordering			Durability		
		Appends and truncates	Single-block overwrites	Renames and unlinks	Safe file flush	Safe renames	Other	Safe file flush	Other	
Leveldb1.10	1‡	1		1	2	1	3	1		**10**
Leveldb1.15	1	1		1	1		2			**6**
LMDB			1							**1**
GDBM	1			1			1		2	**5**
HSQLDB		1		2	1		3	2	1	**10**
Sqlite-Roll									1	**1**
Sqlite-WAL										**0**
PostgreSQL			1							**1**
Git	1			1	2	1	3		1	**9**
Mercurial	2	1		1		1	4		2	**10**
VMWare				1						**1**
HDFS				1			1			**2**
ZooKeeper			1				1	2		**4**
Total	**6**	**4**	**3**	**9**	**6**	**3**	**18**	**5**	**7**	**60**

(a) Types.

Application	Silent errors	Data loss	Cannot open	Failed reads and writes	Other
Leveldb1.10	1	1	5	4	
Leveldb1.15	2		2	2	
LMDB					read-only open†
GDBM		2*	3*		
HSQLDB	2	3	5		
Sqlite-Roll		1*			
Sqlite-WAL					
PostgreSQL			1†		
Git		1*	3*	5*	3#*
Mercurial		2*	1*	6*	5 dirstate fail*
VMWare			1*		
HDFS			2*		
ZooKeeper		2*	2*		
Total	**5**	**12**	**25**	**17**	**9**

(b) Failure Consequences.

Application	ext3-w	ext3-o	ext3-j	ext4-o	btrfs
Leveldb1.10	3	1	1	2	4
Leveldb1.15	2	1	1	2	3
LMDB					
GDBM	3	3	2	3	4
HSQLDB					4
Sqlite-Roll	1	1	1	1	1
Sqlite-WAL					
PostgreSQL					
Git	2	2	2	2	5
Mercurial	4	3	3	6	8
VMWare					
HDFS					1
ZooKeeper	1	1		1	1
Total	16	12	10	17	31

(c) Under Current File Systems.

	Ordering	DO	AG	CA
ext3-w	Dir ops and file-sizes ordered among themselves, before sync operations.	✓	4K	×
ext3-o	Dir ops, appends, truncates ordered among themselves. Overwrites before non-overwrites, all before sync.	✓	4K	✓
ext3-j	All operations are ordered.	✓	4K	✓
ext4-o	Safe rename, safe file flush, dir ops ordered among themselves	✓	4K	✓
btrfs	Safe rename, safe file flush	✓	4K	✓

(d) APMs considered.

Table 3: **Vulnerabilities.** *(a) shows the discovered static vulnerabilities categorized by the type of persistence property. The number of unique vulnerabilities for an application can be different from the sum of the categorized vulnerabilities, since the same source code lines can exhibit different behavior. ‡ The atomicity vulnerability in Leveldb1.10 corresponds to multiple mmap() writes. (b) shows the number of static vulnerabilities resulting in each type of failure. † Previously known failures, documented or discussed in mailing lists. * Vulnerabilities relating to unclear documentation or typical user expectations beyond application guarantees. # There are 2 fsck-only and 1 reflog-only errors in Git. (c) shows the number of vulnerabilities that occur on current file systems (all applications are vulnerable under future file systems). (d) shows APMs used for calculating Table (c). Legend: DO: directory operations atomicity. AG: granularity of size-atomicity. CA: Content-Atomicity.*

user). Git's *fsck-only* and *reflog-only* errors are potentially dangerous, but do not seem to affect normal usage.

We interacted with the developers of eight applications, reporting a subset of the vulnerabilities we find. Our interactions convince us that the vulnerabilities will affect users if they are exposed. The other applications (GDBM, Git, and Mercurial) were not designed to provide crash guarantees, although we believe their users will be affected by the vulnerabilities found should an untimely crash occur. Thus, the vulnerabilities will not surprise a developer of these applications, and we did not report them. We also did not report vulnerabilities concerning partial renames (usually dismissed since they are not commonly exposed), or documented vulnerabilities.

