Séminaire **VERIMAG**
Grenoble, 21/02/2020

Compiler Fuzzing: How Much Does It Matter?

~ *research published at the SPLASH'19 OOPSLA conference* ~

*Michaël Marcozzi[1] *Qiyi Tang[2] Alastair F. Donaldson[3,1] Cristian Cadar[1]

*The presented experimental study has been carried out equally by M. Marcozzi and Q. Tang.

1 Imperial College London

2 University of Oxford

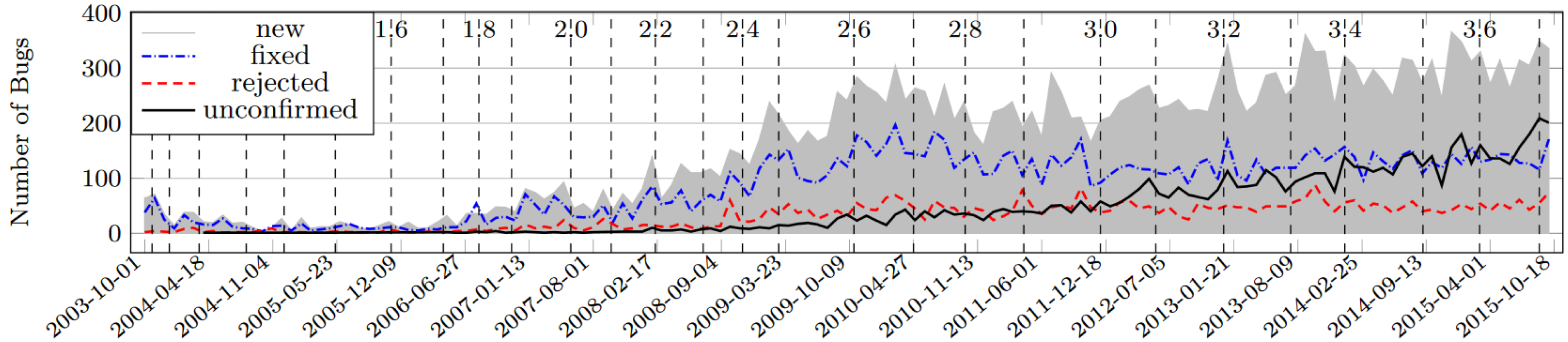
3 Google

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**
  - <u>Formal verification</u>: CompCert verified compiler, Alive optimisation prover, etc.
  - <u>Testing</u>: 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 complier, 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 <u>impact</u> of <u>miscompilation bugs</u> in a <u>mature</u> compiler over <u>real apps</u>
  - ✓ <u>Compare</u> impact of bugs from <u>fuzzers</u> with <u>others</u> (e.g. found by compiling real code)
- Operationally, we aim at **overcoming** the following **challenges**:
  - Take steps towards a <u>methodology</u> to <u>measure the impact</u> of a miscompilation bug
  - Apply it over a <u>significant</u> but <u>tractable</u> 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

- <u>Assumption</u>: 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

Fixing Patch
written by developers

Bug Impact Measurement Methodology

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

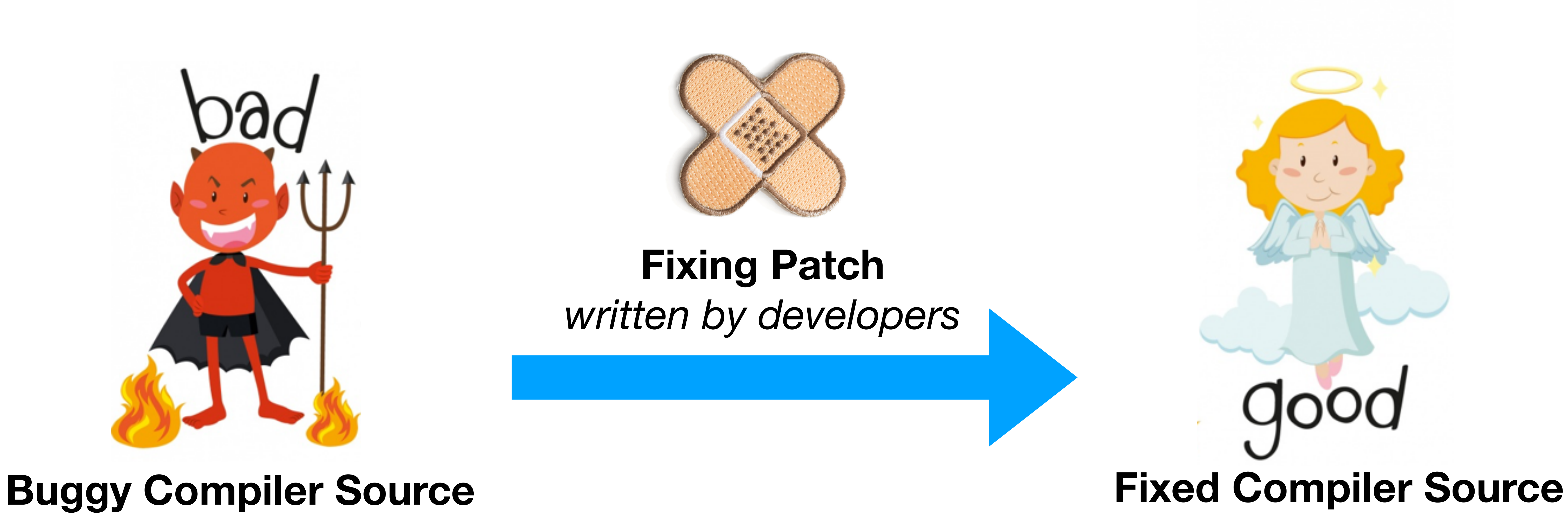
Buggy Compiler Source

Fixing Patch
written by developers

Fixed Compiler Source

Bug Impact Measurement Methodology

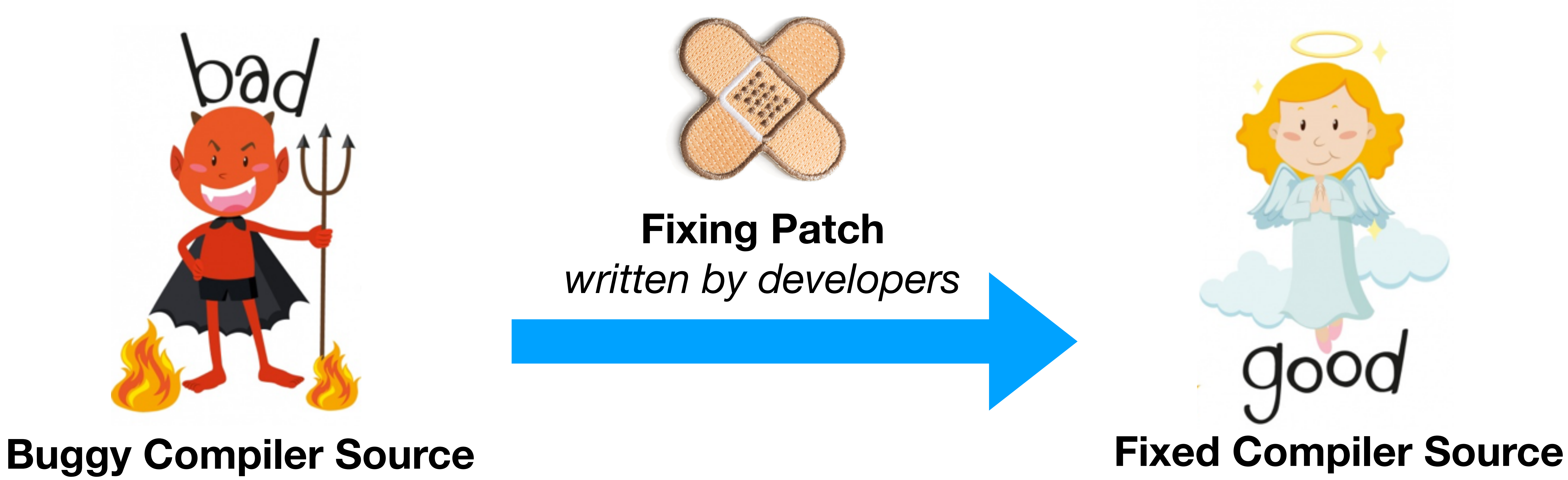
- Assumption: Restrict to **publicly fixed bugs in open-source compilers**, to extract



- 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



- 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

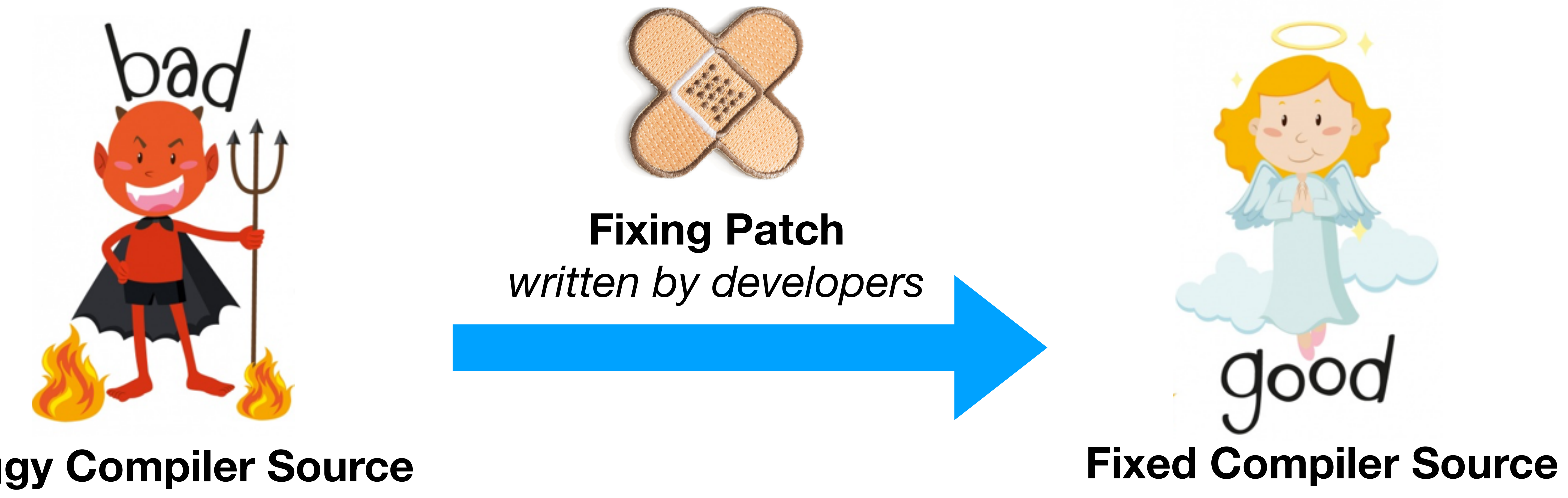
Fixing Patch
written by developers

Buggy Compiler Source → **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?

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

- <u>Assumption</u>: Restrict to **publicly fixed bugs in open-source compilers**, to extract

- <u>Assumption</u>: 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 <u>during compilation</u>?
  2. How much does a triggered bug change the <u>binary code</u>?
  3. Can the binary changes lead to differences in <u>binary runtime behaviour</u>?

Stage 1: Compile-Time Analysis

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

Buggy Compiler Source

fix for LLVM bug #26323

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

Fixed Compiler Source

Stage 1: Compile-Time Analysis

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

Buggy Compiler Source

fix for LLVM bug #26323

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

Fixed Compiler Source

```
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

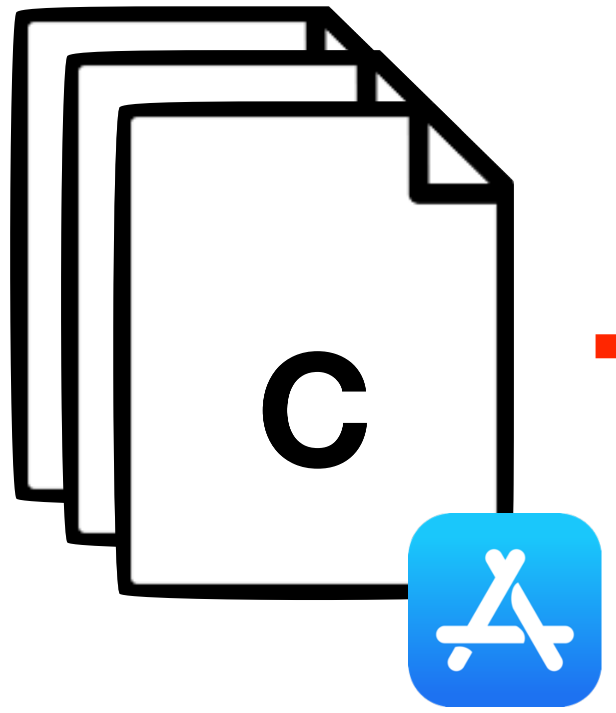
```
if (Not.isPowerOf2())
/* Code transformation */
```

fix for LLVM bug #26323

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

Buggy Compiler Source

Fixed Compiler Source




