IOWA STATE UNIVERSITY
Data Storage Lab

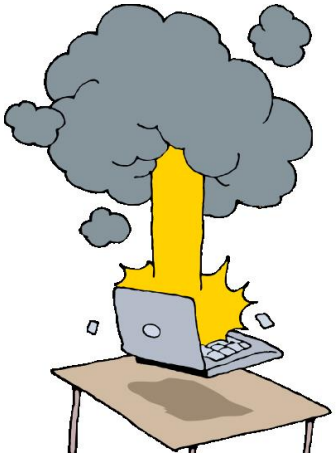
usenix ASSOCIATION

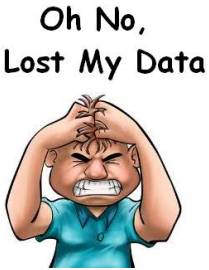
HotStorage '20
JULY 13–14, 2020

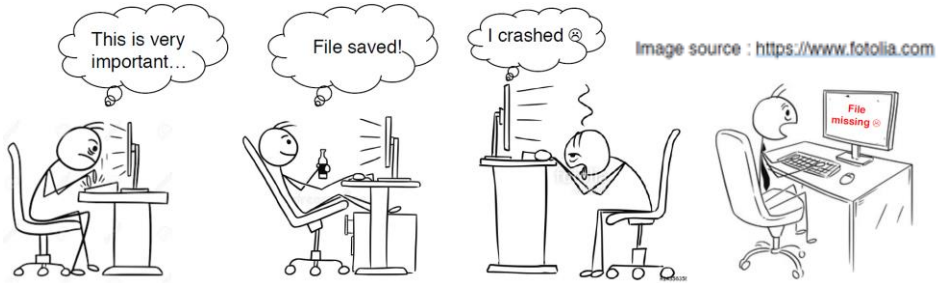
On Failure Diagnosis of the Storage Stack

Duo Zhang, Om Rameshwar Gatla, Runzhou Han, Mai Zheng
Iowa State University

Storage System Failures Are Troublesome

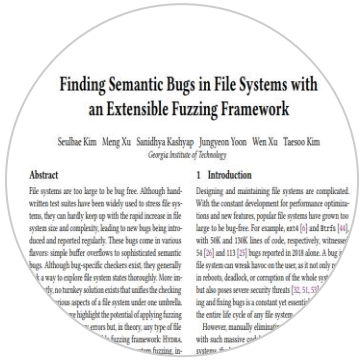






Existing Efforts Are Not Enough

- Mostly focus on *testing*
  - Require a special *testing environment*
    - e.g., a customized kernel
  - Still cannot prevent all failures in *production environment*


Finding Semantic Bugs in File Systems with an Extensible Fuzzing Framework

Seulbae Kim Meng Xu Sanidhya Kashyap Jungyeon Yoon Wen Xu Taesoo Kim
Georgia Institute of Technology

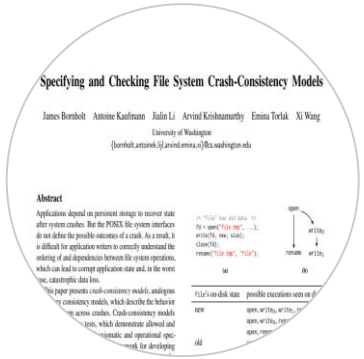
Abstract

File systems are too large to be bug free. Although hand-written test suites have been widely used to stress file systems, they can hardly keep up with the rapid increase in file system size and complexity, leading to new bugs being introduced and reported regularly. These bugs come in various flavors: simple buffer overflows to sophisticated semantic bugs. Although bug-specific checkers exist, they generally ... a way to explore file system states thoroughly. More importantly, no turnkey solution exists that unifies the checking ... various aspects of a file system under one umbrella. ... highlight the potential of applying fuzzing ... errors but, in theory, any type of file ... with such massive ... fuzzing framework. ... from fuzzing, in-

1 Introduction

Designing and maintaining file systems are complicated. With the constant development for performance optimizations and new features, popular file systems have grown too large to be bug-free. For example, ext4 [6] and Btrfs [44] ... with 56K and 136K lines of code, respectively, witnessed 54 [26] and 113 [25] bugs reported in 2018 alone. A bug ... file system can wreak havoc on the user, as it not only ... in reboots, deadlock, or corruption of the whole ... but also poses severe security threats [32, 51, 53] ... ing and fixing bugs is a constant yet essential ... the entire life cycle of any file system ...

However, manually eliminating ...

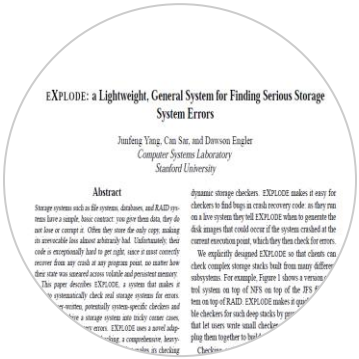

Specifying and Checking File System Crash-Consistency Models

James Bornholt Antoine Kaufmann Jialin Li Arvind Krishnamurthy Emina Torlak Xi Wang
University of Washington
{bornholt,antoink,jialinli,arvind,emina,xi}@cs.washington.edu

Abstract

Applications depend on persistent storage to recover state after system crashes. But the POSIX file system interfaces do not define the possible outcomes of a crash. As a result, it is difficult for application writers to correctly understand the ordering of and dependencies between file system operations, which can lead to corrupt application state and, in the worst ... catastrophic data loss.

