Lab Work – Class 3

Name: ______________________________ ID: ____________

Code Obfuscation is a technique used to hide the actual semantics of a program by means of transformations that make it hard to read and understand. There are several ways to obfuscate code. In this exercise, we will play with one of these ways. Our goals is to replace occurrences of the constant “zero” in the LLVM intermediate representation by boolean expressions that always return false. As an example, consider the program below[1]:

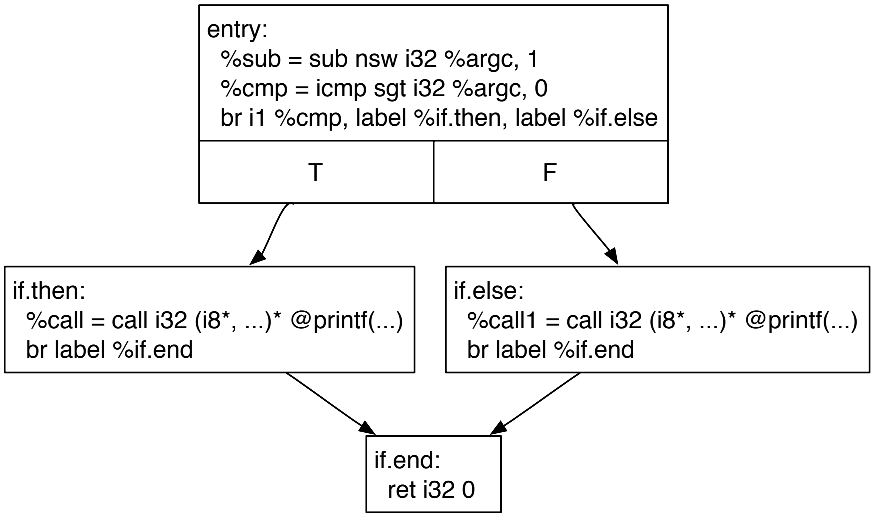
```
#include <stdio.h>

int main(int argc, char** argv) {
  int i = argc - 1;
  if (argc > 0) {
    printf("Arguments found\n");
  } else {
    printf("No arguments\n");
  }
}
```

Let’s compile this program, using the following commands:

```
$> clang -c -emit-llvm ex3_0.c -o ex3_0.bc
$> opt -mem2reg ex3_0.bc -o ex3_0.rbc
```

If we use these commands, then we obtain the CFG below, which is written in LLVM IR:



```
entry:
  %sub = sub nsw i32 %argc, 1
  %cmp = icmp sgt i32 %argc, 0
  br i1 %cmp, label %if.then, label %if.else
```

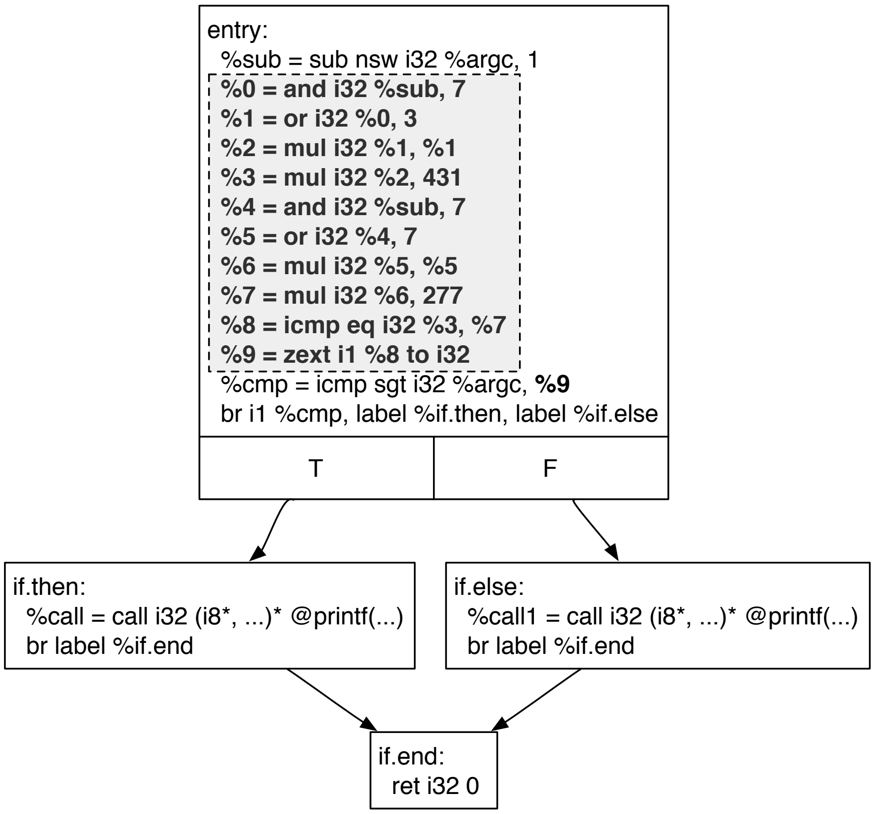
```
if.then:
  %call = call i32 (i8*, ...)* @printf(...)
  br label %if.end
```

```
if.else:
  %call1 = call i32 (i8*, ...)* @printf(...)
  br label %if.end
```

```
if.end:
  ret i32 0
```

