Test Case Generation Based On Fuzzing For C++

Ia Mgvdliashvili

May 13, 2019

Contents

Chapter 1

Introduction

Most software heavily relies on unit tests as its primary source for logic and fault tolerance verification. This approach has been largely considered as essential, but it has some inherent difficulties associated with it. Although testing single member functions independently is more often than not trivial, most of the time the user will call various combinations of them. It is impossible to write unit tests with all possible function call sequences since such space is effectively infinite. Therefore, the need arises for the developer to personally determine which function call sequences are most meaningful.

Other than that, a lot of times the behavior of the function will depend on internal state of the instance, which is in itself reached after certain function calls.

1.1 Background

1.1.1 Fuzzing

Dynamic analysis, or fuzzing, is a popular and effective method of finding vulnerabilities in software. Fuzz testing reaches impressive results in exposing interface vulnerabilities in very short amount of time.

Fuzzing heavily relies on the concept of Fuzz target - a function that accepts an array of bytes and then uses it in user defined way against the API under test. This API usually has a single endpoint that consumes any kind data. Anything that causes an exception, abort, exit, crash, assert failure, timeout is considered a bug[1]. That means, discovery of the first instance of any one of them will cause the fuzzer to halt and inform us about the input that caused the bug, along with some other information.

There are a number of tools available for fuzzing, including AFL and Radamsa. One of the most notable implementations is LibFuzzer, LLVM's tool for coverage guided, evolutionary fuzzing engine[2]. The code coverage information for LibFuzzer is provided by SanitizerCoverage instrumentation, and I will discuss it in the next subsection.

1.1.2 SanitizerCoverage library

LLVM has an interface for its built-in code coverage instrumentation[3]. The user is able to gather information about the covered regions of the program during runtime. There are several different levels of depth for coverage, and the library also offers rich ways to trace the data flow. This tool was crucial for the development of my program and in the developer's manual, I discuss the library in more detail.

1.2 Using fuzz testing for unit tests

Although fuzz testing has been mostly defined to be for exploiting the vulnerabilities of the program, I tried to apply its coverage based philosophy to explore the possible member function call sequences and pinpoint ones which might be most interesting for the developer.
This also required to change the overall approach with which fuzzing is used.

In 1.1.1 Background on Fuzzing, I talked about the classical assumptions about the fuzz target. In our scenario, we have different expectations - since we are testing an entire unit and not a single API endpoint, some kind of control flow disruptions might be expected. For example, assertions are common in member functions. Therefore, the previous approach of exiting on first such failure should be modified to allow the program to gather information about all possible combinations that result in things like exceptions, so the user will be informed about them and decide what constitutes the normal behavior of their library and what is outside of specifications.

1.3 Program description

To achieve the intended results, I created a program that uses LLVM's SanitizerCoverage library and generation based fuzzing. The test case needs almost minimal setup which consists of the user specifying all the member functions it wants to use in testing, and passing a single function pointer for constructing an instance of the class. Modern C++ tools have aided greatly with this by giving the ability to store pointers to functions with different type signatures. There are still difficulties with regards to determining and passing the function arguments, which is in scope of a larger project. In order for this issue to not interfere with the initial program implementation, I allow users to additionally pass pointers to the functions that will in turn call the specific member function with desired arguments.

1.4 Results

Although the original intention was to discover new test cases, there were some surprising outcomes that could not have been anticipated. For example, the program is very good in minimizing the total number of test cases, since it only saves the new sequence of function calls if new coverage was discovered, or if it replaces one or more other sequences by combining their result together. This outcome would be crucial for reducing the size of test suites, which leads to reduced runtime and maintenance cost.

Chapter 2

User Documentation

This software aids in finding test cases for a single class interface by trying out different permutations of member function calls and discovering ones that give the most coverage. In this chapter, I provide full information about its usage for users of the program. I will be using a simple stack implementation as an example[4].

2.1 Intended audience

This software is intended for C++ developers who would like to discover meaningful function call sequences that provide good coverage for their interface/class. Therefore, at least basic knowledge of C++ is assumed, and the user will need to implement and pass pointers for several functions.

2.1.1 Requirements

Some restrictions apply with regards to the test target class. It should satisfy following for the program to work properly:

The program is intended to test a single unit

Current version can not analyze any of the dependencies of the class in some cases. Although the feature might be able to work with multiple classes and different member functions, only single one is supported at this stage.

