Crash Consistency Validation Made Easy

Yanyan Jiang, Chang Xu, Xiaoxing Ma, Jian Lu (Nanjing University, China)
Haicheng Chen, Feng Qin (Ohio State University, USA)

November 15 @ FSE 2016

Outline

- Crash consistency validation
  - definition, examples, and problem formulation

- How to make it easy?
  - expected/crash snapshots, editing distance, test amplification, and C^3

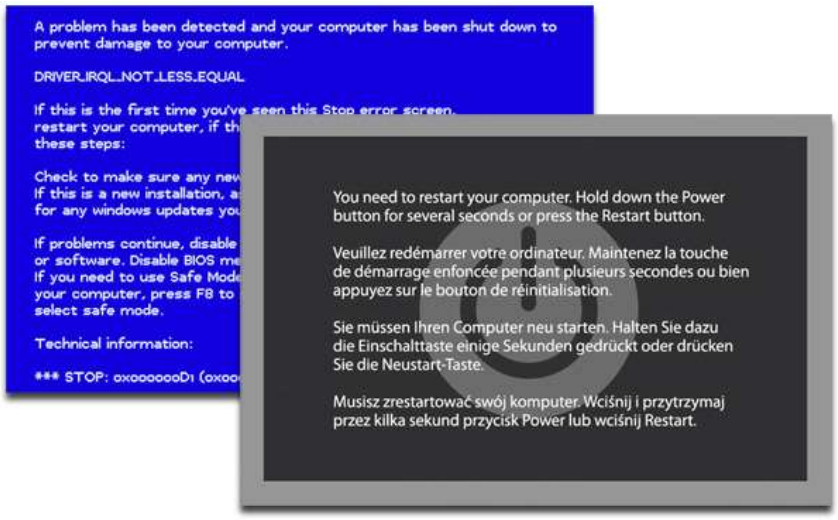
Crash Consistency: Definition

Crash Consistency

- "Ensure that the file system keeps the on-disk image in a *reasonable state* given that *crashes* can occur at arbitrary points in time." [Remzi14]

Crash Consistency Matters: Crash is Inevitable

- Any crash consistency bug will eventually manifest



Software bug

Power loss

Crash Consistency Matters: Crash May Cause Severe Consequences

- When you are saving your paper draft…

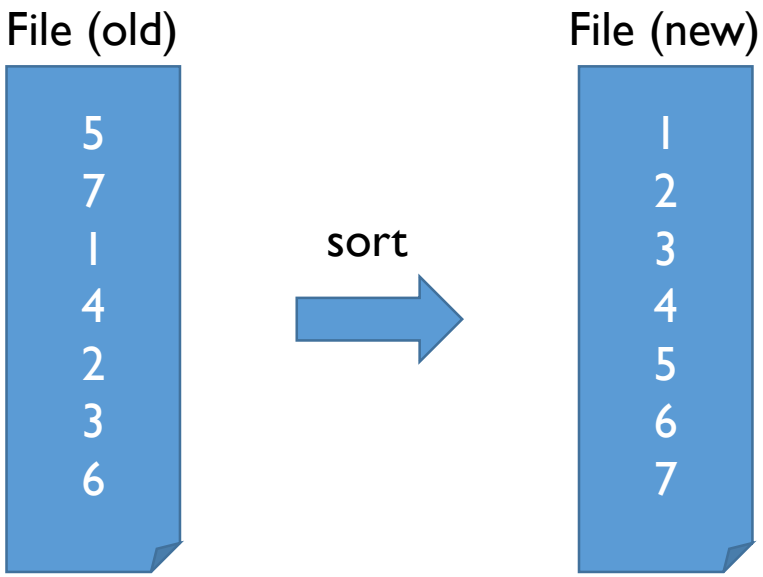
… and your favorite text editor destroys it[1]

[1] May happen in TeXstudio 2.10.8 and TeXmaker 4.5.

Crash Consistency Bug: Examples

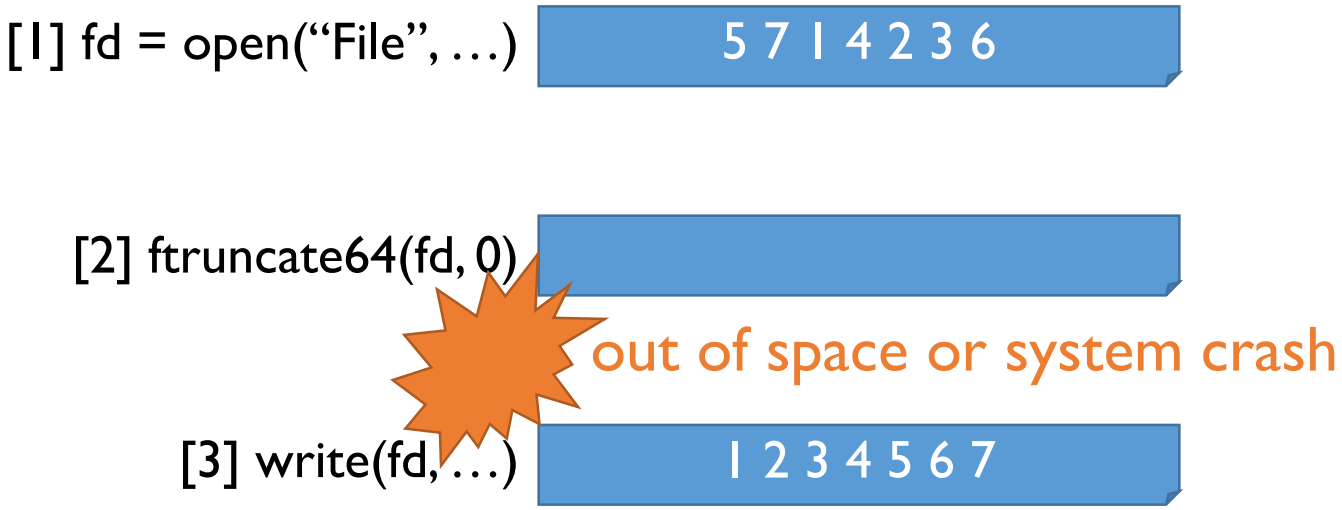
Example 1: GNU Coreutils Sort

- Read lines and output sorted lines
  - there is an option `-o` for backward compatibility
  - in-place sorting (`sort File -o File`)



