Uncovering bugs in code coverage profilers via control flow constraint solving

Yang Wang, Peng Zhang, Maolin Sun, Zeyu Lu, Yibiao Yang, Yutian Tang, Junyan Qian, Zhi Li, Yuming Zhou

Abstract—Code coverage has been widely used as the basis for various software quality assurance techniques. Therefore, it is of great importance to ensure that coverage profilers provide reliable code coverage. However, it is challenging to validate the correctness of the code coverage generated due to the lack of an effective oracle. In this paper, we propose an effective approach based on control flow constraint solving to test coverage profilers and have implemented a coverage bug hunting tool, DOG (finD cOverage buGs). Our core idea is to leverage inherent control flow features to generate control flow constraints that the resulting coverage statistics should respect. If DOG identifies any unsatisfiable constraints, it signifies the presence of incorrect coverage statistics. In such cases, DOG provides detailed diagnostic information about the suspicious coverage statistics for manual inspection. Compared with the state-of-the-art works, DOG has the following prominent advantages: (1) wide applicability: DOG eliminates the need for multiple coverage profilers (as required by differential testing) and program variants (as needed in metamorphic testing), making it highly versatile; (2) unique testing capability: DOG effectively analyzes and utilizes relationships among available coverage statistics, boosting its testing capabilities; and (3) enhanced interpretability: DOG provides clear control flow explanations for incorrect code coverage, enabling the localization of suspicious coverage areas. During our testing period with DOG, we successfully identified and reported 27 bugs in Gcov and llvm-cov, both widely-used coverage profilers. Of these, 17 bugs have been confirmed (11 have been fixed), 3 were deemed expected behaviors by developers, and 7 remain unresolved. Remarkably, 21 out of 24 unexpected bugs had been latent for over two and a half years, and nearly half of the coverage bugs (10 out of 24) were undetectable by state-of-the-art coverage profiler validators. These results demonstrate the effectiveness and importance of using DOG to improve the reliability of code coverage profilers.

Index Terms—Coverage bugs, control flow, constraint solving, coverage profilers, testing

1 Introduction

CODE coverage is a metric that measures the extent to which a test suite exercises a software system [1]. The code coverage statistics generated by coverage profilers have been widely adopted in software quality assurance activities. In recent decades, many studies have used code coverage to guide efficient and effective testing (e.g., fuzzing testing [2], [3], [4], [5], compiler testing [6], [7], [8], mutant testing [9], [10], regression testing [11], [12], and test case generation [13], [14], [15]) and debugging (e.g., fault localization [16], [17], [18] and automated program repair [19], [20], [21]). The accuracy of generated code coverage statistics is of paramount importance, as incorrect measurements could potentially mislead researchers or developers in their software engineering practices. Therefore, it is crucial for coverage profilers to ensure the correctness of the code coverage statistics they generate.

However, coverage profilers themselves are software and are prone to errors. It is challenging to validate the correctness of the generated coverage statistics due to the lack of an effective oracle. Different from the oracle in white box testing that verifies the functionality of a program, the expected coverage statistics for a coverage profiler cannot be directly obtained or specified via any specification. Even if we obtain the oracle through heavy manual verification, it still requires a lot of human effort to examine the correctness of the coverage statistics.

To address the aforementioned challenge, the current mainstream solutions involve two approaches: differential testing, utilized in C2V [22], and metamorphic testing, adopted in Cod [23]. C2V operates under the assumption that different coverage profilers should yield identical code coverage statistics for a given input program. By comparing the coverage reports generated by multiple coverage profilers, C2V can reveal code coverage bugs. On the other hand, Cod alleviates the oracle problem through a metamorphic relation. This means that under the identical profiler, an input program should exhibit the same code coverage statistics for executed blocks as its path-equivalent variants, which are generated by removing unexecuted statements. Inconsistencies in coverage reports between an input program and its path-equivalent variants can then be used to detect bugs. However, despite demonstrating their capability to reveal real coverage bugs in popular coverage profilers like Gcov [24] and llvm-cov [25], both C2V and Cod possess inherent limitations that impede their effectiveness in defect detection, thereby hindering their practical applicability.

- *Y. Wang, P. Zhang, M. Sun, Z. Lu, Y. Yang, Y. Zhou are with State Laboratory for Novel Software Technology, Nanjing University.*
- *Y. Tang is with School of Computer Science and Engineering, University of Glasgow.*
- *J. Qian is with the Key Lab of Education Blockchain and Intelligent Technology, Ministry of Education, Guangxi Normal University, and Guangxi Collaborative Innovation Center of Multi-source Information Integration and Intelligent Processing.*
- *Z. Li is with School of Computer Science and Engineering & School of Software, Guangxi Normal University*

*Please note that all acknowledgments should be placed at the end of the paper, before the bibliography (**note that corresponding authorship is not noted in affiliation box, but in acknowledgment section**).*

- **Applicability:** C2V must use appropriate alternative profilers for benchmarking purposes and cannot be applied when there is only one profiler available (e.g., the programming language Cangjie, developed by Huawei, offers only one coverage profiler). For Cod, the strategy of path-equivalent variant generation cannot be employed if there is no unexecuted code to remove (e.g., no statement is marked as unexecuted).
- **Testing capacity:** Inconsistent interpretations of the same coverage semantics by independently implemented coverage profilers can lead to omissions and false alarms. For example, when applying C2V, a phenomenon known as the "weak inconsistency" [23] emerges, where certain complex statements fail to receive coverage from certain profilers, resulting in missed bugs. The test space of Cod is limited as it focuses on the specific mechanism of coverage profilers, which can lead to underutilization of the rich coverage statistics of executed statements. Consequently, the results demonstrate that Cod is notably less effective in uncovering llvm-cov bugs, accounting for only a small fraction of all the bugs reported by Cod (3 out of 23).
- **Interpretability:** Both C2V and Cod rely on text-comparison for their analysis. When coverage inconsistency is uncovered, they offer limited guidance to alleviate the burden of manual inspection. Testers are still required to precisely grasp the program context to determine the oracle, creating a higher threshold for bug comprehension, particularly in large programs. Furthermore, even if some potential bugs are identified, no interpretation is provided to deduplicate and reduce them.

In this paper, we introduce *control flow constraint solving*, a brand-new effective approach for addressing the oracle problem in coverage profiler testing. Based on this idea, we implemented DOG (finD cOverage buGs), a Python3-based coverage profiler validator that successfully detects numerous long-standing bugs in Gcov and llvm-cov. For a given coverage profiler, our core idea is to leverage the control flow features inside input programs to obtain the control flow constraints that the resulting coverage statistics should respect. A control flow graph (CFG) serves as the graph representation of a program that describes the control flow between the basic blocks in the program during execution. A lot of constraints about coverage statistics are implied in this representation, with which it is possible to verify the compatibility between coverage statistics of a set of control-flow-related statements even without a direct oracle. For example, within a single basic block, all statements should have the same coverage statistics since executing the block implies passing all its statements. Additionally, for an if-statement with two branches, the execution count of the if-statement should be equal to the sum of its branches' execution counts (as an execution can only pass through one of its branches).

At a high level, given an input program with coverage statistics, DOG proceeds as follows. First, based on the CFG, DOG derives a control dependence graph (CDG) to depict how one conditional statement governs the execution of other statements. Then, with the control dependence in CDG, DOG extracts a set of control flow constraints and checks whether the coverage statistics of relevant statements in the input program are reasonable. In cases where constraints are found to be unsatisfiable, DOG offers detailed diagnostic logs to assist testers in confirming suspicious coverage statistics. Compared with C2V and Cod, DOG has the following prominent advantages. First, DOG is applicable for any single coverage profiler, eliminating the need for multiple profilers or path-equivalent variants as prerequisites. Second, DOG stands out with its distinctive testing capacity, as it takes a fundamentally different perspective in tackling the oracle problem within coverage profiler testing. It can effectively leverage available coverage statistics of input programs to enhance testing capabilities. Third, DOG empowers testers with control flow reasoning, allowing them to interpret why code coverage might be incorrect. This feature greatly facilitates the manual inspection of unsatisfiable constraints.

In sum, this study makes three main contributions:

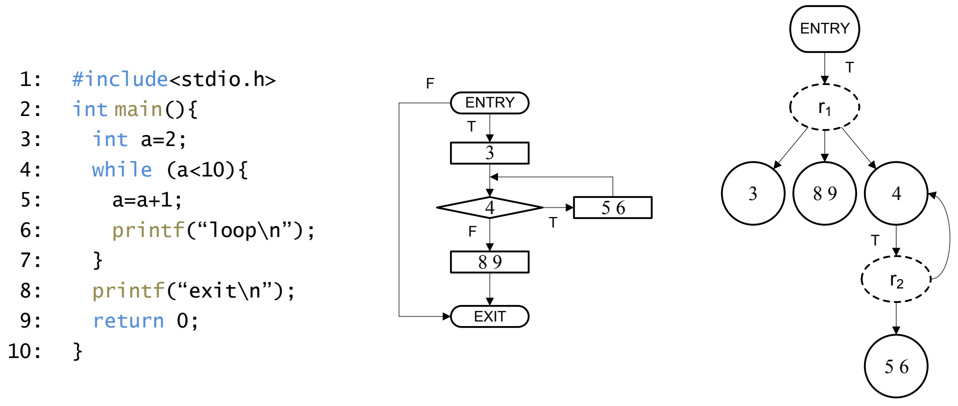
- We propose a brand-new method, control flow constraint solving, which takes a fundamentally different perspective to tackle the oracle problem in coverage profiler testing. This approach has wide applicability, unique testing capacity, and enhanced interpretability.
- Based on control flow constraint solving, we have implemented a coverage profiler validator named DOG in Python3. This tool is open-source and publicly available[1, 2], along with the corresponding datasets, empowering interested researchers to freely reproduce or customize their experiments.
- We revealed 27 longstanding bugs in two widely used and well-tested coverage profilers Gcov and llvm-cov. Among these, 17 bugs got confirmed (11 got fixed), 3 bugs were identified as expected behaviors by developers, and 7 bugs are still pending. Most notably, among 24 bugs revealing unexpected behaviors, 21 had been latent for at least two and a half years by the time we found them, and 10 are completely undetectable by the state-of-the-art coverage profiler validators.

The rest of this paper is organized as follows. Section 2 introduces the preliminaries of coverage profiler and control dependence. Section 3 introduces control flow constraint solving formally and describes the implementation of DOG in detail. Then, our experimental setup and results are respectively presented in Section 4 and Section 5. After that, we discuss the issues in practical applications in Section 6 and possible threats to the validity of our study in Section 7. Finally, Section 8 surveys related works and Section 9 concludes this paper and outlines the direction for future work.

2 Preliminaries

Before formally expounding our method, this section will

[1] https://github.com/NJUocean/DOG
[2] https://zenodo.org/record/8189924



```
 1:  #include<stdio.h>
 2:  int main(){
 3:    int a=2;
 4:    while (a<10){
 5:      a=a+1;
 6:      printf("loop\n");
 7:    }
 8:    printf("exit\n");
 9:    return 0;
10:  }
```

(a) source code (b) CFG (c) CDG

Fig. 1. An illustrating example of source code, CFG, and CDG. (a) is source code, (b) and (c) give the CFG and CDG of the "main" function.

give a brief introduction to related concepts. Among them, the coverage profiler is the source of coverage statistics as well as our test target, and the control dependence plays a pivotal role in generating coverage constraints. In this paper, the related discussion is based on graph representation. When referring to a node, it is referring to the corresponding statement(s).

2.1 Coverage profiler of C/C++

A coverage profiler for C/C++ (such as Gcov) is a tool used to record coverage statistics by collecting runtime information. In general, such a profiler has three steps to collect code coverage at the source level: program instrumentation, data collection, and coverage profiling. First, probes are inserted into the input program, which has no impact on the original execution logic. During the program's execution, these probes will collect the raw coverage data required for the coverage profiler. Finally, with the collected runtime information, the coverage profiler can generate coverage statistics for all instrumented statements. Given a program p and its coverage report, for any statement *stmt* in p, we can obtain its coverage statistics= s according to its corresponding line numbers. $s \geq 0$ indicates that *stmt* is executed s times, while $s = -1$ indicates that the coverage of *stmt* is not recorded.

2.2 Control flow graph

CFG has rich control flow information which is the starting point to control flow constraints. CFG is usually built in the unit of function so we adapt a definition of the function-level CFG from existing literature [26].

Definition 1. (Control Flow Graph) A CFG $G_f = (N_f, E_f, ENTRY, EXIT)$ is a labeled directed graph in which:

— N_f is a set of *content nodes* that represent statements in a function. E_f is a set of edges that represent the control flow between these content nodes;

— N_f is partitioned into two subsets N_f^B and N_f^P. Each node n_b in N_f^B corresponds to statements in a basic block and has only one successor. Each node n_p in N_f^P corresponds to a conditional statement and has at least two distinct successors and attributes "*T*" (true) or "*F*" (false) associated with the outgoing edges;

— the unique start node *ENTRY* is considered the external condition that determines the execution of the function, and it has two distinct successors: the first node in the CFG and the *EXIT* node. The unique stop node *EXIT* has no outgoing edge representing the end of execution;

— for any node $n \in N_f$, n is reachable from *ENTRY* and there exists a path from n to *EXIT*.

For the program presented in Fig. 1(a), Fig. 1(b) gives the CFG of the function "main". In addition, CFG also marks the line numbers of related statements for each node, according to which the coverage statistics of each content node can be obtained from the coverage report.

2.3 Control dependence graph

Control dependence underlies many program analysis and transformation techniques. In this paper, the definition of control dependence to follow is the following classic notion [27].

Definition 2. (Control Dependence) Let x and y be nodes in CFG G_f. y is control-dependent on x iff:

— there exists a directed path from x to y with any z in path (excluding x and y) post-dominated by y;

— x is not post-dominated by y.

If y is control-dependent on x then x must have more than one exit. Following one specific exit E always results in y being executed, while taking other exits may result in y not being executed. Accordingly, one of the control conditions of y' execution is that x takes a particular value such that x exits from E. In Fig. 1(b), node 3 is control-dependent on *ENTRY* and the corresponding control condition is that *ENTRY* takes the value *True*.

