2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)

Hunting for Bugs in Code Coverage Tools via Randomized Differential Testing

Yibiao Yang*, Yuming Zhou*, Hao Sun†, Zhendong Su‡§, Zhiqiang Zuo*, Lei Xu*, and Baowen Xu*
*State Key Lab. for Novel Software Technology, Nanjing University, Nanjing, China
†Unaffiliated
‡Department of Computer Science, ETH Zurich, Switzerland
§Computer Science Department, UC Davis, USA

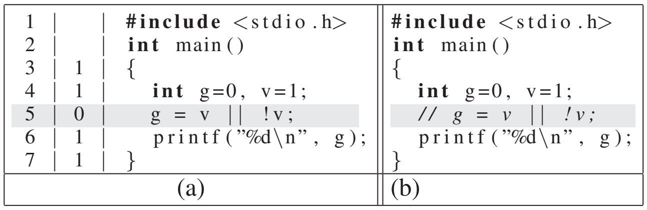
***Abstract*—Reliable code coverage tools are critically important as it is heavily used to facilitate many quality assurance activities, such as software testing, fuzzing, and debugging. However, little attention has been devoted to assessing the reliability of code coverage tools. In this study, we propose a randomized differential testing approach to hunting for bugs in the most widely used C code coverage tools. Specifically, by generating random input programs, our approach seeks for inconsistencies in code coverage reports produced by different code coverage tools, and then identifies inconsistencies as potential code coverage bugs. To effectively report code coverage bugs, we addressed three specific challenges: (1) How to filter out duplicate test programs as many of them triggering the same bugs in code coverage tools; (2) how to automatically reduce large test programs to much smaller ones that have the same properties; and (3) how to determine which code coverage tools have bugs? The extensive evaluations validate the effectiveness of our approach, resulting in 42 and 28 confirmed/fixed bugs for gcov and llvm-cov, respectively. This case study indicates that code coverage tools are not as reliable as it might have been envisaged. It not only demonstrates the effectiveness of our approach, but also highlights the need to continue improving the reliability of code coverage tools. This work opens up a new direction in code coverage validation which calls for more attention in this area.**

***Index Terms*—Code Coverage; Differential Testing; Coverage Tools; Bug Detection.**

I. Introduction

Code coverage [1] refers to which code in the program and how many times each code is executed when running on the particular test cases. The code coverage information produced by code coverage tools is widely used to facilitate many quality assurance activities, such as software testing, fuzzing, and debugging [1]–[15]. For example, researchers recently introduced an EMI ("Equivalence Modulo Inputs") based compiler testing technique [7]. The equivalent programs are obtained by stochastically pruning the unexecuted code of a given program according to the code coverage information given by the code coverage tools (e.g., llvm-cov, gcov). Therefore, the correctness of "equivalence" relies on the reliability of code coverage tools.

In spite of the prevalent adoption in practice and extensive testing of code coverage tools, a variety of defects still remain. Fig. 1(a) shows a buggy code coverage report produced by llvm-cov [16], a C code coverage tool of Clang [17]. Note that all the test cases have been reformatted for presentation in this study. The coverage report is an annotated version of source code, where the first and second column list the line number and the execution frequency, respectively. We can see that the code at line 5 is marked incorrectly as unexecuted by llvm-cov. Given the program *p* and its corresponding code coverage as shown in Fig. 1(a), EMI compiler testing [7] generates its "equivalent" program *p'* as shown in Fig. 1(b) by removing unexecuted code (statement 5). The program *p* and *p'* will be compiled by a compiler under testing and then executed to obtain two different outputs, i.e. 1 and 0, resulting in a bug reported by the EMI approach. However, this is obviously not a real compiler bug. The incorrect code coverage report leads to the false positive in compiler testing. As the code coverage tools offer the fundamental information needed during the whole process of software development, it is essential to validate the correctness of code coverage. Unfortunately, to our best knowledge, little attention has been devoted to assessing the reliability of code coverage tools.



```
1 |   | #include <stdio.h>
2 |   | int main()
3 | 1 | {
4 | 1 |   int g=0, v=1;
5 | 0 |   g = v || !v;
6 | 1 |   printf("%d\n", g);
7 | 1 | }
```

(a)

```
#include <stdio.h>
int main()
{
  int g=0, v=1;
  // g = v || !v;
  printf("%d\n", g);
}
```

(b)

Fig. 1. (a) Bug #33465 of llvm-cov and (b) The "equivalent" program obtained by pruning the unexecuted code (Line #4) of the program in (a)

This work makes the first attempt in this direction. We devise a practical randomized differential testing approach to discovering bugs in code coverage tools. Our approach firstly leverages programs generated by a random generator to seek for inconsistencies of code coverage reports produced by different code coverage tools, and identifies inconsistencies as potential code coverage bugs. Secondly, due to the existence of too many inconsistency-triggering test programs reported and a large portion of irrelevant code within these test programs, reporting these inconsistency-triggering tests directly is hardly beneficial to debugging. Before reporting them, the reduction to those test programs is required [18]. However, it is usually very costly and thus unrealistic to reduce every and each of the test programs [18]. We observe that many test programs trigger the same coverage bugs. Thus, we can filter out many duplicate test programs. Note that 'duplicate test programs' in this study indicates multiple test programs triggering the same code coverage bug. Overall, to effectively report coverage bugs, we need to address the following key challenges:

Challenge 1: Filtering Out Test Programs. To filter out potential test programs triggering the same code coverage bugs, the most intuitive way is to calculate similarities between

1558-1225/19/$31.00 ©2019 IEEE
DOI 10.1109/ICSE.2019.00061

programs using the whole text. However, we use Csmith [19] as the random program generator and two Csmith-generated programs are not meaningfully comparable as they diverge in many ways [20]. In addition, calculating similarities between programs using the whole text is expensive. To tackle this challenge, only lines of code triggering inconsistencies are used for computing similarities between programs.

Challenge 2: Reducing Test Programs. Reducing test programs for code coverage bugs is much more complex than reducing test programs for compiler bugs as the later one only requires testing the behavior of the compiled executables or the exit code of compilers [18]. However, reducing test programs for code coverage bugs involves processing textual code coverage reports and identify inconsistencies. After each iteration of reduction, we need to specify the inconsistency of interest we would like to preserve. In this study, we design a set of inconsistency types over coverage reports as the interest.

Challenge 3: Inspecting Coverage Bugs. With the reduced test program, we need to inspect which code coverage tools have bugs before reporting bug. In practice, it is usually done manually [21]. In other words, developers manually inspect the coverage reports to determine which coverage tools are buggy. To relieve the burden of manual intervention, we summarize a number of rules that code coverage reports must follow. With those rules, we develop a tool to examine part of the inconsistent coverage reports and determine which tools have bugs automatically.

We implemented a differential testing prototype called `C2V` ("Code Coverage Validation") for code coverage tools. In order to evaluate the effectiveness of our approach, we have applied `C2V` to gcov and llvm-cov, two widely used C code coverage tools respectively in the production compilers GCC [22] and Clang [17]. Our evaluation confirms that `C2V` is very effective in finding code coverage bugs: 46 bugs were found (42 bugs confirmed/fixed) for gcov, while 37 bugs were found (28 bugs confirmed/fixed) for llvm-cov.