You need to be able to be compiled separately

In order to analyze , the object file of the test target needs to be compiled with special flags separately from the rest of the project. This means the implementation of the class can not be spread in multiple files, and those files should not contain anything else

2.2 System Requirements

Following dependencies need to be installed in order to

- GNU Make 4.0+

- Clang 7.0+ with AddressSanitizer (included with the compiler by default)

2.3 Installation instructions

run these commands from the directory where you have copied the contents of the project and run commands provided in Listing 1
After tests pass successfully, you can move on to next step and set up the project to test your class.

Listing 1 Installation commands

```
cd <dir> # where dir is the path of the folder


make test-main


make integration-test-run
```

2.4 Instructions using an example

After installation, and successful tests, you need to modify the contents of src/main.cc file and several lines in Makefile. I will demonstrate the instructions using an example.

2.4.1 Sample stack class

Listing 2 shows the definition for the class that the project tests by default. Full implementation can be found in corresponding .cpp file of the same directory. I will go through the example and explain how it can be adjusted for any other class.

2.4.2 Structure of the main file

The user is advised to only change the contents of `main()` function, and replace the `#include` directive. I will explain how to construct the CombinationTester class instance and all necessary dependencies. More details about how classes used in the main.cc file work and way they are engineered can be found in Developer documentation.

Listing 4 shows how the main.cc file should look like. You shoud replace all template arguments with your class instead of stack<int>.

Listing 2 definition for the stack class in examples/stack.h

```
template <typename T>
class stack {
  T *arr;
  int top;
  int capacity;
  bool outPutMessages = false;

public:
  stack(int size = SIZE); // constructor
  ~stack();
  void toggleOutput(bool newValue);
  void push(T);
  T pop();
  T peek();
  int size();
  bool isEmpty();
  bool isFull();
};
```

Listing 3 defining user variables in Makefile

```
TEST_TARGET_FILE := examples/stack.cpp
TEST_TARGET_INC := -I examples
```

Listing 4 Contents of src/main.cc

```
int main(int argc, char **argv) {
  // Function that does all the initialization
  InstanceFunctionPointer<stack<int>> getStackInstance =
      [](int sequenceLength) {
        stack<int> s(sequenceLength);
        return s;
      };
  // Function pointers
  FunctionPointerMap<stack<int>> memberFunctions;
  memberFunctions.insert("pop", &stack<int>::pop);
  memberFunctions.insert("peek", &stack<int>::peek);
  memberFunctions.insert("size", &stack<int>::size);
  memberFunctions.insert("isEmpty", &stack<int>::isEmpty);
  memberFunctions.insert("isFull", &stack<int>::isFull);
  memberFunctions.insertNonVoid("push1",
                                [](stack<int> &a) { a.push(1); });
  CombinationTester<stack<int>> combinationTester(
      4, memberFunctions, getStackInstance, &coverageReporter);
  combinationTester.run();
  coverageReporter.printResults();
  coverageReporter.printResultsToFile();
  std::cout << "done\n";
  return 0;
}
```

Follow these steps to set up the main.cc file:

1. Include your header file and provide it's path

In the beginning of Makefile, set the $(TEST_TARGET_FILE) and $(TEST_TARGET_INC) variables to point to correct file and path, respectively. For example, the stack class located in examples directory of the project path would be set as shown on Listing 3.

2. Create a function that constructs an instance of your class

This function will be called in the beginning of testing each function call sequence. int sequenceLength will be passed and you can use this value if it's applicable.

3. Insert pointers to member functions

Choose which functions you would like to use for testing. Feel free to omit ones that are not crucial to coverage, for example getters. For functions that require arguments, you need to pass pointers (or simply lambda functions)

4. Construct the CombinationTester class

You need to pass the objects that you created in steps 2 and 3. Feel free to tweak the maximum number of combinations to suit your library's needs. If the functions are relatively small but there is a greater number of them, it is more reasonable to settle for numbers lower than 5. In some cases large size of functions requires a very specific state of the class which can only be achieved by more function calls. Such cases will better be resolved by mutation-based fuzzing.

5. Choose the output format

After finishing, you can ask the coverageReporter to show results by simply printing it or writing to a file

2.4.3 Compile and run

If you followed the instructions in the previous section, you are ready to generate test cases for your class using `make run` command.

