Lab 5: Cooperative Bug Isolation

Spring Semester 2020

Submission Due: 30 March, 8am ET

Peer Feedback Due: 6 April, 8am ET

Corresponding Lecture: Lesson 9 (Statistical Debugging)

Objective

In this lab, you will use the data collected by a statistical debugging tool, Cooperative Bug Isolation (CBI) to identify and report bugs in Linux applications. CBI exploits the size of user communities to identify bugs in real-world runs of an application by using a random sampling method to instrument application code, collect data about failures, and isolate the cause of a wide variety of bugs.

On a small program and a large program called `jpegtran`, you will do the following things:

1) Run a program multiple times (to simulate real-world runs of the application by the user community)
2) Generate CBI reports using the runs
3) Locate and inspect bugs

Resources

- Cooperative Bug Isolation webpage:
  http://research.cs.wisc.edu/cbi/
- Dr. Ben Liblit's dissertation on CBI - the most complete resource on this tool. **You will almost certainly need to reference this document to successfully complete the lab**
  http://pages.cs.wisc.edu/~liblit/dissertation/
  http://pages.cs.wisc.edu/~liblit/dissertation/dissertation.pdf
- CBI source code on GitHub (Dr. Liblit welcomes pull requests if you have improvements to the tool):
  https://github.com/liblit/cbiexp

Setup

Inside the `~/cbi` directory, run `init.sh` with the following command:

```
cbi $ . init.sh
```

Next, execute the cbi_prep.sh file with the following command (you will likely be prompted for the VM password, which is 'student'):

```
sudo sh cbi_prep.sh
```

Inside the `cbi` folder, you will see the following structure:

```
cbi
 |
 ├── sampler-1.6.2          // Instrumentor
 ├── cbiexp-0.6             // CBI report generator
 ├── large                  // Large application (jpegtran)
 |    └── gen_report        // CBI report generator for jpegtran
 |        ├── analysis      // Final CBI reports
 |        └── bin                 // Scripts for generating CBI reports
 └── small                  // Small application
     └── gen_report         // CBI report generator for small program
         ├── analysis       // Final CBI reports
         └── bin                // Scripts for generating CBI reports
```

Helpful Details on Affinity Lists from Dr. Liblit

Let's distinguish two different kinds of lists. There's the main list of predictors, and then there's one affinity list associated with each predictor from that main list.

For the main list, we do *not* generate the rankings based on the difference between the initial score and the effective score of a predicate. We order the list based on the effective score of each predicate. The “effective score” is best thought of as an estimate of the residual benefit of including that predicate after taking into account all of the other predicates already placed higher on the list.

Now, it turns out that there is a lot of redundancy among predicates. We might easily have dozens or even hundreds of predicates that are exactly or nearly equivalent. As soon as we pick one predicate from a set of near-equivalents, the other predicates in that same set will see their effective score drop dramatically, because of that redundancy. Suppression of duplicates is mostly a good thing: it's not useful to see the same problem reported a hundred times. However, there's no guarantee that the one predicate we picked is actually the one that makes the most sense to a developer.

So this is where the affinity list comes in. It's our way of saying “We picked the next best predicate we could find, but these other predicates are all quite similar to the one we picked, so maybe you should look them over too.” Perhaps the second or third item on the affinity list is easier for the developer to make sense of.

Something else we see in practice is that developers notice (and reason about) broad patterns in each affinity list. If many of the predicates in an affinity list appear on the same line, or in the same function, or mention the same variable, then this is a diffuse sort of clue that can direct the

developer's attention toward suspicious code. Even if the specific details of any one predictor are not informative, the broad patterns sometimes are.

Inside the report, you can access the affinity list by clicking the "Zoom" button.

Part 1: Locating a Bug in a Small Application

Step 1: Instrumentation

First, you should instrument an application. This process is for extracting necessary information for generating CBI reports in the next step. The instrumentor is a driver script that acts as a transparent wrapper around GCC. For the small application, we already provide a `Makefile` that uses the driver script. You can generate instrumented binary by running the following commands:

```
cbi $ cd small
cbi/small $ make
```

You will now have an executable that is compiled with instrumentation and debugging support. By default, we are using the *branches* scheme (explained below). However, you are free to add other schemes by modifying the `Makefile` located in the `small` directory. The following are the possible options.

branches	For each conditional (branch), count how many times the branch predicate is false or true.
returns	At each scalar-returning call site, count how many times the called function returned a negative, zero, or positive value.
scalar-pairs	At each assignment of a scalar value, count how many times the assigned value is less than, equal to, or greater than each other same-typed in-scope variable.
function-entries	Count how many times each function is called.
float-kinds	At each assignment of a floating point value, count how many times the assigned value is in each of possible categories of $\pm\infty$, 0, and NaN.

