Compiler Fuzzing: How Much Does It Matter?

~ research published at the SPLASH’19 OOPSLA conference ~

*Michaël Marcozzi\textsuperscript{1} *Qiyi Tang\textsuperscript{2} Alastair F. Donaldson\textsuperscript{3,1} Cristian Cadar\textsuperscript{1}

\textsuperscript{1}The presented experimental study has been carried out equally by M. Marcozzi and Q. Tang.
1. **Context**: compiler fuzzing

2. **Problem**: importance of fuzzer-found miscompilations is unclear

3. **Goal**: a study of the practical impact of miscompilation bugs

4. **Methodology** for bug impact measurement

5. Experiments and results

6. Conclusions

7. Future work
Compiler Bugs

- Software developers intensively rely on compilers, often with blind confidence.

- Compilers are software: they have bugs too (~150 fixed bugs/month in LLVM compiler).

- In worst case, unnoticed miscompilation (silent generation of wrong code).

History of LLVM Bug Tracking System (2003-2015) [Sun et al., ISSTA'16]
Compiler Validation (1/2)

- Classical **software validation approaches** have been **applied to compilers**
  - **Formal verification**: CompCert verified compiler, Alive optimisation prover, etc.
  - **Testing**: commercial C test suites, LLVM test suite, etc.
Compiler Validation (2/2)

- Recent surge of interest in **compiler fuzzing**:
  - Automatic and massive random generation of test programs
  - Each program \(P\) is fed to the compiler, automatic miscompilation detection via…
    - **differential testing** (*compile \(P\) with \(N\) compilers, run the \(N\) binaries, detect different outputs*)
    - **metamorphic testing** (*compile and run \(P\) and \(P'\), check output of \(P'\) vs \(P\) is as expected*)
  - e.g. 200+ miscompilations found in LLVM by Csmith\(^1\), EMI\(^2\), Orange\(^3\) and Yarpgen\(^4\)

---

\(^1\) [Yang et al., PLDI’11] [Regehr et al., PLDI’12] [Chen et al., PLDI’13]

\(^2\) Equivalence Modulo Inputs [Le et al., PLDI’14, OOPSLA’15] [Sun et al., OOPSLA’16]

\(^3\) [Nagai et al., T-SLDM] [Nakamura et al., APCCAS’16]

\(^4\) https://github.com/intel/yarpgen
Outline

1. Context: compiler fuzzing

2. Problem: importance of fuzzer-found miscompilations is unclear

3. Goal: a study of the practical impact of miscompilation bugs

4. Methodology for bug impact measurement

5. Experiments and results

6. Conclusions

7. Future work
Importance of Fuzzer-Found Miscompilations (1/2)

• Audience of our talks on compiler fuzzers often question the importance of found bugs.

• In our experience, this is a contentious debate and people can be poles apart:

  In my opinion, compiler bugs are extremely dangerous, period. Thus, regardless of the real-world impact of compiler bugs, I think that techniques that can uncover (and help fix) compiler bugs are extremely valuable.

  One anonymous reviewer of this paper at a top P/L conference

I would suggest that compiler developers stop responding to researchers working toward publishing papers on [fuzzers]. Responses from compiler maintainers is being becoming a metric for measuring the performance of [fuzzers], so responding just encourages the trolls.

'The Shape of Code’ weblog author
(former UK representative at ISO International C Standard)
Importance of Fuzzer-Found Miscompilations (2/2)

• In this work, we consider a **mature compiler** in a **non-critical environment**:
  
  • The compiler has been intensively tested by its developers and users
  
  • Trade-offs between software reliability and cost are acceptable and common
  
• In this context, **doubting the impact of fuzzer-found bugs is reasonable**:

  🤔 It is unclear if mature compilers **leave much space to find severe bugs**
  
  🤔 Fuzzers find bugs with **randomly generated code**, whose patterns may not occur in real code
Outline

1. Context: compiler fuzzing

2. Problem: importance of fuzzer-found miscompilations is unclear

3. Goal: a study of the practical impact of miscompilation bugs

4. Methodology for bug impact measurement

5. Experiments and results

6. Conclusions

7. Future work
Goal and Challenges

• In this work, our **objectives** are to:
  
  ❌ Show specifically that compiler fuzzing matters or does not matter
  
  ✅ Study the **impact** of miscompilation bugs in a **mature** compiler over **real apps**
  