Contributions. We made the following contributions:

- We introduce an effective testing approach to validating the code coverage tools, and have implemented it as a practical tool `C2V` for testing C coverage tools. `C2V` mainly consists of a random program generator, a comparer to identify inconsistencies between coverage reports, a filter to remove test programs triggering same coverage bugs, a test program reducer, and an inspector to automatically determine which coverage tools have bugs for bug reporting.
- We adopted `C2V` to uncover 46 and 37 bugs for gcov and llvm-cov both of which are widely used and extensively tested C code coverage tools, respectively. Specifically, for gcov, 42 bugs have already been confirmed/fixed; for llvm-cov, 28 bugs have been confirmed/fixed.
- Our evaluation indicates that code coverage tools are not as reliable as it might have been envisaged. It opens up a new research direction to improve the reliability of code coverage tools which calls for more attention in this area. Besides, there is a need to examine the influence of those bugs on other techniques which depend on code coverage.

Organization. The rest of this paper is structured as follows. Section II introduces the background on code coverage. Section III describes our approach for code coverage validation. Section IV reports the experimental results in detail. Section V surveys related work. Section VI concludes the paper and outlines the direction for future work.

II. Code Coverage

In this section, we introduce the preliminary knowledge, the importance, and the bug categories of code coverage.

A. Preliminaries

Code coverage is a quality assurance metric used to describe the degree to which the code of a given program is executed when a particular test suite executes [1]. It is suggested that a program with a high test coverage will have a lower chance of containing undetected software bugs compared to a program with a low test coverage.

The code coverage analysis process is generally divided into three tasks: *code instrumentation*, *data gathering*, and *coverage analysis*. Specifically, code instrumentation inserts additional code instructions or probes to monitor whether the specific *program chunk* is executed or not at runtime. The instrumentation can be done at the source level in a separate pre-processing phase or at runtime by instrumenting byte code. Data gathering aims to collect the coverage data produced at test runtime. Finally, coverage analysis aims to analyze the collected results and to provide test strategy recommendations in order to reduce, to feed or to modify the relevant test suite.

Currently, many code coverage tools are available [16], [23]–[31], which support different languages (e.g., C/C++ and Java), instrumentation levels (e.g. source code and byte code level), or coverage criteria. Coverage criteria are the rules or requirements that a test suite needs to satisfy [32]. In the literature, many coverage criteria have been proposed, including statement coverage, branch coverage, path coverage, condition coverage, decision coverage, and data-flow coverage [33]. These criteria can be used to guide the generation of a test suite or to evaluate the effectiveness of a test suite [33].

B. Importance of Code Coverage

Code coverage is widely used in, but not limited to, the following software techniques:

- Coverage-based regression testing. In the context of regression testing, test case prioritization and test suite augmentation are the two widely used techniques [2]–[4], [10], [15], [34]–[36]. The former aims to improve the ability of test cases to find bugs by scheduling test cases in a specific order [10], [15], [37]. One common practice is to achieve a high code coverage as fast as possible [38]. The latter is to generate new test cases to strengthen the ability of a test suite to find bugs [6], [14], [36], [39]. In practice, it is often to generate new test cases to cover the source code affected by code changes.
- Coverage-based compiler testing. Recent years have seen an increasing interest in compiler testing which aims to validate

```
 1 |   | #include <stdlib.h>
 2 |   |
 3 |   | int main (void)
 4 | 1 | {
 5 | 1 |   static int *p;
 6 | 1 |
 7 | 1 |   p = malloc (sizeof (int));
 8 | 1 |   if (p == NULL)
 9 | 1 |     return 0;
10 | 1 |
11 | 1 |   *p = 7;
12 | 1 |   return 1;
13 | 1 | }
```

Fig. 2. Spurious marking (Bug #37092 of llvm-cov)

```
    1 :  1 : int func(int i)
    − :  2 : {
    1 :  3 :   if (i > 0)
    1 :  4 :     return 1;
#####  :  5 :   return 0;
    − :  6 : }
    − :  7 :
    1 :  8 : int  main()
    − :  9 : {
    1 : 10 :   int i = 1, j = 2;
    2 : 11 :   func((i==0) || (i&&j, 1));
    1 : 12 :   return 0;
    − : 13 : }
```

Fig. 3. Wrong frequency (Bug #85163 of gcov)

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 [7], [40], [41]. With the equivalence modulo inputs, we can differentially test compilers.

- Coverage-based fuzzing. Fuzzing technique is one of the most effective testing techniques to find vulnerabilities in software. In practice, coverage-based fuzzing has been widely used [8], [42]. Based on the code coverage information of test cases, it aims to determine which test cases should be retained for fuzzing. In general, a test case is retained if a new basic block is covered.
- Coverage-based debugging. Debugging is a common activity in software development which aims to find 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 [5], [43]–[47]. Under a specific test suite, SBFL leverages the code coverage and the corresponding failed/passed information to infer which code is the root cause of a fault.

As can be seen, the above-mentioned software techniques heavily depend on the code coverage information produced by code coverage tools. Therefore, it is of great importance to guarantee the correctness of the code coverage reports. Otherwise, they might inevitably produce incorrect results or lead to suboptimal decisions.

C. Categories of Code Coverage Bugs

Code coverage bugs can be categorized into the following three general classes:

- *Spurious Marking.* Spurious marking denotes that a program chunk is unexecuted at runtime but is wrongly marked as executed by a code coverage tool. Figure 2 gives such an example. In the `main` function, the two *return*-statements at line 9 and line 12 cannot be executed at the same time. Thus, one of them must be wrongly marked by llmv-cov. At runtime, function `main` returns "1", indicating that line 9 was wrongly marked. A code coverage tool with spurious marking bugs will cause non-executed code wrongly marked as executed. Consequently, for a program under test, a part of elements are indeed untested and hence may have latent bugs undetected.
- *Missing Marking.* Missing marking denotes that a program chunk is actually executed at runtime but is wrongly marked as unexecuted by a code coverage tool. The coverage bug as shown in Figure 1 belongs to this class. A code coverage tool with missing marking bugs will cause a predefined coverage goal never achieved, regardless of how much testing time and resource are allocated.
- *Wrong Frequency.* Wrong frequency denotes that a program chunk is executed *m* times at runtime but is wrongly marked as executed *n* times by a code coverage tool $\{(m \neq n) \wedge (m > 0) \wedge (n > 0)\}$. Figure 3 shows a gcov coverage report, in which the first column lists the execution frequency and the second column lists the line number. As can be seen, the code at line 11 is wrongly marked as executed twice but it was actually executed only once. A code coverage tool with wrong frequency bugs might lead to a suboptimal decision in many coverage-based software engineering activities.

III. Methodology

Figure 4 presents our framework for code coverage validation. In the following, we describe the key steps in this framework. In particular, we will use gcov and llvm-cov as the subject code coverage tools to illustrate our approach to hunting for code coverage bugs if necessary.

A. Generating Test Programs

Our approach starts with a random program generator. The *Generator* randomly generates a large program set P. Each program $p \in P$ will be used as a test program for differentially testing two code coverage tools under examination. In other words, each program will be fed to the two code coverage tools to obtain two raw coverage reports.

In our study, test programs refer to a collection of compilable C programs with the corresponding inputs. We choose Csmith to generate the test programs because of the following reasons: (1) it supports many C language features and can avoid generating programs with undefined behaviors [48], thus outperforming some random C program generators [49], [50]; (2) each generated program is one single file with input self-contained which does not require extra inputs; and (3) it is fast enough to generate programs with tens of thousands of lines within a few seconds.

