Lab 8: Constraint-Based Analysis

Synopsis

Writing a constraint-based static analysis for C programs with LLVM and Datalog.

Objective

In this lab, you will implement a constraint-based analysis to detect exploitable divide-by-zero bugs. A bug is exploitable if hackers can control inputs or environments, thereby triggering unintended behaviors (e.g., denial-of-service) through the bug. For example, a recently reported divide-by-zero bug in the Linux kernel can be exploitable and crash the system; another example was the Log4Shell incident. You will design a static analysis that detects such bugs by combining reaching definition analysis and taint analysis using a datogon engine, souffle.

Setup

The skeleton code for Lab8 is located under `cis547/lab8/`. We will frequently refer to the top level directory for Lab 8 as `lab8` when describing file locations for the lab. Open the `lab8` directory in VSCode following the Instructions from Course VM document

The following commands set up the lab, using the Cmake/Makefile pattern seen before.

```
/lab8$ mkdir build && cd build
/lab8/build$ cmake ..
/lab8/build$ make
```

The above command will generate an executable file 'constraint' in build directory that extracts facts about the program that will be used by `src/analysis.dl` to check if the input program has an exploitable divide-by-zero bug:

```
/lab8$ cd ./test
/lab8/test$ clang -emit-llvm -S -fno-discard-value-names -c simple0.c
/lab8/test$ mkdir -p simple0/input simple0/output
/lab8/test$ ../build/constraint simple0.ll ./simple0/input
/lab8/test$ souffle -F ./simple0/input -D ./simple0/output ../src/analysis.d
```

After completing the lab implementation, you should see the instruction that could potentially cause a divide-by-zero in `./simple0/output/alarm.csv`.

Lab Instructions

In this lab, you will design a reaching definition analysis and taint analysis using datalog.

The main tasks are to design the analysis in the form of logical rules as a Datalog program, and implement a function that extracts logical relations form a test program in the form of Datalog facts for each LLVM instruction.

We will then use the datalog rules with the facts you extracted from each instruction in a program to find any *exploitable* divide-by-zero errors.

In short, the lab consists of the following tasks:

1. Write Datalog rules in `analysis.dl` for taint analysis.
2. Implement the `extractContraints` function in `Extractor.cpp` that extracts Datalog facts from LLVM IR `Instruction` and dumps them to appropriate `.facts` files. Detailed instructions provided in comments.

Relations for Datalog Analysis

The skeleton code in `Extractor.cpp` provides the definitions of various functions that will help you dump the necessary Datalog relations declared in `src/analysis.dl` file.

Here we will explain what each of the datalog relation mean.

The relations for def and use of variables are as follows:

- `def(var, inst)`: Variable `var` is defined at instruction `inst`.
- `use(var, inst)`: Variable `var` is used in instruction `inst`.

The relations for the reaching definition analysis are as follows:

- `kill(curr_inst, old_inst)`: Instruction `curr_inst` kills definition at instruction `old_inst`.
- `next(curr_inst, next_inst)`: Instruction `next_inst` is an immediate successor of instruction `curr_inst`.
- `in(inst, def_inst)`: Definition at defining instruction `def_inst` may reach the program point just before instruction `inst`.
- `out(inst, def_inst)`: Definition at defining instruction `def_inst` may reach the program point just after instruction `inst`.

Note that the `kill` relation can be derived by using the `def` relation by writing a Datalog rule.

The relations for the taint analysis are as follows:

- `taint(inst)` : There exists a function call at intruction `inst` that reads a tainted input.
- `edge(from, to)`: There exists an immediate data-flow from instruction `from` to instruction `to`.
- `Path(from, to)`: There exists a transitive tainted data-flow from instruction `from` to instruction `to`.
- `sanitizer(inst)` : There exists a function call at intruction `inst` that sanitizes a tainted input.
- `div(denom, inst)` : There exists a division operation at instruction inst whose divisor is variable denom.
- `alarm(inst)` : There exists a potential exploitable divide-by-zero error at instruction inst.

Assume that input programs may contain function calls to `tainted_input` and `sanitizer` that read and sanitize a tainted input, respectively. The final output relation for potential bug reports is `alarm`.

You will use these relations to build rules for the definition analysis and taint analysis in `analysis.dl`.

Encoding LLVM Instruction in Datalog

Recall that, in LLVM IR, values and instructions are interchangable. Therefore, all variables X, Y, and Z are an instance of LLVM's `Value` class.

Assume that input C programs do not have pointer variables. Therefore, we abuse pointer variables in LLVM IR as their dereference expressions. Consider the following simplified LLVM program from a simple C program `int x = 0; int y = x;`:

```
x = alloca i32          ; I0
y = alloca i32          ; I1
store i32 0, i32* x     ; I2
a = load i32, i32* x    ; I3
store i32 a, i32* y     ; I4
```

We ignore alloca instructions and consider that each store instruction defines the second argument.

In the case of the above example, you should have `addDef(I0,I2)`, because `I0` corresponds to `x` in LLVM IR and `I2` defines the variable `x`. Likewise, consider each load instruction uses the argument. In the example, you should have `addUse(I0,I3)` and `addDef(I3,I3)` because load instructions define a non-pointer variable which is represented as the instruction itself in LLVM. Finally, you should have `addUse(I3,I4)` and `addDef(I1,I4)` for instruction `I4`.

Defining Datalog Rules

You will write your Datalog rules for taint analysis in `src/analysis.dl` using the relations above.

Please refer to the souffle's documentation for the datalog language and its syntax.

Here's a quick example to help you get started.

Consider an example Datalog rule:

```
A(X, Y) :- B(X, Z), C(Z, Y).
```

It adds a rule which says that there is a relation `A` between `X` and `Y` if, there is a relation `B` between `X` and some `Z` and there is also a relation `C` between that `Z` and `Y`.

Extracting Datalog Facts

You will need to implement the function `extractConstraints` in `Extractor.cpp` to extract Datalog facts for each LLVM instruction. The skeleton code provides a couple of auxiliary functions in `lab8/src/Extract.cpp` and `lab8/src/Utils.cpp` help you with this task:

- `void addX(const InstMapTy &InstMap, ... )`
  - `X` denotes the name of a relation. These functions add a fact to the facts file for `X`.
  - It takes `InstMap` that encodes each LLVM instruction as an integer. This map is initialized in the `main` function.
- `vector<Instruction*> getPredecessors(Instruction *I)`
  - Returns a set of predecessors of a given LLVM instruction `I`.
- `bool isTaintedInput(CallInst *CI)`
  - Checks whether a given LLVM call instruction `CI` reads a tainted input or not.
- `bool isSanitizer(CallInst *CI)`
  - Checks whether a given LLVM call instruction `CI` sanitizes a tainted input or not.

Miscellaneous

For debugging, after you run `constraint` you can inspect the `./test/simple0/input` for files containing the extracted relations.

Format of Input Programs

Input programs in this lab are assumed to have only sub-features of the C language as follows:

- All values are integers (i.e., no floating points, pointers, structures, enums, arrays, etc). You can ignore other types of values.
- Assume that there is no function call to a function with a void return type.
- You must handle the function calls to `tainted_input` and `sanitizer` in a special way which represents their actions as described previously.

Example Input and Output

Your analyzer should run on LLVM IR. For example:

```
/lab8$ cd ./test
/lab8/test$ make loop0
```

If the input program has exploitable divide-by-zero errors, then you should see entries in `loop0/output/alarm.csv`.

```
%div
```

Submission

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

```
lab8$ make submit
...
submission.zip created successfully.
```