Towards Robust File System Checkers

Om Rameshwar Gatla[α], *Muhammad Hameed*[α], *Mai Zheng*[α]

Viacheslav Dubeyko, Adam Manzanares, Filip Blagojevic, Cyril Guyot, Robert Mateescu

[α] **New Mexico State University** **Western Digital Research**

Motivation

Subject: Update: HPCC Power Outage
Date: Monday, January 11, 2016 at 8:50:17 AM Central Standard Time
From: HPCC - Support
Attachments: image001.png, image003.png

TEXAS TECH UNIVERSITY
Information Technology Division

High Performance Computing Center

- Recovery procedure was interrupted
- Severe data loss reported

To All HPCC Customers and Partners,

As we have informed you earlier, the Experimental Sciences Building experienced a major power outage Sunday, Jan. 3 and another set of outages Tuesday, Jan. 5 that occurred while file systems were being recovered from the first outage. As a result, there were major losses of important parts of the file systems for the work, scratch and certain experimental group special Lustre areas.

The HPCC staff have been working continuously since these events on recovery procedures to try to restore as much as possible of the affected file systems. These procedures are extremely time-consuming, taking days to complete in some cases. Although about a third of the affected file systems have been recovered, work continues on this effort and no time estimate is possible at present.

Motivation

Subject: Update: HPCC Power Outage
Date: Monday, January 11, 2016 at 8:50:17 AM Central Standard Time
From: HPCC - Support
Attachments: image001.png, image003.png

TEXAS TECH UNIVERSITY
Information Technology Division

High Performance Computing Center

- Recovery procedure was interrupted
- Severe data loss reported

- Lustre's backend ldiskfs is a variant of EXT4
- Lustre File system checker (lfsck) relies on EXT4 checker (e2fsck)
- Overall recovery is complicated (several days to fix)

To All HPCC Customers and Partners,

As we have informed you earlier, the Experimental Sciences Building experienced a major power outage Sunday, Jan. 3 and another set of outages Tuesday, Jan. 5 that occurred while file systems were being recovered from the first outage. As a result, there were major losses of important parts of the file systems for the work, scratch and certain experimental group special Lustre areas.

The HPCC staff have been working continuously since these events on recovery procedures to try to restore as much as possible of the affected file systems. These procedures are extremely time-consuming, taking days to complete in some cases. Although about a third of the affected file systems have been recovered, work continues on this effort and no time estimate is possible at present.

Motivation

Research questions:
- Are existing checkers resilient to faults?
- How to build a robust checker?

Outline

- Motivation
- Background & Related Work
- Are existing checkers resilient to faults?
- How to build robust checkers?
- Evaluation
- Conclusion

Background & Related Work

File systems are designed to organize data and maintain data integrity

Background & Related Work

File systems are designed to organize data and maintain data integrity

File systems may become corrupt despite various protection techniques
- E.g.: journaling , soft updates, copy-on-write, etc.

Background & Related Work

File systems are designed to organize data and maintain data integrity

File systems may become corrupt despite various protection techniques
- E.g.: journaling , soft updates, copy-on-write, etc.

File system checkers (fsck) recover a corrupted file system back to a consistent state
- E.g.: e2fsck, xfs-repair, etc.
- Some existing checkers exhibit logging mechanism:

File System	Checker	Logging Support
EXT 2/3/4	e2fsck	Yes
XFS	xfs_repair	No
F2FS	fsck.f2fs	No
BTRFS	btrfsck	No

Background & Related Work

Existing work for improving checkers
E.g.: ffsck[@FAST'13], SWIFT[@EUROSYS'12], SQCK[@OSDI'08]

Background & Related Work

Existing work for improving checkers
E.g.: ffsck[@FAST’13], SWIFT[@EUROSYS’12], SQCK[@OSDI’08]

Do not address one fundamental issue: ***Resilience in face of interruption***

Background & Related Work

Existing work for improving checkers
E.g.: ffsck[@FAST'13], SWIFT[@EUROSYS'12], SQCK[@OSDI'08]

Do not address one fundamental issue: ***Resilience in face of interruption***

Our Efforts:

Demonstrate that an interrupted checking could leave the file system in an uncorrectable state

One general solution to this issue

Outline

- Motivation
- Background & Related Work
- Are existing checkers resilient to faults?
- How to build robust checkers?
- Evaluation
- Conclusion

Are existing checkers resilient to faults?

A testing framework to interrupt checker

Two components:

Are existing checkers resilient to faults?

A testing framework to interrupt checker

Two components:

Component 1:
Corrupted images to trigger checker

Are existing checkers resilient to faults?

A testing framework to interrupt checker

Two components:

Component 1:
Corrupted images to trigger checker

Component 2:
Fault injection engine

Component 1: Corrupted images

Two methods to generate corrupted images:

Component 1: Corrupted images

Two methods to generate corrupted images:

Method 1: Collect test images provided by developers
- E.g.: test images in e2fsprogs
- Corruptions envisioned by developers
- Convenient

Component 1: Corrupted images

Two methods to generate corrupted images:

Method 1: Collect test images provided by developers
- E.g.: test images in e2fsprogs
- Corruptions envisioned by developers
- Convenient

Method 2: Corrupt metadata using file system debug tools
- E.g.: debugfs, xfs_db, etc.
- Cover more scenarios
- Flexible

Component 2: Fault Injection Engine

Build a fault injection engine "`rfsck-test`" using iSCSI driver

Component 2: Fault Injection Engine

Build a fault injection engine "`rfsck-test`" using iSCSI driver

Two modes of operation:

1. **Basic mode**
   Single iSCSI drive for one test image

2. **Advanced mode**
   Two iSCSI drives for one test image and one log device

Component 2: Fault Injection Engine

Build a fault injection engine "`rfsck-test`" using iSCSI driver

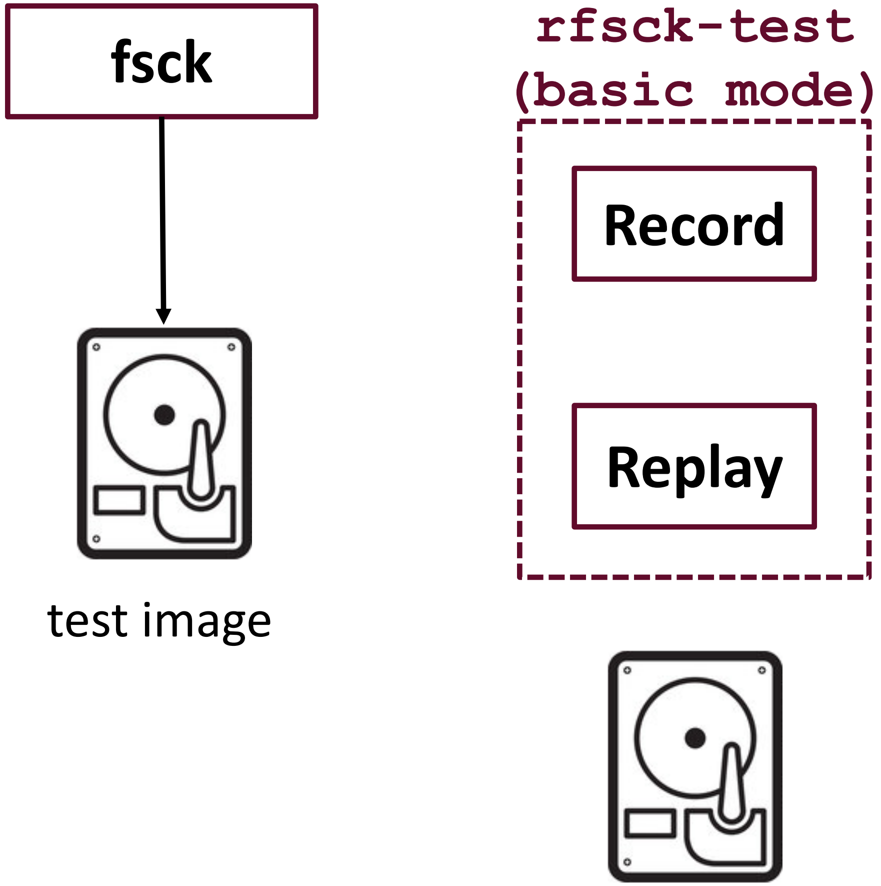
Two modes of operation:

1. **Basic mode** ➡ *For checkers without logging*
   Single iSCSI drive for one test image

2. **Advanced mode** ➡ *For checkers with logging*
   Two iSCSI drives for one test image and one log device

