A Flexible Framework for Expediting Bug Finding by Leveraging Past (Mis-)Behavior to Discover New Bugs

Sanjeev Das,

Kedrian James, Jan Werner, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose

THE UNIVERSITY of NORTH CAROLINA at CHAPEL HILL

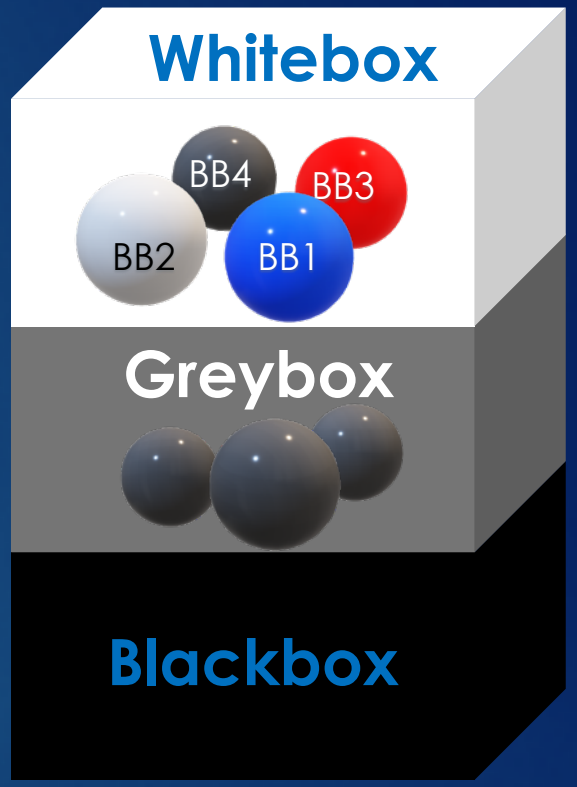
IBM

Fuzzing

Fuzzing is the preeminent technique for finding vulnerabilities

- Random input testing [Miller et al., ACM com.'90]

The success of greybox fuzzers in large stems from the use of a **feedback loop** to prioritize the inputs



Fuzzing Approaches

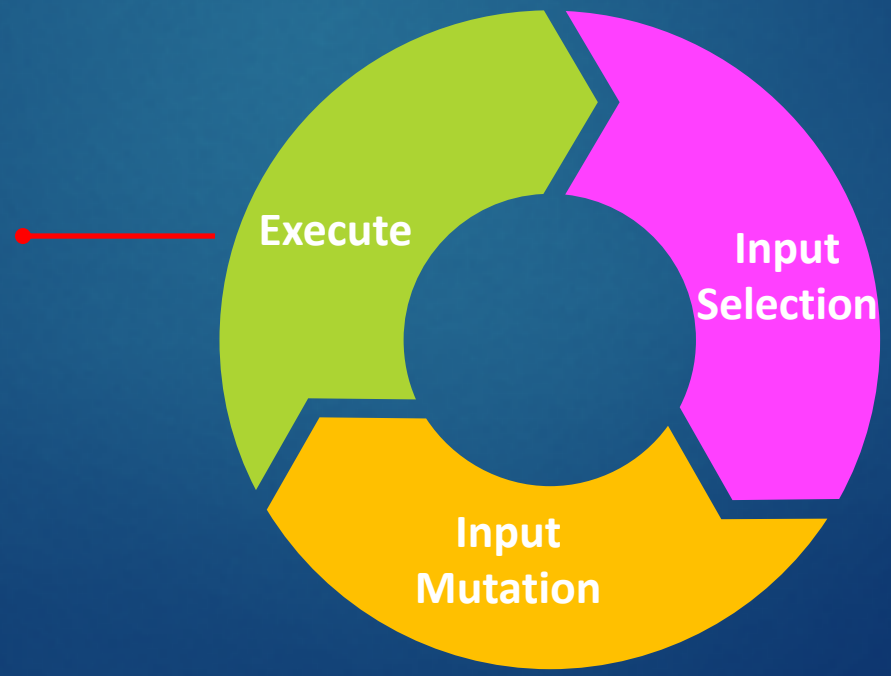
Coverage Guided Fuzzing (CGF)

Prominent grey-box fuzzing approach

- Aims to increase program coverage by exploring new paths to eventually reach buggy paths