Understanding the output

After the program is done running, it will display the results as you indicated in the end of main.cc file. The example is provided in Listing 5, 3 different sequences of function calls that will cover the blocks listed below them. The results are not perfect and contain a small number of redundancy, but all different scenarios are discovered, including the one where calls of pop() occur more times than push().

memory leaks

Since the program will be compiled using the `−fsanitize=address` compiler flag to include the library that is responsible for observing coverage. AddressSanitizer library detects memory issues that your unit might have. For example, if we were to remove the destructor from the stack implementation, we would get the message shown on Listing 6.

Listing 5 Output of program for the stack class

```
sequence: push1 peek pop pop
        in stack<int>::isEmpty() examples/stack.cpp:90
        in stack<int>::isFull() examples/stack.cpp:97
        in stack<int>::peek() examples/stack.cpp:69
        in stack<int>::peek() examples/stack.cpp:71:13
        in stack<int>::pop() examples/stack.cpp:48
        in stack<int>::pop() examples/stack.cpp:52:13
        in stack<int>::pop() examples/stack.cpp:58:9
        in stack<int>::push(int) examples/stack.cpp:30
        in stack<int>::push(int) examples/stack.cpp:39:9
sequence: size push1 peek pop
        in stack<int>::isEmpty() examples/stack.cpp:90
        in stack<int>::isFull() examples/stack.cpp:97
        in stack<int>::peek() examples/stack.cpp:69
        in stack<int>::peek() examples/stack.cpp:71:13
        in stack<int>::pop() examples/stack.cpp:48
        in stack<int>::pop() examples/stack.cpp:58:9
        in stack<int>::push(int) examples/stack.cpp:30
        in stack<int>::push(int) examples/stack.cpp:39:9
        in stack<int>::size() examples/stack.cpp:83
sequence: size push1 pop pop
        in stack<int>::isEmpty() examples/stack.cpp:90
        in stack<int>::isFull() examples/stack.cpp:97
        in stack<int>::pop() examples/stack.cpp:48
        in stack<int>::pop() examples/stack.cpp:52:13
        in stack<int>::pop() examples/stack.cpp:58:9
        in stack<int>::push(int) examples/stack.cpp:30
        in stack<int>::push(int) examples/stack.cpp:39:9
        in stack<int>::size() examples/stack.cpp:83
```

Listing 6 Memory leak discovered when the destructor is missing

```
==30539==ERROR: LeakSanitizer: detected memory leaks


Direct leak of 10656 byte(s) in 645 object(s) allocated from:
    #0 0x4f2f22 in operator new[](unsigned long) ...
    #1 0x4fe6ae in stack<int>::stack(int) examples/stack.cpp:11:11
    #2 0x4ffcb5 in main::$_1::operator()(int) const ...
    #3 0x4ffc79 in main::$_1::__invoke(int) (bin/main+0x4ffc79)
    #4 0x5015c6 in CombinationTester<stack<int> >::run() ...
    #5 0x4ff6c7 in main (bin/main+0x4ff6c7)
    #6 0x7fa3bbaeeb96 in __libc_start_main ...

SUMMARY: AddressSanitizer: 10656 byte(s) leaked in 645 allocation(s).
```

For more information about interpreting and fixing these messages visit documentation for AddressSanitizer[5].

2.5 Troubleshooting

Following issues might arise during different parts of using the program.

2.5.1 Problems during installation

the tests are designed so that all of the underlying infrastructure will be checked. If you start having any problems:

test if clang build works correctly

If there were problems with the make integration−test−run command, it will be because AddressSanitizer is experiencing issues.

2.5.2 Problems during runtime

Out of Memory error for AddressSanitizer

This happens if the source code size is too large, resulting in a lot of guards and callbacks. Please refer to the requirements section. This error could be fixed by advanced tweaking of the AdressSanitizer options, but as for now is not supported in the project.

Chapter 3

Developer Documentation

This chapter discusses the structure and contents of the source directory, essential concepts for development and reasoning behind some of the architecture decisions. It also provides overview of the tools necessary for development, along with their usage.

3.1 Dependencies and internal architecture

Before diving into the specifics, I would like to introduce some material that is necessary for understanding why - and how, everything works. First, I will give a brief overview to the SanitizerCoverage library and introduce the test class which will illustrate core concepts of both the library and my program.

