Debug Information Validation for Optimized Code

Yuanbo Li
Georgia Institute of Technology
USA
yuanboli@gatech.edu

Shuo Ding
Georgia Institute of Technology
USA
sding@gatech.edu

Qirun Zhang
Georgia Institute of Technology
USA
qrzhang@gatech.edu

Davide Italiano
Apple Inc.
USA
davidino@apple.com

Abstract

Almost all modern production software is compiled with optimization. Debugging optimized code is a desirable functionality. For example, developers usually perform post-mortem debugging on the coredumps produced by software crashes. Designing reliable debugging techniques for optimized code has been well-studied in the past. However, little is known about the correctness of the debug information generated by optimizing compilers when debugging optimized code.

Optimizing compilers emit debug information (*e.g.*, DWARF information) to support source code debuggers. Wrong debug information causes debuggers to either crash or to display wrong variable values. Existing debugger validation techniques only focus on testing the interactive aspect of debuggers for dynamic languages (*i.e.*, with unoptimized code). Validating debug information for optimized code raises some unique challenges: (1) many breakpoints cannot be reached by debuggers due to code optimization; and (2) inspecting some arbitrary variables such as uninitialized variables introduces undefined behaviors.

This paper presents the first generic framework for systematically testing debug information with optimized code. We introduce a novel concept called *actionable program.* An actionable program $P_{\langle s,v \rangle}$ contains a program location s and a variable v to inspect. Our key insight is that in both the unoptimized program $P_{\langle s,v \rangle}$ and the optimized program $P'_{\langle s,v \rangle}$, debuggers should be able to stop at the program location s and inspect the value of the variable v without any undefined behaviors. Our framework generates actionable programs and does systematic testing by comparing the debugger output of $P'_{\langle s,v \rangle}$ and the actual value of v at line s in $P_{\langle s,v \rangle}$. We have applied our framework to two mainstream optimizing C compilers (*i.e.*, GCC and LLVM). Our framework has led to 47 confirmed bug reports, 11 of which have already been fixed. Moreover, in three days, our technique has found 2 confirmed bugs in the Rust compiler. The results have demonstrated the effectiveness and generality of our framework.

CCS Concepts: • **Software and its engineering → Software testing and debugging**; **Compilers**.

Keywords: Debug Information, Optimizing Compilers

ACM Reference Format:
Yuanbo Li, Shuo Ding, Qirun Zhang, and Davide Italiano. 2020. Debug Information Validation for Optimized Code. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '20), June 15–20, 2020, London, UK*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3385412.3386020

1 Introduction

Debugging is an essential part of software development. Compilers are typically available with companion source code debuggers. To aid debugging, compilers generate *debug information* together with the machine code [2]. For instance, C compilers offer a flag "-g" to enable debug information. The debug information establishes a mapping between source code and machine code.

Debugging optimized code is a desirable feature [1, 10]. Modern software is compiled with compiler optimizations. Debug information is extremely useful for diagnosing software failures. For instance, to analyze a crash, developers could immediately attach a debugger to the running optimized program compiled with debug information. Therefore, some Linux distributions[1] recommend adding the flag "-g -O2" to customize software builds and provide better quality assurance support. Also, many concerning issues such as performance bugs only exhibit in optimized code, which makes debugging with optimized code mandatory. Some software products are shipped with debug information. For example,

PLDI '20, June 15–20, 2020, London, UK
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7613-6/20/06...$15.00
https://doi.org/10.1145/3385412.3386020

[1] https://wiki.gentoo.org/wiki/Project:Quality_Assurance/Backtraces

```
1 char a,b;
2 int main(){
3   unsigned c;
4   --b;
5   c=2;
6   if(a)
7     b=0;
8   return 0;
9 }
```

(a) Program P.

```
1 char a,b;
2 int main(){
3   unsigned c;
4   --b;
5   c=2;
6   if(a) // c
7     b=0;
8   return 0;
9 }
```

(b) Actionable Program $P_{\langle 6,c\rangle}$.

```
1  char a,b;
2  int main(){
3    unsigned c;
4    --b;
5    opt_me_not(); // b
6    c=2;
7    if(a)
8      b=0;
9    return 0;
10 }
```

(c) Actionable Program $P_{\langle 5,b\rangle}$.

```
1  char a,b;
2  int main(){
3    unsigned c;
4    --b;
5    opt_me_not(); // c
6    c=2;
7    if(a)
8      b=0;
9    return 0;
10 }
```

(d) Non-actionable Program.

Figure 1. Illustrative examples of the actionable programs transformation. The function `opt_me_not()` is defined in another compilation unit. Compiled with GCC and Clang using "`-O3 -g`", the set of available lines is explicitly encoded in the corresponding DWARF debug information. Note that in program P, the breakpoint at line 6 can always be reached in both GDB and LLDB.

the latest Mozilla Firefox on MacOS is built with the "`-g -O3`" flag.[2] The Linux software produced by GNU autotools [7] has the default flag "`-g -O2`" in the release builds.

Since the work of Hennessy [14], the problem of debugging optimized code has been extensively studied in the literature. Two decades ago, a debugger manual explicitly warned users "(in order to) enable debugging, turn off all optimizations."[1, 34] After tremendous developments, debugging optimized code has become a common practice and generating debug information for optimized code is a mature feature in production compilers [13]. For instance, the latest Red Hat developer guide recommends the build flag "`-g -O2`" with GCC for all developers targeting Red Hat Enterprise Linux [26, Chapter 15.5]. Despite the considerable efforts, little progress has been made to validate the debug information generated by compilers. Do compilers always generate correct debug information for optimized code?

This paper fills the gap and presents the first general framework to validate debug information for optimized code. In particular, we restrict the validation to testing the validity of compiler-generated debug information. We focus on two most fundamental tasks in debugging: setting a breakpoint and printing the value of a variable. With these two operations, we consider the following problem: how to generate interesting input programs for testing debug information associated with optimized code? A straightforward approach is adopting existing program generation techniques for compiler testing [19, 35, 36]. Unfortunately, this is inadequate. In compiler testing, each input program contains a fixed set of output statements (*e.g.*, `print` and `return`) and is executed only once *w.r.t.* the program input. However, the act of debugging is more interactive. Even for one program input, users may inspect multiple program points with many variables in the input program. Formally, there are two unique challenges [14] in debug information validation:

- *Code Location Problem.* Compiler optimization rearranges instructions and eliminates statements. Many program points in the source code could be optimized out and may not even be available in the optimized code. Setting breakpoints at those locations does not necessarily stop the program execution in a debugger.
- *Data Value Problem.* Debuggers could literally inspect any variable at a specific program point. Unfortunately, printing an arbitrary variable may cause undefined behaviors such as accessing an uninitialized variable or an out-of-bound array element. The systematic testing must inspect only appropriate variables.

To illustrate the challenges, we consider a concrete example in Figure 1(a). It is a well-formed input program P for compiler testing. However, we could not directly adopt P for testing debug information. Due to the optimization, debuggers cannot stop at some lines in P. Moreover, suppose a debugger manages to stop at line 4. Printing variable `c` results in an undefined behavior since `c` is uninitialized at that program point (data value problem).

To facilitate testing debug information for optimized code, we introduce the *actionable programs*. Specifically, given a well-formed program P, we transform it to an actionable program $P_{\langle s,v\rangle}$ where s denotes a program location and v denotes a variable in P. Let $P_{\langle s,v\rangle}$ and $P'_{\langle s,v\rangle}$ be an unoptimized program and its optimized counterpart, respectively. The key insights behind the actionable program transformation are: (1) the location s in the optimized program $P'_{\langle s,v\rangle}$ can always be reached by a debugger; and (2) printing variable v at location s in $P_{\langle s,v\rangle}$ and $P'_{\langle s,v\rangle}$ does not yield any undefined behaviors. In particular, to tackle the code-location problem, if the location s has been optimized out, we insert a *barrier* (*i.e.*, an unoptimizable function) at s to guarantee that the corresponding breakpoint always hits. To tackle the data value problem, we perform dynamic analysis to ensure that printing variable v at s is always well-formed, which means it doesn't introduce any undefined behaviors. We discuss our dynamic analysis techniques in detail in Section 5. Figure 1(b) gives an actionable program $P_{\langle 6,c\rangle}$ transformed from the original program P in Figure 1(a). The debugger is able to stop at line 6 and printing the variable `c` is well-formed. Figure 1(c) also gives an actionable program $P_{\langle 5,b\rangle}$. Since the

[2] The Windows release of Mozilla Firefox is compiled with "`-Z7 -O2`" which enables CodeView debug information in object files.

debugger cannot stop at line 5 in the original optimized code, our transformation adds an external function `opt_me_not()` at that line. Then, a debugger can stop at the breakpoint at line 5 in the optimized code, and printing variable `b` is also well-formed. However, the program in Figure 1(d) is a non-actionable program since variable `c` is uninitialized even though the whole program is well-formed.

