Uncovering Twilio: Insights into Cloud Communication Services

Ramnatthan Alagappan
ra@cs.wisc.edu

Sourav Das
souravd@cs.wisc.edu

Abstract

We study the protocols and architecture of a popular cloud communication service *Twilio* through gray box methods. We develop a simple VoIP service atop *Twilio* for our study. We provide insights into how different components in a cloud communication service interact with each other in various scenarios. We also measure some guarantees with respect to call and message dequeuing provided by Twilio. Our analysis reveals a number of interesting aspects about the Twilio ecosystem and have strong implications for developers who build applications on top of Twilio APIs.

1 Introduction

Crash recovery is a fundamental problem in systems research [**?**, **?**, **?**, **?**], 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 [**?**, **?**] and DB2 [**?**]) and the development of a working crash recovery algorithm (ARIES [**?**]). Even after ARIES was invented, another five years passed before the algorithm was proven correct [**?**, **?**].

The file systems community has developed a standard set of techniques to provide file-system metadata consistency in the face of crashes: logging [**?**, **?**, **?**, **?**, **?**, **?**], copy-on-write [**?**, **?**, **?**, **?**], soft updates [**?**], and other similar approaches [**?**, **?**]. While bugs remain in the file systems that implement these methods [**?**], the core techniques are heavily tested and well understood.

Many important applications, including databases such as SQLite [**?**] and key-value stores such as LevelDB [**?**], 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 [**?**], 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, it does not fully define how disk state is mutated in the event of a crash. 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 [**?**]. 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-level consistency would be relatively simple (though not trivial) if all state was 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 [**?**, **?**]. 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*. They break down into two global categories: the *atomicity* of operations (e.g., does the file system ensure that `rename()` is atomic [**?**]?), 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 develop 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 legal on-disk crash states. Through this simple approach, BOB can reveal which persistence properties do *not* hold for a given system. We use BOB to study six file systems (Linux 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; ext4 ordered mode has the desired behavior if the right option is enabled.

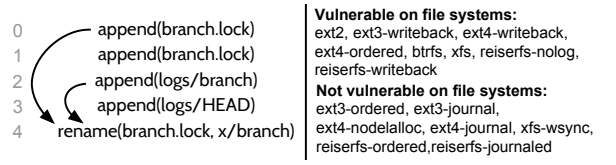

Figure 1: **Git Crash Vulnerability.** *The figure shows part of the Git update protocol. The arrows represent ordering dependencies: if the appends are not persisted before the rename, any further commits to the repository fail. We observe that whether the protocol is vulnerable or not varies even between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

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 complicated [**?**] and spread out over multiple files. To analyze applications, we develop ALICE, a novel framework that enables us to systematically study application-level crash consistency. ALICE takes advantage of the fact that, no matter how complex an application's 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 a example of ALICE in action. The figure shows a part of the update protocol of Git [**?**]. ALICE detected that the appends need to be persisted before the rename; if not, any future commits to the repository fail. A number of file-system features such as delayed allocation and journaling mode determine whether file systems exhibit this behavior. Dangerously, 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 twelve important applications: BerkeleyDB [**?**], LevelDB [**?**], GDBM [**?**], LMDB [**?**], SQLite [**?**], PostgreSQL [**?**], HSQLDB [**?**], Git [**?**], Mercurial [**?**]), HDFS [**?**], ZooKeeper [**?**], and VMWare Player [**?**]. These applications represent software from different domains and at varying levels of maturity.

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 an application's correctness depends on a specific file system's 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 65 vulnerabilities across the applications we studied; several vulnerabilities have severe consequences such as data corruption or application unavailability.

Our study of update protocols and vulnerabilities sheds new light on how applications across domains maintain crash-consistency, and provides a number of interesting insights for file-system developers. For example, we found that many applications use a variant of write-ahead-logging; while applications are very careful about updating data in-place, they are not as careful about logged data, resulting in a number of vulnerabilities. We identify the exact persistence property, *content-atomicity of appends*, that file systems could provide to help applications. Application data loss is often blamed on developers not using `fsync()` [**?**]: on the contrary, we find that buggy recovery code and erroneous, out-of-date documentation play a bigger role in data loss.

ALICE also enables us to determine whether new file-system designs will harm current applications. For example, file-system developers strive to reduce `fsync()` latency while not affecting application correctness; our study suggests one way to achieve this by modifying ext3 (although persistence properties are changed). We used ALICE to model the modified ext3 file system, and verified that the modification did not affect application correctness. 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 [**?**]. With ALICE, testing the impact of changing persistence properties can become part of the file-system design process.

