Heterogeneous Testing for Coverage Profilers Empowered with Debugging Support

Yibiao Yang
State Key Laboratory for Novel Software Technology, Nanjing University
Nanjing, China
yangyibiao@nju.edu.cn

Maolin Sun
State Key Laboratory for Novel Software Technology, Nanjing University
Nanjing, China
merlin@smail.nju.edu.cn

Yang Wang
State Key Laboratory for Novel Software Technology, Nanjing University
Nanjing, China
njuwy@smail.nju.edu.cn

Qingyang Li
State Key Laboratory for Novel Software Technology, Nanjing University
Nanjing, China
liqingyang@smail.nju.edu.cn

Ming Wen
School of Cyber Science and Engineering, Huazhong University of Science and Technology
Wuhan, China
mwenaa@hust.edu.cn

Yuming Zhou
State Key Laboratory for Novel Software Technology, Nanjing University
Nanjing, China
zhouyuming@nju.edu.cn

ABSTRACT

Ensuring the correctness of code coverage profilers is crucial, given the widespread adoption of code coverage for various software engineering tasks. Existing validation techniques, such as differential testing and metamorphic testing, have shown effectiveness in uncovering bugs in coverage profilers. However, these techniques have limitations as they primarily rely on *homogeneous sources*, i.e., different coverage profilers or the profilers themselves, for validation. In this paper, we propose Decov, a novel *heterogeneous testing* technique, to validate coverage profilers using the information provided by debuggers as a *heterogeneous source*. Coverage profilers record *execution counts* for each source line in the program, while debuggers monitor *hit counts* for each source line when running the program in debug mode. Our key insight is that the *execution counts* obtained from coverage profilers should align with the *hit counts* monitored by debuggers, without conflicts. Decov constructs multiple *heterogeneous relations* and utilizes them to uncover bugs in coverage profilers. Through experiments on Gcov and LLVM-cov, two widely used code coverage profilers, we discovered 21 new bug reports, with 19 of them directly confirmed by developers. Notably, developers have resolved 5 bugs in the latest trunk version. Decov serves as a simple yet effective coverage profiler validator and offers a complementary approach to existing techniques.

CCS CONCEPTS

• **Software and its engineering** → **Functionality**; *Compilers*.

ESEC/FSE '23, December 3–9, 2023, San Francisco, CA, USA

ACM ISBN 979-8-4007-0327-0/23/12…$15.00
https://doi.org/10.1145/3611643.3616340

KEYWORDS

Code coverage, coverage profiler, bug detection, debugging support, heterogeneous testing

ACM Reference Format:
Yibiao Yang, Maolin Sun, Yang Wang, Qingyang Li, Ming Wen, and Yuming Zhou. 2023. Heterogeneous Testing for Coverage Profilers Empowered with Debugging Support. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3611643.3616340

1 INTRODUCTION

Code coverage refers to the execution frequency of code lines in a program under specific inputs. As a fundamental measure to approximate dynamic program behavior, code coverage plays a crucial role in various software engineering practices, including software fuzzing [2, 7, 9, 15, 21, 24, 31], test case prioritization [8, 29], fault localization and repair [22, 23], specification mining [14, 32], fault detection [16], and program understanding [6]. For example, in fault localization, code coverage of a target program obtained from passing and failing tests is integrated into a spectrum, and then suspicious values are computed for each statement based on this spectrum [22]. However, incorrect code coverage information can significantly mislead developers [27]. Therefore, ensuring the correctness of code coverage profilers is of utmost importance.

Nevertheless, code coverage profilers are susceptible to bugs, similar to other software systems. A recent study named C2V has identified over 70 bugs in two popular coverage profilers by employing a simple randomized differential testing technique [27]. The effectiveness of C2V's approach can be ascribed to two fundamental factors elucidated in existing literature [26]: First, the validation of coverage profilers has not received sufficient attention from both developers and academic researchers. Second, the absence of effective test oracles makes automatic testing of coverage profilers challenging. In the context of coverage profiler testing, the test oracle refers to the expected execution frequency of each statement

under a given test input. Unlike conventional oracles used to validate program functionalities, obtaining a complete coverage oracle for a target program, especially with respect to specific test inputs, is extremely challenging. Even if programming experts can specify such precise test oracles for a given program, it is impractical due to the significant amount of human intervention required.

To date, two approaches have been proposed to validate coverage profilers. First, C2V employs differential testing to expose bugs by comparing the coverage statistics provided by different coverage profilers [27]. The fundamental assumption of C2V is that distinct coverage profilers should generate identical coverage for the same program with the same inputs. However, different profilers are typically implemented independently and may define inconsistent coverage semantics. For example, Figure 1 (a) and Figure 1 (c) depict the code coverage reports for the same program obtained by LLVM-cov and Gcov, two popular C coverage profilers. LLVM-cov reports that line 13 is not executed as shown in Figure 1 (a), while Gcov does not provide coverage information for line 13, denoted as "-1" in Figure 1 (c). However, line 13 was indeed executed, and C2V missed this bug. Second, Cod proposes the application of metamorphic testing to uncover bugs in coverage profilers by comparing the coverage statistics for path-equivalent programs [26]. Cod detects bugs by identifying inconsistencies in the coverage statistics generated for the original program and its equivalent mutated variants by a single coverage profiler. The key insight of Cod is that when pruning the unexecuted code of a given program with respect to a particular input, the coverage statistics of the remaining code should remain unchanged. Cod has been reported to be effective in identifying deep bugs in coverage profilers [26]. However, Cod has several limitations, such as false negatives. Using the program in Figure 1 as an example, Cod prunes line 13 to generate the "expected" equivalent mutated program as illustrated in Figure 1 (b). Regrettably, even after the pruning of line 13, the coverage statistics for the remaining code remain unchanged, leading to a missed bug by Cod. The limitations of both C2V and Cod stem from the homogeneous nature of the relations they define. Specifically, C2V leverages another coverage profiler, while Cod uses the profilers themselves for validation. In essence, they check whether the homogeneous relations, defined against other profilers or themselves, are violated.

Approach. To address the limitations of existing approaches, we propose Decov, a *heterogeneous testing* technique, to further testing coverage profilers. Specifically, we define heterogeneous relations with the support of the information provided by a *heterogeneous source*, namely *debuggers*, to validate code coverage profilers. The key insight of Decov is that the execution count of a statement obtained from a coverage profiler should not conflict with the hit count obtained by debuggers. Here, the hit count represents the number of times a statement has been executed in a debugger. To discover bugs in profilers, assuming the correctness of the compiler and the debugger[1], and given a deterministic program $\mathcal{P}$ with a fixed input, Decov first compiles $\mathcal{P}$ with the coverage profiling option (e.g., the "--coverage" option in GCC) and then runs the compiled program to obtain its coverage statistics using the coverage profiler. Subsequently, Decov compiles $\mathcal{P}$ with the debugging option (e.g., the "-g" option in GCC) and feeds the compiled program to a debugger. Using the debugger, Decov debugs $\mathcal{P}$ to obtain the hit count for each statement in $\mathcal{P}$. Decov employs two distinct debugging strategies to determine the hit count using the debugger. The first strategy is the "break-continue" approach, which sets breakpoints for each statement and then executes the program using the "`continue`" command within the debugger to obtain the hit count. The "`continue`" resumes program execution at the address where the program last stopped, bypassing any breakpoints set at that address. The second strategy is the "stepping" approach, where the program is stepped line by line in the debugger to obtain the hit count for each statement. Decov combines these two strategies to obtain the hit count for each statement since relying solely on either strategy cannot accurately represent the actual execution frequencies for each statement (see Section 3.2.2 for detailed insights). Finally, Decov compares the coverage statistics obtained by the coverage profiler and the fused hit count for each statement obtained by the debugger to identify potential bugs in coverage profilers. In other words, if the coverage statistic of a statement conflicts with the fused hit count derived from both the "break-continue" strategy and the "stepping" strategy in the debugger, a potential bug is reported. By leveraging the power of debuggers, Decov captures more comprehensive execution frequencies of code lines, thereby facilitating the detection of deep-seated bugs in coverage profilers.

Results. We implemented Decov as a tool and investigated its effectiveness on two widely used coverage profilers Gcov and LLVM-cov, which are integrated with the compiler of GCC and LLVM, respectively. A total of 21 bugs have been reported and 19 of them are confirmed by the associated developers. In summary, we make the following main contributions:

- ***Concept:*** We introduce the novel concept of *heterogeneous testing* for software validation. This concept provides a fresh and innovative perspective, offering a broad range of potential applications and opening up new avenues for software validation.
- ***Originality:*** We leverage the power of *heterogeneous testing* to validate code coverage profilers by utilizing information obtained from debuggers. To the best of our knowledge, we are the pioneers in harnessing this information source specifically for the purpose of validating coverage profilers.
- ***Implementation:*** We have implemented our proposed concept in the form of a prototype called Decov. Decov serves as a straightforward yet effective coverage profiler validator with integrated debugging support. It presents an additional and complementary solution for exposing bugs in coverage profilers, particularly for newly designed programming languages.
- ***Evaluation:*** We conducted an evaluation of Decov by applying it to two widely adopted C code coverage profilers. The evaluation results demonstrate the effectiveness of Decov, as it successfully uncovered 19 confirmed bugs. Furthermore, Decov exhibits the capability to detect most of the bugs reported in prior studies, thereby providing additional affirmation of its efficacy.

