Toward Understanding Compiler Bugs in GCC and LLVM

Chengnian Sun   Vu Le   Qirun Zhang   Zhendong Su
Department of Computer Science, University of California, Davis, USA
{cnsun, vmle, qrzhang, su}@ucdavis.edu

ABSTRACT

Compilers are critical, widely-used complex software. Bugs in them have significant impact, and can cause serious damage when they *silently miscompile* a safety-critical application. An in-depth understanding of compiler bugs can help detect and fix them. To this end, we conduct the first empirical study on the characteristics of the bugs in two mainstream compilers, GCC and LLVM. Our study is significant in scale — it exhaustively examines about 50K bugs and 30K bug fix revisions over more than a decade's span.

This paper details our systematic study. Summary findings include: (1) In both compilers, C++ is the most buggy component, accounting for around 20% of the total bugs and twice as many as the second most buggy component; (2) the bug revealing test cases are typically small, with 80% having fewer than 45 lines of code; (3) most of the bug fixes touch a single source file with small modifications (43 lines for GCC and 38 for LLVM on average); (4) the average lifetime of GCC bugs is 200 days, and 111 days for LLVM; and (5) high priority tends to be assigned to optimizer bugs, most notably 30% of the bugs in GCC's inter-procedural analysis component are labeled P1 (the highest priority).

This study deepens our understanding of compiler bugs. For application developers, it shows that even mature production compilers still have many bugs, which may affect development. For researchers and compiler developers, it sheds light on interesting characteristics of compiler bugs, and highlights challenges and opportunities to more effectively test and debug compilers.

CCS Concepts

•General and reference → Empirical studies; •Software and its engineering → Software testing and debugging;

Keywords

empirical studies, compiler bugs, compiler testing

ISSTA '16, July 18-22, 2016, SaarbrÃijcken, Germany
© 2016 ACM. ISBN 978-1-4503-4390-9/16/07...$15.00
DOI: http://dx.doi.org/10.1145/2931037.2931074

1. INTRODUCTION

Compilers are an important category of system software. Extensive research and development efforts have been devoted to increasing compilers' performance and reliability. However, it may still surprise application developers that, similar to application software, production compilers also contain bugs, and in fact quite many. Furthermore, compiler bugs impact application code, and even lead to severe damage, especially when a buggy compiler is used to compile safety-critical applications.

Different from most of the application bugs, compiler bugs are difficult to recognize as they usually manifest indirectly as application failures. For example, a compiler bug makes a program optimized and transformed into a wrong executable, and this bug can only manifest as the executable misbehaves. Even worse, in most cases the application developer will first assume the misbehavior is caused by some bug introduced by herself/himself, and it may take a long time for her/him to realize that the compiler is the culprit.

In order to better understand compiler bugs, we conduct the first empirical study on their characteristics. Although there have been a number of empirical studies on software bugs [6, 7, 17, 25, 27, 31, 36, 37], none focuses on compiler bugs. For example, Lu *et al.* [17] study the characteristics of concurrency bugs, Sahoo *et al.* [25] investigate the bugs in server software, and Chou *et al.* [7] research the errors in operating system kernels. In contrast, this paper examines the bugs of two mainstream production compilers, GCC and LLVM, in total 39,890 bugs of GCC and 12,842 bugs of LLVM. In order to explore the properties of compiler bug fixes, we also examine 22,947 GCC revisions and 8,452 LLVM revisions, which are fixes to most of the bugs. In particular, we attempt to investigate compiler bugs along four central aspects:

(1) Location of Bugs. We compute the distribution of bugs in compiler components and the source files. The data shows that in both GCC and LLVM, the component C++ is always the most buggy one, with a defect rate twice as much as the second most buggy component. In addition, we find that most of the source files only contain one bug (*i.e.*, 60% for GCC and 53% for LLVM), and most of the top ten buggy source files belong to the front end of C++.

(2) Test Cases, Localization and Fixes of Bugs. We investigate the size of the test cases that trigger compiler bugs, and find that on average GCC test cases have 32 lines of code and those of LLVM have 29 lines. This observation can guide random testing of compilers. For example, instead of generating large simple test programs, compiler testing may be more effective by generating small, yet complex test programs. Moreover, it confirms that most of the bugs can be effectively reduced by techniques such as Delta [38] or C-Reduce [24]. The fixes of compiler bugs are not big either, which on average only touch 43 lines of code for GCC and 38 for LLVM. 92% of them involve fewer than 100 lines of code changes, and 58% for GCC and 54% for LLVM involve only one function. Our findings reveal that most of the bug fixes are local. Even for optimizers, each bug is usually from a single optimization pass.

(3) Duration of Bugs. We compute information on the duration of bugs (*i.e.* the time between when a bug was filed and when it was closed), and show that the average duration of GCC bugs is 200 days and that of LLVM bugs is 111 days. We also show that the GCC community confirms bugs faster than LLVM but spends more time fixing them.

(4) Priorities of Bugs. The field `priority` of a bug report is determined by developers for prioritizing bugs, expressing the order in which bugs should be fixed. We investigate the distribution of bugs over priorities. Of the GCC bugs, 68% have the default P3 priority, and only 6.02% are labeled as P1. We then study how priorities are correlated with components. The inter-procedural analysis component, `ipa`, of GCC is the most "impactful" as 30% of its bugs are labeled as P1. We also study how priorities affect the lifetime of bugs. On average, P1 bugs are fixed the fastest, whereas P5 bugs are fixed the slowest. However, for the rest, the fix time is not strictly correlated with their priorities.

As a proof-of-concept demonstration of the practical potentials of our findings, we design a simple yet effective program mutation algorithm `tkfuzz` to find compiler crashing bugs, by leveraging the second observation that bug-revealing test programs are usually small. Applying `tkfuzz` on the 36,966 test programs in the GCC test suite (with an average of 32 lines of code each), we have found 18 crashing bugs in GCC and LLVM, of which 12 have already been fixed or confirmed.

The results presented in this paper provide further insights into understanding compiler bugs and better guidance toward effectively testing and debugging compilers. To ensure reproducibility and to benefit the community, we have made all our data and code publicly available at http://chengniansun.bitbucket.org/projects/compiler-bug-study/.

Paper Organization Section 2 describes the bugs used in this study and potential threats to validity. Section 3 introduces general properties of these bugs, while Section 4 studies the distribution of compiler bugs in components and source files respectively. In Section 5, we study bug-revealing test cases and bug fixes. Section 6 then investigates the lifetime of bugs, and Section 7 studies the priorities of bugs and their relation to other properties such as components and duration. Section 8 presents our preliminary application of the findings in this paper for compiler testing. We discuss, in Section 9, how to utilize the findings in this paper. Finally, Section 10 surveys related work, and Section 11 concludes.

2. METHODOLOGY

This section briefly introduces the compilers used in our study, and describes how bugs are collected. We also discuss limitations and threats to the validity of this study.

2.1 Source of Bugs

In this paper, we study the bugs in two mainstream compiler systems GCC and LLVM.

GCC GCC is a compiler system produced by the GNU Project supporting various languages (*e.g.* C, C++, and Fortran) and various target architectures (*e.g.* PowerPC, x86, and MIPS). It has been under active development since the late 1980s. The latest version is 5.3, which was released on December 4, 2015.

LLVM LLVM is another popular compiler infrastructure, providing a collection of modular and reusable compiler and toolchain technologies for arbitrary programming languages. Similar to GCC, LLVM also supports multiple languages and multiple target architectures. The project started in 2000 and has drawn much attention from both industry and academia. The latest version is 3.7.1, released on January 5, 2016.

