Automatic Self-Validation for Code Coverage Profilers

Yibiao Yang, Yanyan Jiang, Zhiqiang Zuo, Yang Wang,
Hao Sun, Hongmin Lu, Yuming Zhou, and Baowen Xu
State Key Laboratory for Novel Software Technology
Department of Computer Science and Technology
Nanjing University, Nanjing, China
{yangyibiao, jyy, zqzuo}@nju.edu,cn, dz1933028@smail.nju.edu.cn
shqking@gmail.com, {hmlu, zhouyuming, bwxu}@nju.edu.cn

***Abstract*—Code coverage as the primitive dynamic program behavior information, is widely adopted to facilitate a rich spectrum of software engineering tasks, such as testing, fuzzing, debugging, fault detection, reverse engineering, and program understanding. Thanks to the widespread applications, it is crucial to ensure the reliability of the code coverage profilers.**

Unfortunately, due to the lack of research attention and the existence of testing oracle problem, coverage profilers are far away from being tested sufficiently. Bugs are still regularly seen in the widely deployed profilers, like gcov and llvm-cov, along with gcc and llvm, respectively.

This paper proposes `Cod`, an automated self-validator for effectively uncovering bugs in the coverage profilers. Starting from a test program (either from a compiler's test suite or generated randomly), `Cod` detects profiler bugs with zero false positive using a metamorphic relation in which the coverage statistics of that program and a mutated variant are bridged.

We evaluated `Cod` over two of the most well-known code coverage profilers, namely gcov and llvm-cov. Within a four-month testing period, a total of 196 potential bugs (123 for gcov, 73 for llvm-cov) are found, among which 23 are confirmed by the developers.

***Index Terms*—Code coverage, Metamorphic testing, Coverage profilers, Bug detection.**

I. Introduction

Profiling code coverage data [1] (e.g., executed branches, paths, functions, etc.) of the instrumented subject programs is the cornerstone of a rich spectrum of software engineering practices, such as testing [2], fuzzing [3], debugging [4]–[6], specification mining [7], [8], fault detection [9], reverse engineering, and program understanding [10]. Incorrect coverage information would severely mislead developers in their software engineering practices.

Unfortunately, coverage profilers themselves (e.g., gcov and llvm-cov) are prone to errors. Even a simple randomized differential testing technique exposed more than 70 bugs in coverage profilers [11]. The reasons are two-fold. Firstly, neither the application-end developers nor academic researchers paid sufficient attention to the testing of code coverage profilers. Secondly, automatic testing of coverage profilers is still challenging due to the lack of test oracles. During the code coverage testing, the oracle is supposed to constitute the rich execution information, e.g., the execution frequency of each code statement in the program under a given particular test case. Different from the functional oracle which usually can be obtained via the given specification, achieving the complete code coverage oracles turns out to be extremely challenging. Even though the programming experts can specify the oracle precisely, it requires enormous human intervention, making it impractical.

A simple differential testing approach `C2V` tried to uncover coverage bugs by comparing the coverage profiling results of the same input program over two different profiler implementations (e.g., gcov and llvm-cov) [11]. For instance, if gcov and llvm-cov provide different coverage information for the same statement of the profiled program, a bug is reported. Due to the inconsistency of coverage semantics defined by different profiler implementations, it is rather common that independently implemented coverage profilers exhibit different opinions on the code-line based statistics (e.g., the case in Figure 1) — this essentially contradicts the fundamental assumption of differential testing that distinct coverage profilers should output identical coverage statistics for the same input program.

Approach To tackle the flaws of the existing approach, this paper presents `Cod`, a fully automated *self-validator* of coverage profilers, based on the metamorphic testing formulation [12]. Instead of comparing outputs from two independent profilers, `Cod` takes a single profiler and a program $\mathcal{P}$ (either from a compiler's test suite or generated randomly) as input and uncovers the bugs by identifying the inconsistency of coverage results from $\mathcal{P}$ and its equivalent mutated variants whose coverage statistics are expected to be *identical*. The equivalent program variants are generated based on the assumption that *modifying unexecuted code blocks should not affect the coverage statistics of executed blocks under the identical profiler*, which should generally hold in a non-optimized setting[1]. This idea originates from EMI [2], a metamorphic testing approach which is targeted at compiler optimization bugs.

Specifically, assuming that the compiler is correct[2] and given a deterministic program $\mathcal{P}$ under profiling (either from a compiler's test suite or generated randomly) and fixate its input, `Cod` obtains a reference program $\mathcal{P}'$ by removing the unexecuted statements in $\mathcal{P}$. $\mathcal{P}'$ should strictly follow the same execution path as long as the coverage profiling data of $\mathcal{P}$

[1]According to the developers [13], coverage statistics are only stable under zero optimization level.

[2]We assume this because mis-compilations are rare.

is correct. Therefore, `Cod` asserts that the coverage statistics should be exactly the same over all unchanged statements in $\mathcal{P}$ and $\mathcal{P}'$. However, consider a line of code s in $\mathcal{P}$ and $\mathcal{P}'$, for which the same profiler reported different coverage results, i.e., $\mathcal{C}_{\mathcal{P}}(s) \neq \mathcal{C}_{\mathcal{P}'}(s)$ where $\mathcal{C}_{\mathcal{P}}(s)$ refers to the profiled runtime execution count of statement s in program $\mathcal{P}$. The execution count $\mathcal{C}_{\mathcal{P}}(s)$ is usually a nonnegative number except for a special value -1 indicating the unknown coverage information. This could happen when the coverage profiler failed to obtain the coverage information of statement s due to the information loss caused by the abstraction gap between the source code and the intermediate code transformed during compilation. Given $\mathcal{C}_{\mathcal{P}}(s) \neq \mathcal{C}_{\mathcal{P}'}(s)$, either of the two cases applies:

1) **(Strong Inconsistency)** $\mathcal{C}_{\mathcal{P}}(s) \geq 0 \ \wedge \ \mathcal{C}_{\mathcal{P}'}(s) \geq 0$, meaning that the coverage profiler reports the inconsistent coverage information. Definitely there is a bug in the profiler because $\mathcal{P}'$ should follow exactly the same execution path as $\mathcal{P}$ assuming that the coverage statistics of program $\mathcal{P}$ are correct.
2) **(Weak Inconsistency)** $\mathcal{C}_{\mathcal{P}}(s) = -1 \ \vee \ \mathcal{C}_{\mathcal{P}'}(s) = -1$, indicating an inaccurate statistics because that a non-instrumented line is actually executed in its equivalent. This is also for sure a bug because non-optimized coverage statistics should faithfully reflect the program's execution path.

The self-validator `Cod` fully exploits the inconsistencies between path-equivalent programs with zero false positive. `Cod` addresses the limitation of `C2V` in Section II-C and handles weak inconsistencies whereas `C2V` has to ignore all weak inconsistencies between independent profiler implementations to avoid being flooded by false positives. It is worth noting that such a technique of obtaining path-equivalent programs, firstly known as EMI, was proposed to validate the correctness of compiler optimizations [2]. We found that this idea is also powerful in the validation of coverage profilers. Nevertheless, `Cod` differs from EMI as we proposed the specialized mutation strategies to acquire the program variants, and adopted the results verification criterion in particular for testing coverage profilers. We defer to Section V for the comparison details.

Results We implemented `Cod` as a prototype and evaluated it on two popular coverage profilers, namely gcov and llvm-cov integrated with the compiler gcc and llvm, respectively. As of the submission deadline, a total of 196 potential bugs (123 for gcov, 73 for llvm-cov) are uncovered within 4 months, among which 23 bugs have already been confirmed by the developers. Promisingly, all the detected bugs are new bugs according to the developers' feedback.

Outline The rest of the paper is organized as follows. We introduce the necessary background and a brief motivation in Section II. Section III elaborates on the detailed approach, followed by the evaluation in Section IV. We discuss related work in Section V and conclude the paper in Section VI.

II. Background and Motivation

A. *Coverage Profilers*

Code coverage profiling data (each line of code's execution count in a program execution) is the foundation of a broad spectrum of software engineering practices. Code coverage is the most widely adopted criteria for measuring testing thoroughness, and is also widely used in the automation of software engineering tasks. For example, test input generation techniques leverage code coverage to guide search iterations [3]; fault localization techniques use code coverage to isolate potentially faulty branches [4]–[6].