```
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

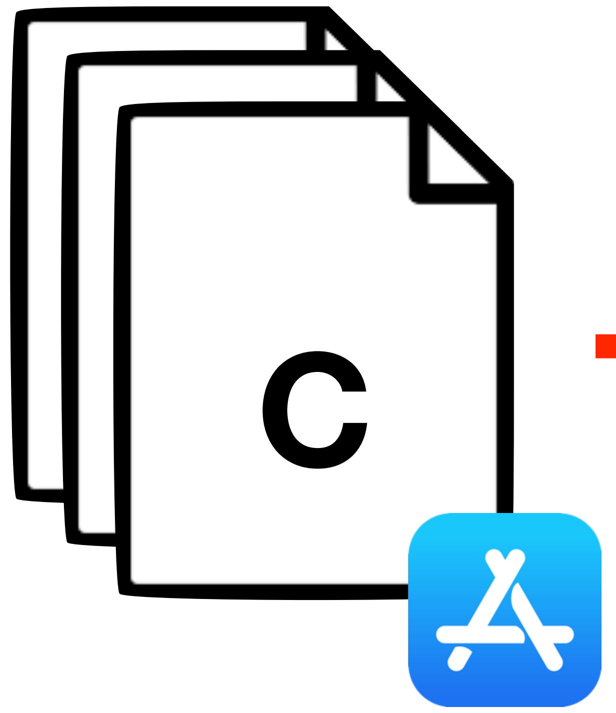
```
if (Not.isPowerOf2())
/* Code transformation */
```

fix for LLVM bug #26323

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

Buggy Compiler Source

Fixed Compiler Source




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

Warning-Laden Compiler

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

Stage 2: Syntactic Binary Analysis

Buggy Compiler

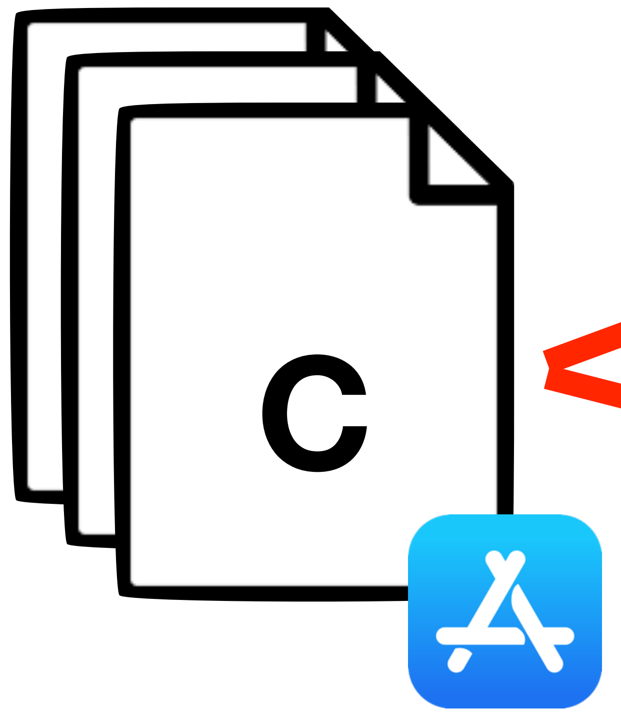

```
if (Not.isPowerOf2())
```

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

Fixed Compiler

Stage 2: Syntactic Binary Analysis

Buggy Compiler

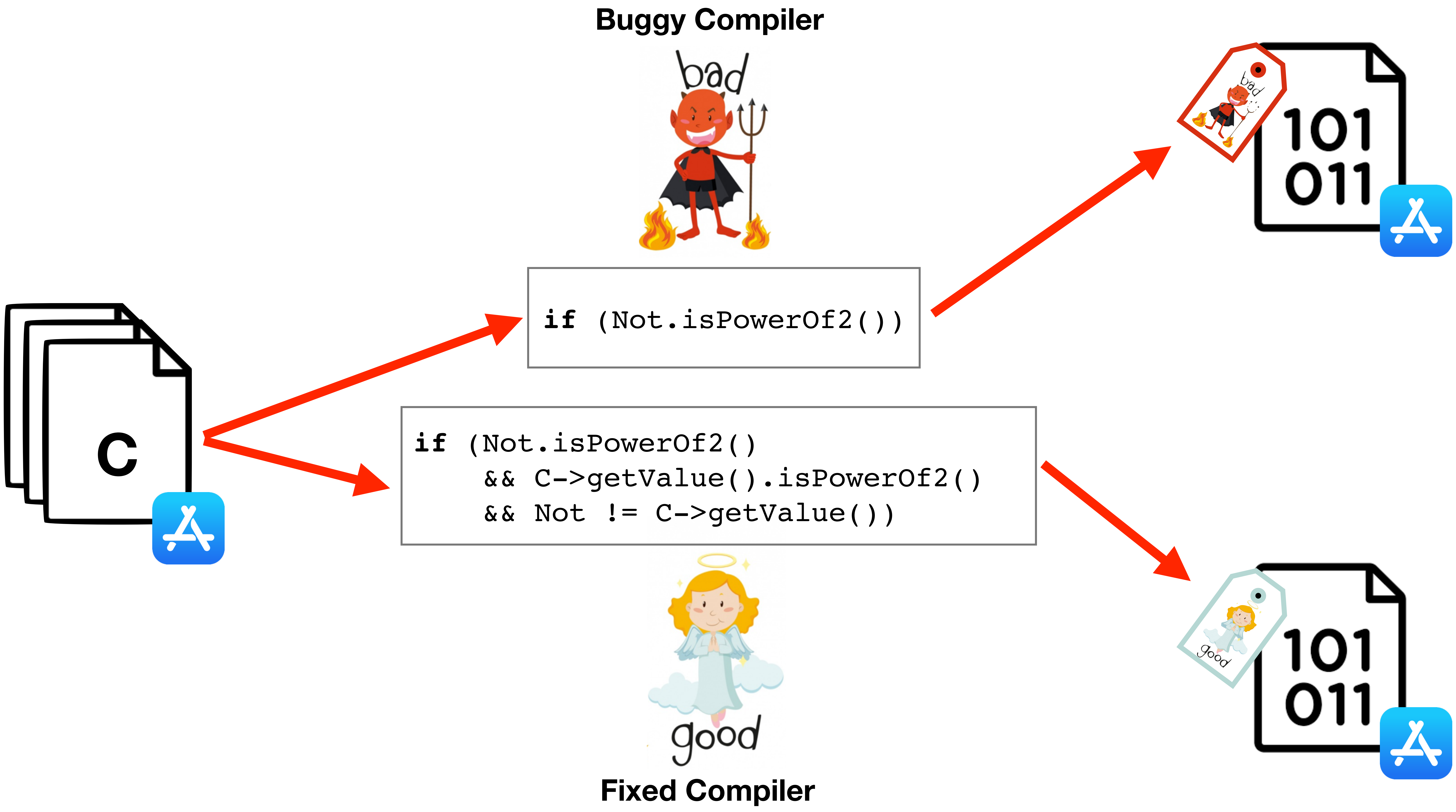



```
if (Not.isPowerOf2())
```

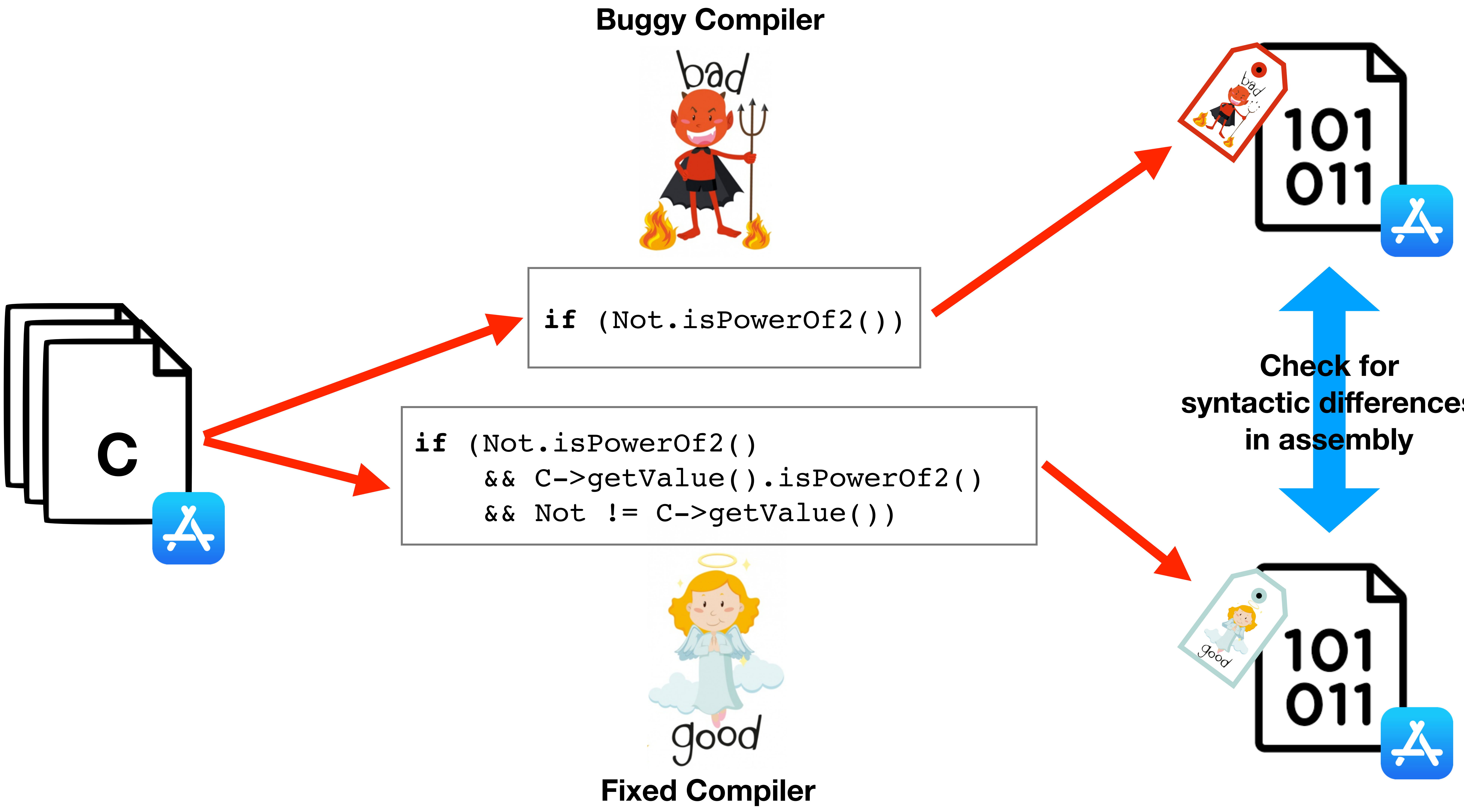
```
if (Not.isPowerOf2()
    && C->getValue().isPowerOf2()
    && Not != C->getValue())
```