Table 1: The information of the bugs used in this study.

Compiler	Start	End	Bugs	Revisions
GCC	Aug-1999	Oct-2015	39,890	22,947
LLVM	Oct-2003	Oct-2015	12,842	8,452

Collection of Bugs Our study focuses on *fixed* bugs. We say a bug is fixed if its `resolution` field is set to `fixed` and the `status` field is set to `resolved`, `verified` or `closed` in the bug repositories of GCC and LLVM.

Identifying Bug Fix Revisions We then identify the revisions that correspond to these fixed bugs. We collect the entire revision log from the code repositories, and for each revision, we scan its commit message to check whether this revision is a fix to a bug.

GCC and LLVM developers usually add a marker in the commit message following one of the two patterns below:

- "`PR` ⟨bug-id⟩"
- "`PR` ⟨component⟩/⟨bug-id⟩"

where the prefix "`PR`" stands for "Problem Report" and ⟨bug-id⟩ is the id of the corresponding bug. We use these patterns to link a revision to the bug that it fixes.

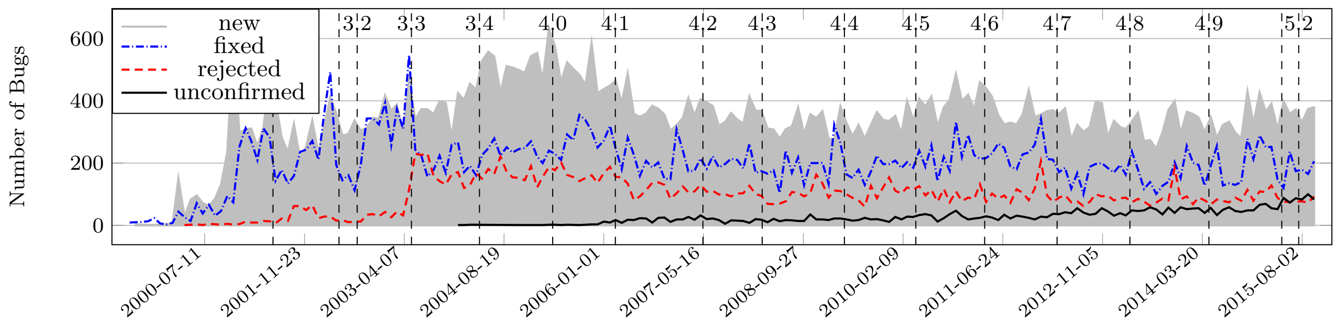
Table 1 shows the numbers of bugs and their accompanied revisions that are used in our study. We analyze 50K fixed bugs and 30K revisions in total, including 1,858 GCC and 1,643 LLVM enhancement requests.

2.2 Threats to Validity

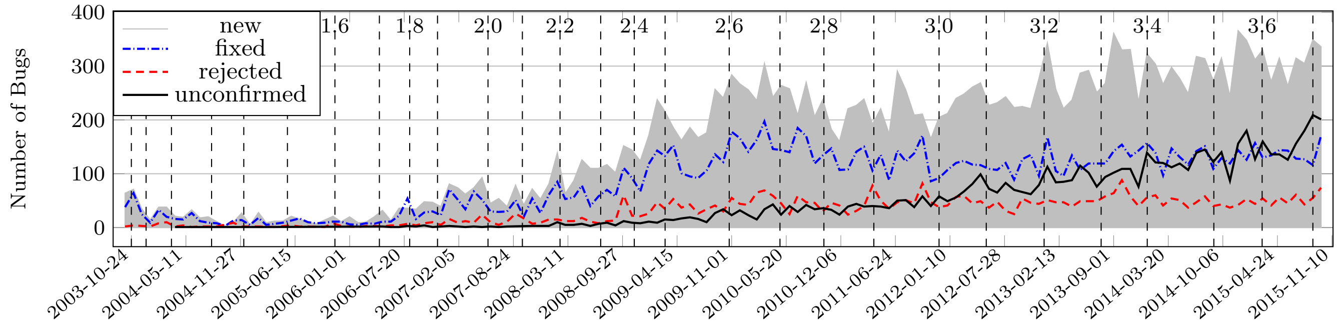
Similar to other empirical studies, our study is potentially subject to several threats, namely the representativeness of the chosen compilers, the generalization of the used bugs, and the correctness of the examination methodology.

Regarding representativeness of the chosen compilers, we use GCC and LLVM, which are two popular compiler projects written in C/C++ with multiple language frontends, a set of highly effective optimization algorithms and various architecture backends. Both are widely deployed on Linux and OS X operating systems. We believe these two compilers can well represent most of the traditional compilers. However, they may not well reflect the characteristics of Just-In-Time compilers (*e.g.*, Java Hotspot virtual machine) or interpreters (*e.g.*, Perl, Python).

Regarding the generalization of the used bugs, we uniformly use all the bug reports satisfying the selection crite-



(a) GCC.



(b) LLVM

Figure 1: The overall evolution history of the bug repositories (in months). The plot filled in gray background shows the new bug reports submitted every month. The blue *dashdotted* plot shows the number of bugs fixed every month. The red *dashed* plot shows the number of bugs that are resolved as *not* fixed (*e.g.* invalid, duplicate, worksforme or wontfix). The black curve shows the number of bug reports per month which have not been confirmed yet. Clearly there is an increasing trend of unconfirmed bugs for LLVM.

ria stated in Section 2.1 with no human intervention. For those unresolved or invalid bugs, we believe that they are not likely as interesting as the bugs investigated in this paper. As for the identification of revisions containing bug fixes, based on the interaction with the GCC and LLVM developers on a large number of bugs we reported before, GCC developers usually link the revisions and the bugs explicitly, while LLVM developers do less often. Given the large number of GCC and LLVM revisions, we believe that the analysis results should be representative.

Regarding the correctness of the analysis methodology, we have automated every analysis mentioned in this paper. We also have hands-on experience on analyzing bug report repositories and code revision histories from our previous studies.

3. GENERAL STATISTICS

This section shows the general statistics of the two bug repositories. Figure 1 shows the overall evolution of GCC and LLVM's bug repositories. In particular, it shows the number of bugs reported, rejected or fixed each month. It also shows the number of bug reports that have *never* been confirmed. Each vertical dashed line represents the date of a compiler release, and the label on the top the version number.

The trends of the plots for GCC are relatively stable in these years compared to those of LLVM. After gaining its popularity recently, LLVM has drawn much more attention than before and more bug reports are being submitted monthly. However, the bug-fixing rate (indicated by the blue dashdotted *fixed* curve) does not increase much and more bugs are being left as unconfirmed (shown as the black *unconfirmed* curve), which is likely due to limited human resources as we were told by active members in the LLVM community that some Apple developers were pulled into the Swift project.[1] This may give external bug reporters the impression that the community is not as responsive as GCC although bugs are fixed as regularly as before.

Table 2: The number and the percentage of bug reports revolved as `invalid`, `worksforme` or `wontfix`.

	invalid	worksforme	wontfix
GCC	7,072/10.4%	1,151/1.7%	1,463/2.2%
LLVM	1,639/6.7%	717/2.9%	593/2.4%

3.1 Rejected Bug Reports

Table 2 shows the information on the bugs which are resolved as `invalid`, `worksforme` or `wontfix`. An invalid bug report is one in which the associated test program is invalid (for example, containing undefined behaviors, a bug in another software, misunderstanding of the language standard, misuse of standard libraries), or the reported "anomalous" behavior is in fact deliberate. If a bug described in a report cannot be reproduced, this report is labeled as `worksforme`.