To obtain code coverage statistics, a code coverage *profiler* maintains each source code line an execution counter and updates them along with the program execution. Specifically, given a program $\mathcal{P}$, a coverage profiler runs $\mathcal{P}$ and outputs each line of code $s \in \mathcal{P}$ a number $\mathcal{C}_{\mathcal{P}}(s) = n$, indicating that s was executed n times. A special value $n = -1$ indicates that the profiler provides no coverage information for this line.

Where should a profiler report a coverage statistics for a line of code s (i.e., whether $\mathcal{C}_{\mathcal{P}}(s) = -1$) is not well defined. Code transformations (e.g., expansion of macros, or compilation from source code to intermediate code) and optimizations may lead to $\mathcal{C}_{\mathcal{P}}(s) = -1$ for a line, and different profilers generally have different opinions upon which lines would have $\mathcal{C}_{\mathcal{P}}(s) \geq 0$. Later we see that this is a major limitation of existing techniques for validating coverage profilers.

B. *Validating Coverage Profilers*

Validating the correctness a coverage profiler is challenging because it is labor-intensive to obtain the ground truth of coverage statistics. Though we have large number of test inputs (any program used to test a compiler also works in testing a profiler), lacking of a test oracle became the problem.

The only known technique to uncover coverage profiler bugs is `C2V` which is based on differential testing [11]. Given a program $\mathcal{P}$ under profiling, `C2V` profiles it using two independently implemented profilers to obtain for each statement s the coverage statistics $\mathcal{C}_{\mathcal{P}}(s)$ and $\mathcal{C}'_{\mathcal{P}}(s)$. When $\mathcal{C}_{\mathcal{P}}(s) \neq \mathcal{C}'_{\mathcal{P}}(s)$, an inconsistency is found. When $\mathcal{C}_{\mathcal{P}}(s) \geq 0 \wedge \mathcal{C}'_{\mathcal{P}}(s) \geq 0$ (a strong inconsistency), `C2V` reports it as a bug candidate and uses clustering to filter out potential false positives and duplicates.

C. *Limitations of Differential Testing*

Though being effective in uncovering coverage profiler bugs, differential testing also has the following major limitations:

First, *differential testing cannot be applied when there is only a single coverage profiler*. This is the case for many mainstream programming languages (e.g, Python and Perl).

Second, *differential testing requires heavy human efforts on analyzing the reports* because it is hard to determine which one is faulty when two profilers disagree on the statistics of a line of code.

Third, *differential testing miss many potential bugs on weak inconsistencies*. Two profilers can have inconsistent (but both

```
✓1:    1:int main()
-1:    2:{
-1:    3:  switch (8)
-1:    4:  {
-1:    5:    case 8:
✓1:    6:      break;
-1:    7:    default:
-1:    8:      abort ();
-1:    9:      break;
-1:   10:  }
✓1:   11:  return 0;
-1:   12:}
```

(a) $\mathcal{C}_\mathcal{P}$ (gcov)

```
 1|  -1|int main()
 2| ✓1|{
 3| ✓1|  switch (8)
 4| ✓1|  {
 5| ✓1|    case 8:
 6| ✓1|      break;
 7| ✓1|    default:
 8| ×0|      abort ();
 9| ✓1|      break;
10| ✓1|  }
11| ✓1|  return 0;
12| ✓1|}
```

(b) $\mathcal{C}_\mathcal{P}$ (llvm-cov)

```
 1|  -1|int main()
 2| ✓1|{
 3| ✓1|  switch (8)
 4| ✓1|  {
 5| ✓1|    case 8:
 6| ✓1|      break;
 7| ✓1|    default:
 8| ×0|      ; // abort ();
 9| ×0|      break;
10| ✓1|  }
11| ✓1|  return 0;
12| ✓1|}
```

(c) $\mathcal{C}_{\mathcal{P}\setminus\{s_8\}\cup\{s'_8\}}$ (llvm-cov)

Fig. 1. The bug case of LLVM #41821. llvm-cov incorrectly reported that the `break` in Line 9 is executed. This bug cannot be detected by differential testing [11] because gcov does not provide coverage statistics for Line 9. Visual conventions of coverage statistics: For a line of code s, gcov and llvm-cov output coverage statistics $c_\mathcal{P}(s)$ in the first and second column, respectively. A `-1` denotes that the profiler does not provide coverage information of s. A check mark or cross mark followed by a number n denotes that $c_\mathcal{P}(s) = n$.

correct) interpretations over the coverage statistics. Section IV reveal that 92.9% of inconsistencies found by differential testing are weak, i.e., $\mathcal{C}_\mathcal{P}(s) \neq \mathcal{C}'_\mathcal{P}(s)$ with $\mathcal{C}_\mathcal{P}(s) = -1 \vee \mathcal{C}'_\mathcal{P}(s) = -1$. Our motivating example in Figures 1 (a)–(b) showed that for 10/12 lines, exactly one of gcov or llvm-cov provides no coverage information. Unfortunately, `C2V` has to ignore *all* of such weak consistencies (i.e., not report any of them as a bug) because the vast majority of them are attributed to the feature of independently implemented profilers.

Finally, *differential testing reports false positive even when two profilers report strongly inconsistent coverage statistics*, because two profilers may disagree over the definition of the execution count of a line, e.g., whether the initialization code of a `for` loop counts for one time of execution.

D. Motivation

The key observation leading to automatic self-validation of a single profiler is that *changing unexecuted statements in a program should not affect the coverage statistics*. Take the program in Figure 1 (a)–(b) as an example. Suppose that we comment out the function call in Line 8 of $\mathcal{P}$ and obtain $\mathcal{P}' = \mathcal{P} \setminus \{s_8\} \cup \{s'_8\}$ as shown in Figure 1 (c) We assert that unchanged statements should have identical coverage statistics, i.e.,

$$\forall s \in \mathcal{P} \cap \mathcal{P}'.\ \mathcal{C}_\mathcal{P}(s) = \mathcal{C}_{\mathcal{P}'}(s),$$

reasonably assuming that:

1) $\mathcal{P}$ is deterministic, contains no undefined behavior, and does not depend on external environment;
2) the coverage statistics is correct; and
3) the executions of $\mathcal{P}$ and $\mathcal{P}'$ are consistent with their semantics.

Since we only remove "unexecuted" statements reported by a coverage profiler, $\mathcal{P}$ and $\mathcal{P}'$ should be semantically equivalent. Furthermore, a profiler (particularly under minimal optimization) should be self-consistent in terms of which statement should have a coverage statistics. Therefore, if there is an inconsistency $\mathcal{C}_\mathcal{P}(s) \neq \mathcal{C}_{\mathcal{P}'}(s)$, no matter whether it is a strong or weak inconsistency, either of the above assumptions is violated. It turns out that we should blame

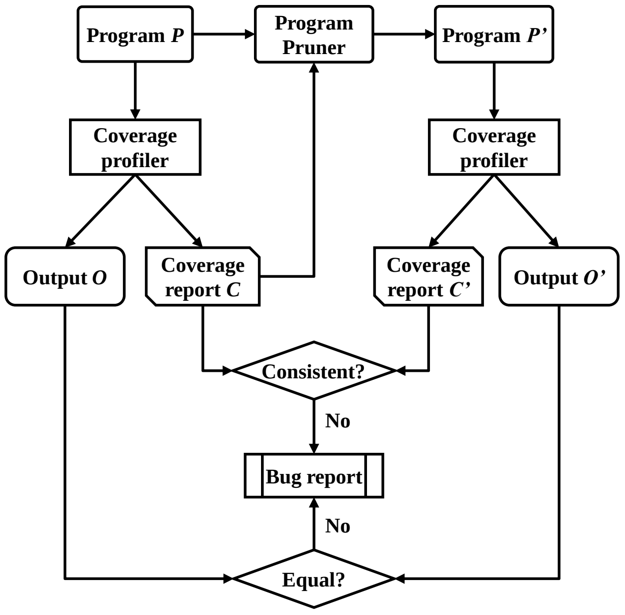


Fig. 2. The framework of `Cod`

the profiler because we have full control over $\mathcal{P}$ (thus easily to guarantee it is deterministic) and there is little chance that the compiler/hardware is defective.

In the motivating example, $\mathcal{C}_\mathcal{P}(s_9) \neq \mathcal{C}_{\mathcal{P}'}(s_9)$ revealed a previously unknown profiler bug in which llvm-cov incorrectly reported an unexecuted `break` as being executed once. This bug case is missed by differential testing (particularly, `C2V`) because all inconsistencies between $\mathcal{C}_\mathcal{P}$ (gcov) and $\mathcal{C}_\mathcal{P}$ (llvm-cov) are weak (Lines 1–5, 7–10, and 12). Generally, weak inconsistencies between different compiler implementations indicate different compilation strategies (thus do not indicate a profiler bug) and should not be reported by `C2V`.

