Compilation Consistency Modulo Debug Information

Theodore Luo Wang
School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
tlwang@uwaterloo.ca

Yongqiang Tian
School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
yongqiang.tian@uwaterloo.ca

Yiwen Dong
School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
yiwen.dong@uwaterloo.ca

Zhenyang Xu
School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
zhenyang.xu@uwaterloo.ca

Chengnian Sun
School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
cnsun@uwaterloo.ca

ABSTRACT

Compilation Consistency Modulo Debug Information (CCMD) is an essential compiler property that a production compiler should support: the compiler should emit the same machine code regardless of enabling debug information. CCMD is vital to developers' experiences with debugging a production binary containing no debug information. To debug such a binary, developers need build another binary with the same compiler flags and enable debug information. Without CCMD, the machine code in the latter binary will be different, which can confuse the debugger, hide the bug, or even cause a miscompilation (as GCC once did with the Linux Kernel).

This paper is the first to introduce to the research community the validation of CCMD, a new research problem that has been overlooked for decades despite its importance. More importantly, we propose the first testing technique Dfusor to automatically validate CCMD for C compilers. At the high level, given a compilable program P as a seed, Dfusor automatically generates compilable program variants via multiple effective program transformations. Such variants can cause a compiler to emit more debug information than it would when compiling P, thus exercising more code paths in the compiler and increasing the chance to find CCMD bugs.

Our extensive evaluations of Dfusor demonstrate that Dfusor can produce variants that exhibit significant increases in the quantity and complexity of the emitted debug information, and thus has found new, real bugs in GCC and LLVM. With a sample of 100 variants derived from distinct seed programs, Dfusor introduces 214% more debug information entries and 36% more distinct debug information entries in the variants than the seeds, and improves the code coverage of GCC and Clang by up to 6.00% and 6.82%. More importantly, Dfusor has found CCMD bugs; within 10 months of development and intermittent testing, Dfusor has found 23 bugs (9 in GCC and 14 in Clang), with 3 confirmed and 18 fixed.

CCS CONCEPTS

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

KEYWORDS

Compiler Testing, Debug Information

ACM Reference Format:
Theodore Luo Wang, Yongqiang Tian, Yiwen Dong, Zhenyang Xu, and Chengnian Sun. 2023. Compilation Consistency Modulo Debug Information. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3575693.3575740

1 INTRODUCTION

Given a source program and its compiled binary, debug information describes the relationship between the source code and the binary code. For example, the relationship includes positions, types and scopes of variables, functions and machine instructions. Debug information can be used by a debugger (*e.g.*, GDB [5], LLDB [33]) to debug a program by investigating the program states at certain program points at runtime.

Debug information is optional, but should be enabled with caution. If enabled, debug information is emitted together with machine code by a compiler, and stored in the binary. However, developers often disable debug information when compiling production binaries (*e.g.*, executables or libraries released to the end users) for two reasons. First, production binaries with debug information are subject to reverse engineering, and are vulnerable to decompilations. Second, enabling debug information can *drastically* increase compilation times and binary sizes: For example, building LLVM 14 with debug information takes 56% more time and produces 30 times larger binaries in file size than building it without debug information. Industrial developers, *e.g.*, Sony Playstation [11], tend to disable debug information to reduce compilation time. The increase in storage requirements also makes enabling debug information infeasible for resource-constrained runtime environments such as embedded or Internet-of-Things devices.

Debugging Production Binaries. As aforementioned, production binaries are usually compiled without debug information. But the shortcoming of doing so is obvious: increased difficulty in debugging production binaries. Assuming that a production binary crashes in the production environment, the operating system then automatically saves the states of the memory, CPU and call stacks, and other environment information into a core dump file [46]. To debug this buggy production binary, the developer needs to recompile the program with the same set of compiler flags but with debug information enabled, and then use a debugger with the new binary and the core dump to reproduce and analyze the bug.

Perhaps less known to developers, debug information can affect the generation of machine code. To emit correct debug information in binaries, compilers record the information (*e.g.*, types, locations, scopes) about symbols and statements during lexing and parsing. Then, compilers carry such information through various optimization algorithms in the middle end, and emit debug information in certain format such as DWARF [3] and PDB [28] during machine code generation in the back end. If debug information is enabled, the need to maintain correct debug information while compiler optimizers are transforming a program may cause the compiler optimizers to behave differently from the scenario where debug information is disabled. Such behavioral differences can further lead to a compiler emitting different machine code, hide application bugs, or in the worse case even trigger miscompilation bugs in the compiler [2, 29, 41, 42].

CCMD. This paper introduces CCMD, the property that compilers should generate the same machine code whether debug information is enabled. If a compiler violates the CCMD property on a program, we say a CCMD bug is triggered in this compiler. To give a concrete example of where debug information changes the generated machine code, consider the compilable program shown in Figure 1a. It is generated by our tool, Dfusor, and triggers a CCMD bug in GCC. Figure 1b highlights the differences in machine code between the binary compiled without debug information (left) and that compiled with debug information (right). The compiler attributes such as `__attribute__((always_inline))` and the `#line` compiler directives complicate the optimization and the debug information of the program, and thus tricked GCC to introduce a difference in a control-flow edge when debug information is enabled by the flag `-g`.

Importance of CCMD. As explained by an LLVM developer, CCMD is an important compiler property as it improves the debugging support of compilers so as to provide smooth, decent debugging experiences to compiler users.

> *The principle here is that if -g changes the generated code, it can potentially hide a bug. That is, you are not actually debugging the exact same program that exhibited the bug you are looking for. One hopes that the two versions of the program (i.e., with and without -g) do have the same behavior, but it is far far better to know that you are debugging the exact same program.* — *by an LLVM developer [35]*

In the aforementioned scenario of debugging a buggy production binary, CCMD ensures that the buggy binary and the debugging binary built with debug information have the same machine code. Any difference in the machine code may cause debugging tools to behave strangely when used on the core dump. For instance, GDB may fail to correctly interpret the core dump with the debugging binary, and show some instructions as "`(bad)`" when disassembling the debugging binary.

Moreover, CCMD can prevent a class of miscompilation bugs that are caused by debug information. In the worst scenario, enabling debug information can cause compilers to miscompile programs [2, 29, 41, 42]. For example, enabling debug information caused GCC 4.9.0 to miscompile the Linux kernel [41] because the presence of debug instructions led to erroneous data dependencies in the instruction scheduling pass, and further led to the register allocator incorrectly spilling a variable before the stack frame was created. On the other hand, GCC 4.9.0 can correctly compile the

```c
# line 6 ""
short a;
__attribute__((optimize(0))) int b();
__attribute__((always_inline)) signed char c();
int b() { char e = c(); return a; }
signed char c(f) {
  d();
# line 7 "attributes_transformed_program0.c"
 int g;
# line 6
 return f;
}
void d() {}
int main() {}
```

(a) A Dfusor-generated program that triggers a CCMD bug (GCC-104237).

```
<b>:
......
mov    -0x8(%rbp),%eax


mov    %al,-0x1(%rbp)
 movzwl 0xeed(%rip),%eax
cwtl
......
<main>:
......
nopw
%cs:0x0(%rax,%rax,1)
 nopl (%rax)
```

```
<b>:
......
mov    -0x8(%rbp),%eax
 nop
mov    %al,-0x1(%rbp)
 movzwl 0xeec(%rip),%eax
cwtl
......
<main>:
......
nopw
%cs:0x0(%rax,%rax,1)
 xchg %ax,%ax
```

(b) Assembly with debug information (left) and that without (right).

Figure 1: (a) is a *minimized* program triggering a CCMD bug, in which each token is necessary to trigger the bug. The left and right of (b) are excerpts of the binaries compiled without debug information (`-O1 -flto`) and with debug information (`-O1 -flto -g`), respectively. The pink lines highlight the differences between the two binaries.

Linux kernel without debug information. Better support for CCMD in GCC could have prevented this miscompilation by keeping the assembly instructions invariant regardless of whether debug information is emitted.

Lastly, compiler developers value CCMD. GCC has implemented CCMD decently and LLVM is in the process of supporting CCMD [4, 7]. LLVM has a script [31] to validate CCMD on their test suite and GCC has a special `-fcompare-debug` flag [12] for the same purpose. Both compiler communities have been actively fixing CCMD bugs.