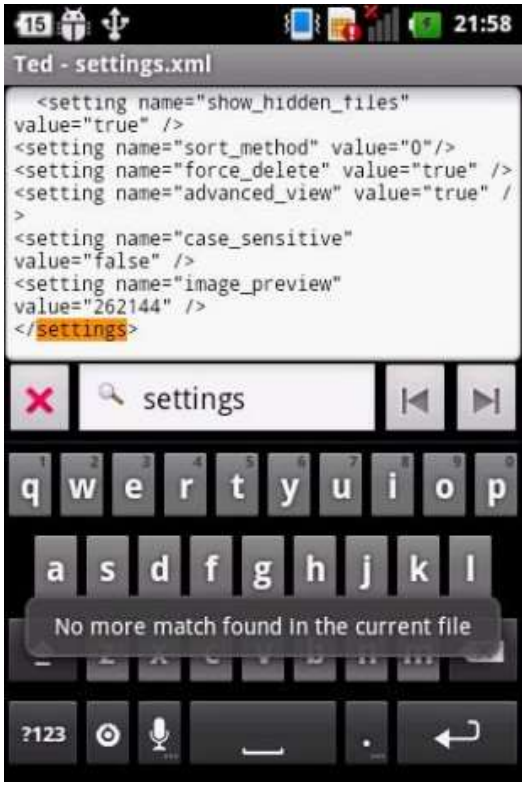
Problematic In-place Sorting

- The actual sorting procedure



Example 2: Ted Text Editor

- An open-source text editor for Android
  - with minimal functionality: viewing, editing, and searching a text file
- The developer is aware of the crash consistency issue



File Operations: Developer's View

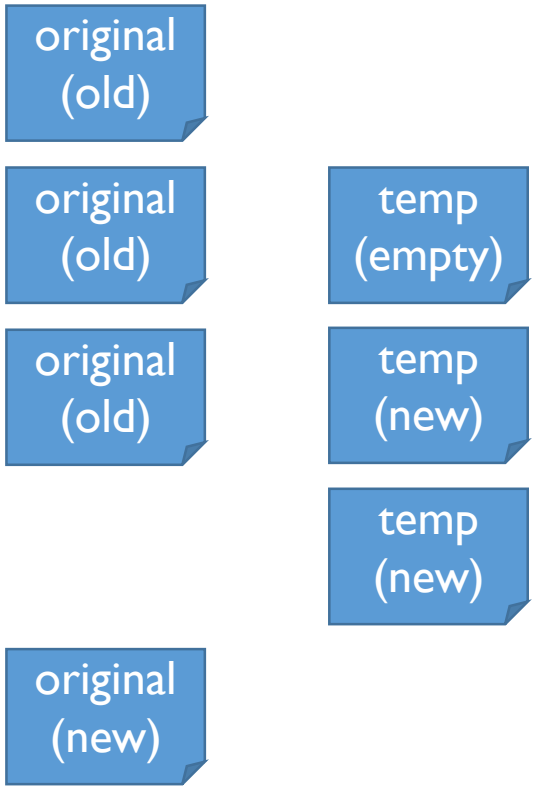
(initial state)

[1] create a temporary file

[2] write data to the temporary file

[3] delete the original file

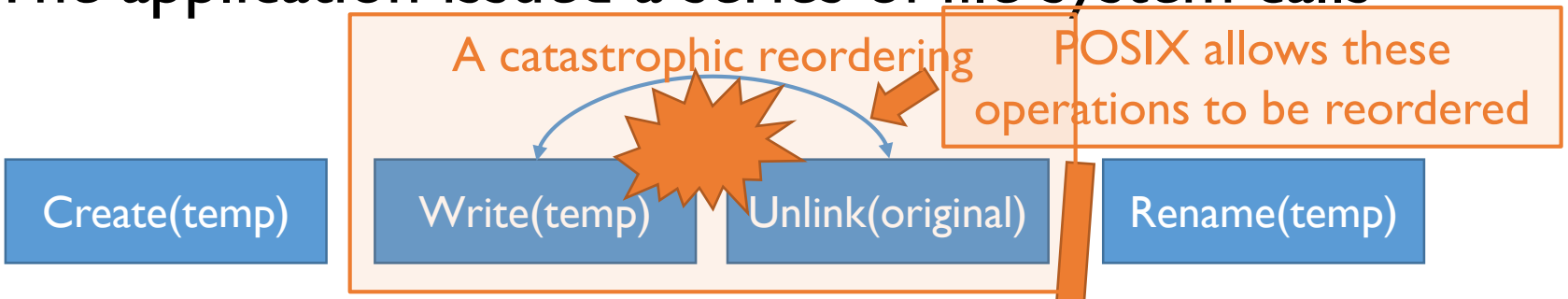
[4] rename the temporary file



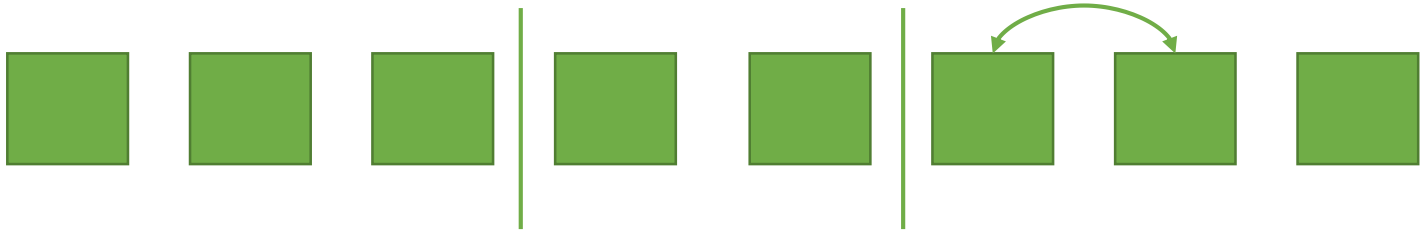
at least one of the original or the temporary file survives the crash