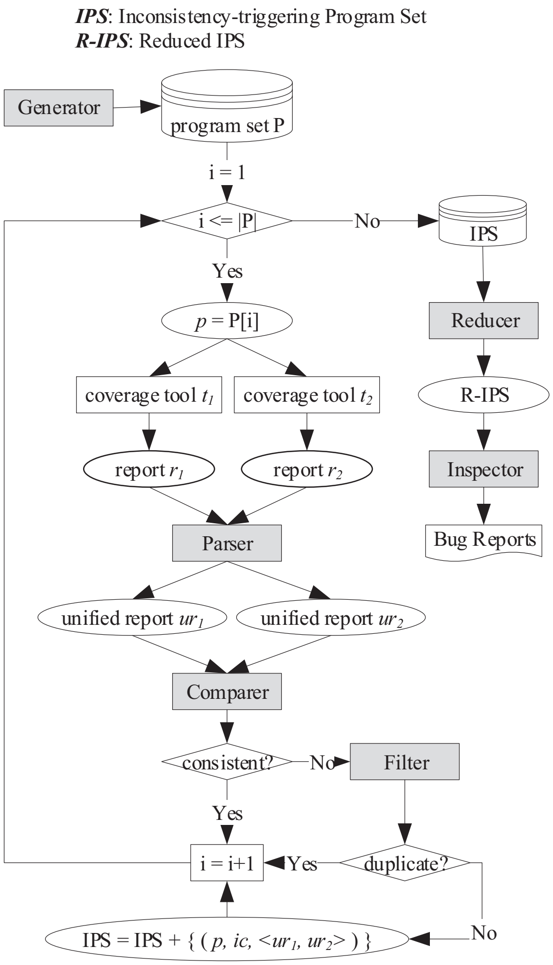


Fig. 4. The framework for code coverage validation

B. Parsing Code Coverage Reports

For each test program p in the generated program set P, we obtain the raw code coverage reports r_1 and r_2 emitted from the coverage tools t_1 and t_2 respectively. However, the raw coverage reports cannot be directly compared since they are presented in different formats. We thus develop a parser to transform r_i into ur_i in a unified format ($1 \leq i \leq 2$). Specifically, ur_i is a sequence of two-tuple, including the monitored program chunk and the corresponding execution frequency. Given a program p with N lines of source code, ur_i is a sequence of N two-tuples (n_j, f_j) in ascending order according to the value of n_j, $1 \leq j \leq N$. Here, n_j is the line number and f_j is the corresponding execution frequency. If line n_j is not an instrumentation site for a particular coverage tool, f_j is assigned -1.

In our study, gcov and llvm-cov are selected as the subject code coverage tools, which are integrated with the mature production compilers GCC and Clang respectively. In the following, we give an illustrating example to demonstrate how the parser works in real world. Figure 5(a) shows the code coverage report r_1 produced by gcov for a program, and Figure 5(b) shows the code coverage report r_2 produced by llvm-cov for the same program. r_1 and r_2 are two annotated versions of the source file. However, there are three differences between r_1 and r_2. First, they have different instrumentation sites. On the one hand, lines 4, 6~8, 12, 19, and 22 are treated as the instrumentation sites by llvm-cov but not by gcov. On the other hand, lines 3 and 18 are treated as an instrumentation site by gcov but not by llvm-cov. Hence, only the *common* instrumentation sites used by gcov and llvm-cov need to be compared later. Second, gcov and llvm-cov have different annotation formats for execution frequency. A non-instrumentation site is noted as hyphen "-" in r_1 but is noted as a null string in r_2 (e.g. line 3). Besides, an unexecuted line is noted as hashes "#####" in r_1 (e.g. line 15), while it is noted as "0" in r_2 (e.g. line 9). Third, coverage statistics are included in r_1 but are not available in r_2. Figure 5(c) lists the unified coverage reports produced by our parser for gcov and llvm-cov respectively. As can be seen, there are 9 common instrumentation sites between them: lines 5, 7, 9, 10, 11, 13, 14, 15, 20, and 21.

C. Comparing Unified Coverage Reports

After obtaining the unified coverage reports ur_1 and ur_2, we use a tool *Comparer* to determine whether they are consistent. If not, the corresponding p and associated coverage reports will be added to a set named *IPS* (Inconsistency-triggering test Program Set). When comparing the unified coverage reports, *Comparer* only takes into account the execution frequencies of the common instrumentation sites among the n coverage tools. During the comparison of ur_1 and ur_2, the following types of cases might occur:

- $Type\ A$: one line code is marked as executed by t_1 but as unexecuted by t_2;
- $Type\ B$: one line code is marked as unexecuted by t_1 but as executed by t_2;
- $Type\ C$: one line code is marked as executed k times by t_1 and as executed l times by t_2 while $k \neq l$.

Consequently, the inconsistent unified reports can be divided into seven categories ($C001 : Type\ C$; $C010 : Type\ B$; $C100 : Type\ A$; $C011 : Type\ B + C$; $C101 : Type\ A{+}C$; $C110 : Type\ A{+}B$; $C111 : Type\ A{+}B{+}C$). If the unified coverage reports corresponding to the test program p are found to have an inconsistency category ic, it will be handled by the filter.

Considering the two unified coverage reports shown in Fig. 5, *Comparer* will compare the execution frequencies of the common 9 instrumentation sites to determine whether ur_1 and ur_2 are consistent. As can be seen, there are four inconsistent execution frequencies in the unified coverage reports between gcov and llvm-cov: $Type\ C$ at line 5, $Type\ C$ at line 13, $Type\ A$ at line 9, and $Type\ B$ at line 15. Consequently, the inconsistency category of ur_1 and ur_2 is found to be $C111$. In other words, the inconsistency introduced by the test program p belongs to the $C111$ category.

```
    - :  1 : int g=1;                    ( 1,-1),
    - :  2 :                             ( 2,-1),
    2 :  3 : int f(int arg)              ( 3, 2),
    - :  4 : {                           ( 4,-1),
    6 :  5 :   for(int i=0;i!=1;++i)     ( 5, 6),
    - :  6 :   {                         ( 6,-1),
    - :  7 :     int f[1];               ( 7,-1),
    - :  8 :     if(0)                   ( 8,-1),
    1 :  9 :       break;                ( 9, 1),
    2 : 10 :     if(arg)                 (10, 2),
    1 : 11 :       break;                (11, 1),
    - : 12 :   }                         (12,-1),
    4 : 13 :   for(;g;)                  (13, 4),
    2 : 14 :     return 0;               (14, 2),
##### : 15 :   return 0;                 (15, 0),
    - : 16 : }                           (16,-1),
    - : 17 :                             (17,-1),
    1 : 18 : int main()                  (18, 1),
    - : 19 : {                           (19,-1),
    1 : 20 :   f(0); f(1);               (20, 1),
    1 : 21 :   return 0;                 (21, 1),
    - : 22 : }                           (22,-1)
```

(a) Raw coverage report by gcov 7.3.0 and unified report ur_1

```
1  |   | int g=1;                  ( 1,-1),
2  |   |                           ( 2,-1),
3  |   | int f(int arg)            ( 3,-1),
4  | 2 | {                         ( 4, 2),
5  | 3 |   for(int i=0;i!=1;++i)   ( 5, 3),
6  | 2 |   {                       ( 6, 3),
7  | 2 |     int f[1];             ( 7, 2),
8  | 2 |     if(0)                 ( 8, 2),
9  | 0 |       break;              ( 9, 0),
10 | 2 |     if(arg)               (10, 2),
11 | 1 |       break;              (11, 1),
12 | 2 |   }                       (12, 2),
13 | 2 |   for(;g;)                (13, 2),
14 | 2 |     return 0;             (14, 2),
15 | 2 |   return 0;               (15, 2),
16 | 2 | }                         (16, 2),
17 |   |                           (17,-1),
18 |   | int main()                (18,-1),
19 | 1 | {                         (19, 1),
20 | 1 |   f(0); f(1);             (20, 1),
21 | 1 |   return 0;               (21, 1),
22 | 1 | }                         (22, 1)
```

(b) Raw coverage report by llvm-cov 6.0.0 and unified report ur_2

Fig. 5. An illustrating example.