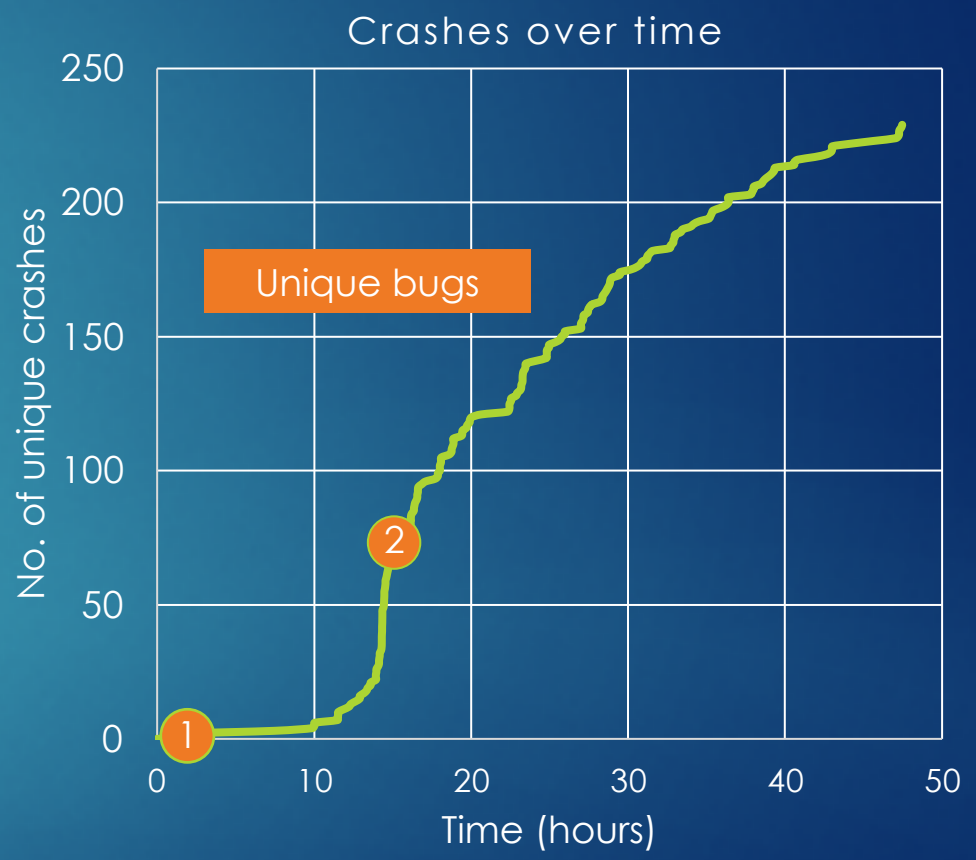
Branch Coverage

Tracks branch coverage to steer input selection



Fuzzing in Practice

Seed collection	Large (diverse) input corpus
Fuzz framework	Spin off fuzzers (CPU dollars)
Crash collection	Obtain large no. of crashes
Crash deduplication	Are these crashes unique?
Bug analysis	Is the bug critical?
Exploitability	Is it exploitable?



*"Improving bug finding **linearly** requires **exponentially** more computational power" [1]*

[1] Boosting fuzzer efficiency: An information theoretic perspective [ASE'20]

Our Focus

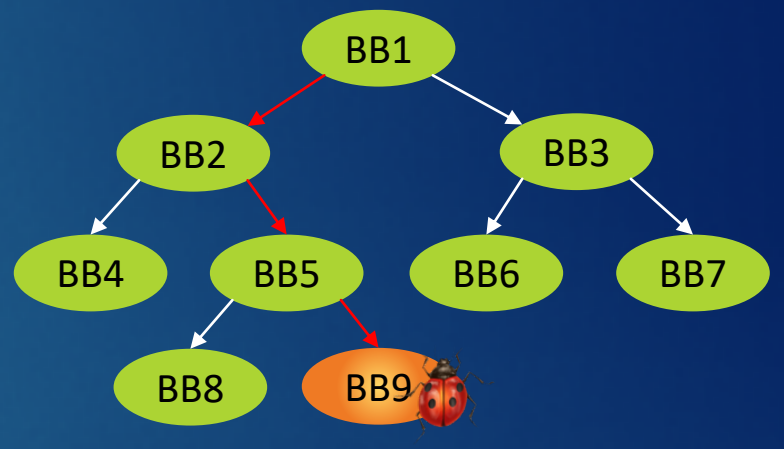
Seed collection	Large (diverse) input corpus
Fuzzing framework	Spin off fuzzers (CPU dollars)
Crash collection	Obtain large no. of crashes
Crash deduplication	Are these crashes unique?
Bug analysis	Is the bug critical?
Exploitability	Is it exploitable?

Limitations of Coverage Guided Fuzzing

Contemporary CGF approach is inefficient

1. Equal priority to all the program paths

- Input that triggers new branch edge is considered interesting
- All branch edges have equal weight
- **Triggers similar paths resulting into similar crashes**
- Slows bug discovery



[BB1 → BB2 → BB4] = [BB1 → BB2 → BB5]

[1] Boosting fuzzer efficiency: An information theoretic perspective [ASE'20]

