Fine-Grained Coverage-Based Fuzzing

Wei-Cheng Wu, Bernard Nongpoh, Marwan Nour, Michaël Marcozzi, Sébastien Bardin, Christophe Hauser

to appear in
ACM Transactions On Software Engineering and Methodology
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)
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
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**
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

```
./buffer-overflow.exe 1111111111111111111111111111111111
Segmentation fault
```
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
}
```
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...

(User-provided) Initial inputs → Seed inputs database → Seed input → Test input → Monitor → Failure observed?

Mutations

Yes → Analyse possible vulnerability!

No → Seed input → Test input → Failure observed?
Coverage-based fuzzing [2/3]

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

**Fine-Grained Coverage-Based Fuzzing**

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

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

![Diagram showing the fuzzing process]

- **Seed inputs database**
- **Seed input**
- **Test input**
- **Mutations**
- **Monitor**
- **Failure observed?**
  - Yes: Analyse possible vulnerability!
- **Branch coverage improved?**
  - Yes

The loop terminates when the fuzzing budget is over!
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 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...

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

<table>
<thead>
<tr>
<th>Coverage objective</th>
<th>Satisfying input</th>
</tr>
</thead>
<tbody>
<tr>
<td>Take THEN branch</td>
<td>engine_speed = 5</td>
</tr>
<tr>
<td></td>
<td>wheels_speed = 0</td>
</tr>
<tr>
<td>Take ELSE branch</td>
<td>engine_speed = 0</td>
</tr>
<tr>
<td></td>
<td>wheels_speed = 0</td>
</tr>
</tbody>
</table>

**Branch Coverage**

*cover both branches*

**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></td>
<td>wheels_speed = 5</td>
</tr>
<tr>
<td>true</td>
<td></td>
</tr>
<tr>
<td></td>
<td>wheels_speed = 0</td>
</tr>
<tr>
<td>false</td>
<td></td>
</tr>
<tr>
<td></td>
<td>wheels_speed = 5</td>
</tr>
<tr>
<td>false</td>
<td></td>
</tr>
<tr>
<td></td>
<td>wheels_speed = 0</td>
</tr>
</tbody>
</table>
• 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**
Principle [2/3]

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 {
    ...}

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 {
    ...}

transformed program
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
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
Simple example of corruption avoidance

```c
if (print("a") || graph_ok) {
    // Proceed
} else { /* Error */
```

program
Simple example of corruption avoidance

```c
if (print("a") || graph_ok) {
    // Proceed
} else { /* Error */ }
```

**Program**

```c
if (print("a") && graph_ok) {}
if (print("a") && !graph_ok) {}
if (!print("a") && graph_ok) {}
if (!print("a") && !graph_ok) {}
if (print("a") || graph_ok) {
    // Proceed
} else { /* Error */ }
```

**Transformed program for MCC**
Simple example of corruption avoidance

```cpp
if (print("a") || graph_ok) {
    // Proceed
} else {
/* Error */
}
```

Program

```cpp
if (print("a") && graph_ok) {}
if (print("a") && !graph_ok) {}
if (!print("a") && graph_ok) {}
if (!print("a") && !graph_ok) {}
if (print("a") || graph_ok) {
    // Proceed
} else {
/* Error */
}
```

Transformed program for MCC

`Prints “a” 4x more! (semantic change)`
Simple example of corruption avoidance

```c
if (print("a") || graph_ok) {
    // Proceed
} else {
    /* Error */
}
```

**Pre-processed program**

```c
int temp = print("a");
if (temp || graph_ok) {
    // Proceed
} else {
    /* Error */
}
```

**Transformed program for MCC**

```c
if (print("a") &&& graph_ok) {}
if (print("a") && !graph_ok) {}
if (!print("a") && & graph_ok) {}
if (!print("a") && !graph_ok) {
}
if (print("a") || graph_ok) {
    // Proceed
} else {
    /* Error */
}
```

Prints “a” 4x more! (semantic change)
Simple example of corruption avoidance

```c
if (print("a") || graph_ok) {
    // Proceed
} else {
    /* Error */
}
```

Program

```c
if (print("a") && graph_ok) {} // Proceed
if (print("a") && !graph_ok) {} // Proceed
if (!print("a") && graph_ok) {} // Proceed
if (!print("a") && !graph_ok) {} // Proceed
if (print("a") || graph_ok) {
    // Proceed
} else {
    /* Error */
}
```

Transformed program for MCC

