AFL-TC: Transforming Fuzzer Test Inputs for Test-Comp (Competition Contribution)

Thomas Lemberger and Henrik Wachowitz*

LMU Munich, Munich, Germany

Abstract. AFL-TC is a tool chain that integrates AFL++ into the environment of Test-Comp. Coverage-guided greybox fuzzers like AFL++ produce raw binary data that is given to programs as input on stdin, without any knowledge of how this data is interpreted. In contrast to that, Test-Comp requires structured XML descriptions of test cases that list a sequence of individual input values, which are read whenever the program calls an input function. Previous adaptations of fuzzers used tool-specific modifications for Test-Comp. Now, AFL-TC demonstrates a flexible solution that decouples the test generation from the Test-Comp format: AFL-TC first runs AFL++ (or any other tester that produces binary input for stdin), then replays each input with a test harness that (a) records how the test input is interpreted by the program and (b) outputs the recording as corresponding XML elements. To provide test cases early, AFL-TC employs a monitor that triggers a transformation whenever new test files are discovered. AFL-TC participated in both Test-Comp categories Cover-Error and Cover-Branches. It placed 6th overall, 4th among active participants, and best in the sub-category `C.coverage-branches.Arrays`.

1 Test-Generation Approach

AFL++ [1] is a prominent coverage-guided greybox fuzzer for C. So far, two variants of AFL++ participated in Test-Comp: FairFuzz [2, 3] in 2019 and cetfuzz [4] in 2024. However, the original AFL++ remained to be evaluated in Test-Comp. We change this with AFL-to-Test-Case (AFL-TC), a tool chain that combines AFL++ with the tool fuzz-to-tc [5]. fuzz-to-tc transforms binary test inputs—as produced by AFL++ and other fuzzers—into structured Test-Comp test cases.

Each test case produced by AFL++ is a file with raw binary content that is provided to the program under test via standard input. How the program reads this binary content into concrete input values is not visible from the outside. As first example, given the binary input in hex notation in Fig. 1, a program could read the first four bytes as an integer n and then read the next n bytes as a sequence of char values. This interpretation of input values is not evident from the binary content itself. As second example, consider the program in Fig. 2. Through the special Test-Comp input method `__VERIFIER_nondet_int()`, the

* Jury member

```
ff ff ff ff 01 00 00 00
01 00 00 00 01 00 00 00
01 00 00 00 01 00 00 00
01 00 00 00 01 00 00 00
01 11 00 00 01 00 00 00
01 00 00 00 01 00 00 00
01 00 00 00 01 00 00 00
01 00 00 00 01 00 00 00
01 00 00 00 01 00 00 00 0a
```

Fig. 1: Binary input generated by AFL++, in hex notation

```c
int main() {
  int x = __VERIFIER_nondet_int();
  if (x != 0) {
    reach_error();
  }
}
```

Fig. 2: C program with reachable error

```
<?xml version="1.0" [...] ?>
<!DOCTYPE testcase [...]>
<testcase>
  <input type="int">-1</input>
</testcase>
```

Fig. 3: Test-Comp test case produced by FUZZ-TO-TC by executing Fig. 2 with Fig. 1

program receives a signed 32-bit integer and stores it in variable x. If $x \neq 0$, it calls the error function `reach_error()`. When AFL++ runs on this program, it does generate the error-triggering binary input in Fig. 1 (shown in hex notation). But the program only reads the first four bytes `ff ff ff ff` and interprets them as the value -1. The remaining bytes of the generated input are superfluous.

In contrast to the raw binary format produced by AFL++, test cases in Test-Comp are represented by an XML file that describes the sequence of concrete input values in the order they are supposed to be read by the program. Whenever the program execution calls an input method, the method returns precisely the next value from the sequence.

To receive structured XML test cases from AFL++, AFL-TC executes a tool chain that consists of three steps: (1) Run AFL++ on the program under test to generate inputs in the default binary format of AFL++; (2) watch for new tests in the relevant output directories of AFL++; (3) whenever a new test is discovered, transform it to the Test-Comp format. Figure 3 shows the Test-Comp test case that this tool chain produces for Fig. 2 (by transforming Fig. 1).

To run AFL++ on Test-Comp benchmark tasks, AFL-TC compiles each program under test against a test harness that defines the Test-Comp-specific input methods. Each method (a) reads the number of bytes that match its expected return type from standard input, (b) casts the read bytes to the expected return type, and (c) returns that value. For the test transformation this harness is extended to output the expected XML element for a value before returning it. This dual-mode behavior is controlled by a compile-time macro. The original test harness without XML output originates from FairFuzz [3].

