Fine-Grained Coverage-Based Fuzzing

Wei-Cheng Wu\textsuperscript{1,2}  Bernard Nongpoh\textsuperscript{1}  Marwan Nour\textsuperscript{1}
Michaël Marcozzi\textsuperscript{1}  Sébastien Bardin\textsuperscript{1}  Christophe Hauser\textsuperscript{2}

published in
ACM Transactions On Software Engineering and Methodology (TOSEM)
and also presented at the
38th IEEE/ACM International Conference on Automated Software Engineering (ASE 2023)

LVP+MTV2 Seminar // Friday, November 17, 2023
This work has been mainly carried out by...

Wei-Cheng Wu  
Ph.D. student  
(also @ USC in Los Angeles)

Dr. Benard Nongpoh  
Postdoc  
(now @ Qualcomm)

Marwan Nour  
M.Sc. Intern  
(from Ecole Polytechnique, now @ EC)
About me // Dr. Michaël Marcozzi

• Permanent researcher @ CEA LIST, Université Paris-Saclay
• My research group focus on software analysis for security
• Invited lecturer @ ENSTA, Institut Polytechnique de Paris
1. **Context**: coverage-based fuzzing
2. **Problem**: branch coverage is shallow
3. **Goal**: enable and evaluate fuzzer guidance with fine-grained metrics
4. **Proposal**: finer-grained objectives as new branches in fuzzed code
5. **Experimental evaluation of impact**
6. **Conclusions**
Fuzzing [1/2]

Fuzzing a program (for security) is...

1. Feed program with massive number of automatically generated inputs
2. Trigger so observable failures (e.g. crashes)
3. Analyse failures to reveal program vulnerabilities to fix or exploit
Fuzzing [1/2]

Fuzzing a program (for security) is...

1. Feed program with massive number of automatically generated inputs
2. Trigger so observable failures (e.g. crashes)
3. Analyse failures to reveal program vulnerabilities to fix or exploit
Fuzzing [1/2]

Fuzzing a program (for security) is...

1. Feed program with massive number of automatically generated inputs
2. Trigger so observable failures (e.g. crashes)
3. Analyse failures to reveal program vulnerabilities to fix or exploit
Fuzzing [2/2]

Fuzzing is **popular** *(why? easy to understand/use, scalable, effective?)*…

• Many recent research papers on improving fuzzers

• “At Google, fuzzing has uncovered tens of thousands of bugs”  [Metzman et al., 2021]

• Fuzzers have found many CVE vulnerabilities in real programs

Number of fuzzing papers/year  [Liang et al., 2018]  Some 2019 CVEs found by AFL++ fuzzer  [AFL++ website]
Coverage-based fuzzing [1/3]

Many fuzzers use branch coverage to guide input generation...

- New inputs are generated by mutating the former inputs that improved branch coverage

- The rationale of this heuristic is...
  - The inputs that improved branch coverage uncovered new interesting program behaviours
  - Mutating these inputs should explore these new behaviours even more

```c
if (input > 5) { // Decision point
  // THEN branch
} else {
  // ELSE branch
}
```
Coverage-based fuzzing [2/3]

More precisely, coverage-based fuzzers implement the following loop...

(User-provided) Initial inputs → Seed inputs database → Mutations → Seed input → Test input → Monitor

Failure observed? → Yes → Analyse possible vulnerability!

Branch coverage improved? → Yes → Test input

No → No → Test input
More precisely, coverage-based **fuzzers** implement the following **loop**...
Coverage-based fuzzing [2/3]

More precisely, coverage-based **fuzzers** implement the following **loop**...
Coverage-based fuzzing [2/3]

More precisely, coverage-based **fuzzers** implement the following **loop**...
Coverage-based fuzzing [2/3]

More precisely, coverage-based **fuzzers** implement the following **loop**...
Coverage-based fuzzing [2/3]

More precisely, coverage-based **fuzzers** implement the following **loop**...
More precisely, coverage-based **fuzzers** implement the following **loop**...

- Seed inputs database
- Mutations
- Seed input
- Test input
- Monitor
  - Failure observed?
  - Branch coverage improved?
    - Yes
      - Yes
      - Analyse possible vulnerability!
Coverage-based fuzzing [2/3]

More precisely, coverage-based **fuzzers** implement the following **loop**...

