Assignment 1– LLVM Playground (Part 2)

HSS
Fall 2024

In this part, you goal is to develop a simple data flow analysis to detect "divide-by-zero" errors in C code.
Logistics:

- **LLVM Primer:** Please make sure that you have skimmed the LLVM Primer presentation (access it from the course webpage) to know the capabilities of LLVM.
- **Setup Repo:** I have created a `github` repo with all the necessary scripts to install LLVM, Z3 and starter code to write a pass. You can access it at: `https://github.com/HolisticSoftwareSecurity/hssllvmsetup`. The repo has examples of analysis (i.e., the passes that do not modify the IR) and instrumentation (i.e., the passes that modify the IR) passes.
- **Development Environment:** I use CLion (`https://www.jetbrains.com/clion/`) while working with LLVM and strongly suggest you to use it. You can get unlimited access using your `@purdue.edu` email.

Setup

The skeleton code for this part is located under `part2_basic_data_flow_analysis` folder of the following repo. We will frequently refer to this top level directory as `dfa` when describing file locations for the lab.

Repo

`https://github.com/HolisticSoftwareSecurity/LLVMPlayground`

Step 1

Clone the above repository to a folder:

```
$ cd
$ git clone https://github.com/HolisticSoftwareSecurity/LLVMPlayground.git
remote: Enumerating objects: 19, done.
remote: Counting objects: 100% (19/19), done.
remote: Compressing objects: 100% (6/6), done.
remote: Total 14 (delta 11), reused 11 (delta 8), pack-reused 0
Unpacking objects: 100% (14/14), done.
```

Step 2

Build the pass using the `CMakeLists.txt` as shown below:

```
$ cd ~/LLVMPlayground/part2_basic_data_flow_analysis
$ mkdir build && cd build
$ cmake -DUSE_REFERENCE=ON ..
$ make
```

Among the files generated, you should now see `DataflowPass.so` in the `build/DivZero` directory. `DataflowPass.so` is built from `DivZero/src/DivZeroAnalysis.cpp` which you will modify shortly. We are now ready to run our bare-bones lab on a sample input C program. One thing to note is the use of the `-DUSE_REFERENCE=ON` flag: this lab comprises two parts and this flag will allow you to focus on the features needed for Part 1 independently of Part 2.

Step 3

Before running the pass, the LLVM IR code must be generated:

```
$ cd ~/LLVMPlayground/part2_basic_data_flow_analysis/DivZero/test
$ clang -emit-llvm -S -fno-discard-value-names -Xclang -disable-O0-optnone -c simple1.c
$ opt -mem2reg -S simple1.ll -o simple1.opt.ll
```

The second line (clang) generates vanilla LLVM IR code from the input C program simple1.c. The last line (opt) optimizes the vanilla code and generates an equivalent LLVM IR program that is simpler to process for the analyzer you will build in this lab; in particular,`-mem2reg` promotes every AllocaInst [1] to a register, allowing your analyzer to ignore handling pointers in this lab. You will extend this to handle pointers in the next part.

Step 4

you will implement your analyzer as an LLVM pass, called `DataflowPass`. Use the `opt` command to run this pass on the optimized LLVM IR program as follows:

```
DivZero/test $ opt -load ../../build/DivZero/libDataflowPass.so -DivZero -disable-output simple1.opt.ll
```

Upon successful completion of this lab, the output should be as follows:

```
Running DivZero on main
Potential Instructions by DivZero:
  %div1 = sdiv i32 %div, %div
```

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.
- You should handle assignments, signed and unsigned arithmetic operations (+, -, *, /), comparison operations (<, <=, >, >=, ==, !=), and branches. All the other instructions are considered to be nop.
- Input programs can have if-statements and loops.
- Assume that user inputs are only introduced via the `getchar` library function. The skeleton code provides an auxiliary function `isInput` that checks whether a given instruction is a function call to `getchar`. You can ignore other call instructions to other functions.

[1] `https://llvm.org/doxygen/classllvm_1_1AllocaInst.html`

Lab Instructions

A full-fledged static analyzer has three components:

1. an abstract domain
2. abstract transfer functions for individual instructions, and
3. combining analysis results of individual instructions to obtain analysis results for entire functions or programs.

In this lab, we will focus only on implementing (ii), and only for a limited subset of instructions as described above. More concretely, your task is to implement how the analysis evaluates different LLVM IR instructions on abstract values from a provided abstract domain.