... paper presents *crash-consistency models*, analogous ... consistency models, which describe the behavior ... across crashes. Crash-consistency models ... tests, which demonstrate allowed and ... matic and operational spec- ... work for developing


EXPLODE: a Lightweight, General System for Finding Serious Storage System Errors

Junfeng Yang, Can Sar, and Dawson Engler
Computer Systems Laboratory
Stanford University

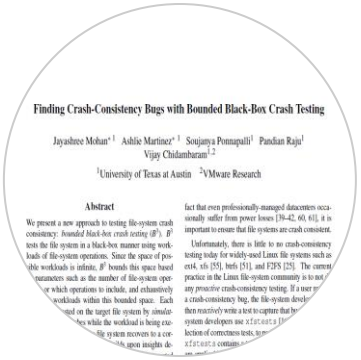
Abstract

Storage systems such as file systems, databases, and RAID systems have a simple, basic contract: you give them data, they do not lose or corrupt it. Often they store the only copy, making its unrecoverable loss almost arbitrarily bad. Unfortunately, their code is exceptionally hard to get right, since it must correctly recover from any crash at any program point, no matter how ... their state was smeared across volatile and persistent memory. This paper describes EXPLODE, a system that makes it ... to systematically check real storage systems for errors. ... errors. EXPLODE uses a novel adap- ... ing a comprehensive, heavy- ... ems, its checking

dynamic storage checkers. EXPLODE makes it easy for checkers to find bugs in crash recovery code: as they run on a live system they tell EXPLODE when to generate the disk images that could occur if the system crashed at the current execution point, which they then check for errors.

We explicitly designed EXPLODE so that clients can check complex storage stacks built from many different subsystems. For example, Figure 1 shows a versatile ... test system on top of NFS on top of the JFS ... ben on top of RAID. EXPLODE makes it easy ... ble checkers for each deep stacks by ... that let users write small checker ... plug them together to build ...

Checkers


Finding Crash-Consistency Bugs with Bounded Black-Box Crash Testing

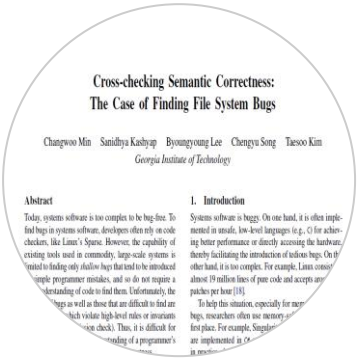
Jayashree Mohan*[1] Ashlie Martinez*[1] Soujanya Ponnapalli[1] Pandian Raju[1]
Vijay Chidambaram[1,2]
[1]University of Texas at Austin [2]VMware Research

Abstract

We present a new approach to testing file-system crash consistency: *bounded black-box crash testing* (B^3). B^3 tests the file system in a black-box manner using workloads of file-system operations. Since the space of possible workloads is infinite, B^3 bounds this space based ... parameters such as the number of file-system operations ... which operations to include, and exhaustively ... workloads within this bounded space. Each ... run on the target file system by *simulating* ... ses while the workload is being exe- ... file system recovers to a cor- ... upon insights de- ... *xfstests* contains ...

fact that even professionally-managed datacenters occasionally suffer from power losses [39–42, 60, 61], it is important to ensure that file systems are crash consistent.

Unfortunately, there is little to no crash-consistency testing today for widely-used Linux file systems such as ext4, xfs [55], btrfs [51], and F2FS [25]. The current practice in the Linux file-system community is to not ... any *proactive* crash-consistency testing. If a user re- ... a crash-consistency bug, the file-system develo- ... then *reactively* write a test to capture that bu- ... system developers use *xfstests* ... lection of correctness tests, to ... *xfstests* contains ...


Cross-checking Semantic Correctness: The Case of Finding File System Bugs

Changwoo Min Sanidhya Kashyap Byoungyoung Lee Chengyu Song Taesoo Kim
Georgia Institute of Technology

Abstract

Today, systems software is too complex to be bug-free. To find bugs in systems software, developers often rely on code checkers, like Linux's Sparse. However, the capability of existing tools used in commodity, large-scale systems is ... limited to finding only *shallow bugs* that tend to be introduced ... simple programmer mistakes, and so do not require a ... understanding of code to find them. Unfortunately, the ... bugs as well as those that are difficult to find are ... which violate high-level rules or invariants ... tion check). Thus, it is difficult for ... understanding of a programmer's

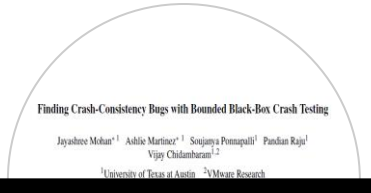
1. Introduction

Systems software is buggy. On one hand, it is often implemented in unsafe, low-level languages (e.g., C) for achieving better performance or directly accessing the hardware, thereby facilitating the introduction of tedious bugs. On the other hand, it is too complex. For example, Linux consists of almost 19 million lines of pure code and accepts around ... patches per hour [18].

To help this situation, especially for more ... bugs, researchers often use memory ... first place. For example, Sparse ... are implemented in C ...