The rest of this paper is structured as follows. We begin by describing persistence properties and showing how they vary across different file systems (§2). We explain the design and implementation of ALICE (§3) and present the results of analyzing application update protocols using ALICE (§4). We discuss factors contributing to the prevalence of crash vulnerabilities (§5), discuss related work (§6), and finally conclude (§7).

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 a simple tool we've built, known as the *Block Order Breaker (*BOB*)*, that explores possible on-disk states by reordering the I/O block stream and examining possible resulting states. BOB is not a complete tool, but can be used to find persistence properties that do *not* hold for a file system implementation. We then discuss our findings for six widely-used Linux file systems (ext2 [**?**], ext3 [**?**, **?**], ext4 [**?**], btrfs [**?**], xfs [**?**], and reiserfs [**?**]).

Application-level crash consistency depends strongly

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.*

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 the 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 The Block Order Breaker (BOB)

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 [**?**]. Between

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 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.*

file systems and their various configurations, it is challenging to know or reason about which persistence properties are provided. For this reason, 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 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. BOB then re-orders the collected blocks and selectively writes some of them to the initial disk state to generate a new legal disk state. In this manner, BOB generates a number of unique disk images corresponding to possible on-disk states after a system crash. BOB then checks whether the persistence property holds (e.g., if the writes were atomic) on each disk image. 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 (a property holds in all situations) is not possible using BOB.

Note that different system calls (e.g., `write()`, `writev()`) 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.).

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 $\times$.

The size of an overwrite or append affects its atomicity. Hence, we show results for single sector, single block, and multi-block overwrite and append operations. As 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 provide atomic single sector overwrites: in some cases (e.g., ext2), this is 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 atomicity at byte granularity [**?**], 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 do 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 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 [**?**].

The append operation has interesting special cases. On file systems with delayed allocation, it may be persisted after other operations. A special exception to this rule is when a file is appended to, and then renamed. Since this idiom is commonly used to atomically update files [**?**], 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. ext2 and btrfs freely re-order directory operations (especially operations on different directories [**?**]) to increase performance.

Figure 3: **Detecting Vulnerabilities using ALICE.** *The figure shows an overview of how* ALICE *converts user inputs into crash states and finally into crash vulnerabilities. The shaded grey boxes are inputs supplied by the user.*

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 very risky to assume that any particular property will be supported by all file systems.

3 The Application-Level 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 assumptions do they make about the underlying file systems? Are such update protocols correct?

To gain insight into these questions, we develop the *Application-Level Crash Explorer (*ALICE*)*, a framework that analyzes application update protocols and discovers crash vulnerabilities. ALICE detects vulnerabilities by testing whether the application is vulnerable to violations of various persistence properties.

We first describe the mechanism used by ALICE to test a persistence property. We then describe the policy used by ALICE: what persistence properties to test. We discuss how vulnerabilities are reported, a few details of the tool implementation, and finally its limitations.

3.1 Testing a Persistence Property

To test whether the application depends on a particular persistence property (e.g., write atomicity), ALICE *violates* that persistence property (e.g., breaks a write into many parts), and generates the corresponding possible on-disk crash states. ALICE then checks whether application invariants are violated for each crash state.

Violating a single persistence property can lead to many crash states. For example, if append atomicity is violated, a file can end up with either a prefix of the data persisted, with random data intermixed with file data, or various combinations thereof. We use a *file-system persistence model* to define the exact intermediate states possible when persistence properties are violated.

Figure 3 shows an overview of how ALICE finds crash vulnerabilities. For each application to be tested, the user supplies a workload exercising that application, and an invariant verifier to check application invariants. ALICE runs the application and collects the resulting system-call

Logical Operation	Micro-operations
overwrite	$N\times$ write_block(data)
append	$N\times$ change_file_size $N\times$ write_block(random) $N\times$ write_block(data)
truncate	$N\times$ change_file_size $N\times$ write_block(random) $N\times$ write_block(zeroes)
link	create_dir_entry
unlink	delete_dir_entry
delete	change_file_size
rename	create_dir_entry delete_dir_entry
write to terminal	stdout

Table 2: **Translating logical operations into micro-operations.** *The table shows how logical operations such as append are translated into micro-operations. N indicates the logical operation may be broken up into many block-sized micro-operations.*

trace. ALICE first converts system calls into logical operations, then converts these logical operations into *micro-operations* as per the persistence model. ALICE then uses micro-operations to generate crash states. ALICE finally runs the application on each crash state and uses the invariant verifier to check if the application is inconsistent. We now discuss these steps in more detail.

3.1.1 Logical Operations

ALICE first converts the trace of system calls in the application workload to *logical operations*. Logical operations abstract away details such as file descriptors and current read and write offsets, 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 writes to `mmap()`-ed regions are all translated into `overwrite` or `append` logical operations. Logical operations are used to produce the protocol diagrams shown in Section 1 and Section 4.2.