III. Approach

A. Metamorphic Testing

A test oracle is a mechanism for determining whether a test has passed or failed. Under certain circumstances, however, the oracle is not available or too expensive to achieve.

This is known as the *oracle problem* [14]. For example, in compiler testing, it is not easy to verify whether the generated executable code by a compiler is functionally equivalent to the given source code. Even if the oracle is available, manually checking the oracle results is tedious and error-prone [15], [16]. As a matter of fact, the oracle problem has been "one of the most difficult tasks in software testing" [16].

Metamorphic testing (MT) was coined by T.Y. Chen in 1998 [12], which can be exploited to alleviate the oracle problem. Based on the existing successful test cases (that have not revealed any failure, such as running too long or returning abnormal values), MT generates *follow-up* test cases by making reference to *metamorphic relations* (MR), which are the necessary properties of the target function or algorithm in terms of multiple inputs and their expected outputs. Let us consider a program $\mathcal{P}$ implementing function $\mathcal{F}$ on domain $\mathcal{D}$. Let t be an initial successful test case, i.e. $t \in \mathcal{D}$ and the execution output $\mathcal{P}(t)$ equals to the expected value $\mathcal{F}(t)$. MT can be applied to generate a follow-up test case $t' \in \mathcal{D}$ base on t and a pre-defined MR (i.e., metamorphic relation). For program $\mathcal{P}$, a MR is a property of its target function $\mathcal{F}$. For instance, suppose $\mathcal{F}(x) = sin(x)$, then the property $sin(\pi - x) = sin(x)$ is a typical MR with respect to $\mathcal{F}$. Hence, given a successful test case, say $t = 1.2$, MT generates its follow-up test case $t' = \pi - 1.2$, and then runs the program over t'. Finally, two outputs (i.e., $\mathcal{P}(t)$ and $\mathcal{P}(t')$) are checked to see if they satisfy the expected relation $\mathcal{P}(t) = \mathcal{P}(t')$. If the identity does not hold, a failure manifests.

In our work, we apply MT to the validation of code coverage profilers. Each program becomes a test case fed to the profilers. Given a program $\mathcal{P}$ and the code coverage profiler under testing, running $\mathcal{P}$ with an input i would produce the execution output $\mathcal{O}$ and a coverage report $\mathcal{C}$ from the coverage profiler, The coverage report $\mathcal{C}$ records which lines of code are executed (or unexecuted) and how many times are executed exactly. Note that this program $\mathcal{P}$ is the initial test case according to the notions of MT. A follow-up test program $\mathcal{P}'$ can be generated based on the following MR:

Given a program, the unexecuted code can be eliminated since these code has no impact with the execution output or the coverage report specifically for the executed part.

In other words, by taking advantage of the coverage information $\mathcal{C}$, we generate $\mathcal{P}'$, a functionally equivalent variant of $\mathcal{P}$ by removing the un-executed code statements of $\mathcal{P}$. We run $\mathcal{P}'$ on the same input i and obtain the output $\mathcal{O}'$ and coverage results $\mathcal{C}'$, accordingly. A bug can then be discovered if 1) the execution output $\mathcal{O}$ is not equal to the new one $\mathcal{O}'$, or 2) there exists inconsistency between the coverage information for executed code inside $\mathcal{C}$ and $\mathcal{C}'$. Figure 2 shows our framework for the self-validation of code coverage profilers.

B. Our Algorithm

Based on our formulation, we implemented a tool Cod for detecting bugs in C code coverage profilers. Cod consists of three main steps: (1) extracting output and coverage information of a given test program, (2) generating equivalent variants based on code coverage report, and (3) comparing the outputs and coverage results to uncover bugs.

Algorithm 1: Cod's process for coverage tool validation

```
   Data: The profiler $\mathcal{T}$ under test, the program $\mathcal{P}$, the input $i$
   Result: reported bugs
1  begin
       /* Step 1: Extract output and coverage
          information                                    */
2      $\mathcal{P}_{exe}$ ← compile($\mathcal{P}$)
3      $\mathcal{O}$ ← getOutput(execute($\mathcal{P}_{exe}$, $i$))
4      $\mathcal{C}$ ← $\mathcal{T}$.extractCoverage(execute($\mathcal{P}_{exe}$, $i$))
       /* Step 2: Generate variants via
          transformation                                 */
5      $\mathcal{P}'$ ← genVariant($\mathcal{P}$, $\mathcal{C}$)
6      $\mathcal{P}'_{exe}$ ← compile($\mathcal{P}'$)
7      $\mathcal{O}'$ ← getOutput(execute($\mathcal{P}'_{exe}$, $i$))
8      $\mathcal{C}'$ ← $\mathcal{T}$.extractCoverage(execute($\mathcal{P}'_{exe}$, $i$))
       /* Step 3: Compare outputs and reports            */
       // First Stage
9      if $\mathcal{O} \neq \mathcal{O}'$ then
10         reportBug()
       // Second Stage
11     else if inconsistent($\mathcal{C}$, $\mathcal{C}'$) then
12         reportBug()

   /* Generate a variant for program $\mathcal{P}$ under
      coverage $\mathcal{C}$                             */
13 Function genVariant(Program $\mathcal{P}$, Coverage $\mathcal{C}$)
14     $\mathcal{P}'$ ← $\mathcal{P}$
15     foreach $s \in$ getStmts($\mathcal{P}$) $\wedge\ \mathcal{C}_{\mathcal{P}}(s) = 0$ do
16         $\mathcal{P}'$.delete($s$)
17     if isCompiable($\mathcal{P}'$) then
18         return $\mathcal{P}'$
19     else
20         genVariant($\mathcal{P}$, $\mathcal{C}$)

   /* Check whether coverages is inconsistent        */
21 Function inconsistent(Coverage $\mathcal{C}$, Coverage $\mathcal{C}'$)
22     foreach $s \in \mathcal{C} \wedge s \in \mathcal{C}'$ do
23         if $\mathcal{C}_{\mathcal{P}}(s) \neq \mathcal{C}'_{\mathcal{P}'}(s)$ then
24             return True
25     return False
```

Algorithm 1 is the main process of Cod. At the first place, Cod compiles the program $\mathcal{P}$ and profiles the execution information with input i to collect: (1) the output $\mathcal{O}$ (which may correspond to a return value or an exit code) and (2) the code coverage information $\mathcal{C}$ (Lines 2 - 4). It then generates the variant $\mathcal{P}'$ with respect to program $\mathcal{P}$ (Line 5) and collects the respective output $\mathcal{O}'$ and code coverage $\mathcal{C}'$ (Lines 6 - 8). Finally, it compares the outputs together with the code coverage reports to validate the coverage profiler. A potential bug in $\mathcal{T}$ is reported if any of them is inconsistent (Lines 9 - 12). We discuss each step in details as follows.

Extracting Coverage Information For each test program $\mathcal{P}$, we first compile it with particular options to generate the executable binary $\mathcal{P}_{exe}$. The options enables the compiler to integrate the necessary instrumentation code into the executable. While executing $\mathcal{P}_{exe}$ with input i, we obtain the output O of the program. Meanwhile, the code coverage report $\mathcal{C}$ can also be readily extracted by the coverage profiler $\mathcal{T}$. Each code coverage report contains the lines of code executed and unexecuted in the test program $\mathcal{P}$ under the input i. Those statements marked as unexecuted will be randomly pruned for the purpose of generating $\mathcal{P}$'s equivalent variants, which will be discussed shortly. `Cod` implemented the supports for both gcov and llvm-cov. Take gcov as an example, `Cod` extracts coverage information by compiling the program $\mathcal{P}$ with the flag: "`-O0 --coverage`" under gcc. It tells the compiler to instrument additional code in the object files for generating the extra profiling information at runtime. `Cod` then runs the executable binary under input i to produce coverage report for the program $\mathcal{P}$.

Generating Variants via Transformation Based on the coverage report $\mathcal{C}$ for the original program $\mathcal{P}$, its variants are generated (Line 5). `Cod` produces the variants by stochastically removing unexecuted program statements from the original program $\mathcal{P}$. Specifically, for each of these removable lines of code, we made a random choice. As such, we obtain a number of variants $\mathcal{P}'$ that should be equivalent to the original program. The function `genVariant` (Lines 13 - 20) describe `Cod`'s process for generating equivalence mutants via transformation. Note that stochastically removing unexecuted program statements would lead to many uncompilable mutants. Only the compilable ones are returned by `genVariant` (Line 17).