Dfusor. Despite the importance of the CCMD property to developers' experiences with compilers, no prior research effort has been put into validating the reliability of CCMD. Therefore, this paper proposes the first such research effort, Dfusor, an effective testing technique to automatically validate CCMD for C compilers. Note that finding CCMD bugs is inherently a research problem aiming at testing the correctness of compiler optimizers *w.r.t.* debug information. Specifically, to find CCMD bugs effectively, there are two orthogonal, technical challenges: ① *how to generate programs with complex logic that can trigger diverse code paths in compiler optimizations*, and ② *how to generate programs with complex debug information that can thoroughly exercise code paths related to debug information in compiler optimizers*. To this end, at the high level, given a compilable program as a seed, Dfusor automatically generates another different, compilable program variant via multiple program transformations (*i.e.*, Randomized Code Lowering, Obfuscation of Code Positions, and Fine Control of Optimizations in §3) to introduce syntactical and semantic complications into the source code, to fully explore and exploit different behaviors of compiler optimizers.

We conducted extensive evaluations of Dfusor. First, we compared the debug information of the same number of seeds and variants generated by Dfusor. The results demonstrated that Dfusor significantly complicates the debug information, resulting in 214% more debug information entries and 36% more distinct ones. Second, the Dfusor-generated variants effectively exercise more code paths in compilers, 6.00% and 6.82% more branches of GCC and Clang triggered by the variants than the seeds, respectively. More importantly, Dfusor has found CCMD bugs. Within 10 months of development and intermittent testing, Dfusor has found 9 bugs in GCC and 14 ones in Clang, with in total 21 already confirmed or fixed. These bugs are distributed among different components of compilers, but are mainly in the middle end and back end.

Contributions. We make the following contributions.

- We introduce CCMD to the research community, a vital property that mature compilers should strive to support to provide excellent debugging experiences to developers.
- We propose Dfusor, the first research effort to automatically validate CCMD for C compilers. Dfusor is a mutation-based testing technique, empowered by three novel program transformations that apply syntactical and semantic mutations on seed programs to generate complex, diverse test programs.
- The evaluation results strongly demonstrate that Dfusor-generated test programs have complex debug information and can exercise different code paths in compilers. More importantly, Dfusor has found new, real bugs in real-world production compilers: 23 bugs in GCC and Clang, with 21 of them confirmed or fixed. Our testing efforts have been well supported and appreciated by the compiler developers.

2 BACKGROUND AND FORMULATION

2.1 Debug Information

To facilitate debugging binary programs, modern compilers emit debug information during compilation. Debug information is usually stored in a format, such as DWARF [3], PDB [28] or STAB [27]. These formats record the metadata of the source code, such as the names and line numbers of each function and variable. Debug information provides a debugger with a mapping between source code and the binary, so that developers can easily manipulate program execution and investigate program states in a debugging session.

As mentioned in §1, developers usually build a debugging binary to analyze a core dump file from a production binary crash. A core dump file records the environment (*e.g.*, memory, registers, stacks) at the crash, and is highly specific to the machine code of the crashing binary. Assume that the debug information is correct. If the debugging binary and the crashing binary have the same machine code, the debugger can render an accurate, intelligible view of the program state upon crash to the developers. The debugger may otherwise behave erroneously, for example, showing some machine instructions as `(bad)` when disassembling the debugging binary.

GCC and Clang have a command-line flag `-g` to control the emission of debug information [6, 32]. An optional integer *level* can be appended to specify the verbosity of debug information; *level* is 0, 1, 2 (default) or 3, from no debug information (`-g0`) to the most detailed debug information (`-g3`).

2.2 Compiler Optimizations

Modern production compilers, such as GCC and Clang, have an optimization stage to optimize the code during compilation, so that the compiled binary uses fewer resources and runs faster than the unoptimized binary. The optimization stage usually consists of a large number of modular optimizing procedures (referred to as passes in GCC and LLVM). For example, the *dead instruction elimination* pass in LLVM removes instructions that are in dead code regions during execution.

It is well known that compiler optimizers have intricate algorithms dedicated to optimizing programs for performance, but perhaps less known is that compiler optimizers also need to handle debug information during optimization. To strive to emit correct debug information, in the front end, compilers store the information (*e.g.*, types, locations, scopes) about symbols and statements during lexing and parsing. In the middle end, compilers run various optimization passes, which are expected to not only optimize the code, but also correctly update the debug information simultaneously so that the debug information is always faithful to the source code. In the back end, debug information is emitted in certain format such as DWARF [3] along with machine code. If any optimization pass fails to properly handle the case where debug information is enabled and the case where it is disabled, compilers are likely to generate different machine code, resulting in a CCMD bug.

2.3 Problem Formulation

We use the following notations to facilitate formalizing the research problem of validating CCMD for compilers.

$\mathbb{L}$	a programming language
$\mathbb{O}$	the set of all possible sets of optimization flags
$\mathbb{G}$	{`-g0`, `-g1`, `-g2`, `-g3`} as described in §2.1
$\mathbb{I}$	the set of all possible sequences of machine instructions
$\mathbb{D}$	the set of all possible sequences of debug information entries
$\mathbb{C}$	$\mathbb{C} : \mathbb{L} \times \mathbb{O} \times \mathbb{G} \to \mathbb{I} \times \mathbb{D}$ is a compiler

A compiler $\mathbb{C}$ for the language $\mathbb{L}$ is a function. It takes as input (p, o, g) where $p \in \mathbb{L}$ is a program, $o \in \mathbb{O}$ is a set of optimization flags and $g \in \mathbb{G}$ is a flag controlling emission of debug information; it outputs a pair (i, d) where $i \in \mathbb{I}$ is a sequence of instructions and $d \in \mathbb{D}$ is a sequence of debug information entries.

Definition 2.1 (CCMD). *A compiler $\mathbb{C}$ is said to support CCMD if and only if*

$$\forall p \in \mathbb{L}, \forall o \in \mathbb{O}, \forall g_1 \in \mathbb{G}, \forall g_2 \in \mathbb{G}.$$
$$(i_1, d_1) = \mathbb{C}(p, o, g_1) \wedge (i_2, d_2) = \mathbb{C}(p, o, g_2) \implies i_1 = i_2$$

Namely, the machine instructions i_1 in $(i_1, d_1) = \mathbb{C}(p, o, g_1)$ is the same as i_2 in $(i_2, d_2) = \mathbb{C}(p, o, g_2)$ for any p, o, g_1 and g_2.

Validation of CCMD. Given the definition above, the research problem of validating CCMD for a compiler $\mathbb{C}$ is to find two inputs (p, o, g_1) and (p, o, g_2) such that the sequence of instructions i_1 in $(i_1, d_1) = \mathbb{C}(p, o, g_1)$ and i_2 in $(i_2, d_2) = \mathbb{C}(p, o, g_2)$ are different.

Note that the problem of validating CCMD is different from prior work on testing compiler optimizations [22, 23, 26, 37, 48] and testing debug information [8, 25].

- The problem of testing compiler optimizations is to find a *well-defined* program $p \in \mathbb{L}$ as well as a set of flags $o \in \mathbb{O}$ and $g \in \mathbb{G}$, such that the emitted binary program i in $(i, d) = \mathbb{C}(p, o, g)$ has different semantics from p. If such a program is found, then a bug in the compiler optimizers is found because a correct compiler should emit a binary program with the same semantics as the source program p.
- Testing debug information is to find an input $p \in \mathbb{L}$, $o \in \mathbb{O}$ and $g \in \mathbb{G} \backslash \{-g0\}$, such that the debug information d in $(i, d) = \mathbb{C}(p, o, g)$ does not correctly describe p.

Neither of the problems above tackles the problem of finding CCMD bugs. This paper is the first such effort.

3 APPROACH

Figure 2 shows the general workflow to validate CCMD of a compiler. At the beginning, in ① we use a program generator such as Csmith [48], tkfuzz [38] and Hermes [37] to generate a random seed program P. In ②, we use the program transformations described in this section to mutate P, so that the mutated program variant P′ can trigger compilers to generate more complex debug information and exercise more code paths of compiler optimizers during compilation than P. In ③ and ④, two binaries are generated by compiling P′ with the same optimization flags but with different debugging flags. Lastly, we compare the machine code of the two binaries in ⑤. If there is any difference, a CCMD bug is found.

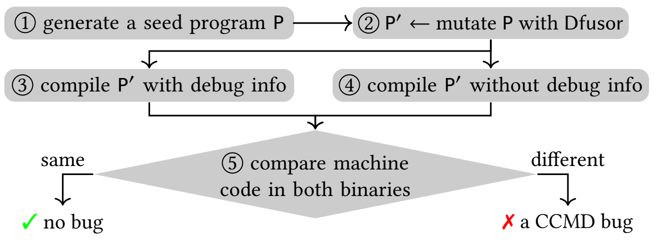

① generate a seed program P
② P′ ← mutate P with Dfusor
③ compile P′ with debug info
④ compile P′ without debug info
⑤ compare machine code in both binaries
same
different
✓ no bug
✗ a CCMD bug

Figure 2: Overall workflow to validate CCMD.