File Operations: Operating System's View

- The application issued a series of file system calls

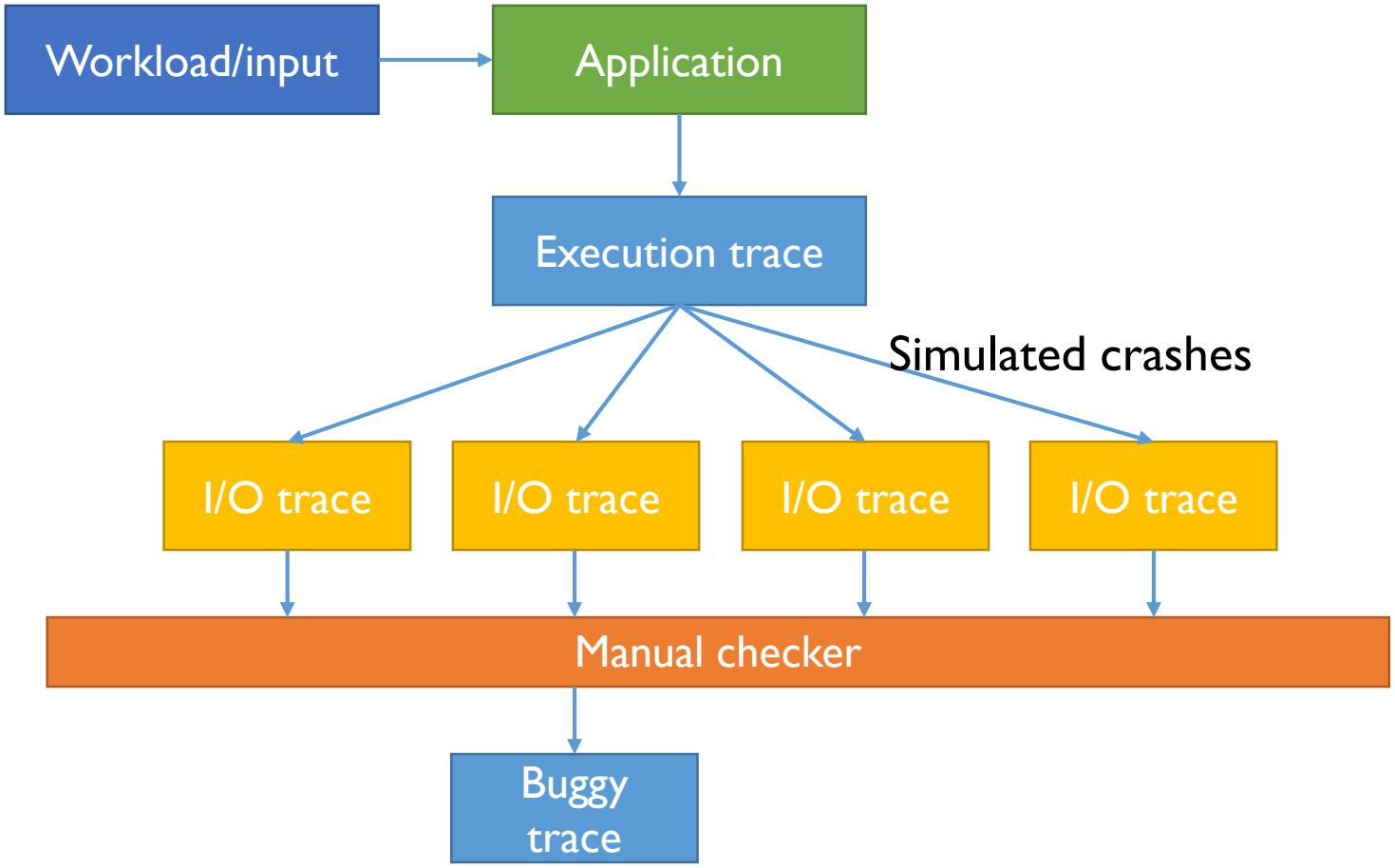


- The file system implementation translates the system call sequence to I/O operations



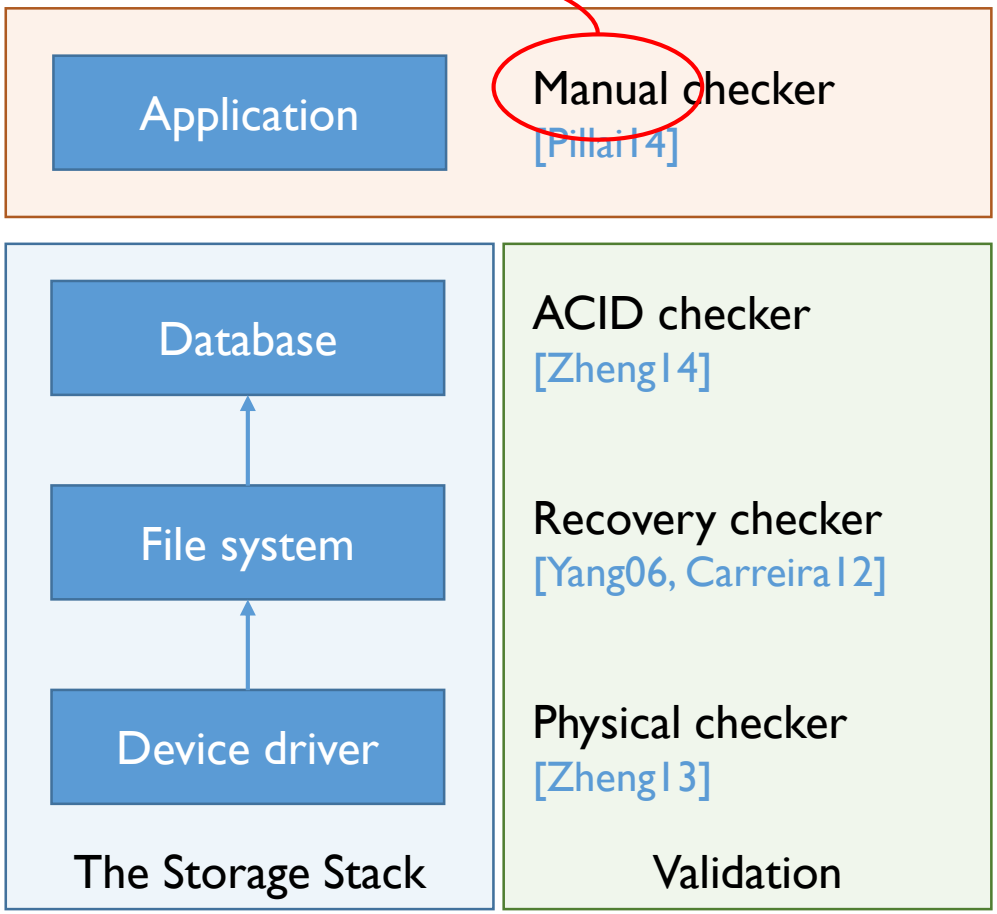
Problem Formulation: Validating Crash Consistency

Crash Consistency Validation: Framework



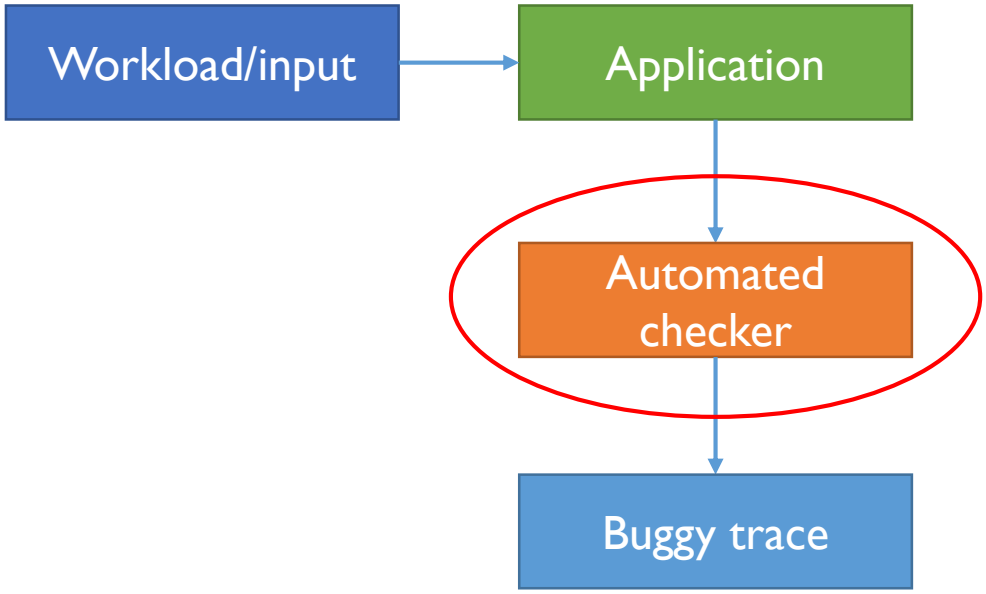
Crash Consistency Validation in the Storage Stack

Can we have a general definition of crash consistency for application software?