However, it is not easy to directly extract control dependence from a CFG. For example, node 8_9 is not control-dependent on node 4 even if it is on the "*F*" branch of node 4, because statements in lines 8 and 9 will be executed no matter what the result of the condition in line 4 is. Moreover, multiple nodes that are control-dependent on the same node may also correspond to different control conditions. Therefore, when dealing with foundational issues of control dependence, researchers often identify control

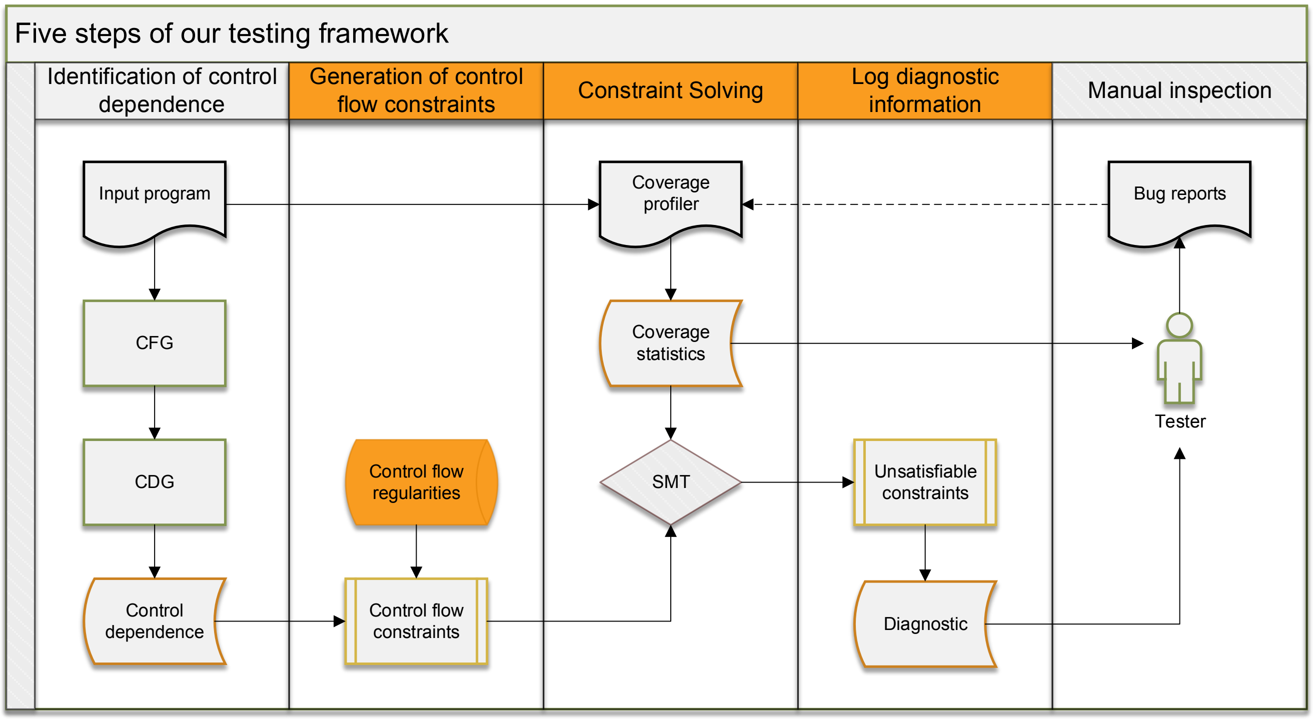

Five steps of our testing framework
Identification of control dependence
Generation of control flow constraints
Constraint Solving
Log diagnostic information
Manual inspection
Input program
CFG
CDG
Control dependence
Control flow regularities
Control flow constraints
Coverage profiler
Coverage statistics
SMT
Unsatisfiable constraints
Diagnostic
Bug reports
Tester

Fig. 2. The framework of control flow constraint solving

dependences from CFG with the assistance of post-dominance and present them in the form of CDG [27]. We follow that practice here and base our presentation on such a definition of the CDG.

Definition 3. (Control Dependence Graph) A CDG G_d = (N_d, R_d, E_d, *ENTRY*) generated from a CFG G_f = (N_f, E_f, *ENTRY*, *EXIT*) is also a labeled directed graph in which:

— N_d is a subset of N_f representing all possible content nodes that can be executed, and if a node is in N_f but not in N_d, then it is a dead node and will not be executed under any conditions;

— R_d is a set of region nodes that have nothing to do with the semantics of the program. They are added to summarize the set of control conditions and group all content nodes with the same set of control conditions together;

— E_d is a set of edges that, together with R_d, represent the control dependence between content nodes: for content nodes X and Y, if Y is control-dependent on X, there exists a region node Z having one incoming edge from X labeled with the associated control condition (“*T*” or “*F*”) and one outcoming edge to Y;

— *ENTRY* is defined in the same way as in G_f, determining the execution of the function, hence it must be identified as the root of G_d. As the end of G_f, *EXIT* is traversed whether the function is executed or not, so it does not involve any control dependence and will not show up in G_d.

From G_d, we can analyze the *control-flow-context* for each content node (i.e., control dependence involving this node) as thoroughly as possible. Fig. 1(c) shows the CDG of function “main” where content nodes are solid while region nodes (i.e., r_1 and r_2) are dotted. Content nodes that are control-dependent on a certain node under the same control condition will be grouped by a region node. Different from content nodes, the coverage of a region node refers to the number of times the corresponding control condition is met and can be obtained from an agent which is a child content node with an in-degree of 1 as such a node is executed only if this control condition is met. For example, the control condition of the execution of node 5_6 is that the predicate expression of node 4 takes the value *True*, and the coverage statistic of r_2 (i.e., the number of times that the predicate expression of node 4 evaluates to *True*) is the same as that of node 5_6. Thus, from Fig. 3(c), we can easily know node 8_9 is control-dependent on *ENTRY* rather than node 4 and has the same control-flow-context as node 3. Both node 8_9 and node 3 can act as agents of r_1, and the agent of r_2 is node 5_6.

3 Approach

In this section, we first show the framework of control flow constraint solving and illustrate how it works in practice with an example bug. Then, we formally introduce the core ideas of our method. Finally, we propose DOG, an automated validator for exposing coverage bugs, and explain the specific details in two algorithms.

3.1 Framework

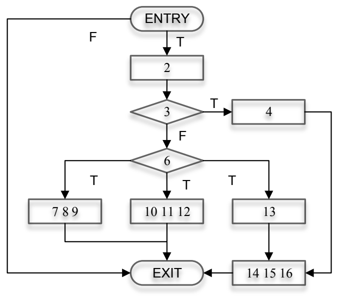
To systematically and effectively expose coverage bugs, we propose a brand-new method called control flow constraint solving for testing coverage profilers. It aims to help developers expand the application scenarios of coverage profiler validation and reduce the human burden of identifying bugs. Fig. 2 shows the framework of our approach,

```
    6: 1:  void foo(int v,int w){
   -1: 2:    int i;
    6: 3:    if (w) {
    2: 4:      goto do_default;
   -1: 5:    }
    4: 6:    switch(v){
    1: 7:      case 0:
    1: 8:        i=27;
    1: 9:        break;
    1:10:      case 1:
    1:11:        i=8;
    1:12:        break;
   -1:13:      default:
    4:14:        do_default:
    4:15:        i=10;
    4:16:        break;
   -1:17:    }
    6:18:  }
     :19:
    1:20:  int main(){
   -1:21:    int i;
    7:22:    for (i=0;i<6;i++){
    6:23:      if (i< 4)
    4:24:        foo(i,0);
   -1:25:      else
    2:26:        foo(i,1);
   -1:27:    }
   -1:28:  }
```

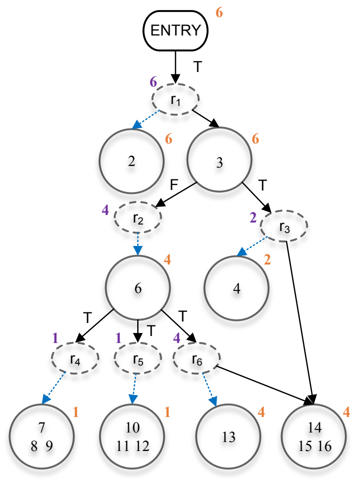
(a) Coverage report by Gcov

```
 1|6|  void foo(int v,int w){
 2|6|    int i;
 3|6|    if (w) {
 4|2|      goto do_default;
 5|2|    }
 6|4|    switch(v){
 7|1|      case 0:
 8|1|        i=27;
 9|1|        break;
10|1|      case 1:
11|1|        i=8;
12|1|        break;
13|4|      default:
14|4|        do_default:
15|4|        i=10;
16|4|        break;
17|4|    }
18|4|  }
19| |
20|1|  int main(){
21|1|    int i;
22|7|    for (i=0;i<6;i++){
23|6|      if (i<4)
24|4|        foo(i,0);
25|2|      else
26|2|        foo(i,1);
27|6|    }
28|1|  }
```

(b) Coverage report by llvm-cov



(c) CFG of “foo”



(d) CDG of “foo”

Fig. 3. An example showing how our method uncovers an llvm-cov bug (#48771) that is not found by existing methods. (a) and (b) are the coverage reports generated by Gcov and llvm-cov for the same input where coverage statistics and line numbers are listed in the first two columns in different order. The coverage of line 13 is wrongly marked as 4 (it is 2 actually) by llvm-cov. (c) is the CFG of function “foo”, and (d) is the CDG derived from (c). The edges from region nodes to agents are also shown as dashed lines.

which consists of the following **five** steps:

Step 1 Identification of control dependence. Given an input program, we focus on its control flow information. By identifying control dependence between content nodes, we derive the CDG for each function from the corresponding CFG.

Step 2 Generation of control flow constraints. With the control dependence, the control flow constraints which describe the relationship between the coverage statistics of graph nodes are generated according to the inherent control flow regularities.

Step 3 Constraint solving. By instantiating a constraint with a given set of coverage statistics, we can encode it as an SMT problem instance. Therefore, the satisfiability of control flow constraints can be checked with the help of an SMT solver.

Step 4 Logging diagnostic information. Each unsatisfiable constraint can correspond to a potential coverage bug. To facilitate manual inspection, we log diagnostic information which includes the coverage statistics of the nodes involved in the solving process and the relationship between them.

Step 5 Manual inspection. Facing a large number of unsatisfiable constraints that may point to the same underlying bug, testers confirm the oracle, deduplicate them with the help of diagnostics, and then report these confirmed and deduplicated potential bugs to developers. Only in this step, manual intervention is required.

3.2 Illustrating example

In the whole approach mentioned above, Steps 2 to 4 embody the core idea of control flow constraint solving, which will be illustrated using a bug example. Fig.3 shows how our method uncovers an llvm-cov bug (#48771 [53]) that is not found by existing methods. Fig. 3(a) and Fig. 3(b) are respectively the coverage report generated by Gcov and llvm-cov. They annotate each line of the original program with a line number and the corresponding coverage statistic in a different format. As highlighted in Fig. 3(b), llvm-cov wrongly reported that line 13 was executed 4 times. This bug is missed by existing methods since Gcov does not provide the corresponding coverage statistic of line 13 for C2V to compare and there are no unexecuted statements for Cod to remove. However, DOG can uncover it by analyzing control dependence. Fig. 3(c) and Fig. 3(d) give the CFG and CDG of the function “foo”, respectively. For the convenience of presentation, we mark the coverage statistics for each node in Fig. 3(d). The numbers in the upper-right corner of the content nodes are the coverage statistics of those nodes, referring to the executions of related statements, which can be obtained from coverage reports directly. The numbers in the upper-left corner of the region nodes are the coverage statistics of the region nodes. Additionally, we mark the edges from region nodes to their agents as dashed lines. For an object x (which can be a statement or a node), we use *Cov(x)* to represent its coverage statistics. Thus, the process of revealing this bug through analyzing the control-flow-context around node 6 can be presented as the following steps:

Generation of control flow constraints. Given the CDG of function "foo", we generate control flow constraints according to the specified semantics. As can be seen, node 6, which corresponds to a switch-statement with a default label on the sixth line, has one father region node (i.e., r_2) and three child region nodes (i.e., r_4, r_5, and r_6). Any result of node 6 will cause a case node to be passed. Therefore, we have the following control flow constraint:

$$Cov(node\ 6) = Cov(r_4) + Cov(r_5) + Cov(r_6) \quad (1)$$

Constraint solving. To check the satisfiability of Eq. (1), we determine the coverage statistics of each node with the coverage statistics and agency relationship. Accordingly, we know that:

$$Cov(node\ 6) = 4 \quad (2)$$

$$Cov(r_4) = Cov(node\ 7\_8\_9) = 1 \quad (3)$$

$$Cov(r_5) = Cov(node\ 10\_11\_12) = 1 \quad (4)$$

$$Cov(r_6) = Cov(node\ 13) = 4 \quad (5)$$

Considering Eq. (2) to (5), the relevant coverage statistics obviously do not satisfy the constraint represented by Eq. (1) (this checking procedure can be achieved by being converted into an SMT-solving problem instance, which will be introduced in Section 3.3.3).

Logging Diagnostic Information. Among nodes 6, r_4, r_5, and r_6, we will try to further infer the most suspicious nodes by checking whether their coverage statistics cause other constraints to be unsatisfiable. As a result, only r_6 is involved in another unsatisfiable constraint:

$$Cov(node\ 14\_15\_16) = Cov(r_3) + Cov(r_6) \quad (6)$$

$$Cov(node\ 14\_15\_16) = 4 \quad (7)$$

$$Cov(r_3) = Cov(node\ 4) = 2 \quad (8)$$

$$Cov(r_6) = Cov(node\ 13) = 4 \quad (9)$$

As a result, it is node 13, the agent of r_6, whose coverage statistic results in two control flow constraints (Eq. (1) and Eq. (6)) to be unsatisfiable. Therefore, node 13 is blamed as the most suspicious. Finally, the diagnostic information shown in Fig. 5 will be logged for testers to understand, classify, and report the bug (details of diagnosis are shown in Section 3.2.4).

3.3 control flow constraint solving

This subsection introduces in detail control flow constraint solving in four parts: insights, control flow regularities, constraint solving, and diagnosis strategy.

3.3.1 Insights

In the process of validating coverage statistics, programmers are required to analyze executions of statements from scratch along the control flow. When the control conditions are ambiguous, testers of developers need to analyze which coverage statistic is most likely to be wrong and even insert counting statements to determine the oracle. By tracing the control flow in the CFG and CDG of real programs, we obtain the following three main observations:

- Insight 1: *Control conditions determine whether the controlled statements are executed.*

  A statement is executed only when the corresponding control condition is met. As a result, statements whose execution dependents on the same control conditions will share the same coverage statistics, but this doesn't necessarily hold in reverse.

- Insight 2: *Control flow is traceable.*

  In a CFG without unexpected exits, the control flow must consistently traverse from the *ENTRY* point to the *EXIT* point, and it does not arbitrarily increase or decrease without valid reasons. Consequently, we can observe that control flow enters and exits a node an equal number of times.

- Insight 3: *The coverage statistics involved in more than one unsatisfiable constraint are more likely to be incorrect.*

  The coverage statistics of a single node can be involved in the solving of different control flow constraints. Consequently, if an individual coverage statistic is incorrect, it can render more than one control flow constraint unsatisfiable simultaneously. That is to say, wrong coverage statistics are more likely to be perceived from different perspectives.

Based on the above insights, we propose the core concept of control flow constraint solving to automate the manual verification of coverage statistics as much as possible. We summarize six control flow regularities according to Insight 1 and Insight 2 and heuristically locate the incorrect coverage statistics with Insight 3, which will be elaborated in detail in the following sections.

