/coverage-analysis

Source: `~/.claude/skills/tob-testing-handbook-skills/skills/coverage-analysis/SKILL.md`

---

name: coverage-analysis type: technique description: > Coverage analysis measures code exercised during fuzzing. Use when assessing harness effectiveness or identifying fuzzing blockers.

Coverage Analysis

Coverage analysis is essential for understanding which parts of your code are exercised during fuzzing. It helps identify fuzzing blockers like magic value checks and tracks the effectiveness of harness improvements over time.

Overview

Code coverage during fuzzing serves two critical purposes:

1. **Assessing harness effectiveness**: Understand which parts of your application are actually executed by your fuzzing harnesses
2. **Tracking fuzzing progress**: Monitor how coverage changes when updating harnesses, fuzzers, or the system under test (SUT)

Coverage is a proxy for fuzzer capability and performance. While coverage is not ideal for measuring fuzzer performance in absolute terms, it reliably indicates whether your harness works effectively in a given setup.

Key Concepts

Concept	Description
Coverage instrumentation	Compiler flags that track which code paths are executed
Corpus coverage	Coverage achieved by running all test cases in a fuzzing corpus
Magic value checks	Hard-to-discover conditional checks that block fuzzer progress
Coverage-guided fuzzing	Fuzzing strategy that prioritizes inputs that discover new code paths
Coverage report	Visual or textual representation of executed vs. unexecuted code

When to Apply

Apply this technique when:

- Starting a new fuzzing campaign to establish a baseline
- Fuzzer appears to plateau without finding new paths
- After harness modifications to verify improvements
- When migrating between different fuzzers
- Identifying areas requiring dictionary entries or seed inputs
- Debugging why certain code paths aren't reached

Skip this technique when:

- Fuzzing campaign is actively finding crashes
- Coverage infrastructure isn't set up yet
- Working with extremely large codebases where full coverage reports are impractical
- Fuzzer's internal coverage metrics are sufficient for your needs

Quick Reference

Task	Command/Pattern
LLVM coverage instrumentation (C/C++)	`-fprofile-instr-generate -fcoverage-mapping`
GCC coverage instrumentation	`-ftest-coverage -fprofile-arcs`
cargo-fuzz coverage (Rust)	`cargo +nightly fuzz coverage <target>`
Generate LLVM profile data	`llvm-profdata merge -sparse file.profraw -o file.profdata`
LLVM coverage report	`llvm-cov report ./binary -instr-profile=file.profdata`
LLVM HTML report	`llvm-cov show ./binary -instr-profile=file.profdata -format=html -output-dir html/`
gcovr HTML report	`gcovr --html-details -o coverage.html`

Ideal Coverage Workflow

The following workflow represents best practices for integrating coverage analysis into your fuzzing campaigns:

```
[Fuzzing Campaign]
       |
       v
[Generate Corpus]
       |
       v
[Coverage Analysis]
       |
       +---> Coverage Increased? --> Continue fuzzing with larger corpus
       |
       +---> Coverage Decreased? --> Fix harness or investigate SUT changes
       |
       +---> Coverage Plateaued? --> Add dictionary entries or seed inputs
```

Key principle: Use the corpus generated *after* each fuzzing campaign to calculate coverage, rather than real-time fuzzer statistics. This approach provides reproducible, comparable measurements across different fuzzing tools.

Step-by-Step

Step 1: Build with Coverage Instrumentation

Choose your instrumentation method based on toolchain:

LLVM/Clang (C/C++):

```
clang++ -fprofile-instr-generate -fcoverage-mapping \
  -O2 -DNO_MAIN \
  main.cc harness.cc execute-rt.cc -o fuzz_exec
```

GCC (C/C++):

```
g++ -ftest-coverage -fprofile-arcs \
  -O2 -DNO_MAIN \
  main.cc harness.cc execute-rt.cc -o fuzz_exec_gcov
```

Rust:

```
rustup toolchain install nightly --component llvm-tools-preview
cargo +nightly fuzz coverage fuzz_target_1
```

Step 2: Create Execution Runtime (C/C++ only)

