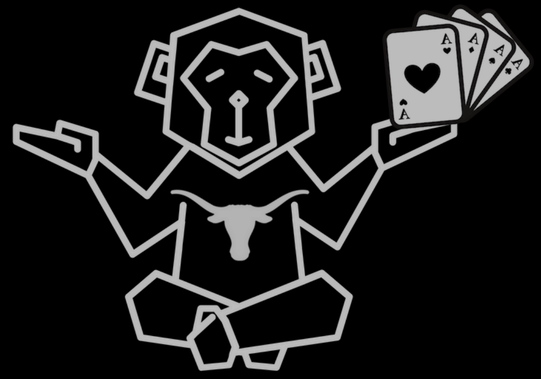


CRASH MONKEY & ACE

Systematically Testing File-System Crash Consistency

[Published at OSDI 2018]

Jayashree Mohan

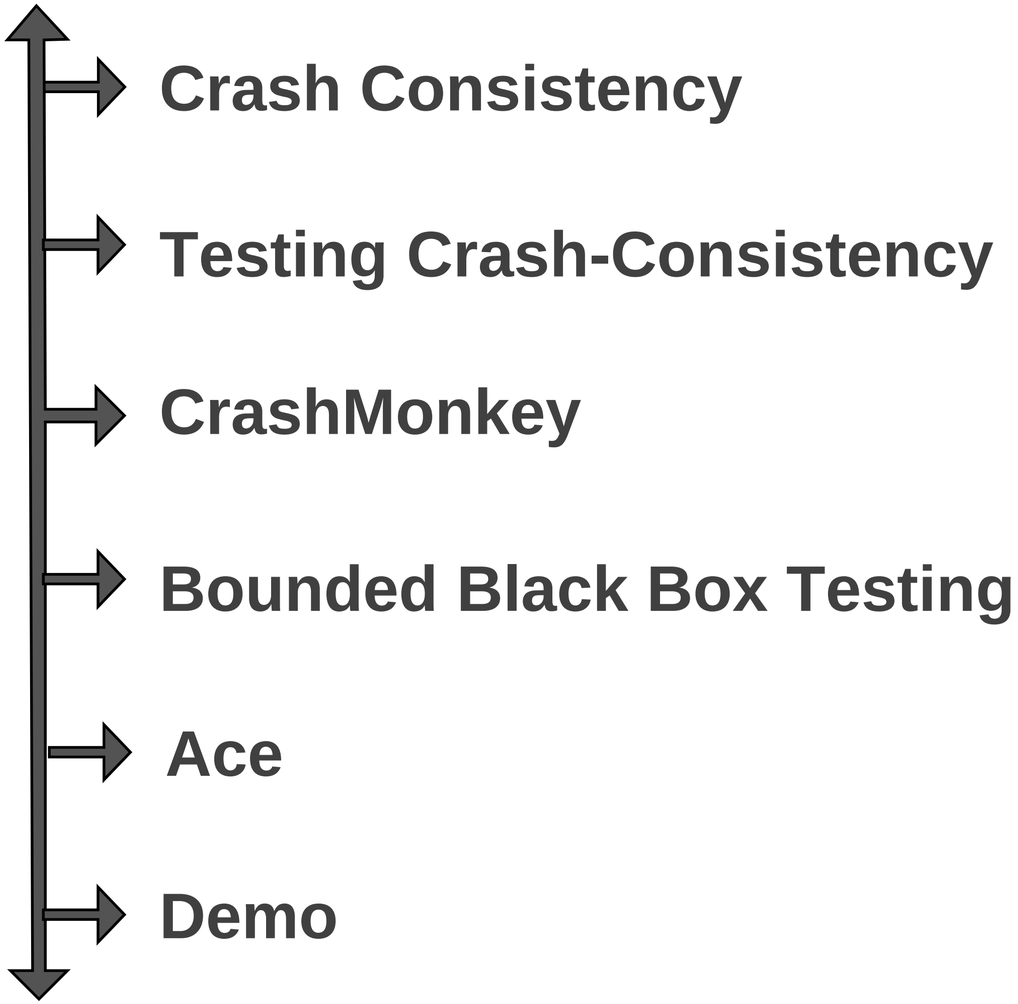
University of Texas at Austin

jaya@cs.utexas.edu

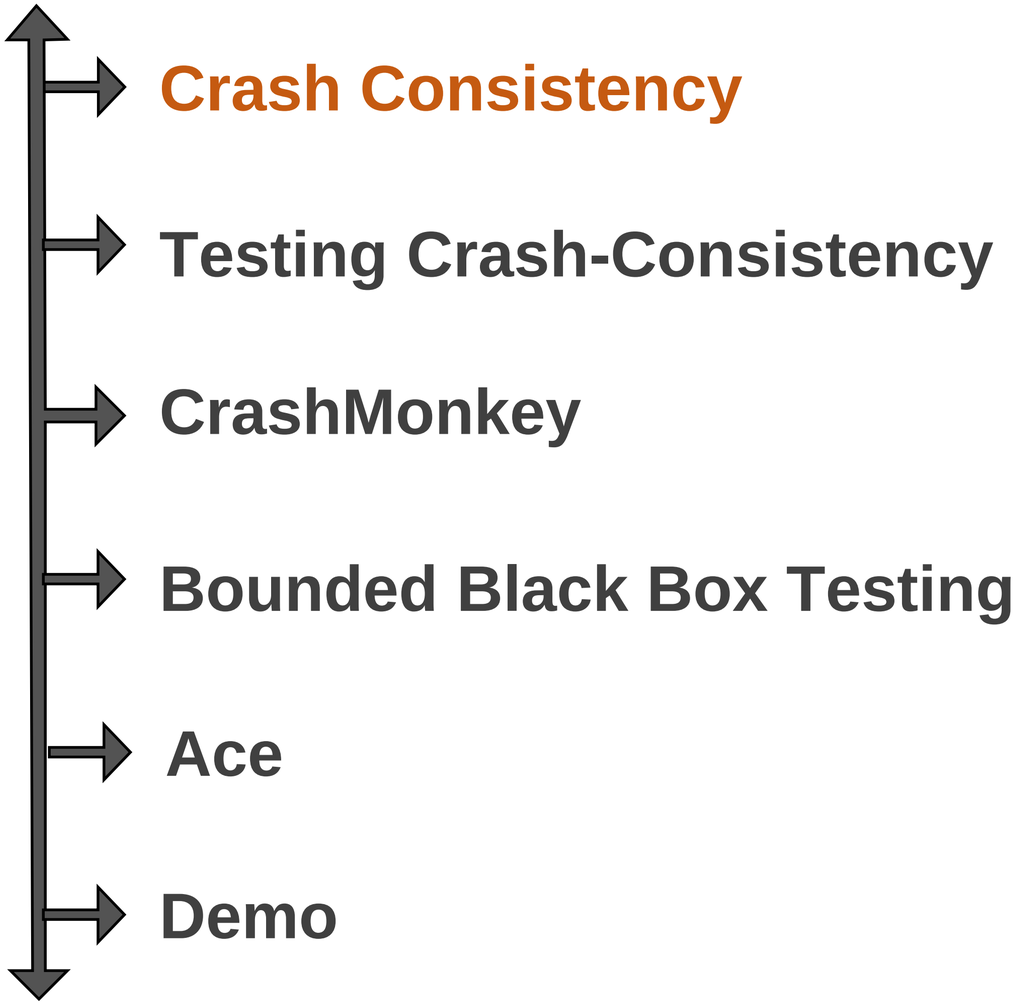
TEXAS
The University of Texas at Austin

UT SaSLab

Agenda


Crash Consistency
Testing Crash-Consistency
CrashMonkey
Bounded Black Box Testing
Ace
Demo

Agenda



Crash Consistency

Testing Crash-Consistency

CrashMonkey

Bounded Black Box Testing

Ace

Demo

What is a File System?

- A file system is a structured representation of data and a set of **metadata** describing this data.
  - Data includes abstractions like files and directories

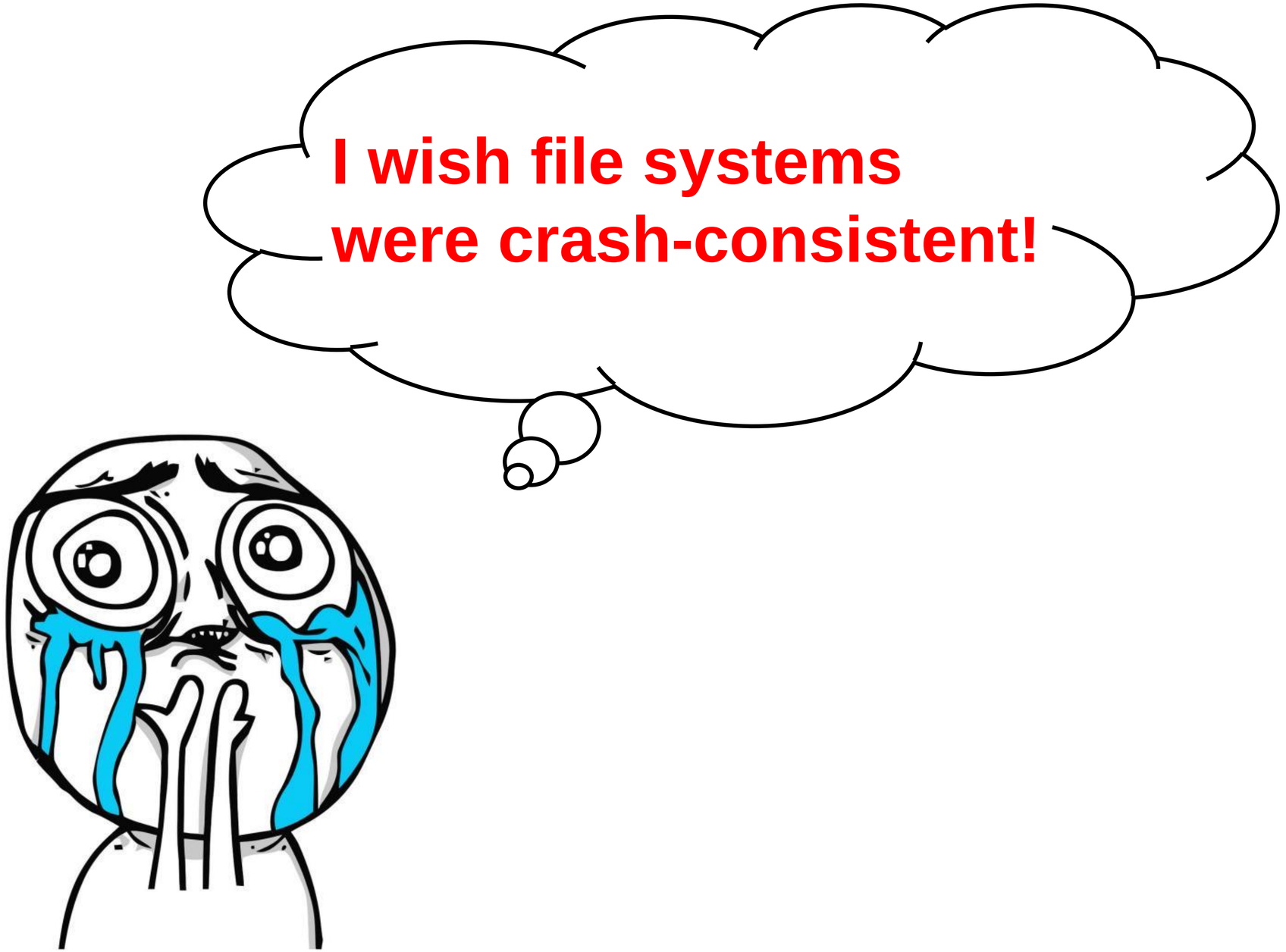
- File system data structures are persisted
  - Stored on hard disk, SSDs etc

What is a crash?

- An event that results in interruption of ongoing processes in the system
- Loss of current working state in memory
- Storage left in an intermediate state

Crashes

Image source : https://www.fotolia.com


I wish file systems
were crash-consistent!

File System Crash Consistency

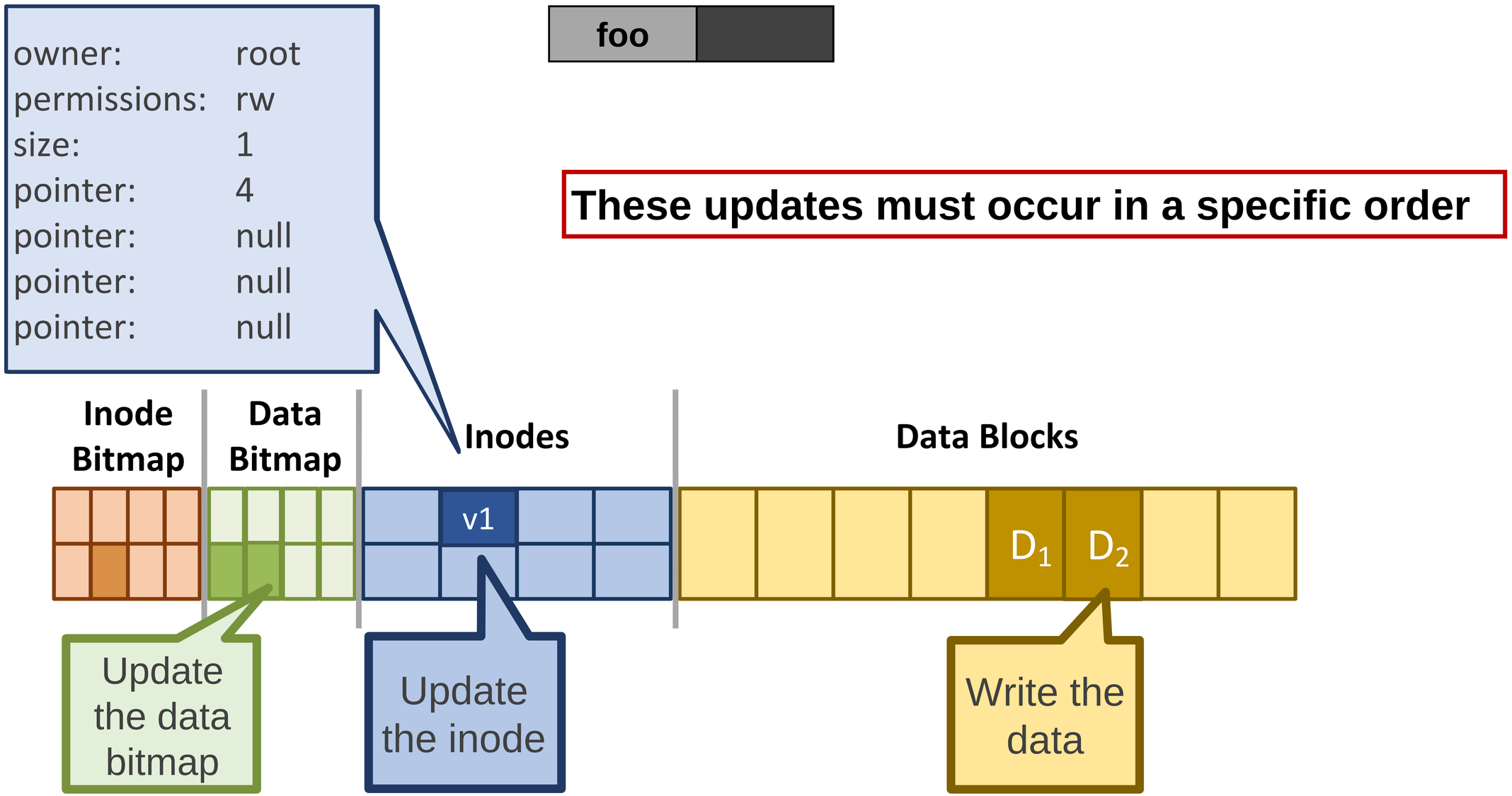
1. **Ordering :** Filesystem operations change multiple blocks on storage that needs to be ordered
   - Inode, bitmaps, data blocks, superblock

2. **Persistence :** Data structures are cached for better performance
   - Great for reads!
   - But writes have to ensure that modified data in cache is written back to disk

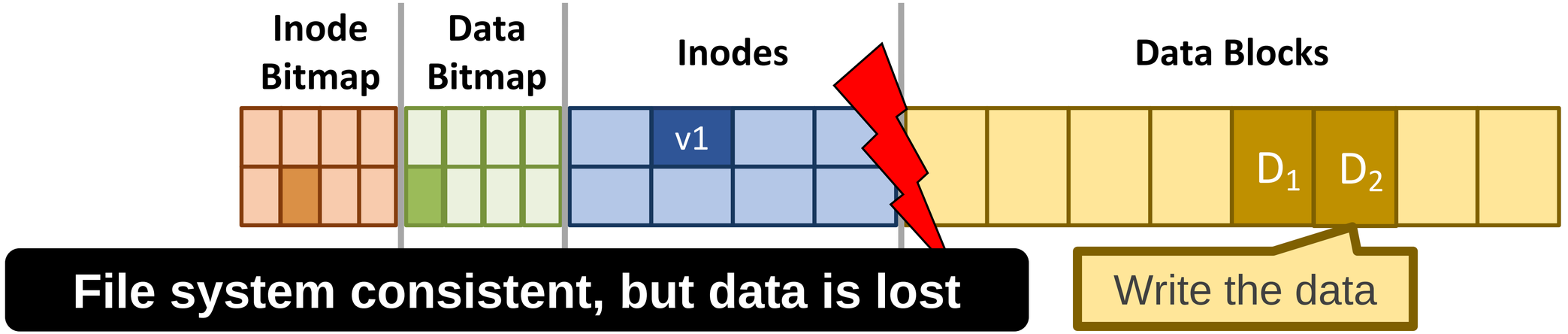
Example of a crash scenario

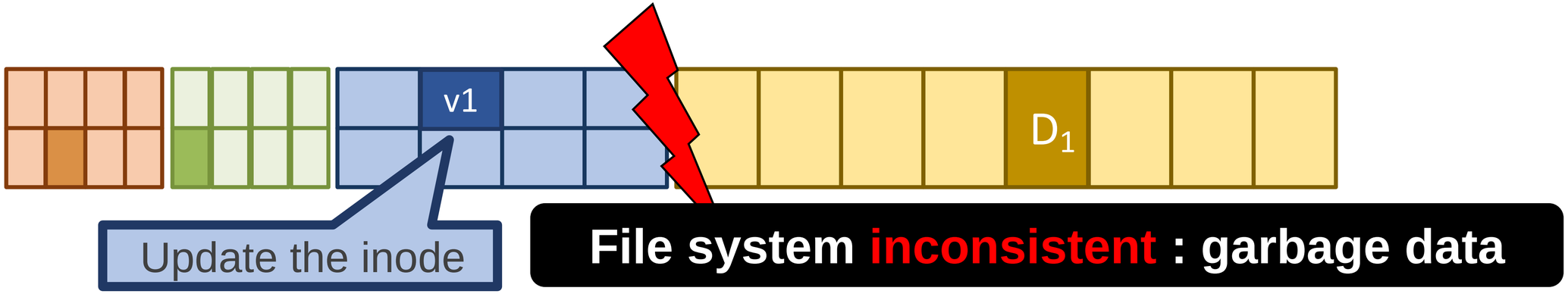
- Let’s consider what happens during a file append

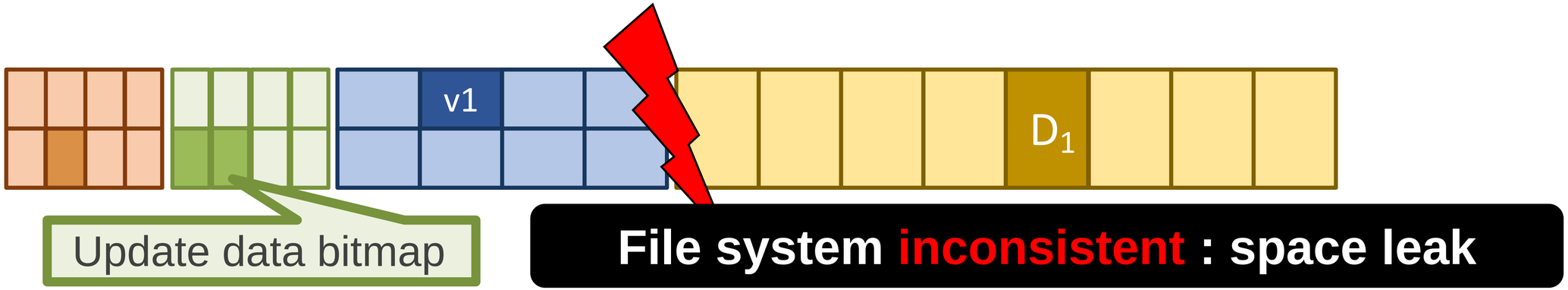
File Append Example



Example adapted from https://cbw.sh/static/class/5600/slides/10_File_Systems.pptx


Inode Bitmap
Data Bitmap
Inodes
Data Blocks
v1
D1
D2
File system consistent, but data is lost
Write the data


v1
D1
Update the inode
File system inconsistent : garbage data


v1
D1
Update data bitmap
File system inconsistent : space leak

How did FS developers handle this problem?