Developers have acted on five of the vulnerabilities we find: one (LevelDB-1.10) is now fixed, another (LevelDB-1.15) was fixed parallel to our discovery, and three (HDFS, and two in ZooKeeper) are under consideration. We have found that developers often dismiss other vulnerabilities which do not (or are widely believed to not) get exposed in current file systems, especially relating to out-of-order persistence of directory operations. The fact that only certain operating systems allow an `fsync()` on a directory is frequently referred to; both HDFS and ZooKeeper respondents lament that such an `fsync()` is not easily achievable with Java. The developers suggest the SQLite vulnerability is actually not a behavior guaranteed by SQLite (specifically, that durability cannot be achieved under `rollback` journaling); we believe the documentation is misleading.

Of the five acted-on vulnerabilities, three relate to not explicitly issuing an `fsync()` on the parent directory after creating and calling `fsync()` on a file. However, not issuing such an `fsync()` is perhaps more safe in modern file systems than out-of-order persistence of directory operations. We believe the developers' interest in fixing this problem arises from the Linux documentation explicitly recommending an `fsync()` after creating a file.

Summary. ALICE detects 60 vulnerabilities in total, with 5 resulting in silent failures, 12 in loss of durability, 25 leading to inaccessible applications, and 17 returning errors while accessing certain data. ALICE is also able to detect previously known vulnerabilities.

4.4 Common Patterns

We now examine vulnerabilities related with different persistence properties. Since durability vulnerabilities show a separate pattern, we consider them separately.

4.4.1 Atomicity across System Calls

Four applications (including both versions of LevelDB) require atomicity across system calls. For three applications, the consequences seem minor: inaccessibility during database creation in GDBM, dirstate corruption in Mercurial, and an erratic `reflog` in Git. LevelDB's vulnerability has a non-minor consequence, but was fixed immediately after introducing LevelDB-1.15 (when LevelDB started using read()-write() instead of mmap()).

In general, we observe that this class of vulnerabilities seems to affect applications less than other classes. This result may arise because these vulnerabilities are easily tested: they are exposed independent of the file system (i.e, via process crashes), and are easier to reproduce.

4.4.2 Atomicity within System Calls

Append atomicity. Surprisingly, three applications require appends to be *content-atomic*: the appended portion should contain actual data. The failure consequences are severe, such as corrupted reads (HSQLDB), failed reads (LevelDB-1.15) and repository corruption (Mercurial). Filling the appended portion with zeros instead of garbage still causes failure; only the current implementation of delayed allocation (where file size does not increase until actual content is persisted) works. Most appends seemingly do not need to be *block-atomic*; only Mercurial is affected, and the affected append also requires content-atomicity.

Overwrite atomicity. LMDB, PostgreSQL, and ZooKeeper require small writes ($<$ 200 bytes) to be atomic. Both the LMDB and PostgreSQL vulnerabilities are previously known.

We do not find any multi-block overwrite vulnerabilities, and even single-block overwrite requirements are typically documented. This finding is in stark contrast with append atomicity; some of the difference can be attributed to the default APM (overwrites are content-atomic), and to some workloads simply not using overwrites. However, the major cause seems to be the basic mechanism behind application update protocols: modifications are first logged, in some form, via appends; logged data is then used to overwrite the actual data. Applications have careful mechanisms to detect and repair failures in the actual data, but overlook the presence of garbage content in the log.

Directory operation atomicity. Given that most file systems provide atomic directory operations (§2.2), one would expect that most applications would be vulnerable to such operations not being atomic. However, we do not find this to be the case for certain classes of applications. Databases and key-value stores do not employ atomic renames extensively; consequently, we observe non-atomic renames affecting only three of these applications (GDBM, HSQLDB, LevelDB). Non-atomic unlinks seemingly affect only HSQLDB (which uses unlinks for logically performing renames), and we did not find any application affected by non-atomic truncates.

