EXPLODE: a Lightweight, General System for Finding Serious Storage System Errors

Junfeng Yang

Joint work with Can Sar, Paul Twohey, Ben Pfaff, Dawson Engler and Madan Musuvathi
“Mom, Google Ate My GMail!”

**Mom, Google Ate My GMail!**

**Update:** If you are ready to give up your Gmail and are considering switching to all-Google Mail, one of us has done, be warned that the simple and straightforward approach may not have worked. An increasing number of Gmail subscribers are complaining that their accounts have been deleted without notice.

For nearly 10 days, a Reddit user has been cleaning out the Gmail inbox. The user recently wrote to us this message:

> Just when we thought we had it right, it went wrong. (If you are one of the victims, try and get to Google people and see what's going on.) Gmail Threads, a user written plugin for Gmail, is now orphaned.

Not only are we surprised that these things happen to a company like Google but we are pissed about it. Our internet life.

---

**TechCrunch**

Gmail Disaster: Reports Of Mass Email Deletions

Michael Arrington

Just a week after I wrote “Uh Oh, Gmail Just Got Perfect” a number of users started complaining that all of their Gmail emails and contacts were auto-deleted.

The first message, posted on the Google Groups forum on December 19, stated “Found my account clean...nothing in Inbox, contacts, sent mail...How can all these information residing in different folders disappear?...How to write to gmail help team to restore the account...is it possible?...Where to report this abuse?...Any help...Welcome...Thanks in advance ps101”
Flash: Software wings its way to Mars rovers

By Patricia Daukantas, GCN Staff

NASA's twin Mars rovers have been receiving medicinal shots of software over the agency's Deep Space Network.

The updates let Earth-based scientists and engineers send updates to the rovers' computer software stores, said Roger Klemm, a flight software developer at NASA's Jet Propulsion Lab.

Last week, Spirit was struggling to recover from a mysterious computer crash, Klemm said. Engineers were trying to reboot itself, Klemm said.

Engineers command it to restart the rover's computer, Klemm said. They discovered the problem, however, by running a checkdisk routine.

Each rover has 256 megabytes of memory. Certain blocks are reserved for dedicated purposes. The rest, about 100 megabytes, is used as a flash file system. The program that sends the updates is called Spirit's flash format.
Why check storage systems?

- **Storage system errors:** some of the most serious
  - machine crash
  - data loss
  - data corruption

- **Code complicated, hard to get right**
  - *Conflicting goals: speed, reliability (recover from any failures and crashes)*

- **Typical ways to find these errors:** ineffective
  - Manual inspection: *strenuous, erratic*
  - Randomized testing (e.g. unplug the power cord): *blindly throwing darts*
  - Error report from *mad users*
Goal: build tools to automatically find storage system errors

Sub-goal: comprehensive, lightweight, general
EXPLODE [OSDI06]

- Comprehensive: adapt ideas from model checking
- General, real: check live systems
  - Can run (on Linux, BSD), can check, even w/o source code
- Fast, easy
  - Check a new storage system: 200 lines of C++ code
  - Port to a new OS: 1 kernel module + optional modification
- Effective
  - 17 storage systems: 10 Linux FS, Linux NFS, Soft-RAID, 3 version control, Berkeley DB, VMware
  - Found serious data-loss in all
- Subsumes FiSC [OSDI04, best paper]
Outline

Overview

- Checking process

- Implementation

- Example check: crashes during recovery are recoverable

- Results
Long-lived bug fixed in 2 days in the IBM Journaling file system (JFS)

- **Serious**
  - Loss of an entire FS!
  - Fixed in 2 days with our complete trace

- **Hard to find**
  - 3 years old, ever since the first version

---

Dave Kleikamp (IBM JFS): “I really appreciate your work finding and recreating this bug. I'm sure this has bitten us before, but it's usually hard to go back and find out what causes the file system to get messed up so bad”
Events to trigger the JFS bug

`creat("/f");`

5-char system call, not a typo

`fsck.jfs`

File system recovery utility, run after reboot

Buffer Cache (in mem)

Disk

Orphan file removed.
Legal behavior for file systems
Events to trigger the JFS bug

```
creat("/f");
bug under low mem (design flaw)
flush "/"
crash!
fsck.jfs
```

Buffer Cache (in mem)

Disk

File system recovery utility, run after reboot

dangling pointer!