1. Don’t bother to ensure consistency
   - Run a program that fixes the file system during bootup
   - File system checker (*fsck*)
   - Results in data loss, but fixes inconsistency

ext2

2. Use a transaction log to make multi-writes atomic
   - Log stores a history of all writes to the disk
   - After a crash the log can be “replayed” to finish updates
   - Journaling file system (ext4, f2fs, btrfs, xfs)

ext4

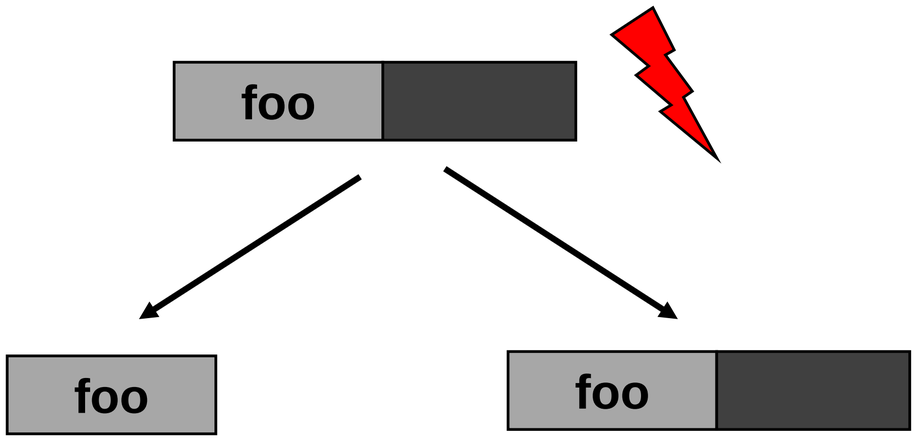
File System Crash Consistency

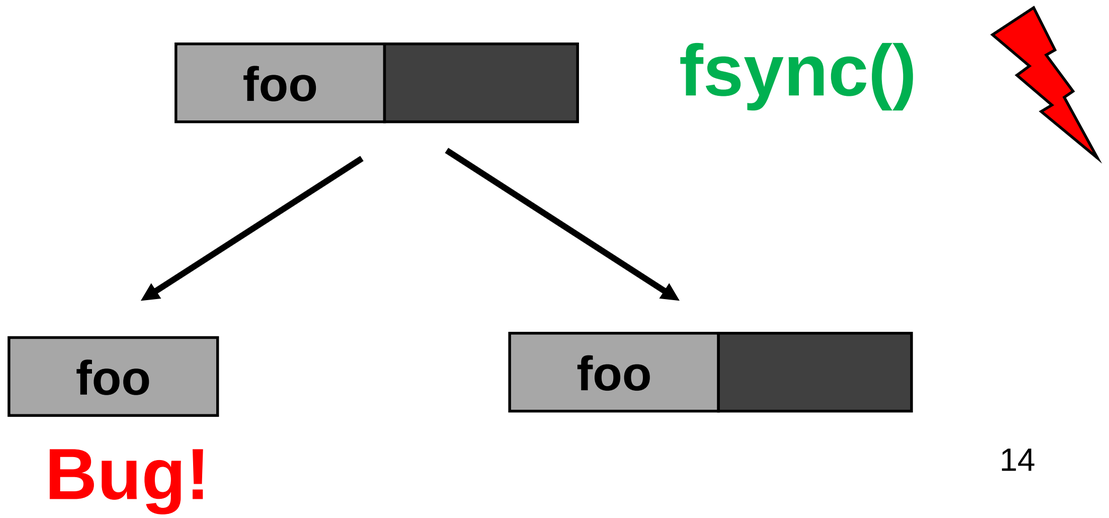
1. **Ordering :** Filesystem operations change multiple blocks on storage that needs to be ordered
   - Inode, bitmaps, data blocks, superblock

2. **Persistence :** Data structures are cached for better performance
   - Great for reads!
   - But writes have to ensure that modified data in cache is written back to disk

The Persistence Operations

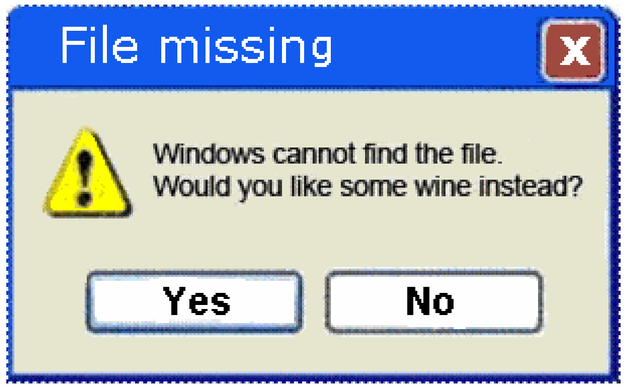
- Journaling file systems aim to ensure crash consistency
- But, can result in data loss if file system operations are not persisted explicitly
- Changes are in memory until explicitly flushed (or a file system background checkpoint at regular timeouts)
- fsync(), fdatasync(), sync



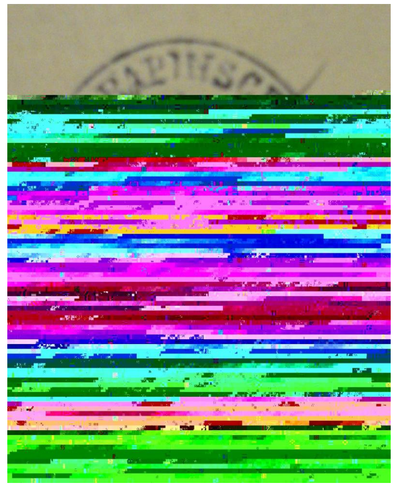


Crash Consistency

- On crash, all the in-memory component of a file system structure is lost.
- Ensuring crash consistency requires that all necessary information to recover the correct state be **persisted in order**.



Metadata Corruption



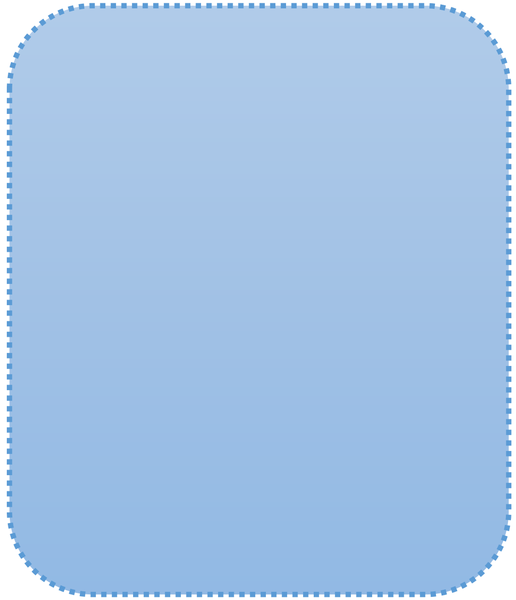
Data Corruption

Unmountable FS

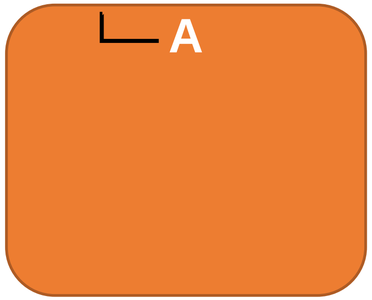
Rename atomicity bug in btrfs

Memory

Storage

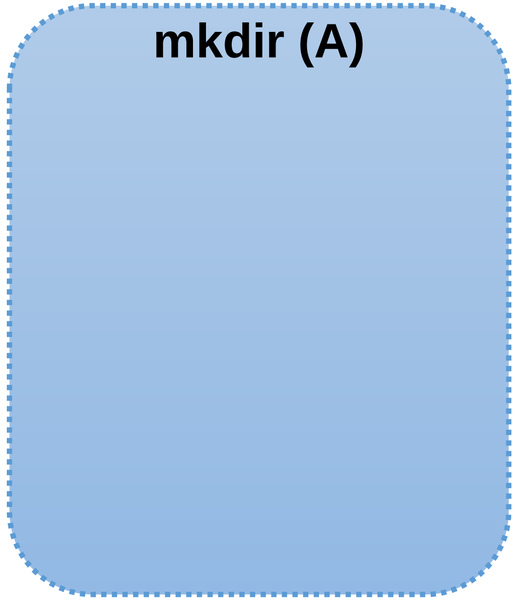


Rename atomicity bug in btrfs

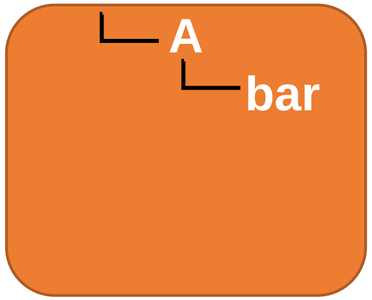


Memory

Storage



Rename atomicity bug in btrfs

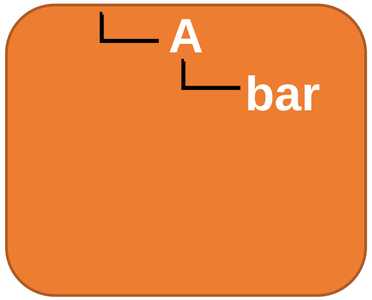


Memory

Storage

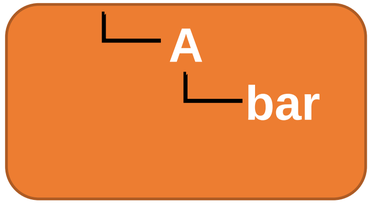
mkdir (A)
touch (A/bar)

Rename atomicity bug in btrfs



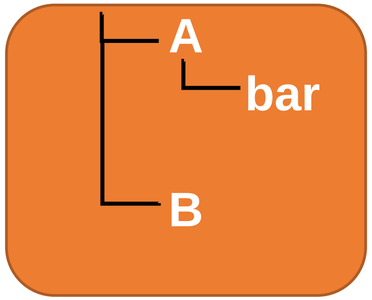
Memory

Storage



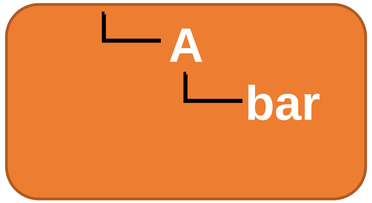
mkdir (A)
touch (A/bar)
fsync (A/bar)

Rename atomicity bug in btrfs



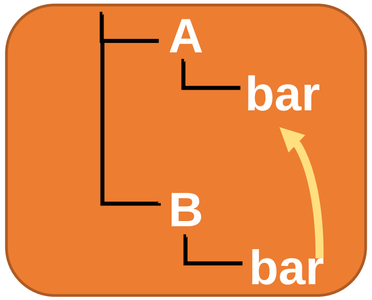
Memory

Storage



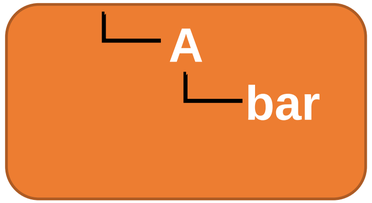
mkdir (A)
touch (A/bar)
fsync (A/bar)
mkdir (B)

Rename atomicity bug in btrfs



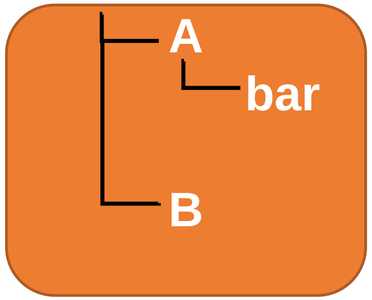
Memory

Storage



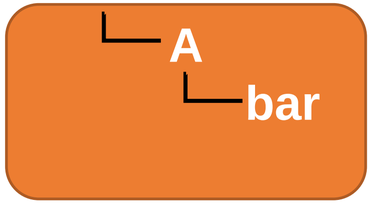
mkdir (A)
touch (A/bar)
fsync (A/bar)
mkdir (B)
touch (B/bar)

Rename atomicity bug in btrfs



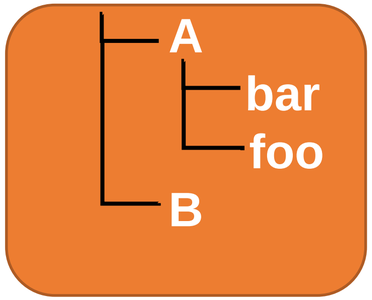
Memory

Storage



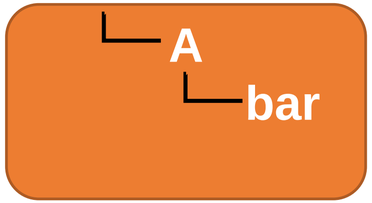
mkdir (A)
touch (A/bar)
fsync (A/bar)
mkdir (B)
touch (B/bar)
rename (B/bar, A/bar)

Rename atomicity bug in btrfs



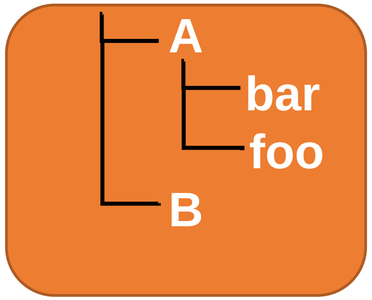
Memory

Storage



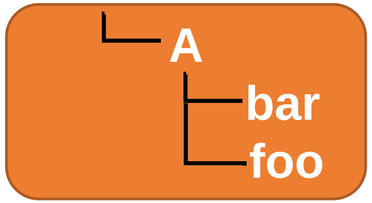
mkdir (A)
touch (A/bar)
fsync (A/bar)
mkdir (B)
touch (B/bar)
rename (B/bar, A/bar)
touch (A/foo)

Rename atomicity bug in btrfs



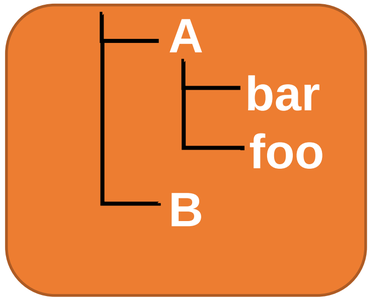
Memory

Storage



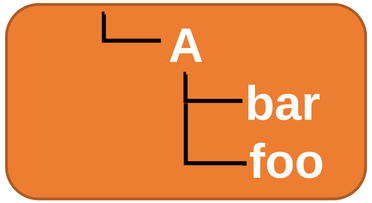
mkdir (A)
touch (A/bar)
fsync (A/bar)
mkdir (B)
touch (B/bar)
rename (B/bar, A/bar)
touch (A/foo)
fsync (A/foo)

Rename atomicity bug in btrfs



Memory

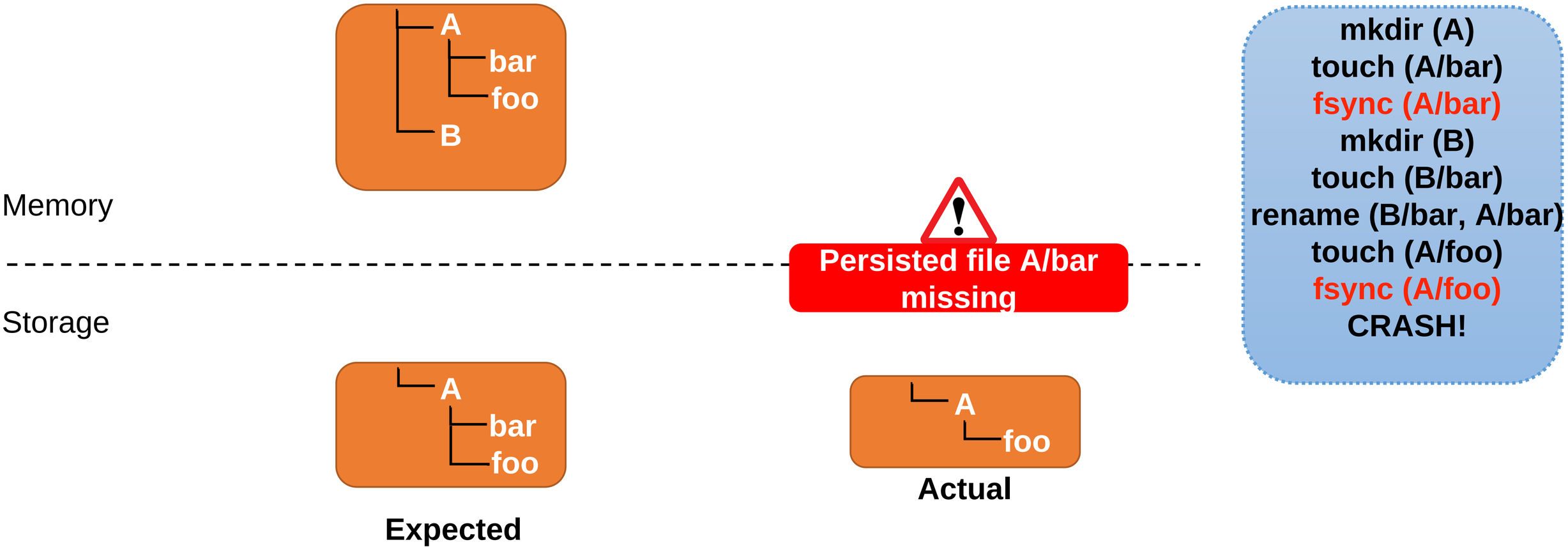
Storage



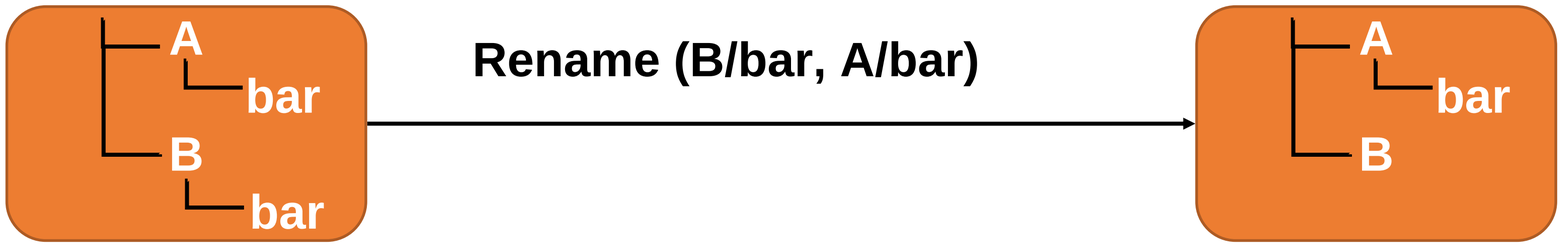
Expected

mkdir (A)
touch (A/bar)
fsync (A/bar)
mkdir (B)
touch (B/bar)
rename (B/bar, A/bar)
touch (A/foo)
fsync (A/foo)
CRASH!

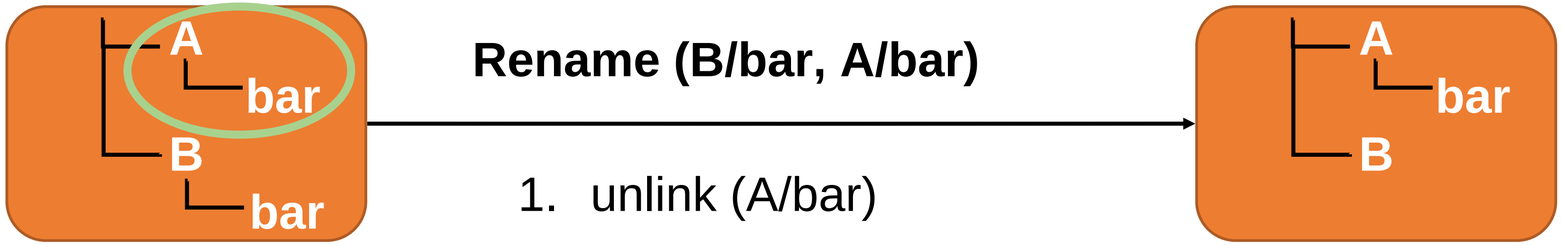
Rename atomicity bug in btrfs



What just happened?



What just happened?

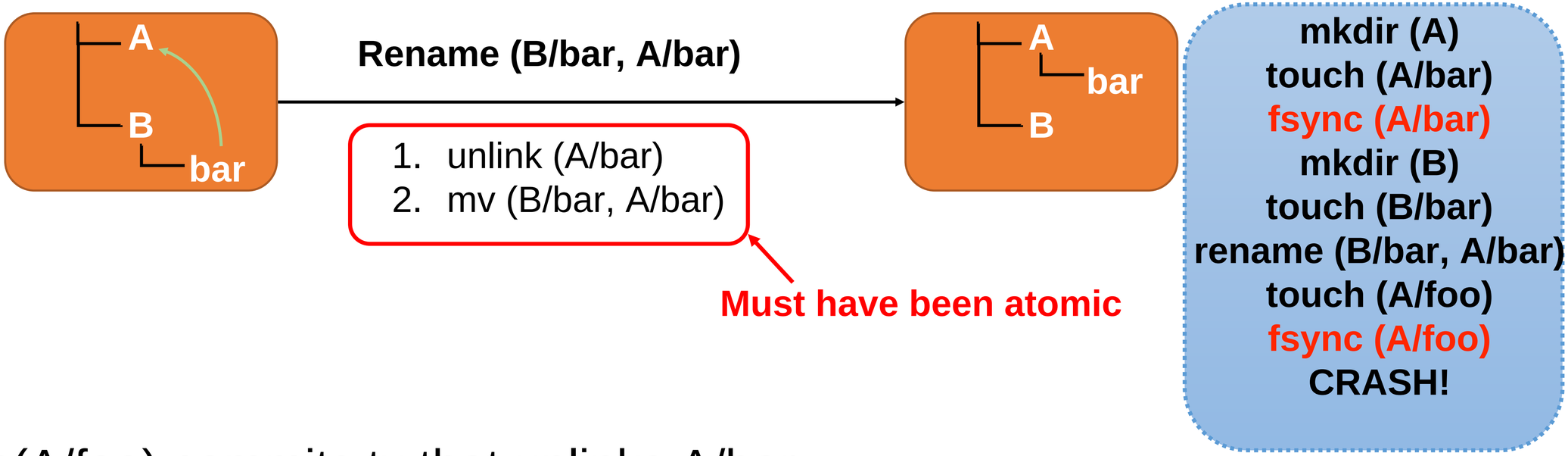


1. unlink (A/bar)

What just happened?

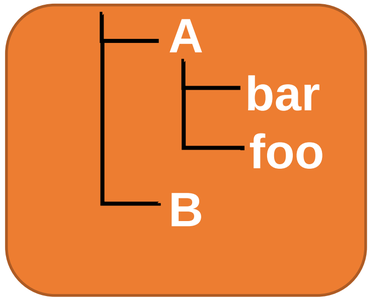
1. unlink (A/bar)
2. mv (B/bar, A/bar)

What just happened?



