Lab 5: Statistical Debugging

Synopsis

Building a statistical debugger for remote program monitoring and debugging.

Objective

In this lab, you will implement cooperative bug isolation (CBI) to statistically localize error locations in a program. You need to implement an LLVM pass that instruments each branch and function call to report the values of their conditions and return values. You will then generate a number of sample inputs using a fuzzer. With these inputs, you will obtain the runtime data from the instrumentation and measure different types of scores that help isolate bugs and guide debugging effort.

Setup

The skeleton code for Lab 5 is located under `cis547vm/lab5/`. We will frequently refer to the top level directory for Lab 5 as `lab5` when describing file locations for the lab. Open the `lab5` directory in VSCode following the Instructions from Course VM document.

The lab in split into three parts:

1. In part 1, you will implement two instrumentations, one to instrument each conditional branch instruction reporting the condition. And another that instruments all function calls, to report their return values.
2. In part 2, you will run a fuzzer to generate a set of inputs that will be used to collect data for performing statistical analysis.
3. In part 3, you will run the target program with the inputs generated by the fuzzer to collect cbi logs and analyze the data to report various metrics.

To build the instrumentation, you can use the following commands:

```
lab5$ mkdir build && cd build
lab5/build$ cmake ..
lab5/build$ make
```

You should now see `CBIInstrumentPass.so`, `libruntime.so`, and `InstrumentPass.so` in the build 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 program runs that pass or fail, you will use a `fuzzer`:

```
lab5$ cd test
lab5/test$ make
lab5/test$ rm -rf fuzz_output_sanity && mkdir fuzz_output_sanity
lab5/test$ timeout 1 fuzzer ./sanity fuzz_input fuzz_output_sanity 10
```

Or alternatively you can have the `Makefile` to do this for you using:

```
lab5/test$ make fuzz-sanity
```

To setup the `cbi` tool you can run:

```
lab5$ make install
```

This will install the `cbi` tool, just like what you had in `lab4` for the `delta-debugger` tool.

You can then run `cbi` to generate a report of analyzing the runs of a program using:

```
lab5/test$ cbi ./sanity ./fuzz_output_sanity
```

Running `cbi` after completing `CBIInstrument.cpp` but before completing `cbi.py` should give you the following report:

```
== S(P) ==

== F(P) ==

== Failure(P) ==

== Context(P) ==

== Increase(P) ==
```

Lab Instructions

In short, the lab consists of the following tasks:

1. Implement the `instrumentBranch` function to insert a `__cbi_branch__` call for a predicate (conditional).
2. Modify `runOnFunction` to instrument all conditional branching instructions using `instrumentBranch`.
3. Implement the `instrumentReturn` function to insert a `__cbi_return__` call to log the return value of function calls.
4. Modify `runOnFunction` to instrument all *call* instructions that return an integer usinf `instrumentReturn`.
5. Using the feedback profile you construct via 1 through 4, modify `cbi/data_format.py` to implement formula for calculating: `Failure(P)`, `Context(P)`, and `Increase(P)` in the `PredicateInfo` class.
6. Modify the `cbi/cbi.py` file using the instruction provided in the file, and compute `S(P)`, `SObs(P)`, `F(P)`, and `FObs(P)` which should be stored in the corresponding data structure(`PredicateInfo`).

Part 1: Instrumentation

In this part, you will edit the `lab5/src/CBIInstrument.cpp` file to instrument branches and return instructions with code to extract and monitor predicates.

`lab5/lib/runtime.c` contains functions that you will be instrumenting:

```
void __cbi_branch__(int line, int col, int cond);

void __cbi_return__(int line, int col, int rv);
```

These functions appends corresponding predicate information as the following json objects respectively:

```
{"kind": "branch", "line": line, "col": col, "value": true/false}
{"kind": "return", "line": line, "col": col, "value": rv}
```

to the running process's cbi log file (`target.cbi.jsonl`).

Like you did in Lab 2, your LLVM pass should instrument the code with these functions. Your pass should instrument each conditional branch with code that records whether the branch conditional is `true` or `false` on execution. Likewise, instrument each call instruction that returns an integer to record its return value at runtime. The log file generated by these instrumentations will create a *feedback profile* for you to perform statistical debugging and generate a *feedback report*.

Revisiting Instrumentation.

By now you should feel comfortable working with the LLVM compiler infrastructure, but for a refresher, consult Lab 2 and see the paragraphs titled "Inserting Instruction into LLVM Code" and "Loading C functions into LLVM".

Part 2: Fuzzer

Once you have implemented the instrumentation, you will run the fuzzer to generate a number of passing as well as failing inputs to a target program.

This is done identically as in lab 3 and lab 4 using:

```
lab5/test$ make fuzz-sanity
```

Depending on your machine, you may need to increase the timeout in the `test/Makefile` to allow for the fuzzer to run for a long enough time to generate a reasonable number of inputs.