Problem Formulation

- Design a *generic test oracle* for validating application software's crash consistency



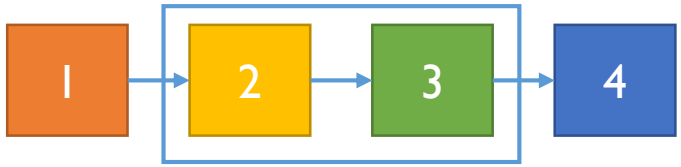
Crash Consistency Validation Made Easy

Overview

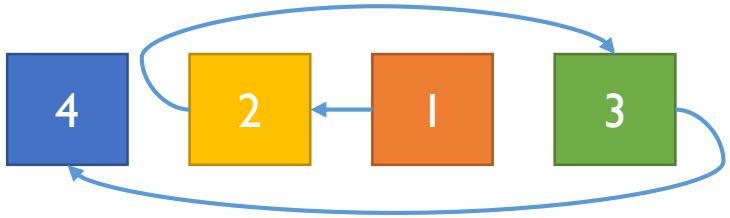
- A three-step approach
  - find expected "correct" file system snapshots (ES)
  - simulate system crash file system snapshots (CS)
  - consistency of a CS ⇔ can be "transformed" to an ES
- Plus a test amplification
  - insert benign file system synchronization operations

The Root Cause of Crash Consistency Bugs

- The developer expects that file system operations are *atomic* and *persistent*

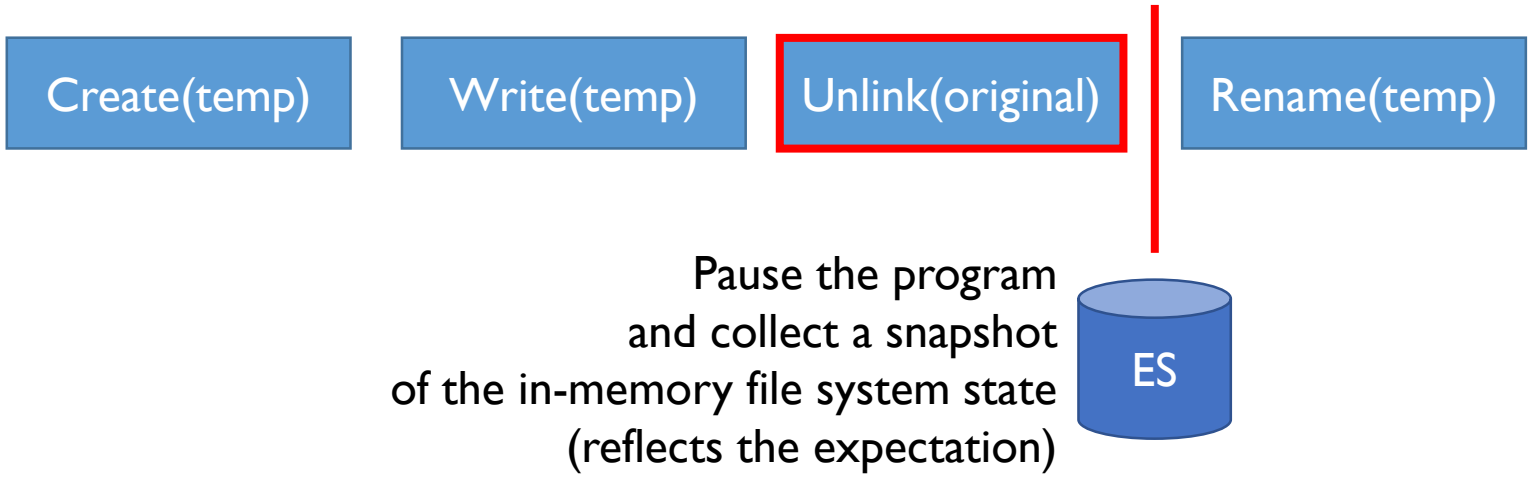


- But the file system implementation provides no such guarantee due to buffering, journaling, …



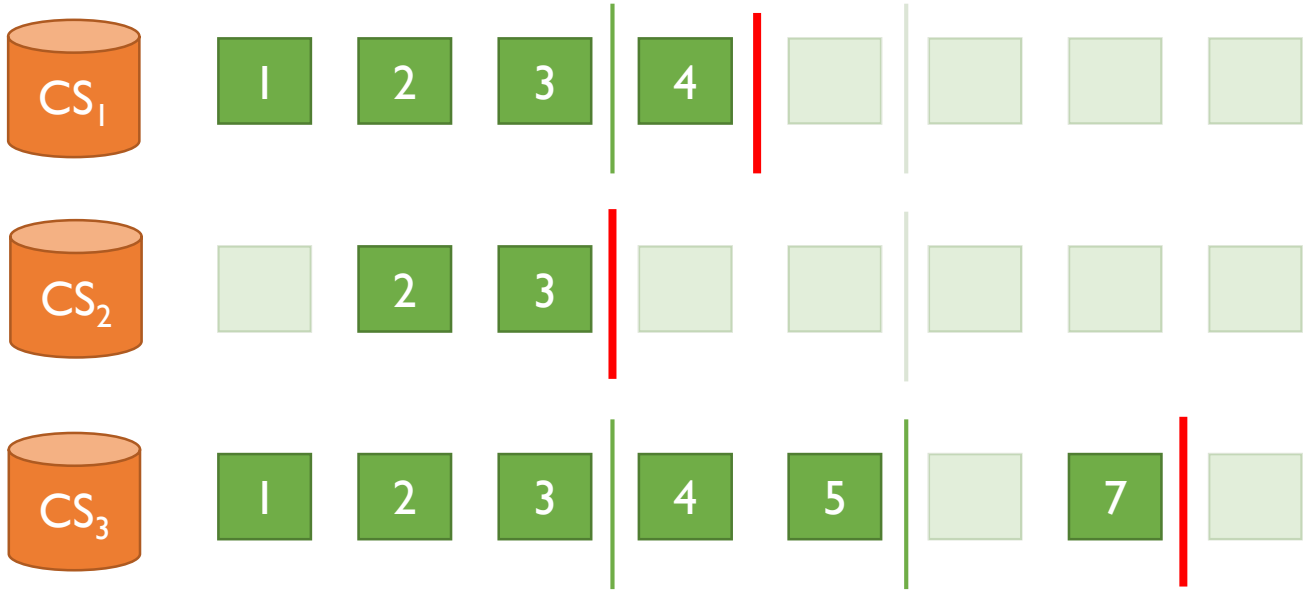
Collecting Expected Snapshots (ESs)