Outline. Section 2 briefly describes the background and motivates our study. Section 3 elaborates on our approach in detail. Section 4 presents the experiment setup and Section 5 presents the evaluation to Decov. We discuss Decov in Section 6 and the related works in Section 7. Section 8 concludes our paper.

[1]We assume this because, with optimization disabled, mis-compilations and mis-debugging are rare.

```
 1| -1|#include <setjmp.h>
 2| -1|
 3| -1|struct T {const int x;} g;
 4| -1|typedef struct T S;
 5| -1|
 6| -1|jmp_buf b;
 7| ✓1|void f() {longjmp(b,1);}
 8| -1|
 9| ✓1|int main() {
10| ✓1|  if (setjmp(b)) {
11| ✓1|    return 0;
12| ✓1|  }
13| ×0|  S *const s[2] = {&g, &g};
14| ✓1|  while(1) f();
15| ×0|}
```

(a) $C_{\mathcal{P}}$(LLVM-cov)

```
 1| -1|#include <setjmp.h>
 2| -1|
 3| -1|struct T {const int x;} g;
 4| -1|typedef struct T S;
 5| -1|
 6| -1|jmp_buf b;
 7| ✓1|void f() {longjmp(b,1);}
 8| -1|
 9| ✓1|int main() {
10| ✓1|  if (setjmp(b)) {
11| ✓1|    return 0;
12| ✓1|  }
13| ×0|  //S *const s[2] = {&g, &g};
14| ✓1|  while(1) f();
15| ×0|}
```

(b) $C_{\mathcal{P}\setminus\{s_{13}\}}$(LLVM-cov)

```
-1:1:#include <setjmp.h>
-1:2:
-1:3:struct T {const int x;} g;
-1:4:typedef struct T S;
-1:5:
-1:6:jmp_buf b;
✓1:7:void foo() {longjmp(b,1);}
-1:8:
✓1:9:int main() {
✓1:10:  if (setjmp(b)) {
✓1:11:    return 0;
-1:12:  }
-1:13:  S *const s[2] = {&g, &g};
✓1:14:  while(1) foo();
-1:15:}
```

(c) $C_{\mathcal{P}}$(Gcov)

```
$ clang -g a.c
$ lldb a.out
(lldb) b a.c:13
Breakpoint 1: a.out at a.c:13:12
(lldb) run
-> 13 S *const s[2] = {&g, &g};
(lldb) continue
Process exited
```

(d) $O_{\mathcal{P}}$(LLDB)

Figure 1: LLVM bug 45166. LLVM-cov incorrectly reported that Line 13 is not executed as shown in (a). When setting breakpoint at Line 13, Line 13 is hit by LLDB in (d). This bug cannot be detected by Cod as removing Line 13 does not affect the coverage statistics of the remaining code as shown in (b). Besides, this bug is also missed by C2V as Gcov does not provide coverage information for Line 13 as shown in (c). '−1' denotes that the Line is not instrumented and then no coverage information is provided. A check or cross mark followed by a number n denotes the execution count of that Line.

2 BACKGROUND AND MOTIVATION

In this section, we introduce the background for coverage profilers and debuggers. Then, we describe existing coverage validation techniques and discuss their limitations.

2.1 Coverage Profilers

Code coverage refers to the execution frequency of code lines with respect to a program's execution under particular test cases. The source-level coverage information is widely adopted to support many software engineering tasks. As the most widely used criterion in measuring the testing adequacy of test cases, it is deeply used for automated software testing. For instance, in the context of regression testing, we often generate test cases covering changed source code to augment a test suite [3]. To collect code coverage information, a compiler needs to emit additional code along with the executable. Along with a program's execution, the required information will be generated and later used by coverage profilers. With the generated runtime information, a coverage profiler produces for each code line $s \in \mathcal{P}$ an execution count $C_{\mathcal{P}}(s) = n$. Due to the default optimizations within the compiler, it is rather common that some source code lines are not instrumented, and thus the coverage profiler cannot provide coverage information for those lines. In this paper, we use $n = -1$ to denote unknown coverage information.

2.2 Debuggers

Debuggers are used to locate and fix programming errors in a target program. It allows developers to see what is going on *inside* the target program at runtime or what is the target program doing at the moment it crashes. In practice, popular debuggers such as GDB and LLDB support a rich set of debugging actions including setting breakpoints on different program locations, stepping program line by line, and inspecting program state at runtime such as the stack frame and the values of variables. With the provided debugging actions, developers can start a program, make a program stop on specified breakpoints, and examine what has happened when the program stops. This helps developers observe the behavior of the target program at runtime. To aid debugging activities, compilers generate debug information together with the machine executable code [1]. For instance, in the case of C compilers, users can enable debug information by specifying the "-g" flag. The debug information contains references to functions, variables, and line numbers in the source code. When a debugging session is initiated, the debugger leverages this debug information to establish a meaningful connection between the executing program and its source code, enabling developers to navigate and understand the program's behavior seamlessly.

2.3 Coverage Profilers Validation

Code coverage profiler validation is challenging as it involves intensive human efforts to validate the ground truth of the coverage statistics. There are only two existing techniques for coverage profiler validation: C2V and Cod. C2V uses differential testing to validate code coverage profilers [27]. Given a program $\mathcal{P}$, C2V uses two independently implemented profilers to obtain the coverage statistics, i.e., $C_{\mathcal{P}}(s)$ and $C'_{\mathcal{P}}(s)$. When $C_{\mathcal{P}}(s) \neq C'_{\mathcal{P}}(s) \wedge C_{\mathcal{P}}(s) \geq 0 \wedge C'_{\mathcal{P}}(s) \geq 0$, C2V regards it as a potential bug and employs clustering to filter out potential duplicated test programs. Cod uses metamorphic testing to validate code coverage profilers. Given a program $\mathcal{P}$, Cod generates $\mathcal{P}$'s equivalent variant $\mathcal{P}'$ by pruning unexecuted statements in $\mathcal{P}$ according to $\mathcal{P}$'s coverage statistic $C_{\mathcal{P}}$. Then, Cod obtains the variant's coverage statistic $C_{\mathcal{P}'}$ by using the same coverage profiler. For any statement s in both $\mathcal{P}$ and $\mathcal{P}'$ (i.e., $s \in \mathcal{P}$ and $s \in \mathcal{P}'$), a potential bug will be found in coverage profiler C if $C_{\mathcal{P}}(s) \neq C_{\mathcal{P}'}(s)$.

2.4 Limitations of Existing Works

Limitations of the differential testing technique. According to the prior study [27], the main limitations of the differential testing technique are summarized as follows: First, C2V *cannot be applied if there is no comparable coverage profiler.* This is common for many mainstream programming languages, e.g., Python and Perl. Second, *differential testing may miss many potential bugs as independently implemented coverage profilers have different opinions on which code*

should have coverage information. A prior study showed that, for Gcov and LLVM-cov, about 50% lines of code in those test programs are non-common instrumentation lines [27]. According to the prior work [26], only half lines of code have coverage statistics for both Gcov and LLVM-cov. In other words, the correctness of the coverage statistics for about half of a program is still unknown in C2V.

Limitations of the metamorphic testing technique. To address the disadvantages of differential testing, Yang et al. [26] proposed Cod to validate code coverage profilers via metamorphic testing. One major flaw of Cod is that *it cannot be applied if no code line can be pruned to generate equivalent programs.* As a result, it may lead to many false negatives. This will frequently occur in two cases: (1) *No source line of code is reported as unexecuted.* In other words, each source line in program $\mathcal{P}$ is either executed or with no coverage information, i.e., $\forall s \in \mathcal{P}, C_{\mathcal{P}}(s) > 0 \vee C_{\mathcal{P}}(s) = -1$. In this situation, Cod cannot be applied to generate equivalent variants; (2) *All the unexecuted source lines are incomplete statements.* In Cod, the path-equivalent programs are generated by pruning unexecuted statements. However, only complete statements can be pruned, whereas coverage statistics from profilers are provided at the line level rather than the statement level. Consequently, many source lines that contain incomplete statements, such as `for/while/switch` expressions placed on different lines, cannot be pruned. Another reason why Cod may generate false negatives is that *pruning uncovered code lines might not affect the coverage statistics of the remaining code lines.* This is exemplified by the case shown in Figure 1, where pruning line 13 does not impact the coverage statistics of the remaining code lines. Consequently, Cod fails to identify this bug as well.

