FAST'24

ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL

ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

Metis: File System Model Checking via Versatile Input and State Exploration

22nd USENIX Conference on File and Storage Technologies (FAST 2024)

Yifei Liu[1], Manish Adkar[1], Gerard Holzmann[2], Geoff Kuenning[3], Pei Liu[1], Scott A. Smolka[1], Wei Su[1], and Erez Zadok[1]

[1] Stony Brook University; [2] Nimble Research; [3] Harvey Mudd College

Outline

- **Background and Motivation**
- Metis Design
- RefFS Design
- Evaluation
- Conclusions

Background: File System Testing

- File system bugs: widespread and serious
- Various testing techniques invented

Regression Testing	Model Checking	Fuzzing	Automatic Test Generation
Linux Test Project xfstests fsx exerciser	FiSC (OSDI '04) eXplode (OSDI '06) MCVFS (VMCAI '09)	Syzkaller Janus (S&P '19) Hydra (SOSP '19)	B3 (OSDI '18) Dogfood (ICSE '20) Chipmunk (EuroSys '23)
Ensuring updates preserve existing functionality	Verifying file system correctness against an abstract model	Finding bugs or crashes through semi-random inputs	Automatically creating test workloads to check file systems

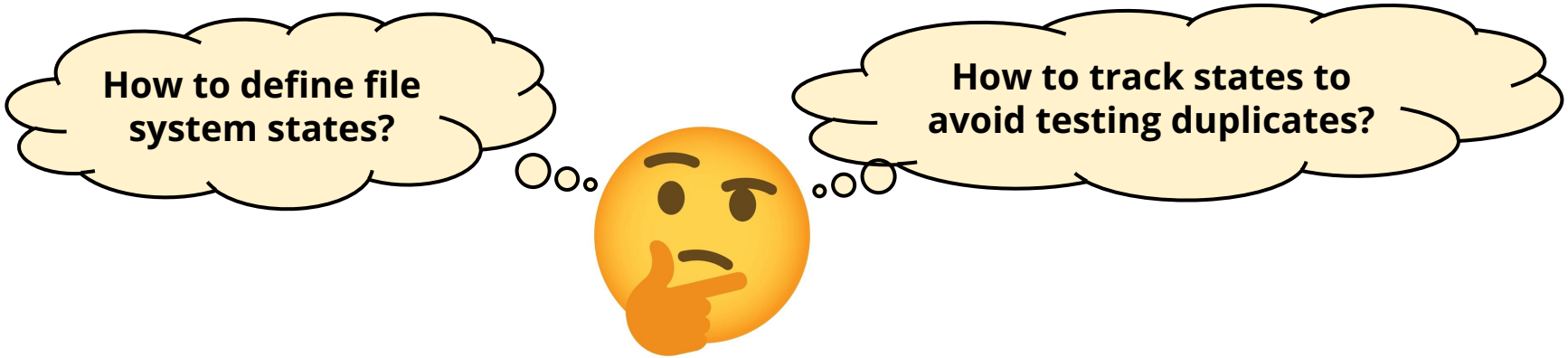
Background: File System Test Inputs

- Large test input space for file systems
  - **Linux:** over 400 system calls, only a handful for file systems
  - **Input space:** various arguments, arbitrary values, combinations
- Maximizing value from the input space
  - Diverse selection of test inputs
  - Tailoring of test input distribution based on strategy
- Input coverage for file system testing[1]
  - **Completeness:** covers enough different inputs
  - **Versatility:** designs test cases to achieve desired input coverage

[1] Liu, Yifei, et al. “Input and Output Coverage Needed in File System Testing”, ACM HotStorage, 2023.

Motivation: File System States

- Inputs should be assessed with file system states
  - E.g., writing to existing vs. brand-new file
- **Ideal case:** test diverse inputs across various states
  - Cover corner cases
  - Don't waste resources by revisiting states