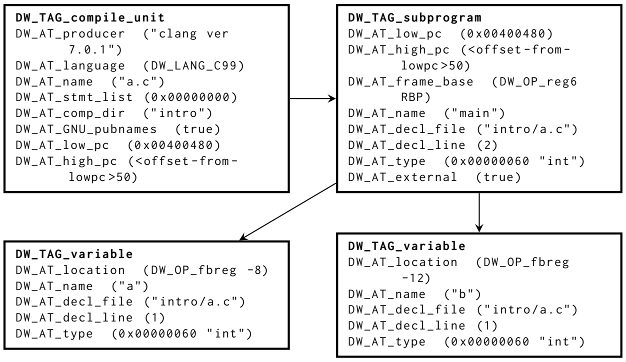
To our knowledge, our work is the first practical effort to test debug information for optimizing compilers. It is a relatively unexplored area and debuggers have received much less attention than other parts of the toolchain such as compilers. Existing work in this direction focuses on testing the interactive aspects of debuggers for dynamic languages [20]. It tests the functional correctness of the debuggers without taking the compiler optimization into consideration. Our work provides a generic debugging information testing framework for optimizing compilers. Specifically, we leverage both the unoptimized version $P_{\langle s,v \rangle}$ and the optimized version $P'_{\langle s,v \rangle}$ of actionable programs. The runtime value of v at location s in $P_{\langle s,v \rangle}$ serves as the reference. We compare the reference against the value of v at s retrieved by a debugger in $P'_{\langle s,v \rangle}$.

We have implemented the actionable program transformation for testing debug information generation in optimizing compilers. Our evaluation based on two widely-used C compilers (*i.e.*, GCC and LLVM) yields promising results. Our framework has led to 47 confirmed bug reports. The majority of those bugs reveal that compilers generate wrong debug information which causes debuggers to output wrong results for optimized code. Moreover, many of our reported bugs are latent which affect at least three recent releases of both compilers. The open-source community appreciates our initial effort in improving the debugging support in optimizing compilers. In particular, all of our reported bugs have been confirmed and about 23% of them have already been fixed.

This paper makes the following contributions:

- We formulate the problem of actionable program generation to facilitate systematic debug information testing for optimized code.
- We propose a practical realization of actionable program transformation.
- We lead the initial effort to improve the correctness of debugging support for optimizing compilers. Our technique has found 47 confirmed bugs in mature C compilers and 2 confirmed bugs in Rust compiler.

The remainder of this paper is structured as follows. Section 2 motivates our work via two concrete examples, and Section 3 gives the problem statement. We present our framework in Section 4 and experimental results in Section 5. Section 6 summarizes the lessons learned. Finally, Section 7 surveys related works, and Section 8 concludes.



```
DW_TAG_compile_unit
DW_AT_producer  ("clang ver
      7.0.1")
DW_AT_language  (DW_LANG_C99)
DW_AT_name  ("a.c")
DW_AT_stmt_list (0x00000000)
DW_AT_comp_dir  ("intro")
DW_AT_GNU_pubnames  (true)
DW_AT_low_pc  (0x00400480)
DW_AT_high_pc (<offset-from-
      lowpc>50)
```

```
DW_TAG_subprogram
DW_AT_low_pc  (0x00400480)
DW_AT_high_pc (<offset-from-
      lowpc>50)
DW_AT_frame_base  (DW_OP_reg6
      RBP)
DW_AT_name  ("main")
DW_AT_decl_file ("intro/a.c")
DW_AT_decl_line (2)
DW_AT_type  (0x00000060 "int")
DW_AT_external  (true)
```

```
DW_TAG_variable
DW_AT_location  (DW_OP_fbreg -8)
DW_AT_name  ("a")
DW_AT_decl_file ("intro/a.c")
DW_AT_decl_line (1)
DW_AT_type  (0x00000060 "int")
```

```
DW_TAG_variable
DW_AT_location  (DW_OP_fbreg
      -12)
DW_AT_name  ("b")
DW_AT_decl_file ("intro/a.c")
DW_AT_decl_line (1)
DW_AT_type  (0x00000060 "int")
```

Figure 2. DWARF debug information of the program in Figure 1(a). The program is compiled with Clang-7 with "-g". We omit some information for brevity.

2 Background and Motivating Examples

To inspect the value of a variable in optimized code, a source code debugger needs to query debug information generated by a compiler. In practice, both debuggers and compilers, like other software, can contain bugs. To improve debugging, previous work focuses on testing the correctness of debuggers [20]. Our work offers a new perspective and focuses on testing the debug information generated by optimizing compilers. This section provides a gentle introduction to debug information (Section 2.1) and motivates our work with two real-world examples (Section 2.2).

2.1 Debug Information

To aid debugging activities, compilers emit debug information together with the machine executables. Debug information contains references to functions, variables, and line numbers in the source code. When a debugging session starts, a debugger interprets the debug information to relate the running program to its source code. Almost all optimizing compilers support the flag "`-g`" to generate the debug information in the operating system's native format such as STABS [23], PDB [24], and DWARF [12]. Some compilers support specialized flags to further support its source code debugger. For example, GCC has a "`-ggdb`" flag to generate more precise information for GDB. DWARF is a popular debug information format used by mainstream compilers and debuggers. Unless otherwise stated, we always refer to the DWARF format of debug information in latter sections.

Figure 2 gives the DWARF debug information of the program in Figure 1(a). The DWARF information is organized in a tree structure. Node "`DW_TAG_compile_unit`" contains meta information about the compilation. It establishes the connection between the source code file (`a.c`) and the addresses ("`0x400480`" and offset 50) in the object file. It has a child node of type "`DW_TAG_subprogram`" which denotes a function ("`main()`"). Fields "`DW_AT_low_pc`" and "`DW_AT_high_pc`" represent, respectively, the beginning and end addresses of the function in the object file. In this example, the addresses

```
1 char a, b;
2 int main() {
3   int c = --a;
4   int l = -8L;
5   l = c;
6   c > (b = 0); // l
7 }
```

(a) Program $P_{\langle 6,l\rangle}$.

```
$ clang-9 -g a.c -O2
$ lldb-trunk a.out
(lldb) b 6
Breakpoint 1: a.out at a.c:6:10.
(lldb) r
-> 6      c > (b = 0); // l
(lldb) p l
(int) $0 = -8
```

(b) Wrong output at "-g -O2".

```
DW_TAG_variable
DW_AT_location (DW_OP_consts
      -8, DW_OP_stack_value)
DW_AT_name    (``l'')
DW_AT_decl_file    (``...'')
DW_AT_decl_line       (4)
DW_AT_type    (0x008f``int'')
```

Wrong DWARF info.

```
DW_TAG_variable
DW_AT_name    (``l'')
DW_AT_decl_file  (``...'')
DW_AT_decl_line       (4)
DW_AT_type    (0x008b``int'')
```

DWARF info after bug fix.

(c) DWARF information on variable l.

Figure 3. Clang-9.0.0 generates wrong debug information at "-g -O2". The bug has been fixed. We use the latest development version of LLDB in this example.

```
1 int a;
2 int main() {
3   int i;
4   for (; a < 10; a++)
5     i = 0;
6   for (; i < 6; i++)
7     ;
8   opt_me_not(); // i
9 }
```

(a) Program $P_{\langle 8,i\rangle}$.

```
$ gcc-8 a.c opt_me_not.c -g -O3
$ gdb a.out
(gdb) b 8
Breakpoint 1: file a.c, line 8.
(gdb) r
Breakpoint 1, main () at a.c:8
8         opt_me_not(); // i
(gdb) p i
$1 = 0
```

(b) Wrong output at " -g -O3".

```
DW_TAG_variable
DW_AT_name    ("i")
DW_AT_decl_file          ("...")
DW_AT_decl_line          (3)
DW_AT_decl_column        (0x07)
DW_AT_type    (0x003d "int")
DW_AT_const_value        (0x00)
```

Wrong DWARF info.

```
DW_TAG_variable
DW_AT_name    ("i")
DW_AT_decl_file    ("...")
DW_AT_decl_line       (3)
DW_AT_decl_column     (0x07)
DW_AT_type    (0x003d "int")
DW_AT_location     <loclist
      at offset 0x00000002
      with 1 entries follows>
```

DWARF info after bug fix.

(c) DWARF information on variable i.

