Imperial College London

MEng Individual Project

Imperial College London

Department of Computing

C-Locate: Automatic Tool to Locate Miscompilation Bugs

Supervisor:
Prof. Cristian Cadar

Author:
Wenteng Zhao

Second Marker:
Prof. Alastair Donaldson

June 18, 2023

Abstract

Compilers are one of the oldest and most essential software ever written. Compilers transform high-level programming language into machine-understandable code, and concurrently, enable the detection of potential errors and the optimization of the code for increased efficiency. Ensuring the reliability of compilers is crucial for all software developments. Compiler fuzzing tools and programs have been developed to find various bugs in different languages and compilers.

Diagnosing a compiler bug may be as challenging as identifying them in the first place. This is particularly true for miscompilation bugs, which compilers would successfully compile but silently generate wrong codes and produce an unexpected result. Many compilers provide compiling or debugging options to help compiler engineers diagnose bugs, but using these options needs much human effort.

This project presents C-Locate, a tool which aims to automatically locate and narrow down miscompilation compiler bugs in a large program that requires minimum human effort. This tool is mainly built for LLVM Clang, but the ideas can be extended to other compilers and programming languages. C-Locate aims to, given a miscompilation bug and a large program that's affected by this bug, 1. Locate the miscompiled source file in the large program. 2. Locate the compiler optimization pass that introduces the bug 3. Locate which git commit in the compiler repository that introduces the bug 4. Locate the exact miscompiled function in the miscompiled source file. C-Locate has been tested on a known compiler bug in a real program (ZShell), and a few manually-injected compiler bugs in a few large projects (ZShell, LevelDB and SQLite). C-Locate manages to provide the correct and useful information in most cases.

Acknowledgements

I wish to express my deepest appreciation to my supervisor, Prof. Cristian Cadar, for his unwavering support and mentorship throughout this project. His broad knowledge base, keen intellectual insight, and rigorous approach to research have significantly influenced this project and overall development.

I also wish to extend my gratitude to my second marker, Prof. Alastair Donaldson, whose valuable feedback greatly enhanced the quality of this project and refined my understanding of the subject matter.

Finally, I would like to thank my family and friends, who have provided unyielding support and encouragement throughout this journey.

Contents

Chapter 1

Introduction

1.1 Motivation

The importance of compilers in computing is multifaceted. Compilers serve as a critical tool in software development, enabling developers to write in high-level languages such as Python, Java, or C/C++, thereby increasing productivity and facilitating easier debugging and maintenance. Compilers also perform code optimization, improving the efficiency of the final executable program in terms of both space and speed. Additionally, by translating to machine language, compilers allow software to become platform-specific, which enhances performance.

Since compilers play a crucial role and are the foundation of software development, ensuring the reliability of compilers is critically important. A reliable compiler should always produce expected output, which then enables software developers to create applications that operate as anticipated. However, although researchers and engineers have spent decades developing, testing and fixing compilers, there are still bugs in compilers. Indeed, there are more than hundreds of various bugs introduced, reported and fixed in compilers every day.

There are mainly two kinds of bugs in compilers, crashing bugs and miscompilation bugs. Crashing bugs are serious issues that occur when a compiler fails during the compilation process, leading to a system crash. Miscompilation bugs are when compilers successfully compile but silently generate wrong code or incorrectly translate the source code into machine code. Snippet 1.1 is a good example in C language of a miscompilation bug reported in the CSMith paper[1]. There is a great paper discussing how much compiler fuzzing matters[2] in real-world production. In this paper, there are examples of how miscompilation bugs can affect real-world large projects such as ZShell, LevelDB and SQLite, causing their fixed test suite to fail. Compared to crashing bugs, miscompilation bugs are particularly harder to detect and diagnose and potentially have a greater impact because the compiler doesn't crash or produce any error message and the produced executable programs wouldn't behave as intended due to incorrect translation.

```c
int foo (void) {
  signed char x = 1;
  unsigned char y = 255;
  return x > y;
}
```
Snippet 1.1: The version of GCC that shipped with Ubuntu Linux 8.04.1 for x86, at all optimization levels, compiles this function to return 1; the correct result is 0.

Many compiler fuzzing tools and techniques have been developed to find bugs in compilers, including AFL, CSMith, XSmith and Equivalence Module Inputs (EMI) testing. Some tools have been developed to help compiler engineer to diagnose a bug by reducing the size of test cases, like C-Reduce. However, there are very less tools that help compiler engineer to diagnose a miscompilation bug within a large program with multiple source files, which is arguably the hardest kind of compiler bug to diagnose.

1.2 Contribution

In this project, we present C-Locate, an automatic tool to help compiler engineers diagnose a miscompilation bug in a large program with multiple source files by locating the bug and reducing the problem. C-Locate is built with a focus on C language and LLVM Clang compilers, some functionalities might only work on Clang now, but the idea can be extended to other languages and compilers.

C-Locate takes a larger program, a buggy compiler and other essential information that triggers a miscompilation bug as input and locates the miscompiled source file, the optimization pass that introduced the bug, the git commit in the compiler repository that introduced the bug, and the exact function that's getting miscompiled. Since, for a miscompilation bug in a large program, compilers successfully and silently compile, compiler engineers have very less information about the bug and would need much effort to locate the bug. C-Locate provides much useful information in locating and diagnosing the bug.

Some compilers provide options to help with debugging, but using these options requires much human effort. C-Locate automates the uses of these options and systematically produces useful information for compiler engineers to diagnose miscompilation bugs which minimises the human effort needed. C-Locate uses a compiler option in LLVM C Compiler(Clang), opt-bisect, which bisects the run of optimization passes, to locate the buggy optimization pass. C-Locate also uses a set of git commands to locate the buggy git commit.

The final output of C-Locate provides the following information:

Locate the Miscompiled Source File(s)

C-Locate aims to work on programs with multiple source files. This functionality locates the miscompiled source file(s) of the program using the idea of mixed-compiling, which is using a correct version of compiler and a buggy version of compiler to compile each source file separately. Details of this idea will be discussed in later sections. Locating the miscompiled source file is very useful since it reduces the problem from multiple source files into a single source file, which largely reduces the code size that's needed to be investigated in the program and narrows down the big problem.

Locate the Buggy Optimization Pass

After locating the miscompiled source file, C-Locate bisects the optimization being run on this source file using the opt-bisect option to locate which optimization pass introduces the bug. Locating the buggy optimization pass in the buggy compiler is very useful. Knowing which optimization pass introduces the bug largely reduces the code size that's needed to be investigated in the buggy compiler which also narrows down the big problem.

Locate the Buggy Git Commit ID

Bugs are usually introduced by recent commits. This functionality of C-Locate locates which commit of the compiler repository introduced the bug. Looking at the code changes that introduce the bug is also helpful in fixing the bug and narrowing down the problem.

Locate the Miscompiled Function

There may be multiple functions in the miscompiled source file. This functionality aims to further reduce the miscompiled code size by locating which function in the source file is getting miscompiled. This is done by extracting and copying each function into a new source file. Then each function respectively by using the idea of mixed-compiling again like the first functionality. Locating the miscompiled source file is very useful since it further reduces the code size and narrows down the problem, also compiler engineer may be able to construct a much simpler test case with this function.

Chapter 2

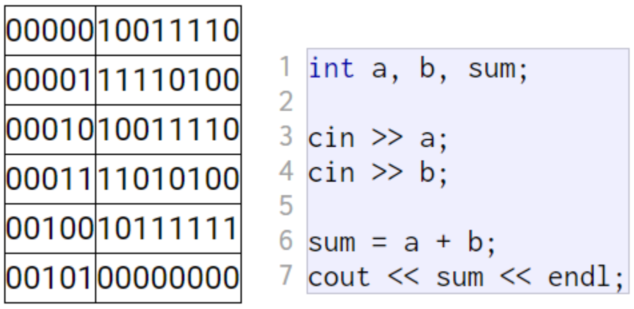
Background & Related Work

This chapter is about the definition and importance of compilers, some concepts of compiler testing and implementations of these concepts, some ideas and criteria for reporting compilation bugs, and some related works.

2.1 Compilers

In general, compilers are computer software that translates one computer language(source language) into another computer language(target language). In this project, we are using a more specific and primary definition of compilers. We define a compiler as a program/software that translates a higher-level programming language into a lower-level machine code[3]. Examples of higher-level programming languages are Java, C, C++ and so on. Machine code is either assembly code or executable binary.

Computers can understand only one language, which is the machine language, code consisting of only binaries(zeros and ones). A simple program that takes two numbers as input and outputs the addition of these two numbers written in binary would be like figure 2.1.



00000	10011110
00001	11110100
00010	10011110
00011	11010100
00100	10111111
00101	00000000

```
int a, b, sum;

cin >> a;
cin >> b;

sum = a + b;
cout << sum << endl;
```

Figure 2.1: Programs with exact same purposes in binary and C++[4]

As one can imagine, programming in zeros and ones directly is very tedious and error-prone. One can easily flip a digit and the whole program crashes. And, It is really hard to debug such a program written in binary. To make programming easier, computer engineers have developed higher-level programming languages. Higher-level programming languages are closer to human language and are easier for programmers to write, understand and inspect programs written in these languages. In figure 2.1, the program written in C++ achieves the exact same purpose as the binary program. Even people without any experience in C++ would be able to appreciate how much easier it is to understand code written in C++ compared to the program written in binary.

Since computers only understand binary, but programmers would like to write code in higher-level languages, we would need a program to translate(re-write) from higher-level language to binary. For compiled languages like C++, this translation is done by programs called compilers.

Compilers are one kind of the most important and essential software ever written. There is an interesting metaphor, on how essential compilers are, by Georgios Varisteas, a Former Research Associate at the University of Luxembourg, who says, asking the question "What is the importance of compilers" is the same as asking "What is the importance of vehicles in driving?". Coding in a higher programming language without a compiler is like driving without vehicles. In many contexts, a compiler is the programming language.

2.2 General Steps Involved in Compiling a Large Program with Compiler

Step 1: Preprocessing: The compiler preprocesses the source code, which involves handling preprocessor directives (#include, #define, etc.) and expanding macros.

Step 2: Compilation and Object File Generation: The preprocessed source code is then compiled into machine-readable object code by the compiler. This step involves lexical analysis, parsing, semantic analysis, optimization, and code generation. The compiler produces one or more object files (.o or .obj) containing machine code and symbols for each source file. These object files represent the compiled code for individual translation units

Step 3: Linking: The linker combines the object files along with any necessary library files to create the final executable or shared library. It resolves symbols and addresses, performs static or dynamic linking, and handles dependencies between different object files and libraries.

Step 4: Symbol Resolution and Relocation: The linker resolves references to symbols, both within the program and external libraries, by matching them to their definitions. Then, the linker adjusts the memory addresses of the object code and resolves any relocations needed for the program to run correctly at the specified memory locations.

Step 5: Link-Time Optimization (optional): Some compilers and linkers perform additional optimizations at link-time, taking advantage of global program analysis across multiple translation units to generate more efficient code.

Step 6: Output Generation: Finally, the linker produces the executable file or shared library, which can be directly executed or used by other programs.

2.3 Importance of Compilers Testing

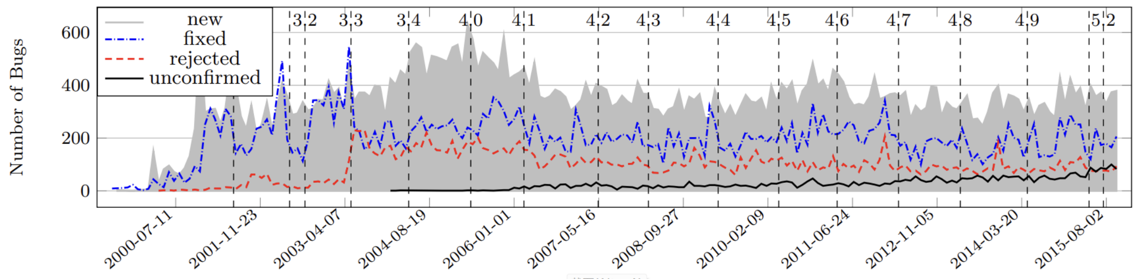
Compiler testing can be divided into three aspects, testing on floating-point precision, on performance in program execution time, and on correctness. The precision of compilers is affected by the order of arithmetic calculation. The same equation might be evaluated into slightly different results in decimal when it is compiled by different compilers. Some compiler optimization changes the order of evaluation and leads to overflow or underflow of floating point numbers. For some physical or chemical simulations, precision is very important because a tiny loss in precision may lead to a significantly different result. Whereas for programs on financial predictions, a slight loss in floating-point precision might not be an issue because only two decimal digits are needed. Performance in program execution time is where most optimizations in compilers aim to reduce. However, some optimizations might have a negative impact on certain workloads. For example, the optimization "loop-unrolling" might push more pressure on register allocation and lead to worse performance. Testing on performance is also important. Testing on the correctness, however, is the most crucial part of compiler testing and is also where most of the efforts are spent on. Correctness is the most essential quality a compiler should guarantee. Generating wrong code is much worse than generating bad code. In our project, we focus on and discuss more the aspect of testing the correctness of compilers.

