

IOWA STATE UNIVERSITY

Data Storage Lab



usenix
ASSOCIATION

HotStorage '20

JULY 13-14, 2020

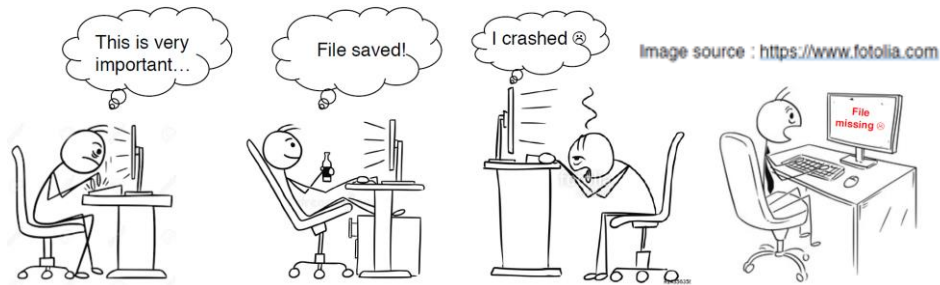
On Failure Diagnosis of the Storage Stack

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

Seulhee Kim, Meng Xu, Sanidhya Kashyap, Jungyeon Yoon, Wen Xu, Tamaso 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 the system, they can hardly keep up with the rapid increase in the system size and complexity. Finding bugs by testing is tedious and reported regularly. These bugs come in various flavors: simple buffer overflows to sophisticated semantic bugs. Although bug-specific checkers exist, they generally only help to reduce the number of bugs. We propose a new way to explore the file system states thoroughly. Here, we help to reduce the number of bugs by finding the semantic aspects of a file system under one umbrella, we highlight the potential of applying fuzzing to file systems but, in theory, any type of file system can be fuzzed. We propose a new framework, *FileFuzz*, to find semantic bugs in file systems.

1 Introduction

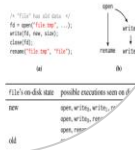
Designing and maintaining file systems are complicated. With the constant development for performance optimizations and new features, popular file systems have grown too large to be bug-free. For example, *ext4* [1] and *extfs* [34] with 58K and 138K lines of code, respectively, whereas *fs* [24] and *ext4* [1] [35] bugs reported in 2018 alone. A bug in a file system can wreak havoc on the user, as it not only corrupts data, but also causes security threats [32, 33, 36]. The entire life cycle of any file system is complex. However, manually eliminating such massive code is a daunting task.

Specifying and Checking File System Crash-Consistency Models

James Bornholt, Antoine Kaufmann, Bailin Li, Arvind Krishnamurthy, Emina Torkal, Xi Wang
University of Washington
{bornholt, antoine.k, arvind.krishnamurthy, emina.torkal, xi.wang}@u.washington.edu

Abstract

Applications depend on persistent storage to recover state after system crashes. But the POSIX file system interface does not define the possible outcomes of a crash. As a result, it is difficult for application writers to correctly understand the meaning of and dependencies between the 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 consistency models, which describe the behavior of file systems across crashes. Crash-consistency models are formalized as state transition systems, which demonstrate allowed and disallowed sequences of system operations and application operations. We use these models to develop a framework for developing



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

Jinfaeng Yang, Cui Sun, and Dawson Engler
Computer Systems Laboratory
Stanford University

Abstract

Storage systems such as the *filesystem*, *database*, 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 an irreparable loss almost always bad. Unfortunately, their code is exceptionally hard to get right, since it must correctly recover from any crash at any program point, so subtle how data can be corrupted across reliable and persistent storage. This paper describes EXPLODE, a system that makes it easy to systematically check and manage systems for errors. EXPLODE, generally, performs specific checks and manages storage systems as they come across. EXPLODE uses a novel algorithm to check a comprehensive, hierarchical set of checks on storage systems.

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 recovery 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 virtualized system on top of NFS on top of the *ext4* filesystem on top of RAID. EXPLODE makes it easy for checkers for each deep check by providing that let users write small, checkers that play them together to build a comprehensive set of checks.

