DEBCOVDIFF: Differential Testing of Coverage Measurement Tools on Real-World Projects

Wentao Zhang, Jinghao Jia, Erkai Yu, Darko Marinov, Tianyin Xu
University of Illinois Urbana-Champaign, Urbana, IL, USA
{wentaoz5, jinghao7, erkaiyu2, marinov, tyxu}@illinois.edu

***Abstract*—Measuring code coverage is a critical practice in software testing. Incorrect or misleading coverage information reported by automatic tools can increase the software development cost and lead to negative consequences especially for safety-critical software. Ensuring the correctness of coverage measurement tools is therefore important. Prior studies have applied various techniques to find bugs in Gcov and LLVM-cov, the two most widely used coverage tools for C/C++. However, those studies had two limiting factors. First, they used only small, often synthetic, programs, potentially missing bugs in real-world scenarios. Second, they focused only on basic line coverage, neglecting advanced metrics that are both more complex to implement and commonly required for safety-critical software.**

This paper presents the first empirical study of coverage measurement tools for real-world projects. We implement DEBCOVDIFF, a testing framework that takes Debian packages as the input programs and performs differential testing of Gcov and LLVM-cov, for line coverage and two advanced coverage metrics. We design robust differential oracles to (1) filter out discrepancies arising from subtle differences in the tool output presentation, (2) overcome the nondeterministic nature of certain packages, and (3) support advanced coverage metrics. From results on 47 Debian packages, we identify 34 new bugs, including 2 crashing bugs and 32 deeper bugs that produce wrong coverage reports.

I. INTRODUCTION

Measuring code coverage [1] is a critical step in software testing. Code coverage reflects which portion of code-under-test is covered during execution and gives an important indicator of test-suite adequacy [2]. Various coverage metrics have been proposed, including line coverage, branch coverage, and modified condition/decision coverage (MC/DC) [3]–[5]. Beyond measuring test-suite adequacy, code coverage has been widely used in fault localization [6], fuzz testing [7], compiler validation [8], flaky-test detection [9], etc. Code coverage is incorporated in software development cycles, e.g., by companies such as Google [10], [11], and for certification in safety-critical industries, such as aviation [12] and automotive [13].

To facilitate the measurement of code coverage, automatic tools have been developed, either open-source [14], [15] or commercial [16], [17], for different programming languages [18], [19]. Ensuring the correctness of these coverage measurement tools is crucial given the diverse uses of code coverage. For example, Google reports how difficulties with the tools frustrate the developers and make them less willing to adopt code coverage in their workflow [10]. Moreover, incorrect or misleading coverage reported by the tools can delay software releases [10], reduce bug-detection effectiveness [2], and give developers a false sense of assurance [12], [13].

While ensuring the reliability of coverage measurement tools is important, it is also challenging due to the absence of specifications for code coverage and the difficulty in crafting precise test oracles for tool output. Prior work [20]–[23] used several techniques—including differential testing [24] and automated test generation [25]—to reveal bugs in Gcov [14] and LLVM-cov [15], the two most widely used coverage tools for C/C++. For example, by identifying different line coverage counters between reports produced by the two tools, C2V [20] found 42 and 28 bugs in Gcov and LLVM-cov, respectively.

However, prior studies [20]–[23] have two limitations. First, they used only small, synthetic programs as test inputs, (e.g., using Csmith [25]) or samples of the tool's compiler test cases (GCC and LLVM). The program diversity is limited by the design of the synthesizer [26] or developers' manual efforts. On the other hand, real projects exhibit high diversity in terms of language syntax, project-specific idioms, software bugs, undefined behavior [27], and project scale and organization, which is hard to capture by small test inputs. While testing the tools with synthetic inputs is valuable, testing *only* with synthetic inputs can be insufficient to assure the reliability for real software. Bugs found with synthetic programs may not occur in real projects [28]; more importantly, synthetic programs may miss bugs that do occur in real projects [26].

Second, prior work focused only on basic line coverage, neglecting advanced metrics such as branch coverage (III-D2) and MC/DC (III-D3). However, these advanced metrics are potentially harder to implement correctly [29]–[31]; moreover, they are arguably more important and commonly required for safety-critical software [12], [13]. Testing coverage measurement tools for advanced metrics is more challenging than for line coverage because of the lack of standard representations in coverage reports (see Section III-D).

To illustrate the above two points, Listings 1 and 2 show a bug revealed in our work and reported as LLVM#131505. It originates from line 313 in `hostname.c` of the package `hostname` (Listing 1), where LLVM-cov reports the second condition (macro `IN6_IS_ADDR_MC_LINKLOCAL`) to have two outcomes, and strikingly, this condition was evaluated as `true` 18.4E (18.4×10^{18}) times during test execution. Gcov reports this condition to have four outcomes and none executed, so our differential oracle flags the difference. (Prior work on line coverage [20]–[23] misses this bug because it is about branch coverage.) The LLVM-cov report is self-contradictory: (1) all lines in the snippet are reported not executed (line coverage 0, left of '|'), (2) the preceding condition (`IN6_IS_ADDR_LINKLOCAL`

```
312  0 | if (IN6_IS_ADDR_LINKLOCAL(&sin6->sin6_addr) ||
     ----------------------------------------------------
     | Branch (2 outcomes): [0,0]
     ----------------------------------------------------
313  0 |     IN6_IS_ADDR_MC_LINKLOCAL(&sin6->sin6_addr))
     ----------------------------------------------------
     | Branch (2 outcomes): [18.4E,1]
     ----------------------------------------------------
314  0 |   continue;
```

Listing 1. A bug found with `hostname-3.23+nmu1/hostname.c`.

```
     // test.h
1    | #define FOO(x) ((x)[0] && x[1])
     // main.c
1    | #include <test.h>
2    | int g = 1, buf[100];
3    1 | int main(void) {
4    1 |     if (g || FOO(buf)) { return 1; }
     ----------------------------------------------------
     | Branch (4 outcomes): [1,0,18.4E,1]
     ----------------------------------------------------
     | MC/DC (2 conditions): [F,F]
     ----------------------------------------------------
5    0 |     return 0;
6    1 | }
```

Listing 2. Reduced input program for LLVM#131505.

at line 312) is reported never evaluated (both its `true` and `false` outcomes are "executed" 0 times, while it should have been `false` sometimes for the second condition to even be evaluated at all), and (3) the value of 18.4E is too large given the test execution time.

To report this new bug to the developers, we reduced the code (Listing 2) and summarized the triggering conditions as a program with (1) a macro from a system header (line 1 of `main.c`), where (2) the macro definition involves subscripting the argument, `x`, in parentheses (line 1 of `test.h`). The bug stems from the way LLVM-cov tries to improve performance of coverage measurement by not adding a counter for each basic block but computing some dependent counters based on other counters. Specifically, LLVM-cov calculates one condition counter value from the relationship $C_{Total} = C_{True} + C_{False}$, and actually maintains only two counter variables, deriving the third value. In this case, an incorrect $C_{False} = 1$ and the correct $C_{Total} = 0$ lead to the wrong $C_{True} = -1$; the counter value "18.4E" is -1 in 64-bit two's complement. In addition to this negative value, the same code makes LLVM-cov report lower numbers of branch outcomes and MC/DC conditions (4 and 2, respectively) than Gcov (6 and 3, respectively).

From this example, we can distill at least two lessons. First, one metric (e.g., branch coverage) may be wrong even if another metric (e.g., line coverage) is correct. Second, program synthesizers, such as Csmith [25], lack expressiveness and may not generate specific complex conditions needed to reveal bugs, e.g., this case needs multiple files (`.h` and `.c`). In contrast, real projects are naturally organized in multiple files, which can challenge coverage measurement tools.

To this end, we developed a new testing framework, named DEBCOVDIFF, that takes real code, namely Debian packages, as the input programs and performs differential testing of coverage measurement tools, namely Gcov and LLVM-cov. Debian packages have been used for differential testing of compilers [28], but applying the methodology to coverage measurement tools faces new challenges—while compiler bugs manifest through miscompilation [28], coverage measurement bugs require specialized test oracles [20]–[23]. Moreover, coverage measurement involves additional steps over general compiler testing ("build-run-measure" vs. "build-run") and relies on additional toolchains (e.g., `compiler-rt`, `llvm-profdata`, and `llvm-cov`). Specifically, to address advanced coverage metrics (namely, branch coverage and MC/DC) that lack standard representations in coverage reports, DEBCOVDIFF develops ways to canonicalize coverage reports for differential analysis. To utilize real code as test inputs, DEBCOVDIFF addresses nondeterminism of program execution by automatically filtering out coverage reports of nondeterministic code paths. (Prior work [20]–[23] assumed small synthetic code to be deterministic.)