3.1.1 SanitizerCoverage library

The program relies primarily on LLVM's built in coverage instrumentation to measure coverage of different function cal sequences. Basic understanding of how these functions work is necessary for development.

With `−fsanitize−coverage=trace−pc−guard` flag, the clang compiler will insert the following code on every edge of the control flow. Every edge will have its own $\text{guard}_{\text{variable}}$ (uint32_{t}), and in the end the instrumentation will look as given on Listing 7. Here, "pc" stands for "program counter", and I used this term in the source code and tests as well to describe the parts of the program.

Listing 7 How SanitizerCoverage instrumentation looks

```
if(*guard)
    __sanitizer_cov_trace_pc_guard(guard);
```

There is another function that will be called at least once per dynamic shared object (it may be called more than once with the same values of start/stop).

__sanitizer_cov_trace_pc_guard_init(uint32_t *start, uint32_t *stop);

These callbacks are not implemented in the Sanitizer run-time and should be defined by the user. This mechanism is used for fuzzing the Linux kernel, as well as the LibFuzzer library mentioned earlier.

Let me illustrate this using an example. First let us introduce a simple class on Listing 8. Most functions have descriptive names and their implementation does exactly that, so I will explain only what f4() does, and also note that f4() returns a simple integer value without any calculations.

As you will observe on Listing 9, there are 3 different scenarios for f4(), each one being a simple return statement. SanitizerCoverage will insert guards on entry point of each of them. The rest of the functions will simply have guards inserted in the beginning.

SanitizerCoverage library offers three different levels for observing the control flow of the program. Instrumentation points can be an edge, basic blocks, and

Listing 8 Definition of the class used in integration tests

```
class IntegrationTestClass {
public:
  void increaseCounter();
  void setToggle(bool);
  int f2();

private:
  bool toggle = false;
  int counter = 1;
  bool counterIs2();
  int f4();
};
```

function entry points. For this project, using the default edge one was more appropriate, since the target of fuzzing will be small and we can afford detailed coverage. There are different ways offered for tracing data flow, for example callbacks for comparison operations and switches. These could aid in refining the program for more complex applications later.

In the next section, I will continue the IntegrationTestClass example. It serves as a good demonstration for showing how the program works, before going into the details about each unit.

3.1.2 Example

Let us get back to our test class and think about how we would accomplish to cover all three blocks of f4(). The first two return statements are reached if

Listing 9 Implementation of IntegrationTestClass::f2() provided in test/integrationTestClass.cc, with inserted guards highlighted (TODO)

```
  if (toggle) {
    if (counterIs2()) {
      return f4();
    } else {
      return 0;
    }
  } else {
    return 7;
  }
}
```

setToggle(true) has been previously called, since the default value of toggle is false. Calling f4() without doing that first results in entering the third branch.

As you see, the program manages to find all of the scenarios that we dicssed, as shown on Listing 10.

You will observe that tweaking the number of maximum sequence length will increase the number of covered blocks per unit test (and have larger sequences of function calls) while significantly decreasing the total number of unit tests, until it collapses to very few, maybe even one. It is not reasonable to test the entire logic in a single test, and also not clear when to stop replacing shorter sets of coverage with longer ones (with longer sequences). This also varies depending on the complexity of the unit. However, it is always guaranteed that a longer sequence will never replace a shorter one if their coverage is the same, as observed when tweaking the number from 6 to 7 for IntegrationTestClass.
Right now, it is recommended to start with smaller number of function calls. It is an interesting task to determine the threshold after which having a larger coverage

Listing 10 snippet from output of make integration−test−run shows that the program covered all different cases for IntegrationTestClass

```
sequence: increaseCounter toggleTrue f2 increaseCounter f2
        in IntegrationTestClass::counterIs2()
        ↪  test/integrationTestClass.cc:24
        in IntegrationTestClass::f2() test/integrationTestClass.cc:12
        in IntegrationTestClass::f2() test/integrationTestClass.cc:15:14
        in IntegrationTestClass::f2() test/integrationTestClass.cc:17:7
        in IntegrationTestClass::f4() test/integrationTestClass.cc:28
        in IntegrationTestClass::increaseCounter()
        ↪  test/integrationTestClass.cc:4
        in IntegrationTestClass::setToggle(bool)
        ↪  test/integrationTestClass.cc:8
sequence: increaseCounter increaseCounter f2 toggleTrue f2
        in IntegrationTestClass::counterIs2()
        ↪  test/integrationTestClass.cc:24
        in IntegrationTestClass::f2() test/integrationTestClass.cc:12
        in IntegrationTestClass::f2() test/integrationTestClass.cc:15:14
        in IntegrationTestClass::f2() test/integrationTestClass.cc:20:5
        in IntegrationTestClass::f4() test/integrationTestClass.cc:28
        in IntegrationTestClass::increaseCounter()
        ↪  test/integrationTestClass.cc:4
        in IntegrationTestClass::setToggle(bool)
        ↪  test/integrationTestClass.cc:8
sequence: increaseCounter f2 toggleTrue f2
```

per unit test stops being important.