```c
int temp = print("a");
if (temp && graph_ok) {} // Proceed
if (temp && !graph_ok) {} // Proceed
if (!temp && graph_ok) {} // Proceed
if (!temp && !graph_ok) {} // Proceed
if (temp || graph_ok) {
    // Proceed
} else {
    /* Error */
}
```

Pre-processed program

```c
int temp = print("a");
if (temp | | graph_ok) {
    // Proceed
} else {
    /* Error */
}
```

Transformed pre-processed program for MCC

Prints "a" 4x more! (semantic change)
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)
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

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

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++

/qsym

5 x (24h fuzzing campaign) per program

to improve statistical significance
Main experimental setup

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

Original programs

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

AFL++

/qsym

Averaged...
- fuzzer's throughput
- # saved seeds
- # covered branches
- # planted bugs that were detected

5 x (24h fuzzing campaign) per program to improve statistical significance
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></th>
<th></th>
<th></th>
<th>AFL++ with WM</th>
<th></th>
<th></th>
<th></th>
<th>AFL++ with MCC + WM</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Throughput</td>
<td>Seeds</td>
<td>Branches</td>
<td>Bugs</td>
<td>Throughput</td>
<td>Seeds</td>
<td>Branches</td>
<td>Bugs</td>
<td>Throughput</td>
<td>Seeds</td>
<td>Branches</td>
<td>Bugs</td>
</tr>
<tr>
<td>base64</td>
<td>+29%</td>
<td>+2%</td>
<td>-2</td>
<td>-</td>
<td>-40%</td>
<td>+1%</td>
<td>-</td>
<td>-</td>
<td>-5%</td>
<td>+2%</td>
<td>+2</td>
<td>-</td>
</tr>
<tr>
<td>uniq</td>
<td>+16%</td>
<td>-5%</td>
<td>+7</td>
<td>-</td>
<td>-1%</td>
<td>+12%</td>
<td>+10</td>
<td>-</td>
<td>-21%</td>
<td>+13%</td>
<td>+6</td>
<td>-</td>
</tr>
<tr>
<td>md5sum</td>
<td>+18%</td>
<td>-34%</td>
<td>-41</td>
<td>-</td>
<td>+3%</td>
<td>-31%</td>
<td>-41</td>
<td>-</td>
<td>+25%</td>
<td>-24%</td>
<td>-12</td>
<td>-</td>
</tr>
<tr>
<td>who</td>
<td>-6%</td>
<td>+19%</td>
<td>+135</td>
<td>+153</td>
<td>-3%</td>
<td>+28%</td>
<td>+6</td>
<td>+98</td>
<td>-19%</td>
<td>+22%</td>
<td>-1</td>
<td>-</td>
</tr>
<tr>
<td>lua</td>
<td>-8%</td>
<td>+6%</td>
<td>-65</td>
<td>-</td>
<td>-33%</td>
<td>+7%</td>
<td>-159</td>
<td>-</td>
<td>-36%</td>
<td>+6%</td>
<td>-99</td>
<td>-</td>
</tr>
<tr>
<td>exif</td>
<td>-21%</td>
<td>-19%</td>
<td>-41</td>
<td>+1</td>
<td>-12%</td>
<td>-5%</td>
<td>-13</td>
<td>+1</td>
<td>-27%</td>
<td>-25%</td>
<td>-98</td>
<td>+1</td>
</tr>
<tr>
<td>snipng_read</td>
<td>-48%</td>
<td>+2%</td>
<td>-293</td>
<td>-</td>
<td>-72%</td>
<td>+39%</td>
<td>-578</td>
<td>-</td>
<td>-64%</td>
<td>+48%</td>
<td>-373</td>
<td>-</td>
</tr>
<tr>
<td>tiff_read_rgb</td>
<td>-7%</td>
<td>+64%</td>
<td>-33</td>
<td>-1</td>
<td>-3%</td>
<td>+45%</td>
<td>-12</td>
<td>-</td>
<td>-13%</td>
<td>+95%</td>
<td>-16</td>
<td>-</td>
</tr>
<tr>
<td>tiffcp</td>
<td>-49%</td>
<td>-2%</td>
<td>-268</td>
<td>-2</td>
<td>-48%</td>
<td>+11%</td>
<td>-354</td>
<td>-1</td>
<td>-45%</td>
<td>+15%</td>
<td>-158</td>