3.1.2 File-System Persistence Model

The *persistence model* defines the atomicity and ordering persistence properties of logical operations, thus defining which crash stats are possible. The persistence model should be sufficiently flexible to model any modern file system. Therefore, the model is constructed based on a minimally POSIX compliant, ext2-like file system, but can be readily changed to mimic modern file systems such as ext4 or btrfs.

The model has three classes of logical entities: *file inodes*, *directory entries*, and *data blocks*. Each logical operation operates on one or more of these entities. An infinite number of instances of each logical entity exist, and they are not allocated or de-allocated by logical operations, but rather simply changed.

To capture intermediate states, the model breaks logical operations further down into *micro-operations*, i.e., the smallest atomic modification that can be performed upon each logical entity. There are four such micro-ops:

```
open(path="/x2VC") = 10
Micro-operations: None
Dependencies: None

pwrite(fd=10, offset=0, size=1024)
Micro-operations:
#1 write_block(inode=12, offset=0, size=512)
#2 write_block(inode=12, offset=512, size=512)
Dependencies: None

fsync(10)
Micro-operations: None
Dependencies: None

pwrite(fd=10, offset=1024, size=1024)
Micro-operations:
#3 write_block(inode=12, offset=1024, size=512)
#4 write_block(inode=12, offset=1536, size=512)
Dependencies: #1, #2

link(oldpath="/x2VC", newpath="/file")
Micro-operations:
#5 create_dir_entry(dirinode=2, entry="file", inode=12)
Dependencies: #1, #2

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

Listing 1: **Annotated Update Protocol.** *The listing shows an example update protocol. The micro-operations generated for each system call are shown along with their dependencies. The inode number of `x2VC` is 12. The inode number of the root directory is 2. Some details of listed system calls have been elided for brevity.*

- *write_block*: an atomic 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 with old file contents
- *change_file_size*: atomically increases or decreases the size of a file
- *create_dir_entry*: atomically creates a directory entry
- *delete_dir_entry*: atomically delete a directory entry

The atomicity properties defined by the model control what micro-operations constitute a logical operation. Ordering properties control which micro-operations can reach disk before other micro-operations. We first describe how the model handles each property, and then illustrate with the update protocol shown in Listing 1.

Atomicity. Table 2 shows how logical file system operations are broken into micro-ops. Overwrites and appends can be atomic at different granularities: the smallest atomic write is of size one byte. For example, in Listing 1, the model's block size is 512 bytes: hence, each `pwrite()` is broken into 512 byte-sized *write_block* operations. The append and truncate operations increase (or decrease) the file size, and fill the extended file region with new data (zeroes in the case of truncate). The *random* and *zeroes* options of *write_block* represent uninitialized and zeroed blocks. If the last link to a file is removed via unlink, the file is deleted. Renaming a file is broken down into a non-atomic pair of unlink and link operations.

Ordering. In the model, all operations following a sync operation on file A depend upon writes to file A preceding the sync operation. For example, in Listing 1, the #3 and #4 *write_block* operations depend upon the #1–2 *write_block* operations due to the intervening `fsync()`. Therefore, ALICE will not construct a crash state that has operations #3 or #4 without operations #1 or #2.

Durability. The application may guarantee certain state to be durable after some terminal output. To check for this in a crash state, we add a special micro-op termed *stdout*, that captures writes by the application to terminal output. These micro-ops depend upon writes synced by preceding sync operations. All operations following the stdout operation depend on it. In Listing 1, terminal output is represented as a *stdout* micro-op. Since it depends only on micro-ops #1–2, a crash state could contain the effects of #1–2 and the *stdout* operation. If the application guarantees that the terminal message indicates all writes (including #3–4) are durable, this is a durability violation.

3.1.3 Constructing and Checking a Crash State

Micro-operations are generated according to the persistence model. Based on persistence property being tested, ALICE selects micro-operations to be applied in sequence to the initial crash state to generate a new crash state.

A user-supplied *Invariant Verifier (IV)* is run on each crash state to check if the application is consistent (e.g., that puts are ordered for the `LevelDB` asynchronous-put workload). The IV also uses terminal output to verify durability for certain applications. If the IV finds an invariant is violated, a vulnerability has been found.

3.2 Persistence Properties Tested

We now describe the different persistence properties tested by ALICE, and how each property is tested.

