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

Thanumalayan Sankaranarayana Pillai
Vijay Chidambaram
Ramnatthan Alagappan, Samer Al-Kiswany
Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau

WISCONSIN
UNIVERSITY OF WISCONSIN–MADISON

WISDOM

Crash Consistency

Maintaining data invariants across system crash

- Ex: database transactions should be atomic

Important in many systems

- Databases
- File Systems
- Key-Value Stores

File-System Crash Consistency

Many techniques to ensure file system remains consistent after a crash

- Journaling
- Copy-on-write
- Soft Updates

Techniques ensure internal file-system structures are consistent

Application-Level Crash Consistency

Many applications run over file systems

- Ex: SQLite, LevelDB, Git, etc.

Application-Level Crash Consistency

Many applications run over file systems

- Ex: SQLite, LevelDB, Git, etc.

Provide user guarantees across system crashes

- Ex: SQLite txs provide ACID guarantees

Interact with file systems via POSIX calls

Crash Recovery is Hard

Databases took a long time to get it right

- Commercial database - System R (1981)
- Crash Recovery algorithm - ARIES (1992)
- ARIES proved correct (1997)

Applications must achieve high performance and correctness

- Mutating persistent state synchronously too slow
- Leads to complex update protocols

Belief: all POSIX file systems implement calls the same way

Application using POSIX interface should work on any POSIX-compliant file system

POSIX specifies what happens in memory

- Does not specify what happens on crash

Example

Goal: atomically update multiple blocks of file

/dir/file	my foo

Initial state

→

/dir/file	my bar

Final state

Solution: use write-ahead-logging

Protocol:

1. Write to log (/dir/log)
2. Update /dir/file
3. Delete /dir/log

On crash:

read log,

do steps 2 and 3

Example

1. Write log

```
open(“/dir/log”, O_CREAT|O_WRONLY| O_TRUNC)
pwrite(“/dir/log”, log_entry, 0, 6)
```

2. Update /dir/file

```
pwrite(“/dir/file”, data, 3, 3)
```

3. Delete /dir/log

```
unlink(“/dir/log”)
```

Example

```
1    open("/dir/log", O_CREAT|O_WRONLY| O_TRUNC)
2    pwrite("/dir/log", log_entry, 0, 6)
3    pwrite("/dir/file", "bar", 3, 3)
4    unlink("/dir/log")
```

Works in ext3 data journaling mode

Possible disk states after a crash:

After 1

/dir/file	my foo
/dir/log	

After 2

/dir/file	my foo
/dir/log	3, 3, bar

Middle of 3

/dir/file	my boo
/dir/log	3, 3, bar

After 4

/dir/file	my bar

Example

```
open("/dir/log", O_CREAT|O_WRONLY| O_TRUNC)
pwrite("/dir/log", log_entry, 0, 6)
pwrite("/dir/file", "bar", 3, 3)
unlink("/dir/log")
```

Fails in ext3 ordered mode!

Example

```
open(“/dir/log”, O_CREAT|O_WRONLY| O_TRUNC)
pwrite(“/dir/log”, log_entry, 0, 6)
pwrite(“/dir/file”, “bar”, 3, 3)
unlink(“/dir/log”)
```

Fails in ext3 ordered mode!

Example

```
1   open("/dir/log", O_CREAT|O_WRONLY| O_TRUNC)
2   pwrite("/dir/log", log_entry, 0, 6)
2.5 fsync("/dir/log")
3   pwrite("/dir/file", "bar", 3, 3)
3.5 fsync("/dir/file")
4   unlink("/dir/log")
```

Fails in ext3 ordered mode!

Example

```
1    open("/dir/log", O_CREAT|O_WRONLY| O_TRUNC)
2    pwrite("/dir/log", log_entry, 0, 6)
2.5  fsync("/dir/log")
3    pwrite("/dir/file", "bar", 3, 3)
3.5  fsync("/dir/file")
4    unlink("/dir/log")
```

Fails in ext3 ordered mode!

/dir/file	my boo
/dir/log	

Fails in ext3 writeback mode!

/dir/file	my foo
/dir/log	&%^

Example

```
1   open("/dir/log", O_CREAT|O_WRONLY| O_TRUNC)
2   pwrite("/dir/log", log_entry + cxsum, 0, 6)
2.5 fsync("/dir/log")
3   pwrite("/dir/file", "bar", 3, 3)
3.5 fsync("/dir/file")
4   unlink("/dir/log")
```

Fails in ext3 ordered mode!

/dir/file	my boo
/dir/log	

Fails in ext3 writeback mode!

/dir/file	my foo
/dir/log	&%^

Example

```
1   open("/dir/log", O_CREAT|O_WRONLY| O_TRUNC)
2   pwrite("/dir/log", log_entry + cxsum, 0, 6)
2.5 fsync("/dir/log")
3   pwrite("/dir/file", "bar", 3, 3)
3.5 fsync("/dir/file")
4   unlink("/dir/log")
```

Fails in ext3 ordered mode!

/dir/file	my boo
/dir/log	

Fails in ext3 writeback mode!

/dir/file	my foo
/dir/log	&%^

May fail in new POSIX file system

/dir/file	my boo

Example

```
1   open("/dir/log", O_CREAT|O_WRONLY| O_TRUNC)
2   pwrite("/dir/log", log_entry + cxsum, 0, 6)
2.5 fsync("/dir/log")
2.7 fsync("/dir")
3   pwrite("/dir/file", "bar", 3, 3)
3.5 fsync("/dir/file")
4   unlink("/dir/log")
```

Fails in ext3 ordered mode!

/dir/file	my boo
/dir/log	

Fails in ext3 writeback mode!

/dir/file	my foo
/dir/log	&%^

May fail in new POSIX file system

/dir/file	my boo

How do file-systems vary in implementing POSIX calls?

Built Block-Order-Breaker(BOB) and analyzed 6 file systems

How do applications maintain crash consistency?

Built Application-Level Intelligent Crash Explorer (ALICE) and analyzed 11 applications

Outline

Introduction

Background

Analyzing file systems with BOB

Analyzing applications with ALICE

Application Study

Conclusion and Future Work

POSIX Standard

POSIX standard is extremely weak

Example: POSIX `fsync()` need not flush data

From the man page for Mac OS X `fsync()`:

```
Specifically, if the drive loses power or the
OS crashes, the application may find that only
some or none of their data was written.
```

Flushing data to disk requires `F_FULLFSYNC fcntl`

Unwritten Standard

Developers coded to an unwritten standard

- Based on ext3 (default Linux fs for many years)

Widely held belief:

- Correct behavior == ext3 behavior
- POSIX guarantees not widely known

All was well until…

ext4 introduced delayed allocation

- data writes could be persisted after `rename()`

Changed guarantees given to applications

Broke hundreds of applications

- `write (tmp); rename(tmp, old)` led to zero length files

Caused wide-spread data loss

Application developers:
your file system is broken!

FS developers:
your application is broken!
It doesn't follow POSIX!

File-system developers added code to bring back old behavior in certain cases

All this could have been avoided if application requirements are known to developers

Our tool, ALICE, allows developers to do this

Outline

Introduction

Background

Analyzing file systems with BOB

Analyzing applications with ALICE

Application Study

Conclusion and Future Work

Analyzing File Systems

File systems implement POSIX calls differently

Study behavior via Persistence Properties:

- define how system calls are persisted
- affect which on-disk states are possible after a system crash
- two classes: atomicity and ordering

Persistence Properties: Example

Consider the following code snippet:

```
write(f1, "AA")
write(f2, "BB")
```

f1	
f2	

Atomicity

f1	AA
f2	BB

f1	A
f2	B

Only size updated

f1	YY
f2	ZZ

Ordering

f1	AA
f2	

f1	AA
f2	BB

f1	
f2	BB

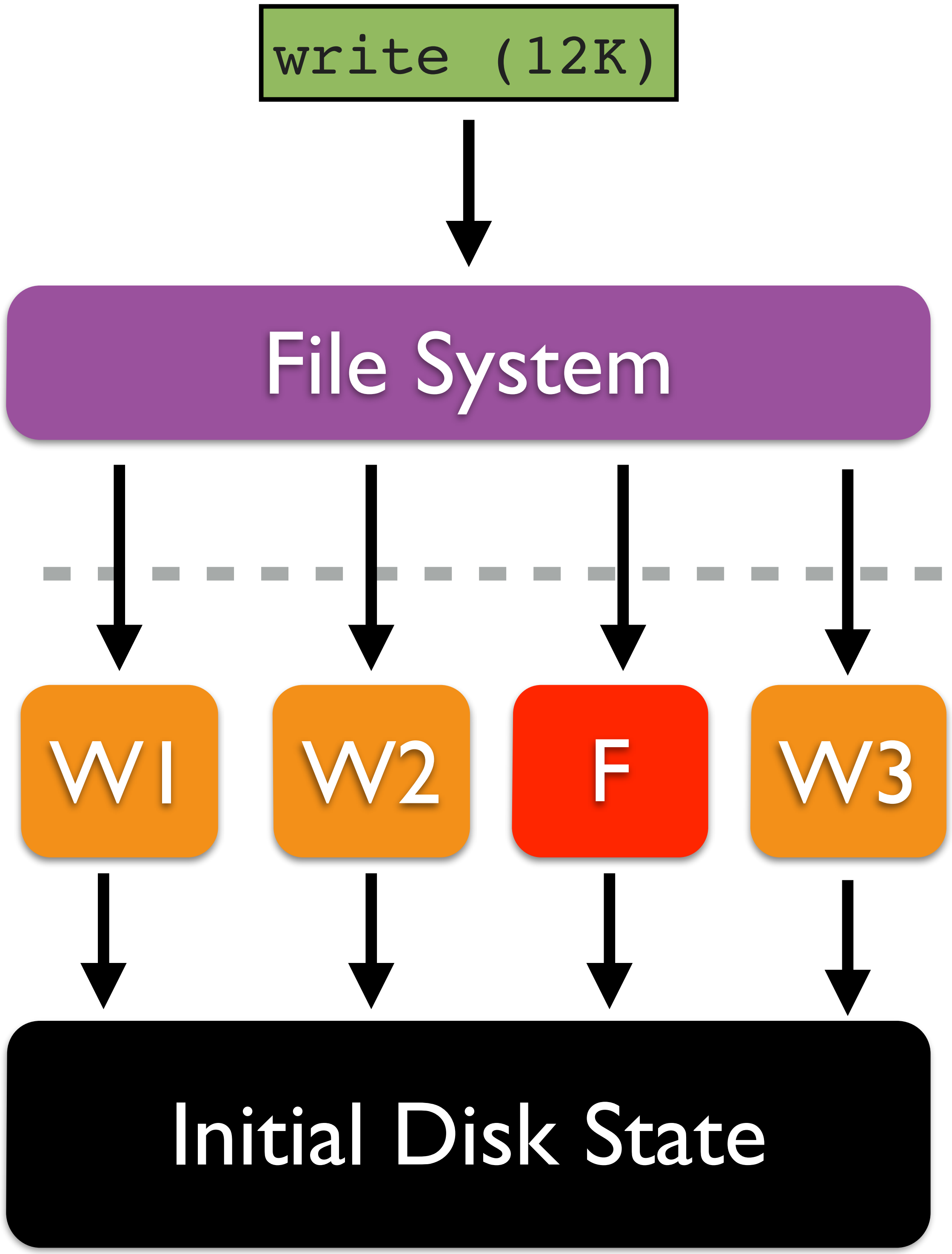
Block-Order-Breaker

Built Block-Order-Breaker (BOB) to study persistence properties

Methodology:

- Re-order block IO to construct legal disk images
- Inspect file-system state on constructed images
- Test whether persistence properties hold

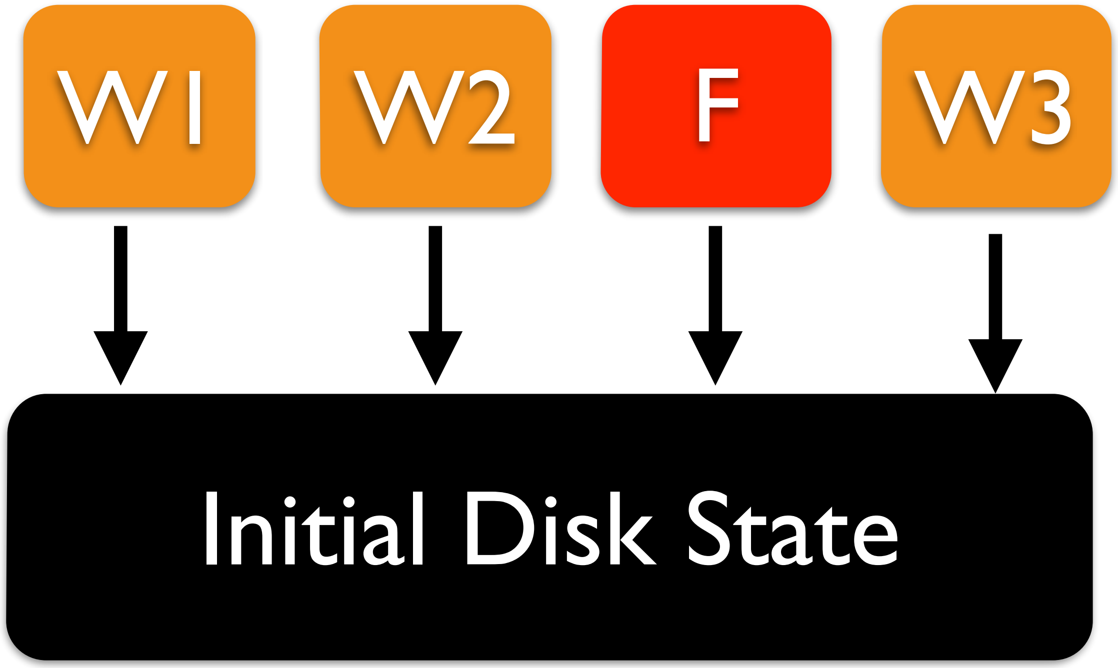
Block-Order-Breaker (BOB)



Test workload designed to stress persistence property

Capture block-level trace of writes, flushes, barriers

Block-Order-Breaker (BOB)



Reconstruct crash state

States limited by flushes and barriers

Block-Order-Breaker (BOB)