  ✅ Compare impact of bugs from **fuzzers** with others (e.g. found by compiling real code)

• Operationally, we aim at **overcoming** the following **challenges**:

  • Take steps towards a **methodology** to **measure the impact** of a miscompilation bug
  
  • Apply it over a **significant** but **tractable** set of bugs and real applications
Outline

1. Context: compiler fuzzing

2. Problem: importance of fuzzer-found miscompilations is unclear

3. Goal: a study of the practical impact of miscompilation bugs

4. **Methodology** for bug impact measurement

5. Experiments and results

6. Conclusions

7. Future work
Bug Impact Measurement Methodology

• Assumption: Restrict to **publicly fixed bugs in open-source compilers**, to extract

---

**Fixing Patch**

*written by developers*
Bug Impact Measurement Methodology

- **Assumption**: Restrict to publicly fixed bugs in open-source compilers, to extract

![Buggy Compiler Source](image)

Fixing Patch
*written by developers*
Bug Impact Measurement Methodology

- **Assumption**: Restrict to **publicly fixed bugs in open-source compilers**, to extract...
Bug Impact Measurement Methodology

- **Assumption**: Restrict to **publicly fixed bugs in open-source compilers**, to extract

  ![Buggy Compiler Source](image1.png) → **Fixing Patch written by developers** → ![Fixed Compiler Source](image2.png)

- **Assumption**: impact of miscompilation bug = **ability to change semantics of real apps**
Bug Impact Measurement Methodology

• **Assumption**: Restrict to **publicly fixed bugs in open-source compilers**, to extract

![Diagram showing the transition from a buggy compiler source to a fixed compiler source with a fixing patch written by developers.]

• **Assumption**: impact of miscompilation bug = **ability to change semantics of real apps**

• We **estimate** the **impact** of the compiler **bug over a real app** in **three stages**: 
Bug Impact Measurement Methodology

- **Assumption**: Restrict to *publicly fixed bugs in open-source compilers*, to extract

- **Assumption**: impact of miscompilation bug = *ability to change semantics of real apps*

- **We estimate** the *impact* of the compiler *bug over a real app* in *three stages*:
  1. Is the buggy compiler code reached and triggered *during compilation*?
Bug Impact Measurement Methodology

- **Assumption**: Restrict to **publicly fixed bugs in open-source compilers**, to extract

- **Assumption**: impact of miscompilation bug = **ability to change semantics of real apps**

- We **estimate** the **impact** of the compiler **bug over a real app** in **three stages**:
  1. Is the buggy compiler code reached and triggered **during compilation**?
  2. How much does a triggered bug change the **binary code**?
Bug Impact Measurement Methodology

- **Assumption**: Restrict to **publicly fixed bugs in open-source compilers**, to extract

![Buggy Compiler Source](image1) → ![Fixing Patch](image2) → ![Fixed Compiler Source](image3)

- **Assumption**: impact of miscompilation bug = **ability to change semantics of real apps**

- We **estimate** the **impact** of the compiler **bug over a real app** in **three stages**:
  1. Is the buggy compiler code reached and triggered **during compilation**?
  2. How much does a triggered bug change the **binary code**?
  3. Can the binary changes lead to differences in **binary runtime behaviour**?
Stage 1: Compile-Time Analysis

If (Not.isPowerOf2())
/* Code transformation */

If (Not.isPowerOf2())
&& C->getValue().isPowerOf2()
&& Not != C->getValue())
/* Code transformation */

Fix for LLVM bug #26323

Buggy Compiler Source

Fixed Compiler Source
Stage 1: Compile-Time Analysis

Buggy Compiler Source

if (Not.isPowerOf2())
/* Code transformation */

warn("Fixing patch reached!");
if (Not.isPowerOf2()) {
    if (!C->getValue().isPowerOf2()
        &
        Not != C->getValue())
        warn("Bug triggered!");
else /* Code transformation */
}

Fixed Compiler Source

if (Not.isPowerOf2() && C->getValue().isPowerOf2() && Not != C->getValue())
/* Code transformation */

fix for LLVM bug #26323
Stage 1: Compile-Time Analysis

Buggy Compiler Source

```
if (Not.isPowerOf2())
/* Code transformation */
```

Fixed Compiler Source

```
if (Not.isPowerOf2())
&& C->getValue().isPowerOf2()
&& Not != C->getValue())
/* Code transformation */
```