2.5 Motivation

Although existing techniques are effective in exposing bugs in code coverage profilers, they are both limited by the fact that they use homogeneous sources for validation. Particularly, C2V uses two different profiler implementations to seek inconsistencies while Cod defines metamorphic relations with the coverage statistics by the profilers themselves. Therefore, it motivates us to look for other information from a heterogeneous source (i.e., other than the coverage statistics generated by profilers) for coverage profiler validation. In this study, we resort to the information provided by debuggers. As mentioned before, our key insight is that the coverage statistics of a statement by a coverage profiler should be consistent with the hit count obtained by debuggers.

Let us use a concrete example in Figure 1 to illustrate our idea. This is a real bug of LLVM-cov exposed by Decov. In Figure 1 (a), Line 13 in the program is reported as not executed by LLVM-cov. Cod will then prune this unexecuted line, which should not affect the coverage statistics of the remaining code according to Cod's predefined metamorphic relations. Figure 1 (b) is the coverage statistics of the equivalent program after pruning Line 13. We can find that the coverage statistics for the common statements are consistent (i.e., the coverage statistics of Lines 7, 9-12, 14 in Figure 1 (a) and Figure 1 (b)). As a result, no bug is reported by Cod. However, LLDB will reach this line if we set a breakpoint at Line 13 as shown in Figure 1 (d). This line will be hit once when stepping this program in LLDB. That is to say, the hit count of Line 13 is supposed to be 1.

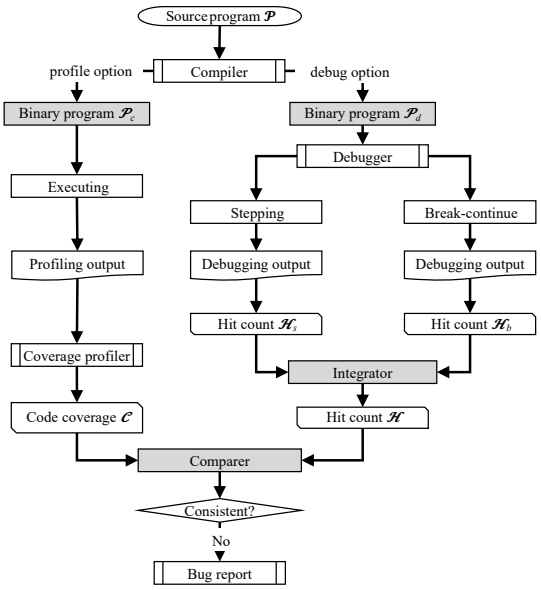


Figure 2: The framework of Decov

Such a result indicates that line 13 should be executed when given the same input, which contradicts the coverage statistics generated by the profiler. Therefore, it motivates us to leverage the information provided by debuggers to design heterogeneous relations to expose potential bugs in coverage profilers.

3 APPROACH

In this paper, we propose to validate code coverage profilers with the help of debugging support.

3.1 Formulation

In this study, for a given statement s in program $\mathcal{P}$, we use $C_{\mathcal{P}}(s)$ to denote its coverage statistic provided by a coverage profiler and $\mathcal{H}_{\mathcal{P}}(s)$ to denote its hit count recorded by a debugger. Here, a statement s is a source line of code with executable code elements, i.e., a non-blank source line of code, while lines consisting solely of braces or comments are excluded from our analysis. Based on our observations, we propose the following *heterogeneous relations* by comparing $C\mathcal{P}(s)$ and $\mathcal{HP}(s)$ for program $\mathcal{P}$ with the same input.

> *If a statement has both coverage information and debug information, the coverage statistic provided by a coverage profiler should not conflict with the hit count provided by a debugger.*

We call this relation *heterogeneous relation* since the information is provided by a heterogeneous source, i.e., the hit count by a debugger. This relation should be satisfied since (1) when a statement is not hit by a debugger after setting a breakpoint against it or not hit when stepping the program line by line, it indicates that the statement has not been executed when feeding the input, and thus a coverage profiler should not report that the statement is executed; (2) when a statement is hit by a debugger, i.e., the statement has actually been executed when feeding the input, the coverage statistic by a profiler should not conflict with the hit count by the debugger if it is available.

3.2 Framework

Based on the formalized heterogeneous relations as described above, we implemented a tool named Decov for testing C code coverage profilers. Decov consists of four main steps: (1) extract code coverage statistics for each line of code by a coverage profiler after executing the program, (2) step the program line by line in a debugger and extract the hit count information $\mathcal{H}_s$, (3) run the program in the debugger via setting breakpoints for each line of code and extract the hit count information $\mathcal{H}_b$, (4) integrate $\mathcal{H}_s$ and $\mathcal{H}_b$ to obtain the fused hit count $\mathcal{H}_p$, and (4) compare the coverage statistics and the fused hit counts to expose bugs.

Figure 2 shows our framework for testing coverage profilers. First, Decov specifies the coverage profiling options of a compiler to compile the program $\mathcal{P}$ and executes the binary with input i to obtain the profiling output, which records the coverage statistics for each line $C_{\mathcal{P}}$. Then, Decov specifies the debug option of a compiler to compile the same program and executes the program $\mathcal{P}$ in a debugger with two different debugging strategies. One debugging strategy is stepping the program line by line and the other one is setting breakpoints for each line of code to run the program. Decov extracts the hit count of each statement for different debugging strategies, which is denoted as $\mathcal{H}_s$ and $\mathcal{H}_b$, respectively. After that, Decov integrated $\mathcal{H}_s$ and $\mathcal{H}_b$ to obtain the fused hit count $\mathcal{H}_P$. Finally, it compares the coverage statistics and the hit count of each line of code to validate the coverage profiler T. A potential bug in T will be reported if any of them violates the predefined relations. We describe each of the steps in detail as follows.

3.2.1 Extract Coverage Statistics. For a given test program $\mathcal{P}$, we specify the coverage profiling options for the compiler to generate the profiled executable as $\mathcal{P}_c$. The compiler will generate the information required by the coverage profilers and also integrate instrumentation code into the binary. Then Decov executes $\mathcal{P}_c$ to generate the profiling output. With the profiling output and the source program, we can obtain the code coverage report $C_{\mathcal{P}}$ by the coverage profiler T. The code coverage report is the coverage statistics of each line for the test program $\mathcal{P}$ under the input i. Decov can be applied to both the Gcov and the LLVM-cov profilers. For instance, to extract the coverage information by Gcov, Decov specifies the "`-O0 --coverage`" flag to GCC when compiling the program $\mathcal{P}$ to $\mathcal{P}_c$. This flag instructs the GCC compiler to include additional code in the executable, enabling the generation of supplementary profiling information during program execution. Subsequently, Decov executes the resulting executable $\mathcal{P}_c$ with input i, thereby generating the coverage report for program $\mathcal{P}$.

3.2.2 Extract and Fuse Hit Count. To extract the hit counts for the test program $\mathcal{P}$, we specify the debug options for compilers to generate the executable as $\mathcal{P}_d$. The debug options enable the compiler to generate debug information along with the executable binary. The debug information establishes a mapping between the source code and the executable. Specifically, Decov adopts two different debugging strategies to obtain the hit count. The first one is the "stepping" strategy, in which case Decov runs the program $\mathcal{P}_d$ via stepping line by line in the debugger, and then extracts the hit count $\mathcal{H}_S$ for each statement. At each step, the current stack frame will be examined and recorded to the debug output O. Taking GDB as an example, Decov uses the debug action sequence "`start` → while(true) {`frame` → `step`}" to query the current stack frame iteratively until the program $\mathcal{P}_d$ exits for the "stepping" strategy. Here, the debug action "`start`" instructs GDB to run the program and stops at the beginning of the main procedure of $\mathcal{P}_d$. The debug action "`frame`" examines the stack frame when GDB stops at any program location of $\mathcal{P}_d$, and "`step`" continues to run the program in GDB until reaches a different source line. The second one is the "break-continue" strategy, in which case Decov sets breakpoints for all statements with debug information, runs the program to observe its execution, and then obtains the hit count $\mathcal{H}_{\mathcal{B}}$. When the debugger stops at any breakpoint, the current stack frame will be examined and recorded. For GDB, Decov uses the debug action sequence "`break s1` → `break s2` → ... → `run` → while(true) {`frame` → `continue`}" to query the stack frame iteratively until the program $\mathcal{P}$ exits for the "break-continue" strategy. Here, the debug action "`continue`" resumes program execution at the location where it stopped. With the debug action sequence and the executable, Decov executes program $\mathcal{P}_d$ to obtain the debugging output. Then, Decov extracts the hit count $\mathcal{H}_{\mathcal{S}}$ and $\mathcal{H}_{\mathcal{B}}$ with respect to the two debugging strategies. Finally, Decov integrates the hit count $\mathcal{H}_{\mathcal{S}}$ and $\mathcal{H}_{\mathcal{B}}$ into $\mathcal{H}_{\mathcal{P}}$. We adopt the integration strategy because the hit count from the "break-continue" strategy may be inconsistent with the hit count from the "stepping" strategy, and thus the hit count obtained from either of these strategies in isolation cannot precisely represent the actual execution frequencies for each statement. For instance, under the "stepping" strategy, if a source line is a function call to a user-defined function, the debugger hits the line twice: once when reaching the function call and again after returning from the callee. Conversely, in the "break-continue" strategy, the function call is hit only once. Consequently, if a function call has been executed m times, the hit count obtained by the "break-continue" strategy is $1 * m$, while the hit count from the "stepping" strategy is $2 * m$. Therefore, the hit count from the "stepping" strategy for a function call does not accurately represent the actual execution frequencies. Similarly, for a "`for (init; condition; increment)`" loop, it is hit only once in the "break-continue" strategy when the execution reaches the loop. Regardless of how many times the `condition` and `increment` expressions are executed, the loop is never hit again. Thus, the hit count from the "break-continue" strategy for the loop cannot accurately represent the actual execution frequencies. Due to such expected behaviors of debuggers, we thus integrate the hit counts from these two different debugging strategies to obtain an integrated hit count for each line of code. In particular, if a source line of code is a loop statement, its coverage statistic will be compared against the hit count from the "stepping" strategy. If it is a function call statement, the coverage statistic will be compared against the "break-continue" strategy.

