Finding and Analyzing Compiler Warning Defects

Chengnian Sun Vu Le Zhendong Su

Department of Computer Science, University of California, Davis, USA

{cnsun, vmle, su}@ucdavis.edu

ABSTRACT

Good compiler diagnostic warnings facilitate software development as they indicate likely programming mistakes or code smells. However, due to compiler bugs, the warnings may be erroneous, superfluous or missing, even for mature production compilers like GCC and Clang. In this paper, we (1) propose the *first randomized differential testing technique to detect compiler warning defects* and (2) describe our *extensive evaluation in finding warning defects in widely-used C compilers*.

At the high level, our technique starts with generating random programs to trigger compilers to emit a variety of compiler warnings, aligns the warnings from different compilers, and identifies inconsistencies as potential bugs. We develop effective techniques to overcome *three specific challenges*: (1) How to generate random programs, (2) how to align textual warnings, and (3) how to reduce test programs for bug reporting?

Our technique is very effective — we have found and reported 60 bugs for GCC (38 confirmed, assigned or fixed) and 39 for Clang (14 confirmed or fixed). This case study not only demonstrates our technique's effectiveness, but also highlights the need to continue improving compilers' warning support, an essential, but rather neglected aspect of compilers.

CCS Concepts

•Software and its engineering → Compilers; Software testing and debugging; •Human-centered computing → *Usability testing;*

1. INTRODUCTION

Compiler warnings are diagnostic messages emitted during compilation on questionable constructs in language conforming code. A warning message describes the reason of the warning and contains the location information of the problematic code fragment (*e.g.*, column number, line number and affected file). Developers use warning messages to detect bugs at compile time by matching their code against certain patterns, which are either behaviors undefined in programming language standards or have been found to be likely programming mistakes. For example, the Security Engineering group at Microsoft utilizes compiler warnings to discover potential security exploits in the process of security code reviews [18]; the maintenance engineers at Hewlett-Packard improve the quality of code base in routine maintenance by correcting code on which compilers warn [31].

```
1  /* file=s.c */
2  int f(int a) {
3    int i = 0;
4    if (a) {
5      i++;
6    } else; /* a stray semicolon here */
7      i *= 2;
8    return i;
9  }
```

```
GCC 5.0 Output:
s.c:6:9: warning: suggest braces around empty body in an ‘else’
                                                 statement [-Wempty-body]
} else; /* a stray semicolon here */
       ^
```

Figure 1: Bug #18877 of Clang. The function has an empty `else` branch. The statement `i *= 2` on line 7 is not controlled by the `else` branch due to the semicolon on line 6. GCC emits a warning on this issue whereas Clang misses it.

Although compiler diagnostics is widely used and important, it can still have bugs, similar to the other compiler components (*e.g.*, optimizers and code generators). These bugs can negatively affect a compiler's usability and developer productivity. Figure 1 shows an example of such a warning bug in Clang 3.6. In the code snippet of Figure 1,[1] there is an inadvertent semicolon placed immediately after the `else` branch on line 6, which makes the statement `i *= 2` on line 7 unconditionally executed. GCC emits a warning on this case, whereas Clang considers this code snippet free of problems. This warning bug of Clang delays the discovery of the coding error to the testing stage, while it could have alerted the developer to the error earlier at compile time. More details on compiler warning defects will be discussed in Section 2.3.

Although there has been extensive work on software testing and analysis, little attention has been devoted to testing compilers' diagnostic support. This paper introduces the first effort in this direction: (1) a *practical differential testing approach* for validating warning support in compilers, and (2) an *extensive evaluation* in testing two production C compilers, GCC and Clang. Our technique and its

ICSE '16, May 14-22, 2016, Austin, TX, USA

© 2016 ACM. ISBN 978-1-4503-3900-1/16/05…$15.00

DOI: http://dx.doi.org/10.1145/2884781.2884879

[1]This accepted test program was reported at https://llvm.org/bugs/show_bug.cgi?id=18877#c5, which is different from the initial test program reported in the description of this bug report.

accompanying tool, Epiphron[2], leverage programs generated by a random program generator to detect inconsistent warning messages from different compilers under test. The central assumption of our approach is that production compilers are mature and reliable, and should emit same/similar warnings on the same given program. Any inconsistent warning behavior indicates a likely compiler bug.

To effectively identify/report compiler warning defects, Epiphron overcomes three key technical challenges: (1) How to generate adequate test programs to stress test compiler warning diagnostics, (2) how to align textual warnings from different compilers to identify inconsistencies, and (3) how to reduce test cases that trigger warning inconsistencies before reporting them to compiler developers?

Challenge 1: Generating Effective Test Programs. Testing compiler warning diagnostics mainly targets compiler front-ends, whereas traditional compiler testing [21–23, 37] focuses on the correctness of compiler optimizers and code generators. This difference induces different requirements on the generated test programs. For our purpose, test programs should cover various language constructs to fully exercise the warning diagnostics, yet it is unnecessary to execute them. Therefore, we do not need to ensure that the test programs are free of undefined behaviors, a critical requirement for traditional compiler testing [21–23, 37]. Furthermore, to better design a program generator, we have empirically studied the characteristics of all the historical warning bugs in GCC and Clang *fixed* before 2014. We observe that most of these bugs are unrelated to the bodies of conditional statements, and not within obviously dead code regions (*e.g.* unreachable conditional branches). We leverage this finding in our program generator, which significantly reduces false positives in differential testing.

Challenge 2: Aligning Warnings. We cannot directly compare the warning messages from different compilers to identify inconsistent warning behaviors, because the messages are in natural language and different compilers may present them quite differently. To tackle this challenge, for each compiler, we design specific parsers to extract computer-recognizable warning records from its natural language warning descriptions. We also design a warning taxonomy to assign each warning record a type. Based on the types of and the information in the extracted records, we align the warning records from the two compilers. Any aligned pair with inconsistent records indicates a potential warning defect.

Challenge 3: Reducing Test Cases. Once we find a warning bug, before reporting it, we need to reduce the bug-triggering test program by removing parts of the program irrelevant to the bug. The reduced program helps developers triage/fix the bug. However, reducing warning bugs is much more complex than reducing regular compiler bugs. In particular, reducing miscompilation or crashing bugs only requires testing the behavior of the compiled executables or the exit code (an integer) of compilers [28, 33, 38]. However, reducing warning bugs involves processing the textual warning output of compilers, and needs expressive predicates to specify the inconsistency of interest which we would like to preserve after each iteration of reduction. We base our reduction process on the alignment algorithm, and further design a set of generic predicates over the aligned warning pairs.

We have applied Epiphron to GCC and Clang, two mature and widely used production compilers. Our evaluation shows that Epiphron is very effective in finding warning bugs, even though both compilers' code bases for C programming language standards `C89` and `C99` have already been stable. We have found and reported 60 bugs to GCC (38 accepted), and 39 bugs to Clang (14 accepted).

Contributions. Below summarizes our main contributions:

[2] Epiphron was the Greek god of prudence and shrewdness.

- We introduce an effective random testing technique to validate the warning support of compilers, and have realized it as a practical tool Epiphron for testing C compilers. Epiphron includes a program generator specifically designed for testing warning diagnostics, an alignment tool analyzing textual warnings to identify warning inconsistencies between compilers, and a test program reducer to facilitate bug reporting.
- Epiphron has helped discover and report 60 bugs to GCC and 39 bugs to Clang, both of which are widely-used and well-tested production compilers. Specifically, for GCC, 38 bugs have already been accepted/fixed and 12 bugs are pending developers' response; for Clang, 14 bugs have been accepted/fixed and 25 bugs are pending developers' response.
- Our evaluation itself (*i.e.* reported bugs) serves as a convincing empirical evidence, calling for more attention on testing compiler warning diagnostics. It opens up a new research direction to improve the usability of compilers to benefit both novice and experienced developers.