Part 3: CBI

The `cbi` program will execute the target program on each of the `input` files from the `fuzzer` output directory. This includes both successful program runs as well as erroneous program runs. Each run will generate a corresponding feedback profile as a `.cbi.jsonl` file. The resulting directory tree will look like this:

```
fuzz_output_sanity1
├── success
│   ├── input0
│   ├── input0.cbi.jsonl
│   └── ...
└── failure
    ├── input0
    ├── input0.cbi.jsonl
    ├── input1
    ├── input1.cbi.jsonl
    └── ...
```

The `cbi` program then parses each of the `.cbi.jsonl` file into a list of `CBILogEntry` objects, and passes it to the `cbi` function in `cbi/cbi.py` where you will generate the 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, `Failure(P)`, calculates how often predicate `P` is true in a failing run. Another metric, `Context(P)`, measures the background chance of failure when predicate `P` is observed. Finally, `Increase(P)` calculates the likelihood that `P` influences the success or failure of the program.

You will first need to implement the formula for `Failure(P)`, `Context(P)`, and `Increase(P)` in the `PredicateInfo` class.

To help you get started, here's what you will be filling in for `Increase(P)`:

```
class PredicateInfo:
  ...
  @property
  def increase(self):
      """
      The Increase value of the predicate.

      :return: The increase value.
      """
      # TODO: Implement the calculation of the increase value.
      return self.failure - self.context
  ...
```

The `PredicateInfo` class is a simple data structure that holds the all the requires information for a predicate.

In the `cbi.py` file, you will go over all the CBILogs and for each predicate you will populate its corresponding `PredicateInfo` object with the information required for the report.

This includes:

- `s`: The number of successful runs where the predicate was true at least once.
- `s_obs`: The number of successful runs where the predicate was either true or false at least once, in a.
- `f`: The number of failing runs where the predicate was true at least once.
- `f_obs`: The number of failing runs where the predicate was either true or false at least once.

We have provided you with a few function declarations in `cbi/cbi.py` that may help you break down this task into more reasonable chunks.

It will also be helpful to look at the `cbi/data_format.py` file for the data format that you will be using and the available convenience functions.

In particular you have:

```
class CBILogEntry:
  """
  Data class for a single entry in a CBI log.

  :param kind: The kind of entry [branch/return].
  :param line: The line number specified in the log.
  :param column: The column number specified in the log.
  :param value: The value specified in the log.
  """
  ...

class ObservationStatus(Enum):
  """
  Enum for the Observation Status of a predicate.

  :param NEVER: The predicate was never observed.
  :param ONLY_TRUE: The predicate was observed and was always true.
  :param ONLY_FALSE: The predicate was observed and was always false.
  :param BOTH: The predicate was observed at least once as true and false.
  """
  ...

class Predicate:
  """
  Data class for a predicate.

  :param line: The line number of the predicate in the program.
  :param column: The column number of the predicate in the program.
  :param pred_type: The type of the predicate.
    One of the predicate types from PredicateType.
  """
  ...

class Report:
  """
  Data class for the CBI Report.

  :param predicates_info_list: A list of PredicateInfo objects of
    all predicates in the program.
  """
  ...
```

Note: Your instrumentation will record where a branch or return occurs 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) is true`, and `(p == 10) is false`, which you would represent as `BranchTrue` and `BranchFalse`. You may find the `PredicateType.alternatives` function helpful for this.

Once you populate the `predicate_infos` dictionary in the `cbi` function, the skeleton code will go through it and print out the report as well as store a detailed report to `target.report.json` file.

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.

```
lab5$ cd test
lab5/test$ make test2
```

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

```
lab5/test$ make fuzz-test2
lab5/test$ cbi ./test2 fuzz_output_test2
```

You should expect to generate output similar to the following:

```
== S(P) ==
Line 010, Col 007,    BranchFalse: 68.0
Line 010, Col 007,     BranchTrue: 8
Line 014, Col 007,    BranchFalse: 65.0
Line 014, Col 007,     BranchTrue: 3.0
== F(P) ==
Line 010, Col 007,    BranchFalse: 0
Line 010, Col 007,     BranchTrue: 145.0
Line 014, Col 007,    BranchFalse: 0
Line 014, Col 007,     BranchTrue: 0
== Failure(P) ==
Line 010, Col 007,    BranchFalse: 0
Line 010, Col 007,     BranchTrue: 1.0
Line 014, Col 007,    BranchFalse: 0
Line 014, Col 007,     BranchTrue: 0
== Context(P) ==
Line 010, Col 007,    BranchFalse: 0.6807511737089202
Line 010, Col 007,     BranchTrue: 0.6807511737089202
Line 014, Col 007,    BranchFalse: 0.0
Line 014, Col 007,     BranchTrue: 0.0
== Increase(P) ==
Line 010, Col 007,    BranchFalse: -0.6807511737089202
Line 010, Col 007,     BranchTrue: 0.3192488262910798
Line 014, Col 007,    BranchFalse: 0.0
Line 014, Col 007,     BranchTrue: 0.0
```

You can download the input files we used to generate the report above from here.

Submission

Once you are done with the lab, you can create a `submission.zip` file by using the following command:

```
lab5$ make submit
...
submission.zip created successfully.
```

Then upload the submission file to Gradescope.