Figure 4. GCC-8.3 generates wrong debug information at "-g -O3". Function opt_me_not() is defined in file opt_me_not.c. The bug has been fixed in the latest development branch. We use the latest development version of GDB in this example.

are the same as those in node "DW_TAG_compile_unit" because the compilation unit contains only one function. The node "DW_TAG_variable" shows variable information such as variable name, line number and type. Field "DW_AT_location" contains the variable location "DW_OP_fbreg - x" where "DW_OP_fbreg" denotes the frame's base address. To summarize, the DWARF debug information provides a mapping to bridge the source code and the machine executable.

2.2 Wrong Debug Information for Optimized Code

Mainstream optimizing compilers (*e.g.*, GCC and LLVM) support debugging with optimized code. Compiler optimizations not only affect the program itself but also impact the debug information generation. Even if compilers generate the correct optimized code, they may produce wrong debug information with it. We motivate our work with two concrete examples using the stable releases of GCC and LLVM, respectively. To read the debug information with the optimized code, we use GDB and LLDB, respectively.

LLVM Bug 43893. Figure 3 triggers an LLVM bug where the compiler generates wrong debug information at "-g -O2".[3] This bug affects the *latest* Clang-9 release. It has been fixed in the development version of Clang.

The expected value of variable l at line 6 should be −1. However, compiled with "-g -O2", LLDB outputs an incorrect value −8 as shown in Figure 3(b). The bug is the result of the InstCombine pass in LLVM dropping an unused variable but not correctly marking the debug information associated to the variable as unavailable. Thus, the DWARF info contains the result of the assignment instead of showing that it has been optimized out.

GCC Bug 89463. Figure 4 gives a GCC bug where the compiler fails to update the debug information when optimizing the code.[4] This bug was introduced in GCC version 7 release and affects the recent stable version of GCC-8.3. It has been fixed in the development version.

The expected value of variable i at line 8 in Figure 4(a) is 6. However, compiled with "-g -O3", the value of i at line 8 becomes 0 in GDB (Figure 4(b)). The bug was introduced by the dead code elimination optimization. When removing the loop body at line 6 in Figure 4(a), it does not clean the degenerated phi nodes in the intermediate representation. Those phi nodes prevents updating the debug information. Figure 4(c) illustrates the debug information of variable i. The old version of GCC fails to update variable i. Therefore, it remains as a constant. The latest version of GCC correctly clean the phi node and optimizes out the variable.

It is interesting to note that the wrong variable information in both examples is only observable in a debugger. GCC and LLVM generate the correct optimized machine executables (*i.e.*, we can print the correct runtime values by replacing function opt_me_not() with a printf() function). It demonstrates that the state-of-the-art optimizing compilers may generate wrong debug information for optimized code. Without our actionable program transformation, traditional program generation techniques could not detect the issue in the second example since the entire function body will be optimized out (*i.e.*, code location problem) and the value of variable i at a different location may be uninitialized (*i.e.*, data value program).

[3] https://bugs.llvm.org/show_bug.cgi?id=43893.
[4] https://gcc.gnu.org/bugzilla/show_bug.cgi?id=89463.

3 Problem Formulation

We introduce a systematic testing framework for validating debug information. This section formalizes the testing problem and our actionable program transformation.

3.1 Testing for Debug Information Validation

As discussed in Section 2.1, the debug information establishes the connection between the executable program P and the original source code. To interpret the debug information, we leverage debuggers in our work. In particular, a debugger *DBG* allows a programmer to observe the program execution. It takes as inputs a target program and an action sequence:

- *Target Program* P. It is the main subject in debugging sessions. Typically, developers use a debugger to inspect runtime properties in P. The executable P must be compiled with debugging support enabled (*e.g.*, with flag "-g" in most compilers).
- *Action Sequence* $\mathcal{A}$. It defines the debugging activity in debugging sessions. Specifically, the action sequence contains the debugger commands to inspect the program P. For instance, common actions include setting a breakpoint, doing a single step-in execution, and printing variables.

In practice, popular debuggers such as GDB and LLDB support a rich set of debugging actions.[5] Our work focuses on the most fundamental debugging operations: Setting a break point s in P and printing the value of variable v at s. We thus define a minimal *value-printing* sequence $A_{\langle s,v\rangle}$ where $A_{\langle s,v\rangle}$ represents "$break(s) \longrightarrow run \longrightarrow print(v) \longrightarrow quit$". Therefore, the debugger $DBG(P, A_{\langle s,v\rangle})$ outputs a number (*i.e.*, the value of v) after completing the debugging session.

Example 3.1. Consider the program P in Figure 1(a). Suppose we want to inspect the value of variable a at line 5. In unoptimized code, we can set a breakpoint at line 5 and print variable a. Therefore, we use the value-printing sequence $A_{\langle 5,a\rangle} = break(5) \rightarrow run \rightarrow print(a) \rightarrow quit$. The debugger should output that a's value is 0, *i.e.*, $DBG(P, A_{\langle 5,a\rangle}) = 0$.

Our goal is to systematically test debug information with optimized code P'. Hence, we need to compare the debugger output $DBG(P', A_{\langle s,v\rangle})$ with a reference. To obtain the reference value, we insert a print statement to print the value of v at s in the source code, then compile and run the program P without optimization. We use $PRINT(P, s, v)$ to denote the printed reference value.

Finally, the correct output of $DBG(P', A_{\langle s,v\rangle})$ should be the same as $PRINT(P, s, v)$. Let P be an unoptimized program and P' an optimized program generated from the same source code. Given a value-printing sequence $A_{\langle s,v\rangle}$, we have the following definition on correctness.

Definition 3.2 (Correctness)**.** The debug information for optimized code P' is correct iff $DBG(P', A_{\langle s,v\rangle}) \equiv PRINT(P, s, v)$.

Note that our work focuses on identifying *incorrect* debugger outputs. In the presence of code optimization, sometimes it is better to discard the debug information, in which case the debugger will simply indicate that a variable has been optimized out. Intuitively, discarding too much debug information renders an unuseful debugger implementation. It is indeed a quality-of-implementation issue based on the DWARF standard [9, Chapter 1.3.12]. Testing the *completeness* is orthogonal to our focus on testing the correctness. We believe that developers will generally only delete debug information when it is impossible to provide any reliable debug information. In Section 5.3, we explicitly give an example where developers fixed the bug via removing debug information, and we also explain why it is a reasonable choice.

3.2 Problem Statement

The core problem in testing debug information is how to generate suitable input programs. Compiler optimizations present unique challenges in the context of debug information validation. Specifically, the testing programs used in compiler testing cannot be directly adopted due to the code location and data value problems discussed in Section 1. Recall that to debug a program P, a debugger also needs a action sequence $A_{\langle s,v\rangle}$. We thus propose the actionable program transformation to support debugging with the action sequence $A_{\langle s,v\rangle}$ in program P.

Definition 3.3 (Actionable Program)**.** For a specific optimization process, an actionable program $P_{\langle s,v\rangle}$ contains a program location s and a variable v, such that both in the case of unoptimized program and optimized program, the debugger is able to stop at s, and printing v at s is well-defined.

A debugger can always apply the value-printing sequence $A_{\langle s,v\rangle}$ on actionable programs $P_{\langle s,v\rangle}$ and their optimized counterparts $P'_{\langle s,v\rangle}$. Therefore, we could leverage actionable programs for the subsequent testing process. To sum up, we consider the following technical problem in our paper.

> Given a program P, generate a set of actionable programs $\mathcal{P}$ for the subsequent systematic testing process.

The size of set $\mathcal{P}$ corresponds to the executed statements and well-defined variables in program P. Let s_i denote the program location in P and v_i the number of well-defined variables at each location. We have $|\mathcal{P}| = \sum_{i \in I} s_i v_i$, where I denotes the set of executed lines.

4 Approach

This section presents the actionable program generation and the systematic testing algorithm. Our approach generates a set $\mathcal{P}$ of actionable programs based on a given program input program P. The input programs are well-formed, closed programs (*i.e.*, they do not take inputs) which is standard in compiler testing [19, 35].

[5] https://lldb.llvm.org/lldb-gdb.html lists some common commands.

Section 4.1 describes a pre-processing process. Section 4.2 tackles the data value problem. Section 4.3 proposes a method to address the code location problem. Section 4.4 provides the actionable program generation algorithm and describes the overall framework to test debug information.

4.1 Pre-Processing

The goal of the pre-processing is to build necessary data structures used in our main algorithm. Given a well-formed test program P, we generate the line coverage information $\mathcal{S}$. For each covered line location s, we build a set $\mathcal{V}_s$ consisting of all in-scope variables in P at location s.

To collect the line coverage information, we instrument the program and perform a dynamic analysis. This coverage analysis is supported by compilers and popular program profilers. According to the line coverage information, we build a set $\mathcal{S}$ such that every element in $\mathcal{S}$ is a line covered in the program P. To construct the variable set $\mathcal{V}_s$, we traverse the abstract syntax tree (AST) of program P. For each line location s, we collect all global variables and local variables in the scope based on the location s on the AST. The coverage information set $\mathcal{S}$ and line augmentation (section 4.3) address the code location problem. Finally, our algorithm utilizes the variable sets $\mathcal{V}_s$ to generate actionable programs.