For C/C++ projects, create a runtime that executes your corpus:

```
// execute-rt.cc
#include <stdio.h>
#include <stdlib.h>
#include <dirent.h>
#include <stdint.h>

extern "C" int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size);

void load_file_and_test(const char *filename) {
    FILE *file = fopen(filename, "rb");
    if (file == NULL) {
        printf("Failed to open file: %s\n", filename);
        return;
    }
```

```
    fseek(file, 0, SEEK_END);
    long filesize = ftell(file);
    rewind(file);

    uint8_t *buffer = (uint8_t*) malloc(filesize);
    if (buffer == NULL) {
        printf("Failed to allocate memory for file: %s\n", filename);
        fclose(file);
        return;
    }

    long read_size = (long) fread(buffer, 1, filesize, file);
    if (read_size != filesize) {
        printf("Failed to read file: %s\n", filename);
        free(buffer);
        fclose(file);
        return;
    }

    LLVMFuzzerTestOneInput(buffer, filesize);

    free(buffer);
    fclose(file);
}

int main(int argc, char **argv) {
    if (argc != 2) {
        printf("Usage: %s <directory>\n", argv[0]);
        return 1;
    }

    DIR *dir = opendir(argv[1]);
    if (dir == NULL) {
        printf("Failed to open directory: %s\n", argv[1]);
        return 1;
    }

    struct dirent *entry;
    while ((entry = readdir(dir)) != NULL) {
```

```
        if (entry->d_type == DT_REG) {
            char filepath[1024];
            snprintf(filepath, sizeof(filepath), "%s/%s", argv[1], entry->d_name);
            load_file_and_test(filepath);
        }
    }

    closedir(dir);
    return 0;
}
```

Step 3: Execute on Corpus

LLVM (C/C++):

```
LLVM_PROFILE_FILE=fuzz.profraw ./fuzz_exec corpus/
```

GCC (C/C++):

```
./fuzz_exec_gcov corpus/
```

Rust: Coverage data is automatically generated when running `cargo fuzz coverage`.

Step 4: Process Coverage Data

LLVM:

```
# Merge raw profile data
llvm-profdata merge -sparse fuzz.profraw -o fuzz.profdata

# Generate text report
llvm-cov report ./fuzz_exec \
  -instr-profile=fuzz.profdata \
  -ignore-filename-regex='harness.cc|execute-rt.cc'

# Generate HTML report
llvm-cov show ./fuzz_exec \
  -instr-profile=fuzz.profdata \
  -ignore-filename-regex='harness.cc|execute-rt.cc' \
  -format=html -output-dir fuzz_html/
```

GCC with gcovr:

```
# Install gcovr (via pip for latest version)
python3 -m venv venv
source venv/bin/activate
pip3 install gcovr

# Generate report
gcovr --gcov-executable "llvm-cov gcov" \
  --exclude harness.cc --exclude execute-rt.cc \
  --root . --html-details -o coverage.html
```

Rust:

```
# Install required tools
cargo install cargo-binutils rustfilt

# Create HTML generation script
cat <<'EOF' > ./generate_html
#!/bin/sh
if [ $# -lt 1 ]; then
    echo "Error: Name of fuzz target is required."
    echo "Usage: $0 fuzz_target [sources...]"
    exit 1
fi
FUZZ_TARGET="$1"
shift
SRC_FILTER="$@"
TARGET=$(rustc -vV | sed -n 's|host: ||p')
cargo +nightly cov -- show -Xdemangler=rustfilt \
  "target/$TARGET/coverage/$TARGET/release/$FUZZ_TARGET" \
  -instr-profile="fuzz/coverage/$FUZZ_TARGET/coverage.profdata" \
  -show-line-counts-or-regions -show-instantiations \
  -format=html -o fuzz_html/ $SRC_FILTER
EOF
chmod +x ./generate_html

# Generate HTML report
./generate_html fuzz_target_1 src/lib.rs
```

Step 5: Analyze Results

Review the coverage report to identify:

- **Uncovered code blocks**: Areas that may need better seed inputs or dictionary entries
- **Magic value checks**: Conditional statements with hardcoded values that block progress
- **Dead code**: Functions that may not be reachable through your harness
- **Coverage changes**: Compare against baseline to track improvements or regressions

Common Patterns

Pattern: Identifying Magic Values

Problem: Fuzzer cannot discover paths guarded by magic value checks.