Compilers are part of programmers' trusted development base. Many programmers trust compilers and believe compilers are reliable. When a program goes wrong, programmers tend to think there are bugs in the program itself but not the compiler. However, there are bugs in compilers. And when a bug in a compiler is triggered by a program, programmers would spend many efforts debugging the program itself until they eventually realize the bug is introduced by the compiler.

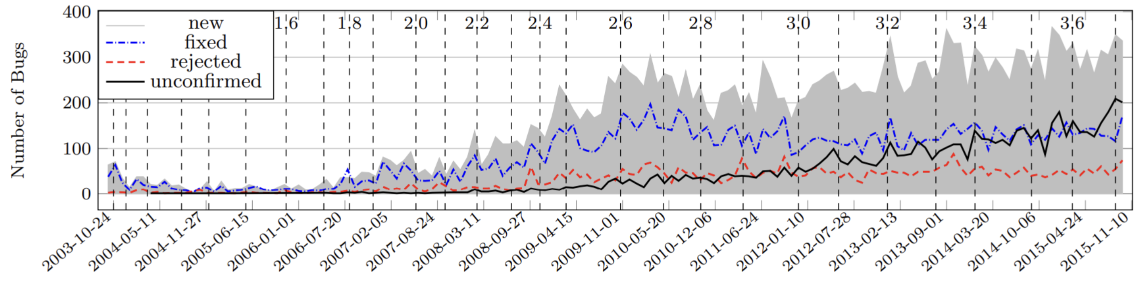
Indeed, there are more bugs than expected. Figure 2.2 is a graph of the history of the bugs tracking system of GCC and LLVM compilers[5]. GCC and LLVM are the two of the most common C compilers in use. We can see there are hundreds of new bugs reported and fixed in these compilers every year.

Bugs in compilers can affect the quality, reliability and security of current and future programs. Some of these miscompiled programs are used in real-world production and even in some critical fields. Many HPC(high-performance computing) workloads, for example, weather forecasting, chemical reaction simulations, physical movement simulation, and so on, use compiler optimizations to improve their performance. A bug in compilers producing slightly different output can lead to a huge difference in the result of these workloads. Some of these bugs affect programs silently, without generating any warnings, and it's hard for programmers to realize that the generated outputs for such big programs are wrong.

Compiler is one of the most fundamental software tools and almost all software systems rely on it. Therefore, it is vitally important to guarantee the reliability of compilers. Compiler testing is an effective and widely-recognized way of ensuring the correctness of compilers[6].



(a) GCC.



(b) LLVM

Figure 2.2: History of bugs tracking system of GCC and LLVM

2.4 Bugs in Compilers

There are two kinds of bugs in compilers, crash bugs and miscompilation bugs. This section provides a brief definition of these two kinds of bugs and provides a few examples of miscompilation bugs that affect large programs.

Crashing Bug

If a program triggers a crash bug in a compiler, then running the generated executable of this program compiled by this compiler would crash. A typical example would be snippet 2.1. In snippet 2.1, running an executable of a program compiled by a version of GCC leads to a segmentation fault. Whereas, in snippet 2.2 running the executable of the same program compiled by other(correct) compilers produces an empty output. This is an example of a crash bug, the output of the buggy compiler leads to a crash whereas the correct output should be empty.

```
#  When Compiling with GCC-10 (gcc-10 (Ubuntu 10.2.0-5
   ubuntu1~20.04)):
> gcc-10 -w -O2 r.c -pedantic -Wall -Wextra
> ./a.out
> Segmentation fault (core dumped)
```

Snippet 2.1: A crash bug reported in GCC 10.2.0.[7]

```
#  Same program in llvm, gcc-9, gcc-8, and gcc-7 exit
   without any output.
> clang-11 -w -O0 r.c -pedantic -Wall -Wextra -fsanitize=
   undefined
> ./a.out
```

```
>
```

Snippet 2.2: Correct output of crash bug in snippet 2.1

Miscompilation Bug

If a program triggers a miscompilation bug in a compiler, then running the generated executable of this program by this compiler would silently generate a wrong output, without any crashing or warning. A typical example would be snippet 2.3 and snippet 2.4. For the same program, two versions of GCC produce different outputs.

```
> gcc-9 ex2.c -o ex
> ./ex
> 0
```

Snippet 2.3: A misocompilation bug reported in GCC-9 [8]

```
> gcc-6 ex2.c -o ex
> ./ex
> 1
```

Snippet 2.4: Correct output of the miscompilation bug in snippet 2.3 [8]

It can already be seen that this kind of bug is much harder to diagnose and also much more significant. First of all, since the wrong output is generated silently, programmers might not realize the output is wrong unless using cross-referencing with different compilers or running a full test suite. In real-life production, it is unlikely for programmers to cross-check every program. Moreover, in many cases, programmers don't even know which compiler is correct and much more effort is needed to analyze this kind of bug.

Bugs Introduced by Optimization

Most bugs in compilers are introduced by optimization passes[9]. It is very unlikely for a developed compiler to have bugs when it's compiling without any optimization option. Optimizations introduce bugs because optimizations involve transformations of code. These transformations might be invalid in some cases, and this is when bugs are introduced. For example, when the program in snippet 2.5 is optimized by a version of the LLVM compiler, it will produce an output of 1, whereas the correct output should be 5. [9]

```c
void foo(void) {
   int x;
   for (x = 0; x < 5; x++) {
      if (x) continue;
      if (x) break;
   }
   printf("%d", x);
}
```

Snippet 2.5: A program triggers a bug in a version of LLVM when optimized[9]

This bug is introduced by a loop optimization pass in LLVM called "scalar evolution analysis". This optimization pass aims to induce the value of variables after the loop. And in the case of snippet 2.5, the optimization sees line 5 and wrongly induces the loop would only run once and the value of x after the loop would be 1.

Miscompilation Bugs in Large Program

Test Failure in Z Shell (caused by bug 29031).[10][2] Zsh (Z shell) is an advanced command-line shell that provides powerful features and customization options for improved productivity and user experience. The buggy compiler miscompiles the function ‘paramsubst’ of the ‘subst.c’ file, a buggy optimization pass wrongly hosting a value out of a loop and cause test script file ‘D04Parameter04.ztst’ to fail.

Test Failure in SQLite (caused by bug 13326)[11][2] SQLite is a lightweight, embedded, and self-contained relational database management system that allows for efficient storage, management, and querying of structured data, without requiring a separate server process. The buggy compiler miscompiles the a line of code in the test bed of test incrblob-2.0.4, causing a value to be wrong inside the test, and ends up reporting an error when this value is detected not as expected.

Test Failure in LevelDB (caused by user-reported bug 27903) LevelDB is an open-source key-value storage library developed by Google, which provides a fast and reliable solution for storing and retrieving data in a sorted manner. It is commonly used as a building block for creating databases and other data storage systems.The buggy compiler miscompiles the function ‘MakeFileName’ from the ‘filename.cc’ file, causing one or more default test cases to fail.

2.5 Current Compiler Testing Techniques

Since compiler testing is so important, researchers and compiler engineers have spent decades developing tools to test compilers. This section discusses three main methodologies to test compilers. First, is to use fixed test suites, second is to use the idea of randomised testing, and third is to use the idea of Equivalence Modulo Inputs testing.

2.5.1 Fixed Test-suite

Like many other kinds of software, there are fixed test-suite for compilers. Many open-source compilers, like GCC[12] and LLVM[13], contain their own test suite in their project and can be run easily following their test-suite guide. Companies that work on compilers may have their own Jenkins test[14] and Smoke test[15] in their CI/CD (Continuous Integration / Continuous Delivery) pipelines[16].

Indeed, every program written in a programming language can be used to test the compilers of this language. Compiler engineers like to use existing High-Performance

Computing (HPC) workloads, like programs on weather forecasting, and math libraries, like OpenBlas, to test all precision, performance and correctness of compilers.

2.5.2 Randomize Testing / Fuzzing

Fixed test suites are inadequate and not complete. Humans can only think of limited test cases. Random testing[17], also called fuzzing[18], is a black-box testing method to test programs by generating random and independent test inputs to this program. The goal of fuzzing is to find inputs that trigger interesting behaviour of the program. Interesting behaviour can be, the program crashes, generates wrong outputs, or runs infinitely, depending on the context. In general, there are five steps in random testing.

Step 1: Define Input Domain

Step 2: Generate inputs independently and randomly from the input domain

Step 3: Test the program on these inputs

Step 4: Compare the result with the system specification

Step 5: If something interesting happens, takes some actions

Randomize Testing in Compiler – CSmith

Compiler fuzzing[2] is widely used in compiler testing and many bugs have been found by using this method, an example is CSmith. CSmith[1] is a random generator of C programs that statically and dynamically conform to the C99 standard. CSmith turns out to be very useful in stress testing C compilers. Lot of bugs in open-source C compilers are reported by CSmith.

The goal of CSmith is to generate a well-formed and single-meaning program with a checksum of variables as output and this generated program needs to be expressive, i.e., support many language features (combinations).

CSmith generates a random program in the following steps.

Step 1: Csmith randomly selects an allowable production from its grammar at the current program point. There is a probability distribution table for selecting each production and a filter to reject invalid production.

Step 2: If the selected production requires a target—e.g., a variable or function—then the generator randomly selects an appropriate target or defines a new one.

Step 3: If the selected production allows the generator to select a type, Csmith randomly chooses one.

Step 4: If the selected production is nonterminal, the generator recurses.

Step 5: Csmith executes a collection of dataflow transfer functions.

Step 6: Csmith executes a collection of safety checks.

The output of a CSmith program would be a checksum of all the variables. The generated checksum of variables can be used to test the correctness of a compiler by cross-checking with different compilers. Cross-checking is achieved by using a voting mechanism. Figure 2.3 is a demonstration of how the voting mechanism works. CSmith uses three compilers, each compiler compiles and runs CSmith-generated programs individually, and the outputs are compared. If all the outputs are the same, then all compilers are correct. If there are differences in output, then the majority result is considered correct, and the compiler produces the minority result is considered buggy.

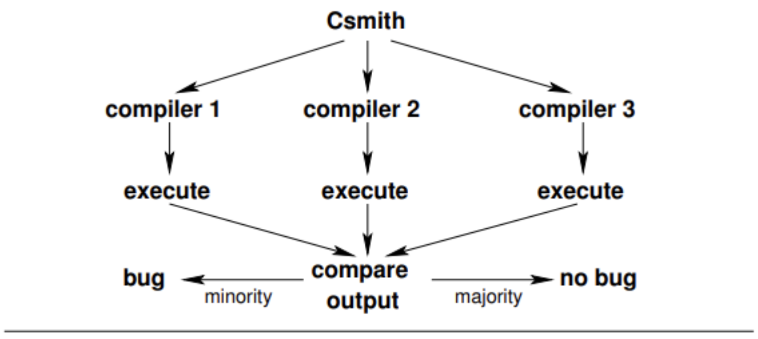


Figure 2.3: Finding bugs in three compilers using randomized differential testing [1]

CSmith manages to find and report a lot of bugs in different versions of GCC and LLVM. Since CSmith uses the idea of random differential testing, these bugs have a wide range of diversity. Most of the reported bugs are difficult to discover by humans. Other contribution of CSmith includes, CSmith advances the state of the art in compiler testing. Csmith tests a large subset of C language while avoiding undefined and unspecified behaviours.

2.5.3 Equivalence Modulo Inputs(EMI) Testing in Compiler

Equivalence modulo inputs (EMI), is a simple, widely applicable methodology for validating optimizing compilers. The idea is to take a real-world code and transform it to produce different, but equivalent variants of the original code.

In general, EMI testing has two steps. For a given real-world program P. Tools using EMI testing method would

Step 1: Generate many equivalent variant programs Q.
The equivalent variant program means, for a given input set I, output of Q + output of P. Step 1 is achieved by adding codes to program P that are for sure not going to be executed (dead code) to generate program Q. The added code is guaranteed not going to be executed because the "Profile and Mutate" strategy is used. The program P is profile and only the part of the dead code is pruned or mutated.

Step 2: Check whether every program Q is valid by comparing it output with program P. Outputs of program Q and P should be equivalent.

A concrete example of EMI testing would be: Given a program P, if we would like to use the equivalence modulo input method to test the compiler, we would add dead codes, for example, “if false xxx”, that would never be executed to program P. Adding this kind of dead code produces a variant of program P, let’s call it program Q. Program Q should produce the same output as P since the added statement is dead code. If the outputs of P and Q differ, it means there is a bug in the compiler.