4.2 Shadow Validation

The naïve approach to generate all actionable programs for a single program location s is to enumerate all available variables at s. However, inspecting arbitrary variables at a specific program point can introduce undefined behaviors (data value problem). We propose a *shadow validation* technique to address the data value problem. Specifically, given a well-formed test program P, a line location s and a variable v, the shadow validation determines whether inspecting variable v at location s is well-formed.

The basic idea of shadow validation is to depict the value-printing operation on program P in the debugging session as a value-printing statement in a modified program P_1. P_1 is almost identical to P except for the print statement. We say that P_1 is the *shadow* program of P. Therefore, if P_1 is a well-formed program, the value-printing operation on program P is well-formed as well. Specifically, at the location s in program P, we insert a new print statement "`print(v)`" to produce P_1. We instrument the shadow program P_1 and perform dynamic analysis to detect whether P_1 yields undefined behaviors. Based on the dynamic analysis, the shadow validation procedure determines whether inspecting the variable v at location s in P is well-formed. We give an example to illustrate the shadow program construction.

Example 4.1. Consider the well-formed program P in Figure 1(a). Suppose we would like to print the value of variable `c` at the beginning of line 5. Figure 5(a) presents the constructed shadow program. The program is in C, and thus we use `printf()` as the print statement. The shadow program appends the print statement for variable `c` before line 5. Note that the shadow program P_1 involves printing an uninitialized variable `c`. Therefore, the dynamic analysis on the shadow program P_1 detects the error and rejects P_1, *i.e.*, P_1 is a non-actionable program.

```
1  char a,b;
2  int main(){
3    unsigned c;
4    --b;
5    printf("%d", c);
6    c=2;
7    if(a)
8      b=0;
9    return 0;
10 }
```

(a) Shadow program P_1.

```
1  char a,b;
2  int main(){
3    unsigned c;
4    --b;
5    opt_me_not(); // b
6    c=2;
7    if(a)
8      b=0;
9    return 0;
10 }
```

(b) Result program P_2.

Figure 5. Shadow validation and line augmentation.

4.3 Line Augmentation

Testing debug information requires debuggers to stop at the desired locations. In the pre-processing step, we have collected the line coverage information $\mathcal{S}$, but setting a breakpoint at a covered line $s \in \mathcal{S}$ does not guarantee that debuggers can stop at s in the presence of compiler optimizations. To tackle this problem, we consider two cases.

First, if the debugger is able to stop at line s, we can just add the meta information of the variable v to the corresponding line as a comment. The problem is to figure out all lines that the debugger is able to stop at. A naïve approach is to try all lines and see whether the debugger is able to stop or not, but that is time-consuming. Instead, we leverage the line number table in the DWARF information, which contains the mapping between the source code and the memory addresses in the executable program. DWARF compresses this data by encoding it as a sequence of instructions called a line number program, which is interpreted by a simple finite state machine to recreate the complete line number table [12]. Thus, we can use this information to find out lines that the debugger is able to stop at. Although we are testing the DWARF information itself, our main concern is the variable values in the DWARF information, and we assume that the line number information is correct.

Second, if the debugger cannot stop at line s, we propose the line augmentation technique to solve this code location problem. Given a well-formed program P as well as a line s and the specified variable v, the line augmentation procedure produces a modified program P_2. Under the condition that the line s is executed in P, the modified program P_2 guarantees that debuggers can stop at line s in the optimized machine executable. To achieve this, the line augmentation procedure inserts an unoptimizable function call (*i.e.*, `opt_me_not()`) before line s. In addition, it also appends variable information

```
Procedure ActionableProgramGeneration($P$).
Input  : A well-formed test program $P$;
Output : a set of programs $\mathcal{P}$.
1  Run pre-processing on test program $P$
2  Let $\mathcal{S}$ be the set of covered lines in $P$
3  $\mathcal{P} \leftarrow \emptyset$
4  foreach $s \in \mathcal{S}$ do
5      Let $\mathcal{V}_s$ be the set of all in-scope variables at line $s$ in $P$
6      foreach $v \in \mathcal{V}_s$ do
7          $ret \leftarrow$ ShadowValidation($P, s, v$)
8          if $ret$ is not null then
9              if $s$ has not been optimized out then
10                 $P' \leftarrow P$ and the meta information
                     indicating $s$ and $v$
11             else
12                 $P' \leftarrow$ LineAugmentation($P, s, v$)
13             $\mathcal{P} \leftarrow \mathcal{P} \cup \{P'\}$
14 return $\mathcal{P}$
```

on v as meta information such as comments. This meta information specifies which variable to inspect for debuggers. An unoptimizable function creates a barrier at the location for compiler optimization. Debuggers can stop at the barrier in the optimized program which is exactly our desired location.

To create an unoptimizable function, we prevent compilers from performing interprocedural optimization by limiting the knowledge of the function to compilers. There are both static and dynamic ways to achieve this goal. For instance, we can put the implementation of `opt_me_not()` into another compilation unit, and link `opt_me_not()` with the program P statically. We can also build `opt_me_not()` in a shared library such that it can only be linked dynamically. Therefore, compilers have no knowledge about `opt_me_not()` when processing P and the function becomes unoptimizable. We provide an example to illustrate line augmentation.

Example 4.2. Consider the program P in Figure 1(a). We perform line augmentation for line 5 with variable `b`. We insert an `opt_me_not()` function before line 5 and produce an new program P_2. Function `opt_me_not()` is an external function defined in another compilation unit. Compilers have no knowledge about `opt_me_not()` when optimizing program P_2, and thus cannot eliminate the function call. We also add extra meta information in a comment to specify that debuggers can inspect variable `b`. Figure 5(b) gives the result program P_2 after line augmentation.

4.4 Main Algorithm

In this section, we describe our main algorithms for actionable program generation and systematic testing.

```
Algorithm 1: Systematic testing algorithm.
Input : A well-formed test program $P_w$;
$\mathcal{P} \leftarrow$ ActionableProgramGeneration($P_w$)
foreach $P_{\langle s,v \rangle} \in \mathcal{P}$ do
    $A_{\langle s,v \rangle} \leftarrow$ ExtractActions($P_{\langle s,v \rangle}$)
    $P' \leftarrow$ compile $P_{\langle s,v \rangle}$ with optimization and "-g"
    if $DBG(P', A_{\langle s,v \rangle})$ is not null and
      $DBG(P', A_{\langle s,v \rangle}) \neq PRINT(P, s, v)$ then
        Report a bug with program $P_{\langle s,v \rangle}$, action $A_{\langle s,v \rangle}$
          and the used optimization flag
```

Actionable Program Generation. The actionable program generation procedure creates a set $\mathcal{P}$ of actionable programs from a well-formed test program P. Procedure ActionableProgramGeneration describes the process.

- *Actionable program candidate enumeration.* The goal of this step is to enumerate all possible candidate pairs (s, v) for actionable program generation. We utilize the coverage information $\mathcal{S}$ and variable sets $\mathcal{V}_s$ from the pre-processing step. Lines 4-6 enumerate all possible covered lines $s \in \mathcal{S}$ paired with a variable in its corresponding in-scope variable set $\mathcal{V}_s$. Each pair is a candidate choice to form an actionable program. Later steps decide whether the candidate actionable programs formed from the pairs conform to our actionable program criterion.
- *Shadow validation.* For each choice of pair (s, v), we first perform the shadow validation in line 7. If printing variable v at program point s in a debugging session involves any undefined behaviors, the shadow validation procedure returns a `null` value. Otherwise, the pair passes the shadow validation.
- *Line augmentation.* If the previous shadow validation procedure returns a non-`null` value, we consider two cases. If the line s has not been optimized out according to the DWARF line number table, we just need to add the meta information of the variable v to the line s in the source code as a comment. Otherwise, we perform the line augmentation procedure in line 12. The modified program returned from the line augmentation procedure is a proper actionable program $P_{\langle s,v \rangle}$. We add the new generated actionable program to our result set $\mathcal{P}$ in line 13.

Here we provide a concrete example to illustrate the process of actionable program generation.

Example 4.3. Consider the program in Figure 1(a). To generate actionable programs, there are many candidate pairs of line location s and variable v. We present two candidates described in Figure 1(c) and 1(d). The first program represents the pair of the location line 5 and variable `b`. Because inserting a print statement for variable `b` in the program is well-formed, the shadow validation returns a non-`null` value (line 7). Because line 5 is optimized out, line augmentation encodes the meta information and generates a new program P'

(line 9-10). Example 4.2 further describes the line augmentation. Figure 1(c) gives the resulting actionable program. The other program we present in Figure 1(d) is not an actionable program. It represents the pair of line location 5 and variable c. As shown in Example 4.1, the shadow program involves printing uninitialized variable c, so the shadow validation returns a null value. Procedure ActionableProgramGeneration discards this candidate program (line 8).