Outline

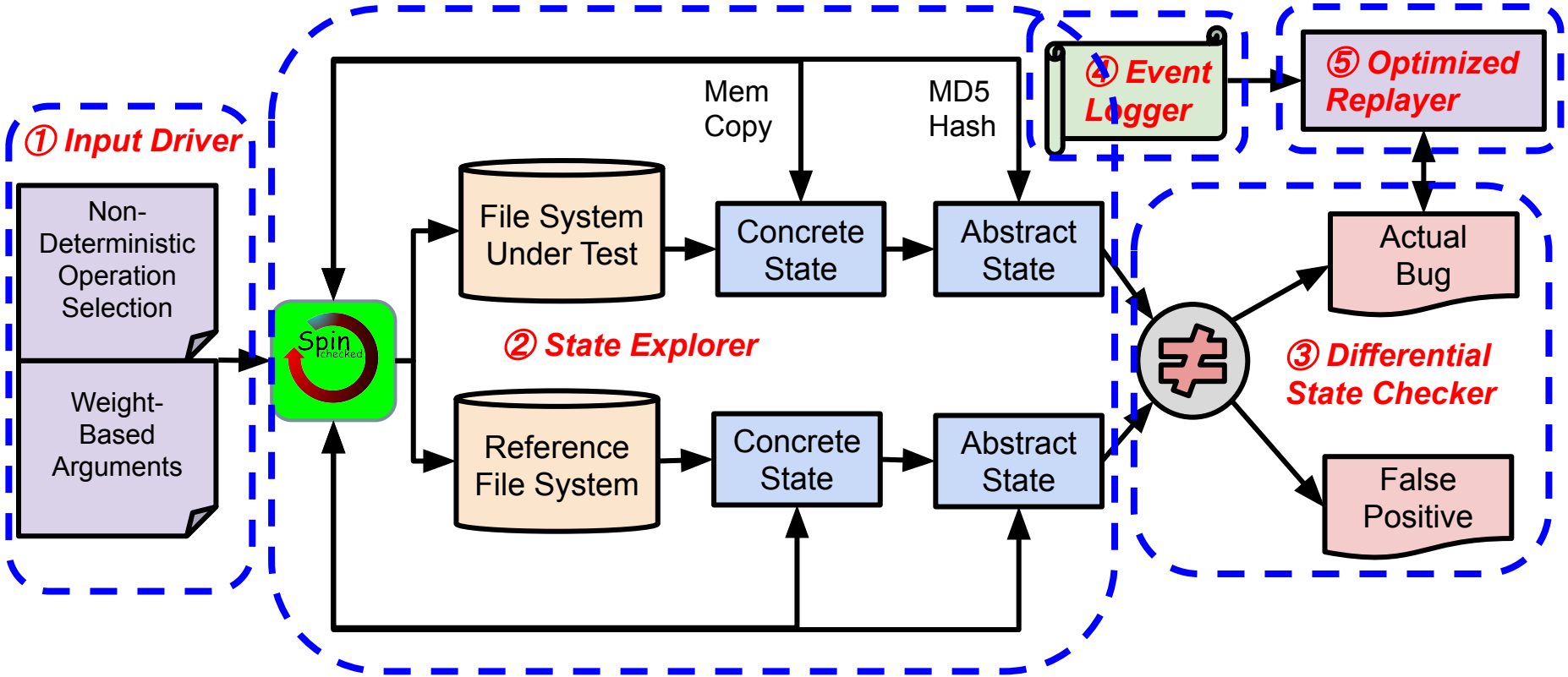
- ~~Background and Motivation~~
- **Metis Design**
- RefFS Design
- Evaluation
- Conclusions

Our Work: Metis

Metis: Combines ***model checking*** & ***differential testing***

1. Achieve both input and state coverage
2. No need to create an abstract model
3. No need to modify or instrument OS kernel
4. Simplify bug reproduction
5. Scale up testing with resources

Metis Architecture

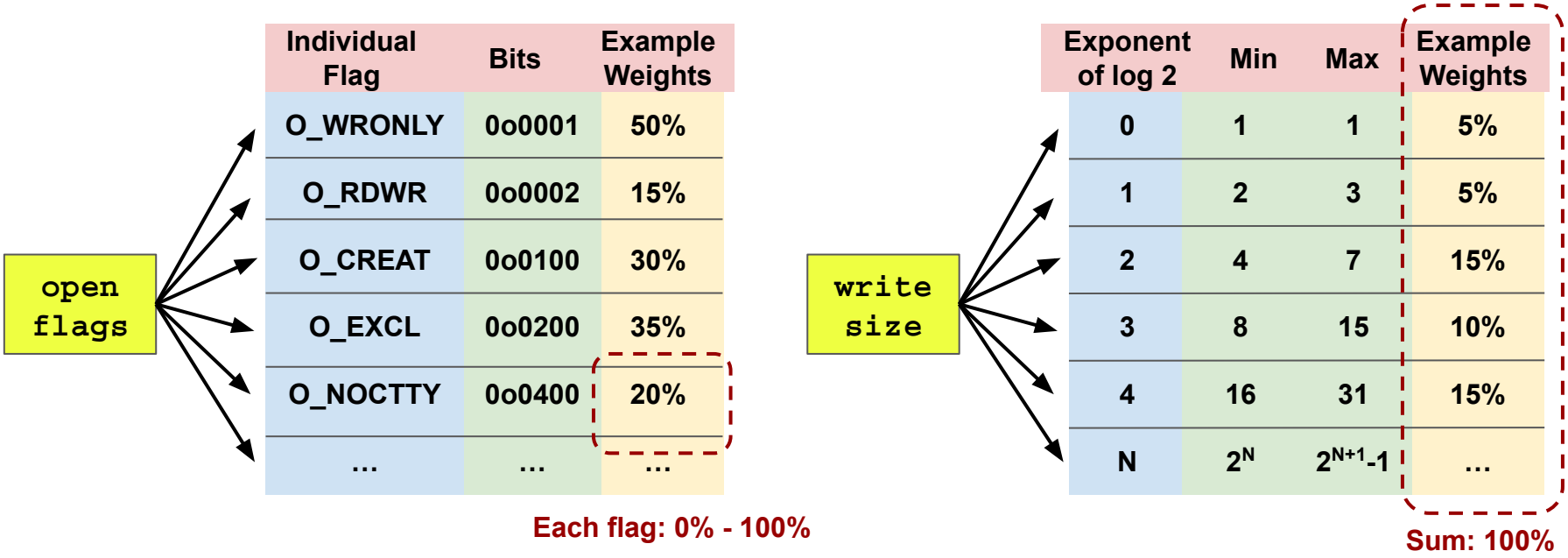


Metis Design: Input Driver

- Metis file-system operations: **meta-operations** and **single syscalls**
- **Syscall Arguments:** input space partitioning
  - Divided arguments into four categories
    - **Identifiers:** e.g., file descriptors
    - **Bitmaps:** e.g., `open` flags
    - **Numeric arguments:** e.g., `write` size
    - **Categorical arguments:** e.g., `lseek` whence
  - Partitioned the input space using type-specific methods

Metis Design: Input Driver (cont.)