Fix for LLVM bug #26323

```
warn("Fixing patch reached!");
if (Not.isPowerOf2()) {
    if (!!(C->getValue().isPowerOf2()
        && Not != C->getValue()))
        warn("Bug triggered");
    else /* Code transformation */
} 
```

Warning-Laden Compiler
Stage 1: Compile-Time Analysis

Buggy Compiler Source

```
if (Not.isPowerOf2())
/* Code transformation */
```

Fixed Compiler Source

```
if (Not.isPowerOf2())
&& C->getValue().isPowerOf2()
&& Not != C->getValue())
/* Code transformation */
```

**Fix for LLVM bug #26323**

**Warning-Laden Compiler**

```
warn("Fixing patch reached!");
if (Not.isPowerOf2()) {
  if (!C->getValue().isPowerOf2()
    && Not != C->getValue())
    warn("Bug triggered!");
  else /* Code transformation */
}
```

**grep logs**

"Fixing patch reached!"
"Bug triggered!"
Stage 2: Syntactic Binary Analysis

Buggy Compiler

```c
if (Not.isPowerOf2())
```

Fixed Compiler

```c
if (Not.isPowerOf2())
    && C->getValue().isPowerOf2()
    && Not != C->getValue())
```
Stage 2: Syntactic Binary Analysis

Buggy Compiler

if (Not.isPowerOf2())

Fixed Compiler

if (Not.isPowerOf2())
&& C->getValue().isPowerOf2()
&& Not != C->getValue())
Stage 2: Syntactic Binary Analysis

Buggy Compiler

```c
if (Not.isPowerOf2())
```

Fixed Compiler

```c
if (Not.isPowerOf2()
    && C->getValue().isPowerOf2()
    && Not != C->getValue())
```
Stage 2: Syntactic Binary Analysis

Buggy Compiler

```
if (Not.isPowerOf2())
```

Fixed Compiler

```
if (Not.isPowerOf2() && C->getValue().isPowerOf2() && Not != C->getValue())
```
Stage 2: Syntactic Binary Analysis

Buggy Compiler

\[
\text{if (Not.isPowerOf2())}
\]

Fixed Compiler

\[
\text{if (Not.isPowerOf2())}
\]
\[
\&\& \text{C->getValue().isPowerOf2()}
\]
\[
\&\& \text{Not != C->getValue())}
\]

Check for syntactic differences in assembly

Textual comparison opcode-by-opcode

mov $5, %eax
addl $4, %esp
Stage 2: Syntactic Binary Analysis

Buggy Compiler

if (Not.isPowerOf2())

if (Not.isPowerOf2() && C->getValue().isPowerOf2() && Not != C->getValue())

Fixed Compiler

Check for syntactic differences in assembly

Textual comparison opcode-by-opcode

→ Limit false positives (registers, etc.)
→ No false negatives with our bugs

"bad"

"good"
Stage 2: Syntactic Binary Analysis

Buggy Compiler

\[
\text{if} \ (\text{Not}. \text{isPowerOf2}())
\]

Fixed Compiler

\[
\text{if} \ (\text{Not}. \text{isPowerOf2}()) \\
\&\& \ C->\text{getValue}().\text{isPowerOf2}() \\
\&\& \ \text{Not} \ != \ C->\text{getValue}()
\]

If non-reproducible build process, some assembly differences might not be caused by the fixing patch.
Stage 3: Dynamic Binary Analysis
Stage 3: Dynamic Binary Analysis
Stage 3: Dynamic Binary Analysis
Stage 3: Dynamic Binary Analysis

- **Test divergence** ≠ **Miscompilation** (flaky tests)

- **No test divergence** ≠ **No miscompilation** (test suite strength)
Stage 3: Dynamic Binary Analysis
Stage 3: Dynamic Binary Analysis

Sample of syntactic differences in assembly from Stage 2

```
mov $5, %eax  |  addl $4, %esp
addl $4, %esp |  mov $5, %eax
             |  addl $4, %esp |  mov $5, %eax
addl $4, %esp |  mov $5, %eax
mov $5, %eax  |  addl $4, %esp
             |  mov $5, %eax
```
Stage 3: Dynamic Binary Analysis

Sample of syntactic differences in assembly from Stage 2
Stage 3: Dynamic Binary Analysis

1. int func(){
   ...  
12. x = f(x,y);