Systematic Testing. Algorithm 1 describes our testing framework. In line 1, we collect all the actionable programs generated from a well-formed test program P_w. For each actionable program, we extract the meta information and encode it as the corresponding debugger actions (line 3). If the line s has not been optimized out, the meta information includes the variable name in the comment in the corresponding line. Otherwise, the meta information includes the location of the opt_me_not() function call, and the variable name in the comment. According to the information, we can generate corresponding debugger actions including breakpoint setup and variable inspection. After compiling the program in optimized (P') version (line 4), we feed the actions to the debugger to inspect the variable v at the line s, and compare the output with the reference value $PRINT(P, s, v)$. The value $PRINT(P, s, v)$ is obtained by inserting print statements to the source code, compiling it without optimization, and running it. We accept that debuggers can report the value of the variable that has been optimized out in P'. In this case, we regard the debugger returns a null value. If debugger returns a non-null value for program P', we compare it with the reference value (line 5). The discrepancy between values in $PRINT(P, s, v)$ and $DBG(P', A_{\langle s,v\rangle})$ may indicate a bug in compiler generated debug information (line 6).

5 Experiments

We have applied our framework to two mainstream optimizing C compilers (*i.e.*, GCC and LLVM). We have also been extensively experimenting with our technique and reporting bugs to the open source community.[6]

We summarize some highlights from our experiments.

- Our framework is extremely effective, and it has led to 47 confirmed bug reports in GCC and LLVM. These bugs have been actively addressed by developers. Developers have already fixed 11 out of 47 reported bugs.
- Despite three decades of research and development in debugging optimized code, there is plenty of room to improve the debug information generation in production optimizing C compilers. The bugs discovered in our experimental study affect all major releases of GCC and LLVM at all optimization levels. Some issues are latent. For instance, one of the GCC bugs affects GCC-4.4 which was released about nine years ago.

```
break 5
run
print a
kill
quit
```

(a) Action script.

```
$ gcc -O2 -g a.c opt_me_not.c
$ gdb -x cmds -batch a.out
Breakpoint 1 at 0x4f4: file a.c, line 5.

Breakpoint 1, main () at a.c:5
5         opt_me_not(); // a
$1 = 1
Kill the program being debugged? (y or n)
```

(b) GDB output.

Figure 6. Debugging with an action script (*i.e.*, cmds) on an actionable program $P_{\langle 5,a\rangle}$ (*i.e.*, a.c). In this example, GDB reads a pre-defined value-printing sequences $A_{\langle 5,a\rangle}$ in Figure 6(a) and outputs "a = 1" in Figure 6(b).

- Our initial effort has been well received by the LLVM, LLDB, GCC, and GDB developers as well as the DWARF standards committee. One LLVM developer mentions, "Thanks for the PR, this was likely a serious quality-of-debug-experience issue for anyone hit by it."[7] One DWARF committee member comments,

  > [Your bug reports] seem clear and reasonably complete. Generating correct debug info for optimized code is important.

5.1 Experimental Setup

Our primary focus is testing the debug information for optimized C code, so we choose two most popular C compilers (*i.e.*, GCC and LLVM). Our experiments were conducted on the latest development versions of both compilers. We use the compiler flag "-g" to generate the DWARF debug information. To produce optimized code, we use optimization flags "-O1", "-O2", and "-O3" in both compilers. GCC has an additional flag "-Og" which provides better debugging support in optimized code. We have tested that flag as well.

To interpret the debug information, we utilize the corresponding source code debuggers, *i.e.*, GDB for GCC and LLDB for LLVM. Both GDB and LLDB used in our experiments are of the latest development versions. All experiments were conducted on a 64-bit machine with an Xeon Silver 4110 processor and 16 GB memory, running Ubuntu 18.04. By default, the two compilers generate 64-bit executables.

Implementation. We describe the components in our framework as follows.

- We implemented the source code transformation described in Shadow Validation (Section 4.2) with Clang's Libtooling library [27]. Given a location s and a variable v, our tool automatically translates the input program P and constructs the corresponding shadow program. The dynamic analysis is crucial in our testing process. To avoid undefined behaviors, we leverage both CompCert's reference interpreter [21] and Clang's undefined behavior sanitizer [28]. CompCert is a verified compiler and its interpreter has been widely-used

[6] Project website: https://www.cc.gatech.edu/~qrzhang/projects/debug/debug.html.

[7] https://bugs.llvm.org/show_bug.cgi?id=39896.

in compiler testing to detect undefined behaviors in input programs [19, 36]. Clang's sanitizer offers runtime monitoring to detect undefined behaviors. It has been used not only in compiler testing [19, 36] but also in other domains such as software debloating [15].

- We implemented the Line Augmentation (Section 4.3) using Clang's Libtooling as well. In particular, we leverage Clang's `RecursiveAstVisitor` to traverse the program AST and build the data structures of the available variables in each scope. We describe the body of `opt_me_not()` in an external file and link it statically with the testing program *P*. In practice, this approach is usually sufficient to prevent compiler from optimizing out the call to function `opt_me_not()` in *P*. We can also insert "`__asm__ volatile ("" : : : "memory");`" in function `opt_me_not()` to further prevent any optimizations in C compilers.
- The systematic testing algorithm is implemented using BASH scripts. Both GDB and LLDB provide the feature to read the action commands from a pre-defined file. Therefore, rather than invoking the debuggers interactively, we pass the file that contains our action sequence to the debuggers. If the debugger can stop at one line multiple times, we only consider the first stop. Figure 6 gives an example action sequence file to inspect the program $P_{\langle 5,a\rangle}$. The program is compiled with "`-g -O2`" and GDB outputs "`a = 1`", *i.e.*, we have $GDB(P'_{\langle 5,a\rangle}, A_{\langle 5,a\rangle}) = 1$.

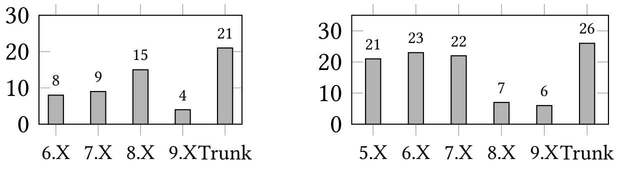
Input Programs. Our actionable program transformation takes as inputs some well-formed seed programs *P*. We pick C testing programs from the unit test-suites of many open-source projects such as GCC, LLVM, CompCert [21], Frama-C [29], and the Rose compiler [30]. Those programs are usually well-formed and have been widely-used in compiler testing work [19, 36]. We also use Csmith [35] – a random C program generator – to produce input programs. Csmith adopts a set of heuristics to avoid generating test programs with undefined behaviors. Furthermore, we check all input programs with CompCert's reference interpreter and Clang's sanitizer to make sure that they are well-formed.

Test Case Reduction. In compiler testing, it is important to reduce test cases before reporting. In particular, test case reduction constructs a small input program that triggers the same compiler bug. Small and simplified test cases ultimately help developers locate the root causes. In our work, we utilize C-reduce [25] for test case reduction. To preserve the well-formedness of the test programs, we adopt the dynamic analysis described in our implementation. The reduction process is fully automated. For each reduced test case, we manually inspect the program before reporting. Our reduction process is quite effective and we haven't reported any ill-formed test cases at the time of writing.

Table 1. Overview of bugs reported for trunk versions of GCC and Clang.

Compiler	Summary			
	Reported	Fixed	Duplicate	Invalid
GCC	21	4	3	2
Clang	26	7	0	1

Compiler	Classification	
	Crash	Wrong Output
GCC	0	21
Clang	3	23



(a) Affected GCC versions.

(b) Affected LLVM versions.

Figure 7. Summary of compiler versions that generate wrong debug information.

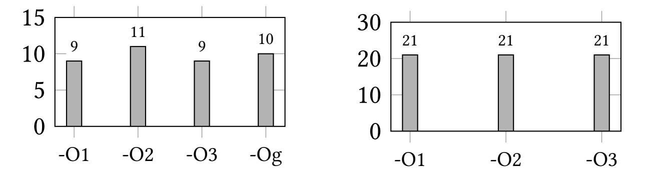
5.2 Quantitative Results

Overall Results. Table 1 summarizes all bugs we have found. In particular, we have reported 47 bugs to both GCC and LLVM developers, and we classify them into the following two types.