Limitations of Coverage Guided Fuzzing

Contemporary CGF approach is inefficient

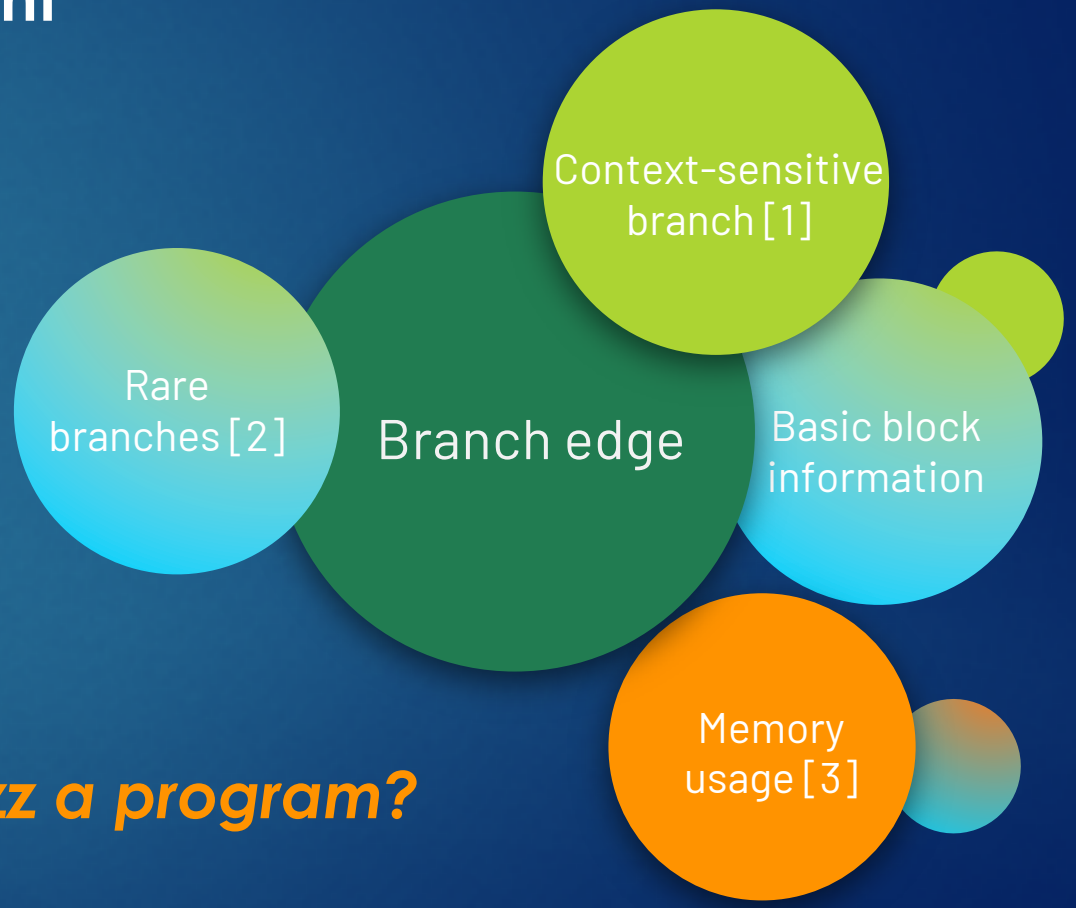
2. Coverage metric is limited

- Most approaches track branch edges
- But lack contextual information

- Recent works utilize more information

- *"No grand slam coverage metric that outperforms all the others"* [1]

- ***What signal can we use to efficiently fuzz a program?***



[1] Be Sensitive and Collaborative: Analyzing Impact of Coverage Metrics in Grey-box Fuzzing [RAID'19]
[2] FairFuzz: A Targeted Mutation Strategy for Increasing Greybox Fuzz Testing Coverage [ASE'18]
[3] Memlock: Memory usage guided fuzzing [ICSE'20]

Our Approach

- **Microarchitectural information as a signal to build a coverage metric**
- **Vulnerability-aware input selection**
- **Configurable input selection strategy**

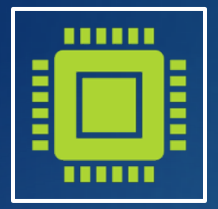
Our Approach

1

Microarchitectural information as a signal to build a coverage metric for a program

- Hardware Performance Counters (HPCs) are special set of registers
- Monitor and measure hardware events
  - Instruction, memory, cache, translation look-aside buffer
  - Deeper code path - retired branches/misses
  - Rare branches - branch misses
  - Memory information – load/store
- Widely used in debugging, profiling, performance analysis, security
- **Low performance overhead**

Challenges



There are *many* HPC events, but only limited no. of events can be monitored simultaneously

Due to limited no. of CPU counters

Which events should we choose?

How can we use HPC information to guide fuzzing?

Observations

"No clear evidence that the vulnerability rate, vulnerability types of widely used software decrease over time" **[Alexopoulos et al., TOPS'20]**

Code analysis platform based on known vulnerabilities has been popular, e.g., CodeQL

Our Approach

2 Vulnerability-aware selection of a coverage metric

- Bugs in the earlier version of a program provide a good signal to find **similar** or **dissimilar** bugs in the new version
- ***Similar coding practices can lead to the similar mistakes being repeated in the new version*** *[Murtaza et al., J Syst Softw'16]*