[1]https://developer.apple.com/swift/

A bug report is resolved as `wontfix` if the affected version, component or library of the compiler is not maintained although the reported bug can be confirmed.

3.2 Duplicate Bug Reports

Reporting bugs in a bug tracking system is uncoordinated. Different people may file multiple reports on the same bug. In this case, the latter bug reports are referred to as *duplicates*.

Table 3: Distribution of duplicate bug reports. The first row lists the number of duplicate bugs, and the second shows the number of bugs with the given number of duplicates.

(a) Duplicate bugs of GCC.

#Duplicate	0	1	2	3	4	5	≥6
#Report	35,933	2,924	596	215	98	41	83

(b) Duplicate bugs of LLVM.

#Duplicate	0	1	2	3	4	5	≥6
#Report	12,157	570	72	17	13	5	8

Table 3 shows the information on duplicate bug reports in the GCC and LLVM bug repositories. 9.9% of GCC bugs and 5.3% of LLVM bugs have duplicates. The duplicate rates are similar to other open source projects such as Eclipse, OpenOffice and Mozilla [30].

The GCC bug with the most duplicates is #4529 that has 60 duplicates. The compiler crashes when it is compiling the Linux kernel 2.4. The LLVM bug with the most duplicates is #2431 with 15 duplicates. It is a meta-bug report to manage all the crashing bugs related to the component `gfortran`.

3.3 Reopening Bugs

A bug report is labeled as fixed if a revision has been committed to fix the bug and passed certain test cases. However, sometimes the bug may be found to be not correctly or fully fixed later. In this case, the bug report will be reopened. There is another scenario of reopening a bug. If the bug repository administrator rejects a bug report first as he deems the bug invalid (*e.g.*, not reproducible) and later finds it valid, then the bug report will be reopened.

Both scenarios are normal but undesirable. Reopening a bug could mean careless bug fixes, inadequate communication between developers and testers or users, or inadequate information to reproduce the bug [40]. It can even become a bad practice if a bug report is reopened more than once, as this behavior is usually regarded as an anomalous software process execution [4, 28], and should be avoided in software development and management process.

Table 4: Breakdown of reopened bugs. The first row is the number of reopened times, and the second is the percentage of the corresponding bugs.

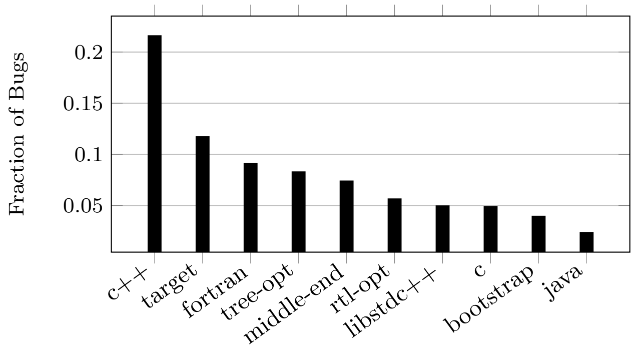
(a) Reopened bugs of GCC.

#Reopen	0	1	2	3	4	8
%	96.95	2.85	0.15	0.04	0.01	0.003

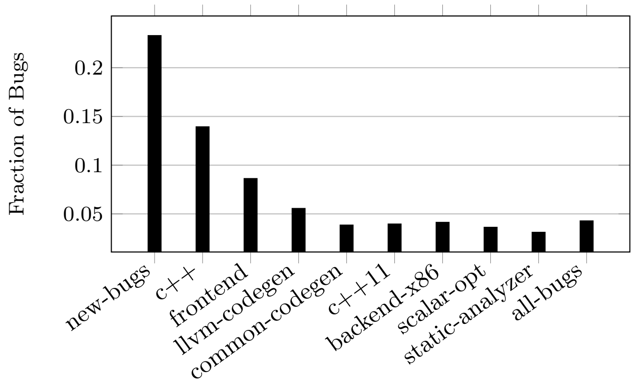
(b) Reopened bugs of LLVM.

#Reopen	0	1	2	3
%	94.596	5.007	0.374	0.023

Table 4 shows the reopening information of bugs in GCC and LLVM. Nearly 97% of the GCC bugs and 94% of the LLVM bugs are fixed directly without being ever reopened. The most frequently reopened bug of GCC is #29975[2] (reopened eight times). It is a meta bug to track Fortran bugs revealed by compiling the CP2K program.[3] Another GCC bug #52748[4] was reopened three times before it was fixed. It is related to the new feature `decltype` of C++11 standard, and took developers several attempts to correctly fix this bug.



(a) The top ten buggy components out of 52 in GCC. The most buggy component is C++, containing 22% of the 39,890 bugs, nearly twice as buggy as the second one. And these ten components account for 79% of the GCC bugs.



(b) The top ten buggy components out of 96 in LLVM. The most buggy component is new-bugs, containing 23% of the 12,842 bugs. The second component is C++, 14% and the third is Frontend 9%. And these ten components account for 79% of the LLVM bugs.

Figure 2: Distribution of Bugs in Components

4. LOCATION OF BUGS

This section answers the following question: Where are the bugs? Subsection 4.1 shows how the bugs are distributed in various compiler components, while Subsection 4.2 shows how bugs are distributed in different source files.

4.1 Distribution of Bugs in Components

Figure 2 shows the distribution of bugs in compiler components. We only show the top ten buggy components for each compiler, accounting for the majority of the bugs (79%

[2]https://gcc.gnu.org/bugzilla/show_bug.cgi?id=29975
[3]http://www.cp2k.org/
[4]https://gcc.gnu.org/bugzilla/show_bug.cgi?id=52748

Table 5: The top 10 buggy files of GCC and LLVM.

(a) GCC.

File	#	Description
cp/pt.c	817	C++ templates
cp/decl.c	638	C++ declarations and variables
cp/parser.c	595	C++ parser
config/i386/i386.c	569	code generation on IA-32
cp/semantics.c	457	semantic phase of parsing C++
fortran/resolve.c	452	type resolution for Fortran
cp/cp-tree.h	415	parsing and type checking C++
fold-const.c	386	constant folding
cp/typeck.c	374	type checking C++
cp/call.c	354	method invocations of C++

(b) LLVM.

File	#	Description
SemaDecl.cpp	301	semantic analysis for declarations
DiagnosticSemaKinds.td	268	definitions of diagnostics messages
SemaExpr.cpp	232	semantic analysis for expressions
SemaDeclCXX.cpp	221	semantic analysis for C++ declarations
X86ISelLowering.cpp	194	lowering LLVM code into a DAG for X86
SemaTemplate.cpp	127	semantic analysis for C++ templates
SemaExprCXX.cpp	124	semantic analysis for C++ expressions
SemaOverload.cpp	123	semnatic analysis for C++ overloading
lib/Sema/Sema.h	122	semantic analysis and AST building
SemaTemplateInstantiateDecl.cpp	119	C++ template declaration instantiation

for GCC and 79% for LLVM). These components touch all critical parts of compilers: front end (*e.g.* syntactic and semantic parsing), middle end (*e.g.* optimizations) and back end (*e.g.* code generation).

As the plots show, C++ is the most buggy component in GCC, accounting for around 22% of the bugs. It is much more buggy than the other components as the bug rate of the second most buggy component is only half. In LLVM, bug reports can be submitted without specifying their components. Therefore a large number of bug reports are closed with the component `new-bugs`. However, based on the distribution of bugs in source files (which will be discussed in the next section), C++ is also the most buggy component in LLVM.