Existing Efforts Are Not Enough

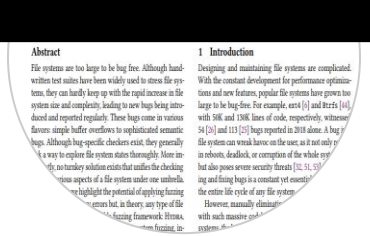
- Mostly focus on *testing*
  - Require a special *testing environment*
    - e.g., a customized kernel
  - Still cannot prevent all failures in *production environment*


Finding Crash-Consistency Bugs with Bounded Black-Box Crash Testing

Jayashree Mohan[*1] Ashlie Martinez[*1] Soujanya Ponnapalli[1] Pandian Raju[1]
Vijay Chidambaram[1,2]

[1]University of Texas at Austin [2]VMware Research

What to do if failures happen ?

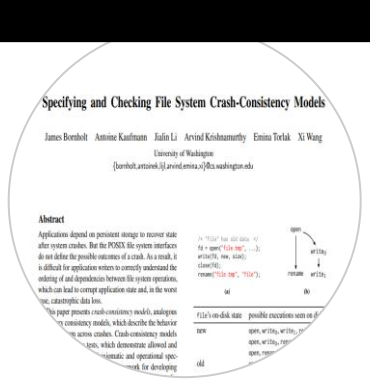

Abstract

File systems are too large to be bug free. Although hand-written test suites have been widely used to stress file systems, they can hardly keep up with the rapid increase in file system size and complexity, leading to new bugs being introduced and reported regularly. These bugs come in various flavors: simple buffer overflows to sophisticated semantic bugs. Although bug-specific checkers exist, they generally [illegible] a way to explore file system states thoroughly. More importantly, no turnkey solution exists that unifies the checking [illegible] various aspects of a file system under one umbrella. [illegible] highlight the potential of applying fuzzing [illegible] errors but, in theory, any type of file [illegible] fuzzing framework. However, [illegible] from fuzzing, in-

1 Introduction

Designing and maintaining file systems are complicated. With the constant development for performance optimizations and new features, popular file systems have grown too large to be bug-free. For example, ext4 [6] and Btrfs [41] with 56K and 130K lines of code, respectively, witnesses 54 [26] and 113 [25] bugs reported in 2018 alone. A bug in a file system can wreak havoc on the user, as it not only [illegible] in reboots, deadlock, or corruption of the whole system, but also poses severe security threats [32, 51, 53]. Finding and fixing bugs is a constant yet essential [illegible] the entire life cycle of any file system.

However, manually eliminating [illegible] with such massive code [illegible] systems. [illegible]


Specifying and Checking File System Crash-Consistency Models

James Bornholt Antoine Kaufmann Jialin Li Arvind Krishnamurthy Emina Torlak Xi Wang

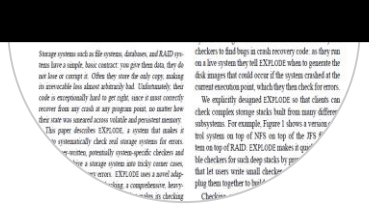
University of Washington

{bornholt,antoink,jialin,arvind,emina,xi}@cs.washington.edu

Abstract

Applications depend on persistent storage to recover state after system crashes. But the POSIX file system interfaces do not define the possible outcomes of a crash. As a result, it is difficult for application writers to correctly understand the ordering of and dependencies between file system operations, which can lead to corrupt application state and, in the worst case, catastrophic data loss.

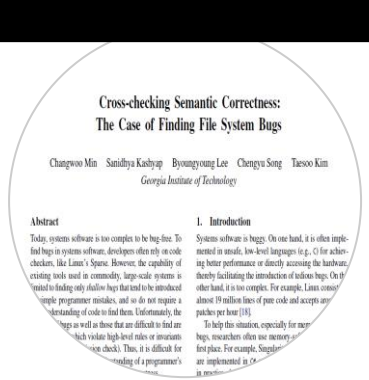
This paper presents *crash-consistency models*, analogous to memory consistency models, which describe the behavior [illegible] across crashes. Crash-consistency models [illegible] tests, which demonstrate allowed and [illegible] matic and operational specifications [illegible] work for developing


Storage systems such as file systems, databases, and RAID systems have a simple, basic contract: you give them data, they do not lose or corrupt it. Often they store the only copy, making its unrecoverable loss almost arbitrarily bad. Unfortunately, their code is exceptionally hard to get right, since it must correctly recover from any crash at any program point, no matter how these state was smeared across volatile and persistent memory. This paper describes EXPLODE, a system that makes it [illegible] to systematically check real storage systems for errors. [illegible] errors. EXPLODE uses a novel adaptive [illegible] ge a storage system into tricky corner cases, [illegible] ooling a comprehensive, heavy- [illegible] rules in checking

checkers to find bugs in crash-recovery code. as they run on a live system they tell EXPLODE when to generate the disk images that could occur if the system crashed at the current execution point, which they then check for errors.

We explicitly designed EXPLODE so that clients can check complex storage stacks built from many different subsystems. For example, Figure 1 shows a version [illegible] test system on top of NFS on top of the JFS [illegible] ben on top of RAID. EXPLODE makes it easy [illegible] ble checkers for such deep stacks by [illegible] that let users write small checker [illegible] plug them together to build [illegible]

Checkers


Cross-checking Semantic Correctness: The Case of Finding File System Bugs

Changwoo Min Sanidhya Kashyap Byoungyoung Lee Chengyu Song Taesoo Kim

Georgia Institute of Technology

Abstract

Today, systems software is too complex to be bug-free. To find bugs in systems software, developers often rely on code checkers, like Linux's Sparse. However, the capability of existing tools used in commodity, large-scale systems is limited to finding only *shallow bugs* that tend to be introduced [illegible] simple programmer mistakes, and so do not require a [illegible] derstanding of code to find them. Unfortunately, the [illegible] bugs as well as those that are difficult to find are [illegible] hich violate high-level rules or invariants [illegible] sion check). Thus, it is difficult for [illegible] nding of a programmer's