Fixed Compiler

Stage 2: Syntactic Binary Analysis



Stage 2: Syntactic Binary Analysis



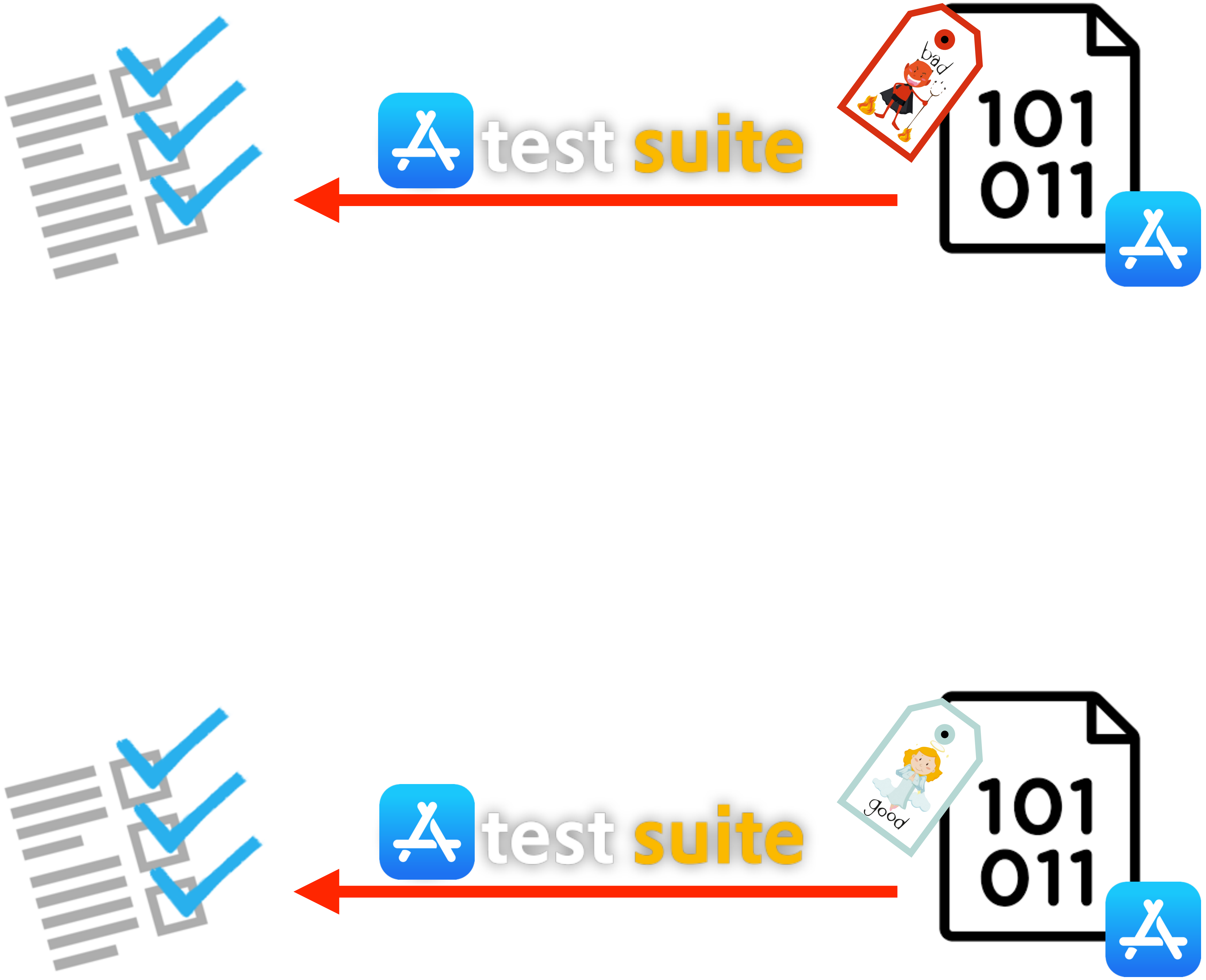
Stage 2: Syntactic Binary Analysis

Stage 2: Syntactic Binary Analysis

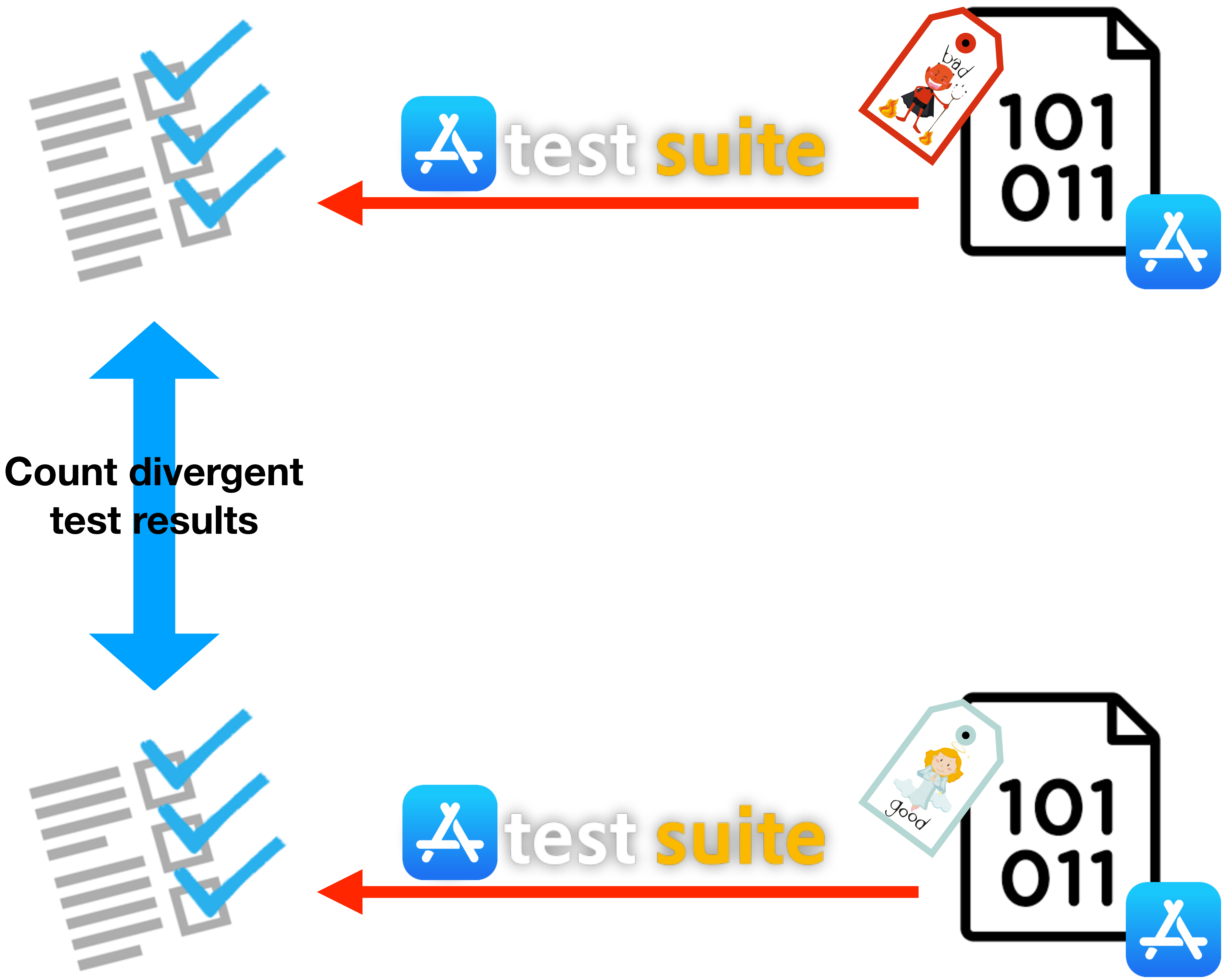
Stage 2: Syntactic Binary Analysis

Stage 3: Dynamic Binary Analysis

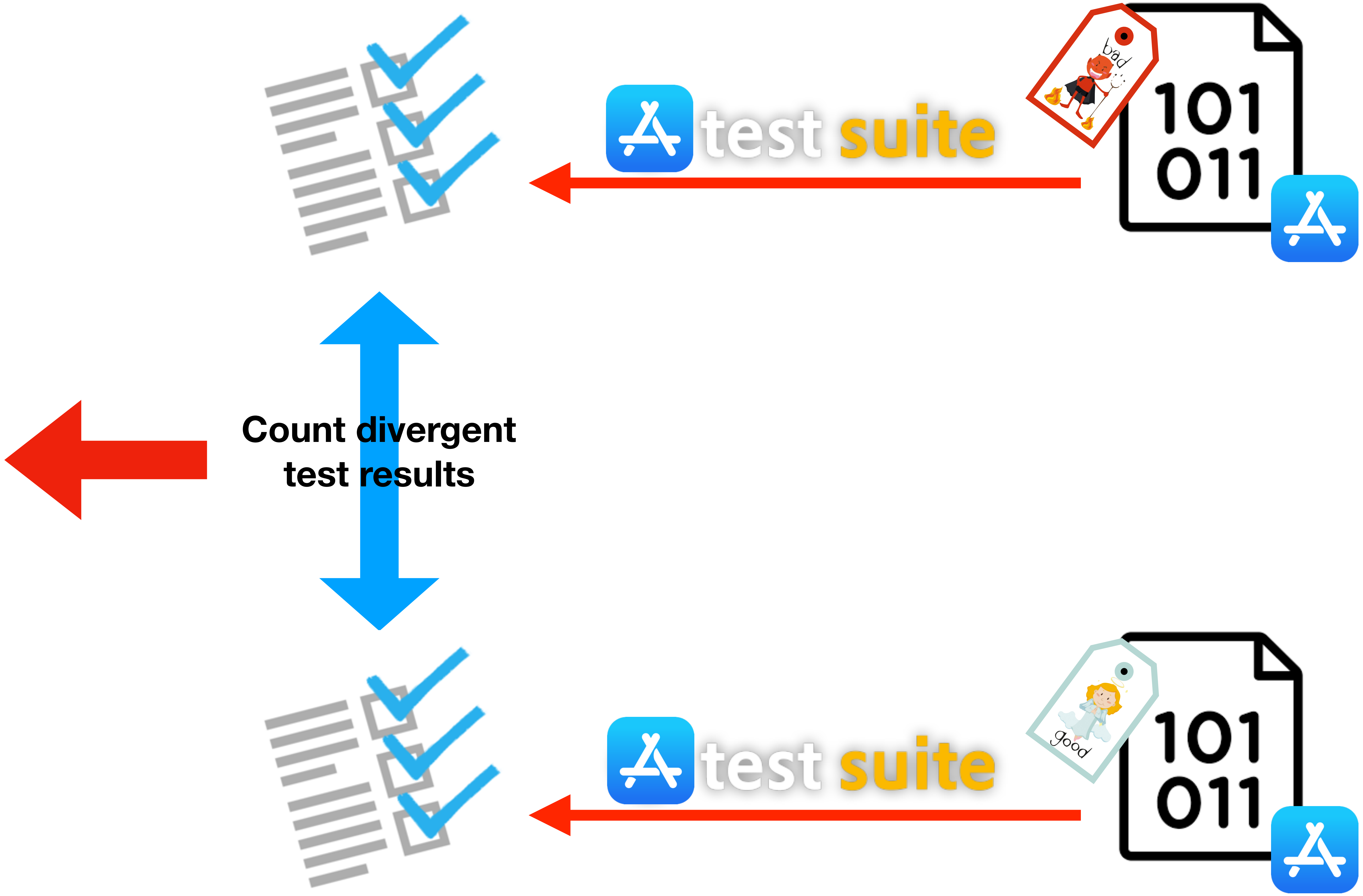
Stage 3: Dynamic Binary Analysis



Stage 3: Dynamic Binary Analysis



Stage 3: Dynamic Binary Analysis



Test divergence
≠
Miscompilation
(flaky tests)

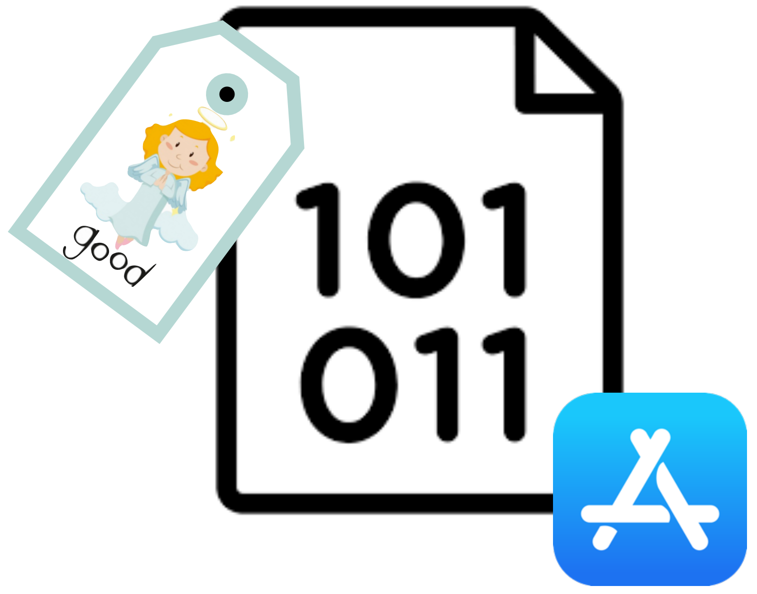
No test divergence
≠
No miscompilation
(test suite strength)

Stage 3: Dynamic Binary Analysis

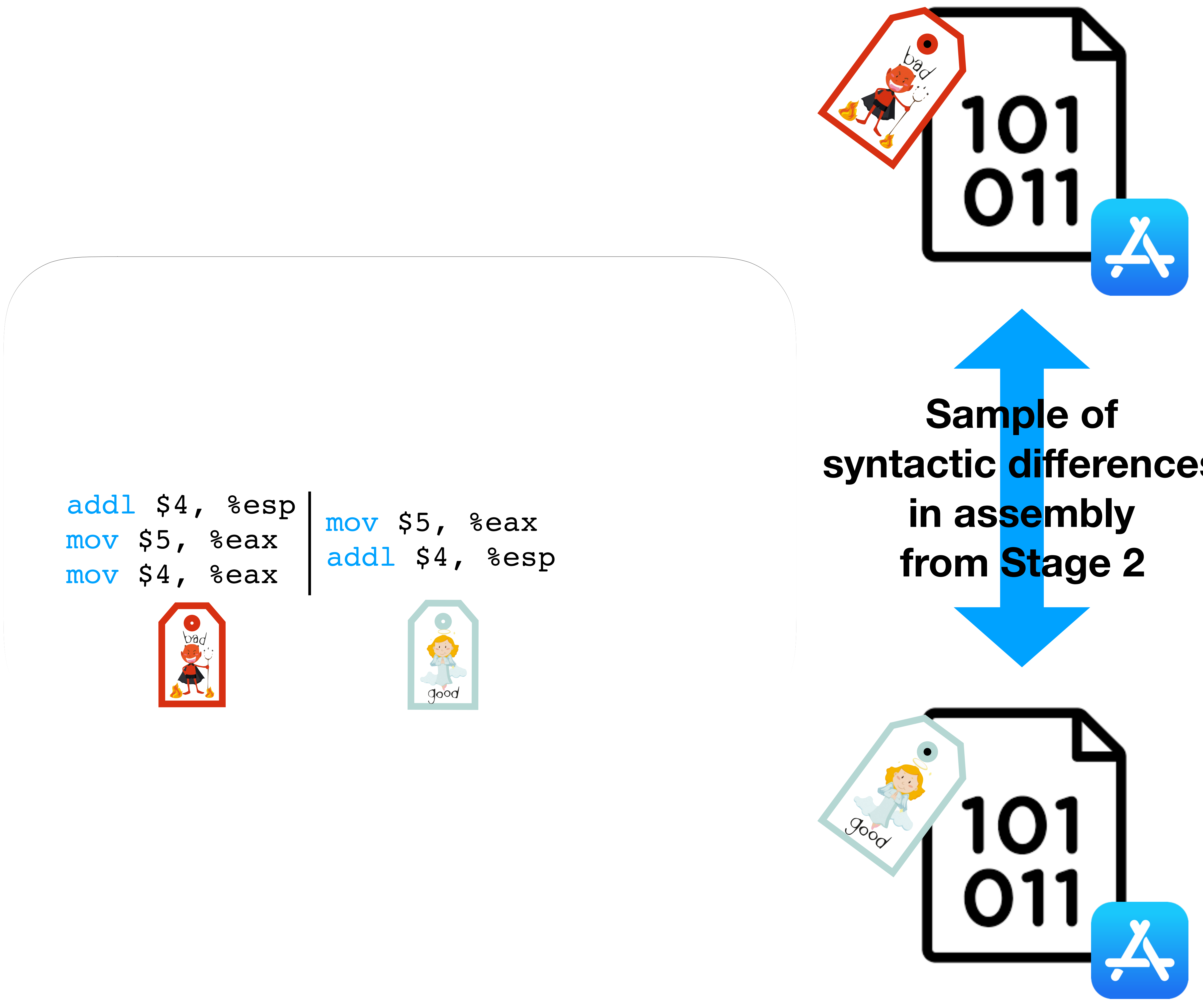
Stage 3: Dynamic Binary Analysis


```
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 $4, %eax  |

mov $5, %eax | addl $4, %esp
```