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 (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 this results in application invariants being violated. The intermediate states for file-system operations are listed in Table 2. Appends have an intermediate state where the block is filled with random data. In terms of splitting a system call into smaller micro-operations, the write, append, and truncate operations are handled in two ways: split into *block* sized micro-operations, and split into three micro-ops irrespective of the size of the resulting micro-op.

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 till B except for A.

3.3 Static Vulnerabilities

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 code line. ALICE uses stack trace information to correlate all 10 system calls to the source line, and reports it as a single **static** vulnerability. In the rest of this paper, we only discuss static vulnerabilities.

Our estimate of static vulnerabilities is conservative. In most applications, the source line in the inner-most stack frame points to a wrapper to the C-library; hence we manually examine the stack to find a representative application source line. However, when not sure about the representative frame, we choose inner frames rather than outer ones.

3.4 Implementation

ALICE consists of around 4000 lines of Python code, and 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, we hash crash states and only check 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. We filtered out system calls involved with these files.

3.5 Limitations

ALICE is not complete, in that there may be vulnerabilities that are not detected by ALICE, especially for applications with large and particularly complex update protocols. It also requires the user to write application workloads and invariant verifiers; 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 to the file system; 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; we believe it would be straight-forward to extend ALICE to do so.

4 Application Vulnerabilities

We studied 12 widely used applications representing different domains, and ranging in maturity from a few years-old to decades-old. We studied four key-value stores (BerkeleyDB [**?**], LevelDB [**?**], GDBM [**?**], LMDB [**?**]), three relational databases (RDBMS) (SQLite [**?**], PostgreSQL [**?**], HSQLDB [**?**]), two version control systems (Git [**?**], Mercurial [**?**]), two distributed systems (HDFS [**?**], ZooKeeper [**?**]) and a virtualization software (VMWare Player [**?**]). We studied two versions of LevelDB (1.10 and 1.15), since LevelDB's update protocol varies significantly between the versions.

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). In the rest of this section, we aim to answer the following questions:

1. How important are discovered vulnerabilities? (§4.3)
2. Are there any patterns among vulnerabilities? (§4.4)
3. How many vulnerabilities are exposed on current file systems? (§4.5)
4. Can ALICE validate new file-system designs? (§4.6)

4.1 Workloads and Checkers

Most applications had configuration options that changed the update protocol or application crash guarantees. Our workloads tested a total of 36 such configuration options across the 12 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 do not clearly define their crash guarantees. Furthermore, many applications provide users with many manual recovery methods to be used after a crash. Our checkers invoke all recovery techniques we are aware of, and are hence complex.

Key-value stores and Relational Databases. Each workload tested different parts of the protocol, typically opening a database, and inserting enough data to trigger checkpoints. The BerkeleyDB checker always verifies the database and invokes recovery if needed. The LevelDB checker always invokes the `RepairDB` command. We do not switch on SQLite options that increase performance by risking correctness atop specific file systems [**?**].

Version Control Systems. Git's crash guarantees are fuzzy; mailing-list discussions suggest that Git expects a fully ordered file system [**?**]. 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 repostiory state. The checkers remove any left-over lock files, and perform all documented recovery techniques (except cloning or restoring a backup).

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 created files (until 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 system 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 uses different checkers for each type of corruption, and report vulnerabilities for each checker separately.

Summary. If application invariants for the tested configuration are explicitly documented, we consider violating those invariants as failure; otherwise, ALICE considers violating a lay user's expectations as failure. We are careful about any recovery procedures that need to be followed on a system crash. Since it is not possible to detail the exact checkers, we plan to make the checkers, along with the ALICE suite, publicly available for verification and reuse.

4.2 Overview

We now discuss the logical protocols of the applications we studied. Figure 4 visually represents update protocols from different classes of applications. The figure shows the logical operations in the protocol (organized as modules), the ordering ensured by the application via `fsync()` calls, and discovered vulnerabilities.

4.2.1 Databases and Key-Value Stores