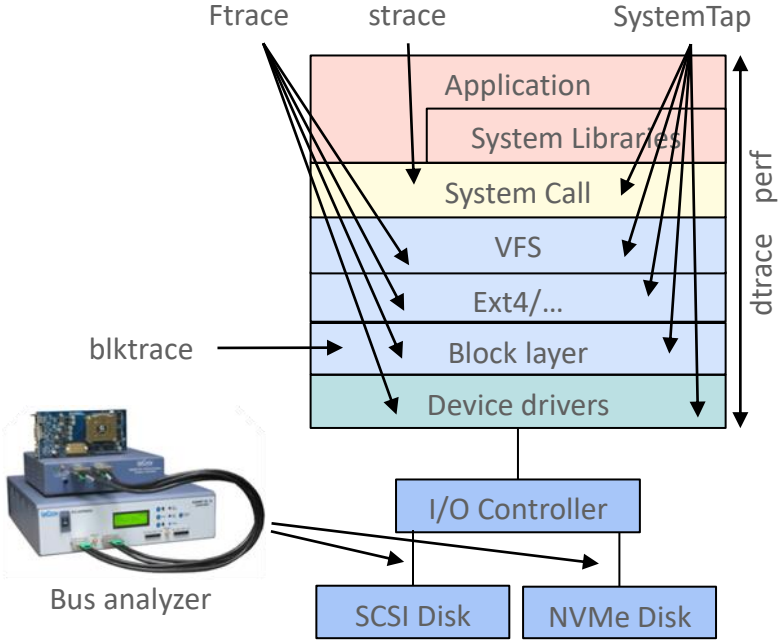
1. Introduction

Systems software is buggy. On one hand, it is often implemented in unsafe, low-level languages (e.g., C) for achieving better performance or directly accessing the hardware, thereby facilitating the introduction of tedious bugs. On the other hand, it is too complex. For example, Linux consists almost 19 million lines of pure code and accepts [illegible] patches per hour [18].

To help this situation, especially for [illegible] bugs, researchers often use memory [illegible] first place. For example, Sing[illegible] are implemented in C [illegible] in [illegible]

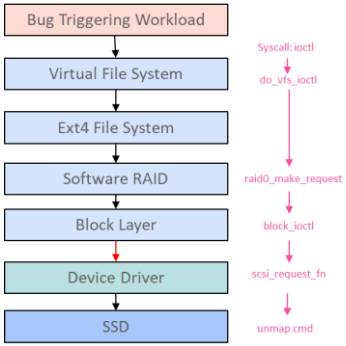
Practical Diagnosis Tools & Limitations

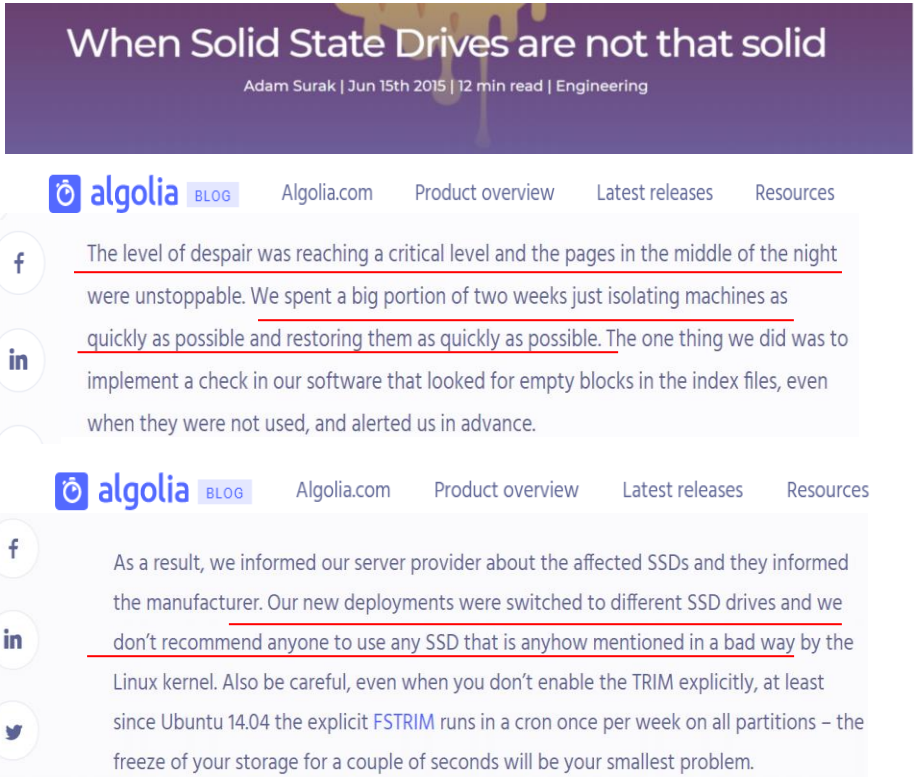
- Practical diagnosis tools
  - Software-based
    - e.g., GDB, SystemTap, Ftrace
  - Hardware-based
    - e.g., Bus analyzer
- Limitations
  - Require substantial manual efforts
    - e.g., GDB single-stepping
  - Require special hardware
  - Only cover partial storage stack