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

Jayashree Mohan¹, Ashlie Martinez², Soujanya Ponnampalli¹, Pandian Raju¹, Vijay Chidambaram^{1,2}

¹University of Texas at Austin ²VMware Research

Abstract

We present a new approach to testing file-system crash-consistency: *bounded black-box crash testing* (B³C²). B³C² tests the file-system in a black-box manner using workloads of file-system operations. Since the space of possible workloads is infinite, B³C² bounds this space based on sequences such as the number of file-system operations which operations to include, and exhaustively workloads within this bounded space. Each workload on the target file-system is simulated by a system developer using a crash-consistency checker. This checker then executes the workload in a controlled environment while the workload is being executed. The system developer uses a checker to capture the execution of correctness tests, to ensure that the system is crash-consistent.

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

Changwoo Min, Sanidhya Kashyap, Byoungyeon Lee, Chengyu Song, Tamaso 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* *Spice*. However, the capability of existing tools used in commodity, large-scale systems is limited to finding only shallow bugs that tend to be introduced by simple programmer mistakes, and so do not require a deep 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 (e.g., *memory* checks). Thus, it is difficult for a programmer's understanding of a program's semantics to be captured by a checker.

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 about 19 million lines of pure code and accepts about 100,000 patches per year [31]. To help this situation, especially for memory bugs, researchers often use memory checkers in the first place. For example, *SingleStep* [32] is 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 Mohan¹ Ashlie Martinez² Soujanya Ponnampalli¹ Pandian Raju¹
Vijay Chidambaram^{1,2}
¹University of Texas at Austin ²VMware 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 the systems, they can hardly keep up with the rapid increase in the system size and complexity. Finding new bugs being introduced and repaired regularly. These bugs occur in various flavors: simple buffer overflows to sophisticated semantic bugs. Although bug-specific checkers exist, they generally lack a way to explore the system states thoroughly. Here, we propose a methodology to find bugs that underlie the checking process. We highlight the potential of applying fuzzing to find bugs but, in theory, any type of file system testing framework, like the existing framework, FITRA, can be used to find bugs.

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 [1] and btrfs [34] with 59K and 138K lines of code, respectively, whereas XFS [24] and HFS [15] bugs reported in 2018 alone. A bug in a file system can wreak havoc on the user, as it not only corrupts data, but also poses security threats [2, 3, 12, 13, 14]. The entire life cycle of a file system under one umbrella, from design to deployment, is a constant yet essential process. However, manually eliminating bugs is a tedious task.

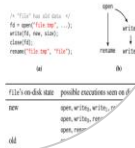
Specifying and Checking File System Crash-Consistency Models

James Bornholt¹ Antoine Kaufmann¹ Bailin Li¹ Arvind Krishnamurthy¹ Emina Torlak¹ Xi Wang¹
¹University of Washington
{bornholt, antoine.k, bailin.li, arvind.krishnamurthy, emina.torlak, xi.wang}@u.washington.edu

Abstract

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

This paper presents crash-consistency models, a methodology for specifying consistency models, which describe the behavior of file systems across crashes. Crash-consistency models are formalized as state transition systems, which demonstrate allowed and disallowed sequences of system operations and operations 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. Clients rely on this contract, making it imperative that these systems hold. Unfortunately, this code is exceptionally hard to get right, even if it must correctly recover from any crash at any program point, so stateless data can be corrupted across reliable and persistent storage. This paper describes EXOGENE, a system that makes it easy to systematically check and manage systems for crashes. EXOGENE, previously generic, checks and manages storage systems as they evolve, using a novel adaptation of EXOGENE to a comprehensive, heavyweight, on-the-fly checking methodology.

checkers to find bugs in crash recovery code, as they run on a live system they tell EXOGENE when to generate the disk images that could occur if the system crashed at the current recovery point, which they then check for errors. We explicitly designed EXOGENE so that clients can check complex storage stacks built from many different subsystems. For example, Figure 1 shows a virtualized system on top of NFS on top of the ZFS file system on top of RAID. EXOGENE makes it easy to checkers for such deep stacks by providing that let users write small, checkers that play them together to build a checker.