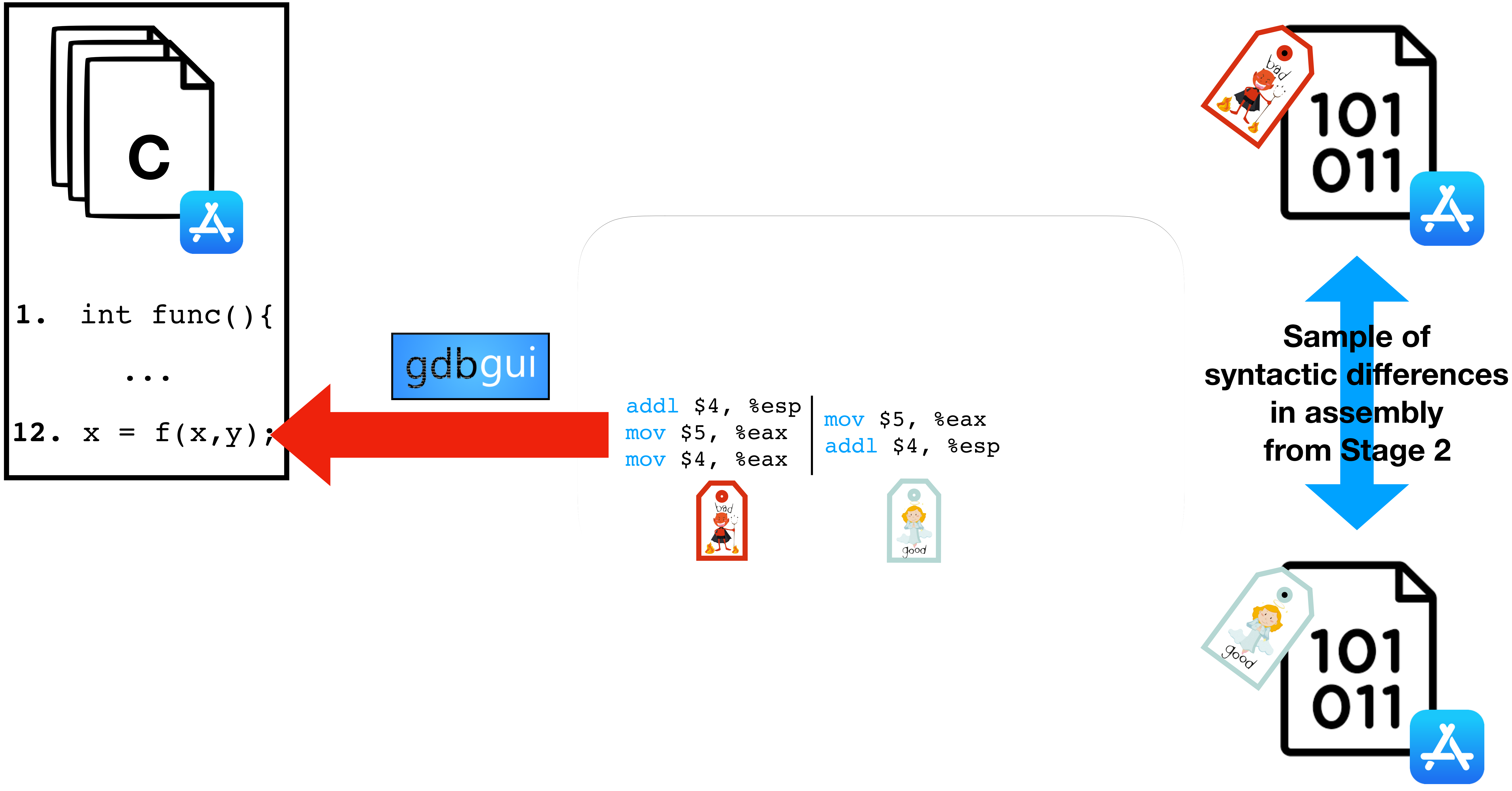
Sample of syntactic differences in assembly from Stage 2



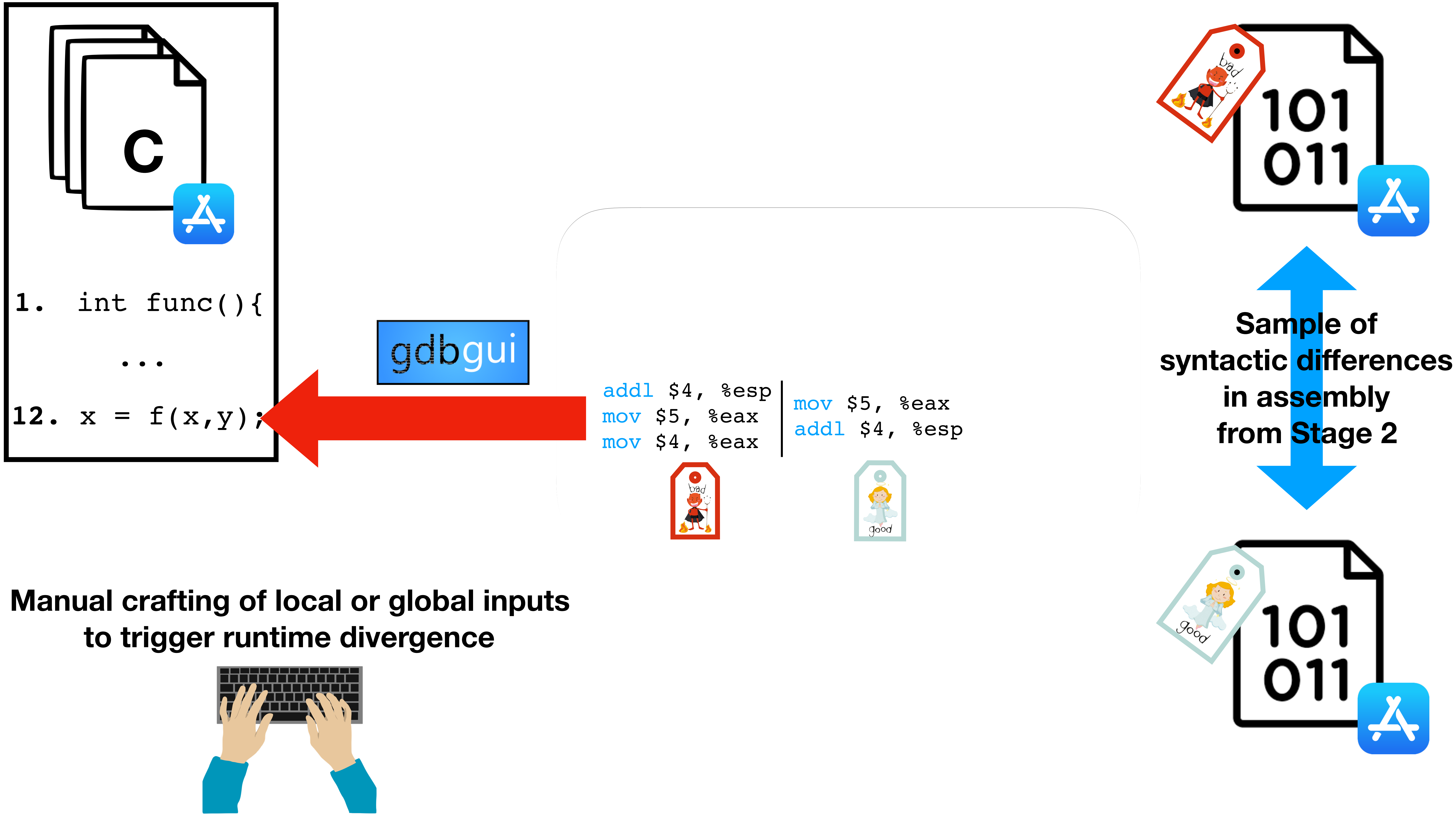
Stage 3: Dynamic Binary Analysis



Stage 3: Dynamic Binary Analysis



Stage 3: Dynamic Binary Analysis



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

Experiments (2/2)

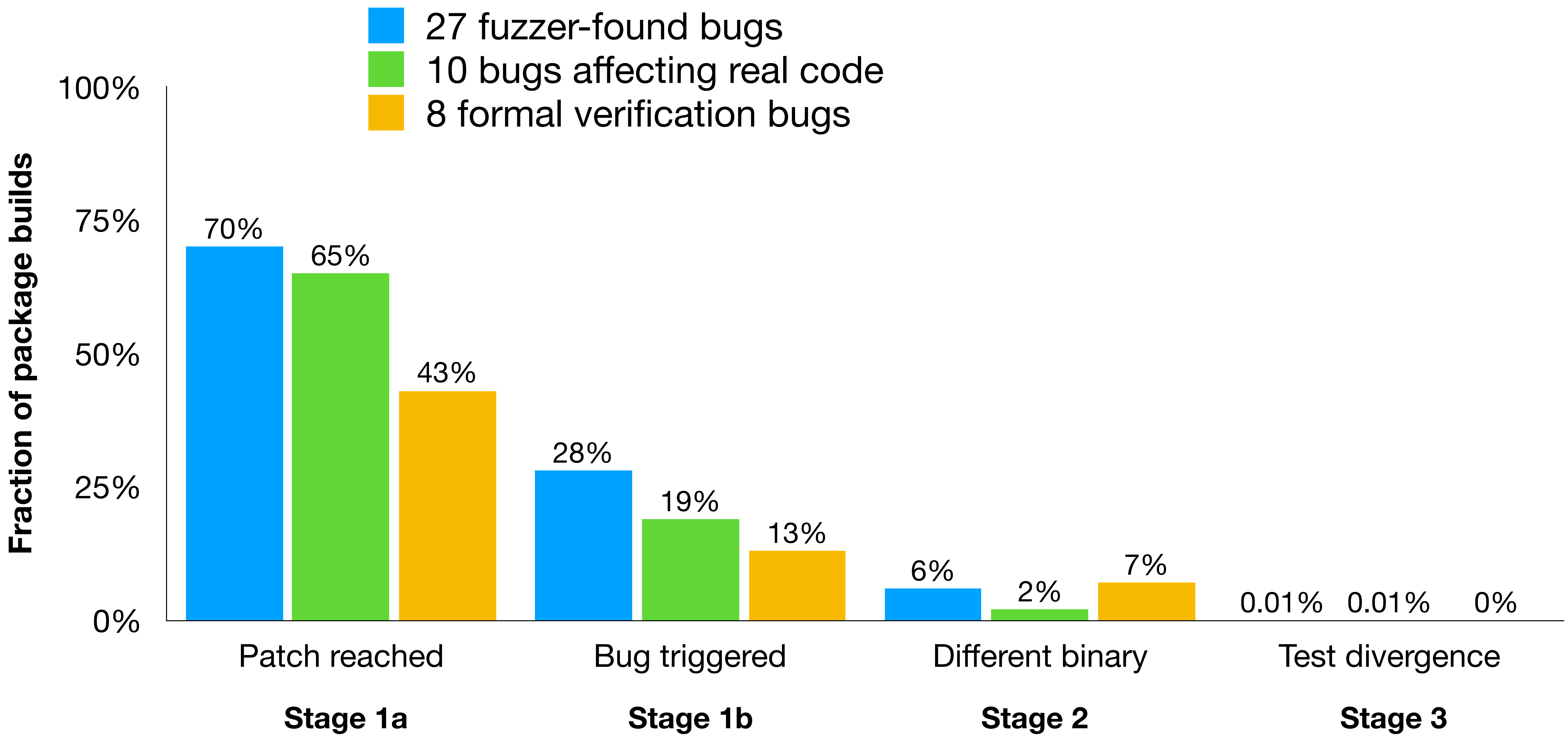
We **apply** our bug impact measurement **methodology over** a **sample** of:

- 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

APACHE™ samba > grep

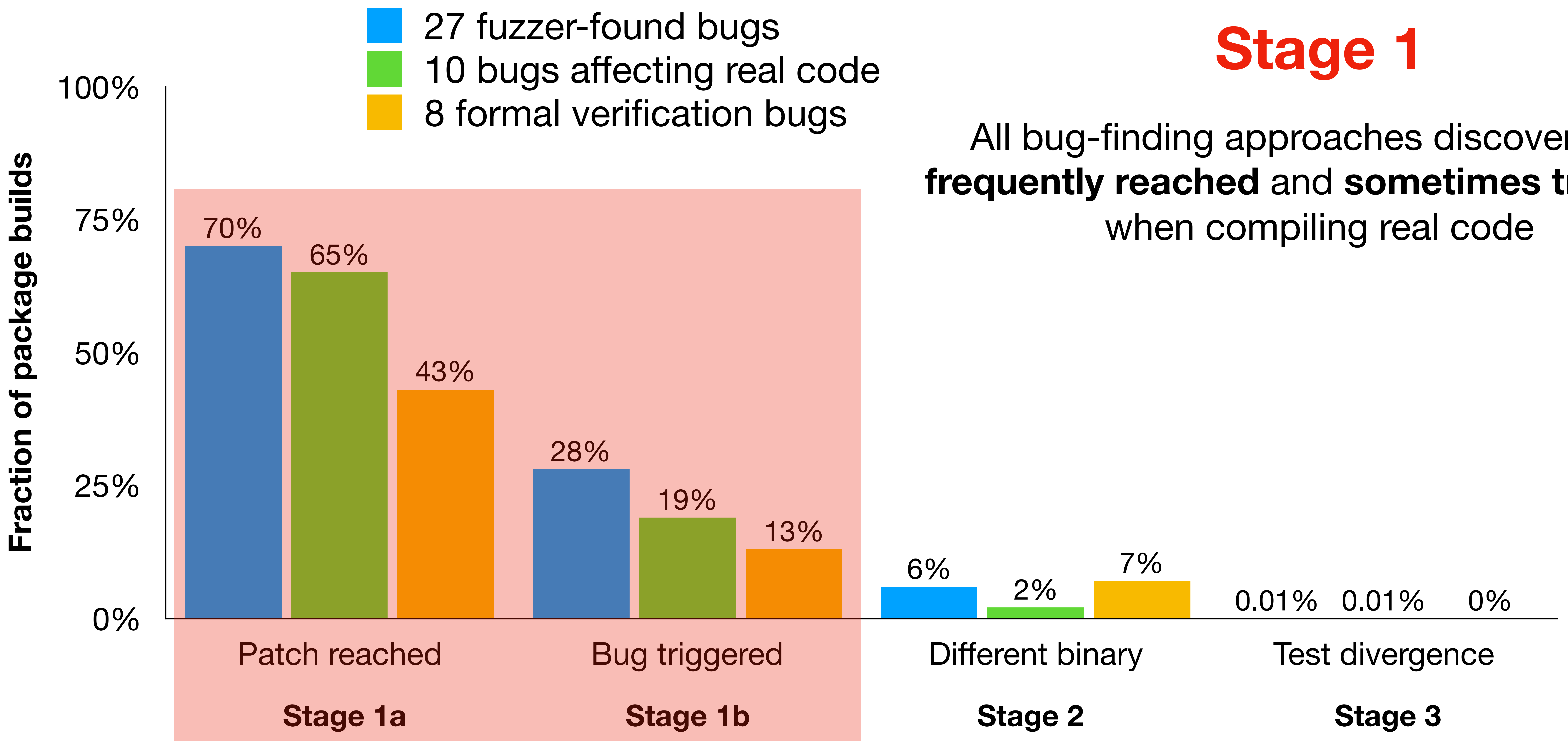
A lot of manual effort and 5 months of computation happen here

Results



27 fuzzer-found bugs
10 bugs affecting real code
8 formal verification bugs
Fraction of package builds
100%
75%
50%
25%
0%
70%
65%
43%
28%
19%
13%
6%
2%
7%
0.01%
0.01%
0%
Patch reached
Bug triggered
Different binary
Test divergence
Stage 1a
Stage 1b
Stage 2
Stage 3