2.6 Criteria of Reporting Compilation Bugs

After bug cases are found in compilers, we would need to find a way to report these bugs. An ideal compiler bug report should follow these criteria.[19]

1: The bug report should contains a small, easy-to-read, self-contained, and well-defined test program.

2: The identification of the buggy compiler. Including its name, version, target machine and OS, how it was built and so on.

3: Instructions on how to reproduce the bug.

4: Correct and actual output of this program.

2.7 The Test-Case Reduction Problem

The 2nd, 3rd and 4th criteria in section 2.6 can be achieved in a systematic way. The 1st criterion is the most difficult to meet. When a compiler bug occurs in a large program and needed to be reported, the circumstances leading to it must be narrowed down, or else it is a very complicated environment to debug. Manual test-case reduction is both difficult and time-consuming. This is defined as the test-case reduction problem, which is to reduce the size of test cases that trigger the bug. There are different approaches to solving this problem, which are Delta debugging(DD), Hierarchical Delta Debugging(HDD), and C-Reduce is an implementation in solving test-case reduction problem in C compilers.

2.7.1 Delta Debugging(DD)

Delta Debugging[20][21] is an automated technique for paring down large failing inputs and is widely used in software debugging. The goal of delta debugging is to generate a "minimal" input from a large buggy program. The minimal input should still induce the bug. Delta debugging is achieved by selectively, systematically, and iteratively removing a portion of inputs. The reduced input can be accepted or rejected by validating whether this input still induces the bug. Iteration runs until every portion of input has been tried to remove.

Hierarchical Delta Debugging(HDD)

Hierarchical Delta Debugging(HDD)[22] is an algorithm using the idea of delta debugging on a hierarchical structure input like trees. HDD algorithm uses the benefit of hierarchical structure, speeds up delta debugging and produces higher quality reduced input. For example, the DD algorithm removes a string at each iteration, but the HDD algorithm removes a node or a sub-tree at each iteration. Since C programs follow a hierarchical structure, the HDD algorithm can be used in C compiler bug case reduction.

2.7.2 C-Reduce

C-Reduce[19] is a tool that aims to solve the test-case reduction problem in C compiler bug reports using the idea of delta debugging. Provided there is a compiler bug in a complicated program, C-Reduce is an automated tool to narrow down this complicated program into a simplified program. This simplified program should still trigger the bug.

C-Reduce uses three new approaches to test-case reduction. 1. Seq-Reduce: This is only used in CSmith-generated programs and is done by randomly changing the sequence in CSmith generation. 2. AST-Based Reduction: This is also only used in CSmith-generated programs and is done by instrumenting CSmith's generated code, removing dead codes and references. 3. General C-Reduce: this is a general approach that can be used in any C program. This approach exhaustively performs different classes of transformations at each state, accepts or rejects each transformation, and goes to the next states until all states finish.

C-Reduce advances the state art in the test-case reduction problem in C compilers by achieving two goals. First, C-Reduce manages to get away from local minima in the reduction of test programs. Second, C-Reduce does not produce test programs with undefined behaviours. C-Reduce uses generalized transformations and a generalized search to get away from local minima. Generalized transformations are transformations including function-inline and replacing an aggregate type with its constituent scalar. This method deals with more scattered points in the program. Generalized search in C-Reduce is a non-greedy search, so avoid sticking to local minima. C-Reduce uses two approaches to address the test case validity. The first is by avoiding the generation of such variants in CSmith. The second is to use an automatic static analyzer + semantic-based interpreter to double-check the validity.

Limitation of C-Reduce[19]

1: C-Reduce only takes a single C compilation unit as input. When a buggy program consists of multiple source files and we don't know which one is miscompiled. Also, it's pretty hard to define C-Reduce's input script when a test case fails in a large program.

2: Sometimes a reduced test case still does not provide sufficient information for compiler engineers to debug. There are many other approaches to provide more information to compiler engineers to diagnose a bug.

3: Reduction would fail for bugs stemming from internal resource limits in a compiler i.e., register spilling.

4: C-Reduce is Not suitable for non-deterministic programs. This is a common limitation of all testing techniques. It's very difficult to test programs with randomness.

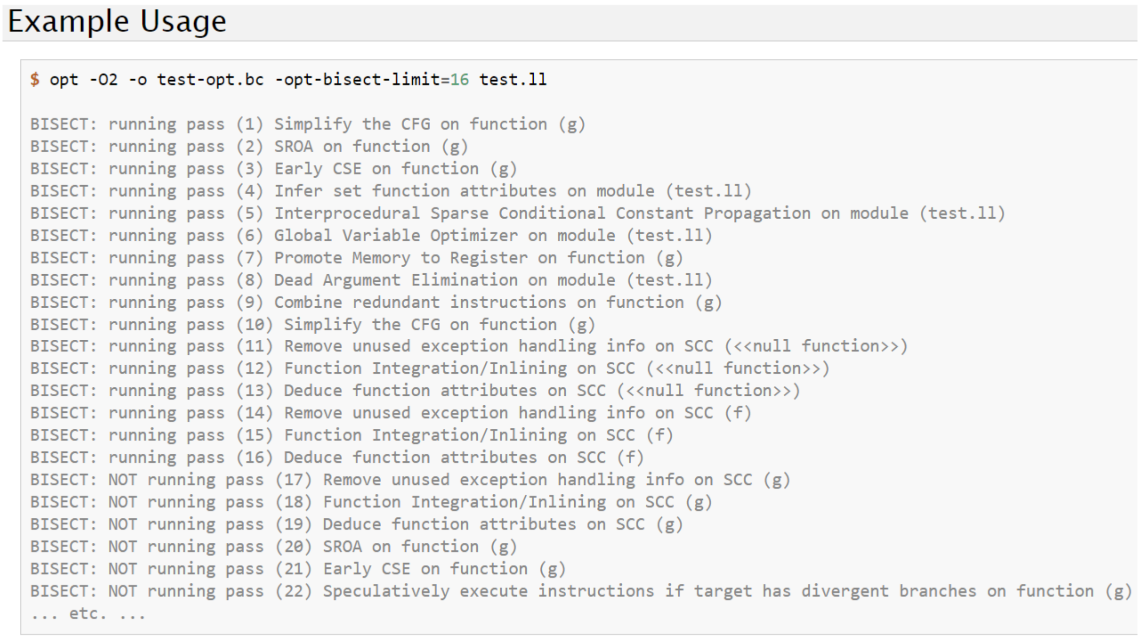
2.8 Compiler Options for Debugging Compiler Bugs

In this section, we discuss some useful compiler options for debugging compiler bugs. We mainly discuss options in the LLVM compiler, other compilers have similar options that achieve the same functionalities. Some of these options are used in C-Locate to help provide useful information to compiler engineers. Running these options needs much human effort but C-Locate finds a way to automate the process.

2.8.1 LLVM Opt-bisect-limit

As we've discussed in section 2.4, most bugs in compilers are introduced in optimization passes. When a bug occurs, if we know which pass introduces the bug, it would make our life easier in debugging. LLVM compilers provide an option "-mllvm -opt-bisect-limit"[23] to help programmers to bisect the buggy optimization pass.

Compilers run optimization passes in a fixed order, and "opt-bisect-limit" is an option to limit the total number of optimization passes being run. An example usage would be figure 2.4. We can see by using this option, we can locate the optimization passes easily.


Example Usage

```
$ opt -O2 -o test-opt.bc -opt-bisect-limit=16 test.ll

BISECT: running pass (1) Simplify the CFG on function (g)
BISECT: running pass (2) SROA on function (g)
BISECT: running pass (3) Early CSE on function (g)
BISECT: running pass (4) Infer set function attributes on module (test.ll)
BISECT: running pass (5) Interprocedural Sparse Conditional Constant Propagation on module (test.ll)
BISECT: running pass (6) Global Variable Optimizer on module (test.ll)
BISECT: running pass (7) Promote Memory to Register on function (g)
BISECT: running pass (8) Dead Argument Elimination on module (test.ll)
BISECT: running pass (9) Combine redundant instructions on function (g)
BISECT: running pass (10) Simplify the CFG on function (g)
BISECT: running pass (11) Remove unused exception handling info on SCC (<<null function>>)
BISECT: running pass (12) Function Integration/Inlining on SCC (<<null function>>)
BISECT: running pass (13) Deduce function attributes on SCC (<<null function>>)
BISECT: running pass (14) Remove unused exception handling info on SCC (f)
BISECT: running pass (15) Function Integration/Inlining on SCC (f)
BISECT: running pass (16) Deduce function attributes on SCC (f)
BISECT: NOT running pass (17) Remove unused exception handling info on SCC (g)
BISECT: NOT running pass (18) Function Integration/Inlining on SCC (g)
BISECT: NOT running pass (19) Deduce function attributes on SCC (g)
BISECT: NOT running pass (20) SROA on function (g)
BISECT: NOT running pass (21) Early CSE on function (g)
BISECT: NOT running pass (22) Speculatively execute instructions if target has divergent branches on function (g)
... etc. ...
```

Figure 2.4: Example usage of llvm opt-bisect-limit [1]

2.8.2 LLVM Bugpoint

LLVM Bugpoint[24] is an automated test case reduction tool that uses the idea of cross-checking and delta debugging. It also makes use of its understanding of C program and LLVM IR. The "LLVM -Bugpoint" miscompilation debugger splits the test program into two pieces, running the optimizations specified on one piece, linking the two pieces back together, and then executing the result.

2.9 Git Options for Debugging

This section discusses a few Git options that we use in C-Locate to help provide useful information in Debugging. Running these options needs much human effort but C-Locate find a way to automate the process.

2.9.1 Git-log

Git-log[25] is a command in the Git version control system that allows you to view the commit history of a Git repository. It displays a chronological list of commits, showing details such as commit messages, authors, dates, and commit hashes. It's a useful tool for inspecting the history of changes in a repository, understanding when and why specific changes were made, and tracking the evolution of the codebase over time.

2.9.2 Git-bisect

Git-bisect[26] is a command in Git that helps you find a specific commit that introduced a bug or caused a regression in your codebase. It uses a binary search algorithm to efficiently narrow down the problematic commit by performing a series of tests or evaluations. You mark a known good commit and a known bad commit, and Git-bisect automatically checks out and allows you to test intermediate commits, helping you pinpoint the exact commit where the issue was introduced. It significantly aids in identifying and isolating the problematic changes, making it easier to track down and fix bugs in your codebase.

2.10 Other Useful Tools or Libraries

2.10.1 Clang LibTooling

Clang LibTooling[27] is a library provided by the Clang compiler infrastructure that enables the creation of custom tools and utilities for C++ code analysis and manipulation. It offers a high-level C++ API that provides access to the Clang compiler's Abstract Syntax Tree (AST) and allows for static analysis, code rewriting, and source-to-source transformations. I used Clang LibTooling library to implement the functionality of locating the miscompiled function.

2.11 Conclusion

Compilers are one of the most important software ever written, and ensuring the reliability of compilers plays a crucial role in software development. Compiler testing is critical to ensure a compiler is bug-free. Many tools and ideas have been developed for compiler fuzzing to find bugs in compilers. After a bug is detected, diagnosing and fixing it is as important. However, there aren't many tools that help compiler engineer to diagnose a compiler bug.

There are mainly two kinds of bugs in compilers, cashing bugs and miscompilation bugs. Crashing bugs are easier to debug because you know where in the compiler it crashes. Miscompilation bugs are much harder to diagnose because there isn't any information provided, except an incorrect output is produced. So, we do need more information about miscompilation bugs to help compiler engineer to diagnose and fix this kind of bug.

C-Reduce is a very helpful tool in helping compiler engineers diagnose a compiler bug by solving the test case reduction problem, which reduces the code size for a test case. However, a major limitation of C-Reduce is that it mainly works in a test-case environment. When the buggy program becomes large and has a complicated testing and compiling process, C-Reduce is not very helpful.

There are many compiler options and git options that help compiler engineers to diagnose compiler bugs, like LLVM opt-bisect or Git git-bisect. However, running these options sometimes need quite a lot of human effort and is also not efficient in large program cases.

In this project, we present C-Locate, a tool that takes in a buggy compiler and miscompiled program with other essential information, then output useful information to the compiler engineer to help diagnose a miscompilation bug in a large program. C-Locate achieves this by manipulating the compilation and object file generation in the compiling process, and automating some compiler and git options to minimise the human effort needed.

Chapter 3

C-Locate

C-Locate project consists of two tools, C-Locate and C-Locate-Function. C-Locate implements the functionalities of locating the buggy source files, the buggy optimization pass and the buggy commit. These three functionalities should work for both C and C++ programs. The C-Locate-Function provides the last functionality of locating and extracting the buggy function, which for current implementation only works for C programs.