4.4.3 Ordering between System Calls

Applications are extremely vulnerable to system calls being persisted out of order; we find 27 vulnerabilities.

Safe renames. On file systems with delayed allocation, a common heuristic to prevent data loss is to persist all data (including appends and truncates) of a file before subsequent renames of the file [14]. We find that this heuristic only matches (and thus fixes) three discovered vulnerabilities, one each in Git, Mercurial, and LevelDB-1.10. A related heuristic, where updating *existing* files by opening them with O_TRUNC flushes the updated data while issuing a close(), does not affect any of the vulnerabilities we discovered. Also, the effect of the heuristics varies with minor details: if the safe-rename heuristic does not persist file truncates, only two vulnerabilities will be fixed; if the O_TRUNC heuristic also acts on new files, an additional vulnerability will be fixed.

Safe file flush. An fsync() on a file does not guarantee that the file's directory entry is also persisted. Most file systems, however, persist directory entries that the file is dependent on (e,g., directory entries of the file and its parent). We found that this behavior is required by three applications for maintaining basic consistency.

4.4.4 Durability

We find vulnerabilities in seven applications resulting in durability loss. Of these, only two applications (GDBM and Mercurial) are affected because an fsync() is not called on a file. Six applications require fsync() calls on directories: three are affected by *safe file flush* discussed previously, while four (HSQLDB, SQLite, Git, and Mercurial) require other fsync() calls on directories. As a special case, with HSQLDB, previously committed data is lost, rather than data that was being committed during the time of the workload. In all, only four out of the twelve vulnerabilities are exposed when full ordering is promised: many applications do issue an fsync() call before durability is essential, but do not fsync() all the required information.

4.4.5 Summary

We believe our study offers several insights for file-system designers. Future file systems should consider providing ordering between system calls, and atomicity within a system call in specific cases. Vulnerabilities involving atomicity of multiple system calls seem to have minor consequences. Requiring applications to separately flush the directory entry of a created and flushed file can often result in application failures. For durability, most applications seem to explicitly flush some, but not all, of the required information; thus, providing ordering among system calls can also help durability.

4.5 Impact on Current File Systems

Our study thus far has utilized an abstract (and weak) file system model (i.e., APM) in order to discover the broadest number of vulnerabilities. We now utilize specific file-system APMs to understand how modern protocols would function atop a range of modern file systems and configurations. Specifically, we focus on Linux ext3 (including writeback, ordered, and data-journaling mode), Linux ext4, and btrfs. The considered APMs are based on our understanding of file systems from Section 2.

Table 3(c) shows the vulnerabilities reported by ALICE, while 3(d) shows the considered APMs. We make a number of observations based on Table 3(c). First, a significant number of vulnerabilities are exposed on *all* examined file systems. Second, ext3 with journaled data is the safest: the only vulnerabilities exposed relate to atomicity across system calls, and a few durability vulnerabilities. Third, a large number of vulnerabilities are exposed on btrfs as it aggressively persists operations out of order [11]. Fourth, some applications show no vulnerabilities on any considered APM; thus, the flaws we found in such applications do not manifest on today's file systems (but may do so on future systems).

Summary. Application vulnerabilities are exposed on many current file systems. The vulnerabilities exposed vary based on the file system, and thus testing applications on only a few file systems does not work.

4.6 Evaluating New File-System Designs

File-system modifications for improving performance have introduced wide-spread data loss in the past [14], because of changes to the file-system persistence properties. ALICE can be used to test whether such modifications break correctness of existing applications. We now describe how we use ALICE to evaluate a hypothetical variant of ext3 (data-journaling mode), *ext3-fast*.

Our study shows that ext3 (data-journaling mode) is the safest file system; however, it offers poor performance for many workloads [35]. Specifically, `fsync()` latency is extremely high as ext3 persists *all* previous operations on `fsync()`. One way to reduce `fsync()` latency would be to modify ext3 to persist only the synced file. However, other file systems (e.g,. btrfs) that have attempted to reduce `fsync()` latency [13] have resulted in increased vulnerabilities. Our study suggests a way to reduce latency *without* exposing more vulnerabilities.