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

Changwoo Min¹ Sudhitha Kashyap¹ Byoungyoung Lee¹ Chengyu Song¹ Taesoo Kim¹
¹Georgia Institute of Technology

Abstract

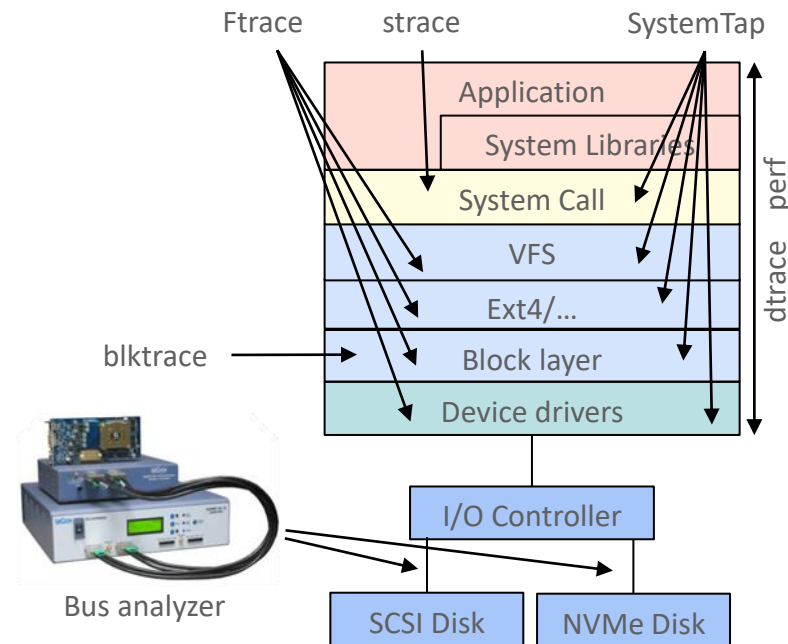
Today, systems software is too complex to be bug-free. To find bugs in systems software, developers often rely on code checkers, like Linux's Sparse. However, the capability of existing tools used in commodity, large-scale systems is limited to finding only shallow bugs that tend to be introduced by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

1 Introduction

Systems software is bugs. 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 about 19 million lines of pure code and accepts about 100 patches per hour [18]. To help this situation, especially for memory bugs, researchers often use memory checkers in first place. For example, Sanitizers are implemented in C++.

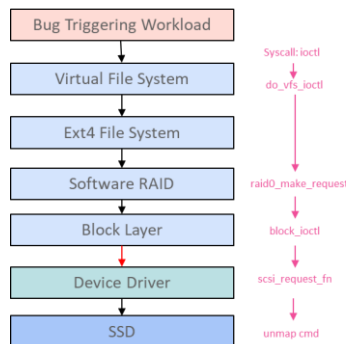
Practical Diagnosis Tools & Limitations

- 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



When Solid State Drives are not that solid

Adam Surak | Jun 15th 2015 | 12 min read | Engineering



algolia [BLOG](#)

[Algolia.com](#)

[Product overview](#)

[Latest releases](#)

[Resources](#)



The level of despair was reaching a critical level and the pages in the middle of the night were unstoppable. We spent a big portion of two weeks just isolating machines as quickly as possible and restoring them as quickly as possible. The one thing we did was to implement a check in our software that looked for empty blocks in the index files, even when they were not used, and alerted us in advance.



algolia [BLOG](#)

[Algolia.com](#)

[Product overview](#)

[Latest releases](#)

[Resources](#)