The second instruction of this program, e.g., `%cmp = icmp sgt i32 %argc, 0`, uses the constant zero explicitly. We could replace this constant with a comparison that always returns false. As an example, the CFG below implements this change:

[1] `http://homepages.dcc.ufmg.br/∼fernando/classes/dcc888/Obfuscate/lab3/ex3_0.c`



```
entry:
  %sub = sub nsw i32 %argc, 1
  %0 = and i32 %sub, 7
  %1 = or i32 %0, 3
  %2 = mul i32 %1, %1
  %3 = mul i32 %2, 431
  %4 = and i32 %sub, 7
  %5 = or i32 %4, 7
  %6 = mul i32 %5, %5
  %7 = mul i32 %6, 277
  %8 = icmp eq i32 %3, %7
  %9 = zext i1 %8 to i32
  %cmp = icmp sgt i32 %argc, %9
  br i1 %cmp, label %if.then, label %if.else
```

```
if.then:
  %call = call i32 (i8*, ...)* @printf(...)
  br label %if.end
```

```
if.else:
  %call1 = call i32 (i8*, ...)* @printf(...)
  br label %if.end
```

```
if.end:
  ret i32 0
```

In this example we have replaced "zero" with the comparison below, in which a_1 and a_2 are different integer numbers produced randomly:

$$431 * ((x|a_1)^2) \neq 277 * ((y|a_2)^2)$$

In this question you will code such an obfuscator. Part of the solution of this exercise has been already done. The files are available for download [2]. In that package, you will find the following files:

- `Makefile`: type `make` to compile the program, if it is on an LLVM tree such as `lib/Transforms`
- `ObfuscateZero.h`: header file of the obfuscator.
- `ObfuscateZero.cpp`: implementation of the obfuscation routines.

1. The implementation file, e.g., `ObfuscateZero.cpp`, contains a routine `createExpression` whose body is incomplete:

```
Value *ObfuscateZero::createExpression
(Type* IntermediaryType, const uint32_t p, IRBuilder<>& Builder) {

  // BEGIN HELP: You can use these declarations, or you can remove them.
  std::uniform_int_distribution<size_t> Rand(0, IntegerVect.size() - 1);
  std::uniform_int_distribution<size_t> RandAny(1, 10);

  size_t Index = Rand(Generator);
```

[2] http://homepages.dcc.ufmg.br/∼fernando/classes/dcc888/Obfuscate/lab3/Obfuscator.zip

```
  Constant *any = ConstantInt::get(IntermediaryType, 1 + RandAny(Generator)),
           *prime = ConstantInt::get(IntermediaryType, p),
           *OverflowMask = ConstantInt::get(IntermediaryType, 0x00000007);
  // END HELP.

  Value *Tot;
  // Tot = ... "insert code here";

  return Tot;
}
```

The goal of this exercise is to implement the missing parts of `createExpression`. This code must create a sequence of LLVM instructions that implements an expression E, such that if we call `createExpression` with different prime numbers, then we will have two different expression, E_1 and E_2, which are different. To create instructions, you must use `IRBuilder`, a helper class available in the LLVM API.

2. Try obfuscating our example program, given in the beginning of this exercise. Once you produce an obfuscated program, try to optimize the original and the obfuscated program with `opt -O3`:

```
$> clang -c -emit-llvm ex3_0.c -o ex3_0.bc
$> opt -mem2reg ex3_0.bc -o ex3_0.rbc
$> opt -load Obfuscator.dyLib -obfZero ex3_0.rbc -o ex3_0.obf.rbc
$> opt -O3 ex3_0.obf.rbc -o ex.opt.obf.rbc
$> opt -O3 ex3_0.rbc -o ex.opt.orig.rbc
$> ls -la *.rbc
-rw-r--r--  1 fernando  staff  1344 Jan 19 23:03 ex.opt.obf.rbc
-rw-r--r--  1 fernando  staff  1296 Jan 19 23:03 ex.opt.orig.rbc
-rw-r--r--  1 fernando  staff  1616 Jan 19 11:36 ex3_0.obf.rbc
-rw-r--r--  1 fernando  staff  1584 Jan 19 10:21 ex3_0.rbc
```

If your obfuscator is good, then it resists an optimization. Are the resulting codes that you produce different? In other words, have you coded a good obfuscator?

3. How can you be sure that `createExpression`(431) and `createExpression`(277) are always different? Notice that if `createExpression` simply returns the argument that it receives, then an optimizer is likely to remove your obfuscations away.