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 location information.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

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

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.
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 mechanisms directly 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:

- 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.
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 parses the compilation output of GCC or Clang to highlight problematic code. This type of warning bugs will make Eclipse behave bizarrely.

**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. 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 $P$, 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:

---

1. [https://eclipse.org/ctd/](https://eclipse.org/ctd/)
2. [http://llvm.org/bugs/show_bug.cgi?id=18504#c1](http://llvm.org/bugs/show_bug.cgi?id=18504#c1)
• **Random Testing**  We first use $P$ 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.

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

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 immutuable 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, 5Clang 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 = -0x4578A0B8CA1DE677LL ^ (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
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

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.

### 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.

- `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>
<thead>
<tr>
<th>File</th>
<th>Line</th>
<th>Column</th>
<th>Message</th>
<th>Misc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>s.c</td>
<td>3</td>
<td>23</td>
<td>large integer implicitly truncated to unsigned type</td>
<td>Target=unsigned</td>
</tr>
</tbody>
</table>

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: \( w_1 \) and \( w_2 \), warning records parsed from two compilers
Output: symmetric difference between \( w_1 \) and \( w_2 

Function Align \(( \text{Set } w_1, \text{Set } w_2) \)

1. \( \text{rm}_1 \leftarrow \emptyset \) \hspace{1cm} /* a set of elements to remove from \( w_1 \) */
2. \( \text{rm}_2 \leftarrow \emptyset \) \hspace{1cm} /* a set of elements to remove from \( w_2 \) */
3. /* Step 1. remove equivalent pairs */
   4. foreach \((a, b) \in (w_1 \times w_2)\) do
      5. if \((a, b) \) is an equivalent pair then
         6. \( \text{rm}_1 \leftarrow \text{rm}_1 \cup \{(a, b)\}, \text{rm}_2 \leftarrow \text{rm}_2 \cup \{b\} \)
   7. /* Step 2. compute pairs with unmatched columns */
   8. columns = \( \emptyset \) \hspace{1cm} /* a set of pairs with unmatched columns */
   9. foreach \((a, b) \in ((w_1 \setminus \text{rm}_1) \times (w_2 \setminus \text{rm}_2))\) do
      10. if \((a, b) \) has only unmatched columns then
          11. columns = columns \( \cup \{(a, b)\} \)
   12. /* Step 3. compute pairs with missing records */
   13. missing = \( \emptyset \) \hspace{1cm} /* a set of pairs with missing records */
   14. foreach \(a \in (w_1 \setminus \text{rm}_1)\) do
      15. missing = missing \( \cup \{(a, \_\_\_\_\_\_)\} \)
   16. foreach \(b \in (w_2 \setminus \text{rm}_2)\) do
      17. missing = missing \( \cup \{\_\_\_\_, b\} \)
   18. return \((\text{columns}, \text{missing})\)

Algorithm 3: Reducing a Test Program

Input: \( p \), a test program
Input: \( c_1 \) and \( c_2 \), two compilers under testing
Input: \( \text{pred} \), a predicate over the aligned warnings of \( c_1 \) and \( c_2 \) specifying the symptom of a warning difference between \( c_1 \) and \( c_2 \)
Output: \( \text{min} \), a minimal test program reduced from \( p \) satisfying the predicate \( \text{pred} \)

Function Reduce \((p, \text{pred}, c_1, c_2)\)

1. \( \text{min} \leftarrow p \)
2. while \( \text{true} \) do
   3. if use C-Reduce [33] or Delta [28, 38] then
   4. \( \text{temp} \leftarrow \text{true} \) \hspace{1cm} /* cannot be further reduced */
   5. break
   6. \( w_1 \leftarrow \text{Parse}(c_1, \text{warn}(\text{temp})) \)
   7. \( w_2 \leftarrow \text{Parse}(c_2, \text{warn}(\text{temp})) \)
   8. alignment \( \leftarrow \text{Align}(w_1, w_2) \)
   9. if \( \text{pred}(\text{alignment}) \) then \( \text{min} = \text{temp} \)
10. return \( \text{min} \)

\( P_a \): 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 compilation 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 \( \text{min} \). This invariant is encoded in the parameter \( \text{pred} \), a predicate for testing whether the alignment of two warning sets \( w_1 \) and \( w_2 \) still preserve the inconsistency. The \( \text{reduce()} \) 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:

```bash
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.