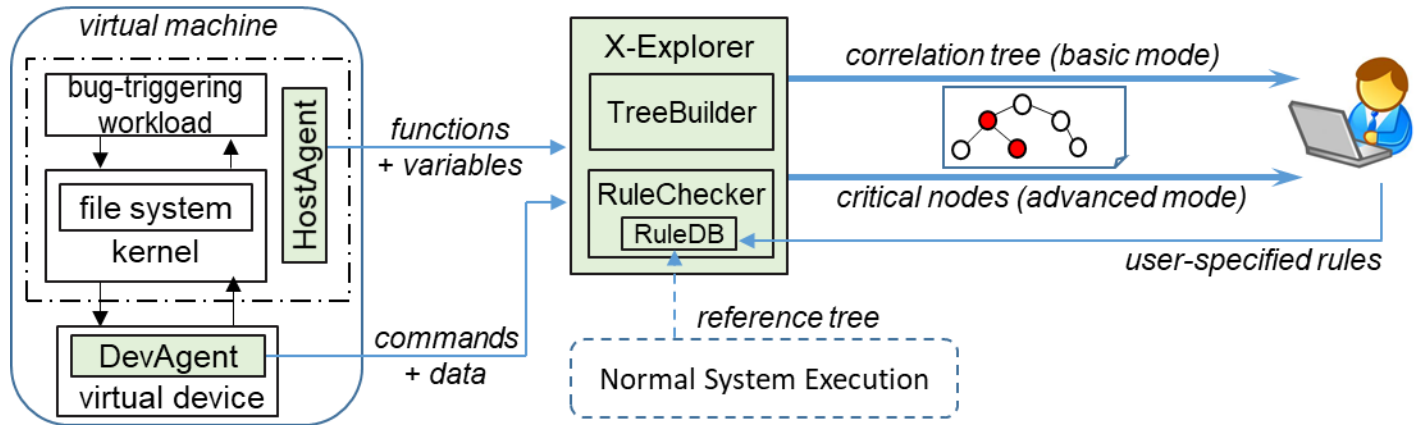
As a result, we informed our server provider about the affected SSDs and they informed the manufacturer. Our new deployments were switched to different SSD drives and we don't recommend anyone to use any SSD that is anyhow mentioned in a bad way by the Linux kernel. Also be careful, even when you don't enable the TRIM explicitly, at least since Ubuntu 14.04 the explicit **FSTRIM** runs in a cron once per week on all partitions – the freeze of your storage for a couple of seconds will be your smallest problem.



