Understanding the Fault Resilience of File System Checkers

Om Rameshwar Gatla, Mai Zheng

New Mexico State University

Motivation

TEXAS TECH UNIVERSITY
Information Technology Division

High Performance Computing Center

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.

User home areas have been recovered successfully. At present, no user logins are being permitted while recovery efforts proceed on the remaining Lustre areas. Your understanding and patience are appreciated.

- Multiple power outages
- Recovery procedure was interrupted
- Resulted in severe data loss

Motivation

- Need to understand the behavior of local checkers under faults first

Motivation

- Need to understand the behavior of local checkers under faults first

- **Research Question:**

  Does running the checker after an interrupted-check successfully return the file system to a consistent state? If not, what goes wrong?

Related Work

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

Related Work

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

- One common assumption
  - checkers can finish ***without interruption***

Related Work

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

- One common assumption
  - checkers can finish ***without interruption***

We study behavior of checkers ***with interruptions***

Challenges

- **Challenge #1:**

  How to generate images that contain corruptions and require complex recovery?

- **Challenge #2:**

  How to interrupt recovery systematically?

  - Difficult to simulate system crash and power outages

Methodology

- Generate ***test images*** that contain corruption:
  - Method #1: Test images provided by developers
  - Method #2: Manipulate metadata using file system debugger (e.g., `debugfs`, `xfs_db`)

Methodology

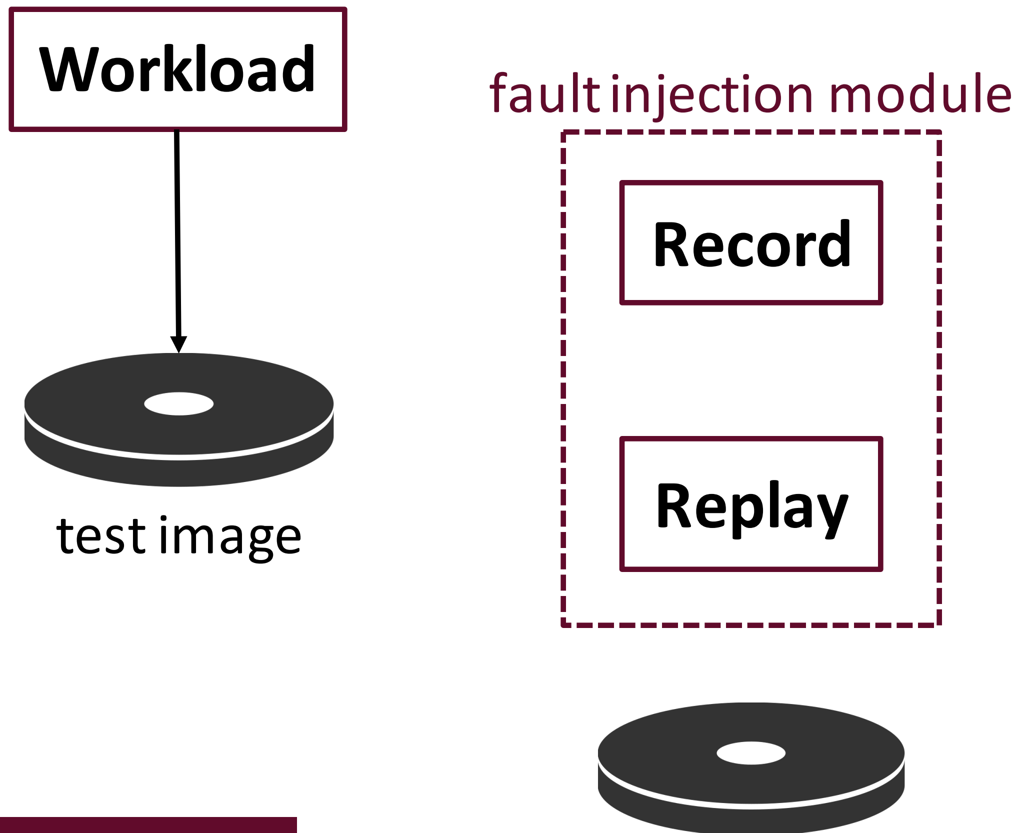
- Develop ***Fault Injection Module*** to generate faults in a systematic and controllable way

Methodology

- Develop ***Fault Injection Module*** to generate faults in a systematic and controllable way
  - Adopt “clean power fault” model[@OSDI’14]
    - Clean termination of I/O stream
      - No reordering
      - Lower bound of failure impact
  - Customize an iSCSI driver to record & replay I/O commands