Paper Organization. Section 2 presents the definition of warnings and how they are identified. Section 3 introduces our approach for finding compiler warning defects, while Section 4 presents the detailed results on our efforts in finding GCC and Clang warning defects. Section 5 discusses Epiphron's false positive rate and applicability of static analysis checkers to detect warning bugs. Finally, we survey related work (Section 6) and conclude (Section 7).

2. COMPILER WARNINGS

A good compiler not only compiles source code correctly, but also emits useful warnings to alert developers to potentially problematic code fragments. A warning should contain the location information of the problematic code fragment (*i.e.*, file name, line number and column number), and a message describing the potential problem. Some modern compilers may produce extra information, such as the warning type and suggestions to eliminate the warning.

Let us take the code in Figure 1 as an example. GCC 5.0 warns that the body of the `else` statement is empty. The prefix "`s.c:6:9:warning:`" indicates that the current message is a warning, and the problematic code is on line 6, column 9 of file "`s.c`". The potential problem is an `else` statement with an empty body. The postfix "`[-Wempty-body]`" is the name of the warning checker that emits this warning, which can also serve as the warning type. GCC also prints the problematic code fragment to help developers identify the problem easily. It also provides a suggestion to silence this warning. If the code is intended, developers may suppress the warning by following the suggestion.

2.1 Compiler Warning Mechanism

Matching code against certain patterns at the compilation stage underlies the mechanism of compiler warning generation. These patterns can be classified into two general classes:

- **Bad Practice** This type consists of patterns that have been found to be likely programming mistakes in practice. The example in Figure 1 belongs to this category because although the code conforms to the C standard, it is usually a bug or at least code smell in practice.
- **Undefined Behavior** This type consists of behaviors that are undefined according to the programming language standard. Examples include using an uninitialized variable, accessing an array index that is out-of-bound, and dereferencing a NULL pointer.

```c
// file = s.c
char a[] = {0xFFFF, 4, 0xEFF};
```

```
GCC 4.9 Output:
s.c:2:1:warning:overflow in implicit constant conversion
s.c:2:1:warning:overflow in implicit constant conversion
```

(a) Erroneous message (GCC Bug #60455)

```c
// file = s.c
int f(unsigned char a, unsigned char b) {
  const unsigned l = 4294967295u;
  return (l ^ a) != b;
}
```

```
GCC 4.9 Output:
s.c:4:18:warning:comparison of promoted ~unsigned with unsigned
```

(b) Spurious warning (GCC Bug #60090)

```c
// file = s.c
int f(unsigned a, int *b) {
  return a > (~(95 != *b));
}
```

```
GCC 4.9 Output:
s.c:3:12:warning:comparison between signed and unsigned
                                            integer expressions
```

(c) Missing warning (Clang Bug #18504)

Figure 2: Three Compiler Warning Bugs

Compilers need different levels of program information to correctly generate warnings. While some types of warnings only require syntactic information (*e.g.*, the one in Figure 1), many others depend on semantic information only available via static program analysis.

2.2 Importance of Compiler Warnings

Compiler warnings are important to both novice and experienced developers. Allain suggests that compiler warnings should be treated carefully, because they provide a means to catch bugs early, including those that are difficult to find during testing [8]. Indeed, large software companies have been using compiler warnings to improve code quality for years.

Software Maintenance. Software engineers at Hewlett-Packard use compiler warnings to clean up source code in their routine maintenance [31]. During a maintenance activity, they increase the level of compiler diagnostics to obtain a large number of compiler warnings. A team of engineers is then assembled to resolve each warning. Doing so not only helps refactor buggy, dangerous or wasteful code, but also makes the system ready for new compilers.

Security Code Review. The Security Engineering group at Microsoft utilizes compiler warnings to discover potential vulnerabilities during security code reviews [18]. They enable compiler diagnostics at the highest level to identify areas of code that require extra scrutiny. Their experience has shown that some warnings are actually bugs or at least hide real bugs, which may be exploitable vulnerabilities.

2.3 Categories of Compiler Warning Bugs

Compiler warning bugs can negatively impact developers' productivity, and we categorize them into three general classes:

Erroneous Messages. Warning messages can be wrong. The compiler may use a misleading or confusing sentence to describe the underlying problematic code fragment, or produce an incorrect location. Figure 2a shows a GCC bug where the compiler emits two overflow warnings with incorrect locations. Incorrect or bogus warning messages frustrate developers, wasting their effort in realizing that the warnings are incorrect. Modern integrated development environments (IDEs) are also impacted because they rely on compiler output to render errors/warnings. For example, Eclipse C/C++ Development Tooling[3] parses the compilation output of GCC or Clang to highlight problematic code. This type of warning bugs will make Eclipse behave bizarrely.

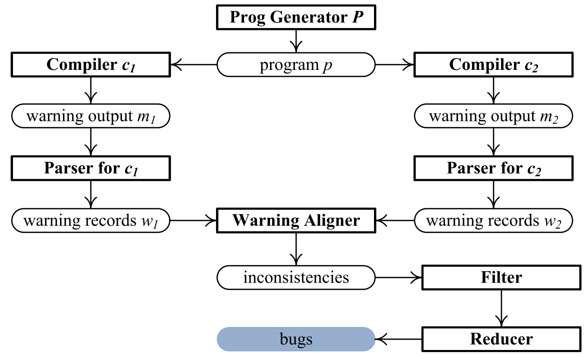

Figure 3: Overall Framework of Our Approach

Spurious Warnings. Compilers emit superfluous warnings for benign code fragments. In Figure 2b, GCC emits a sign comparison warning (*i.e.*, "comparison of promoted ∼unsigned with unsigned") in the returned expression. However, there is no *bitwise not* operation (∼) in the expression. Spurious warnings waste development time and resource. For instance, during a routine maintenance activity at Hewlett-Packard, a subsystem generated 499 new warnings, which took a team of engineers to resolve. If a considerable number of warnings are superfluous, a large amount of developers' time will be wasted. Moreover, bogus warnings may even cause the software build process to fail if the compiler is configured to treat warnings as errors (*i.e.*, the flag `-Werror` of GCC and Clang).

Missing Warnings. Compilers may overlook a potentially buggy code fragment and thus miss a warning. Figure 2c shows a Clang bug where the compiler fails to report a problematic comparison between a signed integer and an unsigned integer. Note that a missing warning is not necessarily a feature request. This example is a real bug as confirmed and explained by the developer.[4] Missing warnings can prevent developers from finding bugs at early development stages. For example, the design decision of GCC not to warn on declared but unused static constants [3] hides a bug in GDB [4]. In contrast, Clang has added a new warning flag `-Wunused-const-variable` to catch such unnoticed bugs.

All three types of warning defects above are exacerbated when novice developers are involved, because they are usually unfamiliar with the programming language (as stated by Peter Norvig [30]).