- ES: file system snapshots after a metadata operation
  - collected by ptrace system call interception



Simulating Crash Snapshots (CSs)

- CS: simulated crash sites by dropping I/O requests w.r.t. the barrier semantics
  - synthesized by a virtual disk driver [Yang06, Zheng14, Pillai14]



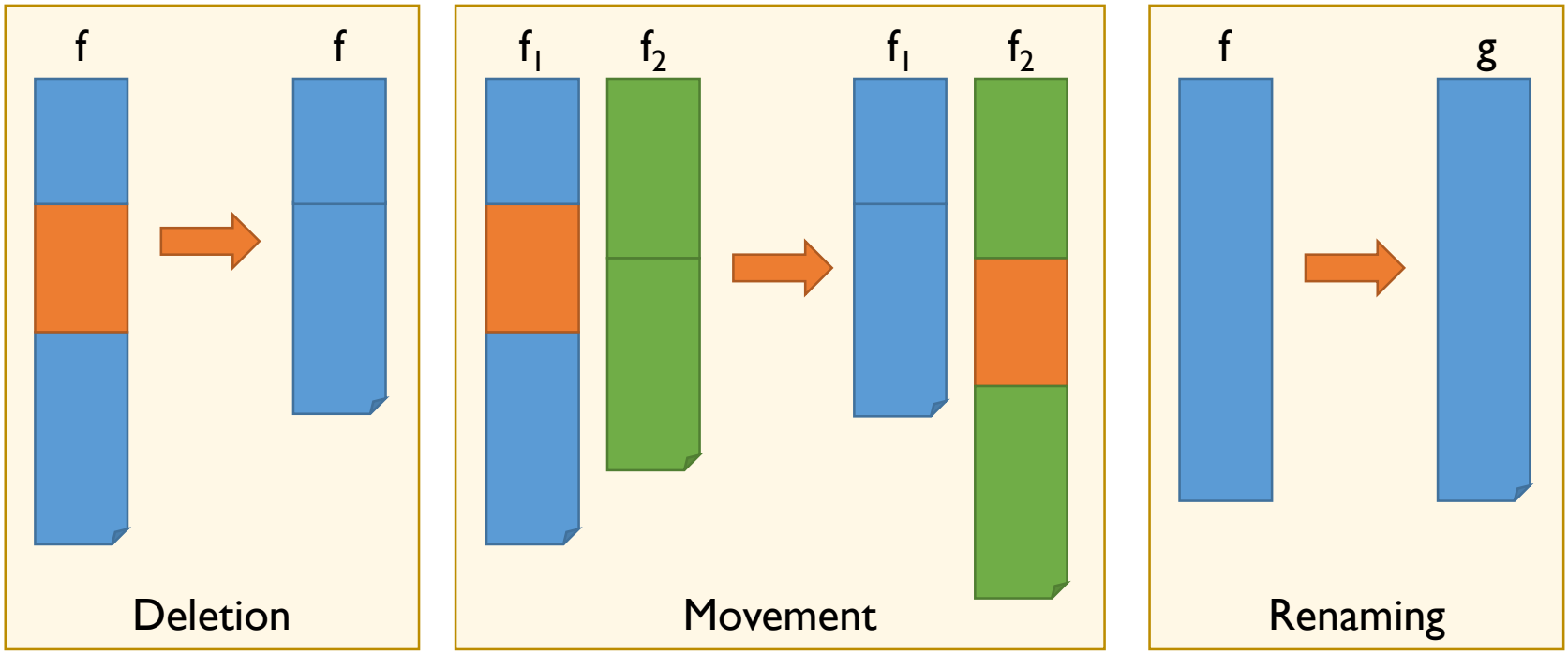
Defining Crash Consistency (the Oracle)

- The general oracle: a CS is consistent if and only if it can be transformed to an ES via *simple recovery operations*
  - the *general oracle* that validates crash consistency for any application

Editing-distance(CS, ES) < Threshold

Editing-distance(CS, ES)

- The minimum number of recovery operations to transform CS to ES

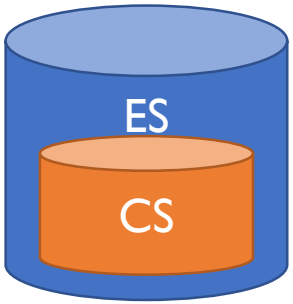


Calculating the Editing Distance

- Editing-distance(CS, ES) < k
  - NP-Complete, intractable

Reducible from bin-packing

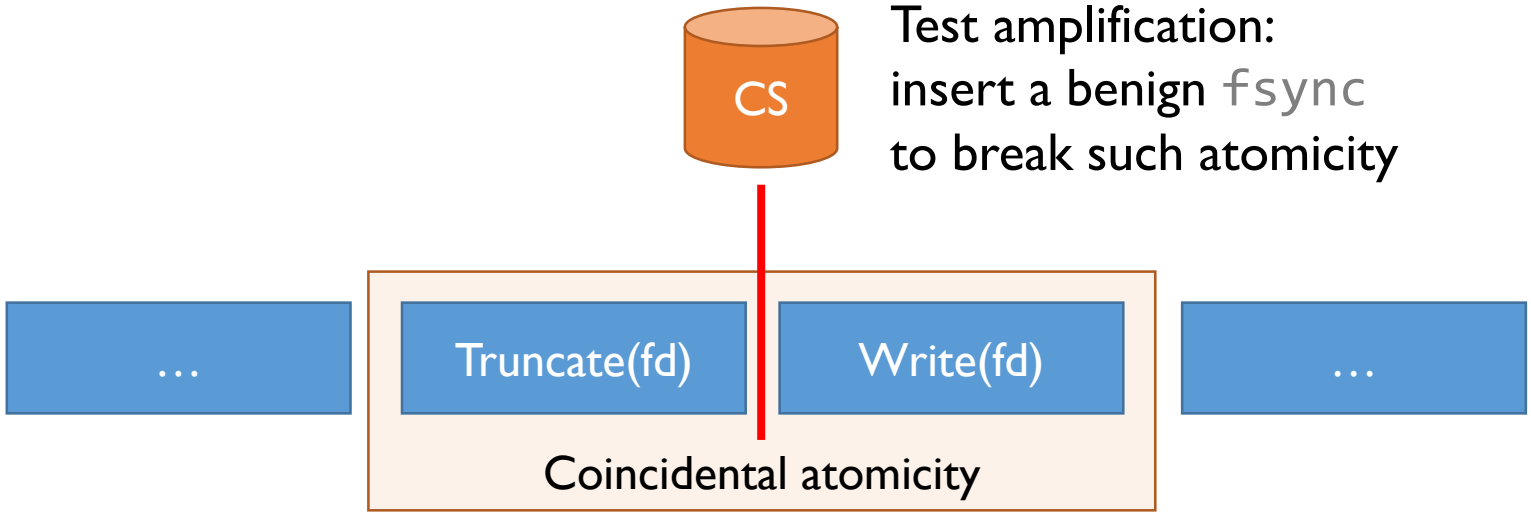
- Editing-distance(CS, ES) < ∞
  - for each byte value b, ES contains no less b than CS
  - linear-time algorithm



