Detecting Compiler Warning Defects Via Diversity-Guided Program Mutation

Yixuan Tang, He Jiang, *Member, IEEE*, Zhide Zhou, Xiaochen Li, Zhilei Ren, and Weiqiang Kong, *Member, IEEE*

Abstract—Compiler diagnostic warnings help developers identify potential programming mistakes during program compilation. However, these warnings could be erroneous due to the defects of compiler warning diagnostics. Although the existing technique (i.e., Epiphron) can automatically generate test programs for compiler warning defect detection, the effectiveness of Epiphron on defect-finding is still limited, due to the limitation for generating warning-sensitive test program structures. Therefore, in this paper, we propose a DIversity-guided PROgram Mutation approach, called DIPROM, to construct diverse warning-sensitive programs for effective compiler warning defect detection. Given a seed test program, DIPROM first removes its dead code to reduce false positive warning defects. Then, the abstract syntax tree (AST) of the test program is constructed; DIPROM iteratively mutates the structures of the AST to generate warning-sensitive program variants. To effectively construct diverse warning-sensitive structures, DIPROM applies a novel diversity-guided strategy to generate program variants in each iteration. With the generated program variants, differential testing is conducted to detect warning defects in different compilers. In the experiments, we evaluate DIPROM with two popular C compilers (i.e., GCC and Clang). Experimental results show that DIPROM significantly outperforms three state-of-the-art approaches (i.e., HiCOND, Epiphron, and Hermes) by up to 18.93%∼76.74% in terms of the bug-finding capability on average. Meanwhile, DIPROM is efficient, which spends less time on finding the same average number of warning defects. We at last applied DIPROM to the latest development versions of GCC and Clang. After two months' running, we reported 8 new warning defects; 5 of them have been confirmed/fixed by developers.

Index Terms—Compiler Testing, Differential Testing, Program Mutation, Test Program Generation

1 Introduction

Compiler warnings are widely used by developers to detect potential programming mistakes at compilation time [1], [2]. A warning diagnostic message provides the reason of the warning and the location information of the problematic code (e.g., the line and the column) to help developers analyze programming mistakes. However, similar to other application software, compilers still contain bugs. In fact, more than 100 compiler warning defects are reported for widely used compilers such as GCC and LLVM [2]. Due to compiler defects, warning diagnostic messages could be erroneous, spurious, and missing. Buggy compiler warnings negatively impact the usability of compilers and the productivity of developers [2]. For example, Clang misses a warning diagnostic in the program which may lead to illegal memory access [3]. The consequence of such a warning defect could be severe, since it is also a type of software vulnerability that may result in disastrous software failures especially in safety-critical domains. Therefore, it is crucial to ensure the quality of compiler warnings.

To detect compiler defects, compiler testing is an effective approach [4], [5], [6]. Compiler testing usually employs program generation tools to generate test programs with various language features. These test programs are fed into different compilers to trigger unexpected or inconsistent behaviors, which may indicate compiler defects [7], [8], [10]. To support the above process, Csmith [10] is the most widely used program generation tool in the compiler testing area. However, Csmith only supports a subset of C language features which limits its capability at detecting compiler warning defects. Recently, program mutation becomes increasingly more important for validating compilers, which constructs test programs by modifying parts of existing seed programs, such as Orion [20], Athena [36], and Hermes [49]. However, all of them are not specific for compiler warning testing. Hence, Sun et al. [2] propose Epiphron, the state-of-the-art compiler warning detection approach, which can generate test programs with nearly all C language features. Epiphron first produces massive compilable test programs according to a set of randomly selected C language grammars. Then, Epiphron intentionally inserts warning-free bodies into conditional statements of the generated test programs, such as adding an empty statement ";" to if statements, and "break" into loop statements. The modified test programs are used to trigger and detect buggy compiler warnings. However, according to our preliminary analysis, we find that the defect-finding ability of Epiphron is still limited. First, merely inserting warning-free bodies into conditional statements may reduce the diversity of the modified test programs. Second, compiler defects are not only within a special statement (e.g., "if" and "for" statement); Epiphron may not well detect warning defects in other types of problematic statements.

- *Y. Tang, H. Jiang, Z. Zhou, Z. Ren, and W. Kong are with School of Software, Dalian University of Technology (DUT), Dalian, China, and Key Laboratory for Ubiquitous Network and Service Software of Liaoning Province. H. Jiang is also with DUT Artificial Intelligence Institute, Dalian, China. E-mail: tangyixuan@mail.dlut.edu.cn, jianghe@dlut.edu.cn (corresponding email), cszide@gmail.com, zren@dlut.edu.cn, and wqkong@dlut.edu.cn.*
- *X. Li is with the SnT Centre for Security, Reliability and Trust, University of Luxembourg, Luxembourg. E-mail: xiaochen.li@uni.lu*

Manuscript received August XX, 2020; revised April XX, 2021.

```c
// file = s.c
int main(){
  // incomplete loop condition in 'for' statement
  for(int i = 0;i < 2;){
  }
}
```

```
Clang 10 outputs:
s.c:4:18: warning: variable 'i' used in loop
    condition not modified in loop body [-Wfor-
    loop-analysis]
4 |  for(int i = 0;i <2;){
```

(a) GCC warning defect #92210

```c
// file = s.c
// missing parameters of function declaration
static int func_1();
int func_1(int a){
  return a;
}
```

```
Clang 10 outputs:
s.c:3:18: warning: this function declaration is
    not a prototype [-Wstrict-prototypes]
3 | static int func_1();
```

(b) GCC warning defect #92209

Fig. 1. Examples of GCC warning defects within questionable structures ($https://gcc.gnu.org/bugzilla/show_bug.cgi?id = 92210$ and $https://gcc.gnu.org/bugzilla/show_bug.cgi?id = 92209$).

To explain our observation, Fig. 1 shows two warning defects that we detected within the loop statement and the declaration statement. In the code snippet of Fig. 1(a), the loop conditional statement "i++" is discarded in Line 4, which makes the "for" statement non-terminally executed. Clang warns on this incomplete structure and provide a concrete warning message, whereas GCC considers the "for" statement as free of problems. In this scenario, if developers use GCC and neglect the problematic statement, the infinite loop in the program may exhaust the resources of CPU without outputting any expected results. In Fig. 1(b), the code snippet unintentionally misses a parameter type of the function declaration in Line 3. Clang warns on this situation with the correct location, while GCC incorrectly reports the location of this warning. If compilers provide wrong warning messages or even miss these traps and pitfalls of programs, they could output unexpected results and prevent developers from enhancing the quality of code at the compilation time. According to the principles of compiler warnings [2], the above problematic code snippets are referred as "warning-sensitive" structures. Therefore, it is crucial to construct test programs with diverse warning-sensitive structures to thoroughly test compiler warnings.

In this paper, we propose DIPROM (**DI**versity-guided **PRO**gram **M**utation), an effective technique to generate test programs with diverse warning-sensitive structures for compiler warning testing. First, given a seed test program, DIPROM removes the dead code regions based on the coverage information of the seed program to avoid false positives. Warning defects on dead code are often not regarded as the real compiler defects because it is a compiler vendor's design decision whether to warn on unreachable code [2]. We call the remained code in the seed as a live test program. Second, DIPROM generates program variants by iteratively employing 63 mutators (i.e., the pruning operation and the inserting operation) of the live test program. To generate diverse program variants, DIPROM employs Markov Chain Monte Carlo (MCMC) sampling to guide mutator selection. More specifically, during each iteration, DIPROM considers two capabilities of each mutator: the first capability is to diversify the newly generated program variant and the existing program variants; and the second capability is to generate compilable program variants. Based on the two capabilities, DIPROM calculates a priority score for each mutator, and utilizes the priority score as a discipline in the Metropolis choice to select each mutator. Since DIPROM can disrupt statements by the pruning operation and insert external statements by the inserting operation, the control- and data-flow among the program variants tend to be different. Thus, DIPROM produces a large number of diverse warning-sensitive structures in program variants. Last, these program variants are compiled by different compilers; the warning diagnostic messages emitted by compilers are obtained and aligned. Any inconsistent warnings among compilers could indicate a compiler warning defect.

We evaluate DIPROM over two popular C compilers (four versions in total), i.e., GCC [11] and Clang [12], under three testing scenarios (i.e., the cross-compiler scenario, the cross-version scenario, and the cross-optimization scenario). Evaluation results show that DIPROM performs better than the comparative approaches, i.e., HiCOND [25], Epiphron [2], and Hermes [49]. In total, DIPROM detects 76.74%, 34.30%, and 18.93% more warning defects on average than HiCOND, Epiphron, and Hermes, respectively. In addition, we examine the influence of mutation operations and the diversity-guided mutation strategy in DIPROM. We design three variants of DIPROM, i.e., DIPROM$_{prune}$ (i.e., DIPROM with only pruning operators), DIPROM$_{insert}$ (i.e., DIPROM with only inserting operators), and DIPROM$_{random}$ (i.e., DIPROM without guidance). During the given testing period, DIPROM outperforms its variants by detecting 9.46%∼28.21% more warning defects on average. Furthermore, we apply DIPROM to the latest development versions of GCC and Clang. Based on the analysis results of DIPROM, we reported 8 new warning defects in two months; 5 warning defects have been confirmed/fixed.

The major contributions of this paper are as follows:

- We present a novel approach, DIPROM, the first effort leveraging the mutation based approach for compiler warning testing. DIPROM designs 63 mutators for C test programs and employs the MCMC sampling to guide mutator selection, which help generate diverse warning-sensitive structures to examine compiler warning diagnostics.
- We conduct extensive experiments on GCC and Clang and show that, DIPROM is more effective than the comparative approaches in detecting warning defects.
- We apply DIPROM to the latest development versions of GCC and Clang. DIPROM helps developers detect 8 warning defects; 5 of them have been confirmed/fixed.

The remainder of this paper is organized as follows. Section 2 presents the background of compiler warnings. We demonstrate the framework in Section 3 and detail the implementation of our approach in Section 4. The experiential setup and the experiential results are shown in Section 5 and Section 6, respectively. In Section 7, threats to validity are discussed. We review the related work in Section 8. Finally, Section 9 concludes this paper.

2 Background

In this section, we introduce the background of compiler warnings, including the importance of compiler warnings, the principles of compiler warnings, the categories of compiler warning defects, and the compiler warning testing.

2.1 Importance of Compiler Warnings

Compiler warnings are important to both novice and experienced developers, since warning diagnostics often indicate potential problems in the programming code or even vulnerabilities in well constructed systems [2], [16]. Developers can leverage these warning diagnostics to detect bugs, especially in the early lifecycle of software [17]. Indeed, many large companies in different fields have been leveraging warning diagnostics to improve code quality over the years.

Code Maintenance. Software engineers in Google utilize compiler warnings to check incorrect dependencies in Google's Java codebase [13]. They enable two warning flags (i.e., "*-direct-dependency*" and "*-indirect-dependency*") to capture the dependencies in each JAR on the classpath. The unneeded dependencies marked by the compiler are subsequently removed by engineers, while the missing direct dependencies are added to prevent the code building failure. Doing so not only helps resolve conflicts when upgrading a subset of dependencies in the older version to a new version, but also decreases the incompatibilities for different functionalities and APIs.

Code Review. The Security Engineering team at Microsoft leverages compiler warnings to identify security vulnerabilities during code reviews [14]. They increase the level of compiler diagnostics and check each warning where the compiler warns. Their experience shows that several warnings are actually bugs, or even vulnerabilities in the program. Furthermore, several common vulnerabilities (e.g., buffer overflow, uninitialized variable, and missing parameter) could be easily issued by compiler warnings [15], [16].

2.2 Principles of Compiler Warnings

Compiler warns on problematic code fragments and provides warning diagnostic messages to discover potential programming mistakes. Typically, a warning contains the location information of the problematic code fragment, the reason of the warning, the warning diagnostic flag, and the specific problematic code in the code fragment. For example, in Fig. 1(a), Clang 10 outputs a warning message with the prefix "s.c:4:18", where "s.c" indicates the file name and "4:18" shows the warning location on Line 4 and Column 18. Subsequently, Clang reports a compiler warning message as: "variable '*i*' used in loop condition not modified in loop body". The "[-Wfor-loop-analysis]" is a warning diagnostic flag which warns on potential problems in the for-loop structure. The specific problematic code is presented below, corresponding to the location of the warning. Such a detailed warning message can help developers identify the problematic code fragment easily, especially in large software projects.

In general, the problematic code in which the compiler warns can be divided into two categories [2]. One is the warning-sensitive structures in the code (or bad code [2]), such as programming mistakes and code smells. Examples in Fig. 1 belong to this category because although the code is valid and executable, it is deprecated in the C standard and unsafe in practice. The other one is the undefined behavior in programming language standards, such as the null pointer dereferencing, the signed integer overflow, and the spatial memory safety violation [10]. These problematic codes induce compilers to check the syntactic and semantic information via static program analysis and output warnings on them. Therefore, they have a high probability of triggering compiler warning defects if compilers have poor warning checkers.

2.3 Categories of Compiler Warning Defects

Compilers could output buggy warning diagnostic messages for the problematic code. For example, GCC misses the warning diagnostic messages in Fig. 1(a) and reports the wrong location of the warning diagnostic messages in Fig. 1(b). In such situations, compiler warning defects are triggered. According to the root causes of buggy warning diagnostics, compiler warning defects can be classified into three categories, including erroneous warnings, spurious warnings, and missing warnings [2].

Erroneous warnings refer to those that contain confusing reasons of the warning or the incorrect location of the problematic code. Spurious warnings are the redundant warnings which are usually false positives on benign code. Missing warnings discard a potential bug in problematic code under two situations. One is that the compiler ignores the warnings although there are specific warning diagnostic flags to produce such warnings. This is because the corresponding warning diagnostic flag only considers limited warning examples and thus misses the diagnostics on other cases. In addition, a warning is missing when there are incompatible warning diagnostic flags among different compilers or different versions during upgradation. In this situation, the warning defect can be avoided by adding new warning diagnostic flags or expanding the warning scopes of existing warning diagnostic flags, which can be also regarded as an enhancement of compiler warning checkers.