3. APPROACH

Our approach is based on the concepts of random and differential testing. It takes as input a pair of compilers $C = \{c_1, c_2\}$ and a random program generator $\mathcal{P}$, and outputs inconsistent warning behaviors between c_1 and c_2. Figure 3 shows the overall framework of the proposed technique, which contains two major steps:

[3]https://eclipse.org/cdt/

[4]http://llvm.org/bugs/show_bug.cgi?id=18504#c1

- **Random Testing** We first use $\mathcal{P}$ to generate a random program p. The two compilers c_1 and c_2 compile p and emit two raw warning output m_1 and m_2, which are parsed into two sets of warnings w_1 and w_2.
- **Differential Testing** We compute the symmetric difference between the two warning sets w_1 and w_2 (*i.e.*, $w_1 \setminus w_2 \cup w_2 \setminus w_1$) as potential warning defects for further investigation.

This process should be repeated indefinitely until reaching a global fixpoint (*i.e.*, all inconsistencies are known) or having exhausted the resource budget.

Because the warning messages m_1 and m_2 are in natural language and have different natural language descriptions across different compilers, it is difficult to directly compute set difference on m_1 and m_2. Therefore, we first invoke a compiler-specific parser to process the warning messages into a set of records (m_1 to w_1, and m_2 to w_2). Each record stores a warning's location, type and other relevant information.

Next, the component "`Warning Aligner`" aligns w_1 and w_2 into a list of pairs based on the parsed records, and computes the symmetric difference between w_1 and w_2 as potential warning bugs in either c_1 or c_2. The component "`Filter`" removes known inconsistencies (*i.e.*, false positives and reported bugs). Finally, we reduce the test program that triggers each remaining inconsistency to obtain a minimized test program that still triggers the same inconsistency, and report it if it is indeed a warning bug.

3.1 Generating Test Programs

Testing compiler warning diagnostics mainly targets compiler frontends, whereas traditional compiler testing [21, 23, 37] usually focuses on the correctness of compiler optimizers and code generators. This difference induces different requirements on the generated test programs. For our purpose, test programs should cover various language constructs to fully exercise the warning diagnostics, yet it is not necessary to execute them. Therefore, we do not need to ensure the test programs free of undefined behaviors, a property otherwise critical to traditional compiler testing [21, 23, 37].

Observations from Historical Warning Bugs. To design an effective program generator, we empirically studied all historical warning bugs that were *fixed* before January 2014. In total we investigated 150 bugs of GCC and 80 of Clang. After analyzing the associated test cases, we have the following two findings on the problematic statement s on which compilers warn:

1. s is *not* within an obviously dead code region. In other words, there is no warning bug on an unreachable statement.
2. s is usually *not* control-dependent on a conditional statement (*e.g.*, `if`, `for` and `while`). That is, compilers only analyze statements locally to emit warnings. It does not matter whether the statements are within the body of a conditional statement or not.

Epiphron Program Generator. We design the Epiphron program generator that supports nearly all the language constructs of the C language. It produces random compilable test programs by unrolling the grammar of the C language. At each step it picks a random, viable grammar production to generate a construct (*e.g.*, a statement or an expression). Epiphron generates much more diverse programs than Csmith [37] and Orion [21].

We further improve Epiphron by leveraging the two findings above to reduce false positives of differential testing. In particular, when Epiphron generates a conditional statement, it intentionally constructs warning-free body such as an empty statement "`;`" for `if` statements, and "`break`" for loop statements.

```
1  char* g() {
2    char *p = "hello";
3    p[0] = 'd'; /*Segmentation fault here.*/
4    return p;
5  }
```

```
Clang 3.4 Output:
s.c:2:9:warning: initializing 'char *' with an expression
          of type 'const char [6]' discards qualifiers
      [-Wincompatible-pointer-types-discards-qualifiers]
```

Figure 4: Bug #18801 of Clang discovered by CVS. The function tries to modify a string literal via a pointer referencing the literal. According to the C standard [19], the string literal `"hello"` always has static storage duration and is immutable on most architectures. The statement on line 3 modifies it, which is undefined behavior and causes an illegal memory access on x86 Linux. In Clang 3.5, the command option `-Weverything` does not enable `-Wwrite-string`, thus missing the warning.

The above design is quite effective at differentially testing GCC and Clang because it helps avoid certain false positives *by construction*. Indeed, it is a compiler vendor's design decision whether to warn on problematic code in obviously dead code regions, thus warning inconsistencies on dead code are often not confirmed as real bugs. For example, GCC might emit various warnings on dead code, while Clang only produces one warning that the code region is unreachable.

3.2 Selecting Reference Compilers

The assumption of our approach is that provided that the two compilers c_1 and c_2 are mature and defect-free, ideally they should emit the same set of warnings (*i.e.*, $w_1 = w_2$) for the same program p. This assumption is vital for effective differential testing, which states that any discovered inconsistent warning behavior between c_1 and c_2 is likely a bug in either c_1 or c_2 (or both).

The selection of c_1 and c_2 for differential testing is very important in our approach, because a bad selection can cause many false positives that require manual investigation. In this paper, we adopt the following three strategies for choosing the right compilers for effective differential testing.

3.2.1 Differential Testing Strategies

Cross-Compiler Strategy (CCS). This strategy selects two different compilers that have been developed independently. Given a programming language, we can select two of its mature and competing compilers for warning inconsistency checking. GCC and Clang are a good example here. Both compilers are mature and under active development for years. In particular, Clang is designed to be a drop-in replacement for GCC and supports all of the GCC command arguments and their semantics. The motivating example shown in Figure 1 is uncovered using this strategy.

Cross-Version Strategy (CVS). This strategy selects different versions of a compiler for differential testing. It specifically targets regressions in compiler warning support, which correspond to bugs introduced in the newer version. For example, we can use Clang 3.4 as a reference compiler to test Clang 3.5.[5]

Figure 4 shows a bug in Clang 3.5 that is discovered by this strategy. Clang has a command option `-Weverything` which enables all diagnostics [1]. However, in Clang 3.5, this invariant is broken,

[5] Clang 3.7 is the current development version of Clang; the latest stable release is Clang 3.6.

```
1  int *const a = 0;
2  unsigned fn1() {
3    unsigned short s = ~0x4578ADBCAA1DE677LL ^ (a == 0);
4    return s;
5  }
```

(a) A function with an integer overflow at line 3

```
s.c:3:23:warning: negative integer implicitly converted
                                 unsigned type [-Wsign-conversion]
s.c:3:23:warning: negative integer implicitly converted
                              to unsigned type [-Wsign-conversion]
```

(b) Duplicate warnings by GCC `-O0` (non-optimized)

```
s.c:3:23: warning: large integer implicitly truncated
                           to unsigned type [-Woverflow]
```

(c) A warning by GCC `-O1` (optimized)

Figure 5: Bug #60083 of GCC discovered by COS. GCC `-O0` emits two duplicate warnings, whereas GCC `-O1` correctly emits only one warning

as `-Wwrite-string` is excluded from `-Weverything`, which warns if an immutable string literal is assigned to a mutable pointer, *e.g.* `char *p = "hello";`. The function `g` tries to modify a string literal via a non-const pointer p, which triggers a segmentation fault on line 3. In order to fix the bug, the developer should use an array to copy the string literal, instead of using the pointer p to reference it, *i.e.* `char p[] = "hello";`. Clang 3.4 is able to detect this problem with `-Weverything`, and outputs the warning as shown below the function `g`. In contrast, Clang 3.5 deems that the code is benign.

Cross-Optimization Strategy (COS). This strategy selects the same compiler, but compiles the generated random program p under different optimization levels. For example, given GCC to test, we can instruct GCC to compile p *without* optimizations (with flag `-O0`) as c_1 and compile p *with* optimizations (with flags `-O1`, `-Os`, `-O2` or `-O3`) as c_2.