Comparing the Outputs and Coverage Reports Having the outputs and the coverage reports for the orginal program $\mathcal{P}$ and its variants $\mathcal{P}'$, we detect bugs in the code coverage tool $\mathcal{T}$ by checking the existence of inconsistency. More specifically, We first compare the outputs of $\mathcal{P}$ and $\mathcal{P}'$. If they are not identical, a potential bug would be reported in the code coverage tool. Otherwise, the code coverage reports are further compared to seeking for inconsistencies. Note that only the code coverage of the common lines of code between the programs $\mathcal{P}$ and $\mathcal{P}'$ (i.e. those lines of code left in the variant program), will be considered for comparison. If the code coverage reports is not consistent over the common lines, a potential bug is reported as well (Lines 9–12).

C. Illustrative Examples

In the following, we take our reported three concrete bug examples to illustrate how `Cod` works. Three bugs are newly discovered by `Cod` and confirmed by the GCC developers.

Bug Example Exposed by Different Outputs Figure 3 shows a real bug example exposed via different outputs of two "equivalent" programs in gcov [17], a C code coverage tool integrated in GCC [18]. Figure 3 (a) and (b) are the code coverage reports produced by gcov for the original program $\mathcal{P}$ and its equivalent program $\mathcal{P}'$ (by removing an unexecuted Line 8), respectively. Note that all the test programs are reformatted for presentation. As can be seen, a code coverage report is an annotated version of the source code augmented with the execution frequency of each line. The first and second column list the execution frequency and the line number. The frequency number "-1" in the first column indicates that the coverage information is unknown.

In this example, we first utilize gcc to compile the program $\mathcal{P}$ and then execute it to produce the output and coverage report (shown as Figure 3 (a)). Note that the output in this case is 0. According to the original code coverage report of $\mathcal{P}$, `Cod` decides to remove the 6th statement from the original program, resulting in an equivalent program $\mathcal{P}'$ shown as Figure 3(b). Next, we compile and execute $\mathcal{P}'$ to get the new output and coverage report. Here, the output turns to be 1.

Since the outputs of these two program are not equal, $\mathcal{P}$ and $\mathcal{P}'$ are somehow not equivalent, meaning that we actually deleted some executed code. The code coverage tool wrongly marked some executed statements as not executed. A potential bug is identified. We reported this bug to Bugzilla. The gcov developers quickly confirmed and fixed it.

Bug Example Exposed by Strongly Inconsistent Coverage Figure 4 illustrates another real bug example uncovered by strongly inconsistent code coverage reports between the program and its "equivalence" variant. Figure 4 (a) shows the coverage report for $\mathcal{P}$. We can read from it that Line 10 is not executed at all (i.e., the execution count is 0). `Cod` prunes Line 10 to generate the equivalent program $\mathcal{P}'$. After compiling and executing $\mathcal{P}'$, another coverage report shown as Figure 4 (a) is produced. As can be seen, there exists an strong inconsistency in term of the execution frequency of Line 6, indicating a potential bug. This bug is submitted and confirmed already by gcov developers.

Bug Example Exposed by Weakly Inconsistent Coverage Figure 5 presents another confirmed real bug example found via the weakly inconsistent code coverage reports between the program and its equivalent variant. In Figure 5 (a), Line 6 in $\mathcal{P}$ is not executed (i.e., the execution count is 0). `Cod` gets rid of Line 6 to generate the equivalent program $\mathcal{P}'$. Upon compiling and executing $\mathcal{P}'$, another coverage report shown as Figure 5 (a) is generated. Apparently, the weakly inconsistency with respect to the execution frequency of Line 5 appears, indicating a potential bug.

IV. Evaluation

This section presents our evaluation of `Cod`. We evaluated `Cod` using the most popular practical code coverage profilers: gcov and llvm-cov and a set of testing programs for testing compilers, and compared the results with existing differential technique `C2V` [11].

A. Evaluation Setup

Profilers for Validation We evaluated `Cod` using the latest versions of gcov and llvm-cov, the most popular two code coverage profilers of C programs, as our experimental subjects. Both profilers are:

1) *popular* in the software engineering community;

```
  -1:   1:#include <stdio.h>
  -1:   2:int *p=0, a=0, b=2;
  ✓1:   3:int *foo() {
  ✓1:   4:  int *r = (int *)1;
  -1:   5:  while (1) {
  ×0:   6:    r = (int)(a+p) & ~1;
  ✓1:   7:    if (a < b) return r;
  -1:   8:  }
  -1:   9:  return r;
  -1:  10:}
  ✓1:  11:void main () {
  ✓1:  12:  int *r = foo();
  ✓1:  13:  printf("%d\n", r);
  ✓1:  14:}
```

(a) $\mathcal{C}_{\mathcal{P}}$ (gcov, output: 0)

```
  -1:   1:#include <stdio.h>
  -1:   2:int *p=0, a=0, b=2;
  ✓1:   3:int *foo() {
  ✓1:   4:  int *r = (int *)1;
  -1:   5:  while (1) {
  -1:   6:    // r = (int)(a+p) & ~1;
  ✓1:   7:    if (a < b) return r;
  -1:   8:  }
  -1:   9:  return r;
  -1:  10:}
  ✓1:  11:void main () {
  ✓1:  12:  int *r = foo();
  ✓1:  13:  printf("%d\n", r);
  ✓1:  14:}
```

(b) $\mathcal{C}_{\mathcal{P}\setminus\{s_6\}\cup\{s'_6\}}$ (gcov, output: 1)

Fig. 3. A real bug example exposed by Cod via different outputs. This is Bug #89675 of gcov 8.2.0. In (a), Line #6 is marked as not executed; (b) is the "equivalent" program by deleting Line #6 from the original program in (a). The outputs of these two "equivalent" programs are not identical, indicating a bug in gcov 8.2.0.

```
   ✓1:   1:int foo() {
   ✓1:   2:  int h=2, f=1, k=0;
   ✓1:   3:  int y=18481, x=y;
   ✓1:   4:  if(y!=0 && (k<=x>>4)) {
  ✓1*:   5:    h=y>0 ? 2:1;
   ✓2:   6:    if (f) {
   ✓1:   7:      h^=3;
   -1:   8:    }
   -1:   9:  } else {
   ×0:  10:    h = 0;
   -1:  11:  }
   ✓1:  12:  return h;
   -1:  13:}
   ✓1:  14:void main() { foo(); }
```

(a) $\mathcal{C}_{\mathcal{P}}$ (gcov)

```
   ✓1:   1:int foo() {
   ✓1:   2:  int h=2, f=1, k=0;
   ✓1:   3:  int y=18481, x=y;
   ✓1:   4:  if(y!=0 && (k<=x>>4)) {
  ✓1*:   5:    h=y>0 ? 2:1;
   ✓1:   6:    if (f) {
   ✓1:   7:      h^=3;
   -1:   8:    }
   -1:   9:  } else {
   -1:  10:    // h = 0;
   -1:  11:  }
   ✓1:  12:  return h;
   -1:  13:}
   ✓1:  14:void main() { foo(); }
```

(b) $\mathcal{C}_{\mathcal{P}\setminus\{s_{10}\}\cup\{s'_{10}\}}$ (gcov)

Fig. 4. A real bug example discovered by Cod, with confirmed bug id #89470 of gcov 8.2.0. When the unexecuted Line #10 is pruned from the original program in (a), the code coverage of Line #6 is inconsistent between that of the original program and the new program in (b), which indicates a bug. A star after a number in Line #5 denotes that this number may be inaccurate.

```
  ✓1:   1:void foo(int x, unsigned u) {
  ✓1:   2:  if ((1U << x) != 64
  ✓1:   3:      || (2 << x) != u
  -1:   4:      || (1 << x) == 14
  ✓1:   5:      || (3 << 2) != 12)
  ×0:   6:    __builtin_abort ();
  ✓1:   7:}
  ✓1:   8:int main() {
  ✓1:   9:  foo(6, 128U);
  ✓1:  10:  return 0;
  -1:  11:}
```

(a) $\mathcal{C}_{\mathcal{P}}$ (gcov)

```
  ✓1:   1:void foo(int x, unsigned u) {
  ✓1:   2:  if ((1U << x) != 64
  ✓1:   3:      || (2 << x) != u
  -1:   4:      || (1 << x) == 14
  -1:   5:      || (3 << 2) != 12)
  -1:   6:    ; // __builtin_abort ();
  ✓1:   7:}
  ✓1:   8:int main() {
  ✓1:   9:  foo(6, 128U);
  ✓1:  10:  return 0;
  -1:  11:}
```

(b) $\mathcal{C}_{\mathcal{P}\setminus\{s_5\}\cup\{s'_5\}}$ (gcov)

Fig. 5. A real bug example discovered by Cod, with confirmed bug id #90439 of gcov 9.0. When the unexecuted Line #5 is pruned from the original program in (a), the code coverage of Line #5 is weakly inconsistent between that of the original program and the new program in (b).