- fsync(A/foo) commits tx that unlinks A/bar
- Which means step 1 above is persisted, but rename is not persisted
- **End up losing file A/bar**

Rename atomicity bug in btrfs



Memory

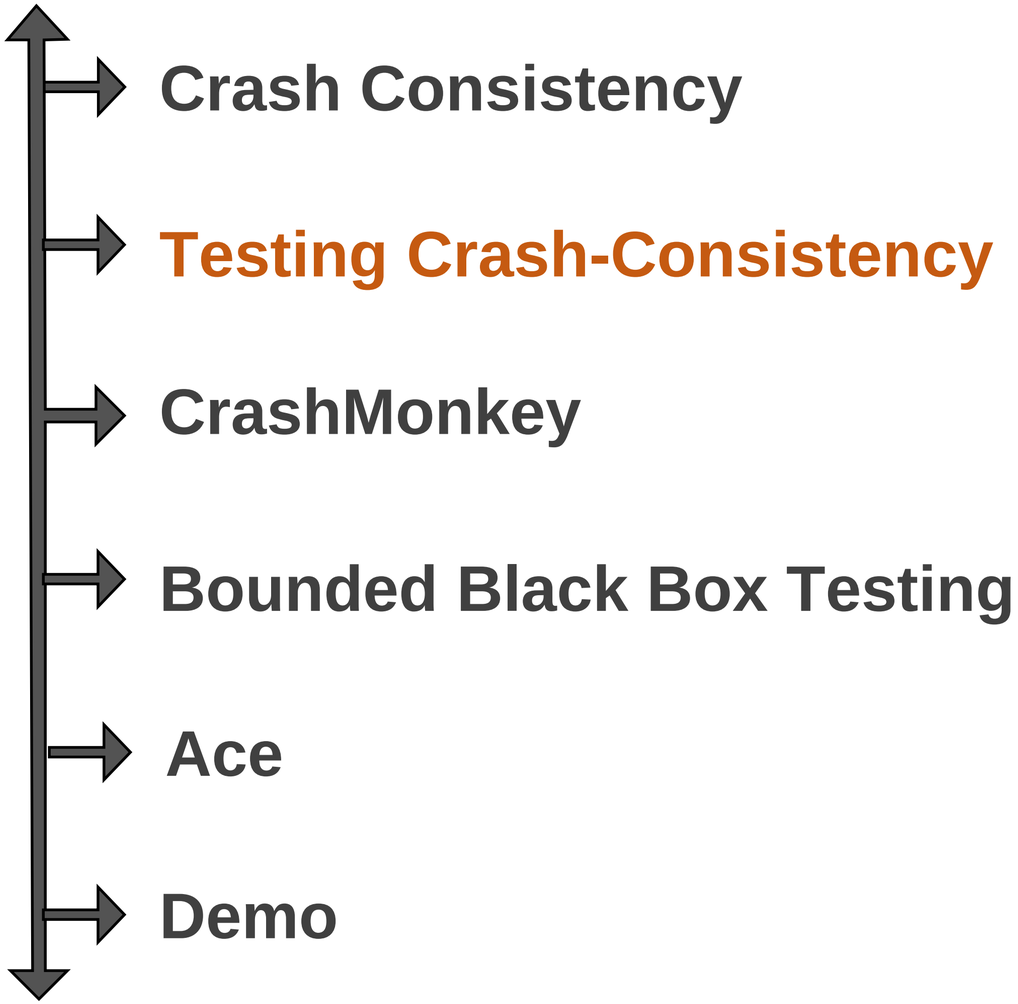
Storage

Persisted file A/bar missing

mkdir (A)
touch (A/bar)
fsync (A/bar)
mkdir (B)
touch (B/bar)
rename (B/bar, A/bar)
touch (A/foo)
fsync (A/foo)
CRASH!

Exists in the kernel since 2014!
Found by ACE and CrashMonkey

Agenda


Crash Consistency
Testing Crash-Consistency
CrashMonkey
Bounded Black Box Testing
Ace
Demo

Testing Crash Consistency Today

- Build FS from scratch

Verified Filesystems

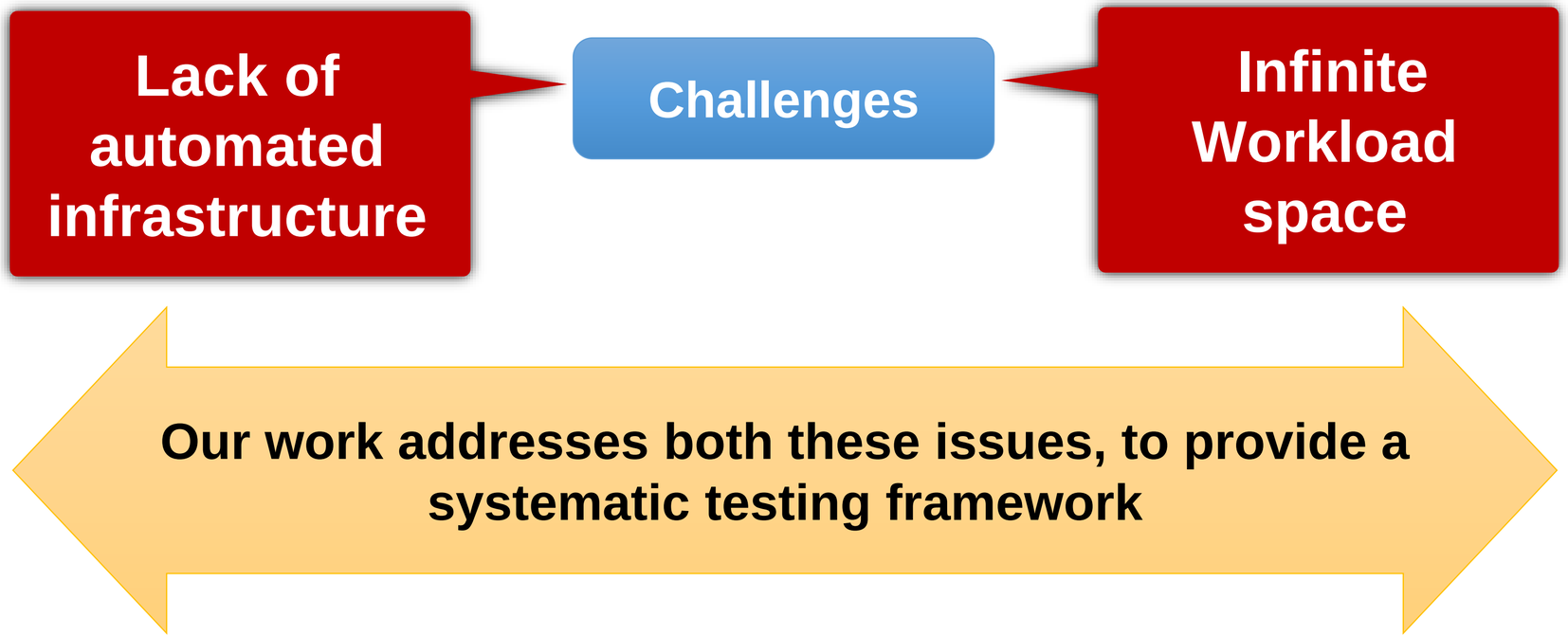
Model Checking

- Annotate filesystems
- Hard to do for existing FS

- State of the Art : xfstest suite
  - Collection of 500 regression tests

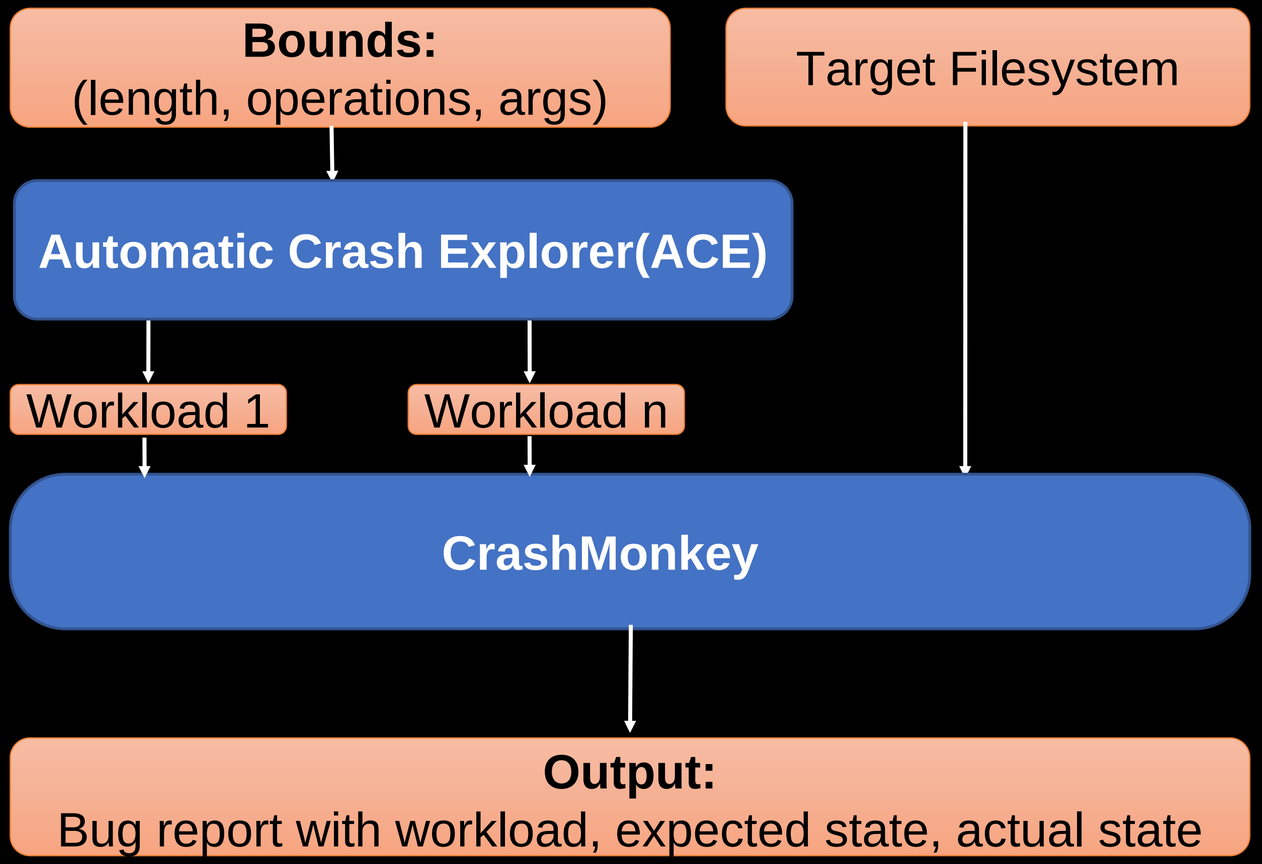
Only 5% of tests in xfstest check for file system crash consistency

Challenges in crash consistency testing



Bounded Black-Box Crash Testing (B^3)

New approach to testing file-system crash consistency



- Focus on reproducible bugs resulting in metadata corruption, data loss.
- Focus on bugs where explicitly persisted data/metadata is corrupted
- Found 10 new bugs across btrfs and F2FS;
- Found 1 bug in FSCQ (verified file system)

www.github.com/utsaslab/crashmonkey

CrashMonkey and Ace : Features

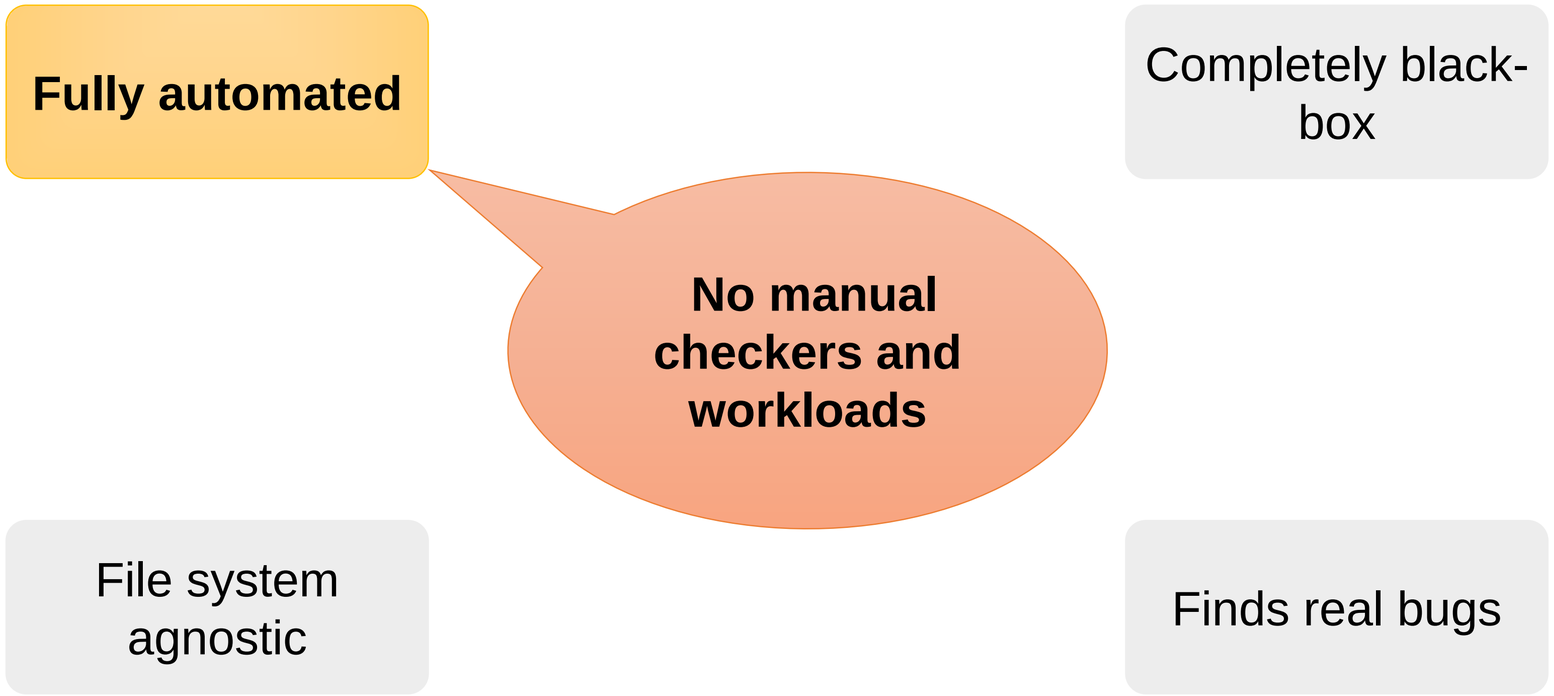
Fully automated

Completely black-box

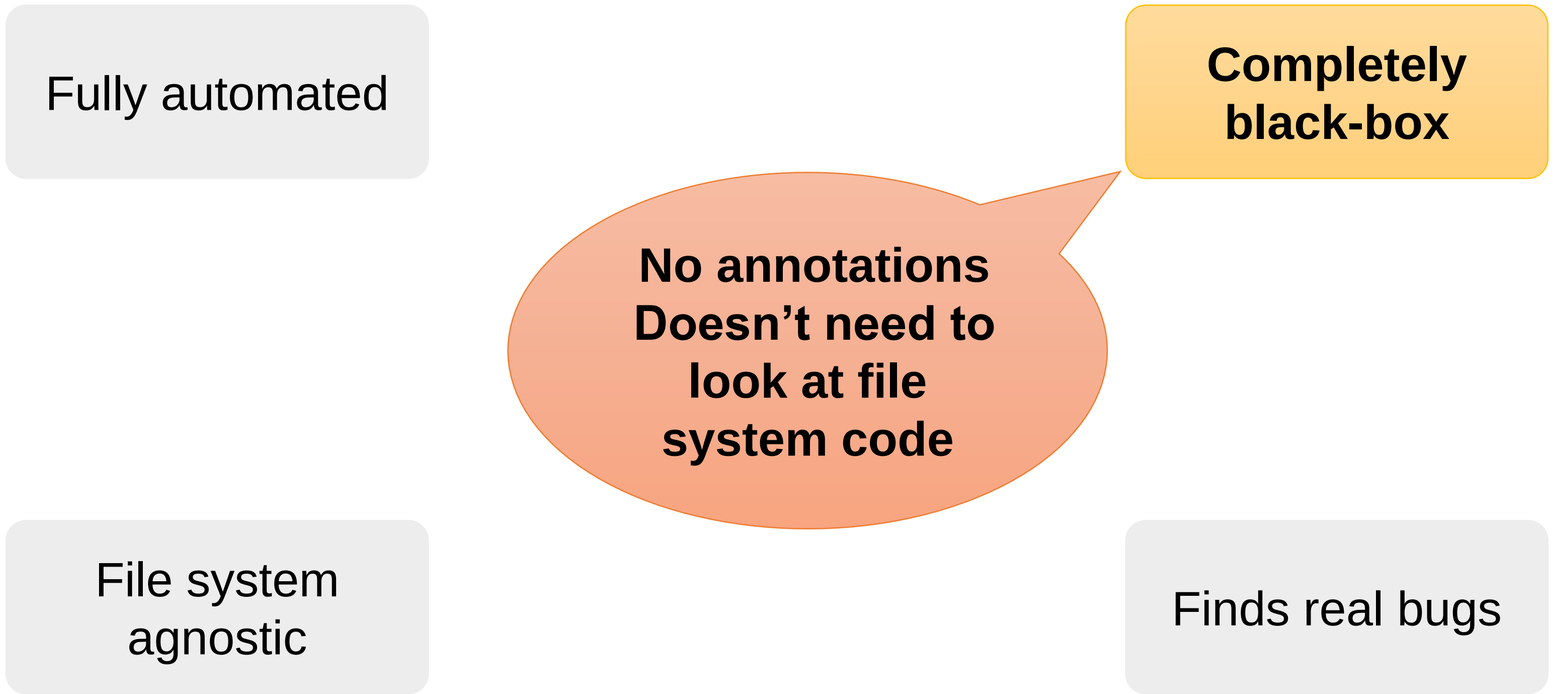
File system agnostic

Finds real bugs

CrashMonkey and Ace : Features



CrashMonkey and Ace : Features



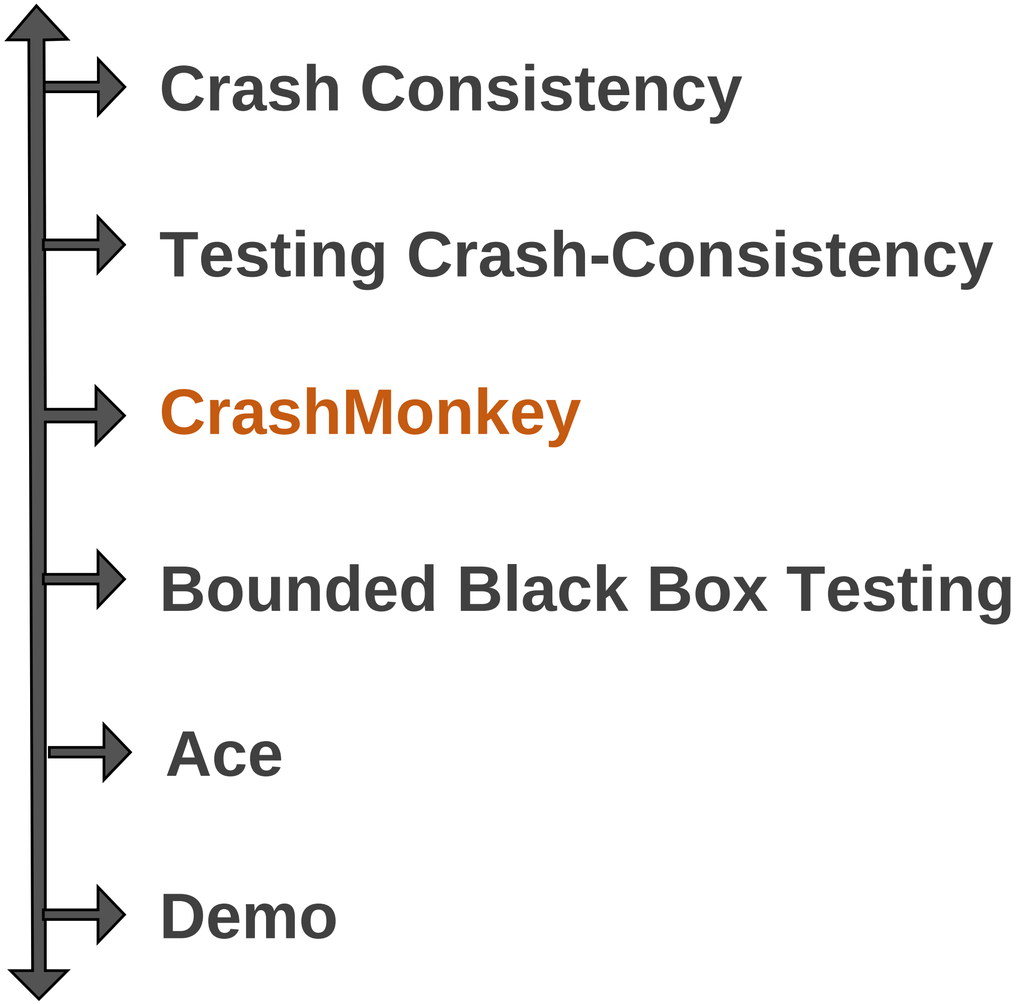
CrashMonkey and Ace : Features

CrashMonkey and Ace : Features

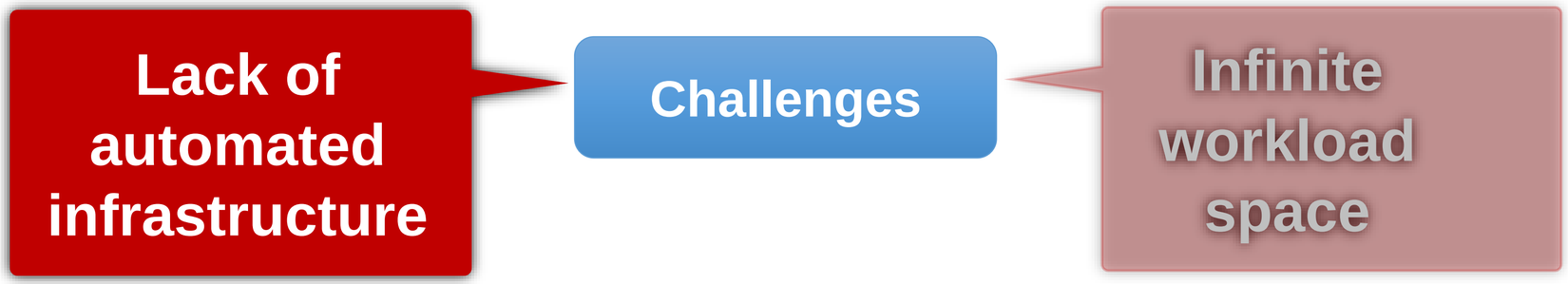
B3 vs other approaches

Metric	Verified FS	Model Check	xfstests	B3
Fully Automated	✗	✗	✗	✓
Black Box	✗	✗	✓	✓
FS agnostic	✗	✗	✓	✓
Find previously unknown bugs	✓	✓	✗	✓

Agenda


Crash Consistency
Testing Crash-Consistency
CrashMonkey
Bounded Black Box Testing
Ace
Demo