3.1 General Run of C-Locate

3.1.1 Compiling & Installing C-Locate

There is a 'Makefile' provided in the C-Locate source directory, running a "make" command will compile C-Locate and C-Locate-Function. Snippet 3.1 is an example to compile C-Locate with g++, where the compiler here can be changed by setting 'CC' environment variable in 'Makefile'. C-Locate-Function has a fixed compile command and libraries with a specific version of clang-11 and llvm-11 needed because Clang LibTooling version 11 library is used to do the front-end parsing of C programs. Using other versions might as well be fine, but not guarantee to be compatible.

```
> make
g++  -c ./source/compiler.cpp
g++  -c ./source/linker.cpp
g++  -c ./source/parser.cpp
g++  -c  ./source/loader.cpp
g++  -o C-Locate ./compiler.o ./linker.o ./main.o ./parser
   .o ./loader.o

clang++-11 -std=c++17 ./source/main2.cpp `llvm-config-11
   --cxxflags --ldflags --libs --system-libs` -
   lclangTooling -lclangASTMatchers -lclangFrontend -
   lclangSerialization -lclangDriver -lclangParse -
   lclangSema -lclangEdit -lclangAnalysis -lclangAST -
   lclangLex -lclangBasic -o C-Locate-Function
```

Snippet 3.1: Compiling C-Locate

3.1.2 Running C-Locate

C-Locate and C-Locate-Fucntion can both be run using a single-line terminal command with one input file. Snippet 3.2 is an example to run C-Locate with input file "test_input.config" and running C-Locate-Function with input file "input_to_2nd_tool". Input to C-Locate-Function is automatically generated by C-Locate, which will be explained later.

```
> ./C-Locate test/testcase7/test_input.config
> ./C-Locate-Function input_to_2nd_tool.txt
```

Snippet 3.2: Running C-Locate

3.1.3 Input to C-Locate

C-Locate takes a file as input in the format as snippet 3.3.

```
#true or false to use make
using_make_to_compile=true

#Script to configure software with correct version \
#of compiler. (Optional)
configure_script=
#Script to run make
make_script=make VERBOSE=1
#Script to run test
check_script=make TESTNUM=D04 check
#Script to clean make
clean_script=make clean

#All directory
built_directory=/home/alan/FYP-Reporting-Compilation-
   Bugs/test/testcase7/zsh-5.9/build
#directory of Correct compiler
correct_compiler=clang
#(Optional) compiler option of correct compiler
correct_compiler_option=
#directory of buggy compiler
buggy_compiler=/home/alan/FYP-Reporting-Compilation-Bugs/
   test/testcase7/llvm-project-331
   fb804c96008bbceec8e5f305fc27f06d58ba6/build/bin/clang
#(Optional) compiler option of buggy compiler
buggy_compiler_option=
#directory of source files.
source_file_directory=/home/alan/FYP-Reporting-
   Compilation-Bugs/test/testcase7/zsh-5.9/
#true or false
is_git_directory=false
buggy_compiler_git_directory=
buggy_compiler_compile_script=
```

Snippet 3.3: Input file to C-Locate

These inputs include:

using_make_to_compile ("true" or "false")

This is a field telling C-Locate whether the buggy program is compiled using a Makefile. If this field is set to false, C-Locate would try to compile all source files using the given compilers and options in the most basic way, which is to compile each source file separately and link them. If this field is set to true, users would need to provide the following extra information to let C-Locate work.

configure_script (Optional)

This field would only be used if "using_make_to_compile" is set to true, and it's still optional in this case. This field tells C-Locate how to configure the input program with a **correct** version compiler that produces **correct** output. If the program is already configured with a **correct** version compiler, this field can be empty.

make_script (Required when using_make_to_compile=true)

This field would only be used if "using_make_to_compile" is set to true. This field tells C-Locate how to run the make command.

check_script (Required when using_make_to_compile=true)

This field would only be used if "using_make_to_compile" is set to true. This field tells C-Locate how to run the test command. It's preferable that this command would only run a single failing test case instead of running the whole test suite.

clean_script (Required when using_make_to_compile=true)

This field would only be used if "using_make_to_compile" is set to true. This field tells C-Locate how to run a clean command to clean all built files.

built_directory (Required when using_make_to_compile=true)

This field would only be used if "using_make_to_compile" is set to true. This field tells C-Locate the built directory of the program when running "make_script". This field should be consistent with the configuration of this program.

correct_compiler (Required)

This field provides a correct compiler's absolute directory to C-Locate. It can also be a single command like "clang" if this compiler is installed in the system or PATH. C-Locate uses this compiler and its output as oracles to determine when the program is miscompiled.

correct_compiler_option (Optional)

This field tells which option the correct compiler is using. This can be left empty if none of the compiler options is used or it's already been configured.

buggy_compiler (Required)

This field provides the known buggy compiler's absolute directory to C-Locate. It can also be a single command like "clang" if this compiler is installed in the system or PATH. C-Locate tries to diagnose the bug in this compiler.

buggy_compiler_option (Optional)

This field tells which option the buggy compiler is using. This can be left empty if none of the compiler options is used or it's the same as the configuration. However, this field should normally be filled with some value because that's when a bug is introduced in the buggy compiler and a program gets miscompiled.

source_file_directory (Optional)

This field tells C-Locate where the parent directory of the miscompiled program is. It should be an absolute directory to prevent potential issues.

is_git_directory ("true“ or "false")

This field tells C-Locate whether the buggy compiler is in a git repository. Setting this field to true will turn on the functionality of locating the git commit that introduces the bug.

buggy_compiler_git_directory (Required when is_git_directory=true)

This field would only be used if "is_git_directory" is set to true. This field tells C-Locate where the git directory of the buggy compiler lies, this should be consistent with the buggy compiler provided before.

buggy_compiler_compile_script (Required when is_git_directory=true)

This field would only be used if "is_git_directory" is set to true. This field tells C-Locate how to compile and install the compiler when it's switched to a different git commit.

3.1.4 Output of C-Locate

The Buggy Source File

C-Locate will output which source file is getting miscompiled in the given source file directory. Snippet 3.4 is a snapshot of C-Locate outputting the miscompiled source file in the absolute path.

```
==================================================
Buggy File(s) are
/home/FYP/test/testcase7/zsh-5.9/Src/subst.c
```

```
==================================================
```

Snippet 3.4: C-Locate outputs the miscompiled source file

The Optimization Passed that Introduces the Bug

C-Locate will output which compiler optimization pass introduced the bug. Snippet 3.5 are snapshots of C-Locate outputting which optimization pass introduces the bug.

```
==============================================
Buggy optimization passes are between
63 and 64
==============================================
Corresponding to
==============================================
BISECT: running pass (61) Simplify the CFG on function (
    paramsubst)
BISECT: running pass (62) SROA on function (paramsubst)
BISECT: running pass (63) Early CSE on function (
    paramsubst)
BISECT: running pass (64) Early GVN Hoisting of
    Expressions on function (paramsubst)
BISECT: NOT running pass (65) Simplify the CFG on function
     (untok_and_escape)
BISECT: NOT running pass (66) SROA on function (
    untok_and_escape)
BISECT: NOT running pass (67) Early CSE on function (
    untok_and_escape)
BISECT: NOT running pass (68) Early GVN Hoisting of
    Expressions on function (untok_and_escape)
==============================================
```

Snippet 3.5: C-Locate outputs the buggy opt pass

The Git Commit that Introduces the Bug

C-Locate will output the git commit ID of the buggy compiler that introduced the bug. Commits before this commit are bug-free. Snippet 3.6 is a snapshot of C-Locate outputting which commit ID introduced the bug.

```
==============================================
Buggy git commit ID is
d3b10150b683142a7481893ddffe9206e
1
==============================================
```

Snippet 3.6: C-Locate outputs the buggy commit ID

An Input File to C-Locate-Function

C-Locate will generate a file that in most cases can directly be used as the input to C-Locate-Function, which will be discussed in the next section.

3.1.5 Input File to C-Locate-Function

C-Locate-Function takes a file as input in the format as snippet 3.7. This input file will be automatically generated by running C-Locate. It can also be manually inputted.

```
buggy_file=/home/alan/FYP-Reporting-Compilation-Bugs/test/testcase7/zsh-5.9/Src/subst.c

buggy_compiler_command=/home/alan/FYP-Reporting-Compilation-Bugs/test/testcase7/llvm-project-331fb804c96008bbceec8e5f305fc27f06d58ba6/build/bin/clang -c -I. -I../Src -I../../Src -I../../Src/Zle -I../../Src  -DHAVE_CONFIG_H -Wall -Wmissing-prototypes -O2  -o subst.o ../../Src/subst.c -fPIC

correct_compiler_command=clang -c -I. -I../Src -I../../Src -I../../Src/Zle -I../../Src  -DHAVE_CONFIG_H -Wall  -Wmissing-prototypes -O2  -o subst.o ../../Src/subst.c -fPIC

built_directory=/home/alan/FYP-Reporting-Compilation-Bugs/test/testcase7/zsh-5.9/build/Src

check_directory=/home/alan/FYP-Reporting-Compilation-Bugs/test/testcase7/zsh-5.9/build

make_script=make VERBOSE=1

check_script=make TESTNUM=D04 check

clean_script=make clean
```

Snippet 3.7: Input file to C-Locate-Function

This file includes the following fields:

buggy_file

This field provides the absolute path of the miscompiled source file that's been located by C-Locate.

buggy_compiler_command

This field provides the exact command that's used to compile the miscompiled source file with the buggy compiler.

correct_compiler_command

This field provides the exact command that's used to compile the miscompiled source file with the correct compiler.

built_directory

This field provides the exact directory where the object file of the miscompiled source file is produced. Note that this might be different from the input before in C-Locate because the produced object file might be in the subdirectory of the built directory.

check_directory

This field provides the directory to run the test. This directory is usually the same as the "built_directory" input to C-Locate.

Other fields are just the same as the inputs to C-Locate and are also needed to be used by C-Locate-Function.

3.1.6 Output of C-Locate-Function

The Miscompiled Function Name

C-Locate-Function would output the name of function that's getting miscompiled like snippet 3.8.

```
========================================
Miscompiled Function is:
paramsubst
========================================
```

Snippet 3.8: C-Locate-Function outputs the miscompiled function name

Three files that can function the same as the original miscompiled source file

C-Locate-Function would generate three files. One source file contains only the miscompiled function like snippet 3.9. Another source file contains all other functions in the original source file and a common header to share between these two generated source files.

```
#include "myHeader.h"
/**/
LinkNode
my_magic_123paramsubst
(LinkList l, LinkNode n, char **str,
int qt, int pf_flags, int *ret_flags)
{
    char *aptr = *str, c, cc;
    char *s = aptr, *ostr = (char *) getdata(n);
    int colf;
    /*...*/
    /*...*/
    /*...*/
    if (eval)
   *str = (char *) getdata(n);
    return n;
}
```

Snippet 3.9: C-Locate-Function outputs the miscompiled function body

3.2 Design & Implementation

3.2.1 Mixed-Compiling

C-Locate uses the idea of mixed-compiling as the core of the implementation of functionalities. Mixed-compiling is an idea to compile source files with different compilers. In C-Locate, the idea of mixed-compiling is implemented in a way that one source file is compiled with a buggy compiler, while all other source files are compiled with a correct compiler. In this way, we can test each file, in turn, to know which one is getting miscompiled. Figure 3.1 is a diagram to demonstrate how mixed-compiling is implemented in C-Locate, the detailed algorithms for each functionality are discussed in later sections respectively.

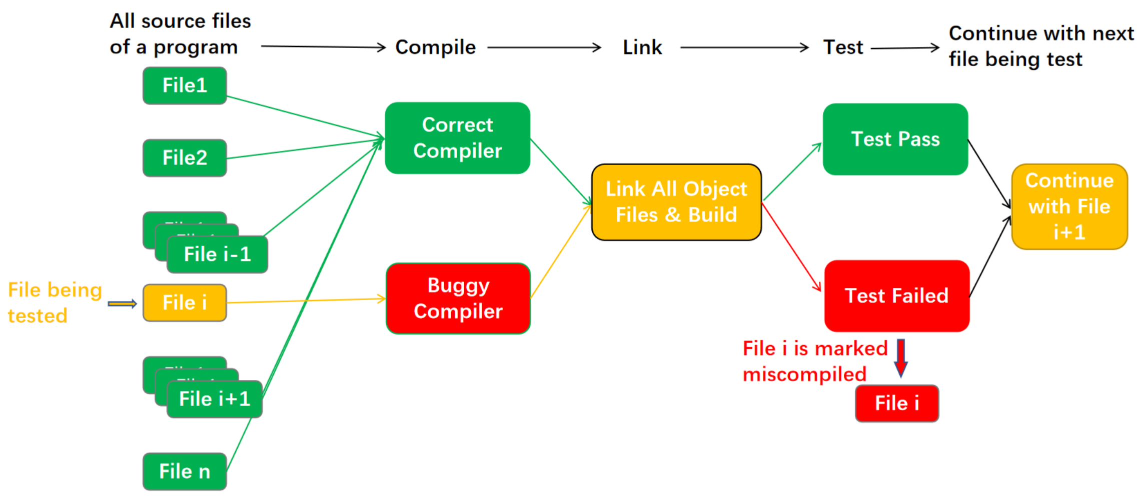


Figure 3.1: Main idea of mixed-compiling implemented in C-Locate

3.2.2 Locating the Miscompiled Source File

This functionality is achieved using the idea of mixed-compiling and delta debugging. The process can be generalized into few steps:

Step 1: Compile a source file with the buggy compiler

Step 2: Compile all other source files with the correct compiler

Step 3: Run the test and see whether it fails.