3.3.2 Control flow regularities

Given a program p, through program analysis, we can not only obtain the control flow information within each function but also the call relationships between functions. Based on this information as well as insight 1 and insight 2, Table 1 briefly summarizes six control flow regularities, each serving different objectives. The regularity SB targets the consistency of coverage statistics of statements within a specific content node. The regularity SF depicts the consistency of coverage statistics of content nodes that share the same father region nodes. The regularities IL and ON concentrate on the control flows into and out of content nodes. The last two regularities FCL and FEL focus on the relationship between the coverage statistics of functions and function calls/exits.

The detailed specifications of these six control flow regularities are as follows:

SB (Same-Block). If a content node n corresponds to k statements ($stmt_i$, $1 \leqslant i \leqslant k$) in the source program, we have:

TABLE 1
CONTROL FLOW REGULARITIES

Name	Full Name	Object	Description
SB	Same-Block	Any content node *n*	Statements related to *n* should have consistent coverage statistics.
SF	Same-Fraternity	Content nodes having the same parents in CDG	Nodes sharing the same father region nodes have the same control conditions and thus have the same coverage statistics.
IL	Inflow-Lossless	Any content node *n*	The coverage statistic of a content node is equal to the sum of the coverage statistics of all its father region nodes.
ON	Outflow-Nonincreasing	Any content node *n*	The coverage statistic of a content node is greater than or equal to the sum of the coverage statistics of all its child region nodes.
FCL	Function-Call-Lossless	Any function f_x	A function executes as many times as it is called.
FEL	Function-Exit-Lossless	Any function f_x	A function exits as many times as it is executed.

$$Cov(n) = Cov(stmt_1) = \cdots = Cov(stmt_k)$$

A content node corresponds to adjacent statements within a single basic block or a conditional statement. Therefore, they should have consistent coverage statistics. Satisfying the regularity SB is also the premise of obtaining coverage statistics for each content node from the coverage report. For example, in Fig. 3(c), associated statements of each content node have consistent coverage statistics so that the node coverage can be obtained.

SF (Same-Fraternity). If there are *m* content nodes (n_i, $1 \leqslant i \leqslant m$) sharing the same parents in CDG, we have:

$$Cov(n_1) = \cdots = Cov(n_m)$$

The parent-child relationship in CDG exists only between a region node and its father/child content nodes, representing control dependence between these content nodes. We also define a brother relationship to capture the peer relationships between content nodes: for any two content nodes, if their execution depends on the same control conditions, they are considered brother nodes to each other and will have the same number of executions regardless of the order in the source program. A node and all its brothers make up a fraternity and all fraternities are mutually exclusive. Since the region nodes summarize the control conditions, brother nodes should have the same father region nodes in CDG. For example, in Fig. 3(d), node 2 and node 3 are brother nodes forming a fraternity of size two. However, node 4 and node 14_15_16 are not brother nodes because node 14_15_16 has one more parent region node r_6 which means that node 14_15_16 can be executed under more control conditions (the predicate expression of node 3 takes the value *True* and the default-case of node 4 is caught) than node 4 (the predicate expression of node 3 takes the value *True*).

IL (Inflow-Lossless). If content node *n* has *a* father region nodes ($F_i(n)$, $1 \leqslant i \leqslant a$) in CDG, we have:

$$Cov(n) = \sum_{i=1}^{a} Cov(F_i(n))$$

In the perspective of control dependence, the control flow into a content node comes from those content nodes it is control-dependent on and not from the father nodes in CFG. Every time a content node is executed, there is one corresponding control condition, on which it depends, being met. Since the coverage statistics of region nodes record the number of times each control condition is met, *Cov*(*n*) should be equal to the sum of the coverage statistics of all its father region nodes. For example, Eq. (6) in Section 3.2 is a control flow constraint of type IL.

ON (Outflow-Nonincreasing). If content node *n* has *b* child region nodes ($C_j(n)$, $1 \leqslant j \leqslant b$) in CDG, we have (by insight 2):

$$Cov(n) \geq \sum_{j=1}^{b} Cov\left(C_j(n)\right)$$

If *n* controls the execution of other content nodes, those nodes may get control flow from *n* and be executed. For example, Eq. (1) in Section 3.2 is generated according to ON. However, there is no control dependence between a content node and conditional branches sometimes. As there is no suitable example in Fig. 3, let us return to Fig. 1. As stated in Section 2.2, node 8_9 on the "F" branch of node 4 is control-dependent on *ENTRY* instead of node 4. Node 4 only governs the execution of node 5_6 on the "T" branch. Therefore, in the perspective of control dependence, we adopt a similar expression as IL, but with inequalities. The exact form of control flow constraints generated depends on the control-flow-context of *n* (e.g., for a switch-node, the equal sign can only be taken when controlling the execution of a default-case (like node 6 in Fig. 3(c)) because any result of the switch-node will cause a case to be executed).

FCL (Function-Call-Lossless). For any function f_x, we have:

$$Called(f_x) = Cov(f_x)$$

Called(f_x) is the number of times f_x is called (it defaults to 1 for function "main") which can be calculated by summing the coverage statistics of the call sites. *Cov*(f_x) is the number of function executions which is given in the coverage report on the line of the function declaration. Take the function "foo" in Fig. 3 as an example. "foo" is called in lines 24 and 26 of the source code so that *Called*(*foo*) is the sum of *Cov*(*24*) and *Cov*(*26*). Besides, the coverage of "foo" is given in line 1 so that *Cov*(f_x) is 6.

FEL (Function-Exit-Lossless). For any function f_x, we have:

```
 1  from z3 import *
 2  s = Solver()
 3  n = Int('n') # node 6
 4  cn = IntVector('cn', 3) # r4, r5, and r6
 5  s.add(Sum([c for c in cn]) == n) # Eq.(1)
 6  s.add(n == 4) #  Eq.(2)
 7  s.add(cn[0] == 1) #  Eq.(3)
 8  s.add(cn[1] == 1) #  Eq.(4)
 9  s.add(cn[2] == 4) #  Eq.(5)
10  res = str(s.check())
```

Fig. 4. A sample code snippet showing how to encode Eq. (1) to (5) in Section 3.2 to an SMT problem instance in Python3 with z3.

$$Cov(f_x) = Exit(f_x)$$

Cov(f_x) is the same as introduced in FCL. Similarly, *Exit*(f_x) is the number of times a function exits and can be obtained by analyzing the coverage statistics of potential exits of f_x. Potential exits are some content nodes post-dominated by *EXIT* directly. Also, take the function "foo" in Fig. 3 as an example, it can exit directly from three content nodes (node 7_8_9, node 10_11_12, and node 14_15_16) which governs no other nodes so that *Exit*(f_x) is equal to the sum of coverage statistics of them, i.e., 6.

3.3.3 Constraint solving

With control flow regularities, we can generate six types of control flow constraints according to control dependence and function call relationships. Thus, we can solve these constraints one by one to hunt coverage bugs by checking whether associated coverage statistics are reasonable. In this way, we convert the task of coverage profiler testing into a set of local SMT-solving problems to verify the satisfiability of control flow constraints. Specifically, the constraint-solving process of a function is progressive. Satisfying the regularity SB is considered the basis to obtain coverage statistics for content nodes, so we first take one round of traversal to solve SB-constraints. Next, we traverse all fraternities to solve SF-constraints and choose appropriate agents for region nodes. With available coverage statistics of graph nodes, the IL-constraint and ON-constraint are usually solved individually for each content node *n*, but if the coverage statistic of *n* is not available, we combine the two constraints as a whole, where *n* only acts as a bridge to connect its father region nodes and child region nodes. After processing all the content nodes, we finally focus on the function itself to solve its FCL-constraint and FEL-constraint.

In practice, for any control flow constraint, all nodes involved are referred to as its *players*, and we can leverage the coverage statistics of all its players to construct an SMT problem instance to conduct constraint solving. For example, the control flow constraint represented by Eq. (1) in Section 3.2 has four players nodes 6, r_4, r_5, and r_6. Fig. 4 shows a sample code snippet of how to construct an SMT problem according to Eq. (1) to (5) in Python3 with z3 [68]. After initializing an SMT solver *s* (Line 2), we create an integer variable *n* (Line 3) and an integer array variable *cn* of length 3 (Line 4) representing node 6 as well as its three

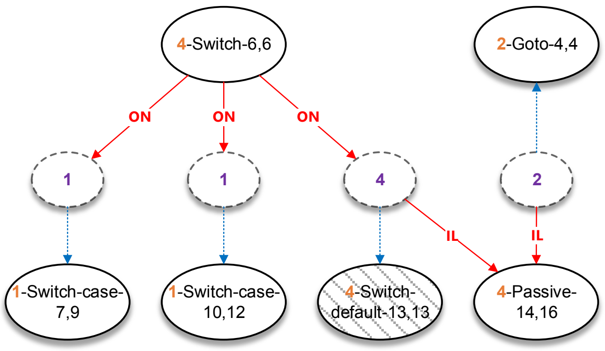


Fig. 5. Graphic diagnostic information of llvm-cov bug #48771 which describes two unsatisfiable constraints in function "foo". It is derived based on the CDG. For each region node, we record its coverage statistic. For each content node, we record its coverage statistic, semantic type, and the start and end line numbers of the related statements.

child region nodes r_4, r_5, r_6. These variables can be associated and assigned values according to the control flow constraint (Line 5) and coverage statistics (Lines 6-9). Finally, *s.check()* is called to obtain the result *res* which is a string variable "unsat" (Line 10).

3.3.4 Diagnosis strategy

Considering that the six regularities are general ground features that the coverage statistics should satisfy, once any control flow constraint is unsatisfiable, on the premise that the form of constraint is correct, a potential coverage bug is uncovered, and all players are suspicious until the true *suspects* (i.e., the nodes with incorrect coverage statistics) are identified. To determine the suspects as well as their oracle, it is necessary to understand the *context* (i.e., who the players are, how they relate to each other concerning control dependence as well as their coverage statistics) of potential bugs, so we provide detailed diagnostic information for testers.

It is worth noting that the difficulty of confirming suspects is highly dependent on the complexity of the context, and we make a slight difference between the different types of constraints. An SB-constraint has only one content node, and if it is unsatisfiable, the node must be the suspect and the corresponding suspicious coverage statistics are easy to confirm by examining the adjacent statements. As for FCL-constraints, they usually refer to the interactions between functions so that corresponding unsatisfiable constraints need to be inspected manually with the function call relationships. Therefore, for these two types of unsatisfiable constraints, we prefer to just log diagnostic information and leave them to human hands. The rest four regularities SF, IL, ON, and FEL, focusing on the compatibility of coverage statistics of multiple nodes in a function, are called *compatibility-related regularities*. For the simplicity of the presentation, in the following, we use *compatibility-related constraints* and *compatibility-related bugs* to denote constraints and potential bugs associated with compatibility-related regularities. Since the coverage statistics of players of different compatibility-related constraints within the same function can intersect, we can perform heuristic inference of suspects according to insight 3: a node whose coverage statistic is involved in more than one unsatisfiable constraint (either directly or indirectly through agency relationship) is more likely to be a suspect.

In practice, graphics can better convey information than text. As shown in Fig. 5, the graphic diagnostic information of the illustrating bug (llvm-cov bug #48771 [53]) visualizes its context, which is related to two unsatisfiable compatibility-related constraints. It is derived based on the CDG, where the content nodes are more informative (connected by dashes are coverage statistics, semantic type, and the start and end line numbers), and the types of unsatisfiable constraints (i.e., ON and IL) are marked on the related edges. As stated in Section 3.2, two unsatisfiable constraints have a shared player whose agent node is accused as a suspect and highlighted with a shadow.

3.4 DOG

Based on control flow constraint solving, we have engineered DOG, an automated validator for coverage profilers. This subsection details the underlying procedures of DOG in Algorithm 1 and Algorithm 2 where Algorithm 1 presents the parameterized pseudocode of DOG and Algorithm 2 presents the implementation of the constraint solving procedure. Note that any italicized "function" in Algorithm 1 and Algorithm 2 represent abstract procedures rather than actual functions. In addition to being explicitly initialized as a set or dictionary, any variable is an abstract object and is not instantiated by any specific class.

Algorithm 1. The main procedure (Line 1) takes as inputs a coverage profiler under test, *profiler,* and an input program, *program*, and finally outputs the diagnostics of all potential bugs inside *program*. To perform constraint solving, we need to obtain the necessary information from *program*. Line 2 leverages *profiler* to generate the coverage report *crp* for *program* which logs the coverage statistics of each code line. Line 3 performs static analysis of *program* to parse control flow information *GF* containing the CFG of each function as well as the calling relationships. Line 4 adopts the Lengauer-Tarjan algorithm [28] to compute dominators to construct the post-dominator tree (PDT) *TP* capturing the post-dominance relationships in each function. Finally, with *GF* and *TP*, *GD* can be built using the corresponding algorithm [27], from which the CDG of each function can be easily obtained (Line 5). The set *PB* is used to collect the context of all potential bugs in *program* and is initialized to an empty set (Line 6). Line 7 initializes an empty dictionary *NC* to save coverage statistics of graph nodes. Lines 8-39 loop in function units to solve control flow constraints. Control flow constraints on different objects are solved by calling the function *Solving* (Lines 14, 20, 28, 31, 34, and 37). If any result is *UNSAT*, then the diagnostic of the unsatisfiable constraint is added to *PB* (Lines 16, 22, 30, 33, 36, and 39). Lines 9-10 obtain the CFG and CDG of the current function. Lines 11-18 iterate over all nodes in GD_f, initialize their coverage statistics to -1(i.e., unavailable), and further computes the coverage statistics of the content nodes by checking whether each content node satisfies SB regularity. Then, since the brother relation is mutual, we solve SF-constraints in units of fraternity to avoid redundant computation and determine agents

Algorithm 1 DOG's pseudocode

```
1   procedure DOG(profiler, program):
2       crp ← GenerateCoverage(profiler, program)
3       GF ← StaticAnalyze(program)
4       TP ← Lengauer-Tarjan(CFG)
5       GD ←DependenceAnalyze(CFG, TP)
6       PB ← Ø /* potential bugs */
7       NC ← Dict() /* coverage statistics of  graph nodes */
8       foreach function f in program do
9           $GF_f$ ← ObtainCFG(GF, f)
10          $GD_f$ ←ObtainCDG(GD, f)
            /* check SB regularity and get node coverage */
11          foreach node n in $GD_f$ do
12              NC[n] ← -1
13              if n is a content node then
14                  res, diag ← Solving(SB, n, $GD_f$, NC)
15                  if res = UNSAT then
16                      PB.add(diag)
17                  else
18                      NC[n] ← ComputeCov(n, crp)
            /* solve SF-constraints for fraternities */
19          foreach fraternity ft in $GD_f$. do
20              res, diag ← Solving(SF, ft, $GD_f$, NC)
21              if res = UNSAT then
22                  PB.add(diag)
23              elif ft has a single father region node fr then
24                  if there is a node t in ft and NC[t] ≠ -1 then
25                      The agent of fr ← t
26                      NC[fr] ← NC[t]
            /* solve IL/ON-constraints for content nodes */
27          foreach content node n in $GF_f$ do
28              res, diag ← Solving(IL, n, $GD_f$, NC)
29              if res = UNSAT then
30                  PB.add(diag)
31              res, diag ← Solving(ON, n, $GD_f$, NC)
32              if res = UNSAT then
33                  PB.add(diag)
            /* solve FCL/FEL-constraints for function */
34          res, diag ← Solving(FCL, f, $GD_f$, NC)
35          if res = UNSAT then
36              PB.add(diag)
37          res, diag ← Solving(FEL, f, $GD_f$, NC)
38          if res = UNSAT then
39              PB.add(diag)
40      InferAndLog(PB)
```

and coverage statistics of region nodes (Lines 19-26). Next, DOG further traverses all content nodes to solve IL-constraints and ON-constraints (Lines 27-33). Afterward, the FCL-constraint and FEL-constraint of the current function can be solved (Lines 34-39). Finally, DOG logs diagnostics of all potential bugs after inferring the suspects (Lines 40).