D. Filtering out Test Programs

Intuitively, using a test program set P with a larger size has a higher chance to discover code coverage bugs. Therefore, in practice, we prefer to generate a very large number of test programs and further resulting in a large number of inconsistency-triggering test programs. This will lead to the following two problems. On the one hand, it may be unrealistic to inspect all the inconsistency-triggering test programs, as the reduction is very costly [18] and inspection resources are often limited. On the other hand, we observe that many test programs trigger the same code coverage bugs. Therefore, we can filter out duplicate test programs.

In Fig. 4, we use *Filter* to determine whether the inconsistency-triggering test program is "duplicate", i.e, triggering the same code coverage bug with other test programs. To filter out potential "duplicate" test programs, the most intuitive way is to calculate similarities between programs using the whole text. However, we use Csmith [19] as the random program generator and two Csmith-generated programs are not meaningfully comparable as they diverge in many ways [20]. In addition, calculating similarities between programs using the whole text is expensive. To tackle this challenge, only lines of code triggering inconsistencies are used for computing similarities between programs. More specifically, we first transform the inconsistency-triggering lines of code in each program into a token based string with variable names insensitive and then use a token based similarity algorithm to calculate similarities between programs. We calculate program similarity with variable name insensitive since the variable names in Csmith-generated programs have no specific meaning. For example, we assume "`int` $i = 0;$ $i = 1;$" are inconsistency-triggering lines of code in program p. We first transform these two statements into "`int identifier equal numeric_constant semi identifier equal numeric_constant semi`" with the help of the clang compiler. After that, we calculate its similarities with other programs. In this study, we use Jaccard index similarity as it is the most widely used token based similarity algorithm [51]. If all the similarity coefficients between p and each of the programs in *IPS* are less than 0.8, a tuple $(p,\ ic,\ \ < ur_1, ur_2 >)$ will be added to the set *IPS*. Otherwise, p will be filtered out.

E. Reducing Test Programs

An inconsistency-triggering test program only indicates that some of the two code coverage tools are buggy but does not inform which one contains bugs. Therefore, there is a need to inspect the inconsistency-triggering test program to obtain the correct coverage report (as the coverage oracle). After that, for each code coverage tool, we can easily determine whether it is buggy by comparing the corresponding coverage report with the coverage oracle. Since a test program may have a large code size, it is time-consuming and error-prone to determine the corresponding coverage oracle. Furthermore, if a large test program is submitted as a bug report, it is also difficult for developers to determine the location of bugs in the code coverage tool(s). To tackle this problem, we use *Reducer* to reduce each inconsistency-triggering test program in *IPS* by removing the code irrelevant to the inconsistency.

For each inconsistency-triggering test program p in *IPS*, *Reducer* takes the following steps to obtain the reduced test program. At step 1, the tool C-Reduce [18] is applied to p to obtain the reduced version p'. if p' has a smaller size than p, go to step 2; otherwise, stop the reduction. At step 2, feed p' to the n coverage tools, collect the coverage reports, and determine the inconsistency category. If the inconsistency category triggered by p' is the same as the inconsistency category triggered by p, assign p' to p. The above process is repeated until p cannot be further reduced. Consequently, we reduce *IPS* to obtain *R-IPS*, the set of reduced test programs.

F. Inspecting Reduced Test Programs

With the reduced test program, we need to inspect which code coverage tools have bugs before reporting. In practice, it is usually done manually [21]. In other words, developers manually inspect the coverage reports to determine which coverage tools are buggy. To relieve the burden of manual intervention, we summarize the following rules that code coverage reports must comply with:

- *Identical Coverage*: Assuming statements `s1` and `s2` in the same block: {`s1; s2;`}. If `s1` is not a jump statement (i.e, `break`, `goto`, `return`, `exit`, or `abort` statement) and `s2` is not a `label` statement nor a `loop` (`for` or `while`) statement, `s1` and `s2` should have identical coverage.
- *Unexecuted Coverage*: Assuming statements s1 and s2 in the same block: {`s1; s2;`}. If `s1` is a `return`, `break`, `goto`, or `exit` statement and `s2` is not a labeled statement, `s2` should be never executed.
- *Ordered Coverage*: Assuming statements `s1` and `s2` form: `s1; if (...) {s2; ...}`. If `s2` is not a labeled statement, the execution time of `s1` should be no less than `s2`.

With the above rules, we develop the tool *Inspector* to examine the inconsistent coverage reports and determine which tools have bugs automatically. There is still some inconsistent coverage reports in *R-IPS* that can not be inspected automatically by our tool. We inspect those coverage reports manually. This process does not require too much human inspection effort, as the reduced test programs only have a few lines (usually less than 13 lines in our study).

G. Reporting Test Programs to Developers

For each test program in *RS-IPS*, this step simply generates bug reports for the corresponding buggy tool(s). A bug report mainly consists of the reduced test program and the affected versions. If a test program triggers multiple bugs, multiple separate bug reports will be generated.

IV. Evaluation

In this section, we first present the subject coverage tools and the testing environment. Then, we describe our experimental results in detail.

A. Subject Code Coverage Tools

In this study, we select gcov and llvm-cov as our subject code coverage tools. We choose these two code coverage tools since: (1) they have been widely used in software engineering community; and (2) they have been integrated with the most widely-used production compilers, i.e. GCC and Clang. More specifically, we chose gcov in latest development trunk of GCC and llvm-cov in the latest LLVM development trunk.

For gcov, the command flags we used to compile a given source file, e.g., `t.c`, and produce the corresponding coverage report is as follows:

```
# gcc -O0 --coverage t.c; ./a.out; gcov t.c
```

The *gcov*-generated coverage report is stored in a file named `t.c.gcov`. For llvm-cov, we use the following command:

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

The *llvm-cov*-generated coverage report is stored in a file named `t.c.lcov`. Then these two produced coverage reports will be parsed into unified format and compared to detect inconsistency. It is worth noting that we are using `-O0` option to turn off compiler optimizations. It make sense to compare the coverage reports produced in this way.

B. Testing Environment Setup

Our evaluation was conducted on a Linux server with Intel(R) Xeon(R) CPU@2.00GHz (60 cores) and 32GB RAM. The server is running on Ubuntu 17.10 (x86_64). We spent non-continuous four months, of which over one month we devoted to developing various tools. The rest of time was spent in testing the two code coverage tools, filtering out test programs, reducing test programs, and inspect test programs. Initially, we only used Csmith-generated programs as the test programs. Later, we made use of the programs selected from GCC's and Clang's test-suites as our test programs since they may cover many C semantics that Csmith does not cover. This is further confirmed in Section IV-C as a number of bugs are detected by programs inside test-suites.

C. Experimental Results

Inconsistent Coverage Reports. Table I shows the statistics of inconsistency-triggering test programs over Csmith-generated programs, GCC's test-suite, and Clang's test-suite. Column 2 shows the total number of test programs and Column 3 shows the number of test programs which run out of time (10 seconds in our experiment). We used 1 million Csmith-generated programs and collected $2,756$ and 106 C compilable programs respectively from GCC's and Clang's test-suites. Note that there are more than tens of thousands of test programs in GCC and Clang's test-suites. Only those C files that can be compiled independently are considered here. Among them, $182,927$ programs executed more than 10 seconds and hence were excluded for further analysis. The remaining test programs were fed to `C2V`. Column 4 is in the form of '*a / b*', where '*a*' refers to the total number of test programs which can lead to inconsistencies and '*b*' refers to the percentage of '*a*' over the number of all the test programs `C2V` analyzed (i.e. Column 2 − Column 3). We found $261,347$ programs leading to inconsistent coverage reports ($261,065$, 262, and 20 respectively from Csmith-generated programs, GCC's test-suite, and Clang's test-suite). About 31.95% Csmith-generated programs caused inconsistent coverage reports, much higher than those from GCC's test-suite and Clang's test-suite. Columns 5 to 11 display the distributions of inconsistency-triggering test programs over 7 different categories. In the third rows of these columns, the number in parentheses indicates the number of test programs after filtering potential test programs that trigger the same code coverage bugs. Most of inconsistent reports fell into the $C010$ category, indicating that the majority of inconsistencies belong

TABLE I
STATISTICS OF INCONSISTENCY-TRIGGERING TEST PROGRAMS.