For the category Cover-Error, AFL-TC monitors only the crash directory of AFL++, and only keeps tests whose execution actually call the error function. For the category Cover-Branches, AFL-TC monitors both the crash and queue directories and keeps all generated tests.

2 Software Architecture

Figure 4 shows the workflow of AFL-TC. First, AFL-TC passes the program P and the test harness H to the AFL++ compiler `afl-clang-lto` (and, as fallback, `afl-gcc-fast`). This creates an AFL-instrumented binary. Next, AFL-TC starts an AFL++ run on the AFL-instrumented executable to generate binary test files.

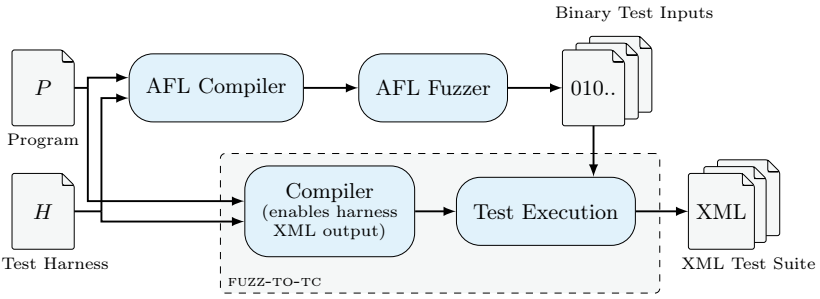

Fig. 4: Workflow of AFL-TC

AFL++ writes newly created tests to an output directory `queue/`. Tests that trigger a program crash are then moved to a separate directory `crash/`. While AFL++ runs, AFL-TC enters a loop to monitor these output directories. For test goal `coverage-error` (Test-Comp category Cover-Error), it watches only the directory `crash/`. For test goal `coverage-branches` (Test-Comp category Cover-Branches), it watches both directories. Whenever the monitor discovers a new test file it passes the currently existing tests to FUZZ-TO-TC for transformation.

FUZZ-TO-TC is a standalone Go tool. It receives the coverage goal, a directory with test cases to transform, the program under test as source code, the machine model to compile the program for, and, for metadata generation, the name of the original test generator (here: AFL++). FUZZ-TO-TC first compiles the program against our test harness with enabled XML output and then runs the resulting executable once for each binary test case, using that test case as program input. For each execution, FUZZ-TO-TC captures the produced XML output and saves it as a Test-Comp-conforming test case. For `coverage-error` tasks, FUZZ-TO-TC only saves test cases that actually reach the error function during execution. FUZZ-TO-TC also creates a necessary metadata file for the test suite. To avoid duplicate tests when FUZZ-TO-TC is run multiple times, it saves each test case under the name of the SHA1 hash of its contents.

3 Strengths and Weaknesses

Strengths. AFL-TC is a tool chain that adapts AFL++ for Test-Comp, but the framework behind AFL-TC can be used with other binary-format fuzzers. The test-generation harness and FUZZ-TO-TC are tool-agnostic. AFL++ itself uses coverage-guided greybox fuzzing, which reaches broad code coverage fast [1] and has found many bugs in real-world software [6]. In Test-Comp 2026, AFL-TC reaches the 6$^{\text{th}}$ place overall and 4$^{\text{th}}$ among active participants [7]. It outperforms CETFUZZ [4], a derivative of AFL++, justifying the participation of the original AFL++ to improve the comparability of participants' performance. AFL-TC reached the highest score among participants in sub-category `C.coverage-branches.Arrays`.

Weaknesses. Greybox fuzzing struggles with complex input constraints that require specific byte sequences. Tasks where reaching the target requires navigating

a complex series of branches (e.g., tasks from XCSP) are challenging for AFL-TC. Unlike symbolic-execution tools, AFL-TC does not reason about program paths on a semantic level, limiting its effectiveness on programs that require precise input sequences. We use the CmpLog instrumentation [8] to mitigate this issue. This instrumentation enables the tracing of comparison operations and better steering of mutators towards branches that are not covered yet [9].

AFL++ is not built for low CPU time usage. It runs as many threads and iterations as possible, consuming significant CPU resources. In the Test-Comp scenario, where CPU time per task is strictly limited, this leads to frequent timeouts. It also restricts the strategies we can use for test-case generation. For example, it is too resource-intensive in Test-Comp to start with deterministic test-input mutations, the default in AFL++. Instead, we directly rely on random mutations to achieve a broad exploration of the program fast.

4 Tool Setup and Configuration

AFL-TC is available open source [10]. The version used in Test-Comp is archived at Zenodo [11]. Installation is possible via FM-Weck [12] or manual setup. When fuzzing 32-bit programs on a 64-bit system, the `gcc-multilib` package is required. In manual setup, the tool requires an installation of AFL++. Our Test-Comp archive contains pre-compiled binaries of AFL++ for x64 Linux systems. To run the tools with newer versions of AFL++ or on other architectures, we recommend the official Docker container [13] that is provided by the AFL++ team.