This strategy is inspired by discussions within the GCC and Clang communities, which state that warnings/errors should be independent of optimization levels. To quote some compiler developers:

> "We generally don't like for program validity (or warnings) to depend on the chosen optimization level" —by a Clang developer[6]

> "I believe we strive for the warnings be independent of the optimization level" —by a GCC developer[7]

This strategy aims to find inconsistent warnings among different optimization levels. Figure 5 shows a bug in GCC discovered by COS. For the integer overflow on line 4 in Figure 5a, GCC `-O0` (without optimization) emits two duplicate warnings, whereas GCC `-O1` (with optimization enabled) correctly emits a single warning.

3.2.2 *Relationship among the Strategies*

CCS targets a general scope of warning defects, CVS targets regressions of a single compiler, and COS targets inconsistent warnings of a single compiler across different optimization levels. Each has a unique ability in finding compiler warnings defects.

In general, CVS and COS have lower false positive rates because different versions and optimization levels of the same compiler are usually quite consistent in producing warning messages. CCS can detect more types of warning defects than the other two, but it can also report more false positives. In particular, it fails if the compilers c_1 and c_2 support different sets of warning types. Although GCC and Clang share a majority of flags, they still have incompatibilities. For example, GCC has a command option `-Wunused-but-set-variable` to warn on variables that are set but never used, whereas Clang does not. As a result, we cannot use differential testing to validate the correctness of this warning diagnostic. In this regard, the CVS and COS strategies may serve as good complements to CCS, because they test the compiler warnings from different perspectives and only require a single compiler.

Algorithm 1: Parsing the warning output of a compiler

Input: text, the textual warning output of a compiler
Output: a set of parsed warning records

```
Function Parse (String text)
    msg_parsers ← a compiler-specific set of warning message parsers
    list ← split the string text into a list, in which each element is a string
        representation for a single warning
    result ← ∅
    foreach warning w ∈ list do
        foreach message parser p ∈ msg_parsers do
            if p.accept(w) then
                /* If w is parsable by p, parse w to a record
                   */
                record ← p.parse(w)
                result ← result ∪ {record}
                break
    return result
```

3.3 Parsing Warnings

Since compiler warnings are in natural languages and different compilers describe warnings in different ways, it is difficult/impossible to directly compute the symmetric difference of w_1 and w_2. To tackle this challenge, we design a specific parser for each compiler to parse its warning messages.

Algorithm 1 presents the general workflow of parsing the warning output of a particular compiler. Initially, we obtain a string `text` containing all the warning messages, and then split it into a list where each element is a textual representation of an individual warning. For each type of textual warnings, we devise a specific message parser. Each message parser has two functions:

- **`accept()`** This function tests whether a string warning is parsable by this message parser. For each type of warnings, we design a regular expression (RE) as a signature of the warning type. If a warning message matches this RE, then it falls into this type. The design of REs is based on the warning messages embedded in the compiler source code. For example, we design the following RE to parse the warning message component in Figure 5c: `"large integer implicitly truncated to .+ type"`
- **`parse()`** This function parses the warning string to a structured record by extracting the location (*i.e.*, which file, which line, and which column), the warning description, and the type of the warning. For example, the warning in Figure 5c can be parsed by a GCC message parser into the record in Table 1.

Table 1: The warning record parsed from Figure 5c

File	s.c	**Line**	3
Column	23	**Type**	overflow
Message	large integer implicitly truncated to unsigned type		
Misc.	**Target**=unsigned		

In total, we implemented 118 distinct warning message parsers for GCC and 107 ones for Clang, covering 106 distinct types of

Algorithm 2: Aligning two sets of warning records

```
Input: w1 and w2, warning records parsed from two compilers
Output: symmetric difference between w1 and w2
1  Function Align (Set w1, Set w2)
2      rm1 ← ∅            /* a set of elements to remove from w1  */
3      rm2 ← ∅            /* a set of elements to remove from w2  */
       /* Step 1. remove equivalent pairs                          */
4      foreach (a, b) ∈ (w1 × w2) do
5          if (a, b) is an equivalent pair then
6              rm1 ← rm1 ∪ {a}, rm2 ← rm2 ∪ {b}
       /* Step 2. compute pairs with unmatched columns             */
7      columns = ∅        /* a set of pairs with unmatched columns */
8      foreach (a, b) ∈ ((w1 \ rm1) × (w2 \ rm2)) do
9          if (a, b) has only unmatched columns then
10             columns = columns ∪ {(a, b)}
11             rm1 ← rm1 ∪ {a}, rm2 ← rm2 ∪ {b}
       /* Step 3. compute pairs with missing records               */
12     missing = ∅        /* a set of pairs with missing records   */
13     foreach a ∈ (w1 \ rm1) do
14         missing = missing ∪ {(a, ⊥)}
15     foreach b ∈ (w2 \ rm2) do
16         missing = missing ∪ {(⊥, b)}
17     return (columns, missing)
```

warnings. All parsers are precise at parsing target warnings as we design them by referring to the warning message templates encoded in the compiler source code. Our parsers also allow certain degree of variations/flexibilities (*e.g.*, different variable names in warning messages), which significantly reduces the number of parsers we need to implement.

3.4 Aligning Warnings

Inconsistencies among compilers, compiler versions, or optimization levels are identified by aligning the warnings in w_1 and w_2. The result of alignment is a list of pairs, of which the first element is either a warning in w_1 or $\perp$ (*i.e.*, nothing), and the other is either a warning in w_2 or $\perp$. The alignment process produces the following three categories of pairs (a, b):

- **Equivalence** $a \in w_1 \wedge b \in w_2$, and both have the same warning type and the same location (*i.e.* file, line, and column). This category is not of interest.
- **Unmatched Columns** $a \in w_1 \wedge b \in w_2$, and both have the same warning type and are on the same line, in the same file but in different columns. This category may indicate bugs of incorrect column numbers in warnings.
- **Missing Records** $(a \in w_1 \wedge b = \perp) \vee (a = \perp \wedge b \in w_2)$. This category constitutes the main body of inconsistencies for users to investigate.

Algorithm 2 describes the alignment process. It first removes all equivalent pairs from w_1 and w_2 (between lines 4 and 6). It then computes pairs with unmatched columns (between lines 7 and 11). Finally, it constructs the inconsistent pairs from the remaining warnings in $(w_1 \setminus rm_1)$ and $(w_2 \setminus rm_2)$ (between lines 12 and 16).

3.5 Filtering Warning Inconsistencies

After reporting an inconsistency to compiler developers, we need to temporarily stop testing this type of inconsistencies until it is fixed. We also need to eliminate false positive warnings to avoid unnecessary human inspection. The "Filter" component discards such warning pairs produced by the "Warning Aligner" component. The filter determines whether to remove a warning pair (a, b) based on its signature — a triple $(P_a, P_b, \textit{category})$ defined as follows,

Algorithm 3: Reducing a Test Program

```
Input: p, a test program
Input: c1 and c2, two compilers under testing
Input: pred, a predicate over the aligned warnings of c1 and c2 specifying the
       symptom of a warning difference between c1 and c2
Output: min, a minimal test program reduced from p satisfying the predicate
        pred
1  Function Reduce (p, pred, c1, c2)
2      min ← p
3      while true do
           // use C-Reduce [33] or Delta [28,38]
4          temp ← reduce(min)
5          if temp = min then          /* cannot be further reduced */
6              break
7          w1 ← Parse(c1.warn(temp))
8          w2 ← Parse(c2.warn(temp))
9          alignment ← Align(w1, w2)
10         if pred(alignment) then min = temp
11     return min
```

***P_a*:** the warning message parser that successfully parses a

***P_b*:** the warning message parser that successfully parses b

***category*:** the category of this warning pair, *i.e.*, *equivalence*, *unmatched column* or *missing records*

This triple signature is able to precisely differentiate warning pairs because the parsers P_a and P_b capture the exact information of the warnings a and b (*e.g.*, types, content). The filter component in Figure 3 maintains a set S of signatures of warning pairs to filter. If a newly discovered warning pair matches any signature in S then it is removed.