Based on our study, we hypothesize that data that is not *synced* need not be persisted before explicitly synced data for correctness; such data must only be persisted in-order amongst itself. We design *ext3-fast* to reflect this: `fsync()` on a file A persists only A, while other dirty data and files are still persisted in-order.

We modeled *ext3-fast* in ALICE by slightly changing the APM of ext3 data journaling mode, so that synced directories, files, and their data, depend only on previous syncs and operations necessary for the file to exist (i.e., safe file flush is obeyed). The operations on a synced file are also ordered among themselves.

We test our hypothesis with ALICE; the observed ordering vulnerabilities (of the studied applications) are not exposed under *ext3-fast*. The design was not meant to fix durability or atomicity across system calls vulnerabilities, so those vulnerabilities are still reported by ALICE.

We estimate the performance gain of *ext3-fast* using the following experiment: we first write 250 MB to file A, then append a byte to file B and `fsync()` B. When both files are on the same ext3-ordered file system, `fsync()` takes about four seconds. If the files belong to different partitions on the same disk, mimicking the behavior of *ext3-fast*, the `fsync()` takes only 40 ms. The first case is 100 times slower because 250 MB of data is ordered before the single byte that needs to be persistent.

Summary. The *ext3-fast* file system (derived from inferences provided by ALICE) seems interesting for application safety, though further investigation is required into the validity of its design. We believe that the ease of use offered by ALICE will allow it to be incorporated into the design process of new file systems.

4.7 Discussion

We now consider why crash vulnerabilities occur commonly even among widely used applications. We find that application update protocols are complex and hard to isolate and understand. Many protocols are layered and spread over multiple files. Modules are also associated with other complex functionality (e.g., ensuring thread isolation). This complexity leads to issues that are obvious with a bird's eye view of the protocol: for example, HSQLDB's protocol has 3 consecutive `fsync()` calls to the same file (increasing latency). ALICE helps solve this problem by making it easy to obtain logical representations of update protocols as shown in Figure 4.

Another factor contributing to crash vulnerabilities is poorly written, untested recovery code. In LevelDB, we find vulnerabilities that should be prevented by correct implementations of the documented update protocols. Some recovery code is non-optimal: potentially recoverable data is lost in several applications (e.g., HSQLDB, Git). Mercurial and LevelDB provide utilities to verify or recover application data; we find these utilities hard to configure and error-prone. For example, an user invoking LevelDB's recovery command can unintentionally end up *further* corrupting the datastore, and be affected by (seemingly) unrelated configuration options (*paranoid checksums*). We believe these problems are a direct consequence of the recovery code being infrequently executed and insufficiently tested. With ALICE, recovery code can be tested on many corner cases.

Convincing developers about crash vulnerabilities is sometimes hard: there is a general mistrust surrounding such bug reports. Usually, developers are suspicious that the underlying storage stack might not respect `fsync()` calls [36], or that the drive might be corrupt. We hence believe that most vulnerabilities that occur in the wild are associated with an incorrect root cause, or go unreported. ALICE can be used to easily reproduce vulnerabilities.

Unclear documentation of application guarantees contributes to the confusion about crash vulnerabilities. During discussions with developers about durability vulnerabilities, we found that SQLite, which proclaims itself as fully ACID-complaint, does not provide durability (even optionally) with the default storage engine, though the documentation suggests it does. Similarly, GDBM's GDBM_SYNC flag *does not* ensure durability. Users can employ ALICE to determine guarantees directly from the code, bypassing the problem of bad documentation.

5 Related Work

Our previous workshop paper [47] identifies the problem of application-level consistency depending upon file-system behavior, but is limited to two applications and does not use automated testing frameworks. Since we use ALICE to obtain results, our current study includes a greater number and variety of applications.