2.4 Compiler Warning Testing

The recent effort on analysis of compiler warning diagnostics is Epiphron [2]. It takes the C language grammars as inputs, and then generates a large number of test programs based on the randomly selected grammars. Epiphron keeps the undefined behaviors in the test programs since they are likely to trigger compiler warnings as shown in Subsection 2.2. Besides, Epiphron randomly injects warning-free bodies in condition statements, such as an empty statement

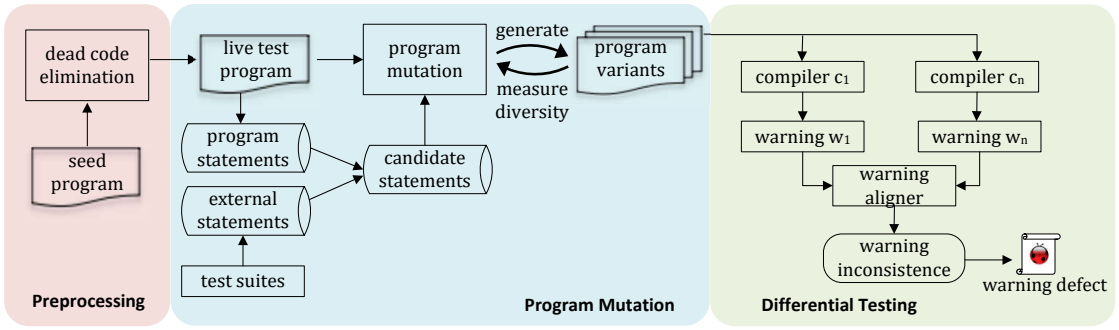

Fig. 2. Framework of our approach.

";" for "if" statements and "break" for loop statements to avoid false positives to some extent.

The generated test program is fed into different compilers (e.g., *c1* and *c2*) to obtain the warning diagnostics (e.g., *w1* and *w2*). The assumption of Epiphron is that the two compilers *c1* and *c2* are largely defect-free; ideally they should produce the same set of warnings (i.e., $w1 = w2$) for the same test program. Any detected inconsistent warnings between *c1* and *c2* is likely a compiler warning defect in either *c1* or *c2* (or both). Specifically, Epiphron leverages a specific warning parser and a warning aligner to analyze compiler warnings produced by each compiler under the compilation time. The warning parser is able to parse each warning diagnostic to a structured record by extracting the location (e.g., lines and columns of the warning), the warning description, and the warning types. The warning aligner determines whether the structured records are consistent between different compilers. If there are inconsistent records, the test program that triggers these inconsistencies is finally reduced and reported to compiler developers.

3 Framework

DIPROM targets generating test programs with diverse warning-sensitive structures for compiler warning testing. As depicted in Fig. 2, DIPROM mainly consists of three steps, including preprocessing, program mutation, and differential testing. Given a seed program, DIPROM first eliminates all the dead code in the program at the preprocessing step. The program without dead code is referred to the live test program. Then in the mutation step, DIPROM employs a diversity-guided mutation to mutate the live test program by pruning and inserting code snippets on it. Finally, based on the generated program variants, DIPROM employs differential testing to identify compiler warning defects.

3.1 Preprocessing

Given a seed test program, DIPROM eliminates all dead code in the program. Dead code usually refers to the unexecuted statements in a program when the program runs with the specific test input [20]. Hence DIPROM can remove the dead code entirely without changing this program's behavior. It is necessary to eliminate dead code in seed programs since warning defects on dead code regions are usually ignored by compilers' developers and it is the compiler vendor's design choice on whether to warn on that code [2]. By doing so, DIPROM can reduce false positives of warning defects in the following procedures. We eliminate the dead code according to the coverage information of the program. Code coverage tools could compute the frequencies that a program's statement executes at runtime when given some specific inputs. We can conveniently use such a tool to identify the executed (i.e., "live") code and the unexecuted (i.e., "dead") code. The dead statements are removed and only the live statements are kept in the test program to perform the following mutation. We refer such test programs to live test programs in the following subsections.

3.2 Program Mutation

To construct warning-sensitive structures in test programs, DIPROM performs two operations (i.e., pruning operation and inserting operation) to prune existing statements from and insert additional statements into the live test program. However, we cannot generate all program variants for a single live test program due to the huge mutation space. Therefore, DIPROM adopts a diversity-guided program mutation to generate diverse program variants for compiler warning testing.

3.2.1 Pruning Operation

We prune statements based on the AST of the live test program according to the predefined probability p^{prune}. That is, each node in the AST would be deleted with the probability p^{prune}. If a leaf node is selected to prune, we simply delete the leaf node; otherwise, if a parent node is to be deleted, we remove all nodes in its subtree, including all children nodes. However, not all AST nodes are proper to prune because syntactic errors may be injected when broking the structure of the AST. For example, if a declared variable is deleted, we will introduce an error (i.e., "use of undeclared identifier") in the following statements which have the "def-use" data dependency on that variable. Thus, in this study, we prevent pruning the AST nodes of identifiers that are declared, defined, or referenced in the test program. In addition, to ensure the pruned variant is syntactically correct, we compile the variant and examine whether an executable file is generated after compilation. If not, we discard this variant and restore the live test program for the next pruning until no errors are introduced.

We define 53 mutators for the pruning operation, allowing the seeds to be fully mutated via rewriting their

TABLE 1
Typical pruning mutators

Mutation type	Typical mutators	Example (Code before mutation →Code after mutation)	Warning diagnostic
Prune variable	Removing the attributes of a variable, e.g., qualifiers, modifiers, and types;	$static\ \ float\ \ a;$ $\rightarrow static\ a;$	warning: type specifier missing, defaults to "int" [-Wimplicit-int]
Prune operator	Removing a operator and its attributes, e.g, unary operators, binary operators, and ternary operators;	$int\ a = 1{>}0?\ 1\ : 0;$ $\rightarrow int\ a = 1{>}0?\ \ : 0;$	warning: ISO C forbids omitting the middle term of $a\ ?\ :$ expression [-Wpedantic]
Prune expression statement	Removing the expression and its attributes, e.g., assignments and arithmetic expressions;	$a = 1\ ;$ $\rightarrow\ ;$	warning: empty expression statement has no effect [-Wextra-semi-stmt]
Prune Control structure	Removing the structure and its attributes, e.g., the loop structure, the branch structure, and the jump structure;	$if\ (a)\ \ goto\ Label\ ;$ $\rightarrow if\ (a);$	warning: suggest braces around empty body in an "if" statement [-Wempty-body]
Prune function	Removing the function declaration and the whole function body;	$int\ func(void)\ ;$ $\rightarrow;$	warning: no previous prototype for function "func" [-Wmissing-prototypes]

syntactic structures. Specifically, we prune five types of structures in the program, including variables, operators, expression statements, control structures, and functions. Table 1 shows some typical mutators. For example, a mutator for "prune expression statement" can randomly remove the whole nodes of an expression statement in the AST. After pruning, DIPROM transforms the pruned AST into a test program which leaves an individual semicolon in the place of the pruned statement. This incomplete structure triggers a warning of LLVM: "empty expression statement has no effect [-Wextra-semi-stmt]".

Given a large seed program with various and abundant control- and data-dependencies, DIPROM is capable of deleting code in different structures on different granularities (e.g., a single statement and a block of code). Therefore, the pruned program variants may have different warning-sensitive structures, which are appropriate for testing compiler warnings.

3.2.2 Inserting Operation

For the inserting operation, we maintain a database of candidate statements and select suitable statements for insertion. We extract candidate statements from existing seed programs generated by Epiphron and the test suites extracted from the compiler under test. Specifically, given a corpus of programs, we traverse the ASTs of the corpus and extract candidate statements[1] at the statement level, the block level, and the function level. Meanwhile, we determine the context required in each extracted statement and construct the corresponding context table ⟨context, statement⟩. Specifically, we extract four types of required context for each piece of candidate statement, including 1) the variables and their types used in the statement but defined or declared outside; 2) the functions and their signatures (i.e., the number of parameters and return types) used in the statement but defined and declared outside; 3) the labels used in the statement but defined outside; and 4) the types of special statements including the "break" statement and the "continue" statement.

When performing the inserting operation, we only insert the candidate statements whose required context could be satisfied by the supplied context in the insertion point. This ensures the new program variant valid and compilable. However, different locations of insertion points are suitable for inserting different types of candidate statements. For example, the "if" statements can only be inserted in the function bodies, and the function definitions can only be added in the global scope of the seed test program. Therefore, we should search a suitable insertion scope for each type of candidate statements. Within the proper insertion scope, we locate the insertion point at the statement level of the AST based on a probability p^{insert}, and then perform a depth-first traversal on the AST. During this traversal, we keep a supplied context table that contains the same necessary information as the required context at the insertion point. With the supplied context table, we could select a compatible candidate statement from the database, in which the required context in the candidate statement is satisfied by the supplied context at the insertion point. Finally, we rename the required context in the candidate statement to the corresponding supplied context to make the inserted test program compilable. Regarding the statement renaming, we prioritize selecting local variables and functions in the supplied context table for references. Otherwise, we search the global context to find a suitable alternative.

Besides, we could insert candidate statements before or after the insertion point with the same probability. Since we perform the insertion operation at the statement level of the AST, it is easy to insert a candidate statement before the live statement. We only need to keep the candidate statements sharing the same parent as the live statement. However, when inserting code after a live statement, we should first determine whether the live statement has child statements.

1. We mainly consider 10 types of candidate statements in the C test programs, including the simple expression statement, the "if" statement, the "for" statement, the "while" statement, the "continue" statement, the "break" statement, the "switch" statement, the "goto" statement, the "return" statement, and the whole function.

TABLE 2
Typical inserting mutators

Mutation type	Typical mutators	Example (Code before mutation →Code after mutation)	Warning diagnostic
Insert expression statement	Inserting a single statement, e.g., an assignment and an arithmetic expression;	$int\ main()\{int\ a=1;...\}$ $\rightarrow int\ main()\{int\ a=1;$ $int\ b=(0!=((-1)\|a));...\}$	warning: bitwise comparison always evaluates to true [-Wtautological-compare]
Insert control structure	Inserting a type of control statement, e.g., the "for" statement and the "break" statement;	$int\ func()\{int\ a;...\}$ $\rightarrow int\ func()\{int\ a;...\ return\ a;\}$	warning: variable "a" is uninitialized when used here [-Wuninitialized]
Insert function	Inserting the function definition and the corresponding function call statement;	$int\ main()\{...unsigned\ int\ a;...\}$ $\rightarrow int\ func(int\ b)\{...return\ b;\}$ $int\ main()\{...unsigned\ int\ a;$ $a=func(0);...\}$	warning: implicit conversion changes signedness: "int" to "unsigned int" [-Wsign-conversion]

If so, we insert the candidate statements as child statements; otherwise, we treat the candidate statement as the sibling nodes of the live statement.

We design 10 mutators for the inserting operation. Table 2 presents some typical mutators. Overall, we insert three types of structures in the program, including expression statements, control structures, and function bodies. For example, when DIPROM inserts a return statement in the live test program and renames the variables in this statement to "a", the mutated program triggers a warning diagnostic, i.e., variable "a" is uninitialized when used here [-Wuninitialized]. Note that, DIPROM could insert a simple assignment statement, but also selecting complicated candidate statements from our database, such as the branch statement (e.g., "if"), the loop statement (e.g., "for"), and the whole function body. Inserting these statements into the seed program could generate warning-sensitive structures to some extent. Furthermore, the inserting operation could change the control- and data-dependencies of the live test program, making the generated program variants diverse and suitable for testing compiler warnings.

3.2.3 Diversity-guided Program Mutation

Due to the huge mutation space, we cannot generate all program variants for a test program. Intuitively, diverse warning-sensitive structures could aid in thoroughly testing compiler warnings. Hence, one of the most effective ways is to generate a program variant that differs from the existing ones as much as possible. In particular, we use the *program distance* to measure the difference between two programs.

DEFINITION 1. (Program Distance) The distance *Dist* between two test programs *P1* and *P2* is a function that measures the differences of their control-flow graphs (CFGs) and their statements. Specifically,

$$Dist(P1,P2)=GED(G1,G2)+d(P1,P2), \quad (1)$$

where *G1* and *G2* are the CFGs of the test programs *P1* and *P2*, respectively. *GED(·)* represents the graph edit distance [26] and *d(·)* represents the Jaccard distance [27] between the statements in *P1* and *P2*. *GED(·)* is defined as follows:

$$GED(G1,G2)=\min_{\lambda\in\gamma(G1,G2)}\sum_{e_i\in\lambda}c(e_i), \quad (2)$$

where e_i is the edit operation given by *deletions*, *insertions*, and *substitutions* on both nodes and edges in CFGs. $\gamma(G1,G2)$ denotes the set of all possible serialized edit operations that transform *G1* into *G2*. $c(e_i)$ is the cost function for the edit operation e_i, measuring the strength of the corresponding operation. Specifically, we set the same strength for each edit operation in this paper. In addition, *d(·)* is formulated as follows:

$$d(P1,P2)=1-\frac{Stmp(P1)\bigcap Stmp(P2)}{Stmp(P1)\bigcup Stmp(P2)}, \quad (3)$$

where *Stmp(P1)* and *Stmp(P2)* indicate each single line of the statements in *P1* and *P2*, respectively.

In Formula 1, we consider both CFGs and Jaccard distance, since the program distance could capture both a simple problematic structure in the variant that does not vary the control flows (such as pruning or inserting a simple straight-line statement), and sophisticated problematic structures that alter the control- and data-dependencies vastly. This helps generate diverse warning-sensitive structures which are likely to exercise the compiler warnings more thoroughly.

For DIPROM, given a seed test program, different mutators are not equally effective to construct diverse warning-sensitive structures. The mutators that are more frequently to generate valid and diverse program variants should be selected with higher probabilities for further mutations. Based on this insight, we propose our diversity-guided program mutation. That is, during the process of constructing program variants, DIPROM first selects a seed test program to mutate, and then selects a mutator to perform in each iteration.

Seed Program Selection. The initial seed test program is generated by Epiphron. The reason for selecting Epiphron-generated programs is that these programs support nearly all the language structures of the C language which are effective for testing compiler warnings than other program generators [2]. The program variants are derived from this initial test program. That is, under the given terminating condition (i.e., the number of generated program variants), the *i-th* variant is mutated on the initial seed test program and is independent of the *(i-1)-th* variant. The reason is that, if the *i-th* variant treats the *(i-1)-th* variant as the seed test

program, the same problematic structure with the *(i-1)-th* variant may be detected which results in duplicate warnings and a burden of the limited computational resources. Therefore, DIPROM constructs each program variant based on the initial seed test program.

Mutator Selection. Based on a seed test program, DIPROM selects mutators to generate program variants. However, different mutators have different capabilities in constructing diverse test programs; the same mutator also performs differently in constructing compilable test programs at different positions of the seed program. Therefore, we design an adaptive procedure to select mutators for generating a set of valid and diverse program variants. Specifically, we compute a priority score for each mutator. We rank the mutators by the priority scores and determine whether a mutator in the ranking list is worth accepting. The priority score for each mutator Mut is defined as follows:

$$Score(Mut) = (\frac{1}{n}\sum_{i=1}^{n} Dist(P, P_i)) * succ(Mut), \quad (4)$$

where n is the number of existing program variants for a seed test program. P is a new program variant generated by applying mutator Mut on the seed test program. *Dist(·)* is the program distance, computed by Formula 1. *succ(·)* is the success rate of generating compilable program variants by Mut, which is formulated as follows:

$$succ(Mut) = \frac{\#compilable_{Mut}}{\#all_{Mut}} * 100\%, \quad (5)$$

where $\#compilable_{Mut}$ is the number of compilable program variants generated by Mut under testing, and $\#all_{Mut}$ is the number of all program variants generated by Mut.