One possible explanation is that C++ has many more features than the other programming languages, supporting multiple paradigms (*i.e.* procedural, object-oriented, and generic programming). It is surprising that most of the research and engineering efforts have been devoted to testing or validating C compilers [3, 11, 12, 13, 14, 23, 29, 34, 35] but little on C++, although C++ is one of the most popular programming languages in industry (among top three according to the TIOBE index [33]).

4.2 Distribution of Bugs in Files

This section analyzes the distribution of bugs in the source files (*i.e.*, how many bugs in each source file). We exclude the source files of test cases as they contribute nothing to the core functionalities of the compilers.

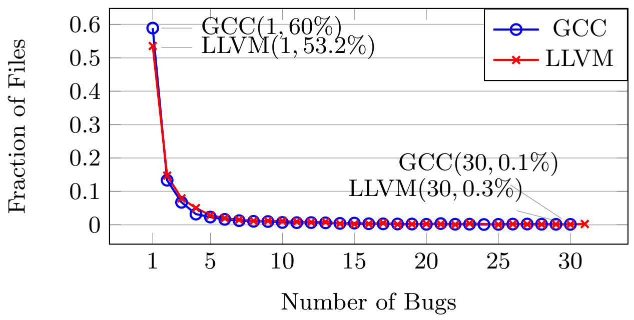
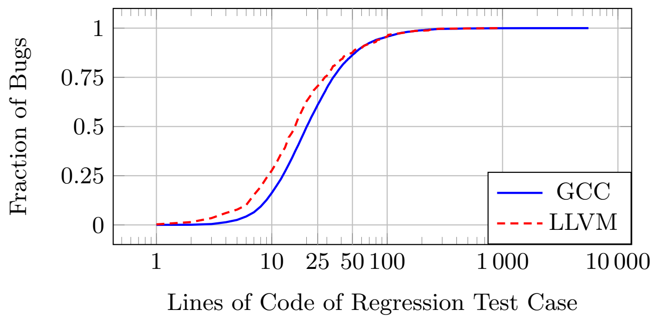


Figure 3: The figure shows the fraction of source files with a given number ($\leq$ 30) of bugs for GCC and LLVM. The files with more than 30 bugs are skipped and their fraction is approximately monotonically decreasing. For GCC, there are 5,039 files in total, and the file with the most bugs is ‘cp/pt.c’ (with 817 bugs); while for LLVM, there are 2,850 files in total, and the most buggy is ‘SemaDecl.cpp’ (with 301 bugs).



(a) This graph shows the empirical cumulative distribution function of the sizes of the test cases that trigger the bugs in GCC and LLVM,

(b) The statistics of the lines of code in regression test cases.

	Mean	Median	SD	Min	Max
GCC	32	21	75	1	5,554
LLVM	29	16	59	1	916

Figure 4: Size of regression test cases.

Figure 3 shows the fraction of source files with a given number of bugs, where the x-axis is the number of bugs, and the y-axis is the fraction of the source files containing that number of bugs. It depicts a phenomenon that half of the source files only contain one bug (60% of GCC and 53% of LLVM), and quite few files have a large number of bugs.

The plots in Figure 3 only show the files with no more than 30 bugs. In order to describe the other extreme of the distribution, Table 5 shows the top ten most buggy source files. Consistent with the distribution of bugs in components in Figure 2, the source files of the C++ component account for most bugs. For example, as perhaps the most complex feature of C++, the C++ template is the most buggy file in GCC. This skewness of bugs again implies that we should devote more efforts to testing C++ compilers, considering that C++ is one of the most wide-used programming languages.

5. REVEALING AND FIXING BUGS

In this section, we investigate the properties of bug-revealing test cases and the bug fixes.

5.1 Size of Bug-Revealing Test Cases

Figure 4a shows the empirical cumulative distribution function over the sizes of the test cases triggering the bugs of GCC and LLVM. As shown, most of the test cases 95%

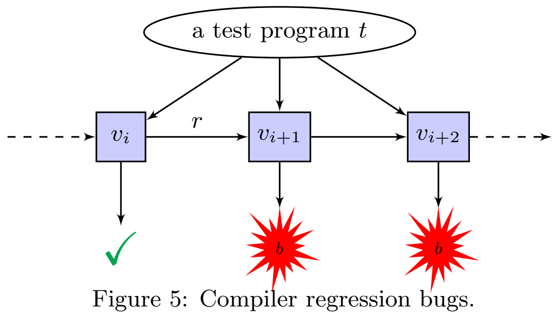


Figure 5: Compiler regression bugs.

are smaller than 100 lines of code, and more than 60% are smaller than 25 lines of code.[5] Table 4b shows the statistics including mean, median and standard deviation (SD).

These test cases are collected from the test suites of GCC and LLVM. For a new bug, besides the fix, the developer often creates a regression test case by extracting the attached bug triggering code in the bug report. Meanwhile a *bug label* is also created by concatenating a prefix `pr` and an infix `bug-id`, which is later used as the file name of the test case or inserted into the test case as a comment. For example, the test case `pr14963.c` is the regression test case for the bug `14963`.

By identifying the bug labels, we have collected 11,142 test cases of GCC and its size distribution is shown as the blue *solid* plot in Figure 4. We only collect 347 of LLVM, and this is why its plot (*i.e.*, the red *dashed* one) is not as smooth that of GCC. However, the overall trend is similar in the two plots, and supports the conclusion stated at the beginning of this section.

This observation can be leveraged for improving the effectiveness of random testing for compiler validation. A random program generator such as Csmith [35] or Orion [11] should focus on generating small but complex test programs rather than big but simple ones.

5.2 Time to Reveal Compiler Regressions

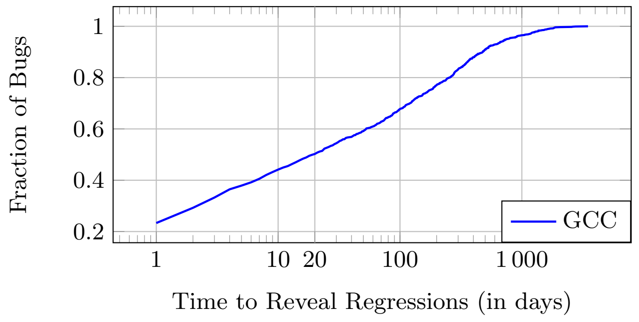
A compiler regression is a bug which is introduced by a revision, breaking the functionality of a feature. Consider the illustration in Figure 5. Let r be the committed revision between two consecutive versions v_i and v_{i+1} in the source code repository. We say revision r introduces a regression bug b into version v_{i+1}

In this section, we investigate how much time it takes for a regression bug to be uncovered by testing. For the regression bug b, its regression revealing time can be obtained by computing the time span between when the revision r introducing b is committed and when the bug report of b is filed. As GCC provides richer meta-information in its bug repository than LLVM, in this study we only focus on the regression bugs of GCC. The following describes how we collect the necessary information for this study.

Identifying Regression Bug Reports If a bug report is confirmed by a GCC developer as a regression, the summary of the bug report will be prefixed with a keyword `regression`.

Identifying Culprit Revisions GCC developers often link the culprit revisions with the regressions as additional comments to the bug reports. Although these correlations are only expressed in natural languages, they follow certain frequent patterns. For example, given a culprit revision `r`, the link can be written as `started with r`, `caused by r`, or `regressed with r`.

[5]We do not exclude comments from counting in this paper.