- Set probabilities (as weights) for each partition

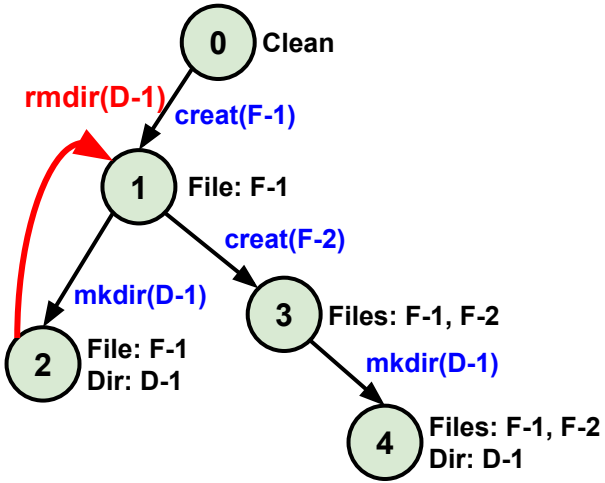


Metis Design: State Explorer

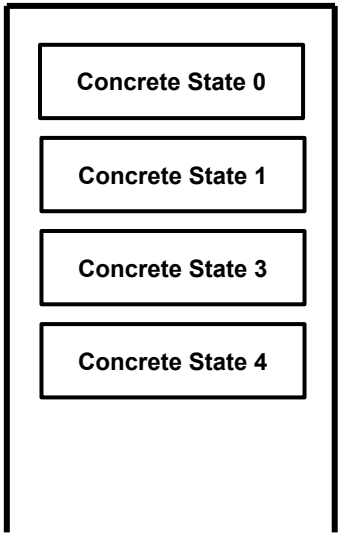
- **State-space exploration:** Depth-First Search (DFS)
  - FS states are the nodes, FS operations are the edges
- FS state definition and tracking
  - **Concrete state**
    - All file system state information
    - For state backtracking and bug reproduction
  - **Abstract state**
    - MD5 hash of file content, directory tree, important metadata
      - Exclude noisy attributes, e.g., `atime` timestamps
    - For identifying and comparing system states
    - Discrepancies are potential bugs

Metis in Action: State Exploration

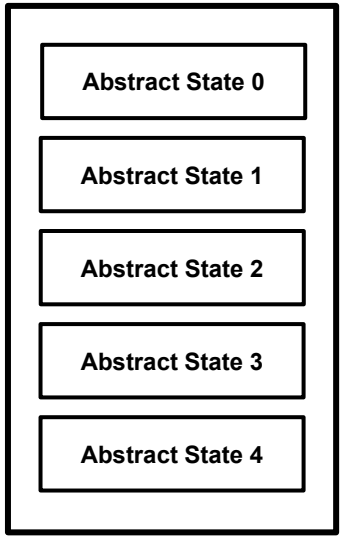
Metis DFS Tree



Concrete State (stack)