By applying DEBCOVDIFF on 47 Debian packages, we conduct the first empirical study of two tools, Gcov and LLVM-cov, on real code. We make these contributions:

- **Framework for real-world software:** We develop DEBCOVDIFF for testing coverage tools on real code. We select 47 Debian packages as the input programs to test two tools (Gcov and LLVM-cov) for three metrics (line coverage, branch coverage, and MC/DC).
- **Differential testing oracles:** DEBCOVDIFF performs differential testing of the coverage reports generated by the two tools, using robust oracles to (1) filter out benign discrepancies arising from subtle differences in the tool output presentation and (2) overcome the nondeterministic nature of certain packages.
- **Bug reports:** We identify 34 new bugs, including 2 crashing bugs (in LLVM-cov) for 6 packages and 32 deeper bugs (in LLVM-cov and Gcov) that produce wrong coverage reports for some of other 41 packages.

The new bugs we find (1) are in very recent versions of Gcov and LLVM-cov (which potentially include fixes for bugs found in prior work [20]–[23]) and (2) stem from real-world projects, thus, arguably more likely to be encountered by developers. Our artifact is at https://github.com/xlab-uiuc/DebCovDiff.

II. BACKGROUND AND SCOPE

Code coverage informs developers which portion of code is executed ("covered"), usually after running tests. Guided by the coverage information—*if* it is correct!—the developers can improve their code or tests accordingly [1].

Various code coverage metrics and tools exist [32]. Gcov [14] and LLVM-cov [15] are the most popular tools for C/C++. In addition to the basic line coverage (related to statement coverage [32]), these tools also measure *branch coverage* (related to decision coverage [32]) which tells whether different outcomes of conditional branches are executed. Both Gcov [31] and LLVM-cov [30] have recently added support for *modified condition/decision coverage* (MC/DC) [5], an even more rigorous metric that requires (1) each Boolean condition in a decision to be executed with both `true` and `false`

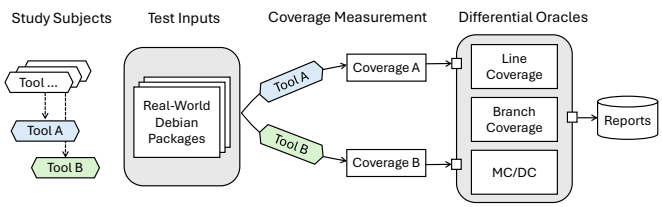

Fig. 1. High-level workflow of DEBCOVDIFF.

outcomes; and (2) the condition in question to independently affect the decision's outcome. Given its power, MC/DC is required for software certification in many safety-critical industries, including aviation [12] and automotive [13].

An important difference is between precise "source coverage" (e.g., produced by Gcov and LLVM-cov) and "approximate coverage" (e.g., used in grey-box fuzzers such as AFL [33] and libFuzzer [34]). In general, the former creates coverage reports as *source code* annotated with extra coverage information (line counters, branch counters, etc.) *after* a certain execution (e.g., program has exited), while the latter is consumed by the fuzzer and more light-weight to be measured at runtime. (Exceptions exist, e.g., classfuzz [7] uses Gcov for fuzzing.) *We focus on source coverage tools based on compiler instrumentation*, namely Gcov and LLVM-cov, because these tools are the most widely used for C/C++ and aim at presenting *precise*, not approximate, coverage.

III. METHODOLOGY

We describe the methodology of our study and the design of DEBCOVDIFF. DEBCOVDIFF follows the common flow of differential testing (Figure 1)—it gives the same test input (program) to two tools, generates respective coverage measurements, and produces a report based on differential oracles.

A. Study Subjects

DEBCOVDIFF can compare coverage measurement tools that produce at least one of line coverage, branch coverage, or MC/DC (§II). We focus on Gcov and LLVM-cov for C/C++, because they are widely adopted, open-source, and have been extensively tested by prior research [20]–[23]. We report all results on the versions from September 2024, specifically GCC `c1fb78fb` and LLVM `ce9a2c65`. We use LLVM-cov in its "source-based code coverage" mode [35] because it (1) supports MC/DC and (2) was used in prior studies [20]–[23]. (The LLVM project provides more modes for coverage instrumentation, including, a mode emulating Gcov, profile-guided optimization, and SanitizerCoverage [36].) To add a tool to DEBCOVDIFF, we provide information on (1) the executable programs for compilation (`CC`, `CXX`, `LD`, `AR`, etc.), e.g., `clang++`; (2) the build flags for coverage measurement, e.g., `-fprofile-instr-generate`, `-fcoverage-mapping`, `-fcoverage-mcdc`; and (3) the postprocessing steps, e.g., invocation of `llvm-profdata` and `llvm-cov`. Following prior work [20]–[23], we compile input programs with `-O0` to disable all optimizations, because coverage reports get even noisier with optimizations enabled [20].

B. Test Inputs

We test coverage tools with real-world projects. We choose Debian packages, because they (1) are used by prior work on testing compilers (but not coverage tools) [28], (2) represent popular and diverse real-world software, (3) are extensively tested under default settings for distribution and *may* build with the coverage tools, (4) can be compiled and tested using a general infrastructure `sbuild` [37], and (5) offer "reproducible builds" to some degree [38]–[40], which is important to reduce nondeterminism that could mislead differential testing.

We select a subset of all Debian packages, aiming for importance and representativeness, using the following procedure:

- Debian 12.8 has 63,417 binary packages for amd64 architecture in the "main" component [41].
- We start from the 103 binary packages whose "priority" field [42] is in the highest groups "required", "important", and "standard": 33, 32, and 38 packages, respectively.
- Because multiple *binary* packages come from the same *source* package (e.g., the binary packages `mount`, `fdisk`, and more come from the source package `util-linux`), we identify 78 unique source packages for 103 binary packages and use source packages as the unit of our study.
- 57 out of 78 packages are written primarily in C/C++. We attempt to measure coverage (§III-C) for all 57.
- For 32 out of 57 packages, we can successfully generate coverage reports using both Gcov and LLVM-cov, so we conduct differential testing (§III-F) with all 32 packages.
- We randomly pick 10 more packages from lower-priority groups; 9 out of 10 packages successfully generate coverage reports using both Gcov and LLVM-cov.

In total, we use 41 packages to conduct differential testing. (Sections IV-A and IV-F describe why coverage reports were not generated for 26 packages.) Our final selection of 41 packages spans a diverse range of functionalities, including essential system utilities (e.g., `coreutils`, `util-linux`), networking tools (`inetutils`, `iproute2`, `traceroute`), compression libraries (`bzip2`, `xz-utils`, `lz4`), services and their management (`apache2`, `cron`, `dbus`), and user-facing tools (`nano`, `wget`, `man-db`). We believe that our selection is both relevant and heterogeneous to represent real code for evaluating coverage tools.

C. Coverage Measurement

We use three general steps to obtain coverage measurement: (1) compile packages with coverage instrumentation, (2) execute compiled binary, and (3) generate coverage reports. We integrate these steps into Debian's own `sbuild` [37] infrastructure, which initiates the build for each package in a clean `chroot`. To make our results reproducible and to enable easier inspection, we allocate an isolated `chroot` and reuse it for the same $\langle \text{tool}, \text{flags} \rangle$ pair.

By default, `sbuild` builds every package using the GCC version of that distro (namely, GCC 12.2.0 in Debian 12.8), although an override can be explicitly specified in certain

packages. To experiment with arbitrary $\langle \text{tool}, \text{flags} \rangle$ pairs, we build on top of the best practices for building Debian packages with different compilers [28], [43]. We replace the installed `/usr/bin/{gcc,g++,cpp}` with a wrapper script so we can (1) drop in the compiler-toolchain-under-test and (2) force the desired build flags for coverage measurement (as registered to DEBCOVDIFF, §III-A). Our wrapper script provides higher flexibility than prior work [28] for hooking compiler invocations and manipulating build flags. Marcozzi et al. [28] specify build flags via `/etc/dpkg/buildflags.conf`, which works for most packages, but flags from this file are *appended* to the package-specific flags, so using their approach would have not allowed us to force `-O0` on all packages. As in [28], [43], we use the "hold" feature of `dpkg` to prevent our scripts from being overwritten by the system package manager.

To run the compiled code and collect its coverage, we use two approaches. First, we add *simple commands* to run the executable binary produced for each package. For example, for `grep`, our commands call `grep` once to find a string and once to not find a string. The `lzo2` package produces a library; to run its code, we build and run one of its sample applications. Second, we experiment with *existing tests* shipped with Debian packages, namely `dh_auto_test` [44], to understand how more thorough test suites affect our results (see Section IV-G).

After program execution, DEBCOVDIFF runs the postprocessing steps to generate coverage reports (§III-A). We save coverage reports in JSON (already supported by both Gcov and LLVM-cov). LLVM-cov does not include line coverage in its JSON output, so we collect its line coverage in the LCOV [45] format. All output is unified (§III-D) and compared (§III-F).