Our Approach

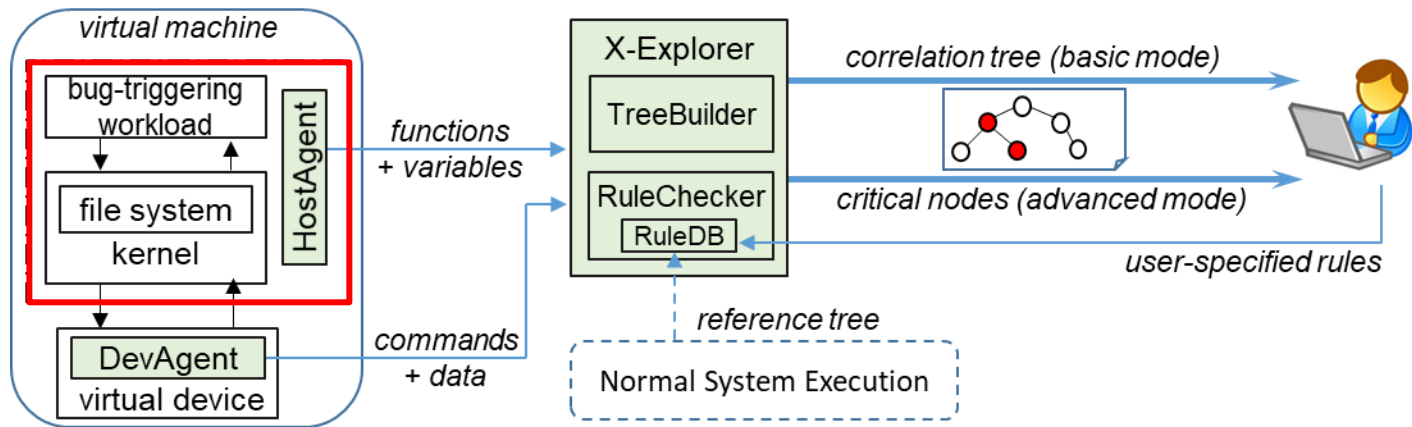
X-Ray: A Cross-Layer Approach

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



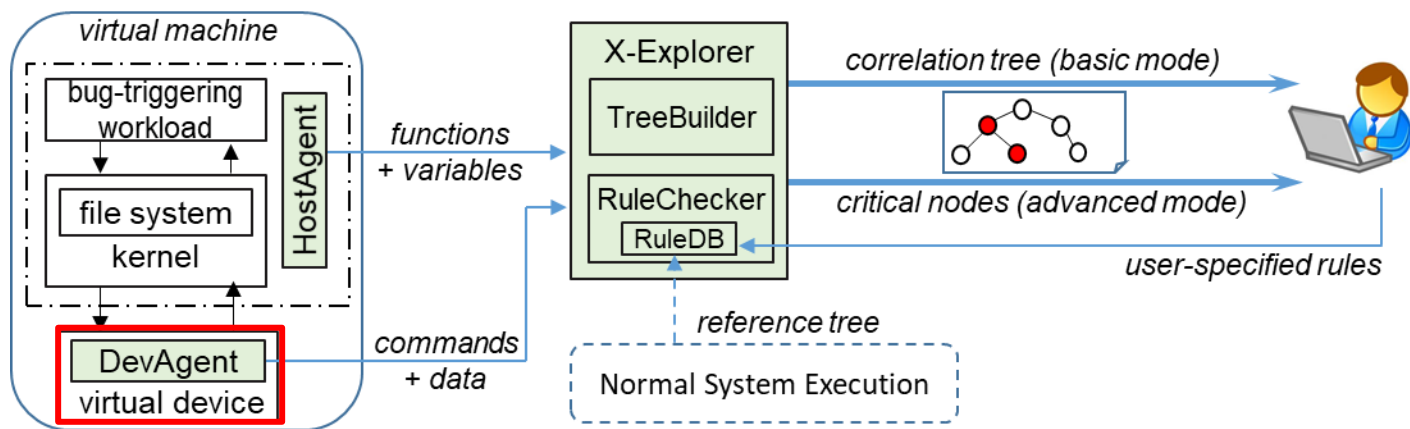
X-Ray: A Cross-Layer Approach

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



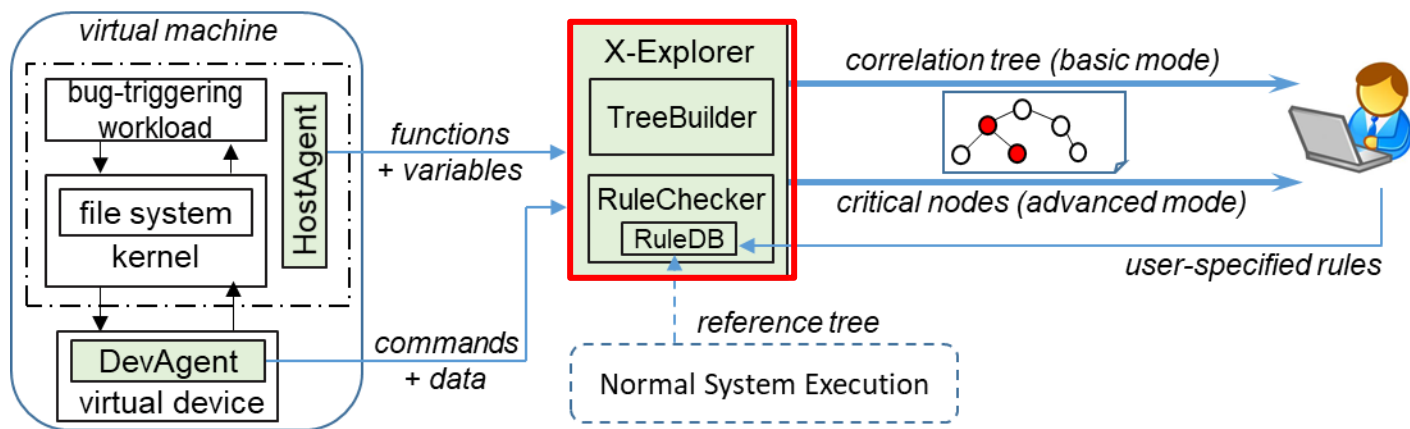