At the beginning of testing, we randomly select and accept a mutator, and then update the priority score of that mutator. Mutators are ranked in the descending order according to the priority scores. However, we do not directly select the mutator ranked at the first position for the next mutation, since the ranking is based on the historical results of these mutators which may not perfectly predict the further results. Ideally, each mutator should have some probabilities to be selected and the mutators ranked at higher positions should be more easily selected than those with lower positions. Thus, the task of mutator selection can be regarded as a problem of sampling from a probability distribution. In our context, the next mutator Mut_b selected for the *i-th* mutation only depends on the updated ranking list of priority scores of the current mutator Mut_a for the *(i-1)-th* mutation. It is a typical Markov Chain (MC). Therefore, to solve the sampling problem, DIPROM employs the Metropolis-Hastings (MH) algorithm, a popular Markov Chain Monte Carlo method. Given a proposal distribution of the samples, the MH algorithm randomly samples the next state (i.e., mutator) from the current state (i.e., mutator) according to the proposal distribution. During the process, an acceptance probability is introduced to determine whether to perform the state transition (i.e., move from the current state to the next state). Therefore, the MH algorithm finally generates a sequence of samples whose distribution closely approximates the proposal distribution. Here, following the prior work [36], [50], we set the proposal distribution to be the geometric distribution, which is the probability distribution of the number X of Bernoulli trials needed to get one success. If the probability of success on each trial is p, the probability that the *k-th* trial is the first success can be calculated as $Pr(X = k) = (1-p)^{k-1}p$.

During each mutation, the mutators are selected randomly, and therefore the proposal distribution is symmetric. Given a current mutator Mut_a, the acceptance probability of the next mutator Mut_b is computed as follows:

$$\begin{aligned} Pb(Mut_b|Mut_a) &= min(1, \frac{Pr(Mut_b)}{Pr(Mut_a)}) \\ &= min(1, (1-p)^{k_b - k_a}), \end{aligned} \quad (6)$$

where k_b and k_a are the positions of Mut_b and Mut_a in the ranking list. Note that, if Mut_b is ranked higher than Mut_a (i.e., $k_b < k_a$), Mut_b is always accepted; otherwise, Mut_b still has a certain probability $(1-p)^{k_b - k_a}$ to be accepted. If Mut_b is finally accepted, DIPROM updates its priority score and re-ranks these mutators for the next iteration. If not, DIPROM re-selects a mutator until it is accepted for the mutation.

Parameter Estimation. We set the success probability of each Bernoulli trial p by satisfying the following three conditions:

$$\begin{aligned} &0.95 \leqslant \sum_{k=1}^{63} Pr(X = k) \leqslant 1, \\ &p > \frac{1}{63}, \\ &\varepsilon < (1-p)^{63-1}p, \end{aligned} \quad (7)$$

where ε is a very small deviation (e.g., 0.001). The first condition ensures that the accumulative probability approaches 1. The second condition guarantees that the first mutator (having the highest priority score) should be selected with the probability larger than $\frac{1}{63}$ (63 is the number of mutators for pruning and inserting). The third condition ensures that the mutator with the lowest priority score still has some probabilities to be selected. Therefore, the initial value of p should be in the range of (0.0464,0.0651). In this study, we set p to be $\frac{4}{63} \approx 0.063$.

3.2.4 Overall Algorithm

We formally present DIPROM in Algorithm 1, Algorithm 2, and Algorithm 3. The function *GuideGen* in Algorithm 1 generates a set of program variants via diversity-guided mutation. Algorithm 2 performs the pruning operation via traversing the AST node with the function *Prune*. Algorithm 3 employs the inserting operation to generate program variants with the function *Insert*.

The inputs of Algorithm 1 are the seed programs, the number of generated program variants, and a set of mutators. Through a series of mutation operations, Algorithm 1 finally outputs a set of variants. In Algorithm 1, Lines 2-4 initialize several variables. Lines 5-7 eliminate the dead code in the seed program, and therefore construct a live program P without unexecuted statements in it. Line 8 randomly selects a mutator as the current one (i.e., Mut_a). Lines 9-29 conduct the diversity-guided mutation to generate the expected number of program variants. Line 10 gets the position of Mut_a in the ranking list of mutators. Lines 11-15 select the next mutator Mut_b. Based on the mutator Mut_b,

Algorithm 1: Diversity-guided Program Mutation

Input: SP, Epiphron-generated seed test program
N, number of program variants
Mut, a list of mutators $[Mut_i | i \in 1...63]$
Output: PV, a set of program variants

1 **Function** `GuidedGen` (*SeedProgram SP, VariantNum N, Mutators Mut*) *:ProgramVariant PV*:
2 $num \leftarrow 0$ /* the number of variants */
3 $score \leftarrow \varnothing$ /* the priority scores */
4 $PV \leftarrow \varnothing$ /* a list of variants */
/* compile a program without opt. */
5 $P_{exe} \leftarrow compile(SP, \text{“}-O0\text{”})$
/* collect the coverage statements */
6 $C \leftarrow coverage(P_{exe}.Execute())$
7 $P \leftarrow SP \cap C$ /* get the live test program */
8 $Mut_a \leftarrow Mut_{i \leftarrow random(1...63)}$
9 **while** $num \leqslant N$ **do**
10 $k_a \leftarrow rank(Mut_a.score)$
11 **do**
12 $Mut_b \leftarrow Mut_{i \leftarrow random(1...63)}$
13 $k_b \leftarrow rank(Mut_b.score)$
14 $f \leftarrow random(0, 1)$
15 **while** $f \geq (1-p)^{k_b - k_a}$
16 **if** Mut_b *is a pruning mutator* **then**
17 $P' \leftarrow$ `Prune` $(AST(P), Mut_b, False)$
18 **else**
19 $P' \leftarrow$ `Insert` $(AST(P), Mut_b, False)$
20 **if** P' *is compilable* **then**
21 $PV \leftarrow PV \cup P'$
22 $num \leftarrow num + 1$
23 $Mut_b.dist \leftarrow Avg(\sum_{i=0}^{n} dist(P', P_i))$
24 $Mut_b.succ \leftarrow succ(Mut_b)$
25 $Mut_b.score \leftarrow Mut_b.dist * Mut_b.succ$
26 $Mut_a \leftarrow Mut_b$
27 **else**
28 $Mut_b.succ \leftarrow succ(Mut_b)$
29 $Mut_b.score \leftarrow Mut_b.dist * Mut_b.succ$
30 return PV

Algorithm 2: Pruning operation

Input: ast, AST of a seed test program
Mut, a pruning mutator
$flagPro$, a global variable initialized to False
Output: P', a program variant

1 **Function** `Prune` (*AST ast, Mutator Mut, Global flagPro*) *:ProgramVariant P'*:
2 **if** *flagPro* **then**
3 return
4 $node \leftarrow traversal(ast, Mut)$
5 **if** *FlipCoin(node)* **then**
6 $ast' \leftarrow delete(node, ast)$
7 $P' \leftarrow transform(ast')$
8 $flagPro \leftarrow True$
9 return P'
10 **else**
11 **foreach** $node' \in node.Children()$ **do**
12 `Prune` (*node', Mut, flagPro*)

Algorithm 3: Inserting operation

Input: ast, AST of a seed test program
Mut, a inserting mutator
$flagPro$, a global variable initialized to False
Output: P', a program variant

1 **Function** `Insert` (*AST ast, Mutator Mut, Global flagPro*) *:ProgramVariant P'*:
2 **if** *flagPro* **then**
3 return
4 $node \leftarrow traversal(ast, Mut)$
5 **if** *FlipCoin(node)* **then**
6 $supplied \leftarrow extract(context, node)$
7 **do**
8 $num \leftarrow size(sample_{Mut})$
9 $sample \leftarrow sample_{i \leftarrow random(1...num)}$
10 $required \leftarrow extract(context, AST(sample))$
11 **while** $required \in supplied$
12 $sample' \leftarrow sample.Rename(supplied, required)$
13 $P' \leftarrow insert(P, transform(node), sample')$
14 $flagPro \leftarrow True$
15 return P'
16 **else**
17 **foreach** $node' \in node.Children()$ **do**
18 `Insert` (*node', Mut, flagPro*)

Line 16 determines whether it is a pruning mutator. If Mut_b is a pruning mutator, Line 17 selects the function *Prune* to generate a program variant P' by pruning statements on the AST of P; otherwise, Line 19 employs the function *Insert* to generate the program variant P'. If the variant P' is compilable, Lines 21-26 accept the variant and update the priority score of mutator Mut_b. Specifically, Line 21 adds the variant into the set of program variant PV; Line 22 updates the current number of accepted program variants; Line 23 updates the capability of Mut_b (i.e., $Mut_b.dist$) on generating diverse program variants by calculating the average program distance between P' and the existing program variants in PV according to Formula 1; Line 24 updates the success rate of Mut_b (i.e., $Mut_b.succ$) according to Formula 5. Based on the $Mut_b.dist$ and the $Mut_b.succ$, we calculate the current priority score (i.e., $Mut_b.score$) of Mut_b in Line 25 according to Formula 4, and take Mut_b as the current mutator for the next iteration in Line 26. If P' has not yet been adopted in PV, the priority score of Mut_b is updated by just changing its success rate in Lines 27-29.

In function *Prune* of Algorithm 2, since we only need to generate one program variant by pruning a node and its children nodes, Line 2 leverages a global variable *flagPro* to determine whether a program variant is generated during the pruning operation. If *flagPro* is *True*, Line 3 returns NULL to finish the pruning process. Otherwise, Line 4 performs a depth-first traversal on the AST to select a pruned node according to the mutator. Then, Line 5 uses a function *FlipCoin* to stochastically decide whether the selected node should be kept or removed. If *FlipCoin* returns *True*, we prune the node and its children in Line 6. Next, we transform the pruned AST into the corresponding program variant in Line 7 and set the variable *flagPro* to *True* in Line 8. Line 9 returns the generated program variant. For the case that *FlipCoin* returns *False*, Lines 11-12 traverse the children nodes and recursively invoke function *Prune* to process each children node.

Algorithm 3 conducts the inserting operation to generate a program variant. Similar to Algorithm 2, a global variable

flagPro is used in Line 2 to determine whether a program variant is generated during the inserting process. If yes, Line 3 returns NULL to finish the inserting operation. Line 4 selects an insertion point via a depth-first traversal on the AST according to the mutator. A function *FlipCoin* is employed to determine whether the selected point should be mutated in Line 5. If *FlipCoin* returns *True*, Line 6 extracts the supplied context information at the insertion point. Lines 7-11 sample a suitable candidate statement in which the required context in the candidate statement is satisfied by the supplied context at the insertion point. Specifically, Line 8 determines the number of candidate statements for a given inserting mutator; Line 9 randomly samples a candidate statement, and Line 10 extracts the required context of the sample. Lines 12-13 rename the sample and insert the sample into the live test program. Line 14 set the variable *flagPro* to *True* and the generated program variant is returned in Line 15. If *FlipCoin* returns *False*, Lines 17-18 traverse the children nodes and recursively invoke function *Insert* to perform the mutation process.

3.3 Differential Testing

In this study, we employ differential testing [9] to detect compiler warning defects. Differential testing is a widely used software testing technology that has been utilized in many studies for detecting compiler bugs [2], [6], [10], [42]. It assumes that the comparable compilers are implemented based on the same specification and should output the same warning diagnostics for the same test program. When compilers under test produce different diagnostics, at least one implementation contains warning defects. During this process, the warning diagnostics produced by multiple compilers are obtained and aligned.

To conveniently align the warning diagnostics, similar to prior work on compiler warning testing [2], we design a message parser to split the diagnostics into a list of pairs for aligning the diagnostics. More specifically, the message parser examines whether a warning diagnostic is parsable and extracts corresponding warning information, including the location, the warning description, and the type of the warning. All the extracted warning information is assigned to a set w. Given two sets of extracted warning information *w1* and *w2* from two compilers, for each warning record *record1* in *w1*, we search whether there is a similar warning record *record2* in *w2*, and then divide them into three categories:

- **Equivalent warning.** Both *record1* in *w1* and *record2* in *w2* have the same location and the same type of the warning.
- **Unmatched location.** Both *record1* and *record2* have the same type of the warning and are on the same line, whereas the columns of the location for *record1* and *record2* are unmatched.
- **Missing warning.** For *record1* in *w1*, there is no waning in *w2* that has the same warning type and the same location as *record1*, and vice versa.

For the inconsistent pairs in the latter two categories, we check whether the identified warning defect is a real compiler warning defect. Test programs that trigger real warning defects are finally reduced by reduction tools, such as C-Reduce [33] and Delta [34], [35]. The reduced test program can be reported to compiler developers.

4 Implementation

In this section, we highlight the details and the design choice in implementing DIPROM for C compiler warning testing[2].

4.1 Coverage Collection

In the preprocessing, we eliminate the dead code regions leveraging the coverage information of an Epiphron-generated test program. Particularly, we employ Gcov[3], a widely used utility in the GNU Compiler Collection, to extract coverage information for the test program. It profiles coverage at the line level and generates a coverage file labeling how many times a line of code has been executed [21]. In the coverage file, an unexecuted statement is labeled as "#####" at the beginning of the statement. Notably, Gcov can also present ambiguous coverage information occasionally. For example, in the "for" statements of a program, Gcov marks the sub-statements inline the body of "for" as executed while incorrectly marks the circulation condition as unexecuted. To avoid this problem, we follow the prior work [20] to remedy the ambiguity by checking whether some of the sub-statements of an unexecuted statement are executed. If so, we re-label the executed times of that statement as the same frequencies as the sub-statements. Since the Epiphron-generated test program takes no input at the running time, we check and re-label the coverage information, and remove all the unexecuted statements; only the live statements are kept in the test program to perform mutations.

4.2 Extracting Candidate Statements

Candidate statements are used to insert into the live test programs. We extract them by traversing the AST of the programs. To correctly apply the candidate statements, we should extract the required context of each candidate statement and compare the required context with the supplied context at the insertion point. Besides, we remove the duplicate candidate statements to avoid unnecessary comparison.