This paper adapts ideas from past work on dynamic program analysis and model checking. EXPLODE [53] has a similar flavor to our work: the authors use *in-situ* model checking to find crash vulnerabilities on different storage stacks. ALICE differs from EXPLODE in four significant ways. First, EXPLODE requires the target storage stack to be fully implemented; ALICE only requires a model of the target storage stack, and can therefore be used to evaluate application-level consistency on top of proposed storage stacks, while they are still at the design stage. Second, EXPLODE requires the user to carefully annotate complex file systems using *choose()* calls; ALICE requires the user to only specify a high-level APM. Third, EXPLODE reconstructs crash states by tracking I/O as it moves from the application to the storage. Although it is possible to use EXPLODE to determine the root cause of a vulnerability, we believe it is easier to do so using ALICE since ALICE checks for violation of specific persistence properties. Fourth, EXPLODE stops at finding crash vulnerabilities; by helping produce protocol diagrams, ALICE contributes to understanding the protocol itself. Like BOB, EXPLODE can be used to test persistence properties; however, while BOB only re-orders block I/O, EXPLODE can test re-orderings caused at different layers in the storage stack.

Zheng et al. [54] find crash vulnerabilities in databases. They contribute a standard set of workloads that stress databases (particularly, with multiple threads), and check ACID properties; the workloads and checkers can be used with ALICE. Unlike our work, Zheng et al. do not systematically explore vulnerabilities of each system call; they are limited by the re-orderings and non-atomicity exhibited by a particular (implemented) file system during a single workload execution. Thus, their work is more suited for finding those vulnerabilities that are *commonly* exposed under a given file system.

Woodpecker [15] can be used to find crash vulnerabilities when supplied with suspicious source code patterns to guide symbolic execution. Our work is fundamentally different to this approach, as ALICE does not require prior knowledge of patterns in checked applications.

Our work is influenced by SQLite's internal testing tool [43]. The tool works at an internal wrapper layer within SQLite, and is not helpful for generic testing.

RACEPRO [25], a testing tool for concurrency bugs, records system calls and replays them by splitting them into small operations, but does not test crash consistency.

OptFS [9], Featherstitch [16], and transactional file systems [17, 39, 42], discuss new file-system interfaces that will affect vulnerabilities. Our study can help inform the design of new interfaces by providing clear insights into what is missing in today's interfaces.

6 Conclusion

In this paper, we show how application-level consistency is dangerously dependent upon file system *persistence properties*, i.e., how file systems persist system calls. We develop BOB, a tool to test persistence properties and show that such properties vary widely among file systems. We build ALICE, a framework that analyzes application-level protocols and detects crash vulnerabilities. We analyze 11 applications, and find 60 vulnerabilities, some of which result in severe consequences like corruption or data loss. We present several insights derived from our study. The ALICE tool, and detailed descriptions of the vulnerabilities found in our study, can be obtained from our webpage [2].

Acknowledgments

We thank Lorenzo Alvisi (our shepherd) and the anonymous reviewers for their insightful comments. We thank members of ADSL, application developers and users, and file system developers, for valuable discussions. This material is based upon work supported by the NSF under CNS-1421033, CNS-1319405, and CNS-1218405 as well as donations from EMC, Facebook, Fusion-io, Google, Huawei, Microsoft, NetApp, Samsung, Sony, and VMware. Vijay Chidambaram and Samer Al-Kiswany are supported by the Microsoft Research PhD Fellowship and the NSERC Postdoctoral Fellowship, respectively. Any opinions, findings, and conclusions, or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the NSF or other institutions.

References

[1] Necessary step(s) to synchronize filename operations on disk. http://austingroupbugs.net/view.php?id=672.

[2] Tool and Results: Application Crash Vulnerabilities. http://research.cs.wisc.edu/adsl/Software/alice/.

[3] Apache. Apache Zookeeper. http://zookeeper.apache.org/.

[4] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 0.8 edition, 2014.

[5] Steve Best. JFS Overview. http://jfs.sourceforge.net/project/pub/jfs.pdf, 2000.

[6] Remy Card, Theodore Ts'o, and Stephen Tweedie. Design and Implementation of the Second Extended Filesystem. In *First Dutch International Symposium on Linux*, Amsterdam, Netherlands, December 1994.