Reconstruct crash state

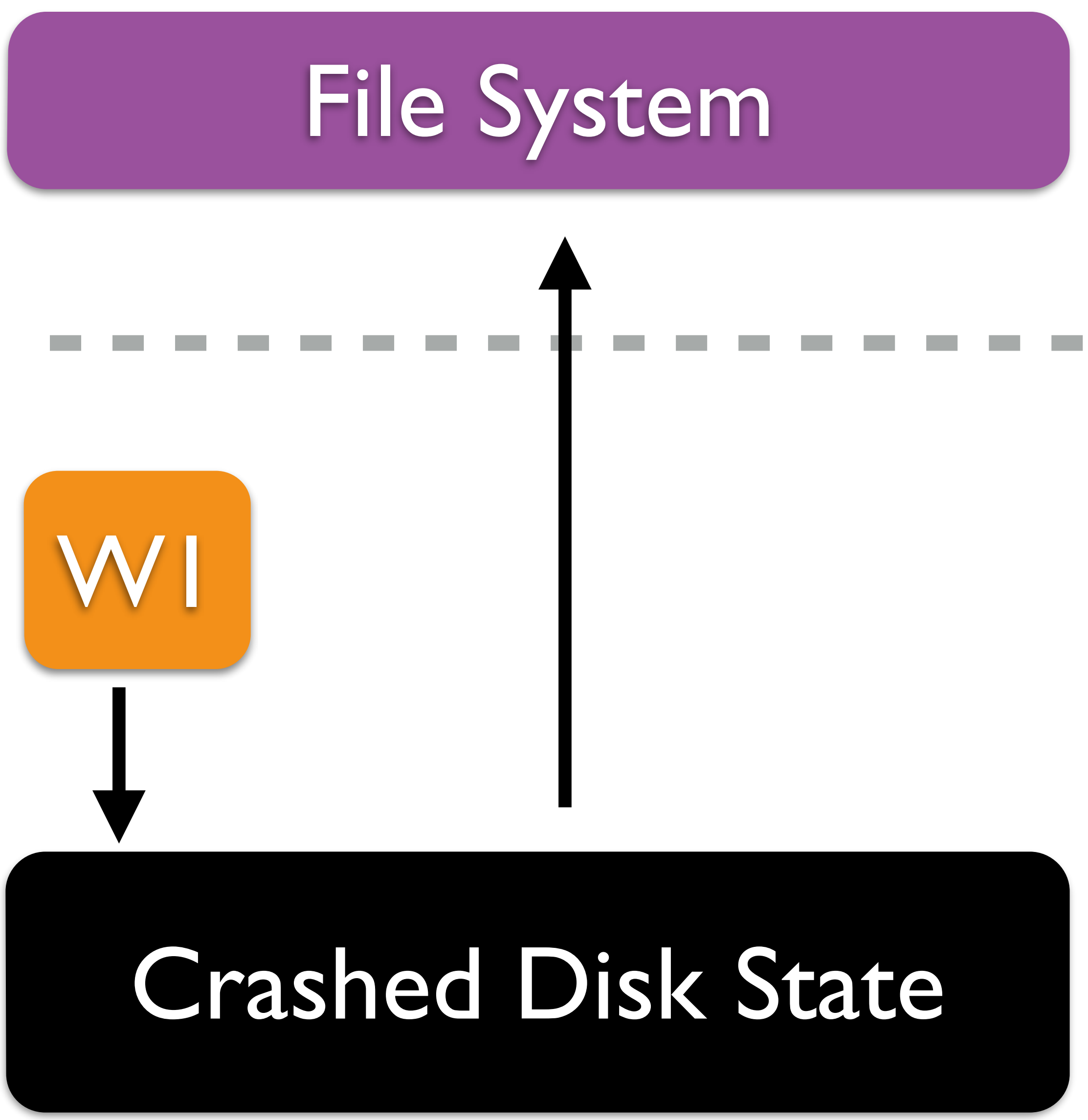
States limited by flushes and barriers

Block-Order-Breaker (BOB)

Reconstruct crash state

States limited by flushes and barriers

Block-Order-Breaker (BOB)



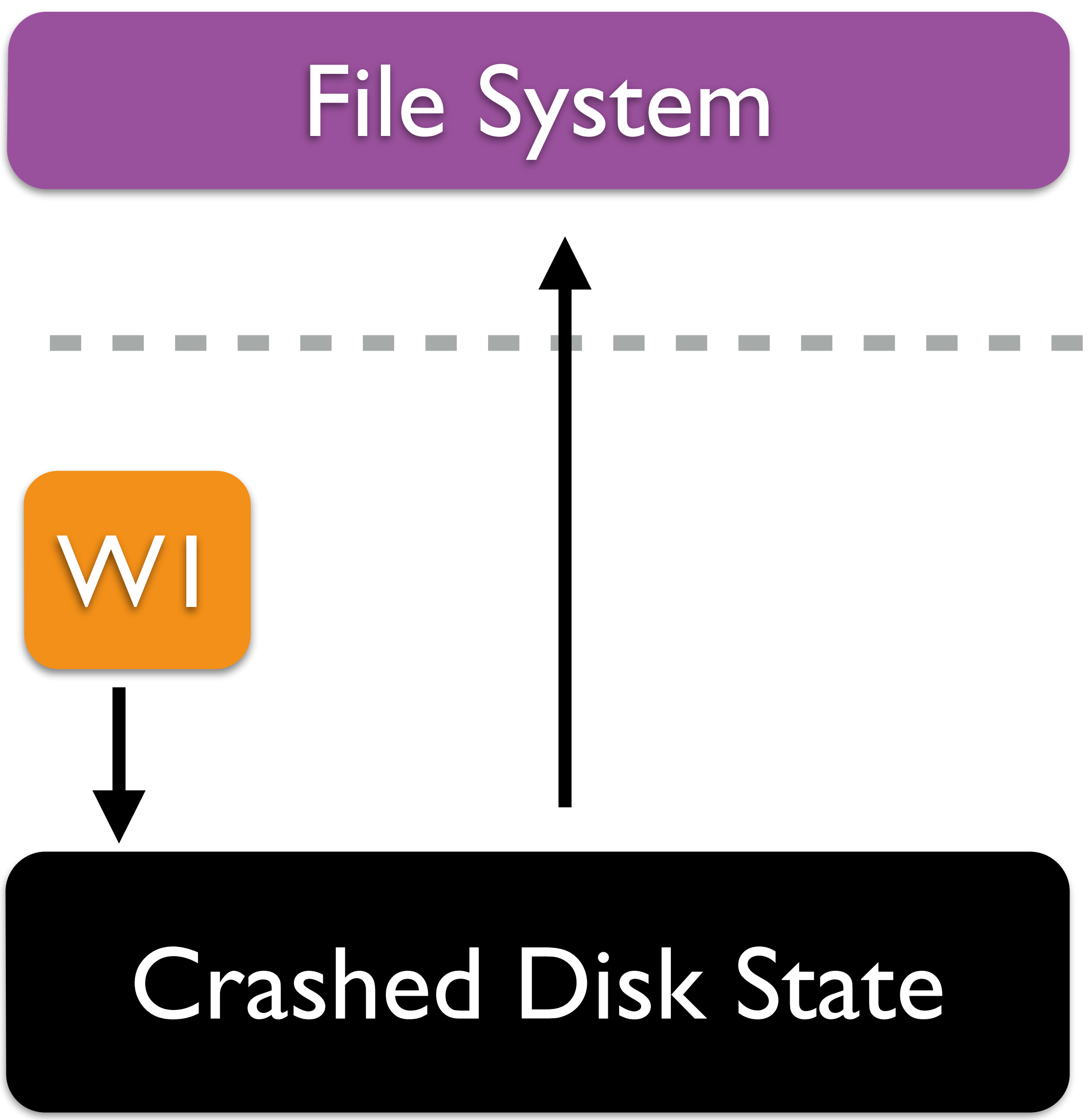
Mount file system on crashed disk state

Reconstruct crash state

States limited by flushes and barriers

Block-Order-Breaker (BOB)

Is entire `write()` data present?



Check if persistence property violated

Mount file system on crashed disk state

Reconstruct crash state

States limited by flushes and barriers

Caveats

BOB can used to find:

- Which properties don't hold
- The exact case where the property fails

BOB does not explore all workloads

BOB cannot be used to prove a file system has a specific persistence property

Studying File Systems

Uses BOB to study how properties varied over file systems

Studied six file systems

- ext2, ext3, ext4, btrfs, xfs, reiserfs
- A total of 16 configurations

Study Results: Atomicity

ext2 async
ext2 sync
ext3 writeback
ext3 ordered
ext3 journal
ext4 writeback
ext4 ordered
ext4 no-delalloc
ext4 journal
btrfs
xfs default
xfs wsync

Study Results: Atomicity

	Single Sector
ext2 async	
ext2 sync	
ext3 writeback	
ext3 ordered	
ext3 journal	
ext4 writeback	
ext4 ordered	
ext4 no-delalloc	
ext4 journal	
btrfs	
xfs default	
xfs wsync	

`write(512)` atomic?

Study Results: Atomicity

	Single Sector
ext2 async	
ext2 sync	
ext3 writeback	
ext3 ordered	
ext3 journal	
ext4 writeback	
ext4 ordered	
ext4 no-delalloc	
ext4 journal	
btrfs	
xfs default	
xfs wsync	

Study Results: Atomicity

	Single Sector	Multi Sector
ext2 async		
ext2 sync		
ext3 writeback		
ext3 ordered		
ext3 journal		
ext4 writeback		
ext4 ordered		
ext4 no-delalloc		
ext4 journal		
btrfs		
xfs default		
xfs wsync		

`write(1GB)` atomic?

Study Results: Atomicity

	Single Sector	Multi Sector
ext2 async		x
ext2 sync		x
ext3 writeback		x
ext3 ordered		x
ext3 journal		x
ext4 writeback		x
ext4 ordered		x
ext4 no-delalloc		x
ext4 journal		x
btrfs		x
xfs default		x
xfs wsync		x

Study Results: Atomicity

	Single Sector	Multi Sector	Append Content
ext2 async		x	
ext2 sync		x	
ext3 writeback			
ext3 ordered			
ext3 journal			
ext4 writeback			
ext4 ordered		x	
ext4 no-delalloc		x	
ext4 journal		x	
btrfs		x	
xfs default		x	
xfs wsync		x	

`open(file, O_APPEND)`
`write(12K)` atomic?

Study Results: Atomicity

	Single Sector	Multi Sector	Append Content
ext2 async		x	x
ext2 sync		x	x
ext3 writeback		x	x
ext3 ordered		x	
ext3 journal		x	
ext4 writeback		x	x
ext4 ordered		x	
ext4 no-delalloc		x	
ext4 journal		x	
btrfs		x	
xfs default		x	
xfs wsync		x	

Study Results: Atomicity

	Single Sector	Multi Sector	Append Content	Directory Operation
ext2 async		x	x	
ext2 sync		x	x	
ext3 writeback				
ext3 ordered				
ext3 journal				
ext4 writeback				
ext4 ordered		x		
ext4 no-delalloc		x		
ext4 journal		x		
btrfs		x		
xfs default		x		
xfs wsync		x		

`rename(old, new)` atomic?

Study Results: Atomicity

	Single Sector	Multi Sector	Append Content	Directory Operation
ext2 async		x	x	x
ext2 sync		x	x	x
ext3 writeback		x	x	
ext3 ordered		x		
ext3 journal		x		
ext4 writeback		x	x	
ext4 ordered		x		
ext4 no-delalloc		x		
ext4 journal		x		
btrfs		x		
xfs default		x		
xfs wsync		x		

Study Results: Ordering

ext2 async
ext2 sync
ext3 writeback
ext3 ordered
ext3 journal
ext4 writeback
ext4 ordered
ext4 no-delalloc
ext4 journal
btrfs
xfs default
xfs wsync

Study Results: Ordering

Overwrite
-> any op

ext2 async
ext2 sync
ext3 writeback
ext3 order
ext3 journ
ext4 writeb
ext4 ordered
ext4 no-delalloc
ext4 journal
btrfs
xfs default
xfs wsync

`write(4K) -> rename()`

Study Results: Ordering

	Overwrite -> any op
ext2 async	x
ext2 sync	
ext3 writeback	x
ext3 ordered	x
ext3 journal	
ext4 writeback	x
ext4 ordered	x
ext4 no-delalloc	x
ext4 journal	
btrfs	
xfs default	x
xfs wsync	x

Study Results: Ordering

	Overwrite -> any op	Append -> any op
ext2 async	x	x
ext2 sync		
ext3 writeback	x	x
ext3 ordered	x	
ext3 journal		
ext4 writeback	x	x
ext4 ordered	x	
ext4 no-delalloc	x	
ext4 journal		
btrfs		x
xfs default	x	x
xfs wsync	x	

Study Results: Ordering

	Overwrite -> any op	Append -> any op	Dir op -> any op
ext2 async	x	x	x
ext2 sync			
ext3 writeback	x	x	
ext3 ordered	x		
ext3 journal			
ext4 writeback	x	x	
ext4 ordered	x		
ext4 no-delalloc	x		
ext4 journal			
btrfs		x	x
xfs default	x	x	
xfs wsync	x		

Study Results: Ordering

	Overwrite -> any op	Append -> any op	Dir op -> any op	Append(f) -> rename(f)
ext2 async	x	x	x	x
ext2 sync				
ext3 writeback	x	x		x
ext3 ordered	x			
ext3 journal				
ext4 writeback	x	x		x
ext4 ordered	x			
ext4 no-delalloc	x			
ext4 journal				
btrfs		x	x	
xfs default	x	x		
xfs wsync	x			

Study Results: Ordering

	Overwrite -> any op	Append -> any op	Dir op -> any op	Append(f) -> rename(f)
ext2 async	x	x	x	x
ext2 sync				
ext3 writeback	x	x		x
ext3 ordered	x			
ext3 journal				
ext4 writeback	x	x		x
ext4 ordered	x			
ext4 no-delalloc	x			
ext4 journal				
btrfs		x	x	
xfs default	x	x		
xfs wsync	x			

File-System Study Results

Persistence properties vary widely among file systems

- Even within different configurations of same file system

Applications should not rely on them

Testing application correctness on single file system is not enough

Outline

- Introduction
- Background
- Analyzing file systems with BOB
- Analyzing applications with ALICE
- Application Study
- Conclusion and Future Work

Application-Level Intelligent Crash Explorer (ALICE)

ALICE: tool to find Crash Vulnerabilities

Application Crash Vulnerabilities

- code that depends on specific persistence properties for correct behavior
- ex: if file system doesn't persist two system calls calls in order, it leads to data corruption