Later in the lab, you will run this executable to discover test cases that cause the program to crash; you may also run it with a debugger like `gdb`, `clang` or `lldb`.

Step 2: Generating CBI Reports

By running the instrumented application multiple times, you can generate a CBI report. Run the following commands to run the application automatically via a script and then open the generated report in the Firefox web browser.

```
cbi/small $ cd gen_report
cbi/small/gen_report $ make
cbi/small/gen_report $ firefox analysis/summary.xml
```

The script will run the instrumented application multiple times with various inputs in `args.txt` and generate a CBI report. You can provide different inputs by modifying `args.txt`; however the provided inputs are sufficient for finding the expected bug(s). The final CBI report will be displayed after the last command. Your starting point when inspecting these results should be the file `all_hl_corrected-exact-complete.xml`, which can be viewed by clicking the link `exact`.

You may find it helpful to look at the affinity lists of the selected predictors. Recall that the affinity lists show other predicates that are highly correlated with the selected predicates, with the most highly correlated predicates listed first.

Step 3: Locating and Fixing Bugs

You should determine where the bugs represented by the CBI report are located. Your next task is to modify the `test.c` file to fix the bugs found in the CBI report. You should only fix bugs reported by CBI. There may be other bugs in the program, but it's possible that fixing them could have unintended consequences so please only correct any CBI-reported bugs. Grading will be based on the script's output, so make sure to remove any print statements you may have added for debugging before submitting the file.

Once you have corrected the source code, run the `make` command again in the `cbi/small` directory to compile your changes. You should then follow the steps to run your instrumented code and produce a new CBI report to verify that CBI no longer finds any bugs.

When searching for the cause of bugs, you may also find it useful to step through parts of the execution of your test case in a debugger such as `gdb` or `lldb`. However, `gdb` and `lldb` are fairly complex programs, so if you are not already familiar with debugging C programs in a UNIX environment, you may be better off directly examining the source code by hand instead of using a debugger.

Note that a CBI report **will not** be generated if there are not at least two failing inputs, so a completely fixed file will cause report generation to fail with the error `There should be at`

`least two inputs causing a failure`. There are more than two test cases per bug in the provided input files that will cause a failure on the unfixed code.

Part 2: Locating a Bug in a Large Realistic Application

For a realistic program `jpegtran`, you can instrument and generate a CBI report in a similar manner to the small application. By default, we are using the branches, returns, and scalar-pairs schemes. You can change the schemes by properly modifying `cbi/large/conf` and recompiling the program.

```
cbi $ cd large
cbi/large $ ./conf
cbi/large $ make
```

By running the instrumented application multiple times, you can generate a CBI report.

```
cbi/large $ cd gen_report
cbi/large/gen_report $ make
cbi/large/gen_report $ firefox analysis/summary.xml
```

The results in the `analysis` directory represent numerous runs of the CBI instrumented `jpegtran` code using the image file `testimg.jpg` with various command-line options in `options.txt` as input. You can provide different command-line options by modifying `options.txt`. The final CBI report will be displayed after the last command. Your starting point when inspecting these results should be the file `all_hl_corrected-exact-complete.xml`, which can be viewed by clicking the link `exact`.

You should determine where the bugs represented by the CBI report are located. Your next task is to modify the `jpegtran` source to fix the bugs found in the CBI report. Any bugs that CBI reports can be corrected in `transupp.c`, so please make your fixes in this file since there are hundreds of files in the `jpegtran` source and we want to simplify the submission of your fixes. You should only fix bugs reported by CBI. There may be other bugs in the program, but it's possible that fixing them could have unintended consequences so please only correct any CBI-reported bugs. You likely will learn a lot by running the programs directly from the command line, adjusting the options based on what you learn from analyzing the CBI report and the source code.

Once you have corrected the source code, run the `make` command again in the `cbi/large` directory to compile your changes. You should then follow the steps to run your instrumented code and produce a new CBI report to verify that CBI no longer finds any bugs.

Part 3: Writing a Report

Record your responses to the following questions in a PDF file named `report.pdf`. Please note the maximum number of words for each response below. Headers and the like do not count against these limits. There is no specific format requirement for the report, but as you will be reviewing several of your classmates' reports, keep in mind things that you would like to see when reading these reports, like repeating the question text in the report and making sure that each rubric item is clearly answered will be appreciated by your classmates and TAs.