2) *integrated* in the most widely used production compilers, i.e. GCC and Clang;
3) *extensive validated* by existing research, both for the compilers and the profilers.

Following the existing research [11], we use the default complier flags to obtain coverage report for gcov and llvm-cov under zero-level optimization. Given a piece of source code `test.c`, the following commands are used to produce the coverage report `test.c.gcov`:

```
gcc -O0 --coverage -o test test.c
./test
gcov test.c
```

For llvm-cov, we use the following commands to produce the coverage report `test.c.lcov`:

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

Evaluation Steps To run either differential testing or Cod, we obtain code coverage statistics for the 26,530 test programs

TABLE I
STATISTICS OF BUG-TRIGGERING TEST PROGRAMS.

Profilers	Different Outputs	Inconsistent Reports	
		Strong	Weak
gcov	1	69	54
llvm-cov	0	62	11

in the test-suite shipped with the latest gcc release (7.4.0) and 5,000 random programs generated by csmith [19]. All evaluated programs contain neither external environmental dependency nor undefined behavior. We run Cod over all the test programs and collect all reported inconsistencies for a manual inspection. We also compare these results with the state-of-the-art differential testing technique C2V [11].

Testing Environment We evaluated gcov shipped with the latest version of gcov (until gcc 9.0.1-20190414) and llvm-cov (until llvm 9.0.0-svn358899) during our experiments. All experiments were conducted on a hexa-core Intel(R) Core(TM) CPU@3.20GHz virtual machine with 10GiB of RAM running Ubuntu Linux 18.04.

B. Experimental Results

Inconsistent Reports For each of the test cases in our testbed, only one variant was generated by using Cod for the validation. The only variant is generated by removing all the unexecuted statements reported by coverage profilers from the original test cases. It is obvious that generating more variants for each test program may trigger more inconsistencies over the test programs and probably detect more bugs in those coverage profilers. Table I shows the statistics of bug-triggering test programs over two code coverage profilers under test, i.e., gcov and llvm-cov. Column 2 refers to the total number of the pairs of test program with its variant, which can lead to different execution outputs, and Column 3 shows the total number that can impose inconsistent coverage reports.

The single case in which the variant outputs a different value (Figure 3) is due to the incorrect coverage statistics causing Cod to create functionally different "equivalent" mutated variants. Others inconsistencies also due to profiler bugs, which are discussed as follows.

Bugs Found We manually inspected all cases and found that *all* reported (strong and weak) inconsistencies revealed defects in the profiler. By far, we reported a total of 26 bugs to the developers of gcov and llvm-cov. The manual classification and reporting of profiler bugs is still on-going. We believe that more bugs will be reported in the future.

23/26 bugs are confirmed[3] by the developers as listed in Table II. One of the remaining three is still in the pending

[3]Consistent with C. Sun et al's [20] and V. Le et al's [21] studies, due to the bug management process of LLVM is not as organized as that of GCC, if a llvm-cov bug report has been CCed by Clang developers and there is no objection in the comments, we label the bug as confirmed. In addition, as stated by developers, if someone does not close the reported bug as "invalid", then the bug is real in LLVM Bugzilla.

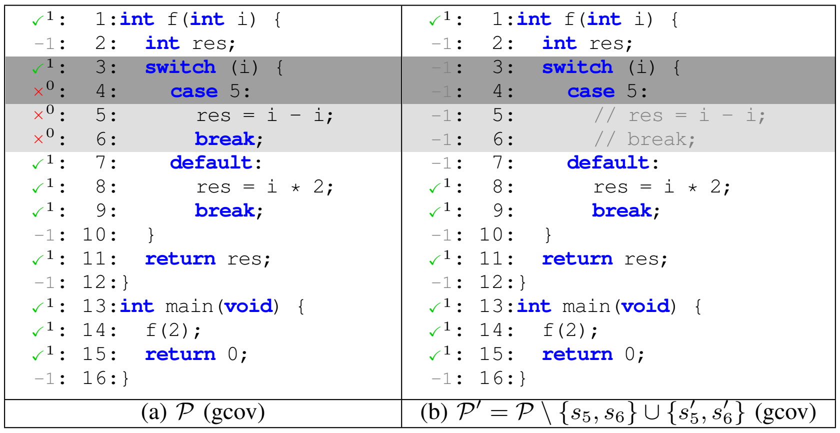
TABLE II
LIST OF CONFIRMED OR FIXED BUGS. PN DENOTES A NORMAL PRIORITY. DIFFTEST DENOTES WHETHER THE BUG CAN BE FOUND BY A DIFFERENTIAL TESTING.

ID	Profiler	Bugzilla ID	Priority	Status	Type	DiffTest
1	gcov	88913	P3	Fixed	Wrong Freq.	✓
2	gcov	88914	P3	Fixed	Wrong Freq.	✓
3	gcov	88924	P5	New	Wrong Freq.	✓
4	gcov	88930	P3	Fixed	Wrong Freq.	✓
5	gcov	89465	P3	Fixed	Missing	×
6	gcov	89467	P3	Fixed	Wrong Freq.	✓
7	gcov	89468	P5	New	Wrong Freq.	×
8	gcov	89469	P5	New	Wrong Freq.	✓
9	gcov	89470	P5	New	Wrong Freq.	✓
10	gcov	89673	P5	New	Spurious	×
11	gcov	89674	P5	New	Spurious	×
12	gcov	89675	P3	Fixed	Missing	×
13	gcov	90023	P5	New	Spurious	×
14	gcov	90054	P3	Fixed	Missing	✓
15	gcov	90057	P3	Fixed	Wrong Freq.	✓
16	gcov	90066	P5	New	Wrong Freq.	×
17	gcov	90091	P3	New	Wrong Freq.	✓
18	gcov	90104	P3	New	Wrong Freq.	×
19	gcov	90425	P5	New	Wrong Freq.	×
20	gcov	90439	P3	New	Missing	×
21	llvm-cov	41051	PN	New	Wrong Freq.	✓
22	llvm-cov	41821	PN	New	Spurious	×
23	llvm-cov	41849	PN	New	Missing	×

confirmation state, one was marked as duplicate, and only one was rejected by the developer (gcov #90438). This rejected case is controversial because gcc is performing optimization even under the zero optimization levels (as shown in Figure 6), which may mislead a developer or an automated tool that are based on the branch information in the coverage statistics.

Following the notions from C2V, code coverage bugs inside coverage profilers can categorized as *Spurious Marking*, *Missing Marking*, and *Wrong Frequency*. As shown in Column 6 of Table II, we can find that Cod is able to detect all three types of bugs in coverage profilers. 14 bugs belong to *Wrong Frequency*, 5 bugs belong to *Missing Marking*, and the rest 4 bugs is *Spurious*. Besides, most of bugs are *Wrong Frequency* bugs, i.e., the execution frequencies is wrongly reported.

Among all these bugs, nearly half (12/26) cannot be manifested by differential testing. Considering that differential testing leverages the coverage statistics of an independent profiler implementation (which produces correct coverage information in all these cases, and thus differential testing is essentially comparing with a golden version) while Cod is merely self-validation, we are expecting Cod to be effective and useful in finding code coverage profiler bugs.



```
✓1:  1:int f(int i) {
-1:  2:  int res;
✓1:  3:  switch (i) {
×0:  4:    case 5:
×0:  5:      res = i - i;
×0:  6:      break;
✓1:  7:    default:
✓1:  8:      res = i * 2;
✓1:  9:      break;
-1: 10:  }
✓1: 11:  return res;
-1: 12:}
✓1: 13:int main(void) {
✓1: 14:  f(2);
✓1: 15:  return 0;
-1: 16:}
```

(a) $\mathcal{P}$ (gcov)

```
✓1:  1:int f(int i) {
-1:  2:  int res;
-1:  3:  switch (i) {
-1:  4:    case 5:
-1:  5:      // res = i - i;
-1:  6:      // break;
-1:  7:    default:
✓1:  8:      res = i * 2;
✓1:  9:      break;
-1: 10:  }
✓1: 11:  return res;
-1: 12:}
✓1: 13:int main(void) {
✓1: 14:  f(2);
✓1: 15:  return 0;
-1: 16:}
```

(b) $\mathcal{P}' = \mathcal{P} \setminus \{s_5, s_6\} \cup \{s'_5, s'_6\}$ (gcov)

Fig. 6. In the case of gcov #90438, gcov refuses to report coverage information for the `case` statement after removing Lines #5–6, but reports the execution of its default branch, which may mislead a developer or an automated tool. Note that though Line #4 is not covered, it is not removed otherwise will result in a compilation error.

