Fine-Grained Coverage-Based Fuzzing

Bernard Nongpoh, Marwan Nour, Michaël Marcozzi, Sébastien Bardin

Université Paris-Saclay, CEA, List
Palaiseau, Paris Metropolitan Area, France
first.last@cea.fr

International Fuzzing Workshop (FUZZING) 2022
San Diego, CA, USA

April 24, 2022
Feedback-based Fuzzing Process

Guided Random Input Generator → Test Cases → Feedback

Monitor
Program execution → Crashes
Crash Triage

Feedback → Branch Coverage

FUZZING 2022 • Fine-Grained Coverage-Based Fuzzing • April 24, 2022
Problem

Branch Coverage is a shallow metric
Goals

- Support fine-grained coverage metrics within state-of-the-art fuzzers
- Leverage decades of software engineering research on defining such metrics
- Make them available in existing fuzzers out-of-the-box (no fuzzer modification)
Contributions

- Developed a Clang pass to **annotate** C code with fine-grained coverage criteria

- Preliminary evaluation using two state-of-art fuzzers, namely **AFL++** and **QSYM** on the LAVA-M benchmarks
Motivating Example

A buggy program checking if an appliance is running outside its allowed temperature range

```c
void check(int current_temp, char *data[]) {
    if (current_temp >= 50) {
        // Deal with appliance running outside the allowed temperature limit
        ...
        // The bug triggers a detectable crash only when current_temp==50
        // and when rare specific values are present in data
    }
}
```

```c
void check(int current_temp, char *data[]) {
    if (current_temp >= 50 != current_temp > 50) {
    }
    ...
    if (current_temp >= 50) {
        // Deal with appliance running outside the allowed temperature limit
        ...
        // The bug triggers a detectable crash only when current_temp==50
        // and when rare specific values are present in data
    }
}
```
Our Approach

Making test objectives explicit in the code of the fuzzed program with labels

```
statement_1;
if (x==y && a<b)
    {...};
statement_3;
```

Condition Coverage (CC)

```
statement_1;
// l-1: x==y
// l-2: x! =y
// l-3: a<b
// l-4: a>=b
if (x==y && a<b)
    {...};
statement_3;
```

Multiple Condition Coverage (MCC)

```
statement_1;
// l-1: x==y && a<b
// l-2: x!=y && a<b
// l-3: x==y && a>=b
// l-4: x!=y && a>=b
if (x==y && a<b)
    {...};
statement_3;
```

Arithmetic Operator Replacement (AOR) Mutant

```
statement_1;
x=a+b;
statement_3;
```

Weak Mutation Coverage (WM)

```
statement_1;
// l-1: (a+b)!=(a*b)
x=a+b;
statement_3;
```
### Code Coverage Criteria

```java
if (A && B) {
    // Statement-1
} else {
    // Statement-2
}
```

<table>
<thead>
<tr>
<th>Combination</th>
<th>DC</th>
<th>CC</th>
<th>DCC</th>
<th>MCC</th>
</tr>
</thead>
<tbody>
<tr>
<td>A &amp;&amp; B</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>A &amp;&amp; B'</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
</tr>
<tr>
<td>A &amp;&amp; B'</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
</tr>
<tr>
<td>A' &amp;&amp; B'</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

- **Decision Coverage (DC)**
- **Condition Coverage (CC)**
- **Decision Condition Coverage (DCC)**
- **Multiple Condition Coverage (MCC)**
Mutation Coverage

- **Strong Mutation Coverage (SM)**: A Mutant M is covered/killed by a test t if the outputs of $p(t)$ and $M(t)$ differ from each other.

