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
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
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]

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 \rightarrow BB2 \rightarrow BB4] = [BB1 \rightarrow BB2 \rightarrow BB5]

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?

Our Approach

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

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

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**
- Data collection
- Crashing & non-crashing inputs
- Profile HPC events

**Model Building**
- Feature selection
- HPC events selection
- ML based heuristic

**Vulnerability Exploration**
- Configurable input selection strategy
- Fuzzing campaign
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

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

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

Pre-processing  Model Building  Vulnerability Exploration
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

<table>
<thead>
<tr>
<th>Programs</th>
<th>No. of Bugs</th>
</tr>
</thead>
<tbody>
<tr>
<td>libjpeg</td>
<td>9</td>
</tr>
<tr>
<td>libarchive</td>
<td>5</td>
</tr>
<tr>
<td>tiff</td>
<td>5</td>
</tr>
<tr>
<td>libxml</td>
<td>4</td>
</tr>
<tr>
<td>pcre</td>
<td>2</td>
</tr>
<tr>
<td>libplist</td>
<td>2</td>
</tr>
<tr>
<td>libpng</td>
<td>1</td>
</tr>
<tr>
<td>yaml</td>
<td>1</td>
</tr>
</tbody>
</table>
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

![Time-to-Crash ratio graph](image)

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

Libraries: libarchive, libjpeg, libplist, libpng, libxml, pcre, tiff, yaml

**Relative Speedup**: On AFL vs. On Mopt
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

• Expedite bug finding, and to find unique bugs

HPC provides rich information to explore a multitude of coverage metrics

To learn more, please read our paper
THANKS!

Any questions?

sanjeev.das@ibm.com