3.1.3 Documentation

This was a higher-level overview of the core concepts used for the program. The rest of the documentation is provided in the last section of this chapter. All classes are thoroughly documented in header files, and some more detailed explanation and reasoning is sometimes provided in implementation files. For documentation generation, Doxygen library is used with javadoc style.

If you have doxygen installed, Run doxygen Doxyfile to generate documentation in html and latex source. Latex source needs additional compiling which can be done by running the command make in the latex directory. If you would like to change doxygen settings, you can copy the Doxyfile and run doxygen Doxyfile.

3.1.4 code conventions

Code is formatted according to LLVM coding standard[6]. Clang-format[7] is used and can be added to commit hook, or alternatively use clang-format plugin for IDE of your choice.

3.2 Makefile and project layout

As mentioned before, the project is compiled using GNU make, since it was more straightforward to express the different compilation commands and necessary flags for Sanitizer library. Each recipe is self documenting by using @echo to display its purpose when invoked. Since this is the most crucial information, it is highlighted in shell output. After that the compile command is displayed in less bright color, so that the user and developer are not distracted but can still observe which commands are being ran.

3.2.1 Source directory contents

Here is a brief overview of the source directory. Each of the subdirectories is displayed alongside corresponding variable in the Makefile.

$(INC) - include

Header files. Definitions for all classes of the project. It also contains definition of the template classes along with their implementation. The `lib` subdirectory holds the header file of the testing framework.

$(SRCDIR) - src

Implementations of non-template classes from `include` directory and a sample for the main file, which should be replaced by user for its own test target unit.

$(BUILDDIR), $(TARGETDIR) - build, bin

Output for binary and object files

$(TESTDIR) - test

Test directory. Tests are discussed in detail in 3.3 Testing strategy, but compilation instructios will be provided here.

3.2.2 Other Makefile variables

$(SRCEXT)

Most files in the project end with .cpp extension. .cc is reserved for special file types which requre specific compilation instructions. Having different extensions allow to quickly filter out such files.

Object files for classes

Located in src directory, ending with .cpp file extension. Listing 11 shows how they are compiled.

$(CC) and %(CFLAGS)

Clang++ is used for all compilation commands, and all possible warnings are turned on and treated as errors in order to ensure good code quality.

$(INC)

Adds include and include/lib (containing the test library) directories to include path.

3.2.3 Compilation commands

main and default command

make run runs bin/main, which is compiled by default when running make, with all necessary parts, shown on Listing 11. These include:

1. Object files from the src directory

Listing 11 compilation commands for bin/main in Makefile

```
$(BUILDDIR)/%.o: $(SRCDIR)/%.$(SRCEXT)
        @echo -e "\t \e[96mCompiling object\e[90m"
        @mkdir -p $(BUILDDIR)
        $(CC) $(CFLAGS) $(INC) -c -o $@ $<

$(INSERTED_GUARDS): $(TEST_TARGET_FILE)
        @echo -e "\t \e[96mCompiling the test target class with
         ↪  guards\e[90m"
        clang++ -c -g -o $@ $^ -fsanitize-coverage=trace-pc-guard

$(TARGET): $(OBJECTS) $(INSERTED_GUARDS) $(MAINFILE)
        @echo -e "\t \e[96m Linking with sanitizer coverage\e[90m"
        @mkdir -p $(TARGETDIR)
        $(CC) $(INC) $(TEST_TARGET_INC) $(SANITIZERFLAGS) $^ -o $@
```

2. Program Test target class, compiled with instrumentation Steps discussed in Section 3.1.1

3. Compiler flag for AddressSanitizer This flag is responsible for callbacks using the guards instrumented in the test target object file

docs

Generates the documentation as described in Section `documentation`

Listing 12 contents of test/catch2−main.cpp defining the main function for test framework

```
#define CATCH_CONFIG_MAIN
#include "catch.hpp"
```

clean

Empties the output directories and removes all object / target files.

test and integration test

These commands are discussed more later in Section 3.3.