Sample of syntactic differences in assembly from Stage 2
Stage 3: Dynamic Binary Analysis

1. int func(){
   ...
12. x = f(x,y);

Manual crafting of local or global inputs to trigger runtime divergence

Sample of syntactic differences in assembly from Stage 2
Outline

1. Context: compiler fuzzing
2. Problem: importance of fuzzer-found miscompilations is unclear
3. Goal: a study of the practical impact of miscompilation bugs
4. Methodology for bug impact measurement
5. Experiments and results
6. Conclusions
7. Future work
Experiments (1/2)

We apply our bug impact measurement methodology over a sample of:

- 45 miscompilations bugs in the open-source LLVM compiler (C/C++ → x86_64)
- 27 fuzzer-found bugs (12% of miscompilations from Csmith, EMI, Orange and Yarpgen)
- 10 bugs detected by compiling real code and 8 bugs from Alive formal verification tool
We apply our bug impact measurement methodology over a sample of:

- 309 Debian packages totalling 10M+ lines of C/C++ code
- Not part of the LLVM test suite and with a reproducible build process
- Diverse set of applications w.r.t. type, size, popularity and maturity

> grep
A lot of manual effort and 5 months of computation happen here
Results

- 27 fuzzer-found bugs
- 10 bugs affecting real code
- 8 formal verification bugs

- Patch reached
  - Stage 1a: 70%
  - Stage 1b: 65%
  - Different binary: 43%
- Bug triggered
  - Stage 1b: 28%
  - Different binary: 19%
  - Different binary: 13%
- Different binary
  - Stage 2: 6%
  - Stage 3: 2%
- Test divergence
  - Stage 3: 7%

M. Marcozzi
Compiler Fuzzing: How Much Does It Matter?
Results

Stage 1

All bug-finding approaches discover bugs frequently reached and sometimes triggered when compiling real code.

Stage 1a

70% Patch reached
65% 10 bugs affecting real code
43% 27 fuzzer-found bugs

Stage 1b

28% Bug triggered
19% 8 formal verification bugs
13%

Stage 2

Different binary
6% 2% 7%

Stage 3

Test divergence
0.01% 0.01% 0%
Results

Stage 1

All bug-finding approaches discover bugs frequently reached and sometimes triggered when compiling real code.

Yet, bug triggering detection had often to be over-approximated!

Stage 1a: 70% Patch reached, 65% Bug triggered
Stage 1b: 43% Patch reached, 28% Bug triggered, 19% Different binary, 13% Test divergence

Stage 2: 6% Different binary, 2% Test divergence
Stage 3: 7% Test divergence

27 fuzzer-found bugs, 10 bugs affecting real code, 8 formal verification bugs

M. Marcozzi
Compiler Fuzzing: How Much Does It Matter?
Results

Stage 2

Binary differences only affect a small fraction of package builds, deeper inspection shows that only a tiny fraction of package functions are touched.
Results

Stage 2

Binary differences only affect a small fraction of package builds, deeper inspection shows that only a tiny fraction of package functions are touched.

Stage 1a
- Patch reached: 70%
- Bug triggered: 65%

Stage 1b
- Patch reached: 43%
- Bug triggered: 28%

Stage 2
- Different binary: 19%
- Test divergence: 13%

Stage 3
- 27 fuzzer-found bugs
- 10 bugs affecting real code
- 8 formal verification bugs

Number of affected functions (out of 202k):
- 0
- 1000
- 2000
- 3000
- 4000
- 5000
- 6000

Bug identifier:
- Csmith #11964
- Csmith #11977
- EMI #26323
- EMI #28610
- Orange #15959
- Alive #20189
- Real #27903
- Real #33708

M. Marcozzi
Compiler Fuzzing: How Much Does It Matter?
Results

Stage 3

In total, **miscompilations** caused only **three package test failures**

- **Stage 1a**: 70% Patch reached, 65% 27 fuzzer-found bugs, 43% 10 bugs affecting real code, 13% 8 formal verification bugs
- **Stage 1b**: 28% Bug triggered, 19% 27 fuzzer-found bugs, 13% 10 bugs affecting real code, 6% 8 formal verification bugs
- **Stage 2**: 6% Different binary, 2% 27 fuzzer-found bugs, 7% 10 bugs affecting real code, 0% 8 formal verification bugs
- **Stage 3**: 0.01% Test divergence, 0% 27 fuzzer-found bugs, 0% 10 bugs affecting real code, 0% 8 formal verification bugs

In total, miscompilations caused only three package test failures.
Results

Stage 3

In total, miscompilations caused only three package test failures.