D. Coverage Definitions

Prior work [20]–[23] considered only line coverage; we revisit its definition and also define branch coverage and MC/DC. We consider coverage for each code line and for each source file in a package. Let $t \in \{A, B\}$ be a tool under test, and $l_1, \ldots, l_N$ be the lines in an N-line input source file. For each line $L \in \{l_1, \ldots l_N\}$, t produces three coverage values: line coverage L_t, branch coverage $\mathbf{B}_t$, and MC/DC $\mathbf{M}_t$; each can be `None` if the line is not instrumented for that coverage metric. Tools decide differently what lines to instrument; e.g., for line coverage, commonly uninstrumented are empty lines, comment lines, or lines with just a curly brace.

1) Line coverage: L_t is a nonnegative integer value.

2) Branch coverage: $\mathbf{B}_t$ for a line where t finds O_t outcomes (different control flows) is represented as a list $\mathbf{B}_t = [b_1, \ldots, b_{O_t}]$, where b_j represents how many times the j-th outcome was taken. For example, if a line has P_t conditions, i.e., atomic Boolean expressions [32], branch coverage would be $\mathbf{B}_t = [b_1^T, b_1^F, \ldots, b_{P_t}^T, b_{P_t}^F]$, where b_i^T and b_i^F represent how many times the `true` and `false` outcomes of the i-th condition were taken.

Several challenges arise in practice for differential testing. First, a tool may not indicate which outcome belongs to which condition when a line has multiple conditions, which is the case for Gcov. Second, tools may represent outcomes differently, e.g., Gcov treats `switch` statements as a multiple-outcome branch, while LLVM-cov treats each `case` label as a binary-outcome branch. Also, for compile-time constant conditions, LLVM-cov has a special "outcome" called "constant folded", while Gcov does not output anything. Third, the number of outcomes for the same line can differ across tools, i.e., $O_A \neq O_B$, which is a type of inconsistency in our study.

Figure 2 shows a code snippet with a logical expression split across lines and the conditions that Gcov and LLVM-cov include in their branch coverage reports. LLVM-cov report reflects exactly how the source code is written. However, Gcov not only places `c4` off by one line but also presents all conditions at line 3 in one outcome list $[b_1, ..., b_6]$ without indicating which outcome belongs to which condition. Our manual inspection has determined the mapping between outcomes and conditions; it turns out Gcov *altered* the order of the conditions from the order in the source code. As a result, comparing branch coverage across these lines is challenging: for lines 3 and 4, $O_A \neq O_B$; for line 2, the comparison of different conditions misleads the differential oracle.

```
1 return
2   c1 &&
3   c2 && c3 &&
4   c4;
```

t = LLVM-cov	O_t	t = Gcov	O_t
`c1`	2	`c2`	2
`c2, c3`	4	`c1, c3, c4`	6
`c4`	2	None	0

Fig. 2. An example confusing Gcov branch coverage report: Gcov does *not* present branches following the source-code order.

3) MC/DC: $\mathbf{M}_t$ is represented as a list $\mathbf{M}_t = [m_1, ..., m_{Q_t}]$ where Q_t is the total number of conditions reported by t, and m_j is a Boolean value that represents whether the j-th condition is covered. Q_t for MC/DC suffers from similar challenges as O_t for branch coverage. For example, Figure 3 shows that both tools group all four conditions from the same decision, making MC/DC less sensitive to multi-line logical expressions. Nevertheless, Gcov still exhibits the off-by-one-line misplacement, making none of the three lines comparable. Moreover, MC/DC in LLVM-cov excludes 1-condition decisions (e.g., `if (x == y)` has no MC/DC report), while Gcov keeps them in the coverage report.

```
1 return
2   c1 &&
3   c2 && c3 &&
4   c4;
```

t = LLVM-cov	Q_t	t = Gcov	Q_t
None	0	None	0
None	0	`c1, c2, c3, c4`	4
`c1, c2, c3, c4`	4	None	0

Fig. 3. Example confusing MC/DC reports on different lines.

These differences in coverage presentation for branch coverage and MC/DC pose great challenges for DEBCOVDIFF. However, we do not count them as bugs but as different *conventions* between the tools. It is debatable whether these should be treated as bugs, and a standard should be developed for code coverage tools, potentially with some "undefined behavior", much like the standard for the C programming language [46]. We resort to using filters and heuristics to mitigate the misleading effects of differences and to identify real bugs rather than inconsistencies due to conventions.

Algorithm 1 Detecting coverage inconsistencies for a line

Input: Line coverage L_t and history $\mathcal{L}$, branch coverage $\mathbf{B}_t$ and history $\mathcal{B}$, MC/DC $\mathbf{M}_t$ and history $\mathcal{M}$; $t \in \{A, B\}$
Output: Inconsistencies $\mathcal{I}$ at this line, initially an empty set

1: **if** L_A is not `None` **and** L_B is not `None` **then** ▷ Line coverage
2: **if** $L_A \cdot L_B \neq 0$ **and** $L_A \neq L_B$ **and** IsStable(L_A, L_B, $\mathcal{L}$) **then**
3: $\mathcal{I} \leftarrow \mathcal{I} +$ (line_val, L_A, L_B)
4: **end if**
5: **end if**
6: **if** $\mathbf{B}_A$ is not `None` **and** $\mathbf{B}_B$ is not `None` **then** ▷ Branch coverage
7: **if** len($\mathbf{B}_A$) $\neq$ len($\mathbf{B}_B$) **then**
8: $\mathcal{I} \leftarrow \mathcal{I} +$ (branch_num, $\mathbf{B}_A, \mathbf{B}_B$)
9: **else**
10: $\mathbf{B}'_A, \mathbf{B}'_B, \mathcal{B}' \leftarrow$ sort($\mathbf{B}_A, \mathbf{B}_B, \mathcal{B}$)
11: **if** $\mathbf{B}'_A \neq \mathbf{B}'_B$ **and** IsStable($\mathbf{B}'_A$, $\mathbf{B}'_B$, $\mathcal{B}'$) **then**
12: $\mathcal{I} \leftarrow \mathcal{I} +$ (branch_val, $\mathbf{B}_A, \mathbf{B}_B$)
13: **end if**
14: **end if**
15: **end if**
16: **if** $\mathbf{M}_A$ is not `None` **and** $\mathbf{M}_B$ is not `None` **then** ▷ MC/DC
17: **if** len($\mathbf{M}_A$) $\neq$ len($\mathbf{M}_B$) **then**
18: $\mathcal{I} \leftarrow \mathcal{I} +$ (mcdc_num, $\mathbf{M}_A, \mathbf{M}_B$)
19: **elif** $\mathbf{M}_A \neq \mathbf{M}_B$ **and** IsStable($\mathbf{M}_A$, $\mathbf{M}_B$, $\mathcal{M}$)
20: $\mathcal{I} \leftarrow \mathcal{I} +$ (mcdc_val, $\mathbf{M}_A, \mathbf{M}_B$)
21: **end if**
22: **end if**
23:
24: **procedure** IsStable(x_A,x_B,$\mathcal{X}$)
25: **return** $\mathcal{X}$ has T identical records of x_A, x_B (e.g., $T = 6$)
26: **end procedure**

Transforming the source code (e.g., placing all conditions on one line) may solve some of the problems, e.g., in Figures 2 and 3. However, it is important to (1) make tool users aware of the issue when they attempt to measure their real software with various coding styles, or even better (2) change tools to more reasonably handle such software.

E. Nondeterminism in Coverage Measurement

Prior work [20]–[23] tested coverage tools using small programs that rarely exhibit nondeterminism; Cod [21] and Decov [22] explicitly state that deterministic programs are pre-requisites for their testing. In contrast, we use real code from Debian packages that can exhibit nondeterminism, and running the same code on the same input can give a different coverage report even when measured by the same coverage tool. (In general, Debian packages can produce nondeterministic binaries, despite efforts on reproducible builds [39], [40].) To mitigate the nondeterminism, which could lead to false alarms in differential testing, we run our simple commands $T = 6$ times, and the existing tests (which are longer and tend to be more nondeterministic) $T = 60$ times. We store the coverage results for each line across all T runs in the history $\mathcal{L}$, $\mathcal{B}$, $\mathcal{M}$ for the three metrics. Algorithm 1 reports as inconsistencies only code paths with stable coverage numbers, i.e., all T runs are identical. We require each such code path to be deterministic in each of LLVM-cov and Gcov reports. We found that code paths in LLVM-cov's and Gcov's reports were always equi-deterministic—either both deterministic or both nondeterministic—for our simple commands but not necessarily for the existing tests.

