arXiv:1902.09334v1 [cs.SE] 25 Feb 2019

# A Systematic Impact Study for Fuzzer-Found Compiler Bugs

Michaël Marcozzi
Imperial College London
m.marcozzi@imperial.ac.uk

Qiyi Tang
Imperial College London
qiyi.tang11@imperial.ac.uk

Alastair Donaldson
Google
Imperial College London
alastair.donaldson@imperial.ac.uk

Cristian Cadar
Imperial College London
c.cadar@imperial.ac.uk

## Abstract

Despite much recent interest in randomised testing (fuzzing) of compilers, the practical impact of fuzzer-found miscompilations on real-world applications has barely been assessed. We present the first quantitative study of the tangible impact of fuzzer-found compiler bugs. We follow a novel methodology where the impact of a miscompilation bug is evaluated based on (1) whether the bug appears to trigger during compilation; (2) whether the effects of triggering a bug propagate to the binary code that is generated; and (3) whether a binary-level propagation leads to observable differences in the application's test suite results. The study is conducted with respect to the compilation of more than 11 million lines of C/C++ code from 318 Debian packages, using 45 historical bugs in the Clang/LLVM compiler, either found using four distinct fuzzers, the Alive formal verification tool, or human users. The results show that almost half of the fuzzer-found bugs propagate to the generated binaries for some packages, but never cause application test suite failures. User-reported and Alive bugs have a lower impact, with less frequently triggered bugs and also no test failures. The major conclusions are that (1) either application test suites do not reflect real-world usage or the impact of compiler bugs on real-world code is limited, and (2) to the extent that compiler bugs matter, fuzzer-found compiler bugs are first class citizens, having at least as much impact as bugs from other sources.

***Keywords*** compilers, bugs, impact, fuzzing, Clang, LLVM

## 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 [50]; for example, hundreds of bugs in the Clang/LLVM and GCC compilers are fixed each month [17, 31]. 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 test generation has been a topic of interest for many years (see e.g. [5, 19, 23, 42, 46, 56]), and recent years have seen the development of several *compiler fuzzing* tools that employ randomised testing to search for bugs in (primarily C) compilers [25, 38, 57, 58].

**Problem.** Although compiler fuzzers have shown their ability to find hundreds of bugs, including many miscompilations, in widely-used compilers, the practical impact of these miscompilations on real applications has barely been evaluated.[1] It is reasonable to question the importance of the bugs found by compiler fuzzers for at least two reasons. First, by their very nature fuzzers detect miscompilations via *artificial* programs, obtained by random generation of code from scratch or random modification of existing code. It is thus unclear whether the code patterns that trigger miscompilations are likely to be exhibited by applications in the wild. Second, regular testing of an application should flag up many relevant miscompilation bugs as they arise, at least in the compilers used to compile the application under test. It is thus unclear whether fuzzers find bugs that are not already discovered by regular testing. The lack of a fine-grained and quantitative study of the real-world impact of compiler bugs means we have little but anecdotal evidence to support or rebut these points.

**Goal and challenge.** Our aim in this work is to investigate how often miscompilation bugs identified by state-of-the-art fuzzers are triggered when compiling a wide range of applications, and the extent to which this impacts the reliability of these applications. Given a fuzzer-found miscompilation bug, our first challenge is to gauge whether the bug is triggered by code in a chosen set of real-world applications.

[1] A recent empirical study of GCC and LLVM compiler bugs [50] provides numerous insights into bug characteristics, but does not address the extent to which bugs found by fuzzers impact on code found in the wild.

Whenever we find an affected application, our second challenge is to estimate whether the compiler bug can actually make the application misbehave in a runtime scenario, e.g. when running the application test suite.

**Approach.** We take advantage of the large number of fuzzer-found bugs in open-source compilers that have been publicly reported and fixed. Given a fixed bug, we analyse the fix and devise a change to the compiler code that emits a warning when the faulty code is reached and a local faulty behaviour triggered. The warnings issued when compiling a set of applications with the modified compiler provide information on whether the bug is triggered at all. In cases where warnings are issued, we compare the application binaries produced by the faulty and the fixed compiler versions. Cases where the binaries differ shed light on whether triggered bugs have an impact on final generated code (subject to appropriate care to ensure reproducible builds). For applications where differences in generated binaries are detected, we run the applications' standard test suites and look for discrepancies; these identify cases where the difference in generated code can be observed by a concrete application input that would have been detected by standard testing of the application.

**Experiments.** Following this three-stage approach, we present what is to our knowledge the *first ever study* of the real-world impact of fuzzer-found compiler bugs over a large set of diverse applications. In practice, we sample a set of 27 miscompilation bugs detected by four fuzzer families targeting C compilers: Csmith [10, 43, 57], EMI [25–27], Orange3/4 [37, 38] and yarpgen [58]. We estimate the impact of these bugs when compiling 318 Debian packages, such as *Apache*, *Grep* and *Samba*, totalling more than 11M lines of C/C++ code. Finally, we compare the impact of these fuzzer-found bugs with the impact of a set of 10 bugs reported directly by end users, and every of the 8 bugs found as a by-product of applying the Alive formal verification tool [32].

**Contributions.** Our main contributions are:

1. A three-stage methodology (detailed in §3) for assessing the impact of a miscompilation bug on a given application (a) during compilation (is the faulty compiler code reached and triggered?), (b) on generated code (did the fault lead to a difference in the resulting binary?) and (c) at application runtime (did the fault lead to application test failures?)
2. The first systematic quantitative study on the real-world impact of compiler bugs (presented in §4, §5 and §6), which applies our methodology (a) to evaluate the impact of 12% of the miscompilation bugs found by four state-of-the-art fuzzers for Clang/LLVM over 318 diverse Debian packages totalling 11M lines of C/C++, and (b) to compare the impact of these fuzzer-found bugs with 18 bugs found either by users compiling real code or by the Alive tool, over the same Debian packages.

**Summary of main findings.**

1. None of the bugs studied, whether fuzzer-found or not, caused any of the application test suites to fail. On one hand, this may suggest that fuzzers find bugs that cannot be found by regular regression testing. On the other hand, it suggests that either such bugs do not have high impact in practice, or test suites are typically not representative of real-world usage.
2. We did not observe significant differences between the severity levels assigned to bugs by compiler developers and the extent to which they are triggered. While a huge number of bugs remain unfixed in the LLVM database, this result suggests that acuter guidance mechanisms need to be developed on how to make prioritisation decisions.

Regarding the question in the paper's title, our results support the argument that bugs in generally high-quality, widely-used production compilers rarely impact deployed applications to the extent that core functionality (as encoded in the regression test suites) is affected, but that to the extent that compiler bugs matter, fuzzer-found compiler bugs appear to be first class citizens.

All our experimental data are available online [2].

## 2 Background

We make precise relevant details of compilation faults and miscompilation failures (§2.1), provide an overview of the compiler validation tools considered in the study (§2.2), provide relevant background on Clang/LLVM, including details of the bug tracker from which we have extracted compiler bug reports (§2.3), and describe the standard framework used to build and test Debian packages (§2.4), which is at the core of the experimental infrastructure deployed for the study.

### 2.1 Compilation faults and miscompilation failures

We describe the notion of a miscompilation bug using a simple but pragmatic model of how a compiler works. A modern compiler typically converts a source program into an intermediate representation (IR), runs a series of *passes* that transform this IR, and finally emits target code for a specific architecture. For source program $P$ and input $x$, let $P(x)$ denote the set of results that $P$ may produce, according to the semantics of the programming language in question. This set may have multiple values if $P$ can exhibit nondeterminism. Similarly, let $T(x)$ denote the set of results that target program $T$ may produce, according to the semantics of the target architecture. For ease of presentation we assume that source and target programs always terminate, so that $P(x)$ and $T(x)$ cannot be empty. A compilation $P \longrightarrow T$ is *correct*

with respect to input $x$ if $T(x) \subseteq P(x)$. That is, the target program respects the semantics of the source program: any result that the target program may produce is a result that the source program may also produce. Otherwise the compilation exhibits a *miscompilation failure* with respect to input $x$. We call a *compilation fault* any internal compiler operation which is incorrect during a compilation. A miscompilation failure is always caused by a compilation fault, but a compilation can exhibit a fault and no miscompilation failure if, for example, the fault makes the compiler crash or only impacts IR code that will be detected as dead and removed by a later pass. In such a case, the fault is said not to *propagate* to a miscompilation failure.