Step 4: If it fails, then we know this is the file getting miscompiled.

Step 5: Go back to step 1 and iterate until every source file has been tested. (C-Locate doesn't short-run even when Step 4 detects a miscompiled file, just in case there are multiple files getting miscompiled.).

C-Locate doesn't use a divide-and-conquer approach to bisect source files, because when there are multiple miscompiled source files, the divide-and-conquer approach wouldn't work more efficiently than linear search.

The general steps seem straight forwards, but there are a **few challenges** in implementing this functionality:

Getting the Compiling Commands for Each Source File

When "using_make_to_compile" is not set to true, C-Locate would try to compiles all source files in the source file directory with the given compiler and compiler options to create object files. Then, try to links these object files together. In this case, compiling commands are constructed by C-Locate.

When a Makefile is used to construct the miscompiled program, things become more complicated. In this case, C-Locate relies on the echo from "make" or "make VERBOSE=1". For most of the Makefile, there would be echo lines on compiling commands like in the snippet 3.10. C-Locate extracts these commands and replaces the compiler as needed. In the case when "make" doesn't echo any compiling commands, C-Locate won't function very well, and this is a limitation which will be discussed in a later section.

```
clang -c -I. -I../../Src -I../../../Src -I../../../Src/Zle -I../../../Src/Zle  -DHAVE_CONFIG_H -DMODULE -Wall -Wmissing-prototypes -O2 -fPIC -o zle_bindings..o ../../../Src/Zle/zle_bindings.c

clang -c -I. -I../../Src -I../../../Src -I../../../Src/Zle -I../../../Src/Zle  -DHAVE_CONFIG_H -DMODULE -Wall -Wmissing-prototypes -O2 -fPIC -o zle_hist..o ../../../Src/Zle/zle_hist.c

clang -c -I. -I../../Src -I../../../Src -I../../../Src/Zle -I../../../Src/Zle  -DHAVE_CONFIG_H -DMODULE -Wall -Wmissing-prototypes -O2 -fPIC -o zle_keymap..o ../../../Src/Zle/zle_keymap.c

clang -c -I. -I../../Src -I../../../Src -I../../../Src/Zle -I../../../Src/Zle  -DHAVE_CONFIG_H -DMODULE -Wall -Wmissing-prototypes -O2 -fPIC -o zle_main..o ../../../Src/Zle/zle_main.c

clang -c -I. -I../../Src -I../../../Src -I../../../Src/Zle -I../../../Src/Zle  -DHAVE_CONFIG_H -DMODULE -Wall
```

```
    -Wmissing-prototypes -O2 -fPIC -o zle_misc..o
   ../../../Src/Zle/zle_misc.c

clang -c -I. -I../../Src -I../../../Src -I../../../Src/
   Zle -I../../../Src/Zle  -DHAVE_CONFIG_H -DMODULE -Wall
    -Wmissing-prototypes -O2 -fPIC -o zle_move..o
   ../../../Src/Zle/zle_move.c

clang -c -I. -I../../Src -I../../../Src -I../../../Src/
   Zle -I../../../Src/Zle  -DHAVE_CONFIG_H -DMODULE -Wall
    -Wmissing-prototypes -O2 -fPIC -o zle_params..o
   ../../../Src/Zle/zle_params.c
```

Snippet 3.10: "make" echo compiling commands

Working Out the Directory to Run Compile Commands

As we can see in snippet 3.10, some compiling commands would include libraries from a relative path and even the source file can be in another directory. We would need to change to a correct working directory to run compile commands. Moreover, for each source file, the correct working directory may be different.

To solve this issue, we work out the correct working directory to run a compiling command by comparing the absolute path of the object file and the relative path in the compiling commands. C-Locate would scan through the "build_directory" and produce a profile of absolute path to object file like snippet 3.11. By comparing the absolute path and the relative path after '-o' in a compiling command, we could work out which working directory we should be in.

```
"/home/alan/FYP-Reporting-Compilation-Bugs/test/testcase7
   /zsh-5.9/build/Src/Zle/zle.mdhs"

"/home/alan/FYP-Reporting-Compilation-Bugs/test/testcase7
   /zsh-5.9/build/Src/Zle/zle utils..o"

"/home/alan/FYP-Reporting-Compilation-Bugs/test/testcase7
   /zsh-5.9/build/Src/Zle/compcore..o"

"/home/alan/FYP-Reporting-Compilation-Bugs/test/testcase7
   /zsh-5.9/build/Src/Zle/zle.mdh"

"/home/alan/FYP-Reporting-Compilation-Bugs/test/testcase7
   /zsh-5.9/build/Src/Zle/complist,mdhi"

"/home/alan/FYP-Reporting-Compilation-Bugs/test/testcase7
   /zsh-5.9/build/Src/Zle/compmatch.syms"

```

```
"/home/alan/FYP-Reporting-Compilation-Bugs/test/testcase7
   /zsh-5.9/build/Src/Zle/zle utils.syms"
```

Snippet 3.11: C-Locate records the absolute paths of all object files

3.2.3 Locating the Buggy Optimization Pass in the Buggy Compiler

This functionality is achieved by using the 'LLVM opt-bisect' option and also makes use of the idea of bisecting and delta debugging. The generalized algorithm is:

Step 1: Previous functionality has located a miscompiled source file and its corresponding compiling command.

Step 2: At the end of this compiling command, we add an extra option "-mllvm -opt-bisect-limit=n", where n=maximum opt passes / 2.

Step 3: Compile the miscompiled source file with this command and Run the test and see whether it fails.

Step 4: If the test fails, we know the buggy optimization pass is lower than n, bisect downwards.

Step 5: If the test passes, we know the buggy optimization is higher than n, bisect upwards,

Step 6: Repeat until the |upper-bound - lower-bound| < 1

LLVM Opt-bisect

'LLVM opt-bisect' is a compiler option provided by LLVM to limit the number of optimization passes getting run by specifying a limit. In specific, if you run "clang -O2 helloworld.c -mllvm -opt-bisect-limit=n", then clang would try to run all "O2" optimization passes, but stop at the nth pass. For example, if you set n=64, then LLVM would run 64 optimization passes like snippet 3.12. The maximum number of optimization passes is different between files and compilers. C-Locate has a mechanism to get this maximum.

```
BISECT: running pass (61) Simplify the CFG on function (
   paramsubst)
BISECT: running pass (62) SROA on function (paramsubst)
BISECT: running pass (63) Early CSE on function (
   paramsubst)
BISECT: running pass (64) Early GVN Hoisting of
   Expressions on function (paramsubst)
BISECT: NOT running pass (65) Simplify the CFG on function
    (untok_and_escape)
BISECT: NOT running pass (66) SROA on function (
   untok_and_escape)
BISECT: NOT running pass (67) Early CSE on function (
   untok_and_escape)
```

```
BISECT: NOT running pass (68) Early GVN Hoisting of
   Expressions on function (untok_and_escape)
```
Snippet 3.12: Clang opt-limit=64

3.2.4 Locating the Buggy Git Commit in Buggy Compiler Repo

This functionality makes use of "git log" and "git checkout" commands and also the git version control system and also follows the idea of delta debugging and bisecting. For the current design choice, C-Locate only checks past 25 commits. The generalized algorithm is:

Step 1: C-Locate would change the working directory into where the compiler repo lies.

Step 2: Run "git log -n 25" to get information on the past 25 commits. Get commits ID for the past 25 commits.

Step 3: Git check-out to the 25 / 2 commits, re-compiles the compiler.

Step 4: Use the reverted compiler to compile the buggy source file with the corresponding command. Run the test.

Step 5: If the test fails, we know the buggy optimization pass is lower than n, bisect downwards.

Step 6: If the test passes, we know the buggy optimization is higher than n, bisect upwards,

Step 7: Repeat until the |upper-bound - lower-bound| < 1

Git Log

Git log produces output like figure 3.13, C-Locates extract the commit ID by simple string manipulation using pattern matching.

```
commit 0d1a55b58704a4bb331688691b8a350973a56ee6
Author: WentengZ <alan.zwt@hotmail.com>
Date:   Fri May 19 16:24:41 2023 +0100

    Initial commit
```
Snippet 3.13: Git log output

3.2.5 Locating the Miscompiled Function in Source File

This functionality makes use of the idea of mixed-compiling, delta-debugging, and also uses some knowledge of variables and functions in C language. Clang LibTooling libraries are used in this functionality to do front-end parsing. The general idea is, for the miscompiled source file that's been located before, C-Locate extracts each function in this source file to another file, using the same idea of mixed-compiling

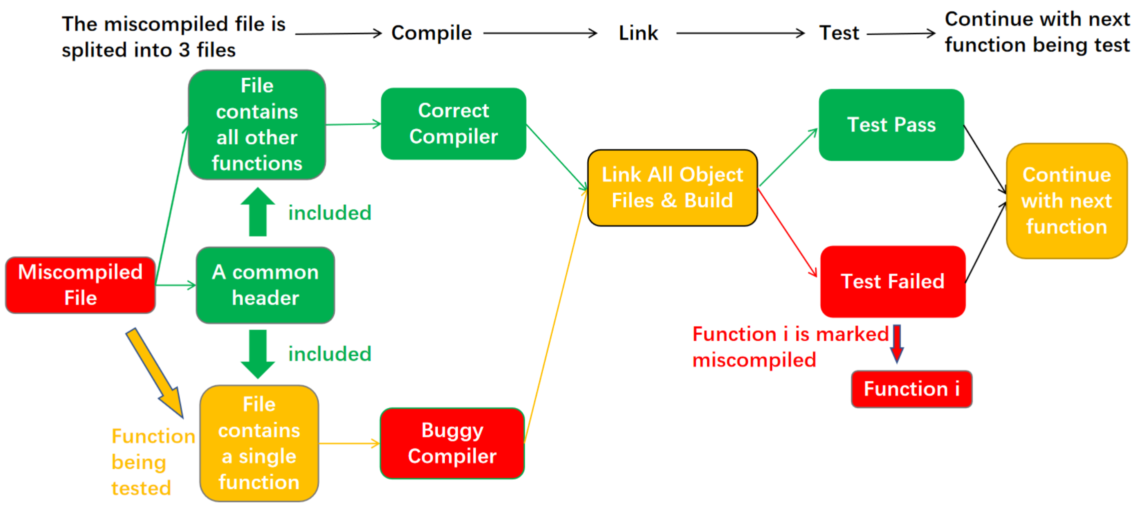

The miscompiled file is splited into 3 files
Compile
Link
Test
Continue with next function being test
File contains all other functions
Correct Compiler
included
Miscompiled File
A common header
included
Function being tested
File contains a single function
Buggy Compiler
Link All Object Files & Build
Test Pass
Test Failed
Function i is marked miscompiled
Function i
Continue with next function

Figure 3.2: Main idea of mixed-compiling implemented in C-Locate-Function

to check which function is getting miscompiled, which is demonstrate in figure 3.2. In general, current implementation of this functionality might **not** work in all cases, but should be extendable. The following is a brief algorithm of how this functionality is implemented:

Step 1: Split the miscompiled source file into three files. One common header, one source file with a single function that's being tested in this iteration, and one source file containing all other stuff.

Step 2: Compile the single-function source file with the buggy compiler and the all-functions source file with the correct compiler.

Step 3: Link the two object files generated last step, and replace the original object file of the original miscompiled file.

Step 4: Recompile the program and run the test.

Step 5: If the test fails, we know this function is getting miscompiled

Step 6: If the test passes, continue iteration.

Step 7: Repeat until all functions have been tested.

How does C-Locate-Function Split the Miscompiled File into 3 Files?

Pre-processing the Source File

Before the original source file is split into three files, C-Locate-Function did some pre-processing.