Algorithm 2. The function *Solving* (Line 1) performs constraint solving based on the type of regularity, object, CDG, and the coverage statistics of graph nodes and returns *result*, the result of constraint solving, as well as related diagnostic information *diagnostic*, if any (Lines 11 and 41). We default that the constraint is satisfiable, so *result* and *diagnostic* are initialized to *SAT* and *None* respectively (Lines 2-

3). Line 4 initializes an empty set *P* to collect all the players involved in the solving procedure, which can be determined according to *regularity* and *object* (Lines 5-28). For regularity SB, *object* is a single content node and there are no other players so we can directly check the consistency of coverage statistics of related statements (Lines 6-11); for regularity SF, *object* is a fraternity, and all nodes inside are players (Lines 12-14); for regularities IL/ON (Lines 15-22), *object* is a content node, the players also include the agents of all the father/child region nodes of it; for regularities FCL/FEL (Lines 23-28), *object* is a function and the players are the *ENTRY* node and its callers/exits. The *ENTRY* node can represent the function itself, the callers are some content nodes from the other function calling *object*, and the exits are those content nodes that can go directly to the *EXIT* node. Next, we initial an SMT solver *solver* and create an integer variable for each player, whose values depend on the corresponding coverage statistics (Lines 29-35). Then, we generate the control flow constraint *constraint* which can associate these variables together to finish the construction of the SMT problem instance. (Lines 36-37). If the result *result* is unsatisfiable, the context of the current potential bug will be extracted as diagnostic information *diagnostic* (Lines 38-40). Finally, both *result* and *diagnostic* will be returned (Line 41).

Algorithm 2 Constraint Solving

```
1  function Solving(regularity, object, cdg, cov):
2      result ← SAT
3      diagnostic ← None
4      P ← Ø /* players */
5      switch regularity do
6          case SB do /* object is a content node */
7              P.add(object)
8              if coverage statistics are inconsistent then
9                  result ← UNSAT
10                 diagnostic ← Extract (cdg, cov, P, regularity)
11             return result, diagnostic
12         case SF do /* object is a fraternity */
13             foreach content node n in object do
14                 P.add(n)
15         case IL do /* object is a content node */
16             foreach father region node frn of object in cdg do
17                 P.add(agent of frn)
18             P.add(object)
19         case ON do /* object is a content node */
20             foreach child region node crn of object in cdg do
21                 P.add(agent of crn)
22             P.add(object)
23         case FCL do /* object is a function */
24             P.add(ENTRY of cdg)
25             P.add(callers of object)
26         case FEL do /* object is a function */
27             P.add(ENTRY of cdg)
28             P.add(exits of object)
29     solver ← InitializeSolver()
30     foreach play p in P do
31         v_p ← IntegerVariable()
32         if cov[p] ≠ -1 then
33             solver.addConstraint("v_p = cov[p]")
34         else
35             solver.addConstraint("v_p ≥ 0")
36     constraint ← ConstrctConstraint(P, regularity)
37     solver.addConstraint(constraint)
38     if solver.check() is unsatisfiable then
39         result ← UNSAT
40         diagnostic ← Extract (cdg, cov, P, regularity)
41     return result, diagnostic
```

4 Experimental Setup

In this section, we describe in detail the experimental setup. First, we list the research questions under investigation. Then, we introduce the experimental environment, subject coverage profilers under test, and the tools used in our study. After that, we characterize the test suite and present the preprocessing approach. Finally, we introduce our bug reduction strategy.

4.1 Research questions

In this paper, we study the following research questions:

RQ1 (Effectiveness of bug finding): *Is DOG competent and effective in exposing coverage bugs?*

The purpose of RQ1 is to investigate whether DOG is effective to validate real coverage profilers. Our original intention is to propose a general and effective testing method, so it is expected that DOG applies to both hand-written and randomly generated test programs and can uncover new real coverage bugs in different independently developed coverage profilers. The answer to RQ1 enables us to understand whether it is worthwhile to use our new method in practice.

To answer RQ1, we assess the effectiveness of DOG by measuring the ability to uncover real bugs in different coverage profilers. For one thing, we leverage both handwritten and randomly generated input programs to reveal new coverage bugs; for another, we reproduce coverage bugs from the code snippets in the bug reports reported by the existing methods to examine the ability of DOG to detect coverage bugs in the earlier versions.

RQ2 (Significance of uncovered bugs): *How significant are the bug-finding results?*

The purpose of RQ2 is to investigate whether our testing work makes sense since meaningful bug-finding results (e.g., long-latent and hard-to-find bugs) are more important for the quality assurance of coverage profilers than bugs that are not intended to be fixed. The answer to RQ2 enables us to understand the uniqueness of DOG's detection capabilities.

To answer RQ2, we evaluate the significance of these bugs found by DOG from two aspects: cross-version lifecycle and uniqueness. For lifecycle, we wonder when these bugs are introduced and whether they exist in the subsequent versions. Therefore, we select nine releases for each subject coverage profiler to observe how the incorrect coverage statistics change from version to version. For uniqueness, we wonder if DOG has a unique capability to reveal hard-to-find bugs, so we investigate whether our new bugs can be detected by the state-of-the-art coverage profiler validators C2V and Cod.

RQ3 (Usefulness of Diagnostics) *How useful are diagnostics for facilitating manual inspection?*

The purpose of RQ3 is to investigate whether our diagnostic is useful in facilitating manual inspection. We mainly focus on potential compatibility-related bugs as they only involve different nodes within a single function, apply to our heuristic inference strategy, and account for the vast majority (84.33% and 95.5% of the potential bugs found in manual input programs and random input programs). Their diagnostics are expected to not only visually describe the context of potential compatibility-related bugs concisely but also point out the specific incorrect coverage statistics as accurately as possible. The answer to RQ3 enables us to understand to what extent diagnostics help testers reduce manual overhead.

To answer RQ3, we measure the usefulness of the diagnostic of potential compatibility-related bugs from two dimensions: reduction of manual overhead and hit rate of inference. For the former, we calculate the ratio of the size of the diagnostic (measured as the number of associated nodes or related lines of code) to that of the whole function. For the latter, we manually check the innocence of all suspects inferred and calculate the percentage of them that does involve incorrect coverage statistics.

RQ4 (Contribution of Regularities): *How do these six types of regularities contribute to the bug-finding results?*

The purpose of RQ4 is to investigate how these regularities work in uncovering real coverage bugs. The regularities we summarize are general ground features that the coverage statistics should satisfy. Constraints from them reveal potential bugs from different perspectives of control flow and infer suspects cooperatively. The answer to RQ4 enables us to know the patterns in which regularities work so that testers can be inspired to adjust the testing strategy according to specific testing goals in the future.

To answer RQ4, we understand their contribution by analyzing the extent of their involvement in coverage profiler testing. For this purpose, we construct a bug database containing these newly uncovered coverage bugs discovered by DOG and those revealed by existing methods in the earlier versions. Accordingly, we determine the set of bugs each regularity can individually detect by only solving the constraints that come from it. In addition, we also show the contribution of compatibility-related regularities to heuristics inference by analyzing the intersection of bugs that can be detected by different compatibility-related constraints.

4.2 Hardware and subjects

In this paper, our evaluation was conducted on a Linux server (running on 64-bit Ubuntu 18.04.6 with Linux kernel 4.15.0-210-generic) with Intel(R) Xeon(R) Gold 5117 @2.00GHz (48 cores) and 128GB RAM. We choose as subjects the two most popular C code coverage profilers Gcov-10.2.0 and llvm-cov-11.0.0, which are the latest release version at the time when our study started at the end of 2020. They are adopted for the following reasons:

- They have been widely used in the community of software engineering.
- They are integrated into well-known production compilers, i.e. GCC and Clang, respectively.
- They are used as the subject profilers in prior studies and thus we can easily and fairly make a comparison with previous approaches.

Therefore, it is of great importance to uncover new bugs in these two widely used and well-tested coverage profilers

4.3 Third-party tools support

Based on control flow constraint solving, we implemented DOG, a coverage profiler validator, in Python3. Just like any other programmers, we adopt some mainstream Python3 libraries like os [66] (for interacting with the operating system) and re [67] (for regular expression operations) to support our functions. At a high level, the framework of our method can be divided into five steps[3], three of which are supported by the following third-party tools: Understand [29] (for Step 1), Z3 [30] (for Step 3), and Graphviz [64] (for Step 4).

- **Understand**. Understand is a static analysis tool for maintaining, measuring, and analyzing critical or large code bases. Our static analysis of the input programs is implemented based on Understand. With its Python API, we can get the control flow information of the functions and the call relationship between the functions.
- **Z3**. Z3 is a high-performance theorem prover that has been wildly applied in a variety of SMT tasks. We leverage the *z3* [68] package in Python3 to perform constraint solving. The satisfiability of control flow constraints can be checked by converting them to SMT problem instances as stated before.
- **Graphviz.** Graphviz is an open-source visual graphics tool from AT&T Research and Lucent Bell Laboratories. We turn to the corresponding Python3 package, graphviz [65], to draw pictures with Python3 syntax for visualization of diagnostics.

4.4 Test programs

We adopt the same test scenario as the existing works, each time checking the irrationality in the coverage statistics of a single program offered by the subject coverage profiler. In terms of test programs selection, we first look for the appropriate input program from the test suite that comes with the release of GCC 10.2.0 which is a manual test suite for GCC compiler testing. There is a total of 38540 C programs in the subject test suite containing rich program semantics without undefined behaviors. And then, before performing constraint solving, we go through the following steps to filter out inappropriate programs and obtain coverage statistics and control flow information:

Step 1 Code formatting. Given that many of these manual programs have inconsistent code styles, we format them into a unified LLVM style so that there will be no more than one statement per line and we can obtain statement

[3] As stated in Section 3.1, the five steps of our testing framework are: Identification of control dependence, Generation of control flow constraint, Constraint solving, Log diagnostic information, and Manual inspection

coverage statistics from line numbers more accurately.

Step 2 Coverage generation. Given any program *test.c*, we tried to generate coverage reports *test.c.gcov* [4] and *test.c.lcov*[5] for it using Gcov and llvm-cov, respectively. Programs that fail to obtain a valid coverage report in 5s, without additional inputs or files, will be discarded.

Step 3 Static analysis. For each test program, we leverage Understand to analyze its control flow information as well as function call relationships. This process can be integrated into the testing process, but for the convenience of reproducing, in advance, we save the information of CFG and function call relationships in the form of files as the input of DOG.

Step 4 Subjective filtration. In practice, we assume that functions exist independently of each other and interact through function calls, so we filter those inputs that define functions inside other functions, or jump between functions through "setjmp" and "longjmp". Besides, programs containing goto-statement whose target label is specified dynamically were also skipped. To facilitate the manual inspection, we also limit the size of programs: (1) no more than 5 defined functions, and (2) no more than 100 lines in source code.

As a result, we finally get 4163 manual programs as our manual test suite. In addition, following previous studies [22], [23], we use Csmith [52] to generate 1000 random programs as a supplement[6]. The semantics of random programs are statically given and they can be correctly parsed. Accordingly, these random programs do not need to be subjectively filtered.

4.5 Bug deduplication

During testing, we encountered many kinds of programs which have at least one potential bug with wrong coverage statistics. In our opinion, it would be irresponsible to submit all of them to developers for confirmation. Dozens of potential bugs may point to the same underlying reason, so, as testers, we should try our best to use our cognition to deduplicate them before reporting.

To reduce the difficulty of bug deduplication, we adopt a two-step classification approach. This categorization aids testers in manually deduplicating bugs within different categories. In the classification process, we refer to two features, one is the type of control flow regularities violated by the potential bugs, and the other is the type of incorrect coverage statistics of the suspects. In the existing works [22], [23], incorrect coverage statistics are distinguished into three types (*Spurious Marking*[7], *Missing Marking*[8], *and Wrong Frequency*[9]), and we inherited the same classification in this paper. After that, testers can deduplicate those potential bugs under the same classification according to the similarity of the context.

TABLE 2
REAL STATUS OF COVERAGE BUGS

Status	Gcov	llvm-cov	Total
Reported	13	14	27
Confirmed	7	10	17
Fixed	1	10	11
Pending	3	4	7
Expected	3	0	3
Duplicate	0	0	0
Won't fix	0	0	0

TABLE 3
TYPES OF INCORRECT COVERAGE STATISTICS

Type	Gcov	llvm-cov	Total
Spurious Marking	1	7	8
Missing Marking	2	1	3
Wrong Frequency	10	6	16

5 EXPERIMENTAL RESULTS

In this section, we will present and analyze our experimental results to answer the proposed research questions.

5.1 RQ1: Effectiveness of bug finding

For Gcov, DOG reports respectively 194 and 48 unsatisfiable constraints in 117 manual programs and random programs. For llvm-cov, DOG reports respectively 272 and 120 unsatisfiable constraints in 120 manual programs and 126 random programs. After de-duplication, we reported 27 potential coverage bugs, 13 in Gcov and 14 in llvm-cov, revealing unusual coverage statistics. Table 2 shows the bug status of all the potential coverage bugs we reported. 17 have been confirmed by developers, 7 are still pending, and 3 were categorized as expected because the anomalous coverage statistics found are expected behaviors that the mechanisms of the coverage profilers could not handle. Table 3 summarizes the types of incorrect coverage statistics of these potential bugs. The potential bugs of both Gcov and llvm-cov cover all three types where the most frequent are *Wrong frequency* (16 out of 27) followed by *Suprious marking* (8 out of 27) and *Missing marking* (3 out of 27).