ALICE Methodology

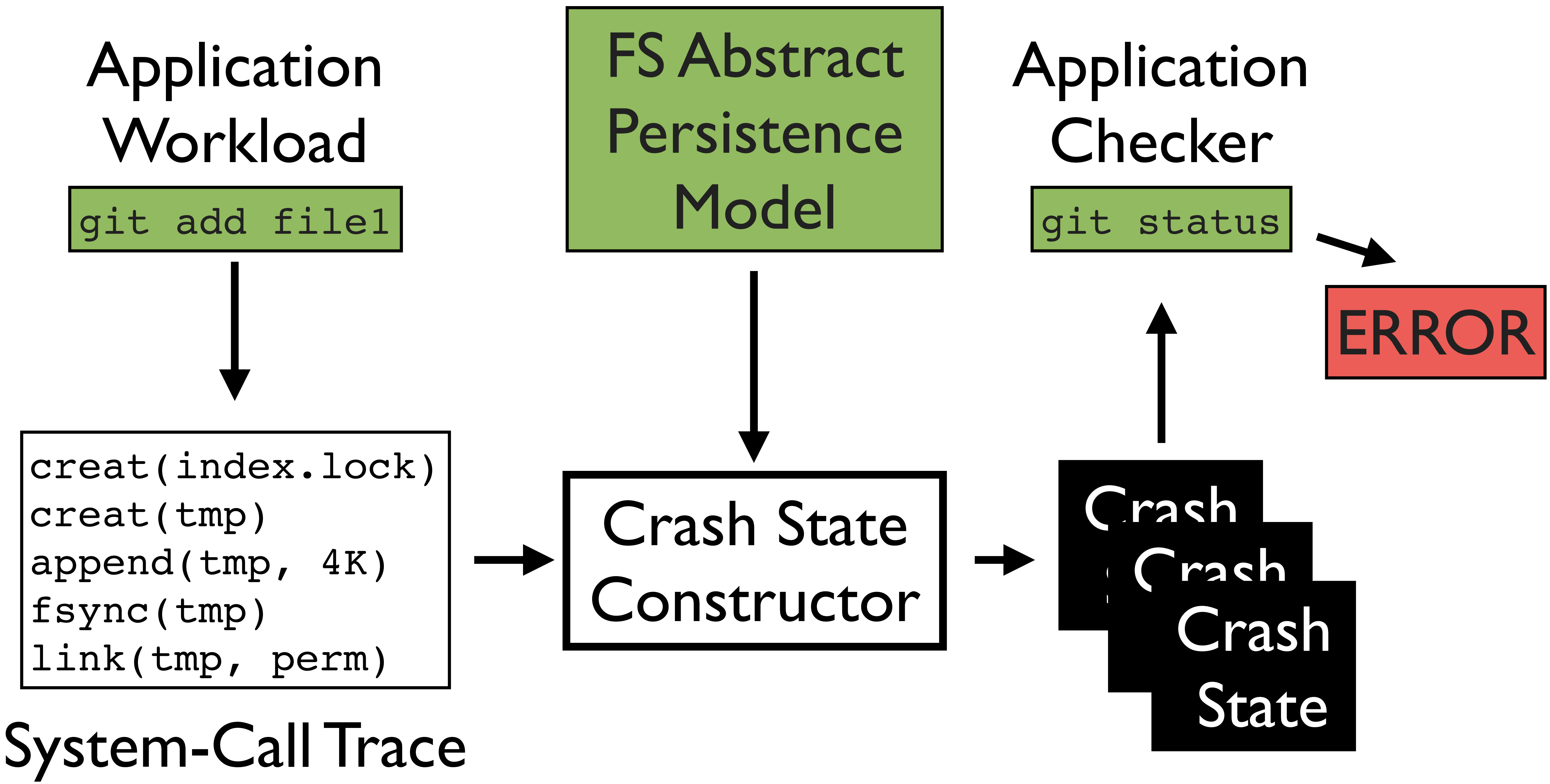
Construct crash state by violating a single persistence property

Run application on crash state (allow recovery)

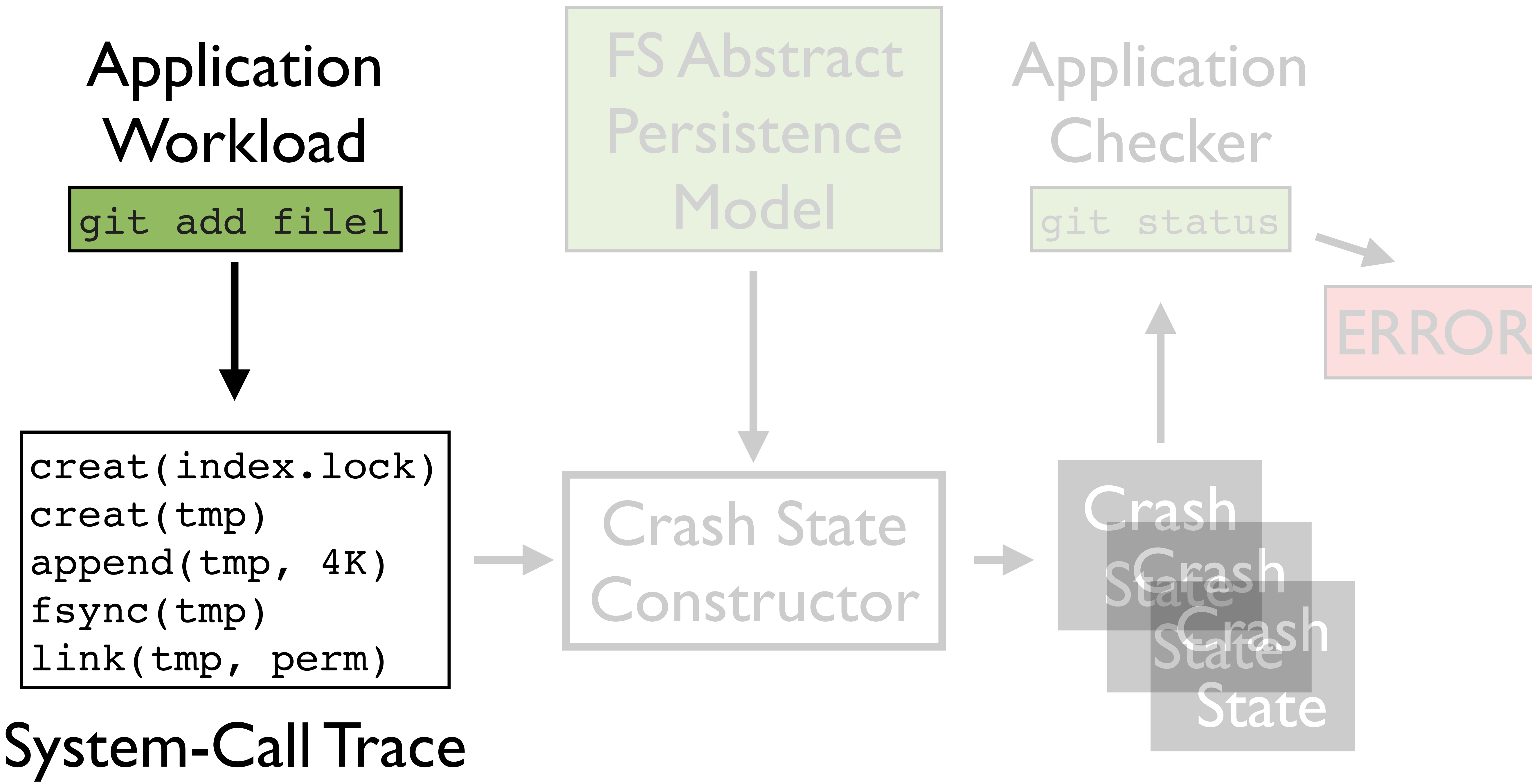
Examine application state

If application inconsistent, it depended on persistence property violated in crash state

ALICE Overview



ALICE Overview



Tracing the Workload

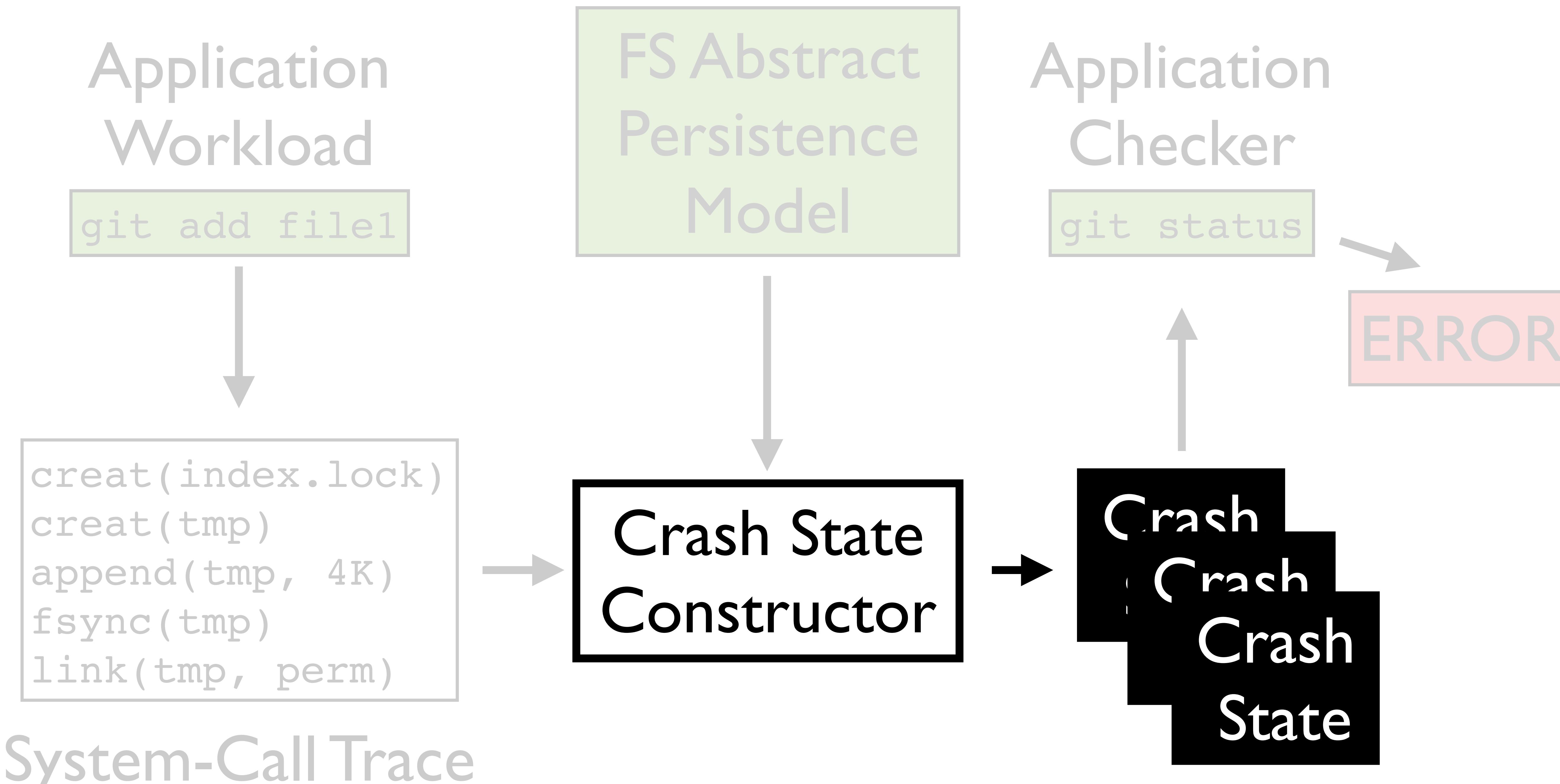
Run the application workload

Collect the system-call traces

System calls converted into logical operations:

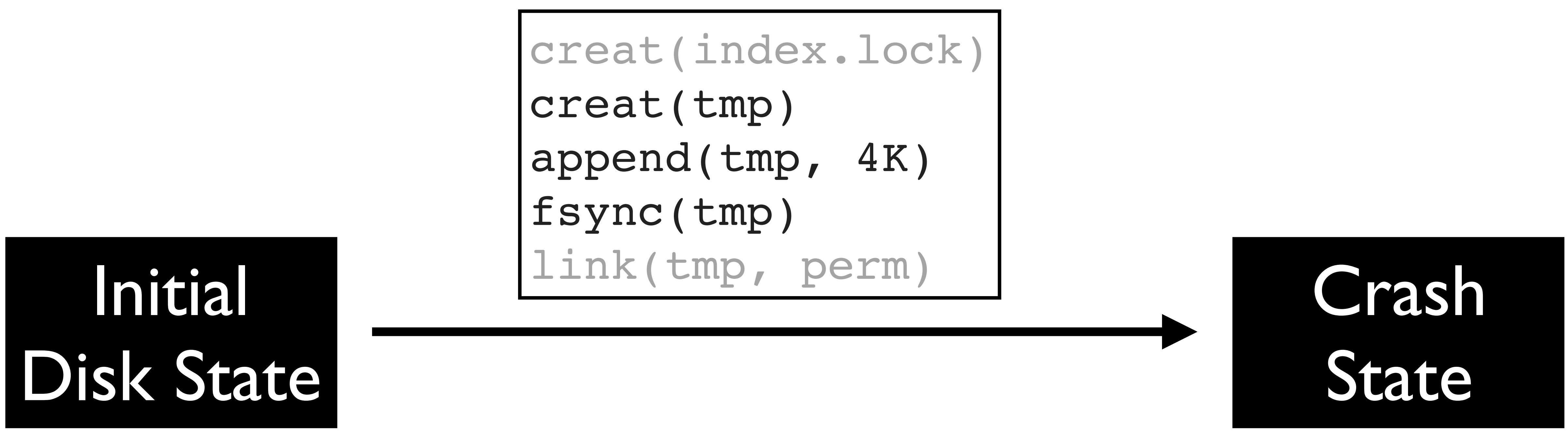
- Abstract away current file offset, fd, etc
- Group `writev()`, `pwrite()` etc into a single type of operation

ALICE Overview



Constructing Crash States

ALICE constructs crash states by applying a subset of operations to the initial disk image



Constructing Crash States

Persistence Properties Violated:

1. Atomicity across system calls

   Method: apply prefix of operations

```
creat(index.lock)
creat(tmp)
append(tmp, 4K)
fsync(tmp)
link(tmp, perm)
```

2. Atomicity within system calls

Constructing Crash States

Persistence Properties Violated:

1. Atomicity across system calls

   Method: apply prefix of operations

```
creat(index.lock)
creat(tmp)
append(tmp, 4K)
fsync(tmp)
link(tmp, perm)
```

2. Atomicity within system calls

   Method: apply prefix + partial operation

```
creat(index.lock)
creat(tmp)
append(tmp, 4K)
fsync(tmp)
link(tmp, perm)
```

Constructing Crash States

Persistence Properties Violated:

1. Atomicity across system calls

   Method: apply prefix of operations

```
creat(index.lock)
creat(tmp)
append(tmp, 4K)
fsync(tmp)
link(tmp, perm)
```

2. Atomicity within system calls

   Method: apply prefix + partial operation

```
creat(index.lock)
creat(tmp)
append(tmp, 512)
…
append(tmp, 512)
fsync(tmp)
link(tmp, perm)
```

Constructing Crash States

Persistence Properties Violated:

1. Atomicity across system calls

   Method: apply prefix of operations

```
creat(index.lock)
creat(tmp)
append(tmp, 4K)
fsync(tmp)
link(tmp, perm)
```

2. Atomicity within system calls

   Method: apply prefix + partial operation

```
creat(index.lock)
creat(tmp)
append(tmp, 512)
…
append(tmp, 512)
fsync(tmp)
link(tmp, perm)
```

Constructing Crash States

Persistence Properties Violated:

3. Ordering among system calls

```
creat(index.lock)
creat(tmp)
append(tmp, 4K)
fsync(tmp)
link(tmp, perm)
```

Method: ignore an operation, apply prefix

Constructing Crash States

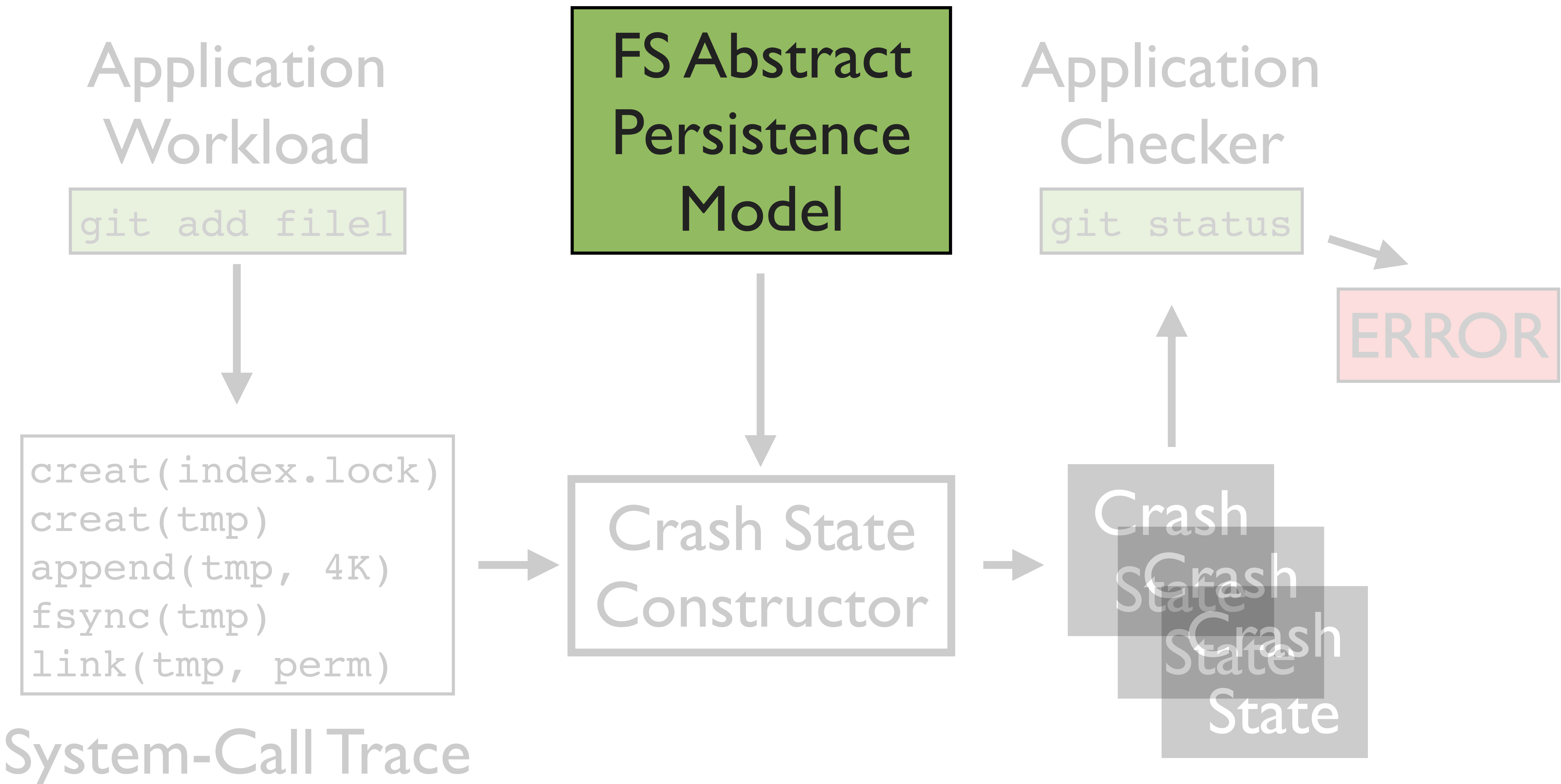
Persistence Properties Violated:

3. Ordering among system calls

```
creat(index.lock)
creat(tmp)
append(tmp, 4K)
fsync(tmp)
link(tmp, perm)
```

Method: ignore an operation, apply prefix

ALICE Overview



FS Abstract Persistence Model

Each file system implements persistence properties differently

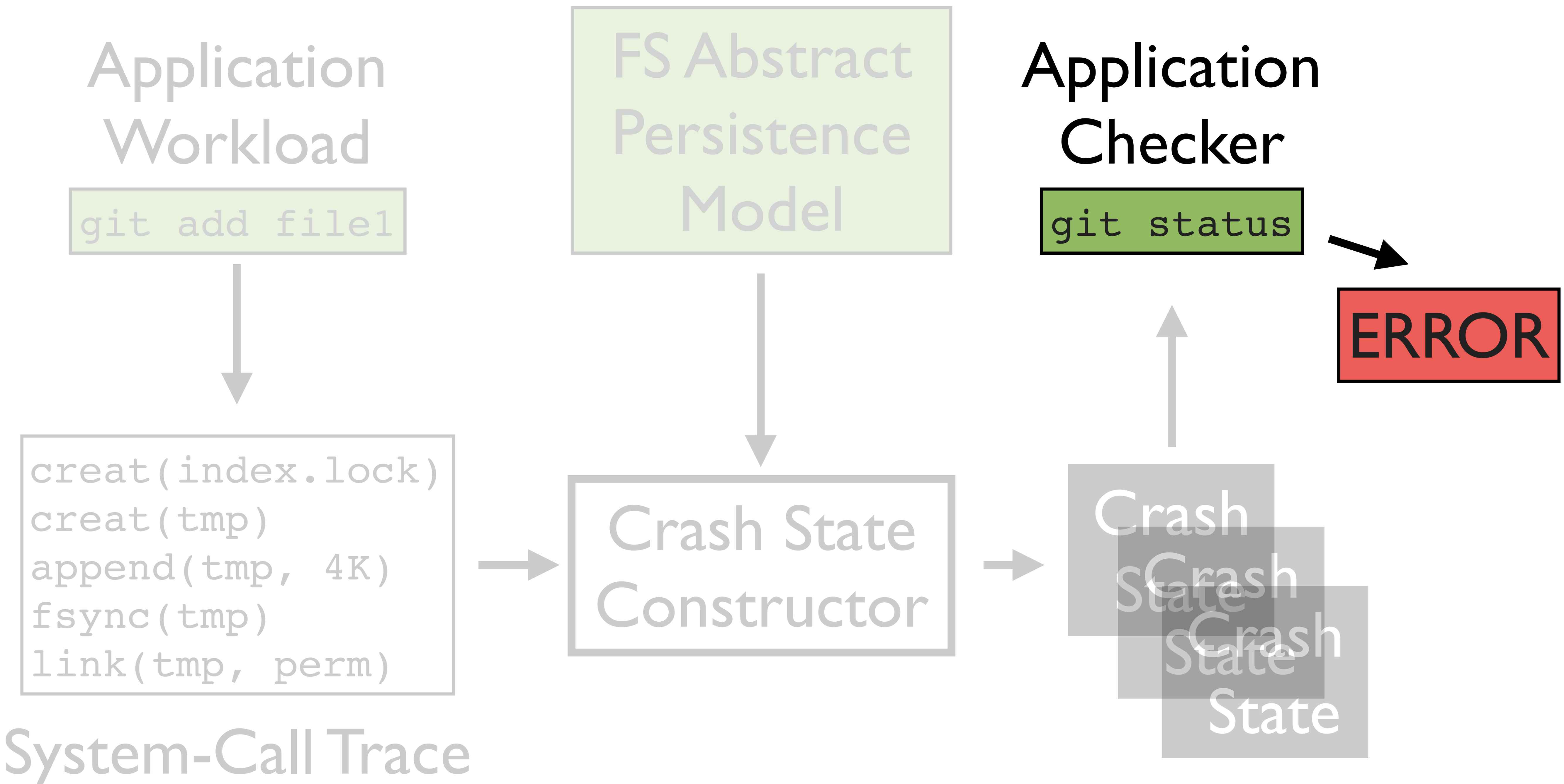
- Ex: ext4 orders writes of a file before its rename

APM defines which crash states are permitted

APM defines atomicity and ordering constraints

APM allow ALICE to model file-system behavior without file-system implementation

ALICE Overview



Finding Crash Vulnerabilities

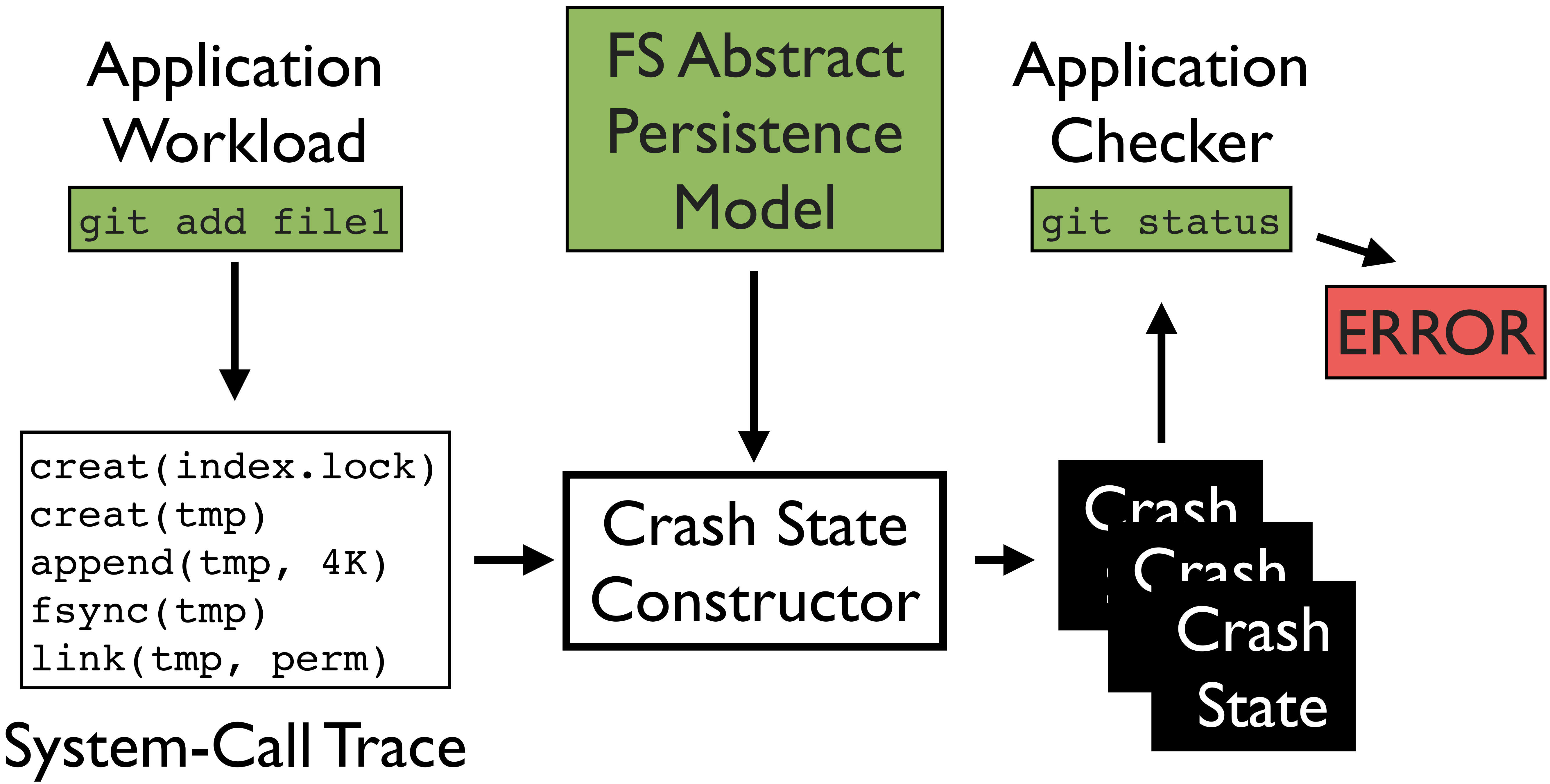
Identify persistence property violated

```
creat(index.lock)
creat(tmp)
append(tmp, 4K)
fsync(tmp)
link(tmp, perm)
```

Identify system calls involved

Identify source code lines involved

ALICE Overview



ALICE Limitations

Not complete

- does not execute all code paths in application
- does not explore all crash states
- does not test combinations of persistence property violations (ex: atomicity + ordering)

Cannot prove an update protocol is correct

Outline

Introduction

Background

Analyzing file systems with BOB

Analyzing applications with ALICE

Application Study

Conclusion and Future Work

Application Study

Used ALICE to study eleven applications

Version Control Systems	git	mercurial	
Key-Value Stores	LEVELDB	GDBM	LMDB
Relational Databases	SQLite	PostgreSQL	HyperSQL
Distributed Systems	hadoop HDFS	Apache ZooKeeper	
Virtualization Platforms	vmware Player		

Study Goals

Analyzed applications using weak APM

- Minimum constraints on possible crash states

Sought to answer:

- Which persistence properties do applications depend upon?
- What are the consequences of vulnerabilities?
- How many vulnerabilities occur on today's file systems?

Did not seek to compare applications

Study: Setup

What is correct behavior for an application?