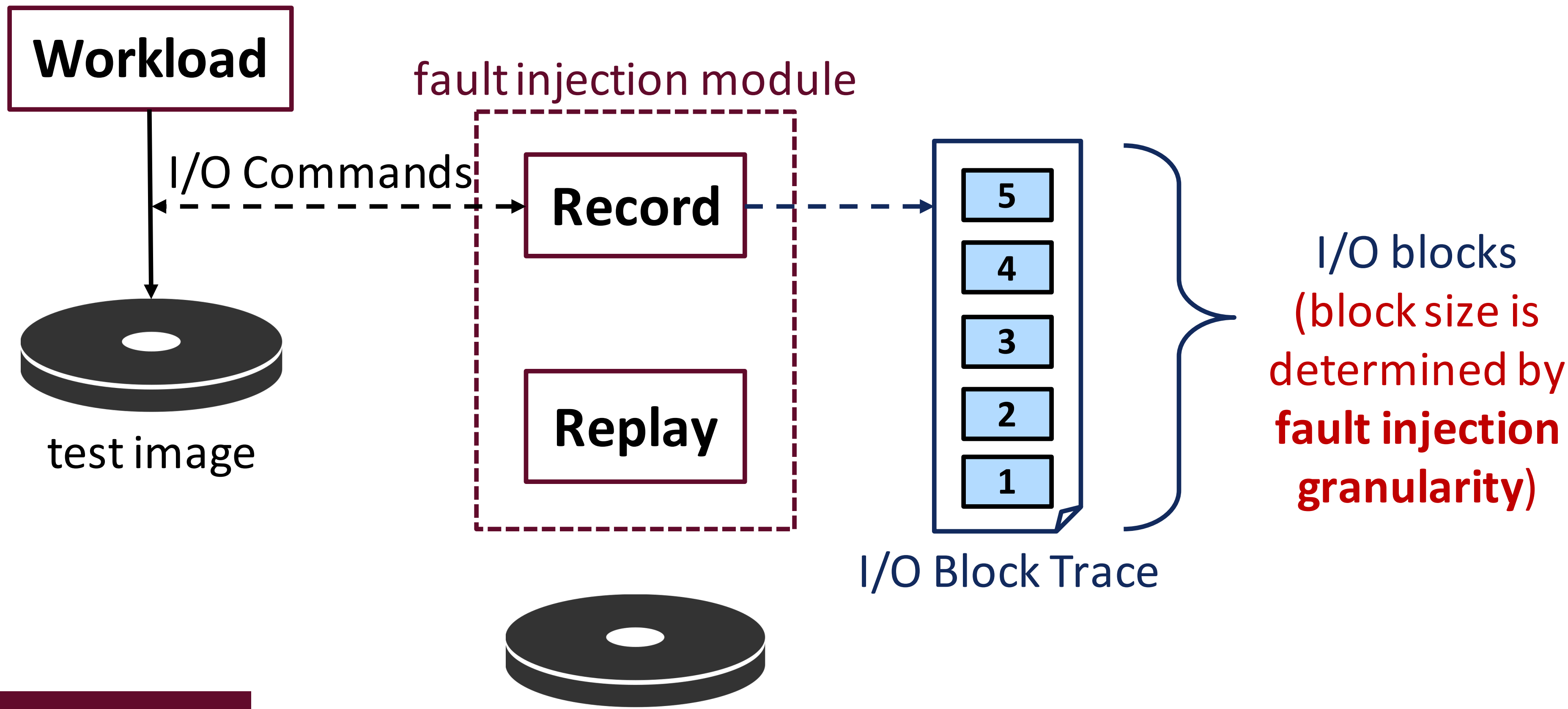
Methodology

- Procedure to emulate clean power fault model



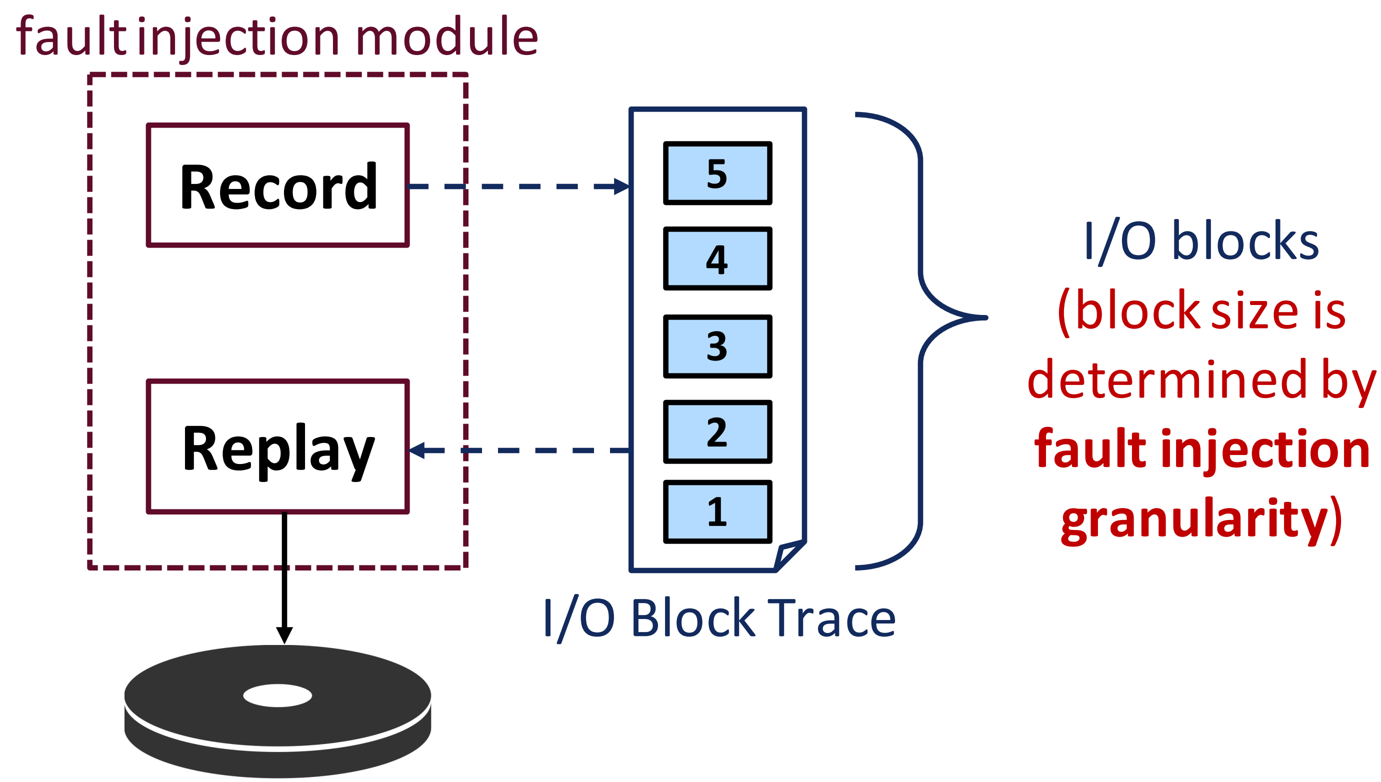
Methodology

- Procedure to emulate clean power fault model



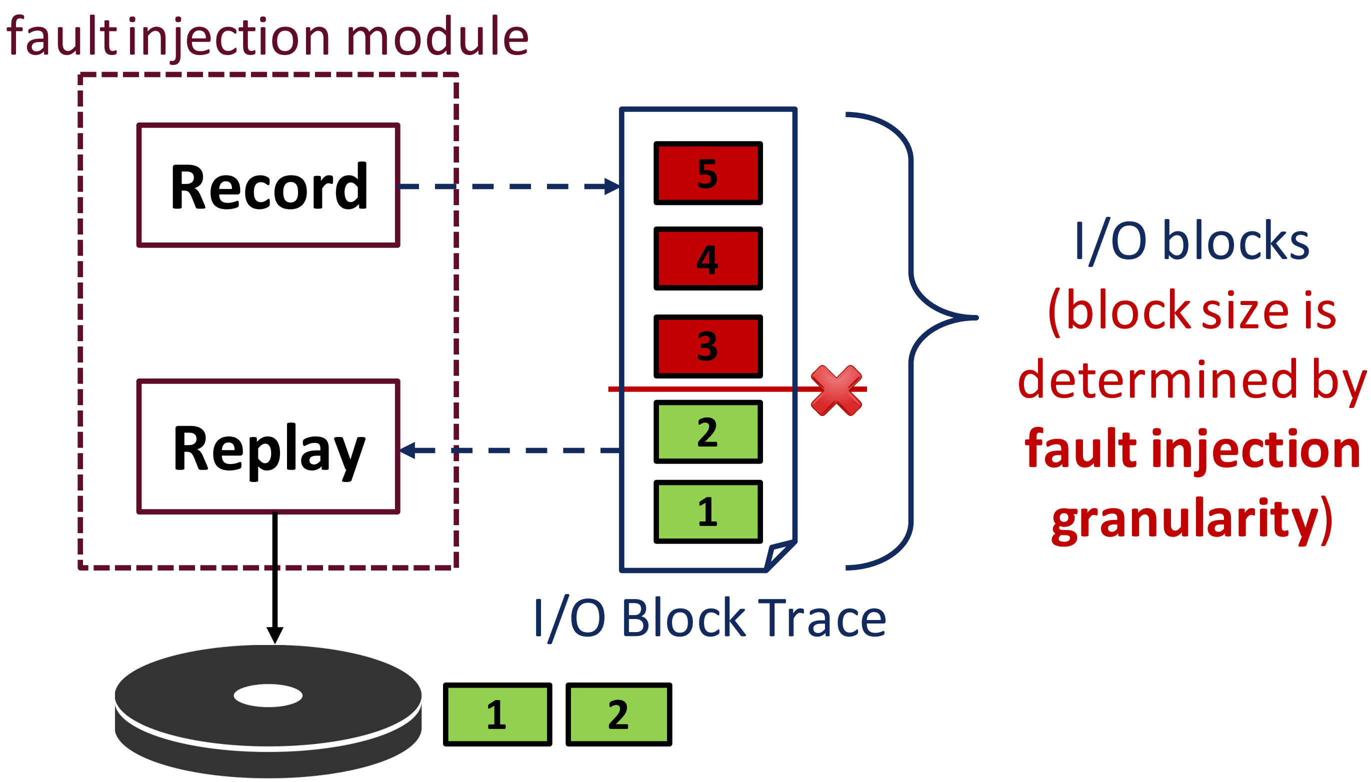
Methodology

- Procedure to emulate clean power fault model



Methodology

- Procedure to emulate clean power fault model



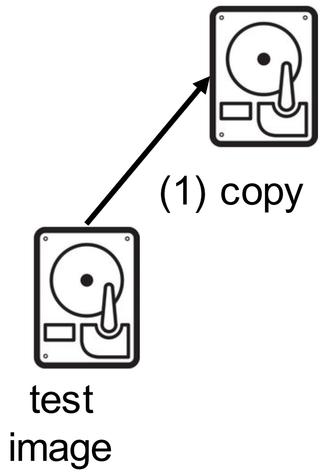
Methodology

- Workflow:

test image

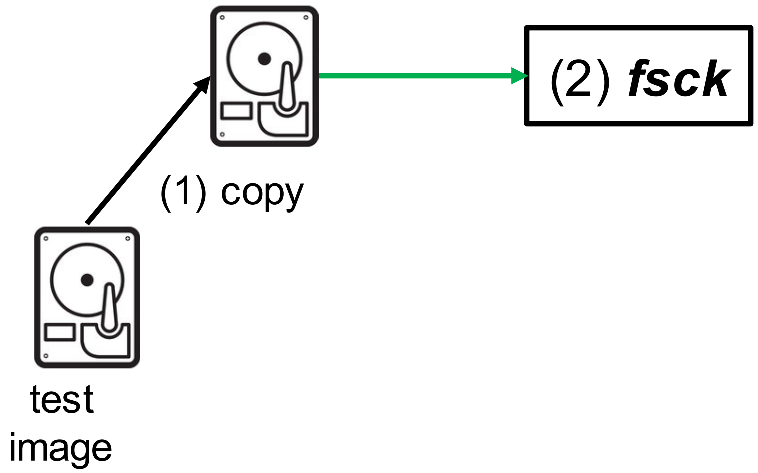
Methodology

- Workflow:



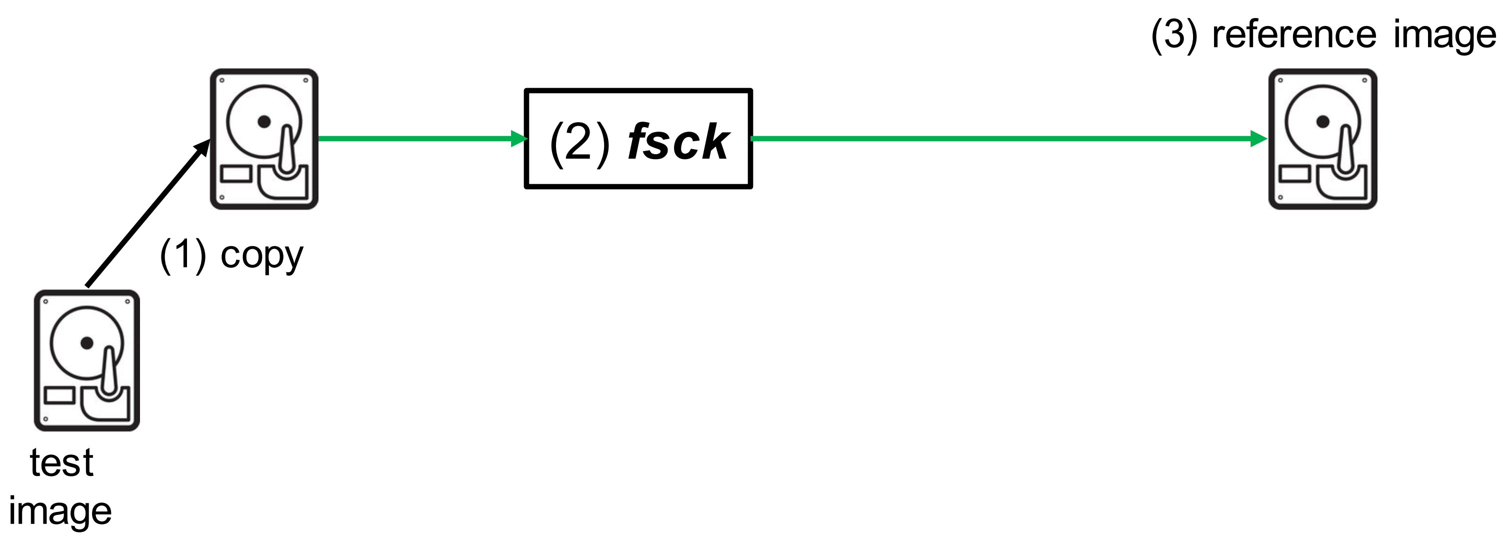
Methodology

- Workflow:



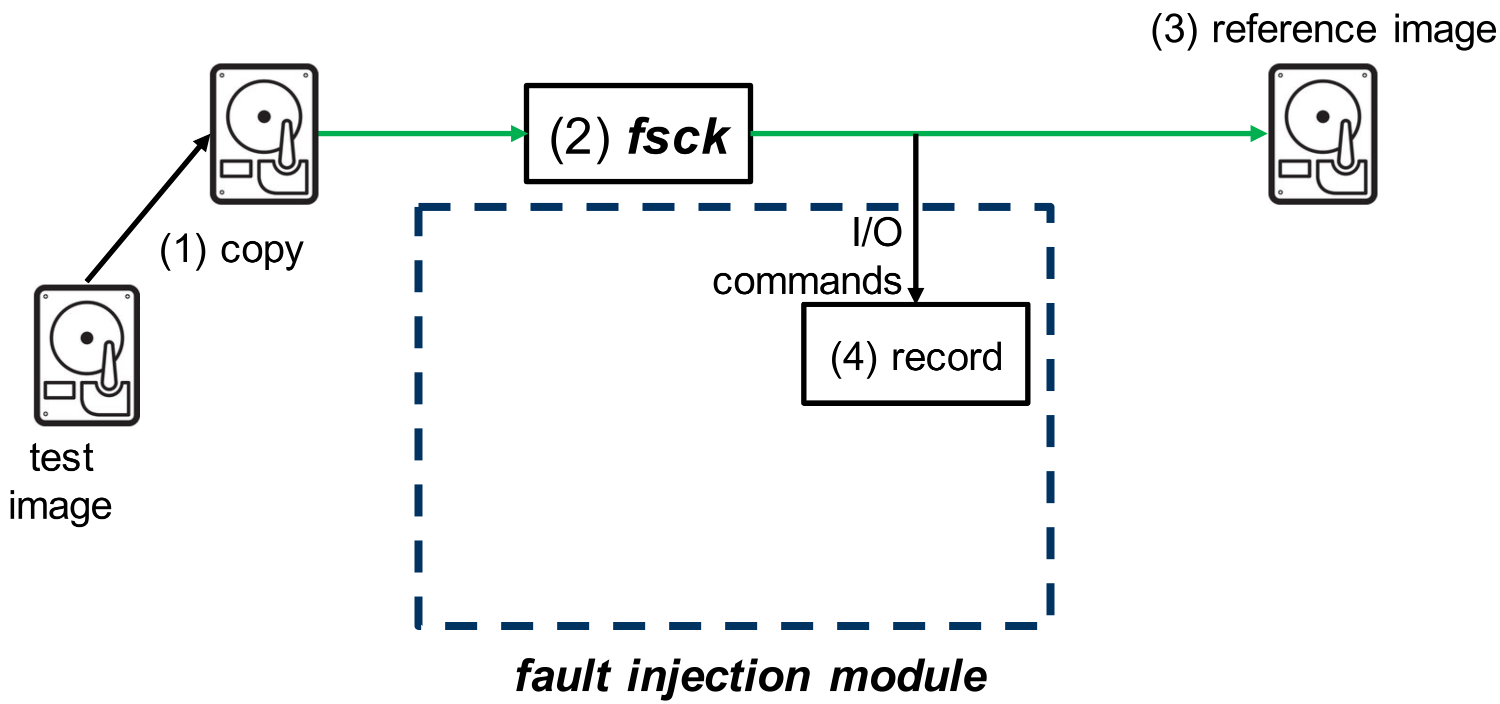
Methodology

- Workflow:



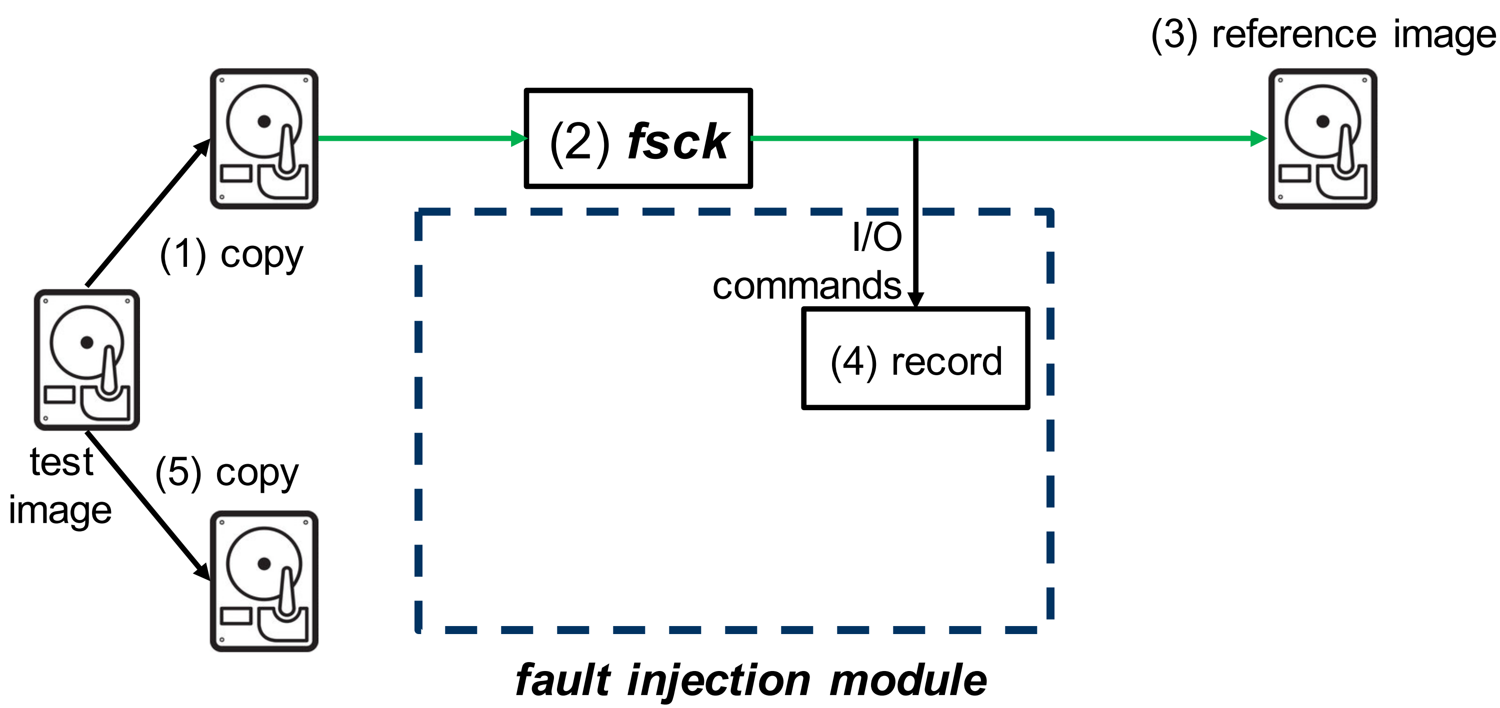
Methodology

- Workflow:



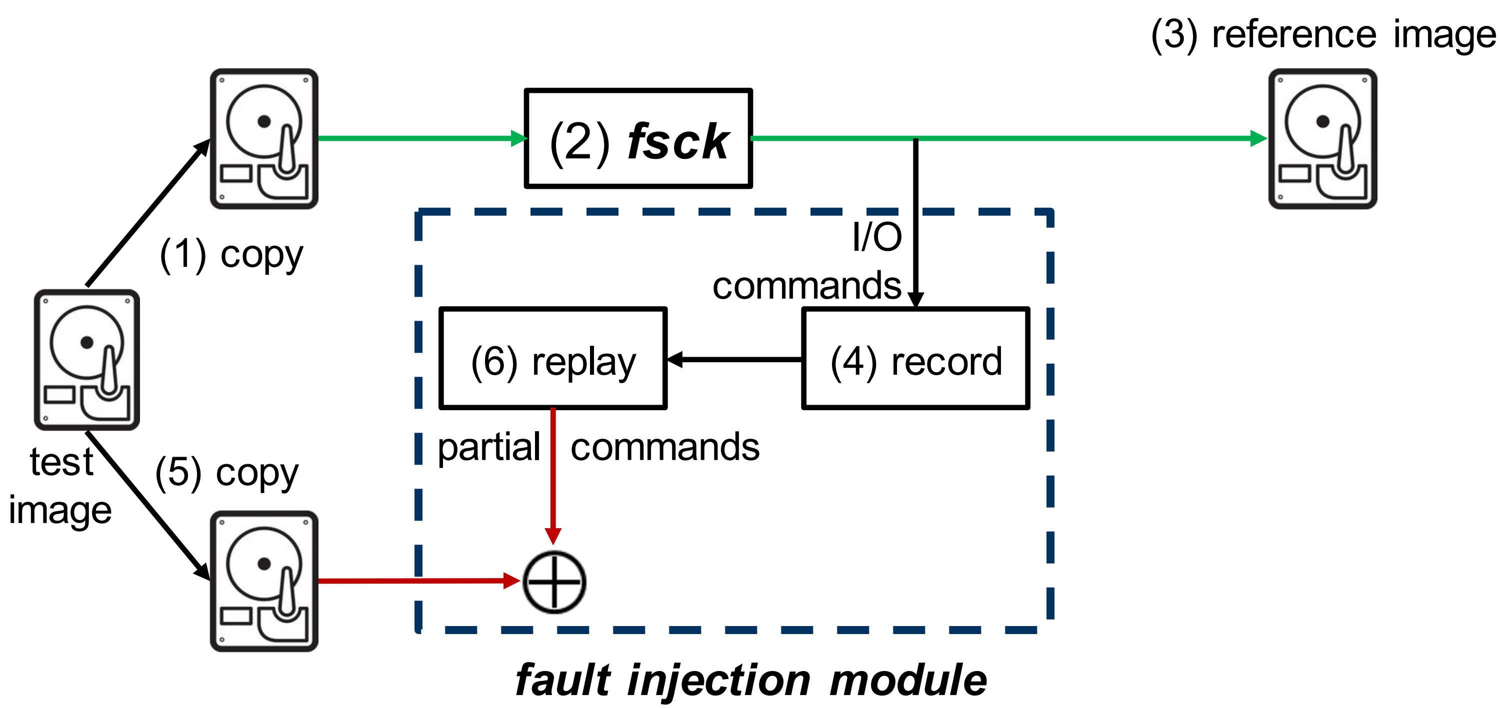
Methodology

- Workflow:



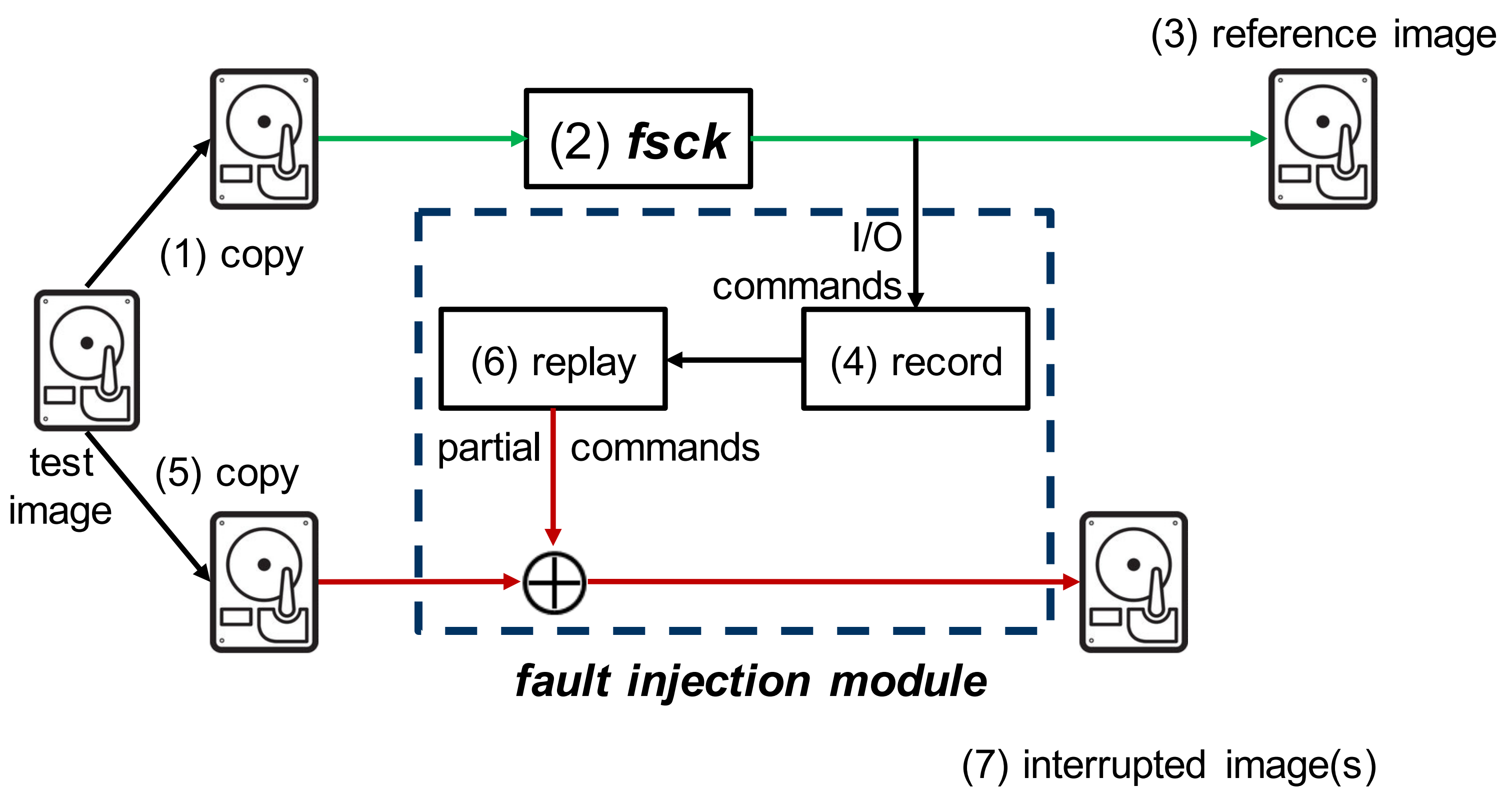
Methodology

- Workflow:



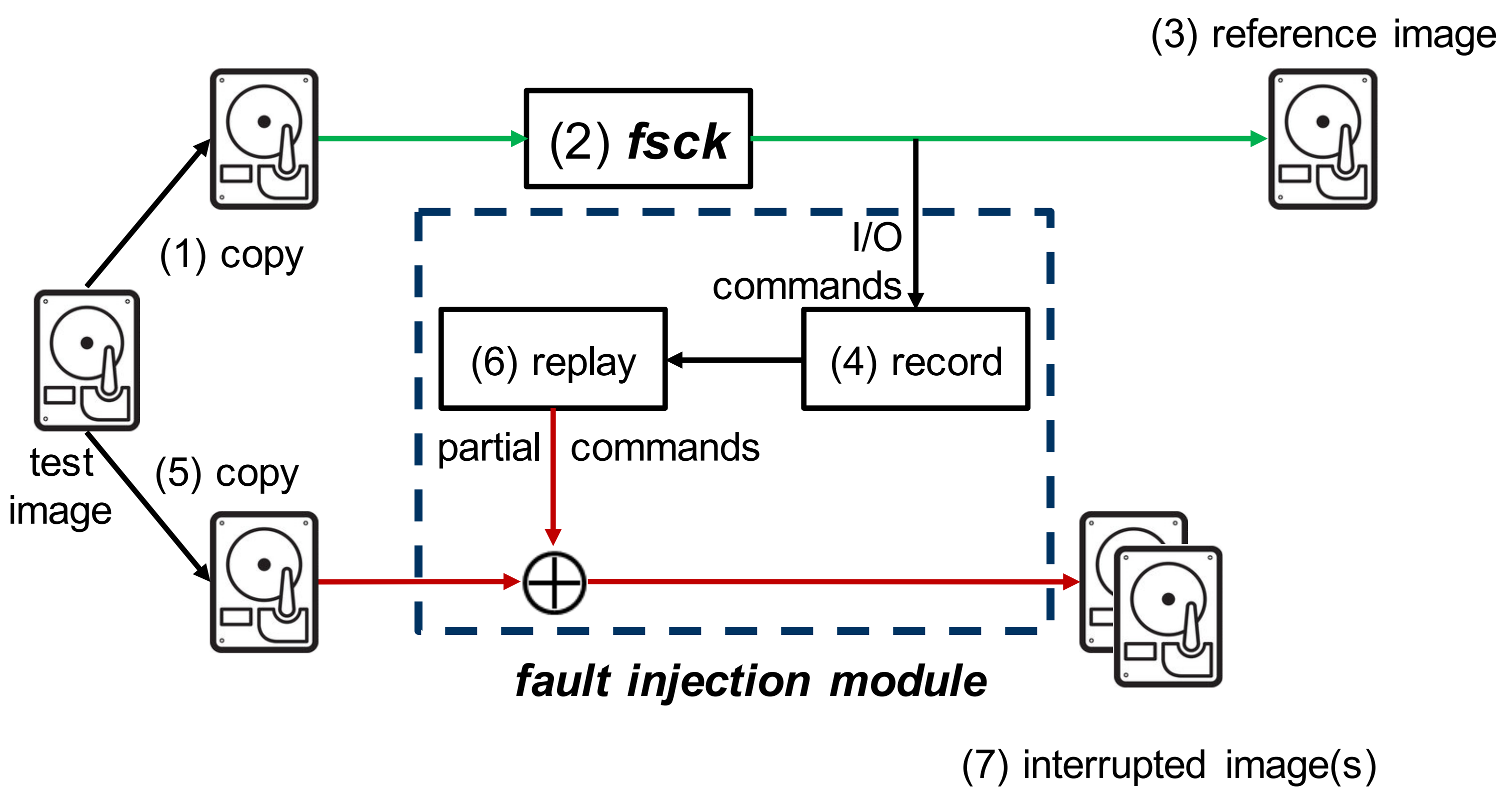
Methodology

- Workflow:



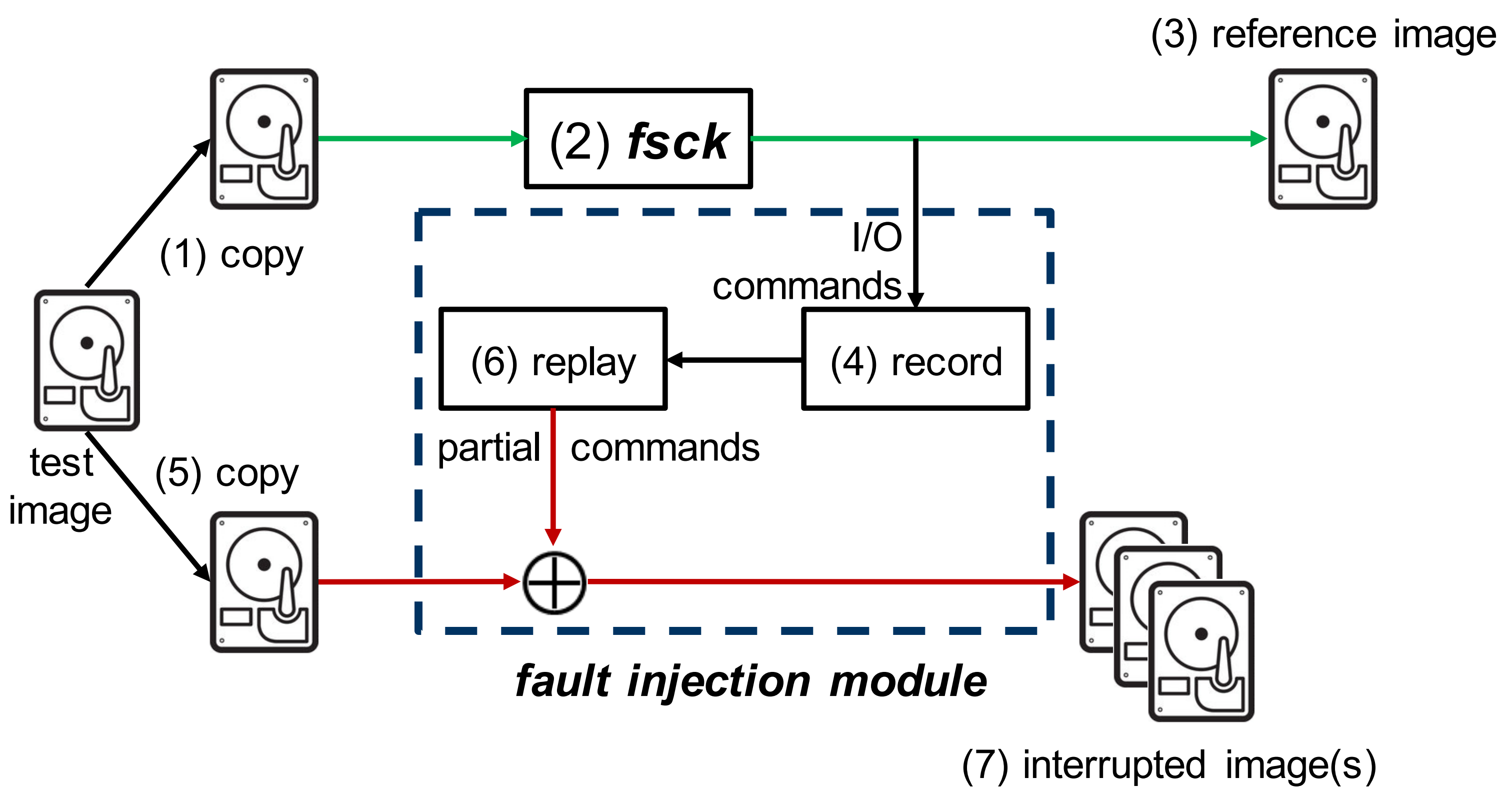
Methodology

- Workflow:



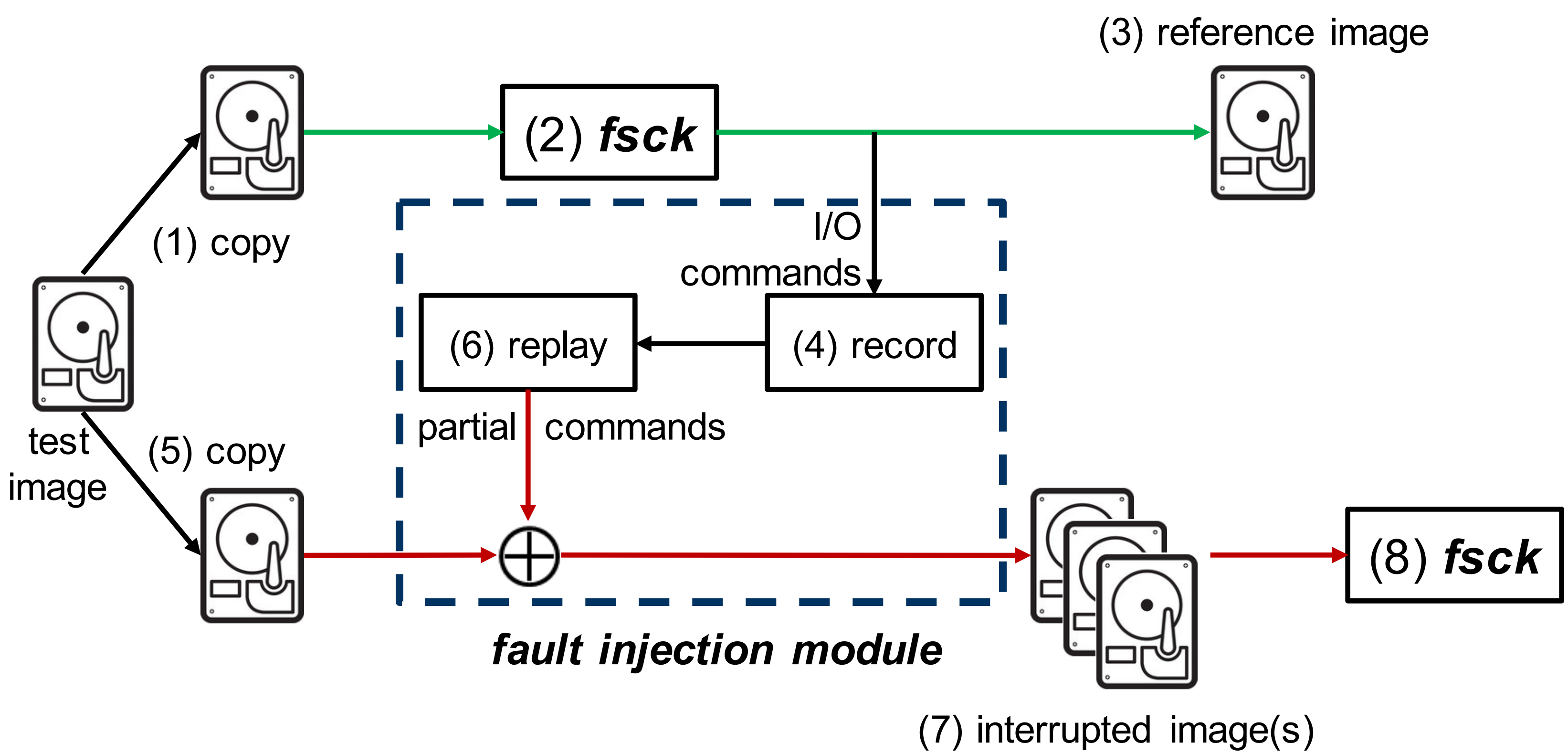
Methodology

- Workflow:



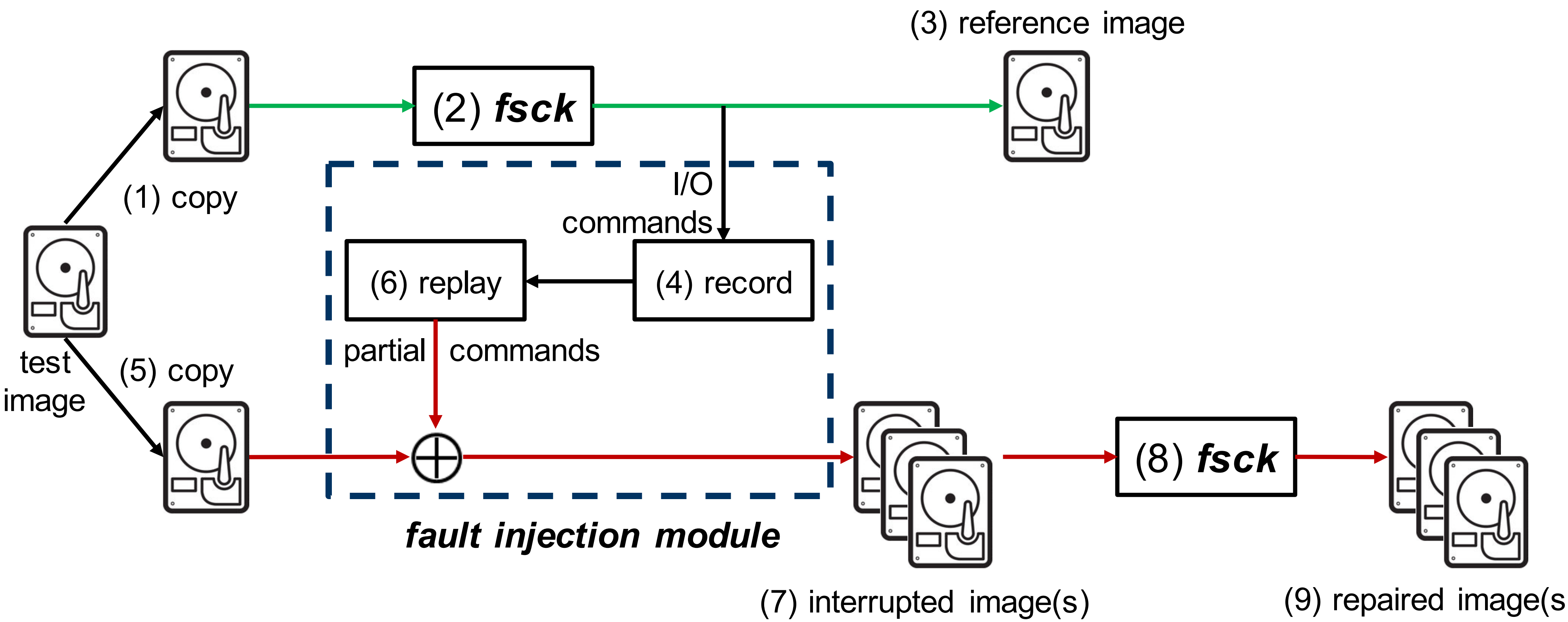
Methodology

- Workflow:



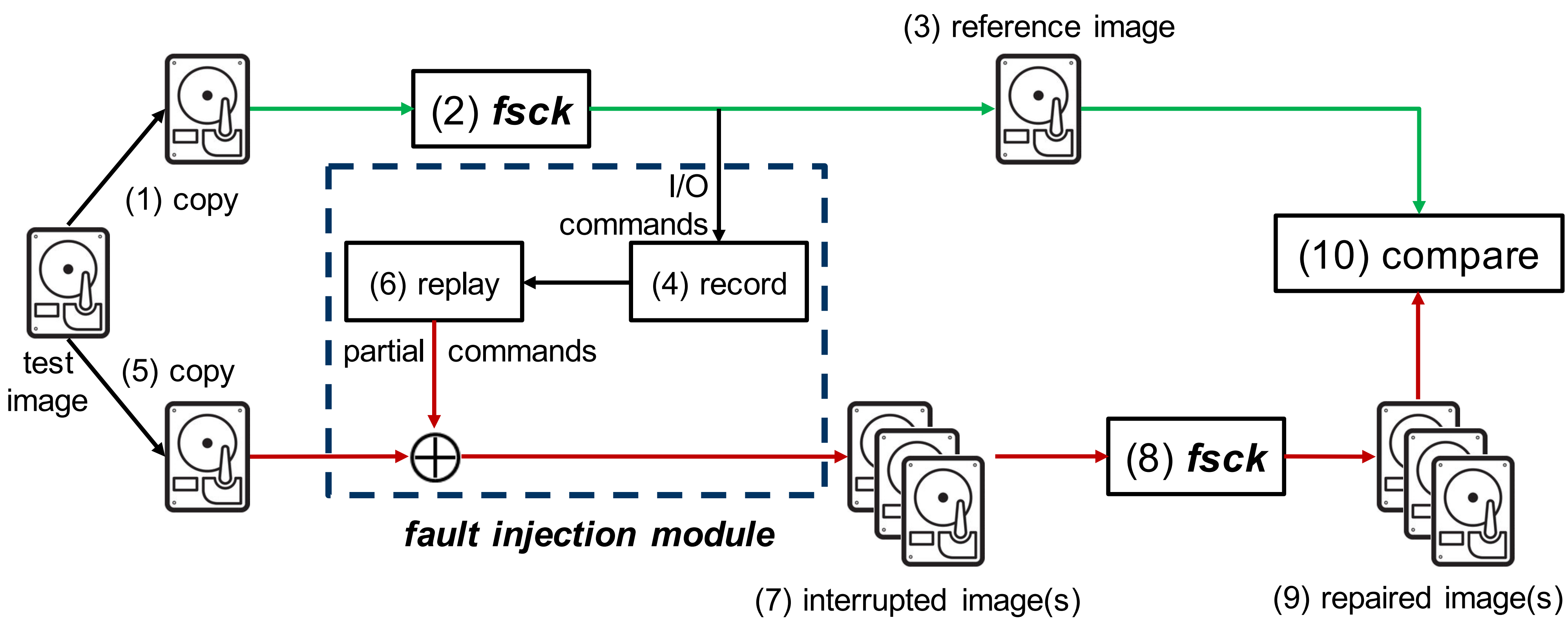
Methodology

- Workflow:



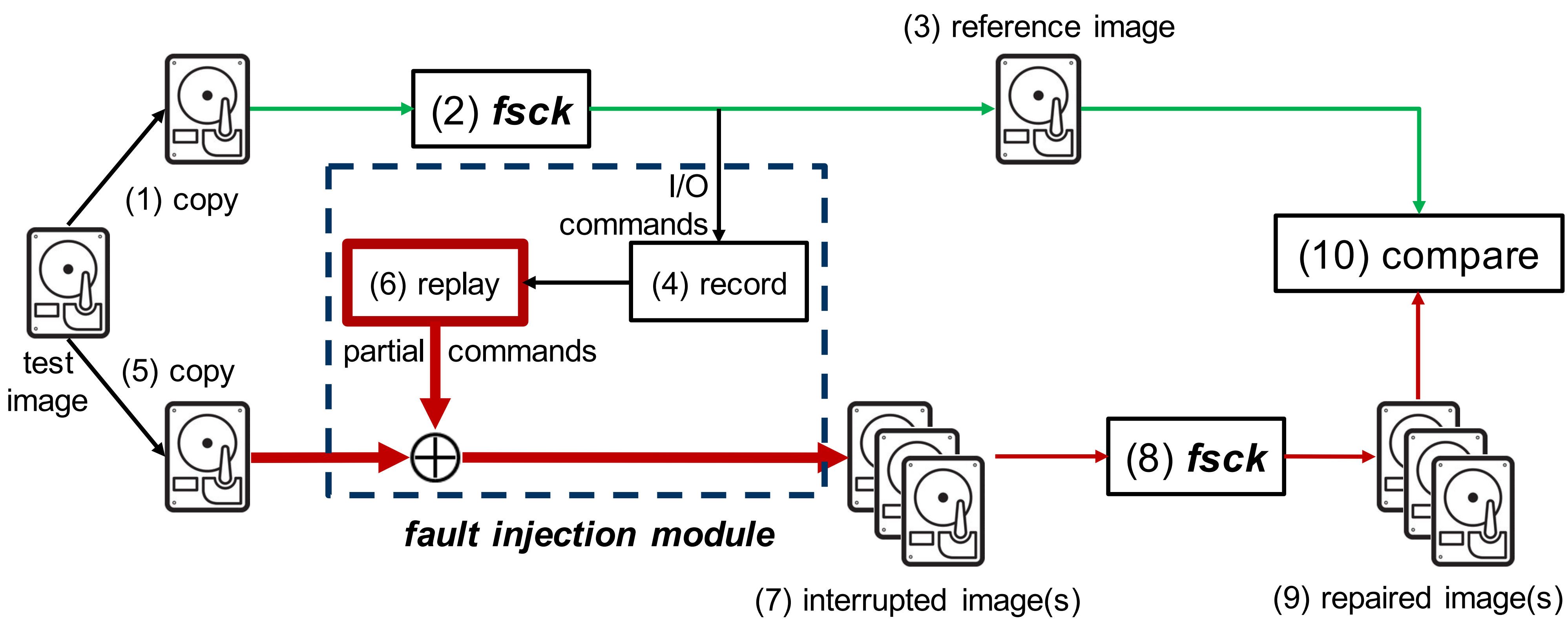
Methodology

- Workflow:



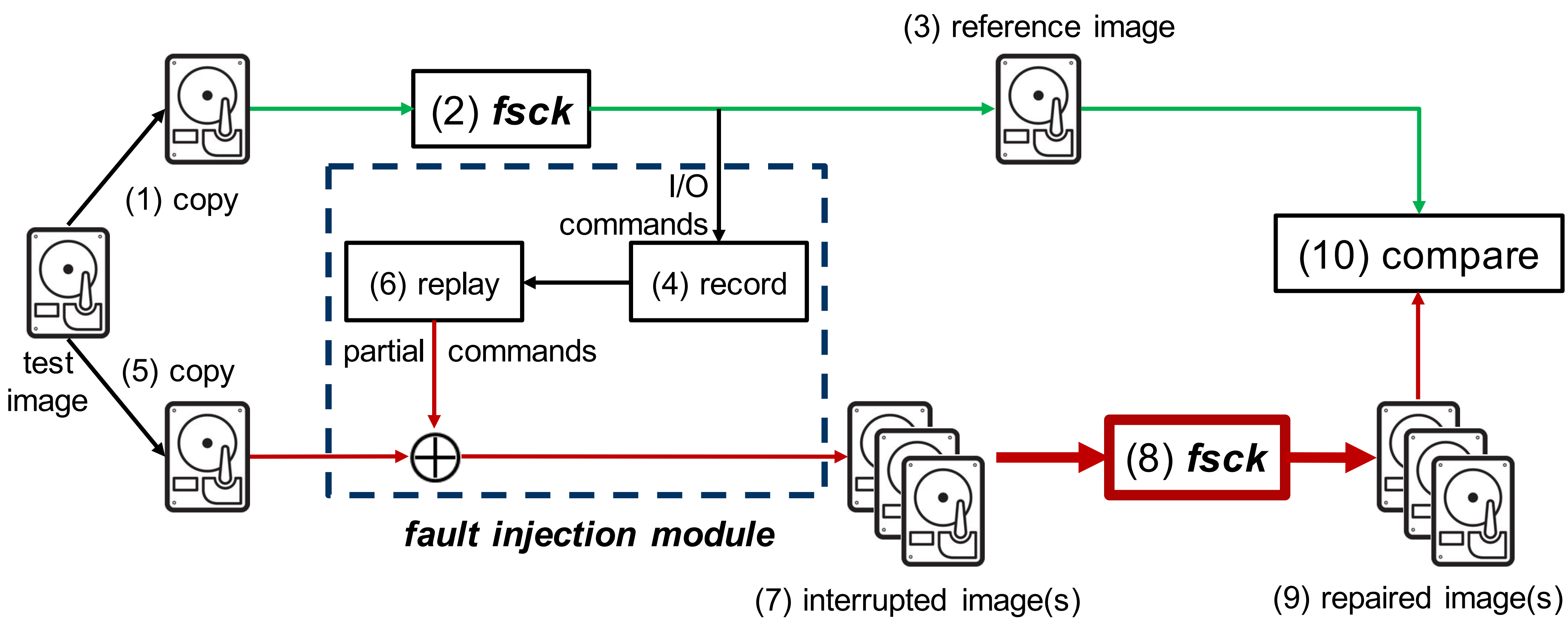
Methodology

- Workflow:



Methodology

- Workflow:



Experimental Results

- Two case studies
  - `e2fsck`: checker for Ext 2/3/4 File Systems
  - `xfs_repair`: checker for XFS File System

Experimental Results

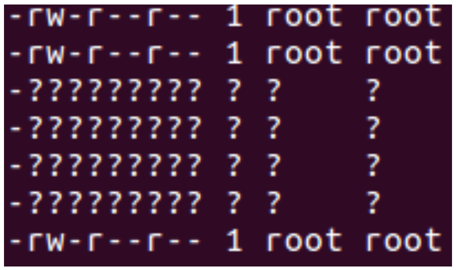
- Two case studies
  - `e2fsck`: checker for Ext 2/3/4 File Systems
  - `xfs_repair`: checker for XFS File System
- Observed 4 types of corruptions:

Un-mountable

File Content Corruption

Misplacement of Files

Others



Experimental Results

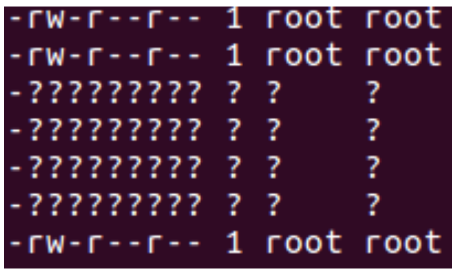
- Two case studies
  - `e2fsck`: checker for Ext 2/3/4 File Systems
  - `xfs_repair`: checker for XFS File System
- Observed 4 types of corruptions:

Un-mountable

File Content Corruption

Misplacement of Files

Others



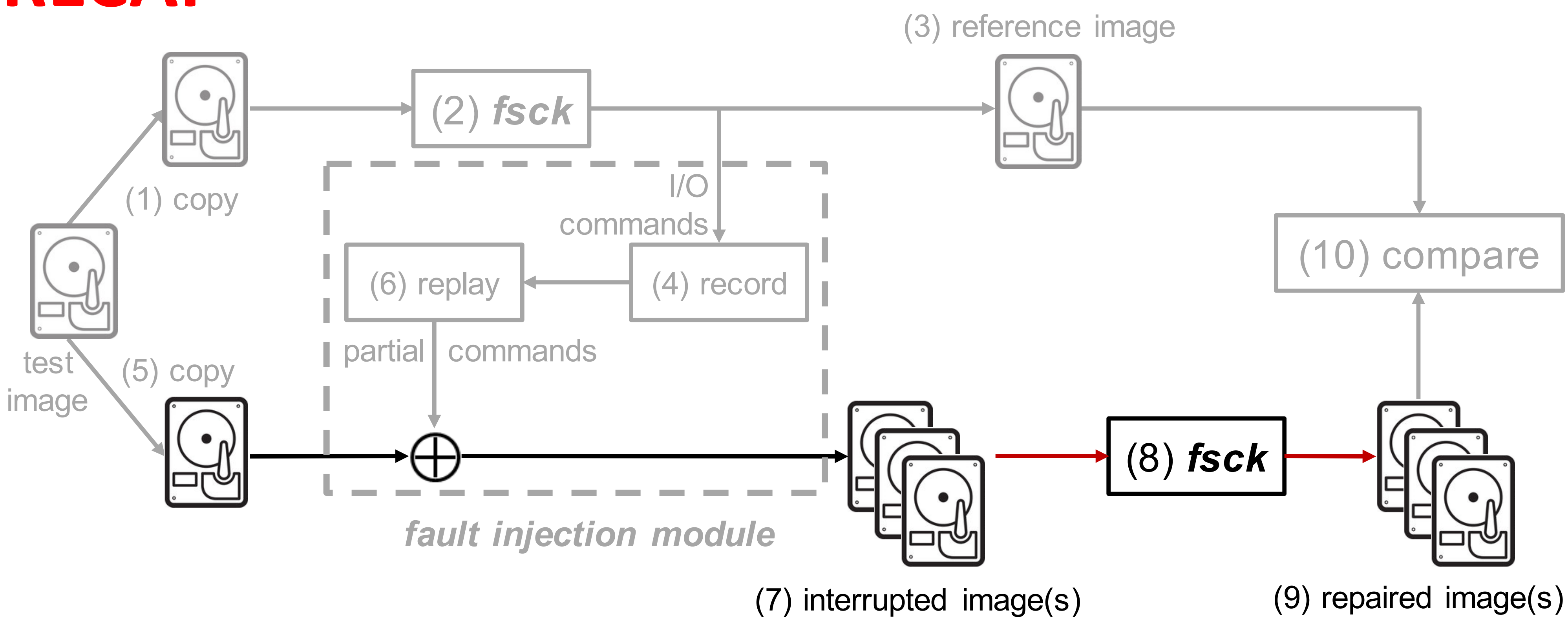
Cannot be fixed by another run of fsck

Case Study: `e2fsck`

- Used 175 test images from `e2fsprogs`
- Block size of all images is 1KB
- Fault injected at two granularities: 512B and 4KB

Methodology

RECAP



1 test image ➡ many interrupted/repaired images

Case Study: `e2fsck`

Fault Injection Granularities	Total number of Test Images	Total number of repaired images
512 B	175	25,062
4 KB	175	3,915

Table 1: Number of test images and repaired images generated under two fault injection granularities

Case Study: `e2fsck`

Fault Injection Granularities	Total number of Test Images	Total number of repaired images
512 B	175	25,062
4 KB	175	3,915

Table 1: Number of test images and repaired images generated under two fault injection granularities

Case Study: `e2fsck`

Fault Injection Granularities	Total number of Test Images	Total number of repaired images
512 B	175	25,062
4 KB	175	3,915

Table 1: Number of test images and repaired images generated under two fault injection granularities

Case Study: `e2fsck`

Fault Injection Granularities	Total number of Test Images	Total number of repaired images
512 B	175	25,062
4 KB	175	3,915

Table 1: Number of test images and repaired images generated under two fault injection granularities

Case Study: `e2fsck`

Fault Injection Granularities	Total number of Test Images	Total number of repaired images
512 B	175	25,062
4 KB	175	3,915

Table 1: Number of test images and repaired images generated under two fault injection granularities

Fault Injection Granularities	Number of images reporting corruption	
	Test Images	Repaired Images
512 B	34	240
4 KB	17	37

Table 2: Number of test images and repaired images reporting corruption under two fault injection granularities

Case Study: `e2fsck`

Fault Injection Granularities	Total number of Test Images	Total number of repaired images
512 B	175	25,062
4 KB	175	3,915

Table 1: Number of test images and repaired images generated under two fault injection granularities

Fault Injection Granularities	Number of images reporting corruption	
	Test Images	Repaired Images
512 B	34	240
4 KB	17	37

Table 2: Number of test images and repaired images reporting corruption under two fault injection granularities

Case Study : `e2fsck`

Corruption Types	512 B	4 KB
Un-mountable	41	3
File Content Corruption	107	10
Misplacement of files	82	23
Others	10	1
Total	240	37

Table 3: Classification of corruptions observed on repaired images

Case Study : `e2fsck`

Corruption Types	512 B	4 KB
Un-mountable	41	3
File Content Corruption	107	10
Misplacement of files	82	23
Others	10	1
Total	240	37

Table 3: Classification of corruptions observed on repaired images

Is it solely caused by asynchronous writes?

- Existing implementation uses asynchronous updates
  - most updates are buffered in memory
  - flush them only at the end of last pass
- Does not guarantee ordering and atomicity

Is it solely caused by asynchronous writes?

- We change the code to enforce synchronous writes
  - **Method 1:** Add `O_SYNC` flag
  - **Method 2:** Invoke `ext2fs_flush()` to flush changes after each pass (5 passes in total)

Enforcing Synchronous Writes

- Surprisingly, the results are worse:

Enforcing Synchronous Writes

- Surprisingly, the results are worse:

BEFORE

Fault Injection Granularities	Number of images reporting corruption	
	Test Images	Repaired Images
512 B	34	240
4 KB	17	37

Table 4: Number of test images and repaired images reporting corruption under two fault injection granularities

Enforcing Synchronous Writes

- Surprisingly, the results are worse:

BEFORE

Fault Injection Granularities	Number of images reporting corruption	
	Test Images	Repaired Images
512 B	34	240
4 KB	17	37

Table 4: Number of test images and repaired images reporting corruption under two fault injection granularities

AFTER

Synchronization Methods	Number of images reporting corruption	
	Test Images	Repaired Images
Sync each write	**45**	**223**
Sync after each pass	**45**	**243**

Table 5: Number of test images and repaired images reporting corruption after enforcing synchronous updates.

Enforcing Synchronous Writes

- Surprisingly, the results are worse:

BEFORE

Fault Injection Granularities	Number of images reporting corruption	
	Test Images	Repaired Images
512 B	34	240

There is strong dependency among updates/passes

AFTER

Sync each write	45	223
Sync after each pass	45	243

Table 5: Number of test images and repaired images reporting corruption after enforcing synchronous updates.

Can Undo Log Handle Interruptions?

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

Can Undo Log Handle Interruptions?

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

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

Can Undo Log Handle Interruptions?

- Modify the existing fault-injection framework to test `e2fsck` with undo log enabled:
  - Add another block device for undo log
  - Add record & replay for undo log

Can Undo Log Handle Interruptions?

- Modify the existing fault-injection framework to test `e2fsck` with undo log enabled:
  - Add another block device for undo log
  - Add record & replay for undo log

- Surprisingly the results are similar:
  - No ordering of writes b/w undo log and block device

Conclusion

- Methodology to study the behavior of file system checker under emulated faults
- Does running the checker after an interrupted-check successfully return the file system to a consistent state? **NO**
- If not, what goes wrong?
  - Strong dependencies among updates/passes, resulting in severe corruption under faults

Make Recovery Procedures Resilient to Faults

Future work

- Build a resilient file system checker
- Port this methodology to other procedures
  - E.g., system updates, etc.

THANK YOU

THANK YOU

QUESTIONS?