X-Ray: A Cross-Layer Approach

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



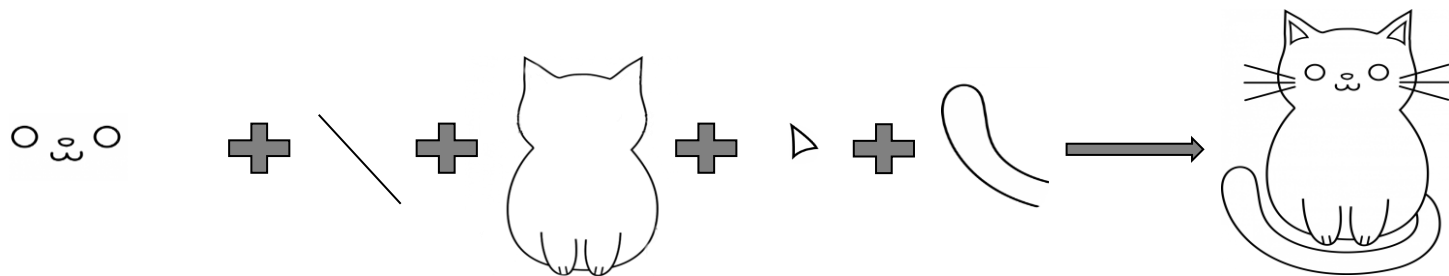
X-Ray: A Cross-Layer Approach

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



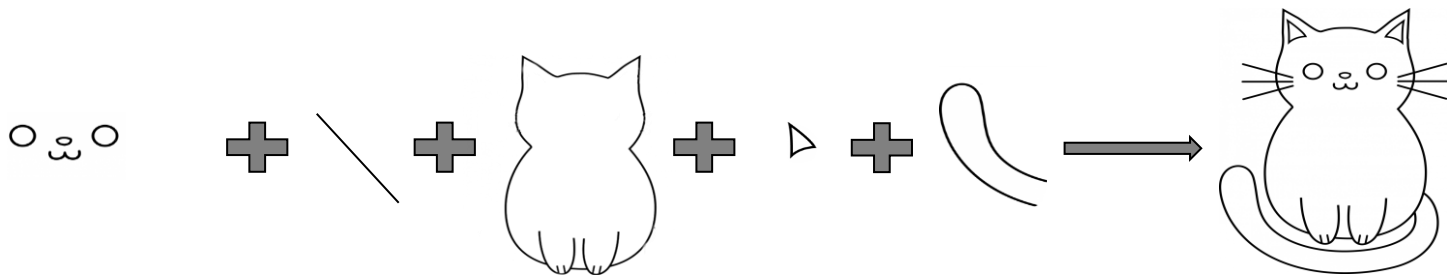
Key Challenge #1

- 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 ?