### 2.2 Fuzzers and compiler validation tools studied

Our study focuses on four compiler fuzzing tools (in two cases the "tool" is actually a collection of closely-related tools), chosen because they target compilation of C/C++ programs and have found bugs in recent versions of the Clang/LLVM compiler framework. We also consider some bugs found via application of the Alive tool for verification of LLVM peephole optimisations. We briefly summarise each tool, discussing related work on compiler validation more broadly in Section 7.

**Csmith.** The Csmith tool [57] randomly generates C programs that are guaranteed to be free from undefined and unspecified behaviour. These programs can then be used for *differential testing* [36] of multiple compilers that agree on implementation-defined behaviour: discrepancies between compilers indicate that miscompilations have occurred. By November 2013 Csmith had been used to find and report 481 bugs in compilers including LLVM, GCC, CompCert and suncc, out of which about 120 were miscompilations [30]. Many bugs subsequently discovered using Csmith are available from the bug trackers of the targeted compilers.

**EMI.** The Orion tool [25] introduced the idea of *equivalence modulo inputs* (EMI) testing. Given a deterministic C program $P$ (e.g. an existing program or a program generated by Csmith) together with an input $x$ that does not lead to undefined/unspecified behaviour, Orion profiles the program to find those statements that are *not* covered when the program is executed on input $x$. A set of programs are then generated from $P$ by randomly deleting such statements. While very different from $P$ in general, each such program should behave functionally identically to $P$ when executed on input $x$; discrepancies indicate miscompilations. Follow-on tools, Athena [26] and Hermes [49], extend the EMI idea using more advanced profiling and mutation techniques; we refer to the three tools collectively as EMI. To date, the project enabled discovering more than 1,600 bugs in LLVM and GCC, of which about 550 are miscompilations [41].

**Orange3/4.** The Orange3 [37] and Orange4 [38] tools can be used to fuzz C compilers via a subset of the C language, focussing primarily on testing compilation of arithmetic expressions. Orange3 generates a program randomly, keeping track during generation of the precise result that the program should compute. Orange4 instead is based on transforming test programs into equivalent forms of which the output always stays the same as for the original test programs before the transformations. These transformations include adding statements to the program or expanding constants into expressions. The tools have enabled reporting 60 bugs in LLVM and GCC, out of which about 25 are miscompilations [3].

**Yarpgen.** The Intel-developed Yet Another Random Program Generator (Yarpgen) tool [58] takes a Csmith-like approach to generating random programs. It accurately detects and avoids undefined behaviour by tracking variable types, alignments and value ranges during program generation. It also incorporates policies that guide random generation so that optimisations are more likely to be applied to the generated programs. It has been used to report more than 150 bugs in LLVM and GCC, out of which 38 are miscompilations [59].

**Alive.** The Alive project [32] provides a language to encode formal specifications of LLVM peephole optimisations, together with an SMT-based verifier to either prove them correct or provide counterexamples demonstrating where they are incorrect. Once an optimisation specification has been proven correct, Alive can generate LLVM-compatible C++ code that implements the optimisation. More than 300 optimisations were verified in this way, leading to the discovery of 8 miscompilation bugs in LLVM as a by-product.

### 2.3 Clang/LLVM framework

The Clang/LLVM framework [24] is one of the most popular compiler frameworks, used by a large number of research and commercial projects. Written in C++, it supports several source languages (like C, C++, Objective-C and Fortran) and many target architectures (x86, x86-64, ARM, PowerPC, etc.). The bugs we study in this work relate to compilation of C/C++ to x86 machine code. The process practically follows the compilation model outlined in §2.1: *Clang* front-end turns C/C++ source into LLVM IR, which is processed (via a series of passes) by the *LLVM Core*, which emits x86 binary.

All *Clang* and *LLVM Core* bugs studied in this paper were reported on the Clang/LLVM online bug tracker [52]. A typical bug report includes a fragment of code that demonstrates the miscompilation, together with the affected Clang/LLVM versions and configurations (target architectures, optimisation levels, etc.). The bug is given a unique ID and classified by severity, being either ranked as an enhancement request, a normal bug or a release blocker. A public discussion usually follows, involving the compiler developers who may end up writing a fix for the bug, if judged necessary. The fix is applied (with attached explanatory comments) directly

within the public Clang/LLVM SVN repository [44]. The revision number(s) where the fix was applied is typically provided to close the bug report.

### 2.4 Debian packages build and test framework

Debian is a well-known open-source operating system and software environment. It provides a popular repository for compatible packaged applications [39], together with a standard framework to facilitate compiling and testing these many packages from sources. This framework is composed, among others, of Simple Build [55], Reproducible Builds [54] and Autopkgtest [34], which we use in this study.

Simple Build provides a standard way to build any package from source in a customised and isolated build environment. The infrastructure provides simple primitives to set up this environment as a tailored Debian installation within a chroot jail [21], to gather packaged sources for any package and compile them within the environment, and to revert the environment to its initial state after building a package.

Reproducible Builds is an initiative to drive developers towards ensuring that identical binaries are always generated from a given source. This makes it possible to check that no vulnerabilities or backdoors have been introduced during package compilation, by cross-checking the binaries produced for a single source by multiple third parties. The initiative notably provides a list of the packages for which the build process is (or can easily be made) reproducible.

Autopkgtest is the standard interface for Debian developers to embed tests suites in their packages. As soon as the developers provide a test suite in the requested format, Autopkgtest enables running of these tests over the package in a Debian environment using a single command. This environment can be the local machine, a local installation within a chroot jail or emulator and a distant machine.

## 3 Methodology

In our study, we focus solely on miscompilation bugs reported in open source compilers, and for which an associated patch that fixes the bug has been made available. We refer to this patch as the *fixing patch* for the bug, the version of the compiler just before the patch was applied as the *buggy compiler*, and the version of the compiler with the patch applied as the *fixed compiler*.

Given a miscompilation bug and fixing patch, together with a set of applications to compile, we evaluate the practical impact of the bug on each application in three stages:

1. We check whether the code affected by the fixing patch is reached when the application is compiled with the fixed compiler. If so, we check whether the conditions necessary for the bug to trigger in the buggy compiler hold, indicating that a compilation fault occurred.
2. We check whether the buggy and fixed compilers generate different binaries for the application. Differences in generated binaries indicate that a compilation fault may have propagated to a miscompilation failure.
3. If stage 2 finds that the binaries differ, we run the application test suite twice, once against each binary. Differences in the test results provide strong evidence that a miscompilation failure has occurred.

Of course, the effectiveness of stage 3 depends entirely on the thoroughness of the application's test suite, and in particular whether the test suite is deemed thorough enough to act as a proxy for typical real-world usage.

We now discuss each of the three stages of our approach in detail, in the process describing the steps we followed to curate a set of bugs with associated compiler versions and fixing patches.

### 3.1 Stage one: fix reached and fault triggered

For each bug, the first stage of our approach relies on isolating a fixing patch and preparing appropriate compile-time checks for the conditions under which the compilation fault would occur. We accomplished this by careful manual review of the bug tracker report associated with each miscompilation bug. We limited our attention to bugs where it was clear from discussion between developers in the bug tracker that the fixing patch was incorporated in a single revision (or several contiguous revisions) of the compiler sources and in isolation from any other code modifications. We found and discarded a small number of bugs whose fixes were applied together with other code modifications and/or via a series of non-contiguous compiler revisions, except in one case (Clang/LLVM bug #27903, reported by an end-user) where we found it straightforward to determine an independent fixing patch for the bug from the two non-contiguous patches that were used in practice. The first patch, meant as temporary, deactivated the faulty feature triggering the miscompilation, while the second patch permanently fixed this feature and reactivated it. Our independent patch simply applies the permanent fix without either deactivating or reactivating the feature.

