Assignment 1– LLVM Playground (Part 3)

HSS
Fall 2024

In this part, you will improve upon your previous “division-by-zero” static analysis to handle pointer aliasing and allocated memory..
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 `part3_pointer_aware_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/part3_pointer_aware_data_flow_analysis
$ mkdir build && cd build
$ cmake ..
$ make
```

Among the files generated, you should now see `DataflowPass.so` in the `build/DivZero` directory, similar to the previous lab. In this lab you will modify `DivZeroAnalysis.cpp`. We are now ready to run our bare-bones lab on a sample input C program.

Step 3

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

```
$ cd ~/LLVMPlayground/part3_pointer_aware_data_flow_analysis/DivZero/test
$ clang -emit-llvm -S -fno-discard-value-names -Xclang -disable-O0-optnone -c pointer0.c -o pointer0.opt.ll
$ opt -load ../../build/DivZero/libDataflowPass.so -DivZero pointer0.opt.ll
```

The second line (clang) generates vanilla LLVM IR code from the input C program simple1.c. The last line (opt) runs the pass over the compiled LLVM assembly code. In prior labs, we used an intermediate step with the argument `-mem2reg` which promoted every AllocaInst [1] to a register, allowing your analyzer to ignore handling pointers in this lab. However, this is no longer needed because you will extend your previous code to handle pointers.

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

```
Running DivZero on f
Potential Instructions by DivZero:
  %div = sdiv i32 1, %2
```

Format of Input Programs

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

- You can ignore precisely handling values other than integers but your LLVM pass must not raise a segmentation fault when encountered with other kinds of values.
- You should handle assignments, signed and unsigned arithmetic operations (+, -, *, /), comparison operations (<, <=, >, >=, ==, !=), and branches. You do not have to handle XOR, OR, AND, and Shift operations precisely but your program must not raise a segmentation fault in these cases.
- 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

In this lab, you will extend your divide-by-zero analysis that you implemented in previous part to analyze and catch potential divide-by-zero errors in the presence of aliased memory locations.

During lecture, you learned that introducing aliasing into a language makes reasoning about a program's behavior more difficult, and requires some form of pointer analysis. You will perform *flow-insensitive* pointer analysis — where we abstract away control flow and build a global points-to graph — to help your sanitizer analyze more meaningful programs.

Part 1: Function Arguments / Call Instructions

Step 1

As you might have noticed, all of the sample programs were basic functions that accepted no arguments. For example:

```
void f() {
  int x = 0;
  int y = 2;
  int z;
  if (x < 1) {
    z = y / x; // divide-by-zero within branch
  }
}
```

The function `f()` has no arguments in its signature. Realistically, functions can accept any number of variables, and even of different types (but for this lab, consider all arguments as `int`'s).

Step 2

Familiarize yourself with the `doAnalysis()` routine that acts as the entrypoint to your divide-by-zero LLVM pass. In previous part, you modified this routine to implement the full chaotic iteration algorithm:

```
void DivZeroAnalysis::doAnalysis(Function &F, PointerAnalysis *PA)
```

The function signature for `doAnalysis()` has now changed slightly to include a `PointerAnalysis` object. We will go over this in part 2.

Step 3

Given an arbitrary function `F` passed into your `doAnalysis()` routine, find the arguments of the function call and instantiate abstract domain values for each argument. Note that the object `F` here is of type Function [2], which can be used to find all the arguments available.

Furthermore, once you've initialized these starting argument abstract values, pass these values into your existing implementation of the divide-by-zero pass such that these variables get propagated throughout the entire reaching definitions analysis.

[2] `https://llvm.org/doxygen/classllvm_1_1Function.html`

Step 4

In addition to handling arguments of the function F being analyzed, we also want to cover other function calls made within the program. We've seen this before with this function:

```
void main() {
  int x = getchar();
  int y = 5 / x;
  return 0;
}
```

In the above example, the `getchar()` is an external function call made without arguments that returns an `int`. Update your analysis to handle arbitrary CallInst [3] instructions, but only if the return type is an `int`.

Part 2: Store / Load Instructions

Step 1

As mentioned above, there's a change made to the former `doAnalysis()` function:

```
void DivZeroAnalysis::doAnalysis(Function &F, PointerAnalysis *PA)
```

In addition, we have modified the signature of the `transfer` function used in previous lab part:

```
void DivZeroAnalysis::transfer(Instruction *I, const Memory *In, Memory *NOut,
      PointerAnalysis *PA, SetVector<Value *> PointerSet)
```

Please make sure when reusing code from the previous assignment that you copy your implementation details and function contents, but leave these two function signatures intact! These arguments will be necessary as we explore pointer aliasing.

To help understand how the code is now tied together, consider the following snippet from `DataFlowAnalysis::runOnFunction()`:

```
// ...
  PointerAnalysis *PA = new PointerAnalysis(F);
  doAnalysis(F, PA);
```

And the following snippet from `DivZeroAnalysis::doAnalysis()`:

```
void DivZeroAnalysis::doAnalysis(Function &F, PointerAnalysis *PA) {
// ...
for (inst_iterator I = inst_begin(F), E = inst_end(F); I != E; ++I) {
  WorkSet.insert(&(*I));
  PointerSet.insert(&(*I));
}
// ...
transfer(I, In, NOut, PA, PointerSet);
```

Again, note that the transfer function now gets PointerAnalysis and a PointerSet as inputs. Keep this in mind when reusing your code from previous lab part.

[3] `https://llvm.org/doxygen/classllvm_1_1CallInst.html`

Step 2

At a high level, you will modify the `transfer()` function in `DivZeroAnalysis.cpp` to perform a more sophisticated divide-by-zero analysis by keeping track of pointers.

The skeleton code for the PointerAnalysis class is in `DivZero/src/PointerAnalysis.cpp` and it includes the implementation of various methods needed to perform this pointer aliasing. By the end of your LLVM pass, the `PointerAnalysis *PA` object will contain the result of the pointer analysis run on the function, and PointerSet will contain all pointers from the `Function`.

We will discuss in more detail what this `PointerAnalysis` class does in the following sections, but read through the code and make sense of what is being done in each of the methods provided.

Modeling LLVM alloca, store, and load. Here we provide an interface for working with pointers in LLVM. You may use this as a fallback, but feel free to model references in LLVM as you wish.

For this lab, we have disabled the mem2reg pass used in previous lab part. As such, LLVM will create a memory cell for every C variable. As a result you will not see phi-nodes, and you will not need the code segments in which you implemented for handling them in previous lab part. Consider the following code:

```
int f(){
  int a = 0;
  int *c = &a;
  int x = 1 /  *c;
  return x;
}
```

```
I1:   %a = alloca i32, align 4
I2:   %c = alloca i32*, align 8
I3:  %x = alloca i32, align 4
I4:   store i32 0, i32* %a, align 4
I5:   store i32* %a, i32** %c, align 8
I6:   %0 = load i32*, i32** %c, align 8
I7:   %1 = load i32, i32* %0, align 4
I8:  %div = sdiv i32 1, %1
I9:  store i32 %div, i32* %x, align 4
I10: %2 = load i32, i32* %x, align 4
I11: ret i32 %2

M[variable(I1)] = 0
M[variable(I2)] = variable(I1)
M[variable(I6)] = M[variable(I2)]
```

As in the previous lab part, the `variable()` method is still used to encode the variable of an instruction.

Building the Points-To Graph. The `PointerAnalysis` class builds a points-to graph that you will use in your `transfer` function. `PointsToInfo` represents a mapping from variables to a `PointsToSet`, which represents the set of allocation sites a variable can point to.

To help model the memory location that corresponds to a variable `\%a` (i.e., `variable(I1)`), we provide a function `address`, which you can use to encode the memory address (`address(I1)`) of a variable when building the `PointsToSet`. Instruction `I2` will be similarly analyzed. At `I5`, the memory location allocated at `I2` (i.e., `address(I2)`) will store the memory location allocated at `I1` (i.e., `address(I1)`).

Additionally, the field `PointsTo` represents the complete points-to graph that you will construct.

The implementation for the `PointerAnalysis` constructor that will go through all the instructions for a given `Function F` and populate `PointsTo` has been completed for you as part of the skeleton code in this assignment. In addition, we have also provided an `alias()` method which returns true if two pointers are aliased.

Step 3

Using the PointerAnalysis object, augment your `transfer()` function in `DivZeroAnalysis.cpp` to take into account pointer aliasing during its analysis. This should be done by adding handling for StoreInst [4] and LoadInst [5] instructions.
LoadInst We can rely on the existing variables defined within the In memory to know what abstract domain should be for the new variable introduced by a load instruction. For example, given a load instruction as follows:

```
%2 = load i32, i32* %1, align 4
```

This is loading the value of the pointer at `%1` into a new variable `%2` of type `i32`. With the addition of pointers, we can also have:

```
%1 = load i32*, i32** %d, align 8
```

This is loading the value of the pointer at `%d` (which itself is a pointer) into a new variable `%1` of type `i32*`. Note the extra * characters in the load instruction's type (load `i32*`) compared to the previous example. You can retrieve this load instruction's type using `getType()` [6], and further check the type using methods like `isIntegerTy()` [7] or `isPointerTy()` [8].
StoreInst Store instruction can either add new variables or overwrite existing variables into our memory maps. For example, given a store instruction as follows:

```
store i32 0, i32* %a, align 4
```

This is storing the value of 0 into variable `\%a`. You should be familiar with retrieving these operands using `getOperand()` [9], but you can also use `getValueOperand()` [10] and `getPointerOperand()` [11] methods respectively. With the addition of pointers, we can also have:

```
store i32* %a, i32** %c, align 4
```

Now we're storing the pointer at `\%a` into variable `\%c`, which is a pointer to a pointer. We can again use type information from `getType()` [12] on each of these operands to determine whether pointer-aliasing may apply. This clearly complicates our abstract domain analysis - if some further instruction updates the value of `\%a`, we not only need to update the abstract value of `\%c`, but also consider updating the abstract value of other pointers that point to `\%a`. This also applies to changes made to `\%c` which is what happens in the `pointer0.c` example.

```
int f() {
  int a = 1;
  int *c = &a;
  int *d = &a;
```

[4]https://llvm.org/doxygen/classllvm_1_1StoreInst.html
[5]https://llvm.org/doxygen/classllvm_1_1LoadInst.html
[6]https://llvm.org/doxygen/classllvm_1_1Value.html#a6393a2d4fe7e10b28a0dcc35f881567b
[7]https://llvm.org/doxygen/classllvm_1_1Type.html#ac6d28a9b11139182134a9618028a0d07
[8]https://llvm.org/doxygen/classllvm_1_1Type.html#a3b996fbf8458aafffc86cb98a68d0a47
[9]https://llvm.org/doxygen/classllvm_1_1User.html#abe1de1520a21f77ac57cc210bf0fb0b4
[10]https://llvm.org/doxygen/classllvm_1_1StoreInst.html#a14298313bdf734e2db5a921cc6e861a0
[11]https://llvm.org/doxygen/classllvm_1_1StoreInst.html#ac03c1c093059ea000216af8dd6f2dbf4
[12]https://llvm.org/doxygen/classllvm_1_1Value.html#a6393a2d4fe7e10b28a0dcc35f881567b

```
  *c = 0;
  ...
}
```

To resolve these cases, we can rely on the points-to graph constructed in `PointerAnalysis`. We'll need to iterate through the provided `PointerSet`: if we come across some instance where there exists a may-alias (`PA->isAlias()` returns true), this essentially means there's an edge that connects the pointer values between two variables. Once we know what connections exist, we will need to get each abstract value, join them all together via `Domain::join()`, then proceed to update the current assignment as well as all may-aliased assignments with this abstract value. This ensures that all pointer references are in-sync and will converge upon a precise abstract value in our analysis.

Submission

Submit your modified file `DivZeroAnalysis.cpp`. Additionally, submit files `Domain.h` and `Domain.cpp`, if you have implemented an alternative abstract domain.