The first pre-processing is, all functions' body is rewritten to call a magic function, which this magic function is exactly the same as the original function but with a different name. Snippet 3.14 is a specific example. The original function in the source file is "stringsubstquote" with some function body, and the function body is rewritten to call "my_magic_123stringsubstquote" with exactly the same parameter. "my_magic_123 stringsubstquote" has a function body which is exactly the same as the original "stringsubstquote". "my_magic_123" is just some magic string added in front to make sure this function name is unique from all other functions. This pre-processing is applied to every function.

```c
static char *
stringsubstquote(char *strstart, char **pstrdpos) {
// Function body is replaced with a call to my magic
   function
  my_magic_123stringsubstquote(strstart, pstrdpos);
}

// Add a magic string in the function name
// to ensure this name is unique
char *
my_magic_123stringsubstquote(char *strstart, char **
   pstrdpos) {
  int len;
  char *strdpos = *pstrdpos, *strsub, *strret;
  /* ... */
  // Exactly same as the original function body
  /* ... */
}
```

Snippet 3.14: Function rewrite

The second preprocessing is, all calls to the original function within this file are replaced by the magic function, just like in snippet 3.15

```c
// Before
str = stringsubstquote(str, &s);
// After
str = my_magic_123stringsubstquote(str, &s);
```

Snippet 3.15: All function calls are replaced

Why Preprocessing?

The main reason for this preprocessing is to prevent potential issues when creating a common header and splitting functions into two files. There are a few reasons:

1. "static" function type in C means this function is only visible within this file. When we split a source file into two, some calls to static functions would become invalid. With preprocessing, we create a non-static function that can be visible in other files. In snippet 3.15, note that the original function is static, whereas the new magic function is not.

2. We preserve all functionalities, input, and output of original functions, so there wouldn't be any problem with other header files and source files if they have included or called these functions.

3. We create our own magic functions that are only visible between our own created new files, so we are free to include them in our new common header files. And also, with the magic string added, we know there wouldn't be a conflict declaration with other source files.

With all this preprocessing, we can easily split the buggy source file into the three files that we want.

Creating the Single-Function Source File

This source file can be created straightforwardly after preprocessing. All we needed is to include the common header file, which will be discussed later, and the function to be tested, note that we are including our created magic function, because this is safe to use and preserve the functionality of the original function. Snippet 3.16 is an example.

```c
#include "myHeader.h"
/**/
LinkNode
my_magic_123paramsubst
(LinkList l, LinkNode n, char **str,
int qt, int pf_flags, int *ret_flags)
{
    char *aptr = *str, c, cc;
    char *s = aptr, *ostr = (char *) getdata(n);
    int colf;
    /*...*/
    /*...*/
    /*...*/
    if (eval)
   *str = (char *) getdata(n);
    return n;
}
```

Snippet 3.16: Single-Function source file

Creating the All-Other-Function Source File

This source file is just like the single-function source file, but contains all other original and magic functions. Also, all variable declaration is preserved in this file. Snippet 3.17 is a brief example.

```c
#include "myHeader.h"

/*All variables declarations*/
char nulstring[] = {Nularg, '\0'};

/*All functions except the extracted one*/
keyvalpairelement(LinkList list, LinkNode node){
  my_magic_123keyvalpairelement(list, node);
}
LinkNode my_magic_123keyvalpairelement(LinkList list,
    LinkNode node) {
/*Function body*/
}

/*All other functions*/
```

Snippet 3.17: All-other-function source file

Creating a Common Header

All others "#define" "#include" and function declarations of our magic function are in this header file. Note that we would only need to include the declaration of our magic functions because all calls to the original function have been replaced, and including our magic functions are known to be safe and won't conflict with any other header files. In general, header files are like snippet 3.18.

```c
/* All the includes in the original source file */
#include "zsh.mdh"
#include "subst.pro"

/* All the defines */
#define LF_ARRAY 1
#define Nularg  ((char) 0xa1)
#define isend(c) ( !(c) || (c)=='/' || (c)==Inpar || (
    assign && (c)==':') )
#define isend2(c) ( !(c) || (c)==Inpar || (assign && (c)
    ==':') )

/* All file scope variables in original source file are
    declared as extern in this header file*/
extern char nulstring[] ;

/* Functions declaration of my magic functions */
LinkNode my_magic_123keyvalpairelement(LinkList list,
    LinkNode node);

mod_export void my_magic_123prefork(LinkList list, int
    flags, int *ret_flags);
```

Snippet 3.18: The common header file

For variables in the original source file, we need some extra handles. We copy all file scope variables' declaration in the source file to the common header file and add an "extern" quantifier. Snippet 3.19 is what exactly happening. We know this is safe because:

1. For variables with initialization in the original source file. If this variable is already defined as "extern" in other header files, it's fine to redeclare it again in our

newly created header, this is valid in C programming language, as long as it's not getting initialized twice, and we know it is only defined once because it is in the original source file. If this variable is not "extern" in any other header file, then we also know it's fine to define it as "extern" in our newly created header file which would only be included my our newly created source file, because 1. there wouldn't be any conflict 2. It's only visible in our newly created source file.

2. For variables without initialization in the original source file. The statement is basically the same as the previous statement. In both cases, this transformation is valid and functions as before.

```c
/* In original source file */
char nulstring[] = {Nularg, '\0'}
int qt;

/* When are declared in the new common header file */
extern char nulstring[];
extern int qt;
```

Snippet 3.19: Variables to the Common Header file

For all other complicated edge cases that C-Locate-Function don't know how to handle, C-Locate-Function put them in the header file. For example, in a case like snippet 3.20, a function is defined depending on a compile-time macro, C-Locate-Funcion doesn't know how to deal with this case yet, so it's included in the header file and this function would not going to be tested.

```c
/* This function definition is depends on compile time
   macro */
#ifdef MULTIBYTE_SUPPORT
#define WCPADWIDTH(cchar, mw) wcpadwidth(cchar, mw)

static int
wcpadwidth(wchar_t wc, int multi_width)
{
    int width;

    switch (multi_width)
    {
    case 0:
 return 1;

    case 1:
 width = WCWIDTH(wc);
 if (width >= 0)
     return width;
 return 0;

    default:
 return WCWIDTH(wc) > 0 ? 1 : 0;
    }
}

#else
#define WCPADWIDTH(cchar, mw) (1)
#endif
```
Snippet 3.20: A Special Edge case

3.3 A Specific Example

A program ZShell is micompiled by the following version of the Clang compiler in snippet 3.21 and causes a ZShell test script to fail in snippet 3.22. ZShell is a large project consisting of multiple source files like in snippet 3.23. These are all the information compiler engineers have when a miscompilation bug is encountered.

```
$ clang -v
clang version 4.0.0 (trunk 278907)
Target: x86_64-unknown-linux-gnu
Thread model: posix
```
Snippet 3.21: A buggy version of Clang

```
*******************************************
60 successful test scipts, 1 failure, 3 skipped
*******************************************
```
Snippet 3.22: Causing a test script to fail

```
> ls zsh-5.9
ChangeLog    Makefile.in    build
Completion   Misc           config.guess
Config       NEWS           config.h.in
Doc          README         config.sub
Etc          Scripts        configure
FEATURES     Src            configure.ac
Functions    StartupFiles   install-sh
INSTALL      Test           make_script.dump
LICENCE      Util           mkinstalldirs
MACHINES     aclocal.m4     stamp-h.in
META-FAQ     aczsh.m4
```
Snippet 3.23: Zshell source directory

By using C-Locate with a corresponding input file, C-Locate should produce output like: 1. The miscompiled source file like snippet 3.24. 2. The buggy optimization pass in the compiler like snippet 3.25. 3. The buggy commit ID in the compiler Repo. like snippet 3.26. And C-Locate-Function is going to output the name of the miscompiled function like snippet 3.27 and extract this function into a single file like snippet 3.28.

```
==================================================
Buggy File(s) are
```

```
/home/FYP/test/testcase7/zsh-5.9/Src/subst.c
==================================================
```

Snippet 3.24: Output the miscompiled source file

```
==============================================
Buggy optimization passes are between
63 and 64
==============================================
Corresponding to
==============================================
BISECT: running pass (61) Simplify the CFG on function (
   paramsubst)
BISECT: running pass (62) SROA on function (paramsubst)
BISECT: running pass (63) Early CSE on function (
   paramsubst)
BISECT: running pass (64) Early GVN Hoisting of
   Expressions on function (paramsubst)
BISECT: NOT running pass (65) Simplify the CFG on function
    (untok_and_escape)
BISECT: NOT running pass (66) SROA on function (
   untok_and_escape)
BISECT: NOT running pass (67) Early CSE on function (
   untok_and_escape)
BISECT: NOT running pass (68) Early GVN Hoisting of
   Expressions on function (untok_and_escape)
==============================================
```

Snippet 3.25: Output the buggy opt pass

```
==============================================
Buggy git commit ID is
9ecdabae840b091f8ab3fd6fe9880
1
==============================================
```

Snippet 3.26: Output the buggy commit ID

```
=======================================
Miscompiled Function is:
paramsubst
=======================================
```

Snippet 3.27: Output the miscompiled function name

```
#include "myHeader.h"
/**/
LinkNode
my_magic_123paramsubst
(LinkList l, LinkNode n, char **str,
int qt, int pf_flags, int *ret_flags)
{
    char *aptr = *str, c, cc;
    char *s = aptr, *ostr = (char *) getdata(n);
    int colf;
    /*...*/
    /*...*/
    /*...*/
    if (eval)
   *str = (char *) getdata(n);
    return n;
}
```

Snippet 3.28: Miscompiled function is extracted to a file

The usefulness, correctness and completeness are discussed in the next chapter of Evaluation, which will discuss how useful is this information in helping compiler engineers diagnose the bug, how correct is C-Locate's output, and how well does C-Locate generalized.

Chapter 4

Evaluation

This chapter will discuss all the evaluations we have done for C-Locate, including discussions on how useful the information that C-Locate provides in helping compiler engineers diagnose a miscompilation bug, how correct is this information, and the complexity of the implemented algorithm.

Evaluation is done by taking a real compiler bug and, either a large program that already triggered this bug or a large program with this compiler bug manually injected. Evaluation has been done on a few compiler bugs and large projects.

4.1 Case Study 1

4.1.1 Test Case Description

This compiler bug is taken from https://bugs.llvm.org/show_bug.cgi?id=29031. There is a small test case provided to triggered this bug, and also ZShell has a section of code that triggered this bug.

Simple Case

Snippet 4.1 shows a compiler bug. For the small test program, a version of clang compiler would do a faulty GVN hoist at O1 or higher optimization level which miscompiled the return value as shown in the snippet 4.2.

```c
int a;
int main()
{
  for (; a < 1; a++)
    for (; a < 1; a++)
      if (a) a++;
  return a;
}
```

Snippet 4.1: Simple case test C program

```
$ clang -O0 small.c; ./a.out; echo $?
2
$ clang-3.8 -O1 small.c; ./a.out; echo $?
```

```
2
# miscompiled case
$ clang -O1 small.c; ./a.out; echo $?
3 # wrong output
```

Snippet 4.2: Simple case miscompiled output

Case in Large Program (ZShell)

This bug is triggered by Zshell, in the function 'paramsubst' of the source file 'subst.c', by the section of code in snippet 4.3. Where the statement "*t = sav" is wrongly hoisted by the same "SVN Hoist" optimization in the simple case. This miscompilation bug causes the test script 'D04Parameter04.ztst' to fail.

```

LinkNode
paramsubst ( LinkList l , LinkNode n , char ** str , int
   qt , int pf_flags , int * ret_flags )
/* ... some code before ... */

    while (*++s) {
        switch (*s) {
        case 'e':
        getkeys |= GETKEY_EMACS;
        break;
        case 'o':
        getkeys |= GETKEY_OCTAL_ESC
        break;
        case 'c':
        getkeys |= GETKEY_CTRL;
        break;

        default:
        *t = sav;
        goto flagerr;
        }
    }
/* ... some code after ... */
```

Snippet 4.3: Case in Zshell

4.1.2 How Much is C-Locate's Outputs Useful?

Given the buggy compiler and ZShell source files as well as the failing test script, how much does each C-Locate's output helps compiler engineers to locate or narrow down the problem?

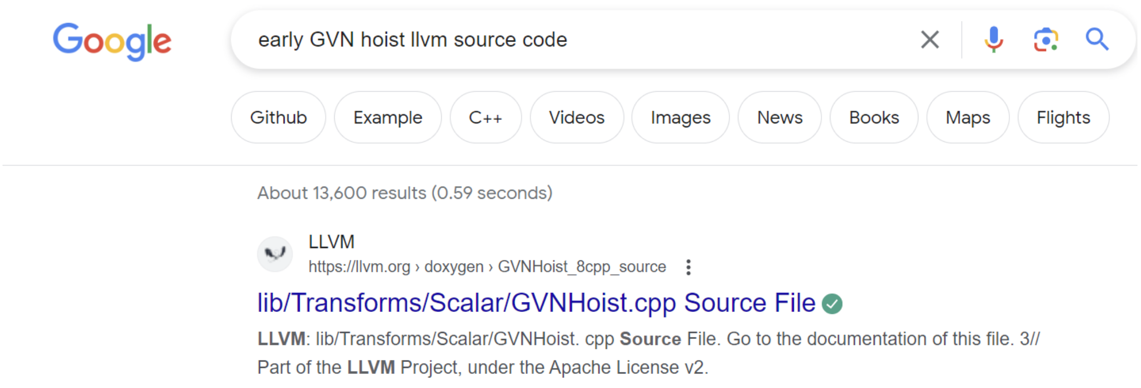
Locate the Miscompiled File

By locating the miscompiled file is "subst.c", C-Locate reduces the possible miscompiled code from more than 70 files to 1 file, which is in this case from more than

80,000 lines of code to around 4000 lines. In this case, we consider C-Locate to be very useful in reducing the problem.