Installation. To install AFL-TC via FM-Weck into directory `afltc/`, run:

```
fm-weck install afltc:testcomp26 -d afltc
```

Use. AFL-TC takes as input a C program (`<program>`), the machine model (`<arch>`) to compile the program under test for, and a coverage goal (`<spec>`). The command-line to run AFL-TC is:

```
afl-tc <program> <arch> <spec>
```

This produces a Test-Comp-conforming test suite in directory `output`. Supported machine models are `32bit` and `64bit`.

Configuration. The concrete AFL++ compilation and fuzzing can be customized by editing the commands `afl-clang-lto`, `afl-gcc-fast`, and `afl-fuzz` in the shell script `bin/afl-tc`. By default, the fuzzing runs with fixed seed 42 (`-s 42`) for reproducibility; uses the CmpLog instrumentation [8] (`AFL_LLVM_CMPLOG=1`) to better handle comparison operands; and lets AFL++ decide on a time limit per run automatically (`-t1000+`).

5 Software Project and Contributors

The underlying fuzzer AFL++ is maintained by Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marcel Heuse. AFL-TC and FUZZ-TO-TC are maintained by Thomas Lemberger and Henrik Wachowitz at LMU Munich.

Data-Availability Statement. AFL-TC is available open source under the Apache-2.0 license at gitlab.com/sosy-lab/software/test-to-witness. The version of AFL-TC that was used in Test-Comp 2026 is archived at https://doi.org/10.5281/zenodo.18060896. All results and artifacts from Test-Comp 2026 are available in the competition report [7]. AFL++ is available open source under the Apache-2.0 license at github.com/AFLplusplus/AFLplusplus.

Funding. This work is supported by the Deutsche Forschungsgemeinschaft (DFG) – ConVeY (378803395) and IdeFix (496588242).

References

1. Fioraldi, A., Maier, D., Eißfeldt, H., Heuse, M.: AFL++: Combining incremental steps of fuzzing research. In: Proc. WOOT. USENIX Association (2020)
2. Lemieux, C., Sen, K.: FairFuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In: Proc. ASE. pp. 475–485. ACM (2018). https://doi.org/10.1145/3238147.3238176
3. Lemieux, C., Sen, K.: FairFuzz-TC: A fuzzer targeting rare branches (competition contribution). Int. J. Softw. Tools Technol. Transf. **23**(6), 863–866 (December 2021). https://doi.org/10.1007/s10009-020-00569-w
4. Krishnan, S., George, S.S., Medicherla, R.K., Divakaran, S.: Gitlab repository of cetfuzz. https://gitlab.com/Sarathkrishnan/cetfuzz, accessed: 2026-01-22
5. Beyer, D., Lemberger, T., Wachowitz, H.: Testing in formal verification via witness generation (empirical evaluation). In: Proc. FASE. LNCS, Springer (2026)
6. Zalewski, M.: American Fuzzy Lop. https://lcamtuf.coredump.cx/afl/, accessed: 2026-01-30
7. Beyer, D.: Evaluating tools for automatic software testing: Test-Comp 2026. In: Proc. FASE. LNCS 16504, Springer (2026)
8. Repository, A.G.: CmpLog instrumentation. https://github.com/AFLplusplus/AFLplusplus/blob/68b492b2c7725816068718ef9437b72b40e67519/instrumentation/README.cmplog.md, accessed: 2026-01-30
9. Aschermann, C., Schumilo, S., Blazytko, T., Gawlik, R., Holz, T.: Redqueen: Fuzzing with input-to-state correspondence. In: Symposium on Network and Distributed System Security (NDSS) (2019). https://doi.org/10.14722/ndss.2019.23371
10. Lemberger, T., Wachowitz, H.: AFL-TC: Test-to-witness translation for afl++. https://gitlab.com/sosy-lab/software/test-to-witness, accessed: 2026-01-22
11. Lemberger, T., Wachowitz, H.: AFL-TC competition participation Test-Comp 2026 (archive). Zenodo (2025). https://doi.org/10.5281/zenodo.18060896
12. Beyer, D., Wachowitz, H.: FM-Weck: Containerized execution of formal-methods tools. In: Proc. FM. pp. 39–47. LNCS 14934, Springer (2024). https://doi.org/10.1007/978-3-031-71177-0_3
13. Heuse, M., Eißfeldt, H., Fioraldi, A., Maier, D., Zalewski, M.: AFL++ docker image. https://hub.docker.com/r/aflplusplus/aflplusplus, accessed: 2026-01-30