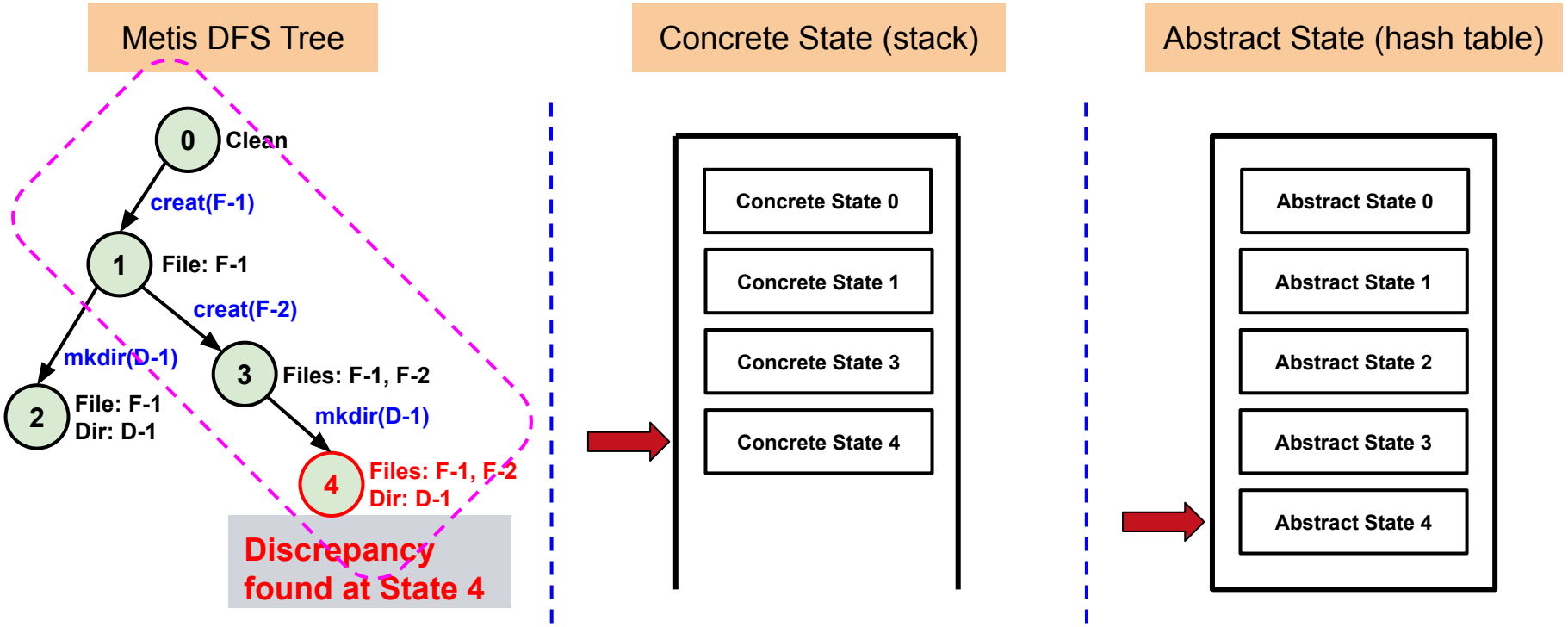


Abstract State (hash table)



If an operation makes file systems reach a previously visited state, Metis reverts the state to the parent state

Metis in Action: State Exploration



Metis replayer can reproduce a potential bug from any point of the exploration path by using concrete states and logs

Tracking Full File System States

- **Save** and **restore** states on backtrack or search limit
  - As a user process, cannot track in-memory file system states

Solutions	Kernel FS	User-space FS
VM Snapshotting	❌ Too slow	
Process Snapshotting	❌ Not applicable	❌ Incompatible with character device
Remount / unmount the file systems before / after each operation	❌ Slow ❌ Hide bugs related to in-memory states ✅ Compatible with all on-disk file systems	
State Save/Restore (SS/R) API	✅ Efficient and preserves in-memory states	
	❌ Challenging to implement on kernel FS	✅ Feasible and less challenging

Parallel State Exploration

- State space is bounded yet huge
  - Exploring with single process is time-consuming
- **Metis uses Swarm verification:**[2] "divide and conquer" the state space
  - **Parallel Verification Tasks (VTs):** check segments of state space
  - VTs scale across CPU cores and machines
  - **Diversification:** techniques that help ensure VTs explore different parts of the space
    - By different combinations of state-exploration parameters

[2] Holzmann, Gerard J., et al. "Swarm verification techniques", IEEE Transactions on Software Engineering, 2010.

Outline

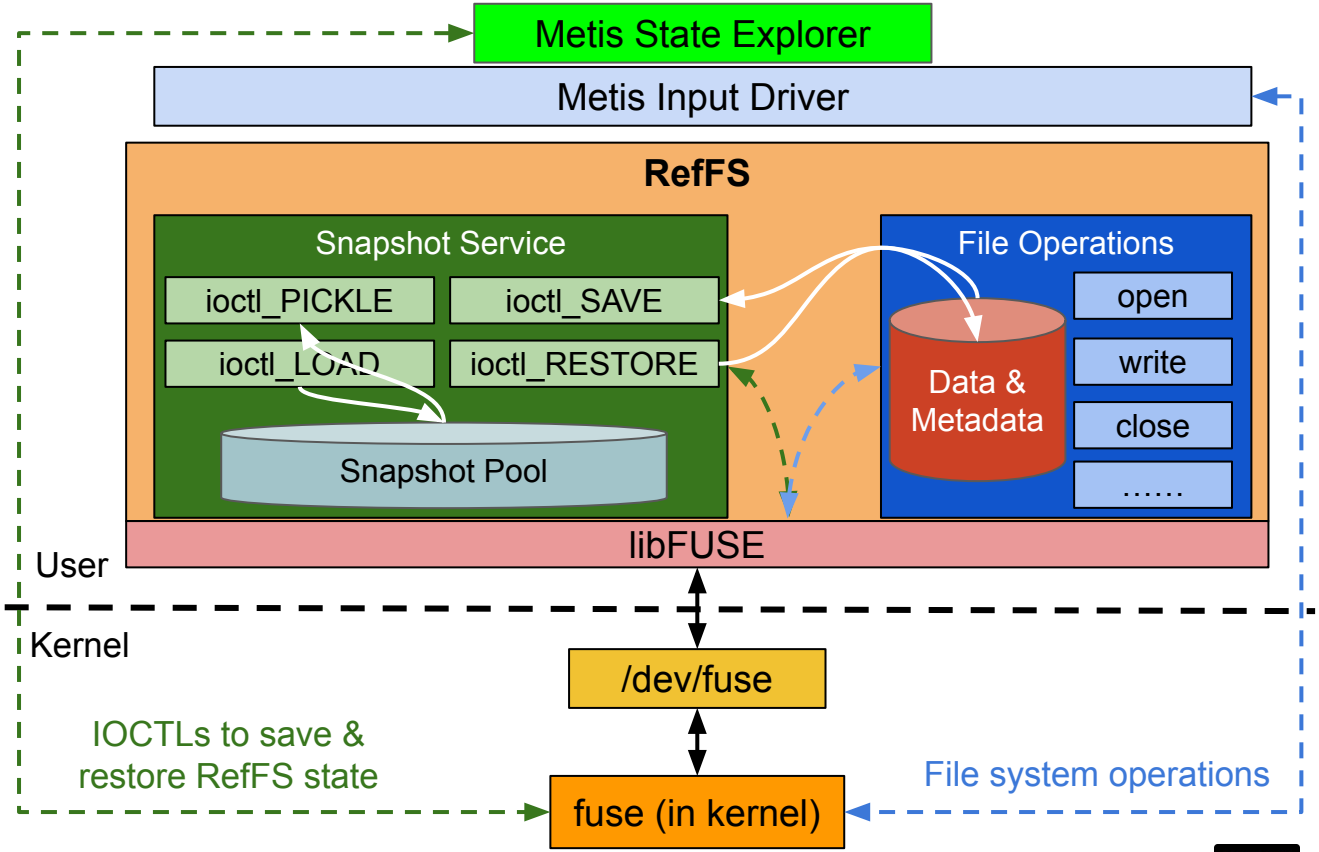
- ~~Background and Motivation~~
- ~~Metis Design~~
- **RefFS Design**
- Evaluation
- Conclusions

