

Lab 5: Cooperative Bug Isolation (CBI)

Fall 2021 Semester

Due: 8 November, 8:00 am Eastern Time

Corresponding Lecture: Lesson 9 (Statistical Debugging)

Resources

- Enumerating basic blocks and instructions in a function:
 - <http://releases.llvm.org/8.0.0/docs/ProgrammersManual.html#basic-inspection-and-traversal-routines>
- Instrumenting LLVM IR
 - <http://releases.llvm.org/8.0.0/docs/ProgrammersManual.html#creating-and-inserting-new-instructions>
- Important classes
 - <http://releases.llvm.org/8.0.0/docs/ProgrammersManual.html#the-function-class>
 - https://llvm.org/doxygen/classllvm_1_1CallInst.html
 - https://llvm.org/doxygen/classllvm_1_1DebugLoc.html
 - https://llvm.org/doxygen/classllvm_1_1BranchInst.html

Objective

In this lab, you will implement cooperative bug isolation (CBI) to statistically localize error locations. You need to implement an LLVM pass that instruments each branch and function call to report the values of their conditions and return values. With the large number of inputs, you will obtain the data at runtime and measure different types of scores that help users find bugs.

You should find this lab familiar. The basic structure of the first part will be very similar to Lab 1 (Fuzzing). If you find you're having trouble implementing the instrumentation, we recommend you refer back to that lab.

Setup

Download `cbi.zip` from Canvas. When uncompressed, it will produce the directory `cbi/`

This lab builds off the work you did in Lab 1 (Fuzzing). We have provided a reference solution in the form of a binary, `reference/Instrument.so`, and `reference/fuzzer`. If you prefer to use your own solution from the previous lab, copy `Instrument.so` and `fuzzer` from `LAB1/build` to `LAB5/reference`.

The following commands set up the lab (from the lab's directory):

```
$ mkdir build && cd build
$ cmake ..
$ make
```

You should now see `CBIInstrumentPass.so` and `cbi` in the current directory.

The `cbi` tool performs statistical debugging for a program using a feedback profile (which you will generate) for successful and erroneous program runs. To help generate programs runs that pass and fail, you will use your `fuzzer`:

```
$ cd test
$ make
$ mkdir fuzz_output
$ timeout 10 ../build/fuzzer ./fuzz0 fuzz_input fuzz_output 10
$ ../build/cbi ./fuzz0 fuzz_output
```

You should see the sample output as follows:

```
== S(P) ==
== F(P) ==
== Failure(P) ==
== Context(P) ==
== Increase(P) ==
```

Lab Instructions

In this lab, you will need to edit the `src/CBIInstrument.cpp` file to implement the cooperative bug isolation which will instrument branch and return instructions with code to extract and monitor predicates. `lib/runtime.c` contains functions that you will use in your lab:

```
// Append predicate information as "branch,line,col,cond" to the running process
// cbi file
void __cbi_branch__(int line, int col, int cond) { /**/ }

// Append predicate information as "return,line,col,rv" to the running process
// cbi file
void __cbi_return__(int line, int col, int rv) { /**/ }
```

Like you did in Lab 1, your LLVM pass should instrument the code with these functions.

Your pass should instrument each conditional branch with code records whether the branch conditional is true or false on execution. Likewise, instrument each integer-returning call instruction with code to record the return value. This will create a *feedback profile* for you to perform statistical debugging and generate a *feedback report*.

In short, the lab consists of the following tasks:

1. Implement the `instrumentCBIBranches` function to insert a `__cbi_branch__` call for predicate (conditional). **Note: `__cbi_branch__` takes a 32-bit integer as a parameter for condition. You'll need to convert the boolean (1-bit integer) condition to a 32-bit integer. `True == 1` (not `-1`) and `False == 0`**
2. Modify `runOnFunction` to instrument all conditional branching instructions with the predicate recording logic
3. Implement the `instrumentCBIReturns` function to insert a `__cbi_return__` call for a return value
4. Modify `runOnFunction` to instrument all 32-bit integer return instructions with the return recording logic
5. Using the feedback profile, you construct in 1-4, modify `generateReport` to implement statistical debugging. You should compute `F(P)`, `S(P)`, `Failure(P)`, `Context(P)`, and `Increase(P)` which should be stored in the corresponding data structures in `include/Utils.h`

Revisiting Instrumentation. By now you should feel comfortable working with the LLVM compiler infrastructure, but for a refresher, consult Lab 0 and Lab 1.

CBI File Infrastructure. The `cbi` executable will execute the input program on each of the trace input files from a `fuzzer` output directory. This includes both successful program runs (`fuzz_output/success`) and erroneous program runs (`fuzz_output/failure`). Each run will generate an analogous feedback profile for each input file. The resulting directory tree will look like this:

```
fuzz_output/  
  success/  
    input1  
    input1.cbi  
    input2  
    input2.cbi  
    ...  
  failure/  
    input1  
    input1.cbi  
    ...
```

You will use these `.cbi` files to generate the feedback report.