	#Test Programs	#Time out	Inconsistency-triggering Test Programs(After filtering)							
			#Num(After filtering) / %P	C001	C010	C100	C011	C101	C110	C111
Csmith-generated	1,000,000	182,927	261,065 (758) / 31.95%	3,331 (119)	238,554 (251)	1,625 (36)	4,547 (115)	87 (33)	12,470 (151)	451 (53)
GCC's test-suite	2,756	15	262 / 9.56%	81	124	15	30	8	1	3
Clang's test-suite	106	15	20 / 21.98%	9	5	4	1	0	1	0

TABLE II
STATISTICS OF LINES OF CODE FOR THE ORIGINAL AND REDUCED VERSION OF CSMITH-GENERATED TEST PROGRAMS

	min	mean	median	max
Original	41	210	167	874
Reduced	2	10	8	27
Relative	–	4.76%	4.79%	–

TABLE III
INFORMATION OF ALL REPORTED BUGS

	gcov	llvm-cov	Total
Confirmed	42	28	70
Pending	1	5	6
Duplicate	0	4	4
Rejected	3	0	3
Reported	46	37	83

TABLE IV
BUG TYPES OF CONFIRMED BUGS

	gcov	llvm-cov	Total
Spurious Marking	18	16	34
Missing Marking	13	5	18
Wrong Frequency	11	7	18
Total	42	28	70

to the cases where some code is marked as unexecuted by gcov but as executed by llmv-cov. $C101$ category has the minimal number of inconsistent reports, indicating that inconsistencies of $Type\ A$ and $Type\ C$ rarely occur simultaneously. In addition, we can found that our method is very efficient for filtering potential "duplicate" test programs.

For those Csmith-generated inconsistency-triggering programs, our filtering strategy led to 758 test programs for reduction and inspection. Table II lists their code size before and after reduction. We can see that the mean lines of code drops from 210 to 10, thus helping effectively report code coverage bugs. For those inconsistency-triggering test programs from GCC's and Clang's test-suites, we intended to inspect all of them. The reasons are two-fold. First, programs in these test-suites are usually unique in program structures (unlike the randomly generated programs produced by Csmith). Second, the total number is not large (i.e. 262 from GCC's test-suite and 20 from Clang' test-suite). In summary, we analyzed about 1000 inconsistency-triggering test programs.

Bug Count. We filed a total of 83 bug reports for gcov and llvm-cov during our testing period. They can be found under "`yangyibiao@nju.edu.cn`" in GCC's and LLVM's Bugzilla databases. Table III shows the details of all the bugs we have reported so far. As shown in Column 4, till April 16, 2018, we have reported 83 bugs, of which 70 bugs are confirmed by developers. Of the 70 confirmed bugs, 11 are resolved as "fixed", 17 as "won't fix", and 2 as "works for me" in latest version. It is worth noting that "won't fix" indicates a confirmed bug but will not be fixed by developers. There are total 6 bugs are pending for developers' responses which are not listed in Table V. Consistent with C. Sun et al's [21] and V. Le et al's [52] studies, due to the bug management process of LLVM is not 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. Another 4 reported bugs were marked as *duplicate* by developers, since similar test programs trigger the same coverage bug inside gcov or llvm-cov. The remaining 3 reports were rejected by GCC developers. Two of them is GCC's default optimization strategy and the other is with invalid code. Table V lists all the confirmed bugs in detail, including the identity, priority, current report status, bug types, the origins of the bug-triggering test programs (i.e. Csmith-generated, GCC's or Clang's test-suite), and affected versions. Note that 'New' indicates *confirmed* in GCC's Bugzilla, and 'Assigned' refers to that the confirmed bugs are under the process of *fixing*.

Bug type. We categorize coverage bugs into three classes as mentioned in Section II-C: *Spurious Marking*, *Missing Marking*, and *Wrong Frequency*. Table IV shows the breakdown of the bug types of all the confirmed bugs. Most of the bugs are spurious marking bugs, i.e. unexecuted code is wrongly marked as executed.

Bug importance. As shown in Column 4 of Table V, all our confirmed bugs have the default priority P3 except 14 of them are reset to P5 by developers. Besides, 13 of our reported coverage bugs have been fixed by developers. Note that 3 bugs are confirmed as 'Works' which means that they are fixed in developers' version. Thus, we consider these three bugs as fixed by developers. This together shows that our reported bugs are important and worth the effort.

Program source. As shown in Column 7 of Table V, test programs from all the three main sources (i.e. Csmith-generated, GCC's test-suite, and Clang's test-suite) can trigger coverage bugs. Two programs from Clang' test-suite trigger coverage bugs of gcov, and a number of programs from GCC's test-suite can also help find coverage bugs for llvm-cov as well. It is worth noting that test programs from different sources may induce the same coverage bugs. It indeed happened in our experiment, and we only reported once.

Affected versions. We only tested gcov and llvm-cov inside the latest development trunks of GCC and Clang respectively. When we find a test case that reveals a bug in gcov or llvm-cov, we also check the corresponding compiler's stable releases against the same test case. Column 8 of Table V shows the