3.3 Testing strategy

The project is thoroughly tested using the catch2 framework. Tests are represented with Behavior-driven Given-When-Then style, and described scenarios carefully follow documentation. This library was chosen for its minimalistic setup and ability to describe the test cases with full sentences.

3.3.1 Catch2 library

Catch is a multi-paradigm test framework for C++, distributed as a single header file. Though that does not mean that it needs to be compiled into every translation unit. Since it needs to be defined just once and used for any test case, it is possible and more efficient[8] to compile it to an object file which is later included in tests, as displayed on Listings 12 and 13.

After this, we can compile the test files. Unit tests follow the convention by having the corresponding class name followed by "-test" suffix, and their

Listing 13 compiling contents of test/catch−main.cpp

```
TEST_MAIN := $(TESTDIR)/catch2-main.cpp
TEST_LIB := $(BUILDDIR)/catch2-main.o
$(TEST_LIB): $(TEST_MAIN)
        @echo -e "\t \e[96m Compiling the test library\e[90m"
        $(CC) $(INC) -c $^ -o $@
```

compilation recipe is shown in Listing 14. The exception to this is the integration test, which was discussed in Section 3.1.2. I will provide some more details about it in this section.

The library offers two different ways of structuring test cases, and I chose the GIVEN-WHEN-THEN one because of its behavior driven style syntax, which allowed me to express expectations more concisely.

3.3.2 An example test case

The library was created by test driven development, and all features discussed in documentation have a corresponding test case. I will walk through one example in more detail. Unit with most scenarios was CoverageReporter, since it needs to decide when the new coverage was meaningful/worth storing. I will show how I handled one of the scenarios.

Let us discuss what should happen if a new function sequence recorded set of pc blocks that has already been saved earlier. Listing 15 shows the setup for the scenario and first test when the new coverage is covered by the sequence of same length.

Listing 14 including compiled catch2−main.cpp in tests and compiling them

```
TESTEXT := -test.$(SRCEXT)
TEST_SOURCES := $(shell find $(TESTDIR) -type f -name *$(TESTEXT))
TEST_OBJECTS := $(patsubst $(TESTDIR)/%,$(BUILDDIR)/%,\
$(TEST_SOURCES:.$(SRCEXT)=.o))

TEST_TARGET := $(TARGETDIR)/test
$(TEST_TARGET): $(TEST_OBJECTS) $(OBJECTS) $(TEST_LIB)
        @mkdir -p $(TARGETDIR)
        @echo -e "\t \e[96mLinking tests\e[90m"
        $(CC) $^ -o $(TEST_TARGET)

$(BUILDDIR)/%-test.o: $(TESTDIR)/%$(TESTEXT)
        @echo -e "\t \e[96mCompiling unit test\e[90m"
        @mkdir -p $(BUILDDIR)
        $(CC) $(CFLAGS) $(INC) -c -o $@ $<
```

Listing 15 Setup for scenario "coverage for new sequence already exists" in test/coverageReporter−test.cpp, and case when new coverage is reached with longer sequence compared to what was recorded earlier

```
TestData testData;
CoverageReporter cr;
cr.startCoverage(testData.sequence1);
pc_set coverage = {"pc1", "pc2"};
cr.addPCForSequence("pc1");
cr.addPCForSequence("pc2");
cr.flush();
auto sequence = cr.coverageSequences.find(coverage)->second;
REQUIRE(sequence == testData.sequence1);

GIVEN("new coverage with longer sequence") {
  cr.startCoverage(testData.longSequence);
  cr.addPCForSequence("pc1");
  cr.addPCForSequence("pc2");
  cr.flush();
  THEN("existing sequence for coverage should not be replaced") {
    auto sequence = cr.coverageSequences.find(coverage)->second;
    REQUIRE(sequence == testData.sequence1);
  }
```

Listing 16 Case when new coverage has a shorter sequence in "coverage for new sequence already exists" scenario

```
GIVEN("new coverage with shorter sequence") {
  cr.startCoverage(testData.shortSequence);
  cr.addPCForSequence("pc1");
  cr.addPCForSequence("pc2");
  cr.flush();
  THEN("existing sequence for coverage should be updated") {
    auto sequence = cr.coverageSequences.find(coverage)->second;
    REQUIRE(sequence == testData.shortSequence);
  }
```

Same should happen if the sequence is the same size, and that is checked as well. However, if the same coverage was achieved by a shorter sequence, we would like to reflect that. Listing 16 shows how I check the behavior.