3.6 Reducing Test Programs

We generate large programs to increase the likelihood of triggering bugs. Once a test program p triggers a warning inconsistency (a, b) between two compilers c_1 and c_2, it is necessary to reduce p to a smaller size by removing program elements irrelevant to the inconsistent pair (a, b). This step is important, as it not only helps us understand the bug and avoid reporting a duplicate, but also assists developers in triaging/fixing the bug. This reduction process is generally more complex than the reduction process of compiler miscompilations and crashes [21, 37]. Test reduction for a miscompilation or a crashing bug only requires testing the behavior of the executables or exit statuses (integers) of compilers, whereas in our case, we need to tackle the textual warning output of compilations and need more expressive predicates to specify the symptoms of (a, b).

Algorithm 3 describes the overall procedure to reduce p. The invariant throughout the reduction process is that after reduction both compilers c_1 and c_2 still output the same inconsistent warning pair for the reduced program *min*. This invariant is encoded in the parameter *pred*, a predicate for testing whether the alignment of two warning sets w_1 and w_2 still preserve the inconsistency. The `reduce()` function on line 4 can be implemented with standard reduction tools such as C-Reduce [33] or Delta [28, 38]. We encapsulate all the parsing and aligning functionalities as a library and specify the invariant predicate as a Boolean method in a modern programming language on top of the library. Our reduction process is effective. A test program with several thousand lines of code can usually be reduced to a few lines (usually within five lines).

4. EMPIRICAL EVALUATION

We have been experimenting with Epiphron on GCC and Clang for six months. Although the two compilers are mature and stable, in the past six months, we are still able to report 60 bugs to GCC, of which 38 have been confirmed, assigned or fixed; and 39 bugs to Clang, of which 14 have been confirmed or fixed.

4.1 Testing Setup

Hardware and Compiler. Our evaluation has been conducted on a Linux PC with Intel(R) Core(TM) i7 CPU@2.67GHz and 12GB RAM. For each compiler (*i.e.* GCC and Clang), we test its daily built development trunk, because developers fix bugs in trunk more promptly than in stable versions. This reason further enables us to remove the filter on the reported bugs (described in Subsection 3.5) timely so that we can stress test more warning types. Moreover, developers usually implement new languages features or fix bugs in trunk, yet the source code of warning diagnostics is much more stable than other components. Therefore, the development trunk is not necessarily more buggy than stable versions in terms of warning diagnostics. All our reported bugs except three affect the latest stable versions. In the cross-version strategy, we use GCC 4.8.2 and Clang 3.4 as reference compilers.

Warning Flags. By default, both GCC and Clang do not enable all warnings. For Clang, we use the following command flags to compile each source file:

```
clang -Weverything -pedantic -std=c<89|99|11>
```

The flag `-Weverything` enables all the diagnostics available in Clang [1], `-std` specifies which version of the C standard should be used for checking and compiling code, and `-pedantic` instructs the compiler to adhere strictly to the C standard. GCC does not have a flag to enable all warning diagnostics. Even `-Wall` and `-Wextra` together only enable a subset of warnings. We have to manually specify other warning flags of interest. The whole command line of flags of GCC is shown below; the interested reader may refer to [5] for more information.

```
gcc -Wall -Wextra -pedantic -std=c<89|99|11> -Wpadded -Wundef\
 -Wformat=2 -Winit-self -Wuninitialized -Wpacked -Wconversion\
 -Wfloat-equal -Wlogical-op -Wswitch-default -Wshadow -Wvla  \
 -Wmissing-prototypes  -Wcast-qual -Wcast-align -Wswitch-enum\
 -Wsign-conversion -Wwrite-strings -Wredundant-decls         \
 -Wmissing-field-initializers
```

Testing Period and Testing Strategies. We spent non-continuous six months on this project, of which over four months was devoted to studying the characteristics of historical warning bugs, designing algorithms, and developing various tools (*e.g.*, program generator, aligner, reducer). The rest of time was spent in testing GCC and Clang. Initially, we tested all the three strategies — CCS, CVS and COS. All of them detected bugs. However, later we started to focus on CCS as both CVS and COS became saturated. This is expected as discussed in Subsection 3.2.2

4.2 Quantitative Results

We next discuss some statistical properties of the discovered bugs.

Detected Bugs. Table 2 shows the details of all the bugs we reported so far. In total, we have reported 99 bugs, of which 52 are confirmed by developers and 21 are already fixed. There are still 33 bug reports pending developers' response. Note for Clang, we only have 14 out of 39 confirmed, which is likely due to limited human resources as we were told by active members of the LLVM community that some Apple developers went to work on Swift.[8]

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

Table 2: Information of All Reported Bugs

	GCC	Clang	Total
Reported	60	39	99
Confirmed	38	14	52
Pending	12	25	37
Rejected	10	0	10

Table 3 further lists the details of all confirmed bugs, including their identities, bug-triggering command flags, priorities and severities assigned by developers, current report statuses, bug types, and the differential testing strategies.

Bug Types. We categorize warning defects into three classes as mentioned in Section 1: *Erroneous Message*, *Spurious Warning* and *Missing Warning*. Table 4 shows the breakdown of the bug types of all the *confirmed* bugs.

Bug Importance. In GCC and Clang's bug repositories, the importance of bugs is described as the combination of two fields, `priority` and `severity`. Priority is used by developers to prioritize bugs to fix; severity measures the impact of bugs, ranging from the most severe, release blocker to the least enhancement. Both fields are adjusted by developers when they confirm bugs.

As shown in Table 3, all our confirmed bugs have the default priority P3, and none of them is downgraded to P4 or P5. Only two of our bugs are labeled as `minor` by developers, and the rest have the normal severity. This demonstrates the importance and necessity of detecting compiler warning bugs. Compiler developers also care about warning bugs, and in fact 21 are already fixed in the latest GCC and Clang releases.

Size of Reported Test Cases. All of the test programs that we reported to GCC and Clang bug tracking systems are under five lines of code. The size of the original test programs generated by Epiphron is around 2,000 lines of code on average. This demonstrates that our reduction process is quite effective at minimizing test programs.

4.3 Assorted Confirmed Bug Samples

This section samples some bugs detected by Epiphron to demonstrate its ability to find a broad range of warning defects. These bugs have real impact on developers and some are even related to security-critical problems, such as Clang bug #18905 discussed in Section 4.3.3.