RefFS: The Reference File System

- Reference file system must exhibit correct behavior
- Tried Ext4 as initial reference file system
  - Lacks state save/restore (SS/R) operations
  - Difficult to debug and verify due to its complexity
- **RefFS:** new file system designed to function as reference file system
  - Small, user-space, easy to debug
  - Optimized for SS/R via four snapshot `ioctl` APIs
  - Thoroughly checked and improved RefFS by using Metis

RefFS Architecture and its Snapshot API

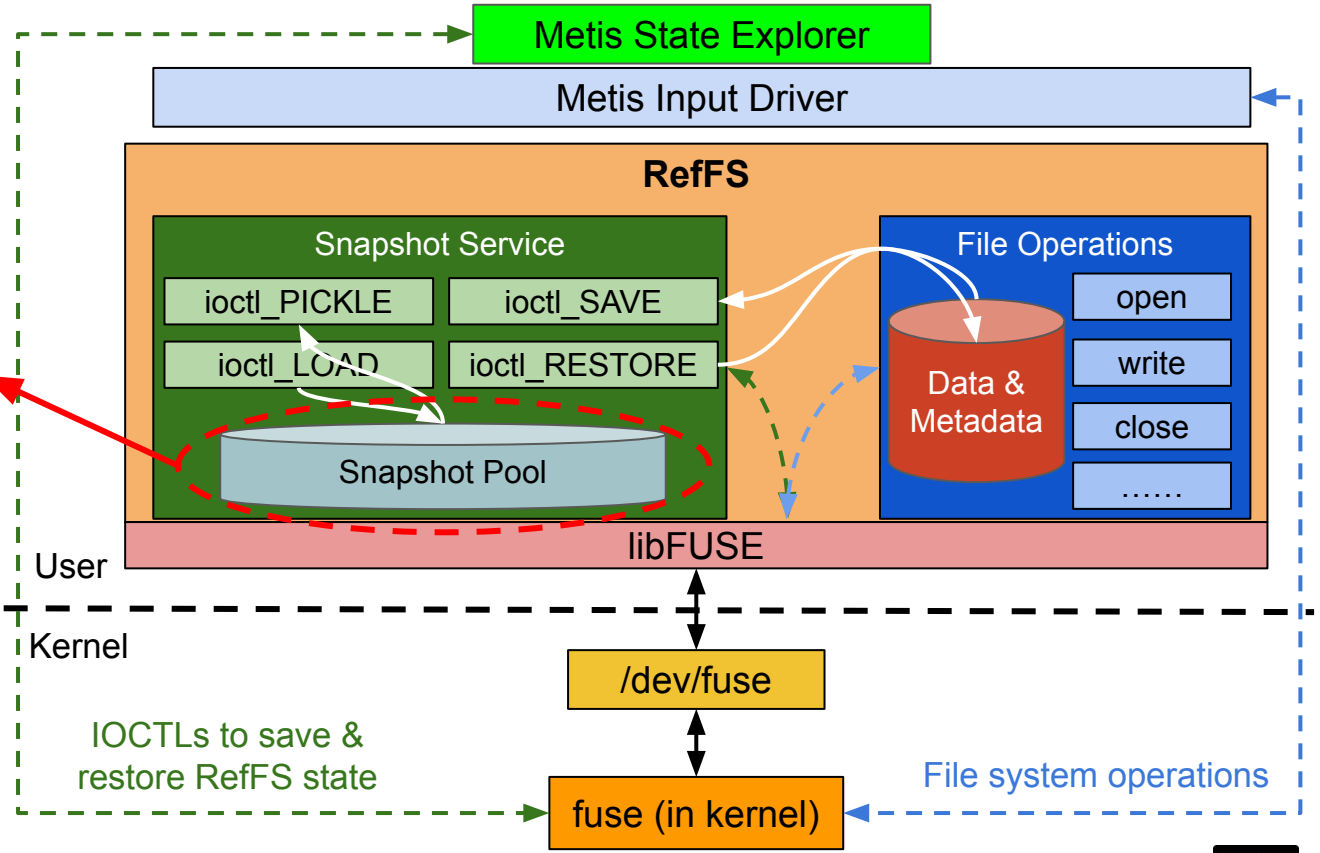
RefFS:
FUSE-based, in-memory, supports most POSIX file system operations