A Real-World Case: Diagnosis Is Challenging

- Algolia data center incident:
  - Servers crashed and files corrupted for unknown reason
  - After weeks of diagnosis, Samsung SSDs were mistakenly blamed
  - After one month, a Linux kernel bug was identified as root cause

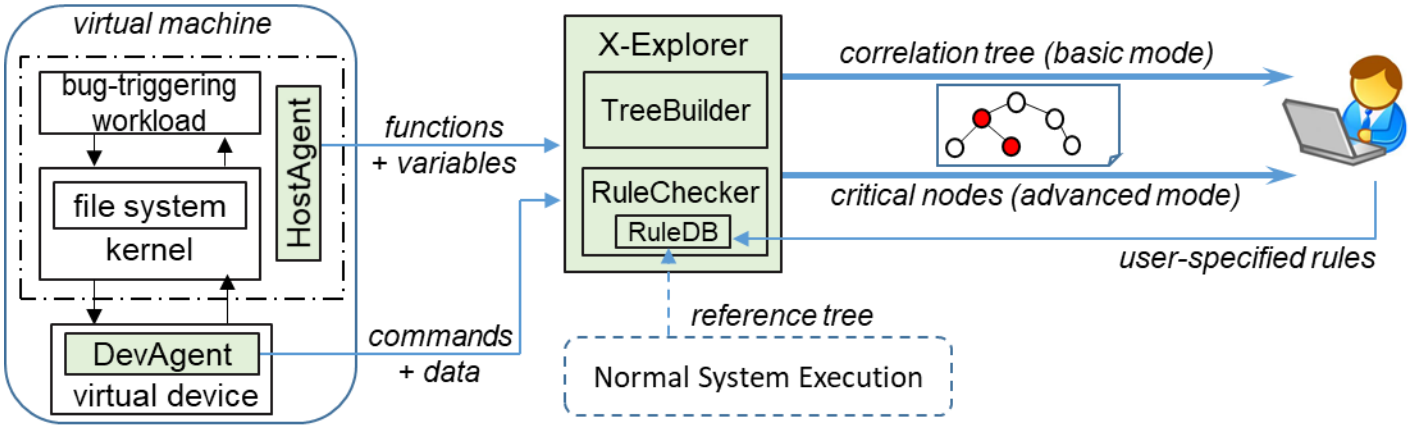




Our Approach

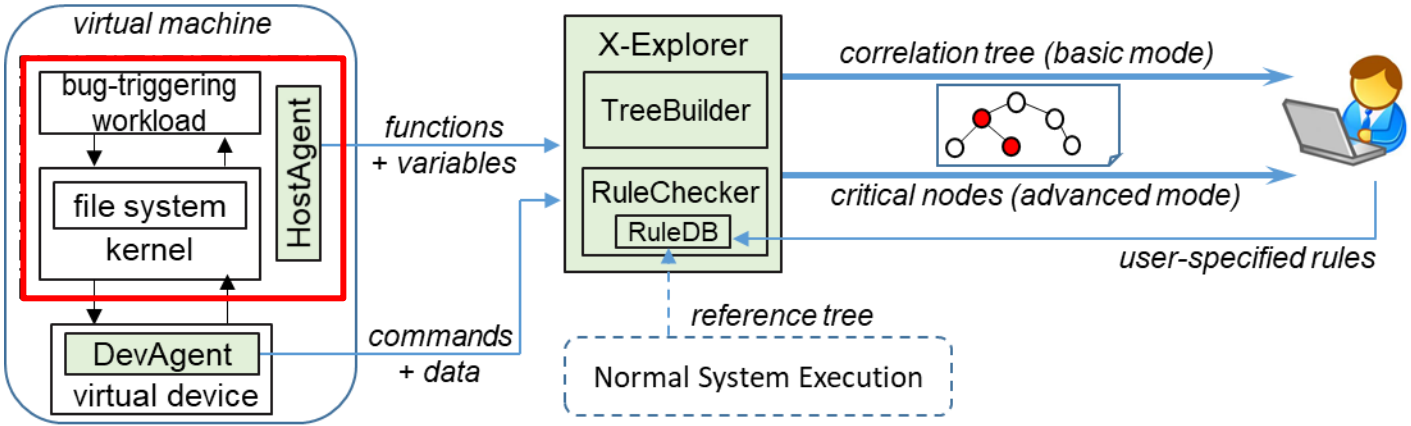
X-Ray: A Cross-Layer Approach

- Support unmodified software stack
- Intercept device activity without relying on kernel or special hardware
- Visualize multi-layer correlation
- Narrow down root cause (semi)automatically