TABLE III
SUMMARY OF THE TEST PROGRAMS WITH INCONSISTENT COVERAGE REPORTS BY Cod ON THE CONSISTENT TEST PROGRAMS BY C2V.

# weakly consistent under C2V	# inconsistent in terms of Cod			
	gcov		llvm-cov	
	Strong	Weak	Strong	Weak
3745	10	23	19	9

TABLE IV
SUMMARIZATION OF THE COMMON AND NON-COMMON INSTRUMENTATION SITES BETWEEN GCOV AND LLVM-COV FOR THE TEST PROGRAMS IN GCC TESTSUITES 7.4.0.
C / $\overline{C}$: NUMBER OF COMMON / NON-COMMON INSTRUMENTATION SITES.

	$\overline{C}$		C	
	Total	Avg.	Total	Avg.
#	83026	16.49	98523	19.56
%	45.73%	-	54.27%	-

C. Discussions

Statistics of Inconsistencies Table III summarizes the test programs in which inconsistencies are identified by Cod but unable to be identified by C2V. All these inconsistencies are true positives: one is either a bug or may mislead a developer or automatic tool.

While using these test programs to test gcov by Cod, we respectively identified 23 weak and 10 strong inconsistencies from these test programs. For llvm-cov, 28 weak and 19 strong inconsistencies are identified. This indicates that Cod has its unique ability to identify many inconsistencies that C2V unable to given the same test programs. We thus believe that Cod is more powerful and useful than C2V.

Weak Inconsistencies Between Independently Implemented Coverage Profilers As aforementioned, independently implemented code coverage profilers might have different interpretations for the same code. This is the major source of *weak inconsistencies* that C2V cannot recognize as a bug.

To further understand weak inconsistencies among profilers, we collect the common instrumentation sites between gcov 9.0.0 and llvm-cov 9.0 for the test programs using programs in GCC testsuites 7.4.0. A code line s is a common instrumentation site $s \in C$ if $\mathcal{C}^G_{\mathcal{P}}(s) \neq -1 \ \wedge \ \mathcal{C}^L_{\mathcal{P}}(s) \neq -1$, where $\mathcal{C}^G_{\mathcal{P}}(s)$ and $\mathcal{C}^L_{\mathcal{P}}(s)$ refer to the profiled runtime execution count of code line s in program $\mathcal{P}$ respectively by gcov and llvm-cov. When $\mathcal{C}^G_{\mathcal{P}}(s) \neq \mathcal{C}^L_{\mathcal{P}}(s) \ \wedge \ (\mathcal{C}^G_{\mathcal{P}}(s) = -1 \vee \mathcal{C}^L_{\mathcal{P}}(s) = -1)$, s is an non-common instrumentation site $s \in \overline{C}$.

Only 5036 test programs in GCC testsuites 7.4.0 can be successfully compiled and further processed by both gcov and llvm-cov. Table IV summarizes the total number and total percentage of common instrumentation sites and non-common instrumentation sites. The second and the forth columns respectively show the total number/percentage C and $\overline{C}$. The third and the last columns respectively shows the average C and $\overline{C}$. From Table IV, we can found that about 46% code lines are $\overline{C}$ and each test program has about 16 code lines are $\overline{C}$.

Table V summarizes the statistics of the proportion of $\overline{C}$ for the 5036 test programs in GCC testsuite 7.4.0. We calculate the proportion as $p = |\overline{C}|/(|C|+|\overline{C}|)$ for each test program. Then, we can obtain how many test programs falls into different intervals as listed in the second row of Table V. From Table V, we can find that about 40%∼70% code lines in most test programs are in $\overline{C}$. This indicates that most code lines of each program is instrumented by only one of the two coverage profilers. Besides, we also found that only 1.14% test programs have exactly the same instrumentation sites under the two profilers.

Overall, our core observation is that different coverage profilers indeed have quite different interpretations on a same piece of code.

Reliability of Code Coverage Profilers Under Compiler Optimizations Finally, even though coverage profilers provide only faithful statistics under the zero optimization level, we

TABLE V
DISTRIBUTION OF THE PERCENTAGE OF NON-COMMON INSTRUMENTATION SITES.
$p = |\overline{C}|/(|C| + |\overline{C}|)$: THE PROPORTION OF NON-COMMON INSTRUMENTATION SITES.

	p										
	0	0∼10%	10%∼20%	20%∼30%	30%∼40%	40%∼50%	50%∼60%	60%∼70%	70%∼80%	80%∼90%	90%∼100%
#	61	21	91	228	703	1722	1433	678	276	84	58
%	1.14%	0.39%	16.70%	4.26%	13.13%	32.16%	26.76%	12.66%	5.15%	15.69%	1.08%

TABLE VI
SUMMARY OF INCONSISTENT REPORTS WITH DIFFERENT OPTIMIZATION LEVELS.

Optimization level	Inconsistent lines			
	gcov		llvm-cov	
	Strongly	Weakly	Strongly	Weakly
-O0	69	54	62	11
-O1	1115	635	62	11
-O2	678	936	63	11
-O3	679	937	63	11
-Os	799	977	63	11
-Ofast	677	927	61	11

still wonder whether inconsistencies reported by `Cod` for optimized binaries may reveal bugs in a profiler. Therefore, we conducted our experiments under optimized compiler settings, and the results are summarized in Table VI.

After the manual inspection of a few cases, we found that gcov generally does *not* provide comprehensive coverage statistics for optimized code (sometimes even obviously wrong), however, llvm-cov is much more reliable–coverage statistics barely change across optimization levels.

We attempted to report such an obviously incorrect coverage statistics of gcov to the developers, as shown in Figure 7 (gcov #90420, under `-O3` optimization level). Line #11 cannot be executed for 11 times in any circumstance, however, the developer rejected this bug report and argued that "it's the nature of any optimizing compiler. If you want to have the best results, then don't use `-O3`, or any other optimization level." This case is also controversial, however, revealed that providing guarantee of coverage statistics under compiler optimizations would be a worthwhile future direction.

V. RELATED WORK

This section surveys some related work on coverage profiler testing, metamorphic testing, testing via equivalence module inputs, and techniques relied on code coverage.

A. Code Coverage Profiler Testing

To the best of our knowledge, `C2V` [11] is the first and also the state-of-the-art work for hunting bugs in code coverage profilers. It feeds a randomly generated program to both gcov and llvm-cov, and then reports a bug if there exist inconsistencies between the produced coverage reports. Within non-continuous four months of testing, `C2V` uncovered 83 bugs, among which 42 and 28 bugs are confirmed/fixed by gcov and llvm-cov, respectively. In essence, `C2V` is a randomized differential testing approach. As stated in Section II-C, `C2V` suffers from a bunch of drawbacks. The work presented in this paper attempts to fill the gap.

B. Metamorphic Testing

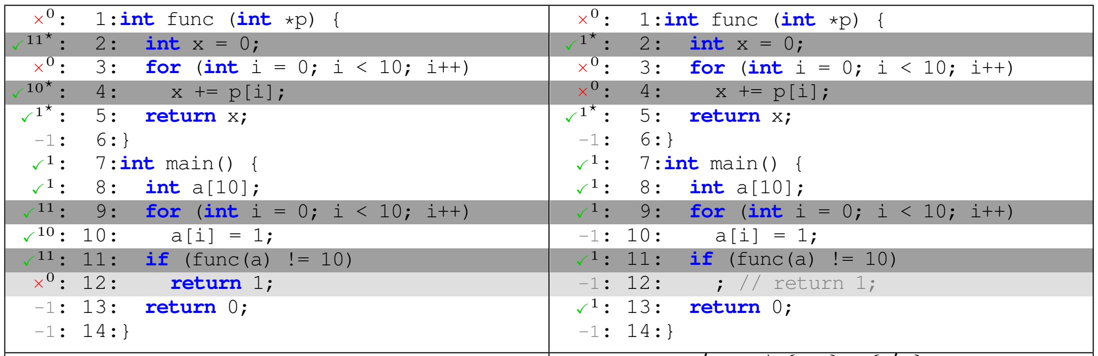
As a simple but effective approach to alleviating the oracle problem, metamorphic testing (MT) exploits the metamorphic relation (MR) among multiple inputs and their expected outputs, to generated follow-up test cases from existing ones, and verifies the corresponding outputs against the MR. Since its first publication in 1998, MT has been successful applied in a variety of domains including bioinformatics [22], [23], web services [24], [25], embedded systems [26], components [27], databases [28], machine learning classifiers [29], online search functions and search engines [30], [31], and security [32].