This section details the program transformations used in Dfusor, which are designed to complicate debug information and trigger different code paths of compiler optimizers so as to find CCMD bugs. As described in §2.1, debug information is the metadata of source code, describing various aspects (*e.g.*, names, types, locations) of program elements (*e.g.*, variables, functions). To thoroughly test how reliably a compiler handles debug information, we need to generate test programs that require complex, diverse debug information to describe the program elements in the test programs; meanwhile these test programs should be complex enough to fully exercise different behaviors of compiler optimizers, because CCMD bugs are essentially bugs in compiler optimizers as described in §2.2.

3.1 The While Language

To facilitate presentation, we use an imperative While language to illustrate the core concepts of the program transformations used in Dfusor. Note that the implementation of these program transformations supports all features of C.

$\langle stmt \rangle$	::=	$\langle var \rangle$ `=` $\langle expr \rangle$ | $\langle label \rangle$ `:` $\langle stmt \rangle$ | `{` $\langle stmt \rangle^*$ `}`
	|	`if` $\langle expr \rangle$ $\langle stmt \rangle$ `else` $\langle stmt \rangle$
	|	`while` $\langle expr \rangle$ `do` $\langle stmt \rangle$ | `goto` $\langle label \rangle$
$\langle var \rangle$	::=	variables
$\langle expr \rangle$	::=	$\langle var \rangle$ | $\langle literal \rangle$ | $\langle expr \rangle$ $\langle bop \rangle$ $\langle expr \rangle$ | $\langle uop \rangle$ $\langle expr \rangle$
$\langle label \rangle$	::=	statement labels
$\langle literal \rangle$	::=	literals, *e.g.*, strings and numbers
$\langle bop \rangle$	::=	binary operators, *e.g.*, `+`, `-`, `*`, `and`, `or`, `>`, `>>`
$\langle uop \rangle$	::=	unary operators, *e.g.*, `!`, `-`

Figure 3: Syntax rules for a While language.

Figure 3 lists the syntax rules for the While language: statements $\langle stmt \rangle$, expressions $\langle expr \rangle$, binary operators $\langle bop \rangle$ and unary operators $\langle uop \rangle$. The notation `{` $\langle stmt \rangle^*$ `}` represents a compound statement that has zero or more statements, and `{}` represents an empty statement that does nothing.

3.2 RCL: Randomized Code Lowering

Given a seed P, Randomized Code Lowering (RCL) aims to generate more variables and diversify the syntactical structures of P. RCL automatically transforms P into a new variant P′ by randomly selecting statements and expressions in P and lowering each of them into multiple small statements. For example, a `while` loop can be converted to its equivalent form comprising `if` and `goto`

Algorithm 1: Randomized Statement Lowering

```
fun LowerStmt(stmt):
    result ← [ ] // initially, an empty list
    LowerStmtRec(stmt, result)
    return result
fun LowerStmtRec(stmt, result):
    if FlipCoin() then                    // randomly skip lowering stmt
        result.Append(stmt)
        return
    switch stmt do
        case $var = expr$ do
            $e'$ ← LowerExpr($expr$, result)
            result.Append($var = e'$)
        case $label$ : $s$ do
            result.Append($label$ : {})
            LowerStmtRec($s$, result)
        case { $stmt\_list$ } do
            $new\_stmts$ ← [ ]
            foreach $s \in stmt\_list$ do
                LowerStmtRec($s$, $new\_stmts$)
            result.Append({ $new\_stmts$ })
        case if $expr$ $s_1$ else $s_2$ do
            $e'$ ← LowerExpr($expr$, result)
            $lt, lf$ ← create two unique labels
            result.Append(if $e'$ goto $lt$ else goto $lf$)
            result.Append($lt$ : {})
            LowerStmtRec($s_1$, result)
            result.Append($lf$ : {})
            LowerStmtRec($s_2$, result)
        case while $expr$ do $body$ do
            $head, lt, lf$ ← create three unique labels
            result.Append($head$ : {})
            $e'$ ← LowerExpr($expr$, result)
            result.Append(if $e'$ goto $lt$ else goto $lf$)
            result.Append($lt$ : {})
            LowerStmtRec($body$, result)
            result.Append(goto $head$)
            result.Append($lf$ : {})
        otherwise do result.Append(stmt)
```

statements; a long expression `a+b*c` might be transformed to one statement `t=b*c` and one expression `a+t`. Such transformations create additional variables in the resulting variant, and introduce different syntactical structures.

Algorithms 1 and 2 describe how to transform a statement stmt and an expression expr, respectively. Both algorithms are similar to the classical algorithms [1] in standard compilers to convert source code to intermediate representations (*e.g.*, three-address code), but with noticeable differences. At the high level, the purposes are different: Algorithms 1 and 2 aim to *diversify* syntactical structures; in contrast, the classical algorithms aim to *simplify* the syntactical structures for program analysis and optimizations by *uniformly* transforming different language constructs into a small set of syntactical structures (*e.g.*, all loop statements transformed to `if` and `goto` statements, all complex expressions transformed to either binary or unary expressions). At the low level, Algorithms 1 and 2 are non-deterministic, to make each run of the algorithms able to generate programs with different syntactical structures, whereas the classical compiler algorithms are deterministic.

Algorithm 2: Randomized Expression Lowering

```
fun LowerExpr(expr, stmt_list):
    if FlipCoin() then return expr
    switch expr do
        case $uop\ e$ do
            $e'$ ← LowerExpr($e$, stmt_list)
            $var$ ← create a variable with a unique name
            stmt_list.Append($var = uop\ e'$)
            return $var$
        case $e_1\ bop\ e_2$ do
            $e_1'$ ← LowerExpr($e_1$, stmt_list)
            $e_2'$ ← LowerExpr($e_2$, stmt_list)
            $var$ ← create a variable with a unique name
            stmt_list.Append($var = e_1'\ bop\ e_2'$)
            return $var$
        otherwise do return expr
```

3.3 OCP: Obfuscation of Code Positions

A code position is a pair (`file`, `line`), specifying the exact position of a syntactical element. Code positions account for a large portion of a program's debug information. The program transformation Obfuscation of Code Positions (OCP) aims to complicate debug information by obfuscating the positions of program elements. Although the transformation RCL in §3.2 complicates the code locations to some extent as RCL breaks down complex statements and expressions and introduces new variables, RCL is not aggressive enough.

3.3.1 Methods to Obfuscate Code Positions. Given the seed program P in Figure 4, there are multiple ways to change code positions. The right side of Figure 4 shows the code position information of P. We use P to illustrate the following methods, as well as the strengths and weaknesses of each method.

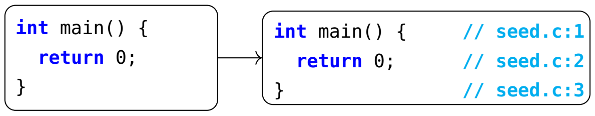
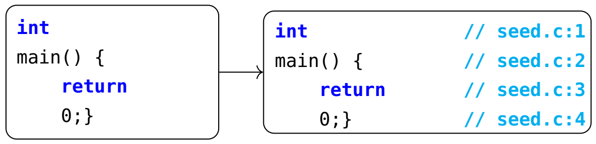

```
int main() {
  return 0;
}
```
```
int main() {      // seed.c:1
  return 0;       // seed.c:2
}                 // seed.c:3
```

Figure 4: `seed.c` (left) and its code positions (right).

Method 1: Mutating Whitespaces. The most naive way is to randomly insert or delete whitespaces (*i.e.*, newlines, tabs, spaces) betweeen two tokens in P. Figure 5 shows such a variant generated by mutating the whitespaces in P. The comments show the code positions of all lines. It is easy to implement this transformation, but its weakness is obvious: it only changes line information and it has no effect on file information.

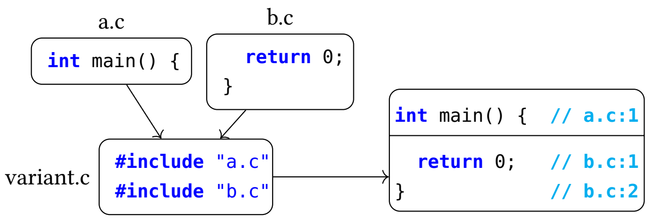


```
int
main() {
    return
    0;}
```

```
int               // seed.c:1
main() {          // seed.c:2
    return        // seed.c:3
    0;}           // seed.c:4
```

Figure 5: Variant (left) by mutating whitespaces at random locations in `seed.c` and the resulting code positions (right).

