Compiler Fuzzing: How Much Does It Matter?

~ research published at the SPLASH’19 OOPSLA conference ~

*Michaël Marcozzi1  *Qiyi Tang2  Alastair F. Donaldson3,1  Cristian Cadar1

*The presented experimental study has been carried out equally by M. Marcozzi and Q. Tang.
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**
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\(^4\)
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
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 affecting **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.png)

  **Buggy Compiler Source**

  ![](fixing_patch.png)

  **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

- **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

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

- **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

  ![Diagram showing the transition from buggy to fixed compiler source through 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:**
  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

  ![](image)

  Buggy Compiler Source → Fixing Patch written by developers → Fixed Compiler Source

• **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

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

Buggy Compiler Source

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

Fixed Compiler Source

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

Buggy Compiler Source

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

Fixed Compiler Source

```cpp
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 */
```

Warning-Laden Compiler

```
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 */

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

Warning-Laden Compiler

grep logs
"Fixing patch reached!"
| "Bug triggered!"
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

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

Fixed Compiler

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

Buggy Compiler

```
if (!isPowerOf2())
```

Fixed Compiler

```
if (!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())

Check for syntactic differences in assembly
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
Stage 2: Syntactic Binary Analysis

Buggy Compiler

if (Not.isPowerOf2())

Fixed Compiler

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

Check for syntactic differences in assembly

Textual comparison opcode-by-opcode

→ Limit false positives (registers, etc.)
→ No false negatives with our bugs
Stage 2: Syntactic Binary Analysis

Buggy Compiler

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

Fixed Compiler

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

Count divergent test results
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

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

Sample of syntactic differences in assembly from Stage 2

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

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

Sample of syntactic differences in assembly from Stage 2

Compiler Fuzzing: How Much Does It Matter?

M. Marcozzi 16
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

```bash
> 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

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

Stage 1

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

Stage 1a

- Patch reached: 70%
- Bug triggered: 65%

Stage 1b

- Different binary: 43%
- Test divergence: 19%

Stage 2

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

Stage 3

- 70% reach
- 65% reach
- 43% reach
- 19% reach
- 13% reach

M. Marcozzi
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% 13%

Stage 2

- 6% Different binary
- 2% Test divergence
- 7% 0.01% 0.01% 0%

Stage 3

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

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

Stage 2
- 6% 2% 7%

Stage 3
- 0.01% 0.01% 0%
Results

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% of package builds
- Bug triggered
  - 65% of package builds
  - 43% of affected functions

Stage 1b
- Different binary
  - 28% of package builds
  - 19% of affected functions
- Test divergence
  - 13% of package builds
  - 6% of affected functions

Stage 2
- 6% of package builds
- 2% of affected functions
- 7% of package builds

Stage 3
- 0% of package builds
- 0.01% of affected functions
- 0% of package builds

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.

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

Stage 1a
- 70% Patch reached
- 65% 43%

Stage 1b
- 28% 19% 13%

Stage 2
- 6% 2% 7%

Stage 3
- 0.01% 0.01% 0%

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**)

M. Marcozzi

Compiler Fuzzing: How Much Does It Matter?

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

<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>43%</td>
<td>6%</td>
<td>0.01%</td>
</tr>
<tr>
<td>Bug triggered</td>
<td>65%</td>
<td>19%</td>
<td>2%</td>
<td>0.01%</td>
</tr>
<tr>
<td>Different binary</td>
<td>28%</td>
<td>13%</td>
<td>7%</td>
<td>0%</td>
</tr>
</tbody>
</table>

Stage 1a

Stage 1b

Stage 2

Stage 3

Test divergence

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

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

M. Marcozzi

Compiler Fuzzing: How Much Does It Matter?
Results

Stage 1a
- Patch reached: 70% blue
- Bug triggered: 43% green

Stage 1b
- Bug triggered: 28% blue, 19% green, 13% yellow

Stage 2
- Different binary: 6% blue, 2% green, 7% yellow

Stage 3
- Test divergence: 0.01% blue, 0.01% green, 0% yellow

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

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

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

24
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:

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

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

**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:

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

  **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 \(u\text{div}\) 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:

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

**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:

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

  **Wrong modulo binary code generated**
  
  ```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 (\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:

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

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

TEST RUN TIME

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 ($\text{udiv}$) 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:

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

  \[
  \underline{232} \quad \underline{78}
  \]

**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 (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**

\[\begin{array}{c}
232 \\
254 (\text{out of range}) \\
\end{array}\]
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:

  ```c
  zBuf[i] = zSrc[zBuf[i] % (sizeof(zSrc)-1)];
  ```

<table>
<thead>
<tr>
<th>zBuf[i]</th>
<th>zSrc[zBuf[i]] % (sizeof(zSrc)-1)</th>
<th>Garbage value</th>
</tr>
</thead>
<tbody>
<tr>
<td>232</td>
<td>78</td>
<td>254 (out of range)</td>
</tr>
</tbody>
</table>
Stage 3

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

Results

Stage 1a
- Patch reached: 70%
- 27 fuzzer-found bugs
- 10 bugs affecting real code
- 8 formal verification bugs

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

Stage 2
- Different binary: 28%
- 6%
- 2%
- 7%

Stage 3
- Test divergence: 0.01%
- 0.01%
- 0%

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?

Stage 1a
- 27 fuzzer-found bugs
- 10 bugs affecting real code
- 8 formal verification bugs

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

Stage 2
- Test divergence: 0.01% 0.01% 0%

Stage 3
- Miscompilations caused only three package test failures
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
Patch reached: 70%

Stage 1b
Bug triggered: 65%

Stage 2
Different binary: 43%

Stage 3
Test divergence: 0%
Results

Stage 3

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

![Bar chart showing fraction of package builds](chart.png)

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

**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

0.01% 0.01% 0%

Test divergence

Stage 3

---

M. Marcozzi

Compiler Fuzzing: How Much Does It Matter?
Results

Stage 3

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

```
<table>
<thead>
<tr>
<th>Component</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>-</td>
</tr>
<tr>
<td>Different binary</td>
<td>6%</td>
<td>2%</td>
<td>7%</td>
<td>0%</td>
</tr>
<tr>
<td>Test divergence</td>
<td>0.01%</td>
<td>0.01%</td>
<td>0%</td>
<td></td>
</tr>
<tr>
<td>27 fuzzer-found bugs</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10 bugs affecting real code</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 formal verification bugs</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

In total, 27 fuzzer-found bugs, 10 bugs affecting real code, and 8 formal verification bugs were found.
Results

Stage 3

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

What does manual inspection of assembly differences reveal?

![Bar chart showing fraction of package builds across stages]

- **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**: 2% Different binary, 7% 27 fuzzer-found bugs, 6% 10 bugs affecting real code, 0.01% 8 formal verification bugs
- **Stage 3**: Test divergence, 0%
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 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