Table 4 details all the filed bug reports including the subject profiler, the bug ID, the priority of bugs, the type of incorrect coverage, the status of bug reports, and the real status of coverage bugs. All these coverage bugs can be searched with the corresponding bug ID. For Gcov bugs that are reported in Bugzilla, "Priority" indicates the priority that the developer plans to fix the bug, with P1 being the highest, P5 the lowest. All the Gcov bugs are labeled with the default priority P3. For llvm-cov bugs which are managed in GitHub, there are no corresponding labels of

[4] *gcc -O0 –coverage test.c –o test; ./test;gcov test*
[5] *clang -O0 -fcoverage-mapping -fprofile-instr-generate=test.profraw test.c -o test; ./test; llvm-profdata merge test.profraw -o test.profdata; llvm-cov show -instr-profile=test.profdata ./file > test.c.lcov*
[6] *--concise --max-struct-fields 5 --max-funcs 2 --max-array-len-per-dim 5 --max-block-depth 3 --max-block-size 2*
[7] *Spurious Marking*: unexecuted statements are marked as executed.
[8] *Missing Marking:* executed statements are marked as unexecuted.
[9] *Wrong Frequency*: a statement executed m times is wrongly marked as executed n times where $m > 0$, $n > 0$, and $m \neq n$.

TABLE 4
COVERAGE BUGS FOUND BY DOG

	Profiler	ID	Priority	Type	Report Status	Bug Status
1	Gcov	99440	P3	Wrong	NEW	Confirmed
2	Gcov	99441	P3	Wrong	NEW	Confirmed
3	Gcov	99442	P3	Missing	RESOLVED INVALID	Expected
4	Gcov	99443	P3	Missing	RESOLVED FIXED	Expected *
5	Gcov	99444	P3	Wrong	NEW	Confirmed
6	Gcov	99485	P3	Wrong	RESOLVED INVALID	Expected
7	Gcov	100938	P3	Wrong	NEW	Confirmed
8	Gcov	101192	P3	Wrong	NEW	Confirmed
9	Gcov	101193	P3	Wrong	NEW	Confirmed
10	Gcov	101554	P3	Wrong	UNCONFIRMED	Pending
11	Gcov	101569	P3	Wrong	UNCONFIRMED	Pending
12	Gcov	101618	P3	Wrong	RESOLVED FIXED	Fixed
13	Gcov	101644	P3	Spurious	UNCONFIRMED	Pending
14	llvm-cov	48767	-	Spurious	Closed	Fixed
15	llvm-cov	48770	-	Spurious	Closed	Fixed
16	llvm-cov	48771	-	Wrong	Closed	Fixed
17	llvm-cov	48772	-	Spurious	Closed	Fixed
18	llvm-cov	48779	-	Missing	Open	Pending
19	llvm-cov	48782	-	Spurious	Closed	Fixed
20	llvm-cov	48783	-	Spurious	Closed	Fixed
21	llvm-cov	48784	-	Spurious	Closed	Fixed
22	llvm-cov	48827	-	Wrong	Open	Pending
23	llvm-cov	50201	-	Wrong	Open	Pending
24	llvm-cov	50500	-	Wrong	Open	Pending
25	llvm-cov	50610	-	Spurious	Closed	Fixed
26	llvm-cov	50611	-	Wrong	Closed	Fixed
27	llvm-cov	50614	-	Wrong	Closed	Fixed

**: identified by authors*

TABLE 5
DOG'S EFFECTIVENESS ON EARLY BUGS

	Gcov	llvm-cov
Yes	34	19
No	5	4
Total	39	23

priority, and the report status is also somewhat different from Gcov bugs. On the one hand, in Bugzilla, "UNCONFIRMED" means *pending* and if it is confirmed as a new bug, it is marked as "NEW", and the status will eventually change to "RESOLVED FIXED" after being fixed by developers. On the other hand, there are only two statuses "Open" and "Closed" for bug reports of llvm-cov bugs, which are equivalent to *pending* and *fixed*. Since the status of bug reports sometimes does not reflect the real status of bugs, so we validate and unify it based on the comments of developers and the coverage statistics in subsequent versions of subjects. Accordingly, even if it is claimed "RESOLVED FIXED" by developers, we identify the true status of GCC Bug#99443 [59] as *Expected*.

In addition to finding new bugs, we also study whether DOG is effective in revealing bugs in earlier versions. C2V reported 42 bugs of Gcov in version 8.0.0, and 28 bugs of llvm-cov in version 7.0.0. Cod reported 20 bugs of Gcov in version 9.0.1 and 3 bugs of llvm-cov in version 9.0.0. We also validate them manually and filter out those bugs with a status of "WONTFIX" or "DUPLICATE" and finally get 62 valid bugs (39 of Gcov and 23 of llvm-cov). With the code snippets in corresponding bug reports, these bugs in earlier versions are reproduced to verify DOG. As shown in Table 5, DOG can uncover 34 and 19 bugs respectively, demonstrating the comprehensiveness of DOG's detection capabilities. Those coverage bugs missed by DOG, where it is hard to conduct valid constraint solving as there are no enough available coverage statistics in the simple code snippets, suggest the uniqueness of existing methods. When reviewing those early bugs, Gcov bug #99444 [60] in Fig. 6 is found to be a reintroduced case of a fixed Gcov bug #85332 [55] (uncovered by C2V in version 8.0.0) as their input programs are almost the same. The coverage statistic of "case 0:" should be 1, however, after being marked as "RESOLVED FIXED", it changed from 2 to 3, but still incorrect. It makes us suspect that the developers got the root cause wrong and made a bad fix.

Answer: DOG is effective. It can not only find a substantial number of diverse and new bugs in two well-tested coverage profilers but also works for revealing bugs in previous versions.

5.2 RQ2: Significance of uncovered bugs

To understand the significance of our bug-finding, we measure the 24 unexpected coverage bugs (except 3 expected bugs #99442 [58], #99443 [59], and #99485 [61]) from two aspects: lifecycle and uniqueness.

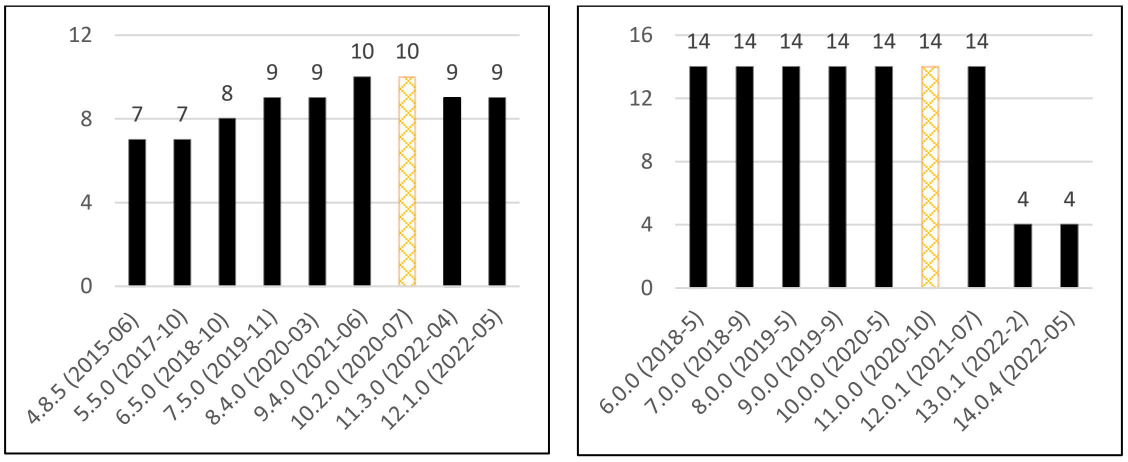
```
  1: 1:  int doit(int sel, int n, void *p)
 -1: 2:  {
  1: 3:    int *const p0 = p;
 -1: 4:
  1: 5:    switch (sel) {
 -1: 6:    {
  2: 7:      case 0:
  3: 8:        do {*p0 += *p0;} while (--n);
  1: 9:        return *p0 == 0;
 -1:10:
  0:11:      default:
  0:12:        abort ();
 -1:13:  }
 -1:14:  }
```

(a) Gcov bug #85332

```
  1: 1:  int doit(int sel, int n, void *p){
  1: 2:    int *const p0 = p;
 -1: 3:
  1: 4:    switch (sel) {
  3: 5:    case 0:
 -1: 6:      do {
  3: 7:        *p0 += *p0;
  3: 8:      } while (--n);
  1: 9:      return *p0 == 0;
  0:10:    default:
  0:11:      __builtin_abort();
 -1:12:    }
 -1:13:  }
```

(b) Gcov bug #99444

Fig. 6. Gcov bug #99444 is a reintroduced case of Gcov bug #85332.



(a) GCov (b) llvm-cov

Fig. 7. Unexpected coverage bugs that affect corresponding release versions of GCov (a) and llvm-cov (b). The abscissa corresponds to the specific versions and release date. The netted bars represent the versions of our subjects.

Lifecycle. We analyze the influence of our bug-finding on previous versions of coverage profilers. For Gcov, we consider 9 official release versions from 4.8.5 (released on June 30, 2015) and later. For llvm-cov, we also consider 9 versions which are from 6.0.0 (released on May 8, 2018) and later. Fig. 7 shows the lifecycle of unexpected bugs over versions where the netted bars represent the experiment version of our subjects. Quantitatively, 7 out of 10 Gcov bugs and all llvm-cov bugs were present from the selected earliest version (Gcov 4.8.5 and llvm-cov 6.0.0) and were not fixed until later versions of the target versions in our experiments. Specifically, during the tracing process of Gcov bugs in the 9 versions, we find that even the same input program has inconsistent coverage statistics in different versions. For the reintroduced bug in Fig. 6, the coverage statistic of “case 0:” is also marked as 5 in version 7.5.0. Meanwhile, the coverage statistic of “return *p0 == 0;” was also wrongly marked as 2 in earlier versions. Gcov bug #101569 [63] is also a case of reintroduction whose coverage statistics are only completely correct in version 8.4.0. Since Gcov-4.8.5 was released in June 2015 and llvm-cov-6.0.0 was released in May 2018, we can conclude that most Gcov bugs are latent for over five years and all llvm-cov bugs are latent for over two and a half years.

Uniqueness. We investigate whether our new bugs can be detected by the state-of-the-art coverage profiler validators C2V and Cod. By identifying inconsistencies in the coverage reports expected to be identical, C2V and Cod found a fair number of bugs in earlier versions of Gcov and llvm-cov. However, they are not as good as they used to be. Table 6 shows the effectiveness of existing methods for the bugs discovered by DOG. For C2V, it still works for all Gcov bugs, but only less than one-third of llvm-cov bugs (4/14) can be killed by C2V. The reason is that the mechanism of instrumentation and parsing of llvm-cov tends to record the number of executions for all code lines. However, Gcov probably does not provide coverage statistics for some complex statements. Therefore, some llvm-cov bugs may be missed by C2V (i.e., “Weak Inconsistency” [23] mentioned in Section 3.1). Meanwhile, it is noted that Cod is completely invalid for coverage bugs of both two coverage profilers. There may be two reasons: 1) Cod has a

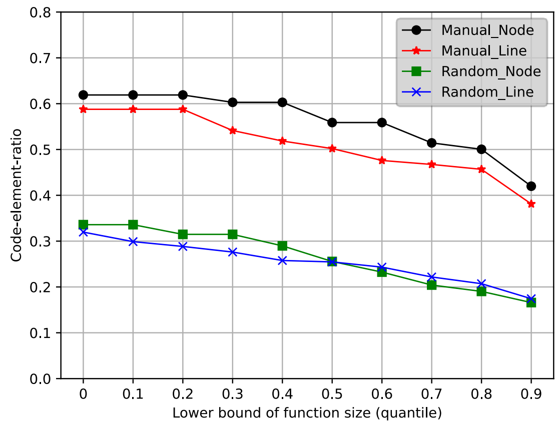


Fig. 8. The average code-element-ratio of diagnostics when the functions exceed a certain size. The abscissa is the quantiles of the function size and the specific number of code elements corresponding to each quantile can be seen in Table 7. Connected by underscores in the legend are the type of functions and code element, respectively.

different preference for inputs. Cod requires unexecuted statements to be modified, while DOG tends to find bugs with inputs having rich execution information as it is more conducive to performing control flow constraint checking; 2) Cod is more of a targeted testing method. In previous work, almost all coverage bugs reported by Cod are Gcov bugs, where half of them have been fixed. That is, Cod targets specific mechanisms of coverage profilers, and this is the reason why Cod can achieve zero false positives. Likewise, the comprehensiveness of Cod is also limited. The above results show that DOG has a unique detection ability to detect bugs that are difficult to be detected by these existing methods.

Answer: Our bug-finding results are significant. More than one-third of them cannot be detected by existing methods. Besides, all the coverage bugs found by DOG are long-latent for years.

5.3 RQ3: Usefulness of diagnostic

By offering insights into why the coverage statistics seem unreasonable based on control dependence, our diagnostic aims to streamline the bug confirmation process and reduce the dependency on extensive code understanding. For compatibility-related bugs, testers can confirm them by solely reading the provided graphic function-level diagnostics (e.g., Fig. 5) rather than comprehending the entire function. The usefulness of such diagnostics can be measured from the following two aspects:

Reduction of manual overhead. During the manual inspection of potential bugs, the manual overhead is roughly positively correlated to the number of code elements that need to be reviewed. Hence, we proposed *code-element-ratio*, which is the ratio of the number of code elements in a diagnostic compared to that of the whole function. The lower the code-element-ratio, the more cost diagnostic can reduce. Moreover, under the same code-element-ratio, the larger the function size, the more reduction. Table 7 shows the distribution of the number of code elements of all 8092

TABLE 6
EXISTING METHODS' EFFECTIVENESS ON NEW BUGS

Method	Testability	Bug	
		Gcov	llvm-cov
C2V	√	10	4
	×	0	10
Cod	√	0	0
	×	0	0
Total		10	14

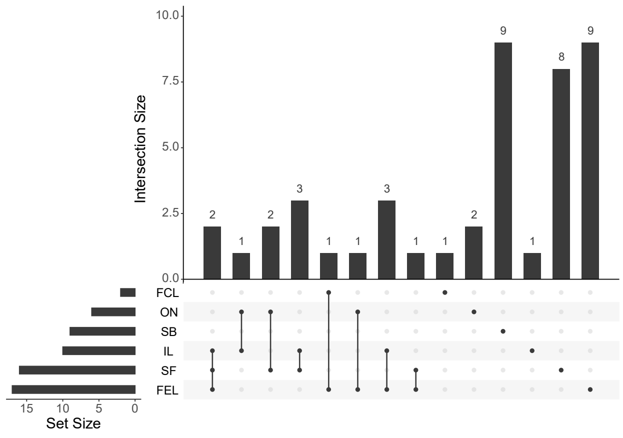
TABLE 7
DISTRIBUTION OF THE NUMBER OF CODE ELEMENTS OF 8092 MANUL FUNCTIONS AND 2756 RANDOM FUNCTIONS

Quantile	Manual functions		Random functions	
	#Node	#Line	#Node	#Line
10%	1	1	2	6
20%	1	2	5	9
30%	2	3	5	12
40%	2	4	7	16
50%	3	5	10	20
60%	3	6	12	25
70%	4	7	16	30
80%	5	8	20	37
90%	8	12	25	49

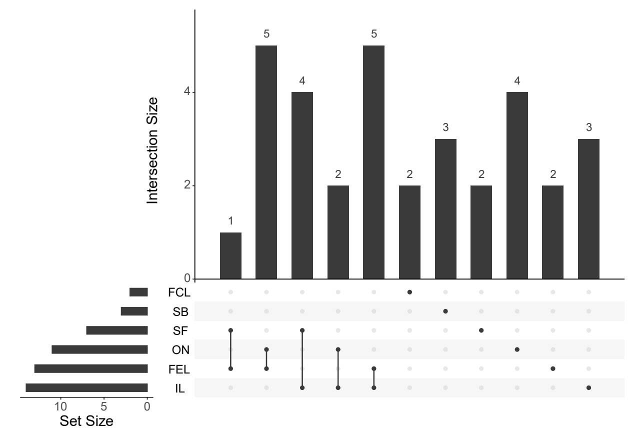
manual functions and 2756 random functions, counted in two ways (content nodes or code lines). In both counting methods, the overall size of manual functions is significantly smaller than that of random functions, with over 80% of manual functions being only a few lines long.