Most databases use a write-ahead logging technique. First, the new data is written to a log. Then, the log is

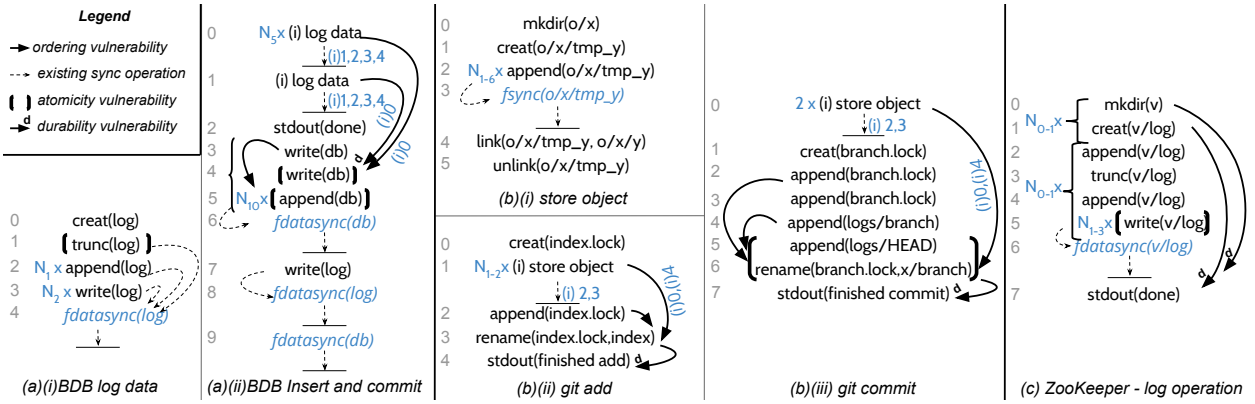

Figure 4: **Protocol Diagrams.** *(a), (b), and (c) show the modularized update protocol for Git, ZooKeeper, and BerkeleyDB-BTree respectively. Repeated operations in a protocol are shown next to each operation, and if the number of repetitions can vary, the variation is specified. Ordering dependencies are indicated with arrows, and dependencies between modules are indicated by the numbers on the arrows, corresponding to line numbers in modules. Dotted, vertical arrows ending at a horizontal line represent ordering dependencies already enforced using sync operations in the application. Dark, bold arrows represent new ordering dependencies discovered;* ***d*** *on the arrow represents a durability requirement. Operations inside dark brackets must be persisted together atomically.*

checkpointed or compacted; i.e., the actual database is carefully overwritten, and the log is deleted.

Figure 4(a) show the protocol used by BerkeleyDB, with its BTree access method. BerkeleyDB adds inserted key-value pairs to the log until it reaches a threshold, and then switches to a new log. Figure 4(a)(i) shows the log appends; Figure 4(a)(ii) shows the complete protocol including checkpointing. During checkpointing, BerkeleyDB first overwrites the database, then appends to it. BerkeleyDB marks the end of checkpointing in the log after persisting the database changes.

In BerkeleyDB and LevelDB-1.15, we found vulnerabilities when initializing or appending to the log file. For example, a crash during initialization of a newly created log file might leave the log uninitialized. BerkeleyDB's recovery code does not handle this situation; the user is unable to open the database. We also found vulnerabilities during database checkpointing. For example, BerkeleyDB does not persist the directory entries of its logs. A crash during checkpoint might cause the logs to vanish; as a result, the database might lose data or simply not open.

Some databases follow radically different protocols than 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 deleting the destination file, and then renaming. We found cases where 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(b). Additionally, 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 found many ordering dependencies in the Git protocol, as shown in Figure 4(b). This is not surprising, since mailing-list discussions suggest Git developers expect total ordering from the file system (which no system provides (§2.2)). We also found a Git vulnerability involving atomicity across multiple system calls; one of the pointer files being updated (via an append) has to happen atomically with another file getting updated (via an update-via-rename). In Mercurial, we find many ordering vulnerabilities for the same reason: the developers have not designed Mercurial to tolerate out-of-order persistence.

4.2.3 Virtualization and Distributed Systems

Surprisingly, VMWare Player's protocol was simple. VMWare maintains a static, constant mapping between blocks in the virtual disk, and blocks 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 minor configuration). Both HDFS and ZooKeeper use write ahead logging. Figure 4(c) shows the ZooKeeper logging module. We found that ZooKeeper does not explicitly persist directory entries of log files; this can lead to lost data. ZooKeeper also requires the log writes to be atomic.

Application	Types										Failure Consequences				Unique static vulnerabilities
	Across-syscalls atomicity	Atomicity				Ordering		Durability			Silent corruption	Data loss	Cannot open	Other	
		Appends and truncates	Multi-block overwrites	Sinlgle-block overwrites	Renames and unlinks	Safe file flush	Safe renames	Other	No commit	Losing old commit					
BDB-BTree		2	1			1			1	1		2	1#,2	1 failed read	**4**
BDB-Hash		2							1	2		3	1#,1		**4**
Leveldb1.10	1*					2		2	1	2	6	3			**7**
Leveldb1.15		1						3			4				**4**
LMDB				1										read-only open	**1**
GDBM	1				1			1	2			2	3		**5**
HSQLDB		1			2	1		3	1	2	2	3	5		**10**
Sqlite-Roll									1			1			**1**
PostgreSQL				1									1#		**1**
Git	1				1	2	1	3	1			1	5	3+	**9**
Mercurial	4	1			1		1	4	2			2	6	7 dirstate fail	**12**
VMWare					1								1		**1**
HDFS					1			1					2		**2**
ZooKeeper				1				1	2			2	2		**4**
Total	**6**	**7**	**1**	**3**	**7**	**6**	**2**	**18**	**12**	**7**	**12**	**19**	**30**	**12**	**65**

Table 3: **Vulnerabilities - Types and Consequences.** *The table shows the discovered static vulnerabilities categorized by the type of persistence property and consequences. 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. SQLite-WAL is not shown because we did not find any vulnerabilities.* ***** *The atomicity vulnerability in Leveldb1.10 corresponds to multiple mmap() writes.* + *There are 2 fsck-only and 1 reflog-only errors in Git.* # *These are known, documented failures.*

4.3 Vulnerabilities Found

ALICE revealed 65 static vulnerabilities in total, corresponding to 162 dynamic vulnerabilities. Altogether, applications failed in more than 4000 crash states. Table 3 presents statistics for each application, classifying vulnerabilities based on the affected persistence property and failure consequence. We reiterate here that 65 is a conservative estimate of the number of logical vulnerabilities. Particularly, in Mercurial, we were unable to correlate system calls with stack traces, since the application is written in Python; we conservatively associate each type of system call to one line of source code.