Challenges with systematic testing



CrashMonkey

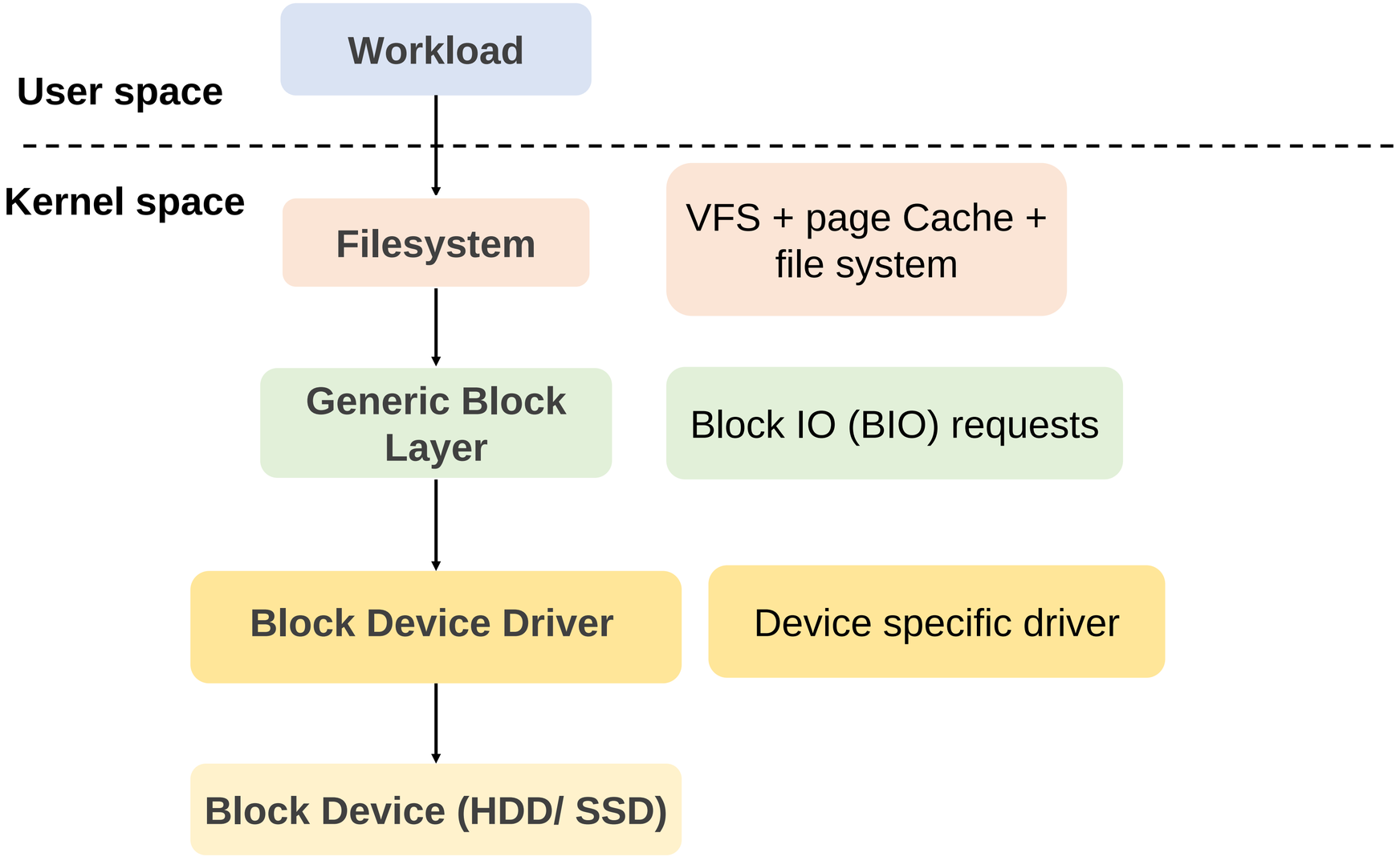
- Efficient infrastructure to record and replay block level IO requests
- Simulate crash at different points in the workload
- Automatically test for consistency after crash.
- Copy-on-write RAM block device

How to crash the file system for testing?

- Actually crash the file system
  1. Randomly power cycle the VM or the server
     - Restarting the VM after crash is slow!
     - Unlikely to reveal bugs
  2. Run the file system in user space
     - Not all file systems can be run as user space processes
     - Redesign file systems

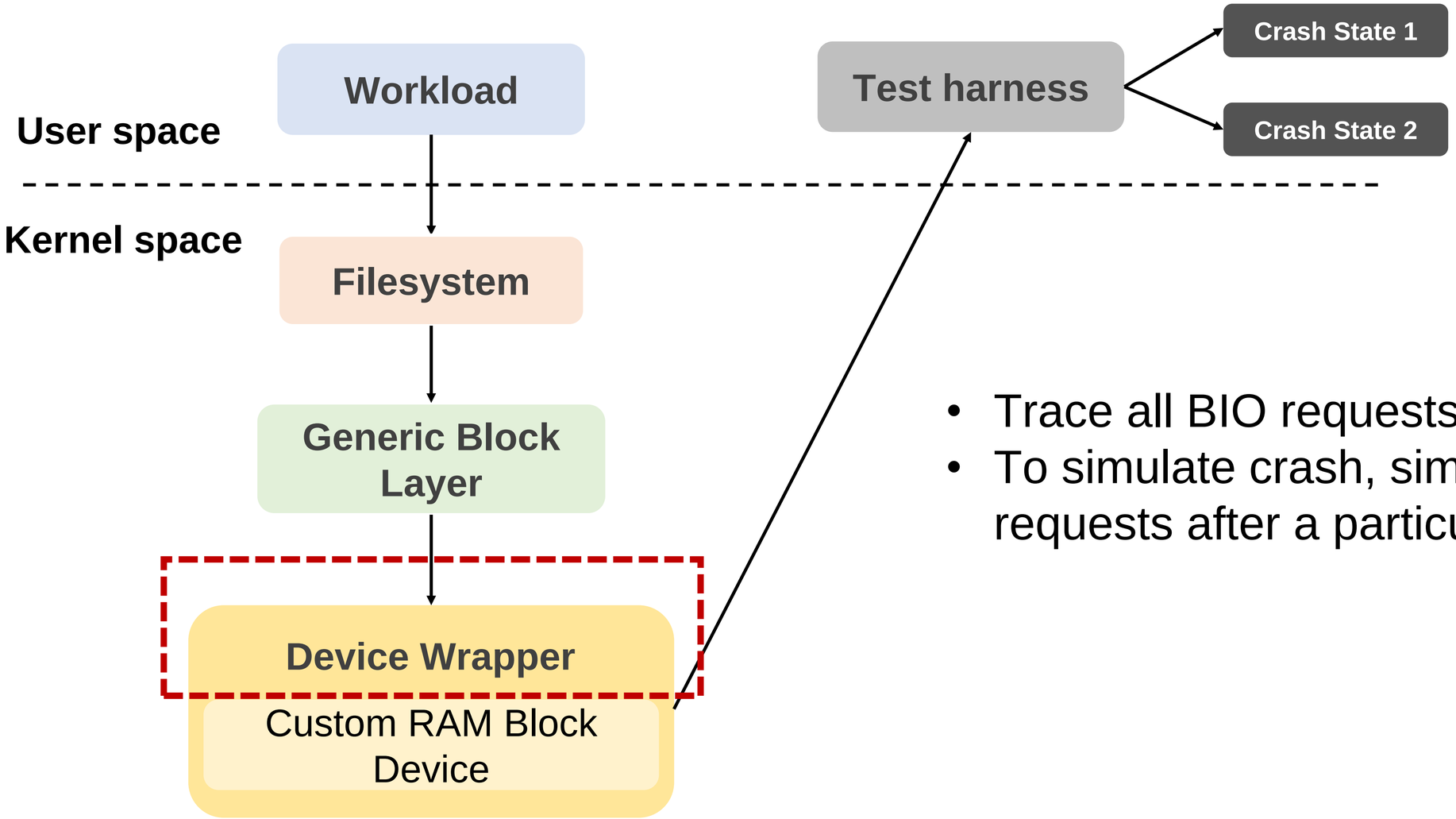
SIMULATE crashes instead of trying to actually crash the system!

How does CrashMonkey simulate crashes?



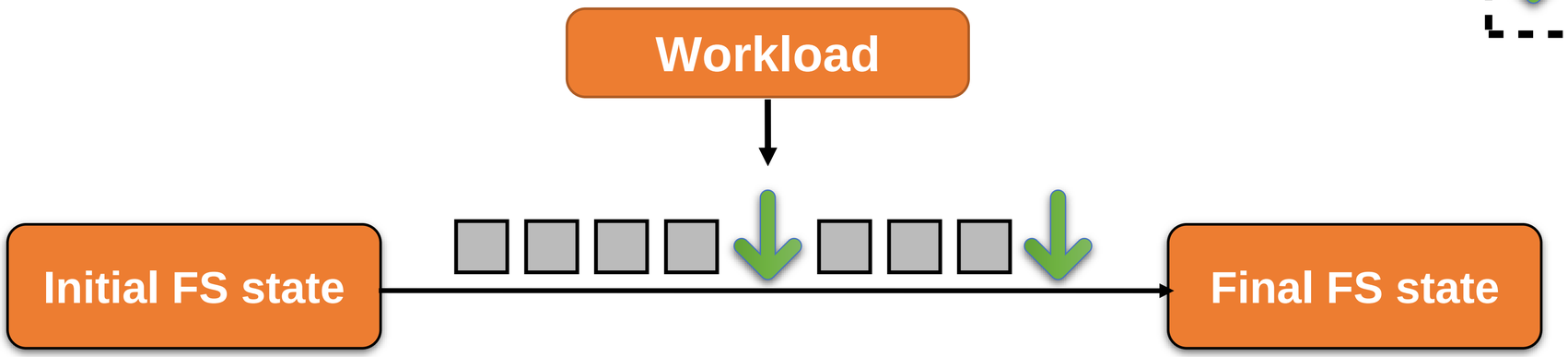
mkdir (A)
touch (A/bar)
fsync (A/bar)
mkdir (B)
touch (B/bar)
rename (B/bar, A/bar)
touch (A/foo)
fsync (A/foo)
CRASH!

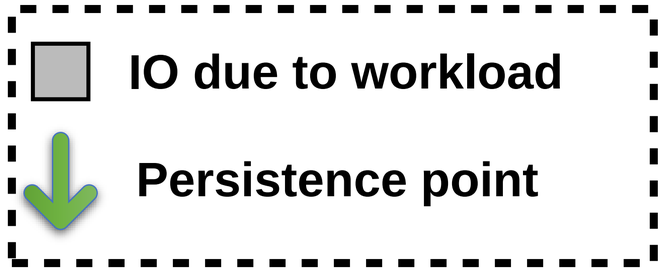
CrashMonkey Architecture



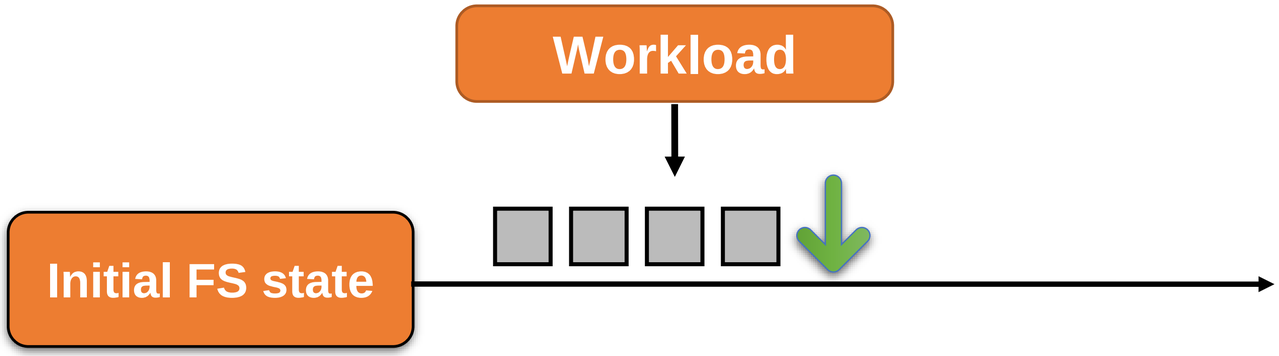
- Trace all BIO requests
- To simulate crash, simply drop the BIO requests after a particular point.