One test failure in zsh (+ one extra test failure in SQLite).

27 fuzzer-found bugs
10 bugs affecting real code
8 formal verification bugs

Stage 1a
Patch reached: 70% 65% 43%
Stage 1b
Bug triggered: 28% 19% 13%
Stage 2
Different binary: 6% 2% 7%
Stage 3
Test divergence: 0.01% 0.01% 0%

Stage 3

Stage 1a
Stage 1b
Stage 2
Stage 3

M. Marcozzi
Compiler Fuzzing: How Much Does It Matter?
Results

Stage 3

- In total, **miscompilations** caused only **three package test failures**
  - One test failure in `zsh`
  - (+ one extra test failure in `SQLite`)
  - One test failure in `leveldb`

Stage 1a
- Patch reached: 70%
- Bug triggered: 65%
- Different binary: 43%

Stage 1b
- Patch reached: 28%
- Bug triggered: 19%
- Different binary: 13%

Stage 2
- Patch reached: 6%
- Bug triggered: 2%
- Different binary: 7%

Stage 3

- Test divergence
  - 0.01% 0.01% 0%

- 27 fuzzer-found bugs
- 10 bugs affecting real code
- 8 formal verification bugs
Test Failure in SQLite

- Miscompilation is **caused by LLVM bug #13326**, found by Csmith
- Bug affects **translation of 8-bits unsigned integer division** from IR (\texttt{udiv}) to x86
- When divisor is constant, **translation is wrong** for 6 of 65k possible divisor values
- In SQLite, the **following line of source code** is miscompiled, triggering a test failure:

  \[
  \text{zBuf}[i] = \text{zSrc}[\text{zBuf}[i]\% (\text{sizeof}(\text{zSrc})-1)];
  \]
Test Failure in SQLite

• Miscompilation is caused by LLVM bug #13326, found by Csmith

• Bug affects translation of 8-bits unsigned integer division from IR (udiv) to x86

• When divisor is constant, translation is wrong for 6 of 65k possible divisor values

• In SQLite, the following line of source code is miscompiled, triggering a test failure:

\[
z\text{Buf}[i] = z\text{Src}[z\text{Buf}[i] \%(\text{sizeof}(z\text{Src})-1)];
\]
Test Failure in SQLite

• Miscompilation is caused by LLVM bug #13326, found by Csmith

• Bug affects translation of 8-bits unsigned integer division from IR (udiv) to x86

• When divisor is constant, translation is wrong for 6 of 65k possible divisor values

• In SQLite, the following line of source code is miscompiled, triggering a test failure:

  \[ z\text{Buf}[i] = z\text{Src}[z\text{Buf}[i] \% (\text{sizeof}(z\text{Src}) - 1)]; \]

  \[ 79 \]

  COMPILE TIME
Test Failure in SQLite

- Miscompilation is **caused by LLVM bug #13326**, found by Csmith
- Bug affects **translation of 8-bits unsigned integer division** from IR (udiv) to x86
- When divisor is constant, **translation is wrong** for 6 of 65k possible divisor values
- In SQLite, the **following line of source code** is miscompiled, triggering a test failure:

  ```c
  zBuf[i] = zSrc[zBuf[i]%((\texttt{sizeof(zSrc)}-1)));  
  ```

  **COMPILE TIME**
Test Failure in SQLite

• Miscompilation is caused by LLVM bug #13326, found by Csmith

• Bug affects translation of 8-bits unsigned integer division from IR (udiv) to x86

• When divisor is constant, translation is wrong for 6 of 65k possible divisor values

• In SQLite, the following line of source code is miscompiled, triggering a test failure:

\[
z\text{Buf}[i] = z\text{Src}[z\text{Buf}[i]%(\text{sizeof}(z\text{Src})-1)];
\]

Wrong modulo binary code generated

\[
79 \quad 78
\]

COMPILE TIME
Test Failure in SQLite

• Miscompilation is **caused by LLVM bug #13326**, found by Csmith

• Bug affects **translation of 8-bits unsigned integer division** from IR (\texttt{udiv}) to x86

• When divisor is constant, **translation is wrong** for 6 of 65k possible divisor values