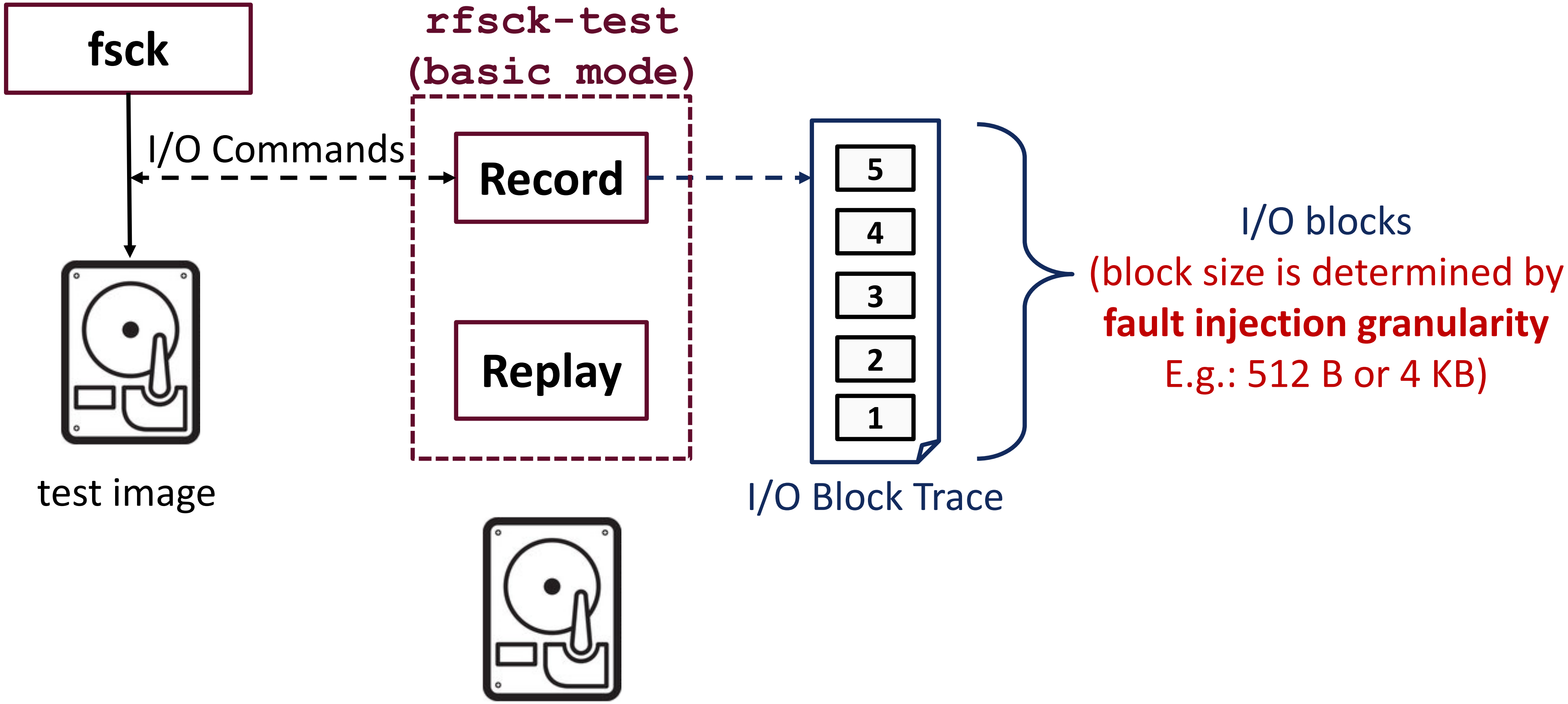
Fault Injection Engine: `rfsck-test`

Basic Mode



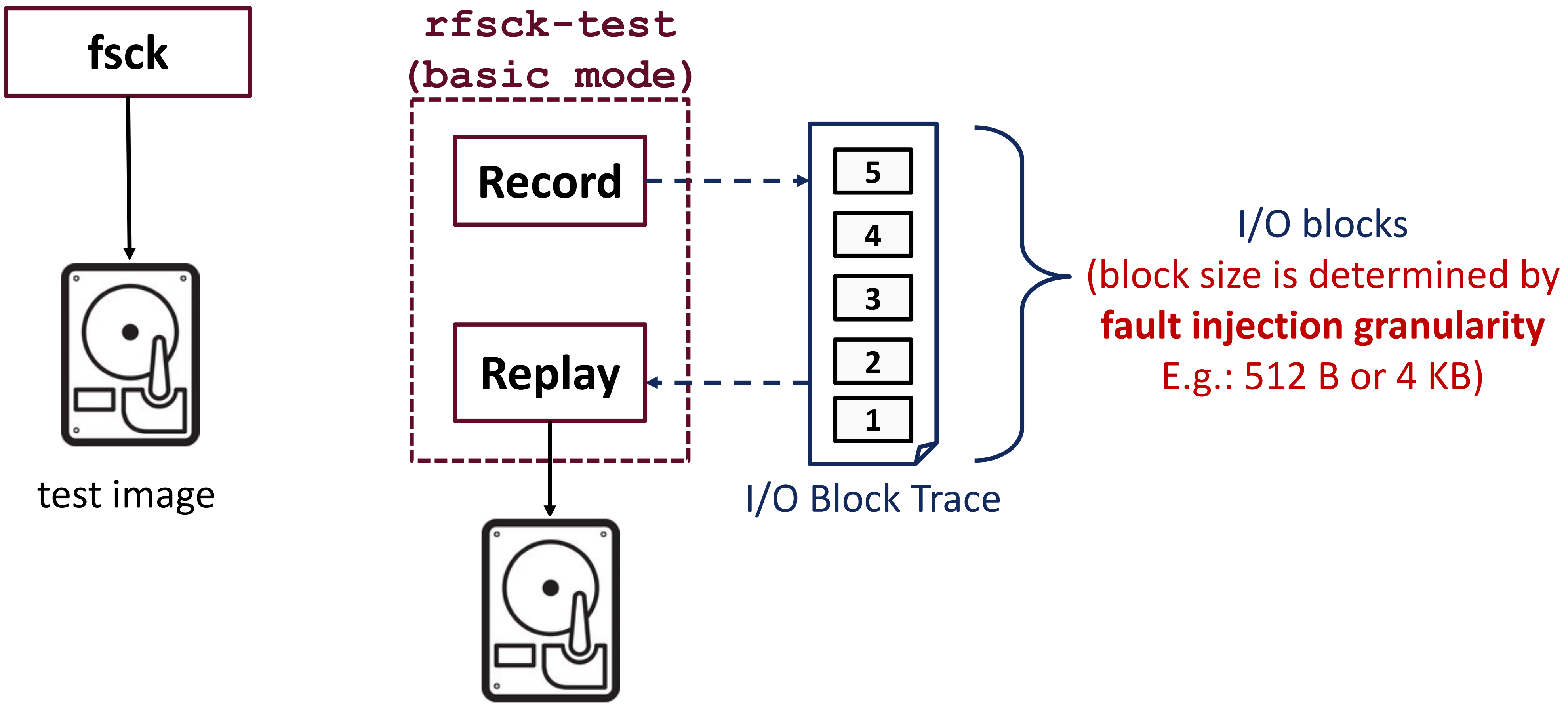
Fault Injection Engine: `rfsck-test`

Basic Mode



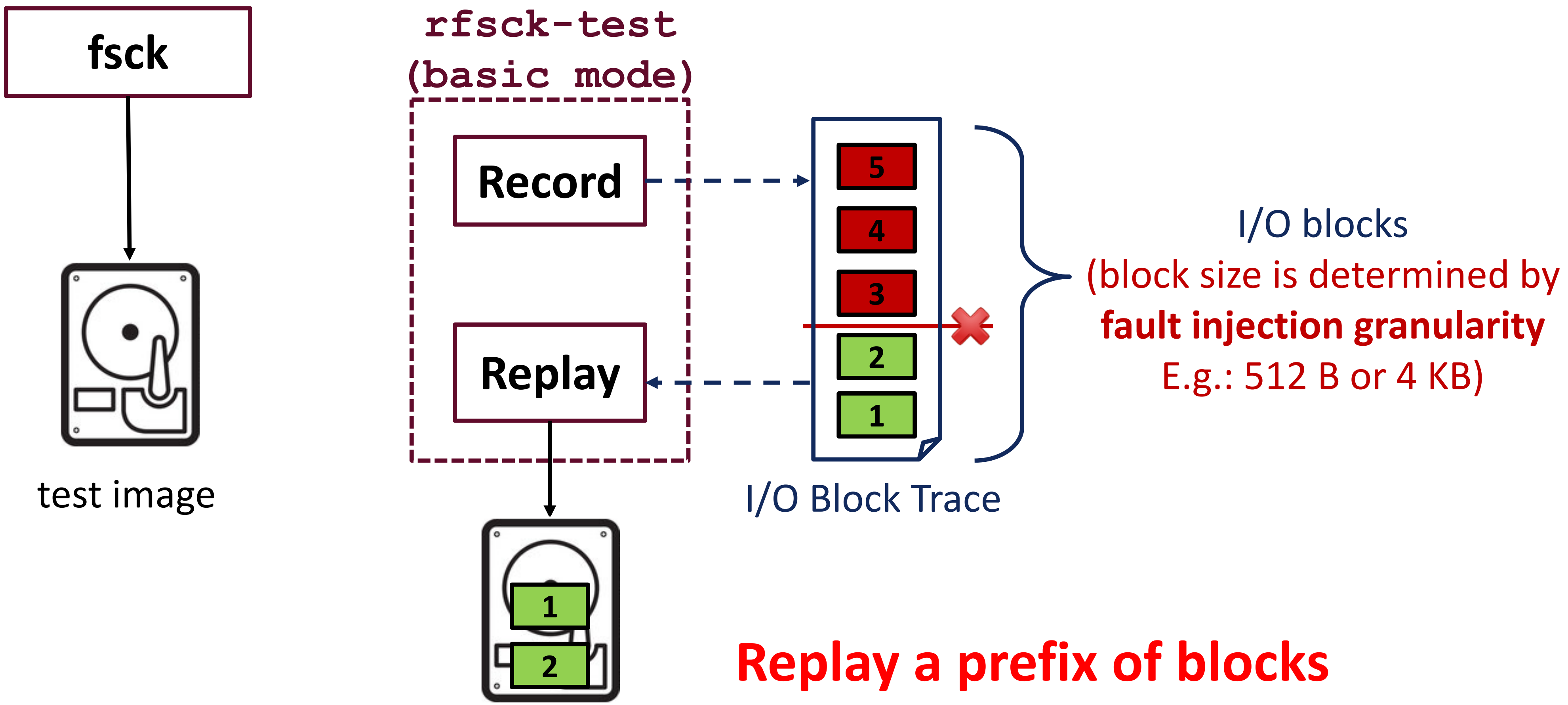
Fault Injection Engine: `rfsck-test`

Basic Mode



Fault Injection Engine: `rfsck-test`

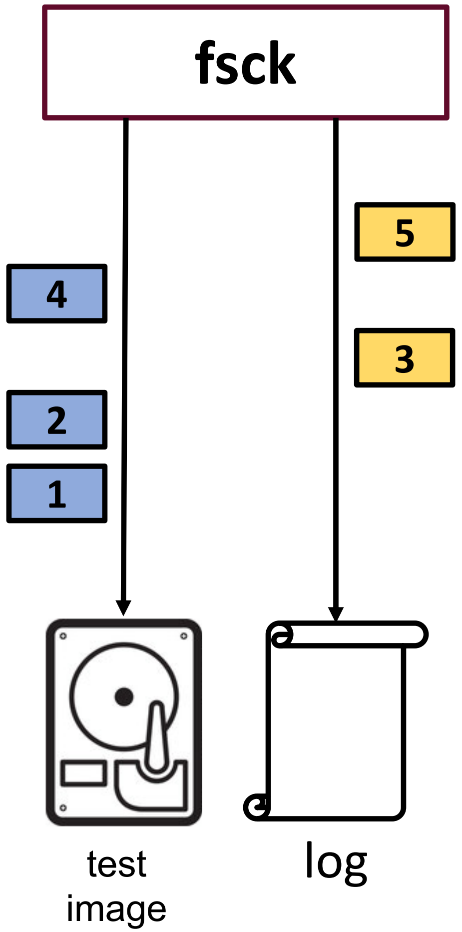
Basic Mode

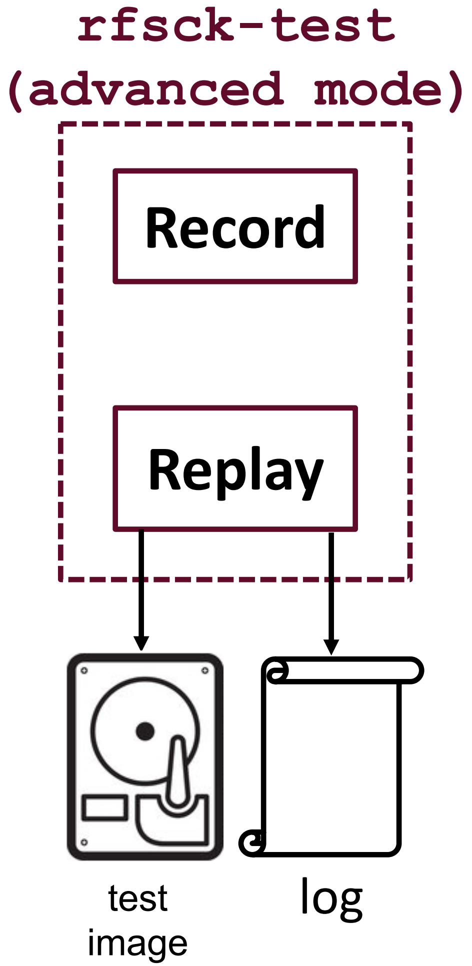


Replay a prefix of blocks

Fault Injection Engine: rfsck-test

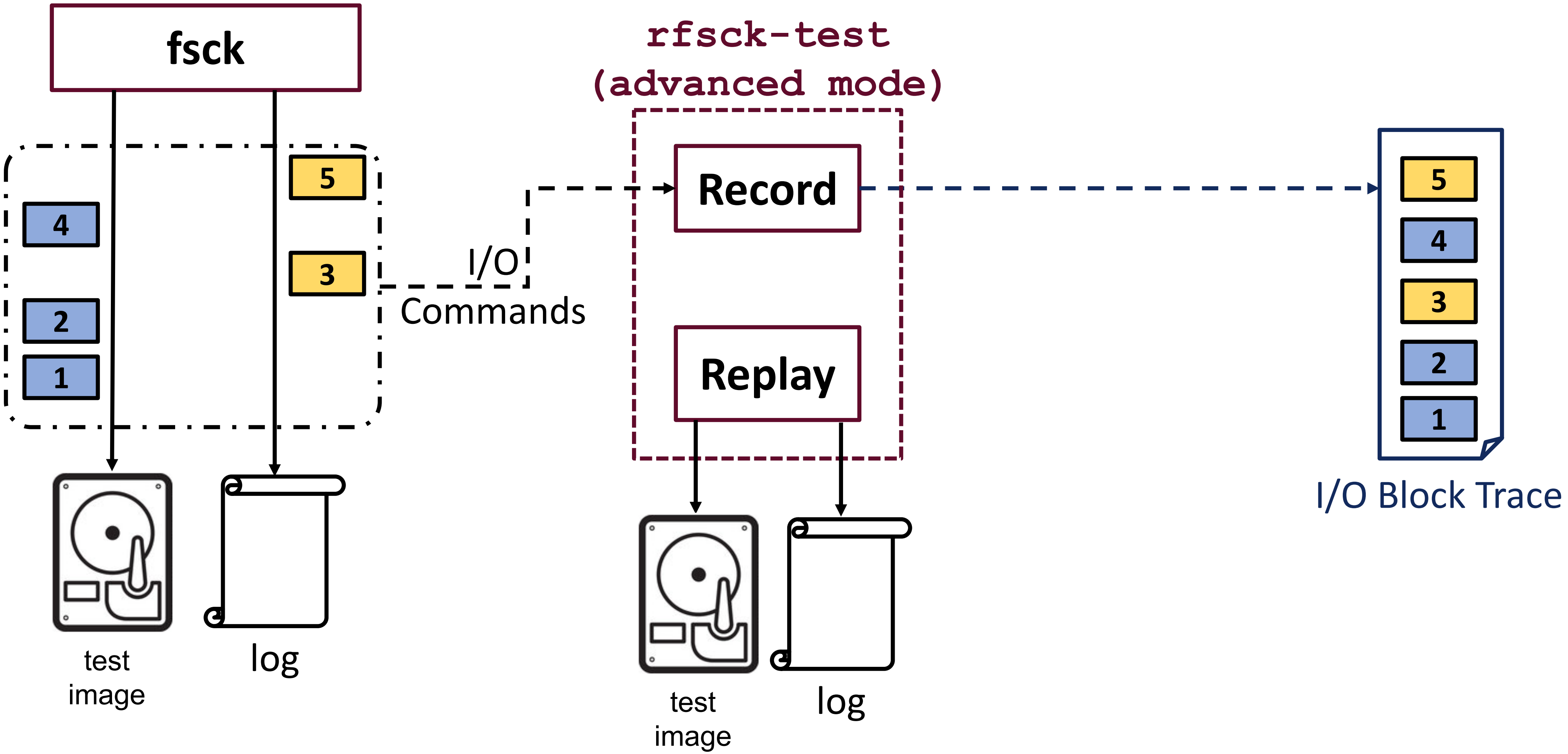
Advanced Mode





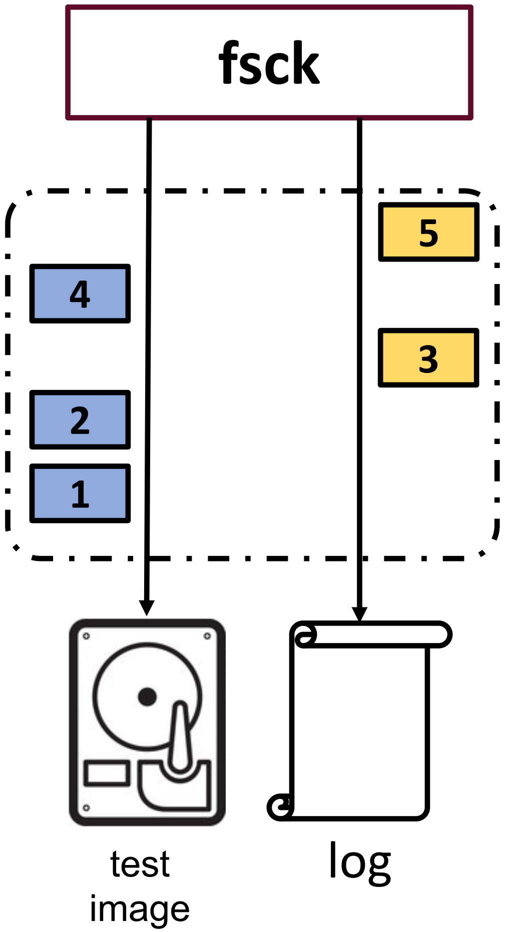
Fault Injection Engine: rfsck-test

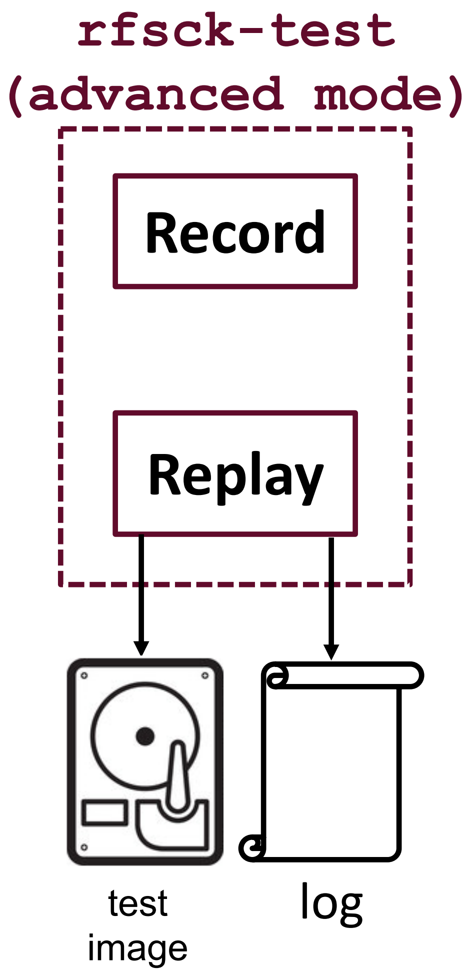
Advanced Mode

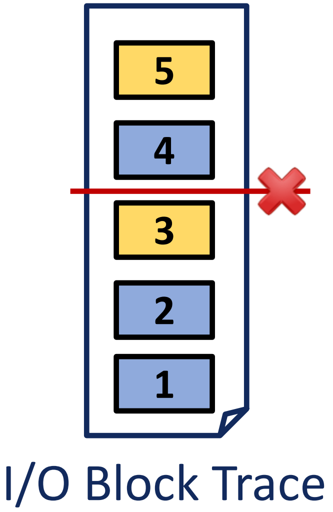


Fault Injection Engine: `rfsck-test`

Advanced Mode

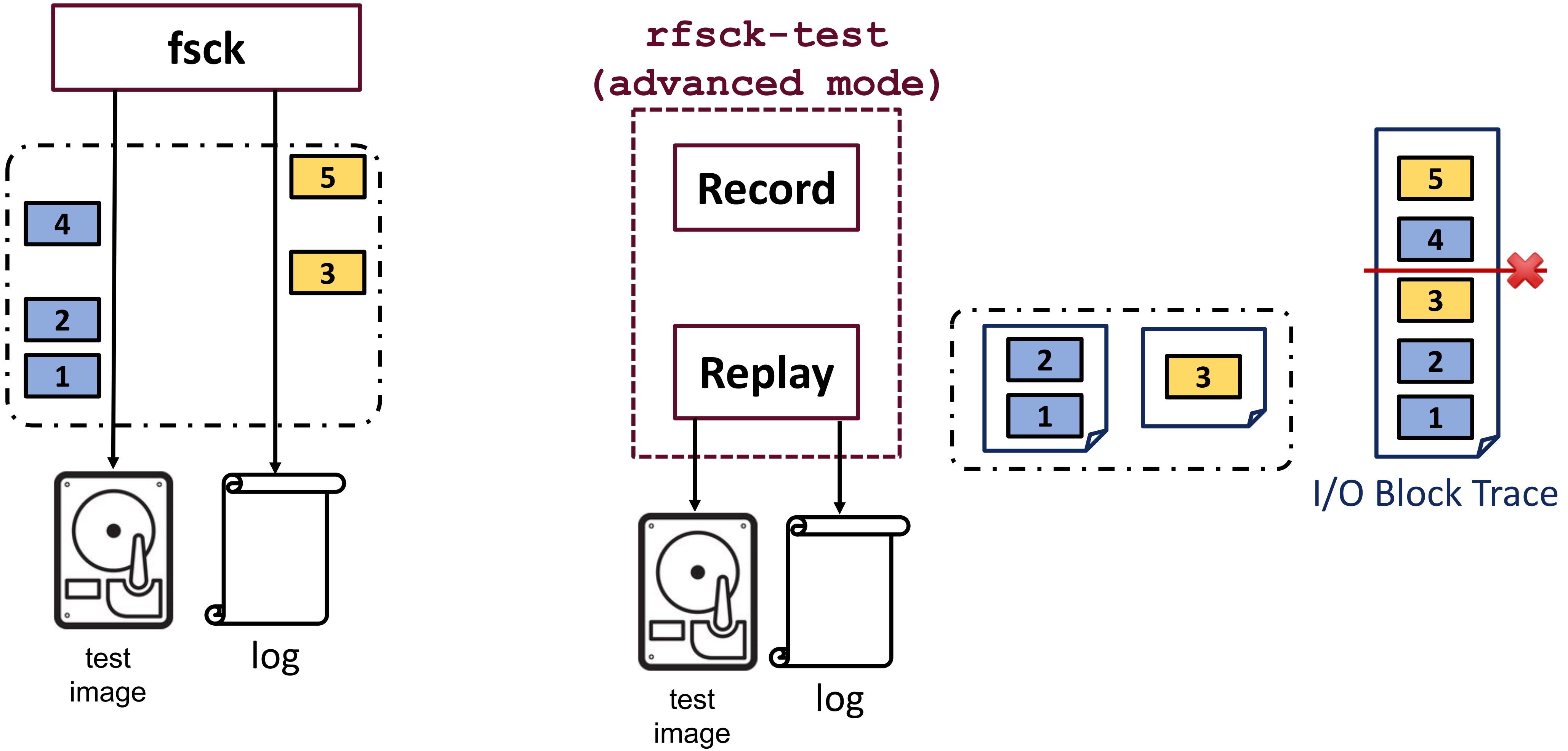






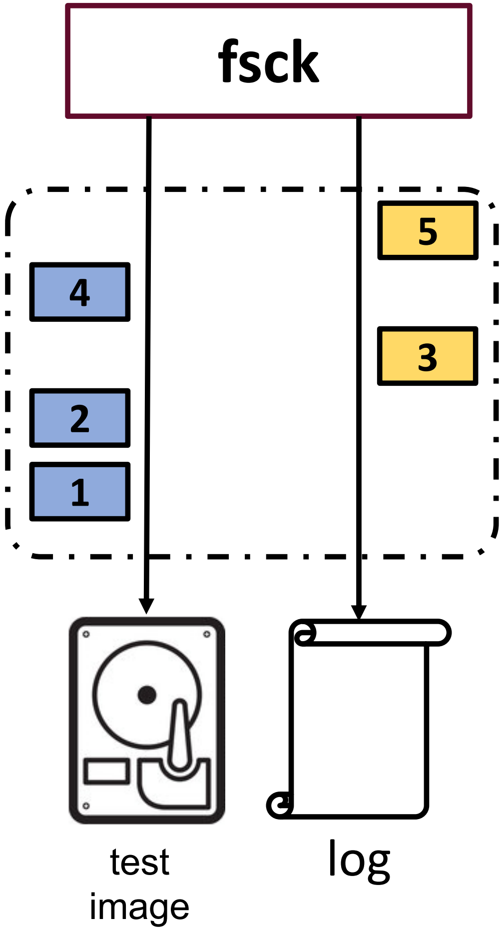
Fault Injection Engine: `rfsck-test`

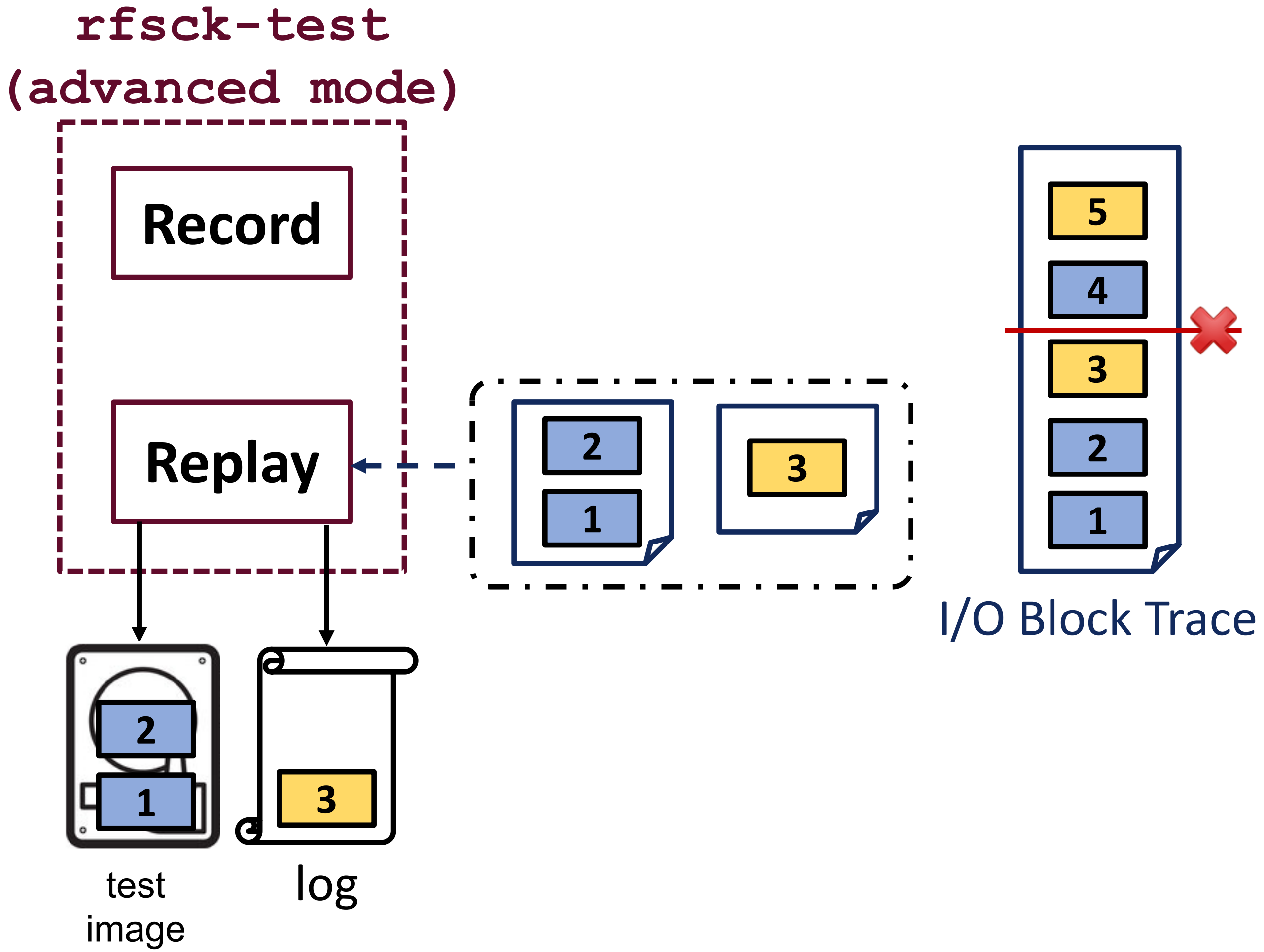
Advanced Mode



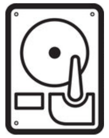
Fault Injection Engine: `rfsck-test`

Advanced Mode

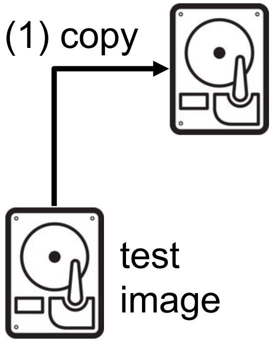




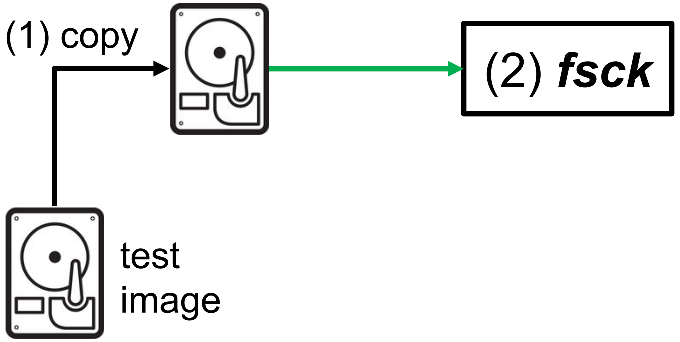
Overall Workflow


test
image

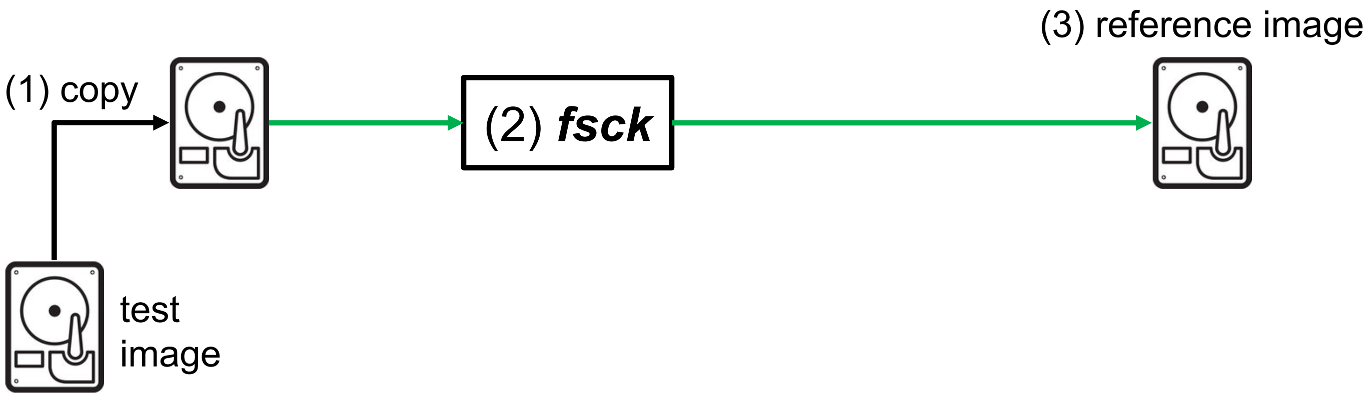
Overall Workflow



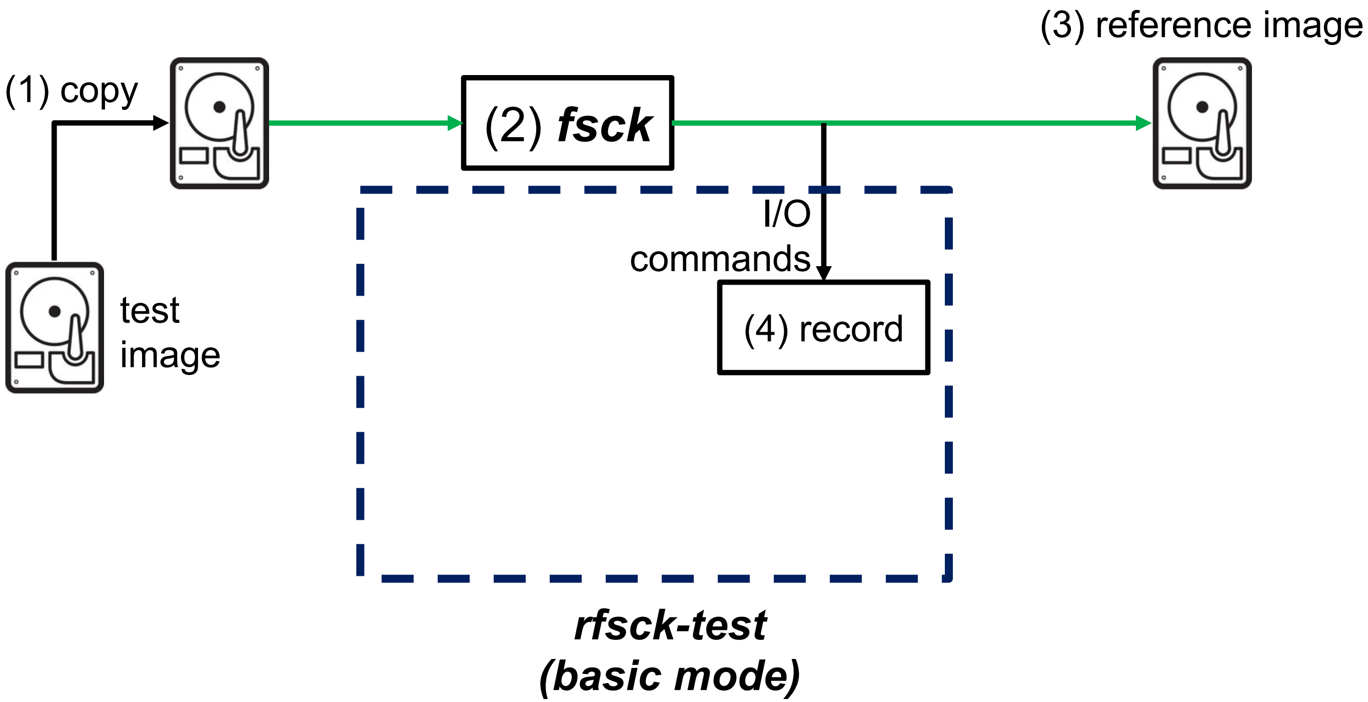
Overall Workflow



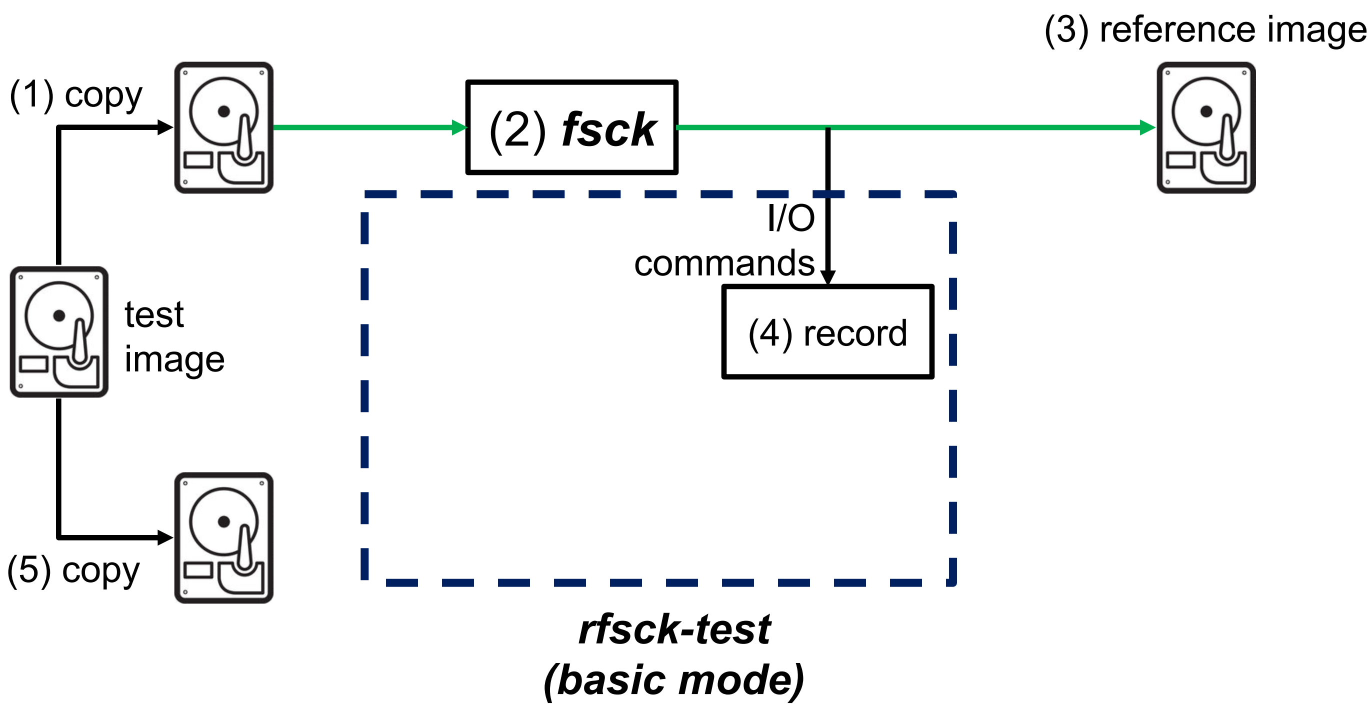
Overall Workflow



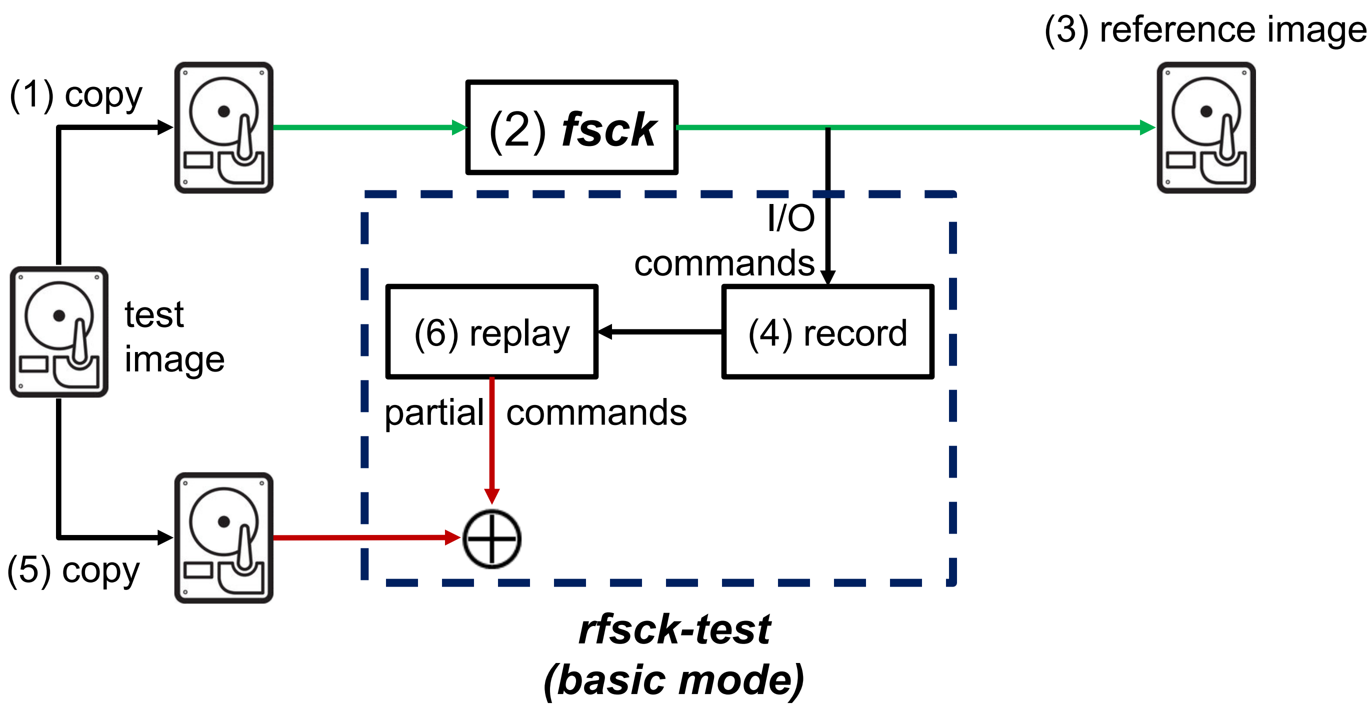
Overall Workflow



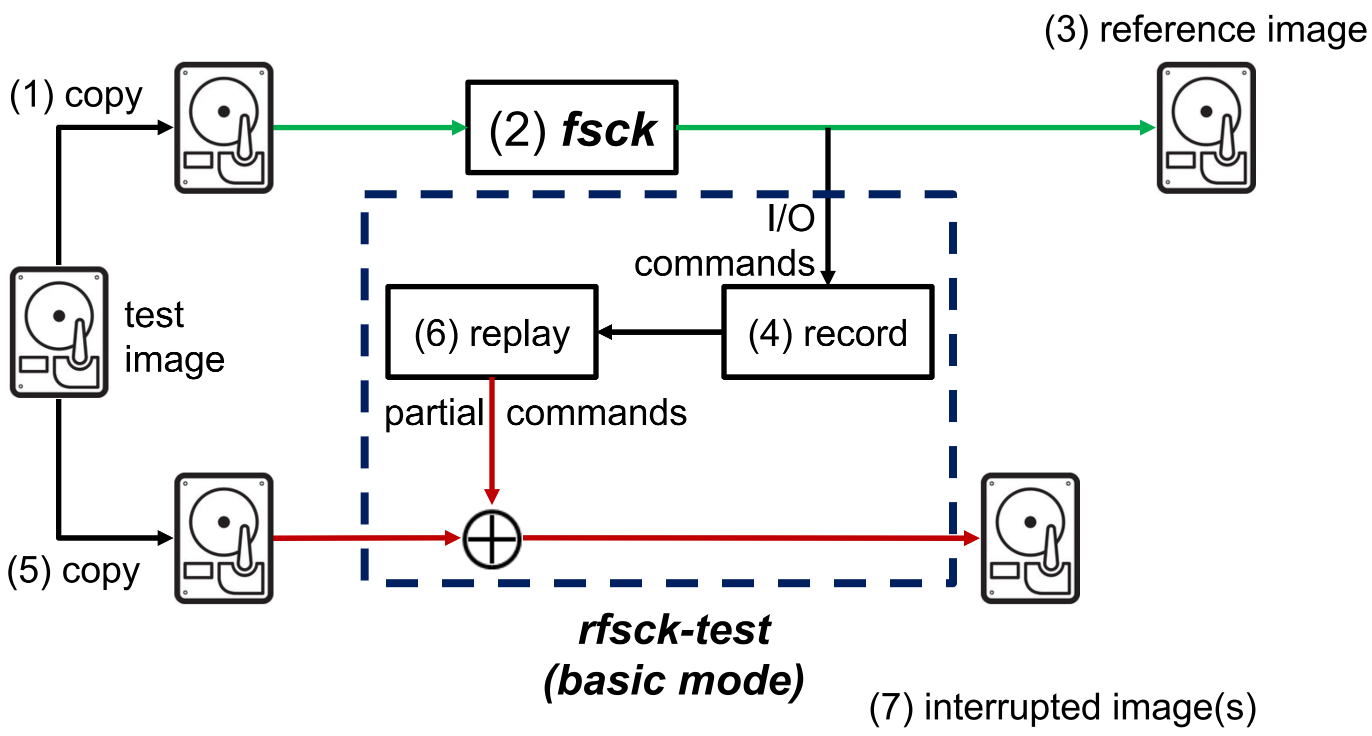
Overall Workflow



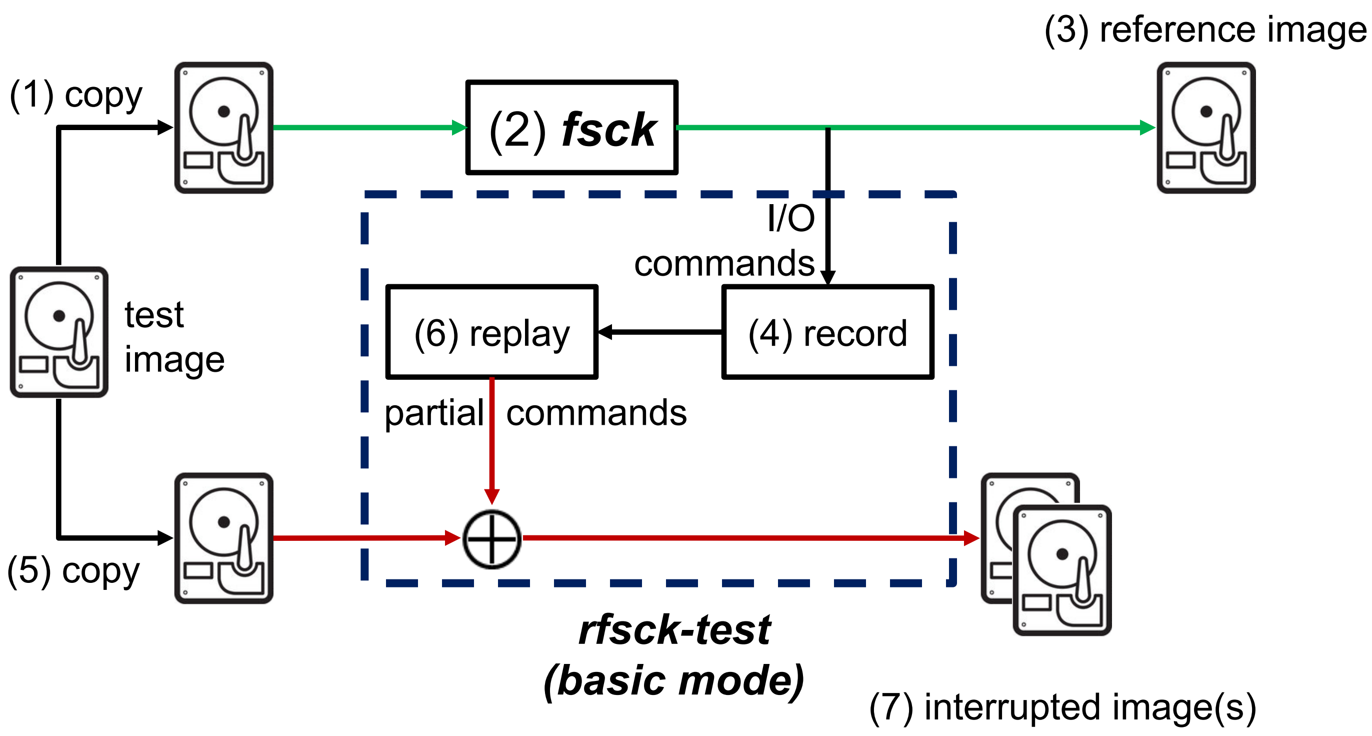
Overall Workflow



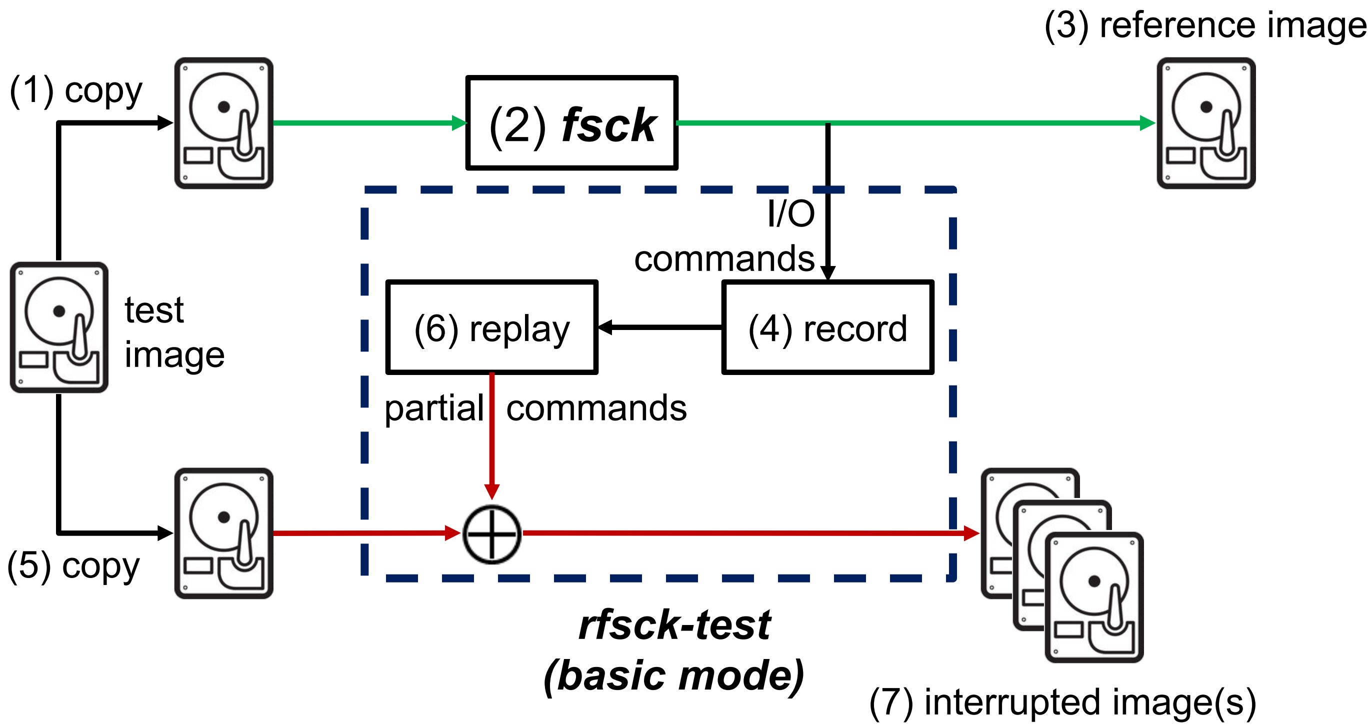
Overall Workflow



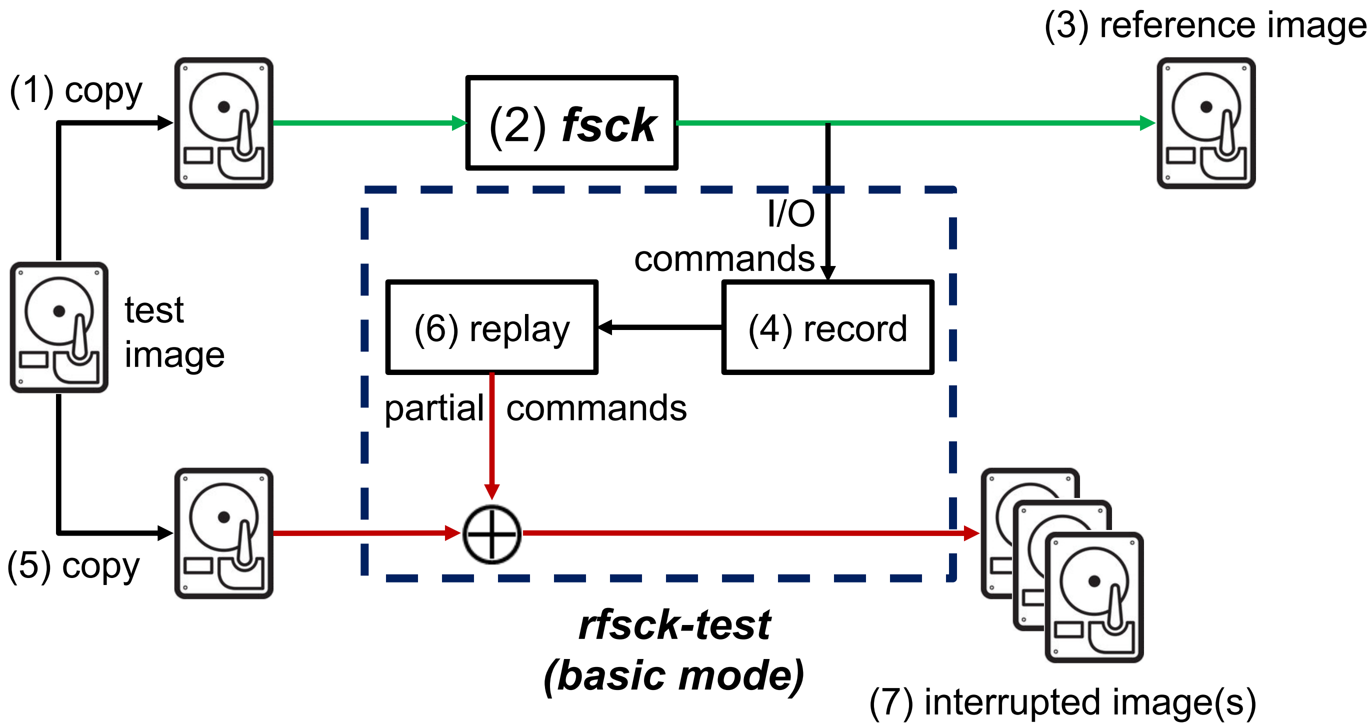
Overall Workflow



Overall Workflow



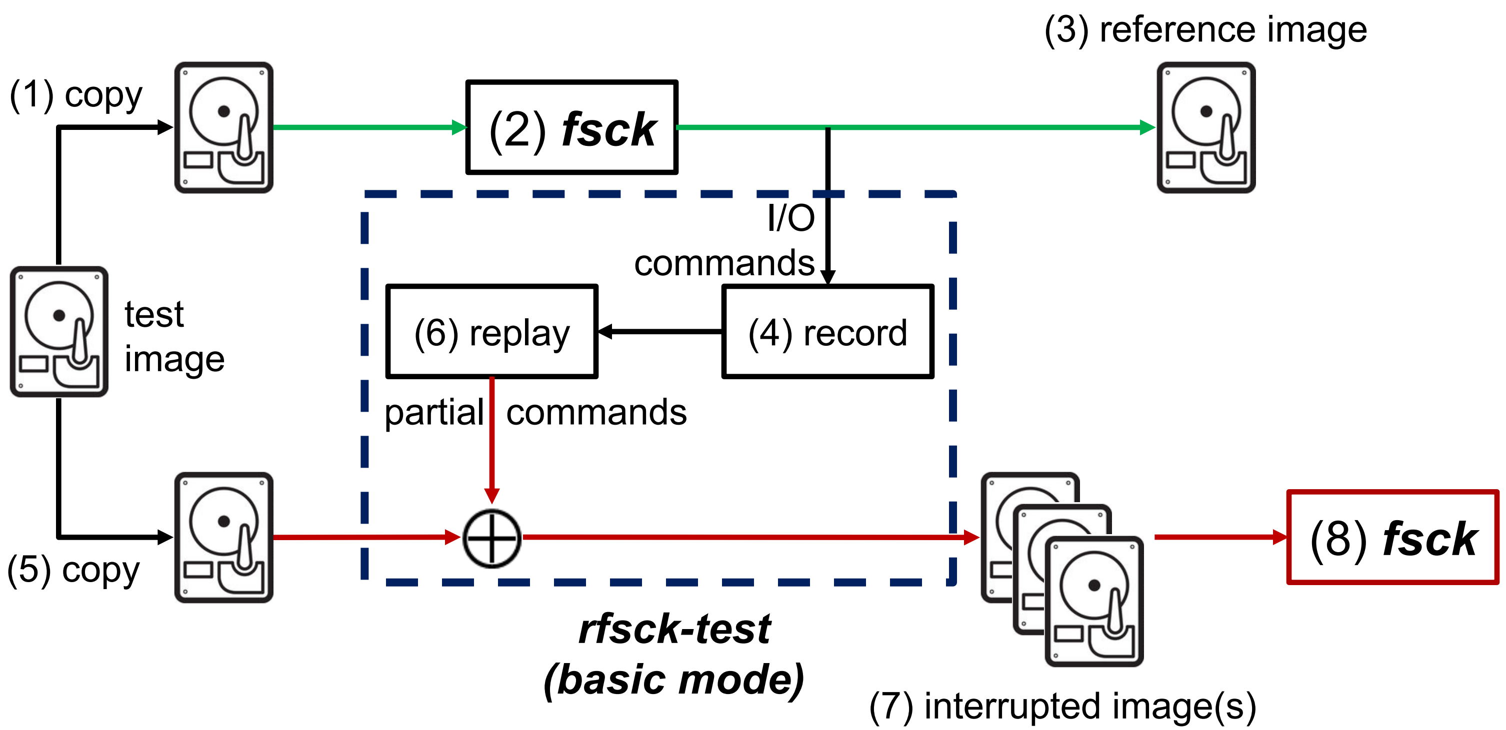
Overall Workflow



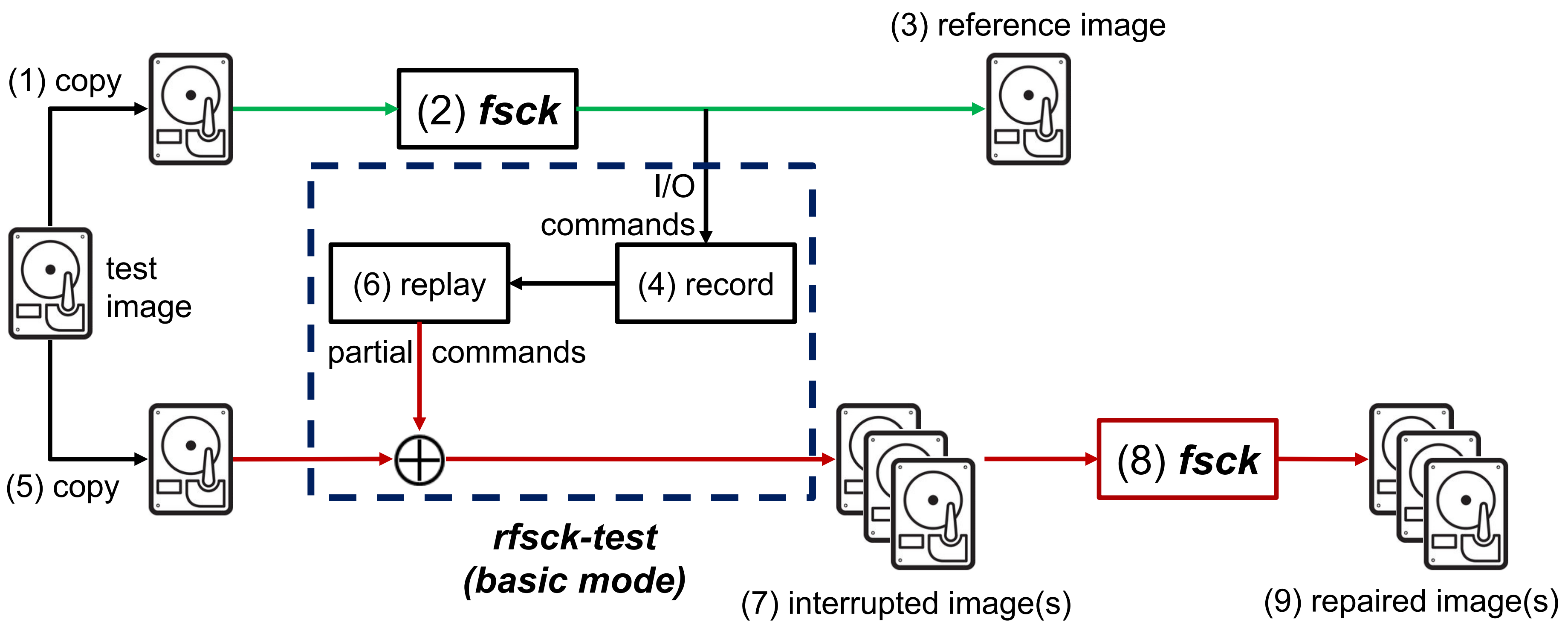
1 test image ➡ many interrupted images

Exhaust all possible fault points during one execution of checker

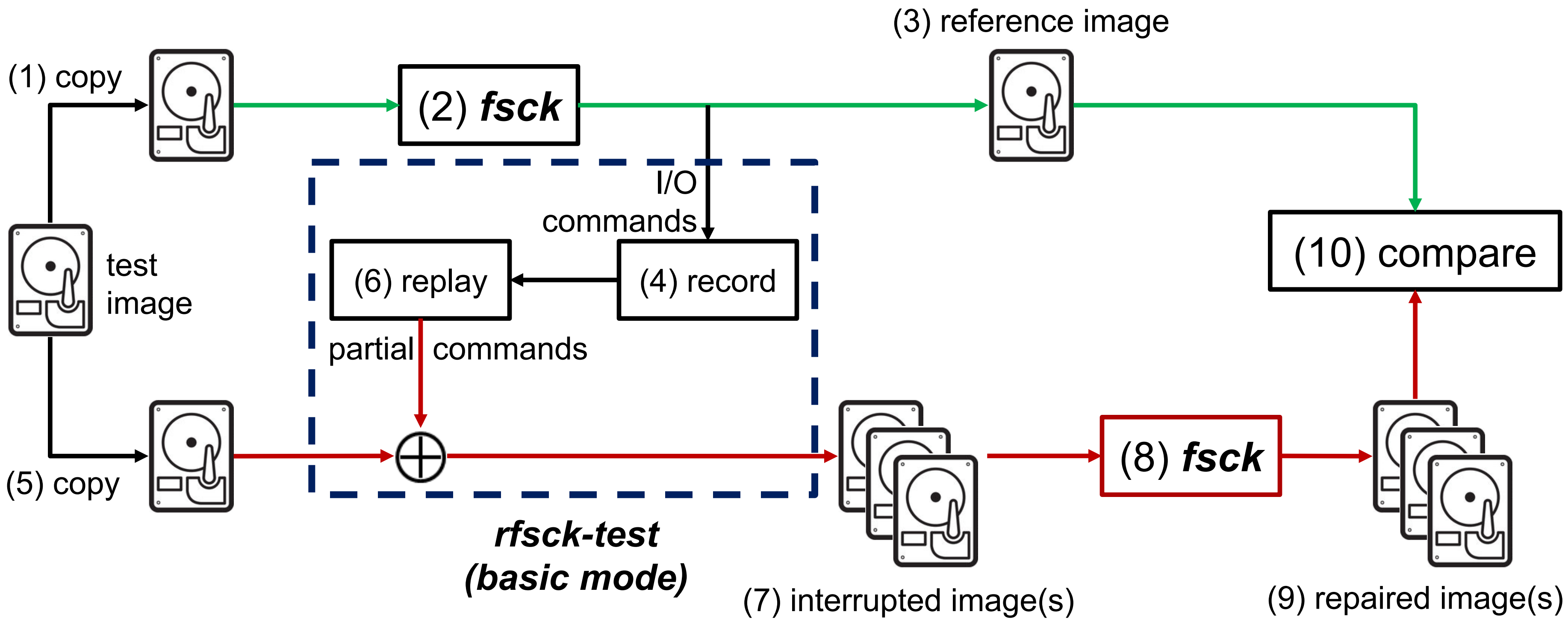
Overall Workflow



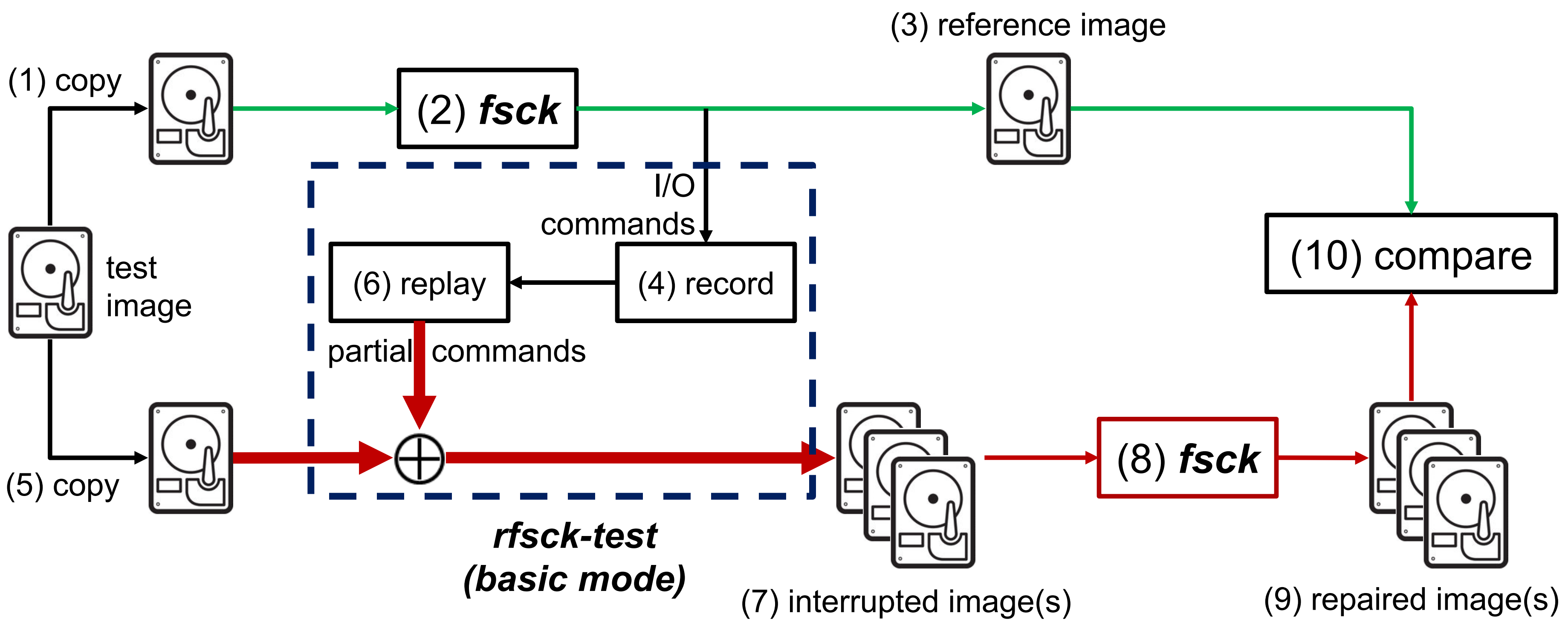
Overall Workflow



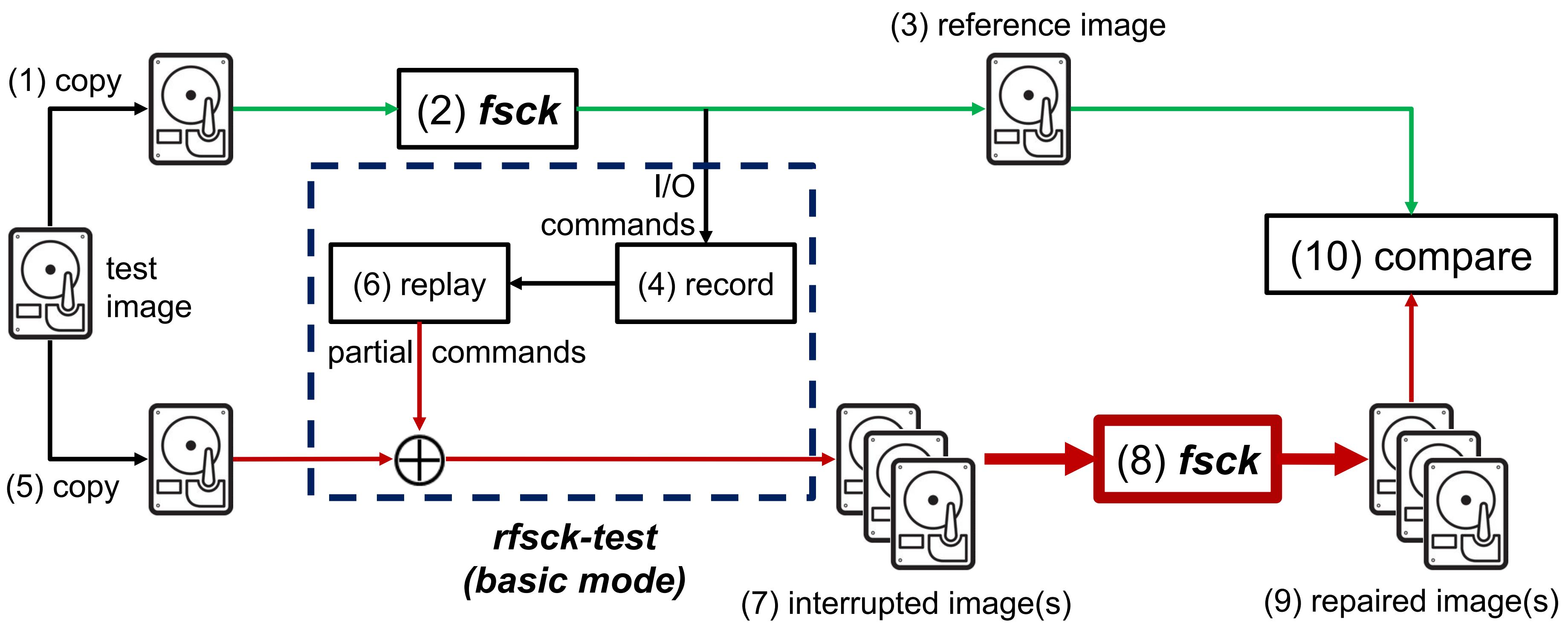
Overall Workflow



Overall Workflow



Overall Workflow



Testing existing checkers

3 case studies performed:

- **`e2fsck`**: checker for EXT 2/3/4 file systems
- **`e2fsck-undo`**: `e2fsck` with logging support
- **`xfs_repair`**: checker for XFS file system

Testing existing checkers

3 case studies performed:

- **`e2fsck`**: checker for EXT 2/3/4 file systems
- **`e2fsck-undo`**: `e2fsck` with logging support
- **`xfs_repair`**: checker for XFS file system

Overall, 4 types of corruptions observed:

Testing existing checkers

3 case studies performed:

- **e2fsck**: checker for EXT 2/3/4 file systems
- **e2fsck-undo**: e2fsck with logging support
- **xfs_repair**: checker for XFS file system

Overall, 4 types of corruptions observed:

Un-mountable

Testing existing checkers

3 case studies performed:

- **e2fsck**: checker for EXT 2/3/4 file systems
- **e2fsck-undo**: e2fsck with logging support
- **xfs_repair**: checker for XFS file system

Overall, 4 types of corruptions observed:

Un-mountable

File Content Corruption

Testing existing checkers

3 case studies performed:

- **`e2fsck`**: checker for EXT 2/3/4 file systems
- **`e2fsck-undo`**: `e2fsck` with logging support
- **`xfs_repair`**: checker for XFS file system

Overall, 4 types of corruptions observed:

Un-mountable

File Content Corruption

Misplacement of Files

Testing existing checkers

3 case studies performed:

- **`e2fsck`**: checker for EXT 2/3/4 file systems
- **`e2fsck-undo`**: `e2fsck` with logging support
- **`xfs_repair`**: checker for XFS file system

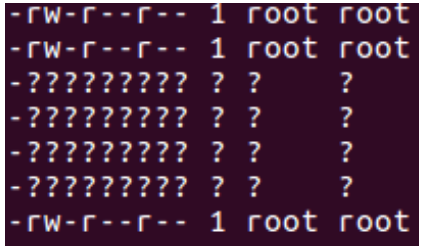
Overall, 4 types of corruptions observed:

Un-mountable

File Content Corruption

Misplacement of Files

Others



Testing existing checkers

3 case studies performed:

- **`e2fsck`**: checker for EXT 2/3/4 file systems
- **`e2fsck-undo`**: `e2fsck` with logging support
- **`xfs_repair`**: checker for XFS file system

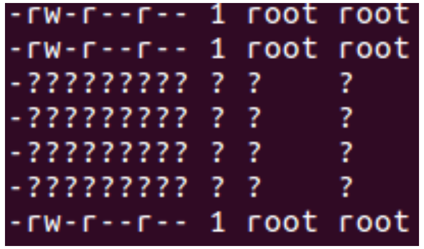
Overall, 4 types of corruptions observed:

Un-mountable

File Content Corruption

Misplacement of Files

Others



Cannot be fixed by another run of fsck

Case Study: `e2fsck`

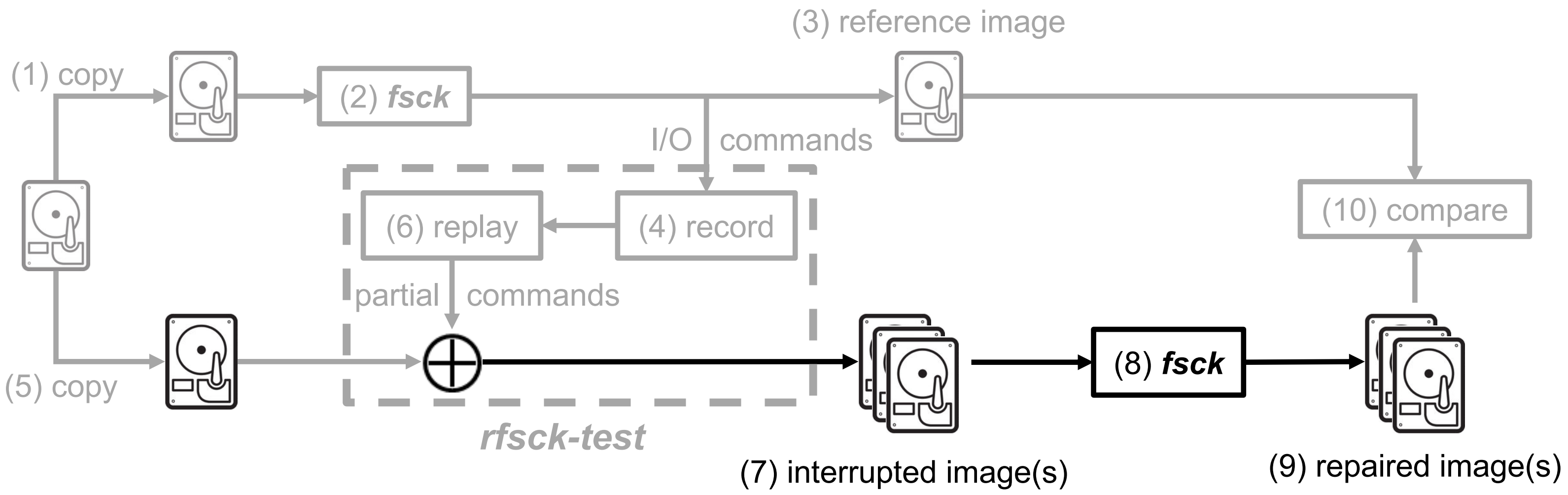
Used 175 test images from `e2fsprogs`

Block size of all images is 1KB

Fault injected at two granularities: 512B and 4KB

Case Study: `e2fsck`

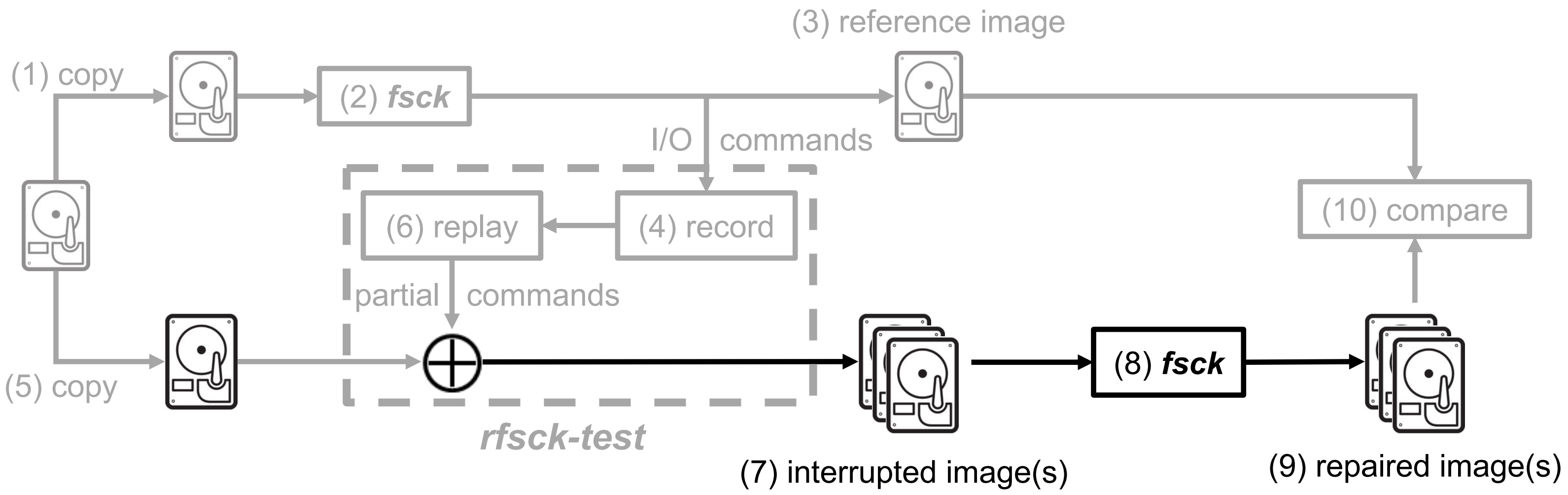
RECAP



1 test image ➡ many interrupted/repaired images

Case Study: `e2fsck`

RECAP



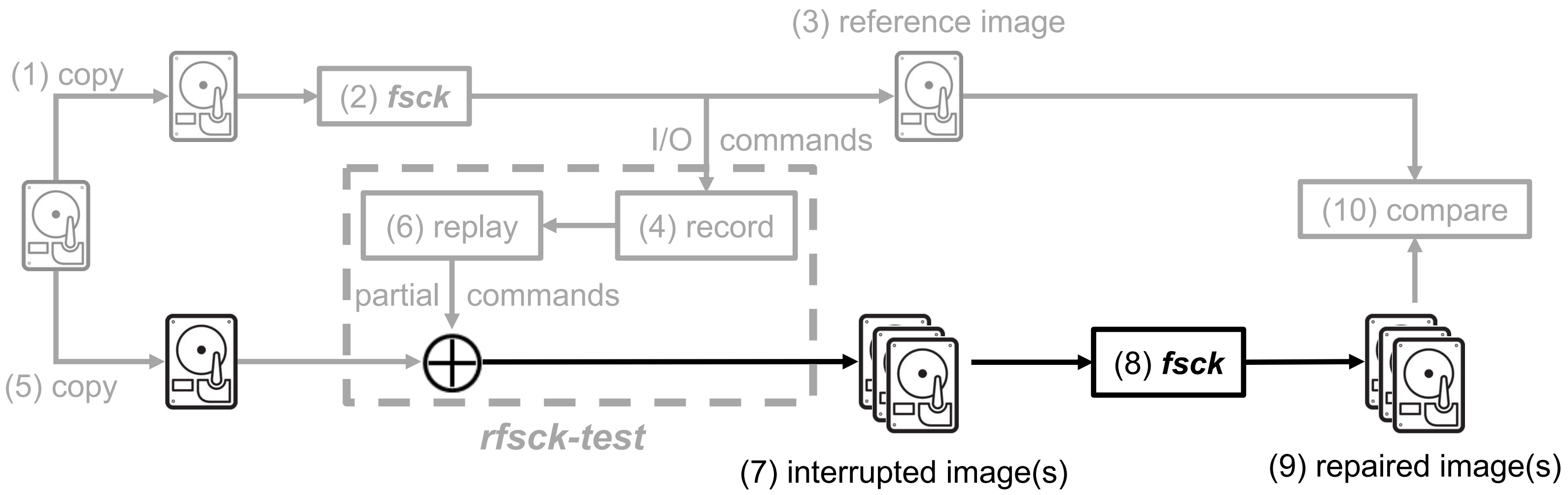
1 test image ➡ many interrupted/repaired images

Fault injection granularity	# of EXT4 test images	# of repaired images generated	# of images reporting corruption	
			test images	repaired images
512 B	175	25,062	34	240
4 KB	175	3,192	17	37

Table 1: Number of test images and repaired images reporting corruption

Case Study: `e2fsck`

RECAP



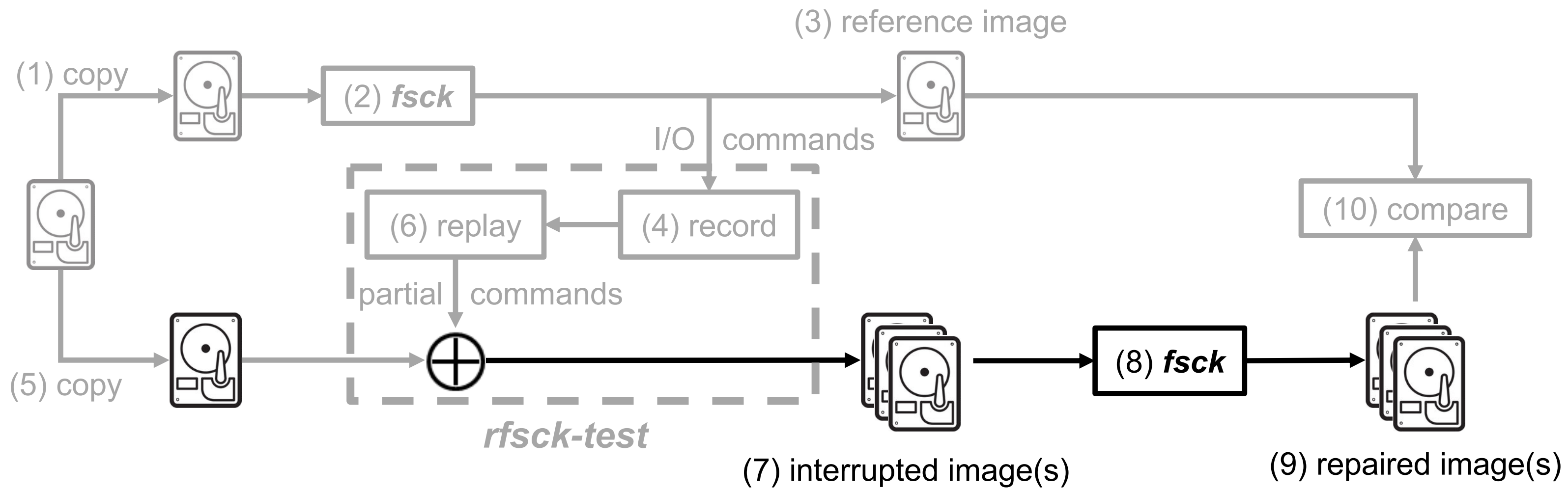
1 test image ➡ many interrupted/repaired images

Fault injection granularity	# of EXT4 test images	# of repaired images generated	# of images reporting corruption	
			test images	repaired images
512 B	175	25,062	34	240
4 KB	175	3,192	17	37

Table 1: Number of test images and repaired images reporting corruption

Case Study: e2fsck

RECAP



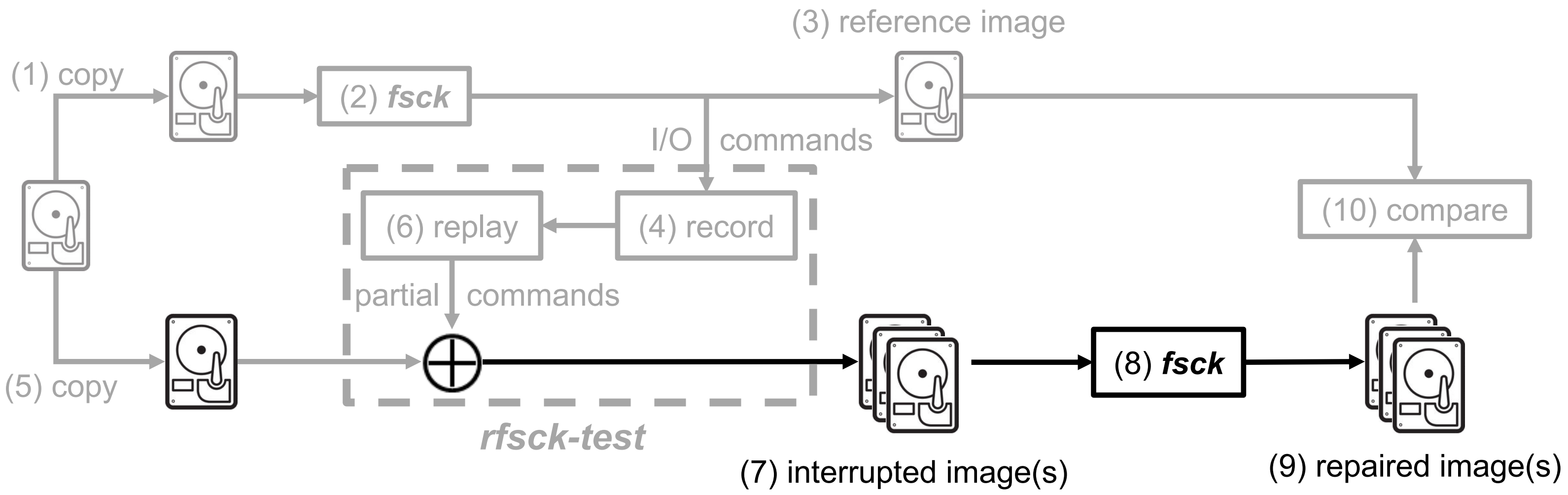
1 test image ➡ many interrupted/repaired images

Fault injection granularity	# of EXT4 test images	# of repaired images generated	# of images reporting corruption	
			test images	repaired images
512 B	175	25,062	34	240
4 KB	175	3,192	17	37

Table 1: Number of test images and repaired images reporting corruption

Case Study: `e2fsck`

RECAP



1 test image ➡ many interrupted/repaired images

Fault injection granularity	# of EXT4 test images	# of repaired images generated	# of images reporting corruption	
			test images	repaired images
512 B	175	25,062	34	240
4 KB	175	3,192	17	37

Table 1: Number of test images and repaired images reporting corruption

Case Study: `e2fsck`

Corruption type	test images		repaired images	
	512 B	4 KB	512 B	4 KB
cannot mount	20	1	41	3
data corruption	9	5	107	10
misplacement	9	11	82	23
others	1	1	10	1

Table 2: Classification of corruptions observed on test and repaired images

Case Study: `e2fsck`

Corruption type	test images		repaired images	
	512 B	4 KB	512 B	4 KB
cannot mount	20	1	41	3
data corruption	9	5	107	10
misplacement	9	11	82	23
others	1	1	10	1

Table 2: Classification of corruptions observed on test and repaired images

Smaller fault injection granularity, more corruption scenarios

Case Study: `e2fsck-undo`

Undo log feature in `e2fsprogs` utilities

E.g.: e2fsck, debugfs, mke2fs, etc.