Method 2: Mutating by Splitting Files. Figure 6 shows another way to obfuscate code positions: splitting a single-file program into multiple files and using `#include` to combine the files together to reassemble the program. The code on the left shows the resulting program with mutated code positions. The advantage of this method over Method 1 is that the file information is mutated. Note that the first line of the code locations in Figure 6 comes from file `a.c` whereas the other two lines are from `b.c`. However there are also several noticeable disadvantages. First, the file name in the `#include` directives has to point to a real, existing file; otherwise the resulting variant does not compile. Second, a large seed file might result in a variant that quite a few files, which complicates the compilation and testing processes and incurs I/O overhead. These challenges can be exacerbated if we aggressively mutate code locations, leading to generate many small files.



```
int main() {
```

```
  return 0;
}
```

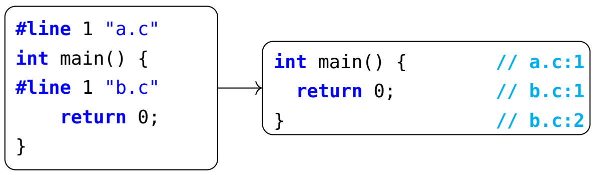
```
#include "a.c"
#include "b.c"
```

```
int main() {   // a.c:1
  return 0;    // b.c:1
}              // b.c:2
```

Figure 6: Variants (left) by splitting `seed.c` to `variant.c`, `a.c` and `b.c`, and the resulting code positions (right).

Method 3: Mutating with Compiler Directive `#line`. This method takes advantage of the C/C++ compiler directive `#line`, which can directly change the file and line information of a line [15]. For example, a directive `#line 999 "foo.c"` changes the code position to `("foo.c", 999)` even if the file `foo.c` does not exist. To achieve the same effect as Figure 6, we only need to insert two `#line` directives into P, as shown in Figure 7. Using `#line` directives has several key advantages compared to Method 1 and 2. First, it is much simpler to implement than Method 1 and 2. Second, it only produces a single source file, which makes compilation, testing and reduction easier than Method 2. For example, once a CCMD bug is found, a single-file program that triggers the bug is also easier to be reduced than a program with multiple files [22, 24]. Third, we can use any random string as a file name, and let compilers emit debug information for the random file name.

Please note that the directives `#line` are more common than one might expect, although it may look unnatural. Specifically, it is prevalently used in compilation. When compilers pre-process macros or includes, compilers automatically insert `#line` directives to specify the positions of the expanded code [18]. The parser generator Bison [9] also uses `#line` to associate parsing errors with lines in the grammar file instead of the generated parser code.



```
#line 1 "a.c"
int main() {
#line 1 "b.c"
    return 0;
}
```

```
int main() {      // a.c:1
  return 0;       // b.c:1
}                 // b.c:2
```

Figure 7: Variant (left) by mutating `seed.c` using compiler directive `#line` and the resulting code positions (right).

3.3.2 *OCP.*

Dfusor combines Method 1 and 3. Given a seed, Dfusor randomly selects a set of lines, and for each line inserts a randomly generated `#line` directive with a random string as the file path and a random integer as the line number. In addition, Dfusor randomly breaks a long line into multiple shorter lines and randomly inserts whitespaces between tokens.

3.4 FCO: Fine Control of Optimizations

As discussed in §2, the root cause of CCMD bugs is usually that compiler optimizations behave differently between when debug information is enabled and when it is disabled. RCL in §3.2 and OCP in §3.3 both strive to complicate debug information of seed programs in hope that *the code paths that handle debug information* in compilers can be thoroughly exercised. Orthogonally, the transformation Fine Control of Optimizations (FCO) in this section aims to thoroughly exercise *the optimization logic* in the middle end and back end.

Most compilers provide coarse-granularity control of optimizations with command-line flags. For example, both GCC and Clang have `-O0`, `-O1`, `-O2`, `-O3` and `-Os`, each of which enables a set of compiler optimizations. They even provide flags to control specific, individual optimization passes (*e.g.*, `-funswitch-loops` to enable loop unswitching [16]).

These coarse-granularity optimization flags are useful to detect CCMD bugs, but are not sufficient to fully trigger different code paths in compiler optimizations. To this end, we propose fine control of optimizations. Concretely, we leverage the compiler attributes to explicitly influence the decisions of compiler optimizers. For example, we can use `__attribute__((always_inline)) signed char c();` to instruct the compiler to always inline calls to `c()`, as shown on line 4 in Figure 1a.

Algorithm 3 lists the main procedure used in Dfusor to do fine-granularity control of optimizations by randomly inserting the following four categories of compiler attributes into the seed program. Currently, Algorithm 3 supports inserting the following four categories of compiler attributes:

- Type Attributes (line 2–4) specify how types should be aligned or whether they should be packed, which might change the memory layout of the members defined in types.

Algorithm 3: Fine Control of Optimizations

```
Input  : P, a seed program
Output : P, the same seed program with various attributes
         inserted in place
1  foreach user-defined type $t$ in P do
2      if FlipCoin() then continue
3      $S_t$ ← randomly sample a set of type attributes
         applicable to $t$
4      Insert each of $S_t$ before the definition of $t$ in P.
5  foreach function $f$ in P do
6      if FlipCoin() then continue
7      $S_f$ ← randomly sample a set of function attributes
         applicable to $f$
8      Insert each of $S_f$ before the declaration of $f$
9  foreach function $f$ in P do
10     foreach variable declaration $v$ in $f$ do
11         if FlipCoin() then continue
12         $a$ ← randomly sample one variable attribute
             applicable to $v$
13         Insert $a$ before $v$
14     foreach label statement $l : s$ in $f$ do
15         if FlipCoin() then continue
16         $a$ ← randomly sample one label attribute
17         Insert $a$ in between $l$ and $s$
```

- Function attributes (line 6–8) allow developers to specify certain function properties which may help compilers optimize the annotated functions. *e.g.*, instructing the compilers to inline or specially optimize the annotated functions.
- Variable attributes (line 11–13) specify extra requirements of the variables that compilers need to accommodate in binaries, *e.g.*, alignment and sections.
- Label attributes (line 15–17) can specify whether the code region following a label is frequently executed or not, thus influencing the optimizing decisions of compiler optimizers.

Table 1 shows the compiler attributes currently used in Algorithm 3. Note that Algorithm 3 is general, and can be easily extended to support other compiler attributes.

Table 1: The compiler attributes used in Algorithm 3.

Category	Attribute Names
Type [19]	`aligned`, `packed`
Function [13]	`aligned(x)`, `always_inline`, `noinline` `artificial`, `cold`, `hot`, `flatten`, `optimize(x)`
Variable [20]	`aligned`, `packed`, `section(x)`
Label [14]	`hot`, `cold`

4 EVALUATION

This section describes our extensive evaluations of Dfusor to find CCMD bugs in two real-world compilers: GCC and Clang. It also presents a thorough analysis of the efficacy of different program transformations in Dfusor.

4.1 Testing Setup

Hardware. We conducted the evaluations on a Linux desktop running Ubuntu 20.04 LTS, with an Intel Core i5-11400 @2.60GHz CPU and 64G RAM.

Compiler Versions. We used Dfusor to test the latest development versions of GCC and Clang by building compiler binaries daily from the repositories of GCC and Clang. As stated in [22, 48], compared to testing stable releases, testing the latest development versions immediately helps developers find bugs early, and facilitates the testing process by preventing reporting duplicate bugs because developers tend to fix bugs quickly in repositories. We used a variety of compiler options for finding CCMD bugs, such as `-O0` to `-O3`, `-Os`, `-g1` to `-g3`, `-flto`, and `-flto=thin`.

Seed Programs. We used three program generators to generate random seed programs:

- Csmith [48] is a random C program generator.
- tkfuzz [38] is a mutation-based program generator which takes as input a seed program and outputs a different program by randomly mutating the seed. Following [38], we used the regression tests of GCC and Clang as seeds for tkfuzz to generate random programs.
- Hermes [37] is a mutation-based program generator similar to tkfuzz but with different mutation algorithms. Following [37], we used Hermes to generate random programs by mutating Csmith-generated programs.

Note that Dfusor is orthogonal to and independent of the source of seed programs. We can also use open-source code as seeds. However, the major obstacle of doing so is that open-source programs are usually large and consist of multiple source files, and thus cannot be effectively reduced by the state-of-the-art reduction tools, a similar problem encountered in [22].

Testing Process. Our testing process is fully automated and runs continuously. In each iteration, we first use a program generator to generate a seed, then apply Dfusor to derive a number n of variants from the seed (we set $n = 10$ by following [22]), and use them to test GCC and Clang. Once a variant triggers a CCMD bug, we use C-Reduce [34] and Perses [39] to reduce the variant. If the variant triggers a new CCMD bug, we report the bug to the developers. The only manual step in the testing is analyzing and reporting bugs.