Some application configuration options change the entire update protocol. For example, the different storage engines provided by BerkeleyDB and SQLite use different protocols, and consequently have different vulnerabilities. Table 3 shows each such configuration separately. Once a storage engine is selected, other configuration options only changed the application invariants. With different configurations of the same update protocol, all vulnerabilities were revealed in the *safest* configuration. Table 3, and the rest of the paper, only show the vulnerabilities found 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 data corruption or data loss. Table 3 shows the vulnerability consequences. Eight applications (and both BerkeleyDB configurations) are affected by data loss, while two (including both LevelDB versions) are affected by silent corruption. The *cannot open* failures include failure to start the server in HDFS and ZooKeeper, and preventing the user from running basic commands (e.g., `git-log`, `git-commit`) in VCS. Many *cannot open* failures are due to fully corrupted datastores, while a few can be solved by application experts. Since we carefully integrated documented crash recovery techniques into our checkers, we believe lay users cannot recover from *cannot open* failures. We also surveyed if any discovered vulnerabilities are previously known, or considered inconsequential. Two BerkeleyDB vulnerabilities and the single PostgreSQL vulnerability are documented; they can be solved only with non-standard (although simple) recovery techniques. The seven *dirstate fail* vulnerabilities in Mercurial are separated from *cannot open* failures since they are less harmful (though frustrating to the lay user). Although Git's `fsck`-only and `reflog`-only errors are potentially dangerous, they do not seem to affect normal usage.

Vulnerabilities exposed by ALICE are hard to reproduce without using it, and thus are difficult to report to application developers. Nevertheless, we reported five vulnerabilities, one each from HSQLDB, LevelDB-1.10, SQLite, GDBM, and BerkeleyDB. HSQLDB and LevelDB-1.10 confirmed the vulnerability. HSQLDB does not intend to fix the vulnerability since they consider it an OS problem (the vulnerability assumes content-atomic appends). When we reported the LevelDB vulnerability, a fix was already underway, though not yet available publicly. SQLite and GDBM developers replied that the expected invariants (durability) were not guaranteed for our workloads; we attribute both found vulnerabilities to erratic documentation. BerkeleyDB developers are investigating our report.

Summary. ALICE detected 65 vulnerabilities in total, with 13 resulting in corruption, 19 in loss of durability, and 31 leading to inaccessible applications. ALICE was able to detect previously known vulnerabilities.

4.4 Common Patterns

We examine vulnerabilities related with different persistence properties. Since durability vulnerabilities can result from the violation of any persistence property, we consider them separately.

4.4.1 Atomicity across System Calls

GDBM, Mercurial, and Git require atomicity across system calls. However, the vulnerability consequences seem

minor: database inaccessibility in GDBM, dirstate corruption in Mercurial, and erratic `reflog` output in Git.

LevelDB-1.10 works correctly only if multiple writes to a `mmap()`-ed file are persisted atomically. We believe this particular vulnerability is an outlier. LevelDB is logically appending to a log using `mmap()`-ed I/O; the vulnerability is similar to a content-atomic append vulnerability.

4.4.2 Atomicity within System Calls

Append atomicity. Surprisingly, four applications require appends to be *content-atomic*: the appended portion should contain actual data. The failure consequences are severe, such as corrupted reads (HSQLDB, BerkeleyDB and 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 do not need to be *block-atomic*; only Mercurial is affected, and the affected append also requires content-atomicity. BerkeleyDB requires that newly truncated (expanded) file regions contain zeroes for correctness.

Overwrite atomicity. LMDB, PostgreSQL, and ZooKeeper require small writes ($<$ 200 bytes) to be atomic. The LMDB and PostgreSQL writes happen to fixed file locations. The PostgreSQL vulnerability is documented. Only BerkeleyDB requires a two-block write to be atomic; a violation results in a *cannot open* failure. Interestingly, the BerkeleyDB source lines issuing the write are also involved in the atomic append vulnerability.

Thus, when compared with append atomicity, multi-block overwrite atomicity vulnerabilities are strikingly less prevalent. Some of the difference can be attributed to the persistence model (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 corruptions of the logged data.

Directory operation atomicity. Given how 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 find this is not the case. Databases do not employ atomic renames extensively; consequently, non-atomic renames only affected two databases (GDBM, HSQLDB). Git, Mercurial, HDFS, and VMWare Player also use rename sparingly, resulting in only one rename vulnerability in each application. Non-atomic unlinks affect only HSQLDB (which uses unlinks for logically performing renames), while non-atomic truncates (that reduce file size) do not affect any application.