Due to the difference in function size, we use the number of code elements corresponding to different quantiles as a lower bound to analyze the average code-element-ratio of diagnostics when the function size exceeds a certain size. Fig. 8 shows the trend of the average code-element-ratio changing with the lower bound of function size. The abscissa is the quantiles of the function size and the specific number of code elements corresponding to each quantile can be seen in Table 7. The points with a lower bound of 0 correspond to the average code-element-ratio of all diagnostics. As can be seen, there is a large difference in the average code-element-ratio for random and manual functions due to different scales (from low to high are 0.32, 0.34, 0.59, and 0.62). However, the general trend of these four curves is similar. As the lower bound of function size increases, the number of functions beyond that size decreases, and the average code-element-ratio goes down indicating that, on average, the relative size of a diagnostic is decreasing. It is noted that these four curves are relatively flat in the beginning, as most potential compatibility-related bugs are not found in too tiny functions. Considering the wealth of control dependence information inside, graphic diagnostic has a good reduction in the manual overhead, especially in large functions.

Hit rate of inference. If a diagnostic has multiple potential compatibility-related bugs, DOG will infer, as far as possible, which nodes are most likely to have incorrect coverage statistics. Table 8 shows the statistics of graphic diagnostics generated by DOG for the 8092 manual functions



(a) 44 Gcov bugs



(b) 33 llvm-cov bugs

Fig. 9. Upset plots for the relationship between sets of coverage bugs that can be detected by different types of control flow constraints.

and 2756 random functions in the test suite. As shown in the first row, when testing Gcov with the manual inputs, DOG generated 89 graphic diagnostics involving a total of 147 unsatisfiable compatibility-related constraints, and 60 nodes were blamed as suspects by heuristic inference. On average, each graphic diagnostic contains about two unsatisfiable compatibility-related constraints and about one suspect. To obtain the hit rate of inference, we manually inspected the generated graphic diagnostics. As shown in the rightmost column, the hit rate of inference is more than 73% for the manual functions. Erroneous inference mainly comes from two aspects. First, many functions are small in size, so the intersection between 2 unsatisfiable compatibility-related constraints is more likely to have more than one node, resulting in not only the real suspect but also some innocent suspicious nodes being blamed. In this case, the range of suspects is also narrowed, that is, the hit rate will be higher if the "hit" is defined as the set of potential suspects inferred from a graphic diagnostic including the real suspect. Second, wrong inferences also come from false alarms of DOG, which are due to the limitation of the static analysis ability. The cause of false alarms will be left for discussion in Section 6.1. In addition, random functions are less susceptible to the aforementioned two factors and have a higher hit rate, because they are larger in size and their static semantics are easier to analyze.

> ***Answer***: Our diagnostic is useful in facilitating manual bug inspection by reducing the manual overhead, especially in large functions, as well as providing suspects inference with a high hit rate.

TABLE 8
STATISTICS OF GRAPHIC DIAGNOSTICS

Tests	#Func	Subject	#GD*	#UCC*	#SUS*	Hit Rate
Manual	8092	Gcov	89	147	60	73.3%
		llvm-cov	109	246	169	78.1%
Random	2756	Gcov	17	33	12	100%
		llvm-cov	130	287	136	84.6%

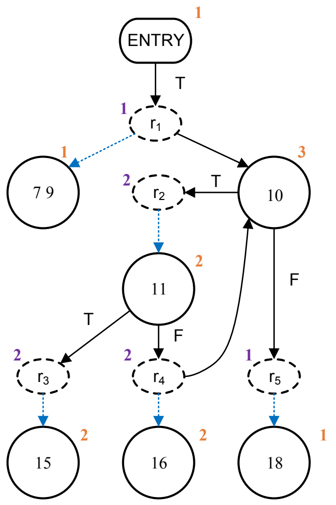
**GD: graphic diagnostic*
**UCC: unsatisfiable compatibility-related constraint*
**SUS: suspect*

5.4 RQ4: Contribution of regularities

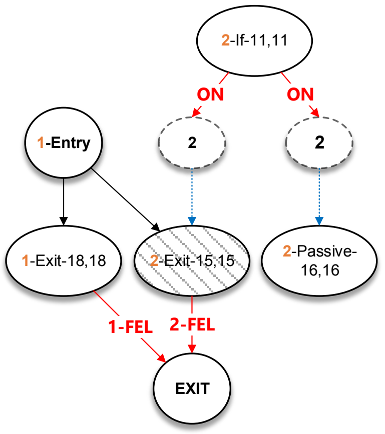
In total, based on six control flow regularities, DOG can respectively detect 44 and 33 new and old bugs in both Gcov and llvm-cov. By analyzing the set of bugs detected by different types of control flow constraints and the relationship between these sets, we can evaluate how these regularities are involved in coverage profiler testing. Since the number of regularities is more than three, we adopt Upset plot to show our results. UpSet plot is a data visualization method first proposed in 2014 [31] and is now frequently used instead of Venn diagram to show set data with more than three intersecting sets, especially in life sciences [32]. Upset shows intersections in the matrix with the rows of the matrix corresponding to the sets and the columns to the intersections between these sets (or vice versa) where the size of the sets and the intersections are shown as bar charts. As shown in Fig. 9, the rows correspond to the bug sets violating different regularity ranked in set size, and the columns correspond to the intersections between these bug sets ranked in degree. The rows in both Fig. 9(a) and Fig. 9(b) show that constraints from all control flow regularities can uncover real coverage bugs but there is a clear difference in the size of the sets: compatibility-related constraints are oriented towards complex relationships within functions and uncover more bugs overall, while the FCL-constraint finds the least number of bugs of both Gcov and llvm-cov. The rightmost six columns in both Fig. 9(a) and Fig. 9(b) indicate that all types of constraints have their unique abilities to reveal some coverage bugs of Gcov and llvm-cov that cannot be detected by other types so that no one regularity can be replaced by the others. The rows in Fig. 9(a) also show that, for Gcov, only the SB-constraint is orthogonal to the others, and the sets of bugs found by any two compatibility-related constraints have an intersection, and there are two bugs even violating three regularities at the same time. The rows in Fig. 9(b) display that, for llvm-cov, both FCL and SB constraints are orthogonal to the

```
 1| |  extern void abort(void)
 2| |
 3|4|  #define f(x) x
 4| |
 5|1|  int main(){
 6|1|  #if f(1) == f /**/(
                  /**/ 1 /**/);
 7|1|    int x;
 8|1|  #endif
 9|1|    x = 0;
10|3|    while (x<2){
11|2|    if (f
12|2|       /**/ (
13|2|         /**/ 0 /**/
14|2|         /**/))
15|2|     abort();
16|2|    x++;
17|2|  }
18|1|    return 0;
19|1|  }
```

(a) Coverage report



(b) CDG

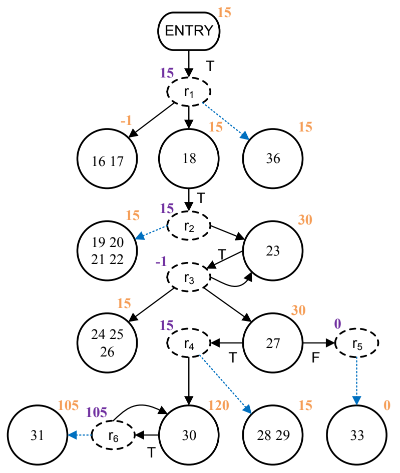


(c) Diagnostic

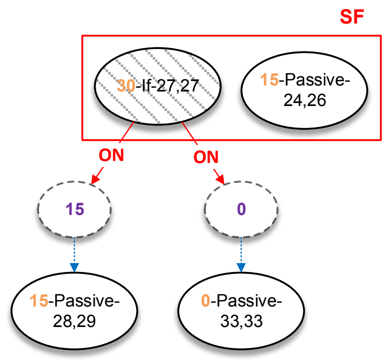
Fig. 10. Coverage report, CDG, and diagnostic of llvm-cov bug #49438. In (a), line 15 is not executed but is wrongly marked as executed twice.

```
                  ...
 15:15:  static int bar(void) {
           ***
 15:18:    if (***) {
             ***
 15:23:      for(***; ***; ***) {
               ***
 30:27:        if(***) {
                 ***
120:30:            while(***)
105:31:              ***
 -1:32:        } else
  0:33:              ***
 -1:34:        }
 -1:35:      }
 15:36:    ***
 -1:37:  }
                  ...
```

(a) Coverage report



(b) CDG

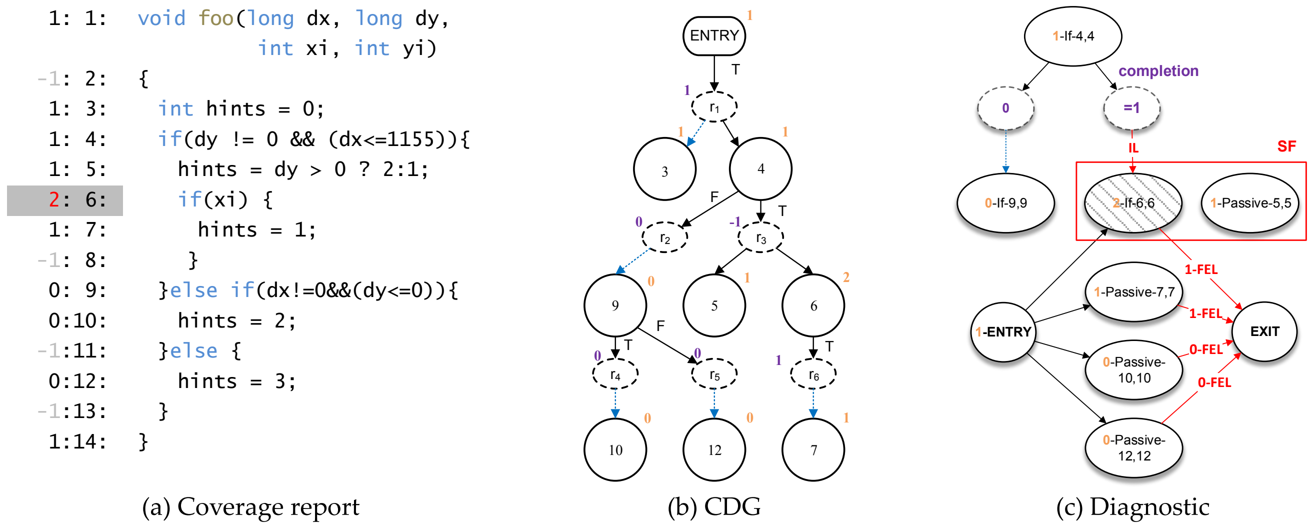


(c) Diagnostic

Fig. 11. Coverage report, CDG, and diagnostic of Gcov bug #99441. In (a), line 27 is only executed 15 times but is wrongly marked as 30 times.

others, and only the bugs detected by FEL have intersections with those of all other compatibility-related constraints. It is not hard to see that FEL contributes the most to inference.

In the example given in Fig. 3, we successfully infer the suspects of a coverage bug uncovered by two unsatisfiable constraints, which are of type IL and ON, respectively. In addition, to visualize the cooperation of the compatibility-related regularities more intuitively, Fig. 10 - Fig. 12 give another three bug examples (llvm-cov bug #49438 [54], Gcov bug #99441[57] and Gcov bug #88930 [56]) to show how we find suspicious nodes and infer the suspect, where from left to right are the segments of coverage report with incorrect coverage statistics, CDG of the corresponding function, and the diagnostic information given by DOG. Fig. 10 is an llvm-cov bug where line 15 is unexecuted but its coverage statistic is incorrectly recorded as 2, and it can be uncovered by DOG with ON and FEL. Fig. 11 is a Gcov bug where the if-statement in line 27 is executed 15 times but its coverage statistic is incorrectly recorded as 30 (the relevant semantic information is hidden for ease of presentation), and it can be uncovered by DOG with SF and ON. Fig 12 is a Gcov bug revealed by C2V in an earlier version where the if-statement in line 6 is executed once but its coverage statistic is incorrectly recorded as 2, and it can be uncovered by DOG with IL, SF, and FEL. Specifically, for the potential bugs uncovered by FEL-constraints, the numbers before the FEL symbol are the actual number of exits of the



```
 1: 1:  void foo(long dx, long dy,
                  int xi, int yi)
-1: 2:  {
 1: 3:    int hints = 0;
 1: 4:    if(dy != 0 && (dx<=1155)){
 1: 5:      hints = dy > 0 ? 2:1;
 2: 6:      if(xi) {
 1: 7:        hints = 1;
-1: 8:       }
 0: 9:    }else if(dx!=0&&(dy<=0)){
 0:10:      hints = 2;
-1:11:    }else {
 0:12:      hints = 3;
-1:13:    }
 1:14:  }
```

(a) Coverage report (b) CDG (c) Diagnostic

Fig. 12. Coverage report, CDG, and graphic diagnostic of Gcov bug #88930. In (a), line 6 is only executed once but wrongly marked as twice, where the relevant semantic information is hidden for ease of presentation.

function from each potential exit.

Take Fig. 12 as an example to show the cooperation of compatibility-related regularities in inference: (1) Node 6 has a brother node 5 with inconsistent coverage so that SF is violated and the coverage of their parent region node r_3 is not available; (2) Since node 4 governs the execution of both two branches (i.e., r_3 and r_4), according to ON, the coverage of r_3 can be completed as:

$$Cov(r_3) = Cov(node\ 4) - Cov(r_2) \tag{10}$$

Therefore, after completing coverage statistics as much as possible, it can be known that node 6 also violates IL; (3) From the perspective of function exit, both node 6 and node 7 are potential exits. Considering that node 7 is control-dependent on node 6, $Cov(r_6) = 1$ and $Cov(node\ 6) = 2$, it turns out that the function "foo" exited once from node 6 through another branch. Accordingly, function "foo" is executed once but is recorded as exiting from node 6 and node 7 once each, causing FEL to be violated. Finally, node 6 is inferred as a suspect for being involved in three unsatisfiable compatibility-related constraints.