```bash
   -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 the time was spent in testing GCC and Clang. Initially, we tested all the three strategies — CCS, CVS and cross-version (CCS-CVS). The rest of time was spent in testing GCC and Clang. We confirmed by developers and 21 are already fixed. There are still 33 bug reports pending developers’ response. Note for Clang, we have only 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.4

4.2 Quantitative Results

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 have only 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.4

Table 2: Information of All Reported Bugs

<table>
<thead>
<tr>
<th></th>
<th>GCC</th>
<th>Clang</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reported</td>
<td>60</td>
<td>39</td>
<td>99</td>
</tr>
<tr>
<td>Confirmed</td>
<td>38</td>
<td>14</td>
<td>52</td>
</tr>
<tr>
<td>Pending</td>
<td>12</td>
<td>25</td>
<td>37</td>
</tr>
<tr>
<td>Rejected</td>
<td>10</td>
<td>0</td>
<td>10</td>
</tr>
</tbody>
</table>

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.

```c
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

<table>
<thead>
<tr>
<th>ID</th>
<th>Flag</th>
<th>Priority</th>
<th>Severity</th>
<th>Status</th>
<th>Bug Type</th>
<th>Strategy</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>GCC 59520</td>
<td>pedantic</td>
<td>P3</td>
<td>Minor</td>
<td>Confirmed</td>
<td>Spurious</td>
</tr>
<tr>
<td>2</td>
<td>GCC 59846</td>
<td>Wtype-limits</td>
<td>P3</td>
<td>Normal</td>
<td>Fixed</td>
<td>Erroreneous Msg</td>
</tr>
<tr>
<td>3</td>
<td>GCC 59871</td>
<td>Wunused-value</td>
<td>P3</td>
<td>Normal</td>
<td>Fixed</td>
<td>Missing</td>
</tr>
<tr>
<td>4</td>
<td>GCC 59932</td>
<td>Wagggressive-loop-optimization</td>
<td>P3</td>
<td>Normal</td>
<td>Confirmed</td>
<td>Spurious</td>
</tr>
<tr>
<td>5</td>
<td>GCC 59940</td>
<td>Wconversion</td>
<td>P3</td>
<td>Normal</td>
<td>Fixed</td>
<td>Erroreneous Msg</td>
</tr>
<tr>
<td>6</td>
<td>GCC 59963</td>
<td>Woverflow</td>
<td>P3</td>
<td>Normal</td>
<td>Confirmed</td>
<td>Spurious</td>
</tr>
<tr>
<td>7</td>
<td>GCC 60018</td>
<td>Wconversion</td>
<td>P3</td>
<td>Normal</td>
<td>Fixed</td>
<td>Spurious</td>
</tr>
<tr>
<td>8</td>
<td>GCC 60021</td>
<td>Wsign-compare</td>
<td>P3</td>
<td>Normal</td>
<td>Confirmed</td>
<td>Spurious</td>
</tr>
<tr>
<td>9</td>
<td>GCC 60036</td>
<td>Wsign-conversion</td>
<td>P3</td>
<td>Normal</td>
<td>Fixed</td>
<td>Spurious</td>
</tr>
<tr>
<td>10</td>
<td>GCC 60083</td>
<td>Wsign-conversion</td>
<td>P3</td>
<td>Normal</td>
<td>Confirmed</td>
<td>Spurious</td>
</tr>
<tr>
<td>11</td>
<td>GCC 60087</td>
<td>Wsign-compare</td>
<td>P3</td>
<td>Normal</td>
<td>Fixed</td>
<td>Erroreneous Msg</td>
</tr>
<tr>
<td>12</td>
<td>GCC 60090</td>
<td>Wsign-compare</td>
<td>P3</td>
<td>Normal</td>
<td>Confirmed</td>
<td>Spurious</td>
</tr>
<tr>
<td>13</td>
<td>GCC 60103</td>
<td>Wsequence-point</td>
<td>P3</td>
<td>Normal</td>
<td>Confirmed</td>
<td>Missing</td>
</tr>
<tr>
<td>14</td>
<td>GCC 60114</td>
<td>pedantic</td>
<td>P3</td>
<td>Normal</td>
<td>Fixed</td>
<td>Erroreneous Msg</td>
</tr>
<tr>
<td>15</td>
<td>GCC 60129</td>
<td>enabled by default</td>
<td>P3</td>
<td>Normal</td>
<td>Assigned</td>