Coverage reveals:

```
// Coverage shows this block is never executed
if (buf == 0x7F454C46) {  // ELF magic number
    // start parsing buf
}
```

Solution: Add magic values to dictionary file:

```
# magic.dict
"\x7F\x45\x4C\x46"
```

Pattern: Handling Crashing Inputs

Problem: Coverage generation fails when corpus contains crashing inputs.

Before:

```
./fuzz_exec corpus/  # Crashes on bad input, no coverage generated
```

After:

```
// Fork before executing to isolate crashes
int main(int argc, char **argv) {
    // ... directory opening code ...
```

```
    while ((entry = readdir(dir)) != NULL) {
        if (entry->d_type == DT_REG) {
            pid_t pid = fork();
            if (pid == 0) {
                // Child process - crash won't affect parent
                char filepath[1024];
                snprintf(filepath, sizeof(filepath), "%s/%s", argv[1], entry->d_name);
                load_file_and_test(filepath);
                exit(0);
            } else {
                // Parent waits for child
                waitpid(pid, NULL, 0);
            }
        }
    }
}
```

Pattern: CMake Integration

Use Case: Adding coverage builds to CMake projects.

```
project(FuzzingProject)
cmake_minimum_required(VERSION 3.0)

# Main binary
add_executable(program main.cc)

# Fuzzing binary
add_executable(fuzz main.cc harness.cc)
target_compile_definitions(fuzz PRIVATE NO_MAIN=1)
target_compile_options(fuzz PRIVATE -g -O2 -fsanitize=fuzzer)
target_link_libraries(fuzz -fsanitize=fuzzer)

# Coverage execution binary
add_executable(fuzz_exec main.cc harness.cc execute-rt.cc)
target_compile_definitions(fuzz_exec PRIVATE NO_MAIN)
target_compile_options(fuzz_exec PRIVATE -O2 -fprofile-instr-generate -fcoverage-mapping)
target_link_libraries(fuzz_exec -fprofile-instr-generate)
```

Build:

```
cmake -DCMAKE_C_COMPILER=clang -DCMAKE_CXX_COMPILER=clang++ .
cmake --build . --target fuzz_exec
```

Advanced Usage

Tips and Tricks

Tip	Why It Helps
Use LLVM 18+ with `-show-directory-coverage`	Organizes large reports by directory structure instead of flat file list
Export to lcov format for better HTML	`llvm-cov export -format=lcov` + `genhtml` provides cleaner per-file reports
Compare coverage across campaigns	Store `.profdata` files with timestamps to track progress over time
Filter harness code from reports	Use `-ignore-filename-regex` to focus on SUT coverage only
Automate coverage in CI/CD	Generate coverage reports automatically after scheduled fuzzing runs
Use gcovr 5.1+ for Clang 14+	Older gcovr versions have compatibility issues with recent LLVM

Incremental Coverage Updates

GCC's gcov instrumentation incrementally updates `.gcda` files across multiple runs. This is useful for tracking coverage as you add test cases:

```
# First run
./fuzz_exec_gcov corpus_batch_1/
gcovr --html coverage_v1.html

# Second run (adds to existing coverage)
./fuzz_exec_gcov corpus_batch_2/
gcovr --html coverage_v2.html

# Start fresh
gcovr --delete  # Remove .gcda files
./fuzz_exec_gcov corpus/
```

Handling Large Codebases

For projects with hundreds of source files:

1. **Filter by prefix**: Only generate reports for relevant directories

   ```
   llvm-cov show ./fuzz_exec -instr-profile=fuzz.profdata /path/to/src/
   ```

2. **Use directory coverage**: Group by directory to reduce clutter (LLVM 18+)

   ```
   llvm-cov show -show-directory-coverage -format=html -output-dir html/
   ```

3. **Generate JSON for programmatic analysis**:

   ```
   llvm-cov export -format=lcov > coverage.json
   ```

Differential Coverage

Compare coverage between two fuzzing campaigns:

```
# Campaign 1
LLVM_PROFILE_FILE=campaign1.profraw ./fuzz_exec corpus1/
llvm-profdata merge -sparse campaign1.profraw -o campaign1.profdata

# Campaign 2
LLVM_PROFILE_FILE=campaign2.profraw ./fuzz_exec corpus2/
llvm-profdata merge -sparse campaign2.profraw -o campaign2.profdata

# Compare
llvm-cov show ./fuzz_exec \
  -instr-profile=campaign2.profdata \
  -instr-profile=campaign1.profdata \
  -show-line-counts-or-regions
```