3.2.3 Compare Coverage Statistics and Hit Counts. Using coverage statistics from the profiler and hit counts for program $\mathcal{P}$ obtained through two debugging strategies, Decov uncovers bugs in code coverage tools by checking whether they violate our predefined relation. More specifically, for a source line s that has debug information and coverage information provided by the coverage profiler (i.e., $C_{\mathcal{P}}(s) \neq -1$), Decov verifies if the coverage statistics $C_{\mathcal{P}}(s)$ conflict with the hit counts obtained from the two debugging strategies.

```
1 #define N 8
2
3 int main(){
4   int i=32;
5   if(i<N)
6     return 1;
7 }
```

(a) $\mathcal{P}$

```
1| ✓1|#define N 8
2| -1|
3| ✓1|int main(){
4| ✓1|  int i=32;
5| ✓1|  if(i<N)
6| ✓1|    return 1;
7| ✓1|}
```

(b) $C_{\mathcal{P}}$(LLVM-cov)

```
$ clang -O0 -g a.c
$ lldb a.out
(lldb) break 6
Breakpoint 1: a.out \
at a.c:6:5
(lldb) run
Process exited
```

(c) $O_{\mathcal{P}}$(LLDB)

Figure 3: LLVM-cov bug 45194. LLVM-cov incorrectly reported that the `return 1;` in Line 6 is executed once. Setting breakpoint in Line 6 will produce a valid breakpoint at this statement. However, when running it in LLDB, Line 6 is not hit by LLDB. That's to say, Line 6 is not executed according to LLDB. Note that, this bug has been fixed by developers.

There are four different situations: (1) If a source line is neither hit by the "stepping" strategy nor hit by the "break-continue" strategy, the source line should not be executed or with no coverage information; (2) If a source line is a loop statement and hit by debugger under the "stepping" strategy, its coverage statistic should be equal to the hit count from the "stepping" strategy or with no coverage information; (3) If a source line is a function call statement and hit by debugger under the "break-continue" strategy, its coverage statistic should be equal to the hit count from the "break-continue" strategy or with no coverage information; (4) If a source line is neither a loop statement nor a function call statement when the line is hit by debugger under either "stepping" strategy or "break-continue" strategy, its coverage statistic should be equal to the hit count from either one of the two debugging strategies. Above all, assuming that $\mathcal{D}(P)$, $\mathcal{L}(P)$, $\mathcal{F}(P)$ represent the set of statements with debug information, the set of loop statements, and the set of function call statements in the program $\mathcal{P}$, respectively for any source line s with coverage information, i.e., $C_{\mathcal{P}}(s) \neq -1$, the following relations should be satisfied:

HR#1:$\forall s \in \mathcal{D}(P),\ \mathcal{H}_{\mathcal{S}}(s) = 0 \wedge \mathcal{H}_{\mathcal{B}}(s) = 0 \Rightarrow C_{\mathcal{P}}(s) = 0$

HR#2:$\forall s \in \mathcal{D}(P),\ s \in \mathcal{L}(P) \wedge \mathcal{H}_{\mathcal{S}}(s) \geq 1 \Rightarrow C_{\mathcal{P}}(s) = \mathcal{H}_{\mathcal{S}}(s)$

HR#3:$\forall s \in \mathcal{D}(P),\ s \in \mathcal{F}(P) \wedge \mathcal{H}_{\mathcal{B}}(s) \geq 1 \Rightarrow C_{\mathcal{P}}(s) = \mathcal{H}_{\mathcal{B}}(s)$

HR#4:$\forall s \in \mathcal{D}(P),\ \mathcal{H}_{\mathcal{S}}(s) \geq 1 \vee \mathcal{H}_{\mathcal{B}}(s) \geq 1 \Rightarrow$
$$C_{\mathcal{P}}(s) = \mathcal{H}_{\mathcal{B}}(s) \vee C_{\mathcal{P}}(s) = \mathcal{H}_{\mathcal{S}}(s)$$

Note that, if a statement is a mix of loop and function call, e.g., function call as the loop-expression, it will neither considered as a loop-statement nor a function call statement. It is also worth noting that only the source lines with debug information are considered for further comparison. This is because the hit count monitored by the debugger for a source line without debug information is inaccurate. Therefore, in such cases, we refrain from utilizing this information to validate coverage profilers.

3.3 Illustrative Examples

In this section, we use three concrete bug examples to illustrate how Decov works. All these bugs are detected by Decov and have been confirmed by developers.

Bug Example with Bug Type of Spurious Marking. Figure 3 is a bug of LLVM-cov exposed by Decov with the bug type of *Spurious Marking*, i.e., there exists an unexecuted statement that is wrongly marked as executed by the coverage profiler [27]. That is to say, there is a statement hit by LLDB but not marked as executed by LLVM-cov. Figure 3 (a) and (b) show the program and the coverage report produced by LLVM-cov, respectively. Note that all the presented test programs are reformatted to ease presentation. As can be seen, the coverage report produced by the coverage profiler in Figure 3 (b) is an annotated version of the source code, which includes the execution frequency along with each line. The first column is the line number and the second column is the execution count. The "-1" shown in the second column indicates that the coverage profiler does not provide coverage information for this line as it is not instrumented. In this example, to obtain the hit count for Line 6 in program $\mathcal{P}$, we first use Clang to compile $\mathcal{P}$ with the debug information enabled. Then, Decov sets breakpoints for Line 6 in program $\mathcal{P}$. Specifically, Decov uses the debug action sequence "`run` → `continue` → `continue`→ ..." to execute $\mathcal{P}$ and then obtain the debug output $O_{\mathcal{P}}$ as shown in Figure 3 (c). We can find that when running the program in the debugger, the process is directly exited without break at Line 6. Consequently, the hit count of Line 6 is 0. Furthermore, when stepping program $\mathcal{P}$ line by line, Line 6 is not hit in LLDB either. However, $C_{\mathcal{P}}(s_6)$ is 1, and the predefined relation *HR* is violated. Therefore, a potential bug is uncovered in LLVM-cov, which is reported as LLVM bug 45194. It is worth noting that this bug was initially marked as resolved, then subsequently reopened, and ultimately resolved by the developers.

Bug Example with Bug Type of Missing Marking. Figure 4 shows a real bug example exposed by a statement hit by LLDB but not marked as executed by LLVM-cov. The type of this bug is *Missing Marking*, i.e., there exists an executed statement that is wrongly marked as uncovered [27]. Figure 4 (a) and (b) are the program and the code coverage report produced by LLVM-cov, respectively. In this example, Decov steps program $\mathcal{P}$ line by line. Specifically, Decov uses the debug action sequence "`break main` → `run`→ `step` → `step` → `...`" to execute $\mathcal{P}$ and then obtains the debug output O as shown in Figure 4 (c). We can find that LLDB stops at Line 4 two times, and thus the hit count of Line 4 is 2. However, $C_{\mathcal{P}}(s_4)$ is 0. The predefined relation between the coverage statistics and the hit count is not satisfied. Thus, a potential bug is uncovered in LLVM-cov. We reported this bug, and it was promptly confirmed by the LLVM developers.