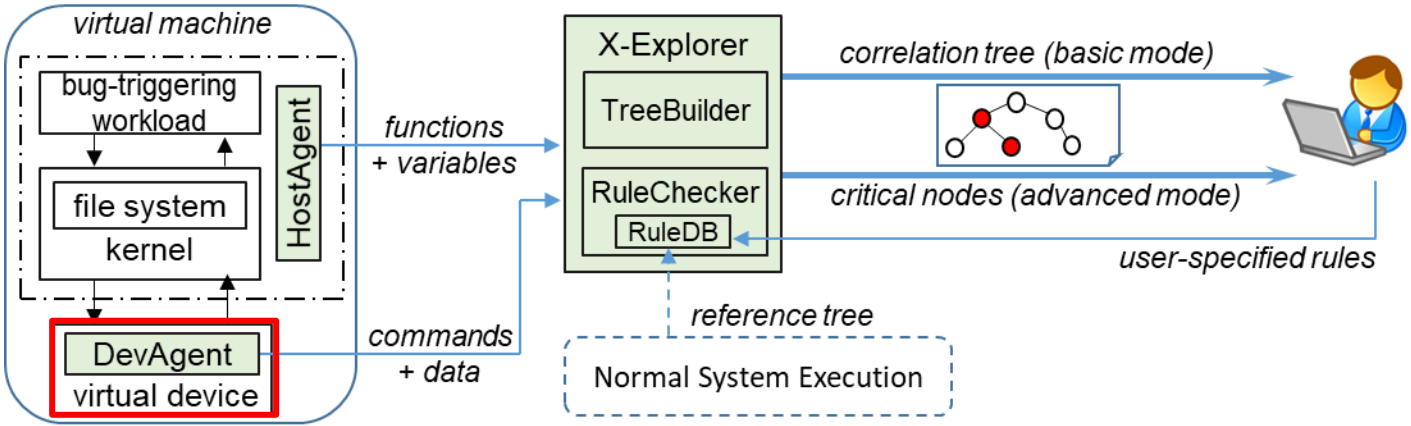
X-Ray: A Cross-Layer Approach

- HostAgent: help understand host-side system activities
  - Trace host-side events
    - e.g., syscalls, kernel functions



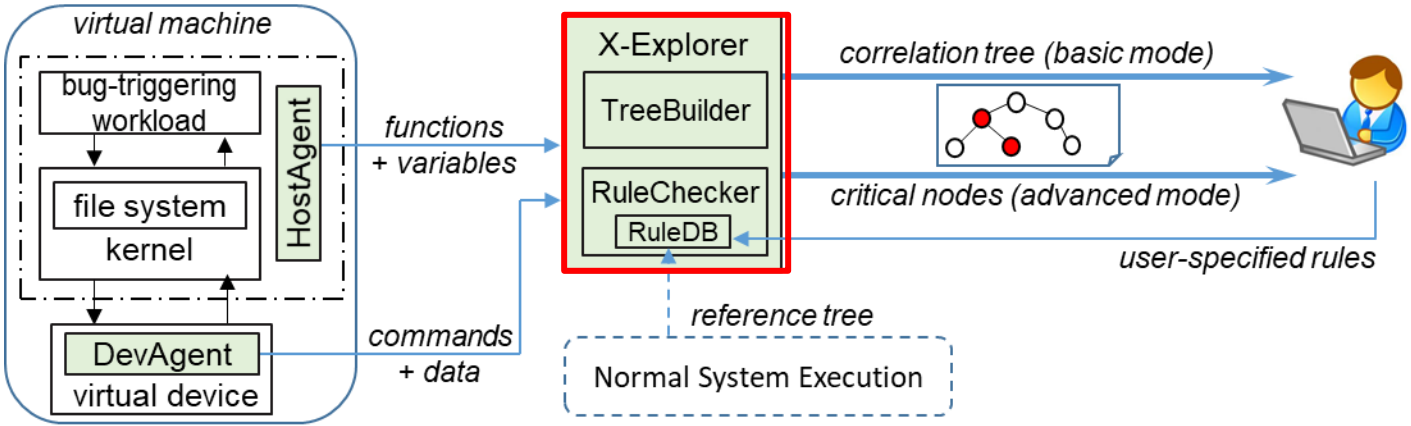
X-Ray: A Cross-Layer Approach

- DevAgent: help understand changes of persistent states
  - Trace device commands
    - e.g., SCSI, NVMe



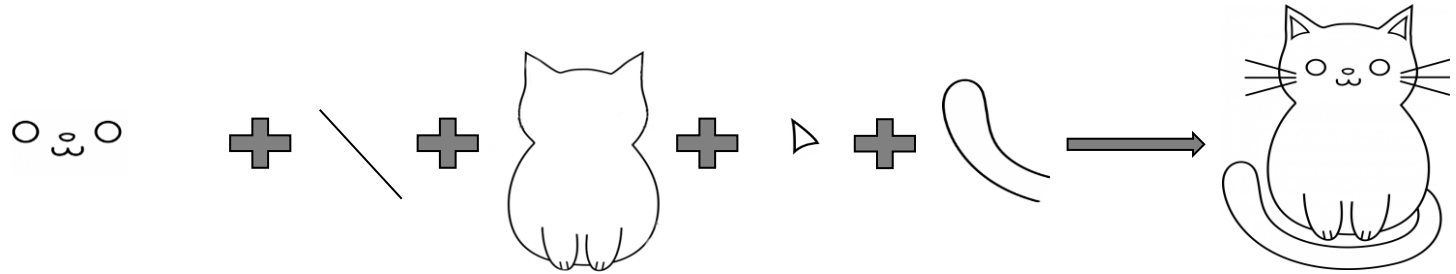
X-Ray: A Cross-Layer Approach

- X-Explorer: facilitate diagnosis in two ways
  - Build and visualize multi-layer correlation (i.e., correlation tree)
  - Highlight critical nodes/paths based on rules