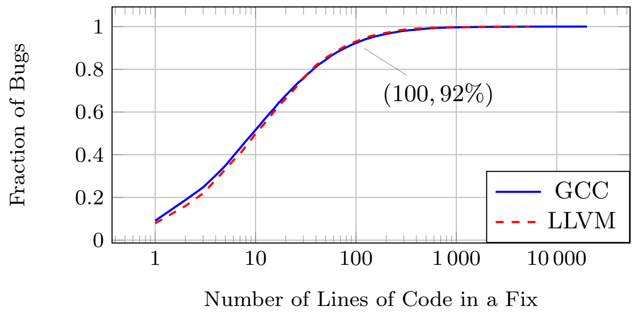


(a) This graph shows the empirical cumulative distribution function of the time interval between when a regression is introduced and when it is triggered.

(b) The statistics of the time to reveal regressions.

	Mean	Median	SD	Min	Max
GCC	163	20	335	1	3492

Figure 6: Time to reveal regression bugs.



(a) This graph shows the number of lines of code in a bug fix. The empirical cumulative distribution curves of GCC and LLVM are almost the same. And most of the bug fixes (92%) contain fewer than 100 lines of code.

(b) The statistics of the lines of code modification in bug fixes.

	Mean	Median	SD	Min	Max
GCC	43	10	264	1	20028
LLVM	38	11	161	1	5333

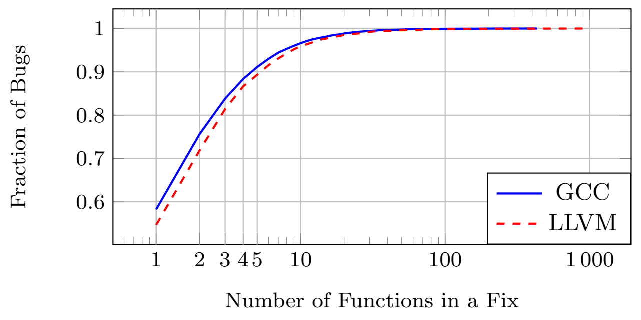
Figure 7: Lines of code modification in bug fixes.

We first collect bug reports of which the summaries contain the word `regression`, and then search the comments in these reports for the patterns of culprit revisions. In total, we have collected 1,248 regression bugs.

Figure 6 shows the empirical cumulative distribution function of the time interval between when a regression is introduced and when it is triggered. On average it takes 163 days, but 50% of these regressions only need 20 days to uncover.

5.3 Size of Bug Fixes

We consider all the 22,947 revisions of GCC and 8,452 revisions of LLVM and exclude the non-functional files (*e.g.*, change logs, documents, executables, test cases) in revisions.



(a) This graph shows the number of functions modified in a bug fix. The empirical cumulative distribution curve indicates that over half of the investigated bugs only involve one function (*i.e.*, 58% for GCC, and 54% for LLVM), and most of the bug fixes (90% or so) involve no more than 5 functions.

(b) The statistics of the number of functions in bug fixes.

	Mean	Median	SD	Min	Max
GCC	2.7	1	7.0	1	434
LLVM	3.4	1	18.4	1	972

Figure 8: Number of functions modified in bug fixes.

5.3.1 *Lines of Code*

We obtain the difference between two versions from the two version control systems respectively, and count the lines of code modification made to the source files.

As Figure 7a shows, 92% of the bug fixes contain fewer than 100 lines of code, and 50% of the bug fixes contain fewer than 10 lines. This indicates that most of the bugs only touch a small portion of the compiler code. Table 7b shows the statistics of the sizes of the bug fixes. On average, the number of lines of code modified in a bug fix is approximately 40, and the median is about 11.

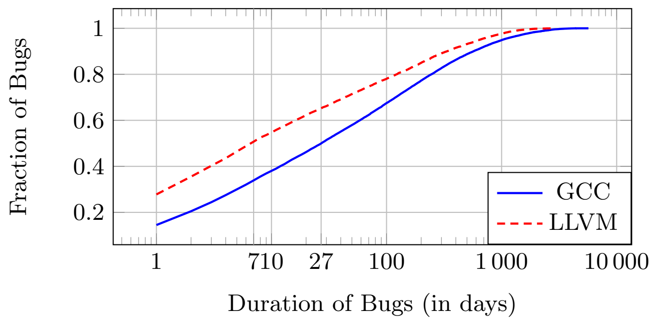
5.3.2 *Number of Functions*

We investigate the number of functions modified in a bug-fix. The information is acquired by (1) retrieving the changed source files at a specific revision, (2) parsing the files, and (3) locating functions of which the lines in files intersect with the line numbers recorded in the version control systems.

Figure 8a shows the relation between the number of functions revised in a revision and the fraction of bugs. Around 90% of the bugs involve at most 5 functions in GCC and LLVM. Moreover, more than 58% of the GCC bugs and 54% of LLVM only involve one single function. Table 8b shows the summary statistics. In terms of number of functions, the median is one and the mean is no more than three.

5.3.3 *Discussion*

The data — the lines of code and number of functions modified in a revision — imply that a compiler bug is usually not so complex that a severe collateral effect is imposed on multiple functions. Considering the plots and tables in Figures 7 and 8, we can conclude that compiler bugs are often local. Although compilers are intricate (such as the complex optimization algorithms), the bug fixes only have limited impact on the entire code-base. On the other hand, this indirectly demonstrates that both compilers are well designed with clear modularity and limited coupling among modules.



(a) This graph shows the empirical cumulative distribution function of bugs through time.

(b) The statistics of the duration of bugs.

	Mean	Median	SD	Min	Max
GCC	200	28	448	1	5686
LLVM	111	7	268	1	2967

Figure 9: This figure shows the statistics of the duration of bugs. Figure (a) is the emprical cumulative distribution function of the duration of bugs, and Table (b) displays the statistics of the duration. Averagely, a bug is fixed within a year. Comparatively, GCC takes longer time to fix bugs than LLVM.

6. DURATION OF BUGS

This section investigates how bugs are distributed through time. A duration spans the time when a bug report is filed in the bug tracking system to the time when the bug is fixed. Ideally, a bug should be fixed right after it is reported, which is zero-day duration. However in reality due to limited resource and various constraints, a bug usually takes time, sometimes years, to be fixed.

6.1 Collecting Duration Data

In the bug tracking systems of GCC and LLVM, when a bug report is filed, its *creation* date is saved in the database, which can be used as the birth date of the bug. The other field of date type in a bug report is the *modification* field, which records the date of the last revision to the report. However, it cannot be used as the killed date of the bug, as even after a bug is resolved as *fixed*, the developers may still edit the bug report, such as adjusting *target version* or adding comments. Hence, using the time interval between the creation date and the modification date is inaccurate. For example, the third bug[6] of GCC was reported on 1999-08-26, the last modification was on 2013-07-23, and therefore the time interval is more than ten years. However, based on its revision history,[7] the bug was already fixed on 2001-01-08, so the duration of bug should be two years, but not ten years.

In order to compute the bug lifetime information accurately, we retrieve both the bug reports and the bug history information from the two bug tracking systems. For the start date of a bug, we use the creation date recorded in the bug report. For the end date of the bug, we scan its revision history in reverse chronological order, and find the date of

[6]https://gcc.gnu.org/bugzilla/show_bug.cgi?id=3
[7]https://gcc.gnu.org/bugzilla/show_activity.cgi?id=3

the last revision to set the `resolution` field to fixed. Taking the bug #3 as an example, its duration is the time interval between its creation date 1999-08-26 in the report and the revision date 2001-01-08 to set it as fixed.