Several representative work are listed below. Chan et al. [24], [25] presented a metamorphic testing methodology for Service Oriented Applications (SOA). Their method relies on so-called metamorphic services to encapsulate the services under test, executes the seed test and the followup test cases, and finally check their results. Zhou et al. [30], [31] employed metamorphic testing to detect inconsistencies in online web search applications. Several metamorphic relations are proposed and utilized in a number of experiments with the web search engines, like Google, Yahoo! and Live Search. Jiang et al. [26] presented several metamorphic relations for fault detection in Central Processing Unit (CPU) scheduling algorithms. Two real bugs are found in one of the simulators under test. Beydeda [27] proposed a selftesting method for commercial offtheshelf components via metamorphic testing. Zhou et al. [33] applied metamorphic testing to self-driving cars, and detected fatal software bugs in the LiDAR obstacle-perception module. Chen et al. [22] presented several metamorphic relations for the detection of faults in two opensource bioinformatics programs for gene regulatory networks simulations and short sequence mapping.

In this paper, we applied MT to a new domain, i.e., validating the correctness of coverage profilers.

C. Testing via Equivalence Modulo Inputs

Testing via equivalence modulo inputs (EMI) [2], [34], [35] is a new testing technique proposed in recent years, being targeted at discovering the compiler optimization bugs. The basic idea of EMI is to modify a program to generate variants



```
  ×0:   1:int func (int *p) {
✓11*:   2:  int x = 0;
  ×0:   3:  for (int i = 0; i < 10; i++)
✓10*:   4:    x += p[i];
 ✓1*:   5:  return x;
  -1:   6:}
  ✓1:   7:int main() {
  ✓1:   8:  int a[10];
 ✓11:   9:  for (int i = 0; i < 10; i++)
 ✓10:  10:    a[i] = 1;
 ✓11:  11:  if (func(a) != 10)
  ×0:  12:    return 1;
  -1:  13:  return 0;
  -1:  14:}
```

(a) $\mathcal{P}$ (gcov)

```
  ×0:   1:int func (int *p) {
 ✓1*:   2:  int x = 0;
  ×0:   3:  for (int i = 0; i < 10; i++)
  ×0:   4:    x += p[i];
 ✓1*:   5:  return x;
  -1:   6:}
  ✓1:   7:int main() {
  ✓1:   8:  int a[10];
  ✓1:   9:  for (int i = 0; i < 10; i++)
  -1:  10:    a[i] = 1;
  ✓1:  11:  if (func(a) != 10)
  -1:  12:    ; // return 1;
  ✓1:  13:  return 0;
  -1:  14:}
```

(b) $\mathcal{P}' = \mathcal{P} \setminus \{s_{12}\} \cup \{s'_{12}\}$ (gcov)

Fig. 7. The bug case of GCC #90420. gcov incorrectly reported that the `if(func(a) != 10)` in Line 11 was executed 11 times. Deleting Line #12 revealed this bug.

with the same outputs as the original program. Initially, Le et al. [2] proposed to generate equivalent versions of the program by profiling program's execution and pruning unexecuted code inside. Once a program and its equivalent variant are constructed, both are fed to the compiler under test, and the inconsistencies of the outputs are checked. Following this work, Athena [34] and Hermes [35] are developed subsequently. Athena [34] generates EMI by randomly inserting code into and removing statements from dead code regions. Hermes [35] complements mutation strategies by operating on live code regions, which overcomes the limitations of mutating dead code regions.

In Cod, we followed the similar way to generate program variants as EMI did, but focused on validating the correctness of coverage profilers instead of optimization bugs in compilers. As such, during the results verification, Cod not only checked the inconsistencies in terms of the outputs, but more importantly the coverage reports. Through our evaluations, it is also shown that only few bugs (1 among 23 confirmed bugs) can be discovered by looking at only the outputs. Moreover, different from EMI performing a random modification, Cod mutates the original program by aggressive statement pruning, thus triggering different coverage behaviors as much as possible.

D. Techniques relied on code coverage

Code coverage is widely adopted in practice and extensively used to facilitate many software engineering tasks, such as coverage-based regression testing, coverage-based compiler testing, and coverage-based debugging. In the context of regression testing, test case prioritization and test suite augmentation are the two widely used techniques [36]–[43]. The former aims to improve the ability of test cases in finding faults by scheduling test cases in a specific order [41], [43], [44]. To achieve a high code coverage as fast as possible is a common practice [45]. The latter is to generate new test cases to strengthen the ability of a test suite in finding faults [42], [46], [47]. In practice, it is often to generate new test cases to cover the source code affected by code changes. Recent years have seen an increasing interest in compiler testing which aims to validate the correctness of compilers. One of the most attractive compiler testing techniques is based on the code coverage of a program's execution to generate equivalence modulo inputs by stochastically pruning its unexecuted code [2], [48]. With the equivalence modulo inputs, we can differentially test compilers. It is obvious that the correctness of "equivalance" relies on the reliability of code coverage. Debugging is a common activity in software development which aims to locating the root cause of a fault. Spectrum-Based Fault Localization (SBFL) is one of the most extensively studied debugging techniques which is heavily based on code coverage [4], [49]–[51]. Under a specific test suite, SBFL leverages the code coverage and the corresponding failed/passed information to statistically infer which code is the root cause of a fault.

As we can see, the correct code coverage information is one of the prerequisites for the techniques above, indicating the importance of our work.

VI. Conclusion

This paper presents Cod, an automated self-validator for code coverage profilers based on metamorphic testing. Cod addressed the limitation of the state-of-the-art differential testing approach, and encouragingly found many previously unknown bugs which cannot be revealed by existing approaches.

Acknowledgment

We thank the anonymous reviewers for their constructive comments. We also thank the GCC and LLVM developers especially Martin Liška for analyzing and fixing our reported bugs. This work is supported by the National Key R&D Program of China (2018YFB1003901), the National Natural Science Foundation of China (61832009, 61432001, 61932021, 61690204, 61772259, 61772263, 61802165, 61802168, 61702256), the Natural Science Foundation of Jiangsu Province (SBK2019022682, BK20170652), the China Postdoctoral Science Foundation (2018T110481), the Fundamental Research Funds for the Central Universities (020214380032, 02021430047). We would also like to thank the support from the Collaborative Innovation Center of Novel Software Technology and Industrialization, Jiangsu, China. Yuming Zhou and Baowen Xu are the corresponding authors.

References

[1] J. C. Miller and C. J. Maloney, "Systematic mistake analysis of digital computer programs," *Commun. ACM*, vol. 6, no. 2, pp. 58–63, Feb. 1963.

[2] V. Le, M. Afshari, and Z. Su, "Compiler validation via equivalence modulo inputs," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. New York, NY, USA: ACM, 2014, pp. 216–226.

[3] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: ACM, 2016, pp. 1032–1043.

[4] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '05. New York, NY, USA: ACM, 2005, pp. 273–282.

[5] Z. Zuo, S.-C. Khoo, and C. Sun, "Efficient predicated bug signature mining via hierarchical instrumentation," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA '14. New York, NY, USA: ACM, 2014, pp. 215–224.

[6] Z. Zuo, L. Fang, S.-C. Khoo, G. Xu, and S. Lu, "Low-overhead and fully automated statistical debugging with abstraction refinement," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '16. New York, NY, USA: ACM, 2016, pp. 881–896.

[7] D. Lo, S.-C. Khoo, J. Han, and C. Liu, *Mining Software Specifications: Methodologies and Applications*, 1st ed. Boca Raton, FL, USA: CRC Press, Inc., 2011.

[8] Z. Zuo and S.-C. Khoo, "Mining dataflow sensitive specifications," in *Formal Methods and Software Engineering*, L. Groves and J. Sun, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 36–52.

[9] S. Park, R. W. Vuduc, and M. J. Harrold, "Falcon: Fault localization in concurrent programs," in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 245–254.

[10] "Lighthouse - a code coverage explorer for reverse engineers," https://github.com/gaasedelen/lighthouse.