![Diagram of fuzzing process]

1. **Seed inputs** database
2. **Seed input**
3. **Test input**
4. **Mutations**
5. **Monitor**
6. **Failure observed?**
   - Yes: Analyse possible vulnerability!
   - No: Go back to seed input
7. **Branch coverage improved?**
   - Yes: Go back to seed input
   - No: Keep fuzzing

The loop terminates when the fuzzing budget is over!
Coverage-based fuzzing [3/3]

Yet, the fuzzing loop alone requires a high budget to find bugs in “difficult” branches...

• A branch in fuzzed code is “difficult” when only activated by tiny fraction of inputs

![Code snippet]

• Code analyses enable fuzzers to be faster at finding inputs entering difficult branches...
  • (Taint tracking) Track comparisons between inputs and constants in fuzzed code (e.g. AFL++ fuzzer)
  • (Symbolic execution) Derive and solve path constraints to enter barely covered branches (e.g. Qsym fuzzer)
Outline

1. **Context**: coverage-based fuzzing

2. **Problem**: branch coverage is shallow

3. **Goal**: enable and evaluate fuzzer guidance with fine-grained metrics

4. **Proposal**: finer-grained objectives as new branches in fuzzed code

5. **Experimental evaluation of impact**

6. **Conclusions**
Fine-grained coverage metrics [1/2]

- Branch coverage is a **shallow metric** of interesting program behaviours.
- Fuzzers may thus **ignore inputs** that were interesting to find and mutate.
- Software testing researchers have for long proposed **finer-grained metrics**.
- **Idea**: guide fuzzers using these **control-flow, data-flow or mutation metrics**.
Fine-grained coverage metrics [2/2]

**For example**, MCC metric considers subtler variations of program logic...

### Program

```java
if (engine_speed > 0 || wheels_speed > 0) {
    // Lock door
} else { ... }
```

### Branch Coverage
- **Coverage objective**: cover both branches
- **Satisfying input**:
  - Take THEN branch: `engine_speed = 5, wheels_speed = 0`
  - Take ELSE branch: `engine_speed = 0, wheels_speed = 0`

### Multiple Condition Coverage (MCC)
- **Coverage objective**: cover whole truth table
- **Satisfying input**:
  - `true || true`: `engine_speed = 5, wheels_speed = 5`
  - `true || false`: `engine_speed = 5, wheels_speed = 0`
  - `false || true`: `engine_speed = 0, wheels_speed = 5`
  - `false || false`: `engine_speed = 0, wheels_speed = 0`
State of the art

- Early research exists for a specific fine-grained metric in a specific fuzzer
- Yet, no clear and general idea of what practical impact is
- Huge effort needed to support all fine-grained metrics in all legacy fuzzers
Outline

1. **Context**: coverage-based fuzzing
2. **Problem**: branch coverage is shallow
3. **Goal**: enable and evaluate fuzzer guidance with fine-grained metrics
4. **Proposal**: finer-grained objectives as new branches in fuzzed code
5. **Experimental evaluation of impact
6. **Conclusions**
Challenges of guiding fuzzers with finer-grained metrics

1. **Harness the wild variety** of legacy fuzzers and fine-grained metrics...

   Provide a **runtime guidance mechanism** that works without modifying legacy fuzzers:
   - *Activate* coverage objectives from most fine-grained metrics *for seed selection*
   - *Trigger* search for inputs that satisfy *difficult fine-grained coverage objectives*

2. **Evaluate impact** of fine-grained metrics *over legacy fuzzers performance*
Outline

1. **Context**: coverage-based fuzzing
2. **Problem**: branch coverage is shallow
3. **Goal**: enable and evaluate fuzzer guidance with fine-grained metrics
4. **Proposal**: finer-grained objectives as new branches in fuzzed code
5. **Experimental evaluation of impact**
6. **Conclusions**
Principle [1/3]

We guide legacy (branch) fuzzers by **transforming the fuzzed program**...