> ***Answer***: All six regularities are valid for finding coverage bugs and no one can be replaced by the others. As for manual inspection, any two compatibility-related regularities can work together to infer suspects. Relatively, FCL has weak effectiveness in testing, and FEL contributes the most to inference.

```
1  extern void abort(void);
2  extern void exit(int);
3  void foo(void) {exit(0);}
4  static void bar(void)
5        __attribute__((alias("foo")));
6  int main(void) {
7    bar();
8    abort();
9  }
```

(a) Function alias

```
1   void foo(int x) {
2   switch(x){
3     case 0:
4     if (0) {

5       printf("0\n");
6     case 1:
7       printf("1\n");
8     }
9   }
10  }
```

(b) Mixed control flow context

Fig. 13. Two manual code snippets causing false alarms. Function "foo" is called by alias in (a), and (b) has mixed control flow context.

6 Discussion

In this section, we discuss the causes of false alarms in practice, the feedback of developers, and the scalability of control flow constraint solving.

6.1 Causes of false alarms

In this paper, DOG leverages Understand to analyze the control flow information as well as function call relationships for input programs. Although the test suite has been preprocessed to avoid complicated situations, there are still a few unusual cases in manual inputs beyond the static analysis capabilities of Understand. Fig. 13 shows two typical code snippets of manual programs leading to false alarms. In Fig. 13(a), the function "bar" has an alias "foo", so when line 7 is executed, it is the function "foo" that gets called causing the program to exit early. DOG is unable to handle dynamic information like this yet and still classifies the statements in lines 7 and 8 as a basic block (i.e., one content node). In Fig. 13(b), the structure of the switch-statement and the if-statement is mixed in an unconventional way where the control flow of statements in lines 6 and 7 comes from both the switch-statement in line 2 and the if-statement in line 4, which cannot be properly parsed by Understand as well.

Overall, the false positives of DOG mainly come from

```
 -1: 1:  #include <stdio.h>
 -1: 2:  char fixed_regs[0x00080000];
  2: 3:
  0: 4:  int main(void) {
  0: 5:    printf("PASSED\n")
  0: 6:    return fixed_regs[0x000ff000];
 -1: 7:  }
```

(a) Gcov bug #99442

```
 -1: 1:  #include<stdio.h>
  2: 2:    void free(void *ptr) {
  2: 3:    printf("passed\n");
  2: 4:  }
   : 5:
  1: 6:  void *foo(void) {
  1: 7:    printf("return\n");
  1: 8:    return 0;
 -1: 9:  }
   :10:
  1:11:  int main(void) {
  1:12:    void *p = foo();
  1:13:    free(p);
  1:14:    return 0;
 -1:15:  }
```

(b) Gcov bug #99485

Fig. 14. Two expected Gcov bugs that were directly marked as "RESOLVED INVALID" by developers.

```
 -1: 1:  #include <x86intrin.h>
 -1: 2:  #include <stdio.h>
   : 3:
 -1: 4:  extern void abort(void);
   : 5:
 -1: 6:  #ifdef __x86_64__
 -1: 7:  #define EFLAGS_TYPE unsigned long long int
 -1: 8:  #else
 -1: 9:  #define EFLAGS_TYPE unsigned int
 -1:10:  #endif
   :11:
  0:12:  int main(void) {
  0:13:    printf("1\n");
  0:14:    EFLAGS_TYPE flags = 0xD7; /*111010111b*/
 -1:15:    __writeeflags(flags);
  0:16:    flags = _readeflags();
  0:17:    printf("2\n");
  0:18:    if ((flags & 0xFF) != 0xD7)
  0:19:      abort();
  0:20:    printf("3\n");
 -1:21:  #ifdef DEBUG
 -1:22:      printf("PASSED\n");
 -1:23:  #endif
 -1:24:  }
```

Fig. 15. Expected Gcov bug #99443. It was marked as "RESOLVED FIXED" by the developer but identified as an expected bug by us according to the comments of the developer.

the restriction of static analysis. All the six control flow regularities we propose are general ground regularities that the coverage statistics should satisfy and there should have no false alarm during control flow rule checking with precise control flow information. DOG indeed achieves zero false alarms for uncovering potential bugs with random programs whose control flow information can be fully parsed correctly.

6.2 Developers' feedback

Overall, developers have confirmed 17 and fixed 11 among those coverage bugs that we reported. However, the reaction was different between the developers of Gcov and llvm-cov. For Gcov bugs, the developers are more aggressive, sometimes updating not only the status of a bug report, but also the corresponding reason, and none of the bugs was downgraded to a priority of P4 or P5. While developers of llvm-cov tend to fix coverage bugs silently, leading bugs have only two statuses: *pending* or *fixed*.

Among the bugs reported by us, there are altogether 3 expected Gcov bugs since their anomalous coverage statistics are considered by the developers to be expected. Fig. 14 shows the two expected bugs which are directly marked as "RESOLVED INVALID" by Gcov developers. As shown in Fig.14(a), a segment error was triggered (Gcov bug #99442), causing all statements to be marked as unexecuted. This is expected by developers [58]:

There is no way really to recover from a segfault in a manner that is suitable for all programs.

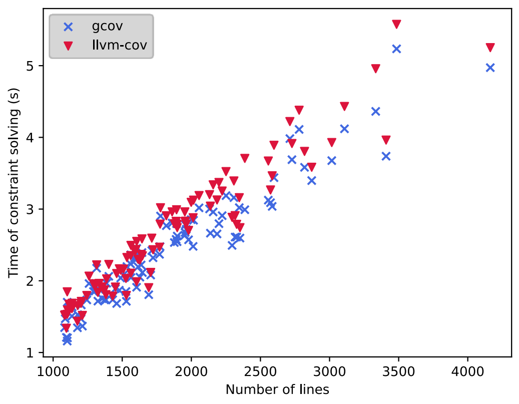
Therefore, when a segment error occurs, the coverage statistics of the executed statements need to be recovered manually. As Fig. 14(b) shows, according to the program semantics, the function "free" should be called only once, but it is executed five times (a total of five times "passed" is printed). The developer explained that the counting mechanism of Gcov can only record the first two times [61]:

Later calls are not counted as counters are already streamed into GCDA file. Well, I tend to close it as invalid.

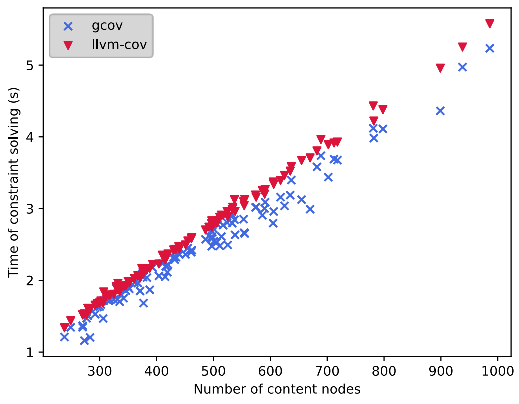
Fig. 15 shows the last special expected bug #99443 where similar to Fig. 14(a), when the program exits from the abort-statement on line 19, all statements were marked as unexecuted (the printf-statements on lines 13 and 17 output successfully). It is also a case of the expected behavior of developers but not documented well, and it is marked as "RESOLVED FIXED" after being documented. Therefore, we tend to identify it as an expected bug.

Despite there being still a few bugs that are pending or waiting to be fixed, developers can get inspiration from them and are indeed actively fixing bugs so we believe there will be more bugs to be confirmed or fixed in the future. For example, in the comments of Gcov bug #101193 revealed by DOG, the first developer thinks that it is not related to bitfields, whereas the second developer disagrees and intends to fix it in the future [62]:

Actually I suspect it is more related to bitfields and spread across multiple lines. There is an optimization done early (before GCOV) in fold-const which combines the above to use and afterwards. I have a few set of patches which allows to get rid of most of the optimization in fold-const dealing with this but not all; This is something which I am going to work towards for GCC 12 but after the current phiopt work.



(a) Measured in lines



(b) Measured in content nodes

Fig. 16. The relationship between the total time of constraint solving and the size of programs, when program size is measured in different ways.

6.3 Scalability

As smaller programs are easier for testers and developers to comprehend, replicate, validate, and subsequently address within the profilers, our experiment was centered around using small programs as inputs to detect coverage bugs. Moreover, considering that we adopt more program analysis techniques than existing works, we will perform a comprehensive analysis of the automation cost and manual effort involved to gain insight into its scalability:

- **Automation cost.** Automation costs primarily arise from two key phases: program analysis and constraint solving. In the phase of program analysis, we employed two algorithms to generate PDT and CDG for each function, respectively. For PDT generation, we utilize the Lengauer-Tarjan algorithm, which efficiently finds dominators in the CFG of a function with a time complexity of $O(V\alpha(V))$ [28], where V represents the number of content nodes and α is the functional inverse of Ackerman's function. With PDT and CFG in hand, the subsequent algorithm allows us to analyze control dependence with a time complexity of $O(V^2)$ [27]. Moving on to the constraint-solving phase, as shown in Table 9, we generate $O(V)$ control flow constraints for each function and all of them are simple linear constraints. To avoid the complexity of constraint-solving instances exploding as the program size increases, we only solve one control flow constraint at a time. During this revision, we additionally employ Csmith to generate 100 random programs with more than 1000 lines to explore the relationship between total constraint-solving costs and program size. Fig. 16 illustrates a linear relationship between constraint-solving time and program size, especially in terms of content nodes. One possible reason is that many lines outside the function are not related to constraint solving, such as macros and global variables. Additionally, we find that the overall time consumption of testing llvm-cov exceeds that of testing Gcov because llvm-cov provides a greater number of coverage statistics to be checked.

TABLE 9
THE NUMBER OF CONTROL FLOW CONSTRAINTS OF EACH TYPE GENERATED FOR A FUNCTION

	SB	SF	IL	ON	FCL	FEL	Total
Number	V	$\leq V$	V	V	1	1	$O(V)$

* *V is the number of content nodes of the CFG of the function*

- **Manual effort.** Manual efforts come from the manual inspection of test results. The difficulty of manual inspection increases as the size increases, and the extent of this increase is difficult to measure. In practice, processing the testing results by hand takes much longer time than the automated execution during the previous testing process. However, our diagnostic information can greatly enhance inspection efficiency by providing control flow reasoning for the presence of unusual coverage statistics, especially in large functions (refer to Section 5.3 for further details). For instance, even statements that are widely separated and seemingly unrelated in terms of control dependencies can be connected through our method, revealing brother relationships (potential bugs uncovered by SF-constraints can be easily confirmed and validated with diagnostics).

7 THREATS TO VALIDITY

We consider the most important threats to the internal and external validity of our study. Internal validity is the degree to which conclusions can be drawn about the causal effect of independent variables on the dependent variables. External validity is the degree to which the results of the research can be generalized to the population under study and other research settings.

7.1 Threats to internal validity

The threats to internal validity mainly come from two aspects. First, our algorithm requires the coverage statistics of graph nodes, but sometimes the coverage statistics of nodes are not available, causing some bugs to be missed

TABLE 10
COMPARISON AMONG C2V, COD, AND DOG

	C2V	Cod	DOG
Methodology	Differential testing	Metamorphic testing	control flow constraint solving
Insight	Multiple coverage profilers should generate consistent coverage statistics for the same program	A coverage profiler should generate consistent coverage statistics for executed statements of path-equivalent variants	Coverage statistics should follow some control flow regularities.
Input	Coverage statistics from different coverage profilers	Coverage statistics for path-equivalent variants	Coverage reports and control flow information
Output	The line with inconsistent coverage statistics	The line with inconsistent coverage statistics	Diagnostic information describing the context of potential bugs
Advantage	Easy to implement	Zero false-positive	Generalizability and interpretability
Shortage	Narrow application and heavy human burden	Limited testing capacity	Restriction of static analysis

during coverage constraint solving. In order to reduce this threat, for graph nodes whose coverage statistics are not available, their coverage statistics are regarded as non-negative integer variables in constraint solving. Besides, we also try to narrow their range based on the available coverage statistics of other nodes in related contexts (e.g., Fig. 12(c)). Second, there is a threat to cause false positives or false negatives in manual work. For one thing, the implementation of the DOG is influenced by the individual's programming ability, for another, we reduce potential coverage bugs in our experiments manually which may lead to mistakes. To alleviate this threat, version control during code development is performed by two authors of this article, and all results of manual deduplication need to be approved by these two authors.

7.2 Threats to external validity

The threats to external validity mainly lie in subjects and test inputs. First, we only choose Gcov and llvm-cov as the subjects. They may not be representative enough for other coverage compilers. In the future, we plan to extend our approach to other C coverage profilers and even coverage profilers for other programming languages. Second, the inputs in our experiment face the threat of diversity in terms of size and semantics. Most functions in our manual test suite are less than 10 lines. We follow the previous works [22], [23] and use the random program generator Csmith to enrich our inputs, but there is still the problem of convergence in these random programs. In future work, we will focus on generating real and diverse test inputs.

8 RELATED WORK

In this section, we introduce the related work in recent years to the automated testing of developer tools. First, we briefly review the work related to coverage profiler testing, after which we present the related testing works about the profiler and debugger.

8.1 Coverage profiler testing

As far as we know, little attention has been paid to the validation of code coverage profilers. Currently, only Yang et al. [22], [23] proposed two coverage profiler validators focusing on this direction, i.e., C2V and Cod. As the first attempt in this direction, C2V leverages a differential testing approach to hunting for bugs in coverage profilers. It assumes that the code coverages given by multiple independently implemented coverage profilers are identical. Therefore, potential bugs could be found by checking the consistency of the coverage reports generated by different coverage profilers. Later, Cod, an automated self-validator for coverage profilers based on metamorphic testing, is proposed to address the limitations of C2V. It uncovers coverage bugs according to a metamorphic relation that modifying unexecuted code blocks in an input program should not affect the coverage statistics of the executed code blocks under the identical profiler.

Inspired by existing works, we propose control flow constraint solving to address the oracle problem within coverage profiler testing, based on which we also implement a profiler validator called DOG. To the best of our knowledge, DOG is the first effort leveraging control flow regularities to validate the coverage profiler, which is our biggest innovation. Table 10 provides a technical comparison of C2V, Cod, and DOG. Clearly, the main difference between these three works is that they adopt different methodologies to tackle the oracle problem. Overall, these three studies have identified a significant number of bugs in various versions of C coverage profilers during different periods. Many of these bugs have been verified and addressed by developers, highlighting the effectiveness of these approaches in enhancing the reliability and maturity of relevant coverage profilers.