- We use guarantees in documentation
- In case of no documentation, we assume typical user expectations ("committed data is durable")

Configurations change guarantees

- We test each configuration separately
- Tested 34 configurations across 11 applications

Post-crash, we run all appropriate application recovery mechanisms

Example: Git

```
mkdir(o/x)
creat(o/x/tmp_y)
append(o/x/tmp_y)
fsync(o/x/tmp_y)
link(o/x/tmp_y, o/x/y)
unlink(o/x/tmp_y)
```

store object

```
do(store object)
creat(branch.lock)
append(branch.lock)
append(branch.lock)
append(logs/branch)
append(logs/HEAD)
rename(branch.lock,x/branch)
stdout("finished commit")
```

git commit

Example: Git

```
mkdir(o/x)
creat(o/x/tmp_y)
append(o/x/tmp_y)
fsync(o/x/tmp_y)
link(o/x/tmp_y, o/x/y)
unlink(o/x/tmp_y)
```

store object

```
do(store object)
creat(branch.lock)
append(branch.lock)
append(branch.lock)
append(logs/branch)
append(logs/HEAD)
rename(branch.lock,x/branch)
stdout("finished commit")
```

Atomicity

git commit

Example: Git

Ordering

```
mkdir(o/x)
creat(o/x/tmp_y)
append(o/x/tmp_y)
fsync(o/x/tmp_y)
link(o/x/tmp_y, o/x/y)
unlink(o/x/tmp_y)
```

store object

```
do(store object)
creat(branch.lock)
append(branch.lock)
append(branch.lock)
append(logs/branch)
append(logs/HEAD)
rename(branch.lock,x/branch)
stdout("finished commit")
```

git commit

Example: Git

Ordering

```
mkdir(o/x)
creat(o/x/tmp_y)
append(o/x/tmp_y)
fsync(o/x/tmp_y)
link(o/x/tmp_y, o/x/y)
unlink(o/x/tmp_y)
```

store object

```
do(store object)
creat(branch.lock)
append(branch.lock)
append(branch.lock)
append(logs/branch)
append(logs/HEAD)
rename(branch.lock,x/branch)
stdout("finished commit")
```

git commit

Example: Git

```
mkdir(o/x)
creat(o/x/tmp_y)
append(o/x/tmp_y)
fsync(o/x/tmp_y)
link(o/x/tmp_y, o/x/y)
unlink(o/x/tmp_y)
```

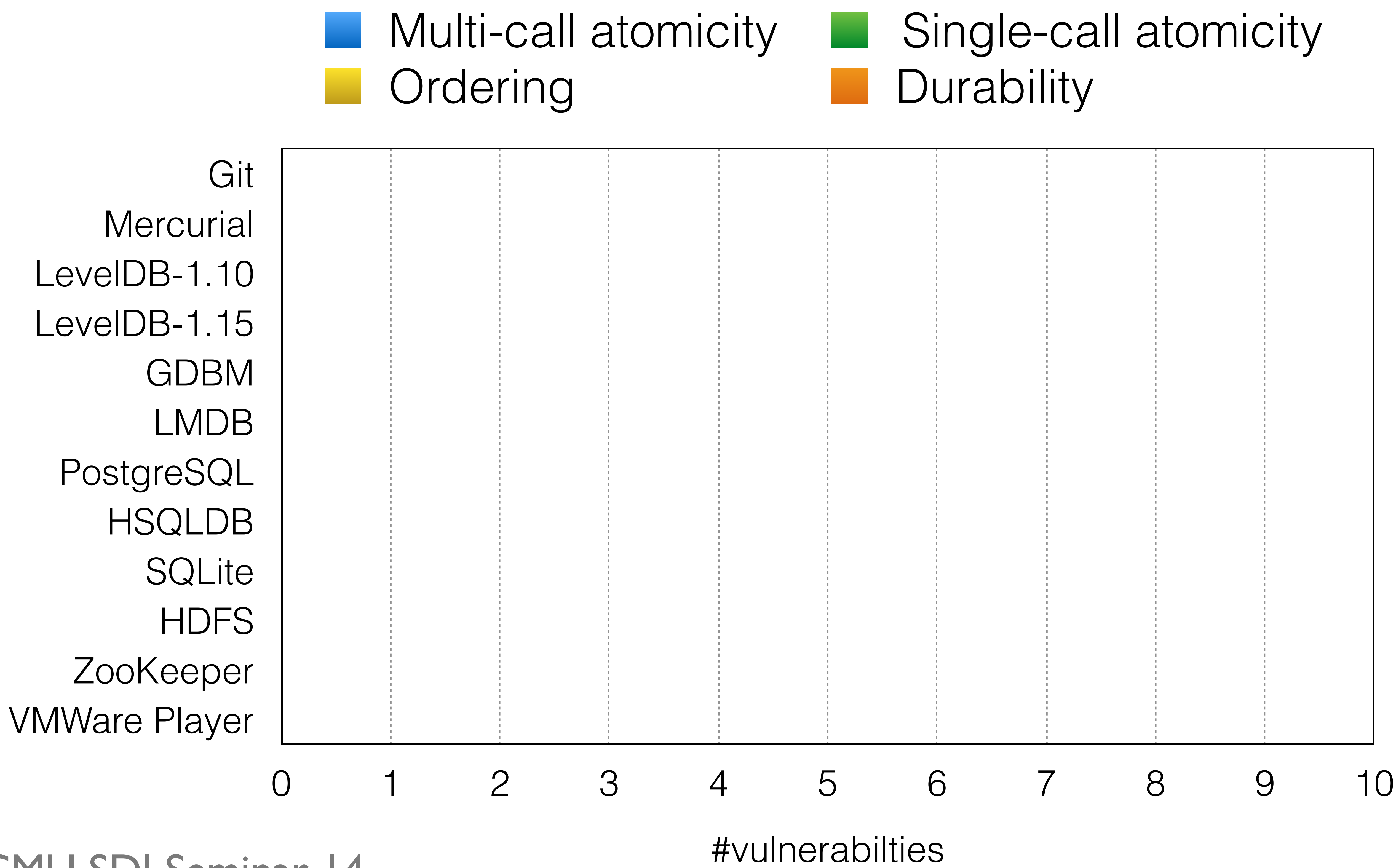
store object

```
do(store object)
creat(branch.lock)
append(branch.lock)
append(branch.lock)
append(logs/branch)
append(logs/HEAD)
rename(branch.lock,x/branch)
stdout("finished commit")
```