We have provided a framework to build your division-by-zero static analyzer. The framework is composed of files `Domain.cpp`, `DataflowAnalysis.cpp`, and `DivZeroAnalysis.cpp` under `DivZero/src/` `DivZeroAnalysis` is a class that extends `DataflowAnalysis`.

Part 1: Transfer Functions

Step 1

Refresh your understanding about program abstractions by reading the article on A Menagerie of Program Abstractions [2] [3].

Once you have a good understanding of abstract domains, study the Domain class to understand the abstract domain that we have defined for you to use in this lab. The files `DivZero/include/Domain.h` and `DivZero/src/Domain.cpp` include the abstract values and operations on them. These operations will perform an abstract evaluation without running the program. As described in the article, we have defined abstract operators for addition, subtraction, multiplication and division.

Step 2

Inspect `DataflowAnalysis::runOnFunction` to understand how, at a high-level, the compiler pass performs the analysis:

```
bool DataflowAnalysis::runOnFunction(Function &F) {
  outs() << "Running " << getAnalysisName() << " on " << F.getName() << "\n"
  for (inst_iterator I = inst_begin(F), E = inst_end(F); I != E; ++I) {
        InMap[&(*I)] = new Memory;
        OutMap[&(*I)] = new Memory;
  }

  doAnalysis(F);
  collectErrorInsts(F);
  ...
}
```

The procedure `runOnFunction` is called for each function in the input C program that the compiler encounters during a pass. Each instruction I is used as the key to initialize a new `Memory` object in the global `InMap` and `OutMap` hash maps. These maps are described in more detail in

[2] https://purs3lab.github.io/hss/static_files/notes/galiosconnection.pdf
[3] https://www.seas.upenn.edu/~edinella/blog-posts/programabstractions/

the next step, but for now you can think of them as storing the abstract values of each variable before and after an instruction. For example, the abstract state might store facts like "at the point before instruction `i`, the variable `x` is positive." Since `InMap` and `OutMap` are global, feel free to use them directly in your code.

Once the In and Out Maps are initialized, `runOnFunction` calls `doAnalysis`: a function that you will implement in Part 2 to perform the chaotic iteration algorithm. For Part 1, you can assume that it simply calls transfer using the appropriate InMap and OutMap maps.

So, at a high level, `runOnFunction` will:

1. Initialize the in and out maps
2. Fill them using a chaotic iteration algorithm
3. Find potential divide by zero errors by using the `InMap` entries for each divide instruction to check whether the divisor may be zero.

Step 3

Understand the memory abstraction in the provided framework. For each `Instruction`, `DivZeroAnalysis::InMap` and `DivZeroAnalysis::OutMap` store the abstract state before and after the instruction, respectively. An abstract state is a mapping from LLVM variables to abstract values; in particular, we have defined `Memory` as a `std::map<std::string, Domain *>`. Since we refer to variables as `std::string`, we have provided an auxiliary function named variable that encodes an LLVM `Value` into our internal string representation for variables. Note that an `Instruction` is also a `Value`. For example, consider the following LLVM program. We have shown the abstract state, denoted M, before and after each instruction:

Instruction		Before Instruction	After Instruction
I1	`%x = call i32 (...) @input()`	{}	{%x: ⊤}
I2	`%y = add i32 %x, 1`	{%x: ⊤}	{%x: ⊤, %y: %x ⊕ 1}

In the first instruction `I1`, we assign an input integer to variable `%x`. In the abstract state, we use an abstract value ⊤(also known as "top" or MaybeZero) since the value is unknown at compile time. Instruction `I2` updates the abstract value of variable `%y` that is computed using the abstract add operation (denoted ⊕) on the abstract value of `%x`. Note that, in the LLVM framework, the object for an assignment instruction (e.g., `call`, binary operator, `icmp`, etc.) also represents the variable it defines (i.e. its left-hand side). Therefore you will use the objects for instructions `I1` and `I2` to refer to variables `%x` and `%y`, respectively, in your implementation. For example, `variable(I1)` will refer to `%x`.

Step 4

Now that we understand how the pass performs the analysis and how we will store each abstract state, we can begin implementation. First, you will implement a function `DivZeroAnalysis::transfer` to populate the `outMap` for each instruction. In particular, given an instruction and its incoming abstract state (`const Memory *In`), `transfer` should populate the outgoing abstract state (`Memory *NOut`).