- *Crash.* The debugger crashes when applying the value-printing sequence to the input program. Typically, the crash is due to some assertion failures. We haven't encountered any crash related to segmentation fault.
- *Wrong Output.* The debugger successfully performs the value-printing tasks on optimized test program P' and exits normally. However, the value is different from the ground truth derived from the unoptimized counterpart *P*, *i.e.*, $DBG(P', A_{\langle s,v\rangle}) \neq PRINT(P, s, v)$. Note that it is an expected behavior for some variables in an optimized program P' to be optimized out.

From Table 1, we can see that all GCC bugs and most LLVM bugs are related to generating wrong debug information. In practice, wrong debug information directly manifests in the debugging process when a debugger inspects variable values. Our testing exposes three crashes in the LLDB debugger when parsing DWARF information. All of our bug reports have been confirmed by the developers. The bug counts reflect our recent efforts in testing debug information. Developers have fixed about 23% of the reported bugs. It shows a strong community interest in improving debugging support in optimizing compilers.

Effectiveness of Line Augmentation. Line augmentation inserts the barrier `opt_me_not()` to cause the compiler to generate a line table entry where the debugger can place a breakpoint. It can potentially increase the number of test



(a)
Affected GCC optimization levels.

(b)
Affected LLVM optimization levels.

Figure 8. Summary of compiler optimization levels generating wrong debug information. In LLVM, the "`-Og`" flag is currently a wrapper of "`-O1`".

cases that we can utilize. Among the 21 reported GCC bugs, 13 of them used line augmentation. Among the 26 reported Clang bugs, 16 of them used line augmentation. Without this technique, the debuggers may not be able to stop at those lines, which means those bugs may not be found.

Affected Compiler Versions. Figure 7 illustrates the compiler versions related to our reported bugs. We chose several recent stable releases of both GCC and LLVM because they represent the recent developments in production C compilers. GCC-6 was released approximately four years ago and LLVM-5 two and a half years ago. We also picked the trunk versions of both compilers. From Figure 7, we can see that all reported bugs affect the latest development version of the two compilers. Many of them affect the latest stable releases. Some bugs were reported before the release dates of GCC-9.X, LLVM-8.X and LLVM-9.X, so they don't affect those stable releases, and this is the reason why the numbers of these 3 versions are significantly smaller than the number of previous versions. We also want to mention that some of the issues have been present for a long time. Specifically, one of the fixed GCC bugs affects GCC-4.4 which was released about nine years ago.

Affected Optimization Levels. Compilers generate wrong debug information at all optimization levels as shown in Figure 8. In particular, Figure 8(a) and Figure 8(b) depict the optimization levels in GCC and LLVM, respectively. GCC has a designated flag "`-Og`" to improve debugging experience. Unfortunately, this flag still introduces many bugs in our GCC experiment. From Figure 8(a), we can notice that the GCC bugs are scattered in different levels. In LLVM, Figure 8(b) suggests that almost all the bugs we found affect optimization levels from "`-O1`" to "`-O3`". This is related to the fact that debug values are sometimes incorrectly updated by passes and subsequent passes don't touch them. Consequently, latter passes do not erase the result of the wrong optimization (by marking the value as unavailable). The empirical result shows that many bugs are, in fact, due to optimizations mistakenly not deleting obsolete debug info. It suggests that newer versions of a compiler could occasionally miss the correct debugging information from earlier versions.

Table 2. LLVM components exposing debugger bugs.

Bug Counts	LLVM Component
6	DWARF Parser
5	SelectionDAG
2	SROA
2	InstCombine
2	EarlyCSE
2	MachineScheduler
1	LoopDeletion
1	SimplifyCFG
1	LSR
1	DeadStoreElimination
1	Inliner
1	CodegenPrepare

Affected Optimization Passes in LLVM LLVM provides a "`-opt-bisect`" flag to stop at some point in the optimization pipeline, not applying subsequent transformations to the input IR. Therefore, we could bisect the pass responsible for updating the wrong debug information. Developers also provided some information about the optimization passes. We have identified the passes associated with all reported LLVM bugs in Table 2. We can also observe that the DWARF parser contains the most bugs we found. Among them, three bugs are LLDB crashes as shown in Table 1. The remaining three bugs cause LLDB to print a wrong value even though the debug information is correct. This is because the parser interprets DWARF incorrectly. In fact, the DWARF specification is a little vague. A developer filed a ticket to clarify some interpretation. The remaining bugs are spread across many different optimization passes, We notice that `SROA`, `InstCombine`, and `SelectionDAG` contain relatively more bugs, probably due to their complexity.

5.3 Case Studies on Sample Bugs

We select and discuss six reported GCC and LLVM bugs. Figure 9 describes the corresponding test programs with bug classifications and status. Despite generating the wrong debug information, compilers produce correct machine executables for all six programs, *i.e.*, we can add a `printf` statement to observe the correct value at runtime.

Five out of these six bugs have been fixed by the developers. Our work focuses on identifying incorrect debugger outputs according to Definition 3.2. For those wrong value bugs, the developers fixed them either by providing the correct debug information (Figure 9(e)) or by noting that the variables have been optimized out (Figures 9(b), 9(c), and 9(f)). We believe developers will generally only delete debug information when it is a good idea to do that. For example, in 9(f), there is a function in LLVM called `salvageDebugInfo` which tries to construct the debug information when an instruction is eliminated. This function is a best-effort function, so it cannot always succeed. Marking the value as optimized out is a reasonable choice because the compiler cannot always make an informative call on whether the value will be correct or

```c
1  static int a[6];
2  int b;
3  static void c () {
4    b = 5 ^ a[b & 5];
5  }
6  int main() {
7    int i, d = 0;
8    c();
9    i = 0;
10   for (; i < 8; i++) {
11   c();
12   if(d)printf("%d",i);//i
13   }
14 }
```

(a) GCC wrong value bug 92387 (new).

```c
1  volatile int a, b;
2  static int c;
3  int main() {
4    int i;
5    b = 0;
6    i = 0;
7    for (; i < 3; i++)
8      a = c;
9    i = 0;
10   for (; i < 5; i++)
11     ;
12   optimize_me_not();
13 }
```

(b) GCC wrong value bug 89792 (fixed).

```c
1  int a[7][8];
2  int main() {
3    int b, j;
4    b = 0;
5    for (; b < 7; b++) {
6      j = 0;
7      for (; j < 8; j++)
8        a[b][j] = 0;
9    }
10   optimize_me_not();
11 }
```

(c) GCC wrong value bug 90716 (fixed).

```c
1  char a;
2  int main() {
3    int i;
4    int b[] = {7, 7, 7};
5    a = b[2];
6    i = 0;
7    for (; i < 5; i++)
8      ;
9    opt_me_not();
10 }
```

(d) LLDB crash bug 40827 (fixed).

```c
1  char a, b;
2  int main() {
3    --b;
4    unsigned l_801 = 36901;
5    ++l_801;
6    if (a) // l_801
7      b = 0;
8  }
```

(e) LLVM wrong value bug 43957 (fixed).

```c
1  char a = 25;
2  short b() {
3    short i = 23680;
4    i = a;
5    opt_me_not(); // i
6    return a;
7  }
8  int main() { b(); }
```

(f) LLVM wrong value bug 39874 (fixed).

Figure 9. Sample test programs that trigger debug information bugs in LLDB, GCC and LLVM. All wrong value bugs can only be observed in debuggers. The binary is compiled with "-g" and optimization flags.

not. Our work does not focus on testing the *completeness* of debug information, *i.e.*, if the debugger displays a variable as "optimized out", could the variable still be in the optimized executable? Identifying the "missing" debug information in optimized code is an exciting future topic.

Figure 9(a). This example exposes a GCC bug that affects GCC trunk. The bug manifests when the source is compiled with "`-O1`". This shows up as a wrong value printed for the variable `i`. The correct value of `i` is 0 at line 12. However, with "-g", GDB prints "`i = 1`".

Figure 9(b). This GCC bug manifests when the program is compiled with "`-O3`". It affects GCC-8 and GCC trunk, and GCC-7 works fine. Printing the value of the variable `i` breaking on line 12 yields the wrong value 0 instead of 5.

```rust
1  fn main() {
2    let mut counter = 0;
3    {
4      let mut tick = || counter += 1;
5      tick();
6      tick();
7      lib::opt_me_not();
8    }
9    assert_eq!(counter, 2);
10 }
```

(a) Program $P_{\langle 7, counter\rangle}$.

```
$ rustc --crate-type=lib opt_me_not.rs -g -C opt-level=1
$ rustc main.rs --extern lib=libopt_me_not.rlib -g -C opt
    -level=1
$ rust-gdb ./main
(gdb) b main.rs:7
Breakpoint 1: file main.rs, line 7.
(gdb) r
Breakpoint 1, at main.rs:7
7           lib::opt_me_not();
(gdb) p counter
$1 = 0
```

(b) Wrong output at "`-C opt-level=1`".