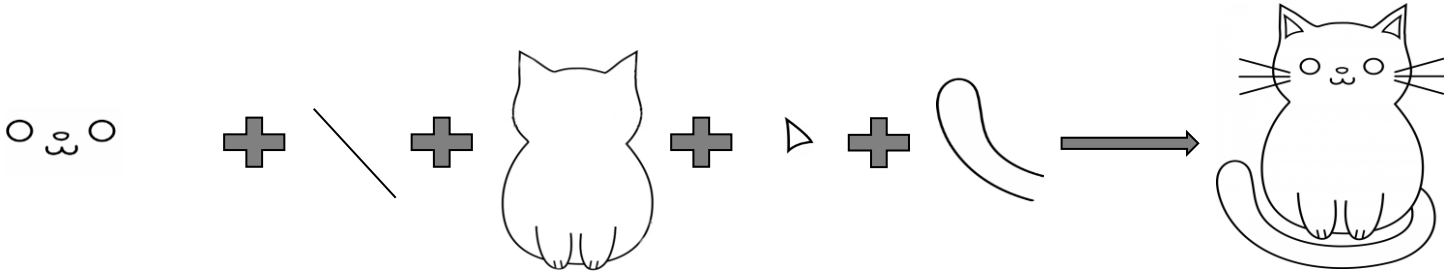
Key Challenge #1

- How to correlate information across layers ?



Key Challenge #1

- How to correlate information across layers ?
  - Cannot use SCSI/NVMe hints ☹
    - Require modification to workload/OS
  - Use timestamp ☺
    - Customized Ftrace frontend
      - Convert execution time to epoch time
    - NTP(Network Time Protocol) based synchronization
      - Solve accuracy problem caused by virtualization



Key Challenge #2

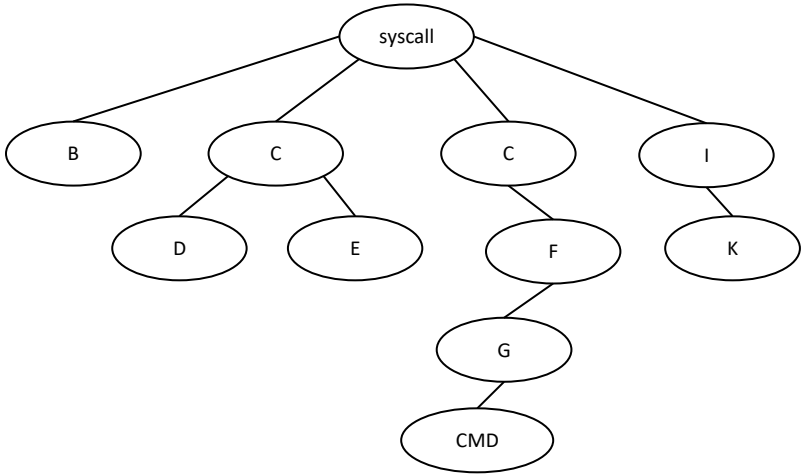
- How to reduce manual efforts ?

Key Challenge #2

- How to reduce manual efforts ?
  - Visualize cross-layer events & dependencies in a correlation tree

Dependency		
Syscall	→	B
Syscall	→	C
C	→	D
C	→	E
Syscall	→	C
C	→	F
F	→	G
G	→	CMD
Syscall	→	I
I	→	K

Tracing log



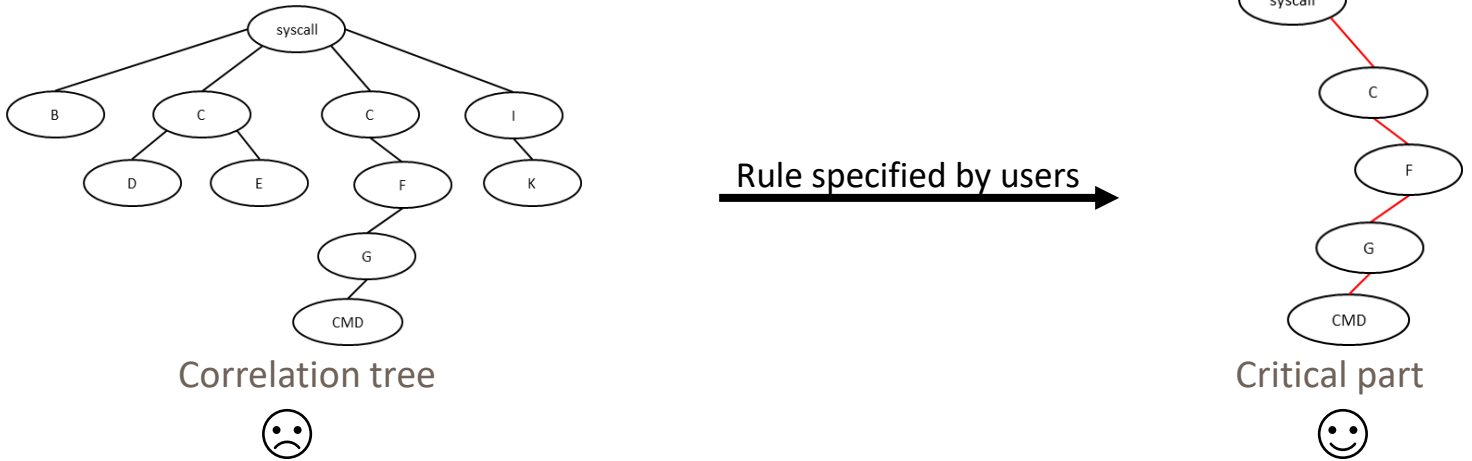
Cross-layer tree

Key Challenge #2

- How to reduce manual efforts ?
  - Visualize cross-layer events & dependencies in a correlation tree
  - Automatically narrow down the root cause via rules