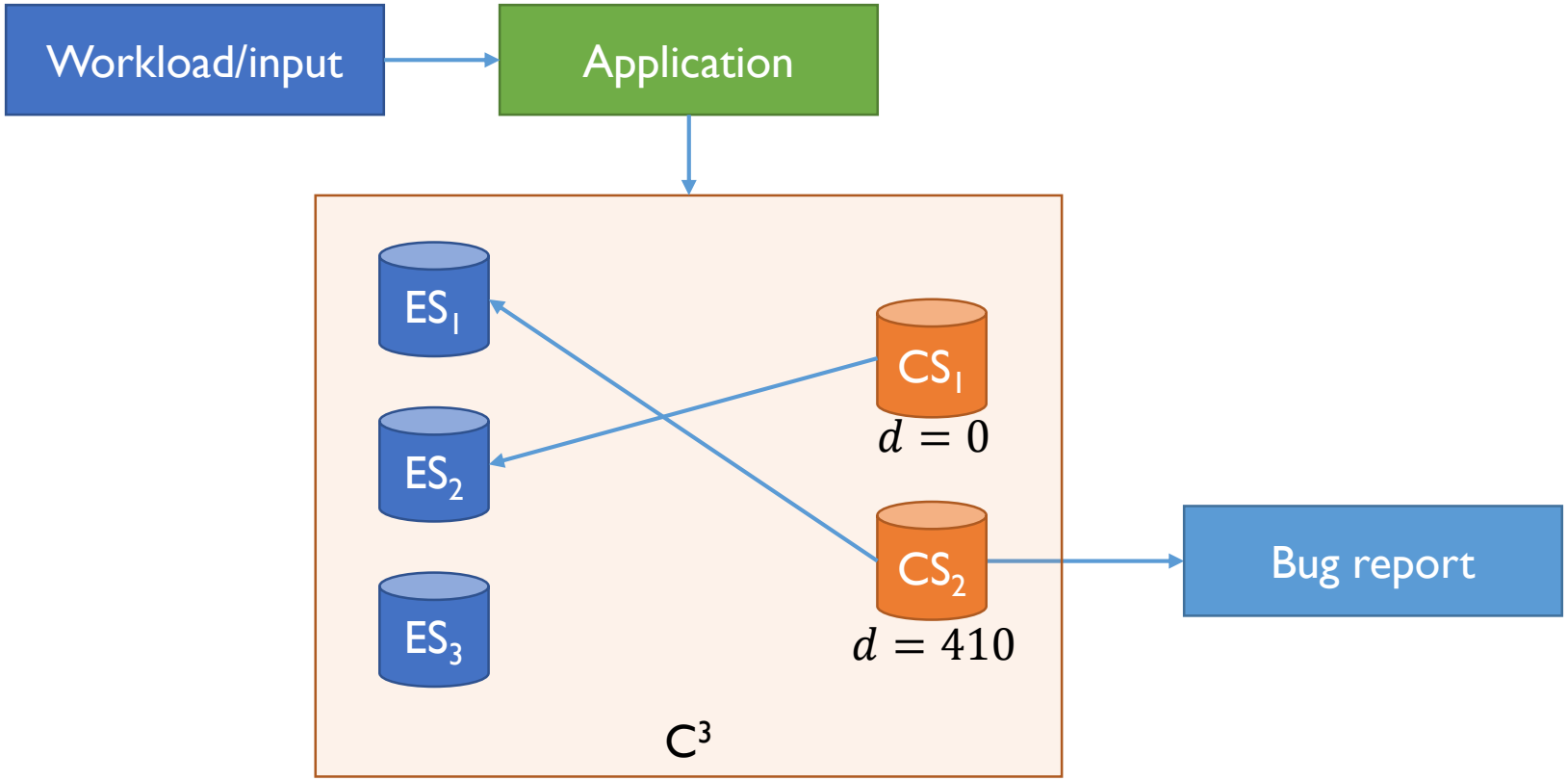
"subset" checking

Amplifying Test Inputs

- File system implementation may treat consecutive small-file operations atomic by chance
  - may lose data: large file, out-of-space, or crash



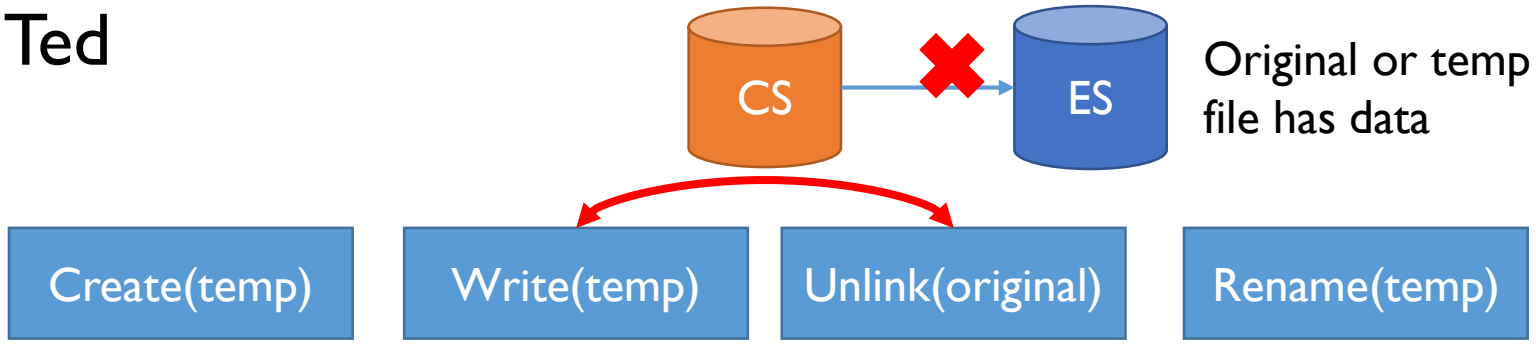
Putting Them Together: C³ Crash Consistency Checker