Case Study: `e2fsck-undo`

Undo log feature in `e2fsprogs` utilities

E.g.: e2fsck, debugfs, mke2fs, etc.

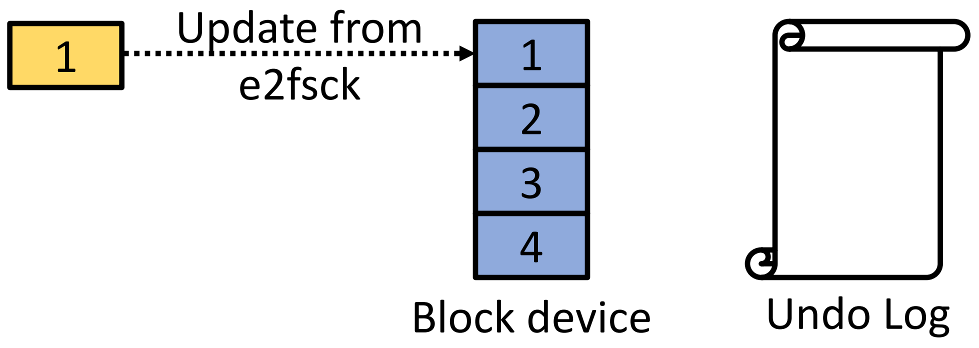
Records data block that is being updated into a log

- undo the changes made (if necessary)

Case Study: `e2fsck-undo`

Undo log feature in `e2fsprogs` utilities
E.g.: e2fsck, debugfs, mke2fs, etc.

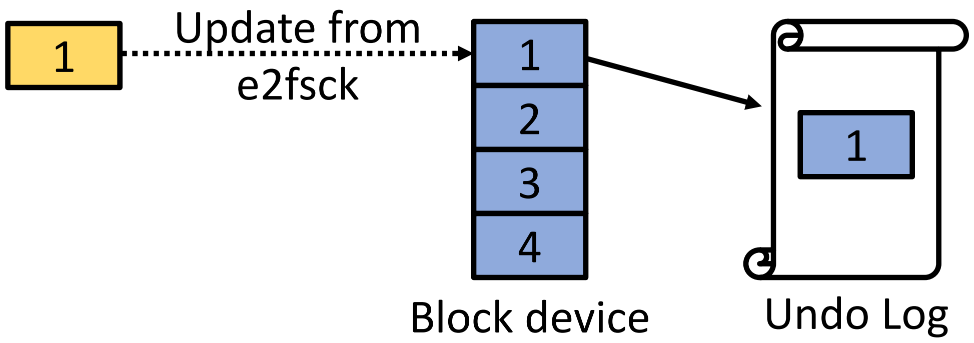
Records data block that is being updated into a log
- undo the changes made (if necessary)



Case Study: `e2fsck-undo`

Undo log feature in `e2fsprogs` utilities
E.g.: e2fsck, debugfs, mke2fs, etc.

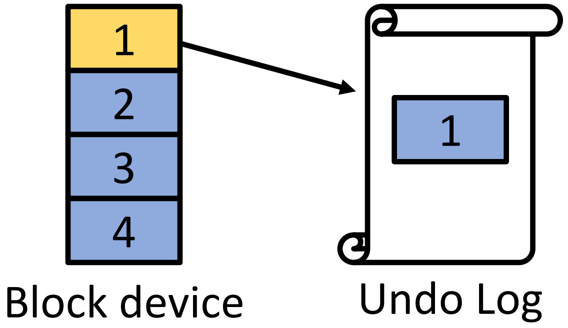
Records data block that is being updated into a log
- undo the changes made (if necessary)



Case Study: `e2fsck-undo`

Undo log feature in `e2fsprogs` utilities
E.g.: e2fsck, debugfs, mke2fs, etc.

Records data block that is being updated into a log
- undo the changes made (if necessary)



Case Study: `e2fsck-undo`

Fault Injection Granularities	Number of images reporting corruption	
	`e2fsck`	`e2fsck-undo`
512 B	34	**34**
4 KB	17	**15**

Table 3: Number of test images reporting corruption under `e2fsck` and `e2fsck-undo`

Undo log fails

Outline

- Motivation
- Background & Related Work
- Research Question
- Are existing checkers resilient to faults?
- How to build robust checkers?
- Evaluation
- Conclusion

Why does `e2fsck-undo` Fail?

Undo log is a Write-ahead log (WAL)

In WAL, it is expected that the log block reaches persistent storage before the updated blocks reaches its storage

Why does `e2fsck-undo` Fail?

Undo log is a Write-ahead log (WAL)

In WAL, it is expected that the log block reaches persistent storage before the updated blocks reaches its storage

Undo log does not enforce such ordering

Why does `e2fsck-undo` Fail?

```
/*open undo log*/
undo_open(...){
    open(...); /*no O_SYNC*/
}
...
/*fix 1st inconsistency*/
undo_write_blk64(...){
    /*write to undo log asynchronously*/
    undo_write_tdb(...){
        ...
        pwrite(...); /*no fsync()*/
    }
    /*write to fs image asynchronously*/
    io_channel_write_blk64(...){...}
}
/*fix 2nd, 3rd, ... inconsistencies*/
...

/*sync buffered writes to fs image*/
ext2fs_flush(...){...}
/*close undo log*/
undo_close(...){...}
```

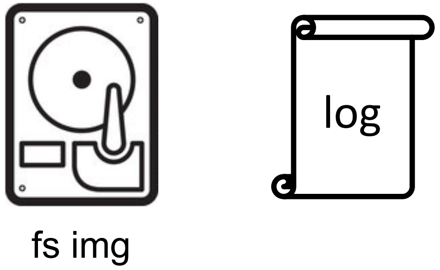
Why does `e2fsck-undo` Fail?

```
/*open undo log*/
undo_open(...){
     open(...); /*no O_SYNC*/
}
...
/*fix 1st inconsistency*/
undo_write_blk64(...){
     /*write to undo log asynchronously*/
     undo_write_tdb(...){
          ...
          pwrite(...); /*no fsync()*/
     }
     /*write to fs image asynchronously*/
     io_channel_write_blk64(...){...}
}
/*fix 2nd, 3rd, ... inconsistencies*/
...

/*sync buffered writes to fs image*/
ext2fs_flush(...){...}
/*close undo log*/
undo_close(...){...}
```

block written to the log

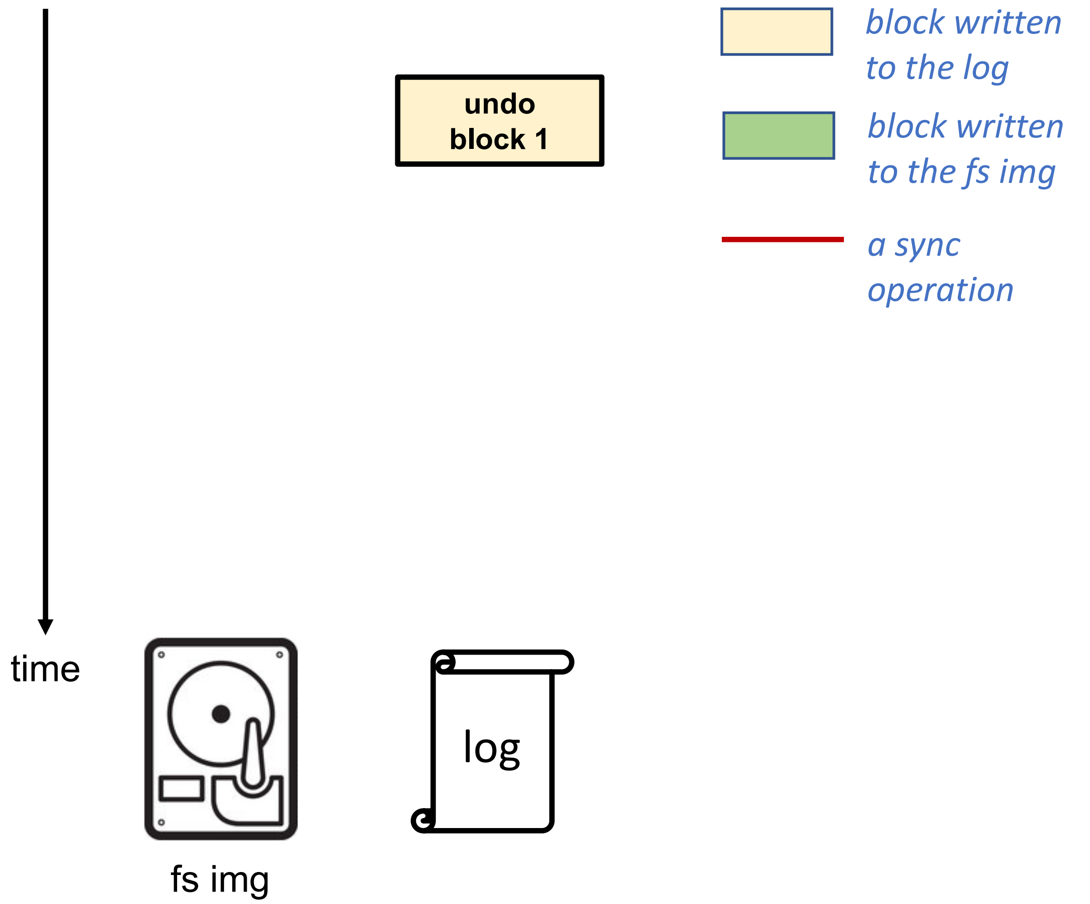
block written to the fs img

a sync operation



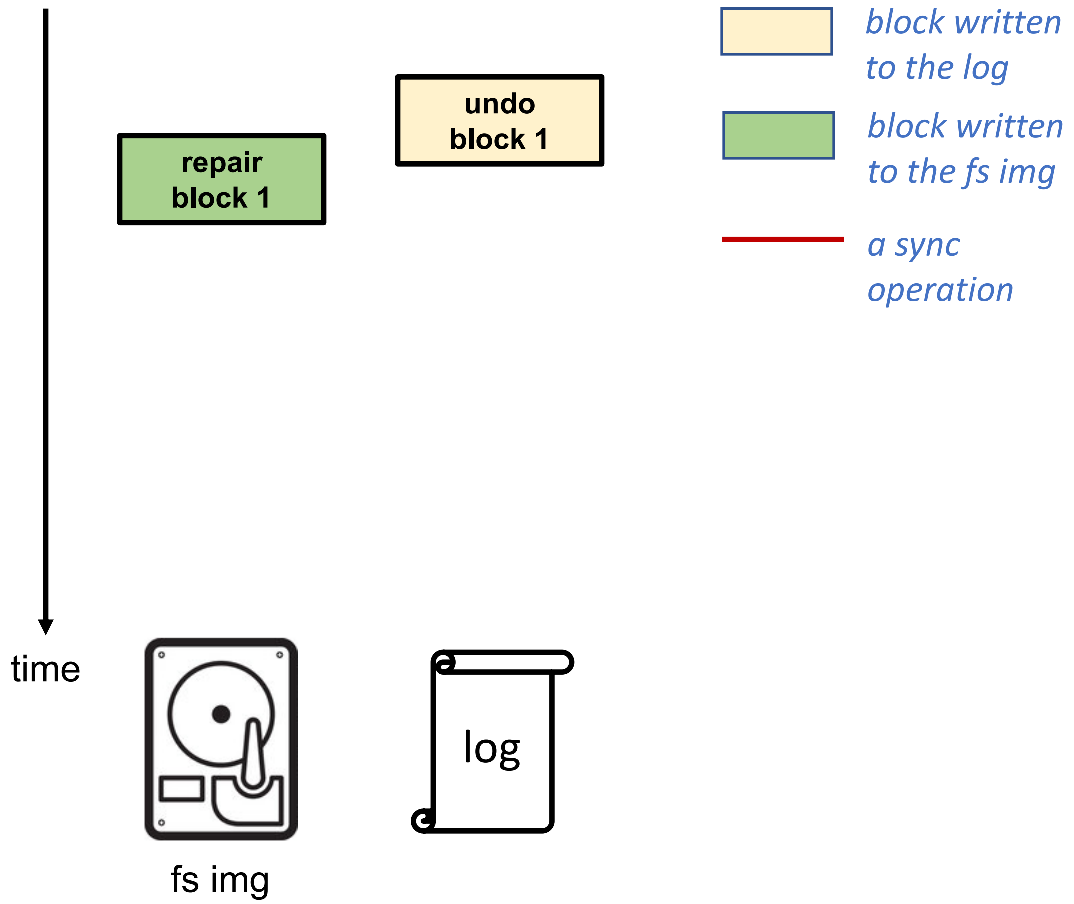
Why does `e2fsck-undo` Fail?

```
/*open undo log*/
undo_open(...){
     open(...); /*no O_SYNC*/
}
...
/*fix 1st inconsistency*/
undo_write_blk64(...){
     /*write to undo log asynchronously*/
     undo_write_tdb(...){
          ...
          pwrite(...); /*no fsync()*/
     }
     /*write to fs image asynchronously*/
     io_channel_write_blk64(...){...}
}
/*fix 2nd, 3rd, ... inconsistencies*/
...

/*sync buffered writes to fs image*/
ext2fs_flush(...){...}
/*close undo log*/
undo_close(...){...}
```



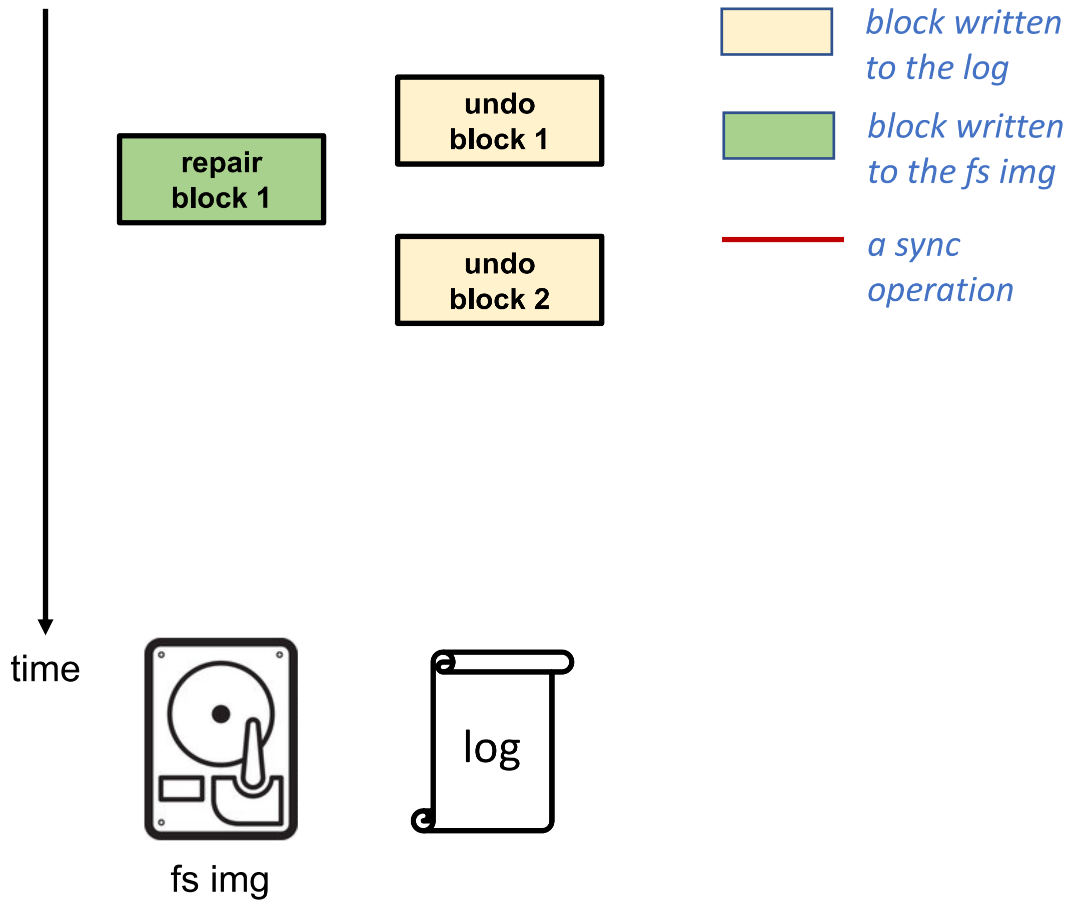
Why does `e2fsck-undo` Fail?

```
/*open undo log*/
undo_open(...){
    open(...); /*no O_SYNC*/
}
...
/*fix 1st inconsistency*/
undo_write_blk64(...){
    /*write to undo log asynchronously*/
    undo_write_tdb(...){
        ...
        pwrite(...); /*no fsync()*/
    }
    /*write to fs image asynchronously*/
    io_channel_write_blk64(...){...}
}
/*fix 2nd, 3rd, ... inconsistencies*/
...

/*sync buffered writes to fs image*/
ext2fs_flush(...){...}
/*close undo log*/
undo_close(...){...}
```



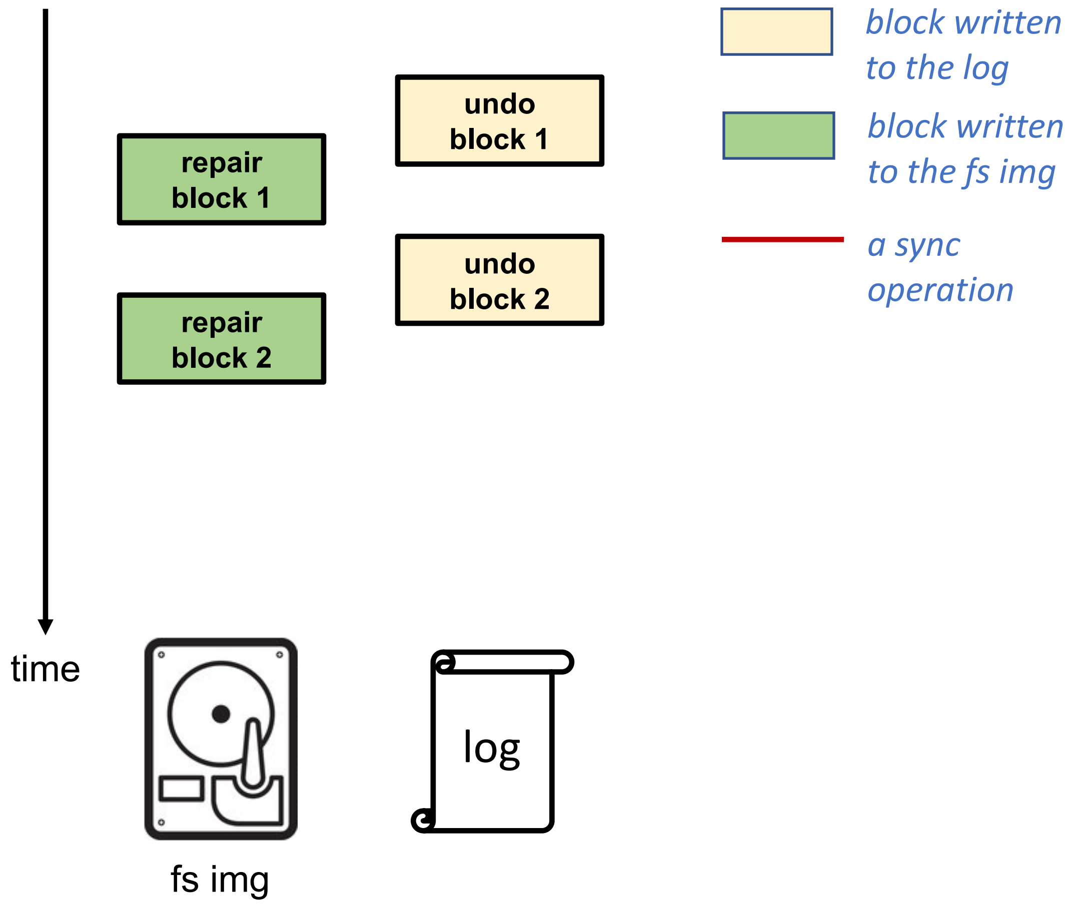
Why does `e2fsck-undo` Fail?

```
/*open undo log*/
undo_open(...){
    open(...); /*no O_SYNC*/
}
...
/*fix 1st inconsistency*/
undo_write_blk64(...){
    /*write to undo log asynchronously*/
    undo_write_tdb(...){
        ...
        pwrite(...); /*no fsync()*/
    }
    /*write to fs image asynchronously*/
    io_channel_write_blk64(...){...}
}
/*fix 2nd, 3rd, ... inconsistencies*/
...

/*sync buffered writes to fs image*/
ext2fs_flush(...){...}
/*close undo log*/
undo_close(...){...}
```



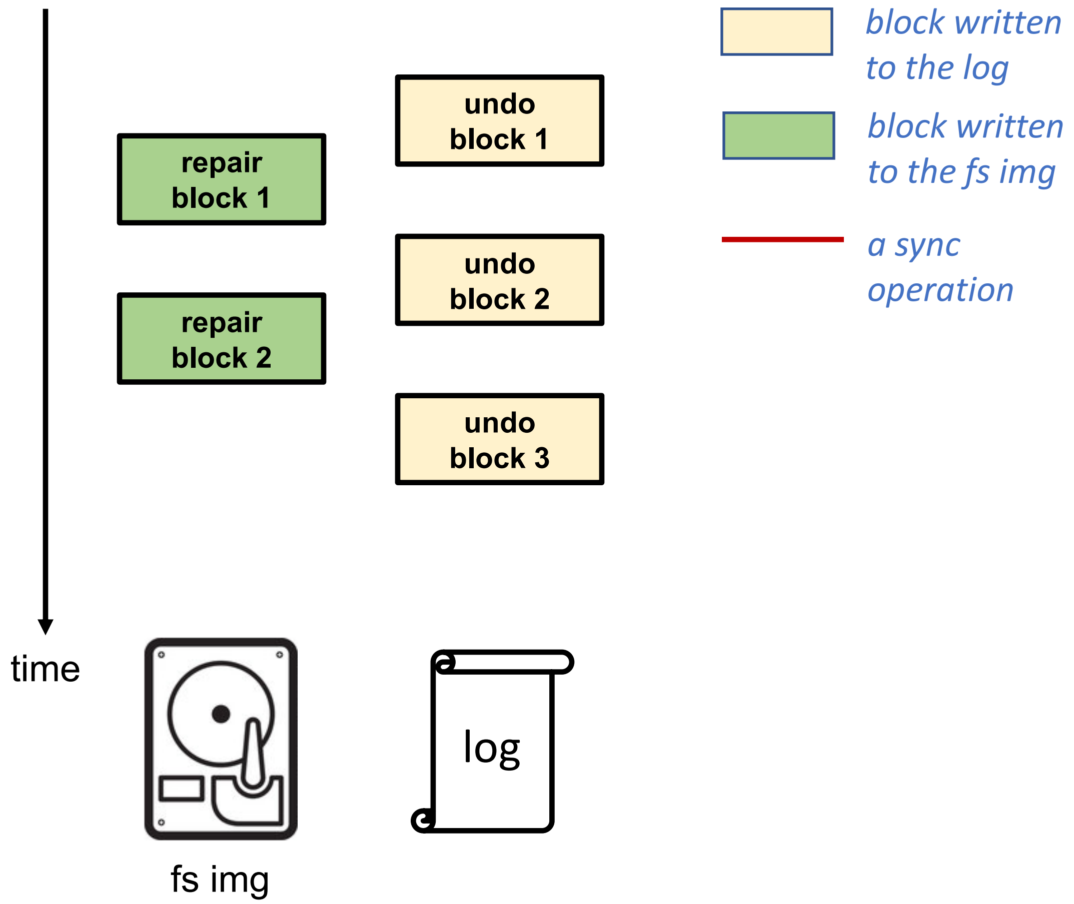
Why does `e2fsck-undo` Fail?

```
/*open undo log*/
undo_open(...){
     open(...); /*no O_SYNC*/
}
...
/*fix 1st inconsistency*/
undo_write_blk64(...){
     /*write to undo log asynchronously*/
     undo_write_tdb(...){
          ...
          pwrite(...); /*no fsync()*/
     }
     /*write to fs image asynchronously*/
     io_channel_write_blk64(...){...}
}
/*fix 2nd, 3rd, ... inconsistencies*/
...

/*sync buffered writes to fs image*/
ext2fs_flush(...){...}
/*close undo log*/
undo_close(...){...}
```



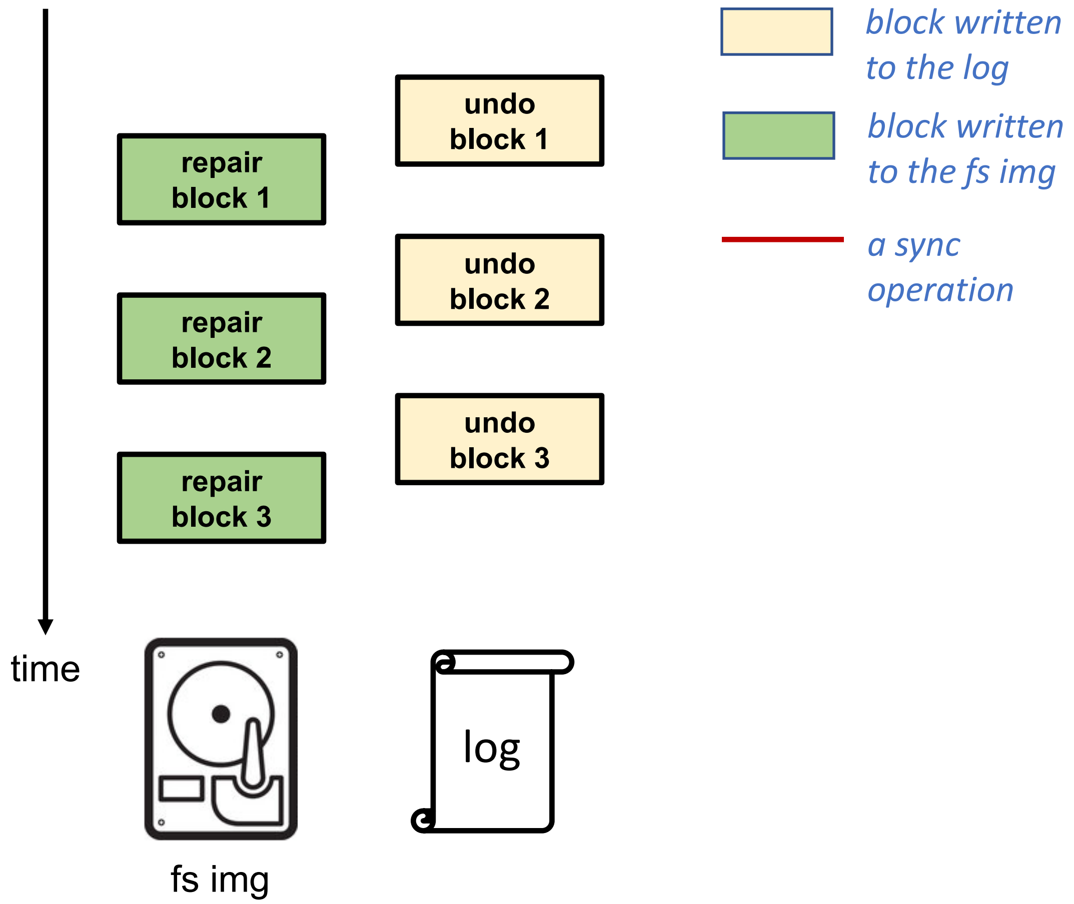
Why does `e2fsck-undo` Fail?

```
/*open undo log*/
undo_open(...){
    open(...); /*no O_SYNC*/
}
...
/*fix 1st inconsistency*/
undo_write_blk64(...){
    /*write to undo log asynchronously*/
    undo_write_tdb(...){
        ...
        pwrite(...); /*no fsync()*/
    }
    /*write to fs image asynchronously*/
    io_channel_write_blk64(...){...}
}
/*fix 2nd, 3rd, ... inconsistencies*/
...

/*sync buffered writes to fs image*/
ext2fs_flush(...){...}
/*close undo log*/
undo_close(...){...}
```