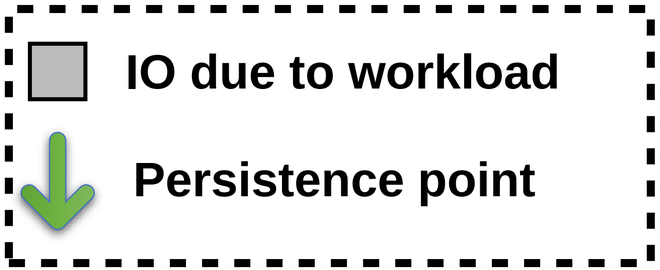
CrashMonkey in Action



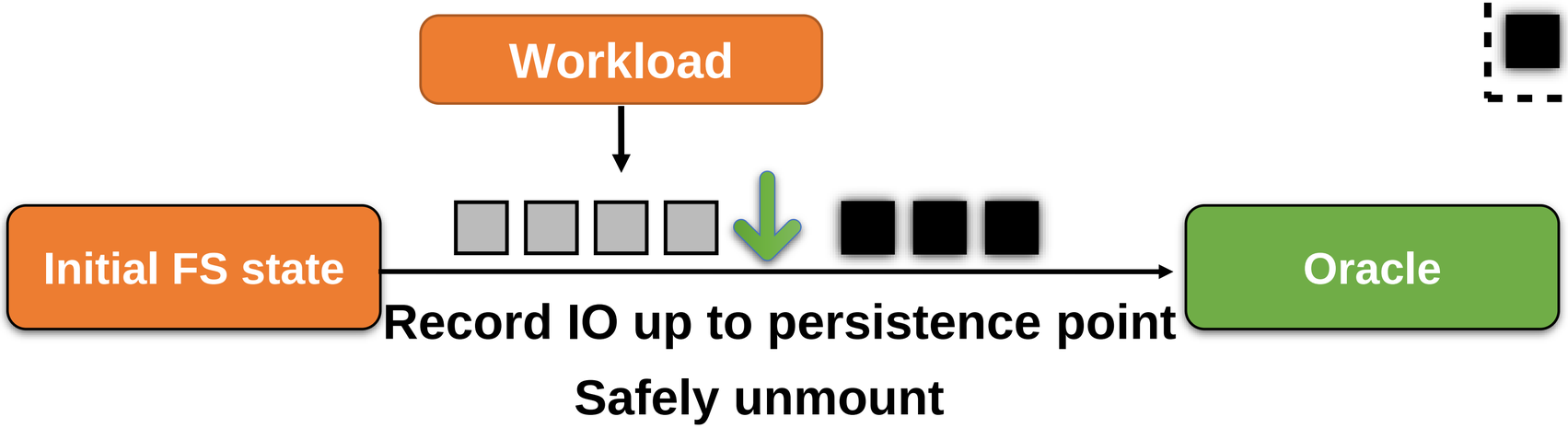


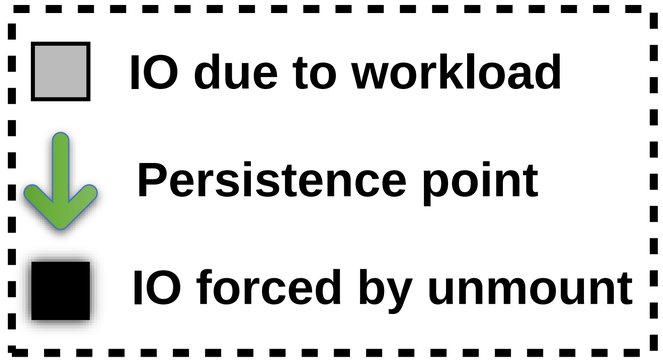
CrashMonkey in Action



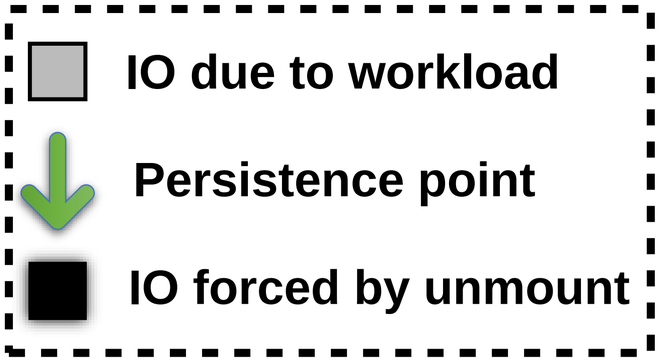


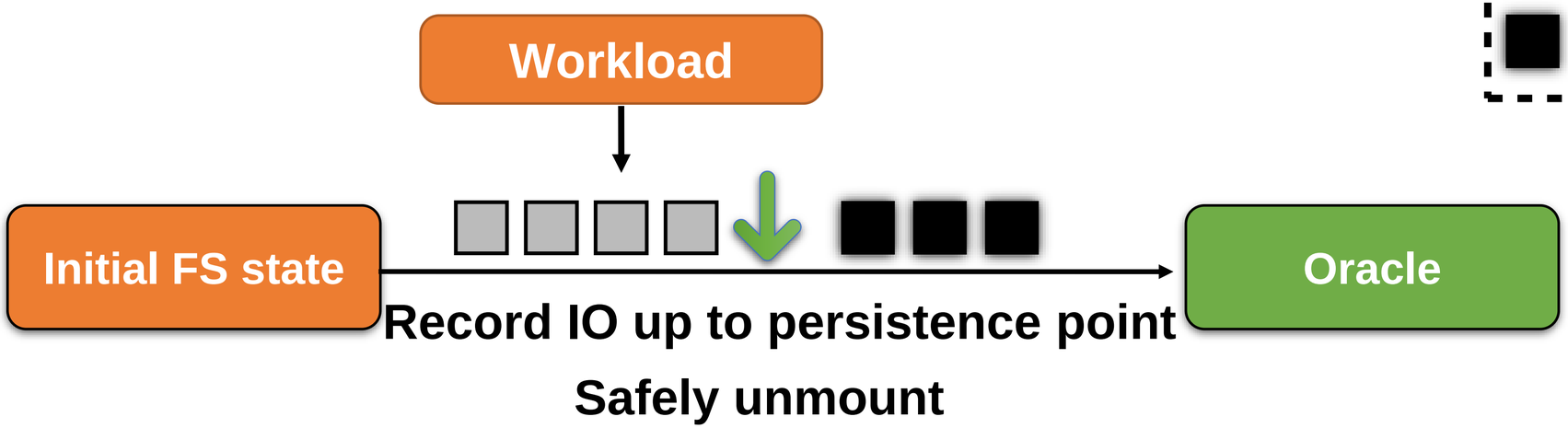
Phase 1 : Record IO

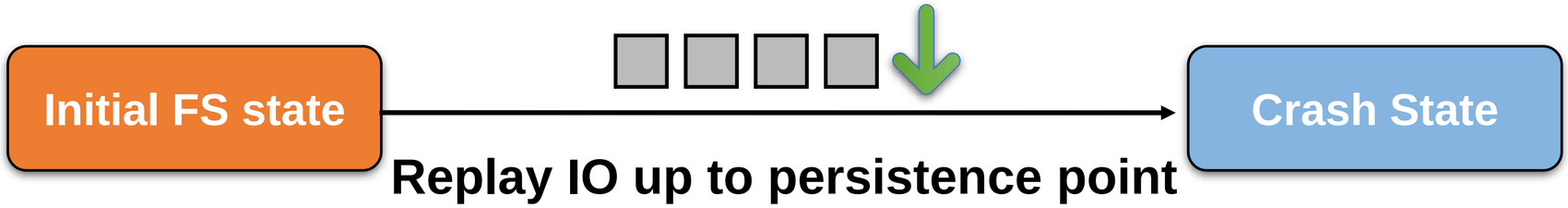




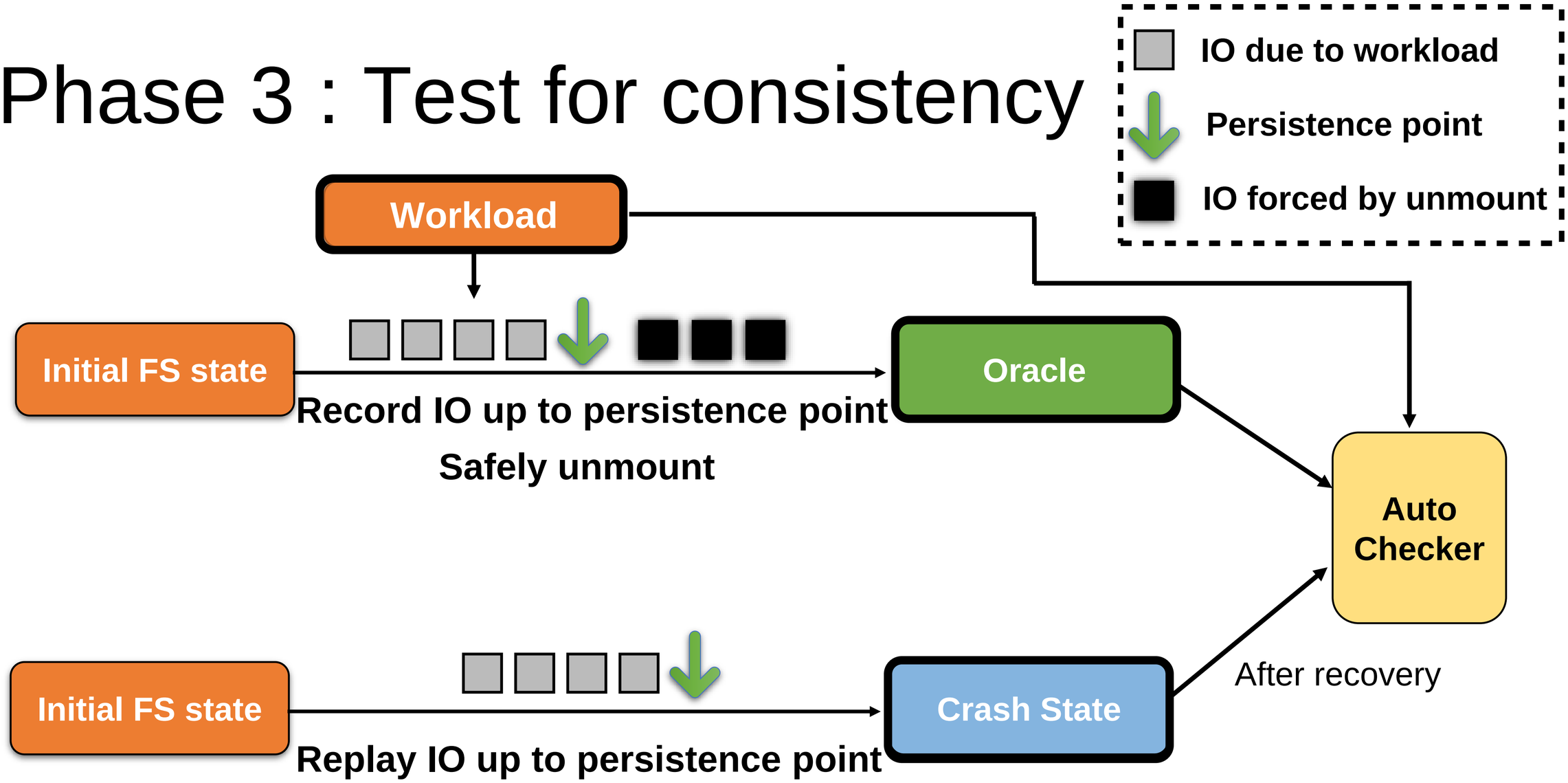
Phase 2 : Replay IO



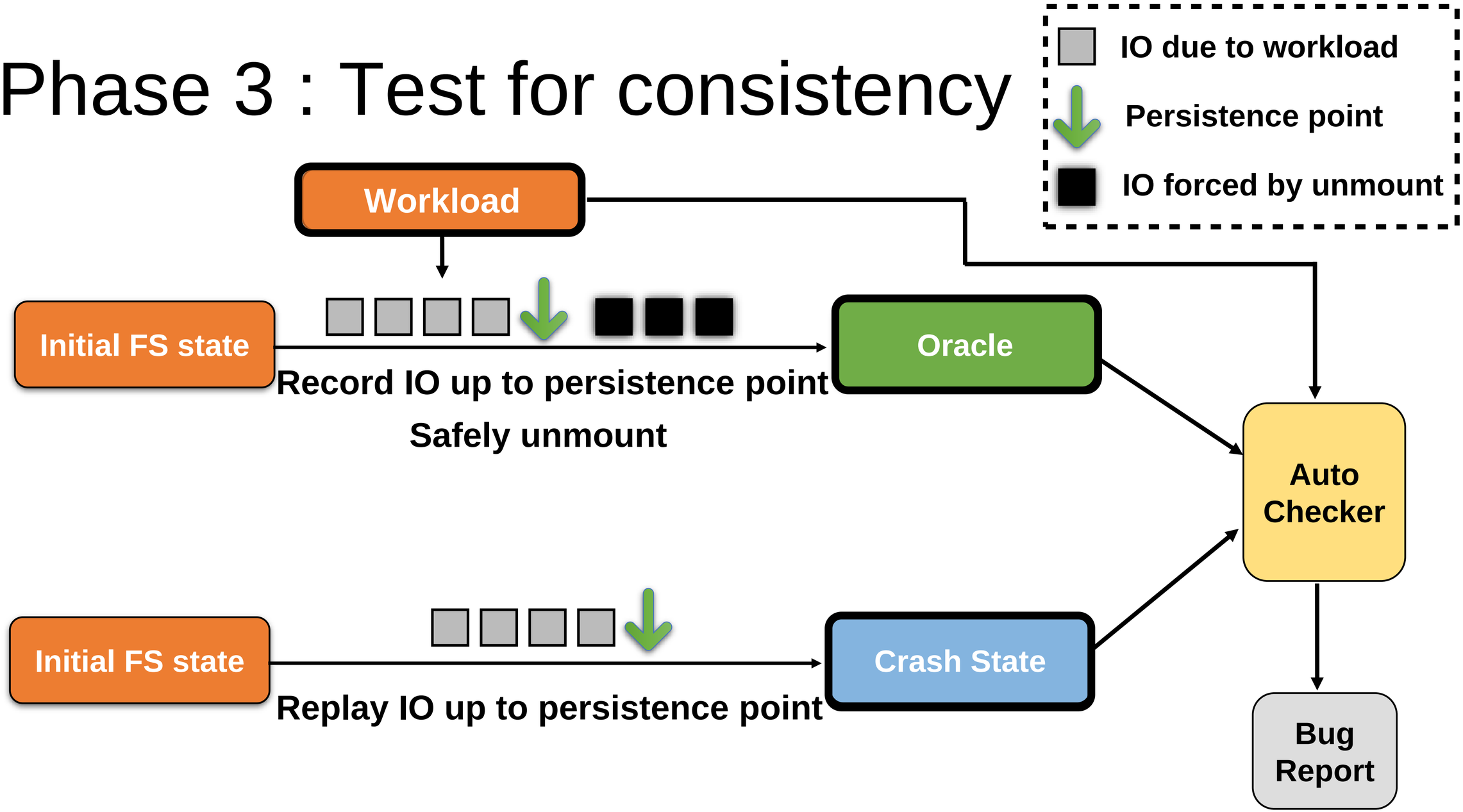




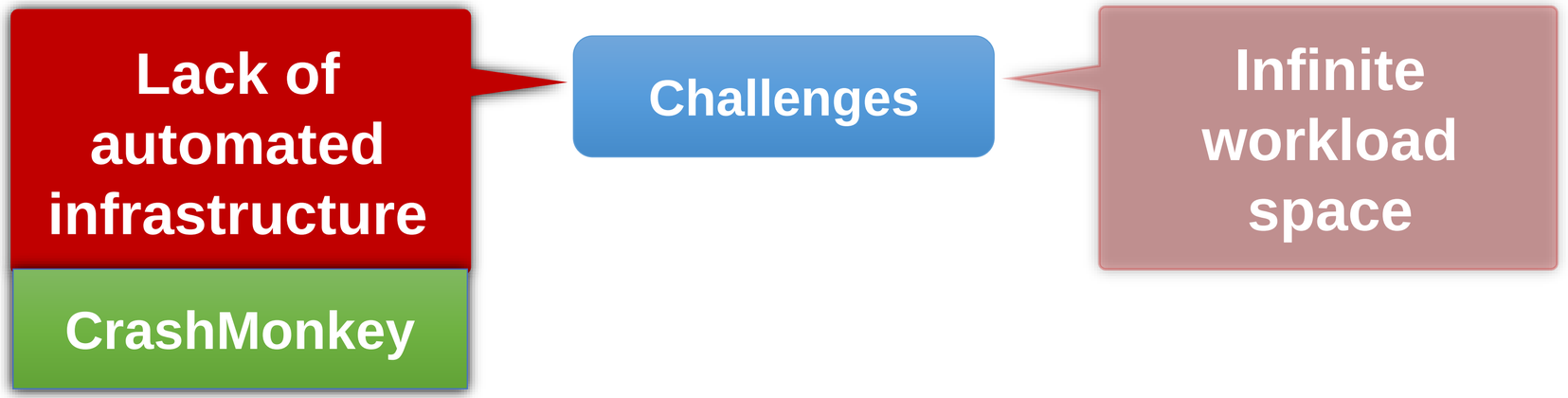
Phase 3 : Test for consistency



Phase 3 : Test for consistency



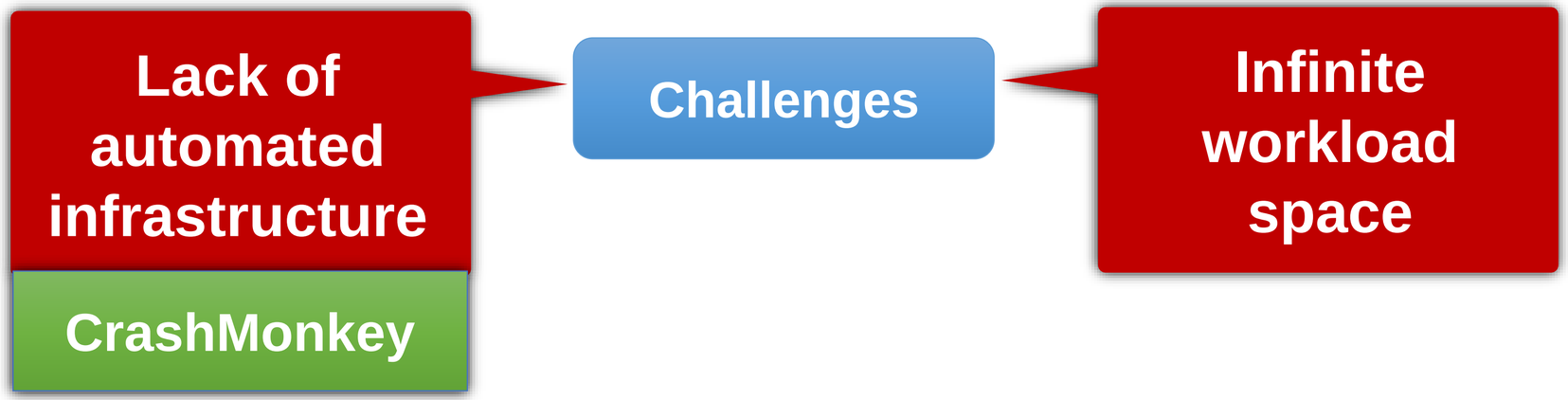
Challenges with Systematic Testing



So Far…

- Given a workload compliant to POSIX API, we saw how CrashMonkey generates crash states and automatically tests for consistency

Challenges with Systematic Testing



So Far…

- Given a workload compliant to POSIX API, we saw how CrashMonkey generates crash states and automatically tests for consistency

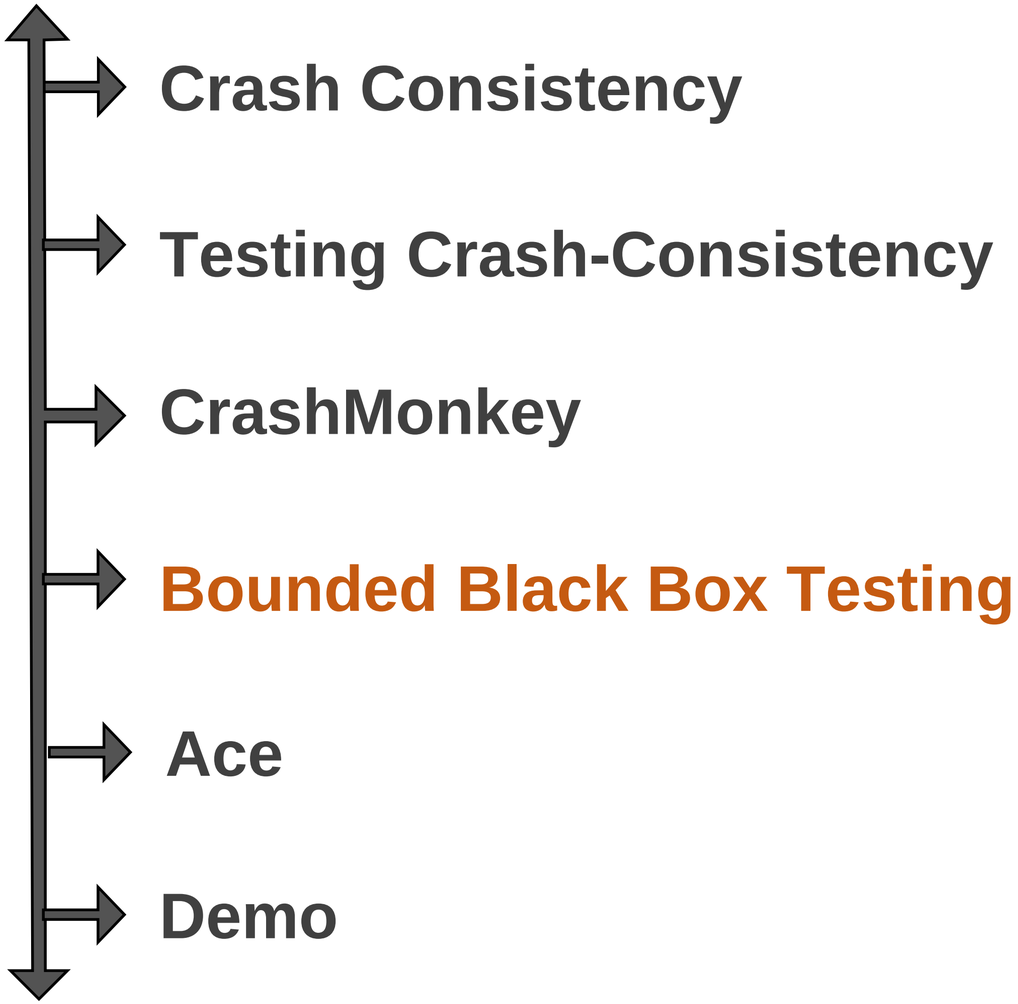
- Next question : How to automatically generate workloads in an the infinite workload space?

Exploring the infinite workload space

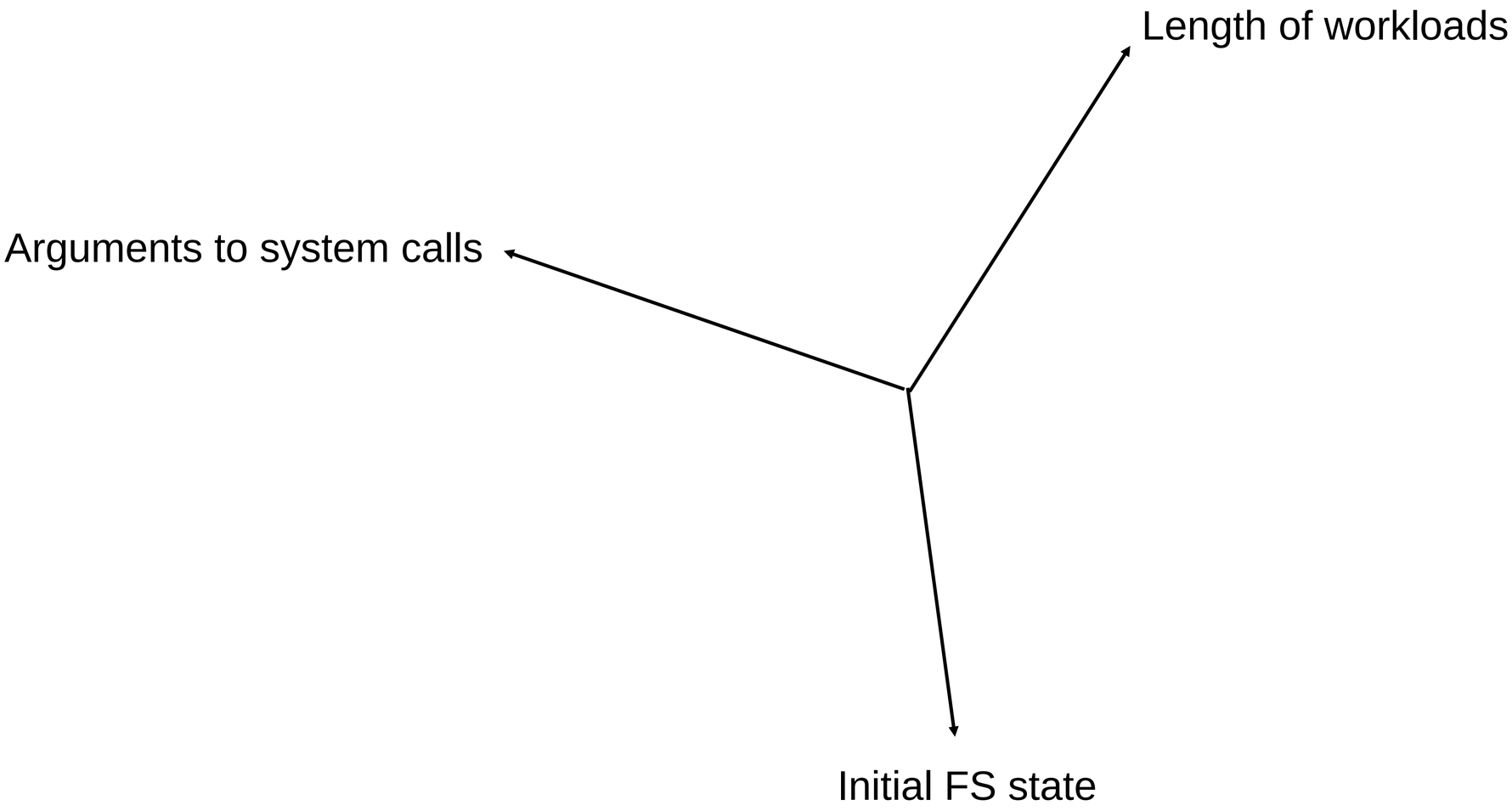
Challenges:

- Infinite length of workloads
- Large set of filesystem operations
- Infinite parameter options (file/directory names, depth)
- Infinite options for initial filesystem state
- When in the workload to simulate a crash?

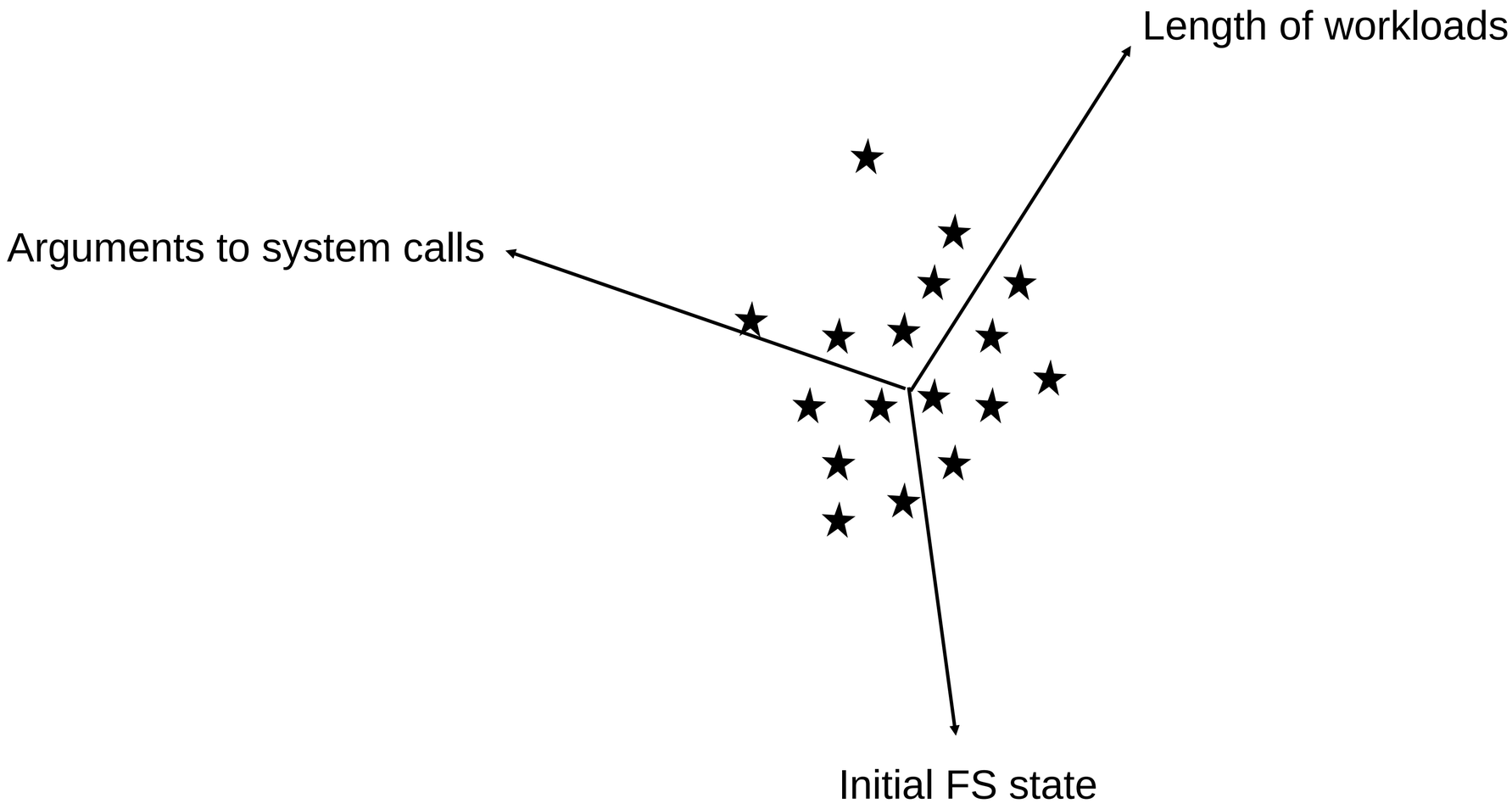
Agenda


Crash Consistency
Testing Crash-Consistency
CrashMonkey
Bounded Black Box Testing
Ace
Demo