4.3.1 *Erroneous Messages*

GCC bug #60350. GCC emits two warnings suggesting that the variables `pf` and `pv` may be used before they are initialized. However, both warnings point to a wrong location: line 5, containing neither `pf` nor `pv`.

```
1  void a(int i) {
2    int (*pf)[2]; int (*pv)[i + 1];
3    (i ?
4        pf
5          :   // <-- two warnings here.
6        pv);
7  }
```

4.3.2 *Spurious Warnings*

GCC bug #60036. The following function triggers a regression since GCC 4.8. GCC emits a conversion warning on the expression '`f ^= fn1() > a`' on line 4 suggesting that there is a conversion from `unsigned int` to `int` and it may cause the signedness of the result to change. However, as the sub-expression '`fn1() > a`'

Table 3: Confirmed Bugs

		ID	Flag	Priority	Severity	Status	Bug Type	Strategy
1	GCC	59520	pedantic	P3	Minor	Confirmed	Spurious	CCS
2	GCC	59846	Wtype-limits	P3	Normal	Fixed	Erroneous Msg	CCS
3	GCC	59871	Wunused-value	P3	Normal	Fixed	Missing	CCS
4	GCC	59932	Waggressive-loop-optimization	P3	Normal	Confirmed	Spurious	CCS
5	GCC	59940	Wconversion	P3	Normal	Fixed	Erroneous Msg	CCS
6	GCC	59963	Woverflow	P3	Normal	Fixed	Erroneous Msg	CCS
7	GCC	60018	Wconversion	P3	Normal	Confirmed	Spurious	**COS**
8	GCC	60021	Wsign-compare	P3	Normal	Confirmed	Spurious	**COS**
9	GCC	60036	Wsign-conversion	P3	Normal	Fixed	Spurious	***CVS***
10	GCC	60083	Wsign-conversion	P3	Normal	Confirmed	Spurious	**COS**
11	GCC	60087	Wsign-compare	P3	Normal	Fixed	Erroneous Msg	CCS
12	GCC	60090	Wsign-compare	P3	Normal	Confirmed	Spurious	**COS**
13	GCC	60103	Wsequence-point	P3	Normal	Confirmed	Missing	**COS**
14	GCC	60114	pedantic	P3	Normal	Fixed	Erroneous Msg	CCS
15	GCC	60129	enabled by default	P3	Normal	Assigned	Erroneous Msg	CCS
16	GCC	60139	pedantic	P3	Normal	Fixed	Erroneous Msg	CCS
17	GCC	60170	Wtype-limits	P3	Normal	Confirmed	Missing	**COS**
18	GCC	60257	Woverride-init	P3	Normal	Fixed	Erroneous Msg	CCS
19	GCC	60279	Wuninitialized	P3	Normal	Confirmed	Erroneous Msg	CCS
20	GCC	60350	Wmaybe-uninitialized	P3	Minor	Confirmed	Erroneous Msg	CCS
21	GCC	60351	enabled by default	P3	Normal	Fixed	Erroneous Msg	CCS
23	GCC	60439	Wswitch	P3	Normal	Fixed	Missing	CCS
22	GCC	60440	Wreturn-type	P3	Normal	Confirmed	Spurious	CCS
24	GCC	60455	Woverflow	P3	Normal	Fixed	Erroneous Msg	CCS
25	GCC	61852	Wimplicit-function-declaration	P3	Normal	Fixed	Erroneous Msg	CCS
26	GCC	61854	pedantic	P3	Normal	Fixed	Missing	CCS
27	GCC	61861	Wdiscarded-qualifiers	P3	Normal	Confirmed	Erroneous Msg	CCS
28	GCC	61864	Wcovered-switch-default	P3	Normal	Confirmed	Missing	CCS
29	GCC	64423	Wchar-subscripts	P3	Normal	Fixed	Erroneous Msg	CCS
30	GCC	64440	Wdiv-by-zero	P3	Normal	Fixed	Missing	CCS
31	GCC	64577	Wpadded	P3	Normal	Confirmed	Missing	CCS
32	GCC	64609	Wbool-compare	P3	Normal	Confirmed	Missing	CCS
33	GCC	64610	Wbool-compare	P3	Normal	Fixed	Missing	CCS
34	GCC	64637	Wunused-value	P3	Normal	Confirmed	Erroneous Msg	CCS
35	GCC	64639	Wunused-value	P3	Normal	Confirmed	Missing	CCS
36	GCC	64648	Wunused-value	P3	Normal	Confirmed	Erroneous Msg	CCS
37	GCC	65430	Wsequence-point	P3	Normal	Confirmed	Missing	CCS
38	GCC	67243	Wvla	P3	Normal	Confirmed	Erroneous Msg	CCS
39	Clang	18504	Wsign-compare	P3	Normal	Confirmed	Missing	CCS
40	Clang	18796	Wtautological	P3	Normal	Confirmed	Missing	CCS
41	Clang	18801	Weverything	P3	Normal	Confirmed	Missing	***CVS***
42	Clang	18803	Wsequence-point	P3	Normal	Confirmed	Missing	CCS
43	Clang	18877	Wempty-body	P3	Normal	Confirmed	Missing	CCS
44	Clang	18905	Wformat	P3	Normal	Fixed	Missing	CCS
45	Clang	18923	Wc++-compat	P3	Normal	Confirmed	Missing	CCS
46	Clang	22059	Wshift-count-negative	P3	Normal	Fixed	Missing	CCS
47	Clang	22318	Wuninitialized	P3	Normal	Confirmed	Missing	CCS
48	Clang	22899	Winteger-overflow	P3	Normal	Fixed	Missing	CCS
49	Clang	23903	Wstrict-overflow	P3	Normal	Confirmed	Missing	CCS
50	Clang	24026	Wshift-negative-value	P3	Normal	Fixed	Missing	CCS
51	Clang	24238	Wtautological-overlap-compare	P3	Normal	Confirmed	Missing	CCS
52	Clang	24451	error	P3	Normal	Confirmed	Spurious	CCS

Table 4: Bug Types of Confirmed Bugs

	GCC	Clang	Total
Erroneous Message	18	0	18
Spurious Warning	8	1	9
Missing Warning	12	12	24
Total	38	14	52

returns either 1 or 0, the scenario reported in the warning will never happen.

```
extern int fn1();
unsigned fn(int a) {
  unsigned f = 9;
  f ^= fn1() > a;
  return f;
}
```

4.3.3 *Missing Warnings*

Compared to erroneous messages and spurious warnings which may take developers extra time to analyze, missing warnings sometimes have a severe negative impact on software development, as they hide bugs from developers and delay bug-fixing.

Clang bug #18796. In the following code, the compiler is expected to emit a warning indicating that the expression ‘`a < 0L`’ is always false, because the parameter `a` is unsigned, and thus its minimum value is 0. However, as the two operands of the operator

`<` are of different types, the parameter `a` is automatically promoted to a signed long, of which the minimum value becomes a negative number. As a result, Clang misses this warning.

```
int fn(unsigned a) { return a < 0L; }
```

Clang bug #18905. The following program has a bug which may lead to illegal memory access. The problem is that the format string `s` is not null-terminated (*i.e.* not ending with '`\0`'), and the function `printf` prints it. The consequence of such a bug can be severe, as it is also a type of software vulnerability, which could be potentially used in security exploits. Clang fails to identify this problem.

```
void fn() { const char s[1] = "format"; printf(s); }
```

4.4 Unconfirmed Bugs

We still have a number of bugs pending developers' confirmation. This is especially true for Clang. The following shows two of them, which we believe will be eventually accepted.