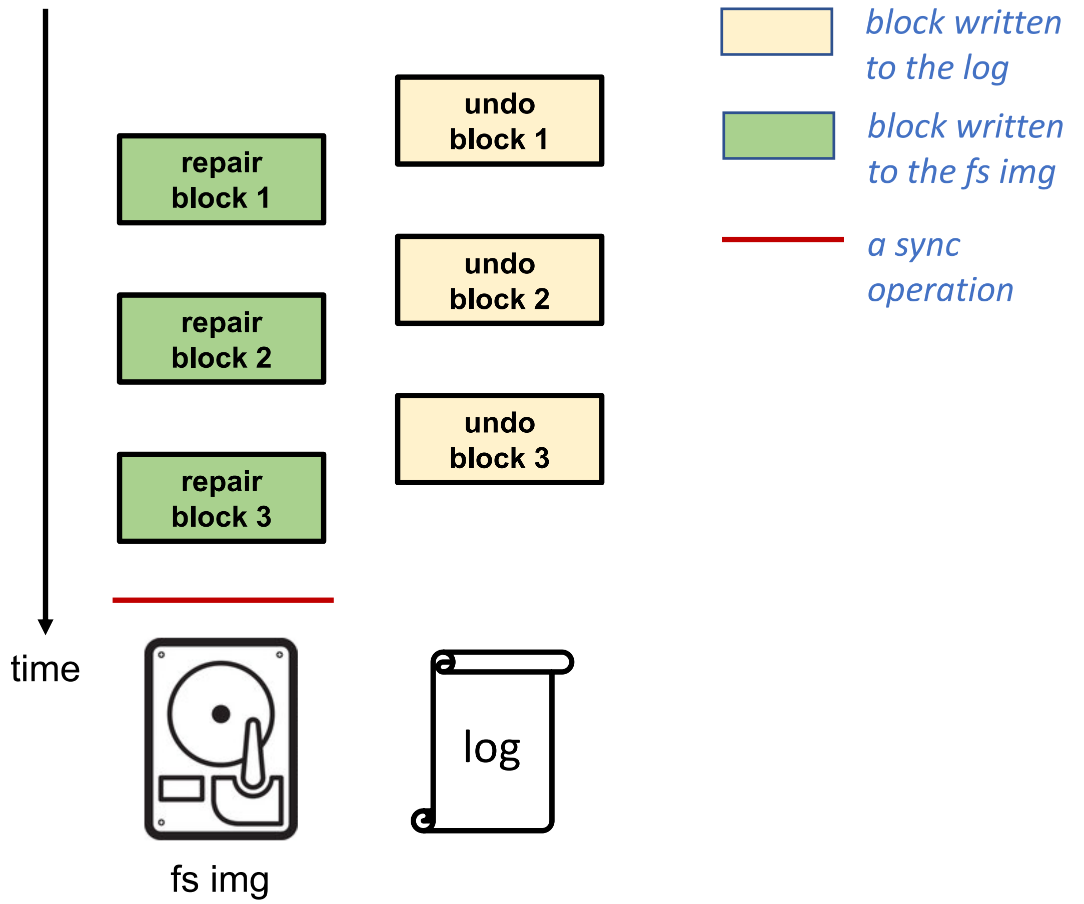
Why does `e2fsck-undo` Fail?

```
/*open undo log*/
undo_open(...){
    open(...); /*no O_SYNC*/
}
...
/*fix 1st inconsistency*/
undo_write_blk64(...){
    /*write to undo log asynchronously*/
    undo_write_tdb(...){
        ...
        pwrite(...); /*no fsync()*/
    }
    /*write to fs image asynchronously*/
    io_channel_write_blk64(...){...}
}
/*fix 2nd, 3rd, ... inconsistencies*/
...

/*sync buffered writes to fs image*/
ext2fs_flush(...){...}
/*close undo log*/
undo_close(...){...}
```



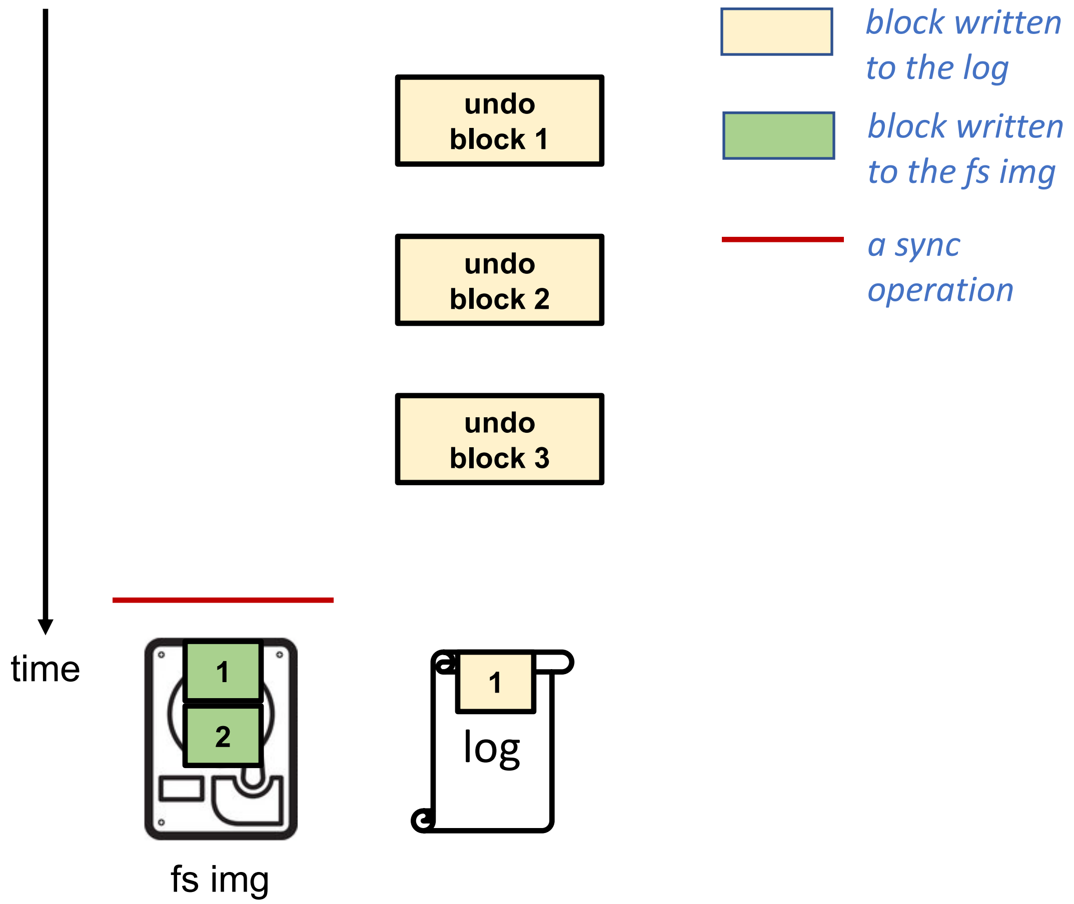
Why does e2fsck-undo Fail?

```
/*open undo log*/
undo_open(...){
    open(...); /*no O_SYNC*/
}
...
/*fix 1st inconsistency*/
undo_write_blk64(...){
    /*write to undo log asynchronously*/
    undo_write_tdb(...){
        ...
        pwrite(...); /*no fsync()*/
    }
    /*write to fs image asynchronously*/
    io_channel_write_blk64(...){...}
}
/*fix 2nd, 3rd, ... inconsistencies*/
...

/*sync buffered writes to fs image*/
ext2fs_flush(...){...}
/*close undo log*/
undo_close(...){...}
```



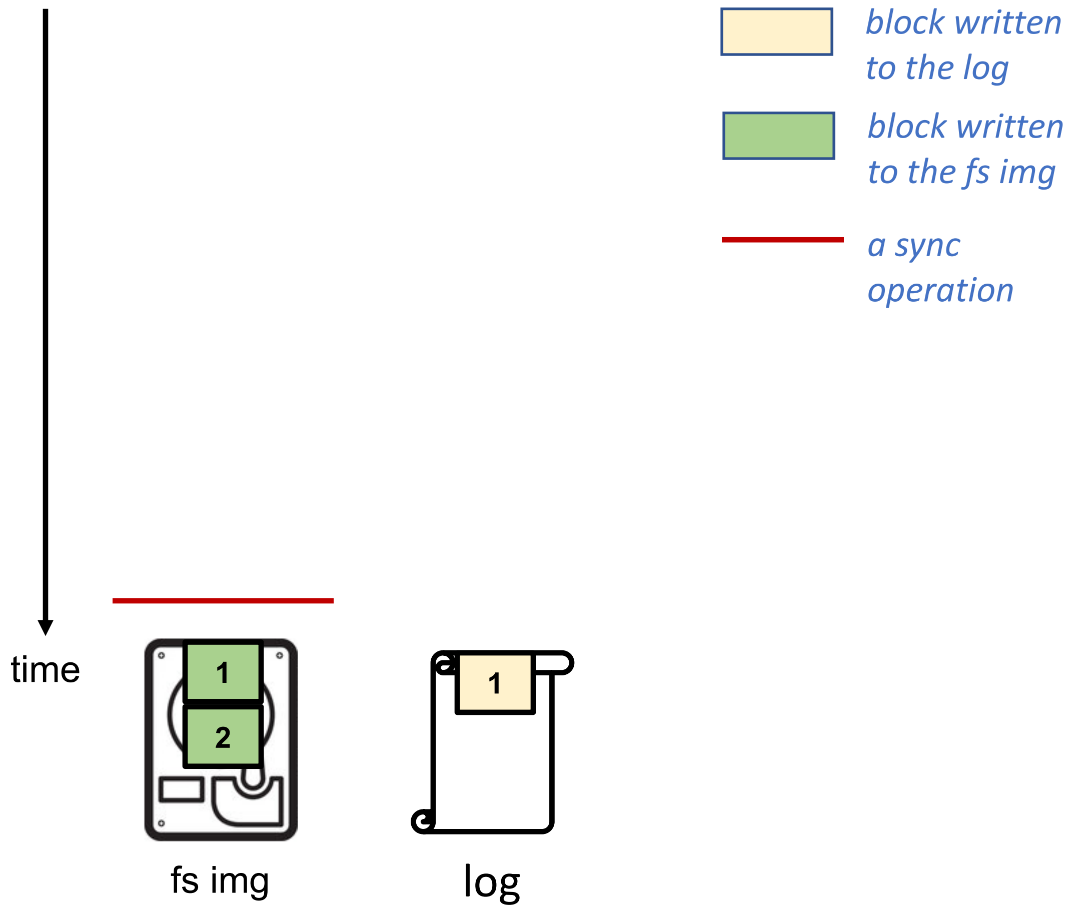
Why does `e2fsck-undo` Fail?

```
/*open undo log*/
undo_open(...){
    open(...); /*no O_SYNC*/
}
...
/*fix 1st inconsistency*/
undo_write_blk64(...){
    /*write to undo log asynchronously*/
    undo_write_tdb(...){
        ...
        pwrite(...); /*no fsync()*/
    }
    /*write to fs image asynchronously*/
    io_channel_write_blk64(...){...}
}
/*fix 2nd, 3rd, ... inconsistencies*/
...

/*sync buffered writes to fs image*/
ext2fs_flush(...){...}
/*close undo log*/
undo_close(...){...}
```



Why does `e2fsck-undo` Fail?

```
/*open undo log*/
undo_open(...){
    open(...); /*no O_SYNC*/
}
...
/*fix 1st inconsistency*/
undo_write_blk64(...){
    /*write to undo log asynchronously*/
    undo_write_tdb(...){
        ...
        pwrite(...); /*no fsync()*/
    }
    /*write to fs image asynchronously*/
    io_channel_write_blk64(...){...}
}
/*fix 2nd, 3rd, ... inconsistencies*/
...

/*sync buffered writes to fs image*/
ext2fs_flush(...){...}
/*close undo log*/
undo_close(...){...}
```



No ordering guarantee

Robust File System Checker

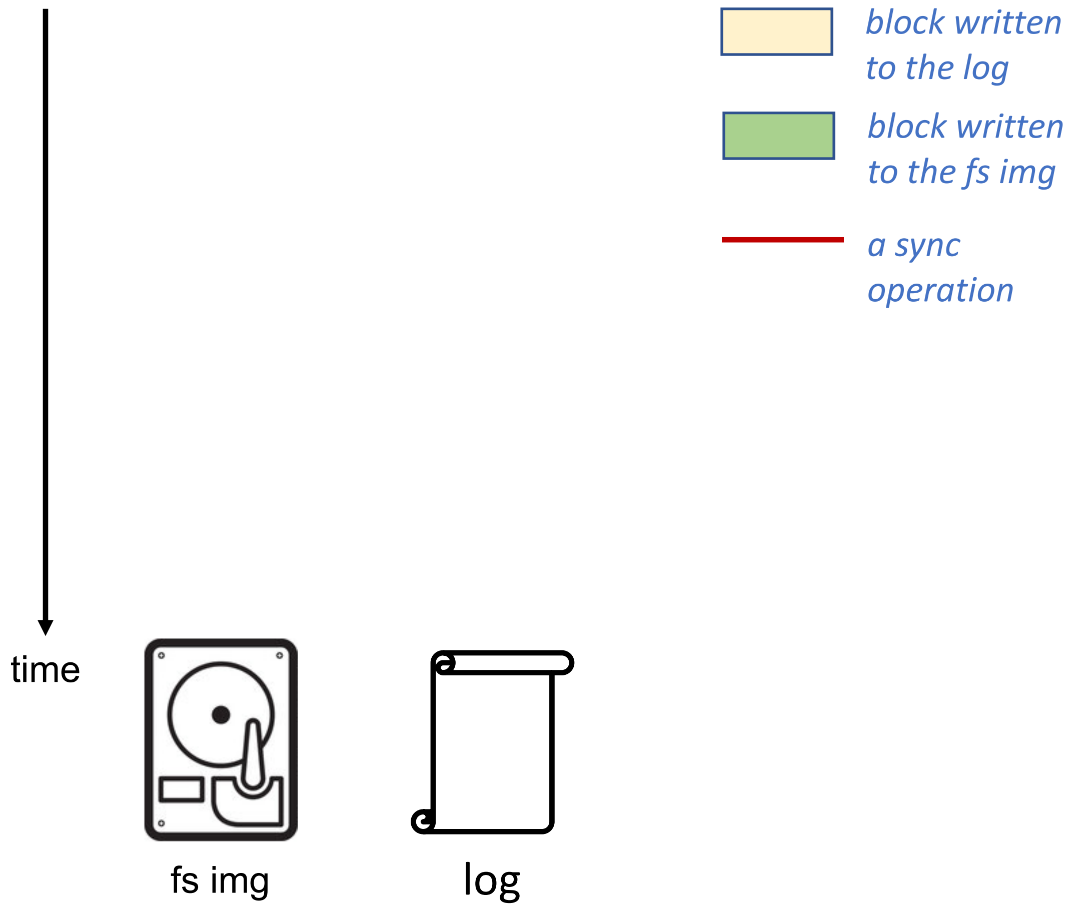
One simple fix: `"e2fsck-patch"`

Enforce synchronous I/O to the log device

Add "O_SYNC" flag while opening the log device

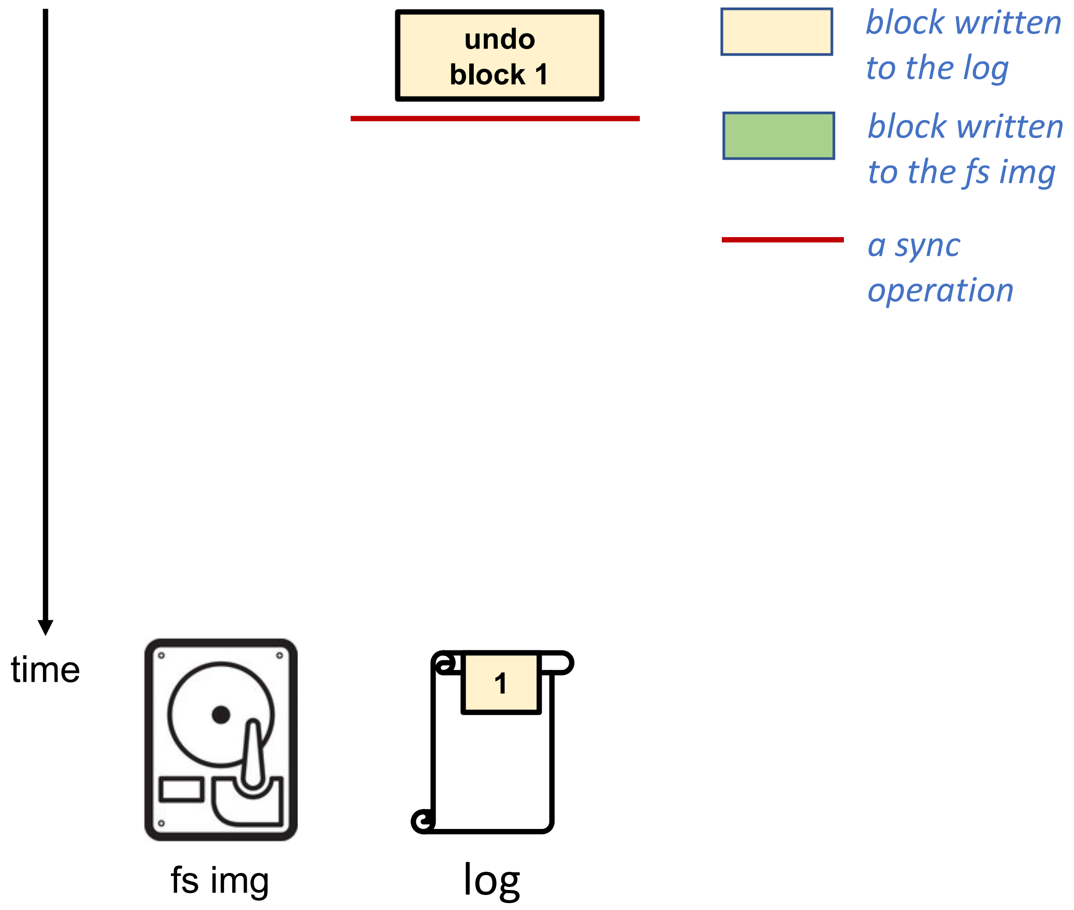
One simple fix: `e2fsck-patch`

```
/*open undo log*/
undo_open(...){
    open(...);   /*add O_SYNC */
}
...
/*fix 1st inconsistency*/
undo_write_blk64(...){
    /*write to undo log asynchronously*/
    undo_write_tdb(...){
        ...
        pwrite(...);
    }
    /*write to fs image asynchronously*/
    io_channel_write_blk64(...){...}
}
/*fix 2nd, 3rd, ... inconsistencies*/
...

/*sync buffered writes to fs image*/
ext2fs_flush(...){...}
/*close undo log*/
undo_close(...){...}
```



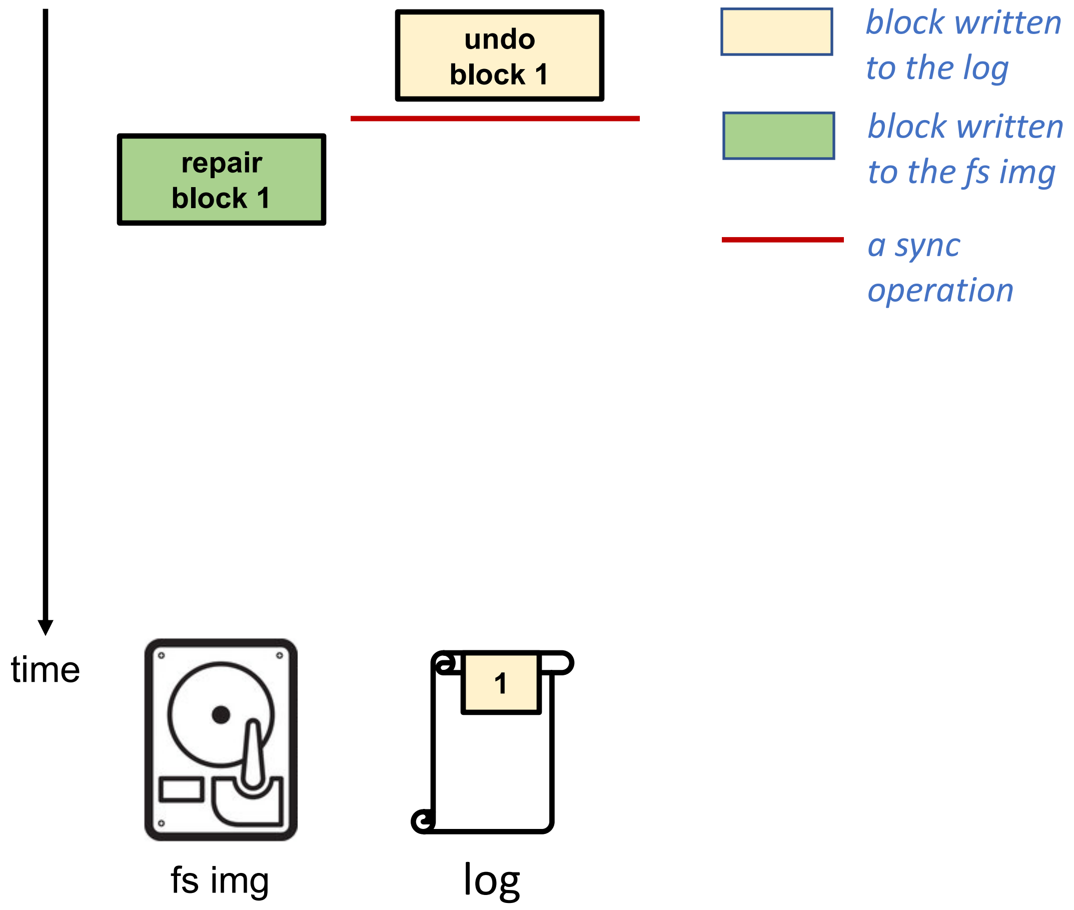
One simple fix: `e2fsck-patch`

```
/*open undo log*/
undo_open(...){
      open(...);   /*add O_SYNC */
}
...
/*fix 1st inconsistency*/
undo_write_blk64(...){
      /*write to undo log asynchronously*/
      undo_write_tdb(...){
            ...
            pwrite(...);
      }
      /*write to fs image asynchronously*/
      io_channel_write_blk64(...){...}
}
/*fix 2nd, 3rd, ... inconsistencies*/
...

/*sync buffered writes to fs image*/
ext2fs_flush(...){...}
/*close undo log*/
undo_close(...){...}
```



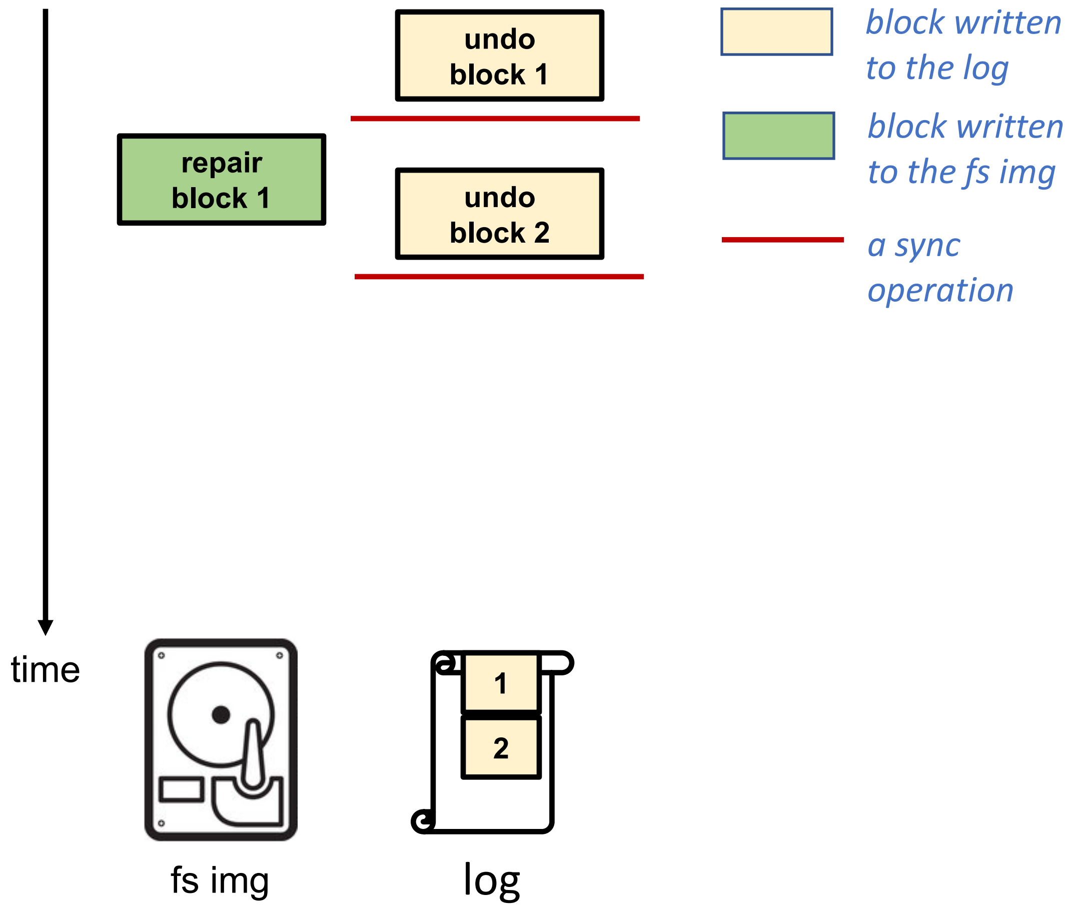
One simple fix: `e2fsck-patch`

```
/*open undo log*/
undo_open(...){
    open(...);  /*add O_SYNC */
}
...
/*fix 1st inconsistency*/
undo_write_blk64(...){
    /*write to undo log asynchronously*/
    undo_write_tdb(...){
        ...
        pwrite(...);
    }
    /*write to fs image asynchronously*/
    io_channel_write_blk64(...){...}
}
/*fix 2nd, 3rd, ... inconsistencies*/
...

/*sync buffered writes to fs image*/
ext2fs_flush(...){...}
/*close undo log*/
undo_close(...){...}
```



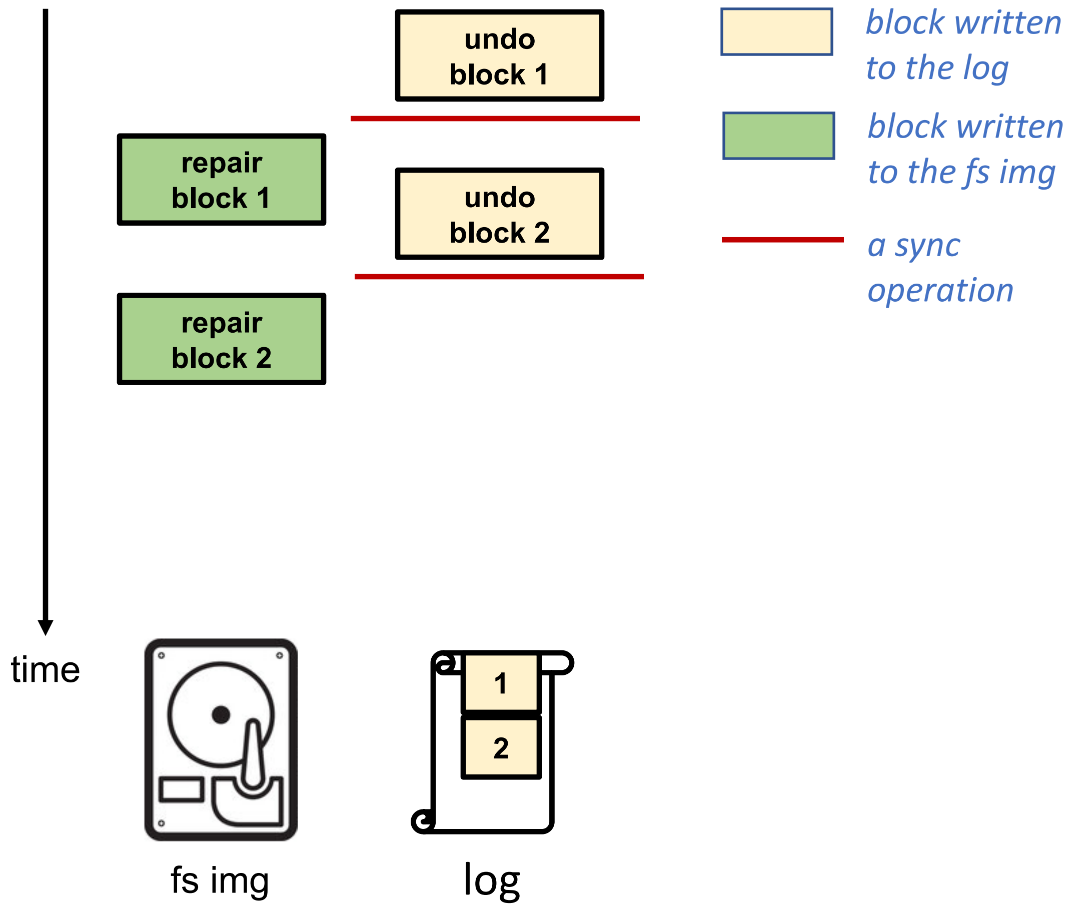
One simple fix: e2fsck-patch

```
/*open undo log*/
undo_open(...){
      open(...);   /*add O_SYNC */
}
...
/*fix 1st inconsistency*/
undo_write_blk64(...){
      /*write to undo log asynchronously*/
      undo_write_tdb(...){
            ...
            pwrite(...);
      }
      /*write to fs image asynchronously*/
      io_channel_write_blk64(...){...}
}
/*fix 2nd, 3rd, ... inconsistencies*/
...

/*sync buffered writes to fs image*/
ext2fs_flush(...){...}
/*close undo log*/
undo_close(...){...}
```



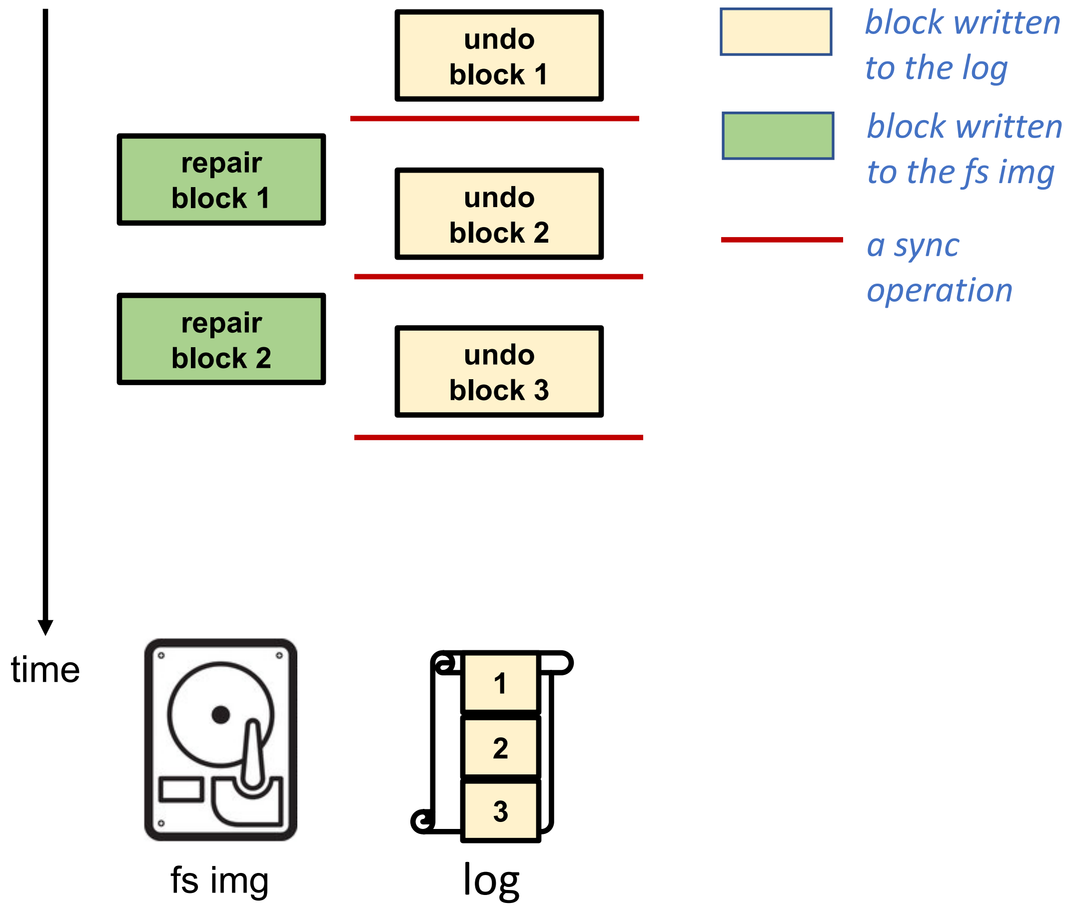
One simple fix: `e2fsck-patch`

```
/*open undo log*/
undo_open(...){
     open(...);   /*add O_SYNC */
}
...
/*fix 1st inconsistency*/
undo_write_blk64(...){
     /*write to undo log asynchronously*/
     undo_write_tdb(...){
          ...
          pwrite(...);
     }
     /*write to fs image asynchronously*/
     io_channel_write_blk64(...){...}
}
/*fix 2nd, 3rd, ... inconsistencies*/
...

/*sync buffered writes to fs image*/
ext2fs_flush(...){...}
/*close undo log*/
undo_close(...){...}
```