Locate the Buggy Optimization Pass

C-Locate outputs the buggy optimization pass to be "Early GVN Hoisting of Expressions on function (paramsubst)". This information doesn't seem to be directly useful. However, from our experiences of working on compilers, a simple next step would be to Google this optimization pass, and Google would tell you which source file in the compiler source code is responsible for this optimization, just like snippet 4.1. With this information, C-Locate reduces the potential buggy source files that compiler engineers need to inspect from hundreds to 1 file, which is in this case from more than 500,000 lines of code to 1000 lines of code.



About 13,600 results (0.59 seconds)

LLVM
https://llvm.org › doxygen › GVNHoist_8cpp_source

lib/Transforms/Scalar/GVNHoist.cpp Source File

LLVM: lib/Transforms/Scalar/GVNHoist. cpp **Source** File. Go to the documentation of this file. 3// Part of the **LLVM** Project, under the Apache License v2.

Figure 4.1: Google the buggy optimization pass

Locate the Buggy Git Commit ID

We didn't manage to find a buggy compiler that's under a Git repo for this test case because this bug is around 7 years ago, so we didn't test this functionality of C-Locate with this case.

Locate the Miscompiled Function

C-Locate locates the miscompiled function to be "paramsubst", which in this case reduces the miscompiled code that needs to be inspected from 4000 lines to 2000 lines. Moreover, this functionality provides a potential for further development.

Correctness and Completeness of C-Locate's Outputs

For this test case, all C-Locate's outputs are correct, except the functionality of locating the buggy Git commit is not being tested. We compared all C-Locate's outputs with analysis from this paper[2] and also the bug report pages and all result matches.

Time Taken

C-Locate takes a few minutes to run, and C-Locate-Function takes less than 2 minutes to run. In this case, we consider it quite efficient because it only takes the time to make a coffee to locate all the information. Compared to manually diagnosing, C-Locate saves a lot of time.

```
> time ./C-Locate test/testcase7/test_input.config
real    8m0.052s
user    4m57.372s
sys     0m37.249s
```

Snippet 4.4: Time taken to run C-Locate

```
> time ./C-Locate-Funtion input_to_C-Locate-Function.txt
real    1m41.220s
user    1m5.078s
sys     0m7.300s
```

Snippet 4.5: Time taken to run C-Locate-Function

4.2 Case Study 2

4.2.1 Test Case Description

This is a bug recorded on https://github.com/llvm/llvm-project/issues/59650. It's a miscompilation bug in a version of the Clang++ compiler and there is a small C++ program to trigger this bug. We would like to test C-Locate in both C and C++ programs.

Simple Case

This is a miscompilation bug introduced by a loop vectorize optimization, some scalar is wrongly identified as invariant variable and leads to a wrong output in value. When it's compiled with -O3, the output is 3 where as the correct output is 0. This bug is confirmed by LLVM experts and fixed. For more detail, visit the bug report website.

```
/* ... Full code is in appendix */

int main() {
    int result = CheckPadding(data_length41,
                              sizeof(data_length41))
    std::cout << result << std::endl;
}
```

Snippet 4.6: Simple case test C++ program

```
> clang++ bug.cpp && ./a.out
0
# With -O3, the result is miscompiled to 3.
> clang++ bug.cpp -O3 && ./a.out
```

```
3
```
Snippet 4.7: Miscompiled output

Case in Large Program

This bug wasn't found in any large program. To test C-Locate, we inject this bug into a large C++ project, LevelDB. To inject this bug, we randomly chose a source file in LevelDB and injected the bug like the following snippet. We have repeated this process to inject this bug into several source files respectively. We use "filename.cc" as the source file being injected with a bug as an example to demonstrate in this case study. In this case, we insert an assertion of the return value of this function should be correct, so when it's miscompiled, this assertion would fail.

```
static std::string
MakeFileName(const string& dbname,
             uint64_t number,
             const char* suffix) {

  char buf[100];

  /* insert the miscompilation bug here */
  assert((int)CheckPadding(data_length41,
                           sizeof(data_length41))==0);
  /* ---------------------------------- */

  std::snprintf(buf, sizeof(buf), "/%06llu.%s",
                static_cast<unsigned long long>(number),
                suffix);
  return dbname + buf;
}
```
Snippet 4.8: Miscompiled output

This will cause a test script in LevelDB to fail like the following snippet.

```
> ./c_test
c_test: /home/alan/FYP-Reporting-Compilation-Bugs/test/
   testcase8/leveldb/db/filename.cc:132: bool leveldb::
   ParseFileName(const std::string &, uint64_t *, leveldb
   ::FileType *): Assertion `(int)CheckPadding(
   data_length41, sizeof(data_length41))==0' failed.
Aborted
```
Snippet 4.9: Cause a LevelDB test script to fail

4.2.2 How Much is C-Locate's Outputs Useful?

Locate the Miscompiled File

The following snippet is output from C-Locate.

```
=====================================================
Buggy File(s) are
/home/FYP/test/testcase8/leveldb/db/filename.cc
=====================================================
```

Snippet 4.10: Output the miscompiled source file

C-Locate successfully locates the file being miscompiled. In this case, we reduced the miscompiled code from all LevelDB source files to one file, which only contains 195 lines of code. And also, by locating this miscompiled file, we largely reduce to the complexity of running other functionalities of C-Locate.

Locate the Buggy Optimization Pass

The following snippet is output from C-Locate.

```
==============================================
Buggy optimization passes are between
4335 and 4336
==============================================
Corresponding to
==============================================
BISECT: running pass (4332) LoopRotatePass on Loop at
   depth 1 containing: %14<header><latch><exiting>
BISECT: running pass (4333) LoopDeletionPass on Loop at
   depth 1 containing: %14<header><latch><exiting>
BISECT: running pass (4334) LoopDistributePass on
   _Z12CheckPaddingPhj
BISECT: running pass (4335) InjectTLIMappings on
   _Z12CheckPaddingPhj
BISECT: NOT running pass (4336)LoopVectorizePass on
   _Z12CheckPaddingPhj
BISECT: NOT running pass (4337) LoopLoadEliminationPass
   on _Z12CheckPaddingPhj
BISECT: NOT running pass (4338) InstCombinePass on
   _Z12CheckPaddingPhj
BISECT: NOT running pass (4339) SimplifyCFGPass on
   _Z12CheckPaddingPhj
==============================================
```

Snippet 4.11: Output the buggy opt pass

C-Locate outputs the buggy optimization pass to be "(4336) LoopVectorizePass on _Z12CheckPaddingPhj ". This output is correct and the same as the bug report. With this information, just like case study 1, we can largely reduce the problem to a smaller section of code.

Locate the Buggy Git Commit ID

The following snippet is output from C-Locate.

```
==============================================
Buggy git commit ID is
```

```
cedfd7a2e536d2ff6da44e89a024b
1
==================================================
```

Snippet 4.12: Output the buggy commit ID

By comparing to the analysis in the bug report, this is the confirmed buggy commit. And in this case, knowing the commit ID becomes very useful in fixing this bug. By going to this commit, we can see that there are less than 10 lines of change within this commit, and a safety check to do vectorization had been removed. Although there might be a better fix, simply putting the safety check back can fix this bug.

Locate the Miscompiled Function

As mentioned, this functionality C-Locate-Function is only built for C programs right now, this program is a C++ project, so this functionality is not being tested.

Correctness and Completeness of C-Locate's Outputs

As discussed in all previous sections already, outputs from C-Locate are all correct compared to the bug report, except the functionality of locating the miscompiled function is not being tested.

Time Taken

The following snippet is the time taken to run C-Locate:

```
> time ./C-Locate test/testcase7/test_input.config
real    80m0.052s
user    4m57.372s
sys     0m37.249s
```

Snippet 4.13: Time taken to run C-Locate

Although it seems running C-Locate on LevelDB takes such a long time, most of the time is actually spent on running the compilation and building process of levelDB itself. Moreover, although more than hours seems to be a long time, it can be run during a lunch break without any manual operation, which also saves compiler engineers' effort.

4.3 Case Study 3

4.3.1 Fail to Run for SQLite

We would like to test C-Locate on different large programs, so we take **SQLite** and try to inject bugs into it. However, C-Locate **fails** to run in this case. All functionalities have no output anything at all, and we inspect the reason for it.

4.3.2 Reason for Failing

As we've mentioned before, C-Locate relies on Makefiles to echo the compiling commands. However, some Makefiles don't echo compiling commands at all, even running with "make VERBOSE=1", for example, SQLite. In this case, C-Locate fails to run and cannot provide any useful information.

4.4 Case Study 4

4.4.1 Test Case Description

This is a bug reported on https://github.com/llvm/llvm-project/issues/59679. It's a miscompilation bug in a version of the Clang compiler with a small C program to trigger this bug. We inject this known bug into the ZShell project for testing C-Locate.

Simple Case

The following snippet of code triggers this miscompilation bug.

```c
int x_test_only;

__attribute__((noinline))
int test_test_only(int * restrict ptr) {
  *ptr = 1;
  if (ptr == &x_test_only){
    *ptr = 2;
  }
  return *ptr;
}
```

Snippet 4.14: Small test program

This program is miscompiled with the following output:

```
> clang bug.c ; ./a.out
2

# With -O3, this value is miscompiled to 1
> clang bug.c ; ./a.out
1
```

Snippet 4.15: Miscompiled output

Large Program Case

This bug wasn't found in any large program. To test C-Locate, we inject this bug into a large C project, ZShell. To inject this bug, we have randomly chosen a source file in ZShell and injected the bug like the following snippet. Instead of injecting an assertion like in case study 3, we pass on a value to change a variable in this case. We have repeated this process to inject this bug into several source files respectively. We use "sort.c" as an example source file being injected with this bug as an example to demonstrate this case study. In this case, the return value of a string compare function is changed if this file is getting miscompiled.

```
/**/
mod_export int
zstrcmp(const char *as, const char *bs, int sortflags)
{
/* ... Some Operations ...*/

/* Inject a miscompilation bug and pass on a value */
    if (test_test_only(&x_test_only) == 1) {
        ret++;
    }
/* -------------------------------------------------- */

    return ret;
}
```

Snippet 4.16: Inject miscompilation bug in ZShell "sort.c"

4.4.2 How Much is C-Locate's Outputs Useful?

Output of C-Locate

The following snippet is output of C-Locate.

```
=====================================================
Buggy File(s) are
/home/FYP/test/testcase5/zsh-5.9/Src/sort.c
=====================================================
=================================================
Buggy optimization passes are between
45 and 46
=================================================
Corresponding to
=================================================
BISECT: running pass (42) InlinerPass on (test_test_only)
BISECT: running pass (43) PostOrderFunctionAttrsPass on (
   test_test_only)
BISECT: running pass (44) OpenMPOptCGSCCPass on (
   test_test_only)
BISECT: running pass (45) SROAPass on test_test_only
BISECT: NOT running pass (46) EarlyCSEPass  on test_test_only
BISECT: NOT running pass (47) SpeculativeExecutionPass on
    test_test_only
BISECT: NOT running pass (48) JumpThreadingPass on
   test_test_only
BISECT: NOT running pass (49)
   CorrelatedValuePropagationPass on test_test_only
=================================================
=================================================
# This output is a bit suspicious
buggy commits are between
```

```
0 and 25
==============================================
```

Snippet 4.17: Output of C-Locate

The functionality of locating the miscompiled file and the buggy optimization pass still works perfectly, C-Locate outputs the correct and useful information, which largely reduces the code size to be inspected.

The functionality of locating the buggy Git commit ID outputs the whole range being tested. After inspection, we found that this is because the miscompilation bugs are not introduced in a recent commit when it's reported. This bug has been last for long and all the past 25 commits contain this bug. In this case, this functionality is not very useful, but the output is correct.