4.2.1 Candidate Statements Construction

We construct a database of candidate statements by extracting codes from a given program corpus. In practice, we use pycparser[4], a parser for C programs written in Python, to visit the AST of statements at the statement level, the block level, and the function level. We do not extract code at other levels of AST, such as the literal level, since these codes may usually lead to syntactic errors and are difficult to seek suitable insertion points. Nevertheless, the candidate statements in our database have different complexities, ranging from a single statement to an entire function body. We then determine and extract four types of context required in each extracted statement following the prior work [36]:

2. The materials of DIPROM are publicly available at https://github.com/tangyixuan01/DIPROM
3. http://ltp.sourceforge.net/coverage/gcov.php
4. https://pypi.org/project/pycparser/

$$
\begin{aligned}
a ::= & x \mid n \mid op_u\ a \mid a_1\ op_a\ a_2 \\
b ::= & \textbf{true} \mid \textbf{false} \mid \textbf{not}\ b \mid \\
& b_1\ op_l\ b_2 \mid a_1\ op_r\ a_2 \\
S ::= & x := a \mid S_1; S_2 \mid \\
& \textbf{while}(b)\ \textbf{do}\ S \mid \\
& \textbf{if}(b)\ \textbf{then}\ S_1\ \textbf{else}\ S_2
\end{aligned}
$$

Fig. 3. Syntax rules for the WHILE language.

- *The variables and their types used in the statement but defined or declared outside.* We exclude the variables defined and used within the statement because it is not necessary to rename these variables when performing insertion. Note that, if a variable is associated with a user-defined type, we explore the typedef and the macros statement to replace the user-defined type with a real type.
- *The functions and their signatures (i.e., the number of parameters and return types) used in the statement but defined and declared outside.*
- *The label used in the statement but defined outside.* Labels are usually appeared in the "goto" statement referring to a chunk of code snippets. If the extracted statement uses these labels, we have to rename the labels to satisfy the local context of the insertion point.
- *The types of special statements including the "break" statement and the "continue" statement.* The "break" statement can be integrated into either "switch" statements or loop statements; the "continue" statement can be only injected into loop statements.

In practice, we extract all the variables, functions, and labels in a piece of statement and exclude those that are already defined in the extracted statements. We individually extract special statements and mark them with the corresponding types.

4.2.2 Duplicate Candidate Statements Elimination

We eliminate the duplicates to ensure the diversity of candidate statements in our database. For each piece of statement, we compare it with the existing statements in the database to avoid duplicates. Following the presentation of prior work [37], we view a candidate statement as a syntactic skeletal structure $\mathbb{S}$ with placeholders for variables (denoted as hole $\square_v$), function references (denoted as hole $\square_f$), and constants (denoted as hole $\square_c$). In particularly, let us consider a WHILE-style language which has been widely used in prior program-analysis research [37], [38]. Fig. 3 shows the program syntax rules for the WHILE-style language. In the rules, the non-terminals a, b, and S denote arithmetic expressions, boolean expressions, and program statements, respectively. Note that, we use the WHILE-style language for the ease of presentation, and our technique applies to the full C program language.

To obtain a candidate statement with holes, we recursively employ a hole transformation $[\![]\!]$ to the WHILE-style language. Fig. 4 presents the transformed grammars for variables, function references, and constants. For each WHILE-style candidate statement S, we say $\mathbb{S}$ is a skeleton of S iff the abstract syntax tree (denoted as $T_{\mathbb{S}}$) of $\mathbb{S}$ is the same as the transformed abstract syntax tree (denoted as $[\![T_S]\!]$) of S, i.e., $T_{\mathbb{S}} = [\![T_S]\!]$. Holes $\square_v$, $\square_f$, and $\square_c$ are associated with the variable set (i.e., $\mathbb{V}$), the function reference set (i.e., $\mathbb{F}$), and the constant set (i.e., $\mathbb{C}$) which are extracted from the candidate statements. Therefore, replacing holes (e.g., $\square_v$) in $\mathbb{S}$ with the elements (e.g., $v \in \mathbb{V}$) in the correspond set could generate a WHILE-style statement S'. We say $v \in \mathbb{V}$ fills $\square_v$, and S' realizes $\mathbb{S}$. A skeleton $\mathbb{S}$ with n holes could be denoted as a characteristics vector $s_{\mathbb{S}} = \langle \square_1, \square_2, ..., \square_n \rangle$. Thus, a statement S' that realizes $\mathbb{S}$ could be represented as a vector $s_{S'} = \langle v_i, f_i, c_i \rangle$, where $v_i \in \mathbb{V}$, $f_i \in \mathbb{F}$, and $c_i \in \mathbb{C}$. Let us consider a piece of "while" statement in Fig. 5. Fig. 5(a) and Fig. 5(b) show two original "while" statements, i.e., *S1* and *S2*. Fig. 5(c) and Fig. 5(d) are skeletons of *S1* and *S2*, respectively. We can observe that the two statements have the same skeletons. Let s be the set of all elements that sequentially fill the holes in the skeleton of a piece of statement. Thus, in Fig. 5, we have $s_{S1} = \langle a, b, b, a, func\_1, b \rangle$, where $\mathbb{V}_{S1} = \langle a, b \rangle$ and $\mathbb{F}_{S1} = \langle func\_1 \rangle$. Similarly, we have $s_{S2} = \langle c, d, d, d, func\_2, c \rangle$, where $\mathbb{V}_{S2} = \langle c, d \rangle$ and $\mathbb{F}_{S2} = \langle func\_2 \rangle$. The statement *S1* can be transformed to *S2* by replacing all the occurrences of elements in $\mathbb{V}_{S1}$ and $\mathbb{F}_{S1}$ with the elements in $\mathbb{V}_{S2}$ and $\mathbb{F}_{S2}$, respectively. Thus, we have the renaming pairs $R = \{\langle a, c \rangle, \langle b, d \rangle, \langle a, d \rangle, \langle func\_1, func\_2 \rangle, \langle b, c \rangle\}$ that can transform *S1* to *S2*, and vice versa. Consequently, we consider *S1* and *S2* as equivalent statements. Particularly, two elements in a renaming pair should have the same property. For example, in Fig. 5, the elements in a renaming pair $\langle a, c \rangle$ are both variables with the same types; the elements in the pair $\langle$*func_1, func_2*$\rangle$ are both function identifiers having the same return type and the same number of parameters. However, the matching conditions in a renaming pair could also be relaxed when the types of variables are compatible (e.g., variable type of Integer and Float). Similar to the previous work [37], we define the statement equivalence as below:

***DEFINITION* 2.** (Statement Equivalence) Two statements *S1* and *S2* are equivalent, iff:
(1) Both *S1* and *S2* share the same skeletons;
(2) There exist renaming pairs that allow *S1* to convert to *S2*, and vice versa.

Equivalent statements exhibit the same control- and data-dependencies information; there is no need to insert all of them into the database. According to Definition 2, we reject statements that are equivalent to the existing one in our database. This increases the diversity of program variants when performing the inserting operation.

4.2.3 Sources of Candidate Statements

We select two sources for extracting candidate statements, including the Epiphron-generated test programs and the regression test suites.

The Epiphron-generated test programs are a suitable source as they can generate hundreds of statements containing different control- and data-flow. Moreover, the variables, functions, labels, and user-defined types have the same naming conventions, which can simplify the renaming process during the inserting operation. Practically, we collect 10,000 Epiphron-generated programs and extract different candidate statements from them. The candidate statements range from a single line to a chunk of code. After the

$[\![a]\!]_v ::= \square_v \mid n \mid op_u\ [\![a]\!]_v \mid [\![a_1]\!]_v\ op_a\ [\![a_2]\!]_v$
$[\![b]\!]_v ::= \textbf{true} \mid \textbf{false} \mid \textbf{not}\ b \mid [\![b_1]\!]_v\ op_l\ [\![b_2]\!]_v \mid [\![a_1]\!]_v\ op_r\ [\![a_2]\!]_v$
$[\![S]\!]_v ::= \square_v := [\![a]\!]_v \mid [\![S_1]\!]_v; [\![S_2]\!]_v \mid \textbf{while}([\![b]\!]_v)\ \textbf{do}\ [\![S]\!]_v \mid \textbf{if}([\![b]\!]_v)\ \textbf{then}\ [\![S_1]\!]_v\ \textbf{else}\ [\![S_2]\!]_v$

(a) Syntax rules for variables

$[\![a]\!]_f ::= \square_f \mid n \mid op_u\ [\![a]\!]_f \mid [\![a_1]\!]_f\ op_a\ [\![a_2]\!]_f$
$[\![b]\!]_f ::= \textbf{true} \mid \textbf{false} \mid \textbf{not}\ b \mid [\![b_1]\!]_f\ op_l\ [\![b_2]\!]_f \mid [\![a_1]\!]_f\ op_r\ [\![a_2]\!]_f$
$[\![S]\!]_f ::= \square_f := [\![a]\!]_f \mid [\![S_1]\!]_f; [\![S_2]\!]_f \mid \textbf{while}([\![b]\!]_f)\ \textbf{do}\ [\![S]\!]_f \mid \textbf{if}([\![b]\!]_f)\ \textbf{then}\ [\![S_1]\!]_f\ \textbf{else}\ [\![S_2]\!]_f$

(b) Syntax rules for function references

$[\![a]\!]_c ::= x \mid \square_c \mid op_u\ [\![a]\!]_c \mid [\![a_1]\!]_c\ op_a\ [\![a_2]\!]_c$
$[\![b]\!]_c ::= \textbf{true} \mid \textbf{false} \mid \textbf{not}\ b \mid [\![b_1]\!]_c\ op_l\ [\![b_2]\!]_c \mid [\![a_1]\!]_c\ op_r\ [\![a_2]\!]_c$
$[\![S]\!]_c ::= x := [\![a]\!]_c \mid [\![S_1]\!]_c; [\![S_2]\!]_c \mid \textbf{while}([\![b]\!]_c)\ \textbf{do}\ [\![S]\!]_c \mid \textbf{if}([\![b]\!]_c)\ \textbf{then}\ [\![S_1]\!]_c\ \textbf{else}\ [\![S_2]\!]_c$

(c) Syntax rules for constants

Fig. 4. Skeletal program structures for the WHILE-style language

```
// required context
a:int
b:int
func_1:int,1
// statement
while(a > b)
b = a + func_1(b)
```

(a) Statement S1

```
// required context
c:int
d:int
func_2:int,1
// statement
while(c > d)
d = d + func_2(c)
```

(b) Statement S2

```
while(□v > □v)
  □v = □v + □f(□v)
```

(c) Skeleton of S1

```
while(□v > □v)
  □v = □v + □f(□v)
```

(d) Skeleton of S2

Fig. 5. Example of program skeleton.

context extraction and duplicate statement elimination, our database contains 25,200 pieces of candidate statements.

Compilers (e.g., GCC and Clang) have their own regression test suites maintained along with the development. Thus, the regression test suites are also a convenient source for testing compilers. However, several regression programs are expected to trigger compiler bugs like crashes or other code errors. We exclude programs that are not parsable and crash compilers. Finally, we collect respectively 3,015 and 2,007 C programs from GCC 4.8 and Clang 3.9 test suite when evaluating the corresponding compilers in our experiments. Note that, if a test program triggers a regression compiler bug, the test program is usually added to the test suites in the next releases of the compiler. We find that the regression test suites are helpful in generating various warning-sensitive structures in program variants though the number of candidate statements derived from the test suites is tiny and limited. Consequently, there are more than 30,000 pieces of candidate statements in our database.

DIPROM is also capable of inserting real-world code into programs. However, the real-world projects usually involve multiple files and the structures could depend on other structures in different files. Accordingly, it is a challenge to parse the real-world projects and extract the required context of the code. Therefore, we only select the seed test programs and the regression test suites as the sources of candidate statements in this paper.

4.3 Statement Renaming

Since most candidate statements have the same naming conversion as the live programs, it is easy to select and rename a candidate statement that is compatible with the supplied context of the live program at the insertion point. We prioritize renaming variables in the candidate statement to those in the local context with the same type. If there are many suitable variables in the local context, we randomly select one and rename all occurrences of a variable to the selected one. If not, we search the global context to find an alternative. Similarly, we can rename the required function or label in the candidate statements to a proper one that satisfies the context of the insertion point. If there is no alternative in the context of the live program, we re-select a new candidate statement from our database until the statement can be correctly inserted into the live program.

5 Experimental Setup

In this section, we detail the settings for evaluating DIPROM, including Research Questions (RQs), hardware and compilers, comparative approaches, testing scenarios, evaluation metrics, and experimental processes.

5.1 Research Questions

We address the following three research questions (RQs):

RQ1: How does DIPROM perform against the state-of-the-art approaches in compiler warning testing?

We select three existing approaches (i.e., HiCOND, Epiphron, and Hermes) to show the effectiveness of DIPROM on detecting warning defects.

RQ2: How do the two mutation operations and the diversity-guided mutation strategy influence the performance of DIPROM on detecting warning defects?

We have implemented three variants of DIPROM (i.e., DIPROM$_{prune}$, DIPROM$_{insert}$, and DIPROM$_{random}$) to investigate the influences of two mutation operations and the diversity-guided mutation strategy on detecting warning defects.

RQ3: Can DIPROM detect warning defects on the latest development versions of compilers?

To investigate the effectiveness of DIPROM in practice, we have applied DIPROM on the latest development versions of GCC and Clang to detect new compiler warning defects.

5.2 Hardware and Compiler

Our experiments are conducted on four PCs with Intel(R) Core(TM) i7-4790 CPU @3.60GHz running Ubuntu 14.04 operating systems. In our study, we use GCC and Clang as subjects, which are two popular and widely used C compilers studied in the existing work [6], [10], [39], [40], [41]. More specifically, for RQ1 and RQ2, we test two versions of GCC and two versions of Clang, i.e., GCC-4.8.5, GCC-7.1.0, Clang-3.9.0, and Clang-7.0.0. These evaluated compilers include both old release versions and recent release versions of compilers. The reason is that the older releases

usually contain more compiler warning defects and give more statistically significant results, while the recent releases could check whether our approach still works well. For RQ3, we conduct experiments on the latest development versions of GCC and Clang[5]. This is because compiler developers usually enhance new compiler warning diagnostics and prefer to fix bugs in the development trunk than in stable versions, thus encourages us to investigate whether DIPROM can detect warning defects on new releases.

5.3 Comparative Approach

To evaluate the effectiveness of DIPROM in RQ1, we compare DIPROM with three comparative approaches, i.e., HiCOND [25], Epiphron [2], and Hermes [49]. HiCOND could control the test program generator through a test configuration, and then generates both bug-revealing and diverse test programs [25]. Since not every program generator has features relevant to the options of test configurations, we adopt the test configurations [6] provided by the authors of HiCOND for a widely used C program generator Csmith [10]. Epiphron is a state-of-the-art tool for compiler warning testing. Epiphron is able to generate C test programs with undefined behaviors by unrolling the grammar of the C language [2]. Hermes is a mutation based approach, which generates a large number of semantically equivalent program variants by inserting extra code into the existing seed programs [49]. We select Hermes as a comparative approach since both Hermes and DIPROM apply mutation strategies on the live code regions to generate test programs.