As a simpler example, the fixing patch for Clang/LLVM bug #26323 (found by one of the EMI fuzzers) is Clang/LLVM revision 258904 [45], which makes the following change:

```
- if (Not.isPowerOf2()) {
+ if (Not.isPowerOf2()
+     && C->getValue().isPowerOf2()
+     && Not != C->getValue()) {
       /* CODE TRANSFORMATION */  }
```

It is clear from the fixing patch and its explanatory comments on the SVN that the bug is fairly simple and localised, fitting a common bug pattern identified by the Csmith authors [57, §3.7] where the precondition associated with a code transformation is incorrect. As a consequence, the transformation can be mistakenly applied, resulting into a miscompilation. The fix simply strengthens the precondition.

Having identified a fixing patch and understood the conditions under which the compilation fault would trigger, we modify the fixed compiler to print warnings (1) when at least one of the basic blocks affected by the fixing patch is reached during compilation, and (2) when upon reaching the fixing patch, the conditions under which a compilation fault would have occurred had the patch not been applied are triggered. In our running example this involves detecting when `Not.isPowerOf2()` holds but `C->getValue().isPowerOf2()&& Not != C->getValue()` does not. The fixing patch augmented with warning generation is as follows:

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

We sanity-check the correctness of the so crafted *warning-laden compiler* by making sure that the warnings are actually fired when compiling the miscompilation sample provided as a part of the bug tracker report. It is of course possible that the "fixing" patch does not entirely fix the miscompilation, and/or introduces new bugs in the compiler; sometimes bug reports are reopened for just this reason (see [6] for an example). We are reasonably confident that the bugs used in our study do not fall into this category: their fixes, accepted by the open source community, have stood the test of time.

For some patches it was tractable to determine precise conditions under which a compilation fault would have occurred in the buggy compiler. However, in other cases it was difficult or impossible to determine such precise conditions based on available compile-time information only. In these cases we instead settled for *over-approximating conditions*, designed to certainly issue warnings when the fault is triggered, but possibly issuing false positives, i.e. warning that the fault is triggered when it is not. In such cases we made the over-approximating conditions as precise as possible, to the best of our abilities. As an example, Clang/LLVM bug #21242 (reported by the Alive tool) affects a code transformation at IR level: a multiplication operation between a variable and a power of two $x * 2^n$ is transformed into an equivalent shift left operation $x << n$. When $n = 31$, the transformation is faulty for 32 bits signed integers in case when $x = 1$, because the overflow semantics of the first operation is not correctly preserved. As the value of variable $x$ cannot typically be determined at compile-time, we issued a fault warning regardless of the value of $x$, which over-approximates the precise behaviour, where we should warn only when we are sure $x$ is allowed to be 1 at runtime.

### 3.2 Stage two: different binaries

The first stage of our approach provides us with useful information on certain (in the case of precise conditions) and potential (in the case of over-approximating conditions) compilation faults. The second stage goes further and performs a bitwise comparison of the application binaries produced by the buggy and fixed compilers, to understand whether a fault propagated to the final binary. This helps to resolve false alarms resulting from imprecise conditions, as well as cases where a fault is masked at a later stage of compilation.

As an example of the latter situation, suppose a program contains an expression of the form $e * v$, where $e$ is some sub-expression and $v$ is an variable, and suppose that a compilation fault causes $e$ to be transformed to non-equivalent expression $e'$. If it turns out that $v$ is guaranteed to be 0 at the relevant program point, the compilation fault does not propagate to a failure because the semantics of the application is not affected. If the compiler determines that $v$ is 0 and simplifies the overall expression to 0, there will be no evidence of the fault in the generated code, which would thus be detected at stage two.

Observe that for stage two to be meaningful, we need to make sure that the differences in the two generated binaries are only caused by the code differences in the two compilers (i.e. by the fixing patch). In practice, this may not always be the case, as the compilation process of some application may not be reproducible. Build-dependent data are indeed often inserted into the produced binaries, like some build timestamps or the name of the path used for the build. In such cases, we could mistakenly infer that the different binaries are caused by the compilation fault. We discuss in section 4.2 how we have practically ensured that the applications compiled in our study follow a reproducible build process.

Notice that if stage one of our approach determines that no fault was triggered during application compilation, the binaries produced by the buggy and fixed compilers should be identical. In such cases, we still perform stage two to sanity-check our approach: if stage one reports no triggered faults but stage two reports differences in generated binaries, something must be wrong either with the way the warning-laden compiler has been hand-crafted or with the reproducibility of the application build process. In practice, this careful approach led to us to detect that the binaries produced for some applications contained the revision number of the compiler used to build them. As the buggy and fixed compilers correspond to different revisions, the binaries that they produced for these applications were always different, even when no compilation fault occurred. We solved this problem by removing any mention of the revision number within the compilers that we used.

### 3.3 Stage three: different test results

Even if stage two discovers that the binaries produced by the buggy and fixed compilers differ, this does not guarantee that the possible compilation fault detected at stage one propagated to a miscompilation failure. Indeed, stage two only confirmed that the binaries produced by the buggy and fixed compilers are *syntactically* different, which does not prevent them from being *semantically* equivalent. In this case, the divergence observed in the binary compiled by the buggy compiler is not the witness of a miscompilation, as it cannot trigger any incorrect application behaviour.

The purpose of the third stage of our impact evaluation approach is to gain a better confidence in whether the compilation fault did or did not propagate to a miscompilation failure. To do so, we simply run the default test suite of the application once with the binary produced by the buggy compiler, and once with the binary produced by the fixed compiler. If the first binary fails on some tests while the second one does not, it means—modulo some possibly flaky tests [33, 35] or some undesirable non-determinism in the application—that the fault can make the considered application misbehave at runtime and did thus propagate to a failure. If the test results are the same for the two binaries, it is unknown whether this is because they are semantically equivalent or because the test suite is not comprehensive enough to expose the miscompilation failure.

## 4 Experimental setup

We now describe how we chose the applications (§4.2) and bugs (§4.1) considered in the study, and discuss the technical aspects of the chosen experimental process (§4.3) .

### 4.1 Sampling compiler bugs to investigate

Due to the steep learning curve associated with gaining expertise in a production compiler framework, and the intensive manual effort required to prepare warning-laden compilers for each bug we consider, we decided to restrict attention to bugs reported in a single compiler framework. Most publicly-reported compiler fuzzing efforts have been applied to Clang/ LLVM and GCC. Either would have been suitable; we chose to focus on Clang/LLVM as this enabled an interesting comparison between bugs found by fuzzers and bugs found as a by-product of formal verification (the Alive tool is not compatible with GCC).

A total of 1,033 Clang/LLVM bugs are listed within the scoreboards of the four C fuzzers and Alive at time of writing. Relevant properties of these bugs are summarised in Table 1.

Our study requires a fixing patch for each bug, and our aim is to study miscompilations. We thus discarded the 799 bugs that remain unfixed or are not miscompilation bugs.

We then removed any of the remaining bugs for which the affected Clang/LLVM versions are too old to be built from source and to compile packages within a Debian 9 installation (with reasonable effort). In practice, this means that we exclude all the bugs affecting Clang/LLVM versions older than 3.1 (which was released more than 6 years ago).

**Table 1.** Tool-reported Clang/LLVM bugs that we study

| Tool family | Number of Clang/LLVM bugs | | |
|---|---|---|---|
| | Reported | Fixed + Miscompilation | Final Sample |
| Csmith | 164 | 52 | 10 |
| EMI | 783 | 163 | 10 |
| Orange | 12 | 7 | 5 |
| yarpgen | 66 | 4 | 2 |
| Alive | 8 | 8 | 8 |
| **TOTAL** | **1033** | **234** | **35** |

Because the aim of our study is to assess the extent to which miscompilation bugs have high practical impact, we prune those bugs that only trigger when non-standard compilation settings are used, e.g. to target old or uncommon architectures or to enable optimisation levels higher than default for Debian packages (-O2).