TABLE V
CONFIRMED BUGS

		#ID	Prio.	Status	Type	Source	Affected Ver.
1	gcov	83434	P4	New	Missing	Csmith	v7,trunk
2	gcov	83465	P3	Won't fix	Missing	Csmith	v4∼7,trunk
3	gcov	83486	P3	Won't fix	Missing	Csmith	v4∼7,trunk
4	gcov	83505	P5	New	Spurious	Csmith	v4∼7,trunk
5	gcov	83587	P5	New	Missing	Csmith	v7,trunk
6	gcov	83616	P4	Assigned	Missing	Csmith	v7,trunk
7	gcov	83617	P3	Works	Wrong Freq.	Csmith	v4∼7
8	gcov	83678	P3	Won't fix	Spurious	Csmith	v4∼7,trunk
9	gcov	83813	P3	Fixed	Missing	Cmsith	trunk
10	gcov	85163	P5	New	Wrong Freq.	Csmith	trunk
11	gcov	85178	P3	Won't fix	Missing	Csmith	v4∼7,trunk
12	gcov	85179	P4	Assigned	Missing	Csmith	trunk
13	gcov	85188	P3	Assigned	Spurious	Csmith	v4∼7,trunk
14	gcov	85197	P3	Won't fix	Wrong Freq.	Csmith	v4∼7,trunk
15	gcov	85199	P4	Assigned	Missing	Csmith	v4∼7,trunk
16	gcov	85201	P5	Assigned	Wrong Freq.	Csmith	trunk
17	gcov	85202	P3	Won't fix	Spurious	Csmith	v4∼7,trunk
18	gcov	85217	P3	Fixed	Spurious	Csmith	v4∼7,trunk
19	gcov	85218	P3	Won't fix	Spurious	Csmith	v4∼7,trunk
20	gcov	85219	P3	Won't fix	Spurious	Csmith	v4∼7,trunk
21	gcov	85225	P5	Assigned	Wrong Freq.	Csmith	v4∼7,trunk
22	gcov	85243	P3	Won't fix	Spurious	Csmith	v4∼7,trunk
23	gcov	85245	P3	Won't fix	Spurious	Csmith	v4∼7,trunk
24	gcov	85272	P3	Won't fix	Spurious	Csmith	v4∼7,trunk
25	gcov	85273	P3	Won't fix	Spurious	Csmith	v4∼7,trunk
26	gcov	85274	P3	Won't fix	Spurious	Csmith	v4∼7,trunk
27	gcov	85276	P5	New	Wrong Freq.	Csmith	trunk
28	gcov	85294	P3	Won't fix	Spurious	Csmith	v4∼7,trunk
29	gcov	85297	P3	Won't fix	Spurious	Csmith	v4∼7,trunk
30	gcov	85299	P3	Won't fix	Spurious	Csmith	v4∼7,trunk
31	gcov	85332	P3	Fixed	Wrong Freq.	GCC	v7,trunk
32	gcov	85333	P3	Won't fix	Missing	GCC	v4∼7,trunk
33	gcov	85336	P4	New	Wrong Freq.	GCC	v4∼7,trunk
34	gcov	85337	P5	New	Wrong Freq.	Clang	v7,trunk
35	gcov	85338	P3	Fixed	Wrong Freq.	Clang	v4∼7,trunk
36	gcov	85349	P5	New	Spurious	GCC	v6∼7,trunk
37	gcov	85350	P3	Fixed	Spurious	GCC	v4∼7,trunk
38	gcov	85351	P5	Assigned	Missing	GCC	v4∼7,trunk
39	gcov	85367	P3	Works	Wrong Freq.	GCC	v5∼7,trunk
40	gcov	85370	P3	Fixed	Spurious	GCC	trunk
41	gcov	85372	P3	Fixed	Missing	GCC	trunk
42	gcov	85377	P4	New	Missing	GCC	v4∼7,trunk
43	llvm-cov	33465	P3	Fixed	Missing	Csmith	v3∼5,trunk
44	llvm-cov	35404	P3	Confirmed	Spurious	Csmith	trunk
45	llvm-cov	35426	P3	Fixed	Spurious	Csmith	trunk
46	llvm-cov	35495	P3	Fixed	Missing	Csmith	trunk
47	llvm-cov	35556	P3	Fixed	Missing	Csmith	trunk
48	llvm-cov	37001	P3	Confirmed	Spurious	Csmith	trunk
49	llvm-cov	37003	P3	Confirmed	Spurious	Csmith	trunk
50	llvm-cov	37012	P3	Confirmed	Spurious	Csmith	trunk
51	llvm-cov	37017	P3	Confirmed	Spurious	Csmith	trunk
52	llvm-cov	37043	P3	Confirmed	Spurious	Csmith	trunk
53	llvm-cov	37046	P3	Confirmed	Spurious	Csmith	trunk
54	llvm-cov	37070	P3	Confirmed	Missing	Clang	v3∼5,trunk
55	llvm-cov	37071	P3	Confirmed	Spurious	Clang	trunk
56	llvm-cov	37072	P3	Confirmed	Wrong Freq.	GCC	trunk
57	llvm-cov	37081	P3	Confirmed	Missing	Clang	v3∼5,trunk
58	llvm-cov	37083	P3	Confirmed	Wrong Freq.	GCC	trunk
59	llvm-cov	37084	P3	Confirmed	Wrong Freq.	GCC	trunk
60	llvm-cov	37085	P3	Confirmed	Spurious	GCC	trunk
61	llvm-cov	37090	P3	Confirmed	Spurious	GCC	trunk
62	llvm-cov	37092	P3	Confirmed	Spurious	GCC	trunk
63	llvm-cov	37099	P3	Confirmed	Spurious	GCC	trunk
64	llvm-cov	37102	P3	Confirmed	Spurious	GCC	trunk
65	llvm-cov	37103	P3	Confirmed	Wrong Freq.	GCC	v3∼5,trunk
66	llvm-cov	37105	P3	Confirmed	Spurious	GCC	trunk
67	llvm-cov	37107	P3	Confirmed	Wrong Freq.	GCC	v3∼5,trunk
68	llvm-cov	37124	P3	Confirmed	Wrong Freq.	GCC	trunk
69	llvm-cov	37125	P3	Confirmed	Wrong Freq.	Clang	v3∼5,trunk
70	llvm-cov	37126	P3	Confirmed	Spurious	GCC	v3∼5,trunk

```
gcov | llvm-cov | #N | Source Code
10 :      10 |  1 | void func(int i) {
10 :      10 |  2 |   switch (i) {
 1 :      10 |  3 |     case 1: break;
 9 :      10 |  4 |     case 2: ;
 9 :       9 |  5 |     default: break;
 − :      10 |  6 |   }
10 :      10 |  7 | }
 − :         |  8 |
 1 :       1 |  9 | int main() {
11 :      11 | 10 |   for (int i=0;i<10;++i)
10 :      10 | 11 |     func(i);
 1 :       1 | 12 |   return 0;
 − :       1 | 13 | }
```

Fig. 6. One program triggering multiple bugs

affected various versions of GCC and Clang for each bug. Note that as for GCC, we select GCC-4.8.0, GCC-5.4.1, GCC-6.4.0, GCC-7.2.0, and the latest trunk version (GCC-8.0), and for Clang, we select Clang-3.8, Clang-4.0, Clang-5.0, and the latest trunk version (Clang-7.0). As can be seen, 42 out of all the bugs can affect stable releases, and a considerable number of bugs have been latent in the old stable releases (such as GCC-4.8.0 and Clang-3.8) for many years.

D. Interesting Bugs Found

One program triggering multiple bugs. In Figure 6, Column 4 lists a code snippet and Column 3 shows the line numbers. Column 1 and Column 2 display the coverage results by gcov and llvm-cov respectively. This code snippet is a simple *switch*-statement inside a *for*-loop from the caller. The code at line 3 and line 4 actually executes only one time, however, they are wrongly marked as executed 10 times by llvm-cov and line 4 is wrongly marked as executed 9 times by gcov. We have reported this case as Bug #85337 of gcov and Bug #37124 of llvm-cov. This case is quite simple and common in real world. Unfortunately, this single case triggers coverage bugs for both gcov and llvm-cov, indicating that coverage bugs are subtle and prevalent.

Coverage bugs or compiler bugs? Figure 7 shows the coverage report for Bug #37082 of Clang. A definition of `strcmp` function is first given at line 2, while `strcmp` is redefined as the libc function. As a result, the calling of `strcmp` at line 6 will invoke the libc function with internal linkage, instead of the self-defined function (line 2). However, in our test, GCC outputs "−1" whereas Clang outputs "20" for this case. That means, GCC invokes the correct libc function but Clang still invokes the self-defined function at line 2. That is why line 2 is reported as executed one time by llvm-cov but as not executed by gcov. As for this case, the root cause of producing the incorrect coverage report is not the bugs inside llvm-cov, but the bugs inside Clang instead.