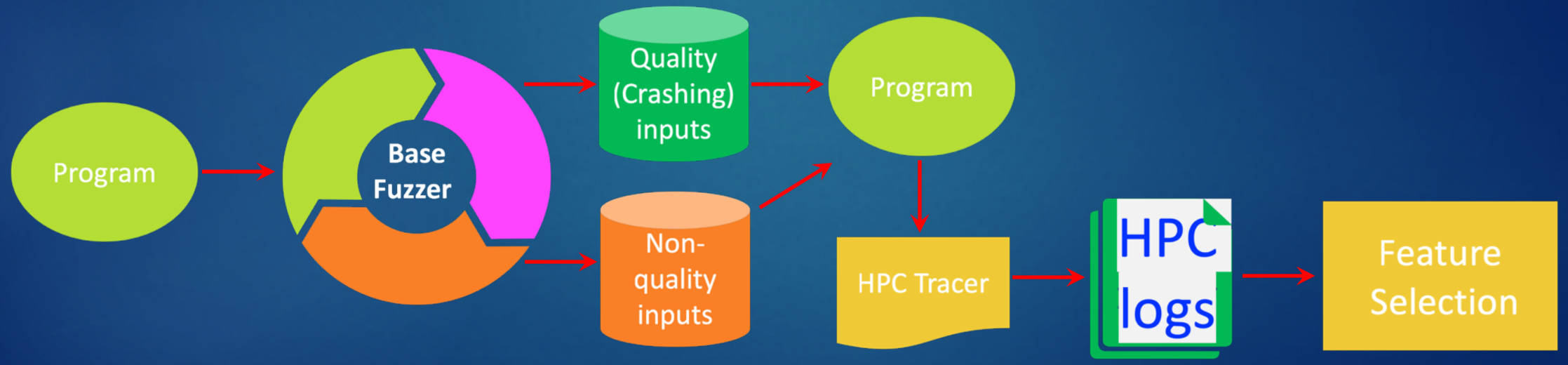
Our Approach: OmniFuzz

Pre-processing	Model Building	Vulnerability Exploration
Data collection	Feature selection	Configurable input selection strategy
Crashing & non-crashing inputs	HPC events selection	Fuzzing campaign
Profile HPC events	ML based heuristic	

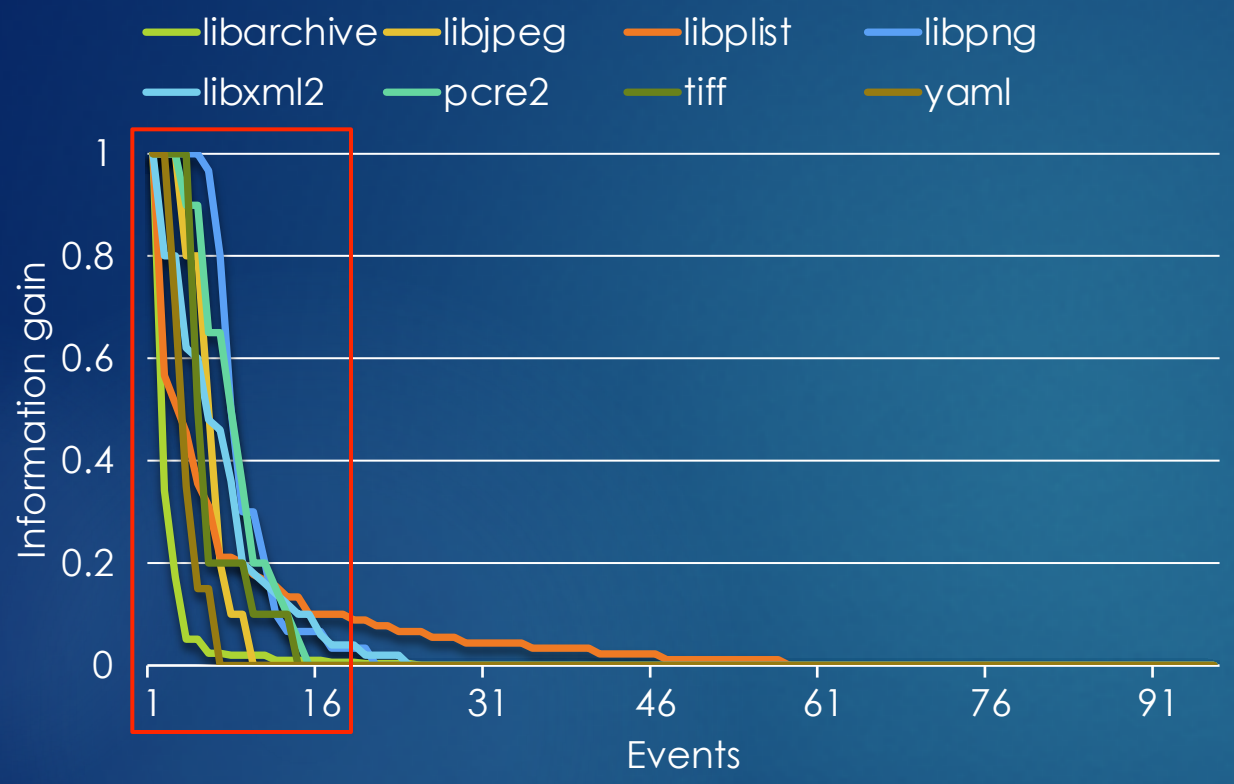
Feature Selection

HPC event selection

- We learn from vulnerabilities in the prior version of program
- We profile 96 HPC events
- Only 4 events can be configured & monitored simultaneously