Finally, we select randomly 10 bugs (if available) to study per tool, among those bugs for which we were able to isolate an independent fixing patch and write a corresponding warning-laden compiler. We end up with a final sample of 35 bugs to analyse, as shown in Table 1. We complete this sample by a set of 10 miscompilation bugs reported directly by the Clang/LLVM end-users. These bugs were selected by searching the Clang/LLVM bug tracker for a set of 20 suitable miscompilation bugs not reported by the fuzzers or Alive authors. Then, we picked the first 10 of these bugs for which we could isolate an independent fixing patch and write a corresponding warning-laden compiler.

### 4.2 Sampling applications to compile

As the set of applications to be compiled, we consider the source packages developed for the last stable release of Debian (version 9). More than 30,000 packages are available. Experimenting with all packages was infeasible in terms of the compute resources available to us. We now describe the process we followed to select a feasible number of packages.

To apply the methodology of §3 to C/C++ compiler bugs in an automated fashion, we required packages that: build in a reproducible fashion (a requirement of stage 2, see §3.2); come with test suites (a requirement of stage 3, see §3.3) that can be executed in a standard fashion (required for automation); and contain more than 1K lines of C/C++ code.

For build reproducibility, the Debian Reproducible Builds initiative lists the approximately 23,000 packages for which builds are believed to be deterministic. Regarding availability of test suites, 5,000 packages support the Debian Autopkgtest command as a unified interface for running tests. We thus restricted attention to the 4,000 packages with reproducible builds and Debian Autopkgtest-compatible test suites.

We then filtered out all packages whose lines of C/C++ code are less than 1K. To do so, we simply count the number of lines of C/C++ code of each surviving package using CLOC [12] and exclude the ones that have less than 1K lines of C/C++ code.

Of the remaining packages, we removed all packages whose build process fails when compiled using version 3.6 of Clang/LLVM. This is middle-ranged in the timeline of compiler versions used in our study. We expect thus version 3.6 to be among the most representative of the features available (or not) in all the other versions.

To make the running time of our analyses more acceptable (it could take more that one week per compiler bug considering all the remaining packages), we sampled half of these packages to end up with a set of 318 source packages, consisting of a total of 8,736,339 lines in C source files, 797,555 lines in C++ source files and 1,492,807 lines in C/C++ header files. In more details, 185 packages consist of between 1K to 10K lines of code, 111 packages between 10K to 100K, 21 packages between 100K to 1M and one package more than 1M. The sampling was performed first by identifying a set of about 50 popular representatives of a wide variety of application types, such as system utilities (*Grep*), web servers (*Apache*), scientific software (*Symmetrica*), network protocols (*Samba*) and printer drivers (*Epson-inkjet-printer-escpr*). Second, we selected randomly other packages to complete the sampling.

To sanity-check reproducibility of builds, we built each remaining package twice with the same compiler (an instance of version 3.6 of Clang/LLVM) and verified that bitwise identical binaries are indeed produced.

### 4.3 Experimental process

We now describe the technical aspects of the process we followed to measure the impact of the 45 sampled compiler bugs over the 318 sampled Debian packages, using the three-stage methodology described in §3. We make our experimental infrastructure available at the companion website of this paper [2]. The website also provides an image of the Debian 9 virtual machine used for running the experiments, preloaded with our complete infrastructure and the warning-laden, buggy and fixed compilers for our example bug #26323.

Experiments were performed over six recent Intel servers, except for the test suite runs, which were conducted within virtual machines set up in the cloud, using Amazon Web Services (AWS) [53]. Each bug was analysed independently in one of the 12 identical Debian 9 virtual machines installed locally on the servers, while the test suite runs necessary for such an analysis were performed over two Debian 9 AWS machines specially created for each bug.

As a preliminary step, we prepared the list of the 318 packages to analyse in the JSON format (*tasks.json*), providing the exact package names and versions that allow Simple Build to automatically download the corresponding source packages from the Debian website.

For each package, the analysis starts by running the *chroot* shell script, which instructs Simple Build to install a fresh local Debian 9 build environment in a chroot jail. The sources of the warning-laden, buggy and fixed compilers are then compiled and installed in this build environment, by running the *compiler-llvm* shell script provided by the experimental infrastructure. Finally, the *steps-llvm* shell script iterates over the packages defined in the *tasks.json* file and performs the three stages detailed in our methodology for each of them.

Stage 1 (§3.1) is performed by setting the warning-laden compiler as the default compiler in the build environment and asking Simple Build to build the package. The resulting build logs are then searched (using *grep*) for the warning messages. In some cases, Simple Build may fail because the package build process is not compatible with the Clang/LLVM versions affected by the bug. These cases are simply logged and Stages 2 and 3 are not carried out.

Stage 2 (§3.2) is performed by setting successively the buggy and fixed compiler as the default compiler in the build environment and asking Simple Build to build each time the package. The two resulting binaries are then compared bitwise using *diff*.

Stage 3 (§3.3) is performed by asking Autopkgtest to execute the package test suite over the two binaries produced at stage 2, if they were different. The two test runs are carried out within the two isolated AWS test environments. The two resulting test logs are then hand-checked and compared. When a difference is spotted, the test runs are repeated several times in fresh test environments to make sure that the results are reproducible and not polluted by flaky tests or non-determinism. For some packages, the testing infrastructure may not be reliable—we log the cases where the infrastructure crashes, making impossible to run the tests.

We estimate the total machine time spent to run the experiments to around 4 months.

## 5 Results

We now analyse the results obtained for every bug and package pair considered in our study. We discuss the results for the fuzzer-found bugs in detail, covering the three stages of our approach in §5.1—§5.3, and turn to a comparison with bugs found from other sources—user-reported bugs and Alive bugs—in §5.4. We also investigate whether there is a correlation between bug severity and impact (§5.5). We wrap up with a discussion of our overall experimental findings in §5.6 The full set of data for all experiments is available from our companion website [2]. For every investigated bug, the downloadable artifacts notably include the warning-laden

fixing patch and the build logs produced at Stage 1, the generated binaries (if different) obtained at Stage 2, and the testing logs gathered at Stage 3 (if performed).

### 5.1 Stage 1: compile-time analysis

Experimental results for the fuzzer-found bugs are presented in Table 2, with Table 3 providing a more condensed view with results aggregated per fuzzer.

**Package build failures.** Fewer than 1% of all package builds failed. All but one of the analysed compiler versions failed to build at least one package; the maximum number of packages that a specific compiler version failed to build was five. Across all compiler versions, build failures were associated with 16 particular packages out of 318 packages total. The package with the highest failure rate is the *Velvet* bioinformatics tool, for which 52% of the builds failed. Manual inspection of build failure logs shows front-end compilation errors, e.g. relating to duplicate declarations and missing header files.

**Reachability of fixing patch.** For each bug, at least one package caused the associated fixing patch to be reached during compilation; i.e. all the fuzzer-found bugs we studied were related to code that could be reached during compilation of standard packages. For 19/27 bugs, the proportion of packages for which the patch was reached is above 50% and it remains high for most of the others bugs. The highest proportion is attained with Yarpgen bug #34381, whose patch is reached for 97% of the packages. This bug affects the generation of x86 binary code for additions. The minimal proportion is attained with Orange bug #26407, whose patch is reached for fewer than 2% of the packages. This bug affects the remainder operation for unsigned numbers when it involves a power of two. In general, the proportion of packages where the patch is reached appears to be much lower for the bugs discovered by Orange than by the other tools. A likely explanation is that Orange focuses on bugs affecting potentially complex arithmetic, which does not appear so commonly in real-world code. For 9/318 packages, the fixing patch was never reached during compilation for any of the considered bugs. Manual inspection of these packages shows either that they only contain a very small amount of code and/or that they use the C/C++ compiler only for a limited part of their build process.

