Proc. ASE 2019, © IEEE

TestCov: Robust Test-Suite Execution and Coverage Measurement

Dirk Beyer
LMU Munich, Germany

Thomas Lemberger
LMU Munich, Germany

***Abstract*—We present TestCov, a tool for robust test-suite execution and test-coverage measurement on C programs. TestCov executes program tests in isolated containers to ensure system integrity and reliable resource control. The tool provides coverage statistics per test and for the whole test suite. TestCov uses the simple, XML-based exchange format for test-suite specifications that was established as standard by Test-Comp. TestCov has been successfully used in Test-Comp '19 to execute almost 9 million tests on 1 720 different programs. The source code of TestCov is released under the open-source license Apache 2.0 and available at https://gitlab.com/sosy-lab/software/test-suite-validator. A full artifact, including a demonstration video, is available at https://doi.org/10.5281/zenodo.3418726.**

***Index Terms*—Test Execution, Coverage, Test-Suite Reduction**

I. Introduction

Modern test-case generators are able to generate system tests that reveal bugs in programs like never before, but executing these tests may lead to system failures, modifications, information leakage, or resource exhaustion. Because of this, program tests are often executed in virtual machines or containers (e.g., Docker). TestCov provides a lightweight solution to this: it uses an overlay file system and Linux control groups to protect the file system from modifications and to prevent unexpected resource usage during test execution, based on the existing benchmarking tool BenchExec [4]. Compared to other containerization technology, BenchExec does not require the installation of any additional software or superuser privileges during usage because it solely relies on features built into the Linux kernel. TestCov provides coverage statistics for line, branch, and condition coverage per test and for the whole test suite, and creates plots to visualize the measured data.

TestCov has been used in the First International Competition on Software Testing (Test-Comp '19) [1] to validate the test suites created by all 9 participants. TestCov uses the simple, XML-based exchange format for test-suite specification that was established as a standard by Test-Comp. All 9 participants support the exchange format. In the past, test-case generators used proprietary formats to output their generated tests, which led to two problems: Test suites can, depending on the format, often only be executed using auxiliary programs or not at all, and test suites generated by different test-case generators can not be directly compared or combined. The XML-based exchange format solves these issues.

Supported in part by DFG grant BE 1761/7-1.

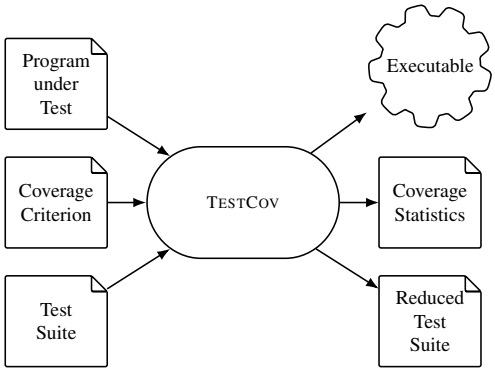

Fig. 1: Inputs and outputs of TestCov

Availability. TestCov is publicly available via GitLab [1] and as an archived package [5].

Related Work. TestCov aims at unifying and executing test suites that are created by test-case generators [7], [13], [6], [2], [9], [10], [8] for C programs. Klee [7] provides a *replay library* that can be used to create a test harness from the program under test with which it is possible to execute individual tests in the proprietary test-case format of Klee. The test cases created by AFL-fuzz[2] can be directly fed to a program. None of the existing test executors supports tests that are created by other test-case generators, nor the execution of a full test suite.

TestCov is based on BenchExec; other tools for containerization are Docker [3], LXC [4], and Snap [5]. Other projects related to the isolation and robust execution of software bugs are BugZoo [6] and the ManyBugs and IntroClass benchmarks [12].

II. Architecture of TestCov

Figure 1 shows the inputs and outputs of TestCov. TestCov gets as input the C program under test, the coverage criterion to check against, and the test suite, and creates an executable program that can be used to feed tests to the program under test, coverage statistics about the test suite, and a reduced test suite that achieves the same coverage (with respect to the coverage criterion) as the original test suite.