Bug Example with Bug Type of Wrong Frequency. Figure 5 illustrates another real bug example for coverage profiler Gcov. This bug is uncovered as the execution count from Gcov exceeds the hit count monitored by the debugger for Line 8. Figure 5 (a), (b), and (c) show the test program $\mathcal{P}$, the code coverage statistics produced by Gcov, and the debug output from GDB, respectively.

```
1  void main() {
2    int s = 0;
3    switch (s) {
4      for (; ;) {
5        s++;
6      case 0:
7        if (s>=2)
8          break;
9      }
10   }
11 }
```

(a) $\mathcal{P}$

```
1|  -1|void main(){
2|  ✓1|  int s = 0;
3|  ✓1|  switch (s){
4|  ×0|    for (; ;){
5|  ✓2|      s++;
6|  ✓3|    case 0:
7|  ✓3|      if (s>=2)
8|  ✓1|        break;
9|  ✓2|    }
10| ✓1| }
11| ✓1|}
```

(b) $C_{\mathcal{P}}$(LLVM-cov)

```
$ clang -g a.c; lldb a.out
(lldb) break main
Breakpoint 1: at a.c:2
(lldb) run
-> 2  int s = 0;
(lldb) ...
(lldb) step
-> 4    for(; ;){
(lldb) ...
(lldb) step
-> 4    for(; ;){
```

(c) $O_{\mathcal{P}}$(LLDB)

Figure 4: LLVM-cov bug 45023. LLVM-cov incorrectly reported the `for(;;)` expression in Line 4 is not executed. However, Line 4 is hit by LLDB when setting breakpoint for it.

```
1 __attribute__ ((always_inline))\
2 int foo (int v) {
3   return (v << 8) | (v >> 8);
4 }
5
6 int main () {
7   int b = 0;
8   return foo(b);
9 }
```

(a) $\mathcal{P}$

```
-1:1:__attribute__ ((always_inline))\
-1:2:int foo (int v) {
×0:3:  return (v << 8) | (v >> 8);
-1:4:}
-1:5:
✓1:6:int main() {
✓1:7:  int b = 0;
✓2:8:  return foo(b);
-1:9:}
```

(b) $C_{\mathcal{P}}$(Gcov)

```
$ gcc -O0 -g a.c
$ gdb a.out
(gdb) break 8
Breakpoint 1 at \
file a.c, line 8.
(gdb) run
-> 8 return foo(b);
(gdb) continue
Process exited
```

(c) $O_{\mathcal{P}}$(GDB)

Figure 5: Gcov bug 93706. Gcov incorrectly reported that the `return foo(b);` in Line 8 is executed twice. However, the debugger only hit Line 8 once when setting breakpoint for Line 8 in GDB. This bug cannot be detected by Cod as removing Line 3 will not affect the coverage statistic of the rest code. In practice, Line 3 is replaced with a blank statement ';' rather than just removed.

From Figure 5 (b), we can see that Line 8 is executed twice in Gcov. When setting a breakpoint at Line 8 in $\mathcal{P}$ and running the program $\mathcal{P}$ in GDB, the debugging process exits after taking the "`run` → `continue`" debugging actions. Therefore, Line 8 is only hit one time by the GDB. Therefore, the predefined relation is not satisfied and indicates a potential bug in Gcov. This bug has already been confirmed by the GCC developers.

4 EXPERIMENTAL SETUP

In this section, we describe the experimental setup in detail, including the research questions, the subject coverage profilers under test, the debuggers used for the validation, and the test programs.

4.1 Research Question

In this paper, we study the following research questions:

- ***RQ1*: (*Effectiveness*) How effective is Decov in uncovering coverage profiler bugs?**
- ***RQ2*: (Significance) How significant are the bugs exposed by Decov?**
- ***RQ3*: (*Complementarity*) How does Decov complement existing techniques?**

4.2 Evaluation Setup

In this section, we describe the coverage profilers, the debuggers, and the test programs chosen for the validation. Note that, all experiments were conducted on a hexa-core Intel(R) Core(TM) CPU@3.20GHz with 10GiB RAM running Ubuntu 20.04.

Subject Coverage Profilers. We applied Decov to the latest trunk versions of Gcov and LLVM-cov, the two popular C code coverage profilers. We chose them as subjects for the following reasons:

(1) They have been *widely used* in the engineering community;
(2) They are *integrated* into the well-known production compilers, i.e., GCC and Clang, respectively.
(3) They are used as the *subject profilers* in prior studies, and thus we can easily and fairly compare with previous approaches.

To extract code coverage reports by using Gcov and LLVM-cov for each test program, we use the compilation flag "-O0" when compiling test programs to profiling executables. At "-O0", most optimizations are completely disabled. This profiling strategy for the coverage profilers is also adopted in prior studies [26, 27]. For a given source file `a.c`, the commands used to produce the code coverage report `a.c.gcov` are as follows:

```
$ gcc -O0 --coverage a.c && ./a.out && gcov a.c
```

For LLVM-cov, the following commands are used to produce the coverage report `a.c.lcov`:

```
$ clang -O0 -fcoverage-mapping -fprofile-instr-generate
  a.c && ./a.out
$ llvm-profdata merge default.profraw -o a.pd
$ llvm-cov show a.out -instr-profile=a.pd a.c > a.c.lcov
```

Debuggers Support Validation. To validate Gcov and LLVM-cov using Decov, we use GDB and LLDB as debugging tools to provide the debugging information. GDB is the GNU debugger while LLDB is the LLVM debugger. These two debuggers are selected due to their association with the respective compiler toolchains. To extract the hit count by using GDB and LLDB for each test program, we compile them with debug options enabled and compiler optimizations disabled. The hit count is only obtained for code lines that have associated debug information. Specifically, when utilizing GDB to obtain the code lines with debug information for a given source file, such as `a.c`, the following commands are employed:

```
$ gcc -O0 -g a.c && gdb ./a.out
(gdb) start
(gdb) maint info line-table
```

For LLDB, the following commands are used to obtain the code lines with debug information:

```
$ clang -O0 -g a.c && lldb ./a.out
(lldb) image dump line-table a.c
```

Having the code lines with debug information, we develop a Python script to run debug sessions non-interactively for GDB and LLDB and extract the hit counts for the program $\mathcal{P}$ under the "stepping" and the "break-continue" debugging strategies, respectively.

Subject Test Programs Consistent with the prior study [26], we use the C programs inside the test-suite of GCC release of 7.4.0 as the subject test programs. In total, there are 26,530 C source programs. We choose these test programs as the subject testing inputs for the validation of C code coverage profilers because (1) they have been used as the subject programs in the prior study [26]; (2) they are open source which can be easily obtained; (3) They are typically used for compiler regression testing, covering a wide range of C semantics; and (4) many of these test programs can be compiled and executed independently, as they do not rely on external libraries and have fixed inputs.

5 EVALUATION

We elaborate on the evaluation for Decov in this section. We apply Decov to validate the two most widely used C code coverage profilers and compare Decov with two state-of-the-art approaches, namely Cod [26] and C2V [27].

5.1 Effectiveness of Decov

Our aim is to find previously unknown bugs for code coverage profilers. To this end, we applied Decov to the latest version of Gcov in GCC (until GCC 11.0.0 20210113) and LLVM-cov in LLVM (until LLVM 12.0.0) during our experiments. As mentioned in Section 4.2, we employ the test programs in the GCC test-suite as the subject test programs. For each of the test programs, we use the aforementioned commands to extract code coverage statistics and monitor the hit counts by debuggers. With the collected code coverage reports and the hit counts of specific statements, Decov uncovers bugs in code coverage profilers by checking whether the predefined heterogeneous relations are violated. For each test program, Decov generates the code coverage reports by the target profilers (i.e., either Gcov or LLVM-cov) and the hit counts with the associated debuggers. In this experiment, we exclude those uncompilable programs with respect to GCC and Clang. Besides, the programs that require more than ten seconds to run are also excluded. This is because our aim is to detect new bugs in code coverage profilers as efficiently as possible. Thus, in this study, Decov tends not to waste time for waiting the test programs that require a long time for the execution to be finished. Decov may generate false positives in specific scenarios. First, an inline assembly statement may possess debug information but never be reached by Gdb. Second, statements like "while(1)" or "for(;;)" may never be hit by LLDB. Third, a statement containing a loop macro might be hit only once in GDB and the execution count is only one in LLVM-cov, while being executed or hit multiple times in Gcov or LLDB. These divergent behaviors are expected due to their varying explanations for such statements. To mitigate false alarms, Decov employs an automated filtering mechanism to exclude such violations.

Table 1: List of confirmed or fixed bugs reported by Decov.