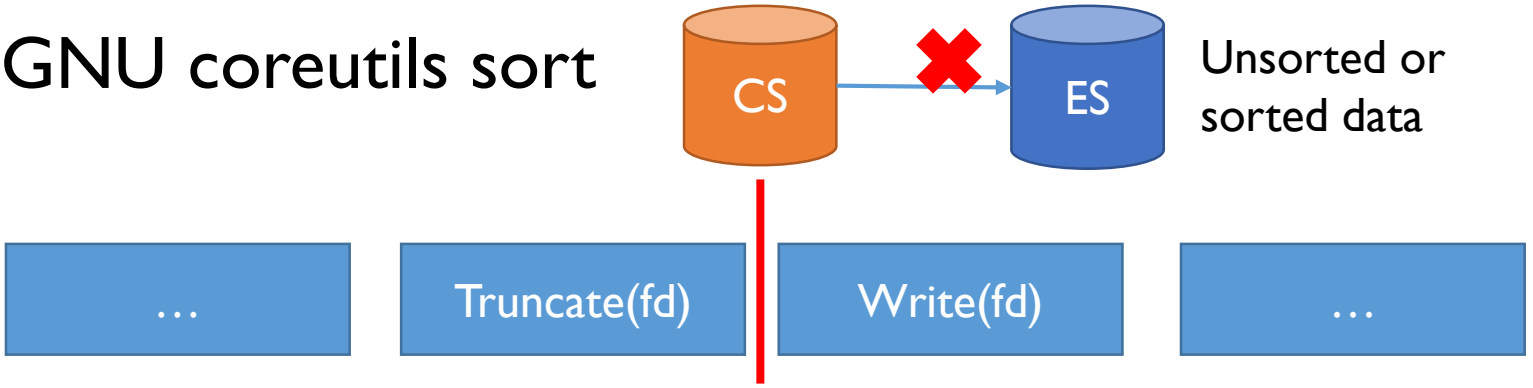
http://jiangyy.github.io/c3/

Detecting Crash Consistency Bugs

- Ted



- GNU coreutils sort



Evaluation Results

Evaluation Subjects

- 25 popular open-source applications
  - 10 command-line utilities



  - 15 productivity applications






Evaluation Settings

- Each subject is tested against a *simple* test input
  - command-line utility: a simple command usage
  - productivity application: replaying a typical use case

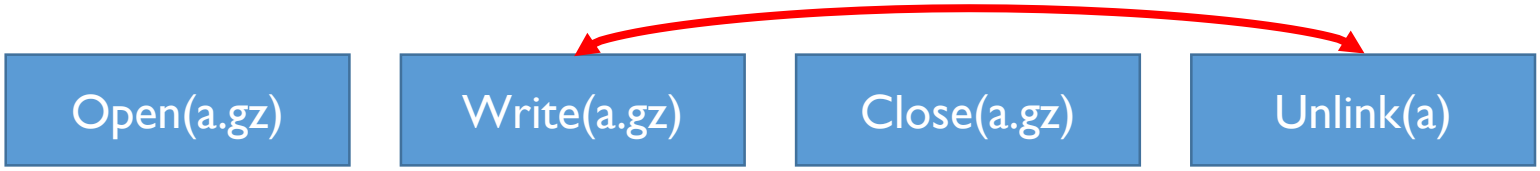
- Evaluated on a virtual machine
  - 2 CPU cores and 2GB RAM
  - Ubuntu Linux (Kernel 4.2)
  - ext4 file system

Bugs Found: Summary

Type	Application	Version	Consequences	Previously Unknown?	Need Test Amplification?
Util.	GNU make	4.1	Incorrect build		
	GNU zip	1.6	Data loss	Yes	
	bzip2	1.0.6	Data loss	Yes	
	GNU coreutils sort	8.21	Data loss	Yes	Yes
	Perl	5.22	Data loss	Yes	
	Python Shelve	2.7.11	Corruption	Yes	
Prod.	Gimp	2.8.14	Data loss		Yes
	CuteMarkEd	0.11.2	Data loss	Yes	Yes
	TeXmaker	4.5	Data loss	Yes	Yes
	TeXstudio	2.10.8	Data loss	Yes	Yes
	Ted	1.0	Data loss	Yes	
	jEdit	5.1.0	Data loss	Yes	
	GitHub Atom	1.5.3	Data loss		Yes
	Adobe Brackets	1.5.0	Data loss	Yes	Yes

Bug Example: GNU zip

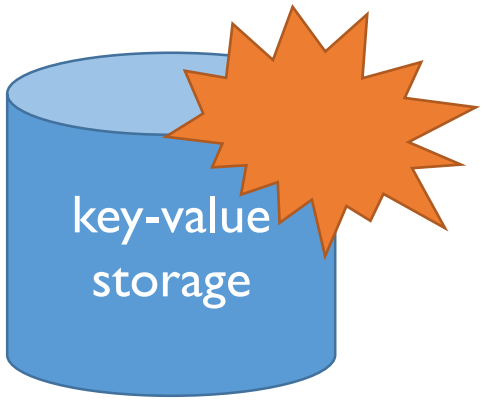
- Compressing a file = deleting it



- We see debates of the developers
  - provide crash consistency → huge overhead

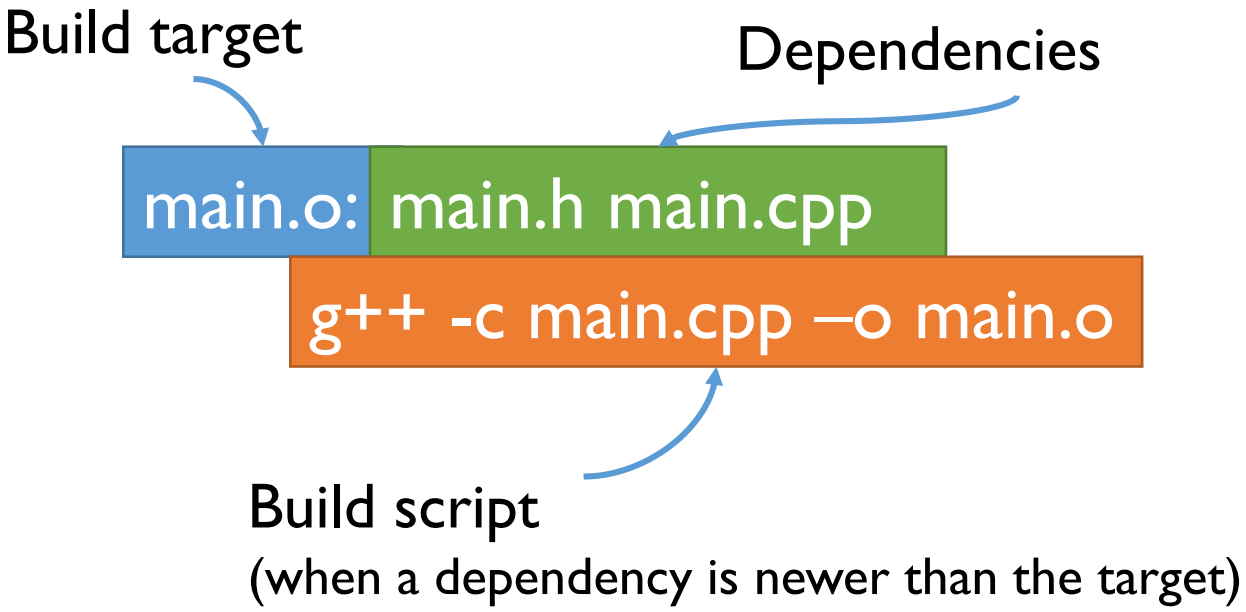
Bug Example: Python Standard Library Shelve (Key-Value Object Storage)