- Objectives from most metrics can be **made explicit as assertions in the fuzzed code** [Bardin et al., 2021]
- Thus, we add a **no-op branch** (if guarded by the assertion predicate) **for each assertion**
Multiple Condition Coverage (MCC)
cover whole truth table

<table>
<thead>
<tr>
<th>Coverage objective</th>
<th>Satisfying input</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td></td>
</tr>
<tr>
<td>true</td>
<td></td>
</tr>
<tr>
<td>false</td>
<td></td>
</tr>
<tr>
<td>false</td>
<td></td>
</tr>
</tbody>
</table>

if (engine_speed > 0 || wheels_speed > 0) {
    // Lock door
} else { ... }

transformed program

if (engine_speed > 0 && wheels_speed > 0) {}
if (engine_speed > 0 && wheels_speed <= 0) {}
if (engine_speed <= 0 && wheels_speed > 0) {}
if (engine_speed <= 0 && wheels_speed <= 0) {}
if (engine_speed > 0 || wheels_speed > 0) {
    // Lock door
} else { ... }
Principle [3/3]

When fuzzing the transformed program with a legacy (branch) fuzzer...  

• ...inputs covering the fine-grained objectives will effortlessly be saved as seeds  

• ...code analyses for difficult branches will help with difficult fine-grained objectives
Practical contributions

We propose a careful no-op branch insertion tool for fine-grained metrics...

• ...which avoids corrupting the program semantics (side-effects, spurious crashes)
• ...which avoids branches being tampered by compiler or fuzzing harness
Possible extensions

No-op branches could be **used as a more general guidance mechanism**...

- They could also be guarded by *predicates written by human developers*...
- ...or by predicates **computed by static analysers** (like fault preconditions)
Outline

1. **Context**: coverage-based fuzzing
2. **Problem**: branch coverage is shallow
3. **Goal**: enable and evaluate fuzzer guidance with fine-grained metrics
4. **Proposal**: finer-grained objectives as new branches in fuzzed code
5. **Experimental evaluation of impact**
6. **Conclusions**
Main evaluation plan

We evaluate the impact of fine-grained metrics over fuzzing...

- ...by running legacy fuzzers over original programs and transformed versions
- ...and comparing throughput, seeds number, covered branches and found bugs
Main experimental setup

Original programs

LAVA-M and MAGMA standard benchmarks

16 C programs

700k LOC with planted bugs
Main experimental setup

Original programs

16 C programs
700k LOC with planted bugs

LAVA-M and MAGMA standard benchmarks

Transformed programs for WM metric

Transformed programs for MCC metric

Transformed programs for WM+MCC metrics

We use **Multiple Condition Coverage** (MCC) and **Weak Mutations** coverage (WM) two common fine-grained metrics, notoriously denser than branch coverage.
Main experimental setup

Original programs
- LAVA-M and MAGMA standard benchmarks
  - 16 C programs
  - 700k LOC with planted bugs

Transformed programs for WM metric
Transformed programs for MCC metric
Transformed programs for WM+MCC metrics

AFL++

5 x (24h fuzzing campaign) per program
*to improve statistical significance*
Main experimental setup

Original programs
- LAVA-M and MAGMA standard benchmarks
  - 16 C programs
  - 700k LOC with planted bugs

Transformed programs
- for WM metric
- for MCC metric
- for WM+MCC metrics

AFL++

5 x (24h fuzzing campaign) per program
to improve statistical significance

Averaged...
- fuzzer’s throughput
- # saved seeds
- # covered branches
- # planted bugs that were detected
2.5 years of CPU computation happen here
Consolidated results for AFL++
(detailed results for AFL++ and QSYM are available in the paper, observations are similar)

<table>
<thead>
<tr>
<th>Executable</th>
<th>AFL++ with MCC</th>
<th>AFL++ with WM</th>
<th>AFL++ with MCC + WM</th>
</tr>
</thead>
<tbody>
<tr>
<td>base64</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>uniq</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>md5sum</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>who</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lua</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>exif</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sndfile</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>libpng_read</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tiff_read_rgba</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tiffcp</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>read_memory</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>xmlf1nt</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sqlite3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>server</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>client</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x509</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Throughput: % Change Seeds: % Change Branches: % Change Bugs: % Change

16/11/2023
## Consolidated results for AFL++

(detailed results for AFL++ and QSYM are available in the paper, observations are similar)

Fuzzer quickly saturates on smaller and simpler programs...