Figure 10. Rust compiler generates wrong debug information with "`opt-level=1`".

Figure 9(c). This GCC bug manifests when the program is compiled with "`-O2`". When the bug was reported, it was a recent regression because it affected GCC trunk and GCC-8 worked fine. Printing the value of `j` breaking on line 10 yields the wrong value 0 instead of 8. The developer confirmed that loop-distribution performs manual removal of the old loop body but it moves debug statements in bogus order.

Figure 9(d). This is an LLDB (debugger bug) which shows up when building with "`-O2`". This is a crash in the DWARF parser when trying to parse DI (debug info) variables. It causes an invalid memory read while trying to copy data from the process being debugged. The bug does not reproduce when the source code is compiled with "`-O0`". LLVM-3.8 and LLDB trunk are affected.

Figure 9(e). This example shows a Clang bug that shows up when building the source with "`-O3`". The correct value of `l_801` is 36902, but LLDB prints the wrong value 36901. Clang trunk is affected.

Figure 9(f). This example shows a Clang bug that reproduces when the source code is compiled with "`-O1`". Trying to print the variable `i` at line 5 yields the wrong value 23680 (*i.e.*, the previous assignment has been missed). This is due to a bug in the `EarlyCSE` pass which performs a round of deletion of trivially dead instructions but does not mark the value as an unavailable DWARF expression. This bug affected Clang-4 to Clang-7.

6 Discussion

This section summarizes the lessons learned from our efforts on testing debug information for optimized code.

6.1 Generality of Program Transformation

We have presented the design and implementation of the actionable program transformation for testing debug information for C compilers, but our framework is general. We have also applied our framework to the Rust compiler. In three days, we have found two confirmed bugs in the Rust compiler. Figure 10(a) gives one of the Rust bugs. In this actionable program $P_{\langle 7, counter \rangle}$, `tick` is a closure to increment the integer `counter`. Because the program calls `tick` twice, the expected value of `counter` at line 7 is 2. The program eventually passes the assertion at line 9. When the program is compiled without optimization, the Rust debugger can report the value correctly. However, when compiled at "`opt-level=1`" which is equivalent to "`-O1`" in LLVM, the Rust debugger reports a wrong value 0 shown in Figure 10(b).

To generalize our framework to Rust, our framework needs to adjust the program instrumentation based on the differences between the grammars. It is worth noting that Rust compiler does not allow undefined behaviors in its safe code section, and thus it can help detect undefined behaviors for our shadow validation procedure. In the line augmentation procedure, we notice that the compilation unit for Rust is crate.[8] We put the implementation of the `opt_me_not()` function into another crate to guarantee that it is unoptimizable. We are unaware of any random program generator for Rust, so we use only the programs from the official Rust test-suite to generate actionable programs.

6.2 Debugger Correctness

Source code debugging relies on both compilers and debuggers. Debuggers read the debug information generated by compilers and perform the actual debugging activity. Our current work only focuses on testing the information generated by compilers. Even with the correct debug information, a debugger can produce wrong outputs. For instance, all three LLDB crashes (Table 1) are associated the optimized code. In these cases, Clang generates the correct debug information because GDB behaves as expected. Validating debuggers for optimized code is an interesting future direction. The recent work by Lehmann and Pradel [20] is a promising technique based on differential testing.

6.3 DWARF Specification

It is interesting to note that one of our reported LLVM bugs eventually led to a ticket submitted to the DWARF standard committee. Our LLVM bug report reveals that LLDB prints incorrect value for some elements of an array which have been optimized out.[9] The DWARF expression emitted for the array contains "`DW_OP_piece: 6`", which indicates that the first 6 bytes of the array have an empty location and have to be printed as such. The DWARF standard illustrates it via an example. However, the main description of "`DW_OP_piece`" in the DWARF standard caused the misunderstandings. Therefore, LLDB has implemented it in an expected way and thus outputs a wrong value. The developer who originally implemented the support in LLDB submitted a comment to the DWARF standard committee to clarify the "`DW_OP_piece`" documentation for parts of values optimized out. Figure 11(a) and 11(b) give our test program and the DWARF expression that triggered the issue, respectively.

```
short a, c;
long b;
short fun1() { return c; }
char d() {
  short l_30[4] = {1};
  a = fun1();
  l_30[3]++;
  opt_me_not(); //break here.
  return b;
}
int main() { d(); }
```

(a) Actionable Program $P_{\langle 8, l\_30 \rangle}$.

```
Begin       End       Expression
0x4004d1    0x4004dd (DW_OP_piece: 6; DW_OP_lit0; DW_OP_stack_value; DW_OP_piece:
       2)
0x4004dd    0x4004e5 (DW_OP_piece: 6; DW_OP_lit0; DW_OP_plus_uconst: 1;
       DW_OP_stack_value; DW_OP_piece: 2)
```

(b) DWARF expression related to array `l_30`.

Figure 11. LLDB bug 39869 related to an issue in the DWARF standard. In the unoptimized code, LLDB prints "`l_30 = ([0] = 1, [1] = 0, [2] = 0, [3] = 1)`". However, when compiled with "`-g -O1`", LLDB incorrectly prints "`l_30 = ([0] = 0, [1] = 0, [2] = 0, [3] = 1)`".

6.4 Avoiding Duplicate Bug Reports

In our testing process, we tried various techniques to avoid reporting duplicates. For every bug, we build a signature based on the affected compiler versions. Before reporting, we compare the signature against all previous signatures to avoid duplicates. Moreover, for all GCC bugs, we performed a repository bisection to locate the culprit revision. For LLVM, we leveraged the pass bisection feature to locate the culprit pass. Based on Table 1, we can see that we have reported three duplicate GCC bugs but every reported LLVM bug is unique. Our findings suggest that the pass bisection is more helpful to eliminate duplicates than repository bisection.

6.5 Threats to Validity

The Shadow Validation procedure (Section 4.2) is based on dynamic analysis, which may not be perfect in practice. CompCert supports only a subset of C. Clang's sanitizer has a few target- and OS-specific limitations described in its own documentation. In our experiments, we rarely encountered the issue with undefined behaviors. Our experience suggests that in practice, CompCert and Clang's sanitizers are sufficient to guarantee the well-formedness of actionable programs.

[8] https://doc.rust-lang.org/rust-by-example/crates.html

[9] https://bugs.llvm.org/show_bug.cgi?id=39869.

Moreover, the random input programs used in our work are generated by Csmith [35]. Csmith has applied various techniques to generate undefined-behavior-free C programs.

The Line Augmentation procedure (Section 4.3) inserts an unoptimizable function call (*i.e.*, `opt_me_not()`) for a location that has already been optimized out. It may inevitably inhibit some optimizations. However, without such a barrier, the particular location could not be inspected by any debuggers in the optimized code. For the non-optimized-out lines, we obtain them by reading the line number table in the DWARF information. There are two potential threats in this process. First, we are making the assumption that the line number information in the DWARF information is always correct, but this may not be true. Second, we have observed that some debuggers can stop at a line which is not in the DWARF line number table. Intuitively, it is acceptable for debuggers to use some other information from the executable binary to decide additional lines to stop. It is an interesting future direction to develop a sound and complete approach to determine all stoppable lines in the executable binary.

7 Related Work

Debugging optimized code has been a long studied problem. Hennessy [14] proposes techniques to recover the values of valid variables in optimized code by reconstructing the original assignments of variables. To avoid misleading developers when they debug optimized code, many frameworks have been proposed to detect [1, 10, 33] and recover [18, 34] variables that are assigned with different values from the unoptimized version. Code location mapping techniques allow programmers to set breakpoints at different line locations [6]. However, this binary to source location mapping is often fragile, sometimes inaccurate or misleading. Tice and Graham [31] propose a framework to create a slightly modified and semantically equivalent version of the source code to facilitate the mapping. In the work of Hölzle et al. [16], it hides the optimization from users and presents information users expect to see in an unoptimized program. Researchers also propose ways to check discrepancies between the optimized and unoptimized programs to avoid debugging in optimized code [17]. The work of Bastian et al. [3] describes two techniques to perform validation and synthesis of the DWARF stack unwinding tables, and their implementation for the x86_64 architecture.

Debugger is a widely used developer tool. Various debuggers have been developed for different programming languages [2]. To ensure the correctness of debuggers, their semantics have been carefully defined [4]. Recently its automated testing has been studied by Lehmann and Pradel [20]. They propose a new framework to generate feedback-directed tests for differential testing [22] on debuggers. For the same program, different behaviors between two equivalent debuggers under same actions indicate bugs in at least one debugger. The work of Tolksdorf et al. [32] presents a metamorphic testing approach for debuggers. However, those works address the testing of functional correctness of debuggers themselves. Our main focus is testing debug information fed to the debuggers for optimized code. To the best of our knowledge, our paper is the first work toward validating the debug information generated by compilers.