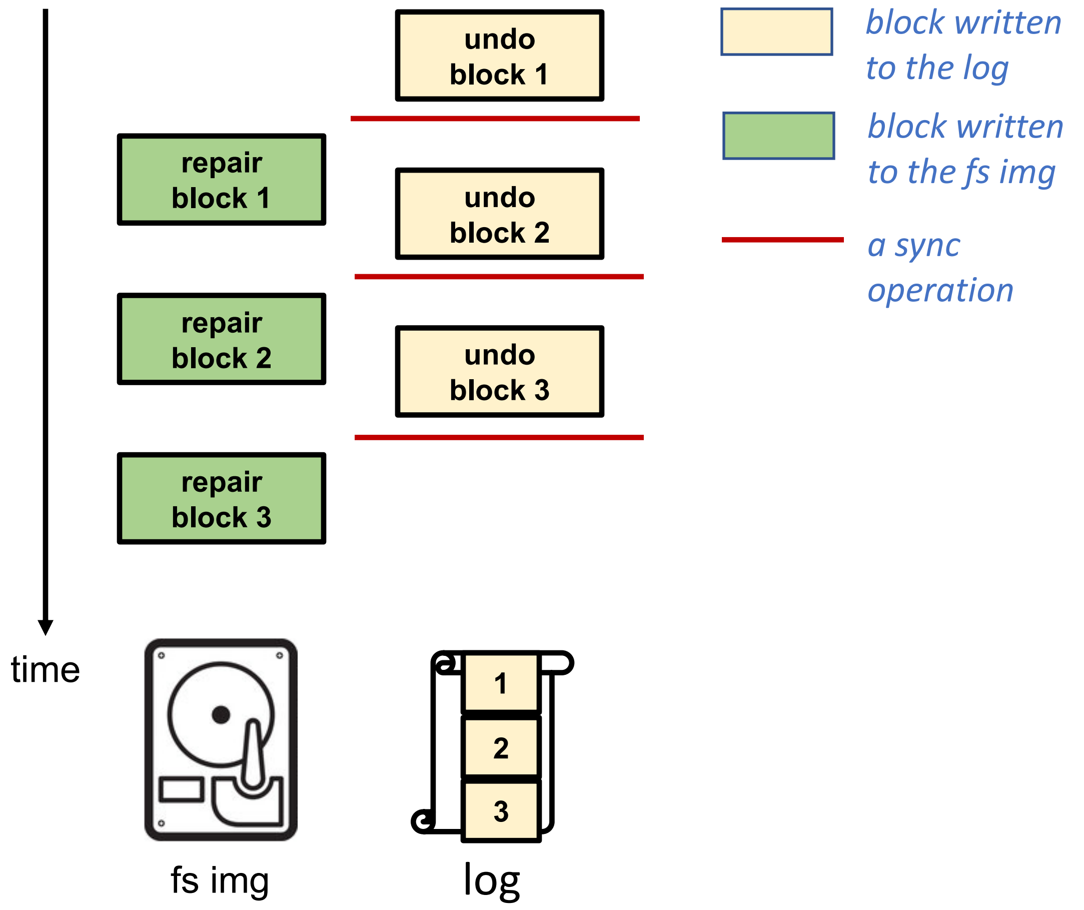
One simple fix: `e2fsck-patch`

```
/*open undo log*/
undo_open(...){
      open(...);   /*add O_SYNC */
}
...
/*fix 1st inconsistency*/
undo_write_blk64(...){
      /*write to undo log asynchronously*/
      undo_write_tdb(...){
            ...
            pwrite(...);
      }
      /*write to fs image asynchronously*/
      io_channel_write_blk64(...){...}
}
/*fix 2nd, 3rd, ... inconsistencies*/
...

/*sync buffered writes to fs image*/
ext2fs_flush(...){...}
/*close undo log*/
undo_close(...){...}
```



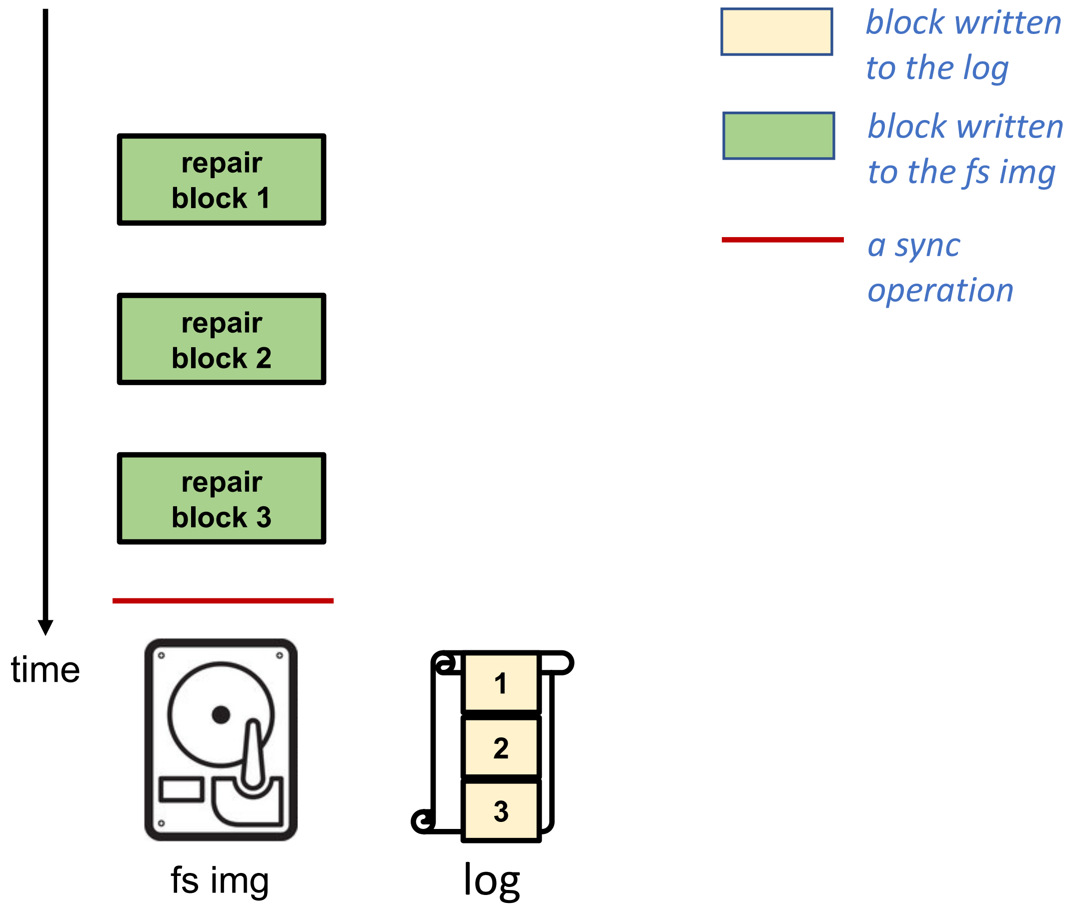
One simple fix: `e2fsck-patch`

```
/*open undo log*/
undo_open(...){
    open(...);   /*add O_SYNC */
}
...
/*fix 1st inconsistency*/
undo_write_blk64(...){
    /*write to undo log asynchronously*/
    undo_write_tdb(...){
        ...
        pwrite(...);
    }
    /*write to fs image asynchronously*/
    io_channel_write_blk64(...){...}
}
/*fix 2nd, 3rd, ... inconsistencies*/
...

/*sync buffered writes to fs image*/
ext2fs_flush(...){...}
/*close undo log*/
undo_close(...){...}
```



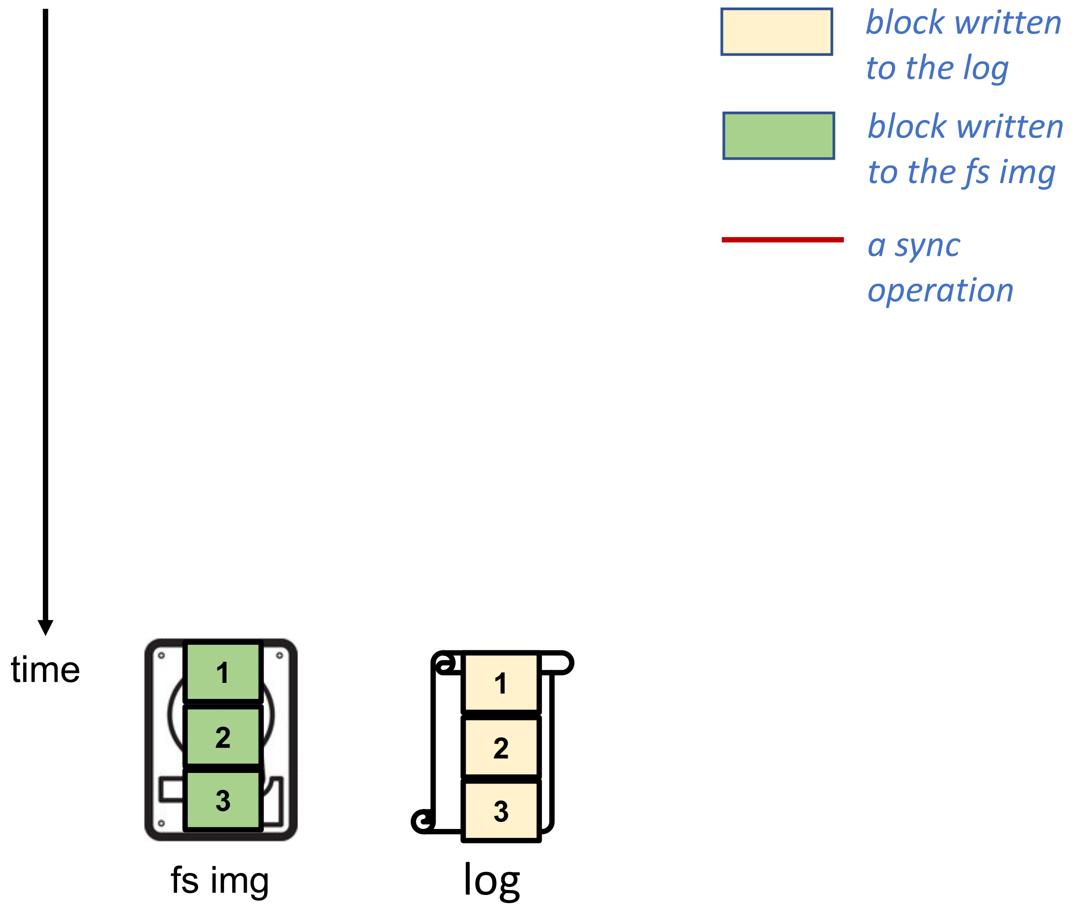
One simple fix: `e2fsck-patch`

```
/*open undo log*/
undo_open(...){
      open(...);   /*add O_SYNC */
}
...
/*fix 1st inconsistency*/
undo_write_blk64(...){
      /*write to undo log asynchronously*/
      undo_write_tdb(...){
            ...
            pwrite(...);
      }
      /*write to fs image asynchronously*/
      io_channel_write_blk64(...){...}
}
/*fix 2nd, 3rd, ... inconsistencies*/
...

/*sync buffered writes to fs image*/
ext2fs_flush(...){...}
/*close undo log*/
undo_close(...){...}
```



One simple fix: `e2fsck-patch`

```
/*open undo log*/
undo_open(...){
      open(...);   /*add O_SYNC */
}
...
/*fix 1st inconsistency*/
undo_write_blk64(...){
      /*write to undo log asynchronously*/
      undo_write_tdb(...){
            ...
            pwrite(...);
      }
      /*write to fs image asynchronously*/
      io_channel_write_blk64(...){...}
}
/*fix 2nd, 3rd, ... inconsistencies*/
...

/*sync buffered writes to fs image*/
ext2fs_flush(...){...}
/*close undo log*/
undo_close(...){...}
```



One simple fix: `e2fsck-patch`

Drawbacks of this approach:

1. Extensive synchronization incurs severe performance overhead
2. Only works with `e2fsck`

One simple fix: `e2fsck-patch`

Drawbacks of this approach:

1. Extensive synchronization incurs severe performance overhead
2. Only works with `e2fsck`

Can we design a generalized logging library with low performance overhead?

Robust File System Checker

Observe similarities among different checkers:

1. Most checkers use write system calls (`pwrite` and its variants)

2. Repairs within independent areas of file system layout
   E.g.: block groups in Ext4, allocation groups in XFS, etc.

3. Subset of total writes may cause severe corruption
   - **Key idea** is to maintain atomicity of checker's writes

Robust File System Checker

Observe similarities among different checkers:

1. Most checkers use write system calls (`pwrite` and its variants)
   - **Redirect all writes to the log**
2. Repairs within independent areas of file system layout
   E.g.: block groups in Ext4, allocation groups in XFS, etc.

3. Subset of total writes may cause severe corruption
   - **Key idea** is to maintain atomicity of relevant writes

Robust File System Checker

Observe similarities among different checkers:

1. Most checkers use write system calls (`pwrite` and its variants)
   - **Redirect all writes to the log**
2. Repairs within independent areas of file system layout
   E.g.: block groups in Ext4, allocation groups in XFS, etc.
3. Subset of total writes may cause severe corruption
   - **Key idea** is to maintain atomicity of relevant writes

Fine-grained logging with safe transactions

General Logging Library: `rfsck-lib`

Design a general redo log library "**`rfsck-lib`**"
- Log format extended from undo log in `e2fsck`

General Logging Library: `rfsck-lib`

Design a general redo log library "**`rfsck-lib`**"

- Log format extended from undo log in `e2fsck`

Fine-grained logging using safe transactions

- Maintain atomicity of relevant writes

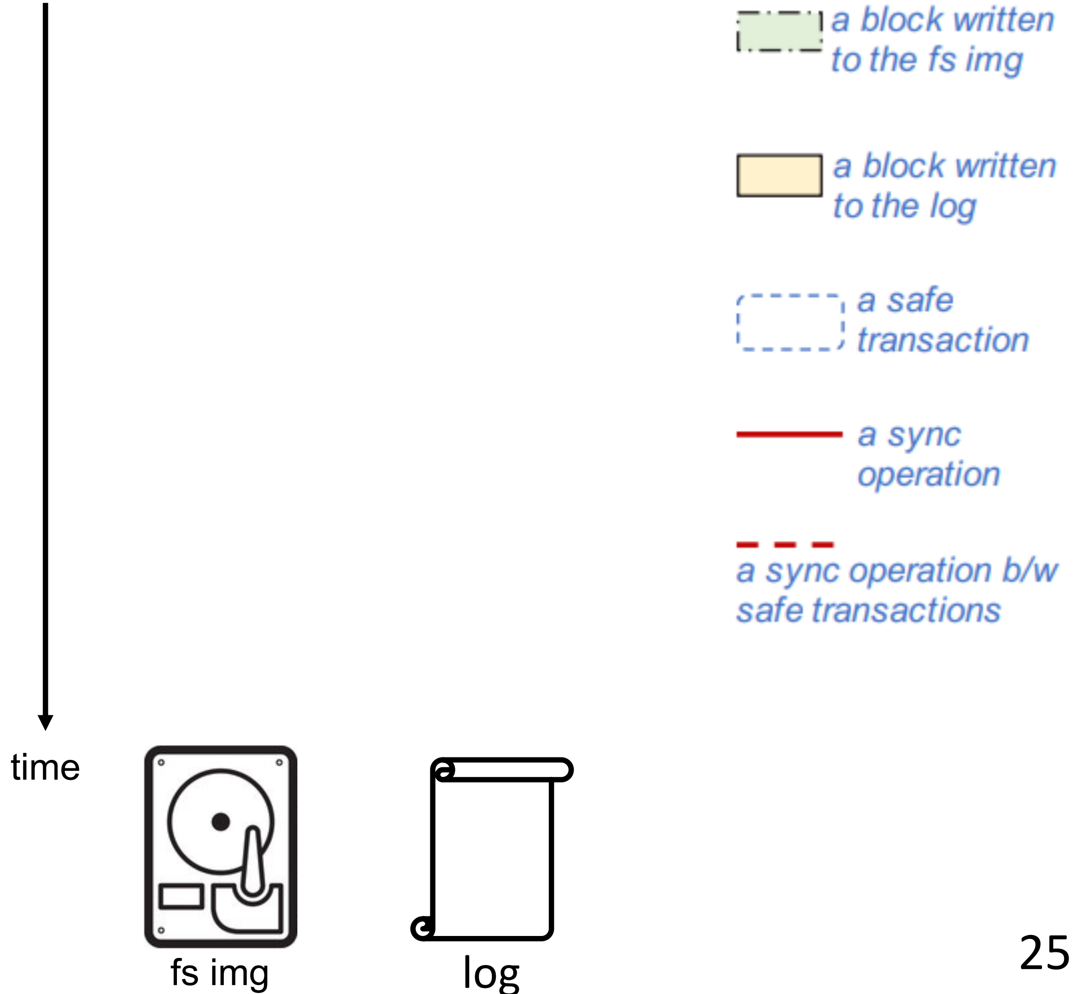
Multiple ways to integrate with tradeoff:

- Mark all repairs as one transaction
- Mark repairs of each pass as one transaction
- Mark repairs for each consistency rule as one transaction

General Logging Library: rfsck-lib

Log Format:

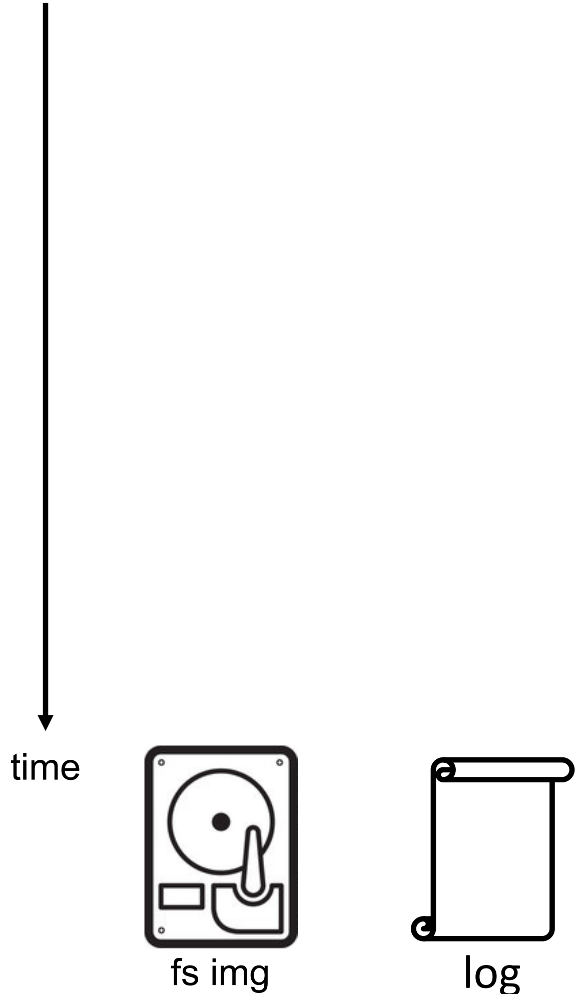
```
/*open redo log*/
redo_open(...) {
        open(...); /* no O_SYNC */
        ...
        rfsck_get_sb(...);   /* fetch superblock */
}
...
/* begin a transaction */
rfsck_txn_begin(...);
...
rfsck_write(...);    /* record an update */
...
/* end a transaction */
rfsck_txn_end(...);
...
rfsck_flush(...);    /* flush updates to the log */
...
rfsck_replay(...);  /* replay updates from log to disk */
```

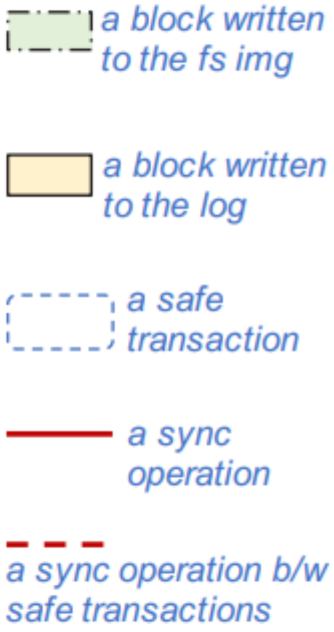


General Logging Library: `rfsck-lib`

Log Format:


```
/*open redo log*/
redo_open(...) {
        open(...); /* no O_SYNC */
        ...
        rfsck_get_sb(...);   /* fetch superblock */
}
...
/* begin a transaction */
rfsck_txn_begin(...);
...
rfsck_write(...);    /* record an update */
...
/* end a transaction */
rfsck_txn_end(...);
...
rfsck_flush(...);    /* flush updates to the log */
...
rfsck_replay(...);  /* replay updates from log to disk */
```

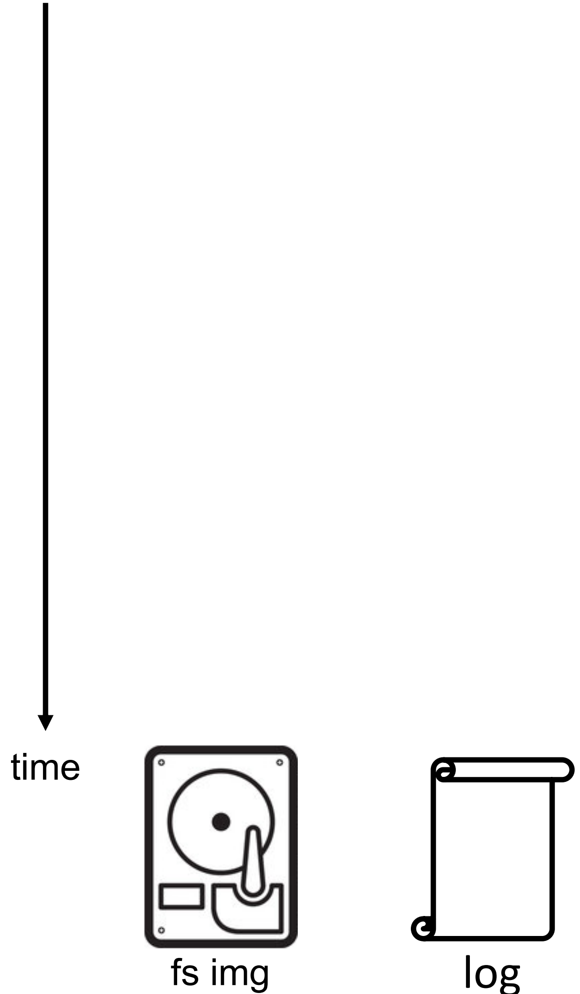


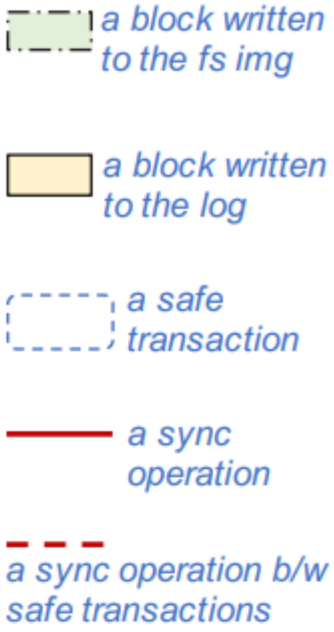


General Logging Library: rfsck-lib

Log Format:

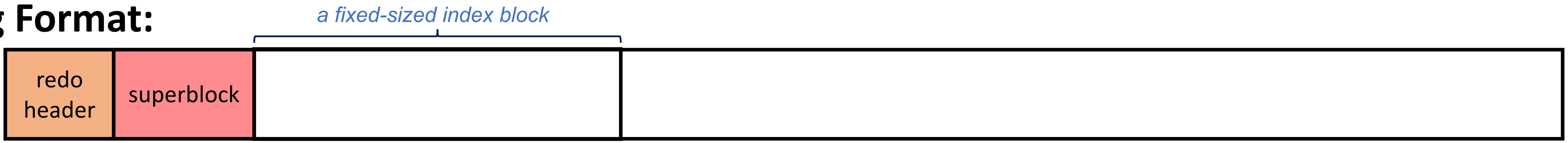

```
/*open redo log*/
redo_open(...) {
        open(...); /* no O_SYNC */
        ...
        rfsck_get_sb(...);   /* fetch superblock */
}
...
/* begin a transaction */
rfsck_txn_begin(...);
...
rfsck_write(...);    /* record an update */
...
/* end a transaction */
rfsck_txn_end(...);
...
rfsck_flush(...);    /* flush updates to the log */
...
rfsck_replay(...);  /* replay updates from log to disk */
```



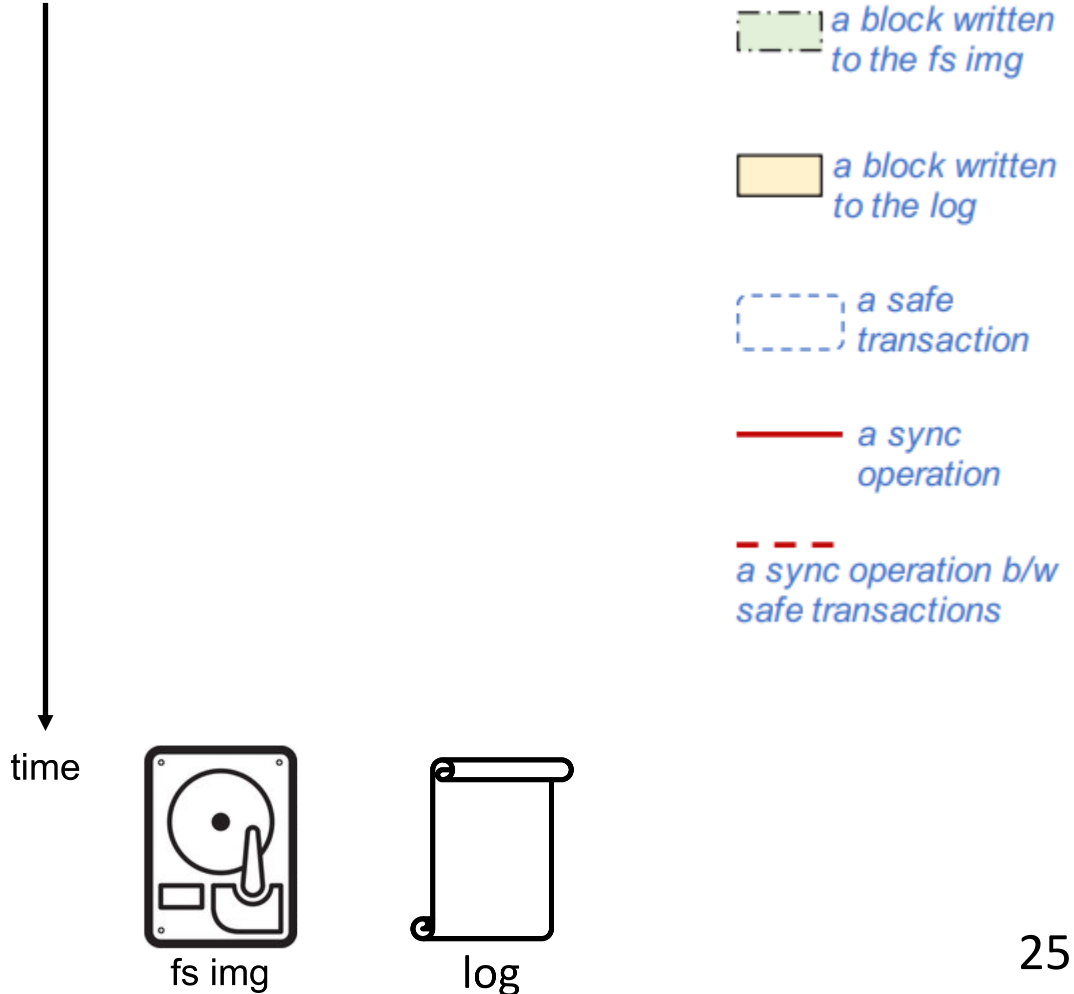


General Logging Library: `rfsck-lib`

Log Format:

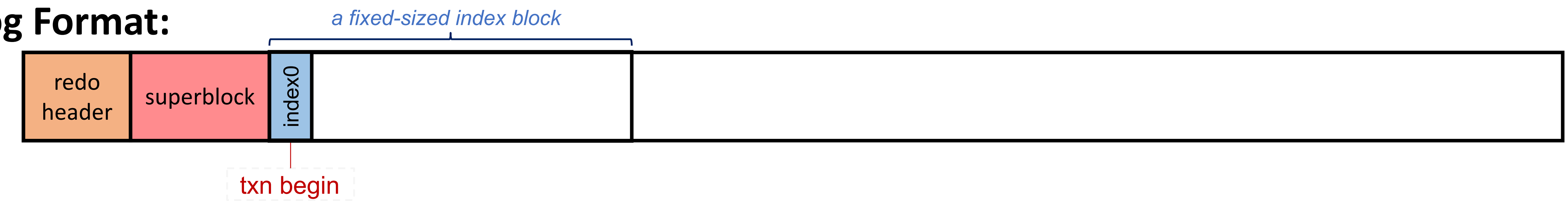



```
/*open redo log*/
redo_open(...) {
        open(...); /* no O_SYNC */
        ...
        rfsck_get_sb(...);   /* fetch superblock */
}
...
/* begin a transaction */
rfsck_txn_begin(...);
...
rfsck_write(...);    /* record an update */
...
/* end a transaction */
rfsck_txn_end(...);
...
rfsck_flush(...);    /* flush updates to the log */
...
rfsck_replay(...);  /* replay updates from log to disk */
```