[1] https://gitlab.com/sosy-lab/software/test-suite-validator
[2] http://lcamtuf.coredump.cx/afl/
[3] https://www.docker.com/
[4] https://linuxcontainers.org/
[5] https://snapcraft.io/
[6] https://github.com/squaresLab/BugZoo

```xml
<?xml version="1.0"?>
<!DOCTYPE test-metadata PUBLIC [...]>
<test-metadata>
  <sourcecodelang>C</sourcecodelang>
  <producer>testcov v3.0</producer>
  <specification>CHECK( FQL(cover EDGES(@CONDITIONEDGE)) )</specification>
  <programfile>example.c</programfile>
  <programhash>eeecda9cbf27c43c9017fa00dd900c19a5ec18d46303f59a6e0357db78c33849</programhash>
  <entryfunction>main</entryfunction>
  <architecture>32bit</architecture>
  <inputtestsuitefile>original-suite.zip</inputtestsuitefile>
  <inputtestsuitehash>11911d658dcfbf8501390bf0faa96eb193b11bb1</inputtestsuitehash>
  <creationtime>2019-06-19T14:17:34Z</creationtime>
</test-metadata>
```

Fig. 2: Example metadata file of a test suite

```xml
<?xml version="1.0"?>
<!DOCTYPE testcase PUBLIC [...]>
<testcase>
  <input>'b'</input>
  <input>10</input>
  <input>0x0f</input>
</testcase>
```

Fig. 3: Example test case of a test suite

```c
#include <stdio.h>
#include <unistd.h>
extern char __VERIFIER_nondet_char();

int main() {
  char x = __VERIFIER_nondet_char();
  if (x == 'a') {
    while (1)
      fork();
  } else {
    remove("important.txt");
    if (access("important.txt", F_OK) != -1) {
      return 1;
    }
  }
}
```

Fig. 4: An example program with side effects

A. Test-Suite Exchange Format

TestCov reads and writes test suites in the XML-based exchange format for test suites, which consists of two parts: a metadata file and a set of test-case files, each defining a single test case. The metadata file is an XML file that describes the test suite and is always named `metadata.xml`. Figure 2 shows an example metadata file with all available fields. Some noteworthy fields are: the programming language of the program under test (`<sourcecodelang>`), the coverage criterion the test suite was created for (`<specification>`), the SHA-256 hash of the program under test (`<programhash>`), the program function that is tested by the test suite (`<entryfunction>`), and the system architecture the program tests were created for (`<architecture>`). If the test suite is the result of another test suite, e.g., because of test-suite reduction, the file name of this *input test suite* (`<inputtestsuitefile>`) and its SHA-256 hash (`<inputtestsuitehash>`) can also be recorded. A test-case file (Fig. 3) contains a sequence of tags `<input>` that describe the sequence of input values. The directory structure of test suites is arbitrary, and they are given to and created by TestCov as zip files for efficient storage and convenient handling. Since the exchange format is used in Test-Comp, many test-case generators support the format: CoVeriTest [2], CPA-Tiger[7], Esbmc [10], FairFuzz [13], Klee [7], PRTest [3], Symbiotic [8], and VeriFuzz [9].

B. Test Execution

For test execution, the program under test is compiled against a test harness that consists of two parts: (1) a method `get_input` for receiving test values, and (2) a new definition for each input method in the program under test that delegates to `get_input`.

Method `get_input` reads test inputs from the standard input as C-format strings and parses them into a C type. It supports hexadecimal (e.g., `0x4a`), integer (e.g., `74`), floating point (e.g., `74.5`) and character representation (e.g., `'J'`) for all primitive types (up to long double), as well as single-line string inputs. Methods from the C standard library are used for parsing.

For each input method, a new definition is introduced that calls `get_input` with the corresponding format type of the return type of the input method. For example:

```c
int inputMethod() {
  int inputVar;
  get_input("%d", &inputVar);
  return inputVar;
}
```

Given a test suite, TestCov first parses the metadata file to check consistency with the input file and coverage criterion. If one of them is not consistent, the user is informed. Then, TestCov iterates over all test-case files, reads the test inputs for each test, executes the compiled program and sequentially passes the test inputs to the execution via standard input.

To ensure that test execution does not get stuck because of a non-terminating test, a time limit is applied for each test execution. In addition, if a test case contains less input values than necessary, TestCov will terminate the execution once all input values are consumed and a new one is requested.