Feature Selection



libjpeg

- Memory instructions
- DTLB load misses
- Arithmetic operations
- TLB flush

libpng

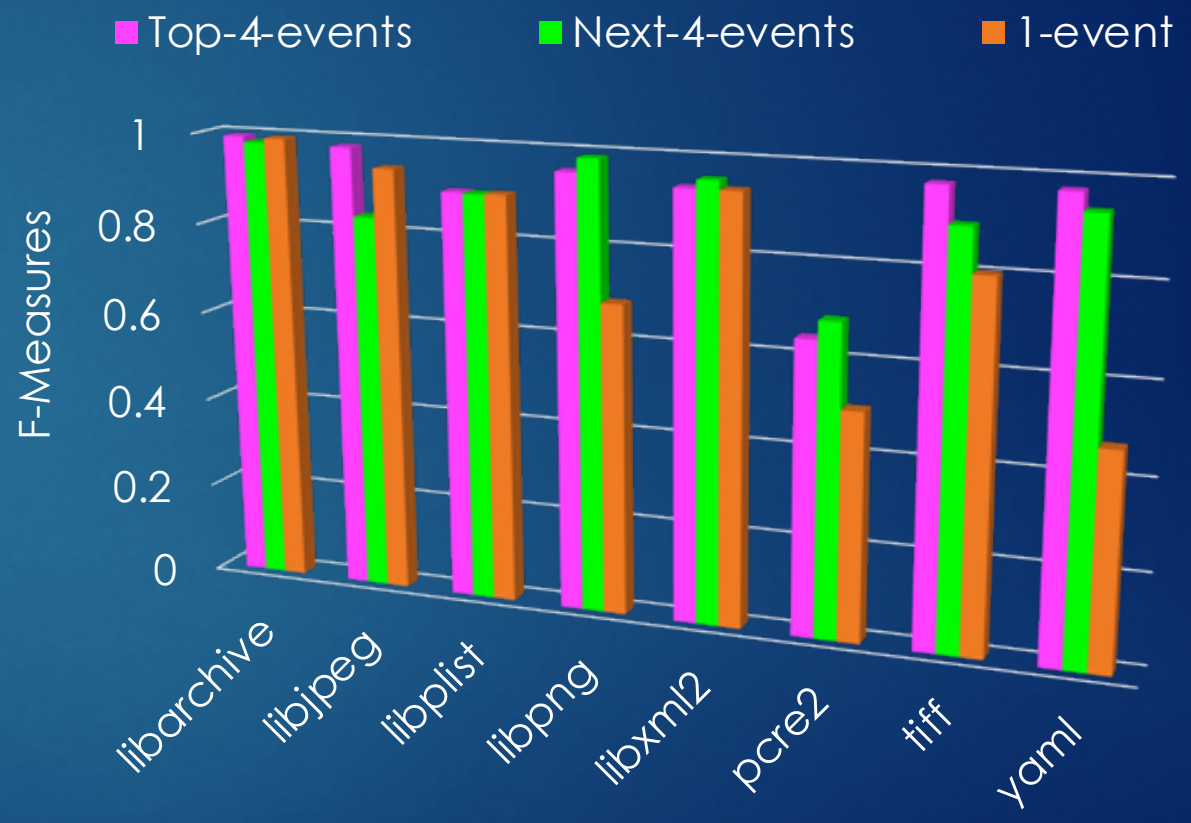
- Memory requests
- Branch misses
- Floating operations
- Memory load uops

- A few events (8-16) have a high **information gain** in identifying crashing and non-crashing inputs
- ***Most notably, top events differ across various programs***

Model Building

Heuristic generation

- We build machine learning classifier to predict crashing and non-crashing inputs
- Model with 4 HPC events perform better than one event
- ***Top 4 events and next 4 events have higher classification accuracy***
- We choose ***multilayer perceptron (MLP)*** for its operational benefits.



Our Approach

3 Configurable input selection strategy

On-the-fly crash deduplication as an **active feedback** mechanism to guide the fuzzing strategy