ID	Profiler	#ID	Current Status	Cod	C2V
1	Gcov	93680	Confirmed	×	✓
2	Gcov	93706	Confirmed	×	✓
3	Gcov	97910	Confirmed	×	✓
4	Gcov	97917	Confirmed	×	✓
5	Gcov	97923	Confirmed	×	×
6	Gcov	97924	Confirmed	×	✓
7	Gcov	97925	Confirmed	×	✓
8	Gcov	105500	Fixed	×	✓
9	LLVM-cov	44940	Confirmed	×	✓
10	LLVM-cov	45023	Confirmed	×	×
11	LLVM-cov	45166	Confirmed	×	×
12	LLVM-cov	45190	Confirmed	×	✓
13	LLVM-cov	45191	Confirmed	×	×
14	LLVM-cov	45194	Fixed	×	×
15	LLVM-cov	45195	Fixed	×	✓
16	LLVM-cov	47607	Confirmed[2]	×	×
17	LLVM-cov	47608	Confirmed	×	×
18	LLVM-cov	48139	Fixed	×	✓
19	LLVM-cov	48162	Fixed	×	✓

The last two columns indicate whether the bug is able (denoted as ✓) or unable (denoted as ×) to be identified by the existing technique of Cod or C2V

Bugs Found. We manually inspected each of the violations in the test programs identified by Decov. Decov provides reports indicating the specific lines in a test program that violate the predefined relations. It is worth emphasizing that the examination of suspicious coverage statistics is not a resource-intensive endeavor, as we can ascertain the presence of a potential bug in the coverage profiler by scrutinizing the coverage statistics surrounding the suspicious lines. To date, we totally reported 21 bugs to the Bugzilla databases of GCC and LLVM. Note that the bug management guidelines of GCC and LLVM are not exactly similar [12, 17, 26]. Specifically, a reported bug in GCC Bugzilla will be initially labeled as "UNCONFIRMED", and the status will turn to "NEW" if developers confirm it. However, in LLVM, a reported bug will be labeled as "NEW" by default, and the status will turn to "CONFIRMED" if developers confirm it. In addition, a bug report reopened by developers is also considered as a confirmed bug report. Worthy noting that

there are still many unfixed bugs reported in prior studies. Some violations detected by Decov are associated with these previously unfixed bug reports. Thus, to avoid duplicate bug reports, we manually inspected each violation and only filed **21** new bug reports to the Bugzilla database of GCC and LLVM. Among the 21 new bug reports, 19 bugs have been directly confirmed by developers as depicted in Table 1, and two bugs are marked as duplicate reports. Of the total 19 confirmed bugs, 5 have been resolved by the developers. For two of these bugs (one in Gcov and the other in LLVM-cov), the developers have provided comments detailing the corresponding fixing commits. Besides, all of the remaining three fixed bugs are attributed to LLVM-cov. These three bugs were initially confirmed in Bugzilla and later migrated from Bugzilla to the LLVM GitHub project repository.[3] Upon further examination, it has been observed that these bugs no longer exist in the latest trunk version of LLVM-cov. Consequently, the LLVM developers have marked these bugs as completed and closed them. In the present context, these three bugs are also considered resolved bug reports.

5.2 Significance of Decov

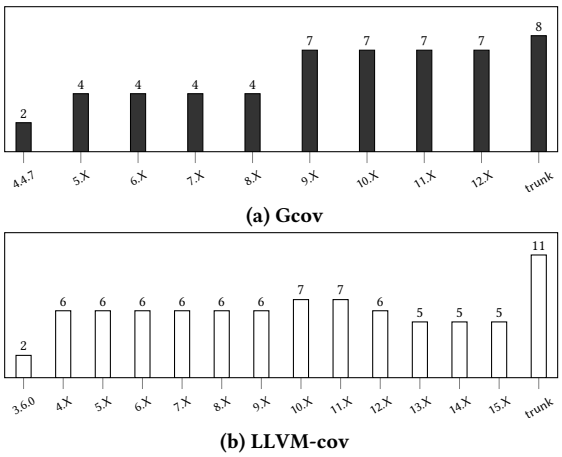
Buggy versions of Profilers. To further understand the importance of the bugs discovered by Decov, we investigate the official release versions of profilers that are affected by the confirmed bugs. Here, we select Gcov-4.4.7 (released on March 13, 2012), LLVM-cov-3.6.0 (released on Feb. 27, 2015), and the subsequent official release versions of the profilers as the subjects. Note that the versions of GNU and LLVM toolchains are consistent with the versions of profilers. As Figure 6 shows, two bugs in Gcov appeared at version 4.4.7, which means they are latent for about eight years and not found by other testing tools. This circumstance is similar to LLVM-cov. Concretely, 2 out of the 11 confirmed bugs were induced in LLVM five years ago. In conclusion, a number of the bugs detected by Decov have long lifespans, which indicates the bugs' significance.

Feedback and discussions. After the bug reports are submitted, many of them invoke developers' lively discussion and may inspire their work in progress. For example, with respect to LLVM-cov bug 44940, two developers express their opinions. Ultimately, inspired by our reported bug, one of the developers comments that they could refine a previous code revision to resolve the reported bug. For the Gcov bugs, the developers usually point out the root causes of the reported issues in the comments. According to the developers' response, we deduce that the bugs identified by Decov are meaningful and contribute well to the compiler community.

5.3 Complementarity of Decov

In this section, we investigate whether Decov is complementary to the state-of-the-art approaches (i.e., Cod [26] and C2V [27]). To this end, we first analyze how many bugs reported by Decov cannot be detected by existing approaches. Then, we investigate whether the bugs reported in prior studies can be also detected by Decov. Furthermore, we analyze the relationship between the identified inconsistencies between Decov and the existing techniques to analyze Decov 's potential ability to expose new bugs.

[3] https://github.com/llvm/llvm-project/milestone/1



(a) Gcov

(b) LLVM-cov

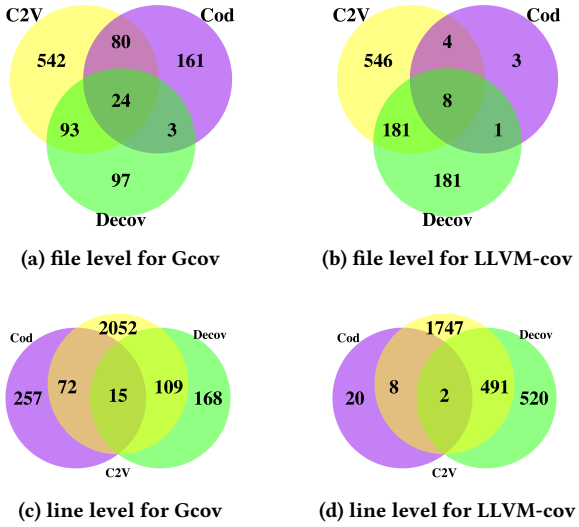
Figure 6: Confirmed bugs that affect corresponding release versions of Gcov and LLVM-cov.

Table 2: The number of bugs reported by Cod and C2V that can also be identified by Decov.

Bug Source	Gcov	LLVM-cov	Total
Cod	18/20	3/3	21/23
C2V	15/25	24/28	39/53

Comparisons via existing bug reports. To have a comprehensive understanding of Decov's capability to uncover bugs compared with the existing techniques, we conducted two investigations. First, we examined whether bugs identified by Decov can be also detected by the state-of-the-art techniques. For this purpose, we employed Cod and C2V to validate the profilers on the versions where we reported the corresponding bugs, using the provided test programs in the bug report. Table 1 presents the results, indicating that none of the bugs could be detected by the existing metamorphic testing technique Cod, while 12 out of the 19 bugs can be identified by the differential testing technique C2V. The results show that a number of bugs reported by Decov are difficult to identify by existing techniques. Second, we investigated how many bugs found by Cod and C2V can be detected by Decov. We employed the test programs from the bug reports provided by Cod and C2V as the subject test programs for the investigation. We checked whether our pre-defined HRs were violated within Decov. If a violation was detected, we consider the corresponding bug to be exposed by our tool Decov. As depicted in Table 2, Decov successfully uncovered 21 out of the 23 bugs identified by Cod and 39 out of the 53 bugs identified by C2V. These results collectively demonstrate the effectiveness of Decov in detecting coverage profiler bugs and its ability to identify the majority of bugs reported by previous studies.