Clang bug #18875. This bug is a missing warning. In the following program, the function `foo` on line 1 accepts a parameter of type `double*`. But it is first cast to a function pointer, which accepts a parameter of type `int*` defined on line 2, and then is called through this pointer on line 6. This behavior is undefined in the C standard, and is implementation-dependent.

```
1  int foo(double *x) {return (int)*x;}
2  typedef int (*F)(int*);
3  int main() {
4    int x = 9;
5    // incompatible pointer cast
6    return ((F)foo)(&x);
7  }
```

GCC bug #60256. This bug is a missing warning of GCC. The function call to `strcpy` on line 5 uses the uninitialized variable `s`. However, GCC does not warn on it, as this call is optimized away based on its semantics (*i.e.*, copying a string to itself is redundant) before a warning can be generated. This "clever" behavior hides the fact that the code is problematic and not portable. When we compile it with Clang, the compiled program triggers a segmentation fault at runtime.

```
1  #include<string.h>
2  void f(void) {  char* s; strcpy(s, s);  }
```

4.5 Debatable Cases (or Compiler Smells)

This section discusses two bugs that were not accepted, but we believe that fixing them is still beneficial and can further improve the usability of compilers.

GCC bug #60121. The following code snippet has an obvious undefined behavior — accessing an array with an index out of its bound. However, when GCC compiles it with an optimization level under `-O2` (*i.e.*, at `-O0` or `-O1`), no warning is emitted on the illegal access on line 2. The reason is that GCC needs to perform value range propagation analysis in order to emit the warning, but the analysis is only enabled above `-O1`. In contrast, Clang has a better design that separates warnings from optimizations, and thus the problem in the code is always alerted.

```
1  int b[1];
2  int f() { return b[9999]; }
```

GCC bug #59939. The following code snippet raises some debate whether to emit warnings for unreachable code. The problem is at the function call '`fn1(a, b)`' on line 3. It expects two *unsigned* parameters, but the actual arguments are both of *signed* type. Moreover, the code is also unreachable, as the left operand of the logical or operator `||` is 1. The current behavior of GCC just simply emits nothing for the code, whereas Clang emits three warnings, two for the signedness changes of the parameters, and one for the unreachable code. A reasonable fix of GCC is to warn on the dead code, but it is nontrivial and takes time as discussed at http://gcc.gnu.org/bugzilla/show_bug.cgi?id=4210#c22.

```
1  int a, b;
2  int fn1(unsigned, unsigned);
3  unsigned int fn2() {  return 1 || fn1(a, b);  }
```

This example also demonstrates the importance and necessity of the program generator of Epiphron. The developers of both GCC and Clang communities are aware of this difference between the two compilers and stand for their current designs. Therefore, avoiding dead code in test programs can prevent such inconsistencies from reaching human investigation.

5. DISCUSSIONS

In this section, we discuss the precision of our technique (*i.e.*, its false positive rate), whether static rule checkers can help detect these warning bugs we have found, and the comparison between Epiphron program generator and Csmith.

False Positive Rate. The false positives of our technique are the warning inconsistencies that are rejected by compiler developers. They generally originate from two sources: (1) the inconsistency is duplicate to an existing bug report; (2) the warning diagnostics of the inconsistency is not supported by both compilers.

As mentioned in Section 3.5, for the first case, after reporting a bug, we temporarily disable checking the same type of inconsistencies until the bug has been fixed; for the second case, we design a list of filters to weed out these known inconsistencies. Moreover, the program generator of Epiphron is designed to reduce false positives by generating warning-free code in bodies of conditional statements (*cf.*, Subsection 3.1).

These two mechanisms work well in practice. Therefore, we compute the false positive rate as a value within the following range,

$$\left[\frac{\texttt{rejected}}{\texttt{reported}}, \frac{\texttt{rejected} + \texttt{pending}}{\texttt{reported}}\right]$$

In our evaluation, the range is $[\frac{10}{99}, \frac{10+37}{99}] = [10\%, 47\%]$. Note that 47% is simply an upper bound of the false positive rate, which is mainly due to a relatively large number of pending bugs (especially for Clang). When reporting a bug, we have carefully checked its validity. We believe some of the pending bugs will eventually be accepted.

Static Analysis Checker. Static analysis checkers use static analysis to detect bugs in source code, *e.g.*, FindBugs [10], Clang Static Analyzer [2] and PMD [6]. They can catch common program flaws and bugs at the early stage of development. However, *all* the bugs reported in this paper are not detectable for them, as these bugs are semantic bugs and specific to compilers. That is, in terms of warning diagnostics, the manifestation of these bugs is just a symptom that the behavior of the compiler does not conform to the developers' intention. Even though these tools were able to detect the type of these bugs, the large code base and the complexity of GCC and Clang would make the checkers hardly scale.

Comparison with Csmith. Csmith is a program generator aiming to stress test compiler optimizers and code generators. It only supports a limited set of C language features. For example, it does not support enumerations or switch statements. Epiphron program generator outperforms Csmith in terms of warning bug detection, as it supports nearly all features of C language. We have already found 14 warning bugs that Csmith cannot detect. For example, Epiphron detected GCC #61864 that involves enumerations and switch statements.

Cascaded Compiler Warnings. A compiler emits an error if the program under compilation does not follow the grammar or the typing rules. This error often results in other related errors, referred to as *cascaded errors*. Differently, compiler warnings are usually not cascaded. Each warning is generated locally and independent of others. As the focus of our work is detecting bugs in compiler warning diagnostics rather than compiler errors, all our test programs are syntactically valid and compilable. Therefore Epiphron is not affected by this complex scenario (*i.e.*, cascaded compiler warnings/errors).

6. RELATED WORK

This section surveys related work on validating/testing compiler and improving warning/error message systems.

6.1 Compiler Testing

Compiler testing still remains the dominant technique for validating the correctness of production compilers. Besides internal regression test suites, compiler developers may use commercial test suites for conformance checking and validation [7, 32]. Since it is expensive to maintain and develop such manually written test suites, people recently leverage randomized testing to complementarily generate massive test cases to further validate compilers [11, 21, 29, 37, 39]. Among them, two notable efforts are Csmith [13, 33, 37] and Orion [21]. Both are proved to be very effective in practice; each has found several hundreds of crashing and miscompilation bugs in production compilers (GCC and LLVM).

Csmith [13, 33, 37] is based on differential testing [27] (which has also been applied to test virtual machines [25] and CPU emulators [26]). Csmith generates random C programs and checks for inconsistent behavior across different compilers or compiler versions. It has also been applied to find bugs in static analyzers, such as Frama-C [15]. The major contributions of Csmith are the number of C language constructs that it supports, and the ability to generate complex programs that are free of undefined behavior most of the time. Orion [21] presents a novel technique to systematically modify existing code (either real or randomly generated) and generate many test cases that are semantically equivalent to the original program *w.r.t.* an input set. Instead of verifying different compilers (or compiler versions) behave exactly the same on a program, it verifies that a compiler must behave the same on all test cases generated from a program under an input set.

Although Epiphron shares the same theme of differential testing, it targets a different class of compiler bugs: compiler warnings. This brings up new technical challenges, as we need to design a new program generator to stress-test warning diagnostics—a component in the frontend, define the "equivalence" of compiler warnings across different compilers, compiler versions, and compiler optimization flags. In contrast, Csmith and Orion aim to test compiler optimizers and code generators with less focus on the diversity of language constructs used in test programs. They pay more attention on validity of the semantics of test programs, and only need to check for equivalence of the execution output, which is well-defined for integer programs.

Another related program generator is CCG [11], which produces random compilable programs to look for crashing bugs in C compilers. However, it only supports a limited set of C features. Therefore it is not as effective as Epiphron in detecting compiler warning bugs. Mutation testing is also related [9, 20]. In particular, we can mutate test programs so that more compiler warnings can be triggered. However, it is not clear how to design effective mutation operators yet. We leave it as future work.