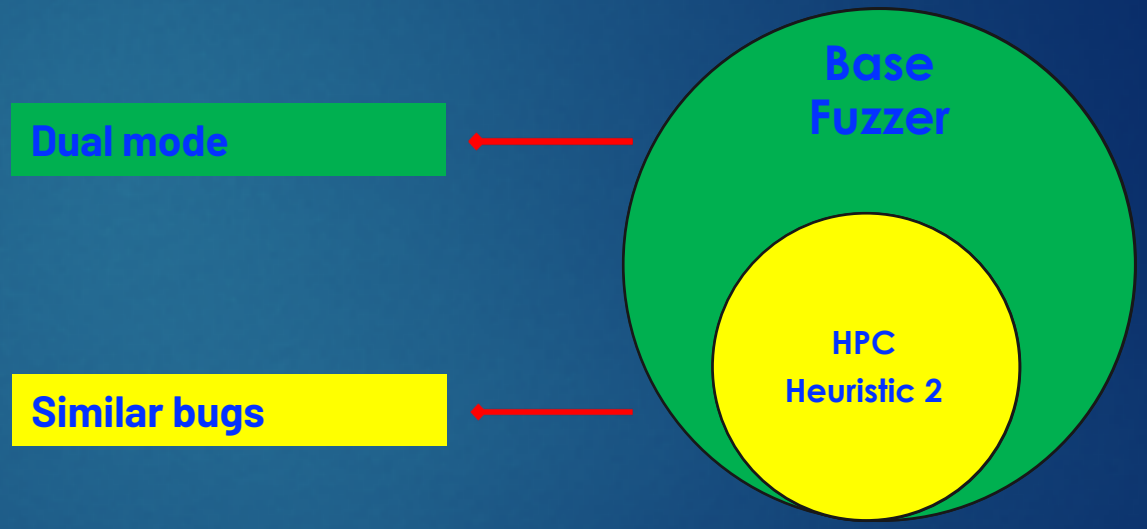
Multiple HPC events to build **multiple coverage metrics** to guide input selection

Vulnerability Exploration

Fuzzing campaign

- Multiple HPC heuristics
  - Best 4 events
  - Next 4 events
- Single mode
  - Similar bugs as past
  - Different bugs than past
- Dual mode
  - Switch coverage metric

Online crash deduplication provides a feedback to switch fuzzing strategies



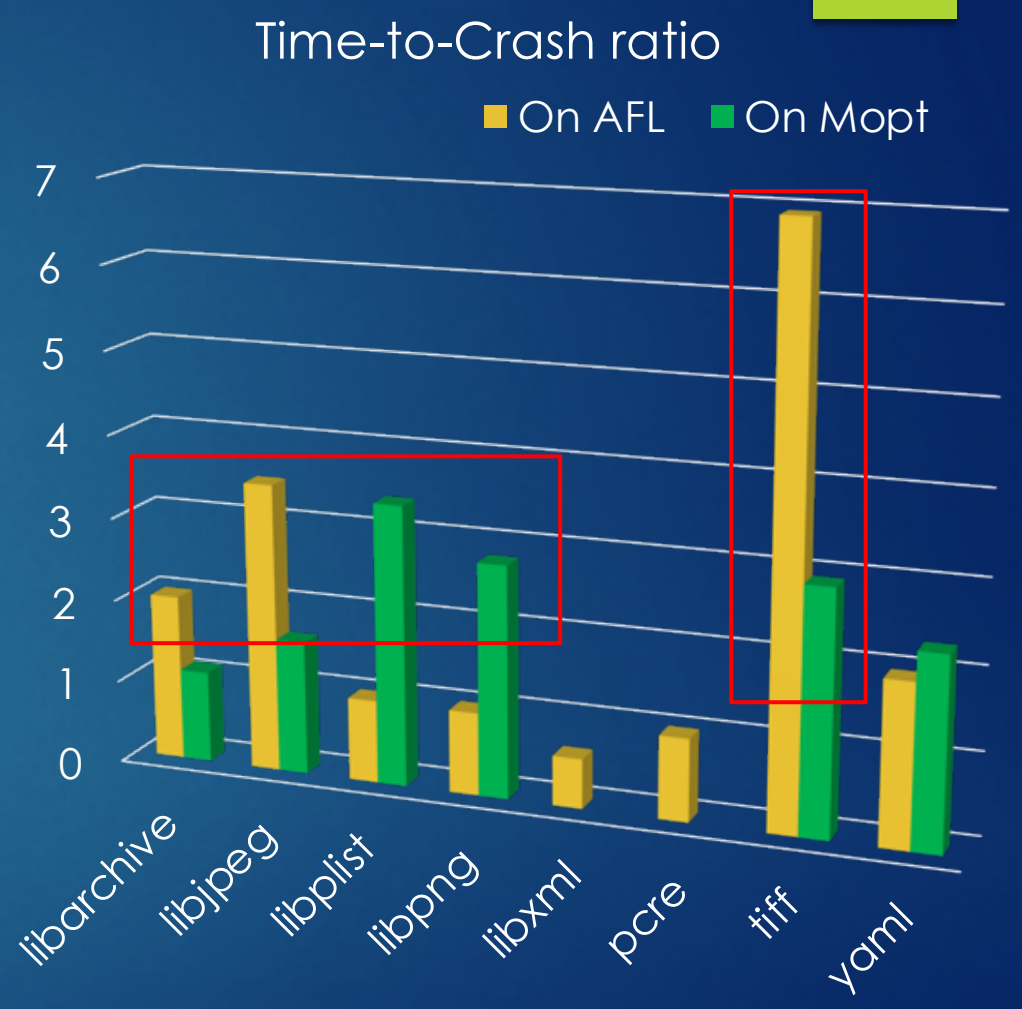
Evaluation

- Extended on three state of the art base fuzzers
  - **AFL, MOpt, Fairfuzz**
- Eight heavily fuzzed libraries
- Evaluated on different version of programs than model building
- Equal amounts of resources
  - One fuzzer instance
  - 24 hours runtime
- For ground truth, we manually analyzed and verified all the bugs