Algorithm 2 Sorting branch coverage and history

1: **procedure** Sort($\mathbf{B}_A, \mathbf{B}_B, \mathcal{B}$)
2: $\texttt{perm}_A \leftarrow \texttt{np.sortarg}(\mathbf{B}_A)$ ▷ Index permutation
3: $\texttt{perm}_B \leftarrow \texttt{np.sortarg}(\mathbf{B}_B)$
4: **for** k in `range`(len($\mathcal{B}$)) **do**
5: $\mathbf{B}_{A,k}, \mathbf{B}_{B,k} \leftarrow \mathcal{B}[k]$
6: $\mathcal{B}'[k] \leftarrow \mathbf{B}_{A,k}[\texttt{perm}_A], \mathbf{B}_{B,k}[\texttt{perm}_B]$
7: **end for**
8: **return** $\mathbf{B}_A[\texttt{perm}_A], \mathbf{B}_B[\texttt{perm}_B], \mathcal{B}'$
9: **end procedure**

F. Differential Testing Oracles

Algorithm 1 shows our procedure for reporting inconsistencies in coverage reports from two tools. The inputs are the coverage reports for each line and the coverage history across T runs. The output identifies five types of inconsistencies, one from prior work on testing line coverage and four new types for branch coverage and MC/DC:

- *line_val*: line coverage counters differ when both are nonzero, called "Type C" in C2V [20]; we leave differences with zero counters (called "Type A" and "Type B" in C2V [20]) as future work.
- *branch_num*, *mcdc_num*: the number of outcomes (O_t in branch coverage) or the number of conditions (Q_t in MC/DC) differs between the two reports.
- *branch_val*, *mcdc_val*: the execution counters (b_j in branch coverage) or coverage of single conditions (m_j in MC/DC) differs between the two reports.

Before reporting any inconsistency, our algorithm ensures that the relevant coverage is stable across all T runs (e.g., line 2 of Algorithm 1) so the difference is less likely caused by execution nondeterminism. Our algorithm checks for each tool (A and B) that the coverage by the tool has the exact same value in all T runs. These checks are *not* needed for the number of outcomes in branch coverage and the number of conditions in MC/DC because these numbers come from the static code instrumentation, not from the dynamic runs.

To tackle different ordering of branch outcomes between the tools, we map outcomes from one tool to the other using the sorting heuristic from Algorithm 2. This approach reduces false positives caused by inconsistent ordering. E.g., for input $\mathbf{B}_A = [0, 1, 42, 10]$ and $\mathbf{B}_B = [11, 42, 0, 1]$, it automatically aligns the difference between outcomes (originally the fourth and first, respectively) after sorting (both become the third). However, sorting does not guarantee that the match is in the right condition order and can lead to false negatives, e.g., $[0, 1]$ and $[1, 0]$ may need to be reported as a difference but do not get reported after sorting.

G. Inspection and Deduplication

We inspect inconsistencies to identify bugs. Determining which tool is buggy is relatively easy based on the code semantics and surrounding coverage. For example, different execution counters may be contradictory within one basic block or violate invariants (e.g., $C_{total} \neq C_{then} + C_{else}$).

Determining which inconsistencies are likely due to the same bug, i.e., *duplicates*, and minimizing code for large codebases are much more challenging. We determine unique bugs based on the language constructs in the input program, e.g., "`push_back` of a char pointer into a `std::string` vector". This criterion matches prior work [20] but may overcount (bugs triggered by different language constructs could be fixed by the same patch) or undercount bugs (similar programs with the same language constructs may require different patches).

```
// LLVM#37125
2 | while (1) { // correct answer should be 1
1 |   last = 1;
1 |   fjmp ();

// LLVM#37081
1 | if ((ret = setjmp(buf)) != 0) {
1 |   printf("True  branch.\n");
1 | } else {
0 |   printf("False branch.\n"); // correct answers
0 |   bar(buf, 2);               // should be 1's
0 | }
```

Listing 3. Likely the same bug reported twice in [20] because of different "types" (numerical relationship between the tool's report and ground truth).

Prior work [20] additionally considers the zero vs. nonzero counter values. However, such difference can lead to duplicates being counted as different bugs. For example, in Listing 3, C2V [20] considers the two reports as different bugs because the type of LLVM#37125 is "wrong frequency", and the type of LLVM#37081 is "missing marking". However, the language construct is the same—"`setjmp`/`longjmp` wrapped in another function". The developers argued that the reports may be duplicate. Our inspection shows that it is likely the same bug: LLVM-cov does not recognize that a function—a wrapper around `longjmp` in this example—may never return to its caller site (`longjmp` can jump to nonlocal destinations) and thus incorrectly assigns the same counter values to the function call and statements after it. Therefore, we do not use differences between zero and nonzero coverage counters as a criterion to determine duplicates. In brief, our criteria are more conservative than C2V [20].

IV. Analysis

A. Tool Crashes

As summarized in Section III-A, for 26 packages, we could not successfully produce coverage reports using both Gcov and LLVM-cov. Our inspection finds 2 crashing bugs in LLVM-cov that come from 6 packages: LLVM#95739 crashes only for MC/DC, while LLVM#95831 crashes for all three metrics. We found no crashing problems in Gcov.

Listings 4 and 5 show our reduced input programs for the two bugs. In Listing 4, LLVM#95739 triggers when a user-defined macro (Line 2) expands to a logical expression that in turn contains another macro (Line 3) defined in a separate system header (Line 1). In Listing 5, LLVM#95831 triggers when an entire header file is included in the middle of a function body. Similar patterns are widely used in real-world code, e.g., 5 packages (`bash`, `diffutils`, `findutils`, `sed`, `tar`) wrap around macros from `<ctype.h>`, `<bits/stat.h>`, etc.

TABLE I
Compared and inconsistent lines, branches, and decisions.

	Lines	Branches	Decisions
Total compared	168,549	44,126	7,527
Median compared	2,207	641	93
Total inconsistent (*_num + *_val)	199	2,098 (1,884 + 214)	30 (29 + 1)
Median inconsistent (*_num, *_val)	1	14 (13, 0)	0 (0, 0)

in a way similar to Listing 4. However, program synthesizers, such as Csmith, are unlikely to generate such patterns, unless specifically tailored. In retrospect, once a bug is found, we can always change a synthesizer to generate such patterns, but that also lowers its probability to generate other code patterns.

```
1 #include <ctype.h>
2 #define IS_WORD_CHAR(ch) \
3   (isalnum (ch) || (ch) == '_')
4 int main(void) { return IS_WORD_CHAR('c'); }
```

Listing 4. Input program for LLVM#95739, reduced from `tar` package.

```
1 void a() {
2 #include <stdint.h>
3 }
```

Listing 5. Input program for LLVM#95831, reduced from `zsh` package.

Both bugs are in LLVM-cov, one manifesting at the Clang frontend and the other during postprocessing (when invoking `llvm-cov`). Both bugs result in assertion failures. Reducing program inputs for these crashes is easier than for coverage inconsistencies, because the invariant ("interestingness" in terms of C-Reduce [47]) for automatic reduction is clear: parsing the tool log should find the assertion failure message.

B. Coverage Report Inconsistencies for Simple Commands

On the 41 Debian packages which produced coverage using both Gcov and LLVM-cov, we perform differential testing. Table I summarizes the results using our simple commands (Section III-C). Figure 4 shows the breakdown across the packages. We make some high-level observations:

- The total number of compared sites decreases from line coverage to branch coverage to MC/DC, as expected from their definitions. Notably, no decision is compared in the `sl` package, because (1) the project is small, with only one source file and one header file; (2) LLVM-cov does not consider 1-condition decisions for MC/DC (Section III-D); and (3) one instance of `if ((c1 ? CONST1 : c2) == CONST2)` is considered a 2-condition decision by Gcov but not considered a decision by LLVM-cov.
- The two types of branch coverage and MC/DC inconsistencies (Section III-F) have big differences. The reason is that the instrumentation produces the number of outcomes and conditions statically, *regardless* of the actual program execution. However, the coverage of individual outcomes depends on the code paths exercised during execution.

As shown in Figure 4, 39 of 41 packages exhibit an inconsistency in at least one of line coverage, branch coverage, or MC/DC. The two exceptions—`shadow` and `dbus`—have the

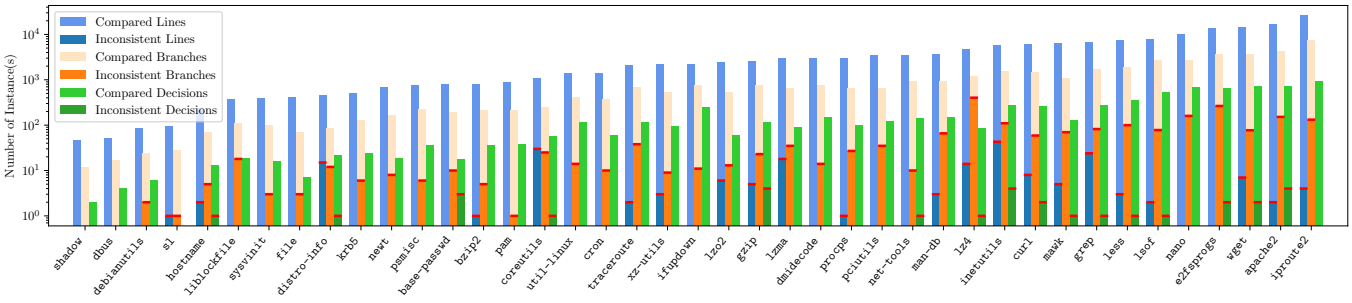

Fig. 4. Compared and inconsistent lines, branches, and decisions per Debian package (sorted by compared lines).

lowest number of compared lines. In terms of particular coverage metrics, 22 packages exhibit line coverage inconsistencies, 39 (38 for branch_num and 18 for branch_val) exhibit branch coverage inconsistencies, and 16 (15 for mcdc_num and 1 for mcdc_val) exhibit MC/DC inconsistencies.