To investigate whether the mutation operations and the diversity-guided strategy contribute to DIPROM in RQ2, we compare DIPROM with its three variants, i.e., DIPROM$_{prune}$, DIPROM$_{insert}$, and DIPROM$_{random}$. DIPROM$_{prune}$ employs the pruning operators to generate program variants by pruning statements from test programs; DIPROM$_{insert}$ only utilizes the inserting operators to generate program variants; DIPROM$_{random}$ generates program variants by randomly employing the mutation operators without the guidance of priority scores. Notably, we employ the same seeds and generate the same number of variants under the same terminating condition for fair comparisons.

5.4 Testing Scenario

We consider three widely used testing scenarios, i.e., cross-compiler scenario (CCS), cross-version scenario (CVS), and cross-optimization scenario (COS) [2], [42].

Cross-Compiler Scenario (CCS) discovers compiler warning defects by comparing the warning diagnostics emitted by independently developed compilers. GCC and Clang are two reference compilers here. Both of them have been developed for years by individual development communities. In practice, for RQ1 and RQ2, we test GCC-4.8.5 and GCC-7.1.0 using Clang-3.6.0 and Clang-4.0.0 as references, respectively. We select these reference compilers because the release time of them are close to GCC-4.8.5 and GCC-7.1.0, which are suitable for differentially testing the compilers under test. Similarly, we test Clang-3.9.0 using GCC-4.8.5 as a reference compiler and test Clang-7.0.0 by referring to GCC-8.1.0. For RQ3, we use GCC-10.0.0 to test Clang-10.0.0, and vice versa.

Cross-Version Scenario (CVS) exposes compiler warning defects by comparing the warning diagnostics emitted from different versions of a single compiler. In this strategy, for RQ1 and RQ2, we test GCC-4.8.5 and GCC-7.1.0 using GCC-4.7.0 and GCC-6.1.0 as reference compilers. Similarly, we use Clang-3.6.0 and Clang-6.0.0 as reference compilers to test Clang-3.9.0 and Clang-7.0.0, respectively. For RQ3, we use GCC-8.1.0 and Clang-9.0.0 as reference compilers to test GCC-10.0.0 and Clang-10.0.0, respectively.

Cross-Optimization Scenario (COS) detects compiler warning defects by comparing the warning diagnostics emitted from different optimization levels of a compiler. For example, we can use GCC to compile a given program without optimization (i.e., -O0) and compile the program with an implemented optimization (i.e., -O1, -O2, -Os, or -O3). In this study, we test GCC and Clang under a level of optimization and use other levels as references.

CCS examines warning diagnostic flags between different compilers and thus targets a large scope of warning defects. CVS focuses on extended warning diagnostic flags as the compiler upgrades, and COS aims at warning defects across different optimization levels of a single compiler. Each of them has a unique ability in finding compiler warnings defects. Indeed, all of them detect compiler warning defects which are shown in Subsection 6.1 and Subsection 6.2.

5.5 Experimental Process

In RQ1, we run DIPROM and three comparative approaches (i.e.,HiCOND, Epiphron, and Hermes) 10 times for each testing scenario (i.e., CCS, CVS, and COS); this setting is also widely used in the evaluation of fuzzing testing [22], [23], [24]. Notably, we install four compilers (i.e., two versions of GCC and two versions of Clang) under test in different

Algorithm 4: Compiler Warning Detection

Input: SP, Epiphron-generated seed test program
N, number of program variants
Mut, a list of mutators $[Mut_i | i \in 1...63]$
Output: WD, a set of compiler warning defects

```
Function WarnDetect (SeedProgram SP, VariantNum N,
  Mutators Mut) :WarningDefect WD:
    WD ← ∅
    while not termination do
        PV ← GuidedGen(SP, N, Mut)
        foreach PV' ∈ PV do
            w ← compile(compiler under test, PV')
            wlist ← compile(reference compilers, PV')
            (record, recordlist) ← parsers(w, wlist)
            foreach record' ∈ recordlist do
                (iw, iw') ← aligner(record, record')
                if (iw,iw') is a real warning defect then
                    WD ← WD ∪ (iw, iw')
    return WD
```

5. In practice, we mainly test four versions of the latest development versions of GCC and Clang, including GCC-10.0.0-20190913, GCC-10.0.0-20191020, GCC-10.0.0-20191110, and Clang-10.0.0.

6. https://github.com/JunjieChen/HiCOND

PCs and thus each compiler is individually evaluated. For each run, we evaluate them with a testing period of 72 CPU hours. Besides, since we need to conduct 480 experiments (DIPROM and three comparative approaches, 10 runs for each compiler under three testing scenarios), we run many experiments simultaneously in each PC so that we can finish all experiments in nearly 60 days. For each experiment, we collect the warning diagnostics emitted by the compiler under test and the reference compilers when giving the same test program as input. The program generator used in HiCOND is Csmith as in the original paper of HiCOND [25]. We use the same Epiphron-generated programs as the seed test programs in Hermes and DIPROM. In addition, the number of generated program variants for each seed in Hermes and HiCOND is identical (i.e., 8 in our experiments) for fair comparisons. During the testing, we record each warning defect and the time for detecting it. In addition, we also record the number of valid test programs generated by each approach for the further analysis.

In RQ2, we apply DIPROM$_{prune}$, DIPROM$_{insert}$, and DIPROM$_{random}$ to test the same compilers under the same 72h testing period. Therefore, we need to conduct 360 experiments (three comparative approaches, 10 runs for each compiler under three testing scenarios). To effectively utilize the CPU computation resources, we also run many experiments simultaneously in each PC so that we could finish the experiments in nearly 45 days. During each experiment, DIPROM$_{prune}$, DIPROM$_{insert}$, and DIPROM$_{random}$ are configured to generate the same number of program variants (i.e., 8 variants in our experiments) for the same Epiphron-generated seed programs that are used for DIPROM. During the test process, we also record the number of valid test programs generated by each approach, each warning defect, and the time for detecting each warning defect.

Finally, we run DIPROM for two months to the latest development versions of GCC and Clang to evaluate the effectiveness of DIPROM in RQ3. For each warning defect detected by DIPROM, we check whether it is a new warning defect that has never been discovered by developers. Subsequently, we reduce the new warning defect by C-reduce and submit a bug report to compiler developers.

We formally present the process of warning defect detection in Algorithm 4. Generally, the function *WarnDetect* consists of five steps. First, we use DIPROM to generate valid program variants as the input of the compiler under test and the reference compiler in Line 4. Second, for each program variant, we collect a series of warning diagnostics during the compilation of each compiler in Lines 5-6. Third, a warning message parser is used to split the warning diagnostics to a list of structured records in Line 8. The aligner is then employed to identify the inconsistent warning diagnostics in Line 10. Finally, we check whether the inconsistent warning diagnostic is a warning defect in Lines 11-12. All experimental procedures are implemented in python and shell script.

5.6 Evaluation Metric

We use two metrics to validate DIPROM and the comparative approaches in RQ1 and RQ2.

The first metric is the number of warning defects detected by each approach under different testing scenarios. For each testing scenario, all approaches (i.e., DIPROM and the comparative approaches) are individually employed to detect compiler warning defects within a given testing period. In our study, we check whether a detected warning defect is a real warning defect by two steps, namely manual analysis and automatically finding correcting versions. Specifically, we first search bug reports about warning defects from GCC Bugzilla[7] and LLVM Bugzilla[8] with the keyword "diagnostics". Notably, we exclude the bug reports with the status "unconfirmed" for GCC and the status "new" for LLVM since these bug reports have not been confirmed by GCC and LLVM developers. We analyze the behaviors of the warning defect which are described in the bug reports. The behaviors of each warning defect include the warning diagnostic flag, the warning message, and the reason for the warning defect. We consider a warning inconsistency defected by DIPROM and the comparative approaches as a real warning defect if it has the same behavior as the historical warning defect described in the confirmed bug report. Otherwise, following the idea of the Correcting Commits [6], we automatically check whether the detected warning defect is fixed by a subsequent version of the compiler under test. This is because compilers are also tested by compiler developers themselves, and the warning defects may be fixed by them without submitting bug reports. If one subsequent commit does not trigger the same warning inconsistency given the same test program as input, we consider the warning inconsistency as a compiler defect, and the corresponding commit fixes this warning defect. Specifically, for two warning defects, if they have the same warning diagnostic flags and the same type of warning defect (i.e., the erroneous warning, the spurious warning, and the missing warning), we treat them as duplicates.

The second metric is the detection time spent on discovering each warning defect. In our experiment, the detection time mainly includes the time for seed program generation, the mutation time, the execution time, and the alignment time. The time for seed program generation refers to the time required by Epiphron to generate seed test programs. Mutation time refers to the time spent on generating program variants. Execution time is the time that compilers compile the program variant and emit the warning diagnostics. Alignment time denotes the time of aligning the warning diagnostics among different compilers and identifying the inconsistent warnings. This metric reflects the performance of DIPROM and the comparative approaches.

6 Experimental Results

6.1 Answer to RQ1: Baseline Approaches Comparison

We analyze the effectiveness and efficiency of DIPROM compared with three comparative approaches from two aspects, including the number of detected warning defects and the time spent on detecting warning defects.

7. https://gcc.gnu.org/bugzilla/
8. https://bugs.llvm.org/

TABLE 3
Number of detected warning defects in different testing scenarios for DIPROM and the comparative approaches

Subject	Scenario	DIPROM		HiCOND					Epiphron					Hermes				
		#Avg. bugs	#TP. (*10^3)	#Avg. bugs	imp.	#TP. (*10^3)	*P*-value	Effect size	#Avg. bugs	imp.	#TP. (*10^3)	*P*-value	Effect size	#Avg. bugs	imp.	#TP. (*10^3)	*P*-value	Effect size
GCC-4.8.5	CCS	25.2	53.39	12.5	101.60%	63.49	<0.001	1.000	16.5	52.73%	63.75	<0.001	1.000	20.8	21.15%	54.29	<0.001	1.000
	CVS	19.9	52.67	10.2	95.10%	53.55	<0.001	1.000	15.5	28.39%	60.98	<0.001	1.000	16.1	23.60%	53.45	<0.001	1.000
	COS	4.8	17.11	3.3	45.45%	17.48	<0.001	0.960	3.6	33.33%	19.22	<0.001	0.920	3.9	23.08%	21.74	0.004	0.830
GCC-7.1.0	CCS	13.9	60.54	7.3	90.41%	68.64	<0.001	1.000	10.9	27.52%	70.78	<0.001	1.000	12.0	15.83%	61.95	0.004	0.855
	CVS	7.9	57.69	4.9	61.22%	65.85	<0.001	1.000	6.4	23.44%	68.04	<0.001	0.925	6.6	19.70%	60.21	0.002	0.870
	COS	3.8	23.72	2.6	46.15%	25.84	0.002	0.860	2.8	35.71%	26.61	0.003	0.845	3.2	18.75%	23.31	0.016	0.750
Clang-3.9.0	CCS	13.1	51.15	8.7	50.57%	59.42	<0.001	1.000	9.9	32.32%	60.21	<0.001	1.000	11.5	13.91%	52.66	<0.001	0.950
	CVS	6.3	51.60	4.1	53.66%	58.61	<0.001	0.980	4.8	31.25%	60.32	<0.001	0.935	5.3	18.87%	51.57	0.002	0.860
Clang-7.0.0	CCS	6.9	54.89	4.0	72.50%	62.65	<0.001	1.000	5.4	27.78%	63.50	<0.001	0.975	6.2	11.29%	55.28	0.001	0.850
	Total	101.8	422.76	57.6	76.74%	475.53	-	-	75.8	34.30%	493.41	-	-	85.6	18.93%	434.46	-	-

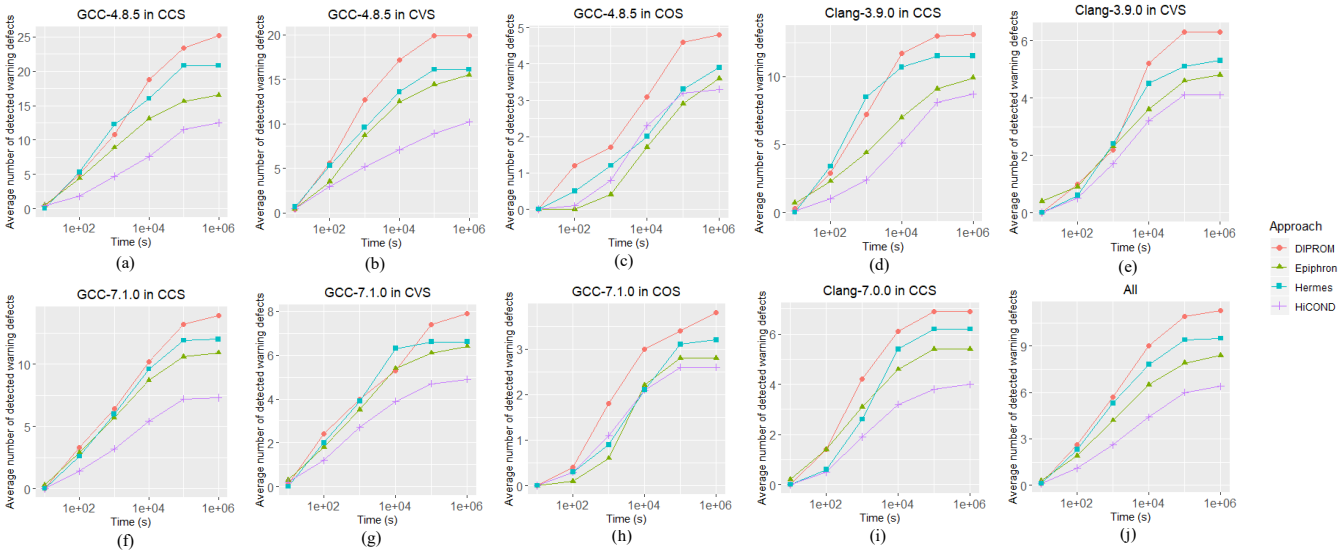

Fig. 6. Time spent on detecting warning defects for DIPROM and the comparative approaches.