RefFS Architecture and its Snapshot API

Snapshot Pool:
Key-value store of snapshots

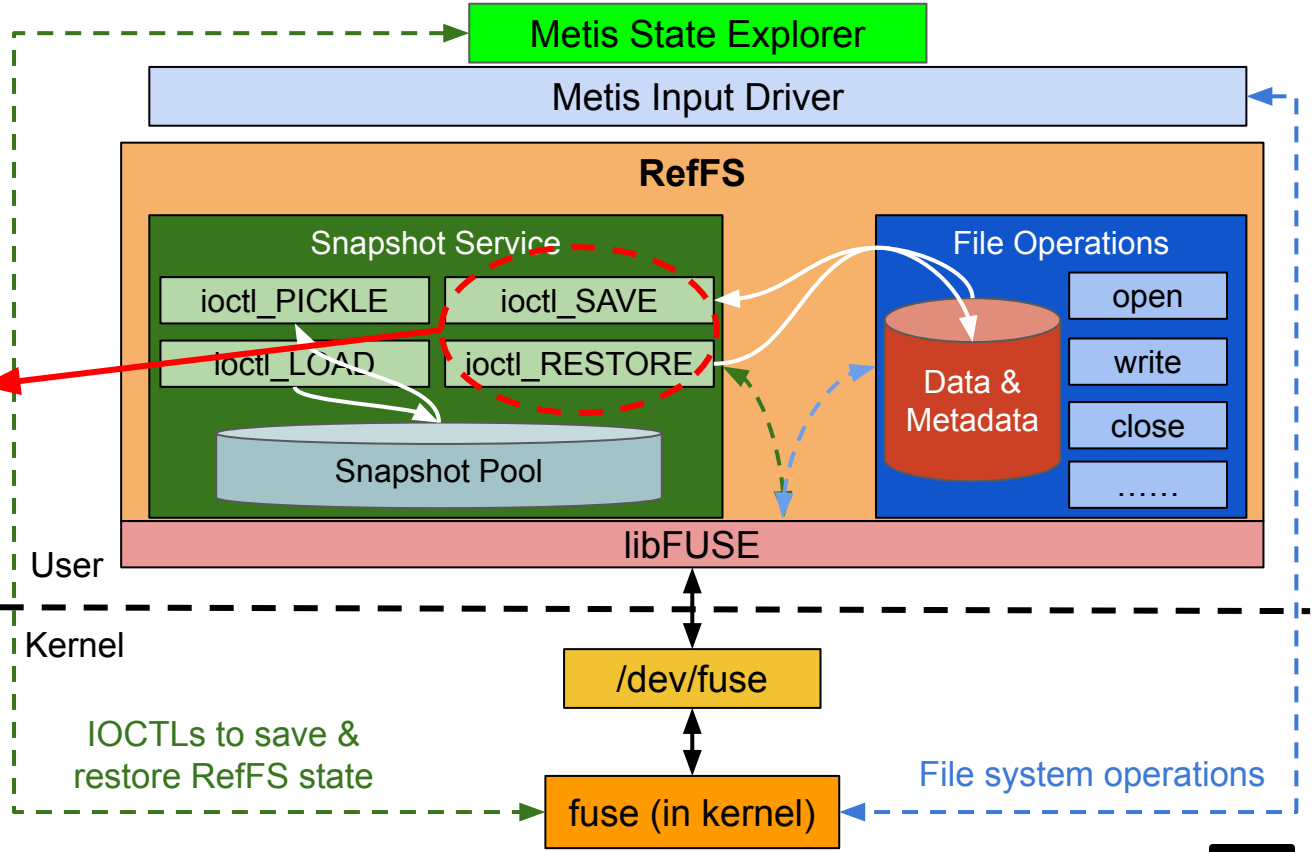
64-bit key (DFS tree location) and corresponding file system snapshot



RefFS Architecture and its Snapshot API

`ioctl_SAVE:`
Saves current concrete file system state as a snapshot

`ioctl_RESTORE:`
Restores file system state from a snapshot



Outline

- ~~Background and Motivation~~
- ~~Metis Design~~
- ~~RefFS Design~~
- **Evaluation**
- Conclusions