• In SQLite, the **following line of source code** is miscompiled, triggering a test failure:

\[
z\text{Buf}[i] = z\text{Src}[z\text{Buf}[i] \% (\texttt{sizeof}(z\text{Src})-1)];\]

78
Test Failure in SQLite

• Miscompilation is **caused by LLVM bug #13326**, found by Csmith

• Bug affects **translation of 8-bits unsigned integer division** from IR (udiv) to x86

• When divisor is constant, **translation is wrong** for 6 of 65k possible divisor values

• In SQLite, the **following line of source code** is miscompiled, triggering a test failure:

\[
\text{zBuf}[i] = \text{zSrc}[\text{zBuf}[i] \% (\text{sizeof(zSrc)}-1)];
\]

**TEST RUN TIME**
Test Failure in SQLite

- Miscompilation is **caused by LLVM bug #13326**, found by Csmith
- Bug affects **translation of 8-bits unsigned integer division** from IR (udiv) to x86
- When divisor is constant, **translation is wrong** for 6 of 65k possible divisor values
- In SQLite, the **following line of source code** is miscompiled, triggering a test failure:

  $$zBuf[i] = zSrc[zBuf[i] % (\text{sizeof}(zSrc)-1)];$$

  \[232\quad 78\]

**TEST RUN TIME**
Test Failure in SQLite

- Miscompilation is **caused by LLVM bug #13326**, found by Csmith
- Bug affects **translation of 8-bits unsigned integer division** from IR (\texttt{udiv}) to x86
- When divisor is constant, **translation is wrong** for 6 of 65k possible divisor values
- In SQLite, the **following line of source code** is miscompiled, triggering a test failure:

\[
z\text{Buf}[i] = z\text{Src}[z\text{Buf}[i] \% (\text{sizeof}(z\text{Src})-1)];
\]

**TEST RUN TIME**

\[232\quad 78\quad 254\text{ (out of range)}\]
Test Failure in SQLite

- Miscompilation is **caused by LLVM bug #13326**, found by Csmith

- Bug affects **translation of 8-bits unsigned integer division** from IR (`udiv`) to x86

- When divisor is constant, **translation is wrong** for 6 of 65k possible divisor values

- In SQLite, the **following line of source code** is miscompiled, triggering a test failure:

  \[
  z\text{Buf}[i] = z\text{Src}[z\text{Buf}[i] \% (\text{sizeof}(z\text{Src}) - 1)];
  \]

  **Garbage value**

  \[
  232 \quad 78
  \]

  **Test Run Time**

  \[
  254 \text{ (out of range)}
  \]
Results

Stage 3

In total, **miscompilations** caused only **three package test failures**

![Bar Chart]

- **Stage 1a**
  - Patch reached: 70%
  - Bug triggered: 65%
  - Different binary: 43%

- **Stage 1b**
  - Patch reached: 28%
  - Bug triggered: 19%
  - Different binary: 13%

- **Stage 2**
  - Patch reached: 6%
  - Bug triggered: 2%
  - Different binary: 7%

- **Test divergence**
  - 0.01% 0.01% 0%

- **Stage 3**

27 fuzzer-found bugs
10 bugs affecting real code
8 formal verification bugs

---

M. Marcozzi

Compiler Fuzzing: How Much Does It Matter?
Results

Stage 3

In total, **miscompilations** caused only three package test failures.

Is it due to **very weak test coverage**?

<table>
<thead>
<tr>
<th>Fraction of package builds</th>
<th>Stage 1a</th>
<th>Stage 1b</th>
<th>Stage 2</th>
<th>Stage 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Patch reached</td>
<td>70%</td>
<td>65%</td>
<td>43%</td>
<td>0%</td>
</tr>
<tr>
<td>Bug triggered</td>
<td>28%</td>
<td>19%</td>
<td>13%</td>
<td>7%</td>
</tr>
<tr>
<td>Different binary</td>
<td>6%</td>
<td>2%</td>
<td>7%</td>
<td>0.01%</td>
</tr>
<tr>
<td>Test divergence</td>
<td>0.01%</td>
<td>0.01%</td>
<td>0%</td>
<td>0%</td>
</tr>
</tbody>
</table>

27 fuzzer-found bugs, 10 bugs affecting real code, 8 formal verification bugs.
Results

Stage 3

In total, **miscompilations** caused only **three package test failures**

Is it due to **very weak test coverage**?