Anti-Patterns

Anti-Pattern	Problem	Correct Approach
Using fuzzer-reported coverage for comparisons	Different fuzzers calculate coverage differently, making cross-tool comparison meaningless	Use dedicated coverage tools (llvm-cov, gcovr) for reproducible measurements

Anti-Pattern	Problem	Correct Approach
Generating coverage with optimizations	`-O3` optimizations can eliminate code, making coverage misleading	Use `-O2` or `-O0` for coverage builds
Not filtering harness code	Harness coverage inflates numbers and obscures SUT coverage	Use `-ignore-filename-regex` or `--exclude` to filter harness files
Mixing LLVM and GCC instrumentation	Incompatible formats cause parsing failures	Stick to one toolchain for coverage builds
Ignoring crashing inputs	Crashes prevent coverage generation, hiding real coverage data	Fix crashes first, or use process forking to isolate them
Not tracking coverage over time	One-time coverage checks miss regressions and improvements	Store coverage data with timestamps and track trends

Tool-Specific Guidance

libFuzzer

libFuzzer uses LLVM's SanitizerCoverage by default for guiding fuzzing, but you need separate instrumentation for generating reports.

Build for coverage:

```
clang++ -fprofile-instr-generate -fcoverage-mapping \
  -O2 -DNO_MAIN \
  main.cc harness.cc execute-rt.cc -o fuzz_exec
```

Execute corpus and generate report:

```
LLVM_PROFILE_FILE=fuzz.profraw ./fuzz_exec corpus/
llvm-profdata merge -sparse fuzz.profraw -o fuzz.profdata
llvm-cov show ./fuzz_exec -instr-profile=fuzz.profdata -format=html -output-dir html/
```

Integration tips:

- Don't use `-fsanitize=fuzzer` for coverage builds (it conflicts with profile instrumentation)
- Reuse the same harness function (`LLVMFuzzerTestOneInput`) with a different main function
- Use the `-ignore-filename-regex` flag to exclude harness code from coverage reports
- Consider using llvm-cov's `-show-instantiation` flag for template-heavy C++ code

AFL++

AFL++ provides its own coverage feedback mechanism, but for detailed reports use standard LLVM/GCC tools.

Build for coverage with LLVM:

```
clang++ -fprofile-instr-generate -fcoverage-mapping \
  -O2 main.cc harness.cc execute-rt.cc -o fuzz_exec
```

Build for coverage with GCC:

```
AFL_USE_ASAN=0 afl-gcc -ftest-coverage -fprofile-arcs \
  main.cc harness.cc execute-rt.cc -o fuzz_exec_gcov
```

Execute and generate report:

```
# LLVM approach
LLVM_PROFILE_FILE=fuzz.profraw ./fuzz_exec afl_output/queue/
llvm-profdata merge -sparse fuzz.profraw -o fuzz.profdata
llvm-cov report ./fuzz_exec -instr-profile=fuzz.profdata

# GCC approach
./fuzz_exec_gcov afl_output/queue/
gcovr --html-details -o coverage.html
```

Integration tips:

- Don't use AFL++'s instrumentation (`afl-clang-fast`) for coverage builds
- Use standard compilers with coverage flags instead
- AFL++'s `queue/` directory contains your corpus
- AFL++'s built-in coverage statistics are useful for real-time monitoring but not for detailed analysis

cargo-fuzz (Rust)

cargo-fuzz provides built-in coverage generation using LLVM tools.

Install prerequisites:

```
rustup toolchain install nightly --component llvm-tools-preview
cargo install cargo-binutils rustfilt
```

Generate coverage data:

```
cargo +nightly fuzz coverage fuzz_target_1
```

Create HTML report script:

```
cat <<'EOF' > ./generate_html
#!/bin/sh
FUZZ_TARGET="$1"
shift
SRC_FILTER="$@"
TARGET=$(rustc -vV | sed -n 's|host: ||p')
cargo +nightly cov -- show -Xdemangler=rustfilt \
  "target/$TARGET/coverage/$TARGET/release/$FUZZ_TARGET" \
  -instr-profile="fuzz/coverage/$FUZZ_TARGET/coverage.profdata" \
  -show-line-counts-or-regions -show-instantiations \
  -format=html -o fuzz_html/ $SRC_FILTER
EOF
chmod +x ./generate_html
```