<table>
<thead>
<tr>
<th>Executable</th>
<th>AFL++ with MCC</th>
<th>AFL++ with WM</th>
<th>AFL++ with MCC + WM</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Throughput</td>
<td>Seeds</td>
<td>Branches</td>
</tr>
<tr>
<td>base64</td>
<td>49%</td>
<td>28%</td>
<td>2</td>
</tr>
<tr>
<td>uniq</td>
<td>45%</td>
<td>23%</td>
<td>2</td>
</tr>
<tr>
<td>md5sum</td>
<td>49%</td>
<td>28%</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>4%</td>
<td>19%</td>
<td>1</td>
</tr>
<tr>
<td>lua</td>
<td>4%</td>
<td>19%</td>
<td>1</td>
</tr>
<tr>
<td>exif</td>
<td>-21%</td>
<td>-19%</td>
<td>1</td>
</tr>
<tr>
<td>sndfile</td>
<td>-21%</td>
<td>-19%</td>
<td>1</td>
</tr>
<tr>
<td>libpng_read</td>
<td>-21%</td>
<td>-19%</td>
<td>1</td>
</tr>
<tr>
<td>tiff_read_rgba</td>
<td>-19%</td>
<td>-19%</td>
<td>1</td>
</tr>
<tr>
<td>tiffcp</td>
<td>-19%</td>
<td>-19%</td>
<td>1</td>
</tr>
<tr>
<td>read_memory</td>
<td>-21%</td>
<td>-19%</td>
<td>1</td>
</tr>
<tr>
<td>xmlint</td>
<td>-21%</td>
<td>-19%</td>
<td>1</td>
</tr>
<tr>
<td>sqlite3</td>
<td>-21%</td>
<td>-19%</td>
<td>1</td>
</tr>
<tr>
<td>server</td>
<td>-17%</td>
<td>-17%</td>
<td>1</td>
</tr>
<tr>
<td>client</td>
<td>-17%</td>
<td>-17%</td>
<td>1</td>
</tr>
<tr>
<td>x509</td>
<td>-17%</td>
<td>-17%</td>
<td>1</td>
</tr>
</tbody>
</table>
Consolidated results for AFL++
(detailed results for AFL++ and QSYM are available in the paper, observations are similar)

Fine-grained metrics slow down the fuzzer
(instrumented program is slower and produces more coverage data)
Consolidated results for AFL++
(detailed results for AFL++ and QSYM are available in the paper, observations are similar)

<table>
<thead>
<tr>
<th>Executable</th>
<th>AFL++ with MCC</th>
<th>AFL++ with WM</th>
<th>AFL++ with MCC + WM</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Throughput</td>
<td>Seeds</td>
<td>Branches</td>
</tr>
<tr>
<td>base64</td>
<td>-29%</td>
<td>+2%</td>
<td>-2</td>
</tr>
<tr>
<td>uniq</td>
<td>-16%</td>
<td>-3%</td>
<td>-7</td>
</tr>
<tr>
<td>md5sum</td>
<td>-18%</td>
<td>-34%</td>
<td>-41</td>
</tr>
<tr>
<td>who</td>
<td>-36%</td>
<td>+19%</td>
<td>-333</td>
</tr>
<tr>
<td>lua</td>
<td>-8%</td>
<td>+6%</td>
<td>-55</td>
</tr>
<tr>
<td>exif</td>
<td>21%</td>
<td>-19%</td>
<td>-41</td>
</tr>
<tr>
<td>ndfifile</td>
<td>48%</td>
<td>1%</td>
<td>-259</td>
</tr>
<tr>
<td>libpng_read</td>
<td>-7%</td>
<td>+64%</td>
<td>-33</td>
</tr>
<tr>
<td>tiff_read_rgb</td>
<td>49%</td>
<td>-2%</td>
<td>-268</td>
</tr>
<tr>
<td>tiffcp</td>
<td>-49%</td>
<td>-9%</td>
<td>-653</td>
</tr>
<tr>
<td>read_memory</td>
<td>84%</td>
<td>+95%</td>
<td>-1442</td>
</tr>
<tr>
<td>xmlint</td>
<td>-72%</td>
<td>+41%</td>
<td>-850</td>
</tr>
<tr>
<td>sqlite3</td>
<td>-19%</td>
<td>-3%</td>
<td>-2489</td>
</tr>
<tr>
<td>server</td>
<td>-17%</td>
<td>-3%</td>
<td>-3</td>
</tr>
<tr>
<td>client</td>
<td>-17%</td>
<td>-2%</td>
<td>-36</td>
</tr>
<tr>
<td>x509</td>
<td>-18%</td>
<td>+1%</td>
<td>-4</td>
</tr>
</tbody>
</table>