“fix” by zeroing, entire FS gone!
Overview

 void mutate() {
  Toy checker: crash after creat("/f") should not lose any old file that is already persistent on disk
  check_crash_now();
}

 void check() {
  User-written checker: can be either very sophisticated or very simple
  int fd = open("/old", O_RDONLY);
  if (fd < 0) error("lost old!");
  close(fd);
}

User-written Checker

create("/f")

EXPLODE Runtime

"crash-disk"

EKM = EXPLODE kernel module
Outline

- Overview
- Checking process
- Implementation
- Example check: crashes during recovery are recoverable
- Results
One core idea from model checking: explore all choices

- Bugs are often triggered by corner cases
- How to find? Drive execution down to these tricky corner cases

**Principle**

When execution reaches a point in program that can do one of N different actions, fork execution and in first child do first action, in second do second, etc.

Result: rare events appear as often as common ones
Crashes (Overview slide revisit)

User-written Checker

EXPLODE Runtime

“crash-disk”

Linux Kernel

Hardware

JFS

/root

old

EKM

/root

fsck.jfs

check(/root old)

✓

fsck.jfs

check(/root)

✗

fsck.jfs

check(/root old f)

✓
External choices

- Fork and do every possible operation

![Diagram](attachment:image.png)

Explore generated states as well

Users write code to check FS valid. EXPLODE “amplifies”
**Internal choices**

- Fork and explore all internal choices

```c
struct block* read_block (int i) {
    struct block *b;
    if ((b = cache_lookup(i)))
        return b;
    return disk_read (i);
}
```
Users expose choices using `choose(N)`

- To explore N-choice point, users instrument code using `choose(N)` (also used in other model checkers)
- `choose(N)`: N-way fork, return K in K'th kid

```c
struct block* read_block (int i) {
    struct block *b;
    if ((b = cache_lookup(i)))
        return b;
    return disk_read (i);
}
```

```c
cache_lookup (int i) {
    if(choose(2) == 0)
        return NULL;
    // normal lookup
    ...
}
```

- Optional. Instrumented only 7 places in Linux
Crash X External X Internal
Speed: skip same states

Abstract and hash a state, discard if seen.
Outline

- Overview
- Checking process

Implementation

- FiSC, File System Checker, [OSDI04], best paper
- EXplode, storage system checker, [OSDI06]

- Example check: crashes during recovery are recoverable
- Results
How to checkpoint and restore a live OS?

S₀ = checkpoint()
enqueue(S₀)
while(queue not empty){
    S = dequeue()
    for each action in S {
        restore(S)
        do action
        S’ = checkpoint()
        if(S’ is new)
            enqueue(S’)
    }
}
FiSC: jam OS into tool

- **Pros**
  - Comprehensive, effective
  - No model, check code
  - Checkpoint and restore: easy

- **Cons**
  - Intrusive. Build fake environment. Hard to check anything new. Months for new OS, 1 week for new FS

- Many tricks, so complicated that we won best paper OSDI 04
EXPLODE: jam tool into OS

FiSC

User Mode Linux

Linux Kernel

Hardware

User-writtten Checker

JFS

EKM = EXPLODE kernel module

User-written Checker

EXPLODE Runtime

JFS

EKM

Linux Kernel

Hardware

User code

Our code

EKM

User code
**EKM lines of code**

<table>
<thead>
<tr>
<th>OS</th>
<th>Lines of code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux 2.6</td>
<td>1,915</td>
</tr>
<tr>
<td>FreeBSD 6.0</td>
<td>1,210</td>
</tr>
</tbody>
</table>

**EXPLODE kernel modules (EKM) are small and easy to write**
How to checkpoint and restore a live OS kernel?

- Hard to checkpoint live kernel memory

- Virtual machine? No
  - VMware: no source
  - Xen: not portable
  - heavyweight