Testing Period. This project of finding CCMD bugs started from May 2021. In the past 10 months, we tested GCC and Clang non-continuously, because at least half of the time was devoted to designing, developing, testing, and refining Dfusor.

4.2 Quantitative Results

Bugs Found by Dfusor. As Table 2 shows, Dfusor found 23 bugs in total: 9 in GCC and 14 in Clang; 21 bugs have been confirmed or fixed. Both GCC and Clang developers treat our reported bugs seriously: the GCC developers confirmed all bugs and fixed 7 of 9 quickly; only 2 bugs in Clang are still awaiting confirmation, and 11 of 14 are already fixed.

Bug Breakdown by Detection Techniques. Figure 8 shows how these bugs are found. The numbers in the intersections refer to the

Table 2: The bugs found by Dfusor.

Bug ID	Status	Flags	Techniques
Clang			
49771	Fixed	`-O1 -g2`	RCL
49924	Fixed	`-O1 -g2`	RCL
50454	Fixed	`-O3 -flto -g2`	RCL
50525	Confirmed	`-O0 -g2`	RCL
50539	Fixed	`-O1 -flto=thin -g2`	RCL
50540	Fixed	`-O2 -flto -g2`	Csmith
50911	Fixed	`-O0 -g2`	Hermes
50912	Fixed	`-O1 -g2`	tkfuzz
51675	Fixed	`-O2 -g2`	OCP
51880	Fixed	`-O -g2`	RCL
51881	Unconfirmed	`-O2 -flto -g2`	Csmith
51882	Fixed	`-O -g2`	RCL
52711	Unconfirmed	`-O0 -g2`	RCL+OCP+FCO
53937	Fixed	`-O -g2`	Hermes
GCC			
100464	Fixed	`-O3 -g2`	Csmith
100781	Fixed	`-O2 -g2`	RCL
102764	Fixed	`-O3 -g2`	Hermes
104156	Fixed	`-O3 -g2`	tkfuzz
104178	Confirmed	`-O3 -g2`	tkfuzz
104237	Fixed	`-O1 -flto -g2`	RCL+OCP+FCO
104337	Fixed	`-O0 -m32 -g3`	RCL+OCP+FCO
104589	Fixed	`-O0 -flto -g2`	RCL+OCP+FCO
107030	Confirmed	`-O2 -flto -g2`	FCO

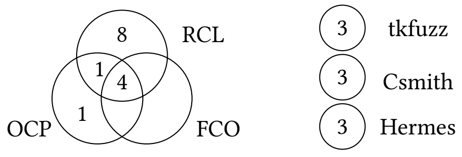


Figure 8: The number of bugs detected by program transformations (left) and seed program generators (right).

number of bugs that are found only when multiple transformations are applied together. For example, four bugs are found by Dfusor when all the three transformations–RCL, OCP and FCO–are applied. Although nine bugs are found directly by the seed program generators (three bugs by each of tkfuzz, Csmith and Hermes), the majority of the bugs (*i.e.*, fourteen) are *only* found by Dfusor. RCL is the most effective technique which finds the most bugs.

4.3 Analysis of Root Causes

To better understand the characteristics of CCMD bugs, we studied the root causes of the reported bugs by reading the fixing commits and the feedback from the compiler developers.

Identifying Fixing Commits. Among the 18 fixed bugs, all 7 GCC bugs have fixing commits, whereas only 4 Clang bugs have fixing commits. For the other 7 fixed Clang bugs without fixing commits, we are not confident that the fixing commits located via `git bisect` is accurate. Thus, in this study we only studied the 11 fixed bugs with explicit fixing commits, including 7 of 9 GCC bugs and 4 Clang bugs. Table 3 lists the commits as well as the analysis of the root causes and fixes.

Table 3: The fixing commits and the root causes of the bugs found by Dfusor.

Bug ID	Commit	Root Cause/Fix
Clang: Middle End		
49771	4316b0e	Debug instrinics were not ignored in loop strength reduce pass.
Clang: Back End		
49924	e90c6f5	Debug intrinsics invalidated registers in machine copy propagation.
51675	c7c84b9	Underflow during DWARF emission caused by a missing width check.
53937	0f56ce0	Debug info passes expected a different variable location mode than the one produced by SelectionDAG.
GCC: Middle End		
100464	a076632	Debug statements caused GIMPLE folding to erroneously set expressions as TREE_ADDRESSABLE.
100781	715914d	Debug statements caused new values to be calculated in value range propagation.
104156	f953c8b	Debug statements were not ignored when looking for out-of-loop uses in the loop unswitching pass.
104337	1d5c758	Variables had their abstract location set incorrectly in return value optimization.
GCC: Back End		
102764	972ee84	In the pass lowering GIMPLE to RTL, debug instructions at the end of a basic block were erroneously considered when setting the current location for outgoing edges.
104237	430dca6	Debug information added extra instruction locations in LTO, leading to a difference in the locations at the ends of a CFG edge.
104589	2e1b003	Similar to the issue exposed by GCC-104237. Fixed by expanding the solution in GCC-104237 to more cases.

Locations of Bugs. As shown in Table 3, all these 11 bugs have root causes in the middle and back end of the compiler. Among all the 23 reported bugs, only one of the confirmed bugs (Clang-50525) has received comments from compiler developers stating that the root causes are in the front end. This distribution of the causes of these bugs is consistent with the characteristics of CCMD bugs as stated in §1: CCMD bugs are usually caused by incorrect handling of debug information inside compiler optimizers.

Commonalities of Fixes. All fixes in Table 3 correct certain mishandling of debug information in compiler optimizations, and some fixes share commonalities. For example, in Clang-49771\49924 and GCC-100464\100781\102764\104156, debug information should be but is not ignored in certain compiler optimizations. For Clang-53937 and GCC-104237\104589, the presence of debug information breaks assumptions of compiler optimizers. As for Clang-51675 and GCC-104337, the variables in compilers related to debug information are incorrectly updated.

In-depth Analysis of Two Bugs. To provide a more concrete understanding of CCMD bugs, we detail the root causes of the following two CCMD bugs.

GCC-104237: Figure 1 shows the test program. GCC inserts a `nop` due to differences between the locations of two instructions connected by a CFG edge. The CFG edge of interest here is in the function `int b()`, where the function `signed char c(f)` is inlined (the inlining is instructed by the compiler attribute on line 4). Instruction locations are streamed into a cache from `lto`, and enabling `-g` leads to two extra cache entries from the variable `g` (highlighted in cyan). This leads to a change in the location for an instruction corresponding to line 11 (highlighted in orange).

Clang-49771: For the test program below, without `-g`, the instruction for creating the terminating condition `!g` and the instruction for the if statement are adjacent to each other. Enabling `-g` inserts a debug intrinsic into the LLVM IR between these instructions. The *Loop Strength Reduction* pass incorrectly counted these debug intrinsics when checking whether the terminating condition is adjacent to the terminating branch.

```
int a; char b, c;
short d() {
  { int e;
  f:;
    int g = b != 7;  if (!g) goto h;
    if (!a) goto i;
    return a;
  i:b = b + 1;
    goto f;
  h:; }
  return c;
}
int main() {}
```

4.4 Improvement of Code Coverage

We conducted experiment to understand the effects of the program transformations of Dfusor in improving code coverage of compilers. We randomly sampled 100 seed programs and mutated them using the three transformations. For each set of program variants, we invoked Clang/GCC to compile them with flag `-g3 -O3` and measured the code covered in the compilations. If these variants improve the code coverage over seeds, it confirms that our transformations can explore more code of the compilers and increase the chance to trigger bugs.

Results. Table 4 shows the results on latest Clang and GCC. The two rows "seeds" list the line/branch/function coverage using the seeds on Clang and GCC respectively. Other rows show the coverage of the program variants. For example, "RCL" refers to the coverage of program variants using RCL, and "RCL+FCO+OCP" refers to the programs using all three transformations. The percentage in parentheses is the improvement ratio of each coverage *w.r.t.* the one of the seeds. For example, $2.22\%=(557,070-544,970)/544,970\times100\%$ is the improvement ratio of RCL *w.r.t.* seeds in line coverage, where 544,970 and 557,070 are their line coverage respectively.

Table 4: Code coverage of different combinations of program transformations. The value in parentheses is the percentage of increased line/branch/function coverage *w.r.t.* the one triggered by the seeds.