**Fault triggering.** We were able to come up with precise fault-triggering conditions for only 19% of the investigated bugs. The main difficulty in writing such conditions was that the fixing patch for many bugs makes it difficult to identify the precise situations where the buggy version of the code would fail. Indeed, for many patches, it is highly complex to determine how the local changes made by the patch precisely impact the global behaviour of the compiler.

Due to our best efforts to make the imprecise patches as precise as possible only 39% of the builds where a fixing patch is reached lead to the fault conditions triggering. In total, 22/27 compiler bugs and 27% of the package builds generated potential faults. This last number falls at 13% when restricted to the cases where the patch is precise and thus the fault is certain.

### 5.2 Stage 2: binary comparison

While Stage 1 returned an 27% possible fault rate, only 6% of the package builds actually led to different binaries being generated by the buggy and fixed compiler. This is mainly false alarms issued by our fault detectors in the case of imprecise conditions. However, manual inspection revealed that it can also be caused by the package build process being more complex than simple code compilation. For example, in the context of Csmith bug #27392, there are three packages, namely *libffi-platypus-perl*, *snowball* and *viennacl* where a fault is detected for sure at Stage 1, but identical binaries are produced at Stage 2. This is due to the fact the binary for which the fault was triggered is not included as a part of the final packaged binary returned by the build process.

For a few of the cases where the compiled binaries were different, we compiled the package sources again with the buggy and fixed compilers, but using a Clang/LLVM flag to make them generate the binaries in human-readable LLVM IR. The manual inspection and comparison of the two resulting LLVM IR files confirmed that the one generated by the buggy compiler exhibited the symptoms described in the corresponding compiler bug report.

Regarding the aggregated numbers obtained for the Csmith bugs (see Table 3), buggy compiler code is reached for 79% of the builds and the fault conditions are triggered 33% of the builds. The numbers similar to those obtained for the EMI bugs, 77% and 30% respectively. However, the possible fault rate at stage 2 decreases almost twice as fast for EMI (5%) as for Csmith (10%), resulting in 4 EMI bugs out of 10 causing binary differences against 8 out of 10 for Csmith. While this trend should be confirmed using a larger number of bugs, a possible explanation is that Csmith was the first tool used to fuzz Clang/LLVM, so it had the privilege to collect some "low-hanging fruit", i.e. those compiler bugs that trigger more often.

In total, 13/27 fuzzer-reported bugs (48%) lead to binary differences. Bugs Csmith #27392 and EMI #29031 account for 68% of the builds that lead to different binaries. Bug #27392 affects loop unrolling in the case where the loop is unrolled by a given factor at compile-time, but the number of necessary loop executions cannot be adjusted to account for unrolling until runtime. The bug occurs when the number of original loop executions modulo the unroll factor is found to be non-zero. To deal with this case, the compiler produces extra target code to find out at runtime that some iterations were left over and execute them. However, the optimisation is faulty because the generated additional target code does not deal correctly with possible overflows. Bug

**Table 2.** Three stage impact analysis for 27 Clang-LLVM bugs found by the 4 C fuzzers families over 237 Debian packages

| BUG | | PACKAGES | 1) BUGGY LLVM CODE | | | 2) COMPILED BINARIES | 3) TEST SUITE RUNS | |
|---|---|---|---|---|---|---|---|---|
| *id* | *severity* | *successful builds* | *reached* | *triggered* | *(precise)* | *different* | *possible* | *different* |
| **Csmith** (10) | | | | | | | | |
| 11964 | enhancement | 317 | 308 | 2 | (no) | 2 | 2 | 0 |
| 11977 | normal | 317 | 303 | 111 | (no) | 21 | 13 | 0 |
| 12189 | enhancement | 317 | 299 | 293 | (no) | 46 | 36 | 0 |
| 12885 | enhancement | 315 | 287 | 1 | (no) | 0 | - | - |
| 12899 | enhancement | 316 | 144 | 6 | (no) | 0 | - | - |
| 12901 | enhancement | 316 | 293 | 288 | (no) | 37 | 32 | 0 |
| 13326 | enhancement | 314 | 126 | 126 | (no) | 1 | 1 | 0 |
| 17179 | normal | 315 | 247 | 3 | (no) | 2 | 2 | 0 |
| 17473 | release blocker | 317 | 286 | 17 | (no) | 11 | 9 | 0 |
| 27392 | normal | 317 | 206 | 206 | (yes) | 203 | 181 | 0 |
| **TOTAL** | | **3161** | **2499** | **1053** | | **323** | **276** | **0** |
| | | **100%** | **79%** | **33%** | | **10%** | **9%** | **0%** |
| **EMI** (10) | | | | | | | | |
| 24516 | normal | 318 | 133 | 0 | (yes) | 0 | - | - |
| 25900 | normal | 316 | 222 | 4 | (no) | 0 | - | - |
| 26266 | normal | 317 | 303 | 196 | (no) | 0 | - | - |
| 26323 | normal | 314 | 281 | 32 | (no) | 12 | 11 | 0 |
| 26734 | normal | 317 | 176 | 5 | (no) | 0 | - | - |
| 27968 | normal | 317 | 123 | 0 | (yes) | 0 | - | - |
| 28610 | normal | 317 | 303 | 298 | (no) | 9 | 8 | 0 |
| 29031 | normal | 316 | 298 | 216 | (no) | 128 | 113 | 0 |
| 30841 | normal | 317 | 307 | 192 | (no) | 0 | - | - |
| 30935 | normal | 317 | 288 | 11 | (no) | 3 | 3 | 0 |
| **TOTAL** | | **3166** | **2434** | **954** | | **152** | **135** | **0** |
| | | **100%** | **77%** | **30%** | | **5%** | **4%** | **0%** |
| **Orange** (5) | | | | | | | | |
| 15940 | normal | 316 | 159 | 19 | (no) | 0 | - | - |
| 15959 | normal | 316 | 108 | 10 | (no) | 9 | 9 | - |
| 19636 | normal | 316 | 7 | 7 | (no) | 0 | - | - |
| 26407 | normal | 317 | 4 | 0 | (yes) | 0 | - | - |
| 28504 | normal | 315 | 17 | 0 | (no) | 0 | - | - |
| **TOTAL** | | **1580** | **295** | **36** | | **9** | **9** | **0** |
| | | **100%** | **19%** | **2%** | | **1%** | **1%** | **0%** |
| **yarpgen** (2) | | | | | | | | |
| 32830 | enhancement | 317 | 302 | 0 | (yes) | 0 | - | - |
| 34381 | enhancement | 317 | 309 | 259 | (no) | 0 | - | - |
| **TOTAL** | | **634** | **611** | **259** | | **0** | **-** | **-** |
| | | **100%** | **96%** | **41%** | | **0%** | **-** | **-** |

**Table 3.** Data of Table 2, aggregated by compiler fuzzer

| TOOL | BUGS | 1) BUGGY LLVM CODE | | | 2) COMPILED BINARIES | 3) TEST SUITE RUNS | |
|---|---|---|---|---|---|---|---|
| | | *reached* | *triggered* | *(precise)* | *different* | *possible* | *different* |
| Csmith | 10 | 10 | 10 | (1) | 8 | 8 | 0 |
| EMI | 10 | 10 | 8 | (0) | 4 | 4 | 0 |
| Orange | 5 | 5 | 3 | (0) | 1 | 1 | 0 |
| yarpgen | 2 | 2 | 1 | (0) | 0 | - | - |
| **TOTAL** | **27** | **27** | **22** | **(1)** | **13** | **13** | **0** |

#29031 affects an optimisation where load and store instructions are hoisted to a dominating code block. The optimisation is faulty due to the dominance relation being computed incorrectly in the presence of loops.

### 5.3 Stage 3: test suite execution

**Failed test suite runs.** About 13% of the test suite runs could not be carried out because the underlying testing infrastructure was not reliable and crashed. The most common reasons for such crashes were missing files or dependencies as well as failures to compile the testing infrastructure itself

**Differences in test results.** Across all bugs and packages, we did not observe any differences in the test results that were obtained; i.e. it would appear that the impact of these compiler bugs is not severe enough to cause failures in the current regression test suites of these packages.