- **Weak Mutation Coverage (WM)**: A Mutant M is covered/killed by a test t if the internal states of $P(t)$ and $M(t)$ differ from each other right after the mutated location.
Fine-grained Coverage-based Fuzzing

Workflow

Source Code

Side-Effects Extraction

Normalized Source Code

Program Annotation

Annotated Source Code

Coverage-based fuzzers

CC

MCC

DC

Weak Mutation Operators

AOR

ABS

WORKFLOW
Side-Effects

Output: \texttt{ab} \neq \texttt{aabbab}

\begin{verbatim}
statement_1;
//l-1: print("a")
//l-2: !print("a")
//l-3: print("b")
//l-4: !print("b")
if(print("a") && print("b"))
  {...};
statement_3;
\end{verbatim}

Program with labels for CC

\begin{verbatim}
statement_1;
if (print("a")) {}
if (!print("a")) {}
if (print("b")) {}
if (!print("b")) {}
if (print("a") && print("b"))
  {...};
statement_3;
\end{verbatim}

Annotated program for CC
Examples of Side-Effects Extraction (1)

Side-effect in an atomic condition at a decision point

```
int foo();
...
if (foo()) { }

⇒
int foo();
....
int temp = foo();
if (temp) { }
```
Examples of Side-Effects Extraction (2)

Side-effect in the second atomic condition of a lazy boolean operator (AND case)
Examples of Side-Effects Extraction (3)

Side-effect in the second atomic condition of a lazy boolean operator (OR case)
Program Annotation

Program with labels for CC

Annotated program for CC

- Off-the-shelf fuzzers will be able to handle **Condition/Branch Coverage** out-of-the-box. *Thanks to This!*
Research Questions

- **RQ1**: Is our code annotation tool effective and useful?  
  - Is it easy to use and does out-of-the-box integration with existing fuzzers work well?  
  - Can it scale to real-world applications?

- **RQ2**: Does our fine-grained approach allow to improve over the baseline state-of-art fuzzers?
Experimental Evaluation

- **Objective**: Fuzzer performance with and without label instrumentation (MCC and WM)
- Coverage-based fuzzers: **AFL++** and **QSYM**
- **Infrastructure**
  - Intel Skylake CPU, with 192GB memory RAM and 72 logical cores running at 2.6GHz.
- Time budget of **24 hours** (repeated 5 times)
- Benchmark: **LAVA-M Benchmark** Suite
## Preliminary Experimental Evaluation

<table>
<thead>
<tr>
<th>Application</th>
<th>LOC</th>
<th>MCC</th>
<th>Weak Mutation</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>ABS</td>
<td>AOR</td>
</tr>
<tr>
<td>uniq</td>
<td>494</td>
<td>204</td>
<td>61</td>
<td>7</td>
</tr>
<tr>
<td>base64</td>
<td>255</td>
<td>26</td>
<td>56</td>
<td>7</td>
</tr>
<tr>
<td>md5sum</td>
<td>663</td>
<td>125</td>
<td>113</td>
<td>20</td>
</tr>
<tr>
<td>who</td>
<td>622</td>
<td>170</td>
<td>180</td>
<td>15</td>
</tr>
</tbody>
</table>

Number of **labels** on LAVA-M benchmark suite
Preliminary Experimental Evaluation

Bugs found on the LAVA-M benchmark by fuzzers
Average number of bugs found by QSYM and QSYM+\textit{lannot} vs time
Label Coverage

Cumulative label coverage (in %) vs time
Conclusion

- Borrow well-established research over fine-grained code coverage criteria and provide off the shelf support in popular fuzzers
- Making test objectives (defined by fine-grained metrics) explicit as new branches in the target program
- Preliminary evaluation on the four LAVA-M benchmark suite
- Tested on two fuzzers: AFL++ and QSYM
- Preliminary findings:
  - On average, 100 more bugs being discovered in total
  - Bugs being uncovered faster during the fuzzing process
  - Label coverage improvement in two applications
Future Direction

- Pruning out *infeasible* labels
- Test on *Magma* (ground-truth benchmark) and *real-world* applications
- Investigate the effect of each coverage criteria on the *fuzzing performance* separately
- Evaluation on *standard metrics* as suggested by the fuzzing community (edge coverage)
- Investigate the *overhead* introduce by labelling in fuzzer throughput
"Making current fuzzer support fine-grained coverage metrics out-of-the-box"

Preprint available at [https://binsec.github.io/](https://binsec.github.io/)

Follow us on Twitter

@BinsecTool, @BernardNongpoh, @__M4rwan, @michaelmarcozzi

The team is looking for Ph.D. Students and PostDoc
Visit [https://binsec.github.io/](https://binsec.github.io/) for more information