Generating the feedback report. During the lesson, you saw how we use several metrics to help determine which predicates correlate with bugs. One such metric, $F(P)$ is the number of failing runs in which P is observed to be true. $S(P)$ is the number of successful runs in which P is observed to be true. $\text{Failure}(P)$, calculates how often predicate P is true in failing runs. Another metric, $\text{Context}(P)$, calculates the background chance of failure when predicate P is observed. Finally, $\text{Increase}(P)$ calculates the likelihood that P influences the success or failure of the program.

The function `generateLogFiles()` will be called for you in `CBI.cpp`. This file will iterate through the `.cbi` files generated and populate the lists `SuccessLogs` and `FailureLogs` with the filenames of each. You'll need to iterate through each file in these data structures to calculate these values. Please use these data structures instead of traversing the directory yourself as the grader will use a different directory during grading.

We have defined maps `F`, `S`, `Failure`, `Context`, and `Increase` in `LAB5/include/Utils.h` that you should populate in `generateReport` located in `LAB5/CBI.cpp`. Notice each is a mapping from `std::tuple<int, int, State>` to `double`. Here, the tuple represents a predicate, which consists of a line, column, and a `State` datatype that encodes the possible predicates a branch or return has.

Note that your instrumentation will record where a branch or return occur and its result, but you need to encode that into a predicate. For example, if we encounter `if (p == 10) { ... }` in the code, we need to store two predicates, $(p == 10)$, and $(p \neq 10)$, which you would represent as `State::BranchTrue` and `State::BranchFalse`).

When doing your calculations for $\text{Failure}(P)$, $\text{Context}(P)$, and $\text{Increase}(P)$, if you encounter a divide-by-zero error please enter 0 for the result.

The skeleton code will go through and print out your maps via `printReport`.

Example Input and Output

Your statistical debugger should run on any C code that compiles to LLVM IR. As we demonstrated in the Setup section, we will compile code to LLVM and instrument the code with the fuzzer and `cbi` passes.

Make sure to re-run the `make` command in the `build` directory if you make code changes to your files in the `src` directory so it will pick up the changes.

```

$ cd test
$ clang -emit-llvm -S -fno-discard-value-names -c fuzz1.c -g
$ opt -load ../build/InstrumentPass.so -Instrument -S fuzz1.ll -o
fuzz1.instrumented.ll
$ opt -load ../build/CBIInstrumentPass.so -CBIInstrument -S fuzz1.instrumented.ll
-o fuzz1.cbi.instrumented.ll
$ clang -o fuzz1 ../lib/runtime.c fuzz1.cbi.instrumented.ll

```

The Makefile provided in the test directory will run these commands for all of the test programs at once.

After, we will run the fuzzer to generate a set of passing and failing inputs for use with the cbi tool.

```

$ mkdir fuzz_output
$ timeout 10 ../build/fuzzer ./fuzz1 fuzz_input fuzz_output 10
$ ../build/cbi ./fuzz1 fuzz_output

```

You should expect to generate output similar (but not exactly the same) to the following:

```

== S(P) ==
Line 10, Col 7, BranchTrue: 0
Line 10, Col 7, BranchFalse: 188
Line 14, Col 7, BranchTrue: 5
Line 14, Col 7, BranchFalse: 183
== F(P) ==
Line 10, Col 7, BranchTrue: 129
Line 10, Col 7, BranchFalse: 0
Line 14, Col 7, BranchTrue: 0
Line 14, Col 7, BranchFalse: 0
== Failure(P) ==
Line 10, Col 7, BranchTrue: 1
Line 10, Col 7, BranchFalse: 0
Line 14, Col 7, BranchTrue: 0
Line 14, Col 7, BranchFalse: 0
== Context(P) ==
Line 10, Col 7, BranchTrue: 0.40694
Line 10, Col 7, BranchFalse: 0.40694
Line 14, Col 7, BranchTrue: 0
Line 14, Col 7, BranchFalse: 0
== Increase(P) ==
Line 10, Col 7, BranchTrue: 0.59306
Line 10, Col 7, BranchFalse: -0.40694
Line 14, Col 7, BranchTrue: 0
Line 14, Col 7, BranchFalse: 0

```

How Do We Use It?