8.2 Compiler testing

Compiler testing is the most attractive area of toolchain testing and has received a lot of attention in the past decade. As surveyed by Chen et al. [33], researchers have mainly carried out studies on the following four aspects of compiler testing: (1) constructing test programs. Besides constructing the validation suites manually [34], [35], the related main techniques of test program generation can be broadly categorized as grammar-based methods [36], [37] and mutation-based methods [6], [38]; (2) alleviating the test oracle problem. Since it is difficult to determine the test oracle in compiler testing, i.e., to determine whether a given test program exposes any undesired behavior, many technologies have been proposed to mitigate this issue, e.g., differential testing [7], [36] and metamorphic testing [6], [8]; (3) optimizing the test process. To improve the test efficiency, many optimization approaches for test-program execution have been proposed which can be divided into two types: i.e., test-program prioritization [39], [40] and test-suite reduction [47], [48]; and (4) post-processing of test results. If the test programs indeed trigger compiler bugs, the next step is to understand and fix these bugs. The related efforts can fall into three groups: test program reduction [43], [44], duplicated bug identification [45], [46], and compiler bug debugging [46], [47].

8.3 Debugger testing

Debugger is also an important developer tool that is widely used. Recently, the testing of debuggers has started to receive attention. For the testing of interactive debuggers, Lehmann and Pradel [48] have proposed a feedback-directed test generator, called DBDB, which generates debugger actions to exercise the debugger. By comparing traces of multiple debuggers differentially, diverging behavior that points to bugs and other noteworthy differences can be found. Built on DBDB's approach for obtaining initial test inputs, Tolksdorf et al. [49] have presented the first metamorphic testing approach for debuggers where both the debugged code and the debugging actions will be transformed in such a way that the behavior of the original and the transformed inputs should differ only in specific ways. Li et al. [50] focus on the validation of the correctness of debug information given by a debugger for optimized code. They verify whether a debugger can stop at a predetermined line and print the correct value of a specific variable without triggering undefined behavior for optimized code. Di Luna et al. [51] proposed Debug^2, a framework to find debug information bugs that rely on trace invariants to perform differential analysis on debug traces of optimized and unoptimized programs, and generalized the work of differential analysis of debug information of optimized code to more aspects than the consistency of variables information.

9 Conclusion and future work

In this paper, our primary objective is to unveil coverage bugs in coverage profilers. To achieve this goal, we present an effective approach called control flow constraint solving, specifically designed to tackle the oracle problem inherent in coverage profiler testing. We identify and summarize six control flow regularities based on control dependence and the pattern of control flow during program execution. By leveraging these regularities, we generate coverage constraints that describe the relationship between coverage statistics. Using control flow constraint solving as the foundation, we have developed a code coverage profiler validator called DOG, implemented in Python3. To assess its effectiveness, we conducted evaluations on two widely used C/C++ coverage profilers, namely Gcov and llvm-cov. As a result, we discovered 27 bug reports exposing abnormal coverage statistics. Out of these, 17 bugs were confirmed (with 11 of them fixed), 3 bugs were identified as expected behaviors by the developers, and 7 bugs are still pending. Most notably, among 24 bugs revealing unexpected behaviors, 21 had been latent for at least two and a half years by the time we found them, and 10 are completely undetectable by the state-of-the-art code coverage profiler validators. The results show that the overhead of inspection of compatibilities-related bugs can be reduced effectively and each regularity makes an irreplaceable contribution to finding bugs. Our work introduces a novel method for uncovering bugs in coverage profilers, offering simplicity and effectiveness when compared with state-of-the-art approaches.

In our future endeavors, we are dedicated to further enhancing the effectiveness of control flow constraint solving through several key areas of focus. First, we aim to bolster the generation of diverse test inputs, enabling us to explore a broader range of scenarios and situations within the code. Second, we are committed to advancing the accuracy of program analysis within control flow constraint solving. Third, we plan to extend the applicability of control flow constraint solving to cover profilers designed for other programming languages. Through these future endeavors, we envision a powerful and flexible toolset that empowers developers and researchers to assess code coverage with increased accuracy and confidence.

Acknowledgment

The authors express their sincere gratitude to the anonymous reviewers for providing highly insightful comments and valuable suggestions. This work was supported in part by grants from Natural Science Foundation of China (xxxxxxx).

References

[1] M. Ivanković, G. Petrović, R. Just and G. Fraser, "Code coverage at Google," *Proc. Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 955-963.

[2] M. Böhme, V.-T. Pham and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," *IEEE Transactions on Software Engineering*, vol. 45, no. 5, 2017, pp. 489-506.

[3] C. Lemieux and K. Sen, "Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage," *Proc. Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 475-485.

[4] X. Xie, L. Ma, F. Juefei-Xu, M. Xue, H. Chen, Y. Liu, J. Zhao, B. Li, J. Yin and S. See, "Deephunter: a coverage-guided fuzz testing framework for deep neural networks," *Proc. Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 146-157.

[5] Q. Zhang, J. Wang, M.A. Gulzar, R. Padhye and M. Kim, "Bigfuzz: Efficient fuzz testing for data analytics using framework abstraction," *Proc. 2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2020, pp. 722-733.

[6] V. Le, C. Sun and Z. Su, "Finding deep compiler bugs via guided stochastic program mutation," *ACM SIGPLAN Notices*, vol. 50, no. 10, 2015, pp. 386-399.

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

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

[9] J. Zhang, L. Zhang, M. Harman, D. Hao, Y. Jia and L. Zhang, "Predictive Mutation Testing," *IEEE Transactions on Software Engineering*, vol. 45, no. 9, 2019, pp. 898-918; DOI 10.1109/tse.2018.2809496.

[10] P. Zhang, Y. Li, W. Ma, Y. Yang, L. Chen, H. Lu, Y. Zhou and B. Xu, "CBUA: A probabilistic, predictive, and practical approach for evaluating test suite effectiveness," *IEEE Transactions on Software Engineering*, 2020, pp. 1-1; DOI 10.1109/tse.2020.3010361.

[11] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *Software testing, verification and reliability*, vol. 22, no. 2, 2012, pp. 67-120.

[12] N. Bin Ali, E. Engström, M. Taromirad, M.R. Mousavi, N.M. Minhas, D. Helgesson, S. Kunze and M. Varshosaz, "On the search for industry-relevant regression testing research," *Empirical Software Engineering*, vol. 24, no. 4, 2019, pp. 2020-2055.

[13] G. Fraser and A. Arcuri, "A large-scale evaluation of automated unit test generation using evosuite," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 24, no. 2, 2014, pp. 1-42.

[14] G. Fraser and A. Arcuri, "1600 faults in 100 projects: automatically finding faults while achieving high coverage with evosuite," *Empirical software engineering*, vol. 20, no. 3, 2015, pp. 611-639.

[15] Y. Kim and S. Hong, "DEMINER: test generation for high test coverage through mutant exploration," *Software Testing, Verification and Reliability*, vol. 31, no. 1-2, 2021, pp. e1715.

[16] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M.D. Ernst, D. Pang and B. Keller, "Evaluating and improving fault localization," *Proc. 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, IEEE, 2017, pp. 609-620.

[17] M. Wen, J. Chen, Y. Tian, R. Wu, D. Hao, S. Han and S.C. Cheung, "Historical Spectrum based Fault Localization," *IEEE Transactions on Software Engineering*, 2019, pp. 1-1; DOI 10.1109/tse.2019.2948158.

[18] Y. Li, S. Wang and T. Nguyen, "Fault localization with code coverage representation learning," *Proc. 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, IEEE, 2021, pp. 661-673.

[19] E.K. Smith, E.T. Barr, C. Le Goues and Y. Brun, "Is the cure worse than the disease? overfitting in automated program repair," *Proc. Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 532-543.

[20] J. Yang, A. Zhikhartsev, Y. Liu and L. Tan, "Better test cases for better automated program repair," *Proc. Proceedings of the 2017 11th joint meeting on foundations of software engineering*, 2017, pp. 831-841.

[21] M. Motwani, M. Soto, Y. Brun, R. Just and C. Le Goues, "Quality of automated program repair on real-world defects," *IEEE Transactions on Software Engineering*, 2020.

[22] Y. Yang, Y. Zhou, H. Sun, Z. Su, Z. Zuo, L. Xu and B. Xu, "Hunting for bugs in code coverage tools via randomized differential testing," *Proc. 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, IEEE, 2019, pp. 488-499.

[23] Y. Yang, Y. Jiang, Z. Zuo, Y. Wang, H. Sun, H. Lu, Y. Zhou and B. Xu, "Automatic self-validation for code coverage profilers," *Proc. 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2019, pp. 79-90.

[24] Gcov. Available: https://gcc.gnu.org/onlinedocs/gcc/Gcov.html.

[25] llvm-cov. Available: https://llvm.org/docs/CommandGuide/llvm-cov.html.

[26] V.P. Ranganath, T. Amtoft, A. Banerjee, J. Hatcliff and M.B. Dwyer, "A new foundation for control dependence and slicing for modern program structures," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 29, no. 5, 2007, pp. 27-es.

[27] J. Ferrante, K.J. Ottenstein and J.D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 9, no. 3, 1987, pp. 319-349.

[28] T. Lengauer and R.E. Tarjan, "A fast algorithm for finding dominators in a flowgraph," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 1, no. 1, 1979, pp. 121-141.

[29] Understand. Available: https://www.scitools.com.

[30] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," *Proc. International conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2008, pp. 337-340.

[31] A. Lex, N. Gehlenborg, H. Strobelt, R. Vuillemot and H. Pfister, "UpSet: visualization of intersecting sets," *IEEE transactions on visualization and computer graphics*, vol. 20, no. 12, 2014, pp. 1983-1992.

[32] K. Gadhave, H. Strobelt, N. Gehlenborg and A. Lex, "UpSet 2: From Prototype to Tool."

[33] J. Chen, J. Patra, M. Pradel, Y. Xiong, H. Zhang, D. Hao and L. Zhang, "A survey of compiler testing," *ACM Computing Surveys (CSUR)*, vol. 53, no. 1, 2020, pp. 1-36.

[34] Perennial, "The Perennial Validation Suite for C and C++," [Online]. Available: https://www.peren.com.

[35] K.-H. Wolf and M. Klimek, "A Conformance Test Suite for Arden Syntax Compilers and Interpreters," *Proc. MIE*, 2016, pp. 379-383.

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

[37] J. Chen, G. Wang, D. Hao, Y. Xiong, H. Zhang and L. Zhang, "History-guided configuration diversification for compiler test-program generation," *Proc. 2019 34th IEEE/ACM International Conference on Automated*

Software Engineering (ASE), IEEE, 2019, pp. 305-316.

[38] Y. Tang, H. Jiang, Z. Zhou, X. Li, Z. Ren and W. Kong, "Detecting compiler warning defects via diversity-guided program mutation," *IEEE Transactions on Software Engineering*, vol. 48, no. 11, 2021, pp. 4411-4432.

[39] J. Chen, Y. Bai, D. Hao, Y. Xiong, H. Zhang, L. Zhang and B. Xie, "Test case prioritization for compilers: A text-vector based approach," *Proc. 2016 IEEE international conference on software testing, verification and validation (ICST)*, IEEE, 2016, pp. 266-277.

[40] J. Chen, Y. Bai, D. Hao, Y. Xiong, H. Zhang and B. Xie, "Learning to prioritize test programs for compiler testing," *Proc. 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, IEEE, 2017, pp. 700-711.

[41] H.S. Chae, G. Woo, T.Y. Kim, J.H. Bae and W.-Y. Kim, "An automated approach to reducing test suites for testing retargeted C compilers for embedded systems," *Journal of Systems and Software*, vol. 84, no. 12, 2011, pp. 2053-2064.

[42] A. Groce, M.A. Alipour, C. Zhang, Y. Chen and J. Regehr, "Cause reduction: delta debugging, even without bugs," *Software Testing, Verification and Reliability*, vol. 26, no. 1, 2016, pp. 40-68.

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

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

[45] Y. Chen, A. Groce, C. Zhang, W.-K. Wong, X. Fern, E. Eide and J. Regehr, "Taming compiler fuzzers," *Proc. Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, 2013, pp. 197-208.

[46] J. Holmes and A. Groce, "Causal distance-metric-based assistance for debugging after compiler fuzzing," *Proc. 2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*, IEEE, 2018, pp. 166-177.

[47] J. Chen, J. Han, P. Sun, L. Zhang, D. Hao and L. Zhang, "Compiler bug isolation via effective witness test program generation," *Proc. Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 223-234.

[48] D. Lehmann and M. Pradel, "Feedback-directed differential testing of interactive debuggers," *Proc. Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 610-620.

[49] S. Tolksdorf, D. Lehmann and M. Pradel, "Interactive metamorphic testing of debuggers," *Proc. Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 273-283.

[50] Y. Li, S. Ding, Q. Zhang and D. Italiano, "Debug information validation for optimized code," *Proc. Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 1052-1065.

[51] G.A. Di Luna, D. Italiano, L. Massarelli, S. Österlund, C. Giuffrida and L. Querzoni, "Who's debugging the debuggers? exposing debug information bugs in optimized binaries," *Proc. Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 1034-1045.

[52] X. Yang, Y. Chen, E. Eide and J. Regehr, "Finding and understanding bugs in C compilers," *Proc. Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, 2011, pp. 283-294.

[53] llvm-cov Bug #48771. Available: https://github.com/llvm/llvm-project/issues/48771.

[54] llvm-cov Bug #49438. Available: https://github.com/llvm/llvm-project/issues/49438.

[55] Gcov Bug #85332. Available: https://gcc.gnu.org/bugzilla/show_bug.cgi?id=85332.

[56] Gcov Bug #88930. Available: https://gcc.gnu.org/bugzilla/show_bug.cgi?id=88930.

[57] Gcov Bug #99441. Available: https://gcc.gnu.org/bugzilla/show_bug.cgi?id=99441.

[58] Gcov Bug #99442. Available: https://gcc.gnu.org/bugzilla/show_bug.cgi?id=99442.

[59] Gcov Bug #99443. Available: https://gcc.gnu.org/bugzilla/show_bug.cgi?id=99443.

[60] Gcov Bug #99444. Available: https://gcc.gnu.org/bugzilla/show_bug.cgi?id=99444.

[61] Gcov Bug #99485. Available: https://gcc.gnu.org/bugzilla/show_bug.cgi?id=99485.

[62] Gcov Bug #101193. Available: https://gcc.gnu.org/bugzilla/show_bug.cgi?id=101193.

[63] Gcov Bug #101569. Available: https://gcc.gnu.org/bugzilla/show_bug.cgi?id=101569.

[64] Graphviz. Available: https://www.graphviz.org/.

[65] graphviz. Available: https://pypi.org/project/graphviz/.

[66] os. Available: https://docs.python.org/3/library/os.html.

[67] re. Available: https://docs.python.org/3/library/re.html.

[68] z3. Available: https://pypi.org/project/z3-solver/