A natural follow-up question is: how high is the coverage achieved by the package test suites? It would be unsurprising if test suites achieving very low coverage were unaffected by compiler bugs, but we might expect higher-coverage test suites to be more discriminating.

To investigate this we set out to use the *gcov* [18] tool to gather statement coverage information for the package test suites. Integrating gcov with our Simple Build + Autopkgtest infrastructure presented challenges, e.g. gcov notes files were not always generated during compilation, sometimes coverage data were not generated during test suite

runs, and sometimes both were generated but were deemed not to match by gcov. We were however able to gather reliable coverage data for 39 packages. Across these packages, we found their test suites to achieve a median of 47% and a mean of 46% statement coverage, with lowest and highest coverage rates of 2% and 95%, respectively. About half of the packages—18/39—had test suites achieving at least 50% statement coverage.

While statement coverage is a limited metric (e.g. high coverage via tests with weak oracles has limited significance), these coverage rates were somewhat higher than our team had predicted. Certainly the coverage seems high enough that we cannot attribute the absence of test suite failures to very poor test coverage.

### 5.4 Comparison with other bug sources

The impact data for the 8 bugs discovered by Alive and our sample of 10 user-reported bugs is reported in Tables 4 and 5.

**Alive bugs.** The average impact of the Alive bugs appears much more limited than the one of the Csmith and EMI bugs. The buggy compiler code is reached twice less often for the Alive bugs, and 6/8 Alive bugs never trigger compared to only 2/20 for Csmith and EMI. The bugs impact profile for Alive is actually closer to the one for Orange: Alive bugs trigger in the context of corner cases (typically overflows) of possibly complex arithmetic, which do not appear so often in real-world code. However, two Alive bugs led to 175 binary differences, which, despite their likely limited impact in practice (they require the smallest representable integer to be manipulated at runtime), confirm some of the practical interest of Alive.

**User-reported bugs.** Contrary to fuzzer-reported bugs, user-reported ones were discovered by spotting miscompilations in the code of a real application. Our results tend to show that this does not make user-reported bugs likelier to trigger when compiling other applications. On the contrary, different binaries are only spotted for 3/10 bugs and 2% of the package builds, far below the values of 11/20 and 5% reached by Csmith and EMI. This trend should still be confirmed using more than 10 user-reported bugs, but the absence of any impact gap in favour of end-users' bugs supports the claim that compiler fuzzers are not less relevant because they discover bugs via randomly generated or mutated code.

### 5.5 Correlation between bug severity and impact

Table 6 aggregates the impact data by the severity level that bugs were assigned to on the Clang/LLVM bug tracker. We expected a higher impact to be associated with bugs with higher severity, but no clear trend emerged in practice and some numbers are even counter-intuitive: the buggy compiler code is reached almost twice as often at the *enhancement* level than at the more severe *normal* level, while different binaries are produced eight times less often at the *release blocker* level than at the two lower levels. While the confidence in these results would be increased by bigger samples at the *enhancement* and *release blocker* levels, the bug severity appears to be a bad predictor of the practical bug impact.

### 5.6 Discussion

To sum up, our top-level findings include that the code associated with fuzzer-found bugs is frequently reached when compiling our set of real-world applications, that bugs do trigger and about half of the bugs propagate to binary-level differences for some packages; however, these differences never cause application test suite failures. The impact of the user-reported and Alive-related bugs is even lower: associated compiler code is not always reached, bugs are triggered less frequently, and also do not lead to test suite failures.

Our major take-aways are that (1) either application test suites do not reflect real-world usage or the impact of compiler bugs (whether fuzzer-found or not) on real-world code is limited, and (2) to the extent that compiler bugs matter, fuzzer-found compiler bugs appear to be first class citizens, having at least as much impact as bugs found via other sources, including user-reported bugs.

## 6 Threats to validity and limitations

**Threats to validity.** A first class of threats to the validity of our experimental results arise because the software artefacts that we used, including the shell scripts, warning-laden fixing patch, compilers, package development framework and system tools, could be defective. However, we have crosschecked our results in several ways. The data computed by the shell scripts were verified by hand for a small number of packages and bugs. A sample set of the warning-laden fixing patches were randomly picked and reviewed by a member of the team who had not been involved in producing them. Each warning-laden compiler was tested over the miscompilation samples provided on the bug tracker. Any arising contradiction between the results of stage one and two, where no potential fault would trigger but the produced binaries would be different, was investigated. A sample of the occurring suspicious behaviours like build failures or triggered bugs leading to similar binaries were investigated for a satisfiable explanation. The test logs were hand-checked and the test runs were repeated several times in clean environments for a dozen of randomly picked bug and package pairs, and also in all the situations where the binaries produced divergent test results or where the test beds crashed. All these sanity checks succeeded.

Another threat is that some of the fixing patches from the Clang/LLVM repository could be incorrect. However, this seems improbable given that all the bug reports from which the patches come from have been closed at least several months and typically several years ago, and they have never been reopened since.

**Table 4.** Three stage impact analysis for the 18 Clang-LLVM bugs found by Alive or end users over 318 Debian packages

| BUG | | PACKAGES | 1) BUGGY LLVM CODE | | | 2) COMPILED BINARIES | 3) TEST SUITE RUNS | |
|---|---|---|---|---|---|---|---|---|
| *id* | *severity* | *successful builds* | *reached* | *triggered* | *(precise)* | *different* | *possible* | *different* |
| **Alive** (8) | | | | | | | | |
| 20186 | normal | 318 | 34 | 0 | (yes) | 0 | - | - |
| 20189 | normal | 318 | 267 | 177 | (no) | 123 | 113 | 0 |
| 21242 | normal | 318 | 254 | 152 | (no) | 52 | 50 | 0 |
| 21243 | normal | 318 | 56 | 0 | (yes) | 0 | - | - |
| 21245 | normal | 318 | 275 | 0 | (yes) | 0 | - | - |
| 21255 | normal | 318 | 10 | 0 | (yes) | 0 | - | - |
| 21256 | normal | 318 | 168 | 0 | (yes) | 0 | - | - |
| 21274 | normal | 318 | 0 | 0 | (yes) | 0 | - | - |
| **TOTAL** | | **2544** | **1064** | **329** | | **175** | **163** | **0** |
| | | **100%** | **42%** | **13%** | | **7%** | **6%** | **0%** |
| **User-reported** (10) | | | | | | | | |
| 13547 | release blocker | 316 | 302 | 280 | (no) | 1 | 1 | 0 |
| 15674 | release blocker | 316 | 302 | 0 | (yes) | 0 | - | - |
| 17103 | release blocker | 315 | 307 | 0 | (no) | 0 | - | - |
| 24187 | normal | 318 | 0 | 0 | (yes) | 0 | - | - |
| 26711 | normal | 317 | 0 | 0 | (yes) | 0 | - | - |
| 27575 | normal | 317 | 134 | 45 | (no) | 0 | - | - |
| 27903 | normal | 317 | 287 | 232 | (no) | 51 | 45 | 0 |
| 31808 | normal | 317 | 230 | 0 | (yes) | 0 | - | - |
| 33706 | normal | 317 | 260 | 41 | (no) | 4 | 3 | 0 |
| 37119 | normal | 312 | 178 | 0 | (no) | 0 | - | - |
| **TOTAL** | | **3162** | **2000** | **598** | | **56** | **49** | **0** |
| | | **100%** | **63%** | **19%** | | **2%** | **2%** | **0%** |

**Table 5.** Data of Table 4, aggregated by bug source

| TOOL | BUGS | 1) BUGGY LLVM CODE | | | 2) COMPILED BINARIES | 3) TEST SUITE RUNS | |
|---|---|---|---|---|---|---|---|
| | | *reached* | *triggered* | *(precise)* | *different* | *possible* | *different* |
| Alive | 8 | 7 | 2 | (0) | 2 | 2 | 0 |
| User-reported | 10 | 8 | 4 | (0) | 3 | 3 | 0 |