B^3 : Bounded Black Box Crash Testing



B^3 : Bounded Black Box Crash Testing



B^3 : Bounded Black Box Crash Testing

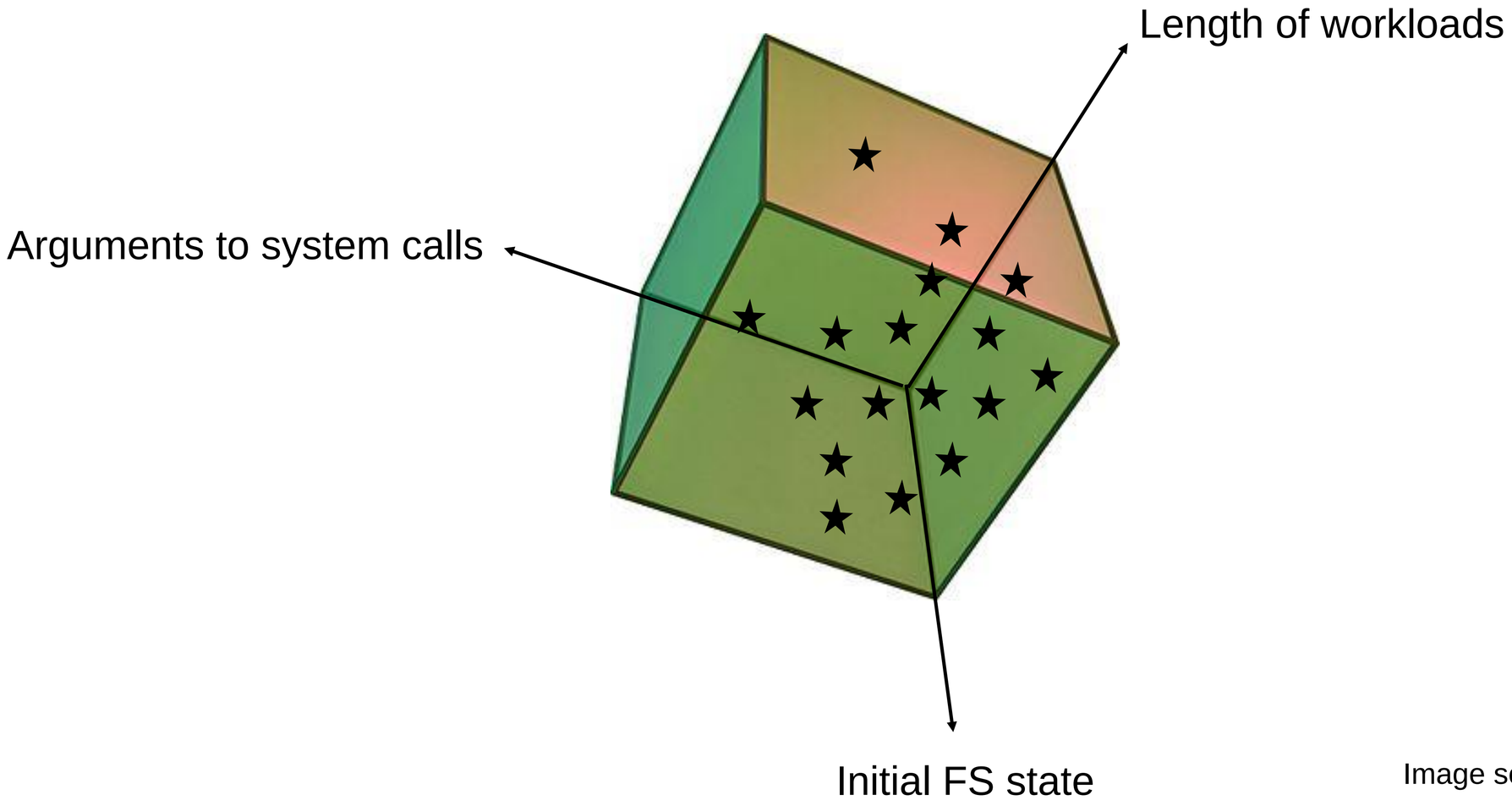
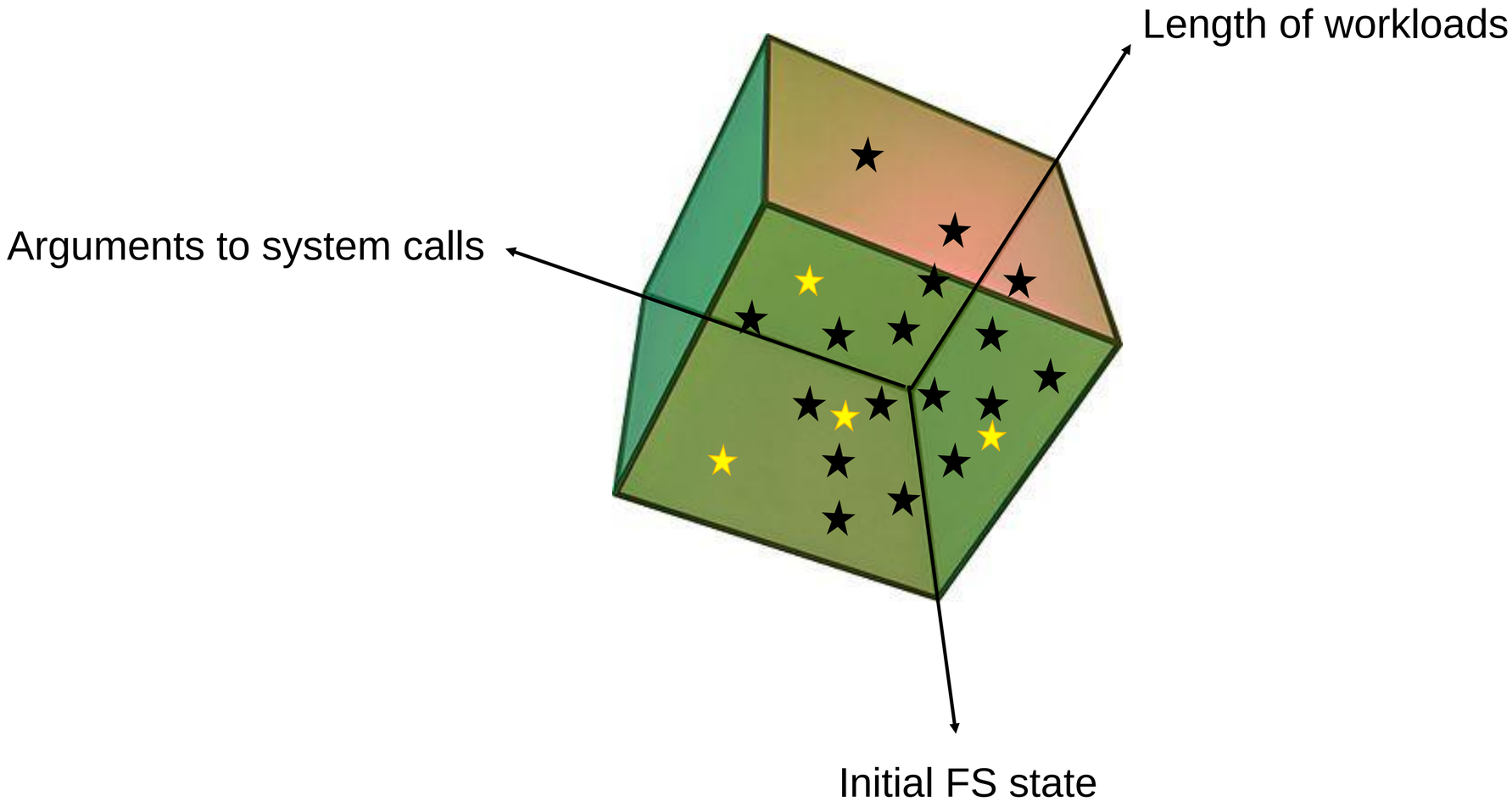
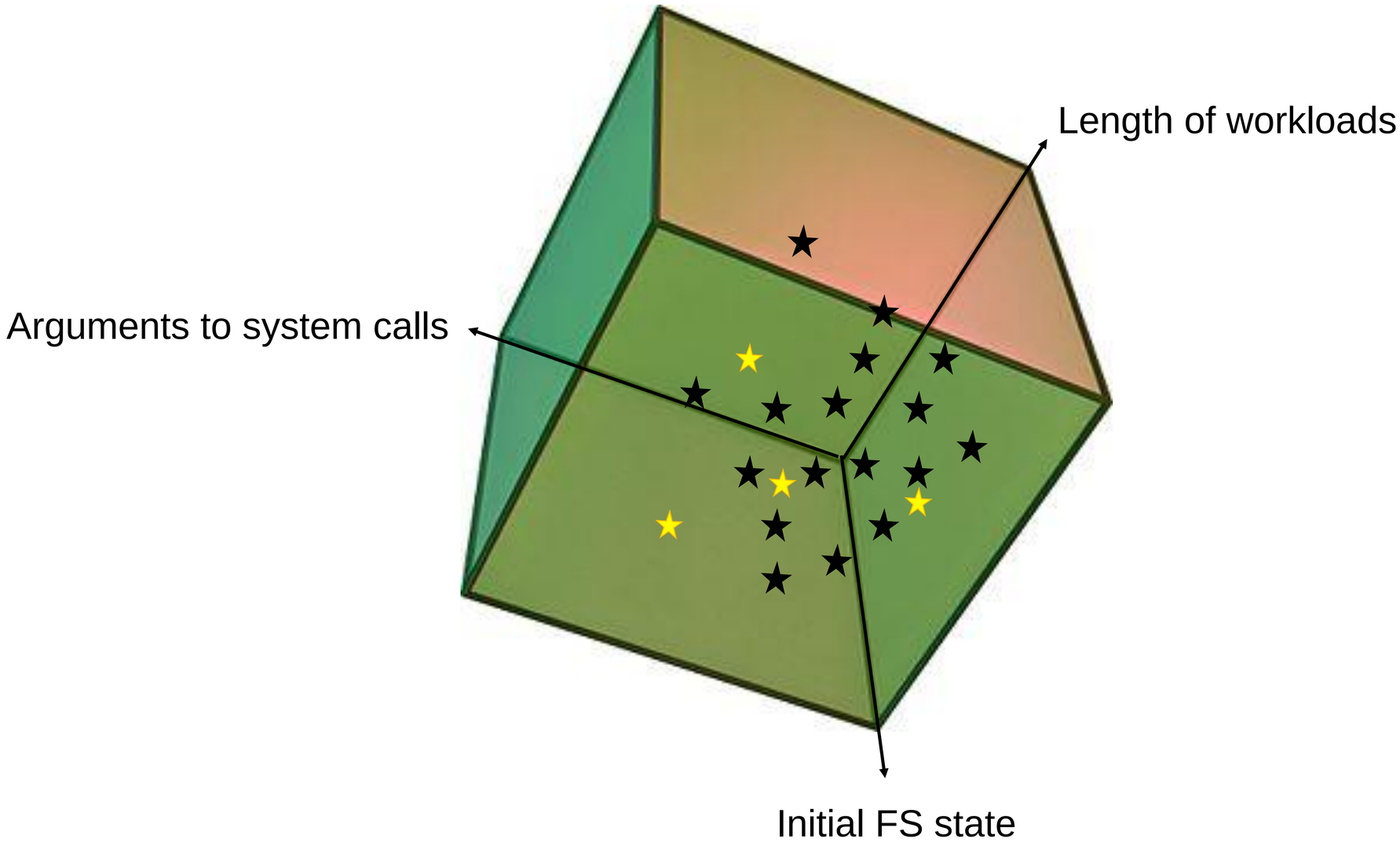


Image source: https://en.wikipedia.org/wiki/Cube

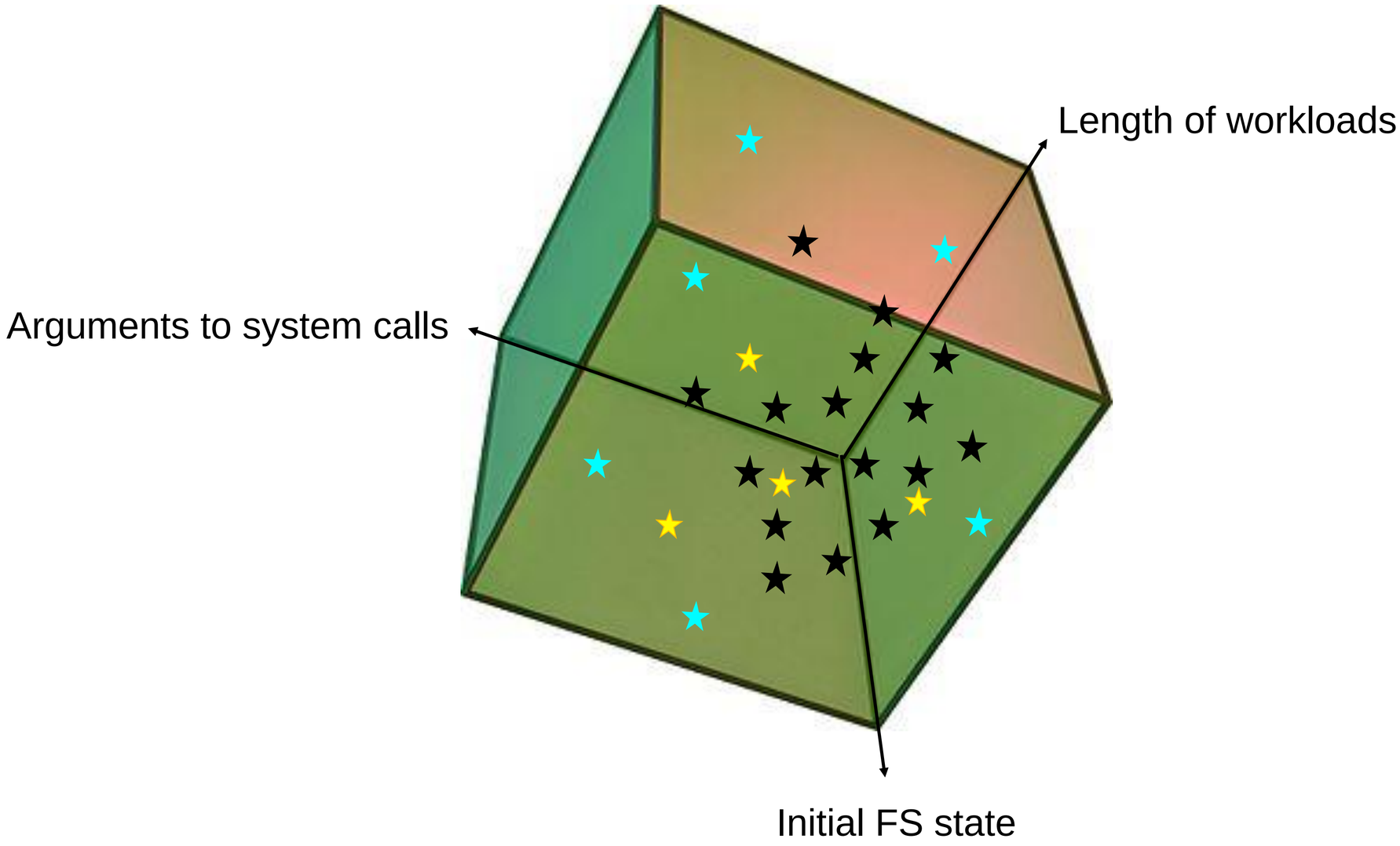
B^3 : Bounded Black Box Crash Testing



B^3 : Bounded Black Box Crash Testing



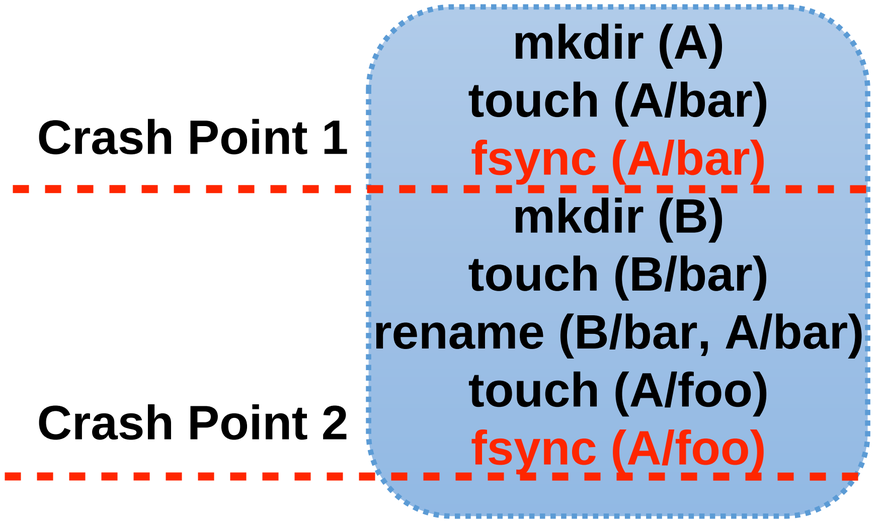
B^3 : Bounded Black Box Crash Testing



B^3 : Bounded Black Box Crash Testing

Choice of crash point

- Only after fsync(), fdatasync() or sync()
- Not in the middle of system call

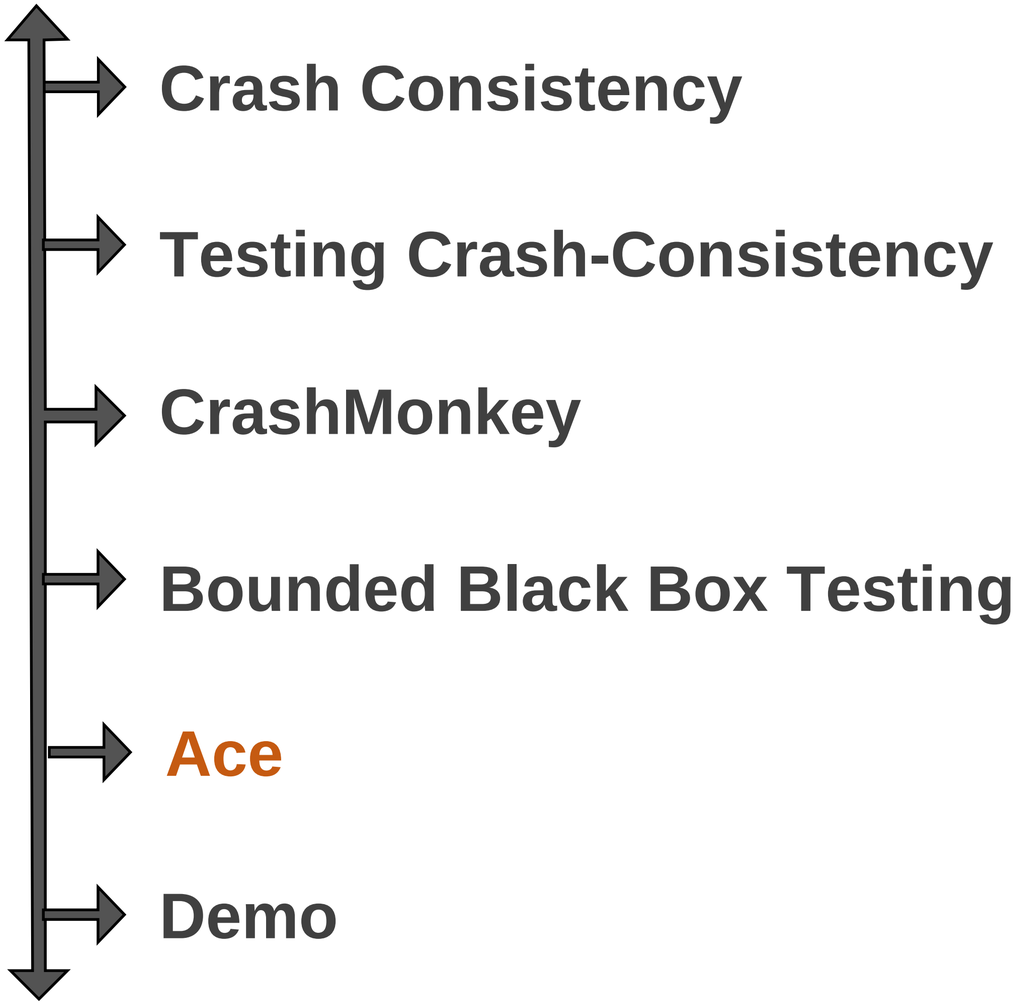


- Developers are motivated to patch bugs that break semantics of persistence operations
- Crashing in the middle of system calls leads to exponentially large crash-states.

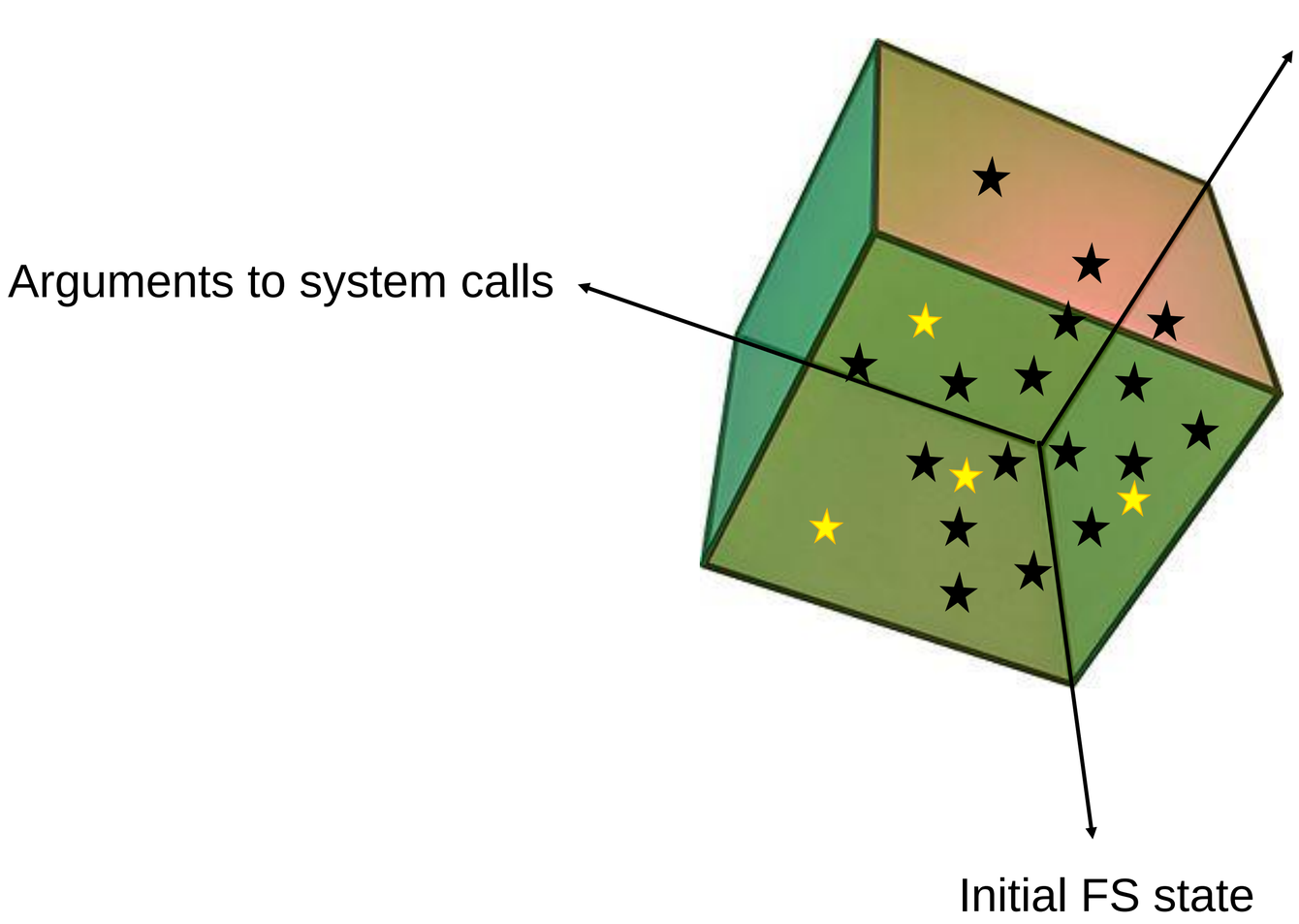
Limitations of B^3

- No guarantee of finding all crash-consistency bugs in a filesystem
- Assumes the correct working of crash-consistency mechanism like journaling or CoW
  - Does not crash in the middle of system calls
- Can only reveal if a bug has occurred, not the reason or origin of bug.
- Needs larger compute to test higher sequence lengths

Agenda


Crash Consistency
Testing Crash-Consistency
CrashMonkey
Bounded Black Box Testing
Ace
Demo

