Eötvös Loránd Tudományegyetem

Faculty of Informatics

Department of Programming Languages

Test case generation based on fuzzing for C++

Supervisor:

Zoltan Porkolab

Associate Professor

Author:

Ia Mgvdliashvili

Computer Science BSc

Budapest, 2019

Test based development is a favorable development method for modern software. We create all the necessary test cases to test the software under development and then we implement the functionality. This is a widely accepted method for library development, when the test cases try to cover all meaningful combinations of API calls. However, in real software systems, the possible combinations can grow exponentially. It is very hard to determine the minimun necessary set of meaningful API call sequences. In this thesis we try to apply fuzzy testing methods for automatically generate API call sequences for testing C++ libraries. We will use the LLVM toolset to exploit the existing code coverage and test input mutation methods. However, our target is not to gereate a random input sequence but a meaningful sequence of API calls. It is also in our plans to analyse the result to create a minimal classification set.

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 libfuzzer 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 LLVM's 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 covereged regions of the program during runtime. There are severeal 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] `https://github.com/CppCon/CppCon2017/blob/master/Demos/Fuzz%20Or%20Lose/Fuzz%20Or%20Lose%20-%20Kostya%20Serebryany%20-%20CppCon%202017.pdf`

[2] `https://llvm.org/docs/LibFuzzer.html`

[3] `https://clang.llvm.org/docs/SanitizerCoverage.html`

1.2 using fuzz testing for ..

Although fuzz testing has been mostly defined to be for exploiting the vulnerabilities of the program, we decided 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 constitues the normal behavior of their library and what is outside of specifications.

1.3 Program description

To acheive the intended results, I created a program that uses LLVM's sanitizer coverage 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 research / 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.

The sanitizer coverage library is able to communicate its results using a single global object. (more about this)

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. For the sample stack class, it discovered that in . 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 section provides full information for users of the program. I am using a simple stack implementation as an example.

2.1 intended audience

this software is intended for c++ developers who would like to increase
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

(. . . .) and test target class should satisfy following:

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.

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

make test-main

make test # to make sure that everything works
```

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 dependencies

2.3 installation instructions

run these commands from the directory where you want to install the project
After tests pass successfully, you can move on to next step and set up the

2.4 Instructions using an example

After installation, and successful tests (TODO)

2.4.1 Sample stack class

`Listing _TODO_` 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

Listing 1 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();
};
```

other class.

2.4.2 Structure of the main file

The user is adviced to only change the contents of `main()` function, and replace the `#include` directive. More details about how these classes work and way they are engineered can be found in the 3. Developer Documentation (TODO)

Listing 2 example file

```
 * libraries needed for running
 */


// replace this line with the header of your own class
#include "stack.h"
```

Global objects

Getting the output

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

2.4.3 Compile and run

using commands
This command will (. . . .relevant section from the makefile)

2.4.4 output

(TODO)

results

memory leaks

After it's finished running

```
==32362==ERROR: LeakSanitizer: detected memory leaks
```

Since the program will be compiled using the '-fsanitize=address' compiler flag, any existing memory leaks will be discovered.
(example with a stack class but missing constructor)
For more info about interpreting and fixing these messages visit AddressSanitizer.

2.5 troubleshooting

There are few mistakes

2.5.1 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

There might be problems with the addressSanitizer. To see if the program can run independently, use the make command, which will compile and run all the source files without the flag. The program will still work and call functions, but the coverage will not be reported. If this step is successful describe how then please check your compiler

2.5.2 running

(TODO reproduce)

Out of Memory error for AddressSanitizer

This happens because .... Please refer to the requirements section. This error could be fixed by tweaking the AdressSanitizer, but as for now is not supported in the project.

Chapter 3

Developer Documentation

This section 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 Information about the project

Before going in-depth about the details of software, I would like to elaborate more on some commands that were dicusses in the User Manual.

3.1.1 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.

(TODO) contents and commends of Makefile will be broken down here. show the file

include

Header files. Definitions for 3 main classes of the project. It also contains definition of the template class along with its implementation.

src

Implementations of classes from /include directory (excluding functionPointerMap which is a template class) and a sample for the main file, which should be replaced by user for its own test target unit.

source file extensions

this makes it easier to create a comprehensive but concise makefile which scans the source folder for .cpp files and .cc is used for main and

test

Test directory. Tests are discussed later in `catch2`

Makefile

The test section of the makefile is discussed in catch2 (TODO link)

3.1.2 code conventions

Code is formatted according to LLVM coding standard[4]. Clang-format is used and can be added to commit hook, or alternatively use clang-format plugin for IDE

[4]`https://llvm.org/docs/CodingStandards.html`

`https://clang.llvm.org/docs/ClangFormatStyleOptions.html` of your choice.

3.2 Dependencies and internal architecture

3.2.1 Catch2 library and tests

The project is thoroughly tested using the catch2 framework. Tests are represented with 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, and facilitates test driven development. You can find test cases corresponding to each scenario discussed in the documentation. (TODO? is it ok to finish with this?).

```
make test-main
```

This compiles the test-main.cpp which defines the main function of catch. Since it needs to be defined just once and used for any test case, it is more efficient to compile it to an object which is later included in tests.

```
make test
```

runs the tests for all units in the project, excluding the combination tester.

The exception to this is the integration test, which will be discussed in the section ([Example]). It serves as a good example for showing how the library works, before going into the details about each unit. First, I will give a brief overview to the Sanitizer Coverage library and introduce the test class which will illustrate core concepts of both the library and my program. (TODO review this sentence)

3.2.2 Sanitizer Coverage library

The program relies primarily on LLVM's built in coverage instrumentation to measure coverage of different function cal sequences. Basic understanding of how

Listing 3 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();
};
```

these functions work is necessary for development.

(paragraph about guards,)

Let me illustrate this using an example. First let us introduce a simple class on `Listing 3`. Most functions have descriptive names and their implementation does exactly that, so I will explain only what `f2()` does, and also note that f4() returns a simple integer value without any calculations.

As you will observe on `Listing 3`, there are 3 different scenarios for `f2()`, and Sanitizer Coverage will insert guards on entry point of each of them. The rest of the functions will simply have guards inserted in the beginning.

Sanitizer Coverage library offers numerous ways to observe the control flow of the program, three default ones being (.... edge, block, explain differences). It also includes (... for switches and). These could aid in refining the program for more complex applications but will not be covered here since only edge case is used currently.

Listing 4 Implementation of `IntegrationTestClass::f2()`, with inserted guards highlighted (TODO)

```
int IntegrationTestClass::f2() {
  if (toggle) {
    if (counterIs2()) {
      return f4();
    } else {
      return 0;
    }
  } else {
    return 7;
  }
```

3.2.3 Example

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 growing), and it might also vary from the complexity of the unit. However, it will never replace . . . , 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 hard to determine the threshold after which having a larger coverage per unit test stops being important. If we restrict the program to only save the function call sequence if it covers a new block, the number of unit tests will (TODO? I think this is actually a reasonable approach, and should be the default and I could say that other alternatives could be found, at the same time, I don't think I have time to implement, test and document even such a small feature, since I did added other ones recently).

Choosing the right combination of number of function calls with regards to number of covered blocks is for another project.

This was a higher-level overview of the core concepts used for the program. The rest of the documentation is provided in the following section.

3.2.4 documentation

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.

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 `my-Doxyfile`.

(TODO the blank rest of the page here to bind second pdf)