6.2 Compiler Errors and Warnings

The general problem of building good warning/error message systems has been long acknowledged [35]. Shneiderman [34] presented a few guidelines on building such systems, and showed that a good system could improve user productivity and satisfaction. Brown [12] articulated the concern that little interest was paid by the community to error message design. His analysis on Pascal compilers showed that the messages were generally disappointing and did not clearly show suggestions for correction. This problem is even more important in the context of learning and teaching, as novice developers may spend hours on a simple error [16].

There have been some efforts to alleviate this problem. For instance, when a program is ill-typed, the compiler (instructed by its type checker) often reports error locations far away from the source problem [36]. Lerner *et al.* [24] proposed a simple solution: instead of reporting imprecise error messages provided by type-checkers, they search for a similar programs that do type-check and present them to users. Coull [14] developed a database with common compilers errors together with their likely solutions. When an error is encountered, the system shows both the original message and its solution. The authors demonstrated that the system has positive impact on the learning of students. Alternatively, users may also look at how their peers fixed the warnings/errors in the similar context, and apply similar changes to their programs [17].

Epiphron is complementary. Despite having the same general goal — to improve warning/error systems — with previous work, Epiphron has a completely different execution. It finds defects in such systems by finding their inconsistencies under the same configuration.

7. CONCLUSION

We have described an approach based on randomized differential testing to finding compiler warning defects and implemented it in the Epiphron tool. Our empirical evaluation has shown that Epiphron is very effective in detecting warning bugs in mature compilers. Within only six months of testing (including four-months development), we have reported 99 bugs, of which 52 have been confirmed, assigned or fixed to-date.

Our work is the very first extensive effort in testing compilers' warning support. We believe that it opens up a new direction of research to improve the correctness and usability of compiler warnings and errors. We are actively pursuing future work to (1) extend the proposed technique to other languages such as C++, (2) design a grey-box approach to testing compiler warnings by incorporating coverage of compilers and (3) support the testing of compiler error messages. The data and source code used in this paper are publicly available at http://chengniansun.bitbucket.org/projects/epiphron.

Acknowledgments

We are grateful to the anonymous reviewers for their insightful comments. We also would like to thank the GCC and Clang/LLVM developers for analyzing and fixing our reported bugs. Our evaluation benefited significantly from the Berkeley Delta [28], and University of Utah's Csmith [37] and C-Reduce [33] tools.

This research was supported in part by the United States National Science Foundation (NSF) Grants 1117603, 1319187, 1349528, and 1528133. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

8. REFERENCES

[1] Clang Compiler User's Manual 3.5. http://clang.llvm.org/docs/UsersManual.html#diagnostics-enable-everything, accessed: 2014-03-04.

[2] Clang Static Analyzer. http://clang-analyzer.llvm.org/, accessed: 2015-08-10.

[3] GCC Bug #28901. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=28901, accessed: 2014-03-04.

[4] GDB Bug. https://sourceware.org/ml/gdb-patches/2014-02/msg00342.html, accessed: 2014-03-04.

[5] Options to Request or Suppress Warnings – GCC. http://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html, accessed: 2014-03-04.

[6] PMD. http://pmd.github.io/, accessed: 2015-08-10.

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

[8] A. Allain. Why Bother with Compiler Warnings. http://www.cprogramming.com/tutorial/compiler_warnings.html, accessed: 2014-12-10.

[9] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, New York, NY, USA, 1 edition, 2008.

[10] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou. Evaluating static analysis defect warnings on production software. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 1–8. ACM, 2007.

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

[12] P. J. Brown. Error messages: The neglected area of the man/machine interface. *Commun. ACM*, 26(4):246–249, Apr. 1983.

[13] Y. Chen, A. Groce, C. Zhang, W.-K. Wong, X. Fern, E. Eide, and J. Regehr. Taming Compiler Fuzzers. In *PLDI*, pages 197–208, 2013.

[14] N. J. Coull. *SNOOPIE: development of a learning support tool for novice programmers within a conceptual framework*. Ph.d. thesis, University of St Andrews, St Andrews, Scotland, UK, Oct. 2008.

[15] P. Cuoq, B. Monate, A. Pacalet, V. Prevosto, J. Regehr, B. Yakobowski, and X. Yang. Testing Static Analyzers with Randomly Generated Programs. In A. Goodloe and S. Person, editors, *NASA Formal Methods*, volume 7226 of *Lecture Notes in Computer Science*, pages 120–125. Springer Berlin Heidelberg, 2012.

[16] T. Flowers, C. Carver, and J. Jackson. Empowering students and building confidence in novice programmers through gauntlet. In *Frontiers in Education, 2004. FIE 2004. 34th Annual*, pages T3H/10–T3H/13 Vol. 1, Oct 2004.

[17] B. Hartmann, D. MacDougall, J. Brandt, and S. R. Klemmer. What Would Other Programmers Do: Suggesting Solutions to Error Messages. In *CHI*, pages 1019–1028, 2010.

[18] M. Howard. A Process for Performing Security Code Reviews. *IEEE Security & Privacy*, 4(4):0074–79, 2006.

[19] International Organization for Standarization. *ISO/IEC 9899:201x:Programming Languages–C*. http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1570.pdf, May 2011.

[20] Y. Jia and M. Harman. An Analysis and Survey of the Development of Mutation Testing. *IEEE Trans. Softw. Eng.*, 37(5):649–678, Sept. 2011.

[21] V. Le, M. Afshari, and Z. Su. Compiler Validation via Equivalence Modulo Inputs. In *PLDI*, 2014.

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

[23] V. Le, C. Sun, and Z. Su. Randomized Stress-testing of Link-time Optimizers. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 327–337, 2015.

[24] B. S. Lerner, M. Flower, D. Grossman, and C. Chambers. Searching for Type-error Messages. In *PLDI*, pages 425–434, 2007.

[25] L. Martignoni, R. Paleari, G. Fresi Roglia, and D. Bruschi. Testing System Virtual Machines. In *ISSTA*, pages 171–182, 2010.

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

[27] W. M. McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998.

[28] S. McPeak, D. S. Wilkerson, and S. Goldsmith. Berkeley Delta. http://delta.tigris.org/.

[29] E. Nagai, A. Hashimoto, and N. Ishiura. Scaling up size and number of expressions in random testing of arithmetic optimization of C compilers. In *SASIMI)*, pages 88–93, 2013.

[30] P. Norvig. Learning Programming (by Humans, by Machine). http://vimeo.com/69631070, accessed: 2014-03-04.

[31] T. Pearse and P. Oman. Maintainability measurements on industrial source code maintenance activities. In *Software Maintenance, 1995. Proceedings., International Conference on*, pages 295–303. IEEE, 1995.

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

[33] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang. Test-Case Reduction for C Compiler Bugs. In *PLDI*, pages 335–346, 2012.

[34] B. Shneiderman. Designing computer system messages. *Commun. ACM*, 25(9):610–611, Sept. 1982.

[35] V. J. Traver. On compiler error messages: What they say and what they mean. *Adv. in Hum.-Comp. Int.*, 2010:3:1–3:26, Jan. 2010.

[36] M. Wand. Finding the Source of Type Errors. In *POPL*, pages 38–43, 1986.

[37] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and Understanding Bugs in C Compilers. In *PLDI*, pages 283–294, 2011.

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

[39] C. Zhao, Y. Xue, Q. Tao, L. Guo, and Z. Wang. Automated Test Program Generation for an Industrial Optimizing Compiler. In *AST*, pages 36–43, 2009.