Bounds chosen by ACE



Bounds picked based on insights from the study of crash-consistency bugs reported on Linux file systems over the last 5 years

Study of crash consistency bugs in the wild

- Study the workload pattern and impacts of crash consistency bugs reported in the **past 5 years**
  - Kernel mailing lists
  - Crash consistency tests submitted to xfstests

- **26 unique bugs** across ext4, F2FS, and btrfs

Study of crash consistency bugs in the wild

Consequence	# bugs
Corruption	17
Data inconsistency	6
Unmountable FS	3
Total	26

Filesystem	# bugs
Ext4	2
F2FS	2
btrfs	24
Total	28

# ops	# bugs
1	3
2	14
3	9
Total	26

Study of crash consistency bugs in the wild

Consequence	# bugs
Corruption	17
Data inconsistency	6
Unmountable FS	3
Total	26

Filesystem	# bugs
Ext4	2
F2FS	2
btrfs	24
Total	28

# ops	# bugs
1	3
2	14
3	9
Total	26

1. Crash consistency bugs are hard to find

- Bugs have been around in the kernel for up to 7 years before being identified and patched
- Usually involve reuse of files/ directories

Study of crash consistency bugs in the wild

Consequence	# bugs
Corruption	17
Data inconsistency	6
Unmountable FS	3
Total	26

Filesystem	# bugs
Ext4	2
F2FS	2
btrfs	24
Total	28

# ops	# bugs
1	3
2	14
3	9
Total	26

1. Crash consistency bugs are hard to find
2. **Small workloads are sufficient to reveal bugs**
   - 2-3 core operations on a new, empty file-system

Study of crash consistency bugs in the wild

Consequence	# bugs
Corruption	17
Data inconsistency	6
Unmountable FS	3
Total	26

Filesystem	# bugs
Ext4	2
F2FS	2
btrfs	24
Total	28

# ops	# bugs
1	3
2	14
3	9
Total	26

1. Crash consistency bugs are hard to find
2. Small workloads are sufficient to reveal bugs
3. **Crash after persistence points**
   - Sufficient to crash after a call to fsync(), fdatasync(), or sync()

Study of crash consistency bugs in the wild

Consequence	# bugs
Corruption	17
Data inconsistency	6
Unmountable FS	3
Total	26

Filesystem	# bugs
Ext4	2
F2FS	2
btrfs	24
Total	28

# ops	# bugs
1	3
2	14
3	9
Total	26

1. Crash consistency bugs are hard to find
2. Small workloads are sufficient to reveal bugs
3. Crash after persistence points
4. **Systematic testing is required**

Study of crash consistency bugs in the wild

Consequence	# bugs
Corruption	17
Data inconsistency	6
Unmountable FS	3
Total	26

Filesystem	# bugs
Ext4	2
F2FS	2
btrfs	24
Total	28

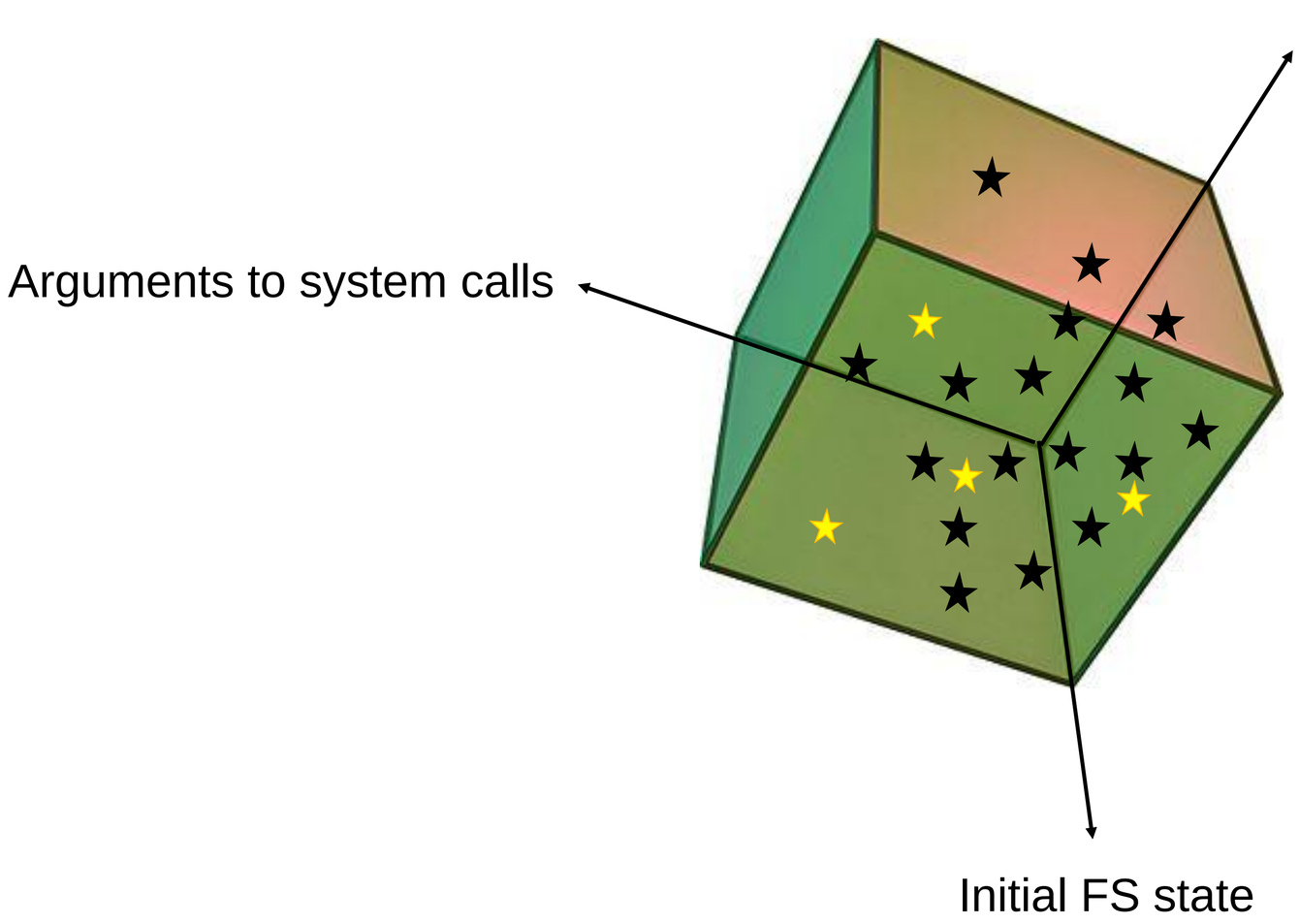
# ops	# bugs
1	3
2	14
3	9
Total	26

1. Crash consistency bugs are hard to find
2. Small workloads are sufficient to reveal bugs
3. Crash after persistence points
4. **Systematic testing is required**

Fallocate : punch_hole : 2015

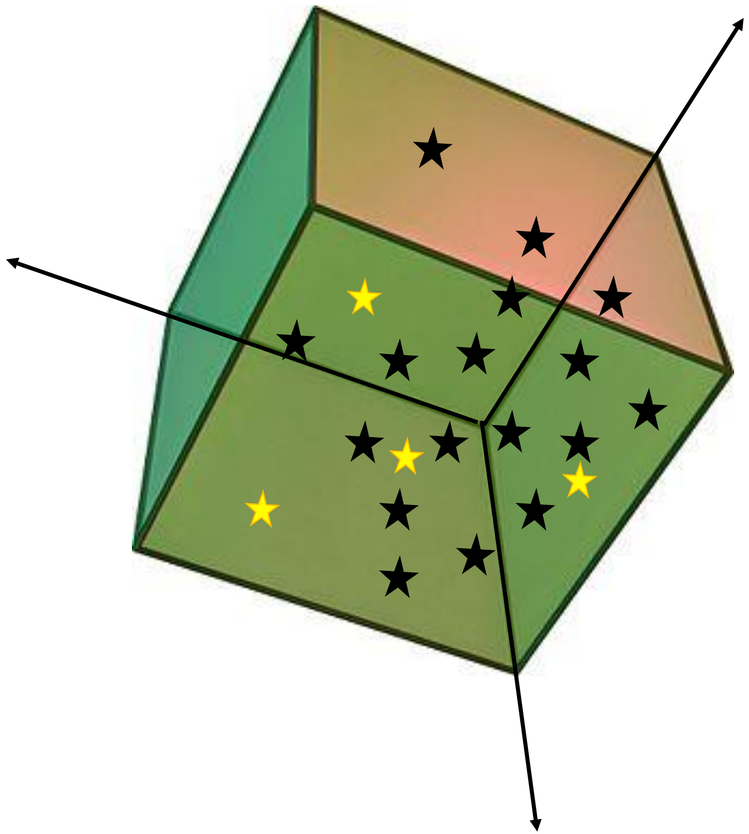
Fallocate : zero_range : 2018

Bounds chosen by ACE



Bounds picked based on insights from the study of crash-consistency bugs reported on Linux file systems over the last 5 years

Bounds chosen by ACE



Length of workloads

Maximum # core ops is 3

Arguments to system calls

Initial FS state

Bounds chosen by ACE

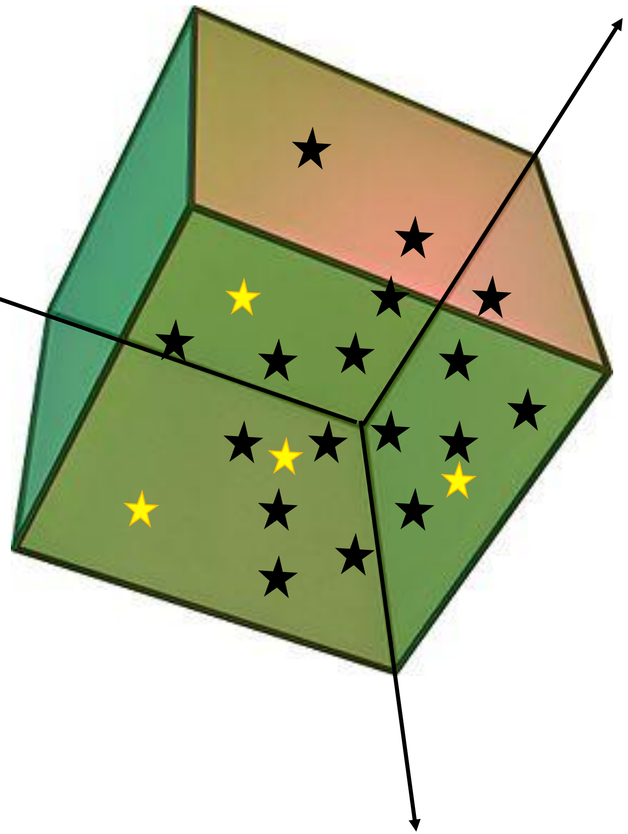
Length of workloads

Maximum # core ops is 3

Arguments to system calls

Root
- **A → (foo, bar)**
- **B → (foo, bar)**

Overwrites to start, middle, end of a file and append



Initial FS state

Bounds chosen by ACE

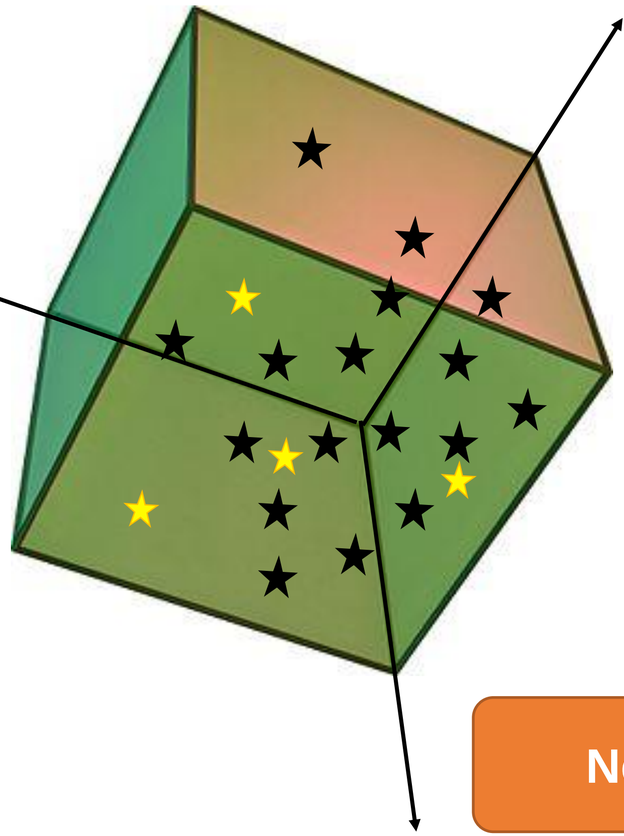
Length of workloads

Maximum # core ops is 3

Arguments to system calls

Root
- **A → (foo, bar)**
- **B → (foo, bar)**

Overwrites to start, middle, end and append



New, 100MB FS

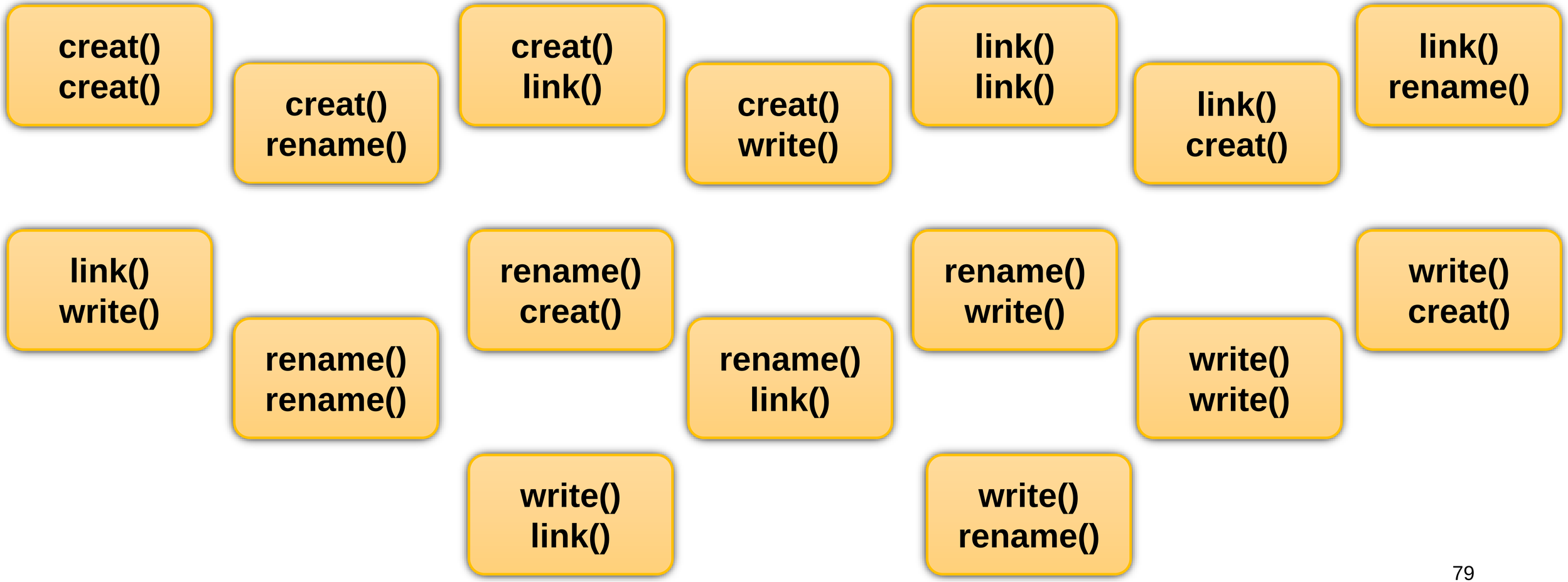
Initial FS state

Phases of ACE

Operation Set

creat()
link()
rename()
write()

Generating skeletons of sequence-2. : 4*4 = 16

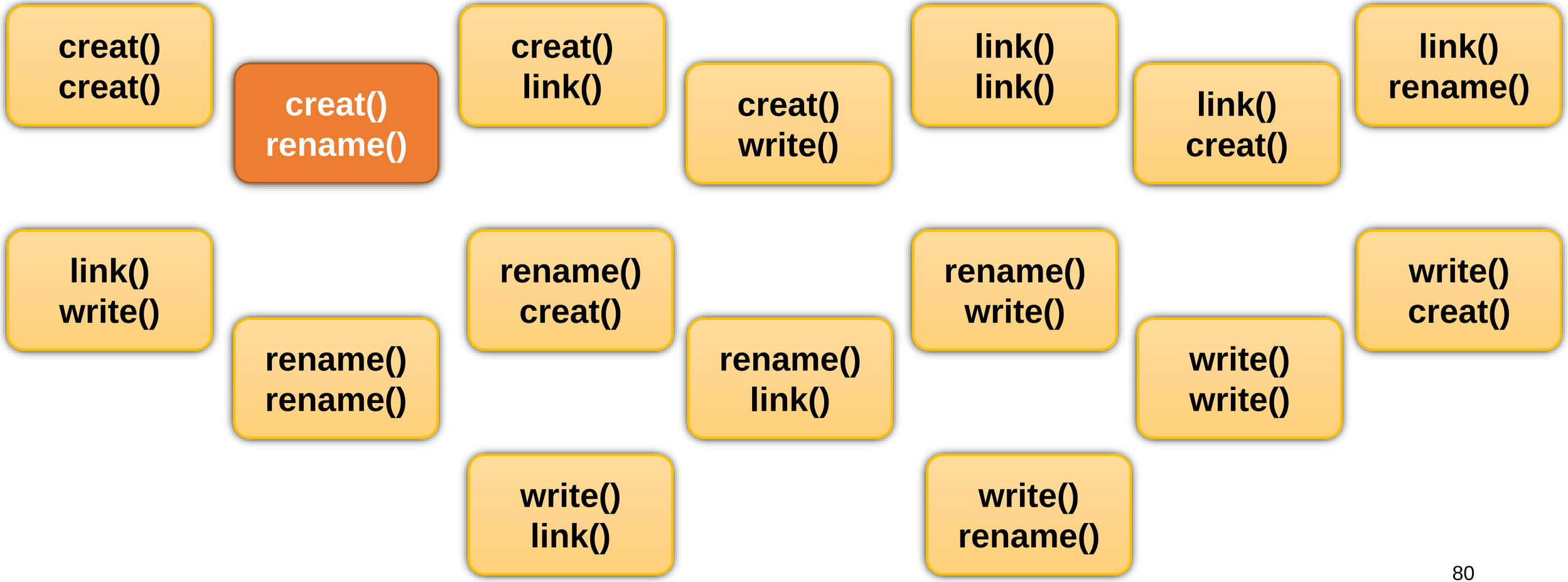


Phases of ACE

Operation Set

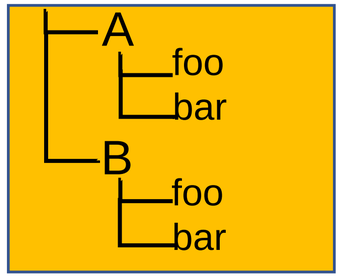
creat()
link()
rename()
write()

Generating skeletons of sequence-2. : 4*4 = 16



Phases of ACE

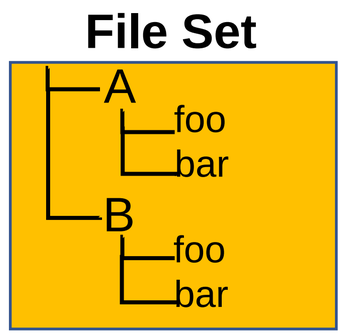
File Set



1. Select Operations

1. **creat()**
2. **rename()**

Phases of ACE



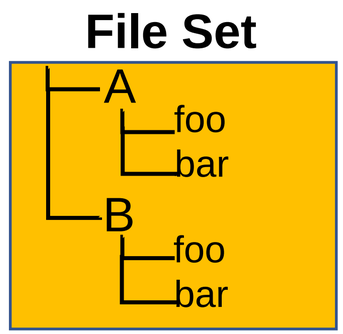
1. Select Operations

1. **creat()**
2. **rename()**

- If metadata operations, pick file or directory names
- If data operations, pick a range of offset and length

2. Select Parameters

Phases of ACE



1. Select Operations

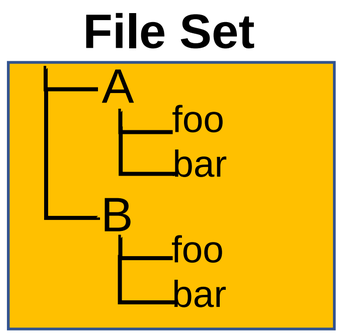
1. **creat()**
2. **rename()**

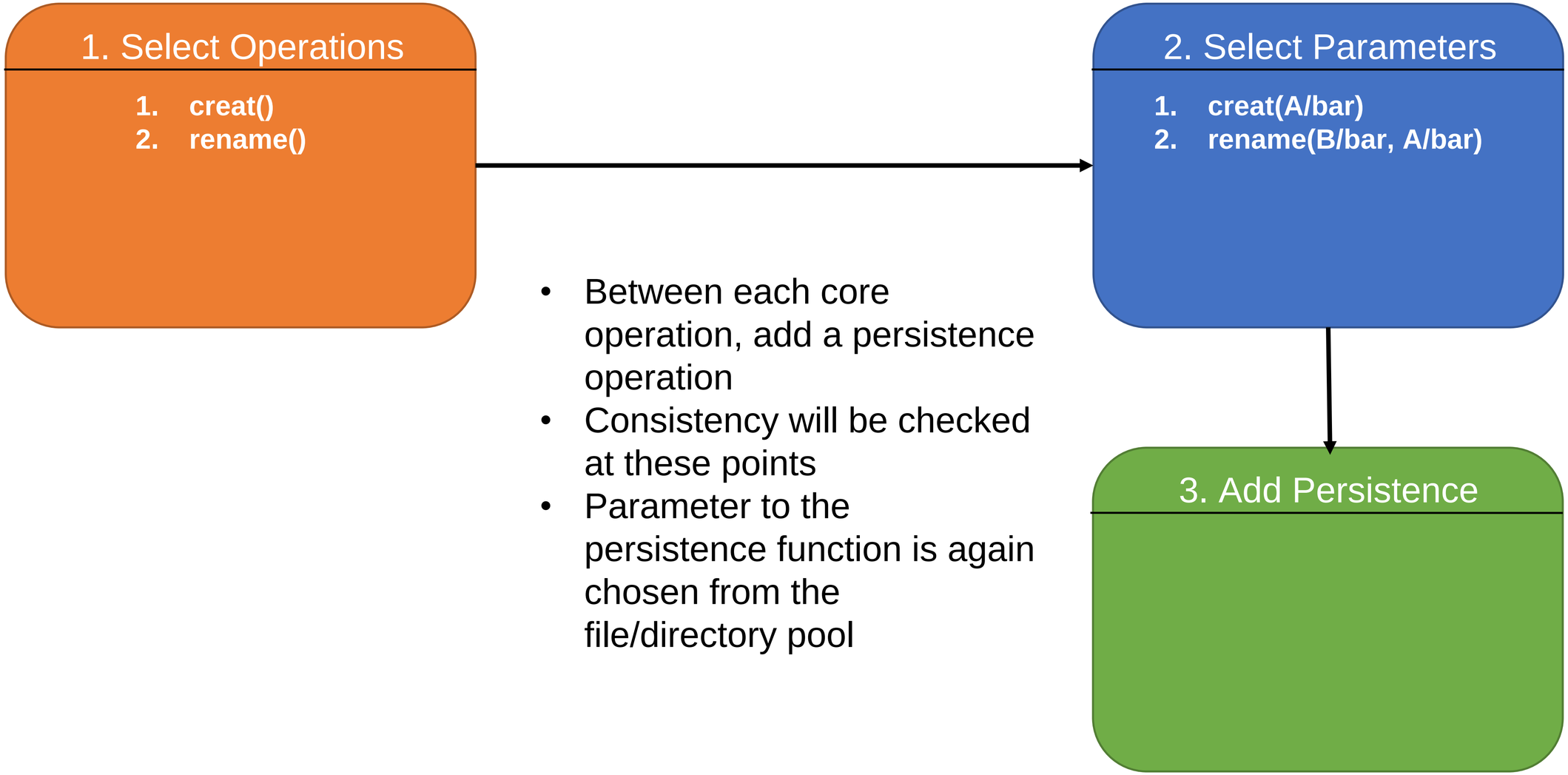
- If metadata operations, pick file or directory names
- If data operations, pick a range of offset and length