6.1.1 Number of Detected Warning Defects

Table 3 shows the average number of detected warning defects discovered by DIPROM and the comparative approaches in 10 runs. The first column is the compiler under test and the second column is the testing scenario for each compiler. Regarding each approach, we present the average number of warning defects (i.e., #Avg. bugs) and the average number of valid test programs (i.e., #TP.) in the following columns. We also show the improvement (i.e., imp.) of DIPROM over the baselines (i.e., three comparative approaches) in terms of the average number of warning defects, i.e., $imp = (DIPROM - baseline)/baseline * 100\%$. Notably, we do not present the results of Clang-3.9.0 in COS and Clang-7.0.0 in both CVS and COS since there is no warning defect discovered by DIPROM and the comparative approaches during the given testing period.

From Table 3, we can see that DIPROM significantly outperforms the comparative approaches in terms of the bug-finding capability. Across all the experiments, the sum of the average numbers of warning defects triggered by DIPROM is 101.8, which outperforms HiCOND (i.e., 57.6), Epiphron (i.e., 75.8), and Hermes (i.e., 85.6) by 76.74%, 34.30%, and 18.93%, respectively. Specifically, we could observe that DIPROM detects more warning defects than HiCOND by 45.45%∼101.60% (column *imp.* of HiCOND). This is because the test programs generated by HiCOND are free from the undefined behaviors, which may have a small probability of triggering compiler warning diagnostics. Therefore, although HiCOND generates more test programs than DIPROM, HiCOND performs worse than DIPROM. In addition, DIPROM detects 23.44%∼52.73% more warning defects than Epiphron (column *imp.* of Epiphron). The reason may be that Epiphron generates test programs according to a set of randomly selected language grammars, which rarely considers how to construct diverse warning-sensitive structures in the generated test programs. Besides, we can see that DIPROM outperforms Hermes on detecting compiler warning defects, which achieves 11.29%∼23.60% improvements. Although Hermes employs the mutation strategy to generate program variants leveraging the same Epiphron-generated test programs as seeds, all the program variants generated by Hermes are enforced to be semantically equivalent to the seed program. Therefore, the control- and data- dependencies in the variants cannot be varied, which may be not capable of thoroughly testing compiler warnings. Interestingly, from Table 3, we can observe that

although DIPROM generates the minimum number of valid test programs, it can construct diverse warning-sensitive test programs triggering a relatively large number of warning diagnostics flags. Accordingly, DIPROM could detect more warning defects than other comparative approaches.

In addition, we conduct the Mann-Whitney U-test [29] with a level of significance 0.05 on the number of detected warning defects between DIPROM and the comparative approaches. The *P*-value (p<0.05) in Table 3 shows that DIPROM performs significantly better than comparative approaches. Furthermore, we also calculate the effect size of the differences between DIPROM and the comparative approaches using the Vargha and Delaneys A_{12} statistics[9] [29]. If the effect of DIPROM is small compared to other approaches, then $A_{12} < 0.5$; otherwise, $A_{12} > 0.5$. From Table 3, we can observe that almost all the effect sizes are greater than 0.8, which indicates that DIPROM has a higher probability of obtaining better results than the comparative approaches. Particularly, the values of effect size for the comparative approaches in CCS and CVS of GCC-4.8.5 are 1.000. This indicates that the numbers of warning defects discovered by DIPROM during 10 runs are larger than those detected by comparative approaches.

At last, we analyze the effectiveness of DIPROM in different testing scenarios and different versions of compilers according to Table 3. Regarding the different testing scenarios, we find that DIPROM always has better performance in CCS. For example, in GCC-4.8.5, DIPROM detects an average number of 25.2 warning defects in CCS, whereas DIPROM only detects 19.9 and 4.8 warning defects on average in CVS and COS, respectively. Particularly, there is no warning defect detected by any of the approaches in COS of Clang-3.9.0 and Clang-7.0.0. This indicates that DIPROM has a higher ability to detect compiler warning defects in the CCS of differential testing. Regarding the different versions of compilers, we can observe that DIPROM detects more warning defects in the older release versions of compilers than in the recent releases. This phenomenon has been already demonstrated in the prior work [25], [28] that the old releases usually contain more compiler bugs than the recent release versions. Even so, we detect an average total number of 25.6 and 6.9 warning defects in GCC-7.1.0 and Clang-7.0.0, respectively, which is larger than those detected by other comparative approaches. This also indicates that DIPROM has a broad capability of detecting compiler warning defects in different versions of compilers.

6.1.2 Time Spent on Detecting Warning Defects

Fig. 6(a)-(i) presents the time spent on detecting warning defects for DIPROM, HiCOND, Epiphron, and Hermes in different testing scenarios. Fig. 6(j) presents the overall results for different approaches. We can observe that DIPROM obviously outperforms comparative approaches in most testing scenarios for both GCC and Clang. For example, from Fig. 6(j), we can see that the average number of warning defects detected by DIPROM is 10.9 during 10^5 seconds, which is larger than those detected by HiCOND (i.e., 6.0), Epiphron (i.e., 7.9), and Hermes (i.e., 9.4) during the same testing period, achieving 81.67%, 37.97%, and 15.96% improvements, respectively. Particularly, in Fig. 6(b), DIPROM could detect an average number of 12.7 warning defects during 10^3 seconds, whereas HiCOND, Epiphron, and Hermes only detect 5.2, 8.7, and 9.6 warning defects in the same testing period, respectively. In addition, when detecting the same number of warning defects, DIPROM spends less time than the comparative approaches. From Fig. 6(j), we can observe that DIPROM only spends 10^3 seconds in detecting an average number of 5.7 warning defects, whereas HiCOND spends nearly 2 magnitudes of the time on detecting the same number of warning defects. It is important and beneficial to detect warning defects earlier because the development resources are always limited and required to find warning defects as soon as possible. Thus, DIPROM is efficient in detecting compiler warning defects.

6.2 Answer to RQ2: Influences of Mutation Operators and Diversity-guided Mutation Strategy

This RQ analyzes the influences of different mutation operators and the diversity-guided mutation strategy.

6.2.1 Number of Detected Warning Defects

Table 4 shows the average number of warning defects discovered by DIPROM and its variants in 10 runs. For each approach, we present the average number of warning defected (i.e., #Avg. bugs) and the average number of valid test programs (i.e., #TP.) in Table 4. Besides, we show the improvement (i.e., imp.) of DIPROM over its three variants on the average number of warning defects and the statistics results of Mann-Whitney U-test [29] for each run. Note that, we do not present the results of Clang-3.9.0 in COS and Clang-7.0.0 in both CVS and COS since there is no warning defect discovered by any of the approaches during the given testing period.

From Table 4, we can see that DIPROM detects more warning defects than its variants. Across all the experiments, the sum of the average numbers of warning defects detected by DIPROM is 101.8, whereas DIPROM$_{prune}$, DIPROM$_{insert}$, and DIPROM$_{random}$ detect 79.4, 89.5, and 93.0 warning defects, achieving 28.21%, 13.74%, and 9.46% improvements, respectively. Specifically, from the columns *imp.* of DIPROM$_{prune}$ and DIPROM$_{insert}$ in Table 4, we can observe that DIPROM detects more warning defects than DIPROM$_{prune}$ and DIPROM$_{insert}$ by 15.93%∼50.00% and 9.52%∼23.08%, respectively. The reason may be that DIPROM integrated two operations would generate more warning-sensitive structures than a single operation. In practice, the effectiveness of DIPROM$_{insert}$ on detecting warning defects is better than DIPROM$_{prune}$. This indicates that inserting additional code may construct more warning-sensitive structures than pruning code from the test programs. Furthermore, we can also observe that DIPROM detects 7.38%∼17.07% more warning defects than DIPROM$_{random}$. This indicates that the diversity-guided mutation in DIPROM brings a positive effort to help detect compiler warning defects. Indeed, the performance of DIPROM$_{random}$ is limited due to the random selection of the mutators during each mutation process. If DIPROM$_{random}$ frequently employs a mutator that could result in program

9. We use the open source code shared by Tim Menzies to calculate A_{12}, https://github.com/txt/ase16/blob/master/doc/stats.md.

TABLE 4
Number of detected warning defects in different testing scenarios for DIPROM and the its variants

Subject	Scenario	DIPROM		$DIPROM_{prune}$					$DIPROM_{insert}$					$DIPROM_{random}$				
		#Avg. bugs	#TP. ($*10^3$)	#Avg. bugs	imp.	#TP. ($*10^3$)	*P*-value	Effect size	#Avg. bugs	imp.	#TP. ($*10^3$)	*P*-value	Effect size	#Avg. bugs	imp.	#TP. ($*10^3$)	*P*-value	Effect size
GCC-4.8.5	CCS	25.2	53.39	16.8	50.00%	56.13	<0.001	1.000	22.7	11.01%	50.23	<0.001	1.000	23.3	8.15%	52.64	0.002	0.870
	CVS	19.9	52.67	15.7	26.75%	54.18	<0.001	1.000	17.3	15.03%	49.89	<0.001	0.940	18.5	7.57%	52.18	0.001	0.870
	COS	4.8	17.11	3.8	26.32%	17.26	<0.001	0.920	3.9	23.08%	16.71	0.001	0.870	4.1	17.07%	16.70	0.012	0.770
GCC-7.1.0	CCS	13.9	60.54	11.5	20.87%	64.47	0.001	0.905	12.4	12.10%	57.26	0.006	0.803	12.8	8.59%	59.83	0.019	0.770
	CVS	7.9	57.69	6.6	19.70%	57.88	0.001	0.910	6.9	14.49%	55.67	0.007	0.815	7.0	12.85%	58.04	0.015	0.775
	COS	3.8	23.72	2.9	31.03%	25.63	0.001	0.865	3.2	18.75%	22.53	0.016	0.750	3.3	15.15%	23.25	0.045	0.700
Clang-3.9.0	CCS	13.1	51.15	11.3	15.93%	53.22	<0.001	0.940	11.6	12.93%	49.60	<0.001	0.940	12.2	7.38%	51.06	0.013	0.780
	CVS	6.3	51.60	4.9	28.57%	52.09	<0.001	0.930	5.2	21.15%	48.76	0.001	0.890	5.4	16.67%	49.83	0.018	0.765
Clang-7.0.0	CCS	6.9	54.89	5.9	16.95%	58.12	<0.001	0.970	6.3	9.52%	51.70	0.012	0.755	6.4	7.81%	54.07	0.013	0.750
	Total	101.8	422.76	79.4	28.21%	438.98	-	-	89.5	13.74%	402.35	-	-	93.0	9.46%	417.60	-	-

errors or the same warning-sensitive structures, there may be a small probability of triggering different warning defects.

To further analyze the reason why DIPROM outperforms $DIPROM_{random}$, we calculate the average program distance according to Formula 1 among program variants generated by DIPROM and $DIPROM_{random}$ for GCC-4.8.5 and Clang-3.9.0, respectively. The results are presented in Fig. 7, where the violin plots show the density of the program distance, and the box plots show the median and interquartile ranges. From the violin plots in Fig. 7, we can observe that the program distance values for DIPROM are distributed relatively more dispersive than those for $DIPROM_{random}$. This indicates that DIPROM could generate program variants that are vastly different from the existing ones to some degree. In addition, from Fig. 7, we find that the median values achieved by DIPROM are larger than those achieved by $DIPROM_{random}$ in both GCC-4.8.5 and Clang-3.9.0. For example, in Clang-3.9.0, the median program distance of program variants generated by DIPROM is 0.37, while it is 0.32 for $DIPROM_{random}$, achieving 15.63% improvement. This illustrates that DIPROM would have a higher probability to generate diverse program variants for compiler warning testing. Furthermore, we conduct Spearman correlation analysis [30] to explore the correlation between the diversity of program variants generated by DIPROM and $DIPROM_{random}$ and the number of warning defects triggered by them. Spearman correlation is a widely used statistics method which is robust to non-normally distributed data [31], [32]. Notably, we use the average program distance to measure the diversity of all the program variants generated by each approach. The Spearman correlation analysis is conducted between the average program distance and the number of detected compiler warning defects for GCC-4.8.5 or Clang-3.9.0 in each run. We find that the Spearman correlation coefficient is 0.636 in GCC-4.8.5 (with *p*-value=0.001) and 0.625 in Clang-3.9.0 (with *p*-value=0.002). This means that there is a positive correlation between the diversity of program variants and the number of detected compiler warning defects.

In Table 4, the Mann-Whitney U-test confirms that DIPROM performs better than its variants (p<0.05). Besides, we conduct the A_{12} statistics on the number of detected warning defects. From Table 4, we can observe that almost all the effect sizes among DIPROM, $DIPROM_{prune}$, $DIPROM_{insert}$, and $DIPROM_{random}$ are greater than 0.5, which indicates that DIPROM has a relatively higher probability of obtaining better results than its variants. Note that, although the absolute difference between DIPROM and $DIPROM_{random}$ is not larger in several testing scenarios on detecting warning defects, DIPROM is still effective via diversity-guided mutation from the statistical analysis.

Fig. 7. Comparison of the diversity of test programs generated by DIPROM and $DIPROM_{random}$.