We inspected 1,240 inconsistencies (all for MC/DC, and a random subset of line and branch coverage). Our inspection found 256 inconsistencies caused by 22 new bugs and two known bugs (§IV-B2), 954 caused by at least 15 convention differences (§IV-D), and 30 caused by using two different compilers, e.g., compiler-specific strings inserted in the binary executable programs leading to benign differences.

1) New Bugs: Table II lists the new bugs that we reported. The top and bottom parts list the 22 and 10 new bugs found for our simple commands (§III-C) and existing tests (elaborated in §IV-G), respectively. We have two key observations:

- Despite substantial research on finding bugs in line coverage [20]–[23], we still find new bugs for line coverage. These new bugs demonstrate that using Debian packages as inputs increases the richness of C/C++ language features over small programs, e.g., automatically generated by Csmith [25]. Additionally, we find 8 bugs that affect branch coverage and MC/DC; note that some bugs affect multiple metrics, e.g., LLVM#101241 and GCC#120319. Interestingly, we find no bug for the mcdc_val inconsistency.
- The fifth column of Table II shows how many of the 41 Debian packages manifest a specific bug, listing in the parentheses the package(s) with the most occurrences. We observe both high numbers of packages (up to 9), which suggest the wide impact of some bugs, and low numbers (1 package), with bugs originating only from package-unique code patterns. In particular, `lzma` and `mawk` expose most bugs. One probable reason for `lzma` is its mixture of C and C++ code, which leads to a greater diversity of language features, e.g., object-oriented programming, C++ standard libraries, or function calling between C and C++ code.

2) Known Bugs: DEBCOVDIFF also finds bugs that are not new but duplicates of bugs already reported but not yet fixed. For Gcov, we found multiple occurrences of wrong line coverage for multiline logical expressions, similar as reported in GCC#97923 [48]; this report has a pending patch. For LLVM-cov, we found multiple occurrences of a bug that is acknowledged in the LLVM-cov documentation since 2016 [49] and hard to resolve due to unpredictable control-flow changes (we call it LLVM#UCF and believe it to be the same cause for Listing 3). We do *not* count these known bugs among our new bugs. To our knowledge, all 34 bugs that we reported to Gcov and LLVM-cov developers are unique and not duplicates of previous reports.

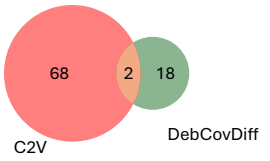

Fig. 5. Line coverage bugs found by C2V [20] and DEBCOVDIFF.

We also compare the bugs we found with the bugs from C2V [20]. Figure 5 shows the Venn diagram of the line coverage bugs found by C2V and DEBCOVDIFF for simple commands. (DEBCOVDIFF also finds 10 more line coverage bugs for existing tests from Debian packages and 4 more non-line bugs.) For the 18 line coverage bugs that DEBCOVDIFF finds, we check their existence in the tool versions targeted by C2V, i.e., GCC 8.0 and LLVM 7.0. 12 of 18 existed in the old releases but were *not* found by C2V (to our ability to check).

Moreover, programs generated by Csmith would miss at least 14 out of 18 bugs even now. To count, we summarize the language constructs that trigger bugs and check whether Csmith generates these language constructs. For each bug, we implement a checker that over-approximates the triggering language constructs. We used Csmith to generate 300,000 programs (100,000 each for the default Csmith configuration, `--lang-cpp`, and `--inline-function`). Only 4 of the 18 combinations of constructs are in at least one of the 300,000 Csmith programs. The language constructs for the remaining 14 bugs are not in any of the 300,000 Csmith programs.

C. Case Study of New Bugs

1) LLVM coverage mapping: LLVM-cov maintains a dedicated coverage mapping [50] that helps it map coverage counters to source code fairly precisely, but it is also hard to implement correctly, especially with macros and headers. In Listing 6, for example, the statement at line 5 should have been measured with a dedicated "code region" [50]. However, LLVM-cov fails to do so because of a macro from system header. The line coverage for line 5 falls back to a less exactly matched block, the entire function, thus the value 1. LLVM#131505 shown in Listings 1 and 2 is similar but triggered by a different set of language constructs.

2) Coding style: In real-world projects, coding style may vary. Long statements are often split into multiple lines ("mul-

TABLE II
BUGS FOUND BY DEBCOVDIFF; L: LINE_VAL, BN: BRANCH_NUM, BV: BRANCH_VAL, MN: MCDC_NUM, MV: MCDC_VAL; N/A: NOT RUN.

#	Bug ID	Tool	Incons. Type	Affected Package(s)	Occur.	Triggering Condition(s)
1	120321	Gcov	L	9 (`lzo2`)	6 + 11	constant, loop, no code, variable scope
2	120478	Gcov	L	2 (`lzo2`)	3 + 2	inline function, multistmt
3	117412	Gcov	L	3 (`lz4`)	3 + 1	dereference, function, multiline, type conversion
4	120482	Gcov	L	1 (`lz4`)	3 + n/a	`const` keyword, inline function
5	120490	Gcov	L	1 (`lz4`)	3 + n/a	inline function, `switch-case`
6	117415	Gcov	L	1 (`apache2`)	1 + n/a	dereference, function, `if`, loop, multiline
7	120348	Gcov	L	1 (`lzma`)	1 + n/a	C++, `extern "C"`, function, loop, `return`, variable scope
8	120484	Gcov	L	1 (`curl`)	1 + n/a	`if`, loop, variable scope
9	120489	Gcov	L	1 (`lz4`)	1 + n/a	`continue`, dereference, loop, variable scope
10	120491	Gcov	L	1 (`lzma`)	1 + n/a	C++, constructor, multiline
11	120492	Gcov	L	1 (`lzma`)	1 + n/a	C++ standard library, type conversion
12	120486	Gcov	L/BN	5 (`distro-info`)	6 + 19	multiline, ternary
13	120319	Gcov	L/BN	1 (`lzma`)	3 + n/a	C++, function, multiline, variable scope
14	120332	Gcov	L/BN/BV	5 (`lzo2`)	4 + 16	`if`, loop, no code, `switch-case`
15	120485	Gcov	BV	1 (`mawk`)	23 + 0	preprocessor directive, header
16	140427	LLVM-cov	L	3 (`mawk`)	2 + 2	`goto`, `if`, loop, macro, `return`
17	114622	LLVM-cov	L	1 (`sl`)	1 + n/a	`break`, header, macro
18	116884	LLVM-cov	L	1 (`grep`)	1 + n/a	`break`, constant, loop
19	105341	LLVM-cov	L/BV	1 (`hostname`)	1 + n/a	concurrency
20	116902	LLVM-cov	BN	1 (`lzma`)	3 + n/a	C++ standard library, function
21	101241	LLVM-cov	BN/MN	1 (`net-tools`)	2 + n/a	comma, constant, tautological compare, type conversion
22	131505	LLVM-cov	BN/BV/MN	1 (`hostname`)	1 + n/a	dereference, header, macro, type conversion
23	121914	Gcov	L	1 (`lzo2`)	0 + 4	inline function
24	121897	Gcov	L	2 (`ifupdown`)	0 + 3	function, variable scope
25	121901	Gcov	L	1 (`bzip2`)	0 + 3	dereference, loop, `malloc`, multiline
26	121896	Gcov	L	1 (`mawk`)	0 + 2	function, `if`, pointer
27	121902	Gcov	L	1 (`ifupdown`)	0 + 1	`continue`
28	158003	LLVM-cov	L	1 (`mawk`)	0 + 4	macro, `switch-case`
29	158080	LLVM-cov	L	2 (`distro-info`, `mawk`)	0 + 4	`break`, `if`, `switch-case`
30	157959	LLVM-cov	L	2 (`lzo2`, `procps`)	0 + 2	`goto`, `if`
31	157946	LLVM-cov	L	1 (`distro-info`)	0 + 1	preprocessor directive, header, macro
32	157981	LLVM-cov	L	1 (`psmisc`)	0 + 1	loop, multiline

```
// test.h
   | #define FOO (void) 0
// main.c
   | #include <test.h>
 1 | int main(int argc, char **argv) {
 1 |     if (argc == 1)
 1 |         return 1;
 1 |     FOO;
 0 |     return 0;
 1 | }
```

Listing 6. Input program for LLVM#114622 reduced from `sl` package. The invariant of $C_{total} = C_{then} + C_{else}$ is violated for the `if` statement; lines 5 and 6 forming the same basic block also wrongly have different line coverage.

tiline" in Table II). In rarer cases, multiple statements are put in one line ("multistmt" in Table II). Gcov's instrumentation makes it hard to precisely associate a basic block to source line(s). Figures 2 and 3 (Section III-D) show two confusing examples, which we do not even count as bugs because we managed to interpret them. However, in worse cases, such as line 6 in Listing 7, Gcov results are definite bugs, as line coverage for the same expression separated at line 6 and 7 differs when there is no change of control flow.