- Writing to the storage corrupts itself
  - neither backend (GDBM, DBM, and BSDDB) is crash-consistent
  - there has been proposal of using SQLite as backend



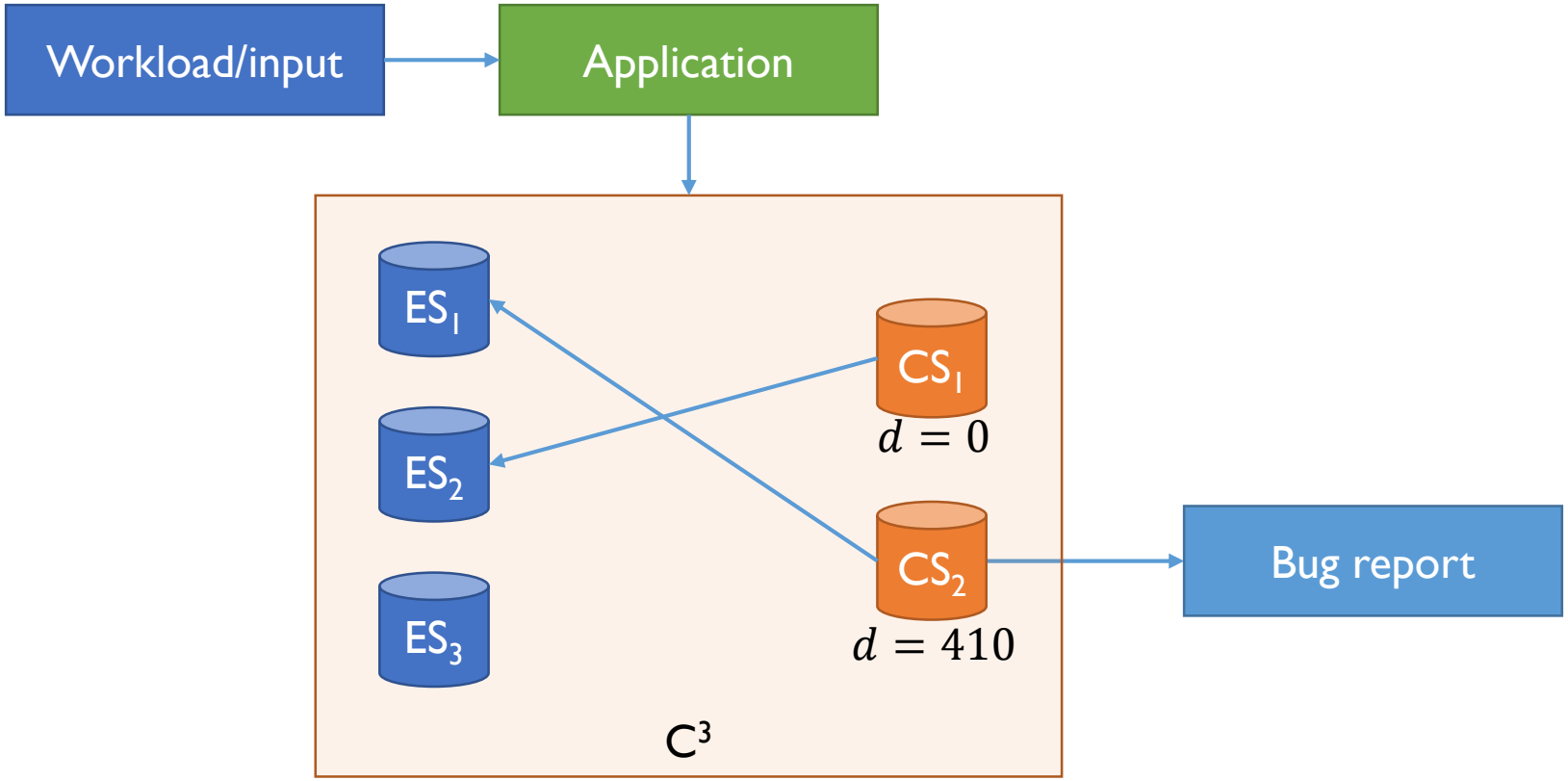
Bug Example: GNU make

- Corrupted build target with an up-to-date timestamp in the file system
  - may lead to incorrect build results



Summary

Crash Consistency Validation Made Easy



Future Directions

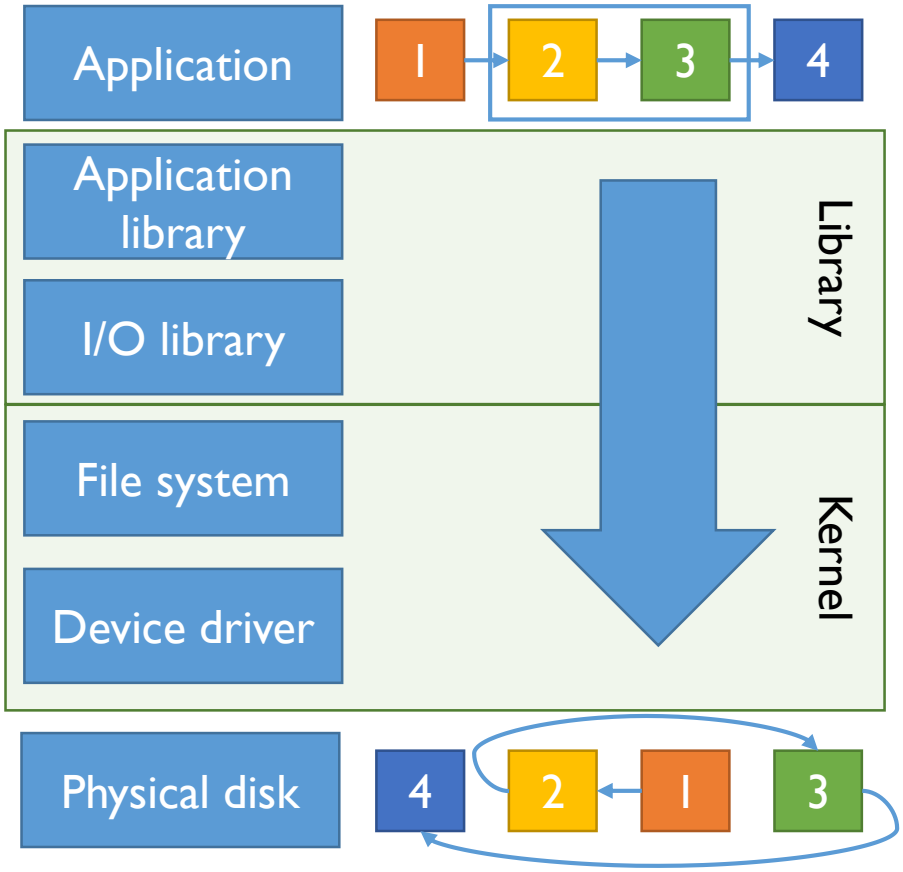
Static analysis

Offline detection

Operating system transactions

In-order file system

Online avoidance



Thank You!