**Table 6.** Results for our 45 LLVM bugs over our 318 Debian packages, aggregated by bug severity

| BUG AGGREGATE | | PACKAGES | 1) BUGGY LLVM CODE | | 2) COMPILED BINARIES | 3) TEST SUITE RUNS |
|---|---|---|---|---|---|---|
| *severity* | *number of bugs* | *successful builds* | *reached* | *triggered* | *different* | *different* |
| enhancement | 8 | 2529 | 2068 | 975 | 86 | 0 |
| | | 100% | 82% | 39% | 3% | 0% |
| normal | 33 | 10454 | 5638 | 1957 | 617 | 0 |
| | | 100% | 54% | 19% | 6% | 0% |
| release blocker | 4 | 1264 | 1197 | 297 | 12 | 0 |
| | | 100% | 95% | 23% | 1% | 0% |

Our results might also have been affected by the unreproducible build process of some Debian packages. However, this is very unlikely as all the used packages were selected among the list of reproducible ones provided by Debian. Moreover, each of the selected packages was built twice and checked for any unreproducible behaviour. Finally, unreproducible packages should have led to contradictions between the warning-laden compiler and the binary comparison, but none were detected.

Common to all studies relying on empirical data, this one may be of limited generalisability. To reduce this threat, we performed our experiments over 318 diverse applications from the well-known Debian repository, including some very popular ones like *Apache* and *Grep*, totalling more than 11 millions lines of code. We have also investigated 45 historical Clang/LLVM bugs. They constitute a sample of about 15% of all the fixed miscompilation bugs reported by the studied tools in the considered compiler. Moreover, our sampling strategy excluded many bugs that are likely to have lower practical impact due to their reliance on specific and uncommon compiler flags.

Recall from §4.2 that we excluded from our study those Debian packages whose builds were non-reproducible or whose test suites did not follow the standard Autopkgtest format. While our coverage analysis in §5.3 shows that the packages that we used had test suites achieving reasonable test coverage, our filtering process may have eliminated packages with even more thorough test suites if they were in a non-standard format, or if the package is not guaranteed to build in a reproducible fashion. For example, SQLite is renowned for having a very rigorous test suite [48], but it does not conform to the Autopkgtest format. Running our study on non-standard test suites would require significant per-package manual effort, but might be interesting to try for selected packages such as SQLite.

**Limitations.** The study could have been conducted using more applications and more compiler bugs, targeting other compilers, particularly GCC, and other source and target languages, such as OpenCL [29]. This would however have dramatically increased the already high manual effort, and required significantly more computing resources.

We have checked whether an application impacted by a compiler bug could fail only by running its Debian-standard test suites. A more thorough, but more complex examination would involve running larger numbers of tests, e.g. via test suites that do not follow the standard Debian model, or investigating ways to monitor failures of applications that have been deployed to end users [8].

## 7 Related Work

**Understanding compiler bugs.** A recent empirical study provides an in-depth analysis of bugs in the GCC and LLVM compilers [50], focusing on aspects such as the distribution of bugs across compiler components, the sizes of triggering test cases associated with bug reports, the lifetime of bug reports from filing to closing, and the developer-assigned priority levels for bugs and how these correlate to compiler components. The study is complementary to ours: beyond a discussion of bug priorities, it is not concerned with the extent to which compiler bugs affect real-world applications, and it does not focus on whether the bugs under analysis are miscompilations, nor whether the bugs were found in the wild or via automated tools such as fuzzers.

Another empirical study [11] compares the equivalence modulo inputs and differential testing approaches to compiler testing (see below). A "correcting commits" metric is proposed that helps to identify distinct compiler bugs from failing tests. Otherwise the focus of the study is on understanding the testing techniques themselves, rather than understanding the real-world impact of the bugs they find.

The paper associated with the Csmith tool [57] focuses to some degree on understanding compiler bugs, e.g. identifying the most buggy files (according to Csmith-found bugs) in versions of GCC and LLVM at the time. This analysis *does* distinguish between wrong code bugs and crash bugs, but is simply concerned with whether bugs exist, rather than (as in our study) whether they affect real-world applications. Two projects associated with Csmith, on automatically reducing test cases that trigger fuzzer-found bugs [43], and on ranking reduced test cases in a manner that aims to prioritise distinct bugs [10], are important for understanding the root causes of fuzzer-found bugs, but do not directly shed light on how likely such bugs are to be triggered by real applications.

Bauer et al. [4] discuss exploiting compiler bugs to create software backdoors, and show a proof-of-concept backdoor based on a simplified version of an LLVM miscompilation bug found by Csmith. The possibility of code written to maliciously exploit a known miscompilation bug presents a compelling argument that miscompilations matter even though they may not otherwise affect real-world code.

An informal online collection of anecdotes about compiler bugs found in the also makes for interesting reading [40].

**Automated compiler testing.** The idea of randomly generating or mutating programs to induce errors in production compilers and interpreters has a long history, with grammar- or mutation-based fuzzers having been designed to test implementations of languages such as COBOL [46], PL/I [19], FORTRAN [7], Ada and Pascal [56], and more recently C [25, 26, 37, 38, 49, 57, 58], JavaScript and PHP [20], Java bytecode [11], OpenCL [29], GLSL [15, 16] and C++ [50] (see also two surveys on the topic [5, 23]). Related approaches have been used to test other programming language processors, such as static analysers [13], refactoring engines [14], and symbolic executors [22]. Many of these approaches are either geared towards inducing crashes, for which the test oracle problem is easy. Those that can find miscompilation bugs do so either via *differential testing* [36], whereby multiple equivalent compilers, interpreters or analysers are compared on random programs, or via *metamorphic testing* [9, 47], whereby a single tool is compared across equivalent programs, or generating programs with known expected results.

Regarding the fuzzers of our study, Orange3 takes the approach of generating programs with known results [37]; Csmith [57] and Yarpgen [58] are intended to be applied for differential testing; while the *equivalence modulo inputs* family of tools [25, 26, 49] as well as Orange4 [38] represent a successful application of metamorphic testing (earlier explored with only limited success [51]).

An recent non-fuzzing compiler testing technique involves *skeletal program enumeration*: exhaustively enumerating all programs (up to $\alpha$-renaming) that have a particular control-flow skeleton [60]. This technique is geared towards finding compiler crashes rather than miscompilations, so the bugs that it finds are not relevant for a study such as ours.

**Compiler verification.** A full discussion of compiler verification is out of scope for this paper, but we mention CompCert [28] as the most notable example of a *formally verified* compiler. CompCert provides an incomparable level of reliability: intensive fuzzing via Csmith and EMI techniques have not discovered any bugs in verified parts of the code base [25, 57] (as should be expected for a formally verified piece of software). One might think that a verified compiler should make the question of whether compiler bugs matter irrelevant by eliminating bugs completely. However, CompCert still faces some major limitations, such as incomplete language support (including no support for C++) and a less mature set of optimizations compared with e.g. GCC

or LLVM. A compromise is to verify certain parts of a compiler, an approach taken by Alive [32], whose bugs we have included in this study.

## 8 Conclusion

Compiler fuzzing tools have proven capable of finding hundreds of errors in widely-used compilers such as GCC and LLVM, but very little attention has been paid to whether these bugs impact real-world applications and whether they find bugs that would not be detected during the regular regression testing process. In this first empirical study investigating these questions, we have shown that almost half of the fuzzer-found bugs in our sample propagate to the compiled binaries of real-world applications, but none of them affect the execution of their regression test suites. On the one hand, this may suggest that fuzzers find bugs that cannot be found by regular regression testing. On the other hand, it suggests that either such bugs do not have high impact in practice, or test suites are typically not representative of real-world usage. At the same time, a comparison with user-reported bugs and bugs found as a side-effect of verifying compiler optimizations shows that such bugs propagate to even fewer compiler binaries, suggesting that to the extent that compiler bugs matter, fuzzer-found compiler bugs appear to be first class citizens.