Comparisons via inconsistent source code. In this section, we made a comparison for Decov with the state-of-the-art approaches based on the identified source code that violates respective relations. Worthy noting that the identified source code that triggers inconsistencies only indicates potential bugs rather than real bugs of coverage profilers due to the presence of false positives. To determine whether it is indeed a real bug, we often need to perform manual inspections. However, this comparison can be viewed as a



(a) file level for Gcov (b) file level for LLVM-cov

(c) line level for Gcov (d) line level for LLVM-cov

Figure 7: The relationship of test programs and code line set violating predefined relations for C2V, Cod, and Decov.

preliminary analysis of their potential bug finding capabilities. To have a fair comparison, we use the test programs in the test-suite of GCC 7.4.0 to evaluate the old version of Gcov and LLVM-cov, respectively. Here, the Gcov in GCC 7.5.0 and the LLVM-cov in LLVM 6.0.0 are used as the subject versions. Additionally, we employ the LLDB associated with the LLVM toolchain while the version of GDB is 8.1 since GDB tends to be separate from the GCC toolchain. It is unfair to make such comparisons for these three tools in the latest version as there are a number of bugs reported by C2V and Cod that have been fixed in the latest version of Gcov and LLVM-cov. Thus, we choose the old version of Gcov and LLVM-cov rather than the latest version. For each of the test programs in the test-suite of GCC 7.4.0, we use C2V, Cod, and Decov to check whether the predefined homogeneous or heterogeneous relations are violated, respectively. Table 3 presents the statistics of the test programs that violate the predefined heterogeneous relations in Decov, C2V, and Cod for Gcov and LLVM-cov. As shown in Table 3, there are 217 and 371 test programs, respectively, that violate at least one of the predefined heterogeneous relations in Decov for Gcov and LLVM-cov.

- **Comparison at the test program level.** Figure 7 (a) and (b) show the relationship between the potential bug-triggering test programs identified by C2V, Cod, and Decov for Gcov and LLVM-cov, respectively. Among them, 217 (4.4%) and 371 (7%) test programs violate the heterogeneous relations according to Decov for Gcov and LLVM-cov, respectively. Of these 217 and 371 test programs, 100 and 182 do not have an inconsistent coverage report pair according to C2V for Gcov and LLVM-cov. Besides, 190 and 362 of them do not violate the metamorphic relations defined in Cod. In other words, there may exist many potential bugs exposed by Decov that cannot be identified by the existing approaches, indicating the uniqueness of Decov. Besides, C2V identified 739 test programs violating those homogeneous relations, and Cod identified 268 and 16 test programs violating those homogeneous relations for Gcov and LLVM-cov, respectively. From Figure 7 (a) and (b), we can find that a large number of potential bugs found by C2V and Cod can be also found by Decov. Besides, it is evident that these three techniques can identify their own distinct violated programs. This demonstrates the complementary nature of these approaches.

Table 3: Test programs violating predefined relations for C2V, Cod, and Decov.

Technique	Profilers	Violated Reports	Consistent Reports
C2V	-	739	4674
Cod	Gcov	268	4441
	LLVM-cov	16	4717
Decov	Gcov	217	4674
	LLVM-cov	371	4905

- **Comparison at source line level.** Figure 7 (c) and (d) present the relationship between the source lines of code that violate respective relations according to C2V, Cod, and Decov for Gcov and LLVM-cov, respectively. At the source line level, we observed that C2V identifies a larger number of inconsistent source lines of code compared to the other two approaches. This is because each test program may contain multiple source lines that violate the predefined relations. Furthermore, Decov identified 168 and 520 source lines of code that solely violate our predefined relations for Gcov and LLVM-cov, respectively. This also demonstrates that Decov complements existing approaches.

6 DISCUSSION

In this section, we discuss how efficient Decov is, how developers fix code coverage bugs, and the threats to validity.

6.1 How efficient is Decov?

In this section, we discuss whether Decov's efficiency is acceptable in practice. To investigate this, we compare the time overhead of Decov with C2V and Cod. To have a fair comparison, we run C2V, Cod, and Decov over the same test programs in the test-suite of GCC 7.4.0. The results are shown in Table 4. In Table 4, the first and the second columns are respectively the technique and the profiler under test. The third column is the number of test programs that are successfully used for testing the profilers by different approaches. In the context of Cod, a test program is regarded as 'successfully used' when Cod is capable of generating compilable equivalent mutations for the given test program. The fourth and fifth column is the average and median time overhead among those test programs. From Table 4, we can find that the average time cost of C2V and Cod is around 0.35 seconds. However, the average time overhead of Decov for Gcov and LLVM-cov are 1.97 and 2.55 seconds on average, respectively. It is obvious that our approach is more expensive than existing approaches as it requires stepping test programs in debuggers. However, to improve the coverage profilers' correctness, we believe this overhead is considered acceptable.

6.2 How do developers fix code coverage bugs?

Bug 45194 in LLVM-cov is a resolved issue that was discovered by Decov. The bug involved incorrect coverage results due to the presence of a macro constant in an if expression. To rectify this

Table 4: Summarization of time overhead for C2V, Cod, and Decov over the programs in test-suite of GCC 7.4.0.

Technique	Profiler	#Test programs	Avg.(s)	Median(s)
C2V	Gcov vs LLVM-cov	5413	0.35	0.32
Cod	Gcov	2936	0.27	0.25
	LLVM-cov	2794	0.39	0.37
Decov	Gcov	4891	1.97	1.35
	LLVM-cov	5276	2.55	1.89

issue, the LLVM developer "Emit gap region after conditions when macro is present". A gap region typically refers to a section of code that is intentionally excluded from the coverage analysis. The bug fix involved modifications in seven files, resulting in 94 additions and 50 deletions. Specifically, a significant alteration of 47 LoC was made in `clang/lib/CodeGen/CoverageMappingGen.cpp`, the component responsible for generating instrumentation-based code coverage mappings for LLVM-cov. Similarly, in the case of fixing bug 105500 in Gcov, the GCC developer improved the accuracy of the mapping strategy for the determination of basic block-line associations. This modification resulted in a substantial change of 121 LoC across two files, allowing for more precise corresponding basic blocks and code lines as reflected in the .gcno files.

6.3 Threats to Validity

The first threat is the assumption that the hit count information from the debugger is accurate. In order to reduce this threat, we randomly selected 100 test programs from the GCC and LLVM test-suite and manually examine the accuracy of the hit count information produced by GDB and LLDB. We found that the hit count information was accurate for these programs. Additionally, upon manual inspection of all the violations found by Decov, we determined that all of them were caused by bugs in the code coverage profilers. Therefore, it is reasonable to believe that GDB and LLDB can provide accurate hit count debug information. The second threat is that our approach may not be able to generalize to other coverage criteria as we only focus on line coverage. The reasons why we only focus on line coverage are as follows: (1) line coverage is easy to obtained and widely applied to a large spectrum of software engineering tasks; (2) line coverage is the basis of other coverage criteria as they can be derived based on line coverage. In our experiment, we reformatted the test programs in test-suite of GCC and LLVM before feeding them to the coverage profilers and the debuggers. This can ensure that most of the statements indeed correspond to one single line. Currently, we are exploring how to generalize our approach to other coverage criteria such as branch coverage and condition coverage.

7 RELATED WORK

Testing for Coverage Profiler Validation. In the field of coverage profiler validation, there are two existing approaches have been proposed, utilizing differential and metamorphic testing techniques, respectively. Differential testing is a widely used technique for addressing the oracle problem in software testing [10, 13, 18, 19], including the profiler testing technique C2V [27]. C2V randomly generates programs, feeds them into different coverage profilers, and reports potential bugs if the resulting coverage reports are inconsistent. C2V has successfully detected 70 bugs for coverage profilers. The metamorphic testing technique, Cod [26], uncovers bugs by comparing the coverage statistics of path-equivalent programs. Cod has exposed a total number of 23 bugs so far. However, as mentioned in Section 2.4, both C2V and Cod still have certain limitations. To address these limitations, we propose to leverage the debugging information to validate code coverage profilers.

Testing based on Metamorphic Relation. The concept of metamorphic testing, initially proposed by T.Y. Chen in 1998, aims to alleviate the test oracle problem [4]. By examining whether the specific metamorphic relations hold across different inputs, this approach can effectively expose software bugs. Metamorphic testing has proven successful in various domains, such as software fuzzing [2, 7, 9, 15, 21, 24, 31], regression testing [5, 8], compiler testing [11], debugger testing [20], coverage profiler testing [26], machine learning classifications [25], and security [30]. In this paper, we extend the application of metamorphic testing by leveraging heterogeneous relations, supported by external debug information, to validate code coverage profilers.

8 CONCLUSION

Assuring the quality of coverage profilers is essential as the code coverage information is of great importance to many software engineering tasks. In this study, we propose to validate coverage profilers through heterogeneous testing. Specifically, we resort to the heterogeneous information provided by debuggers to validate coverage profilers. Our key insight is that the coverage statistics of a statement by a coverage profiler have heterogeneous relations with the hit count by a debugger. We implemented our idea as a prototype, Decov, and the evaluation results showed that it is complementary to existing approaches. Besides, it can also alleviate several limitations of the state-of-the-art approaches. We applied Decov to two most widely-used code coverage profilers and have exposed 21 new bugs. Of the 21 bugs, 19 bugs have been confirmed by developers.

9 DATA-AVAILABILITY STATEMENT

Decov and the scripts for the experiments are publicly available at: **https://doi.org/10.5281/zenodo.8275866** [28].