[7] Donald D Chamberlin, Morton M Astrahan, Michael W Blasgen, James N Gray, W Frank King, Bruce G Lindsay, Raymond Lorie, James W Mehl, Thomas G Price, Franco Putzolu, et al. A history and evaluation of system r. *Communications of the ACM*, 24(10):632–646, 1981.

[8] Donald D Chamberlin, Arthur M Gilbert, and Robert A Yost. A history of system r and sql/data system. In *VLDB*, pages 456–464, 1981.

[9] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Optimistic Crash Consistency. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, Nemacolin Woodlands Resort, Farmington, Pennsylvania, October 2013.

[10] Vijay Chidambaram, Tushar Sharma, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Consistency Without Ordering. In *Proceedings of the 10th USENIX Symposium on File and Storage Technologies (FAST '12)*, pages 101–116, San Jose, California, February 2012.

[11] Chris Mason. Btrfs Mailing List. Re: Ordering of directory operations maintained across system crashes in Btrfs? http://www.spinics.net/lists/linux-btrfs/msg32215.html, 2014.

[12] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better I/O Through Byte-addressable, Persistent Memory. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09)*, Big Sky, Montana, October 2009.

[13] Jonathan Corbet. Solving the Ext3 Latency Problem. http://lwn.net/Articles/328363/, 2009.

[14] Jonathan Corbet. That massive filesystem thread. http://lwn.net/Articles/326471/, March 2009.

[15] Heming Cui, Gang Hu, Jingyue Wu, and Junfeng Yang. Verifying systems rules using rule-directed symbolic execution. In *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*, pages 329–342. ACM, 2013.

[16] Christopher Frost, Mike Mammarella, Eddie Kohler, Andrew de los Reyes, Shant Hovsepian, Andrew Matsuoka, and Lei Zhang. Generalized File System Dependencies. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP '07)*, pages 307–320, Stevenson, Washington, October 2007.

[17] Bill Gallagher, Dean Jacobs, and Anno Langen. A High-performance, Transactional Filestore for Application Servers. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data (SIGMOD '05)*, pages 868–872, Baltimore, Maryland, June 2005.

[18] Gregory R. Ganger and Yale N. Patt. Metadata Update Performance in File Systems. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation (OSDI '94)*, pages 49–60, Monterey, California, November 1994.

[19] GNU. GNU Database Manager (GDBM). http://www.gnu.org.ua/software/gdbm/gdbm.html, 1979.

[20] Google. LevelDB. https://code.google.com/p/leveldb/, 2011.

[21] Robert Hagmann. Reimplementing the Cedar File System Using Logging and Group Commit. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles (SOSP '87)*, Austin, Texas, November 1987.

[22] Dave Hitz, James Lau, and Michael Malcolm. File System Design for an NFS File Server Appliance. In *Proceedings of the USENIX Winter Technical Conference (USENIX Winter '94)*, San Francisco, California, January 1994.

[23] HyperSQL. HSQLDB. http://www.hsqldb.org/.

[24] Dean Kuo. Model and verification of a data manager based on aries. *ACM Trans. Database Syst.*, 21(4):427–479, December 1996.

[25] Oren Laadan, Nicolas Viennot, Chia-Che Tsai, Chris Blinn, Junfeng Yang, , and Jason Nieh. Pervasive Detection of Process Races in Deployed Systems. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, Cascais, Portugal, October 2011.

[26] Linus Torvalds. Git. http://git-scm.com/, 2005.

[27] Linus Torvalds. Git Mailing List. Re: what's the current wisdom on git over NFS/CIFS? http://marc.info/?l=git&m=124839484917965&w=2, 2009.

[28] Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Shan Lu. A Study of Linux File System Evolution. In *Proceedings of the 11th USENIX Symposium on File and Storage Technologies (FAST '13)*, San Jose, California, February 2013.

[29] Cris Pedregal Martin and Krithi Ramamritham. Toward formalizing recovery of (advanced) transactions. In *Advanced Transaction Models and Architectures*, pages 213–234. Springer, 1997.