Code formatting problem. Figure 8 shows the coverage report for Bug #37102 of Clang. Line 8 is marked as executed twice by gcov, but wrongly marked as executed only once in llvm-cov. Along the execution, we can see that each of the four statements at line 8 (i.e. `foo(); goto L2; L1: bar()`) executes actually only one time before the `main` function finishes. However, from the perspective of line coverage, the

```
1 |   | #include <stdio.h>
2 | 1 | static int strcmp(){return 2;}
3 | 1 | #define strcmp __builtin_strcmp
4 |   | int main()
5 | 1 | {
6 | 1 |   int ret = strcmp(”a”,”b”);
7 | 1 |   printf(”%d\n”, ret);
8 | 1 |   return 0;
9 | 1 | }
```

Fig. 7. Coverage or compiler bug (Bug #37082 of llvm-cov)

```
 1 |   | int a = 0;
 2 |   |
 3 | 1 | void foo() { a++; }
 4 | 1 | void bar() { a++; }
 5 |   |
 6 |   | int main()
 7 | 1 | {
 8 | 1 |   foo(); goto L2; L1: bar();
 9 | 1 |
10 | 2 |   L2:
11 | 2 |     if (a == 1)
12 | 1 |       goto L1;
13 | 1 | }
```

Fig. 8. Code formatting problem (Bug #37102 of llvm-cov)

```
 gcov   | llvm-cov | #N | Source Code
    – : |          |  1 | #include <stdio.h>
    – : |          |  2 | #include <setjmp.h>
    – : |          |  3 |
    1 : |        1 |  4 | int foo(jmp_buf b) {longjmp(b,1);}
    – : |          |  5 |
    1 : |          |  6 | int main()
    – : |        1 |  7 | {
    – : |        1 |  8 |   int ret;
    – : |        1 |  9 |   jmp_buf buf;
    – : |        1 | 10 |
    1 : |        1 | 11 |   if(setjmp(buf)) {
#####   :        0 | 12 |     foo(buf);
    – : |        0 | 13 |   }
    – : |        1 | 14 |
    1 : |        1 | 15 |   if(setjmp(buf)!= 0) {
    1 : |        1 | 16 |     ret = 0;
    – : |        1 | 17 |   } else {
    1 : |        0 | 18 |     ret = 1;
    1 : |        0 | 19 |     foo(buf);
    – : |        0 | 20 |   }
    – : |        1 | 21 |
    1 : |        1 | 22 |   printf(”%d”, ret);
    – : |        1 | 23 | }
```

Fig. 9. Non-trivial inspection (Bug #37081 of Clang)

code at line 8 executes twice. For the first time, the first two statements, i.e. `foo()` and `goto L2`, get executed and then the control flow jumps to line 10. After executing the *goto*-statement at line 12, the control flow jumps back to line 8. Then the last two statements, i.e. `L1: bar();`, get executed. Note that the coverage result will be correct if we put the four statements at line 8 into four separate lines.

Non-trivial inspection. Figure 9 follows the same notions with Figure 6. Lines 18 and 19 are marked as executed by gcov but as unexecuted by llvm-cov. Human inspection is conducted to determine whether gcov or llvm-cov produces the incorrect coverage report. But this process is non-trivial. Intuitively, the two branches of the *if*-statement at line 15 should not be executed simultaneously, implying that the coverage report by gcov is incorrect. Besides, this code outputs “0” instead of “1” at runtime, further supporting the implication. However, it is actually llvm-cov that produces the incorrect report (Bug #37081 of Clang). Function `setjmp` and function `longjmp` are provided to perform complex flow-of-control in C. Due to their existence, the execution flows for this code are: (1) the *if*-statement at line 11 takes the false branch, (2) then the *if*-statement at line 15 also takes the false branch, assigning variable `ret` as 1 and calling function `foo` at line 4, (3) function `longjmp` restores the program state when `setjmp` at line 15 are called and returns 1, hence taking the true branch at line 15. As a result, variable `ret` is assigned as 0. (4) the `main` function returns after printing the value of variable `ret`.

E. Limitations

In this study, we assess the reliability of code coverage tools via differential testing. This is a first effort towards this direction. However, our technique has a number of limitations. *First*, most of the test programs we used were generated by Csmith. The Csmith-generated programs only cover a subset of C semantics, which might cause `C2V` to miss a number of code coverage defects. As a complement, we collected 2862 C programs from GCC’s and Clang’s test-suites, and fed them to `C2V`, which can mitigate this problem to some extent. *Second*, we only take account inconsistency-triggering lines of code for computing program similarities to filter out test programs that potentially trigger the same code coverage bugs. In this study, a large number of inconsistency-triggering test programs are filtered out which may miss a number of quality test programs. If we can inspect all Csmith-generated inconsistency-triggering test programs, it is reasonable to expect that more code coverage bugs would be found. *Third*, it is possible that both code coverage tools may have the same bugs. In other words, these two coverage tools might produce same but incorrect code coverage reports for a given program. Our approach can not identify any inconsistencies from such paired coverage reports and further miss this kind of bugs. Therefore, in the future, more research efforts should be paid in this area to improve the quality of code coverage tools. *Forth*, different code coverage tools having different implementations may make the coverage reports difficult to be compared. To mitigate this problem, we have taken the following steps: (1) we reformatted the generated test programs before feeding them to the coverage tools, which led to formatted and comparable coverage reports; (2) before comparing coverage reports, we identified and excluded specific behavioral differences of the coverage tools; and (3) before reporting bugs, we inspected inconsistent coverage reports to determine which tools are buggy. During inspection, false alarms are manually identified. Therefore, we have taken careful steps to reduce false positives resulting from the variability among different tools. Besides, it is also interesting to develop more accurate techniques for coverage reports comparison in the future.

V. Related Work

This section introduces the related work on randomized differential testing, coverage-directed differential testing, and testing via equivalence modulo inputs.

A. *Randomized Differential Testing*

Differential testing is originally introduced by McKeeman [53] which attempt to detect bugs by checking inconsistent behaviors across different comparable software or different software versions. Randomized differential testing is a widely-used black-box differential testing technique in which the inputs are randomly generated [19], [54]. Yang et al. [19] developed Csmith, a randomized test case generation tool that can support a large subset of C features and avoid introducing undefined and unspecified behaviors, to find C compiler bugs. Lidbury et al. [40] developed CLsmith, a tool built on top of Csmith, to validate OpenCL compilers based on differential testing and testing via equivalence modulo inputs (EMI). They presented several strategies for random generation of OpenCL kernels and an injection mechanism which allowed EMI testing to be applied to kernel in order to avoid little or no dynamically-dead code. Their study revealed a significant number of OpenCL compiler bugs in commercial implementations. Sun et al. [21] applied randomized differential testing to find and analyze compiler warning defects across GCC and LLVM. In less than six months, they successfully found 52 confirmed/fixed bugs. Different from prior studies, we apply randomized differential testing to find code coverage bugs which we believe is an important topic.

B. *Coverage-based Differential Testing*

A number of recent studies leverage coverage to improve the effectiveness of differential testing. Chen et al. [55] proposed a coverage-directed fuzzing approach to detecting inconsistencies between different implementations of Java Virtual Machine (JVM). They mutated seeding classfiles, executed mutants on a reference JVM implementation, and used coverage uniqueness as a discipline for accepting representative mutants. The accepted mutants were then used as the inputs to differentially test different JVM implementations. Pei et al. [56] proposed DeepXplore, a whitebox coverage-directed differential testing for detecting inconsistencies between multiple DNNs. They first introduced neuron coverage as a systematic metric for measuring how much of the internal logic of a DNNs had been tested and then used this information to guide the testing process. As can be seen, the prerequisite of the above techniques is to obtain the correct coverage. Our work provides a general and practical approach to finding coverage bugs, thus helping improve the quality of code coverage tools.

C. *Testing via Equivalence Modulo Inputs*

Testing via equivalence modulo inputs is a new testing technique proposed in recent years. In nature, EMI testing is a kind of metamorphic testing, which modifies a program to generate variants with the same outputs as the original program [57], [58]. Le et al. [7] proposed to generate equivalent versions of the program by profiling program's execution and pruning unexecuted code. Once a program and its equivalent variant are constructed, both are used as input of the compiler under test, checking for inconsistencies in their results. So far, this method has been used to detect 147 confirmed bugs in two open source C compilers, GCC and LLVM. Based on this idea, Athena [59] and Hermes [60] are developed subsequently. Athena [59] generates EMI by randomly inserting code into and removing statements from dead code regions. Hermes [60] complements mutation strategies by operating on live code regions, which overcomes the limitations of mutating dead code regions. Le et al. [52] first used Csmith to generate single-file test programs and transformed each single-file test program into multiple compilation units. Then, they stochastically assigned each unit an optimization level to thoroughly exercise link-time-optimizers. They discovered and reported 37 LTO bugs for GCC and LLVM in 11 months. These techniques heavily depend on the code coverage information.