ACKNOWLEDGEMENT

We would like to extend our heartfelt gratitude to the developers of GCC and LLVM, especially Martin Liška, David Blaikie, Richard Biener, Xionghu Luo, Zequan Wu, and Vedant Kumar for their valuable feedback and efforts to address the bug reports we submitted. We are also grateful to all the anonymous reviewers for their insightful feedback. This work is mainly supported by the National Natural Science Foundation of China (No. 62072194, No. 62172205, No. 62002125), the Natural Science Foundation of Jiangsu Province (No. SBK2023022696), the CCF-Huawei Populus euphratica Fund (No. CCF-HuaweiSY2022007). Yuming Zhou and Maolin Sun are the corresponding authors.

REFERENCES

[1] Sanjeev Kumar Aggarwal and M. Sarath Kumar. 2002. Debuggers for Programming Languages. *The Compiler Design Handbook* (2002), 295–328.

[2] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based Greybox Fuzzing as Markov Chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi (Eds.). ACM, 1032–1043. https://doi.org/10.1145/2976749.2978428

[3] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based Greybox Fuzzing As Markov Chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. ACM, New York, NY, USA, 1032–1043. https://doi.org/10.1145/2976749.2978428

[4] Tsong Yueh Chen, Shing C Cheung, and Siu Ming Yiu. 1998. *Metamorphic testing: a new approach for generating next test cases.* Technical Report. Technical Report HKUST-CS98-01, Department of Computer Science, Hong Kong.

[5] Gordon Fraser and Andrea Arcuri. 2014. A Large-Scale Evaluation of Automated Unit Test Generation Using EvoSuite. *ACM Trans. Softw. Eng. Methodol.* 24, 2, Article 8 (Dec. 2014), 42 pages. https://doi.org/10.1145/2685612

[6] Markus Gaasedelen. 2017. *Lighthouse - A Code Coverage Explorer for Reverse Engineers.* Retrieved 2022-10-15 from https://github.com/gaasedelen/lighthouse

[7] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. 2018. CollAFL: Path Sensitive Fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*. 679–696. https://doi.org/10.1109/SP.2018.00040

[8] Milos Gligoric, Alex Groce, Chaoqiang Zhang, Rohan Sharma, Mohammad Amin Alipour, and Darko Marinov. 2015. Guidelines for Coverage-Based Comparisons of Non-Adequate Test Suites. *ACM Trans. Softw. Eng. Methodol.* 24, 4, Article 22 (Sept. 2015), 33 pages. https://doi.org/10.1145/2660767

[9] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. 2008. Automated Whitebox Fuzz Testing. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2008, San Diego, California, USA, 10th February - 13th February 2008*. The Internet Society. https://www.ndss-symposium.org/ndss2008/automated-whitebox-fuzz-testing/

[10] Christian Klinger, Maria Christakis, and Valentin Wüstholz. 2019. Differentially testing soundness and precision of program analyzers. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*, Dongmei Zhang and Anders Møller (Eds.). ACM, 239–250. https://doi.org/10.1145/3293882.3330553

[11] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 216–226. https://doi.org/10.1145/2594291.2594334

[12] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Randomized Stress-Testing of Link-Time Optimizers. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis* (Baltimore, MD, USA) *(ISSTA 2015)*. Association for Computing Machinery, New York, NY, USA, 327–337. https://doi.org/10.1145/2771783.2771785

[13] Daniel Lehmann and Michael Pradel. 2018. Feedback-directed differential testing of interactive debuggers. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu (Eds.). ACM, 610–620. https://doi.org/10.1145/3236024.3236037

[14] David Lo, Siau-Cheng Khoo, Jiawei Han, and Chao Liu. 2011. *Mining Software Specifications: Methodologies and Applications* (1st ed.). CRC Press, Inc., Boca Raton, FL, USA.

[15] Stefan Nagy and Matthew Hicks. 2019. Full-Speed Fuzzing: Reducing Fuzzing Overhead through Coverage-Guided Tracing. In *2019 IEEE Symposium on Security and Privacy (SP)*. 787–802. https://doi.org/10.1109/SP.2019.00069

[16] Sangmin Park, Richard W. Vuduc, and Mary Jean Harrold. 2010. Falcon: Fault Localization in Concurrent Programs. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1* (Cape Town, South Africa) *(ICSE '10)*. ACM, New York, NY, USA, 245–254. https://doi.org/10.1145/1806799.1806838

[17] Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding and Analyzing Compiler Warning Defects. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. ACM, New York, NY, USA, 203–213. https://doi.org/10.1145/2884781.2884879

[18] Maolin Sun, Yibiao Yang, Yang Wang, Ming Wen, Haoxiang Jia, and Yuming Zhou. 2023. SMT Solver Validation Empowered by Large Pre-trained Language Models. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Kirchberg, Luxembourg, September 11-15, 2023*. IEEE.

[19] Maolin Sun, Yibiao Yang, Ming Wen, Yongcong Wang, Yuming Zhou, and Hai Jin. 2023. Validating SMT Solvers via Skeleton Enumeration Empowered by Historical Bug-Triggering Inputs. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 69–81. https://doi.org/10.1109/ICSE48619.2023.00018

[20] Sandro Tolksdorf, Daniel Lehmann, and Michael Pradel. 2019. Interactive Metamorphic Testing of Debuggers. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 273–283. https://doi.org/10.1145/3293882.3330567

[21] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2017. Skyfire: Data-Driven Seed Generation for Fuzzing. In *2017 IEEE Symposium on Security and Privacy (SP)*. 579–594. https://doi.org/10.1109/SP.2017.23

[22] Ming Wen, Junjie Chen, Yongqiang Tian, Rongxin Wu, Dan Hao, Shi Han, and Shing-Chi Cheung. 2021. Historical Spectrum Based Fault Localization. *IEEE Trans. Software Eng.* 47, 11 (2021), 2348–2368. https://doi.org/10.1109/TSE.2019.2948158

[23] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-aware patch generation for better automated program repair. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 1–11. https://doi.org/10.1145/3180155.3180233

[24] Mingyuan Wu, Ling Jiang, Jiahong Xiang, Yanwei Huang, Heming Cui, Lingming Zhang, and Yuqun Zhang. 2022. One Fuzzing Strategy to Rule Them All. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 1634–1645. https://doi.org/10.1145/3510003.3510174

[25] Xiaoyuan Xie, Joshua Wing Kei Ho, Christian Murphy, Gail E. Kaiser, Baowen Xu, and Tsong Yueh Chen. 2011. Testing and validating machine learning classifiers by metamorphic testing. *J. Syst. Softw.* 84, 4 (2011), 544–558. https://doi.org/10.1016/j.jss.2010.11.920

[26] Yibiao Yang, Yanyan Jiang, Zhiqiang Zuo, Yang Wang, Hao Sun, Hongmin Lu, Yuming Zhou, and Baowen Xu. 2019. Automatic Self-Validation for Code Coverage Profilers. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering* (San Diego, California) *(ASE '19)*. IEEE Press, 79–90. https://doi.org/10.1109/ASE.2019.00018

[27] Yibiao Yang, Yuming Zhou, Hao Sun, Zhendong Su, Zhiqiang Zuo, Lei Xu, and Baowen Xu. 2019. Hunting for bugs in code coverage tools via randomized differential testing. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 488–498. https://doi.org/10.1109/ICSE.2019.00061

[28] Yang Yibiao, Sun Maolin, Wang Yang, Li Qingyang, Wen Ming, and Zhou Yuming. 2023. *ESEC/FSE 2023 Artifact for "Heterogeneous Testing for Coverage Profilers Empowered with Debugging Support"*. https://doi.org/10.5281/zenodo.8275866

[29] S. Yoo and M. Harman. 2012. Regression Testing Minimization, Selection and Prioritization: A Survey. *Softw. Test. Verif. Reliab.* 22, 2 (March 2012), 67–120. https://doi.org/10.1002/stv.430

[30] Zhi Quan Zhou and Liqun Sun. 2019. Metamorphic Testing of Driverless Cars. *Commun. ACM* 62, 3 (Feb. 2019), 61–67. https://doi.org/10.1145/3241979

[31] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. 2022. Fuzzing: A Survey for Roadmap. *ACM Comput. Surv.* 54, 11s, Article 230 (sep 2022), 36 pages. https://doi.org/10.1145/3512345

[32] Zhiqiang Zuo and Siau-Cheng Khoo. 2013. Mining Dataflow Sensitive Specifications. In *Formal Methods and Software Engineering - 15th International Conference on Formal Engineering Methods, ICFEM 2013, Queenstown, New Zealand, October 29 - November 1, 2013, Proceedings (Lecture Notes in Computer Science, Vol. 8144)*, Lindsay Groves and Jing Sun (Eds.). Springer, 36–52. https://doi.org/10.1007/978-3-642-41202-8_4

Received 2023-02-02; accepted 2023-07-27