6.2.2 Time Spent on Detecting Warning Defects

Fig. 8(a)-(i) presents the time spent on detecting warning defects in each testing scenario for DIPROM and its variants. Fig. 8(j) shows the overall results. From Fig. 8, we can observe that DIPROM outperforms its variants on detecting the average number of warning defects in almost all the testing scenarios of GCC and Clang. For example, from the overall results in Fig. 8(j), we can see that DIPROM detects an average number of 9.0 warning defects during 10^4 seconds, which is larger than that detected by $DIPROM_{prune}$ (i.e., 6.7), $DIPROM_{insert}$ (i.e., 7.8), and $DIPROM_{random}$ (i.e., 8.0) during the same testing period, achieving 34.33%, 15.38%, and 12.50% improvements, respectively. Particularly, in Fig. 8(a), DIPROM detects 23.4 warning defects on average during 10^5 seconds, whereas $DIPROM_{prune}$, $DIPROM_{insert}$, and $DIPROM_{random}$ only detect an average of 15.2, 20.8, and 21.6 warning defects during the same testing period, respectively. In addition, DIPROM spends less time than its variants in detecting the same number of warning defects. In Fig. 8(j), DIPROM detects an average of

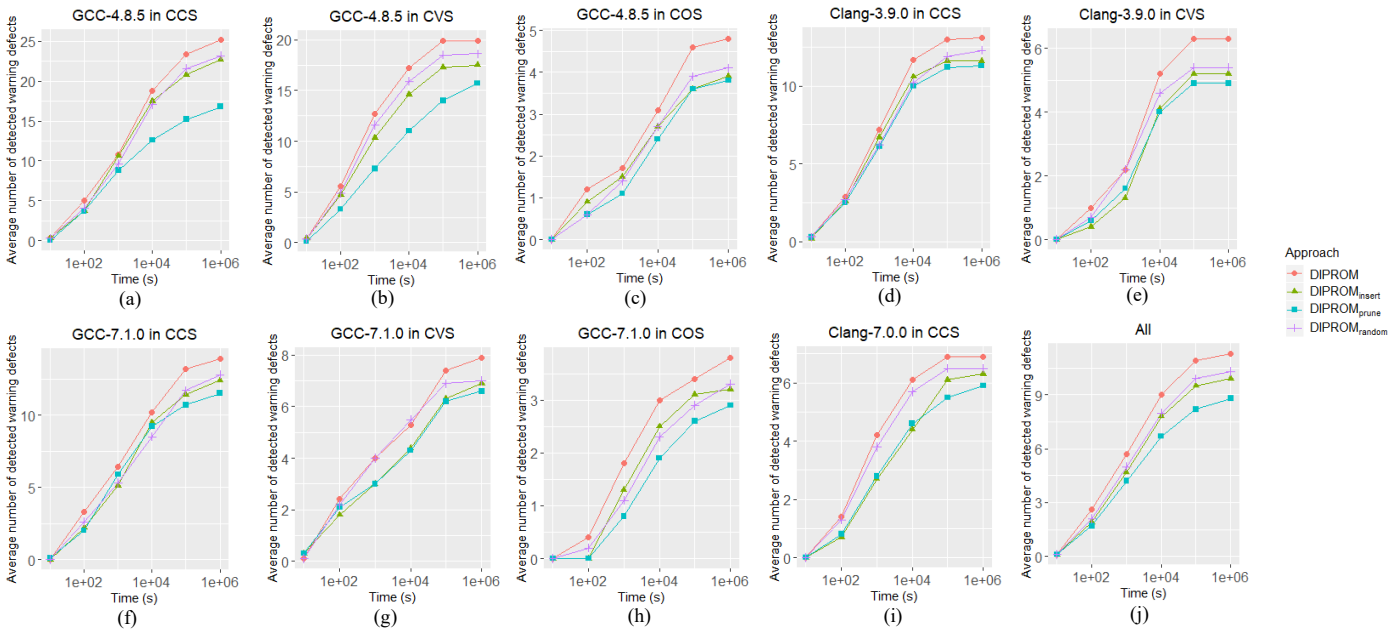

Fig. 8. Time spent on detecting warning defects for DIPROM and its variants.

TABLE 5
Information of reported warning defects

	GCC	Clang	Total
Confirmed	4	1	5
Pending	0	1	1
Duplicate	1	0	1
Suspended	1	0	1
Reported	6	2	8

```
1 // file = s.c
2 typedef int abc;
3 static abc * const f1(void);//wrong location

Clang 10 outputs:
s.c:3:18: warning: 'const' type qualifier on
    return type has no effect [-Wignored-
    qualifiers]
3 | static int32_t * const f1(void);
```

Fig. 9. GCC erroneous warning defect #92392 (*https* : *//gcc.gnu.org/bugzilla/show*$_b$*ug.cgi?id* = 92392).

9.0 warning defects within the testing period of 10^4 seconds, whereas DIPROM$_{prune}$ spends nearly 2 magnitudes of the time on detecting the same number of warning defects. Therefore, DIPROM is efficient in detecting compiler warning defects than its variants.

6.3 Answer to RQ3: New Warning Defects Discovering

6.3.1 Detected warning defects

Table 5 summarizes all the detected warning defects for the latest development versions of compilers we reported so far. In two months, we have reported 8 warning defects, of which 5 have been confirmed by developers. There is still a warning defect pending developers' responses. Note that, since many compiler bugs have been fixed in the latest development versions of GCC and Clang, it is not easy to detect new compiler warning defects in the developed versions of compilers. Despite that, it is still worthy of detecting new warning defects in the development version to ensure the quality of compiler warning diagnostics.

Table 6 details all the confirmed compiler defects, including the defect id, the warning diagnostic flag, the priority of warning defects, the current status, the defect type, the strategy of differential testing, and the affected compiler version. Priority indicates the order in which warning defects should be fixed by developers, ranging from P1 to P5. P1 is the highest level and defects in this level should be fixed soon. As shown in Table 6, all the confirmed defects are labeled with the default priority P3 and none of them is demoted to either P4 or P5. Defect types are categorized into two classes, i.e., the missing warning and the erroneous warning. From Table 6, we can observe that three confirmed warning defects are missing warnings and one of them is fixed by developers. Strategy refers to the testing scenarios used to detect the corresponding warning defects, including CCS, CSV, and COS.

6.3.2 Confirmed defects samples

GCC warning defect #92392. Fig. 9 shows a GCC erroneous warning defect discovered by CCS. GCC emits a wrong warning column for Line 3 that the return type qualifiers *"abc"* is ignored on function return. However, the "const" type qualifier has no effect in the static function and Clang exactly outputs the warning location. As a result, this is probably a problem of keeping track of the qualifiers in GCC.

GCC warning defect #92378. Fig. 10 is a missing warning of GCC. GCC is expected to emit a warning indicating that the array subscript of *a[3]* is above array bounds in Line

TABLE 6
Confirmed warning defects

Number	Defect Id	Warning Diagnostic Flag	Priority	Status	Defect Type	Strategy	Affected Version
1	GCC-92209	Wstrict-prototypes	P3	Confirmed	Erroneous	CCS	GCC-10.0.0
2	GCC-92210	Wfor-loop-analysis	P3	Confirmed	Missing	CCS	GCC-10.0.0
3	GCC-92378	Warray-bounds	P3	Fixed	Missing	COS	GCC-10.0.0
4	GCC-92392	Wignored-qualifiers	P3	Confirmed	Erroneous	CCS	GCC-10.0.0
5	Clang-43573	Wtautological-compare	P3	Confirmed	Missing	CCS	Clang-10.0.0

```
1 // file = s.c
2 #include <stdio.h>
3 int main(){
4   int a[1]={0};
5   printf("%d", a[3]);//no warning here
6 }

Clang 10 outputs:
s.c:5:16: warning: array index 3 is past the end
    of the array (which contains 1 element) [-
    Warray-bounds]
5 |   printf("%d", a[3]);
```

Fig. 10. GCC missing warning defect #92378 (*https://gcc.gnu.org/bugzilla/show_bug.cgi?id* = 92378).

```
1 // file = s.c
2 int main(){
3   int a;
4   int b=(0!=((-1)|((a=1)==1)));//no warning here
5   return 0;
6 }

GCC 10 outputs:
s.c:4:12: warning: bitwise comparison always
    evaluates to true [-Wtautological-compare]
4 | int b=(0!=((-1)|((c = 1) == 1)));
```

Fig. 11. Clang missing warning defect #92479 (*https://bugs.llvm.org/show_bug.cgi?id* = 92479).

5. However, no warning is emitted when GCC compiles it with an optimization level (i.e., *-O1/O2/O3*).

Clang warning defect #92479. Fig. 11 shows a missing-warning defect of Clang. On Line 4, Clang misses the warning of bitwise comparison for the left operand "-1" in the bitwise inclusive OR operator "|"; whereas GCC warns in this situation to help developers examine the semantics of the statement. However, Clang only explicitly checks the value from the IntegerLiteral AST node that ignores the corresponding bits in the operand.

6.3.3 Pending Warning Defect

Clang warning defect #43647. This is a missing-warning defect as shown in Fig. 12. GCC emits a warning indicating that "the comparison will always evaluate as 'false' for the address of *'a'* will never be NULL". Developers could check the code carefully whether it is correct to specify the address of a point instead of the point itself. In contrast, Clang overlooks this warning leaving the questionable code undiscovered.

```
1 // file = s.c
2 int main(){
3   int *a = (void *) 0;
4   int b = (&a) == ((void *)0);//no warning here
5   return b;
6 }

GCC 10 outputs:
s.c:4:16: warning: the comparison will always
    evaluate as 'false' for the address of 'a'
    will never be NULL [-Waddress]
4 |   int b = (&a) == ((void *) 0);
```

Fig. 12. Clang pending warning defect #43647(*https://bugs.llvm.org/show_bug.cgi?id* = 43647).

7 Threats To Validity

7.1 Threats to Internal Validity

The threats to internal validity mainly lie in three aspects. First, as a heuristic algorithm, we set several probabilities in DIPROM to mutate the AST of the test program. For example, when applying the mutation operator on AST, we mutate a parent node or a leaf node based on the probability of 50%. These probabilities may influence the effectiveness of DIPROM. However, to ensure the generality of DIPROM, we set the same probabilities when applying the mutation processes. Second, we identify compiler warning defects in our experiments by both manually analyzing the existing bug reports and automatically checking the correcting versions. However, there is a threat to introduce false positives in these two approaches. To alleviate this threat, all the detected warning defects are checked by two authors of this paper and the results must be agreed by these two authors. Third, we do not compare DIPROM with the baselines on the latest development versions of GCC and Clang. This is because the latest development versions usually contain a small number of compiler bugs, which could not obtain statistically significant results in the evaluation. To reduce this threat, we conduct experiments over two recent versions of GCC and Clang, i.e., GCC-7.1.0 and Clang-7.0.0. The results also indicate that DIPROM is effective in detecting compiler warning defects.

7.2 Threats to External Validity

The threats to external validity mainly lie in the subjects and the seed programs. Subjects refer to the compilers we selected for evaluation and the seed programs are the test programs generated by program generation tools, such as Epiphron and Csmith.

In our experiments, we only used four versions of GCC compilers and LLVM compiler as subjects, including two older releases and two recent releases. These subjects may not be representative enough for other compilers. It is still unknown whether our approaches could be generalized to other C compilers. In the future, we will apply DIPROM to different compilers for compiler warning testing.

Although there exist several test program generators [2], [10], DIPROM employs Epiphron-generated test programs as the seeds. To reduce the threat from the seed program, we randomly generate a series of seed programs by Epiphron and use the same seeds to evaluate DIPROM and other comparative approaches, rather than selecting a set of representative and diverse seed programs. This also indicates that our approach indeed increases the diversity of structures in test programs.

8 Related Work

In this section, we introduce the related work to the compiler testing. In general, the process of compiler testing consists of three main aspects, i.e., test program generation (Section 8.1), test oracle construction (Section 8.2), and test program reduction (Section 8.3).

8.1 Test Program Generation

In the area of compiler testing, test program generation is the initial and the most crucial issue as compilers require a large number of sophisticated test programs as inputs [6], [43], [44], [45]. Our work also targets this issue. Besides the manually constructed validation suites (e.g., Plum Hall [46], SuperTest [47], and Perennial [48]), the main techniques for test program generation can be broadly categorized as grammar based program generation and mutation based program generation.

The grammar based program generation focuses on automatically generating massive test programs to comprehensively test compilers [42]. The notable efforts are Csmith [10], HiCOND [25], and Epiphron [2]. All of them are evaluated to be effective in finding bugs in mature compilers. Csmith generates random test programs by defining and sampling a probabilistic grammar of the C programming language. It has been applied to find hundreds of bugs in GCC and Clang via differential testing. HiCOND could produce a set of configurations for Csmith, and thus generates diverse test programs for compiler testing. Epiphron generates massive test programs based on a set of randomly selected language grammars. Since Epiphron aims at compiler warning testing, it intentionally inserts warning-free bodies into the generated programs. Although we also focus on detecting compiler warning defects, DIPROM is a mutation based approach, which is different from Csmith, HiCOND, and Epiphron.

The mutation based program generation targets modifying existing test programs. The most representative mutation based approach is Equivalence Modulo Inputs (EMI) [20]. Starting with a seed test program, EMI deletes or inserts code in the code regions to make the mutated programs preserve the original semantics w.r.t. the inputs. EMI has three instantiations, i.e., Orion [20], Athena [36], and Hermes [49]. Orion deletes dead code regions of test programs randomly and Athena inserts code into and removes code from dead code regions. Hermes complements the two tools above, which performs mutations on both dead and live code regions to derive semantically equivalent valid variants from existing test programs.

Our work is different from them in three aspects. First, Orion and Athena generate syntactically equivalent program variants by pruning or inserting unexecuted statements (i.e., dead code). They may not construct warning-sensitive structures in the live code. However, compiler warning diagnostics on the dead code are usually ignored by compilers' developers [2]. Therefore, they are not suitable for compiler warning testing. In contrast, DIPROM mutates the live code of test programs to construct warning-sensitive structures in the program variants. Second, although Hermes can insert code in the live code regions, the generated program variants must maintain the EMI (Equivalence Modulo Inputs) property. That is, all the program variants generated by Hermes are enforced to be semantically equivalent to the seed program. Therefore, the control- and data-dependencies of the variants cannot be varied, which may be not capable of thoroughly testing compiler warnings. Different from Hermes, DIPROM attempts to generate program variants with different control- and data- dependencies such that these variants are different from the seed and existing variants. Third, both Orion and Hermes mutate the seed test program randomly, while DIPROM is a diversity-guided mutation approach.

Recently, Chen et al. [50], [51] proposed two non-semantics-preserving mutations for testing JVM implementations, i.e., classfuzz and classming. Classfuzz tests JVM's startup processes by altering classfiles via 129 mutators, and leverages MCMC sampling to guide the mutator selection. The selection of each mutator is based on the success rate of that mutator on generating representative test programs. After that, classfuzz applies the selected mutator to generate a program variant, and determines whether to accept or reject the program variant according to the coverage uniqueness of a JVM when executing the program variant. Classming aims at testing JVM's execution engines based on hooking instruction insertions/deletions. After the mutation process, classming employs the Metropolis algorithm to determine whether to accept a program variant using the seed coverage as the discipline in the Metropolis choice.

Although both classfuzz and classming utilize the MCMC-guided strategy during the mutation process, DIPROM is different from them in three aspects. First, both classming and classfuzz generate serialized program variants using Markov Chains, whereas DIPROM constructs each program variant based on the initial seed test program, i.e., the *i-th* program variant is independent of the *(i-1)-th* program variant. Second, despite that both classfuzz and DIPROM employ the Metropolis algorithm for mutator selection, the ranking of mutator scores in the Metropolis choice of classfuzz only depends on the success rate of each mutator, whereas DIPROM designs a more complex priority metric (i.e., Formula 4) to rank these mutators. Third, DIPROM accepts a program variant when it is compilable, whereas classming utilizes the Metropolis algorithm to accept a program variant and treats the seed coverage

TABLE 7
Technical comparison among DIPROM, classming, classfuzz, and Hermes

Technical difference	DIPROM	classming	classfuzz	Hermes
General process	Iterative mutation	Iterative mutation	Iterative mutation	Random mutation
Mutation objective	The generated program variant is compilable and diverse from existing program variants.	The generated bytecode file is live and diverse from its seed.	The generated bytecode file is diverse from its seed.	The generated program variant is semantically equivalent to its seed.
Type of seed	C source code	Java bytecode file	Java bytecode file	C source code
Mutators	63 mutators for altering both the syntax and control- and data-flows in the seed	5 Inserting/deleting hooking instructions for altering control- and data-flows in the seed	129 mutators for altering Java bytecode syntactically	Inserting three types of EMI snippets (FCB, TG, and TCB)
Mutator selection	Mutator is guided by both the program distance and the success rate.	Mutator is randomly selected.	Mutator is guided by the success rate.	Mutator is randomly selected.
Mutated regions	Live code regions	Live bytecode	Entire bytecode	Entire code regions
Mutant acceptance	Mutant is compilable and different from existing program variants.	Mutant has high seed coverage and is different from the seed.	Mutant is different from the seed.	Mutant is an EMI mutant of the seed.
Test subjects	C compilers	JVMs' verifiers and execution engines	JVMs' startup processes	C compilers

information as a discipline in the Metropolis choice.