3) C++: Multiple aspects of C++ can trigger coverage bugs, such as object-oriented code, routines from the C++ standard library, or mixture of C and C++ code. Listing 8 shows a case where changing the extension of the input file and switching from `gcc` to `g++` introduces bogus branch outcomes in Gcov. With the C version, the `if` predicate is reported to have 4 outcomes—`true` and `false` for `g` and `foo()` respectively. With the C++ version, 2 extra outcomes show

```
   | int g;
 1 | int *foo() { return &g; }
 1 | int main(void) {
 1 |     int *a = &g;
   |     &a;
 2 |     *foo() =
 1 |         *a;
   | }
```

Listing 7. Input program for GCC#117415 reduced from `apache2` package. The whole snippet has a linear control flow and is executed exactly once, but line 6 is wrongly reported to be executed twice.

```
// Compiled with "g++ --coverage"
 0 | int foo(void) { return 1; }
   | int g;
 1 | int main() {
 1 |     if (g && foo()) { return 1; }
-------------------------------------------------------
| Branch (6 outcomes): [0,1,0,0,0,1]
-------------------------------------------------------
 1 |     return 0;
   | }
```

Listing 8. Input program for GCC#120319 reduced from `lzma` package. The `if` predicate has two atomic conditions `g` and `foo()` but GCC wrongly reports 6 outcomes. With the exact same program but renamed with an extension name `*.c` and compiled with `gcc`, the report correctly presents 4 outcomes.

up. Listing 9 shows another C++-related bug where the use of `std::string` in function parameters introduces bogus branch outcomes in LLVM-cov.

4) "No code": Gcov struggles to associate coverage information to source code elements that do not result in object code, e.g., `(void) 0` as a result of macro fallback expansion (GCC#120321), `goto` labels, `case` labels, or deliberate empty

TABLE III
CONVENTION DIFFERENCES BETWEEN GCOV AND LLVM-COV REVEALED BY DEBCOVDIFF.

#	Short Description	Incons. Type	Affected Package(s)	Occur.	Triggering Condition(s)
1	Complex control flow in macros	L	4 (`inetutils`, `lzma`)	9	`goto`, `if`, loop, macro
2	`do-while` statements	L	5 (`hostname`)	6	`if`, loop
3	`else if`	L	2 (`gzip`)	3	`if`
4	Multiline statements	L/BN/BV	19 (`iproute2`)	283	multiline, ternary
5	`switch-case` statements	L/BN/BV	12 (`mawk`)	142	no code, `switch-case`
6	Macros in decisions	BN	11 (`apache2`)	150	macro
7	Decisions in macros	BN	12 (`lz4`)	141	macro
8	Branches in inline functions	BN	1 (`lz4`)	118	inline function
9	Branches for assertions	BN	6 (`lz4`)	52	assertion
10	System header files	BN	7 (`iproute2`)	44	header
11	Constant condition	BN/MN	11 (`iproute2`)	58	constant
12	MC/DC ternary	MN	8 (`inetutils`)	12	ternary
13	MC/DC empty "then" clause	MN	3 (`gzip`)	6	`if`, no code
14	MC/DC inline function	MN	1 (`lz4`)	1	`const` keyword, inline function
15	Masking vs. unique-cause MC/DC	MV	1 (`distro-info`)	1	mixed conjunctive and disjunctive

```
   | #include <string>
   | int g = 1;
   | std::string s = "hello";
 1 | int foo(std::string s) { return 0; }
 1 | int main() {
 1 |     if (g &&
-----------------------------------------------------
| Branch (4 outcomes): [0,1,1,0]
-----------------------------------------------------
 1 |         foo(s))
-----------------------------------------------------
| Branch (2 outcomes): [0,1]
-----------------------------------------------------
 0 |         return 1;
 1 | }
```

Listing 9. Input program for LLVM#116902 reduced from `lzma` package. A single atomic condition `g` is wrongly reported to have 4 outcomes.

```
 1 | int main(void) {
 1 |     int i = 0;
   |     while (1) {
 6 |         0;
 7 |         if (++i >= 7)
 1 |             break;
   |     }
   | }
```

Listing 10. Input program for GCC#120321 reduced from `lzo2` package. The nop statement "`0;`" that is typically reported as "no code" by Gcov is reported with an actual value 6 that contradicts other counters that follow it.

statements (GCC#120332). In Listing 10, Gcov reports for line 4 wrong coverage that seems to refer to the implicit "else" of the following `if` statement, conceptually the empty part between lines 6 and 7. Similarly, Gcov can give `case` labels wrong coverage values based on a different line or even scope.

5) Constants: Because LLVM-cov instruments the source code before most compiler passes, it cannot recognize many constant expressions. We label most inconsistencies stemming from constants as convention differences (§IV-D), but one is a definite bug, LLVM#101241. For the example in Listing 11, the Clang frontend already warns that one condition is a constant: "result of comparison [...] is always `true`". However, LLVM-cov fails to mark the condition as constant, reporting a missed outcome that can never be covered for branch coverage or MC/DC. The reason is that LLVM-cov implements its own constant analysis rather than reusing the one from Clang.

```
   | unsigned char g, h;
 1 | int main(void) {
 1 |     if (g < 256 && h)
-----------------------------------------------------
| Branch (4 outcomes): [1,0,0,1]
-----------------------------------------------------
| MC/DC (2 conditions): [F,F]
-----------------------------------------------------
 0 |         return 1;
 1 | }
```

Listing 11. Input program for LLVM#101241 reduced from `net-tools`. Expression `g < 256` is always `true` for `unsigned char g`. LLVM-cov marks most such conditions as "constant folded" but not this one.

```
switch (c) {
case 0:
case 1:
case 2:
  <statement>;
```

$L_{\text{LLVM-cov}}$	L_{Gcov}
C_0	$C_0 + C_1 + C_2$
$C_0 + C_1$	None
$C_0 + C_1 + C_2$	None
$C_0 + C_1 + C_2$	$C_0 + C_1 + C_2$

Fig. 6. A convention difference for line coverage in `switch-case`.

D. Convention Differences

Some inconsistencies are not bugs in any tool but arguably different, potentially correct choices for measuring coverage. We label 954 inspected inconsistencies as such *convention differences* between the tools, with 15 categories listed in Table III, where one inconsistency may be in several categories.

Of 15 convention differences, 5, 8, and 5 affect line coverage, branch coverage, and MC/DC, respectively. One convention difference can affect multiple metrics. E.g., Section III-D briefly mentioned how tools treat `switch-case` statements differently for branch coverage; more subtlety exists. (1) If a `switch` has no explicit `default` case, LLVM-cov puts the binary outcome for the implicit `default` case at the line of `switch`, which often contradicts with Gcov that puts there the multiple-outcome branch for the whole statement. (2) If multiple `case` labels immediately fall through to the next, Gcov presents the summed number of matches as line coverage for the first label, but "no code" for the rest; LLVM-cov, in contrast, differentiates all labels and presents line coverage in an accumulated way, as shown in Figure 6.

Another noteworthy difference is #15 in Table III: Gcov implements masking MC/DC, while LLVM-cov implements

```
1  2 | int foo(int a, int b, int c) {
2  2 |   return (a && b) || c;
   ----------------------------------------------------
   | MC/DC by LLVM-cov (3 conditions): [F,F,F]
   ----------------------------------------------------
   | MC/DC by Gcov     (3 conditions): [F,F,T]
   ----------------------------------------------------
3  2 | }
4  1 | int main(void) {
5  1 |   foo(0, 0, 0);
6  1 |   foo(1, 0, 1);
7  1 | }
```

Listing 12. Difference between unique-cause MC/DC (reported by LLVM-cov) and masking MC/DC (reported by Gcov).

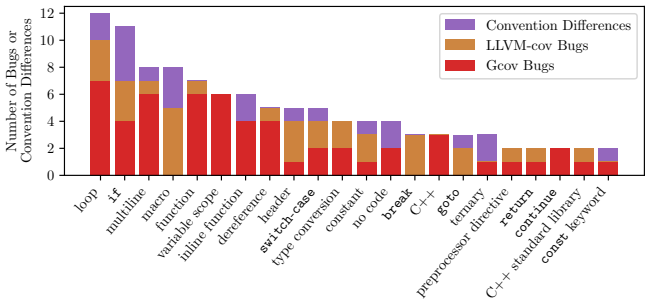

Fig. 7. Programming language constructs that trigger inconsistencies.

unique-cause MC/DC [30], [31]. Listing 12 shows an example where Gcov reports a condition as covered while LLVM-cov reports the opposite; it manifests only for specific test vectors to logical expressions that combine `&&` and `||` operators [51].