[30] Chris Mason. The Btrfs Filesystem. oss.oracle.com/projects/btrfs/dist/documentation/btrfs-ukuug.pdf, September 2007.

[31] Matt Mackall. Mercurial. http://mercurial.selenic.com/, 2005.

[32] Marshall K. McKusick, William N. Joy, Sam J. Leffler, and Robert S. Fabry. A Fast File System for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

[33] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Transactions on Database Systems*, 17(1):94–162, March 1992.

[34] C Mohan, Bruce Lindsay, and Ron Obermarck. Transaction management in the r* distributed database management system. *ACM Transactions on Database Systems (TODS)*, 11(4):378–396, 1986.

[35] Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Analysis and Evolution of Journaling File Systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX '05)*, pages 105–120, Anaheim, California, April 2005.

[36] Abhishek Rajimwale, Vijay Chidambaram, Deepak Ramamurthi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Coerced Cache Eviction and Discreet-Mode Journaling: Dealing with Misbehaving Disks. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN '11)*, Hong Kong, China, June 2011.

[37] Hans Reiser. ReiserFS. www.namesys.com, 2004.

[38] Mendel Rosenblum and John Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10(1):26–52, February 1992.

[39] Frank Schmuck and Jim Wylie. Experience with transactions in quicksilver. In *ACM SIGOPS Operating Systems Review*, volume 25, pages 239–253. ACM, 1991.

[40] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *Proceedings of the 26th IEEE Symposium on Mass Storage Systems and Technologies (MSST '10)*, Incline Village, Nevada, May 2010.

[41] Stewart Smith. Eat My Data: How everybody gets file I/O wrong. In *OSCON*, Portland, Oregon, July 2008.

[42] R. P. Spillane, S. Gaikwad, E. Zadok, C. P. Wright, and M. Chinni. Enabling transactional file access via lightweight kernel extensions. In *Proceedings of the Seventh USENIX Conference on File and Storage Technologies (FAST '09)*, pages 29–42, San Francisco, CA, February 2009. USENIX Association.

[43] SQLite. SQLite transactional SQL database engine. http://www.sqlite.org/.

[44] Sun Microsystems. ZFS: The last word in file systems. www.sun.com/2004-0914/feature/, 2006.

[45] Adan Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the XFS File System. In *Proceedings of the USENIX Annual Technical Conference (USENIX '96)*, San Diego, California, January 1996.

[46] Symas. Lightning Memory-Mapped Database (LMDB). `http://symas.com/mdb/`, 2011.

[47] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Jooyoung Hwang, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau. Towards Efficient, Portable Application-Level Consistency. In *Proceedings of the 9th Workshop on Hot Topics in Dependable Systems (HotDep '13)*, Farmington, PA, November 2013.

[48] The Open Group. POSIX.1-2008 IEEE Std 1003.1. `http://pubs.opengroup.org/onlinepubs/9699919799/`, 2013.

[49] The PostgreSQL Global Development Group. PostgreSQL. `http://www.postgresql.org/`.

[50] Theodore Ts'o and Stephen Tweedie. Future Directions for the Ext2/3 Filesystem. In *Proceedings of the USENIX Annual Technical Conference (FREENIX Track)*, Monterey, California, June 2002.

[51] Stephen C. Tweedie. Journaling the Linux ext2fs File System. In *The Fourth Annual Linux Expo*, Durham, North Carolina, May 1998.

[52] VMWare. VMWare Player. `http://www.vmware.com/products/player`.

[53] Junfeng Yang, Can Sar, and Dawson Engler. EXPLODE: A Lightweight, General System for Finding Serious Storage System Errors. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, Seattle, Washington, November 2006.

[54] Mai Zheng, Joseph Tucek, Dachuan Huang, Feng Qin, Mark Lillibridge, Elizabeth S Yang, Bill W Zhao, and Shashank Singh. Torturing Databases for Fun and Profit. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*, Broomfield, CO, October 2014.