To ensure that test executions do not alter the system of the user, perform malicious actions, or influence each other, TestCov isolates each test execution in a separate container and control group using runexec[8], a tool provided as part of

[7] https://www.es.tu-darmstadt.de/es/team/sebastian-ruland/testcomp19/

[8] https://github.com/sosy-lab/benchexec/blob/2.0/doc/runexec.md

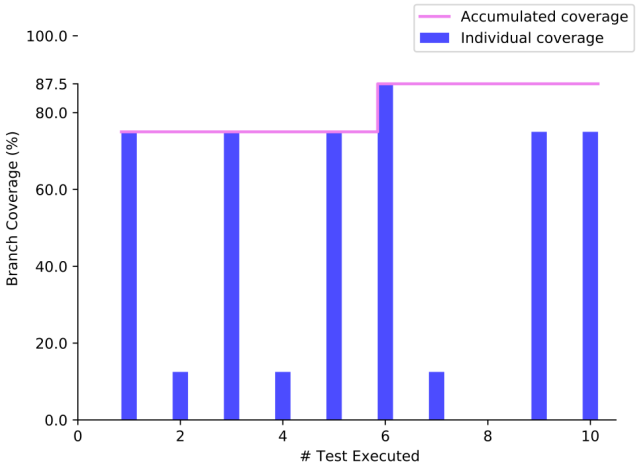

Fig. 5: Plot of individual and accumulated test coverage

```
1  int x = 1;
2  if (x > 0 || x < 0) {
3      // ...
4  }
```

Fig. 6: Code with short-circuit condition `||`

```
1  int x = 1;
2  if (x > 0 || x < 0) {
3      BRANCH_1:;
4      // ...
5  } else {
6      BRANCH_2:;
7  }
```

Fig. 7: Code instrumented to compute branch coverage

BENCHEXEC [4]. We configure it such that test executions in the container have no network access, can not see or modify other system processes, and work on an overlay file system that prevents file modifications in the original system. Files written inside the container are kept in memory and not written to disk. Keeping all file modifications in memory also speeds up test execution in the presence of file operations. Cgroups are a Linux kernel feature that allows to restrict and measure the resource consumption of a process and all its child processes. TESTCOV uses this to restrict memory usage to a user-specified maximum, to restrict computations to a specified number of CPU cores, and to enforce the time limit on test executions.

Figure 4 shows a program with side effects. The program takes a single character as input, here via Test-Comp-specific method `__VERIFIER_nondet_char`. If the input is 'a', a *fork bomb* is started that spawns an unbounded number of processes that will eventually fill the process table of the user's system and make it unusable. Otherwise, the program will delete some file, and check whether the deletion was successful. If TESTCOV is given a test suite that defines test cases for both branches, it executes both branches properly, but the number of processes is limited (by default to 5000 processes), and the file deletion only happens in the execution's container, not on the original file system, and thus, no harm is done to the user's system, while the coverage measurement is still accurate.

C. Coverage Statistics

TESTCOV provides coverage information per test and for the whole test suite, and creates plots for the coverage criterion. TESTCOV reads coverage criteria in the query language FQL [11]. Currently, it supports block, branch, and condition coverage, as well as covering calls to an error-function. To compute coverage, TESTCOV uses GCC instrumentation and LCOV. LCOV stores coverage for each line and program condition in a *tracefile*. LCOV claims to store branch coverage, but considers each condition of a short-circuit boolean operation as a separate branch. For example, the code in Fig. 6 consists of two branches: the if-branch is entered if condition `x > 0 || x < 0` is true, and the (implicit) else branch is entered otherwise. LCOV considers each evaluation of the two conditions `x > 0` and `x < 0` as a separate branch and thus reports that the program has four branches. The evaluation of the first condition (`x > 0`) is always true for that program, so every program execution takes the if-branch. Since condition `x < 0` is never evaluated, LCOV reports a branch coverage of only 25 % instead of the expected 50 %. To circumvent this and implement a proper branch-coverage measurement, TESTCOV adds program labels `BRANCH_i` at the beginning of each program branch of a program (Fig. 7) and uses the line-coverage measurement of LCOV to check which of the added program labels are covered. This way, TESTCOV can accurately measure branch coverage.