The `Instruction` class represents the parent class of all types of instructions. There are many subclasses [4] of `Instruction`. In order to populate the `outMap`, each type of instruction should be handled differently.

[4] https://releases.llvm.org/8.0.0/docs/ProgrammersManual.html#the-instruction-class

Recall for this lab you should handle:

- `BinaryOperators (add, mul, sub, etc.)`[5]
- `CastInst`[6]
- `CmpInst (icmp eq, ne, slt, sgt, sge, etc.)`[7]
- `BranchInst`[8]
- user input via `getchar()` - recall from above that this is handled using `isInput()` from `DivZero/include/DataflowAnalysis.h`

LLVM provides several template functions[9] to check the type of an instruction. We will focus on `dyn_cast<>` for now. In this example, we check if the `Instruction I` is a `BinaryOperator`:

```
if (BinaryOperator *BO = dyn_cast<BinaryOperator>(I)) {
  // I is a BinaryOperator, do something
}
```

At runtime, `dyn_cast` will return `I` casted to a `BinaryOperator` if possible, and null otherwise.

Working with LLVM PHI Nodes. For optimization purposes, compilers often implement their intermediate representation in *static single assignment (SSA)* form and LLVM IR is no different. In SSA form, a variable is assigned and updated at exactly one code point. If a variable in the source code has multiple assignments, these assignments are split into separate variables in the LLVM IR and then merged back together. We call this merge point a phi node.

To illustrate phi nodes, consider the following code:

```
int f() {
  int y = input();
  int x = 0;
  if (y < 1) {
    x++;
  } else {
    x--;
  }
  return x;
}
```

Depending on the value of `y`, we either take the left branch and execute `x++`, or the right branch and execute `x--`.

In the corresponding LLVM IR (below):

```
entry:
  %call = call i32 (...) @input()
  %cmp = icmp slt i32 %call, 1
  br i1 %cmp, label %then, label %else

then:                                             ; preds = %entry
  %inc = add nsw i32 0, 1
  br label %if.end

else:                                             ; preds = %entry
```

[5] https://llvm.org/doxygen/classllvm_1_1BinaryOperator.html
[6] https://llvm.org/doxygen/classllvm_1_1CastInst.html
[7] https://llvm.org/doxygen/classllvm_1_1CmpInst.html
[8] https://llvm.org/doxygen/classllvm_1_1BranchInst.html
[9] https://releases.llvm.org/8.0.0/docs/ProgrammersManual.html#the-isa-cast-and-dyn-cast-templates

```
  %dec = add nsw i32 0, -1
  br label %end

end:                              ; preds = %else, %then
  %x = phi i32 [ %inc, %then ], [ %dec, %else ]
  ret i32 %x
}
```

The update on `x` is split into two variables `%inc` and `%%dec`. `%x` is assigned after the branch executes with the phi instruction; abstractly, `phi i32 [ %inc, %then ], [ %dec, %else ]` says assign `%inc` to `%x` if the then branch is taken, or `%dec` to `%x` if the else branch was taken.

Here is a piece of sample code to help you address `phi` nodes, as the specifics are beyond this course; however, feel free to read up more on SSA if these kind of compiler details pique your interest.

```
Domain *evalPhiNode(PHINode *PHI, const Memory *Mem) {
  Value* cv = PHI->hasConstantValue();
  if (cv){
    // eval cv, manipulate Mem, return
  }
  unsigned int n = PHI->getNumIncomingValues();
  Domain* joined = NULL;
  for (unsigned int i = 0; i < n; i++){
    Domain* V = // eval PHI->getIncomingValue(i), manipulate Mem
    if (!joined){
      joined = V;
    }
    joined = Domain::join(joined, V);
  }
  return joined;
}
```

Step 5

Implement a function `DivZeroAnalysis::check` to check if a specific instruction can incur a divide-by-zero error. You should use `DivZeroAnalysis::InMap` to decide if there is an error or not.

Part 2: Putting it all together - dataflow analysis

Now that you have code to populate in and out maps and use them to check for divide by zero errors, your next step is to implement the chaotic iteration algorithm in function doAnalysis.

First, review the dataflow analysis lecture content. In particular, study the reaching definition analysis and the chaotic iteration algorithm.