<td>Erroreneous Msg</td>
</tr>
<tr>
<td>16</td>
<td>GCC 60139</td>
<td>pedantic</td>
<td>P3</td>
<td>Normal</td>
<td>Fixed</td>
<td>Erroreneous Msg</td>
</tr>
<tr>
<td>17</td>
<td>GCC 60170</td>
<td>Wtype-limits</td>
<td>P3</td>
<td>Normal</td>
<td>Confirmed</td>
<td>Missing</td>
</tr>
<tr>
<td>18</td>
<td>GCC 60257</td>
<td>Woverride-init</td>
<td>P3</td>
<td>Normal</td>
<td>Fixed</td>
<td>Erroreneous Msg</td>
</tr>
<tr>
<td>19</td>
<td>GCC 60279</td>
<td>Wunninitialed</td>
<td>P3</td>
<td>Normal</td>
<td>Confirmed</td>
<td>Erroreneous Msg</td>
</tr>
<tr>
<td>20</td>
<td>GCC 60350</td>
<td>Wmaybe-uninitialized</td>
<td>P3</td>
<td>Normal</td>
<td>Confirmed</td>
<td>Erroreneous Msg</td>
</tr>
<tr>
<td>21</td>
<td>GCC 60351</td>
<td>enabled by default</td>
<td>P3</td>
<td>Normal</td>
<td>Fixed</td>
<td>Erroreneous Msg</td>
</tr>
<tr>
<td>22</td>
<td>GCC 60439</td>
<td>Wswitch</td>
<td>P3</td>
<td>Normal</td>
<td>Fixed</td>
<td>Missing</td>
</tr>
<tr>
<td>23</td>
<td>GCC 60440</td>
<td>Wreturn-type</td>
<td>P3</td>
<td>Normal</td>
<td>Confirmed</td>
<td>Spurious</td>
</tr>
<tr>
<td>24</td>
<td>GCC 60455</td>
<td>Woverflow</td>
<td>P3</td>
<td>Normal</td>
<td>Fixed</td>
<td>Erroreneous Msg</td>
</tr>
<tr>
<td>25</td>
<td>GCC 61852</td>
<td>Wimplicit-function-declaration</td>
<td>P3</td>
<td>Normal</td>
<td>Fixed</td>
<td>Erroreneous Msg</td>
</tr>
<tr>
<td>26</td>
<td>GCC 61854</td>
<td>pedantic</td>
<td>P3</td>
<td>Normal</td>
<td>Fixed</td>
<td>Missing</td>
</tr>
<tr>
<td>27</td>
<td>GCC 61861</td>
<td>Wdiscarded-qualifiers</td>
<td>P3</td>
<td>Normal</td>
<td>Confirmed</td>
<td>Erroreneous Msg</td>
</tr>
<tr>
<td>28</td>
<td>GCC 61864</td>
<td>Wcovered-switch-default</td>
<td>P3</td>
<td>Normal</td>
<td>Confirmed</td>
<td>Missing</td>
</tr>
<tr>
<td>29</td>
<td>GCC 64423</td>
<td>Wchar-subscripts</td>
<td>P3</td>
<td>Normal</td>
<td>Fixed</td>
<td>Erroreneous Msg</td>
</tr>
<tr>
<td>30</td>
<td>GCC 64440</td>
<td>Wdiv-by-zero</td>
<td>P3</td>
<td>Normal</td>
<td>Fixed</td>
<td>Missing</td>
</tr>
<tr>
<td>31</td>
<td>GCC 64577</td>
<td>Wpacked</td>
<td>P3</td>
<td>Normal</td>
<td>Confirmed</td>
<td>Missing</td>
</tr>
<tr>
<td>32</td>
<td>GCC 64609</td>
<td>Wbool-compare</td>
<td>P3</td>
<td>Normal</td>
<td>Confirmed</td>
<td>Missing</td>
</tr>
<tr>
<td>33</td>
<td>GCC 64610</td>
<td>Wbool-compare</td>
<td>P3</td>
<td>Normal</td>
<td>Fixed</td>
<td>Missing</td>
</tr>
<tr>
<td>34</td>
<td>GCC 64637</td>
<td>Wunused-value</td>
<td>P3</td>
<td>Normal</td>
<td>Confirmed</td>
<td>Erroreneous Msg</td>
</tr>
<tr>
<td>35</td>
<td>GCC 64639</td>
<td>Wunused-value</td>
<td>P3</td>
<td>Normal</td>
<td>Confirmed</td>
<td>Missing</td>
</tr>
<tr>
<td>36</td>
<td>GCC 64648</td>
<td>Wunused-value</td>
<td>P3</td>
<td>Normal</td>
<td>Confirmed</td>
<td>Erroreneous Msg</td>
</tr>
<tr>
<td>37</td>
<td>GCC 65430</td>
<td>Wsequence-point</td>
<td>P3</td>
<td>Normal</td>
<td>Confirmed</td>
<td>Missing</td>
</tr>
<tr>
<td>38</td>
<td>GCC 67243</td>
<td>WVla</td>
<td>P3</td>
<td>Normal</td>
<td>Confirmed</td>
<td>Erroreneous Msg</td>
</tr>
<tr>
<td>39</td>
<td>Clang 18504</td>
<td>Wsign-compare</td>
<td>P3</td>
<td>Normal</td>
<td>Confirmed</td>
<td>Missing</td>
</tr>
<tr>