VI. Conclusion and Future Work

We proposed a randomized differential testing approach to hunting code coverage bugs and implemented a tool named `C2V` to test two C code coverage tools, gcov and llvm-cov. Our evaluations where 42 and 28 bugs confirmed from gcov and llvm-cov respectively in a short few months provided a strong evidence that code coverage tools are not as reliable as they might have been envisaged. Overall, our approach has the following main advantages: (1) it simplifies the difficult code coverage validation problem as a simple comparison problem; (2) the comparison between code coverage reports not only checks whether a program chunk gets executed or not, but also the exact execution frequencies. Any discrepancy in these dimensions would alert a potential bug report, which helps find subtle but deep semantic bugs in code coverage tools; and (3) our approach is simple, straightforward, and general. It can be easily applied to validate different code coverage tools, under various programming languages and coverage criteria. In the future, more efforts should be paid on this area and there is a need to examine the influence of those bugs on other techniques which depend on code coverage.

Acknowledgment

We thank Yanyan Jiang, Zhaogui Xu, and 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 Natural Science Foundation of China (61702256, 61772259, 61432001, 61832009, 61772263, 61802168, 61872177), the Natural Science Foundation of Jiangsu Province (BK20170652), the China Postdoctoral Science Foundation (2018T110481), the Fundamental Research Funds for the Central Universities (020214380032, 02021430047), the National Key R&D Program of China (2018YFB1003901). Zhendong Su was supported by United States NSF Grants 1528133 and 1618158, and Google and Mozilla Faculty Research awards. Yuming Zhou (zhouyuming@nju.edu.cn) and Baowen Xu (bwxu@nju.edu.cn) 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] 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.

[3] 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 2013. New York, NY, USA: ACM, 2013, pp. 291–301.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

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

[9] D. Hao, L. Zhang, L. Zang, Y. Wang, X. Wu, and T. Xie, "To be optimal or not in test-case prioritization," *IEEE Trans. Softw. Eng.*, vol. 42, no. 5, pp. 490–505, May 2016.

[10] 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.

[11] 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 2014. New York, NY, USA: ACM, 2014, pp. 215–224.

[12] 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 2016. New York, NY, USA: ACM, 2016, pp. 881–896.

[13] 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.

[14] 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 &#38; Applications*, ser. OOPSLA '13. New York, NY, USA: ACM, 2013, pp. 19–32.

[15] 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.

[16] "llvm-cov," https://llvm.org/docs/CommandGuide/llvm-cov.html.

[17] "Clang," http://clang.llvm.org/.

[18] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang, "Test-case reduction for c compiler bugs," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '12. New York, NY, USA: ACM, 2012, pp. 335–346.

[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] Y. Chen, A. Groce, C. Zhang, W.-K. Wong, X. Fern, E. Eide, and J. Regehr, "Taming compiler fuzzers," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '13. New York, NY, USA: ACM, 2013, pp. 197–208.

[21] 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.

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

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

[24] "Testwell ctc++," http://www.testwell.fi/ctcdesc.html.

[25] "Covtool," http://covtool.sourceforge.net/.

[26] "Jcov," https://wiki.openjdk.java.net/display/CodeTools/jcov.

[27] "Jacoco," https://www.jacoco.org/.

[28] "Emma: a free java code coverage tool," http://emma.sourceforge.net/.

[29] "Atlassian clover," https://www.atlassian.com/software/clover.

[30] "Codecover," http://codecover.org/.

[31] "Openclover," http://openclover.org/.

[32] P. Ammann and J. Offutt, *Introduction to software testing*. Cambridge University Press, 2016.

[33] G. J. Myers, T. Badgett, T. M. Thomas, and C. Sandler, *The art of software testing*. Wiley Online Library, 2004, vol. 2.

[34] 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.

[35] 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 2012. New York, NY, USA: ACM, 2012, pp. 331–341.

[36] 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.

[37] 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.

[38] 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.

[39] K. Jamrozik, G. Fraser, N. Tillmann, and J. De Halleux, "Augmented dynamic symbolic execution," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2012. New York, NY, USA: ACM, 2012, pp. 254–257.

[40] 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.

[41] Q. Zhang, C. Sun, and Z. Su, "Skeletal program enumeration for rigorous compiler testing," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2017. New York, NY, USA: ACM, 2017, pp. 347–361.

[42] C. Lemieux and K. Sen, "Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE 2018. New York, NY, USA: ACM, 2018, pp. 475–485.

[43] 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.

[44] T. Wang and A. Roychoudhury, "Automated path generation for software fault localization," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '05. New York, NY, USA: ACM, 2005, pp. 347–351.

[45] 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.

[46] 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.

[47] J. Xuan and M. Monperrus, "Test case purification for improving fault localization," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 52–63.

[48] X. Wang, N. Zeldovich, M. F. Kaashoek, and A. Solar-Lezama, "Towards optimization-safe systems: Analyzing the impact of undefined behavior," in *Proceedings of the Twenty-Fourth ACM Symposium on*

Operating Systems Principles, ser. SOSP '13. New York, NY, USA: ACM, 2013, pp. 260–275.

[49] E. Eide and J. Regehr, "Volatiles are miscompiled, and what to do about it," in *Proceedings of the 8th ACM International Conference on Embedded Software*, ser. EMSOFT '08. New York, NY, USA: ACM, 2008, pp. 255–264.

[50] A. Balestrat, "Ccg: A random c code generator," *https://github.com/Merkil/ccg*, 2015.

[51] P.-N. Tan, M. Steinbach, and V. Kumar, "Association analysis: basic concepts and algorithms," *Introduction to Data mining*, pp. 327–414, 2005.

[52] 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 2015. New York, NY, USA: ACM, 2015, pp. 327–337.

[53] W. M. McKeeman, "Differential testing for software," *DIGITAL TECHNICAL JOURNAL*, vol. 10, no. 1, pp. 100–107, 1998.

[54] A. Groce, G. Holzmann, and R. Joshi, "Randomized differential testing as a prelude to formal verification," in *Proceedings of the 29th International Conference on Software Engineering*, ser. ICSE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 621–631.

[55] Y. Chen, T. Su, C. Sun, Z. Su, and J. Zhao, "Coverage-directed differential testing of jvm implementations," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '16. New York, NY, USA: ACM, 2016, pp. 85–99.

[56] K. Pei, Y. Cao, J. Yang, and S. Jana, "Deepxplore: Automated whitebox testing of deep learning systems," in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP '17. New York, NY, USA: ACM, 2017, pp. 1–18.

[57] T. Y. Chen, F.-C. Kuo, H. Liu, P.-L. Poon, D. Towey, T. H. Tse, and Z. Q. Zhou, "Metamorphic testing: A review of challenges and opportunities," *ACM Comput. Surv.*, vol. 51, no. 1, pp. 4:1–4:27, Jan. 2018.

[58] S. Segura, G. Fraser, A. B. Sanchez, and A. Ruiz-Corts, "A survey on metamorphic testing," *IEEE Trans. Softw. Eng.*, vol. 42, no. 9, pp. 805–824, Sept 2016.

[59] 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 2015. New York, NY, USA: ACM, 2015, pp. 386–399.

[60] 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 2016. New York, NY, USA: ACM, 2016, pp. 849–863.