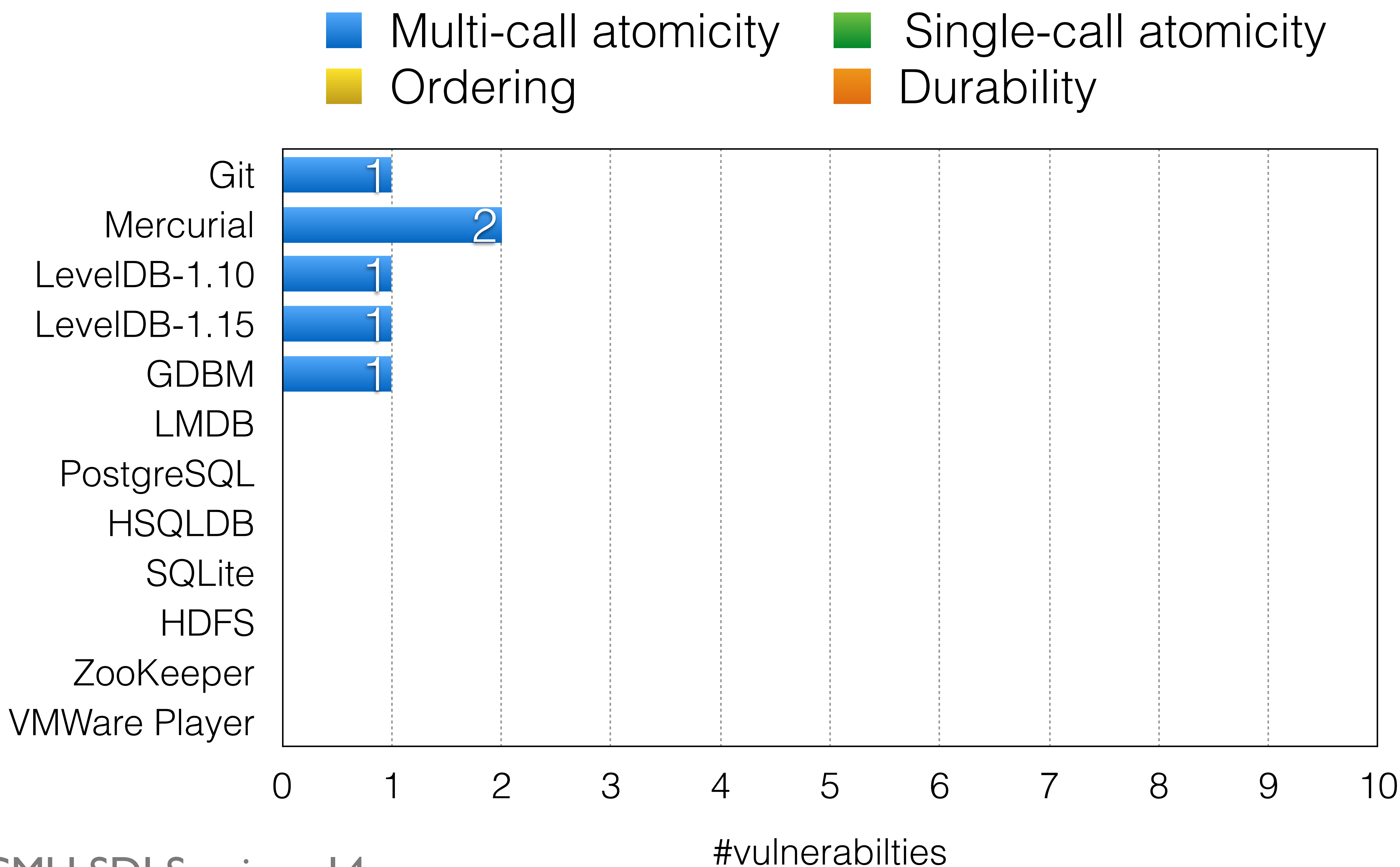
Durability

git commit

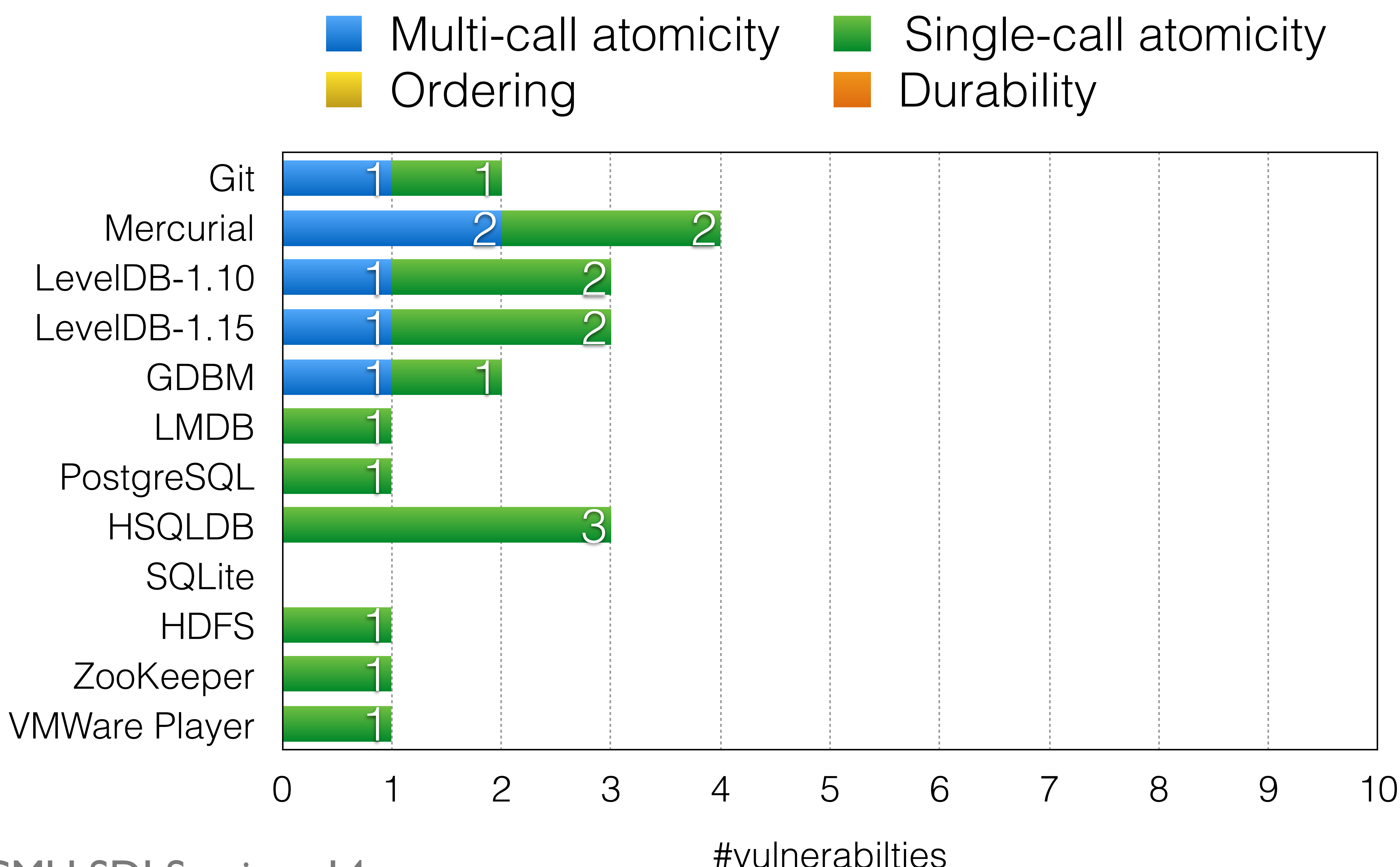
Vulnerability Types



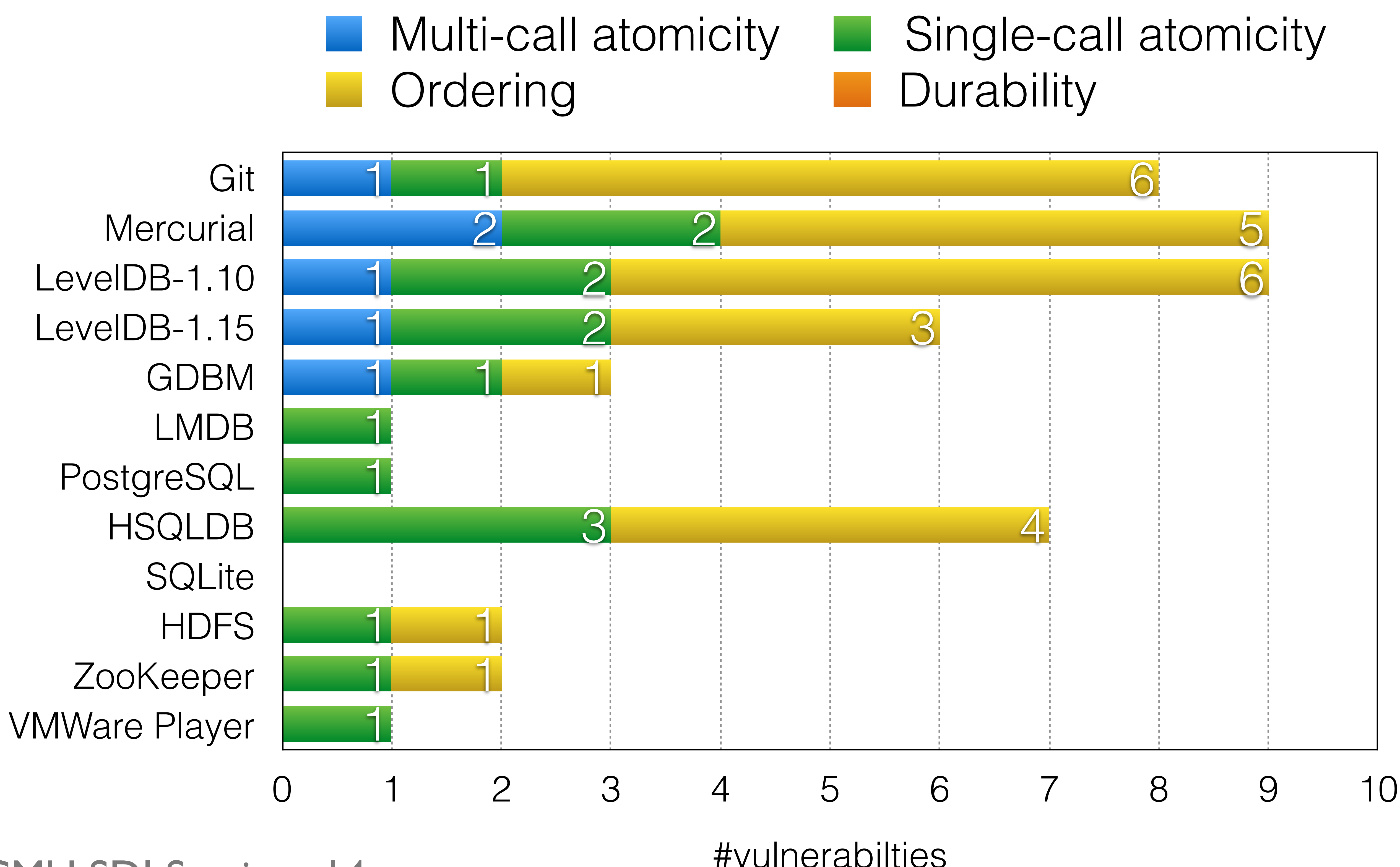
Vulnerability Types



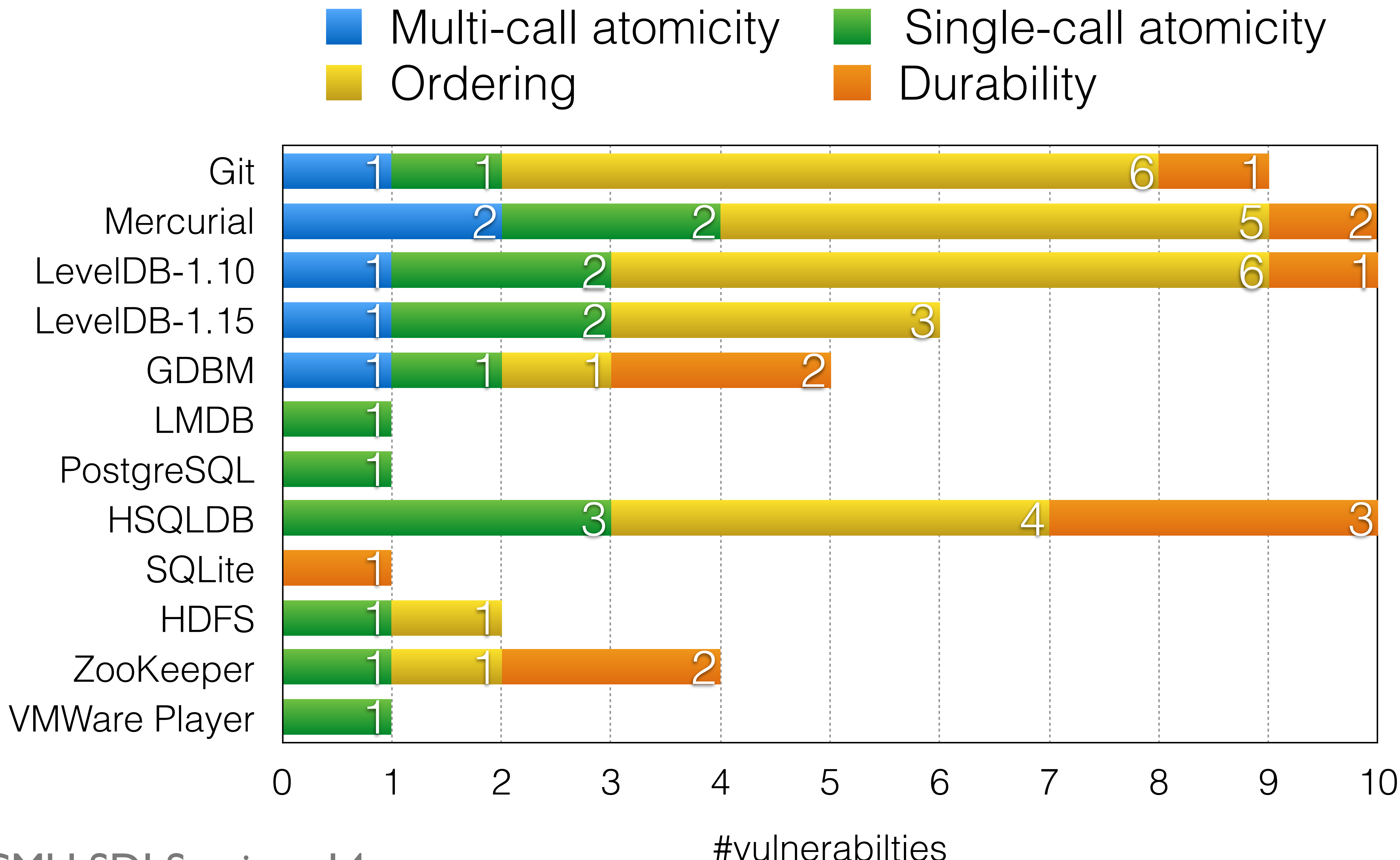
Vulnerability Types



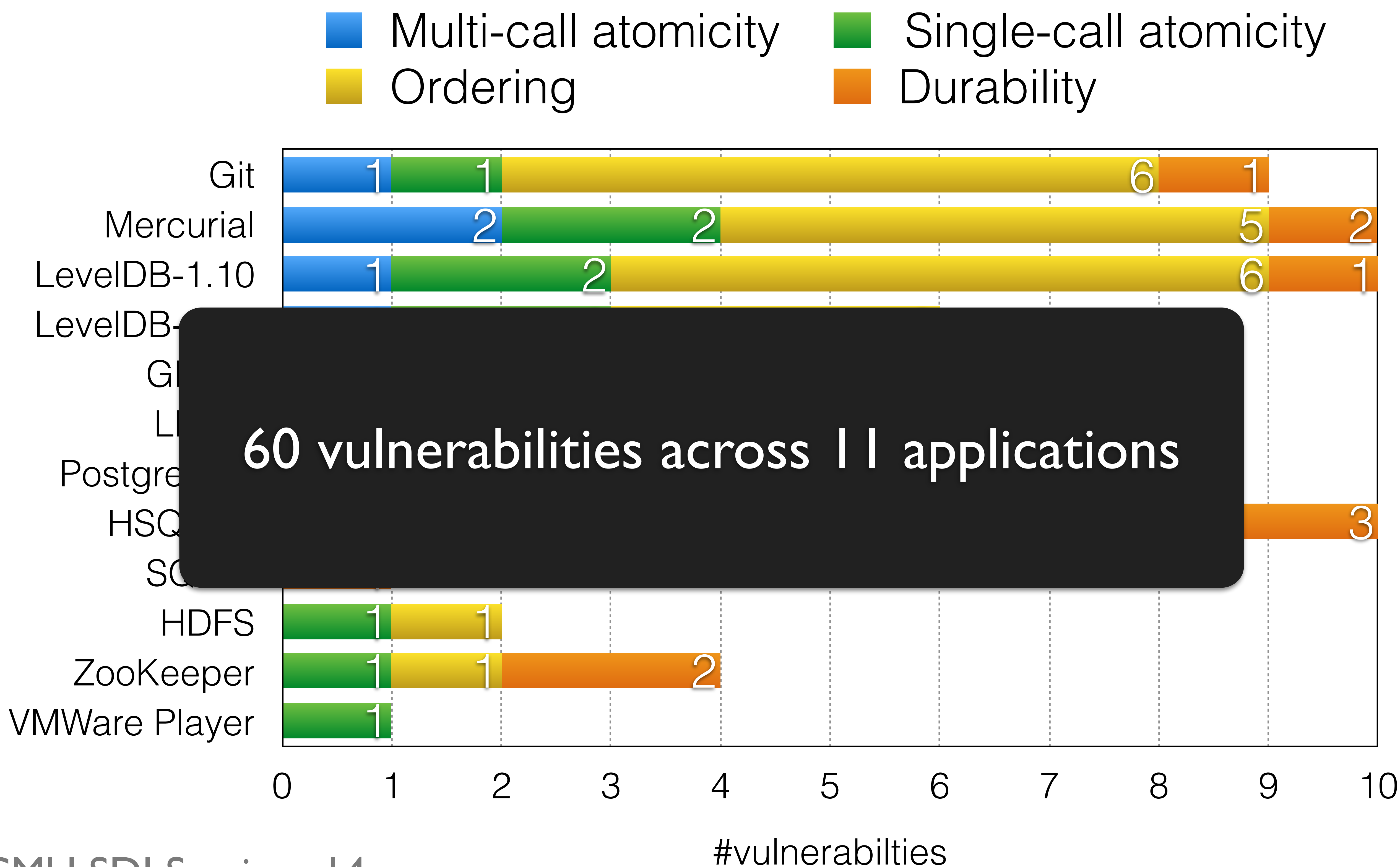
Vulnerability Types



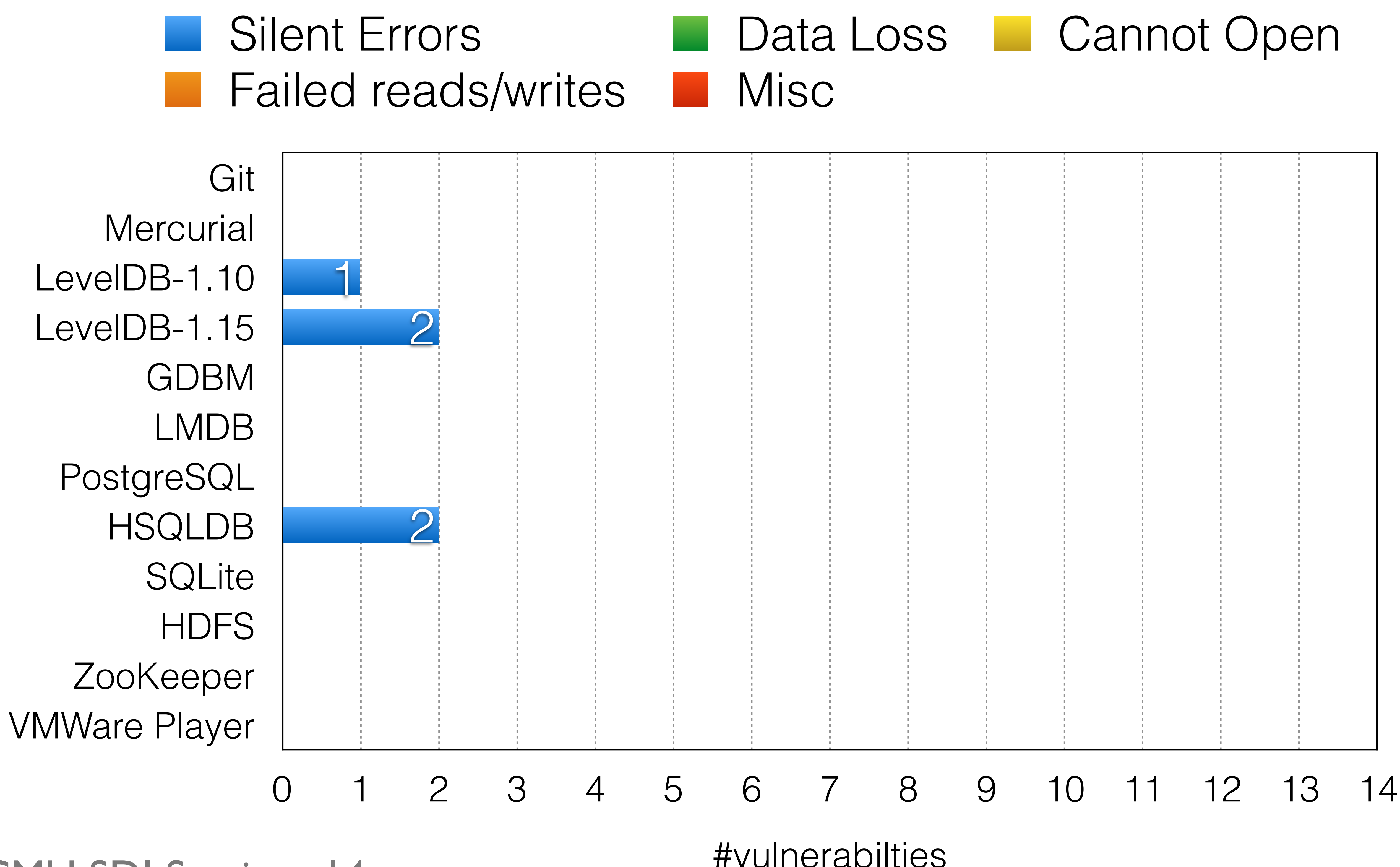
Vulnerability Types



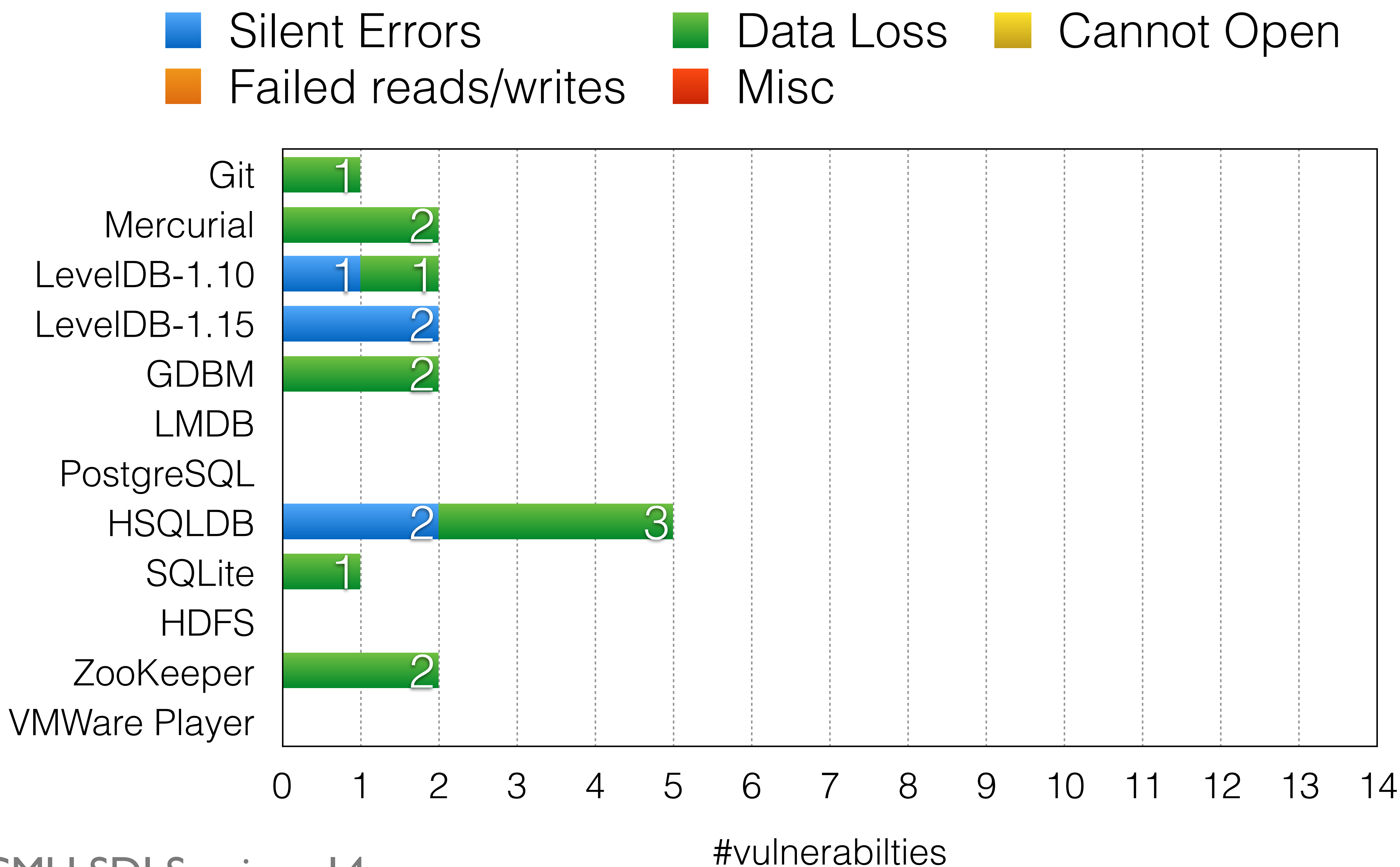
Vulnerability Types



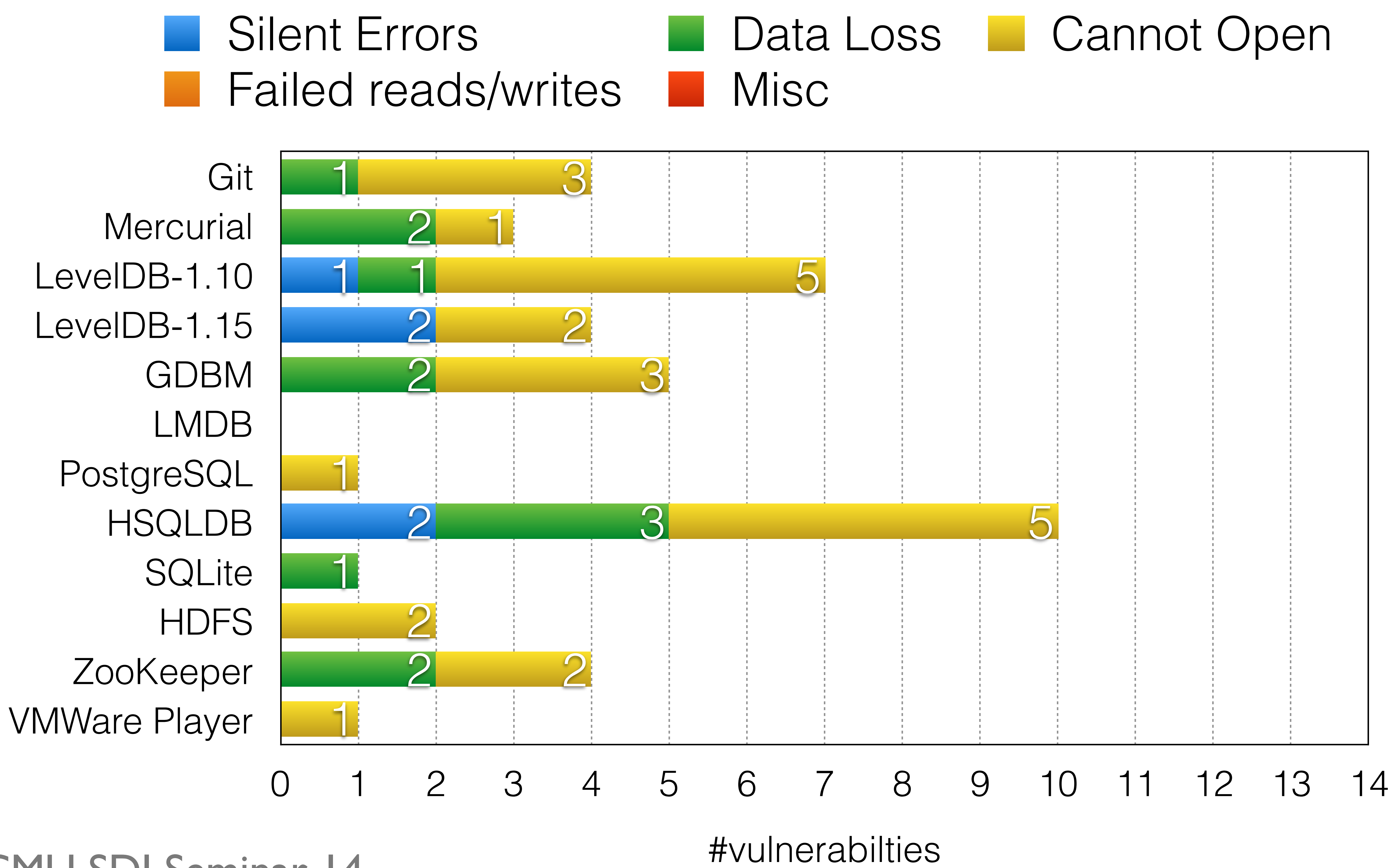
Vulnerability Consequences



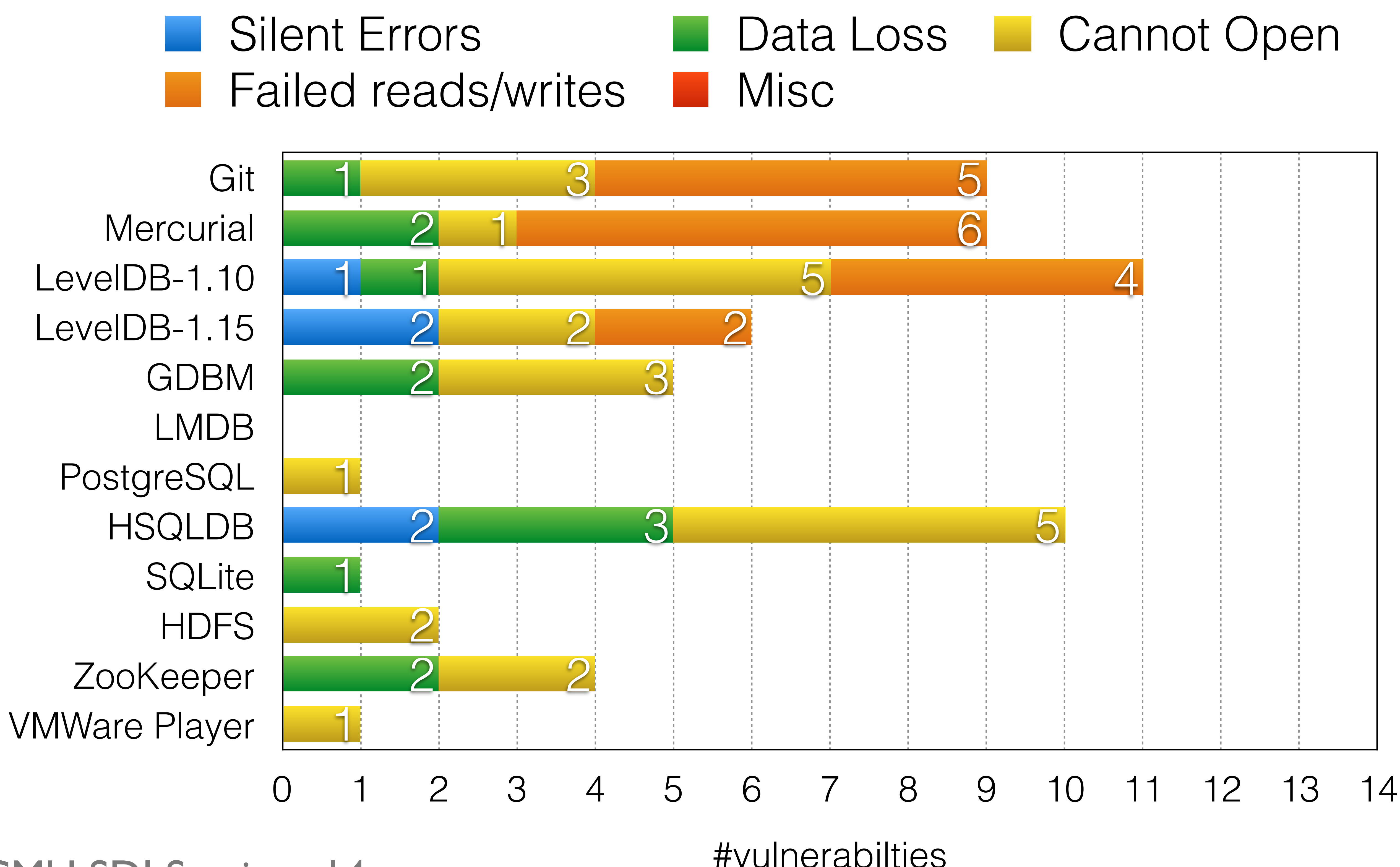
Vulnerability Consequences



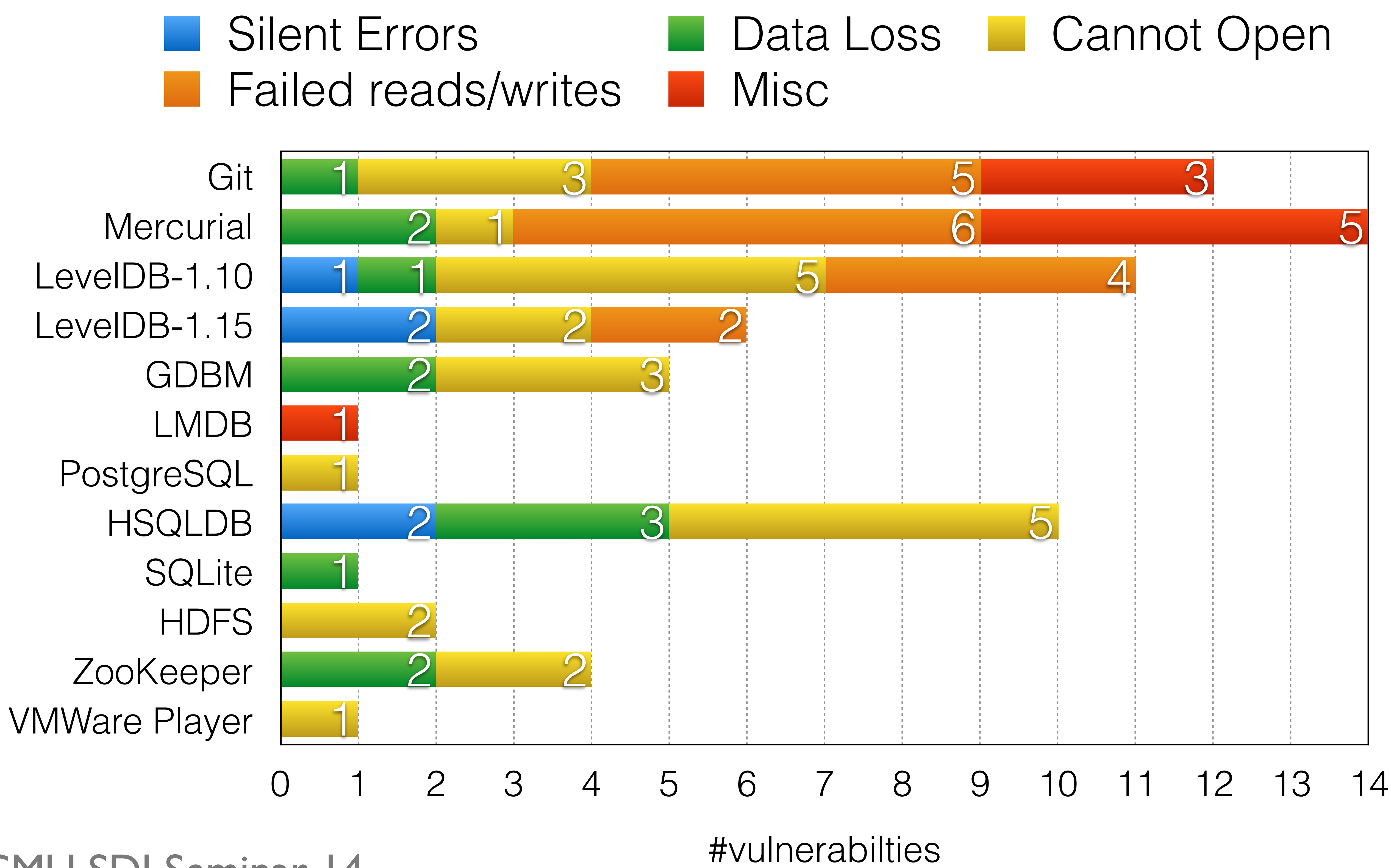
Vulnerability Consequences



Vulnerability Consequences



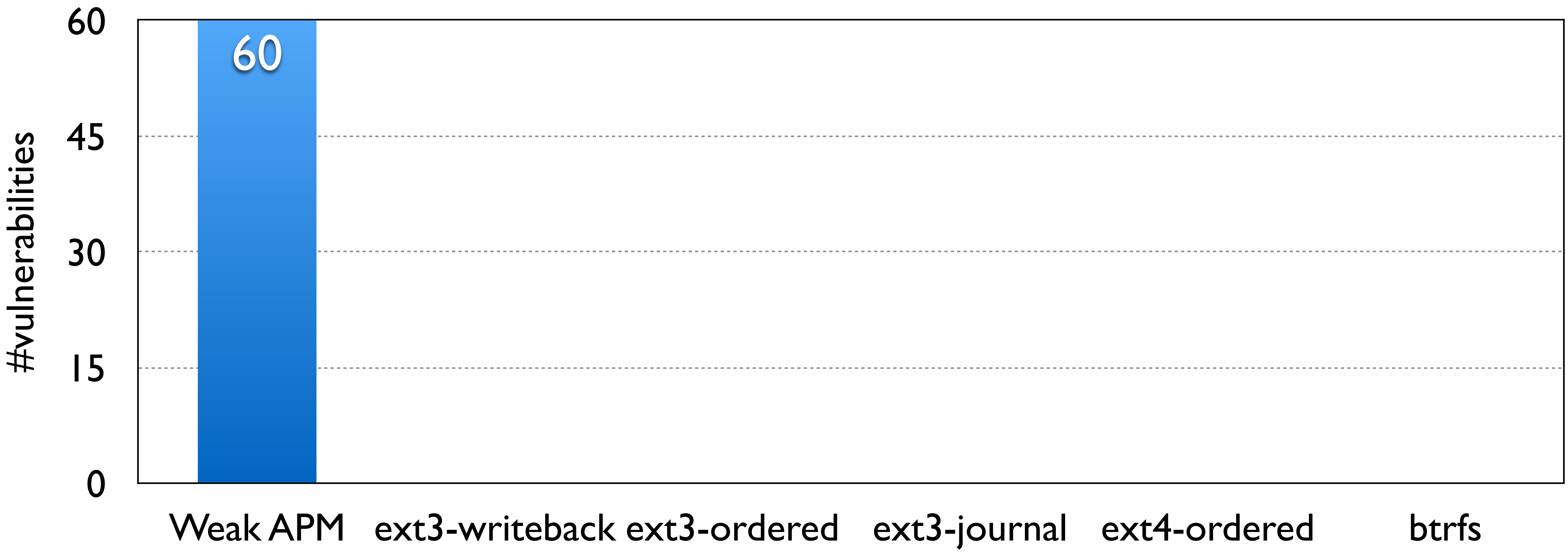
Vulnerability Consequences



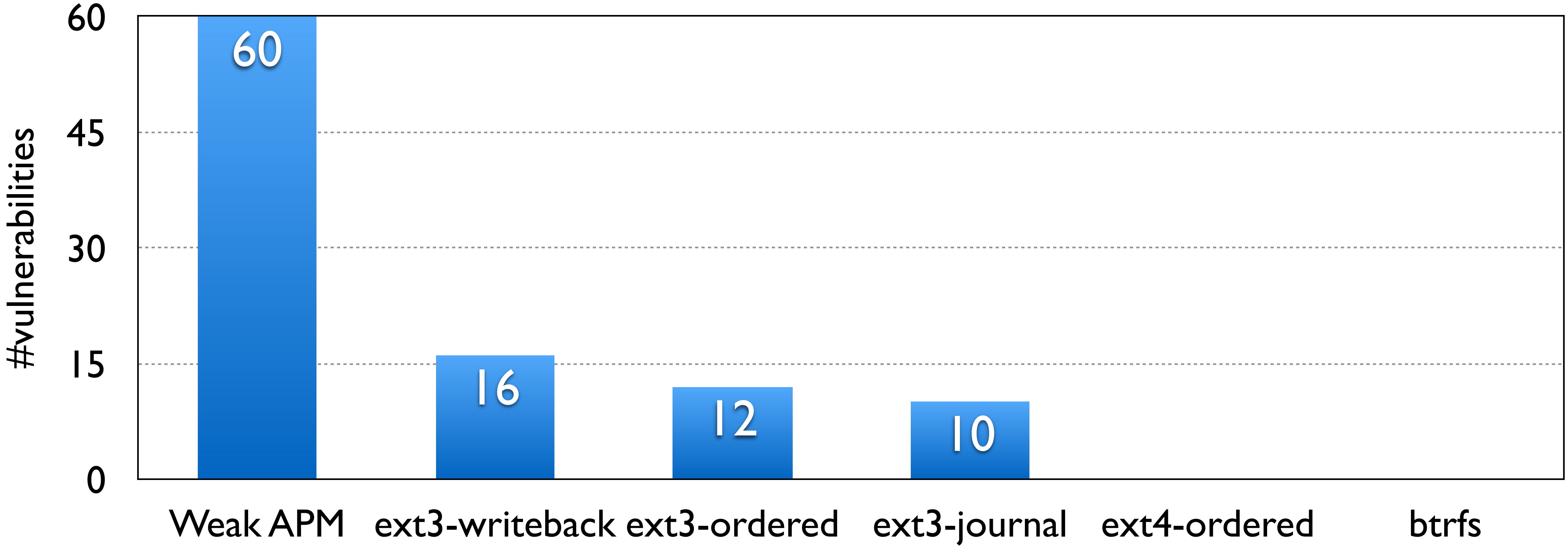
Vulnerability Consequences

Many vulnerabilities result in data loss, silent errors, and failed reads/writes

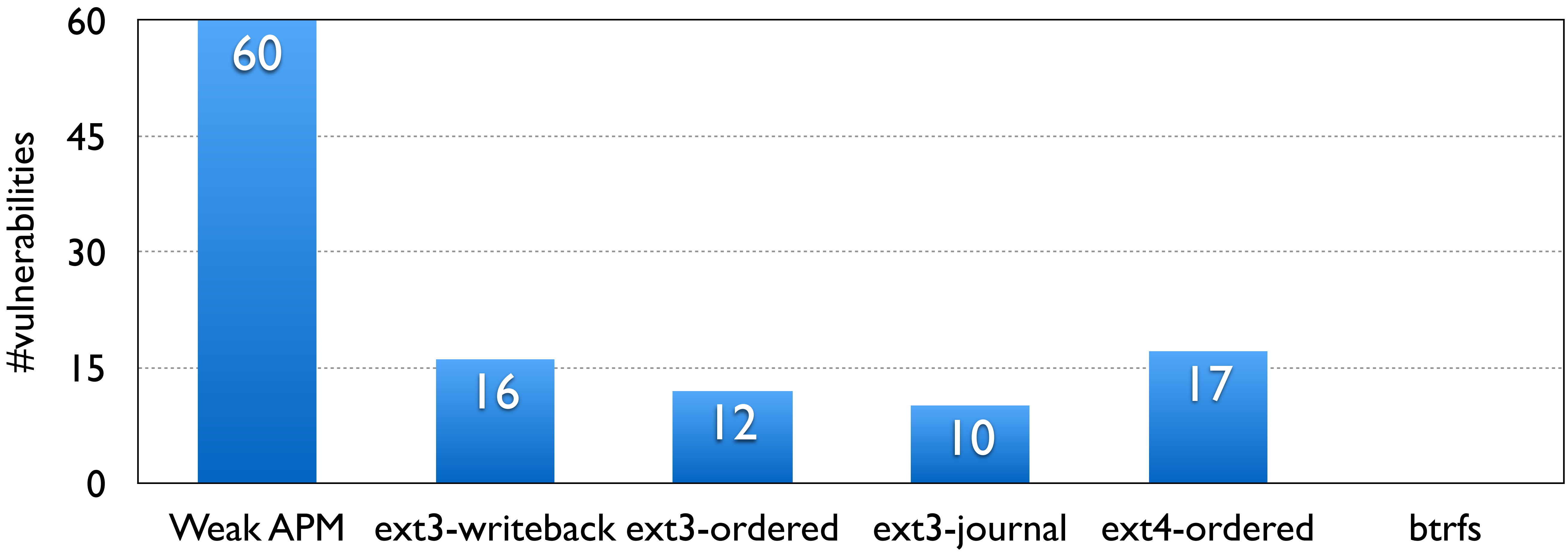
Vulnerabilities on Current File Systems



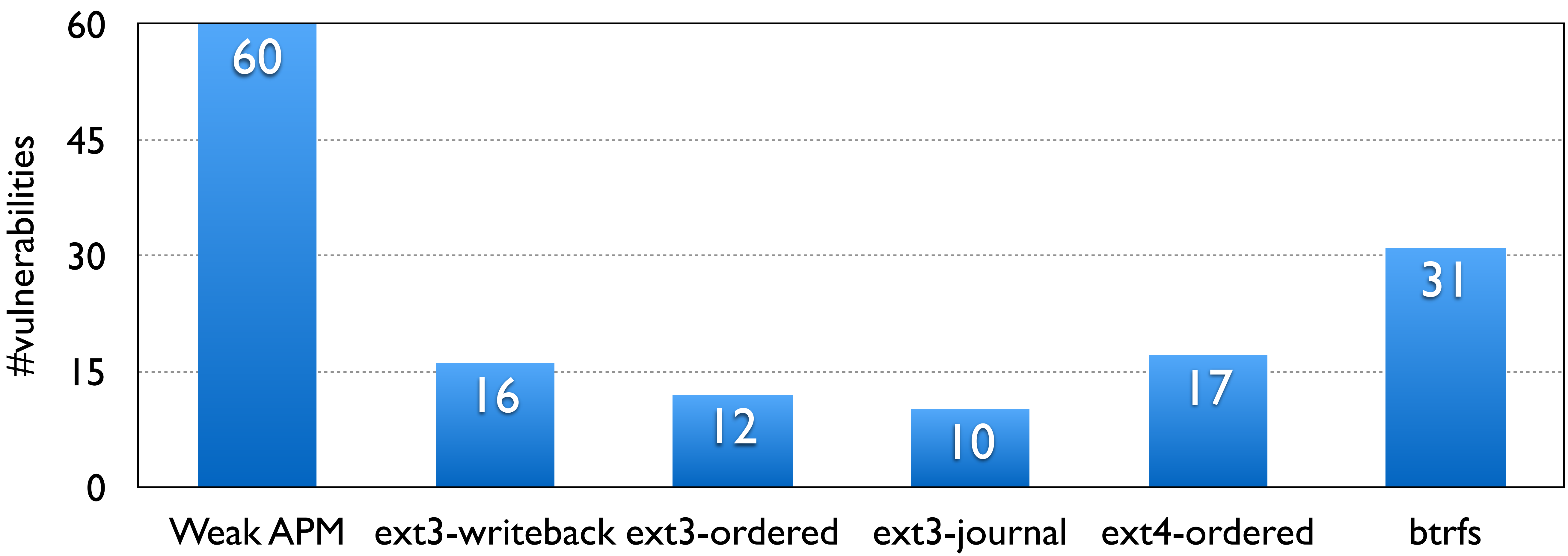
Vulnerabilities on Current File Systems



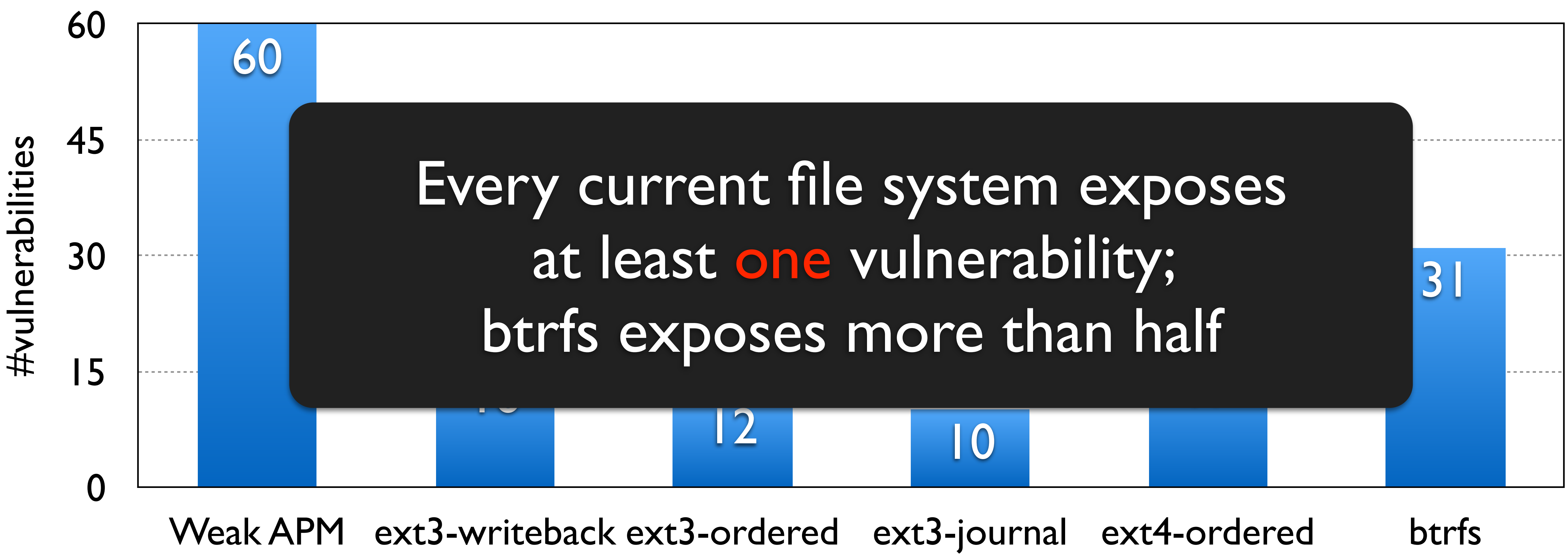
Vulnerabilities on Current File Systems



Vulnerabilities on Current File Systems



Vulnerabilities on Current File Systems



Every current file system exposes
at least one vulnerability;
btrfs exposes more than half

Observations

Applications very careful in overwriting user data