Fine-grained metrics improve performance when fuzzer slowdown is low enough and bug density is high enough (favour local exploration vs. new branch discovery)
Consolidated results for AFL++
(detailed results for AFL++ and QSYM are available in the paper, observations are similar)

<table>
<thead>
<tr>
<th>Executable</th>
<th>AFL++ with MCC</th>
<th>AFL++ with WM</th>
<th>AFL++ with MCC + WM</th>
</tr>
</thead>
<tbody>
<tr>
<td>base64</td>
<td>-2%</td>
<td>+2%</td>
<td>-2%</td>
</tr>
<tr>
<td>uniq</td>
<td>-16%</td>
<td>-3%</td>
<td>+7</td>
</tr>
<tr>
<td>md5sum</td>
<td>-18%</td>
<td>-34%</td>
<td>-41</td>
</tr>
<tr>
<td>who</td>
<td>-6%</td>
<td>+19%</td>
<td>-133</td>
</tr>
<tr>
<td>lua</td>
<td>-8%</td>
<td>+4%</td>
<td>+65</td>
</tr>
<tr>
<td>exif</td>
<td>-21%</td>
<td>-19%</td>
<td>-41</td>
</tr>
<tr>
<td>sndfile</td>
<td>-48%</td>
<td>+1%</td>
<td>-239</td>
</tr>
<tr>
<td>libpng_read</td>
<td>7%</td>
<td>+64%</td>
<td>-33</td>
</tr>
<tr>
<td>tiff_readrgba</td>
<td>-49%</td>
<td>-9%</td>
<td>-653</td>
</tr>
<tr>
<td>tiffcp</td>
<td>-49%</td>
<td>-9%</td>
<td>-653</td>
</tr>
<tr>
<td>read_memory</td>
<td>84%</td>
<td>+85%</td>
<td>1448</td>
</tr>
<tr>
<td>xmllint</td>
<td>-72%</td>
<td>+48%</td>
<td>-830</td>
</tr>
<tr>
<td>sqlite3</td>
<td>-19%</td>
<td>-7%</td>
<td>-2489</td>
</tr>
<tr>
<td>server</td>
<td>-17%</td>
<td>-3%</td>
<td>+5</td>
</tr>
<tr>
<td>client</td>
<td>-17%</td>
<td>2%</td>
<td>+36</td>
</tr>
<tr>
<td>x509</td>
<td>-18%</td>
<td>+1%</td>
<td>-9</td>
</tr>
</tbody>
</table>

Fine-grained metrics improve performance when fuzzer slowdown is low enough and bug density is high enough (favour local exploration vs. new branch discovery)

*Hard to know if these conditions are met before fuzzing (most of the time, no)... :-(*
Outline

1. **Context**: coverage-based fuzzing
2. **Problem**: branch coverage is shallow
3. **Goal**: enable and evaluate fuzzer guidance with fine-grained metrics
4. **Proposal**: finer-grained objectives as new branches in fuzzed code
5. **Experimental evaluation of impact**
6. **Conclusions**
Conclusions [1/2]

Adding no-op branches to fuzzed code...

• Can provide runtime guidance to legacy (branch) fuzzers out of the box
• Can encode guidance from most fine-grained coverage metrics
• Requires careful transformation for not breaking semantics (beware of corner cases)

Future work involves...

• Study tighter integration with fuzzer harness and configuration
• Use to encode human directives or bug preconditions from static analysers
Conclusions [2/2]

**Fine-grained metrics** should **not replace branch coverage** to guide fuzzers...

- Impact is **hard to predict** before fuzzing and usually **neutral or negative**
- Other studies (with tight fuzzer/metric integration) tend to **confirm** this trend
- Yet, they might be useful in **small doses**, to improve **local exploration** where needed

**Future work** involves...

- Investigate favourable circumstances that could make fine-grained metrics profitable
- Notably, use them only in **fragile or sensitive parts** of the code...
Key takeaways

> Carefully adding branches to fuzzed code provides guidance to fuzzers
> Fine-grained metrics slow down fuzzers but favour local exploration