4.4.3 Output of C-Locate-Function

C-Locate-Function doesn't work well in this case due to the limitation in the implemented algorithm. The current algorithm only considers most of the generic and basic cases, but when the miscompiled source file becomes more complicated, C-Locate-Function might fail to run. In this case, C-Locate-Function fails to run with the following output.

```
/usr/bin/ld: sort.o:
in function ‘my_magic_123strmetasort’: (.text+0xa68):
   undefined reference to ‘eltpcmp’
```

Snippet 4.18: C-Locate-Function error message

The issue is because in function strmetasort, there is a use of function parameter of eltpcmp, like the following snippet:

```
mod_export void
strmetasort(char **array, int sortwhat, int *unmetalenp)
{
/* ... */
/* eltpcmp is a function, and it’s passed as a parameter
to another function */
    qsort(sortptrarr, nsort, sizeof(SortElt), eltpcmp);
/* ... */
}
```

Snippet 4.19: C-Locate-Function error message

To recall, in section 3.2.5, we discuss that C-Locate-Function implements a preprocessing to all function names and function uses. All function uses should be replaced with our magic function. In this case, "eltpcmp" should be replaced with our "magic_eltpcmp". However, the current algorithm in C-Locate-Function doesn't consider the case when a function is used as a function pointer without any parameter, so in this case, it fails to run, which brings a limitation of C-Locate-Function.

4.5 Other Cases

We evaluate C-Locate and C-Locate-Function with many other cases with different ways of injecting miscompilation bugs in different large projects.

4.5.1 C-Locate

In general, C-Locate works well in most cases, i.e. the functionality of locating the miscompiled file, buggy optimization pass and buggy git commit, often output the correct result.

4.5.2 C-Locate-Function

However, the functionality of locating the miscompiled function sometimes doesn't work well, i.e. C-Locate-Function would sometime output the wrong result or fail to run, or in some cases need some human effort. We summarized the following cases where C-Locate-Function fails to run or would need human effort.

Not-Directly-included User-defined Macros

C-Locate-Function might fail to run when there are user-defined macros and not directly included in the file. Some human effort would be needed before running C-Locate-Function to solve this issue. The following snippet is an example of such a case.

```c
#include "zsh.mdh"
#include "subst.pro"
/**/
char nulstring[] = {Nularg, '\0'};
```

Snippet 4.20: Code with not-directly-included user-defined macro

"Nularg" is a user-defined macro in other files. Although "Nularg" is included in the header file "zsh.mdh", "zsh.mdh" is generated from "zsh.h" and Clang libtooling is unable to analyze this kind of structure and will disregard this line of code. A possible simple solution to make C-Locate-Function work is to manually add the definition of "Nularg" in this file. Like the following snippet:

```c
#include "zsh.mdh"
#include "subst.pro"
/**/
/* Add this line of definition manually */
define Nularg          ((char) 0xa1)}

char nulstring[] = {Nularg, '\0'};
```

Uses of Function Pointer

As mentioned in a previous case study, when there are uses of function pointer, the preprocessing of C-Locate-Function would fail and hence C-Locate-Function might fail to run. The following snippet is a generalized example.

```c
int bar(/* with some parameters */) {
/* with some function body */
}

int foo(int a, int b){
/* passing function bar as a function pointer. */
    compare(a, b, bar);
}
```

Snippet 4.21: Code with function pointer usage

In this case and with the current implementation, C-Locate-Function would fail to transform this kind of function pointer used to the created magic function that we want, which would lead to an undefined reference issue when compiling. There are possible solutions which involve improvement of C-Locate-Function implementation, but due to time limits, this kind of problem is not solved now.

Other Cases Not Taken Into Account

In general, the currently implemented algorithm in C-Locate-Function works for most of the basic cases, but there are some cases that have not been taken into account yet. For example, "enum" definitions are not handled at all. There might be other cases we haven't encountered that are not being handled.

4.6 Time Complexity of C-Locate Algorithms

Locating the Miscompiled File

The worst-case complexity of this functionality should be O(n), where n is the total number of source files. However, this might take quite a long time to run because, for each file, there is a compilation process and testing process. So, the actual worst-case running time should be n * (compiling_time + testing_time).

Locating the Buggy Opt Pass

The worst-case complexity of this functionality should be O(log n), where n is the total number of source files. It's log n because we used bisection. We would also need to take compiling time and testing time into account, so the worst-case running time is log(n) * (compiling_time + testing_time).

Locating the Buggy Git Commit ID

The worst-case complexity of this functionality should be O(log 25), where 25 is fixed because we only check the past 25 commits. However, in this case, we need to take into account the time of building the compiler. So, the worst-case running time is log(25) * (build_compiler_time + compiling_time + testing_time). Note that the time taken to build a compiler is usually much longer than other processes.

Locating the Miscompiled Function

The worst-case complexity of this functionality should be O(n), where n is the total number of functions in the miscompiled file. So, the worst-case running time is n * (compiling_time + testing_time).

4.7 Limitations and Potential Future Development

Rely on Echo from Makefile

As mentioned in previous sections, C-Locate relies on the echo of Makefile to extract the compiling commands for source files. In some cases when the Makefile of a project doesn't echo compiling commands, C-Locate would not work. A possibly better way is to analyze the Makefile and generate compiling commands from this analysis instead of relying on echo.

C-Locate-Function Limitations

As mentioned in previous sections, there are some cases where C-Locate-Function doesn't work very well. However, all the issues can be solved by improving the algorithm and adding more handles to different cases.

Time Taken Too Long in Some Cases

In some cases, C-Locate might take a long time to run, although most of the time taken is on the compilation process. This problem can be alleviated using multi-threading. Since all sub-processes can be isolated, C-Locate has the potential to support multi-threading in running all the compilation processes, which might largely reduce the running time.

Not Useful for Link-Time-Optimization Bug

With the current implementation of C-Locate, we are assuming the bug is not with the linker. When the bug is due to a link time optimization, C-Locate might not be very useful in diagnosing this kind of bug.

Provide a Simple Driver to Invoke the Miscompiled Function

A potential further functionality of C-Locate-Function would be, after the miscompiled function is located, to provide a simple driver (a main function) that invokes this miscompiled function to trigger the compiler bug. In this case, a simple test case is created for compiler engineers to diagnose the bug.

Chapter 5

Conclusion

In this project, our primary objective was to develop an automated solution to assist compiler engineers in diagnosing miscompilation bugs in large programs. Our accomplishment lies in the creation of a powerful tool called C-Locate, which effectively locates the miscompiled source file, faulty optimization pass, and buggy git commit, and an experimental tool called C-Locate-Function which locates the specific miscompiled function. We successfully adapted the concepts of delta-debugging and bisecting to be used in this large project. Moreover, we introduced the concept of mixed-compiling, which is an idea to compile a program using both correct and buggy versions of compilers. This approach enabled us to apply delta-debugging to identify the miscompiled source file and miscompiled function. Furthermore, we utilize some existing debugging options by automating them, significantly reducing the manual effort involved. Additionally, we devised a method to divide a single miscompiled source file into three distinct files, which allowed us to isolate and test each function separately, and apply the concept of mixed-compiling and delta-debugging.

Based on evaluation studies that investigated the correctness of C-Locates outputs as well as a thorough analysis of the usefulness of C-Locate's output in diagnosing bugs. The final version of C-Locate is able to produce useful and valid information that can largely reduce the human effort needed in diagnosing miscompilation bugs in large programs by largely reducing the lines of code needed to be inspected in the compiler source code and the test program source code.

In conclusion, the results clearly show that using the idea of delta-debugging and mixed-compiling can effectively be used as a method to diagnose a miscompilation bug and reduce the problem. Furthermore, throughout the development process, we have maintained the project's potential for future expansion. Given sufficient time, all the future enhancements discussed in section 4.7 can be readily implemented.

5.1 Ethical Consideration

Though C-Locate is primarily designed to assist compiler engineers in locating and diagnosing compiler bugs, there is potential for misuse. Some individuals with malicious intentions might exploit it to take advantage of compilation bugs. This issue isn't exclusive to this project, but is a broader concern associated with all testing and debugging work.

Appendix A

First Appendix

```
#include <optional>
#include <string>
#include <string_view>
#include <iostream>

__attribute__((always_inline))
static inline unsigned char
constant_time_ge_8(unsigned int a, unsigned int b) {
  return ~(((unsigned int)(a - b)) >> 15);
}

__attribute__((noinline))
unsigned char CheckPadding(unsigned char *data,
                           unsigned int length)
{
    unsigned int padding_length = data[length - 1];
    unsigned char res = 0;

    for (unsigned int i = 0; i < length - 1; i++) {
        unsigned char mask =
                 constant_time_ge_8(padding_length, i);
        unsigned char b = data[length - 1 - i];
        res |= mask & (padding_length ^ b);
    }

    return res;
}

__attribute__((noinline)) unsigned char CheckPaddingNoVec
   (unsigned char *data, unsigned int length)
{
    unsigned int padding_length = data[length - 1];
    unsigned char res = 0;

#pragma clang loop vectorize(disable)
```

```
    for (unsigned int i = 0; i < length - 1; i++) {
        unsigned char mask = constant_time_ge_8(padding_length, i);
        unsigned char b = data[length - 1 - i];
        res |= mask & (padding_length ^ b);
    }

    return res;
}

__attribute__((aligned(16))) unsigned char data_length41[41] = {
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x03, 0x03, 0x03, 0x03};

__attribute__((aligned(16))) unsigned char data_length40[40] = {
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x03, 0x03, 0x03, 0x03};

int main() {
    std::cout << (int)CheckPadding(data_length41, sizeof(data_length41)) << std::endl;
}
```

Snippet A.1: Case Study 2 small test progrm

Bibliography

[1] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. *Proceedings of 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2011)*, June 2011. https://www.cs.utah.edu/~regehr/papers/pldi11-preprint.pdf.

[2] Alastair Donaldson Cristian Cadar Michael Marcozzi, Qiyi Tang. Compiler fuzzing: How much does it matter? *Proceedings of the ACM on Programming Languages (OOPSLA 2019)*, 3(OOPSLA), 2019.

[3] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools(2 ed.)*. Addison-Wesley, 2006.

[4] Cplusplus.com. https://cplusplus.com/doc/tutorial/introduction/.

[5] Chengnian Sun, Vu Le, Qirun Zhang, and Zhendong Su. Ftoward understanding compiler bugs in gcc and llvm. July 2016. https://web.cs.ucdavis.edu/~su/publications/issta16-compiler-bug-study.pdf.

[6] Junjie Chen, Yanwei Bai, Dan Hao, Yingfei Xiong, Hongyu Zhang, and Bing Xie. Learning to prioritize test programs for compiler testing. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 700–711, 2017. http://hongyujohn.github.io/ICSE17-LET.pdf.

[7] Karine EM. Bug 98630 - seg-fault when using a goto after condition (if). 11 Jan 2021. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=98630.

[8] Karine EM. Bug 94809 - [8/9 regression] different results between gcc-9 and gcc-6. 27 April 2020. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=94809.

[9] Christopher Roman and Zhijing Li. Finding and understanding bugs in c compilers. 21 November 2019. https://www.cs.cornell.edu/courses/cs6120/2019fa/blog/bug-finding/.

[10] Zhendong Su. wrong code at -o1 and above on x86_64-linux-gnu in both 32-bit and 64-bit modes (gvn-hoist). 2016. https://bugs.llvm.org/show_bug.cgi?id=29031.

[11] John Regehr. wrong code from %. 2012. https://bugs.llvm.org/show_bug.cgi?id=13326.

[12] Installing gcc: Testing. https://gcc.gnu.org/install/test.html.

[13] Test-suite guide : Llvm. https://llvm.org/docs/TestSuiteGuide.html.

[14] Official website: Jenkins. https://www.jenkins.io/.

[15] Angel Rivera. Smoke testing in ci/cd pipelines. 18 oct 2021. https://circleci.com/blog/smoke-tests-in-cicd-pipelines/.

[16] Gitlab. What is ci/cd? https://about.gitlab.com/topics/ci-cd/.

[17] R. Hamlet. Random Testing. *Encyclopedia of Software Engineering second edition*. Wiley, 2001.

[18] L. Fredriksen B. P. Miller and B. So. An empirical study of the reliability of unix utilities. *Commun. ACM, 33(12):32–44*, 1990.

[19] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. Test-case reduction for c compiler bugs. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 335–346, 2012.

[20] Andreas Zeller. Isolating cause-effect chains from computer programs. *ACM SIGSOFT Software Engineering Notes*, 27(6):1–10, 2002.

[21] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002.

[22] Ghassan Misherghi and Zhendong Su. Hdd: hierarchical delta debugging. pages 142–151, 2006.

[23] Official website: Llvm opt-bisect-limit. https://llvm.org/docs/OptBisect.html.

[24] Official website: Llvm bugpoint. https://llvm.org/docs/Bugpoint.html.

[25] Official website: Git - git-log. https://git-scm.com/docs/git-log.

[26] Official website: Git - git-bisect. https://git-scm.com/docs/git-bisect.

[27] Official website: Clang - libtooling. https://releases.llvm.org/11.0.0/tools/clang/docs/LibTooling.html.