E. Causes of Inconsistencies

Putting bugs and convention differences together, we seek to understand the language constructs that trigger the inconsistencies. Figure 7 reveals Gcov (IR-based instrumentation) and LLVM-cov (frontend instrumentation) have fundamental differences, but neither is superior to the other, e.g.:

- "multiline" (6 of 7 bugs are in Gcov) and "no code" (both bugs are in Gcov) show Gcov's difficulty in mapping basic blocks to accurate source lines;
- "macro" (all 5 bugs are in LLVM-cov) and "header" (3 of 4 bugs are in LLVM-cov) show LLVM-cov's difficulty in handling preprocessing: the coverage component may need to perform analysis by itself (instead of piggybacking on the later, widely-used stages of the compilation pipeline). Certain analyses are even impossible, such as LLVM#UCF.

Control-flow constructs, such as "loop", "`if`", or "ternary", recur, suggesting they are especially hard to make correct.

Figure 8 shows the distribution of bugs and conventions across packages. About half of them appear in multiple packages, with occurrence outliers such as convention #8 exhibiting strong clustering pattern. This pattern suggests that package-specific idioms may help stress test coverage tools.

F. Other Build Failures

Besides tool crashes (§IV-A), other factors prevent differential testing for 20 packages. Of these 20, Gcov did not work for 18 and LLVM-cov for 13 (neither worked for 11). We inspected the failures and summarize their root causes below. Marcozzi et al. [28] report similar problems in their study

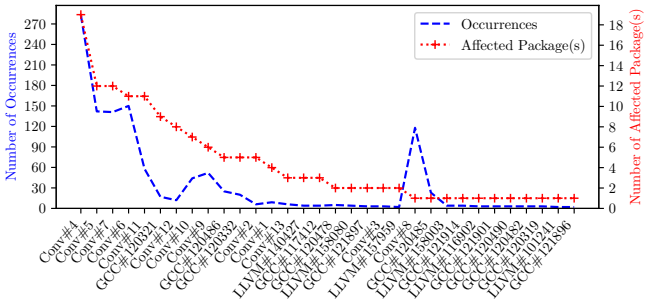

Fig. 8. Impacts of bugs and convention differences (only those with more than 1 occurrences are shown).

where they also used Debian packages as test inputs (to test compilers). Our success rate of generating coverage reports for 41 out of 67 packages (using both Gcov and LLVM-cov) is higher than their success rate [28] of generating coverage reports for 39 out of 309 packages (using only Gcov), presumably because the Debian build system and the reliability of coverage tools improved. These issues do not show functional bugs in the coverage tools but do show potential usability issues as sophisticated configuration is needed to apply these tools to real-world projects. As a baseline, we confirm that all 26 packages successfully build using the default GCC version from `sbuild` without our flags.

The causes of build failures are in three categories. (1) 9 packages cannot build when we replace the default GCC version with our GCC under test (newer than the default) or LLVM/Clang and add coverage-related compiler flags, e.g., one package (`logrotate`) enforces `-Werror`, and the Clang warning for LLVM-cov MC/DC limit [35] becomes a compilation error that breaks the build. (2) 4 packages built but either generated no `*.gcda` files or had empty `*.profraw` files: 2 packages use custom `exit` functions; 1 package (`iputils`) has special capability issues; and 1 package (`glibc`) hardcodes the compiler version and flags (thus not using ours). (3) 7 packages had postprocessing failures for Gcov, reporting either `Cannot open source file` or `stamp mismatch with notes file`.

G. Impact of Existing Tests from Debian Packages

The aforementioned results are obtained from running simple commands as tests (see §III-C). To evaluate how more thorough tests affect our results, we also experiment with the test suites shipped with the Debian packages, namely `dh_auto_test` (which is run during package builds). Of the 41 packages we evaluate, only 15 support the `dh_auto_test` command. Among these 15, we could run differential testing for nine, Gcov has postprocessing failures (the third case in §IV-F) in five, and LLVM-cov reports `mismatched data` with corrupted coverage reports in one (`util-linux`).

Table IV compares the numbers of coverage inconsistencies that DEBCOVDIFF reports for the simple commands (SC) and existing tests (ET) from `dh_auto_test` for the nine evaluated packages. Overall, 238 of inconsistencies are found for both SC and ET. Only 20 inconsistencies (2, 17, and 1 for line,

TABLE IV
INCONSISTENCIES REPORTED FOR SIMPLE COMMANDS (SC) AND EXISTING TESTS (ET) IN 9 DEBIAN PACKAGES.

Inconsistencies	Lines	Branches	Decisions
Only for SC	2	17	1
Common for SC and ET	28	208	2
Only for ET	409	134	12

branch, and MC/DC, respectively) are found for SC but not for ET. Among them, 9 are due to ET not covering the source file covered by SC. Eight are due to LLVM#UCF (§IV-B2)—a function that does not return for SC *does* return for ET. Two cases involve merging coverage when the same header file is included multiple times. As a result, (1) convention #4 in Table III no longer appears when the same function has different bodies due to different `#ifdefs`; or (2) one line is reported to have multiple MC/DC decisions and no longer compared by our algorithm (§III). In one case, both tests reach the same logical expression but take different branches; for ET, the expression takes no short-circuit path, so a convention (#4) gets masked, and the two tools agree by accident. The 20 inconsistencies missed by ET find one bug, but ET misses no bug as its other inconsistencies find the same bug.

Most inconsistencies found for ET, 555 of 793 (70%), are not found for SC. Of these 555, at least 227 are due to convention differences and 300 due to bugs. We identify 18 bugs, of which 8 were also found for SC (6 in Table II and 2 in Section IV-B2), and 10 are new. Their occurrences in ET-only inconsistencies are shown after '+' in Table II ("n/a" means the bug did not appear in any of 9 packages evaluated with ET). We analyze why the 10 bugs found only for ET are missed for SC. Unsurprisingly, ET covers more code than SC. The bugs stem from 25 code locations, all of which have zero coverage for at least one tool (23 for both tools) in coverage reports for SC, and thus are not compared by our algorithm.

V. RELATED WORK

Testing C/C++ Toolchains. Prior work like Csmith [25] has extensively tested C/C++ compiler toolchains. Notably, Equivalence Modulo Inputs (EMI) [8] revealed lots of bugs; Orion generates test variants by randomly deleting unexecuted statements in seed programs to expose miscompilations. Athena [52] improves upon Orion by introducing guided mutations. It inserts new statements into unexecuted regions and explores variants that differ significantly in control- and data-flow from the original, thereby stressing compiler optimizations more thoroughly. Hermes [53] extends the EMI approach by enabling mutations in "live" (executed) code, not just dead code, which substantially broadens the mutation space. Proteus [54] targets a different dimension of the compiler pipeline: link-time optimization (LTO). It splits single translation units into multiple compilation units and tests the optimizer by assigning randomized optimization flags per unit.

A recent extension to EMI is Creal [26], which introduces a novel angle: enriching seed programs by injecting real functions extracted from large codebases. Prior work by Marcozzi et al. [28] uses real code, i.e., Debian packages, to evaluate compiler bugs. Similarly, we use real code to test coverage tools, complementary to the existing approaches that use synthetic test inputs. Thorough testing should ideally use both real code and synthetic tests.

Testing Coverage Tools. Prior work has also explored techniques to reveal bugs in coverage tools. C2V [20] leverages randomized differential testing to compare coverage results between Gcov and LLVM-cov. Using Csmith-generated programs and selected programs from the tool's test suites, C2V identifies discrepancies in line coverage. Cod [21] improves upon C2V by applying metamorphic testing to a single tool. This approach eliminates the need for multiple coverage tools and achieves zero false positive. Decov [22] further broadens validation capabilities by introducing heterogeneous testing, cross-checking coverage results against debugger-reported hit counts using both breakpoints and single-stepping. Complementing these approaches, DOG [23] formulates the validation as a constraint-solving problem, extracts control-flow and control-dependence graphs from programs, and defines control-flow constraints that capture expected relationships among coverage statistics. Violations of these constraints are encoded as SMT problems [55], enabling bug detection without a second tool or program variants.

Our study uses differential testing as C2V [20]. However, no prior work captured all complex patterns from real software, due to the limitations of small test inputs. Moreover, no prior work considered coverage metrics beyond line coverage.

VI. CONCLUSION

Reliability of code coverage tools is important because coverage is used widely in software engineering, including for safety-critical software. Prior work found many bugs in Gcov and LLVM-cov, but (1) using only small, often synthetic, input programs; and (2) only for line coverage. We present our DEBCOVDIFF framework for testing Gcov and LLVM-cov on real code from Debian packages. We identify 34 new bugs, including 2 crashing bugs (in LLVM-cov) and 32 deeper bugs (in LLVM-cov and Gcov) that produce wrong coverage reports for line coverage, branch coverage, or MC/DC.