2. Select Parameters

1. **creat(A/bar)**
2. **rename(B/bar, A/bar)**

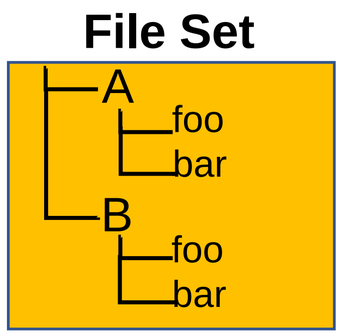
Phases of ACE





- Between each core operation, add a persistence operation
- Consistency will be checked at these points
- Parameter to the persistence function is again chosen from the file/directory pool

Phases of ACE



1. Select Operations

1. **creat()**
2. **rename()**

2. Select Parameters

1. **creat(A/bar)**
2. **rename(B/bar, A/bar)**

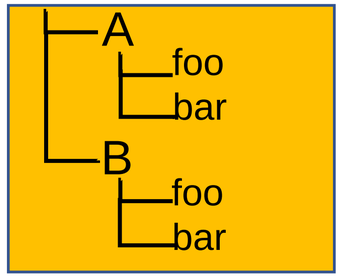
3. Add Persistence

1. **creat(A/bar)**
   fsync(A/bar)
2. **rename(B/bar, A/bar)**
   fsync(A/foo)

- Between each core operation, add a persistence operation
- Consistency will be checked at these points
- Parameter to the persistence function is again chosen from the file/directory pool

Phases of ACE

File Set



1. Select Operations

1. **creat()**
2. **rename()**

2. Select Parameters

1. **creat(A/bar)**
2. **rename(B/bar, A/bar)**

3. Add Persistence

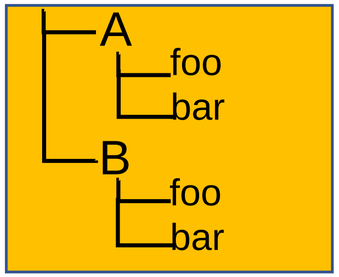
1. **creat(A/bar)**
   fsync(A/bar)
2. **rename(B/bar, A/bar)**
   fsync(A/foo)

4. Add Dependencies

- Add file create/open/close to ensure the workload executes on any POSIX compliant filesystem.

Phases of ACE

File Set



1. Select Operations

1. **creat()**
2. **rename()**

2. Select Parameters

1. **creat(A/bar)**
2. **rename(B/bar, A/bar)**

3. Add Persistence

1. **creat(A/bar)**
   fsync(A/bar)
2. **rename(B/bar, A/bar)**
   fsync(A/foo)

- Add file create/open/close to ensure the workload executes on any POSIX compliant filesystem.

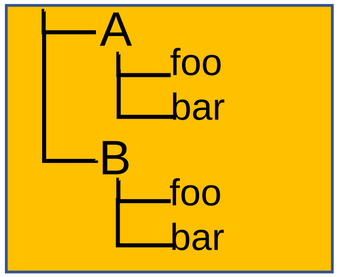
4. Add Dependencies

mkdir(A)

1. **creat(A/bar)**
   fsync(A/bar)
   mkdir(B)
   creat(B/bar)
2. **rename(B/bar, A/bar)**
   creat(A/foo)
   fsync(A/foo)
   close(A/foo)

Phases of ACE

File Set



1. Select Operations

1. **creat()**
2. **rename()**

2. Select Parameters

1. **creat(A/bar)**
2. **rename(B/bar, A/bar)**

3. Add Persistence

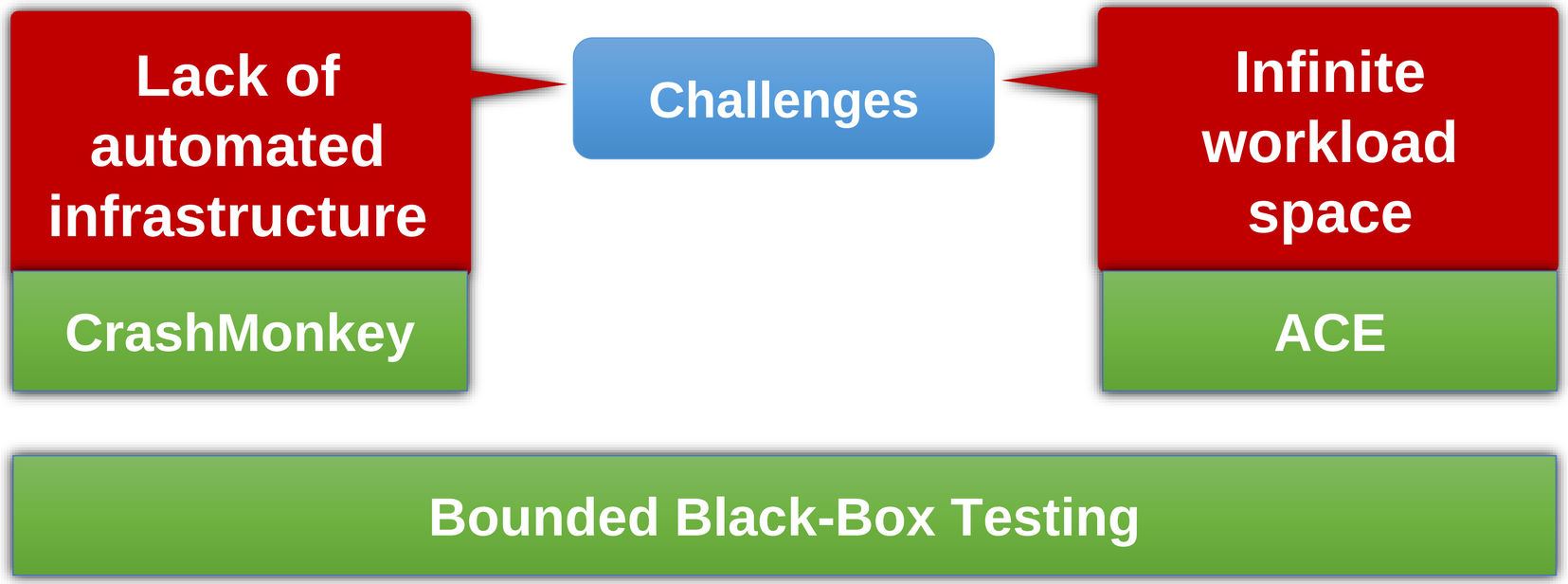
1. **creat(A/bar)**
   fsync(A/bar)
2. **rename(B/bar, A/bar)**
   fsync(A/foo)

4. Add Dependencies

mkdir(A)
1. **creat(A/bar)**
   fsync(A/bar)
   mkdir(B)
   creat(B/bar)
2. **rename(B/bar, A/bar)**
   creat(A/foo)
   fsync(A/foo)
   close(A/foo)

This workload with 2 core operations is the same workload required to trigger rename atomicity bug!

Challenges with Systematic Testing



Evaluation

We seek to answer the following questions:

1. Can B^3 reproduce known bugs in the kernel?
2. Can B^3 find new bugs in Linux Filesystems in a reasonable time?
3. Resource consumption by ACE and CrashMonkey
4. How can we generalize and scale this approach?

Test Setup

- Cluster of 65 nodes
  - 40 cores, 48GB RAM, 128GB SSD
  - 12VMs on every node, each with 2GB RAM, 10GB storage
    - Restricted by storage
  - Total 780 VMs in parallel

- Generate workloads on a local server and distribute over network to the VMs

Results at a glance

Sequence Length	# workloads	# Bugs Reproduced	# Bugs found	Time (min)
Seq-1				
Seq-2				
Seq-3 metadata				
Seq-3 data				
Seq-3 nested				
Total				

Results at a glance

Sequence Length	# workloads	# Bugs Reproduced	# Bugs found	Time (min)
Seq-1				
Seq-2				
Seq-3 metadata				
Seq-3 data				
Seq-3 nested				
Total				

25 million workloads
Needs 15 days of testing on 780 VMs in parallel!

Results at a glance

Sequence Length	# workloads	# Bugs Reproduced	# Bugs found	Time (min)
Seq-1	300			
Seq-2	254K			
Seq-3 metadata	120K			
Seq-3 data	1.5M			
Seq-3 nested	1.5M			
Total	**3.37M**			

Results at a glance

Sequence Length	# workloads	# Bugs Reproduced	# Bugs found	Time (min)
Seq-1	300	3		
Seq-2	254K	14		
Seq-3 metadata	120K	5		
Seq-3 data	1.5M	2		
Seq-3 nested	1.5M	2		
Total	**3.37M**	**26**		

Results at a glance

Sequence Length	# workloads	# Bugs Reproduced	# Bugs found	Time (min)
Seq-1	300	3	3	
Seq-2	254K	14	3	
Seq-3 metadata	120K	5	2	
Seq-3 data	1.5M	2	0	
Seq-3 nested	1.5M	2	2	
Total	**3.37M**	**26**	**10**	

Results at a glance

Sequence Length	# workloads	# Bugs Reproduced	# Bugs found	Time (min)
Seq-1	300	3	3	1
Seq-2	254K	14	3	215
Seq-3 metadata	120K	5	2	102
Seq-3 data	1.5M	2	0	1274
Seq-3 nested	1.5M	2	2	1274
Total	**3.37M**	**26**	**10**	**2866 (2 days)**

Are Verified File Systems Crash Safe?

- Tested FSCQ and Yxv6 with CrashMonkey and Ace
- Found broken fdatasync guarantees in FSCQ
  - Acknowledged and patched

Time to integrate back-box testing in your file system development cycle!

Results

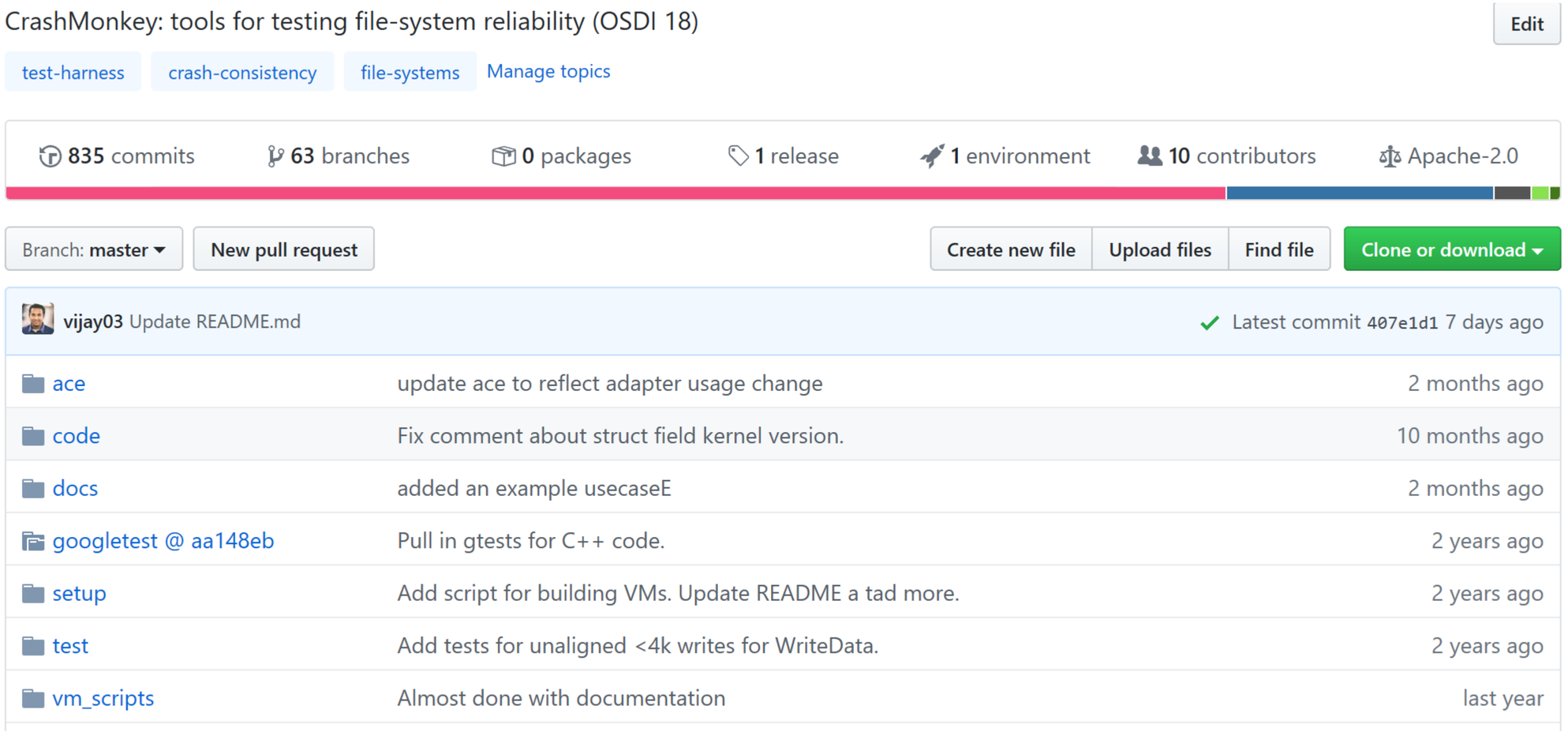
- Reproduced 24/26 known bugs across ext4, btrfs and F2FS
- Found 10 new bugs across btrfs and F2FS
- Found 1 bug in a verified file system, FSCQ

Generalizing Ace

- Open Source
- Easily expandable to test higher sequences exhaustively if more compute is available
- Integrated with Ace fuzzer to test random, but valid sequences
- Adding support for more system calls is straightforward
  - Increases the space of workloads

Using CrashMonkey & Ace

- Open Source : **https://github.com/utsaslab/crashmonkey**



Using CrashMonkey & Ace

Table Of Contents

Advanced Documentation

Push button testing!

Push Button Testing for Seq-1 Workloads

This repository contains a pre-generated suite of 328 seq-1 workloads (workloads with 1 file-system operation) here. Once you have set up CrashMonkey on your machine (or VM), you can simply run :

```
python xfsMonkey.py -f /dev/sda -d /dev/cow_ram0 -t btrfs -e 102400 -u build/tests/seq1/ > outfile
```

Sit back and relax. This is going to take about 12 minutes to complete if run on a single machine. This will run all the 328 tests of seq-1 on a `btrfs` file system `100MB` in size. The bug reports can be found in the folder `diff_results`. The workloads are named j-lang<1-328>, and, if any of these resulted in a bug, you will see a bug report with the same name as that of the workload, describing the difference between the expected and actual state.

More here : **https://github.com/utsaslab/crashmonkey**

Impact of our tools

Our tools found 10 long-standing bugs in btrfs and F2FS in the Linux kernel, and 1 bug in a verified file system, FSCQ.

The tests generated by our tools have been added to xfstests, the file-system test suite for the Linux kernel.

Spurred discussion and effort towards documenting the crash guarantees of various Linux filesystems in the kernel

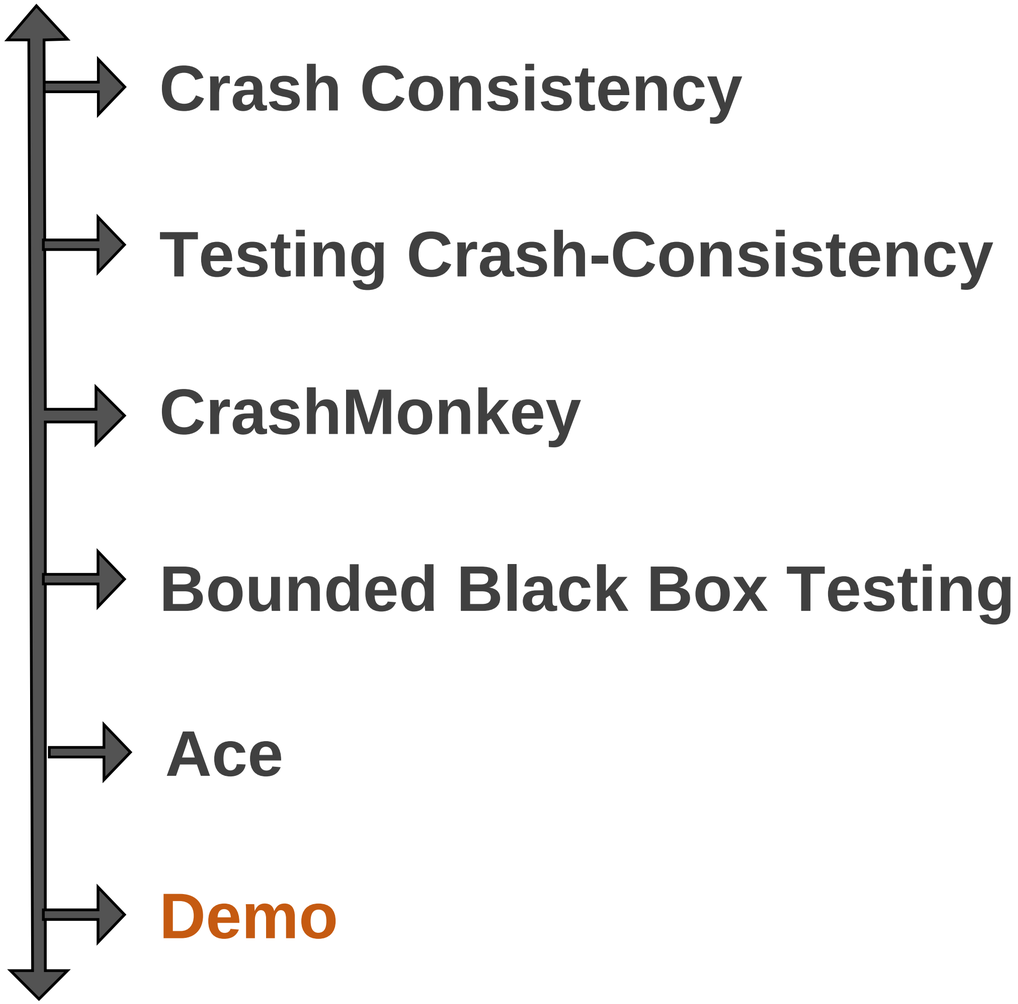
Used in research file systems like BarrierFS to test their crash-consistency guarantees.

What did we learn from B3?

Even if you build verified software, testing is important!

A custom tool that is application-aware is more powerful than a generic testing approach

Agenda


Crash Consistency
Testing Crash-Consistency
CrashMonkey
Bounded Black Box Testing
Ace
Demo

```
root@jayashree-VirtualBox:/home/jayashree/crashmonkey/demo/crashmonkey# ./demo.sh btrfs btrfs_out
```

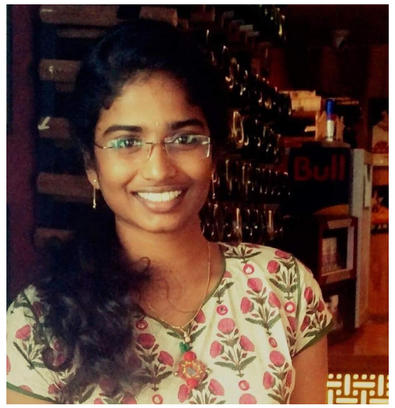
Summary

- File system crash consistency bugs result in severe consequences like metadata corruption and data loss.
- To address the lack of infrastructure for testing such bugs : CrashMonkey
  - Record and replay framework
- Automated workload generation using Ace
  - Study the past bug pattern and explore a bounded space
- Found 10 new bugs in Linux FS and 1 bug in a verified FS

Our team

Jayashree Mohan

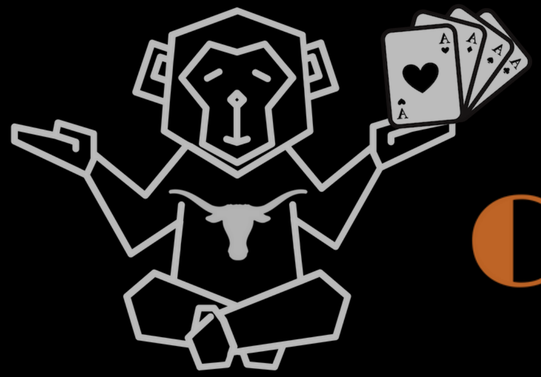
Ashlie Martinez



Soujanya Ponnapalli

Pandian Raju

Vijay Chidambaram



CRASH MONKEY & ACE

- B^3 makes exhaustive testing feasible using informed bound selection
- Easily generalizable to test larger workloads if more compute is available
- Found 10 new bugs across btrfs and F2FS, most of which existed since 2014
- Found 1 bug in FSCQ

Try our tools : **https://github.com/utsaslab/crashmonkey**

Contact : **Jayashree Mohan** (jaya@cs.utexas.edu)

Questions?