1. What do affinity lists show that enables you to find a bug with them? (150 words)
2. Why might multiple predicates appear on the affinity list for the same bug? (100 words)
3. How could a tool like CBI be used to find bugs in software that you write? Explain the CBI process and how it's useful in the "real world". You may use an example from your professional, academic, or hobby experience or you may make up a hypothetical example. (300 words)
4. Explain the process you used to find and fix bugs based on the CBI report. We encourage you to be specific about how the report influenced your testing and debugging. A good response should include a sample test case (a command that you would run outside of CBI to demonstrate a bug found with CBI). (250 words)

Your grade for the report will follow these criteria:

Criterion	**No credit - Does not meet requirements**	**Half credit - Meets some requirements**	**Full credit - Meets all requirements**
Question 1: Affinity lists (2 points)	Not mentioned	Lacking explanation of using affinity lists to find a bug	Clear explanation of using affinity lists to find a bug
Question 2: Multiple predicates (2 points)	Not mentioned	Discusses predicates, but explanations are unclear or inaccurate	Shows strong understanding of how predicates inform bugs
Question 3: Discussion about using CBI (4 points)	Does not demonstrate understanding of CBI	Demonstrates some understanding of CBI	Has a good understanding of CBI and how to use it effectively
Question 3: Using CBI - picks a relevant example (4 points)	Example is not an appropriate situation to use CBI	Example may be an appropriate situation for CBI, but it does not show the strengths of CBI	Choice of example shows a good understanding of the strengths of CBI
Question 4: Describes finding a bug with CBI (4 points)	Does not demonstrate understanding of CBI	Shows some understanding of how CBI can be used to find and fix a bug	Describes and clearly explains how CBI can be used to find and fix a bug
Question 4: Test case	No test case present or	Test case is close to a	Test case is relevant to

(4 points)	inaccurate test case (0 points	correct test case, but would not trigger the desired bug	one of the sample programs and causes one of the bugs to be hit

Part 4: Peer Feedback

Part of this lab includes reviewing reports written by other students. Our goal in using peer feedback on this lab is that, by reviewing other reports is for, you will gain a variety of perspectives on how to best use a CBI tool.

The day after your report is due, you will be assigned reports to review using the Peer Feedback tool (`https://peerfeedback.gatech.edu`). You will be added to the CS 6340 class in Peer Feedback close to the lab submission deadline, but you do not need to log in to Peer Feedback yet. **We will post on Piazza when we have assigned your peer reviews, at which point you will need to log into Peer Feedback to complete them.**

You will provide feedback by rating your classmates' responses using the eight criteria above and providing any overall comments that you think may be helpful. (Note that your official grade on the written report will be assigned using the same criteria by the TAs, who will have access to your peers' feedback on your lab.)

By default, your classmates can see who wrote each report. If you wish to remain anonymous to your peers, you must take the following steps once you gain access to Peer Feedback but before the report is due. If you wish to be known or do not care, you can ignore these instructions. For anonymity, log in to Peer Feedback and set your display name by clicking on your email address at the top of the screen, then click on `Settings` at the top right corner of the banner image and enter non-identifying information in the `Full Name` box, such as `Anonymous Student`, and then press `Update Profile`. Please also ensure that you do not include your identifying information in the `report.pdf` that you submit. Your graders will still be able to tell who you are so we can still properly assign your grade. Please also note that if you have a profile picture set in Canvas, it will be shown to your reviewers.

Items to Submit by 30 March, 8am ET

We expect your submission to conform to the standards specified below. To ensure that your submission is correct, you should run the provided file validator. You should not expect submissions that have an improper folder structure, incorrect file names, or in other ways do not conform to the specification to score full credit. The file validator will check folder structure and names match what is expected for this lab, but won't (and isn't intended to) catch everything.

The command to run the tool is: `python3 zip_validator.py lab5 lab5.zip`

Submit the following files in a single compressed file (`.zip` format) named `lab5.zip`. For full credit, there must not be any subfolders or extra files contained within your zip file.

1. (25 points) Fixed `test.c` from Part 1
2. (35 points) Fixed `transupp.c` from Part 2

Additionally, submit the following single file as a second attachment to your Canvas submission as a single PDF format file (`.pdf` format) named `report.pdf`. **There are two files being submitted to Canvas for this lab because of how the Peer Feedback system works.**

3. (20 points) `report.pdf` from Part 3

For full credit, both your files MUST be included in the same Canvas submission. You will need to add your ZIP file, click the `Add Another File` link, and then add your PDF file to include both files in the submission. Submitting a ZIP file with the PDF inside or including one of the files on a comment will prevent your report from making it into the Peer Feedback system and will result in a grade penalty.

Make sure the spelling of the filenames and the extensions are exactly as above, or else it could break the grading script and cause you to incur a deduction from your grade. Make sure you have not added any additional print statements to your `test.c` file (the ones there originally must remain).

Items to Submit by 6 April, 8am ET

1. (20 points) Feedback for your classmates through Peer Feedback as detailed in Part 4.

The feedback you provide to other students must be submitted by the deadline, and must demonstrate effort on your part. In addition to assessing student responses according to the criteria, you must provide feedback in comments section. While we will not be grading these comments based on accuracy, we expect that your comments are constructive, and show effort on your part to participate. Full credit will not be given to submissions with blank, trivially short, or reused feedback.