Automated Testing for compilers has been studied for many years. A common approach for compiler testing is random test generation. Csmith is currently the state-of-the-art program generator for C compiler testing [8, 35]. Another approach to test compilers is mutation based testing. The work of Le et al. [19] proposes the equivalence modulo inputs (EMI) technique for compiler testing. Testing tools adopting EMI principles mutate programs on unexecuted branches and expect the same result as before. Other approaches include enumerating all testing programs of a given template within bounded size [36]. Recently, deep learning has also been applied in program generation for compiler testing [11]. A survey for compiler test case generators is provided by [5]. All of these techniques emphasize on generating programs as inputs for compilers. Our work focuses on generating breakpoints and valid variables for debuggers.

8 Conclusion

This paper has introduced the first testing framework for debug information validation for optimized code. We have presented a novel concept of actionable programs to facilitate the subsequent systematic testing process. We have evaluated our framework on two well-known C compilers and the Rust compiler. The results are extremely encouraging. Our technique has led to 47 confirmed bug reports in GCC and LLVM, 11 of which have already been fixed. Our work has been well-received by the open-source community. Generating correct debug information for optimized code is inherently difficult. We hope our work motivates more researchers to look into this important area.

Acknowledgments

We would like to thank our shepherd and the anonymous PLDI reviewers for valuable feedback on earlier drafts of this paper, which helped improve its presentation. We thank Michael Eager and Vivek Sarkar for helpful comments on this work. We also appreciate the GCC and LLVM developers' help for analyzing and fixing our reported bugs. Our evaluation benefited significantly from Csmith [35] and C-Reduce [25] tools. This research was supported in part by the United States National Science Foundation (NSF) under Grant No. 1917924. 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] Ali-Reza Adl-Tabatabai and Thomas R. Gross. 1996. Source-Level Debugging of Scalar Optimized Code. In *Proceedings of the ACM SIGPLAN'96 Conference on Programming Language Design and Implementation (PLDI' 96)*. 33–43.

[2] Sanjeev Kumar Aggarwal and M. Sarath Kumar. 2002. Debuggers for Programming Languages. In *The Compiler Design Handbook*. CRC Press, 297–329.

[3] Théophile Bastian, Stephen Kell, and Francesco Zappa Nardelli. 2019. Reliable and Fast DWARF-Based Stack Unwinding. *PACMPL* 3, OOPSLA (2019), 146:1–146:24.

[4] Karen L. Bernstein and Eugene W. Stark. 1995. Operational Semantics of a Focusing Debugger. *Electr. Notes Theor. Comput. Sci.* 1 (1995), 13–31.

[5] Abdulazeez S. Boujarwah and Kassem Saleh. 1997. Compiler Test Case Generation Methods: A Survey and Assessment. *Information & Software Technology* 39, 9 (1997), 617–625.

[6] Gary Brooks, Gilbert J. Hansen, and Steve Simmons. 1992. A New Approach to Debugging Optimized Code. In *Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation (PLDI' 92)*. 1–11.

[7] John Calcote. 2010. *Autotools: A Practioner's Guide to GNU Autoconf, Automake, and Libtool.* No Starch Press.

[8] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Z. Fern, Eric Eide, and John Regehr. 2013. Taming Compiler Fuzzers. In *Proceedings of the ACM SIGPLAN'13 Conference on Programming Language Design and Implementation, (PLDI' 13)*. 197–208.

[9] DWARF Debugging Information Format Committee. 2020. DWARF Debugging Information Format Version 5. http://dwarfstd.org/doc/DWARF5.pdf

[10] Max Copperman. 1994. Debugging Optimized Code Without Being Misled. *ACM Trans. Program. Lang. Syst.* 16, 3 (1994), 387–427.

[11] Chris Cummins, Pavlos Petoumenos, Alastair Murray, and Hugh Leather. 2018. Compiler Fuzzing through Deep Learning. In *Proceedings of the ACM SIGSOFT'18 International Symposium on Software Testing and Analysis, (ISSTA' 18)*. 95–105.

[12] Michael Eager. 2012. Introduction to the DWARF Debugging Format. http://www.dwarfstd.org/doc/Debugging%20using%20DWARF-2012.pdf

[13] Free Software Foundation. 2019. GNU GCC Manual. https://gcc.gnu.org/onlinedocs/gcc-8.3.0/gcc/

[14] John L. Hennessy. 1982. Symbolic Debugging of Optimized Code. *ACM Trans. Program. Lang. Syst.* 4, 3 (1982), 323–344.

[15] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. 2018. Effective Program Debloating via Reinforcement Learning. In *Proceedings of the ACM SIGSAC'18 Conference on Computer and Communications Security (CCS' 18)*. 380–394.

[16] Urs Hölzle, Craig Chambers, and David M. Ungar. 1992. Debugging Optimized Code with Dynamic Deoptimization. In *Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation (PLDI' 92)*. 32–43.

[17] Clara Jaramillo, Rajiv Gupta, and Mary Lou Soffa. 1999. Comparison Checking: An Approach to Avoid Debugging of Optimized Code. In *Proceedings of the ACM SIGSOFT'99 Symposium on the Foundations of Software Engineering (FSE' 99)*. 268–284.

[18] Clara Jaramillo, Rajiv Gupta, and Mary Lou Soffa. 2000. FULLDOC: A Full Reporting Debugger for Optimized Code. In *Proceedings of the International Symposium on Static Analysis (SAS' 00)*. 240–259.

[19] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler Validation via Equivalence Modulo Inputs. In *Proceedings of the ACM SIGPLAN'14 Conference on Programming Language Design and Implementation, (PLDI' 14)*. 216–226.

[20] Daniel Lehmann and Michael Pradel. 2018. Feedback-Directed Differential Testing of Interactive Debuggers. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE' 18)*. 610–620.

[21] Xavier Leroy. 2006. Formal Certification of a Compiler Back-end or: Programming a Compiler with a Proof Assistant. In *Proceedings of the ACM SIGPLAN-SIGACT'06 Symposium on Principles of Programming Languages (POPL' 06)*. 42–54.

[22] William M. McKeeman. 1998. Differential Testing for Software. *Digital Technical Journal* 10, 1 (1998), 100–107.

[23] Julia Menapace, Jim Kingdon, and David MacKenzie. 1992. *The "stabs" Debug Format.* Technical Report. Cygnus support.

[24] Microsoft. 2015. The PDB (Program Database) Symbol File format. https://github.com/Microsoft/microsoft-pdb.

[25] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-Case Reduction for C Compiler Bugs. In *ACM SIGPLAN'12 Conference on Programming Language Design and Implementation (PLDI' 12)*. 335–346.

[26] Vladimir Slavik. 2018. Developer Guide – An Introduction to Application Development Tools in Red Hat Enterprise Linux 7. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/developer_guide/index.

[27] The Clang Team. 2019. LibTooling. https://clang.llvm.org/docs/LibTooling.html

[28] The Clang Team. 2019. UndefinedBehaviorSanitizer. https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html

[29] The Frama-C Team. 2019. Frama-C. https://frama-c.com/

[30] The ROSE Compiler Team. 2019. Rose Compiler. http://rosecompiler.org/

[31] Caroline Tice and Susan L. Graham. 1998. OPTVIEW: A New Approach for Examining Optimized Code. In *Proceedings of the ACM SIGPLAN/SIGSOFT'98 Workshop on Program Analysis For Software Tools and Engineering (PASTE' 98)*. 19–26.

[32] Sandro Tolksdorf, Daniel Lehmann, and Michael Pradel. 2019. Interactive Metamorphic Testing of Debuggers. In *Proceedings of the ACM SIGSOFT'19 International Symposium on Software Testing and Analysis (ISSTA'19)*. 273–283.

[33] Roland Wismüller. 1994. Debugging of Globally Optimized Programs Using Data Flow Analysis. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI' 94)*. 278–289.

[34] Le-Chun Wu, Rajiv Mirani, Harish Patil, Bruce Olsen, and Wen-mei W. Hwu. 1999. A New Framework for Debugging Globally Optimized Code. In *Proceedings of the ACM SIGPLAN'99 Conference on Programming Language Design and Implementation (PLDI' 99)*. 181–191.

[35] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proceedings of the ACM SIGPLAN'11 Conference on Programming Language Design and Implementation, (PLDI' 11)*. 283–294.

[36] Qirun Zhang, Chengnian Sun, and Zhendong Su. 2017. Skeletal Program Enumeration for Rigorous Compiler Testing. In *Proceedings of the ACM SIGPLAN'17 Conference on Programming Language Design and Implementation, (PLDI' 17)*. 347–361.