I also paid attention to error handling. Listing 17 shows the scenario when the user flushes collected coverage but forgets setting a sequence beforehand. This needs to hold true after flushing any number of times, which is why I first start the coverage and flush correctly, but do not call the startCoverage() function for the second time.

3.3.3 Integration test

The example discussed in Section 3.1.2 is used to test the combinationTester class. Since this class only combines the functionalities of other ones without adding a lot of logic and scenarios, I use the test case to determine that the Sanitizer Coverage library properly works and at least one pc is reported during runtime. The compilation commands are very similar to the ones used for main.cc, and they

Listing 17 Asserting that CoverageReporter communicates the error in case of developer forgetting to set the sequence.

```
SCENARIO("user forgot to set current sequence", "[coveragereporter]")
 ↪  {
  CoverageReporter cr;
  std::vector<std::string> sequence{"f1", "f2"};
  cr.startCoverage(sequence);
  cr.addPCForSequence("pc1");
  cr.flush();
  GIVEN("coverage reporter without current sequence") {
    cr.addPCForSequence("pc2");
    THEN("flushing would cause an exception") {
     ↪  REQUIRE_THROWS(cr.flush()); }
  }
```

are provided in Listing 18.

Listing 18 Compiling the integration test Makefile

```
INTEGRATION_TEST := combinationTester-test
INTEGRATION_TEST_CLASS := integrationTestClass
INTEGRATION_TEST_FILE := $(TESTDIR)/$(INTEGRATION_TEST).cc
INTEGRATION_TEST_TARGET := $(TARGETDIR)/integration-test

# compile the integration test class with sanitizer flag
INTEGRATION_TEST_CLASS_FILE :=
↪  $(TESTDIR)/$(INTEGRATION_TEST_CLASS).cc
INTEGRATION_TEST_GUARDS :=
↪  $(BUILDDIR)/$(INTEGRATION_TEST_CLASS)-guards.o
$(INTEGRATION_TEST_GUARDS): $(INTEGRATION_TEST_CLASS_FILE)
        @echo -e "\t \e[96mCompiling the integration test class with
        ↪  guards\e[90m"
        @mkdir -p $(BUILDDIR)
        $(CC) -c -g $^ -fsanitize-coverage=trace-pc-guard -o $@

$(INTEGRATION_TEST_TARGET): $(INTEGRATION_TEST_GUARDS) $(TEST_LIB) \
$(OBJECTS) $(INTEGRATION_TEST_FILE)
        @mkdir -p $(TARGETDIR)
        @echo -e "\t \e[96mLinking integration test...\e[90m"
        $(CC) $(INC) $(SANITIZERFLAGS) $^ -o $@

integration-test-run: $(INTEGRATION_TEST_TARGET)
        @echo -e "\t \e[96mRunning the integration test with
        ↪  $(INTEGRATION_TEST_CLASS)\e[90m"
        ASAN_OPTIONS=strip_path_prefix=`pwd`/ ./bin/integration-test
```

References

[1] Kostya Serebryany, Fuzz or lose! https://github.com/CppCon/CppCon2017/blob/master/Demos/Fuzz%20Or%20Lose/Fuzz%20Or%20Lose%20-%20Kostya%20Serebryany%20-%20CppCon%202017.pdf

[2] Libfuzzer documentation, https://llvm.org/docs/LibFuzzer.html

[3] Clang SanitizerCoverage library Documentation, https://clang.llvm.org/docs/SanitizerCoverage.html

[4] Techiedelight, Stack Implementation in C++, https://www.techiedelight.com/stack-implementation-in-cpp/

[5] Clang AddressSanitizer documentation, https://clang.llvm.org/docs/AddressSanitizer.html

[6] LLVM coding standards, https://llvm.org/docs/CodingStandards.html

[8] Improving catch library compilation, https://github.com/catchorg/Catch2/blob/master/docs/slow-compiles.md#top

[7] Clang-Format documentation, https://clang.llvm.org/docs/ClangFormatStyleOptions.html