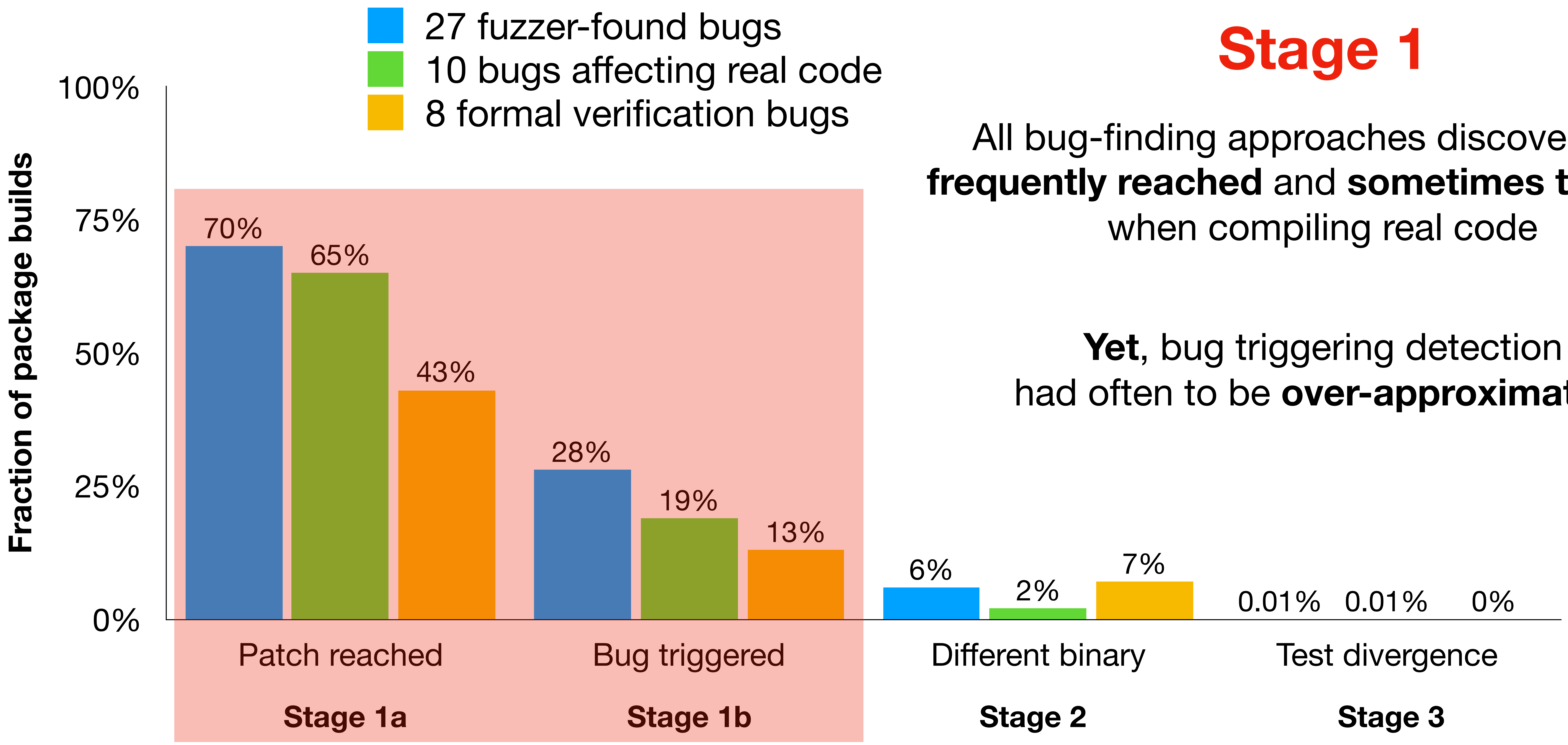
Results



Stage 1

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

Results



Stage 1

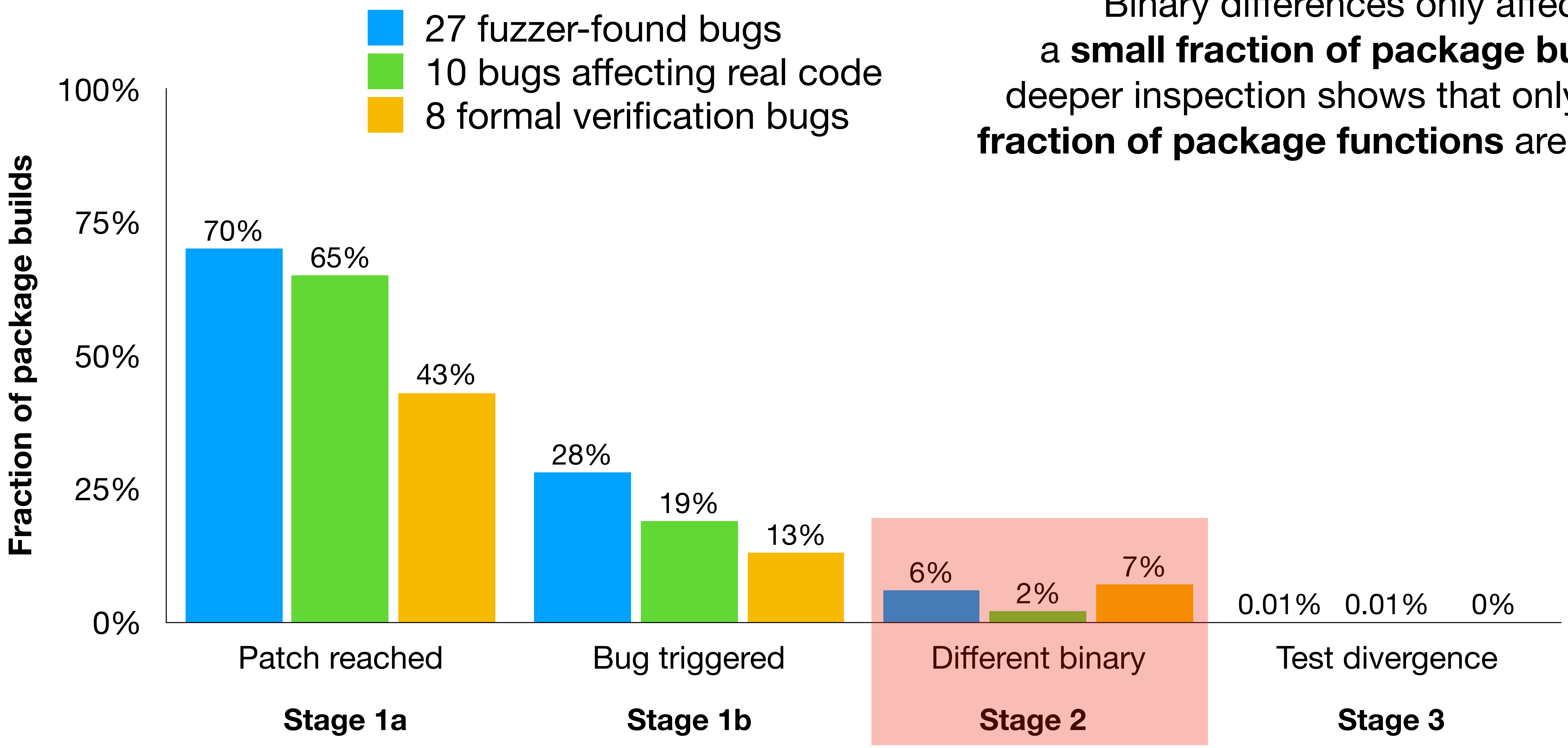
All bug-finding approaches discover bugs **frequently reached** and **sometimes triggered** when compiling real code

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

Results

Stage 2

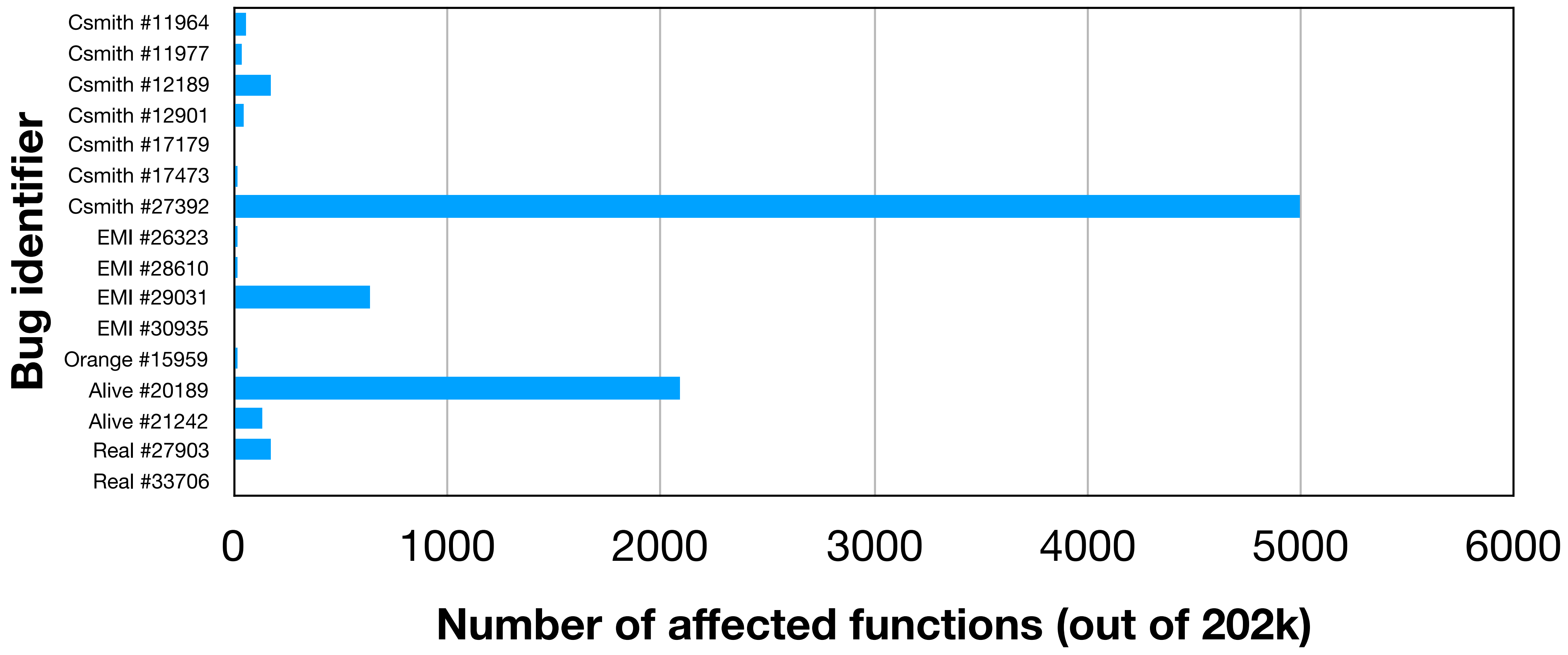
Binary differences only affect a **small fraction of package builds**, deeper inspection shows that only a **tiny fraction of package functions** are touched