Generate report:

```
./generate_html fuzz_target_1 src/lib.rs
```

Integration tips:

- Always use the nightly toolchain for coverage
- The `-Xdemangler=rustfilt` flag makes function names readable
- Filter by source files (e.g., `src/lib.rs`) to focus on crate code
- Use `-show-line-counts-or-regions` and `-show-instantiations` for better Rust-specific output
- Corpus is located in `fuzz/corpus/<target>/`

honggfuzz

honggfuzz works with standard LLVM/GCC coverage instrumentation.

Build for coverage:

```
# Use standard compiler, not honggfuzz compiler
clang -fprofile-instr-generate -fcoverage-mapping \
  -O2 harness.c execute-rt.c -o fuzz_exec
```

Execute corpus:

```
LLVM_PROFILE_FILE=fuzz.profraw ./fuzz_exec honggfuzz_workspace/
```

Integration tips:

- Don't use `hfuzz-clang` for coverage builds
- honggfuzz corpus is typically in a workspace directory
- Use the same LLVM workflow as libFuzzer

Troubleshooting

Issue	Cause	Solution
`error: no profile data available`	Profile wasn't generated or wrong path	Verify `LLVM_PROFILE_FILE` was set and `.profraw` file exists
`Failed to load coverage`	Mismatch between binary and profile data	Rebuild binary with same flags used during execution
Coverage reports show 0%	Wrong binary used for report generation	Use the instrumented binary, not the fuzzing binary
`no_working_dir_found` error (gcovr)	`.gcda` files in unexpected location	Add `--gcov-ignore-errors=no_working_dir_found` flag
Crashes prevent coverage generation	Corpus contains crashing inputs	Filter crashes or use forking approach to isolate failures
Coverage decreases after harness change	Harness now skips certain code paths	Review harness logic; may need to support more input formats
HTML report is flat file list	Using older LLVM version	Upgrade to LLVM 18+ and use `-show-directory-coverage`
`incompatible instrumentation`	Mixing LLVM and GCC coverage	Rebuild everything with same toolchain

Related Skills

Tools That Use This Technique

Skill	How It Applies
libfuzzer	Uses SanitizerCoverage for feedback; coverage analysis evaluates harness effectiveness
aflpp	Uses edge coverage for feedback; detailed analysis requires separate instrumentation
cargo-fuzz	Built-in `cargo fuzz coverage` command for Rust projects

Skill	How It Applies
honggfuzz	Uses edge coverage; analyze with standard LLVM/GCC tools

Related Techniques

Skill	Relationship
fuzz-harness-writing	Coverage reveals which code paths harness reaches; guides harness improvements
fuzzing-dictionaries	Coverage identifies magic value checks that need dictionary entries
corpus-management	Coverage analysis helps curate corpora by identifying redundant test cases
sanitizers	Coverage helps verify sanitizer-instrumented code is actually executed

Resources

Key External Resources

LLVM Source-Based Code Coverage Comprehensive guide to LLVM's profile instrumentation, including advanced features like branch coverage, region coverage, and integration with existing build systems. Covers compiler flags, runtime behavior, and profile data formats.

llvm-cov Command Guide Detailed CLI reference for llvm-cov commands including `show`, `report`, and `export`. Documents all filtering options, output formats, and integration with llvm-profdata.

gcovr Documentation Complete guide to gcovr tool for generating coverage reports from gcov data. Covers HTML themes, filtering options, multi-directory projects, and CI/CD integration patterns.

SanitizerCoverage Documentation Low-level documentation for LLVM's SanitizerCoverage instrumentation. Explains inline 8-bit counters, PC tables, and how fuzzers use coverage feedback for guidance.

On the Evaluation of Fuzzer Performance Research paper examining limitations of coverage as a fuzzing performance metric. Argues for more nuanced evaluation methods beyond simple code coverage percentages.

Video Resources

Not applicable - coverage analysis is primarily a tooling and workflow topic best learned through documentation and hands-on practice.

Revision #5
Created 2026-02-18 08:40:11 UTC by John
Updated 2026-06-21 20:01:23 UTC by John