	Line	Branch	Function
Clang-13.0			
seeds	544,970	149,171	31,703
RCL	557,070 (2.22%)	155,977 (4.56%)	32,377 (2.13%)
RCL+FCO	559,969 (2.75%)	158,066 (5.96%)	32,591 (2.80%)
RCL+FCO+OCP	560,047 (2.77%)	158,121 (6.00%)	32,601 (2.83%)
GCC-11.2			
seeds	265,729	200,724	29,415
RCL	272,797 (2.66%)	207,720 (3.49%)	29,780 (1.24%)
RCL+FCO	281,430 (5.91%)	214,379 (6.80%)	30,774 (4.62%)
RCL+FCO+OCP	281,455 (5.92%)	214,422 (6.82%)	30,774 (4.62%)

Analysis. Clearly, all transformations improve code coverage in all coverage criteria. In Clang, both line and function coverage are increased by over 2%; the branch coverage is improved by up to 6%. The improvement is even higher in GCC. The line and branch coverage are increased by up to 5.92% and 6.82%, respectively; more functions are covered by the variants (up to 4.62%). The results show that these transformations can indeed trigger more code paths in compilers.

The effect of each individual transformation on code coverage varies. First, it is clear that RCL improves code coverage effectively. In both Clang and GCC, the code coverage is improved by over 2%, except that the function coverage of GCC is improved by 1.24%. Second, the effects of FCO are different in two compilers. The RCL+FCO only slightly increased the coverage in Clang. In contrast, the improvement ratio of RCL+FCO (5.91%) in GCC is more than double of RCL's improvement ratio (2.66%) . Lastly, the improvement brought by OCP is not as obvious as the previous two transformations. Nevertheless, it still covers more code in compilers.

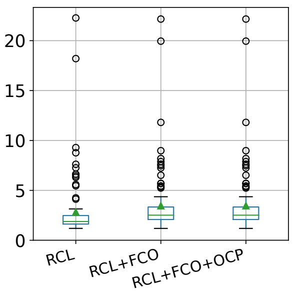
4.5 Improvement of Debug Information

To study the effects of the three program transformations in complicating debug information, we conducted two experiments: *quantity* of debug information and *quality* of debug information. Specifically, by comparing the debug information of the seeds and that of the variants derived from the seeds by various program transformations, we measured whether our program transformations can increase and complicate the debug information quantitatively and qualitatively.

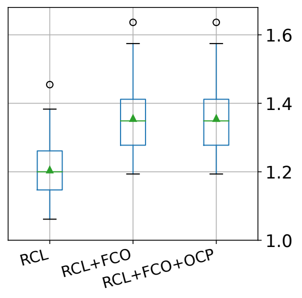
Debugging Information Entry (DIE). In this experiment, we compiled programs using GCC-11.2 with flag `-g3 -O3` and extracted the debug information in DWARF format. The debug information in DWARF is largely comprised of, *debugging information entries*

(DIEs) [3]. Each DIE describes a small unit of the program, such as `subprograms` and `variables`.

4.5.1 Quantity of Debug Information. The quantity of debug information is a direct indicator of how much computation resource is needed to run certain code paths in compiler optimizers to emit debug information. A large amount of debug information usually means that certain relevant code paths in compilers need to run frequently, increasing the chance to find CCMD bugs. We measured the number of DIEs in the debug information (*i.e.*, DIE count) of the compiled variants and compared with that of the compiled seeds. For each variant, we calculated the ratio of its DIE count *w.r.t.* the seed's DIE count. The DIE counts of the seeds range from 559 to 15,165, with mean 2,746 and standard deviation 1,740.



(a)



(b)

Figure 9: (a): The ratio of the DIE count of program variants over that of seeds. (b): The ratio of the number of distinct DIE types for program variants over that for seeds.

Results. Figure 9a shows a boxplot of the results, *i.e.*, the ratios of DIE counts for program variants over the DIE count for seeds. On average, the number of entries using RCL is 2.92x (standard deviation $\sigma = 0.73$) as that of seeds. As for RCL+FCO and RCL+FCO+OCP, the average ratios are identical, *i.e.*, 3.46x ($\sigma = 3.12$). This is because OCP mostly modifies the line information in the `.debug_line` section, which does not affect the number of DIEs. The highest ratios over three boxes are 22.28x, 22.16x and 22.16x, respectively. We also conducted the significance test [47] to examine whether each set of variants has more DIEs than the seeds. The p values of all cases are less than 0.01, indicating that their numbers of DIEs are statistically larger than those of the seeds.

4.5.2 Quality of Debug Information. We believe the quality of debug information is correlated with the diversity of code paths executed when debug information is being emitted. We use the number of distinct DIE types as a proxy metric to measure the quality of debug information. More distinct DIE types means more chances to trigger new code paths in compilers.

To define a distinct DIE type, it is essential to understand the structure of DIEs. Each DIE consists of a `tag` and multiple `attributes`: the `tag` records the class of the DIE, and `attributes` are key-value pairs describing properties of the DIE (name, line number, *etc.*). We define a DIE's type by combining its `tag` and the keys in `attributes`. In other words, the number of distinct DIE types is the number of unique combinations of `tags` and `attribute` keys in each program.

Results. Figure 9b shows the corresponding data regarding distinct DIE types. In the seeds, there are 44 to 65 distinct DIE types (mean 55 and σ=4.57). On average, RCL and RCL+FCO leads to 1.21x and 1.36x distinct entries as the seeds, respectively. RCL+FCO+OCP has the same value as RCL+FCO, since OCP does not modify DIE `attribute` keys. The results demonstrate that the proposed transformations not only increase the size of debug information, but also enrich the diversity of debug information. A significance test also confirms this argument ($p < 0.01$ for all three cases).

5 DISCUSSIONS

5.1 Mitigating CCMD Bugs with `strip`

The command `strip` is a utility program on Unix-like systems that can remove debug information from binaries [10]. Software developers may use `strip` to mitigate CCMD bugs: developers always compile a program with debug information into a binary and then use `strip` to remove the debug information from the binary to obtain another binary that has no debug information. By doing so, both binaries are guaranteed to have exactly the same machine code.

However, this approach is not practical. Compiling a program with debug information *significantly* increases both the binary size and compilation time. To illustrate this, we conduct two experiments. In our first experiment, we measure the time and disk space taken by a full build of LLVM-14. In our second expeiment, we first build LLVM at the commit acf648b, then we checkout the following commit 3089b41 and measure the time taken for an incremental build. The difference between acf648b and 3089b41 is rather small, with the only changes being to one `.cpp` file in the LLVM middle end. Both experiments build LLVM-14 release mode with GCC and various debug flags from `-g0` to `-g3` using a single thread. Without loss of generality, we only build four common components in LLVM: `clang`, `clang-tools-extra`, `compiler-rt` and `polly`.

Table 5: Build times (in hh:mm:ss) and binary size (GB) of LLVM 14, with different degrees of debug information.

	`-g0`	`-g1`	`-g2`	`-g3`
Full Build Time	3:27:02	3:31:19	5:15:39	5:18:50
Incremental Build Time	0:04:54	0:05:21	0:30:54	0:34:47
Binary Size	3.18	14.84	98.73	99.42

Table 5 shows the time taken for each build and the size of the binaries in the `bin` directory of LLVM. Compared to the full build without debug information (`-g0`), the full build with `-g3` takes 56% longer time (~2 hours longer) and the incremental build with `-g3` takes 610% more time (~30 minutes longer) than the incremental build with `-g0`. Builds with `-g3` also produces 30 times larger binaries (~96 GB larger) than `-g0`. From this table, it is easy to see that enabling debug information incurs significant overhead in compilation time and binary size. The overhead is the major reason that industrial developers, *e.g.*, Sony PlayStation [11], prefer disabling debug information; it has also been motivating the developers of GCC and LLVM to make painstaking efforts to implement CCMD natively in GCC and LLVM [21, 30, 36]. Though the `strip`-based approach hides CCMD bugs from users of compilers, it does not help the developers of GCC and LLVM to implement CCMD.

5.2 Finding the Miscompilation Bug [41] via CCMD

We demonstrate that the compiler bug miscompiling the Linux Kernel [41] described in §1 can be detected as a CCMD bug. Though the bug was originally found as a miscompilation, it is also a CCMD bug: enabling debug information causes changes in the machine code. Moreover, we argue that detecting this bug as a CCMD bug is easier than as a miscompilation bug.

This bug stems from GCC mis-handling debug information in the presence of inline assembly. As the vanilla Dfusor does not have transformations that insert inline assembly statements, it would not be able to find this bug. However, Dfusor is general and extensible. To demonstrate that the bug can be detected via CCMD, as a proof of concept, we have developed a simple program transformation that randomly inserts empty inline assembly statements with input and output operands randomly selected from in-scope variables. Within only one day of testing, this transformation found a CCMD bug that reproduces the miscompilation bug,[1] as shown in Figure 10. The pink line highlights the inserted inline assembly statement.

```
1  int a, c = 1;
2  long b;
3  void main() {
4    short d;
5    //undefined behavior:
6    //signed integer overflow by 'c++'.
7    for (; c; c++) {
8      b = 0;
9      for (;;) {
10       asm volatile("" : "+c"(d), "+m"(a));
11       break;
12     }
13   }
14 }
```

Figure 10: A reproduction of the Linux Kernel bug found by an extension of Dfusor. This program is minimized from a program generated by Csmith+Dfusor.