6.2 Duration Analysis

Figure 9 shows the empirical cumulative distribution function of bugs over time, and the statistics of the bug duration. On average, the bugs of GCC are fixed within 111 days, and those of LLVM are fixed in 98 days. The medians of durations are 28 days and 7 days respectively.

Table 6 further breaks down the duration into two segments, *i.e.*, the duration between when a bug is reported and when it is confirmed, and the duration between it is confirmed and fixed. On average, it takes less time for GCC to confirm a bug but longer time to fix a bug than LLVM.

Table 6: The breakdown of the duration.

(a) Duration of bugs between when they are *reported* and *confirmed*.

	Mean	Median	SD	Min	Max
GCC	61	2	205	1	3816
LLVM	95	5	245	1	2967

(b) Duration of bugs between when they are *confirmed* and *fixed*.

	Mean	Median	SD	Min	Max
GCC	139	4	396	1	5427
LLVM	17	1	113	1	2446

7. PRIORITIES OF BUGS

This section first studies the priority of compiler bugs, and then investigates the correlation between priorities and other types of bug information, *i.e.*, compiler components and bug duration.

7.1 Priority Distribution

The field *priority* of a bug report is determined by developers for prioritizing their bugs, expressing the order in which bugs should be fixed by the developers. GCC has five levels: P1, P2, P3, P4 and P5. P1 is the most urgent level and bugs of this level should be fixed soon, and P5 has the lowest priority. P3 is the default priority level. Any new bug report is initially labeled as P3 by default, then the triage team will determine its priority based on the impact of the bug on users and available human and time resource [32]. LLVM developers do not prioritize bugs explicitly in the bug repository, so we only conduct priority-related analyses on the GCC data set.

Table 7: The priority distribution of the GCC bugs.

P1	P2	P3	P4	P5
6.02%	22.92%	68.24%	1.82%	0.99%

Table 7 lists the breakdown of the five priority categories in GCC. As the default priority, P3 accounts for over half of the bugs, whereas P4 and P5 are the extreme cases with the fewest bugs.

7.2 Priority and Component Correlation

We study the correlation between priorities and components of bugs, that is, which compiler component tends to be more important and thus its bugs are more severe than the other components.

Given a compiler component c and a priority level $p \in [1,5]$ corresponding to $\langle$P1, P2, P3, P4, P5$\rangle$, let R be the set of the bugs in c. We define the following function to compute the fraction of the bugs with the given priority p among all the bugs in the component c.

$$\psi(c,p) = \frac{|\{r \in R | \text{the priority of } r \text{ is } p\}|}{|R|}$$

We then define a total order between components in the order of their fractions of five different priorities.

$$\Theta(c_1,c_2) = \theta(c_1,c_2,1)$$

$$\theta(c_1,c_2,p) = \begin{cases} > & \text{if } \psi(c_1,p) > \psi(c_2,p) \\ < & \text{else if } \psi(c_1,p) < \psi(c_2,p) \\ = & \text{else if } p = 5 \\ \theta(c_1,c_2,p+1) & \text{otherwise} \end{cases}$$

Basically, if a component c_1 has a larger fraction of bugs with high priorities than c_2, then c_1 is greater than c_2. Bugs in a "large" component are more likely to be prioritized higher than those in "small" components, and should draw more attention for testing and validation.

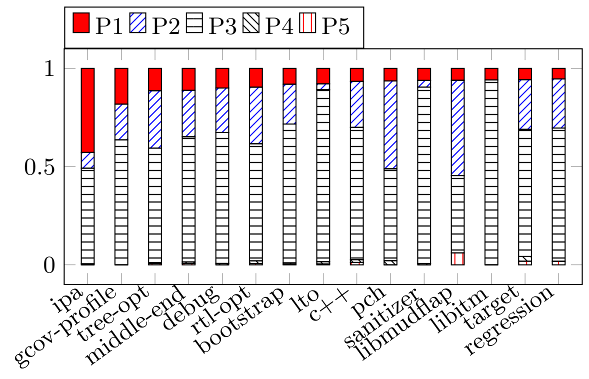


Figure 10: Correlation between priorities and components

We rank all the components of GCC in the descending order of Θ. Figure 10 shows the top 15 components. The first component is `ipa`, which includes the inlining and other inter-procedural optimizations and the infrastructure supporting them. 30% of its bugs are labeled as P1. This ratio is significantly higher than the second component, which is only 14%. Many other optimization components also appear in this list, such as `tree-opt` (optimizations over high-level tree representation), `middle-end` (optimizations over GIMPLE representation [9]), `rtl-opt` (optimizations over low-level architecture-neutral register transfer language representation [10]), `lto` (link-time optimization).

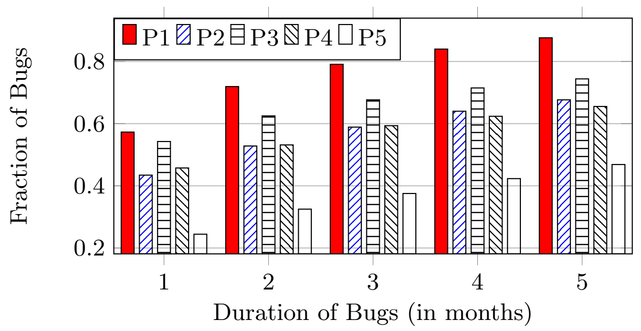
7.3 Priority and Duration Correlation

This subsection studies the correlation between priorities and the lifetime of bugs. Intuitively, bugs with higher priorities should be fixed in a shorter time than those with lower priorities. The results in this section partially invalidate this hypothesis.

Table 11a lists the statistics of the lifetime of bugs for the five priority levels. Each row shows the information of all the bugs that are labeled with a certain priority. For example,

(a) The statistics of the duration of bugs for different priority levels in GCC.

Priority	Mean	Median	SD	Min	Max
P1	72	22	149	1	2196
P2	262	49	502	1	4937
P3	185	21	420	1	5686
P4	268	48	473	1	3326
P5	450	170	617	1	3844



(b) The cumulative breakdown of the bug lifetime for different priorities within 5 months

Figure 11: The correlation between duration and priorities of GCC bugs.

the second row shows the mean, median and standard deviation of all the bugs of P1. Additionally, we performed t-test between every two priorities, and validated that except the difference between P2 and P4 the differences between all others are statistically significant with ($p < 0.001$).

From a different perspective, Figure 11b graphically shows the cumulative breakdown of the bug lifetime for each priority level within five months. For example, 79% of P1 bugs are fixed within three month, 58% for P2, 68% for P3, 59% for P4 and 38% for P5.

Both the table and the figure demonstrate that the time spent in fixing bugs does not strictly follow the order of their priorities. On average, only the bugs with P1 are fixed much faster than the other priorities, and the bugs with P5 take the most time among all the priorities. The P2 bugs take more time than P3. The P1 bug with the maximum lifetime is Bug #25130.[8] It was reopened twice, and took 2,196 days in total to resolve.

8. A PRELIMINARY APPLICATION

This section presents a proof-of-concept application of our findings. Specifically, we have designed `tkfuzz`, a simple yet effective program mutation algorithm for compiler testing, by leveraging the observation in Subsection 5.1 that bug-revealing test cases are usually small. Moreover, as indicated in Subsection 4.1 that C++ is the most buggy component, our algorithm works for C++ too, the first research effort on C++ compiler testing.

`tkfuzz` Given a test program represented as a sequence of tokens, `tkfuzz` randomly substitutes an identifier token (*e.g.*, variable or function names) with another different identifier token in the token list. Then the mutated programs are used to find crashing bugs in compilers.