Key Challenge #2

- How to reduce manual efforts ?
  - Visualize cross-layer events & dependencies in a correlation tree
  - Automatically narrow down the root cause via rules
    - Rules specified by users (e.g., “ancestors of device commands”)

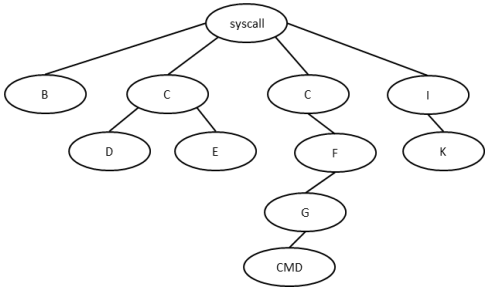


Correlation tree

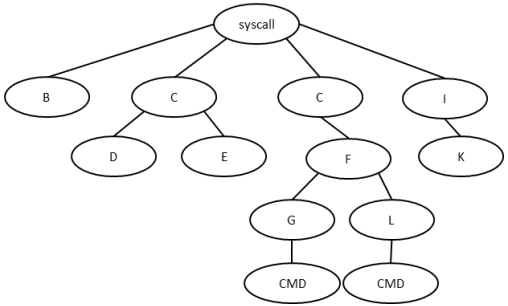
Critical part

Key Challenge #2

- How to reduce manual efforts ?
  - Visualize cross-layer events & dependencies in a correlation tree
  - Automatically narrow down the root cause via rules
    - Rules specified by users (e.g., “ancestors of device commands”)
    - Rules derived from reference execution (e.g., non-failure run due to different kernel version)

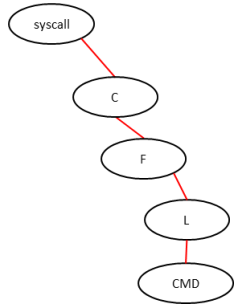


Tree from Abnormal execution



Tree from reference execution

Rules derived from reference

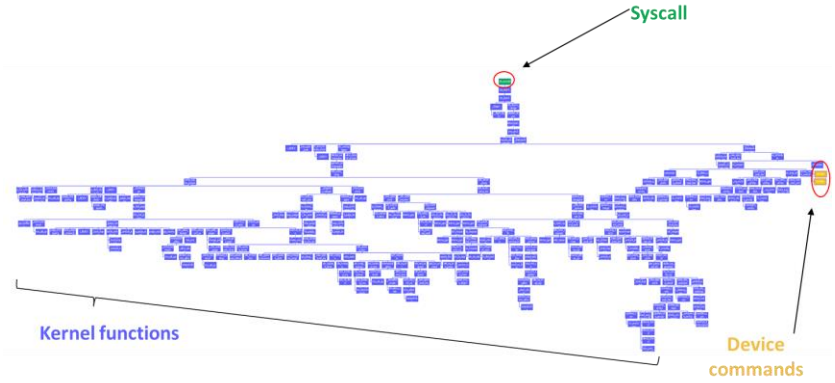


Difference part

Preliminary Results

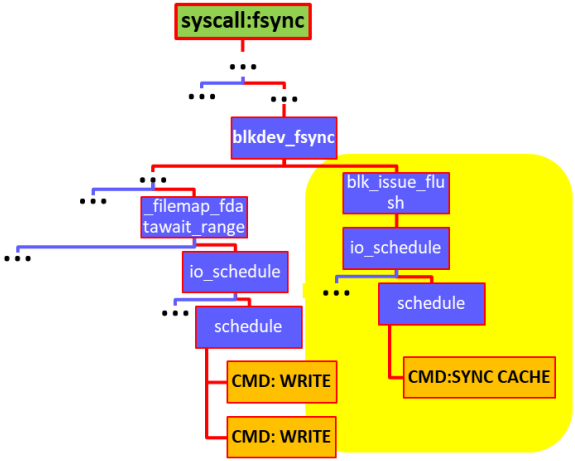
Preliminary Results

- Case Study
  - A kernel bug manifested as serialization errors on SSDs [Zheng *et. al.*@TOCS'16, FAST'13]
  - The problem can be observed in the correlation tree clearly
  - Rules can help narrow down the root cause quickly



Tree from abnormal execution

Rules



Pinpointed root cause

Preliminary Results

- Result summary
  - 5 failure cases reported in the literature
  - 3 simple rules to define critical parts of the correlation trees
  - Reduce the search space for root causes effectively
    - 0.06% - 4.97% nodes of the original trees

Case ID	node count in original tree	node count by Rule#1	node count by Rule#2	node count by Rule#3
1	11,353 (100%)	704 (6.20%)	571 (5.03%)	30 (0.26%)
2	34,083 (100%)	697 (2.05%)	328 (0.96%)	22 (0.06%)
3	24,355 (100%)	1254 (5.15%)	1210 (4.97%)	/
4	273,653 (100%)	10230 (3.74%)	/	/
5	284,618 (100%)	5621 (1.97%)	5549 (1.95%)	/

Conclusion and Ongoing Work

- X-Ray: A cross-layer approach for failure diagnosis
  - Support unmodified software stack
  - Intercept device activity without relying on kernel or special hardware
  - Visualize multi-layer correlation
  - Narrow down root cause (semi)automatically
- Explore more real-world failure cases
- Derive more diagnosis rules
- Automate the comparison based on reference tree

Thanks !

Duo Zhang
duozhang@iastate.edu
https://www.ece.iastate.edu/~mai/lab/dsl.html