Programs	No. of Bugs
libjpeg	9
libarchive	5
tiff	5
libxml	4
pcre	2
libplist	2
libpng	1
yaml	1

On Expediency

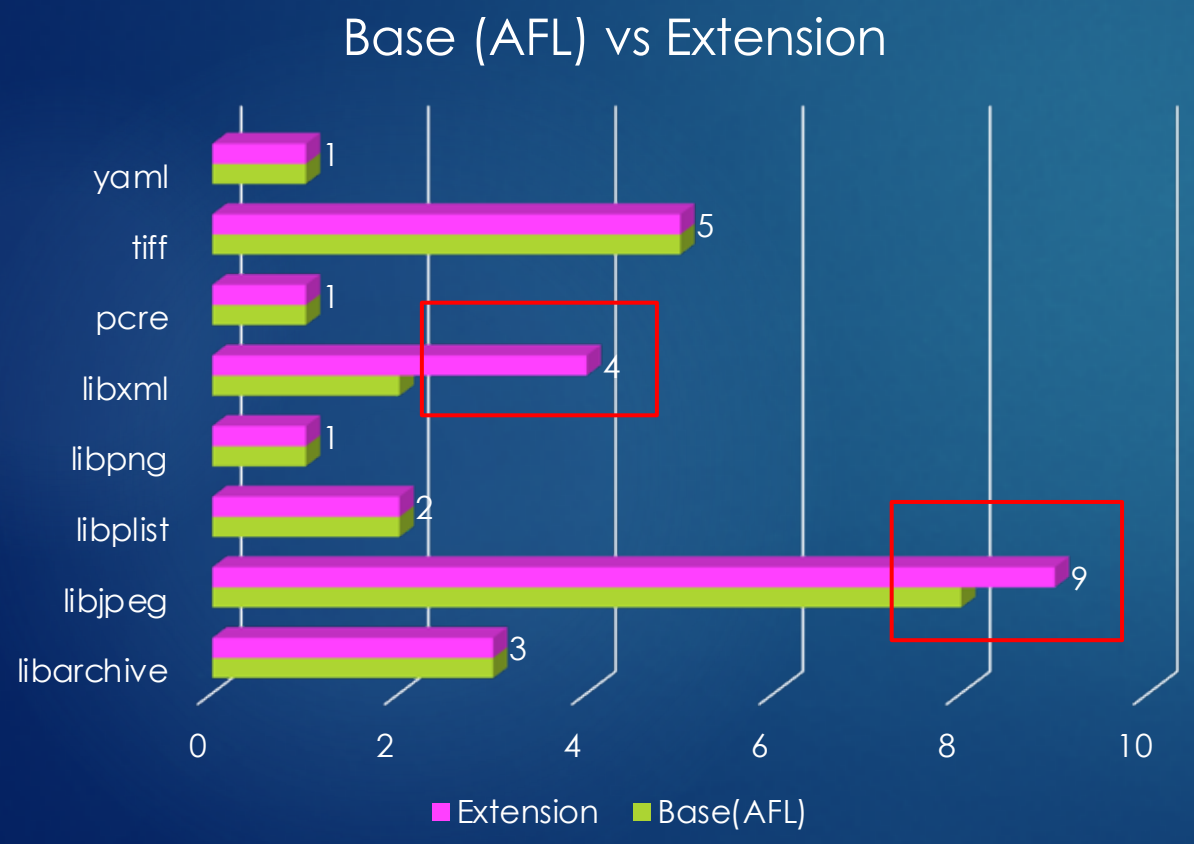
- The uniqueness of a crash is measured by stack trace approach
- We measure speedup relative to a predefined granularity
- **On average omnifuzz is 3.3x faster than the base fuzzers**
- Interestingly, our strategies explore only 62% of the base fuzzer paths on average
- Comprehensive results in the paper