By default, LCOV stores only the accumulated coverage of all program executions in a single tracefile. To get both the accumulated coverage and the individual coverage of each separate test case, TESTCOV manages two separate tracefiles with LCOV: a default one that is newly created for each test execution and only stores the coverage of that execution, and one that contains the accumulated coverage over all test executions. While coverage information per test is usually not interesting for mere test execution, it can provide useful insights for test-suite optimization and reduction. TESTCOV provides coverage statistics as plots and as CSV files that can be easily processed further. It provides a plot (Fig. 5) that shows: (a) the accumulated test coverage (y-axis) after execution of the n-th test (x-axis) of the test suite (step plot, continuous line in Fig. 5), and (b) the test coverage of each test case (bars in Fig. 5). The order of tests in the plot is always the same as the order of execution. It is visible that, for the example, the five tests that reach 75.0 % coverage subsume the three tests that reach 12.5 % coverage, because the accumulated coverage does not increase beyond 75.0 % and 87.5 %, resp., after any of their executions. In addition, it is visible that only the 6th test executed is necessary to achieve the same branch coverage as achieved by all 9 tests of the test suite together, because it provides, on its own, the same coverage as the accumulated coverage of the full test suite.

D. Test-Suite Reduction

TESTCOV provides test-suite reduction through the strategy design pattern, so different algorithms can be added in the existing infrastructure to reduce a given test suite. By default, TESTCOV provides the following test-suite reduction technique: If the coverage criterion is to cover calls to an error-function, TESTCOV creates a new test suite that consists of one test case from the original test suite that covers that error function. If the coverage criterion is to cover lines, branches, or conditions, TESTCOV creates a new test suite that is potentially smaller than the original test suite and that achieves the same coverage.

To do so, it reads the recorded accumulated coverage after each test execution, and a test is only added to the reduced test suite if its corresponding test execution increased the accumulated coverage. TESTCOV executes tests in arbitrary order, so this approach does not necessarily produce a minimal test suite, but no additional test executions or computations are necessary for this simple but effective reduction technique.

III. USAGE

Installation. TESTCOV requires Python 3.6 or newer. The following command line installs TESTCOV and its dependencies (executed from the base directory of the TESTCOV source code):

```
> python3 setup.py install
```

Execution. TESTCOV is started via command line, with three required arguments: (1) **–test-suite** to specify the test suite to execute, (2) **–goal** to specify the coverage criterion, and (3) the program file. A test suite is provided as zip file, and a coverage criterion is provided as text file in FQL syntax. The following example command line runs TESTCOV on test suite `suite.zip`, coverage criterion `criterion.prp`, and program `prog.c`:

```
> testcov –test-suite suite.zip –goal criterion.prp prog.c
```

Directory `output` will contain all output files, i.e., the executable test harness, the reduced test suite, coverage statistics, and plots (in SVG format).

Creation of a separate container for each test execution and coverage measurement increases execution overhead because of the additional file system operations. TESTCOV provides optional arguments to turn these features off if they are not required. The following command-line prints all such arguments:

```
> testcov –help
```

Adaption of test format. To make adaption of the XML-based test format easy for test-case generators, we provide a small Python library called `tsbuilder`[9]. It can be used to programmatically create test-suite metadata and test cases in the established exchange format for test suites.

IV. APPLICATIONS

TESTCOV has been used for Test-Comp '19, where it ran almost 9 million tests created by 9 different test-case generators on 1 720 different programs and 2 different coverage criteria. TESTCOV was used for both execution and coverage measurement during the competition. All results of the competition are available online.[10] The tables that show results for several test-case generators or meta categories (e.g., `Cover-Branches`) only list the coverage computed by TESTCOV. The tables for single test generators and sub-categories (e.g., `coverage-branches.ReachSafety-Arrays`-VERIFUZZ[11]) provide the full data, including a stripped-down version of plots for accumulated test coverage.