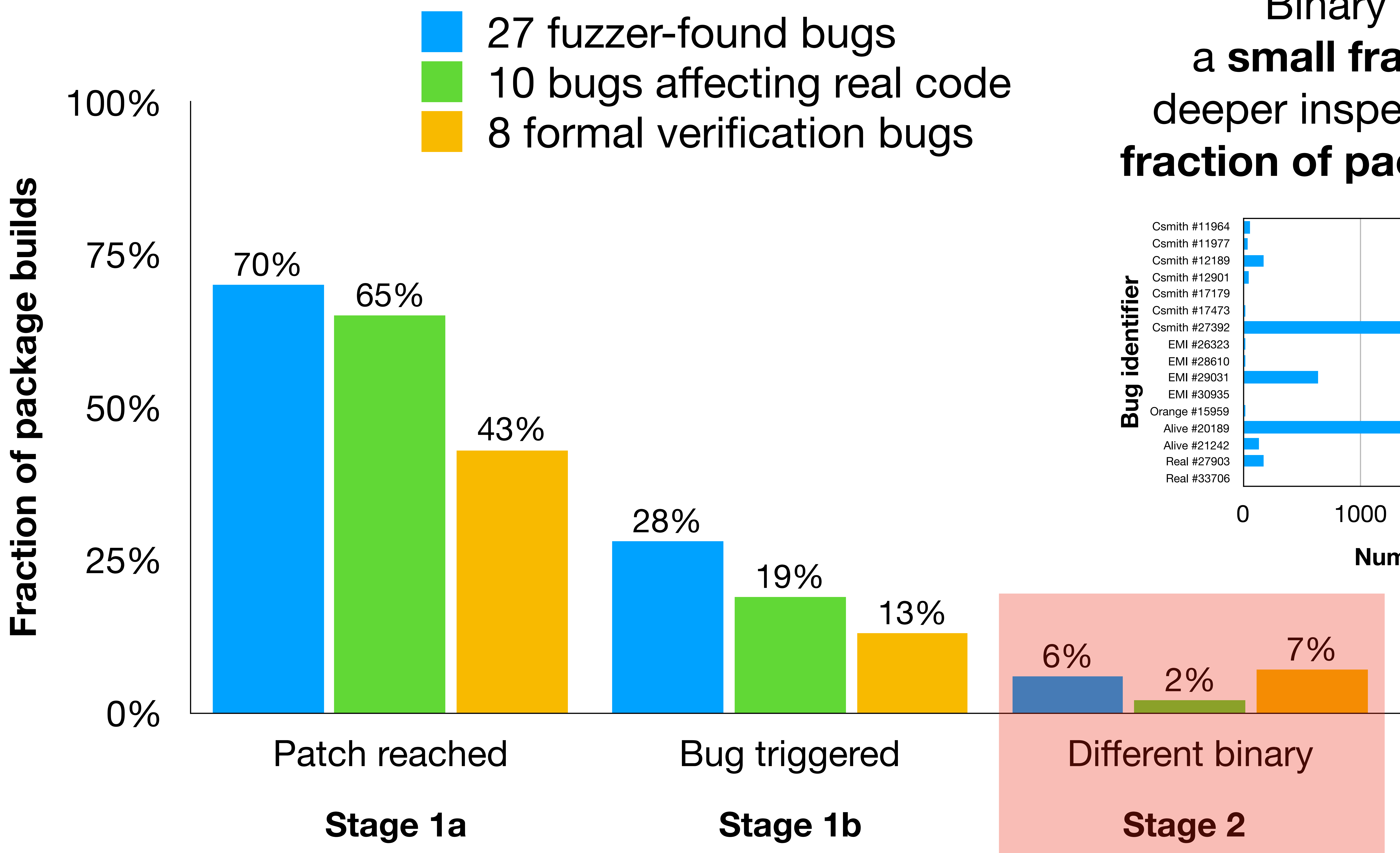


Results

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

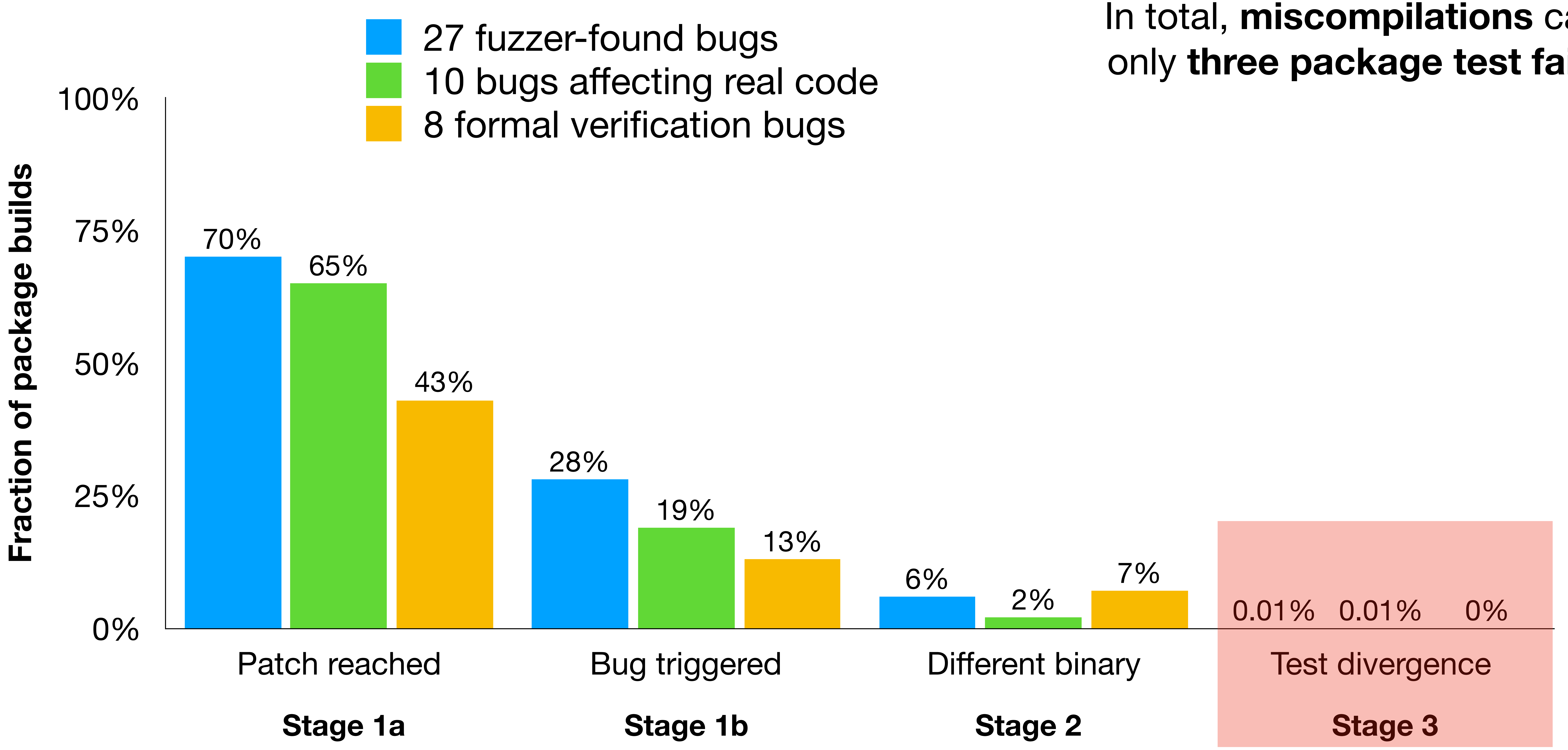




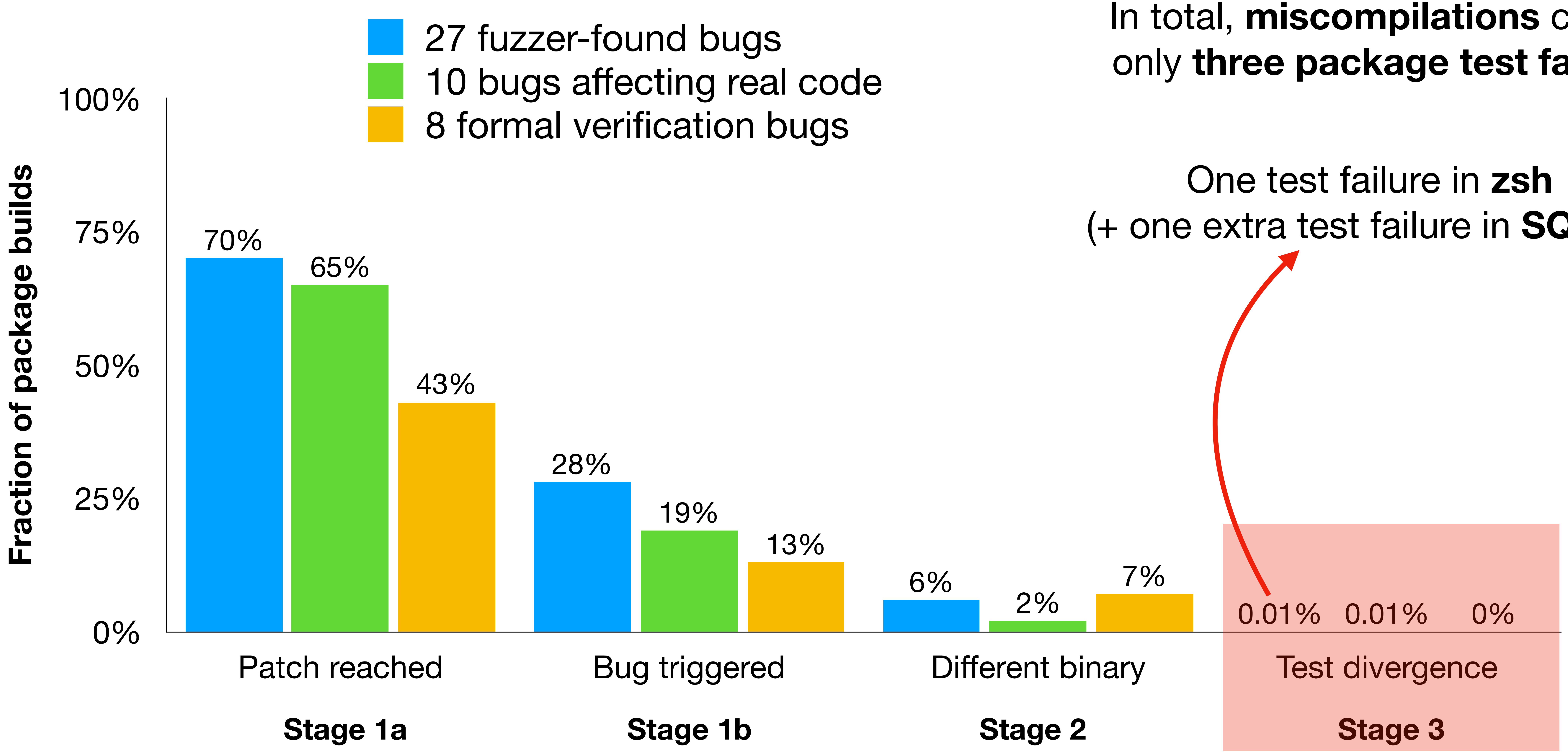
Results

Stage 3

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



Results



Stage 3

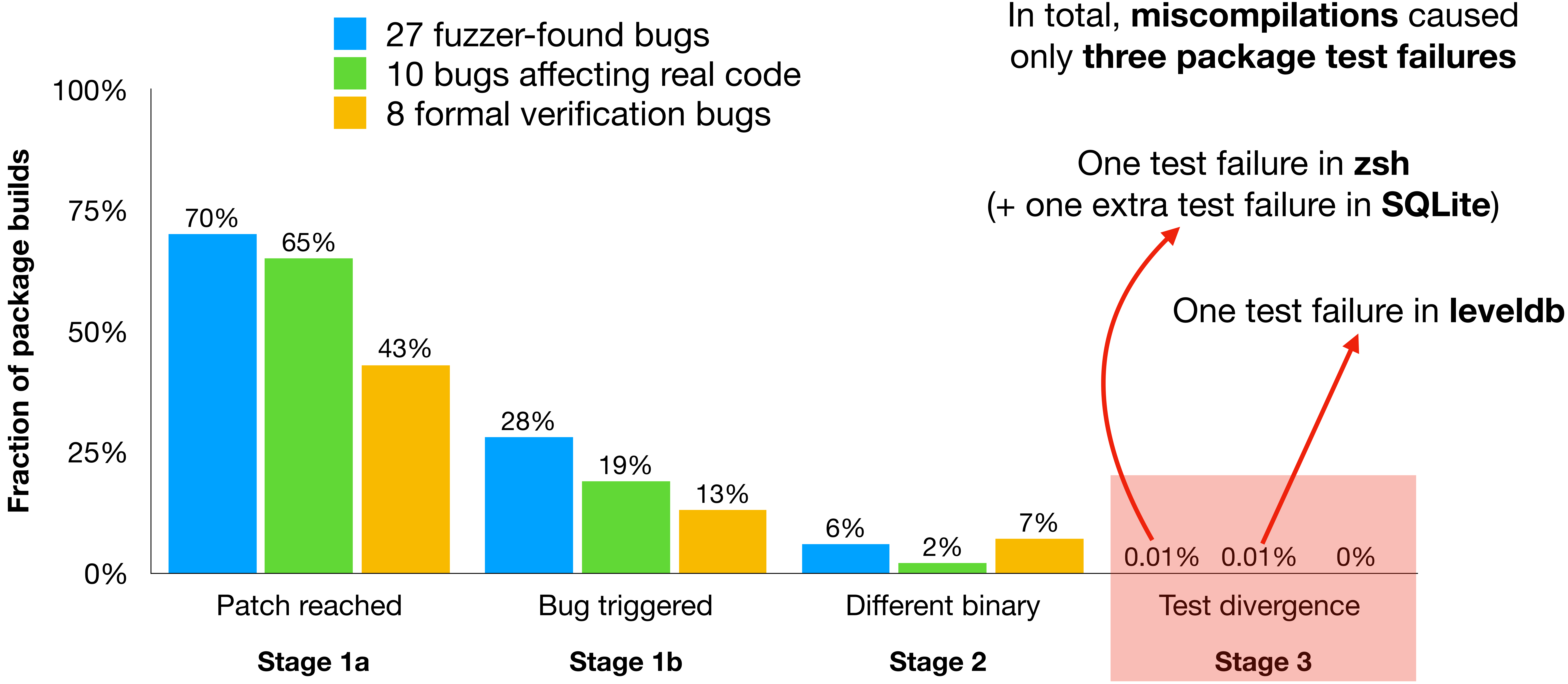
In total, **miscompilations** caused only **three package test failures**

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

Results

Stage 3

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



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)];
```

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 (`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)];
```

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:

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

79

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 (`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:

Wrong modulo binary code generated

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

79

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 (`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)];
```

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:

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

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:

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

232

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:

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

232

78

TEST RUN TIME

254 (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:

Garbage value

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

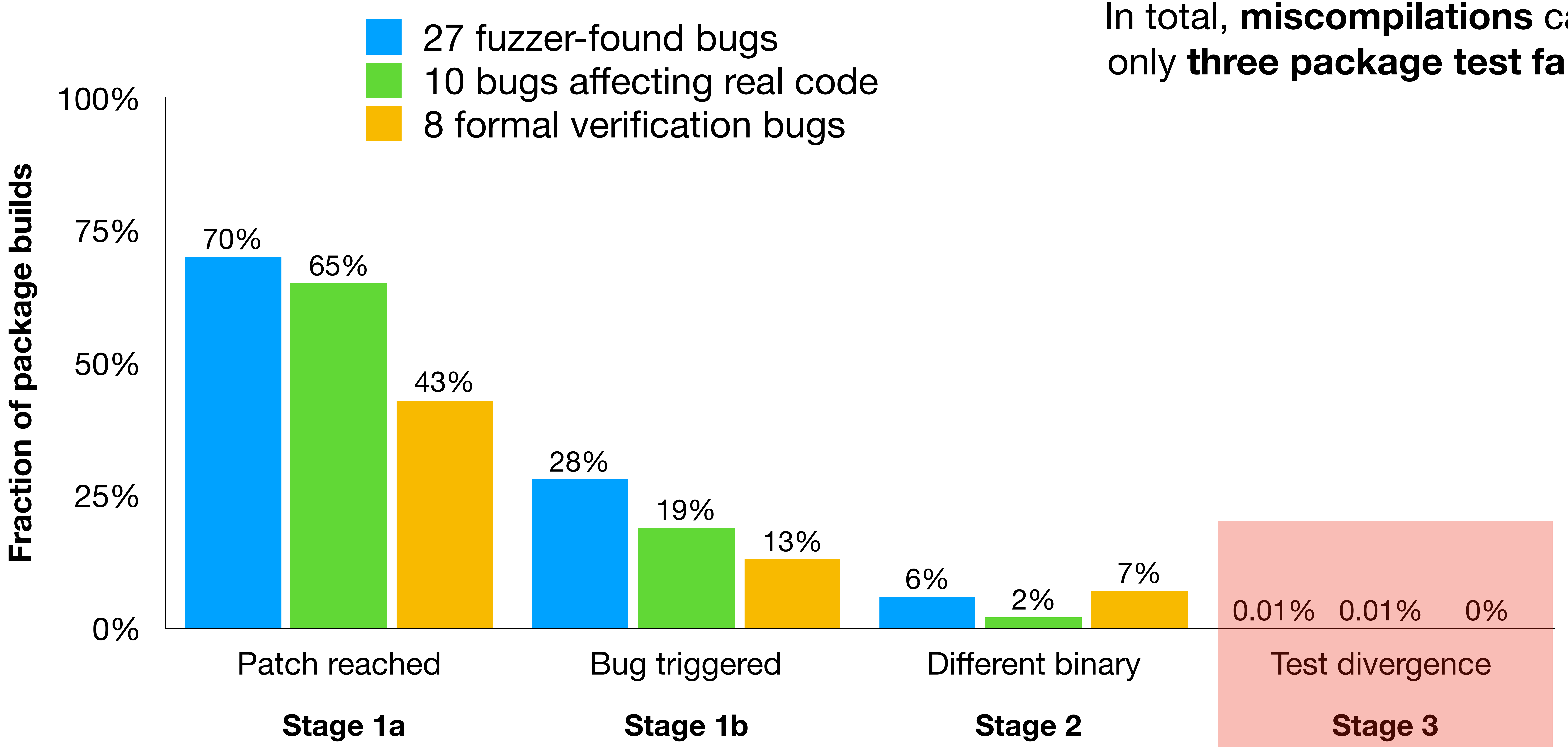
232

78

254 (out of range)

TEST RUN TIME

Results



Stage 3

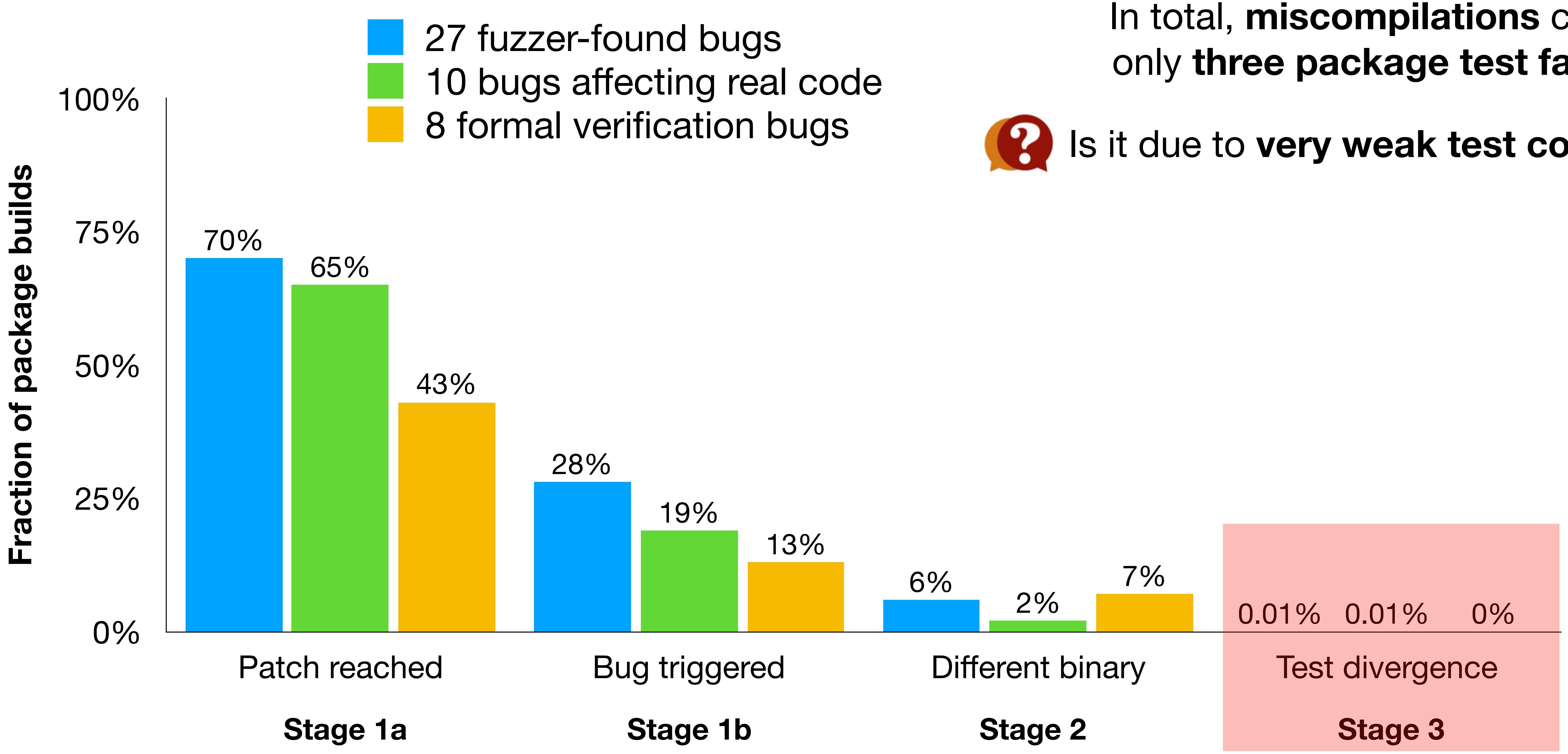
In total, **miscompilations** caused only **three package test failures**

Results

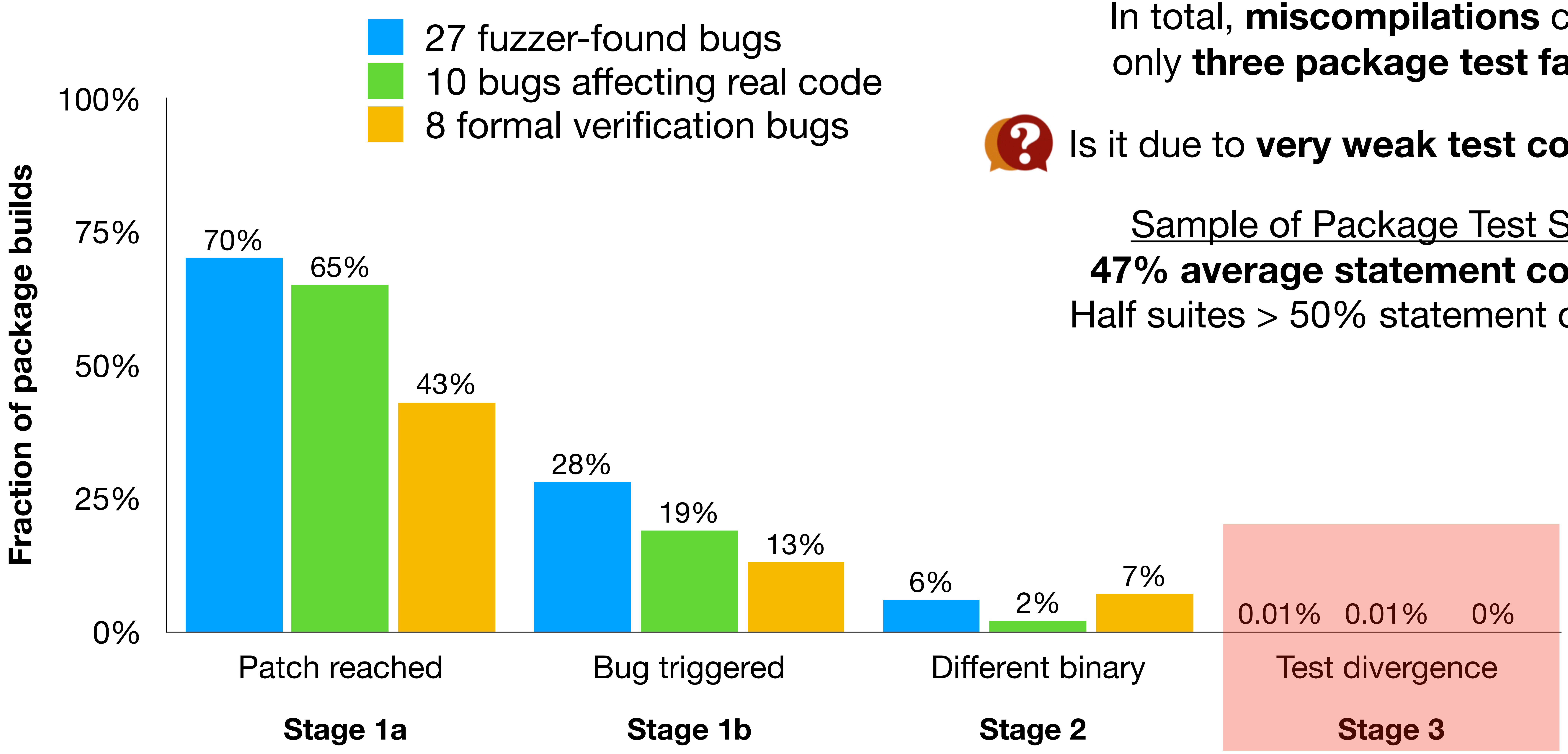
Stage 3

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

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



Results



Stage 3

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

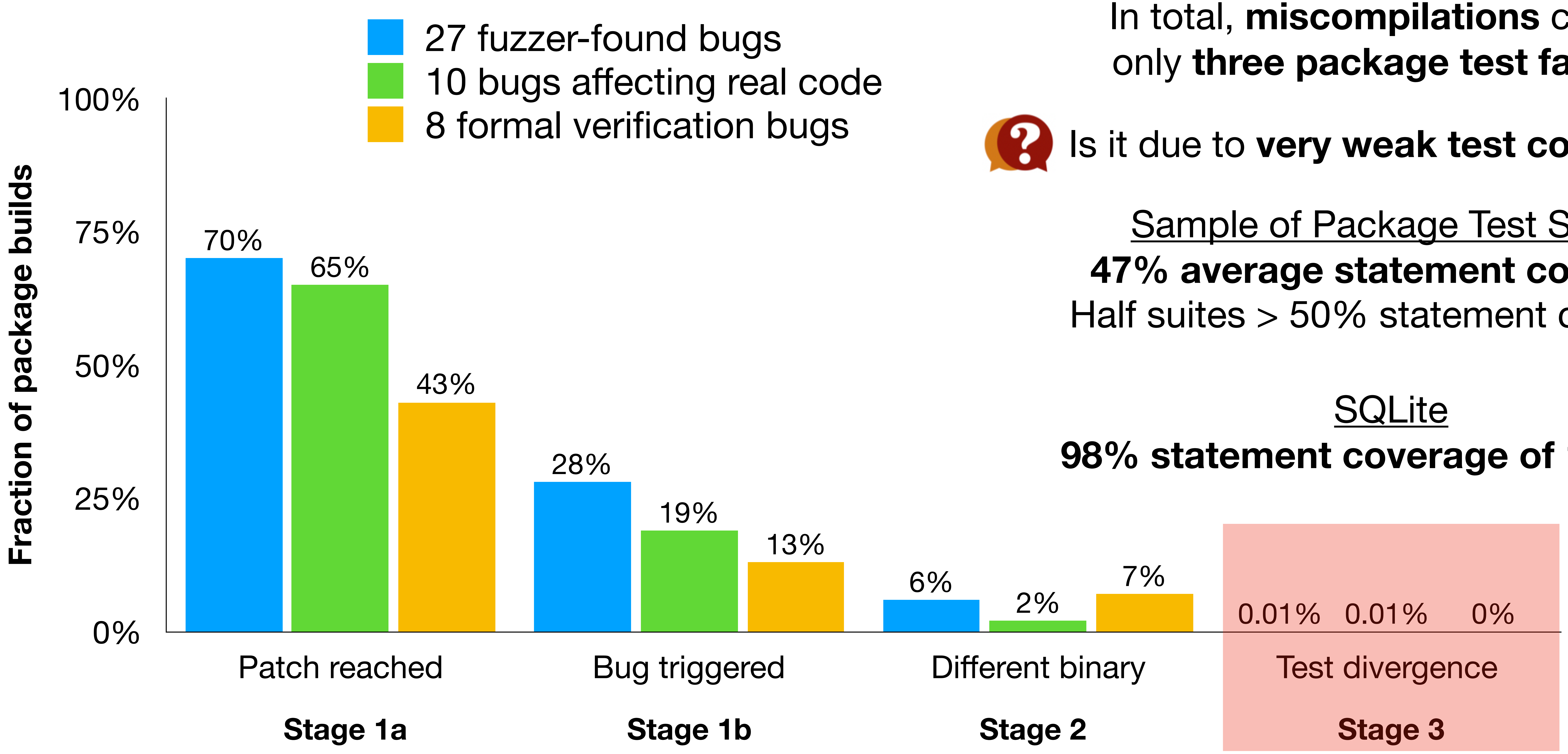
Is it due to **very weak test coverage**?

Sample of Package Test Suites

47% average statement coverage

Half suites > 50% statement coverage

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

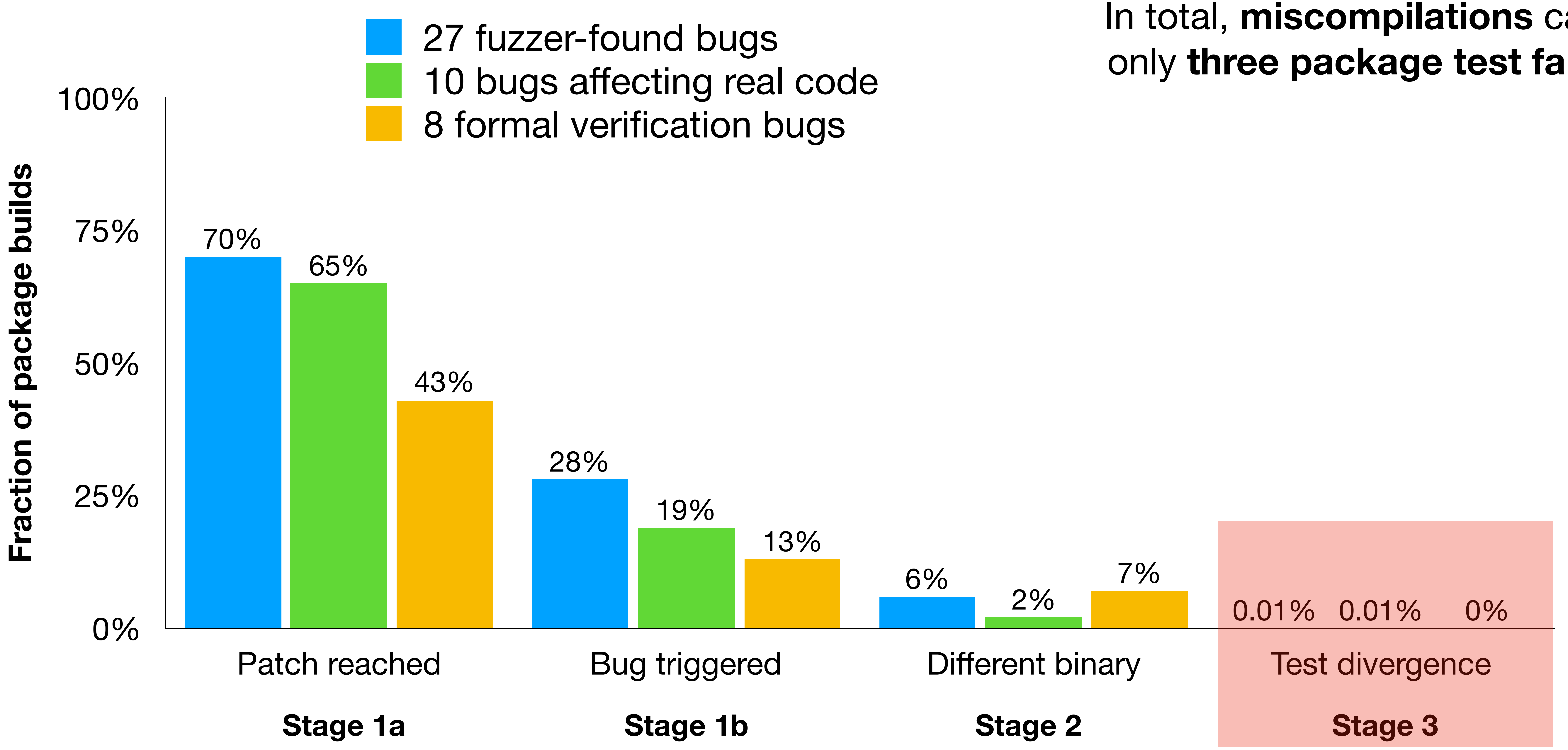
SQLite

98% statement coverage of 151kLoC

Results

Stage 3

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

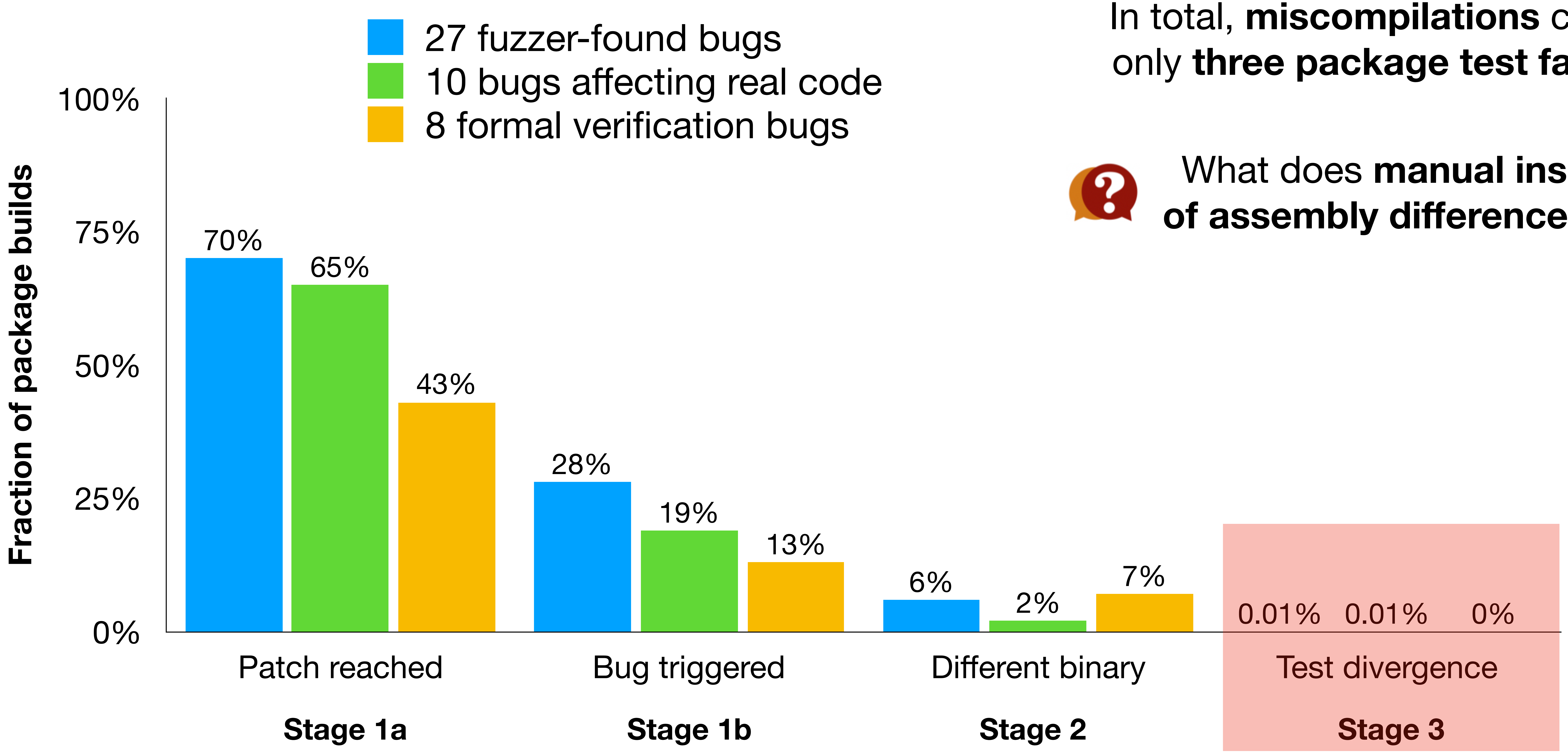


Results

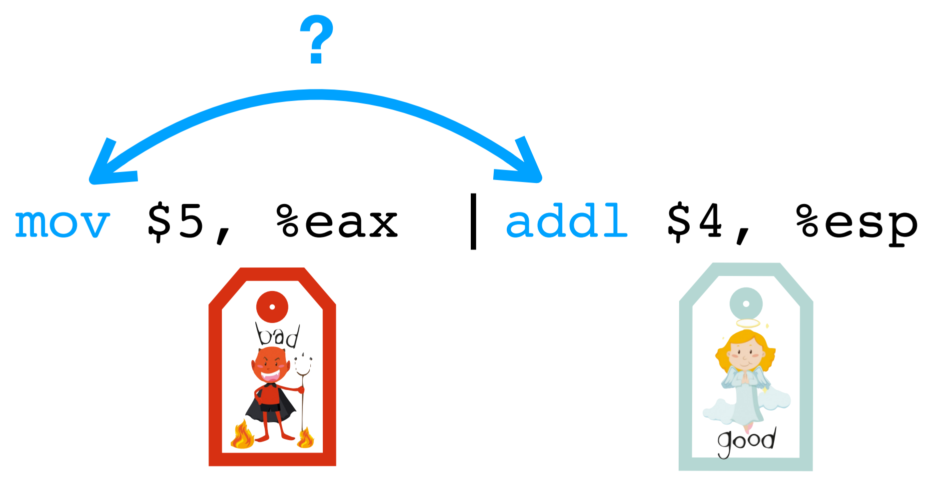
Stage 3

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

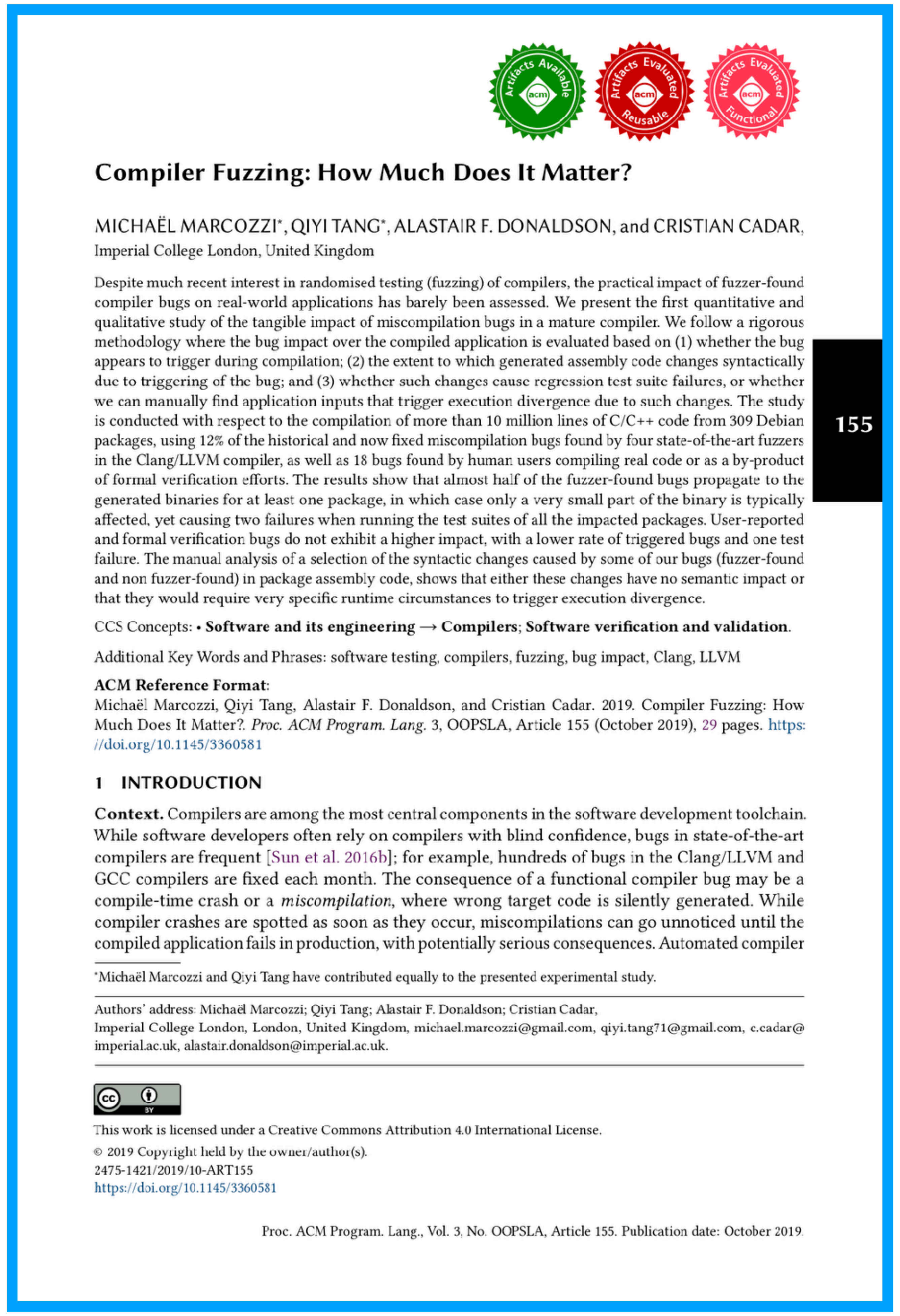
- Our main **research directions** for **even better evaluation** of **compiler bugs impact**:

  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!



Compiler Fuzzing: How Much Does It Matter?

MICHAËL MARCOZZI*, QIYI TANG*, ALASTAIR F. DONALDSON, and CRISTIAN CADAR, Imperial College London, United Kingdom

Despite much recent interest in randomised testing (fuzzing) of compilers, the practical impact of fuzzer-found compiler bugs on real-world applications has barely been assessed. We present the first quantitative and qualitative study of the tangible impact of miscompilation bugs in a mature compiler. We follow a rigorous methodology where the bug impact over the compiled application is evaluated based on (1) whether the bug appears to trigger during compilation; (2) the extent to which generated assembly code changes syntactically due to triggering of the bug; and (3) whether such changes cause regression test suite failures, or whether we can manually find application inputs that trigger execution divergence due to such changes. The study is conducted