<td>-1</td>
</tr>
<tr>
<td>read_memory</td>
<td>-49%</td>
<td>-9%</td>
<td>-653</td>
<td>-2</td>
<td>-52%</td>
<td>+18%</td>
<td>-512</td>
<td>-2</td>
<td>-44%</td>
<td>+18%</td>
<td>-543</td>
<td>-2</td>
</tr>
<tr>
<td>xmllint</td>
<td>-84%</td>
<td>+35%</td>
<td>-1447</td>
<td>-</td>
<td>-63%</td>
<td>+8%</td>
<td>-356</td>
<td>-</td>
<td>-86%</td>
<td>+53%</td>
<td>-1333</td>
<td>-</td>
</tr>
<tr>
<td>sqlite3</td>
<td>-72%</td>
<td>+46%</td>
<td>-850</td>
<td>-1</td>
<td>-49%</td>
<td>+12%</td>
<td>+401</td>
<td>-1</td>
<td>-77%</td>
<td>+54%</td>
<td>-1059</td>
<td>-1</td>
</tr>
<tr>
<td>server</td>
<td>-17%</td>
<td>-3%</td>
<td>+3</td>
<td>-1</td>
<td>-18%</td>
<td>-5%</td>
<td>-26</td>
<td>-1</td>
<td>-33%</td>
<td>-5%</td>
<td>-47</td>
<td>-1</td>
</tr>
<tr>
<td>client</td>
<td>-17%</td>
<td>+2%</td>
<td>+16</td>
<td>-</td>
<td>-27%</td>
<td>-</td>
<td>-20</td>
<td>-</td>
<td>-42%</td>
<td>-1%</td>
<td>-27</td>
<td>-</td>
</tr>
<tr>
<td>x509</td>
<td>-18%</td>
<td>+1%</td>
<td>-9</td>
<td>-1</td>
<td>-21%</td>
<td>+1%</td>
<td>-11</td>
<td>-</td>
<td>-24%</td>
<td>+1%</td>
<td>-13</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)

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>+29%</td>
<td>+2%</td>
<td>-2</td>
</tr>
<tr>
<td>uniq</td>
<td>+16%</td>
<td>-5%</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>+6%</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>+2%</td>
<td>-239</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>+35%</td>
<td>-1447</td>
</tr>
<tr>
<td>xmlint</td>
<td>-72%</td>
<td>+46%</td>
<td>-850</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>+3</td>
</tr>
<tr>
<td>client</td>
<td>-17%</td>
<td>+2%</td>
<td>+16</td>
</tr>
<tr>
<td>x509</td>
<td>-18%</td>
<td>+1%</td>
<td>-9</td>
</tr>
</tbody>
</table>
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>-5%</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>+6%</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>+2%</td>
<td>-239</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>+35%</td>
<td>-1447</td>
</tr>
<tr>
<td>xmlint</td>
<td>-72%</td>
<td>+46%</td>
<td>-850</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>+3</td>
</tr>
<tr>
<td>client</td>
<td>-17%</td>
<td>+2%</td>
<td>+16</td>
</tr>
<tr>
<td>x509</td>
<td>-18%</td>
<td>+1%</td>
<td>-9</td>
</tr>
</tbody>
</table>

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>-5%</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>+6%</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>+2%</td>
<td>-239</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>-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>+35%</td>
<td>-1447</td>
</tr>
<tr>
<td>xmllint</td>
<td>-72%</td>
<td>+46%</td>
<td>-850</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>+3</td>
</tr>
<tr>
<td>client</td>
<td>-17%</td>
<td>+2%</td>
<td>+16</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)
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 Throughput</th>
<th>Seeds</th>
<th>Branches</th>
<th>Bugs</th>
<th>AFL++ with WM Throughput</th>
<th>Seeds</th>
<th>Branches</th>
<th>Bugs</th>
<th>AFL++ with MCC + WM Throughput</th>