[11] 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," in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE '19. Piscataway, NJ, USA: IEEE Press, 2019, pp. 488–499.

[12] T. Y. Chen, S. C. Cheung, and S. M. Yiu, "Metamorphic testing: A new approach for generating next test cases," Technical Report HKUST-CS98-01, Department of Computer Science, Hong Kong, Tech. Rep., 1998.

[13] M. Liška, "Explanations on the coverage results under optimizations." https://gcc.gnu.org/bugzilla/show_bug.cgi?id=90420.

[14] E. J. Weyuker, "On testing non-testable programs," *The Computer Journal*, vol. 25, no. 4, pp. 465–470, 1982.

[15] D. Hamlet, "Predicting dependability by testing," in *ACM SIGSOFT Software Engineering Notes*, vol. 21, no. 3. ACM, 1996, pp. 84–91.

[16] L. Manolache and D. G. Kourie, "Software testing using model programs," *Software: Practice and Experience*, vol. 31, no. 13, pp. 1211–1236, 2001.

[17] "Gcov," https://gcc.gnu.org/onlinedocs/gcc/Gcov.html.

[18] "Gcc," https://gcc.gnu.org/.

[19] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in C compilers," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11. New York, NY, USA: ACM, 2011, pp. 283–294.

[20] C. Sun, V. Le, and Z. Su, "Finding and analyzing compiler warning defects," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 203–213.

[21] V. Le, C. Sun, and Z. Su, "Randomized stress-testing of link-time optimizers," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ser. ISSTA '15. New York, NY, USA: ACM, 2015, pp. 327–337.

[22] T. Y. Chen, J. W. Ho, H. Liu, and X. Xie, "An innovative approach for testing bioinformatics programs using metamorphic testing," *BMC Bioinformatics*, vol. 10, no. 1, p. 24, 2009.

[23] L. L. Pullum and O. Ozmen, "Early results from metamorphic testing of epidemiological models," in *Proceedings of the ASE/IEEE International Conference on BioMedical Computing*, ser. BioMedCom '12.

[24] W. Chan, S. C. Cheung, and K. R. Leung, "Towards a metamorphic testing methodology for service-oriented software applications," in *Proceedings of the 5th International Conference on Quality Software*, ser. QSIC '05. IEEE, 2005, pp. 470–476.

[25] W. K. Chan, S. C. Cheung, and K. R. Leung, "A metamorphic testing approach for online testing of service-oriented software applications," *International Journal of Web Services Research (IJWSR)*, vol. 4, no. 2, pp. 61–81, 2007.

[26] M. Jiang, T. Y. Chen, F.-C. Kuo, and Z. Ding, "Testing central processing unit scheduling algorithms using metamorphic testing," in *Proceedings of the 4th IEEE International Conference on Software Engineering and Service Science*. IEEE, 2013, pp. 530–536.

[27] S. Beydeda, "Self-metamorphic-testing components," in *30th Annual International Computer Software and Applications Conference*, ser. COMPSAC '06.

[28] M. Lindvall, D. Ganesan, R. Árdal, and R. E. Wiegand, "Metamorphic model-based testing applied on NASA DAT: An experience report," in *Proceedings of the 37th International Conference on Software Engineering-Volume 2*. IEEE Press, 2015, pp. 129–138.

[29] X. Xie, J. W. Ho, C. Murphy, G. Kaiser, B. Xu, and T. Y. Chen, "Testing and validating machine learning classifiers by metamorphic testing," *Journal of Systems and Software*, vol. 84, no. 4, pp. 544–558, 2011.

[30] Z. Q. Zhou, S. Zhang, M. Hagenbuchner, T. Tse, F.-C. Kuo, and T. Y. Chen, "Automated functional testing of online search services," *Software Testing, Verification and Reliability*, vol. 22, no. 4, pp. 221–243, 2012.

[31] Z. Q. Zhou, S. Xiang, and T. Y. Chen, "Metamorphic testing for software quality assessment: A study of search engines," *IEEE Transactions on Software Engineering*, vol. 42, no. 3, pp. 264–284, 2016.

[32] T. Y. Chen, F.-C. Kuo, W. Ma, W. Susilo, D. Towey, J. Voas, and Z. Q. Zhou, "Metamorphic testing for cybersecurity," *Computer*, vol. 49, no. 6, pp. 48–55, 2016.

[33] Z. Q. Zhou and L. Sun, "Metamorphic testing of driverless cars," *Commun. ACM*, vol. 62, no. 3, pp. 61–67, Feb. 2019.

[34] V. Le, C. Sun, and Z. Su, "Finding deep compiler bugs via guided stochastic program mutation," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '15. New York, NY, USA: ACM, 2015, pp. 386–399.

[35] C. Sun, V. Le, and Z. Su, "Finding compiler bugs via live code mutation," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '16. New York, NY, USA: ACM, 2016, pp. 849–863.

[36] M. Gligoric, A. Groce, C. Zhang, R. Sharma, M. A. Alipour, and D. Marinov, "Guidelines for coverage-based comparisons of non-adequate test suites," *ACM Trans. Softw. Eng. Methodol.*, vol. 24, no. 4, pp. 22:1–22:33, Sep. 2015.

[37] G. Fraser, M. Staats, P. McMinn, A. Arcuri, and F. Padberg, "Does automated white-box test generation really help software testers?" in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ser. ISSTA '13. New York, NY, USA: ACM, 2013, pp. 291–301.

[38] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: A survey," *Softw. Test. Verif. Reliab.*, vol. 22, no. 2, pp. 67–120, Mar. 2012.

[39] J. A. Jones and M. J. Harrold, "Test-suite reduction and prioritization for modified condition/decision coverage," *IEEE Trans. Softw. Eng.*, vol. 29, no. 3, pp. 195–209, Mar. 2003.

[40] L. Zhang, D. Marinov, L. Zhang, and S. Khurshid, "Regression mutation testing," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ser. ISSTA '12. New York, NY, USA: ACM, 2012, pp. 331–341.

[41] D. Hao, L. Zhang, L. Zhang, G. Rothermel, and H. Mei, "A unified test case prioritization approach," *ACM Trans. Softw. Eng. Methodol.*, vol. 24, no. 2, pp. 10:1–10:31, Dec. 2014.

[42] S. Artzi, J. Dolby, F. Tip, and M. Pistoia, "Directed test generation for effective fault localization," in *Proceedings of the 19th international symposium on Software testing and analysis*. ACM, 2010, pp. 49–60.

[43] Z. Li, M. Harman, and R. M. Hierons, "Search algorithms for regression test case prioritization," *IEEE Trans. Softw. Eng.*, vol. 33, no. 4, pp. 225–237, Apr. 2007.

[44] G. Rothermel, R. J. Untch, and C. Chu, "Prioritizing test cases for regression testing," *IEEE Trans. Softw. Eng.*, vol. 27, no. 10, pp. 929–948, Oct. 2001.

[45] W. E. Wong, J. R. Horgan, S. London, and H. A. Bellcore, "A study of effective regression testing in practice," in *Proceedings of the Eighth International Symposium on Software Reliability Engineering*, ser. ISSRE '97. Washington, DC, USA: IEEE Computer Society, 1997, pp. 264–274.

[46] G. Fraser and A. Arcuri, "A large-scale evaluation of automated unit test generation using evosuite," *ACM Trans. Softw. Eng. Methodol.*, vol. 24, no. 2, pp. 8:1–8:42, Dec. 2014.

[47] Y. Li, Z. Su, L. Wang, and X. Li, "Steering symbolic execution to less traveled paths," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '13. New York, NY, USA: ACM, 2013, pp. 19–32.

[48] C. Lidbury, A. Lascu, N. Chong, and A. F. Donaldson, "Many-core compiler fuzzing," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '15. New York, NY, USA: ACM, 2015, pp. 65–76.

[49] X. Xie, T. Y. Chen, F.-C. Kuo, and B. Xu, "A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization," *ACM Trans. Softw. Eng. Methodol.*, vol. 22, no. 4, pp. 31:1–31:40, Oct. 2013.

[50] S. Yoo, M. Harman, and D. Clark, "Fault localization prioritization: Comparing information-theoretic and coverage-based approaches," *ACM Trans. Softw. Eng. Methodol.*, vol. 22, no. 3, pp. 19:1–19:29, Jul. 2013.

[51] R. Santelices, J. A. Jones, Y. Yu, and M. J. Harrold, "Lightweight fault-localization using multiple coverage types," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 56–66.