## References

[1] 2015. .
[2] 2018. Paper companion website. https://sites.google.com/view/michaelmarcozzi/compiler-bugs.
[3] Orange 3/4 website. 2018. https://ist.ksc.kwansei.ac.jp/~ishiura/pub/randomtest/index.html.
[4] Scott Bauer, Pascal Cuoq, and John Regehr. 2015. Deniable Backdoors using Compiler Bugs. *PoC GTFO* (2015), 7–9.
[5] Abdulazeez Boujarwah and Kassem Saleh. 1997. Compiler test case generation methods: a survey and assessment. 39 (1997), 617 – 625. Issue 9.
[6] Clang/LLVM bug report #21903. 2018. https://bugs.llvm.org/show_bug.cgi?id=21903.
[7] Colin Burgess and M. Saidi. 1996. The automatic generation of test cases for optimizing Fortran compilers. 38 (1996), 111 – 119. Issue 2.
[8] Cristian Cadar, Luís Pina, and John Regehr. 2015. Multi-Version Execution Defeats a Compiler-Bug-Based Backdoor. http://ccadar.blogspot.co.uk/2015/11/multi-version-execution-defeats.html.
[9] T.Y. Chen, S.C. Cheung, and S.M. Yiu. 1998. *Metamorphic testing: a new approach for generating next test cases.* Technical Report HKUST-CS98-01. Hong Kong University of Science and Technology.
[10] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide, and John Regehr. [n. d.]. Taming Compiler Fuzzers.
[11] Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. [n. d.]. Coverage-directed differential testing of JVM implementations.
[12] CLOC - count Lines of Code [n. d.]. CLOC - Count Lines of Code. http://cloc.sourceforge.net/.
[13] Pascal Cuoq, Benjamin Monate, Anne Pacalet, Virgile Prevosto, John Regehr, Boris Yakobowski, and Xuejun Yang. [n. d.]. Testing Static Analyzers with Randomly Generated Programs.
[14] Brett Daniel, Danny Dig, Kely Garcia, and Darko Marinov. [n. d.]. Automated Testing of Refactoring Engines.
[15] Alastair F. Donaldson, Hugues Evrard, Andrei Lascu, and Paul Thomson. 2017. Automated Testing of Graphics Shader Compilers. *Proceedings of the ACM Programming Languages (PACMPL)* 1, OOPSLA (2017), 93:1–93:29.
[16] Alastair F. Donaldson and Andrei Lascu. [n. d.]. Metamorphic testing for (graphics) compilers.
[17] GCC website [n. d.]. GCC website. https://gcc.gnu.org/.
[18] gcov – A Test Coverage Program [n. d.]. gcov – A Test Coverage Program. gcc.gnu.org/onlinedocs/gcc/Gcov.html.
[19] K.V. Hanford. 1970. Automatic generation of test cases. *IBM Systems Journal* 9 (1970), 242–257. Issue 4.
[20] Christian Holler, Kim Herzig, and Andreas Zeller. [n. d.]. Fuzzing with Code Fragments.
[21] Chroot jail manpage. 2018. http://man7.org/linux/man-pages/man2/chroot.2.html.
[22] Timotej Kapus and Cristian Cadar. [n. d.]. Automatic Testing of Symbolic Execution Engines via Program Generation and Differential Testing.
[23] Alexander Kossatchev and Mikhail Posypkin. 2005. Survey of Compiler Testing Methods. *Programming and Computing Software* 31 (Jan. 2005), 10–19. Issue 1.
[24] Chris Lattner and Vikram Adve. [n. d.]. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation.
[25] Vu Le, Mehrdad Afshari, and Zhendong Su. [n. d.]. Compiler Validation via Equivalence Modulo Inputs.
[26] Vu Le, Chengnian Sun, and Zhendong Su. [n. d.]. Finding Deep Compiler Bugs via Guided Stochastic Program Mutation.
[27] Vu Le, Chengnian Sun, and Zhendong Su. [n. d.]. Randomized Stress-testing of Link-time Optimizers.
[28] Xavier Leroy. 2009. Formal verification of a realistic compiler. 52, 7 (2009), 107–115.
[29] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F. Donaldson. 2015. Many-core compiler fuzzing, See [1].
[30] Csmith Bugs List. 2018. https://github.com/csmith-project/csmith/blob/master/BUGS_
[31] LLVM website [n. d.]. LLVM website. http://llvm.org/.
[32] Nuno Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. 2015. Provably Correct Peephole Optimizations with Alive, See [1].
[33] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. [n. d.]. An Empirical Analysis of Flaky Tests.
[34] Debian Autopkgtest manpage. 2018. https://manpages.debian.org/testing/autopkgtest.
[35] Paul Dan Marinescu, Petr Hosek, and Cristian Cadar. [n. d.]. Covrig: A Framework for the Analysis of Code, Test, and Coverage Evolution in Real Software.
[36] W. M. McKeeman. 1998. Differential testing for software. 10 (1998), 100–107. Issue 1.
[37] Eriko Nagai, Atsushi Hashimoto, and Nagisa Ishiura. 2014. Reinforcing random testing of arithmetic optimization of C compilers by scaling up size and number of expressions. *IPSJ Transactions on System LSI Design Methodology* 7 (2014), 91–100.
[38] Kazuhiro Nakamura and Nagisa Ishiura. 2016. Random testing of C compilers based on test program generation by equivalence transformation. In *2016 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)*. 676–679.
[39] Debian packages website. 2018. http://packages.debian.org.
[40] WikiWikiWeb post on compiler bugs. 2018. http://wiki.c2.com/?CompilerBug.
[41] EMI project website. 2018. http://web.cs.ucdavis.edu/~su/emi-project.
[42] Paul Purdom. 1972. A sentence generator for testing parsers. *BIT Numerical Mathematics* 12 (1972), 366–375. Issue 3.

[43] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. [n. d.]. Test-case reduction for C compiler bugs.
[44] Clang/LLVM SVN repository explorer. 2018. http://llvm.org/viewvc.
[45] Clang/LLVM revision #258904 commit details. 2018. http://llvm.org/viewvc/llvm-project?view=revision&revision=258904.

[46] Richard L. Sauder. [n. d.]. A General Test Data Generator for COBOL.
[47] Sergio Segura, Gordon Fraser, Ana Sanchez, and Antonio Ruiz-Cortés. 2016. A Survey on Metamorphic Testing. (2016).
[48] SQLite. 2018. How SQLite is Tested. https://www.sqlite.org/testing.html
[49] Chengnian Sun, Vu Le, and Zhendong Su. [n. d.]. Finding compiler bugs via live code mutation.
[50] Chengnian Sun, Vu Le, Qirun Zhang, and Zhendong Su. [n. d.]. Toward Understanding Compiler Bugs in GCC and LLVM.
[51] Qiuming Tao, Wei Wu, Chen Zhao, and Wuwei Shen. [n. d.]. An Automatic Testing Approach for Compiler Based on Metamorphic Testing Technique.
[52] Clang/LLVM Bug Tracker. 2018. https://bugs.llvm.org.
[53] Amazon Web Services website. 2018. https://aws.amazon.com.
[54] Debian Reproducible Builds website. 2018. https://wiki.debian.org/ReproducibleBuilds.
[55] Debian Simple Build website. 2018. https://wiki.debian.org/sbuild.
[56] B.A. Wichmann. 1998. Some Remarks about Random Testing. http://www.npl.co.uk/upload/pdf/random_testing.pdf.
[57] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. [n. d.]. Finding and Understanding Bugs in C Compilers.
[58] Yarpgen. 2018. https://github.com/intel/yarpgen.
[59] yarpgen website. 2018. https://github.com/intel/yarpgen/blob/master/bugs.rst.

[60] Qirun Zhang, Chengnian Sun, and Zhendong Su. [n. d.]. Skeletal program enumeration for rigorous compiler testing.