[8]https://gcc.gnu.org/bugzilla/show_activity.cgi?id=25130

Table 8: Bugs Found by `tkfuzz`

	Bug ID	Component	Status
1	LLVM-24610	frontend	fixed
2	LLVM-24622	frontend	fixed
3	LLVM-24797	new-bugs	–
4	LLVM-24798	c++	fixed
5	LLVM-24803	new-bugs	–
6	LLVM-24884	new-bugs	–
7	LLVM-24943	new-bugs	–
8	LLVM-25593	new-bugs	–
9	LLVM-25634	new-bugs	–
10	GCC-67405	target	fixed
11	GCC-67581	c++	fixed
12	GCC-67619	middle-end	fixed
13	GCC-67639	middle-end	confirmed
14	GCC-67653	middle-end	fixed
15	GCC-67845	c++	fixed
16	GCC-67846	c++	fixed
17	GCC-67847	c++	fixed
18	GCC-68013	tree-opt	fixed

We applied `tkfuzz` to the GCC test suite to generate mutated programs. The test suite contains 36,966 test programs (24,949 for C and 12,017 for C++); the average size of each test program is only 32 lines of code, including blank lines and comments. Our testing was done on a quad-core Intel 3.40 GHz machine. We have reported 18 bugs, of which 12 have already been accepted or fixed. Table 8 shows the details of these bugs. Note that five bugs are in the two compilers' C++ components, confirming our observation in Subsection 4.1. Although this is a simple application, it clearly demonstrates the practical potentials of our findings in this paper.

9. CALL FOR ACTIONS

We have shown that compiler bugs are common: there is a large number of them. This section discusses several directions that are worthy of pursuing based on the analysis results presented earlier.

Buggy C++ Components It is not very surprising that C++ components are the most buggy in both compilers, as C++ is one of the most complex programming languages. However, it is surprising that little research has been directed at testing C++. Although it is more difficult than testing C compilers [11, 35], it is very worthy because C++ is used as widely as C. Furthermore, based on the most buggy files of the two compilers (*cf.* Table 5), we can take gradual steps toward testing C++ by starting from the most buggy features such as *templates* and *overloaded methods*.

Small Regression Tests Figure 4 shows the size of the regression test cases, which are extracted from the test programs attached to bug reports. 95% of them have fewer than 100 lines of code, and more than 50% fewer than 25 lines of code. This interesting (and surprising) finding indicates that randomized compiler testing techniques can leverage this fact by producing small but complex test programs. This not only can stress test compilers, but may also enhance testing efficiency. Our preliminary application of this finding in Section 8 also demonstrates its potential impact.

Locality of Fixes Section 5.3 shows that most of the bug fixes only touch one function (58% for GCC and 54% for LLVM), and they tend to be local. Thus, it would be promising to investigate techniques that can direct testing at a chosen component, rather than treating the compiler as a monolithic whole.

Duration of Bugs Section 6 shows that the average time to triage and resolve a compiler bug is a few months. This may be because compilers are very complex and compiler bugs are difficult to resolve. One has to understand the root cause of a bug and decide how to fix it; at the same time, it is important not to overlook certain cases and avoid regressions. Thus, practical techniques are needed to aid compiler developers with such tasks. Lopes *et al.*'s recent work on Alive [16] can be viewed as a relevant example in this direction — it helps developers write and debug peephole optimizations.

10. RELATED WORK

We survey two lines of closely related research.

Empirical Studies on Bugs Much work has been devoted to studying the characteristics of various bugs in various software systems. Chou *et al.* [7] conducted an empirical study on approximately one thousand operating system errors. The errors were collected by applying static automatic compiler analysis to Linux and OpenBSD kernels. They found that device drivers had much more bugs than the rest of the kernels. Lu *et al.* [17] studied the characteristics of concurrency bugs by examining 105 concurrency bugs randomly selected from four real-world programs (MySQL, Apache, Mozilla and OpenOffice). Their findings further the understanding of concurrency bugs and highlight future directions for concurrency bugs detection, diagnosis and fixing. Sahoo *et al.* [25] analyzed 266 reported bugs found in released server software, such as MySQL, Apache, and SVN. Based on the findings, they discussed several implications on reproducing software failures and designing automated diagnosis tools for production runs of server software. Li *et al.* [15] analyzed the trend of bugs by applying natural language text classification techniques to about 29,000 bugs of Mozilla and Apache. Thung *et al.* [31] studied the bugs in machine learning systems and categorized them based on the characteristics of bugs. Song *et al.* [26] studied performance bugs in open source projects and based on their characteristics proposed a statistical debugging technique.

Our work complements these previous studies, with a specific focus on compiler bugs. Different from application bugs, compiler bugs can be much difficult to notice and debug, and for application developers, compilers are usually assumed to be bug-free. In this paper, we show that compiler bugs are also common, and more than 65% bugs of GCC and LLVM are reported by external users. For researchers working on compiler testing and validation, we show that certain compiler components have higher bug rates than the others, and should be paid more attention to.

Compiler Testing Due to compilers' complexity, testing is still the major technique to validate the correctness of production compilers. In addition to internal regression test suites, compiler developers can also use external commercial conformance testing suites [1, 22] to further test whether compilers conforms to language standards or specifications. However, such manual test suites may still be inadequate and therefore recently researchers have started to employ randomized testing to further stress test compilers.

One of the most successful approaches is Csmith [5, 24, 35], which has found several hundred bugs in GCC and LLVM. Compared to traditional random C program generators which target at compiler crashes, Csmith is able to generate valid C programs by avoiding introducing undefined behaviors, hence capable of finding mis-compilation bugs. It has also been applied to test virtual machines [18], CPU emulators [19] and static analyzers, such as Frama-C [8].

Another successful compiler testing technique is Equivalence Modulo Inputs (EMI) [11, 12, 13]. It has found several hundred bugs in GCC and LLVM. EMI is a general compiler testing methodology to derive semantically equivalent variants from existing programs. It introduces an alternative view of differential testing. Given a program P, instead of verifying the consistency between executables compiled by multiple compilers or multiple versions of a compiler on P, it tests the consistency *w.r.t.* an input I between executables from P and P' compiled by the same compiler, where P' is an EMI variant of P *w.r.t.* I.

A considerable amount of effort has also been put on testing different compilers or different components in compilers. Zhao *et al.* proposed a tool JTT to test the EC++ embedded compiler [39]. Nagai *et al.* [20, 21] proposed a technique to test the arithmetic optimizers of compilers. CCG is another random C program generator targets compiler crashing bugs [2]. Sun *et al.* [29] proposed an approach to finding bugs in compiler warning diagnostics.

Our study highlights that in GCC and LLVM, C++ has the highest bug rate of all the compiler components, much higher than the C component. It would be interesting to devise effective testing strategies, theories and tools to test C++ compilers, as it is also a popular but more complex programming language widely used in industry. We also analyze the statistics of bug revealing test cases and bug fixes and find that most of the test cases are small and most of the bug fixes are local to a small number of lines. These observations can potentially serve as good heuristics to guide random program generator to generate test programs, small in size but effective at detecting new bugs.

11. CONCLUSION

This paper has presented a study of in total 39,890 bugs and 22,947 bug fixes of GCC, and 12,842 bugs and 8,452 bug fixes of LLVM, and analyzed the characteristics of compiler bugs. In particular, we have shown how bugs are distributed in components and source files (skewness of bugs in a small number of components and files), how bugs are triggered and fixed (both bug-triggering test cases and fixes are small in size), how long bugs live (on average 200 days for GCC and 111 for clang), and how bugs are prioritized.