- None required atomicity for multi-block overwrites

Applications not as careful in appending to logs

- Multi-block appends require prefix atomicity
- Ex: write(“ABC”) should result in “A”/“AB”/“ABC”

Atomicity across system calls doesn't seem useful

Observations

Update protocols spread over layers and files

- Ex: HSQLDB has 3 consecutive `fsync()` calls

Recovery code is poorly written and untested

- Ex: LevelDB recovery does not correct errors

Documentation unclear or misleading

- SQLite by default does not provide durability
- `GDBM_SYNC` does not ensure durability

Reporting Vulnerabilities

Developers generally suspicious when we reported vulnerabilities

- Dev #1: "Maybe it is the disk"
- Dev #2: "File systems don't behave that way"

Tough to reproduce without tools like ALICE

Developers acting on five vulnerabilities

- Vulnerabilities in LevelDB, HDFS, ZooKeeper

Outline

Introduction

Background

Analyzing file systems with BOB

Analyzing applications with ALICE

Application Study

Conclusion and Future Work

Summary

Built BOB to study persistence properties

- Studied 16 configurations of 6 file systems
- Persistence properties vary widely

Built ALICE to study application-level crash-consistency protocols

- Studied 11 applications
- Found 60 vulnerabilities across all applications

Application-Level Consistency in the Cloud

Cloud computing and software-defined storage make this problem worse

- Increased storage-stack diversity
- Multiple storage media, file systems, etc.

Applications deployed in multiple environments

- Cant rely on specific persistence properties

Portability in the Cloud

Need to match application requirements to storage-stack guarantees

Challenges:

- specifying application requirements
- computing how layers build on each other to provide guarantees
- checking if requirements are met

Matching Applications to Stacks

Use a formal language (like Isar) to specify application requirements

Specify high-level design of stack layers in Isar

Use proof-assistants (like Isabelle) to verify that requirements are provided by stack

Do this on-the-fly as storage stacks are constructed

Benefits of Matching

Currently, applications are coarsely matched to storage stacks

Stacks provide either too much or too little

Verifying application correctness allows construction of optimal stacks

- Cheapest stack that satisfies application requirements

Conclusion

Applications are moving to the cloud

- For performance
- For ease-of use or availability
- correctness is often forgotten

To unlock true potential of cloud, portable applications need to be created

Figuring out application requirements is the first step towards this vision

Thank You

Source code

http://research.cs.wisc.edu/adsl/Software/alice/

Questions?

WISCONSIN
UNIVERSITY OF WISCONSIN–MADISON

WISDOM