[Dr. Liblit](#), the inventor of the [CBI technique](#), [says](#) that CBI “is mostly a research topic, but it is out there in the “real” world in two ways:

1. The public deployment of the [Cooperative Bug Isolation Project](#) hunts for bugs in various Open Source programs running under Fedora Linux. You can download pre-instrumented packages and every time you use them you're feeding us data to help us find bugs.
2. Microsoft has released [Holmes](#), an implementation of statistical debugging for .NET. It's nicely integrated into Visual Studio and should be a very easy way for you to use statistical debugging to help find your own bugs in your own code. I've worked closely with Microsoft Research on Holmes, and these are good smart people who know how to put out high-quality tools.”

While we do not have concrete examples that we can share with the class, CBI is the technique that we teach where we have had the most past students contact us and tell us they are using either the CBI implementation from Dr. Liblit's dissertation or their own variation on the technique. Typically, if they are writing C/C++, they use Dr. Liblit's and if they are using another language, they write their own. After doing this lab, you will be equipped to implement the technique on any code that you can process through the LLVM toolchain.

Grading

Your solution will be tested against the provided test cases as well as some additional hidden test cases that are similar to the provided ones.

Items to Submit

For this lab, we will be using Gradescope to submit and grade your assignment. For full credit, the file must be properly named and there must not be any subfolders.

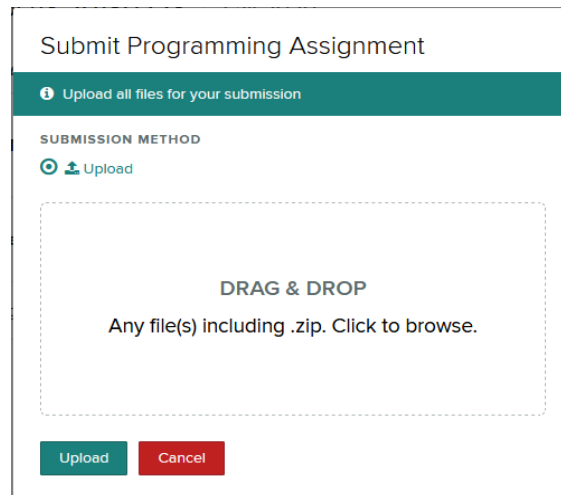
- o Submit file `CBIInstrument.cpp` 40 points
- o Submit file `CBI.cpp` 60 points

Do not submit anything else. Make sure all of your code modifications are in the files listed above as your other files will not be submitted. In particular, past students have modified header files to import additional headers, leading to a compilation error in their submission

Gradescope will have immediate feedback from the autograder as to whether the submission was structured correctly as well as whether your project compiles successfully. As long as Gradescope doesn't report any errors, you can be confident that your project will run and be able to receive a grade. If however, Gradescope reports a compilation error then you will likely receive a 0 on your assignment.

Tips for submitting your assignment with Gradescope

- When you go to the Canvas Assignment, you will see a blank Gradescope submission window embedded in the Canvas Page. Follow the on-screen prompt to submit your files.



Submit Programming Assignment

Upload all files for your submission

SUBMISSION METHOD

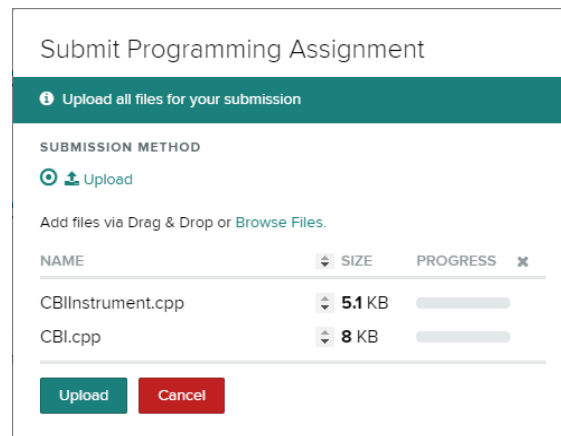
Upload

DRAG & DROP

Any file(s) including .zip. Click to browse.

Upload Cancel

- If there are multiple files in this lab, you can zip the files together and upload the single zip file to the submittal page, or you may select each file individually on the page. If you select the files individually, they must all be uploaded in the same submission.
- A correct submission should look like this image (ignoring size and order):



Submit Programming Assignment

Upload all files for your submission

SUBMISSION METHOD

Upload

Add files via Drag & Drop or Browse Files.

NAME	SIZE	PROGRESS
CBInstrument.cpp	5.1 KB	
CBI.cpp	8 KB	

Upload Cancel

- You may resubmit your assignment as many times as you want up to the assignment due date. The **Active** submission will be the final grade. By default, every time you upload a new submission, it will become the **Active** submission, but you can change the **Active** submission in your **Submission History** window if needed.
- If you resubmit your assignment, you must include all files as part of the new submissions.
- We provide comments as output of your submission. Use these to make sure your submission is as complete as possible. For example, if you see a compile error, you will not receive any points for that part of the assignment.