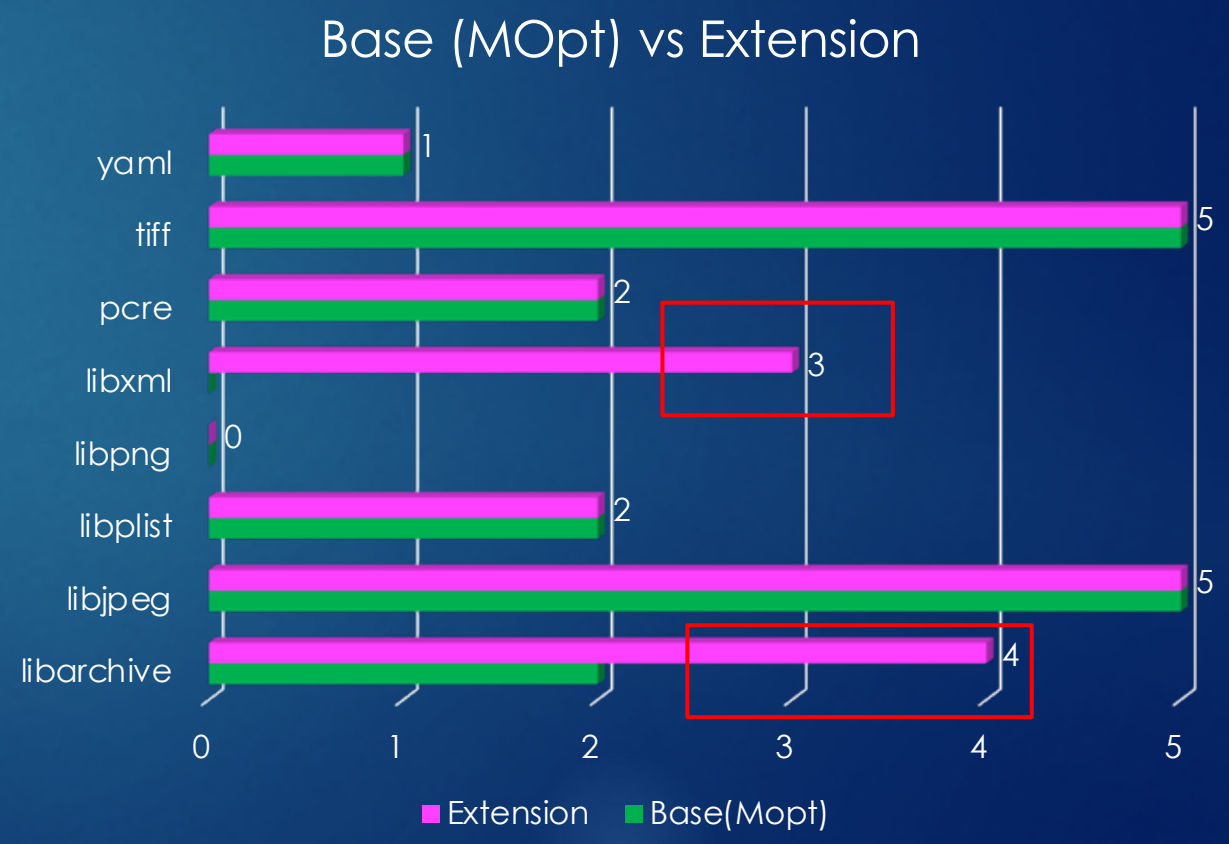


A larger ratio means that our approach finds the bug in less time

Bug Discovery

- Our extensions finds 3 additional bugs on AFL, and 5 on MOpt





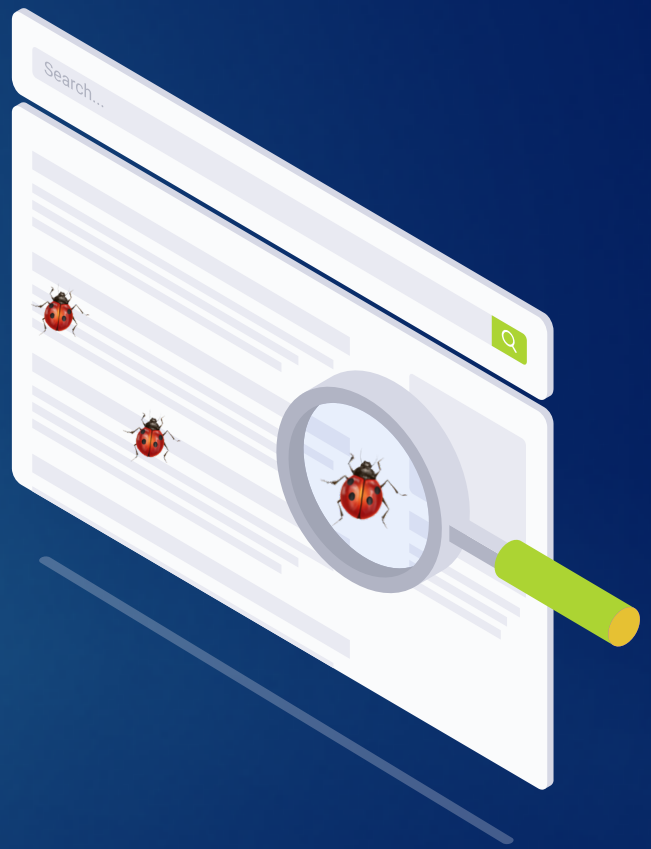
New Vulnerabilities

3 CVEs were assigned

- Libjpeg - CVE-2020-13790
- Libarchive - CVE-2020-21674
- Libarchive - CVE-2019-19221

More bugs reported

- In PCRE, PHP
- Heap-based buffer overflow
- Out-of-bounds read
- Stack overflow, null pointer dereference



Takeaways & Future Direction

HPC provides rich information to explore a multitude of coverage metrics

- Expedite bug finding, and to find unique bugs

To learn more, please read our paper

THANKS!

Any questions?

sanjeev.das@ibm.com