In addition, Table 7 provides a detailed comparison among DIPROM, Hermes, classfuzz, and classming. Clearly, the four approaches take their respective strategies and follow their respective processes to generate program mutants. Regarding the compiler warning diagnostics, abundant compilable and diverse test programs are more suitable for triggering compiler warning defects. To the best of our knowledge, DIPROM is the first effort leveraging mutation based approach to generate programs for this purpose.

8.2 Test Oracle Construction

Test oracle issue is a long-term challenge in compiler testing, since it is difficult to determine whether the actual output of the compiler under test is as expected or not given a test program. In the literature, many technologies of compiler testing has been proposed to mitigate this issue, which fall into three categories, i.e., Randomized Differential Testing (RDT) [8], [10], [18], [19], Different Optimization Levels (DOL) [6], and Equivalence Modulo Inputs (EMI) [20]. RDT is a well-known technology in compiler testing. It uses two or more comparable compilers that implement the same specification. For the same test inputs, these comparable compilers should produce the same results. Hence, a compiler may contain bugs when the generated results are different from the majority of other compilers (e.g., half of the other results). DOL is a variant of RDT, which compares the results under different optimization levels of a single compiler. This is, when a test program is compiled in the same experimental condition but under different optimization levels (i.e., -O1, -O2, or -O3), it may output different results which indicates a bug in the compiler under test. The third technique is called EMI. It detects compiler bugs by comparing the results between a test program and its variants in one compiler. EMI generates a series of variants from an existing test program which are semantically equivalent to this test program. Thus, these variants should produce the same results as the test program under the same test input; if not, the compilers under test could contain bugs. An empirical study shows that different compiler testing technologies are effective at detecting distinct compiler bugs [6]. DOL is more effective at detecting optimization-related bugs, and RDT can substitute EMI in detecting optimization-irrelevant bugs. Our work also leverages RDT and DOL to test compilers. The difference is that, we have implemented RDT under two different testing scenarios, namely the cross-compiler scenario and the cross-version scenario. Different testing scenarios have distinct bug-finding capabilities and are complementary to each other.

8.3 Test Program Reduction

Test program reduction reduces a bug-triggering test program to a smaller one that still exposes the same compiler bug. Zeller et al. [52] developed delta debugging for test program reduction, and instantiated it in *dd* and *ddmin* algorithms. Based on the delta debugging algorithm, Berkeley Delta [34] is proposed to reduce test program at line granularity. By allowing the only whole-line deletions, Berkeley Delta reduces the size of the test programs until most lines in the program are semantically independent from each other. Furthermore, to produce sufficiently small and valid test program, C-Reduce [33] is implemented via a variety of transformations to avoid undefined behaviors. The program transformations in C-Reduce are repeatedly applied until the transformations lead to an error (due to transformation tool crashing) or the space of reductions for the transformations is exhausted. Subsequently, C-Reduce is extended to CL-Reduce [53] for reducing OpenCL kernels that trigger compiler bugs. The reduced test programs are usually small enough to be directly reported to compiler developers for bug reproduction. Our work also employs the C-Reduce tool to reduce C test programs that trigger new compiler warning defects.

9 Conclusion

In this paper, we aim to detect compiler warning defects. To this end, we propose a novel diversity-guided program mutation to generate test programs for compiler warning testing, called DIPROM. DIPROM first removes all the dead code regions in a given test program, which produces a program with only live code. Then, a diversity-guided strategy is utilized to select mutation operators to prune or insert code snippets in the live code, resulting in a set of program variants with diverse warning-sensitive structures. Finally, the generated program variants are utilized to test compiler warnings by differential testing. We evaluated the effectiveness of DIPROM on two popular C compilers, i.e., GCC and Clang. The results show that DIPROM significantly outperforms three state-of-the-art approaches (i.e., HiCOND, Epiphron, and Hermes) by 76.74%, 34.30%, and 18.93% on average in the term of the number of detected warning defects, respectively. In addition, DIPROM spends less time than the comparative approaches on detecting the same number of warning defects. Furthermore, DIPROM has been successfully employed in the latest development versions of GCC and Clang and exposed 8 warning defects within two months.

Acknowledgment

We thank all the compiler developers for their confirmation of our bug reports and the TSE reviewers for their valuable feedbacks on this paper. This work is partially supported by National Natural Science Foundation of China under grants 62032004, 61772107.

References

[1] J. L. Anderson, "How to produce better quality test software," *IEEE Instrumentation and Measurement Magazine*, vol. 8, no. 3, pp. 34-38, 2005.

[2] C. Sun, V. Le, and Z. Su, "Finding and analyzing compiler warning defects," in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 203-213.

[3] Clang Bug #18905. Available: https://bugs.llvm.org/show_bug.cgi?id=18905, accessed: 2021-02-10.

[4] C. Lindig, "Random testing of C calling conventions," In *Proceedings of the 6th international symposium on Automated analysis-driven debugging*, ACM, 2005, pp. 3-12.

[5] E. Nagai, H. Awazu, N. Ishiura, and N. Takeda, "Random testing of C compilers targeting arithmetic optimization," In *Workshop on Synthesis And System Integration of Mixed Information Technologies*, 2012, pp. 48-53.

[6] J. Chen, W. Hu, D. Hao, Y. Xiong, H. Zhang, L. Zhang, and B. Xie, "An empirical comparison of compiler testing techniques," in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 180-190.

[7] T. Yoshikawa, K. Shimura, and T. Ozawa, "Random program generator for Java JIT compiler test system," In *Proceedings of the 3rd International Conference on Quality Software*, 2003, pp. 20-23.

[8] C. Zhao, Y. Xue, Q. Tao, L. Guo, and Z. Wang, "Automated test program generation for an industrial optimizing compiler," In *ICSE Workshop on Automation of Software Test*, 2009, pp. 36-43.

[9] W. M. McKeeman, "Differential testing for software," *Digital Technical Journal*, vol. 10, no. 1. pp. 100-107, 1998.

[10] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers," in *Proceedings of the 32nd Conference on Programming Language Design and Implementation*, 2011, pp. 283-294.

[11] GCC developers. [n. d.]. GCC Testsuites. Available: https://gcc.gnu.org/onlinedocs/gccint/Testsuites.html, accessed: 2021-02-10.

[12] LLVM developers. [n. d.]. LLVM Testing Infrastructure Guide. Available: https://llvm.org/docs/TestingGuide.html, accessed: 2021-02-10.

[13] E. Aftandilian, R. Sauciuc, S. Priya, and S. Krishnan, "Building useful program analysis tools using an extensible java compiler," in *Proceedings of the 12th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2012, pp. 14-23.

[14] M. Howard, "A process for performing security code reviews," *IEEE Security and Privacy*, vol. 4, no. 4, pp. 74-79, 2006.

[15] M. A. Hadavi, H. M. Sangchi, V. S. Hamishagi, and H. Shirazi, "Software security; a vulnerability activity revisit," in *Proceedings of the 3rd International Conference on Availability, Reliability and Security*, 2008, pp. 866-872.

[16] A. Nayyar, A. Kumar, and U. Saxena, "Compiler for detection of program vulnerabilities," *International Journal of Computer Applications*, vol. 104, no. 6, pp. 25-31, 2014.

[17] A. Allain. Why Bother with Compiler Warnings. Available: http://www.cprogramming.com/tutorial/compiler_warnings.html, accessed: 2021-02-10.

[18] W. M. McKeeman, "Differential testing for software," *Digital Technical Journal*, vol. 10, no. 1, pp. 100-107, 1998.

[19] F. Sheridan, "Practical testing of a C99 compiler using output comparison," *Software: Practice and Experience*, vol. 37, no. 14, pp. 1475-1488, 2007.

[20] V. Le, M. Afshari, and Z. Su, "Compiler validation via equivalence modulo inputs," in *Proceedings of the 35th Conference on Programming Language Design and Implementation*, 2014, pp. 216-226.

[21] J. Chen, G. Wang, D. Hao, Y. Xiong, H. Zhang, L. Zhang, and B. Xie, "Coverage prediction for accelerating compiler testing," *IEEE Transactions on Software Engineering*, 2018, To appear.

[22] G. Grieco, M. Ceresa, and P. Buiras, "QuickFuzz: an automatic random fuzzer for common file formats," In *International Symposium on Haskell*, 2016, pp. 13-20.

[23] G. Grieco, M. Ceresa, A. Mista, and P. Buiras, "QuickFuzz testing for fun and profit," *Journal of Systems and Software*, vol. 134, no. 12, pp. 340-354, 2017.

[24] W. Han, B. Joe, B. Lee, C. Song, and I. Shin, "Enhancing memory error detection for large-scale applications and fuzz testing," In *Network and Distributed System Security Symposium (NDSS)*, 2018.

[25] J. Chen, G. Wang, D. Hao, Y. Xiong, H. Zhang, and L. Zhang, "History-guided configuration diversification for compiler test-program generation," In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, 2019, pp. 305-316.

[26] A. Fischer, C. Y. Suen, V. Frinken, K. Riesen, and H.Bunke, "Approximation of graph edit distance based on Hausdorff matching," *Pattern Recognition*, vol. 48, no. 2, pp. 331-343, 2015.

[27] Wikipedia. Jaccard index. Available: http://en.wikipedia.org/wiki/Jaccard_index, accessed: 2021-02-10.

[28] J. Chen, J. Han, P. Sun, L. Zhang, D. Hao, and L. Zhang, "Compiler bug isolation via effective witness test program generation," in *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 223-234.

[29] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *Proceedings of the 33rd International Conference on Software Engineering*, 2011, pp. 1-10.

[30] J. Hauke and T. Kossowski, "Comparison of values of Pearson's and Spearman's correlation coefficients on the same sets of data," *Quaestiones Geographicae*, vol. 30, no. 2, pp. 87-93, 2011.

[31] W. Zou, J. Xuan, X. Xie, Z. Chen, and B. Xu, "How does code style inconsistency affect pull request integration? An exploratory study on 117 GitHub projects," *Empirical Software Engineering*, vol. 24, no. 6, pp. 3871-3903, 2019.

[32] H. Jiang, X. Li, Z. Ren, J. Xuan, and Z. Jin, "Toward better summarizing bug reports with crowdsourcing elicited attributes," *IEEE Transactions on Reliability*, vol. 68, no. 1, pp. 2-22, 2018.

[33] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, X. Yang, "Test-case reduction for C compiler bugs," *In Proceedings of the 33th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2012, pp. 335-346.

[34] S. McPeak, D. S. Wilkerson, and S. Goldsmith. Berkeley Delta. Available: http://delta.tigris.org/, accessed: 2021-02-10.

[35] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 183-200, 2002.

[36] V. Le, C. Sun, and Z. Su, "Finding deep compiler bugs via guided stochastic program mutation," in *Proceedings of the 2015 International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2015, pp. 386-399.

[37] Q. Zhang, C. Sun, and Z. Su, "Skeletal program enumeration for rigorous compiler testing," in *Proceedings of the 38th Conference on Programming Language Design and Implementation*, 2017, pp. 347-361.

[38] F. Nielson, H. R. Nielson, and C. Hankin, "Principles of program analysis," Springer Verlag Berlin, 1999.

[39] Y. Chen, A. Groce, C. Zhang, W.-K. Wong, X. Fern, E. Eide, and J. Regehr, "Taming compiler fuzzers," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2013, pp. 197-208.

[40] 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*, 2015, pp. 327-337.

[41] J. Chen, Y. Bai, D. Hao, Y. Xiong, H. Zhang, and B. Xie, "Learning to prioritize test programs for compiler testing," in *Proceedings of the 39th International Conference on Software Engineering*, 2017, pp. 700-711.

[42] J. Chen, J. Patra, M. Pradel, Y. Xiong, H. Zhang, D. Hao, and L. Zhang, "A survey of compiler testing." *ACM Computing Surveys*, 2020, To appear.

[43] E. Nagai, A. Hashimoto, and N. Ishiura, "Reinforcing random testing of arithmetic optimization of C compilers by scaling up size and number of expressions," *IPSJ Transactions on System LSI Design Methodology*, vol. 7, no. 4, pp. 91-100, 2014.

[44] C. Sun, V. Le, Q. Zhang, and Z. Su, "Toward understanding compiler bugs in gcc and llvm," in *Proceedings of the 25th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2016, pp. 294-305.

[45] Y. Tang, Z. Ren, W. Kong, and H. Jiang, "Compiler testing: a systematic literature analysis," *Frontiers of Computer Science in China*, vol. 14, no. 1, pp. 1-20, 2020.

[46] Plum Hall, Inc, "The Plum Hall Validation Suite for C," [Online]. Available: http://www.plumhall.com/stec.html, accessed: 2021-02-10.

[47] ACE, "SuperTest compiler test and validation suite," [Online]. Available: http://www.ace.nl/compiler/supertest.html, accessed: 2021-02-10.

[48] Perennial, "The Perennial Validation Suite for C and C++," [Online]. Available: https://www.peren.com/, accessed: 2021-02-10.

[49] C. Sun, V. Le, and Z. Su, "Finding compiler bugs via live code mutation," in *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, 2016, pp. 849-863.

[50] Y. Chen, T. Su, C. Sun, Z. Su, and J. Zhao, " Coverage-directed differential testing of JVM implementations," in *Proceedings of the 37th annual ACM SIGPLAN conference on Programming Language Design and Implementation*, 2016, pp. 85-99.

[51] Y. Chen, T. Su, and Z. Su, "Deep differential testing of JVM implementations, " in *Proceedings of the 41st International Conference on Software Engineering*, 2019, pp. 1257-1268.

[52] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 183-200, 2002.

[53] M. Pflanzer, A. F. Donaldson, and A. Lascu, "Automatic test case reduction for opencl," In *Proceedings of the 4th International Workshop on OpenCL*, 2016, pp. 1-12.