Informally, a dataflow analysis creates and populates an **IN** set and an **OUT** set for each node in the program's control flow graph. The *flowIn* and *flowOut* operations are repeated until the algorithm has reached a fixed point.

More formally, the doAnalysis function should maintain a WorkSet that holds nodes that "need more work." When the WorkSet is empty, the algorithm has reached a fixed point. For each instruction in the WorkSet your function should do the following:

1. Perform the flowIn operation by joining all OUT sets of incoming flows and saving the result in the IN set for the current instruction. Here, you will use the entries from the `InMap` and `OutMap` that you populated in Part 1 as the **IN** and **OUT** sets.

2. Apply the `transfer` function that you implemented in Part 1 to populate the **OUT** set for the current instruction.

3. Perform the flowOut operation by updating the `WorkSet` accordingly. The current instruction's successors should be added only if the **OUT** set was changed by the `transfer` function.

We have sketched out the start of this process for you by initializing the `WorkSet` with each instruction in the input C program:

```
void DivZeroAnalysis::doAnalysis(Function &F) {
  SetVector<Instruction *> WorkSet;
  for (inst_iterator I = inst_begin(F), E = inst_end(F); I != E; ++I) {
    WorkSet.insert(&(*I));
  }
  // ...
```

For this lab, we do not need to maintain an explicit control flow graph; LLVM already maintains one in its internals. In order for you to focus on the dataflow portion of this assignment, we have provided two auxiliary functions `getSuccessors` and `getPredecessors` (defined in `DivZero/include/DataflowAnalysis.h`) that lookup and return the successors and predecessors for a given LLVM Instruction.

First, uncomment the functions marked under PART 2, namely `doAnalysis`, `flowIn`, `flowOut`, `join` and `equal`. After doing so, you will implement each part of the algorithm detailed above in the following steps:

Step 1

In `flowIn`, you will perform the first step of the reaching definitions analysis by taking the union of all **OUT** variables from all predecessors of I. You may find the `getPredecessors` method in `DivZero/include/DataflowAnalysis.h` to be helpful here. This should be done in the following function that is templated for you below:

```
void DivZeroAnalysis::flowIn(Instruction *I, Memory *In)
```

Given an `Instruction I` and its **IN** set of variables `Memory In`, you will need to union the **IN** with the **OUT** of every predecessor of `I`. In order to take the union of two memory states, you will need to implement the `join` function templated below:

```
Memory* DivZeroAnalysis::join (Memory *M1, Memory *M2)
```

Within this function, you will also need to consider the `Domain` values when merging these `Memory` objects. Refer to the abstract domain on why this is necessary. Recall that a `join` operation for combining two abstract values is defined in the `Domain` class.

Step 2

Call the `transfer` function that you implemented in Part 1 to populate the **OUT** set for the current instruction.

Step 3

In `flowOut`, you will determine whether or not a given instruction needs to be analyzed again. This should be done in the following function that is templated for you below:

```
void DivZeroAnalysis::flowOut(Instruction *I, Memory *Pre, Memory *Post,
                              SetVector<Instruction *> &WorkSet)
```

Given an `Instruction I`, you will analyze the pre-transfer memory `Pre` and the post-transfer memory `Post`. If there exists a change between the memory values after the transfer is applied, you will need to submit the instruction `I` for additional analysis. To determine if the memory has changed during the transfer function, you will implement the function equal:

```
bool DivZeroAnalysis::equal(Memory *M1, Memory *M2)
```

In this function, you will again consider the `Domain` values when determining whether two `Memory` objects are equal. Recall that an `equal` operation to evaluate equality between two abstract values is defined in the `Domain` class.

Lastly, in `flowOut` be sure that you update the `OutMap` for instruction `I` to include values in `Post`.

Step 4

Recall in Part 1, a reference `doAnalysis` could be used to verify your `check` and `transfer` implementations. Now that you're writing your own version of `doAnalysis`, you may need to rebuild the pass without the reference. First, make sure that the `doAnalysis` function in `DivZeroAnalysis.cpp` is not commented out. Next, follow these steps to compile using your implementation:

```
$ cd ~/LLVMPlayground/part2_basic_data_flow_analysis
$ cd build
$ cmake ..
$ make
```

Submission

Submit your modified file `DivZeroAnalysis.cpp`.