V. CONCLUSION

TESTCOV is a tool for test-suite execution on C programs that reads test suites in the simple and standard exchange format of Test-Comp, and performs a robust and reliable test execution. TESTCOV uses BENCHEXEC, which in turn uses as foundation the containers for isolated execution and control groups for resource control that the operating-system kernel provides. The current version provides both individual and accumulated coverage statistics for four important coverage criteria. TESTCOV has been successfully used for the execution of Test-Comp '19. While TESTCOV is implemented for C programs, the used concepts can be easily transferred to other languages.

ACKNOWLEDGEMENTS

We thank Maximilian Wiesholler for his valuable contributions to the implementation of TESTCOV.

REFERENCES

[1] D. Beyer, "Competition on software testing (Test-Comp)," in *Proc. TACAS (3)*, ser. LNCS 11429. Springer, 2019, pp. 167–175. Available: https://www.doi.org/10.1007/978-3-030-17502-3_11

[2] D. Beyer and M.-C. Jakobs, "CoVeriTest: Cooperative verifier-based testing," in *Proc. FASE*, ser. LNCS 11424. Springer, 2019, pp. 389–408. Available: https://doi.org/10.1007/978-3-030-16722-6_23

[3] D. Beyer and T. Lemberger, "Software verification: Testing vs. model checking," in *Proc. HVC*, ser. LNCS 10629. Springer, 2017, pp. 99–114. Available: https://www.doi.org/10.1007/978-3-319-70389-3_7

[4] D. Beyer, S. Löwe, and P. Wendler, "Reliable benchmarking: Requirements and solutions," *Int. J. Softw. Tools Technol. Transfer*, vol. 21, no. 1, pp. 1–29, 2019. Available: https://www.doi.org/10.1007/s10009-017-0469-y

[5] D. Beyer and T. Lemberger, "Replication package for article 'TestCov: Robust test-suite execution and coverage measurement' in Proc. ASE '19," Zenodo, 2019. Available: https://doi.org/10.5281/zenodo.3418726

[6] J. Burnim and K. Sen, "Heuristics for scalable dynamic test generation," in *Proc. ASE*. IEEE, 2008, pp. 443–446. Available: https://doi.org/10.1109/ASE.2008.69

[7] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proc. OSDI*. USENIX Association, 2008, pp. 209–224.

[8] M. Chalupa, J. Strejcek, and M. Vitovská, "Joint forces for memory safety checking," in *Proc. SPIN*. Springer, 2018, pp. 115–132. Available: https://www.doi.org/10.1007/978-3-319-94111-0_7

[9] A. B. Chowdhury, R. K. Medicherla, and R. Venkatesh, "VeriFuzz: Program-aware fuzzing (competition contribution)," in *Proc. TACAS (3)*, ser. LNCS 11429. Springer, 2019, pp. 244–249. Available: https://doi.org/10.1007/978-3-030-17502-3_22

[10] M. Y. Gadelha, H. I. Ismail, and L. C. Cordeiro, "Handling loops in bounded model checking of C programs via *k*-induction," *Int. J. Softw. Tools Technol. Transf.*, vol. 19, no. 1, pp. 97–114, Feb. 2017. Available: https://www.doi.org/10.1007/s10009-015-0407-9

[11] A. Holzer, C. Schallhart, M. Tautschnig, and H. Veith, "Query-driven program testing," in *Proc. VMCAI*, ser. LNCS 5403. Springer, 2009, pp. 151–166. Available: https://doi.org/10.1007/978-3-540-93900-9_15

[12] C. Le Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. T. Devanbu, S. Forrest, and W. Weimer, "The ManyBugs and IntroClass benchmarks for automated repair of C programs," *IEEE Trans. Software Eng.*, vol. 41, no. 12, pp. 1236–1256, 2015. Available: https://doi.org/10.1109/TSE.2015.2454513

[13] C. Lemieux and K. Sen, "FairFuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage," in *Proc. ASE*. ACM, 2018, pp. 475–485. Available: https://www.doi.org/10.1145/3238147.3238176

[9] https://gitlab.com/sosy-lab/software/test-format/tree/v2.0/python_modules/tsbuilder

[10] https://test-comp.sosy-lab.org/2019/results/

[11] https://test-comp.sosy-lab.org/2019/results/results-verified/verifuzz.2019-02-06_0717.results.test-comp19_prop-coverage-branches.ReachSafety-Arrays.xml.bz2.merged.xml.bz2.table.html