<th>Seeds</th>
<th>Branches</th>
<th>Bugs</th>
</tr>
</thead>
<tbody>
<tr>
<td>base64</td>
<td>-29%</td>
<td>+2%</td>
<td>-2</td>
<td>-</td>
<td>+40%</td>
<td>+1%</td>
<td>-</td>
<td>-</td>
<td>-3%</td>
<td>+2%</td>
<td>+2</td>
<td>-</td>
</tr>
<tr>
<td>uniq</td>
<td>+16%</td>
<td>-5%</td>
<td>-7</td>
<td>-</td>
<td>-1%</td>
<td>+12%</td>
<td>+10</td>
<td>-</td>
<td>-21%</td>
<td>+13%</td>
<td>+6</td>
<td>-</td>
</tr>
<tr>
<td>md5sum</td>
<td>+18%</td>
<td>-34%</td>
<td>-41</td>
<td>-</td>
<td>+3%</td>
<td>-31%</td>
<td>-43</td>
<td>-</td>
<td>+25%</td>
<td>-24%</td>
<td>-12</td>
<td>-</td>
</tr>
<tr>
<td>who</td>
<td>-6%</td>
<td>+19%</td>
<td>+133</td>
<td>+165</td>
<td>-9%</td>
<td>+28%</td>
<td>+6</td>
<td>+98</td>
<td>-19%</td>
<td>+22%</td>
<td>-4</td>
<td>-56</td>
</tr>
<tr>
<td>lua</td>
<td>-8%</td>
<td>+6%</td>
<td>-55</td>
<td>-</td>
<td>-33%</td>
<td>+7%</td>
<td>-159</td>
<td>-</td>
<td>-36%</td>
<td>+6%</td>
<td>-99</td>
<td>-</td>
</tr>
<tr>
<td>exif</td>
<td>-21%</td>
<td>-19%</td>
<td>-41</td>
<td>+1</td>
<td>-12%</td>
<td>-5%</td>
<td>-13</td>
<td>+1</td>
<td>-27%</td>
<td>-25%</td>
<td>-95</td>
<td>+1</td>
</tr>
<tr>
<td>ndfile</td>
<td>-48%</td>
<td>+2%</td>
<td>-239</td>
<td>-</td>
<td>-72%</td>
<td>+39%</td>
<td>-578</td>
<td>-</td>
<td>-64%</td>
<td>+48%</td>
<td>-373</td>
<td>-</td>
</tr>
<tr>
<td>libpng_read</td>
<td>-7%</td>
<td>+64%</td>
<td>-33</td>
<td>-1</td>
<td>-3%</td>
<td>+45%</td>
<td>-12</td>
<td>-</td>
<td>-13%</td>
<td>+95%</td>
<td>-16</td>
<td>-</td>
</tr>
<tr>
<td>tiff_read_rgba</td>
<td>-49%</td>
<td>-2%</td>
<td>-268</td>
<td>-2</td>
<td>-48%</td>
<td>+11%</td>
<td>-354</td>
<td>-1</td>
<td>-45%</td>
<td>+15%</td>
<td>-158</td>
<td>-1</td>
</tr>
<tr>
<td>tiffcp</td>
<td>-49%</td>
<td>-9%</td>
<td>-653</td>
<td>-2</td>
<td>-52%</td>
<td>+18%</td>
<td>-512</td>
<td>-2</td>
<td>-44%</td>
<td>+18%</td>
<td>-543</td>
<td>-2</td>
</tr>
<tr>
<td>read_memory</td>
<td>-84%</td>
<td>+35%</td>
<td>-1447</td>
<td>-</td>
<td>-63%</td>
<td>+8%</td>
<td>-356</td>
<td>-</td>
<td>-86%</td>
<td>+53%</td>
<td>-1333</td>
<td>-</td>
</tr>
<tr>
<td>xmllint</td>
<td>-72%</td>
<td>+46%</td>
<td>-850</td>
<td>-1</td>
<td>-49%</td>
<td>+12%</td>
<td>+401</td>
<td>-1</td>
<td>-77%</td>
<td>+53%</td>
<td>-1333</td>
<td>-</td>
</tr>
<tr>
<td>sqlite3</td>
<td>-19%</td>
<td>-7%</td>
<td>-2489</td>
<td>-</td>
<td>-25%</td>
<td>-10%</td>
<td>-5297</td>
<td>-</td>
<td>-45%</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>server</td>
<td>-17%</td>
<td>-3%</td>
<td>+3</td>
<td>-1</td>
<td>-18%</td>
<td>-5%</td>
<td>-26</td>
<td>-1</td>
<td>-33%</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>client</td>
<td>-17%</td>
<td>+2%</td>
<td>+16</td>
<td>-</td>
<td>-27%</td>
<td>-</td>
<td>-20</td>
<td>-</td>
<td>-42%</td>
<td>-</td>
<td>-4%</td>
<td>-</td>
</tr>
<tr>
<td>x509</td>
<td>-18%</td>
<td>+1%</td>
<td>-9</td>
<td>-1</td>
<td>-21%</td>
<td>+1%</td>
<td>-11</td>
<td>-</td>
<td>-24%</td>
<td>-</td>
<td>-4%</td>
<td>-</td>
</tr>
</tbody>
</table>

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

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