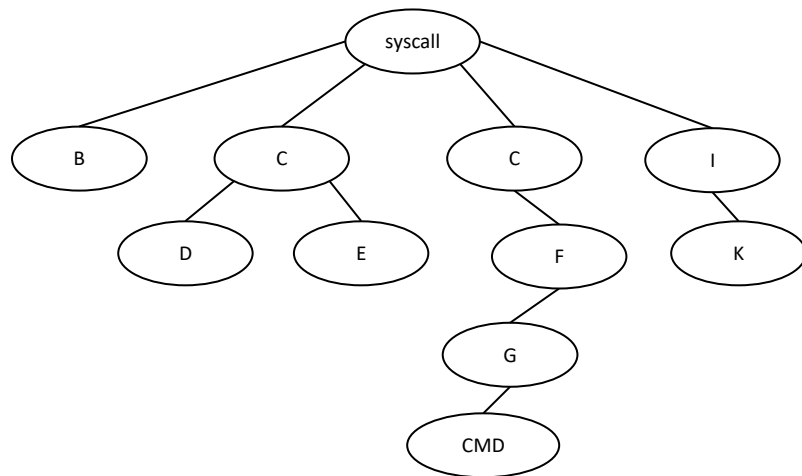
Key Challenge #2

- 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

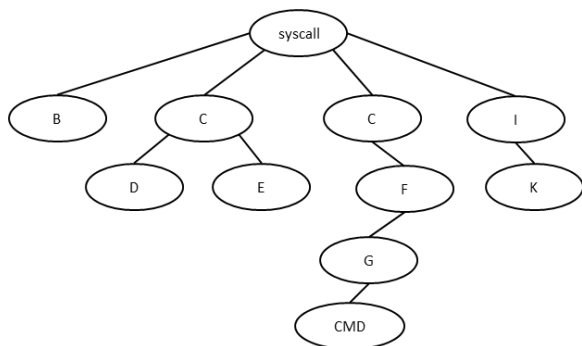


Key Challenge #2

- 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

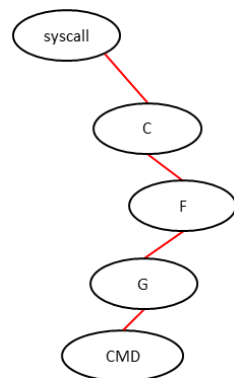
- How to reduce manual efforts ?
 - Visualize cross-layer events & dependencies in a correlation tree
 - Automatically narrow down the root cause via rules
 - Rules specified by users (e.g., “ancestors of device commands”)



Correlation tree



Rule specified by users

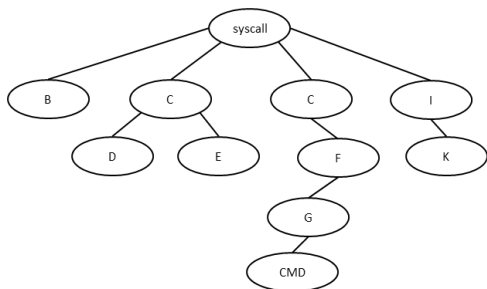


Critical part

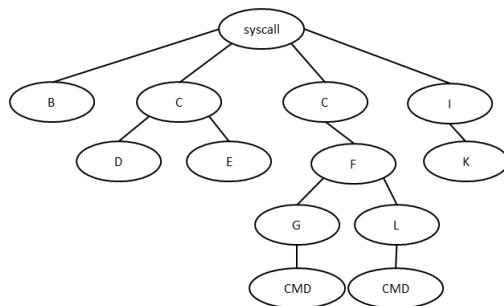


Key Challenge #2

- 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)

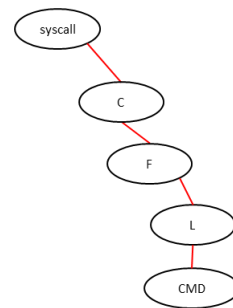
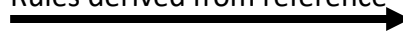


Tree from Abnormal execution



Tree from reference execution

Rules derived from reference



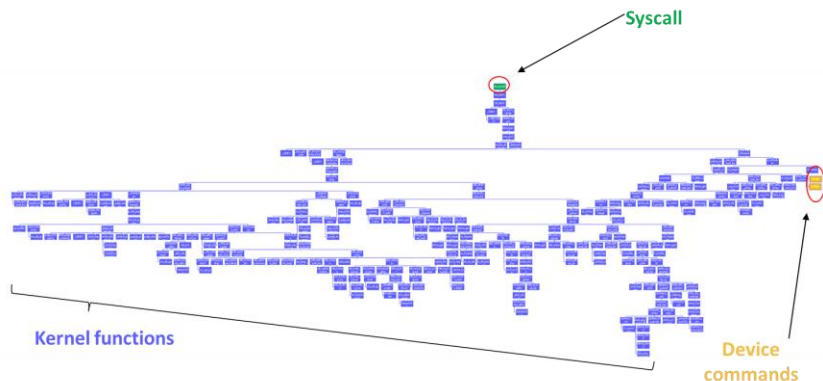
Difference part



Preliminary Results

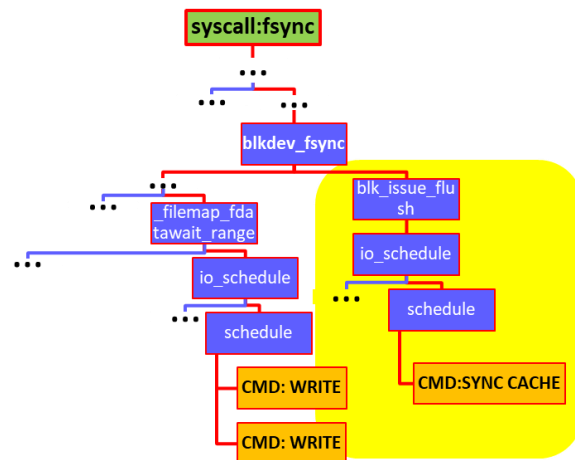
Preliminary Results

- 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>