<td>40</td>
<td>Clang 18760</td>
<td>Wtautological</td>
<td>P3</td>
<td>Normal</td>
<td>Confirmed</td>
<td>Missing</td>
</tr>
<tr>
<td>41</td>
<td>Clang 18801</td>
<td>Weverything</td>
<td>P3</td>
<td>Normal</td>
<td>Confirmed</td>
<td>Missing</td>
</tr>
<tr>
<td>42</td>
<td>Clang 18803</td>
<td>Wsequence-point</td>
<td>P3</td>
<td>Normal</td>
<td>Confirmed</td>
<td>Missing</td>
</tr>
<tr>
<td>43</td>
<td>Clang 18877</td>
<td>Wempty-body</td>
<td>P3</td>
<td>Normal</td>
<td>Confirmed</td>
<td>Missing</td>
</tr>
<tr>
<td>44</td>
<td>Clang 18905</td>
<td>Wformat</td>
<td>P3</td>
<td>Normal</td>
<td>Confirmed</td>
<td>Missing</td>
</tr>
<tr>
<td>45</td>
<td>Clang 18923</td>
<td>WC++-compat</td>
<td>P3</td>
<td>Normal</td>
<td>Confirmed</td>
<td>Missing</td>
</tr>
<tr>
<td>46</td>
<td>Clang 22059</td>
<td>Wshift-count-negative</td>
<td>P3</td>
<td>Normal</td>
<td>Fixed</td>
<td>Missing</td>
</tr>
<tr>
<td>47</td>
<td>Clang 22318</td>
<td>Wunused-value</td>
<td>P3</td>
<td>Normal</td>
<td>Confirmed</td>
<td>Missing</td>
</tr>
<tr>
<td>48</td>
<td>Clang 22899</td>
<td>Woverflow</td>
<td>P3</td>
<td>Normal</td>
<td>Fixed</td>
<td>Missing</td>
</tr>
<tr>
<td>49</td>
<td>Clang 23903</td>
<td>Wstrict-overflow</td>
<td>P3</td>
<td>Normal</td>
<td>Confirmed</td>
<td>Missing</td>
</tr>
<tr>
<td>50</td>
<td>Clang 24026</td>
<td>Wshift-negative-value</td>
<td>P3</td>
<td>Normal</td>
<td>Fixed</td>
<td>Missing</td>
</tr>
<tr>
<td>51</td>
<td>Clang 24238</td>
<td>Wtautological-overlap-compare</td>
<td>P3</td>
<td>Normal</td>
<td>Confirmed</td>
<td>Missing</td>
</tr>
<tr>
<td>52</td>
<td>Clang 24451</td>
<td>Error</td>
<td>P3</td>
<td>Normal</td>
<td>Confirmed</td>
<td>Spurious</td>
</tr>
</tbody>
</table>

Table 4: Bug Types of Confirmed Bugs

<table>
<thead>
<tr>
<th>Bug Type</th>
<th>GCC</th>
<th>Clang</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Erroreneous Message</td>
<td>18</td>
<td>0</td>
<td>18</td>
</tr>
<tr>
<td>Spurious Warning</td>
<td>8</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>Missing Warning</td>
<td>12</td>
<td>12</td>
<td>24</td>
</tr>
<tr>
<td>Total</td>
<td>38</td>
<td>14</td>
<td>52</td>
</tr>
</tbody>
</table>

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

```c
extern int fn1();
unsigned fn(int a) {
unsigned f = 9;
4   f ^= fn1() > a;
5   return f;
6 } `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.

```c
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.

```c
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.

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

**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.

```c
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 }
```

### 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.

```c
1 #include<string.h>
2 int b[1];
3 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.

```c
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,

\[
\begin{bmatrix}
\text{rejected} & \text{rejected + pending} \\
\text{reported} & \text{reported}
\end{bmatrix}
\]

In our evaluation, the range is \([10\%, 10\% + 1.47\%] = [10\%, 47\%]\). 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 misconpilation 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