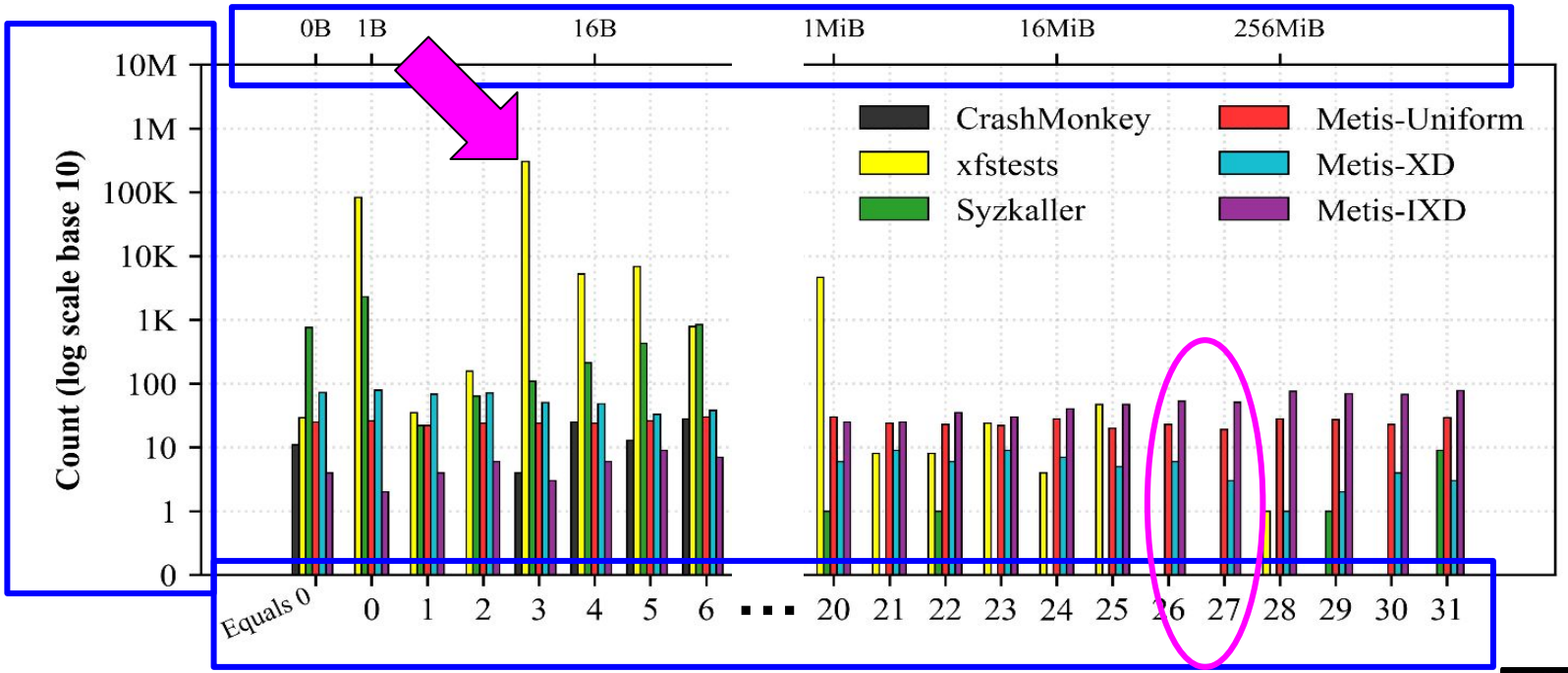
Evaluation: Experimental Setup

- **Hardware Platform**
  - Ubuntu 22.04, dual 6-core Intel Xeon X5650 CPUs, 128GB RAM, 128GB NVMe SSD for swap space
  - Swarm verification execution: 3 identical machines
- **File Systems**
  - **Ext4** (reference to check RefFS), **RefFS** (reference to check others)
  - BetrFS,[3] BtrFS, F2FS, JFFS2, JFS, NILFS2, NOVA, PMFS, XFS
- **Complementary Tools**
  - IOCov [Liu 2023]: Computes input coverage for file system testing
    - **Comparison:** CrashMonkey, xfstests, Syzkaller, Metis
  - RAM disks: Serve as devices for on-disk file systems

[3] Jiao, Yizheng, et al. “BetrFS: A Compleat File System for Commodity SSDs”, EuroSys, 2022.

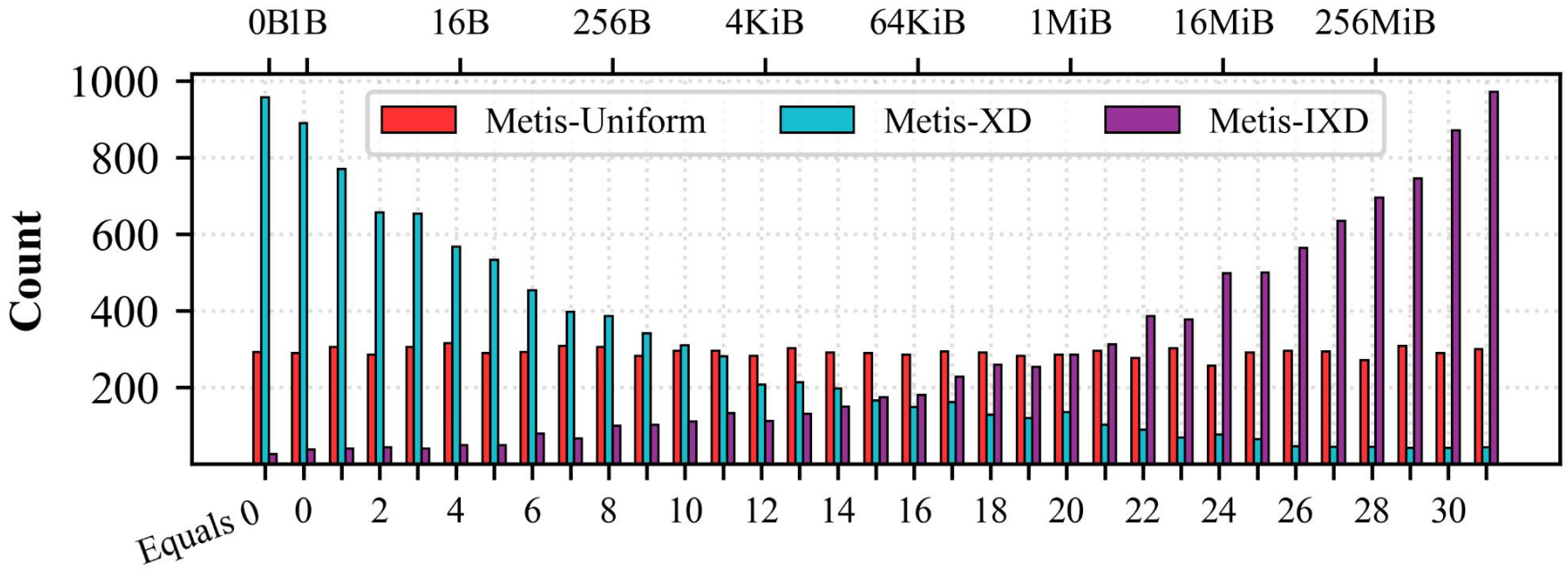
Input Coverage: `write()` sizes [40 mins]