Aside from being able to reproduce the Linux Kernel miscompilation, the new transformation found another 3 new bugs in Clang [43, 44] and GCC [45] within a testing period of 2 weeks (2 CCMD bugs and 1 compiler crash).

Detecting Miscompilations as CCMD Bugs. If a miscompilation is due to mis-handling of debug information, then it can be detected as a CCMD bug. Finding miscompilations is difficult because it requires test programs to be well defined [22, 26, 48], whereas it is relatively easy to generate programs for finding CCMD bugs because the programs only need to be compilable. For example, the program in Figure 10 has an undefined behavior on line 6, which cannot be used to find miscompilations but is sufficient for finding CCMD bugs.

[1]We manually verified that both the CCMD bug and the miscompilation bug have the same root cause. Specifically, we built two versions of GCC: commit `bf95b62` that fixes the miscompilation and the previous commit `6782b1e` that has the miscompilation; `bf95b62` also fixes the CCMD bug whereas `6782b1e` still has the CCMD bug.

5.3 Using Compiler Directives for Finding CCMD Bugs

The bug-triggering programs of CCMD bugs found by Dfusor may inlcude compiler directives. Specifically, among the 23 bugs, 6 include directives in their bug-triggering programs: 3 of them contain common `#include` (Clang-51882) and `#pragma` (Clang-51675\52711), which are inherited from seed programs, while the other 2 bugs have `#line` induced by OCP (GCC-104337\104589). Note that the fixing commits of CCMD bugs are mostly in the middle end and back end of the compilers, rather than the front-end which handles directives (see Table 3).

We attempted to remove the compiler directives from the bug-triggering programs to see whether the bugs could be replicated without them. We found that neither of the two bugs using `#pragma` can be reproduced without `#pragma`, since `#pragma` provides additional information to the compiler [17] and cannot be replaced. The bug using `#include` can be reproduced by replacing the directive with the content of header files.

6 RELATED WORK

This work is the first effort to validate CCMD for production compilers, and Dfusor is the first technique to automatically find violations of CCMD in C compilers. The following discusses two closely related lines of research.

Testing of Debug Information. Several studies focus on testing the correctness of the debug information generated by compilers [8, 25]. Li *et al.* proposed a methodology to validate the correctness of debug information generated for optimized code [25]. Their insight is that the binary code optimized by compilers should stop at the same unoptimized breakpoint and print the same value as the unoptimized one during debugging. Otherwise, compiler optimizations induce errors to the debug information. Di Luna *et al.* designed a framework to find the errors in debug information [8]. They proposed four invariants that should not be violated by any pair of optimized and unoptimized programs. In the execution of randomly generated programs, they dynamically captures such violations to detect errors in debug information.

This paper tackles a different problem as detailed in §2.3: Dfusor validates CCMD but not the correctness of the debug information emitted by compilers. Besides, Dfusor can help tackle the research problem in [8, 25] as well. The state of the art [8, 25] used the programs generated by Csmith in their evaluations. As demonstrated in §4, our program transformations can effectively trigger more code paths in compilers and emit a larger amount of distinct DIEs than Csmith. By using the seed program generated by Dfusor other than Csmith, we believe that the chances to find debug-information-related bugs is generally increases, which we leave as future work.

Testing of Compiler Optimizers. Most prior studies on compiler validation focus on the correctness of compiler optimizers [22, 23, 26, 37, 48]. Csmith [48] and YARPGen [26] randomly generate programs for C/C++ compilers. After compiling these programs with multiple compilers, a bug is found if the binaries behave differently in execution. Le *et al.* proposed Equivalence Modulo Input (EMI) to expose bugs in compiler optimizations [22, 23, 37]. It mutates the seed program in diverse ways while ensuring that the mutants are equivalent to the seed program *w.r.t.* a given input. tkfuzz [38] randomly substitutes variable or function name with a different one to find crash bugs in compilers. Recently, Theodoridis *et al.* proposed a novel methodology to detect the missed optimizations in compiler optimizers [40]. Their methodology inserts markers to the dead code blocks of programs and then analyzes the binary programs optimized by two compilers. If there is any marker that only exists in one of the optimized binary programs, it is implied that one of the compilers misses the optimization opportunity to eliminate the corresponding dead code block.

As detailed in §2.3, our work is different from the prior work. First, our work focuses on CCMD, a *new* research problem that requires special care to generate test programs with both complex debug information and logic, in order to fully exercise the integration of handling debug information and optimizing algorithms inside compilers. Prior work mainly focuses on whether compilers can reliably compile test programs into semantically-equivalent binaries. Second, the program transformations inside Dfusor are different from those used in prior work: the goal of our program transformations is to complicate debug information of test programs and trigger different code paths in compiler optimizers at the same time.

7 CONCLUSION

This paper introduces CCMD, a vital property that compilers should support, *i.e.*, compilers should generate the same machine code, whether debug information is enabled. We also propose Dfusor, the first automatic test generation technique to validate CCMD for C compilers. Within 10 months of development and intermittent testing, Dfusor has found 23 bugs in GCC and Clang, and 21 of them have been confirmed or fixed by the compiler developers. The proposed transformations used in Dfusor can significantly complicate debug information by introducing 214% more DIEs and 36% distinct DIEs, and improve the code coverage of GCC and LLVM by up to 6.82%. We believe that our efforts open up a new research direction to improve the quality and usability of production compilers. As future work, we will explore the possibility of using Dfusor to validate the correctness of debug information [8, 25].

ACKNOWLEDGMENTS

We thank all the anonymous reviewers in ASPLOS'23 for their insightful feedback and comments, which significantly improved this paper. We thank Davide Italiano for introducing this problem to us. We express our appreciation to the LLVM developer Paul Robinson and GCC developers Andrew Pinski, Jakub Jelinek and Richard Biener for their enlightening discussion with us. Our appreciation also goes to all LLVM and GCC developers who confirmed and fixed the bugs reported by us. This research is partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) through the Discovery Grant, a project under Waterloo-Huawei Joint Innovation Lab, and CFI-JELF Project #40736.