with respect to the compilation of more than 10 million lines of C/C++ code from 309 Debian packages, using 12% of the historical and now fixed miscompilation bugs found by four state-of-the-art fuzzers in the Clang/LLVM compiler, as well as 18 bugs found by human users compiling real code or as a by-product of formal verification efforts. The results show that almost half of the fuzzer-found bugs propagate to the generated binaries for at least one package, in which case only a very small part of the binary is typically affected, yet causing two failures when running the test suites of all the impacted packages. User-reported and formal verification bugs do not exhibit a higher impact, with a lower rate of triggered bugs and one test failure. The manual analysis of a selection of the syntactic changes caused by some of our bugs (fuzzer-found and non fuzzer-found) in package assembly code, shows that either these changes have no semantic impact or that they would require very specific runtime circumstances to trigger execution divergence.

CCS Concepts: • **Software and its engineering → Compilers; Software verification and validation.**

Additional Key Words and Phrases: software testing, compilers, fuzzing, bug impact, Clang, LLVM

ACM Reference Format:
Michaël Marcozzi, Qiyi Tang, Alastair F. Donaldson, and Cristian Cadar. 2019. Compiler Fuzzing: How Much Does It Matter?. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 155 (October 2019), 29 pages. https://doi.org/10.1145/3360581

155

1 INTRODUCTION

Context. Compilers are among the most central components in the software development toolchain. While software developers often rely on compilers with blind confidence, bugs in state-of-the-art compilers are frequent [Sun et al. 2016b]; for example, hundreds of bugs in the Clang/LLVM and GCC compilers are fixed each month. The consequence of a functional compiler bug may be a compile-time crash or a *miscompilation*, where wrong target code is silently generated. While compiler crashes are spotted as soon as they occur, miscompilations can go unnoticed until the compiled application fails in production, with potentially serious consequences. Automated compiler

*Michaël Marcozzi and Qiyi Tang have contributed equally to the presented experimental study.

Authors' address: Michaël Marcozzi; Qiyi Tang; Alastair F. Donaldson; Cristian Cadar, Imperial College London, London, United Kingdom, michael.marcozzi@gmail.com, qiyi.tang71@gmail.com, c.cadar@imperial.ac.uk, alastair.donaldson@imperial.ac.uk.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2019 Copyright held by the owner/author(s).
2475-1421/2019/10-ART155
https://doi.org/10.1145/3360581

Proc. ACM Program. Lang., Vol. 3, No. OOPSLA, Article 155. Publication date: October 2019.

> 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