- **Metis-Uniform:** uniform test probabilities to each write size partition
- **XD:** exponentially decaying; **IXD:** inverse exponentially decaying

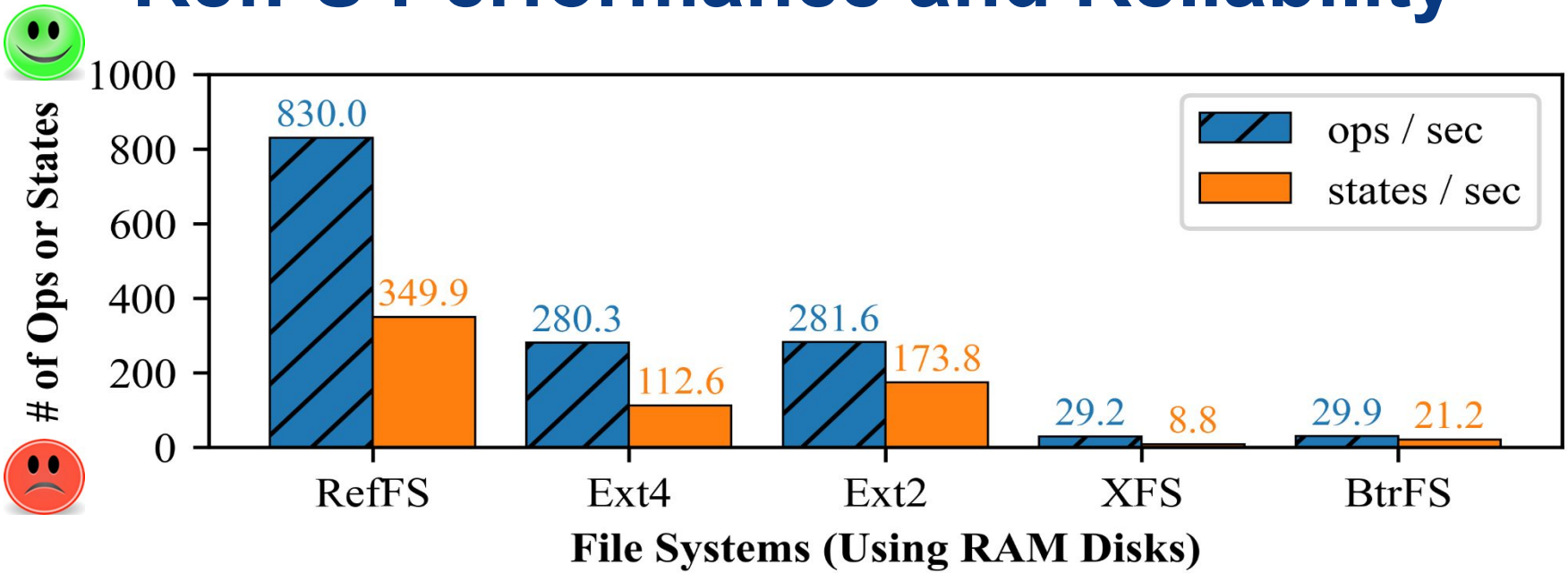


Input Coverage: `write()` sizes [4 hours]

- **4-hour Metis run:** with a longer run, the expected distributions are more accurate



RefFS Performance and Reliability



RefFS explores states 3–28× faster than other mature file systems

Using Ext4 as the reference, we used Metis to find and fix 11 bugs in RefFS

Compared RefFS and Ext4 for 1 month across 18 VTs with > 3B ops, w/o any discrepancy

Evaluation: Bug Finding

- Checked nine existing file systems; identified bugs in seven
- Discovered and confirmed various types of file system bugs

File System	Total Bugs	Deterministic	Reported & Confirmed	New Bugs
BetrFS	3	3	3	2
F2FS	1	0	0	1
JFFS2	3	2	2	2
JFS	2	1	0	2
NILFS2	3	3	0	3
NOVA	2	1	1	2
PMFS	1	0	0	1
Total	**15**	**10**	**6**	**13**

Outline

- ~~Background and Motivation~~
- ~~Metis Design~~
- ~~RefFS Design~~
- ~~Evaluation~~
- **Conclusions**

Conclusions

- File system testing must consider both input and state space
- New model-checking framework, Metis, achieves thorough and versatile coverage of both input and state
- RefFS, an efficient FUSE-based in-memory file system, serves as the reference for Metis
- Evaluation demonstrates the performance and effectiveness of Metis and RefFS
- Found 15 bugs across seven file systems; six were confirmed, and 13 were previously unknown

Metis: File System Model Checking via Versatile Input and State Exploration

Thank You!

Q&A

Metis and RefFS are open-sourced at

https://github.com/sbu-fsl/Metis

https://github.com/sbu-fsl/RefFS

yifeliu@cs.stonybrook.edu