4.4.3 Ordering between System Calls

Applications are extremely vulnerable to system calls being persisted out of order; we found 25 vulnerabilities.

Safe renames. On file systems with delayed allocation, a common heuristic to prevent data loss is to persist all appends to a file before subsequent renames of the file [**?**]. We found that this heuristic only fixes 2 discovered vulnerabilities, 1 each in Git and Mercurial. We believe the heuristic is more useful for a different class of applications (e.g., GNOME applications) than those we study.

Safe file flush. A `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 4 applications for maintaining even basic consistency. Additionally, ZooKeeper requires it for durability.

4.4.4 Durability

We found two distinct durability loss scenarios: data is not committed properly and hence lost; or, data that *was* correctly committed previously is lost later.

Not properly committing data. We find 12 instances where applications fail to correctly commit data. Improper commit happens in one of two ways: applications might simply not `fsync()` written data (e.g., GDBM); applications might not persist the `creat()` or `mkdir()` of the file hierarchy containing the data (e.g., HSQLDB, LevelDB-1.10, ZooKeeper); applications may also fail to persist a directory operation or a small `write()` marking transaction completion (e.g., `unlink()` in SQLite, `rename()` in Git, `write()` in BerkeleyDB).

Losing pre-committed data. Certain vulnerabilities might result in the applications losing data that was already committed. We observe 7 such vulnerabilities, spread across BerkeleyDB, HSQLDB, and LevelDB-1.10. Most were ordering vulnerabilities, while one depended on atomicity of an append (BerkeleyDB). We further analyzed BerkeleyDB to understand the root cause; under the failing crash states, the `DB_Verify` command does not detect any inconsistencies, and hence our checker opens the database without explicitly calling `DB_Recover`. In other databases, though our checkers always call all available recovery commands, we believe the root cause is similar to BerkeleyDB: incorrect recovery code.

4.4.5 Summary

We believe our study offers several insights for file-system designers. Vulnerabilities involving atomicity of multiple system calls seem to have minor consequences. We believe that it is sufficient for future file systems to focus on providing atomicity *within* a system call, and ordering between system calls.

Breaking the atomicity of directory operations does not have severe implications for many applications; this could

[t]0.27

	Ordering	Atomicity		
		DO	AG	CA
ext3-w	Dir ops and file-size changes ordered among themselves, and 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, directory operations ordered among themselves	✓	4K	✓
btrfs	Safe rename, safe file flush	✓	4K	✓

(a)

[t]0.19

	ext3-w	ext3-o	ext3-j	ext4-o	btrfs
BDB-BTree	3	1	1	1	1
BDB-Hash	4	1	1	1	2
Leveldb1.10	2	1	1	1	4
Leveldb1.15	1				3
GDBM	2	2	1	2	3
Git	2	2	2	2	5
Mercurial	3	2	2	4	6
HSQLDB					4
Sqlite-Roll	1	1	1	1	1
HDFS					1
ZooKeeper	1	1		1	1
Total	19	11	9	13	31

(b)

Table 4: **Vulnerabilities on Current File Systems.** *(a) describes the simple file system models used. (b) shows the number of vulnerabilities that occur on current file systems. Applications that do not fail in the considered file systems are omitted. Legend: DO: directory operations. AG: granularity of size-atomicity. CA: Content-Atomicity.*

point to a way to improve file-system performance.

Missing *sync* operations are not the only reason for durability loss – atomicity and ordering vulnerabilities also lead to data loss. Therefore, features such as atomic appends can help prevent data loss.

4.5 Vulnerabilities On Current File Systems

We evaluated the occurrence of vulnerabilities in current file systems by configuring ALICE with simple persistence models of the file systems. The considered persistence models are shown in Table 4(a). Table 4(b) shows the vulnerabilities reported by ALICE for each file system.

We make a number of observations based on Table 4(b). First, a significant number of vulnerabilities are exposed on *all* examined file systems. Second, ext3 in data journaling mode is the safest: the only vulnerabilities exposed are related to durability and atomicity across system calls. Third, a large number of vulnerabilities are exposed on btrfs as it aggressively persists operations out of order [**?**].

Summary. Application vulnerabilities are exposed on many current file systems; the vulnerabilities exposed vary based on the file system, thus testing applications on specific 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 [**?**], because of changes to the file-system persistence model. ALICE can be used to verify that such modifications don't break correctness of existing applications. We describe how we used ALICE to evaluate a hypothetical variant of ext3 (data journaling mode), `ext3-fastfsync`.