- There’s a better solution for storage systems
Checkpoint: save actions instead of bits

state = list of actions
checkpoint $S = \text{save (creat, cache miss)}$
restore = re-initialize, creat, cache miss

re-initialize = unmount, mkfs

Utility that clears in-mem state of a storage system

Utility to create an empty FS

We use it only to reduce intrusiveness
Deterministic replay

- Storage system: isolated subsystem

- Non-deterministic kernel scheduling decision
  - Opportunistic fix: priorities

- Non-deterministic interrupt
  - Fix: use RAM disks, no interrupt for checked system

- Non-deterministic kernel `choose()` calls by other code
  - Fix: filter by thread IDs. No `choose()` in interrupt

- Worked well in practice
  - Mostly deterministic
  - Worst case: auto-detect & ignore non-repeatable errors
Outline

- Overview
- Checking process
- Implementation

Example check: crashes during recovery are recoverable

- Results
Why check crashes during recovery?

- Crashes are highly correlated
  - Often caused by kernel bugs, hardware errors
  - Reboot, hit same bug/error

What to check?

- `fsck once == fsck & crash, re-run fsck`
  - `fsck(crash-disk)` to completion, “/a” recovered
  - `fsck(crash-disk)` and crash, `fsck`, “/a” gone

- Powerful heuristic, found interesting bugs (wait until results)
How to check crashes during recovery?

“crash-disk”

fsck.jfs

EXPLODE Runtime

“crash-crash-disk”

Problem: N blocks $\Rightarrow$ $2^N$ crash-crash-disks.
Too many! Can prune many crash-crash-disks
Simplified example

- 3-block disk, B1, B2, B3
- each block is either 0 or 1
- crash-disk = 000 (B1 to B3)

fsck(000)

Read(B1) = 0
Write(B2, 1)
Write(B3, 1)
Read(B3) = 1
Write(B1, 1)

buffer cache: B2=1
buffer cache: B2=1, B3=1
buffer cache: B2=1, B3=1, B1=1
fsck(000) = 111
Naïve strategy: 7 crash-crash-disks

\[
\begin{align*}
\text{crash-disk} &= 000 \\
\text{fsck}(000) &= 111 \\
\text{Read}(B1) &= 0 \\
\text{Write}(B2, 1) \\
\text{Write}(B3, 1) \\
\text{Read}(B3) &= 1 \\
\text{Write}(B1, 1)
\end{align*}
\]

buffer cache: \( B2=1, B3=1, B1=1 \)

\[
\begin{align*}
\text{fsck}(000 + \{B2=1\}) &= 111 \\
\text{fsck}(010) &= 111? \\
\text{fsck}(001) &= 111? \\
\text{fsck}(011) &= 111? \\
\text{fsck}(100) &= 111? \\
\text{fsck}(110) &= 111? \\
\text{fsck}(101) &= 111? \\
\text{fsck}(111) &= 111?
\end{align*}
\]
Optimization: exploiting determinism

- For all practical purposes, `fsck` is deterministic
  - read same blocks → write same blocks

- `fsck(000) = 111`

- `crash-disk = 000`

- `fsck(000) = 111`

- `Read(B1) = 0`
- `Write(B2, 1)`
- `Write(B3, 1)`
- `Read(B3) = 1`
- `Write(B1, 1)`

- `000 + \{B2=1\}`

- `fsck(010) == 111?`

- `fsck(000)` doesn’t read B2

- So, `fsck(010) = 111`
What blocks does fsck(000) actually read?

\[
\begin{align*}
\text{crash-disk} &= 000 \\
\text{fsck(000)} &= 111 \\
\text{Read(B1)} &= 0 \\
\text{Write(B2, 1)} \\
\text{Write(B3, 1)} \\
\text{Read(B3)} &= 1 \\
\text{Write(B1, 1)}
\end{align*}
\]