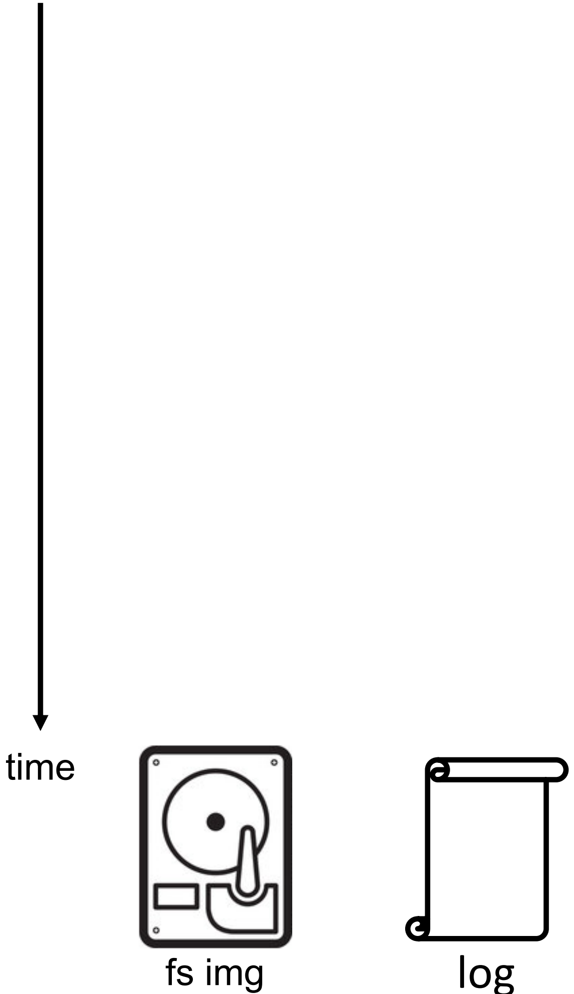


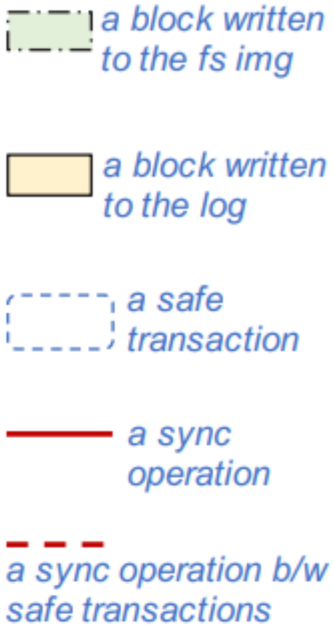
General Logging Library: `rfsck-lib`

Log Format:



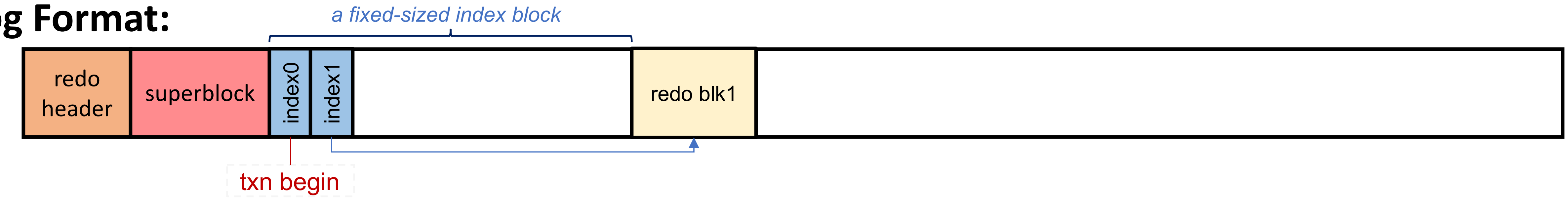

```
/*open redo log*/
redo_open(...) {
        open(...); /* no O_SYNC */
        ...
        rfsck_get_sb(...);   /* fetch superblock */
}
...
/* begin a transaction */
rfsck_txn_begin(...);
...
rfsck_write(...);    /* record an update */
...
/* end a transaction */
rfsck_txn_end(...);
...
rfsck_flush(...);    /* flush updates to the log */
...
rfsck_replay(...);  /* replay updates from log to disk */
```



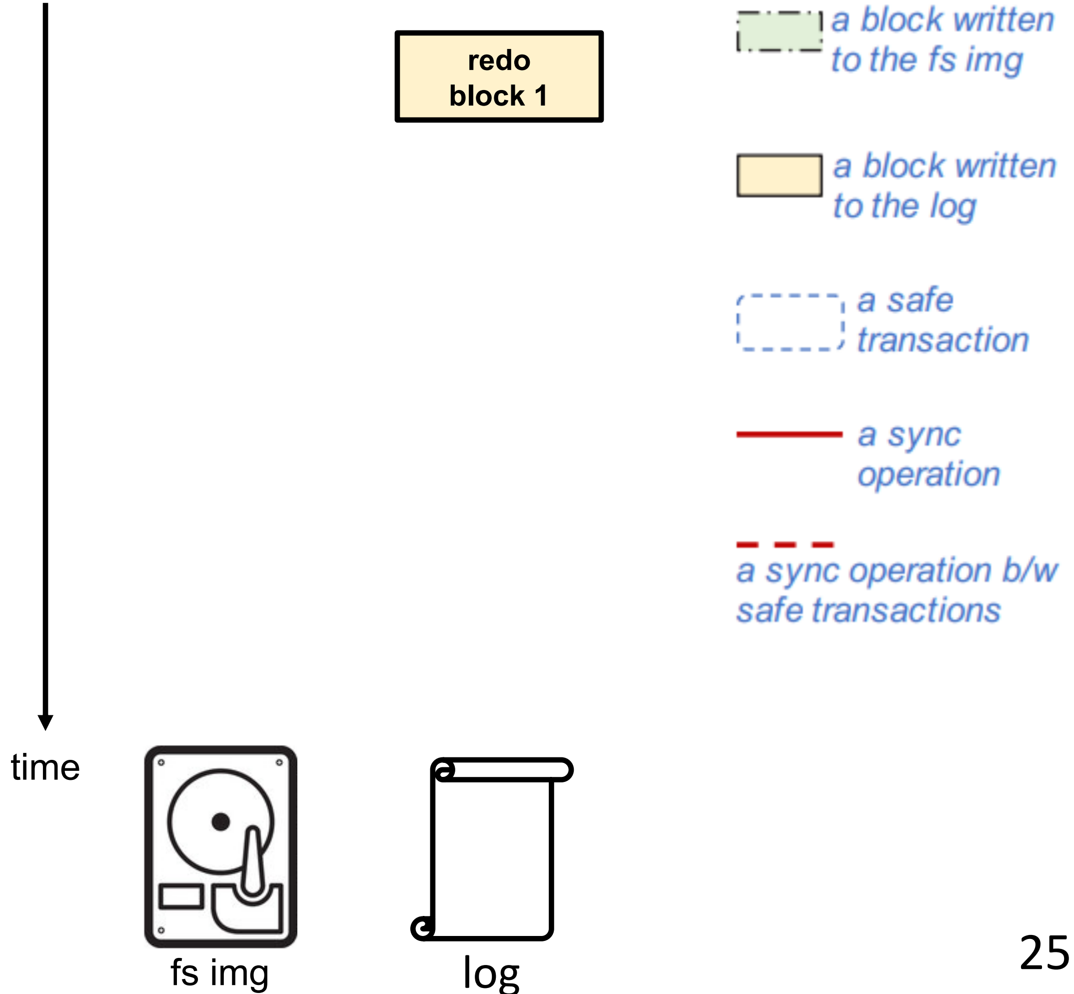


General Logging Library: `rfsck-lib`

Log Format:

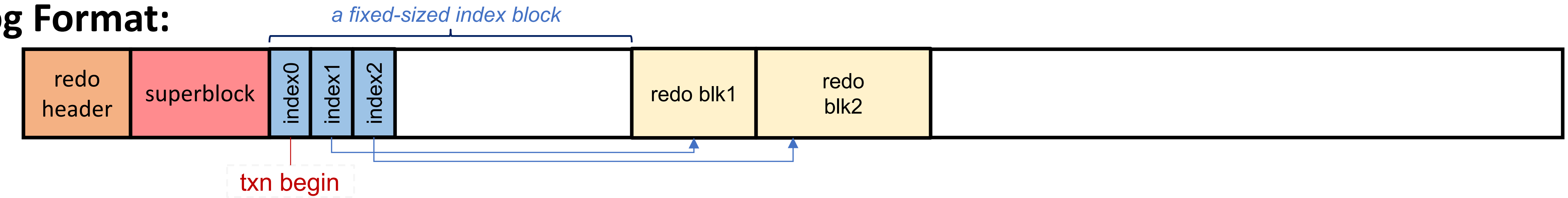



```
/*open redo log*/
redo_open(...) {
        open(...); /* no O_SYNC */
        ...
        rfsck_get_sb(...);   /* fetch superblock */
}
...
/* begin a transaction */
rfsck_txn_begin(...);
...
rfsck_write(...);    /* record an update */
...
/* end a transaction */
rfsck_txn_end(...);
...
rfsck_flush(...);    /* flush updates to the log */
...
rfsck_replay(...);  /* replay updates from log to disk */
```

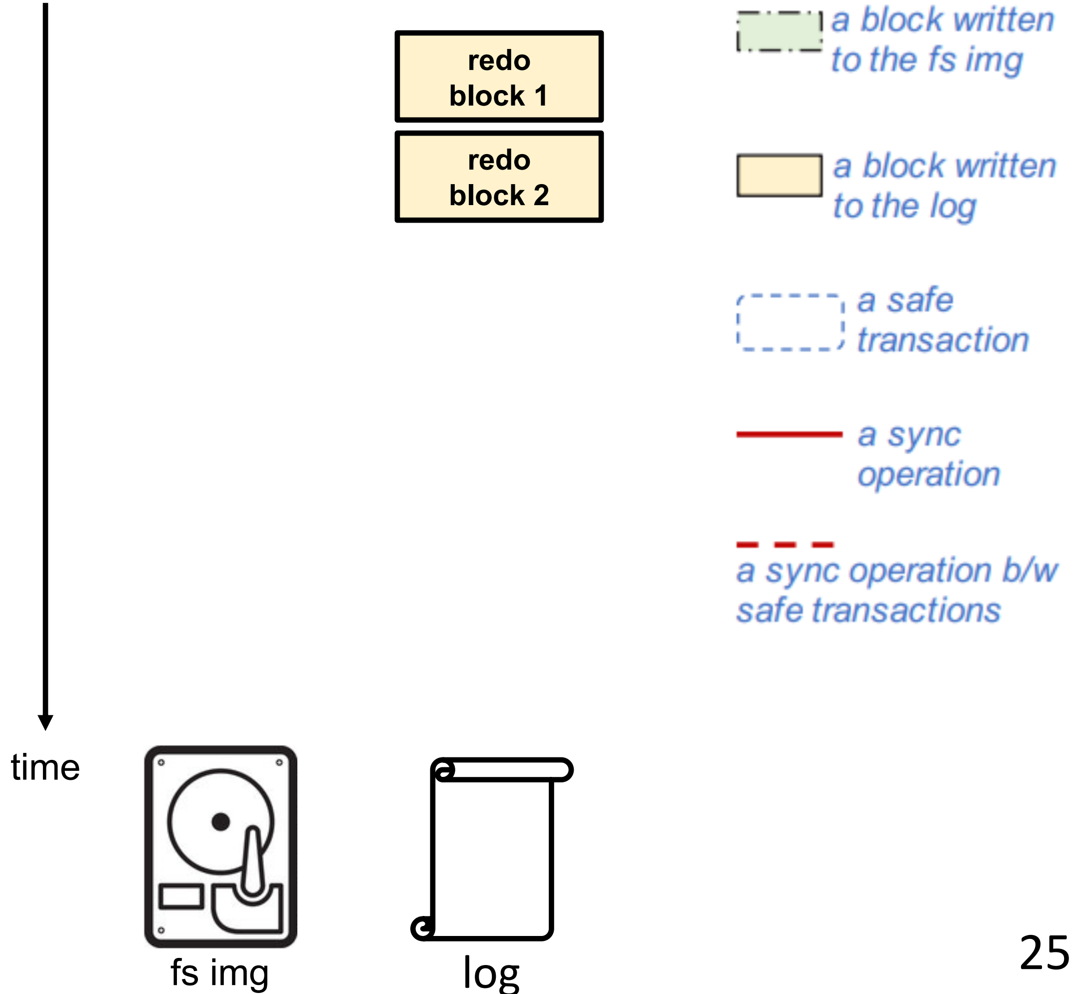


General Logging Library: `rfsck-lib`

Log Format:

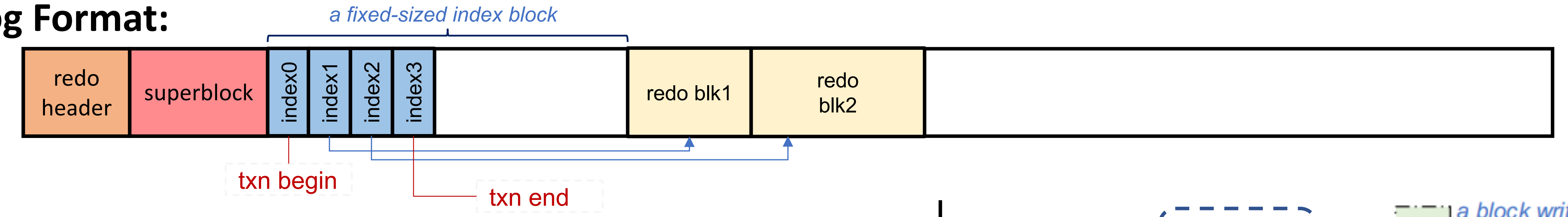



```
/*open redo log*/
redo_open(...) {
        open(...); /* no O_SYNC */
        ...
        rfsck_get_sb(...);   /* fetch superblock */
}
...
/* begin a transaction */
rfsck_txn_begin(...);
...
rfsck_write(...);     /* record an update */
...
/* end a transaction */
rfsck_txn_end(...);
...
rfsck_flush(...);     /* flush updates to the log */
...
rfsck_replay(...);  /* replay updates from log to disk */
```

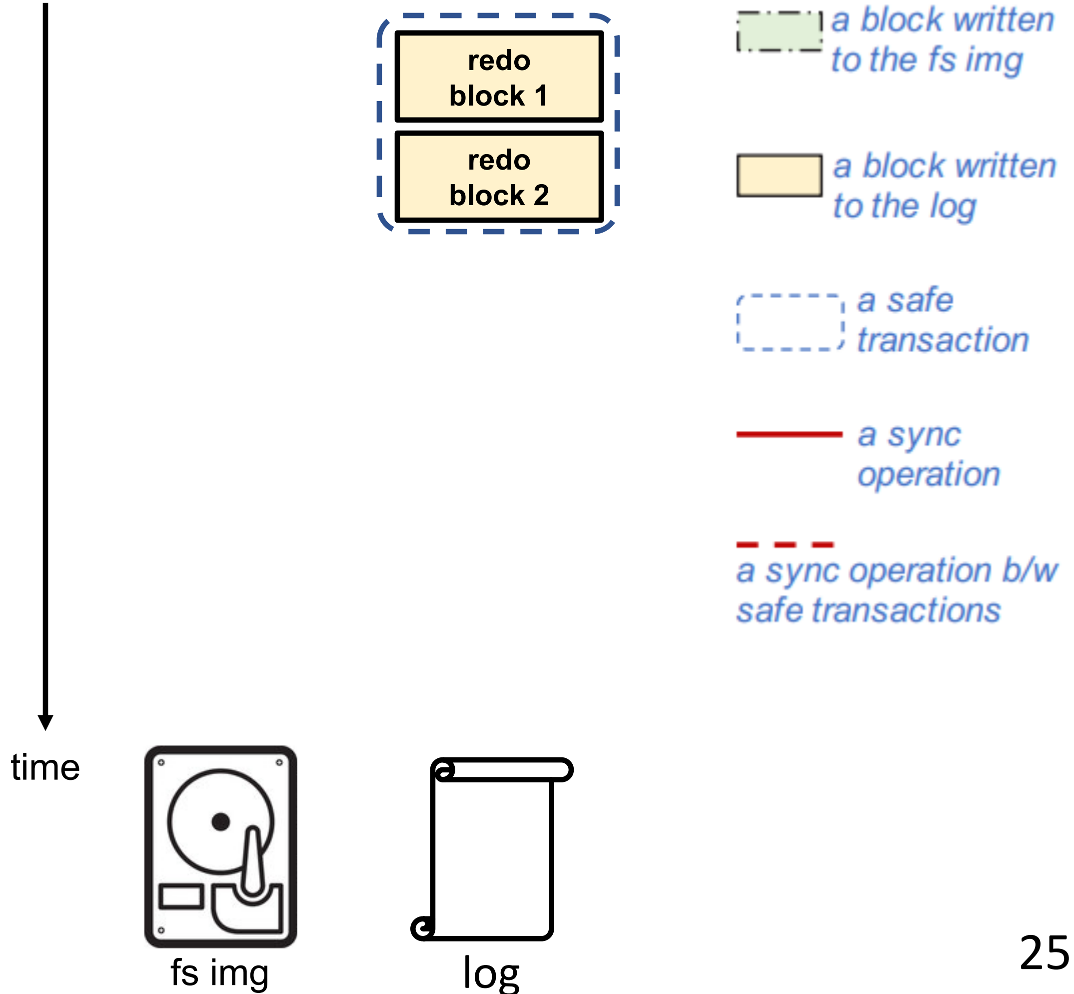


General Logging Library: `rfsck-lib`

Log Format:

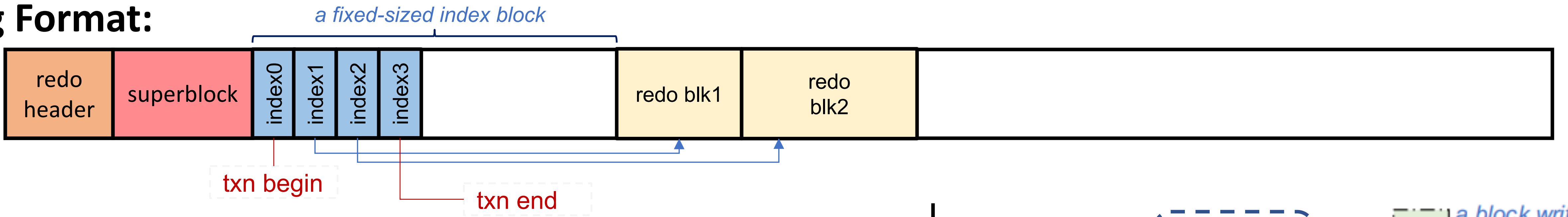



```
/*open redo log*/
redo_open(...) {
        open(...); /* no O_SYNC */
        ...
        rfsck_get_sb(...);   /* fetch superblock */
}
...
/* begin a transaction */
rfsck_txn_begin(...);
...
rfsck_write(...);    /* record an update */
...
/* end a transaction */
rfsck_txn_end(...);
...
rfsck_flush(...);    /* flush updates to the log */
...
rfsck_replay(...);  /* replay updates from log to disk */
```

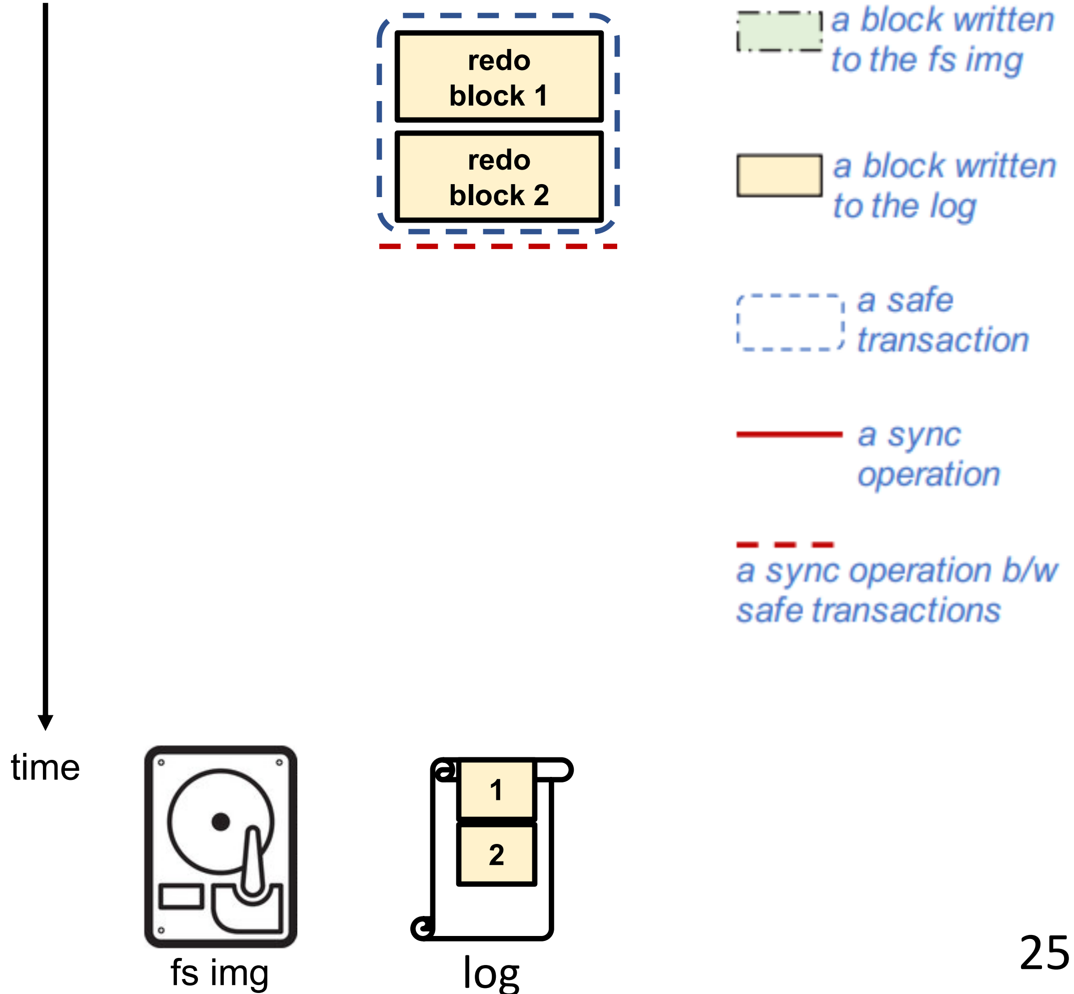


General Logging Library: rfsck-lib

Log Format:

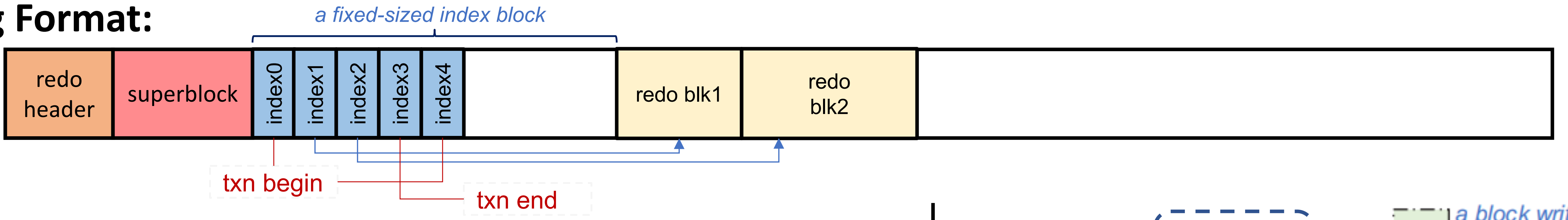



```
/*open redo log*/
redo_open(...) {
        open(...); /* no O_SYNC */
        ...
        rfsck_get_sb(...);   /* fetch superblock */
}
...
/* begin a transaction */
rfsck_txn_begin(...);
...
rfsck_write(...);    /* record an update */
...
/* end a transaction */
rfsck_txn_end(...);
...
rfsck_flush(...);    /* flush updates to the log */
...
rfsck_replay(...);  /* replay updates from log to disk */
```

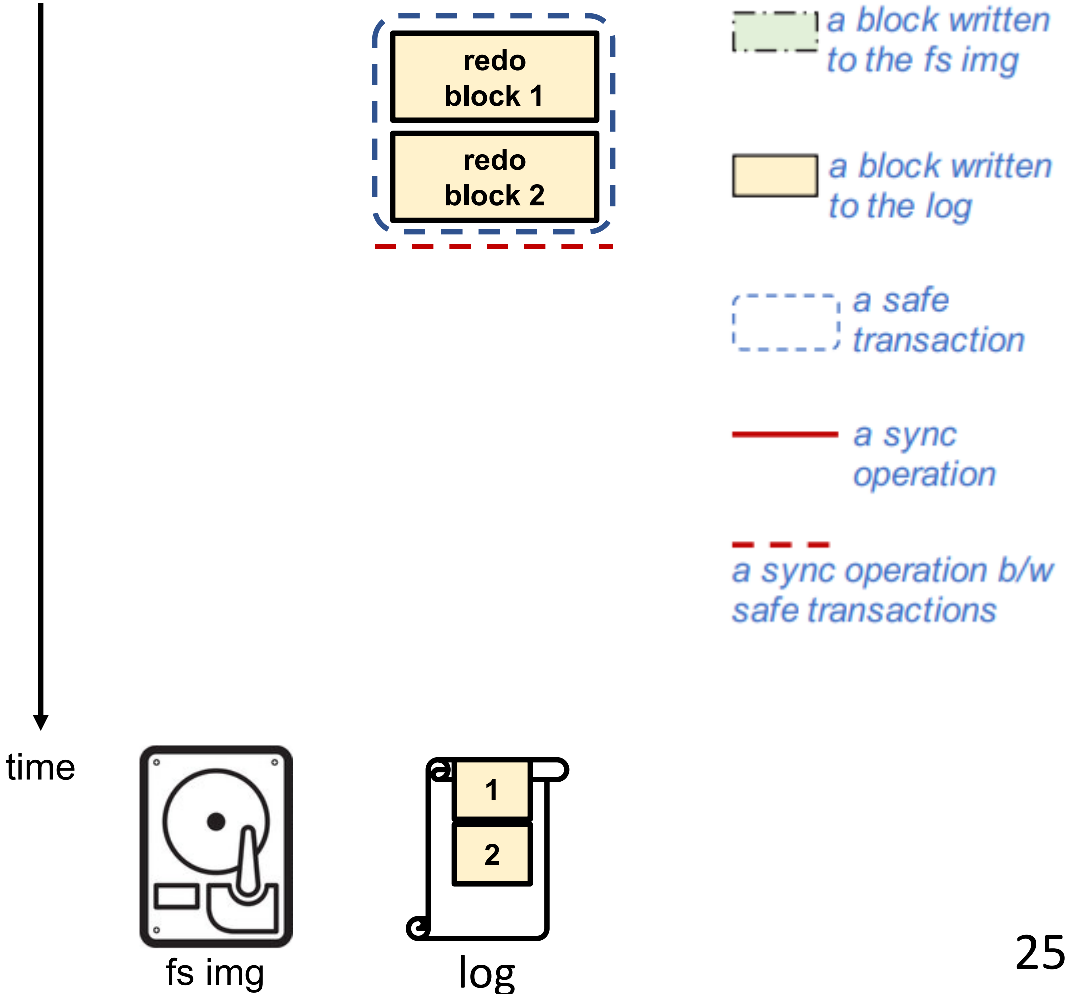


General Logging Library: rfsck-lib

Log Format:

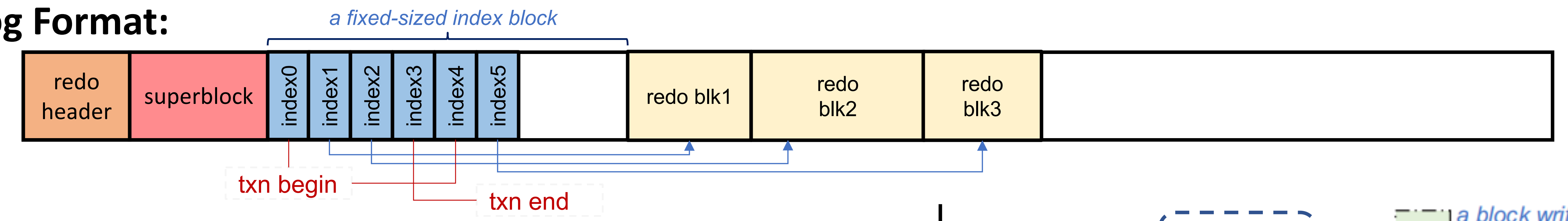



```
/*open redo log*/
redo_open(...) {
        open(...); /* no O_SYNC */
        ...
        rfsck_get_sb(...);   /* fetch superblock */
}
...
/* begin a transaction */
rfsck_txn_begin(...);
...
rfsck_write(...);    /* record an update */
...
/* end a transaction */
rfsck_txn_end(...);
...
rfsck_flush(...);    /* flush updates to the log */
...
rfsck_replay(...);  /* replay updates from log to disk */
```

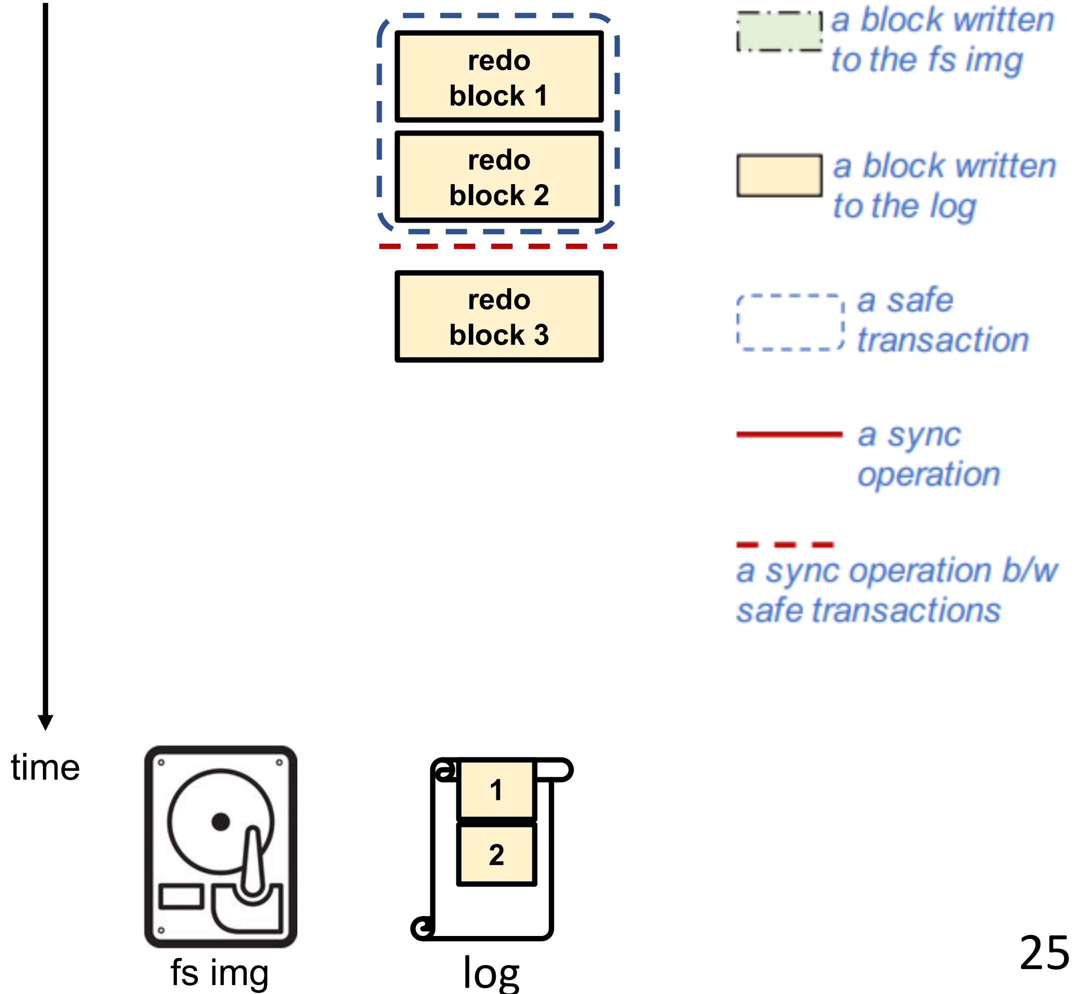


General Logging Library: `rfsck-lib`

Log Format:

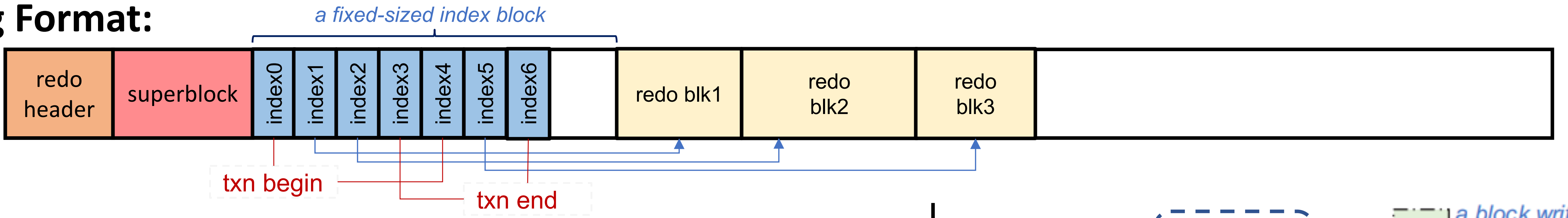



```
/*open redo log*/
redo_open(...) {
        open(...); /* no O_SYNC */
        ...
        rfsck_get_sb(...);   /* fetch superblock */
}
...
/* begin a transaction */
rfsck_txn_begin(...);
...
rfsck_write(...);    /* record an update */
...
/* end a transaction */
rfsck_txn_end(...);
...
rfsck_flush(...);    /* flush updates to the log */
...
rfsck_replay(...);  /* replay updates from log to disk */
```

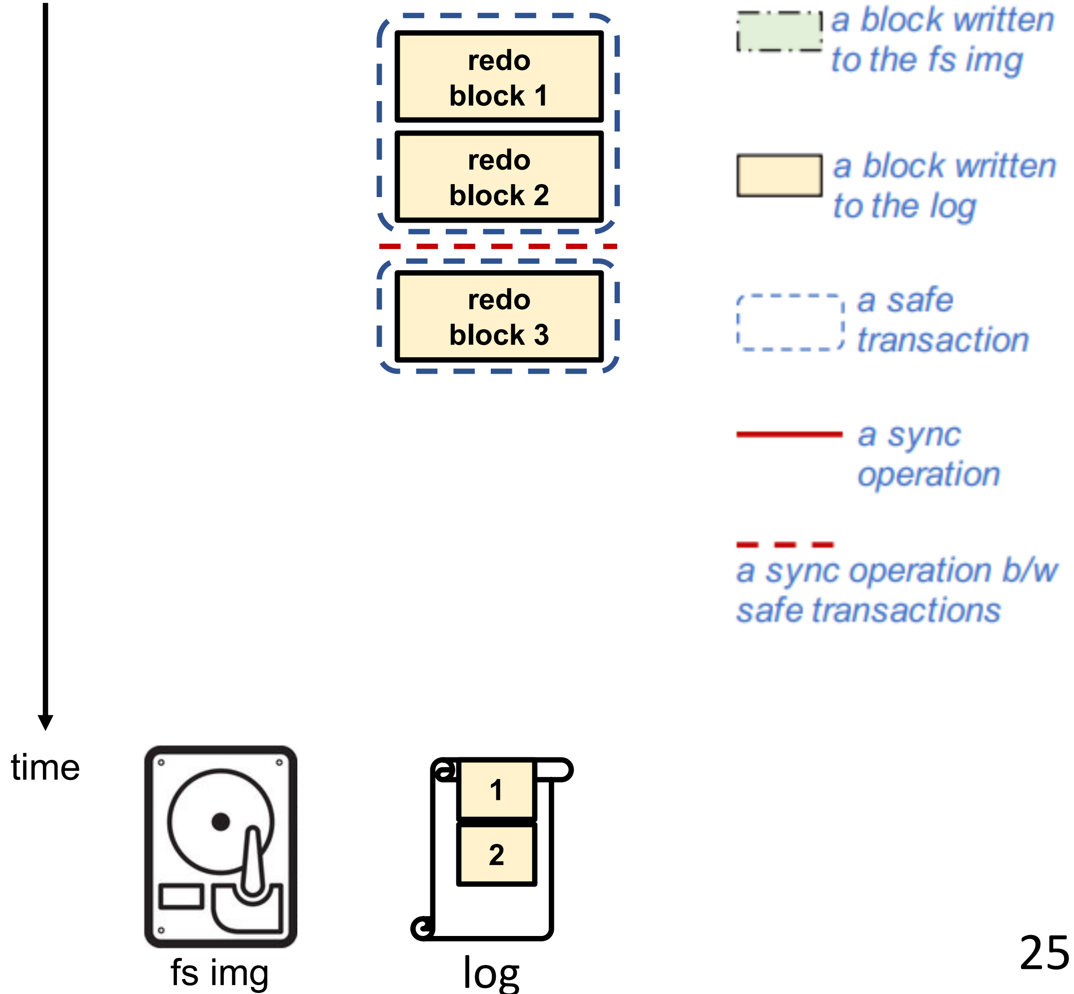


General Logging Library: `rfsck-lib`

Log Format:

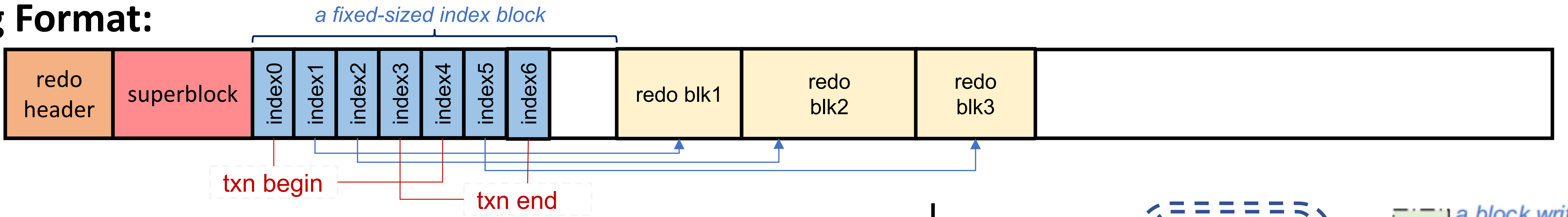



```
/*open redo log*/
redo_open(...) {
        open(...); /* no O_SYNC */
        ...
        rfsck_get_sb(...);   /* fetch superblock */
}
...
/* begin a transaction */
rfsck_txn_begin(...);
...
rfsck_write(...);    /* record an update */
...
/* end a transaction */
rfsck_txn_end(...);
...
rfsck_flush(...);    /* flush updates to the log */
...
rfsck_replay(...);  /* replay updates from log to disk */
```

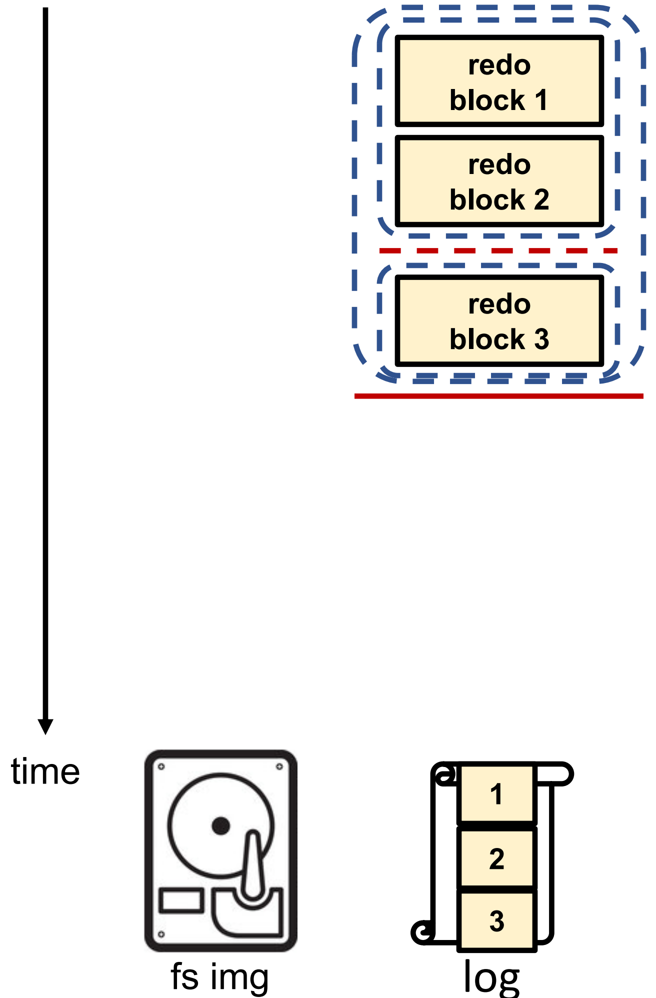


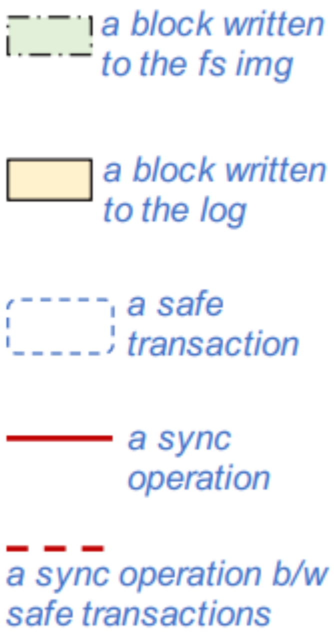
General Logging Library: `rfsck-lib`

Log Format:



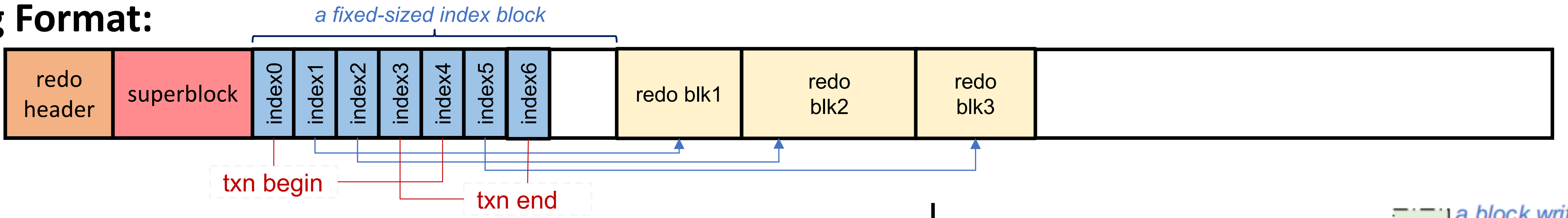

```
/*open redo log*/
redo_open(...) {
        open(...); /* no O_SYNC */
        ...
        rfsck_get_sb(...);   /* fetch superblock */
}
...
/* begin a transaction */
rfsck_txn_begin(...);
...
rfsck_write(...);    /* record an update */
...
/* end a transaction */
rfsck_txn_end(...);
...
rfsck_flush(...);    /* flush updates to the log */
...
rfsck_replay(...);  /* replay updates from log to disk */
```



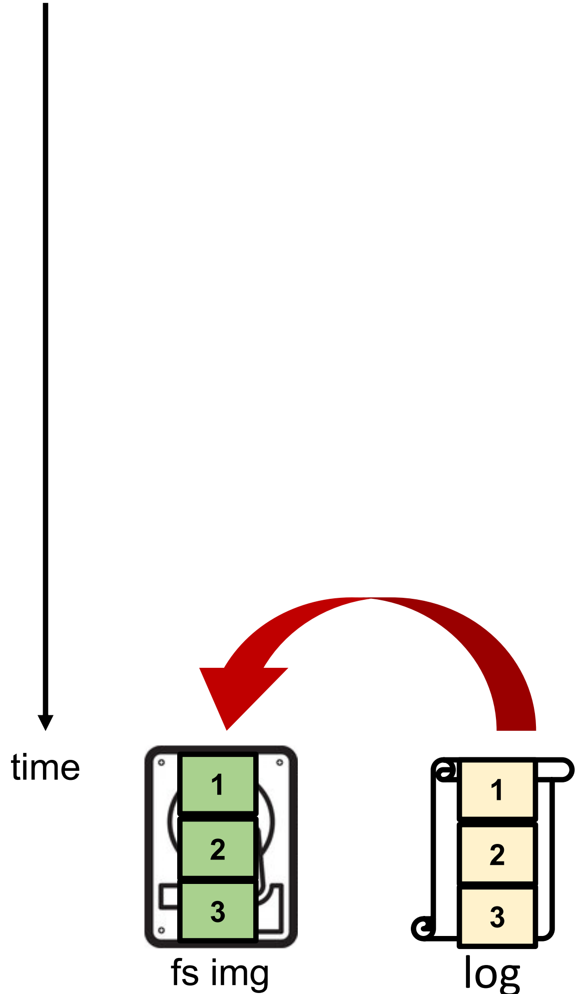


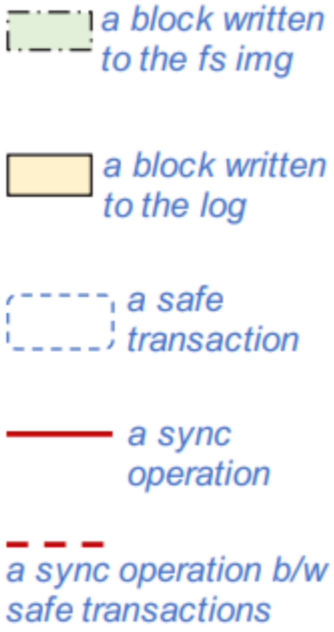
General Logging Library: rfsck-lib

Log Format:




```
/*open redo log*/
redo_open(...) {
        open(...); /* no O_SYNC */
        ...
        rfsck_get_sb(...);   /* fetch superblock */
}
...
/* begin a transaction */
rfsck_txn_begin(...);
...
rfsck_write(...);    /* record an update */
...
/* end a transaction */
rfsck_txn_end(...);
...
rfsck_flush(...);    /* flush updates to the log */
...
rfsck_replay(...);  /* replay updates from log to disk */
```





Outline

- Motivation
- Background & Related Work
- Research Question
- Are existing checkers resilient to faults?
- How to build robust checkers?
- Evaluation
- Conclusion

rfsck-lib: General Logging Library

Integration with existing checkers:

```
rfsck-lib + e2fsck      => rfsck-ext
rfsck-lib + xfs_repair => rfsck-xfs
```

	rfsck-ext	rfsck-xfs
Lines of Code	50	15
Integration	“-R” option	“-R” option
Safe transaction	For each pass	For entire run
Replay log	At the end or at restart points	At the end

Table 4: Integrating `rfsck-lib` with existing checkers

Robustness of `rfsck-lib`

Evaluation of EXT4 checkers		
Test Images	Test images reporting corruption	
	`e2fsck`	`rfsck-ext`
17	17	0

Evaluation of XFS checkers		
Test Images	Test images reporting corruption	
	`xfs_repair`	`rfsck-xfs`
12	12	0

Robustness of `rfsck-lib`

Evaluation of EXT4 checkers		
Test Images	Test images reporting corruption	
	`e2fsck`	`rfsck-ext`
17	17	0

Evaluation of XFS checkers		
Test Images	Test images reporting corruption	
	`xfs_repair`	`rfsck-xfs`
12	12	0

No corruption reported

Performance of `rfsck-lib`

Specifications:
- CPU: Intel Xeon 5160 3GHz
- RAM: 8GB
- OS: Ubuntu 16.04 (Linux Kernel v4.4)
- HDD: WD5000AAKS

Practical File System sizes of 100, 200 & 500 GB

Fill in steps using `fs_mark` tool

Corrupt metadata using `debugfs` & `xfs_db`

Performance of `rfsck-lib`

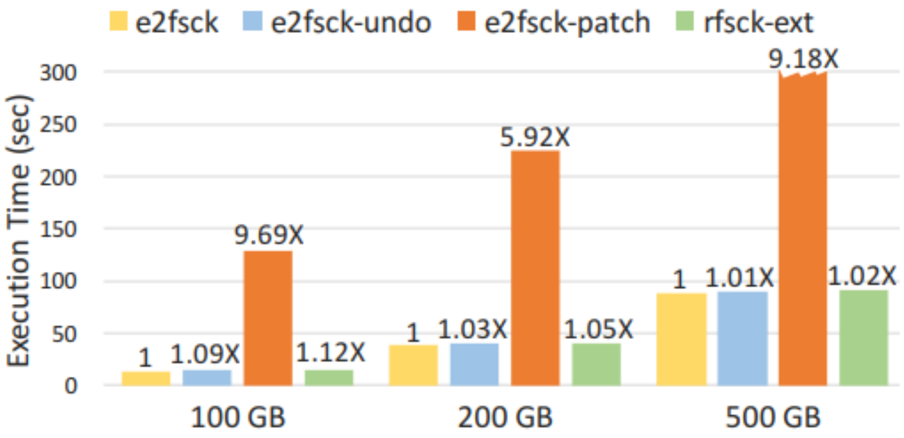


Figure 1: Performance comparison of e2fsck, e2fsck-undo, e2fsck-patch and rfsck-ext

Performance of `rfsck-lib`

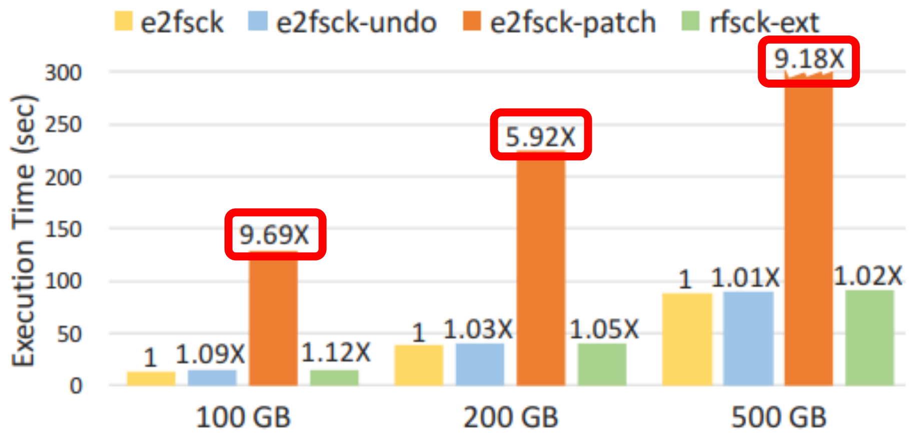


Figure 1: Performance comparison of e2fsck, e2fsck-undo, e2fsck-patch and rfsck-ext

- Degraded performance of `e2fsck-patch` due to extensive synchronization

Performance of `rfsck-lib`

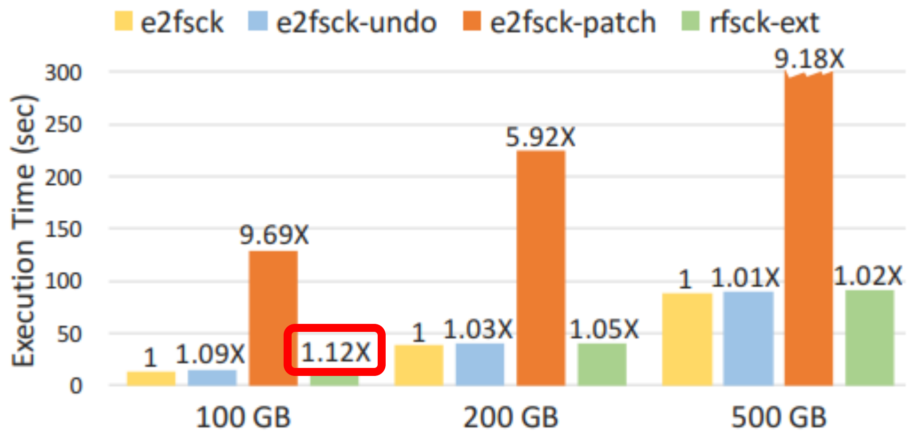


Figure 1: Performance comparison of e2fsck, e2fsck-undo, e2fsck-patch and rfsck-ext

- Degraded performance of `e2fsck-patch` due to extensive synchronization
- `rfsck-ext` incurs a max. overhead of 12%

Performance of `rfsck-lib`

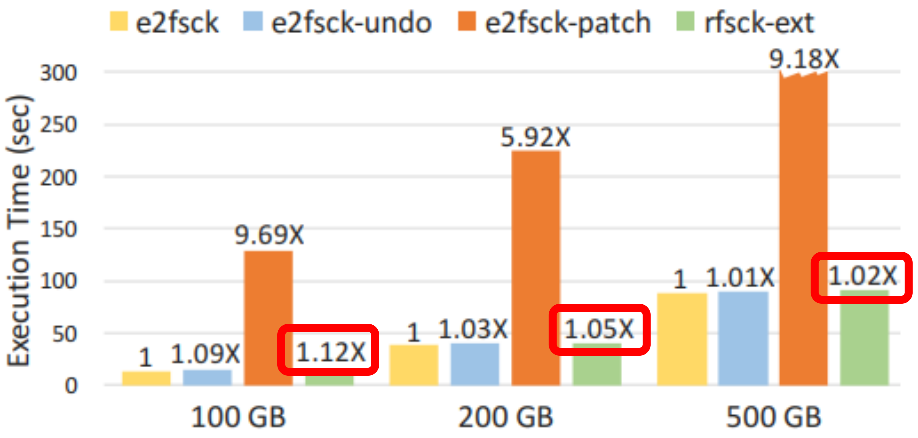


Figure 1: Performance comparison of e2fsck, e2fsck-undo, e2fsck-patch and rfsck-ext

- Degraded performance of `e2fsck-patch` due to extensive synchronization
- `rfsck-ext` incurs a max. overhead of 12%
- Overhead reduces as file system size increases
  - Runtime of checking is dominant, compared to replay

Outline

- Motivation
- Background & Related Work
- Research Question
- Are existing checkers resilient to faults?
- How to build robust checkers?
- Evaluation
- Conclusion

Conclusion

- Are existing checkers resilient to faults?

Conclusion

- Are existing checkers resilient to faults? **NO**
  - Strong dependencies among updates ➡ vulnerabilities in checkers

Conclusion

- Are existing checkers resilient to faults? **NO**
  - Strong dependencies among updates ➡ vulnerabilities in checkers
- How to build a robust checker?
  - One simple fix: e2fsck-patch

Conclusion

- Are existing checkers resilient to faults? **NO**
  - Strong dependencies among updates → vulnerabilities in checkers
- How to build a robust checker?
  - One simple fix: e2fsck-patch
  - General logging library: rfsck-lib
    - Easy to integrate

Conclusion

- Are existing checkers resilient to faults? **NO**
  - Strong dependencies among updates ➡ vulnerabilities in checkers
- How to build a robust checker?
  - One simple fix: e2fsck-patch
  - General logging library: rfsck-lib
    - Easy to integrate
- Consistent with previous studies that show "recovery procedures are imperfect"
  - Why does the cloud stop computing?: Lessons from hundreds of service outages [SoCC'16]
  - Failure recovery: When the cure is worse than the disease [HotOS'13]

Conclusion

- Are existing checkers resilient to faults? **NO**
  - Strong dependencies among updates ➡ vulnerabilities in checkers
- How to build a robust checker?
  - One simple fix: e2fsck-patch
  - General logging library: rfsck-lib
    - Easy to integrate
- Consistent with previous studies that show "recovery procedures are imperfect"
  - Why does the cloud stop computing?: Lessons from hundreds of service outages [SoCC'16]
  - Failure recovery: When the cure is worse than the disease [HotOS'13]
- Raise awareness on vulnerabilities in recovery procedures, and facilitate building fault-resilient systems

Conclusion

COMING SOON GitHub

- Are existing checkers resilient to faults? **NO**
  - Strong dependencies among updates → vulnerabilities in checkers
- How to build a robust checker?
  - One simple fix: e2fsck-patch
  - General logging library: rfsck-lib
    - Easy to integrate
- Consistent with previous studies that show “recovery procedures are imperfect”
  - Why does the cloud stop computing?: Lessons from hundreds of service outages [SoCC’16]
  - Failure recovery: When the cure is worse than the disease [HotOS’13]
- Raise awareness on vulnerabilities in recovery procedures, and facilitate building fault-resilient systems

THANK YOU

BACK UP SLIDES

Framework to interrupting the recovery

Build a fault injection tool "`rfsck-test`" using customized iSCSI driver to emulate faults

Adopt the "Clean power fault" model

- Clean I/O termination
- No ordering of I/O
- Serves as the lower bound of failure impact

Case Study: `xfs_repair`

Generated 20 test images using `xfs_db`

Block size of all images is 4KB

Fault injected at two granularities: 512B and 4KB

Case Study: `xfs_repair`

Fault injection granularity	# of XFS test images	# of repaired images generated	# of images reporting corruption	
			test images	repaired images
512 B	3	1,127	2	443
4 KB	17	1,409	12	737

Table 4: Number of test images and repaired images reporting corruption

Are existing checkers resilient to faults?

No, because there is strong dependency among updates

Also, existing logging mechanism in checkers also fail

2. Framework to interrupting the recovery

Why two modes?

2. Framework to interrupting the recovery

Why two modes?

Some checkers exhibit logging mechanism

- E.g: undo log in e2fsck

Test for resilience with logging mechanism enabled

Conclusion

Analyze the behavior of file system checkers under faults

- May lead to unrecoverable inconsistencies

Analyze the logging mechanism of existing checkers

- Fail the test of resilience

Build a general logging library “rfsck-lib” to strengthen existing checkers

Minimum LoC added for integration

Existing checkers become more robust but induce minimal performance overhead (max 12%)

Robust File System Checker

No, because there is strong dependency among updates

Also, existing logging mechanism in checkers also fail 😐

Robustness of `rfsck-lib`

Evaluated `rfsck-ext` and `rfsck-xfs`

Used `rfsck-test` framework

Used 17 EXT4 & 12 XFS test images

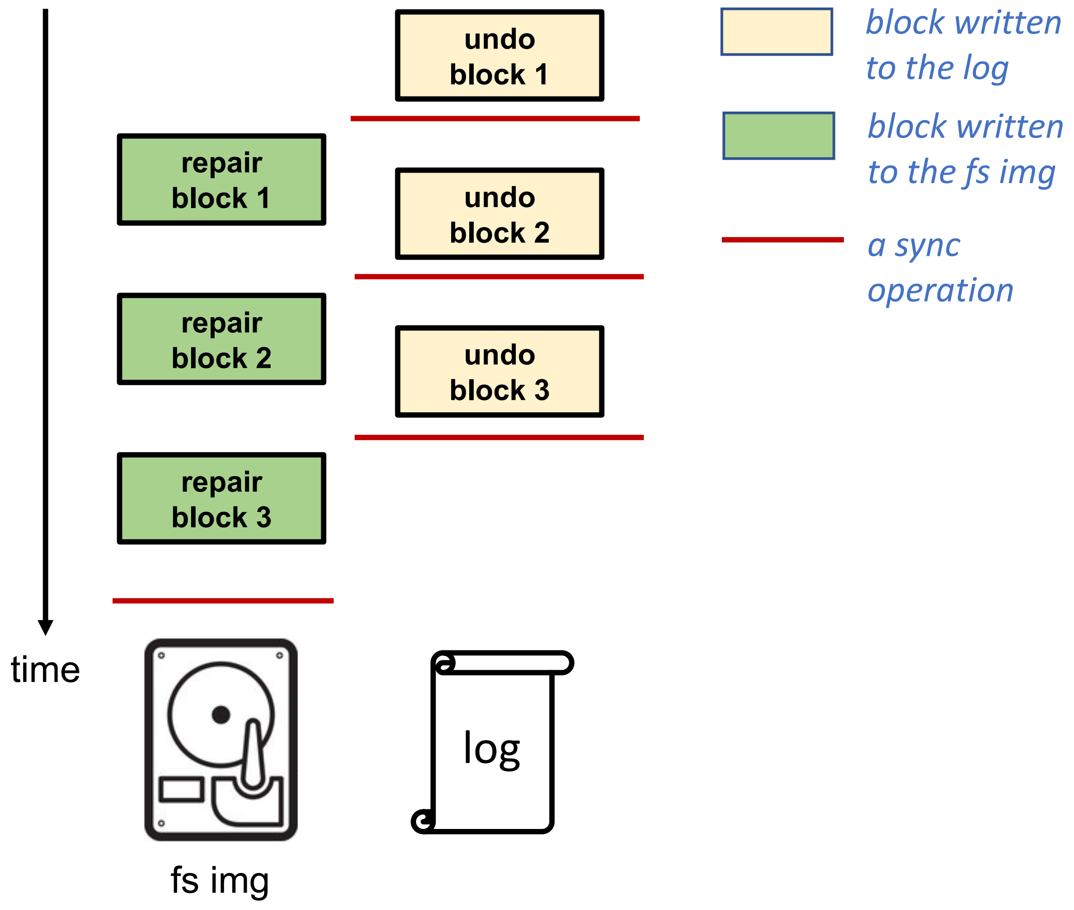
None reported corruption

Conclusion

- Study behavior of existing checkers under faults
  - Interrupted repair may cause irreparable damage
- Build a general logging library "`rfsck-lib`" to address this issue
- Test for robustness using fault injection tool "`rfsck-test`"
- Raise awareness on vulnerabilities in recovery procedures
- Integrate `rfsck-lib` into existing checkers to build more robust checkers

e2fsck-patch: A simple fix

```
/*open undo log*/
undo_open(...){
      open(...);  /*add O_SYNC */
}
...
/*fix 1st inconsistency*/
undo_write_blk64(...){
      /*write to undo log asynchronously*/
      undo_write_tdb(...){
            ...
            pwrite(...);
      }
      /*write to fs image asynchronously*/
      io_channel_write_blk64(...){...}
}
/*fix 2nd, 3rd, ... inconsistencies*/
...

/*sync buffered writes to fs image*/
ext2fs_flush(...){...}
/*close undo log*/
undo_close(...){...}
```



e2fsck-patch

Performance of `rfsck-lib`

- Similar to `rfsck-ext`
- `rfsck-xfs` induces upto 0.8% overhead

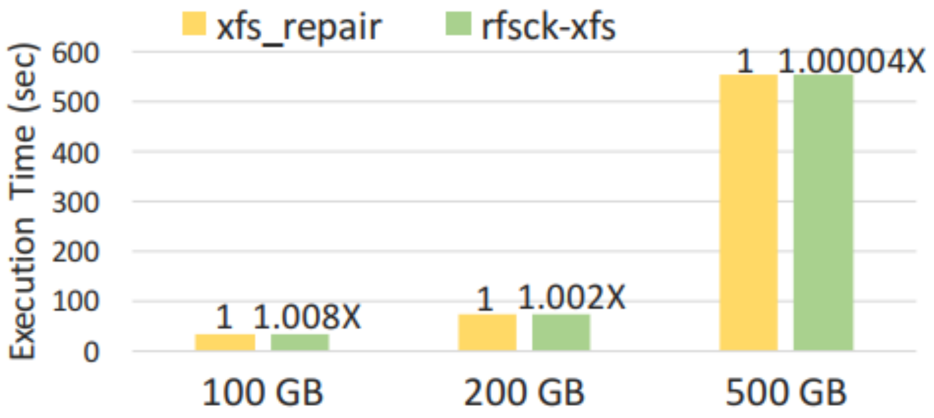


Figure 2: Performance comparison of xfs_repair, rfsck-xfs