Our study shows that ext3 (data journaling mode) is the safest file system; however, it offers poor performance for many workloads [**?**]. 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 [**?**] have resulted in increased vulnerabilities. Our study suggests a way to reduce `fsync()` latency *without* introducing additional vulnerabilities.

Based on our study, we hypothesize that data that is not *synced* need not be persisted before explicitly synced data for correctness. We design `ext3-fastfsync` 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-fastfsync` in ALICE by introducing a slight change in the persistence model of ext3. We changed the ordering dependencies of synced files and their data: previously, they were ordered after all previous operations; we changed them to depend on previous syncs and operations necessary for the file to exist (i.e., safe file flush is obeyed). The overwrites of a synced file are ordered among themselves.

ALICE verified our hypothesis: `ext3-fastfsync` does not expose any of the observed ordering vulnerabilities on the 12 tested applications. The design was not meant to fix durability or atomicity across system calls vulnerabilities, so those vulnerabilities are still reported by ALICE.

We estimated the performance gain of `ext3-fastfsync` 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 4 seconds. If the files belong to different ext3-ordered partitions on the same disk, mimicing the behavior of `ext3-fastfsync`, 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. ALICE helps evaluate the impact of changes to file-system design on application correctness. Evaluating `ext3-fastfsync` required less than 50 lines of code.

5 Discussion

Our study reveals why crash vulnerabilities occur so commonly even among widely used applications.

We found 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 leads to problems 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. ALICE helps solve this problem by making it easy for developers to obtain logical representations of update protocols such as Figure 4.

Another factor contributing to crash vulnerabilities is poorly written, untested recovery code. In BerkeleyDB and 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, Berkeley, Git). Mercurial and LevelDB provide utilities to verify or recover application data; we find these utilities are hard to configure and error-prone. For example, LevelDB's recovery command works correctly only when (seemingly) unrelated configuration options (*paranoid checksums*) are setup properly, and sometimes ends up *further* corrupting the data-store. 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 different corner cases that would occur rarely in practice.

We found that convincing developers about crash vulnerabilities was hard, since the vulnerabilities are hard to reproduce. More troubling is the general mistrust surrounding such bug reports. Usually, developers are suspicious that the underlying storage stack might not respect `fsync()` calls, or that the drive might be corrupt. We hence believe that most vulnerabilities that occur in the wild go unreported, or get associated with a wrong root cause. ALICE can easily reproduce crash vulnerabilities.

Out-of-date, 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) by default 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.

6 Related Work

Pillai et al. [**?**] identify the problem of application consistency depending on file-system behavior, but they are limited by manual testing, and cover only 2 applications.

EXPLODE [**?**] has a similar flavor to our work: the authors use *in-situ* model checking to find crash vulnerabilities on different storage stacks. Our work differs from EXPLODE in 4 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 persistence model. 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 would be easier to do so using ALICE since ALICE checks for violation of specific persistence properties. Furthermore, EXPLODE stops at finding crash vulnerabilities; by helping produce protocol diagrams, ALICE contributes to understanding the protocol itself.

Woodpecker [**?**] uses known patterns of source code lines that cause vulnerabilities to find whether the same pattern occurs in other applications. Our work is fundamentally different as ALICE intends to infer new patterns.

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

OptFS [**?**], Featherstitch [**?**], and transactional file systems [**?**, **?**, **?**], discuss new file system interfaces that will affect vulnerabilities. Our study can help inform the designs of new interfaces, by providing clear insights into what is missing in today's interfaces.

7 Conclusion

In this paper, we show how application-level consistency is dangerously dependent upon how file systems persist system calls. We show that persistence properties (file-system behavior affecting application correctness) vary widely among file systems. We build ALICE, a framework that analyzes application-level protocols and detects crash vulnerabilities. We use ALICE to analyze 12 applications, finding 65 vulnerabilities across tested applications, some of which result in severe consequences like corruption or data loss. We present several insights derived from our study of application update protocols and vulnerabilities. We make both our tool and our data publicly available to stimulate further research in application-level consistency.

References