Read of B3 will get what we just wrote. Can't depend on B3

\[
\text{fsck(000)} \text{ reads/depends only on B1.} \quad \text{It doesn't matter what we write to the other blocks.}
\]

\[
\text{fsck(O***)} = 111
\]
Prune crash-crash-disks matching 0**

\[
\begin{align*}
\text{crash-disk} &= 000 \\
\text{fsck}(000) &= 111 \\
\text{Read}(B1) &= 0 \\
\text{Write}(B2, 1) \\
\text{Write}(B3, 1) \\
\text{Read}(B3) &= 1 \\
\text{Write}(B1, 1) \\
\text{buffer cache: } B2 &= 1, B3 = 1, B1 = 1 \\
\end{align*}
\]

Can further optimize using this and other ideas.
Outline

- Overview
- Checking process
- Implementation
- Example check: crashes during recovery are recoverable

Results
Bugs caused by crashes during recovery

- Found data-loss bugs in all three FS that use logging (ext3, JFS, ReiserFS), total 5

- Strict order under normal operation:
  - First, write operation to log, commit
  - Second, apply operation to actual file system

- Strict (reverse) order during recovery:
  - First, replay log to patch actual file system
  - Second, clear log
  - No order ➔ corrupted FS and no log to patch it!
Bug in fsck.ext3

```c
recover_ext3_journal(...) {
   // ...
   retval = -journal_recover(journal);
   // ...
   // clear the journal
   e2fsck_journal_release(...)
   // ...
}
```

```c
journal_recover(...) {
   // replay the journal
   //...
   // sync modifications to disk
   fsync_no_super (...)
}
```

- Code directly adapted from the kernel
- But, fsync_no_super defined as NOP: “hard to implement”

// Error! Empty macro, doesn’t sync data!
#define fsync_no_super(dev) do {} while (0)
## FiSC Results (can reproduce in EXPLODE)

<table>
<thead>
<tr>
<th>Error Type</th>
<th>VFS</th>
<th>ext2</th>
<th>ext3</th>
<th>JFS</th>
<th>ReiserFS</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data loss</td>
<td>N/A</td>
<td>N/A</td>
<td>1</td>
<td>8</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>False clean</td>
<td>N/A</td>
<td>N/A</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Security</td>
<td>N/A</td>
<td>N/A</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>3 + 2</td>
</tr>
<tr>
<td>Crashes</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>Other</td>
<td>1</td>
<td></td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>2</td>
<td>2</td>
<td>5</td>
<td>21</td>
<td>2</td>
<td>32</td>
</tr>
</tbody>
</table>

32 in total, 21 fixed, 9 of the remaining 11 confirmed
## EXPLODE checkers lines of code and errors found

<table>
<thead>
<tr>
<th>Storage System Checked</th>
<th>Checker</th>
<th>Bugs</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 file systems</td>
<td>5,477</td>
<td>18</td>
</tr>
<tr>
<td><strong>Storage applications</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVS</td>
<td>68</td>
<td>1</td>
</tr>
<tr>
<td>Subversion</td>
<td>69</td>
<td>1</td>
</tr>
<tr>
<td>“EXPENSIVE”</td>
<td>124</td>
<td>3</td>
</tr>
<tr>
<td>Berkeley DB</td>
<td>202</td>
<td>6</td>
</tr>
<tr>
<td><strong>Transparent subsystems</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RAID</td>
<td>FS + 137</td>
<td>2</td>
</tr>
<tr>
<td>NFS</td>
<td>FS</td>
<td>4</td>
</tr>
<tr>
<td>VMware GSX/Linux</td>
<td>FS</td>
<td>1</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>6,008</td>
<td>36</td>
</tr>
</tbody>
</table>

6 bugs per 1,000 lines of checker code
Related work

- FS Testing
- Static (compile-time) analysis
- Software model checking
Conclusion

EXPLODE
- Comprehensive: adapt ideas from model checking
- General, real: check live systems in situ, w/o source code
- Fast, easy: simple C++ checking interface

Results
- Checked 17 widely-used, well-tested, real-world storage systems: 10 Linux FS, Linux NFS, Soft-RAID, 3 version control, Berkeley DB, VMware
- Found serious data-loss bugs in all, over 70 bugs in total
- Many bug reports led to immediate kernel patches