Our aspiring goal is to qualify open-source coverage measurement tools, such as Gcov or LLVM-cov, per DO-330 [56] to be usable for certifying software to DO-178C [12] as required by the US Federal Aviation Administration, European Union Aviation Safety Agency, and Transport Canada for approving all commercial aerospace software. DO-178C requires measurement of branch coverage and MC/DC.

ACKNOWLEDGMENTS

We thank Lanea Rohan, Leslie He, Manvik Nanda, and Tingxu Ren for their help in the project, and Yibiao Yang for answering our technical questions about [20]. The work was supported in part by Boeing 2023-BRT-PA-064 and NSF grants CNS-1956007, CCF-1956374, and CNS-2145295.

References

[1] J. C. Miller and C. J. Maloney, "Systematic mistake analysis of digital computer programs," *Commun. ACM*, 1963.

[2] Y. T. Chen, R. Gopinath, A. Tadakamalla, M. D. Ernst, R. Holmes, G. Fraser, P. Ammann, and R. Just, "Revisiting the relationship between fault detection, test adequacy criteria, and test set size," in *ASE*, 2020.

[3] H. Zhu, P. A. V. Hall, and J. H. R. May, "Software unit test coverage and adequacy," *ACM Comput. Surv.*, 1997.

[4] P. Ammann, J. Offutt, and H. Huang, "Coverage criteria for logical expressions," in *ISSRE*, 2003.

[5] J. J. Chilenski and S. P. Miller, "Applicability of modified condition/decision coverage to software testing," *Software Engineering Journal*, 1994.

[6] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE TSE*, 2016.

[7] Y. Chen, T. Su, C. Sun, Z. Su, and J. Zhao, "Coverage-directed differential testing of JVM implementations," in *PLDI*, 2016.

[8] V. Le, M. Afshari, and Z. Su, "Compiler validation via equivalence modulo inputs," in *PLDI*, 2014.

[9] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, "DeFlaker: Automatically detecting flaky tests," in *ICSE*, 2018.

[10] M. Ivanković, G. Petrović, R. Just, and G. Fraser, "Code coverage at Google," in *ESEC/FSE*, 2019.

[11] M. Ivanković, G. Petrović, Y. Kulizhskaya, M. Lewko, L. Kalinovčić, R. Just, and G. Fraser, "Productive coverage: Improving the actionability of code coverage," in *ICSE-SEIP*, 2024.

[12] RTCA/DO-178C, "Software considerations in airborne systems and equipment certification," Dec. 2011.

[13] ISO 26262:2018, "Road vehicles — functional safety," Dec. 2018.

[14] `gcov`, https://gcc.gnu.org/onlinedocs/gcc/Gcov.html, accessed on Oct. 1, 2025.

[15] `llvm-cov`, https://llvm.org/docs/CommandGuide/llvm-cov.html, accessed on Oct. 1, 2025.

[16] VectorCAST, https://www.vector.com/us/en/products/products-a-z/software/vectorcast/, accessed on Oct. 1, 2025.

[17] Parasoft C/C++test, https://www.parasoft.com/products/parasoft-c-ctest/, accessed on Oct. 1, 2025.

[18] JaCoCo, https://www.jacoco.org/jacoco/, accessed on Oct. 1, 2025.

[19] Coverage.py, https://coverage.readthedocs.io/, accessed on Oct. 1, 2025.

[20] 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 *ICSE*, 2019.

[21] Y. Yang, Y. Jiang, Z. Zuo, Y. Wang, H. Sun, H. Lu, Y. Zhou, and B. Xu, "Automatic self-validation for code coverage profilers," in *ASE*, 2019.

[22] Y. Yang, M. Sun, Y. Wang, Q. Li, M. Wen, and Y. Zhou, "Heterogeneous testing for coverage profilers empowered with debugging support," in *ESEC/FSE*, 2023.

[23] Y. Wang, P. Zhang, M. Sun, Z. Lu, Y. Yang, Y. Tang, J. Qian, Z. Li, and Y. Zhou, "Uncovering bugs in code coverage profilers via control flow constraint solving," *IEEE TSE*, 2023.

[24] W. M. McKeeman, "Differential testing for software," *Digital Technical Journal*, 1998.

[25] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in C compilers," in *PLDI*, 2011.

[26] S. Li, T. Theodoridis, and Z. Su, "Boosting compiler testing by injecting real-world code," in *PLDI*, 2024.

[27] X. Wang, N. Zeldovich, M. F. Kaashoek, and A. Solar-Lezama, "Towards optimization-safe systems: Analyzing the impact of undefined behavior," in *SOSP*, 2013.

[28] M. Marcozzi, Q. Tang, A. F. Donaldson, and C. Cadar, "Compiler fuzzing: How much does it matter?" in *OOPSLA*, 2019.

[29] A. Phipps, "Branch Coverage: Squeezing more out of LLVM Source-based Code Coverage," in *2020 LLVM Developers' Meeting*.

[30] ——, "MC/DC: Enabling easy-to-use safety-critical code coverage analysis with LLVM," in *2022 LLVM Developers' Meeting*.

[31] J. Kvalsvik, "Modified Condition/Decision Coverage in the GNU Compiler Collection," *arXiv:2501.02133*, 2025.

[32] P. Ammann and J. Offutt, *Introduction to Software Testing*, 1st ed., 2008.

[33] AFL, https://lcamtuf.coredump.cx/afl/, accessed on Oct. 1, 2025.

[34] libFuzzer, https://llvm.org/docs/LibFuzzer.html, accessed on Oct. 1, 2025.

[35] "Source-based Code Coverage," https://clang.llvm.org/docs/SourceBasedCodeCoverage.html, accessed on Oct. 1, 2025.

[36] "Clang compiler user's manual," https://clang.llvm.org/docs/UsersManual.html, accessed on Oct. 1, 2025.

[37] "`sbuild` - build Debian packages from source," https://manpages.debian.org/bookworm/sbuild/sbuild.1.en.html, accessed on Oct. 1, 2025.

[38] Debian Wiki, "Reproducible builds," https://wiki.debian.org/ReproducibleBuilds/About, accessed on Oct. 1, 2025.

[39] Z. Ren, H. Jiang, J. Xuan, and Z. Yang, "Automated localization for unreproducible builds," in *ICSE*, 2018.

[40] Z. Ren, S. Sun, J. Xuan, X. Li, Z. Zhou, and H. Jiang, "Automated patching for unreproducible builds," in *ICSE*, 2022.

[41] Debian Wiki, "SourcesList," https://wiki.debian.org/SourcesList, accessed on Oct. 1, 2025.

[42] Debian Policy Manual, "The Debian Archive," https://www.debian.org/doc/debian-policy/ch-archive.html, accessed on Oct. 1, 2025.

[43] S. Ledru, "Build of the Debian archive with Clang," https://clang.debian.net/, accessed on Oct. 1, 2025.

[44] "`dh_auto_test` - automatically runs a package's test suites," https://manpages.debian.org/bookworm/debhelper/dh_auto_test.1.en.html, accessed on Oct. 1, 2025.

[45] LCOV, https://github.com/linux-test-project/lcov, accessed on Oct. 1, 2025.

[46] ISO/IEC 9899:2024, "Information technology — programming languages — C," Oct. 2024.

[47] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang, "Test-case reduction for C compiler bugs," in *PLDI*, 2012.

[48] GCC Bugzilla, "[GCOV] Wrong code coverage for multiple expressions with Logical OR Operator at multiple lines," https://gcc.gnu.org/bugzilla/show_bug.cgi?id=97923, accessed on Oct. 1, 2025.

[49] LLVM Mailing Lists, "r271544 - [docs] Add a limitations section to SourceBasedCodeCoverage.rst," https://lists.llvm.org/pipermail/cfe-commits/Week-of-Mon-20160530/161069.html, accessed on Oct. 1, 2025.

[50] "LLVM Code Coverage Mapping Format," https://llvm.org/docs/CoverageMappingFormat.html, accessed on Oct. 1, 2025.

[51] J. J. Chilenski, "An investigation of three forms of the modified condition decision coverage (MCDC) criterion," Federal Aviation Administration, Tech. Rep., 2001.

[52] V. Le, C. Sun, and Z. Su, "Finding deep compiler bugs via guided stochastic program mutation," in *OOPSLA*, 2015.

[53] C. Sun, V. Le, and Z. Su, "Finding compiler bugs via live code mutation," in *OOPSLA*, 2016.

[54] V. Le, C. Sun, and Z. Su, "Randomized stress-testing of link-time optimizers," in *ISSTA*, 2015.

[55] L. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *TACAS*, 2008.

[56] RTCA/DO-330, "Software tool qualification considerations," Dec. 2011.