Sample of Package Test Suites

- 47% average statement coverage
- Half suites > 50% statement coverage

---

Stage 1a
- 70% Patch reached
- 65% 27 fuzzer-found bugs

Stage 1b
- 28% Bug triggered
- 19% 10 bugs affecting real code
- 13% 8 formal verification bugs

Stage 2
- 6% Different binary
- 2% 0.01%
- 7% 0.01%

Stage 3
- 0% Test divergence

---

M. Marcozzi

Compiler Fuzzing: How Much Does It Matter?
Results

Stage 3

In total, **miscompilations** caused only **three package test failures**

Is it due to **very weak test coverage**?

Sample of Package Test Suites

47% average statement coverage

Half suites > 50% statement coverage

SQLite

98% statement coverage of 151kLoC

Stage 1a

- Patch reached: 70%
- Bug triggered: 43%

Stage 1b

- Bug triggered: 28%
- Different binary: 6%

Stage 2

- Different binary: 2%
- Test divergence: 7%

Stage 3

- Miscompilations: 0.01%
- Miscompilations: 0.01%
- Miscompilations: 0%

Sample of Package Test Suites

47% average statement coverage

Half suites > 50% statement coverage

SQLite

98% statement coverage of 151kLoC
Results

Stage 3

In total, **miscompilations** caused only **three package test failures**

---

- **Stage 1a**: 70% Patch reached, 65% 27 fuzzer-found bugs, 43% 10 bugs affecting real code
- **Stage 1b**: 28% Bug triggered, 19% 8 formal verification bugs, 13% other
- **Stage 2**: 6% Different binary, 2% 0.01% Test divergence, 7% 0.01%
- **Stage 3**: 0%
Results

Stage 3

In total, miscompilations caused only three package test failures.

What does manual inspection of assembly differences reveal?

- 27 fuzzer-found bugs
- 10 bugs affecting real code
- 8 formal verification bugs

Stage 1a: Patch reached
- 70% of builds

Stage 1b: Bug triggered
- 65% of builds
- 43% of builds

Stage 2: Different binary
- 28% of builds
- 19% of builds
- 13% of builds

Stage 3: Test divergence
- 6% of builds
- 2% of builds
- 7% of builds

Stage 3
- 0.01% of builds
- 0.01% of builds
- 0% of builds

In total, miscompilations caused only three package test failures.

What does manual inspection of assembly differences reveal?
Manual Inspection of Assembly Differences

• We inspected about 50 differences in package assembly code.
• For each, we tried and failed to craft inputs triggering a runtime divergence.
• In practice, differences have no or little impact over package semantics:
  • Compiler maintainers often deactivate whole parts of features instead of fixing them.
  • Specific runtime circumstances often necessary for miscompilation to cause failure.
Outline

1. Context: compiler fuzzing
2. Problem: importance of fuzzer-found miscompilations is unclear
3. Goal: a study of the practical impact of miscompilation bugs
4. Methodology for bug impact measurement
5. Experiments and results
6. Conclusions
7. Future work
Conclusions

• Our **two major take-aways** are that miscompilations bugs in a mature compiler…
  
  • **seldom impact** app reliability (as probed by test suites and manual inspection)
  
  • have **similar impact** no matter they were found in **real** or **fuzzer-generated** code
  
• A **possible explainer** for these results is that, in a mature compiler…

 💡 all the bugs **affecting patterns frequent in real code** have **already been fixed**

 💡 only **corner-case bugs remain**, affecting real and generated code similarly
Outline

1. Context: compiler fuzzing
2. Problem: importance of fuzzer-found miscompilations is unclear
3. Goal: a study of the practical impact of miscompilation bugs
4. Methodology for bug impact measurement
5. Experiments and results
6. Conclusions
7. Future work
Future Work

- Our main **research directions** for even better evaluation of compiler bugs impact:

  1. **Better probe differences in assembly**: symbolic execution + multi-version execution
  2. **Exploit methodology and artefact**: replication, more bugs, less mature compiler, etc.
  3. **Consider impact on non-functional properties**: speed, compiler-induced backdoors, etc.
Thank you for listening!

> Open access to paper
https://dl.acm.org/doi/10.1145/3360581

> Fully reusable artefact
https://doi.org/10.5281/zenodo.3403703

www.marcozzi.net
@michaelmarcozzi