REFERENCES

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2nd Edition).* Addison Wesley, Chapter 6. Intermediate-Code Generation.
[2] Richard Biener. 2014. GCC Bug 61801. Retrieved July 10, 2022 from https://gcc.gnu.org/bugzilla/show_bug.cgi?id=61801
[3] DWARF Debugging Information Format Committee. 2017. DWARF Debugging Information Format Version 5. Retrieved February 28, 2022 from https://dwarfstd.org/doc/DWARF5.pdf
[4] LLVM Developers. 2018. [meta] Make llvm passes debug info invariant. Retrieved July 15, 2022 from https://github.com/llvm/llvm-project/issues/37076
[5] The GDB developers. 2022. GDB: The GNU Project Debugger. Retrieved February 28, 2022 from https://www.sourceware.org/gdb/
[6] The GDB developers. 2022. Options for Debugging Your Program. Retrieved February 28, 2022 from https://gcc.gnu.org/onlinedocs/gcc/Debugging-Options.html
[7] The LLVM developers. 2022. Debug information and optimizations. Retrieved February 28, 2022 from https://llvm.org/docs/SourceLevelDebugging.html#debug-information-and-optimizations
[8] Giuseppe Antonio Di Luna, Davide Italiano, Luca Massarelli, Sebastian Österlund, Cristiano Giuffrida, and Leonardo Querzoni. 2021. *Who's Debugging the Debuggers? Exposing Debug Information Bugs in Optimized Binaries.* Association for Computing Machinery, New York, NY, USA, 1034–1045. https://doi.org/10.1145/3445814.3446695
[9] Free Software Foundation. 2021. Bison 3.8.1. Retrieved July 20, 2022 from https://www.gnu.org/software/bison/manual/bison.html
[10] Free Software Foundation. 2021. strip(1). Retrieved July 5, 2022 from https://man7.org/linux/man-pages/man1/strip.1.html
[11] Russell Gallop. 2015. Verifying Code Generation is unaffected by -g/-S. Retrieved July 10, 2022 from https://llvm.org/devmtg/2015-04/slides/Verifying_code_gen_dash_g_final.pdf
[12] GCC. 2022. Developer Options. Retrieved June 14, 2022 from https://gcc.gnu.org/onlinedocs/gcc/Developer-Options.html
[13] GCC. 2022. Function Attributes. Retrieved February 28, 2022 from https://gcc.gnu.org/onlinedocs/gcc/Function-Attributes.html
[14] GCC. 2022. Label Attributes. Retrieved February 28, 2022 from https://gcc.gnu.org/onlinedocs/gcc/Label-Attributes.html
[15] GCC. 2022. Line Control. Retrieved February 28, 2022 from https://gcc.gnu.org/onlinedocs/cpp/Line-Control.html
[16] GCC. 2022. Optimize Options. Retrieved February 28, 2022 from https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html
[17] GCC. 2022. Pragmas. Retrieved Jul 15, 2022 from https://gcc.gnu.org/onlinedocs/cpp/Pragmas.html
[18] GCC. 2022. Preprocessor Output. Retrieved Jul 15, 2022 from https://gcc.gnu.org/onlinedocs/cpp/Preprocessor-Output.html
[19] GCC. 2022. Type Attributes. Retrieved February 28, 2022 from https://gcc.gnu.org/onlinedocs/gcc/Type-Attributes.html
[20] GCC. 2022. Variable Attributes. Retrieved February 28, 2022 from https://gcc.gnu.org/onlinedocs/gcc/Variable-Attributes.html
[21] Jakub Jelinek. 2022. Bug 104237 - [11 Regression] Emitted binary code changes when -g is enabled at -O1 -flto and optimize attribute since r11-3126-ga8f9b4c54cc35062. Retrieved July 15, 2022 from https://gcc.gnu.org/bugzilla/show_bug.cgi?id=104237#c20
[22] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 216–226. https://doi.org/10.1145/2594291.2594334
[23] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Finding deep compiler bugs via guided stochastic program mutation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, Jonathan Aldrich and Patrick Eugster (Eds.). ACM, 386–399. https://doi.org/10.1145/2814270.2814319
[24] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Randomized Stress-Testing of Link-Time Optimizers. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis* (Baltimore, MD, USA) *(ISSTA 2015)*. Association for Computing Machinery, New York, NY, USA, 327–337. https://doi.org/10.1145/2771783.2771785
[25] Yuanbo Li, Shuo Ding, Qirun Zhang, and Davide Italiano. 2020. Debug Information Validation for Optimized Code. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 1052–1065. https://doi.org/10.1145/3385412.3386020

[26] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2020. Random Testing for C and C++ Compilers with YARPGen. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 196 (nov 2020), 25 pages. https://doi.org/10.1145/3428264

[27] Julia Menapace, Jim Kingdon, and David MacKenzie. 1992. *The" stabs" debug format.* Technical Report. Technical report, Cygnus support.

[28] Microsoft. 2022. The PDB (Program Database) Symbol File format. Retrieved February 28, 2022 from https://github.com/Microsoft/microsoft-pdb

[29] Hans-Peter Nilsson. 2010. GCC Bug 45656. Retrieved July 10, 2022 from https://gcc.gnu.org/bugzilla/show_bug.cgi?id=45656

[30] Andrew Pinski. 2022. Bug 104237 - [11 Regression] Emitted binary code changes when -g is enabled at -O1 -flto and optimize attribute since r11-3126-ga8f9b4c54cc35062. Retrieved July 15, 2022 from https://gcc.gnu.org/bugzilla/show_bug.cgi?id=104237#c19

[31] The LLVM Compiler Infrastructure Project. 2022. check_cfc. Retrieved June 14, 2022 from https://github.com/llvm/llvm-project/tree/main/clang/utils/check_cfc

[32] The LLVM Compiler Infrastructure Project. 2022. Controlling Debug Information. Retrieved February 28, 2022 from https://clang.llvm.org/docs/ClangCommandLineReference.html#kind-and-level-of-debug-information

[33] The LLVM Compiler Infrastructure Project. 2022. The LLDB Debugger. Retrieved February 28, 2022 from https://lldb.llvm.org/

[34] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-Case Reduction for C Compiler Bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (Beijing, China) *(PLDI '12)*. Association for Computing Machinery, New York, NY, USA, 335–346. https://doi.org/10.1145/2254064.2254104

[35] Paul Robinson. 2018. Bug 37306 - [fuzzDI] -O1 + '-g' cause the generated code to change. Retrieved February 28, 2022 from https://bugs.llvm.org/show_bug.cgi?id=37306#c7

[36] Paul Robinson. 2021. Bug 50913 - Emitted binary code changes at -O0 when compiling through -S. Retrieved July 15, 2022 from https://github.com/llvm/llvm-project/issues/50913

[37] Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding Compiler Bugs via Live Code Mutation. *SIGPLAN Not.* 51, 10 (oct 2016), 849–863. https://doi.org/10.1145/3022671.2984038

[38] Chengnian Sun, Vu Le, Qirun Zhang, and Zhendong Su. 2016. Toward Understanding Compiler Bugs in GCC and LLVM. In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (Saarbrücken, Germany) *(ISSTA 2016)*. Association for Computing Machinery, New York, NY, USA, 294–305. https://doi.org/10.1145/2931037.2931074

[39] Chengnian Sun, Yuanbo Li, Qirun Zhang, Tianxiao Gu, and Zhendong Su. 2018. Perses: Syntax-Guided Program Reduction. In *Proceedings of the 40th International Conference on Software Engineering*. Association for Computing Machinery, 361–371. https://doi.org/10.1145/3180155.3180236

[40] Theodoros Theodoridis, Manuel Rigger, and Zhendong Su. 2022. Finding Missed Optimizations through the Lens of Dead Code Elimination. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 697–709. https://doi.org/10.1145/3503222.3507764

[41] Linus Torvalds. 2014. Fix gcc-4.9.0 miscompilation of load_balance() in scheduler. Retrieved June 12, 2022 from https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=2062afb4f804afef61cbe62a30cac9a46e58e067

[42] Linus Torvalds. 2014. GCC Bug 61904. Retrieved July 10, 2022 from https://gcc.gnu.org/bugzilla/show_bug.cgi?id=61904

[43] Theodore Luo Wang. 2022. Emitted binary changes when -g is enabled with -O1. Retrieved July 13, 2022 from https://github.com/llvm/lvm-project/issues/56523

[44] Theodore Luo Wang. 2022. Emitted binary changes when -g is enabled with -O3. Retrieved July 16, 2022 from https://github.com/llvm/llvm-project/issues/56575

[45] Theodore Luo Wang. 2022. ICE when compiling inline asm with -m32. Retrieved July 20, 2022 from https://gcc.gnu.org/bugzilla/show_bug.cgi?id=106364

[46] Wikipedia. 2022. Core dump. Retrieved February 28, 2022 from https://en.wikipedia.org/wiki/Core_dump

[47] Frank Wilcoxon. 1945. Individual Comparisons by Ranking Methods. *Biometrics Bulletin* 1, 6 (1945), 80–83.

[48] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 283–294. https://doi.org/10.1145/1993498.1993532

A ARTIFACT APPENDIX

A.1 Abstract

This artifact provides the binary program of Dfusor, *i.e.*, the three program transformers proposed by us to find CCMD bugs. The artifact is packaged as a Docker image for a standard X86 machine.

A.2 Artifact check-list (meta-information)

- **Algorithm: OCP, FCO and RCL**
- **Binary: The binary of Dfusor is provided as a Docker image.**
- **Run-time environment: Docker**
- **Hardware: X86**
- **Output: Program variants transformed by transformers.**
- **Disk space required (approximately)?: 15 GB**
- **Time needed to prepare workflow (approximately)?: 10 mins**
- **Publicly available?: Yes**
- **Archived DOI: 10.5281/zenodo.7217505**

A.3 Description

A.3.1 How to access. The artifact can be downloaded from https://doi.org/10.5281/zenodo.7217505

A.3.2 Hardware dependencies. A standard X86 machine.

A.3.3 Software dependencies. Docker.

A.4 Installation

Please use the following command to install the artifact.

```
docker load --input ASPLOS23-dfusor.tar
```

A.5 Experiment workflow

(1) Generate a program as seed program.
(2) Generate a program variant by applying the transformers to the seed program.

A.6 Evaluation and expected results

The "README.MD" file provides the detailed steps to generate a random program using Csmith and then transform it using Dfusor. It is expected that the seed program is properly transformed by each transformation. Since the three transformations have certain randomness, the program variants are likely to be different from the ones shown in the documentation.

All the bugs reported by this study are reproducible in certain versions of GCC or Clang. The detailed steps, including the program variants, can be found in their bug reports. Due to the randomness in program transformation, it is not expected to find the exact same program variants that trigger the bugs reported in this study.

Received 2022-07-07; accepted 2022-09-22