We believe that our analysis results and findings provide insight into understanding compiler bugs and guidance toward better testing and debugging compilers. All our data and code are publicly available at http://chengniansun.bitbucket.org/projects/compiler-bug-study/.

Acknowledgments

We are grateful to the anonymous reviewers for their insightful comments. This research was supported in part by the United States National Science Foundation (NSF) Grants 1117603, 1319187, 1349528 and 1528133, and a Google Faculty Research Award. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

References

[1] ACE. SuperTest compiler test and validation suite. http://www.ace.nl/compiler/supertest.html.

[2] A. Balestrat. CCG: A random C code generator. https://github.com/Merkil/ccg/.

[3] S. Blazy, Z. Dargaye, and X. Leroy. Formal Verification of a C Compiler Front-End. In *Int. Symp. on Formal Methods (FM)*, pages 460–475, 2006.

[4] N. Chen, S. C. H. Hoi, and X. Xiao. Software Process Evaluation: A Machine Learning Approach. In *ASE*, pages 333–342, Washington, DC, USA, 2011.

[5] Y. Chen, A. Groce, C. Zhang, W.-K. Wong, X. Fern, E. Eide, and J. Regehr. Taming compiler fuzzers. In *Proceedings of the 2013 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 197–208, 2013.

[6] R. Chillarege, W.-L. Kao, and R. G. Condit. Defect Type and Its Impact on the Growth Curve. In *Proceedings of the 13th International Conference on Software Engineering (ICSE)*, pages 246–255, 1991.

[7] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An Empirical Study of Operating Systems Errors. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (SOSP)*, pages 73–88, 2001.

[8] P. Cuoq, B. Monate, A. Pacalet, V. Prevosto, J. Regehr, B. Yakobowski, and X. Yang. Testing static analyzers with randomly generated programs. In A. Goodloe and S. Person, editors, *NASA Formal Methods*, volume 7226 of *Lecture Notes in Computer Science*, pages 120–125. Springer Berlin Heidelberg, 2012.

[9] GCC. GIMPLE – GNU Compiler Collection (GCC) Internals. https://gcc.gnu.org/onlinedocs/gccint/GIMPLE.html, accessed: 2014-06-25.

[10] GCC. RTL – GNU Compiler Collection (GCC) Internals. https://gcc.gnu.org/onlinedocs/gccint/RTL.html, accessed: 2014-06-25.

[11] V. Le, M. Afshari, and Z. Su. Compiler Validation via Equivalence Modulo Inputs. In *Proceedings of the 2014 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2014.

[12] V. Le, C. Sun, and Z. Su. Finding Deep Compiler Bugs via Guided Stochastic Program Mutation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 386–399. ACM, 2015.

[13] V. Le, C. Sun, and Z. Su. Randomized Stress-Testing of Link-Time Optimizers. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA)*, pages 327–337. ACM, 2015.

[14] X. Leroy, A. W. Appel, S. Blazy, and G. Stewart. The CompCert Memory Model, Version 2. Research report RR-7987, INRIA, June 2012.

[15] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai. Have Things Changed Now?: An Empirical Study of Bug Characteristics in Modern Open Source Software. In *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability (ASID)*, pages 25–33, 2006.

[16] N. P. Lopes, D. Menendez, S. Nagarakatte, and J. Regehr. Provably correct peephole optimizations with alive. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 22–32, 2015.

[17] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 329–339, 2008.

[18] L. Martignoni, R. Paleari, G. Fresi Roglia, and D. Bruschi. Testing system virtual machines. In *Proceedings of the 19th International Symposium on Software Testing and Analysis (ISSTA)*, pages 171–182, 2010.

[19] L. Martignoni, R. Paleari, A. Reina, G. F. Roglia, and D. Bruschi. A methodology for testing cpu emulators. *ACM Trans. Softw. Eng. Methodol.*, 22(4):29:1–29:26, Oct. 2013.

[20] E. Nagai, H. Awazu, N. Ishiura, and N. Takeda. Random testing of C compilers targeting arithmetic optimization. In *Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI 2012)*, pages 48–53, 2012.

[21] E. Nagai, A. Hashimoto, and N. Ishiura. Scaling up size and number of expressions in random testing of arithmetic optimization of C compilers. In *Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI 2013)*, pages 88–93, 2013.

[22] Plum Hall, Inc. The Plum Hall Validation Suite for C. http://www.plumhall.com/stec.html.

[23] A. Pnueli, M. Siegel, and E. Singerman. Translation Validation. In *4th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, pages 151–166, 1998.

[24] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang. Test-case reduction for C compiler bugs. In *Proceedings of the 2012 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 335–346, 2012.

[25] S. K. Sahoo, J. Criswell, and V. Adve. An Empirical Study of Reported Bugs in Server Software with Implications for Automated Bug Diagnosis. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 485–494, 2010.

[26] L. Song and S. Lu. Statistical Debugging for Real-world Performance Problems. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 561–578, 2014.

[27] M. Sullivan and R. Chillarege. A Comparison of Software Defects in Database Management Systems and Operating Systems. In *Twenty-Second International Symposium on Fault-Tolerant Computing (FTCS)*, pages 475–484, July 1992.

[28] C. Sun, J. Du, N. Chen, S.-C. Khoo, and Y. Yang. Mining Explicit Rules for Software Process Evaluation. In *ICSSP*, pages 118–125, 2013.

[29] C. Sun, V. Le, and Z. Su. Finding and Analyzing Compiler Warning Defects. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*. ACM, 2016.

[30] C. Sun, D. Lo, X. Wang, J. Jiang, and S.-C. Khoo. A Discriminative Model Approach for Accurate Duplicate Bug Report Retrieval. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 45–54, 2010.

[31] F. Thung, S. Wang, D. Lo, and L. Jiang. An Empirical Study of Bugs in Machine Learning Systems. In *Software Reliability Engineering (ISSRE), 2012 IEEE 23rd International Symposium on*, pages 271–280, Nov 2012.

[32] Y. Tian, D. Lo, and C. Sun. DRONE: Predicting Priority of Reported Bugs by Multi-factor Analysis. In *29th IEEE International Conference on Software Maintenance (ICSM)*, pages 200–209, Sept 2013.

[33] TIOBE. TIOBE Index for May 2016. http://www.tiobe.com/tiobe_index, accessed: 2016-05-15.

[34] J.-B. Tristan and X. Leroy. Formal Verification of Translation Validators: A Case Study on Instruction Scheduling Optimizations. In *Proceedings of the 35th ACM Symposium on Principles of Programming Languages (POPL)*, pages 17–27, Jan. 2008.

[35] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 2011 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 283–294, 2011.

[36] Z. Yin, X. Ma, J. Zheng, Y. Zhou, L. N. Bairavasundaram, and S. Pasupathy. An empirical study on configuration errors in commercial and open source systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP)*, pages 159–172, 2011.

[37] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram. How Do Fixes Become Bugs? In *19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE)*, pages 26–36, 2011.

[38] A. Zeller and R. Hildebrandt. Simplifying and Isolating Failure-Inducing Input. *IEEE Trans. Softw. Eng.*, 28(2):183–200, Feb. 2002.

[39] C. Zhao, Y. Xue, Q. Tao, L. Guo, and Z. Wang. Automated test program generation for an industrial optimizing compiler. In *ICSE Workshop on Automation of Software Test (AST)*, pages 36–43, 2009.

[40] T. Zimmermann, N. Nagappan, P. J. Guo, and B. Murphy. Characterizing and Predicting Which Bugs Get Reopened. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, pages 1074–1083, 2012.