CS453 Automated Software Testing

LLVM Pass and Code Instrumentation

Prof. Moonzoo Kim

CS Dept., KAIST

Motivating Example

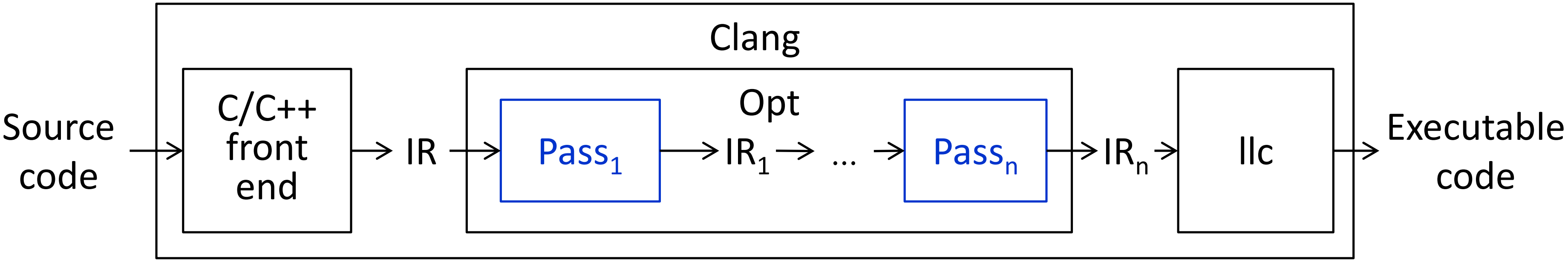
- (30 pts) Find bugs in the following program that has multiple bugs
  - Write down buggy lines, bugs, and explain the bugs as many as possible
  - Write down a code instrumentor using Clang which inserts assert() to report runtime failures due to the bugs you detected.
- For example, to report a div-by-zero crash, your code should insert assert(z!=0) immediately before x=y/z;

```c
//example1.c
#include <malloc.h>
#include <stdio.h>
#include <string.h>
void f() {
  char* mem = NULL;
  int length;
  char buf[100];
  // file descriptor 0 is connected to keyboard
  read(0, &length, sizeof(int));
  int r=read(0, &buf,length>100 ?
        100:length);
  mem = malloc(r + 1);
  buf[r] = 0;
  strcpy(mem, buf);
  printf(mem);
  fflush(stdout);
}
```

Which tool do you prefer for the task? Clang? LLVM IR?

Pass in LLVM

- A Pass receives an LLVM IR and performs analyses and/or transformations.
  - Using `opt`, it is possible to run each Pass.
- A Pass can be executed in a middle of compiling process from source code to binary code.
  - The pipeline of Passes is arranged by Pass Manager



LLVM Pass Framework

- The LLVM Pass Framework is the library to manipulate an AST of LLVM IR (http://llvm.org/doxygen/index.html)
- An LLVM Pass is an implementation of a subclass of the Pass class
  - Each Pass is defined as visitor on a certain type of LLVM AST nodes
  - There are six subclasses of Pass
    - ModulePass: visit each module (file)
    - CallGraphSCCPass: visit each set of functions with caller-call relations in a module (useful to draw a call graph)
    - FunctionPass: visit each function in a module
    - LoopPass: visit each set of basic blocks of a loop in each function
    - RegionPass: visit the basic blocks not in any loop in each function
    - BasicBlockPass: visit each basic block in each function

Control Flow Graph (CFG) at LLVM IR

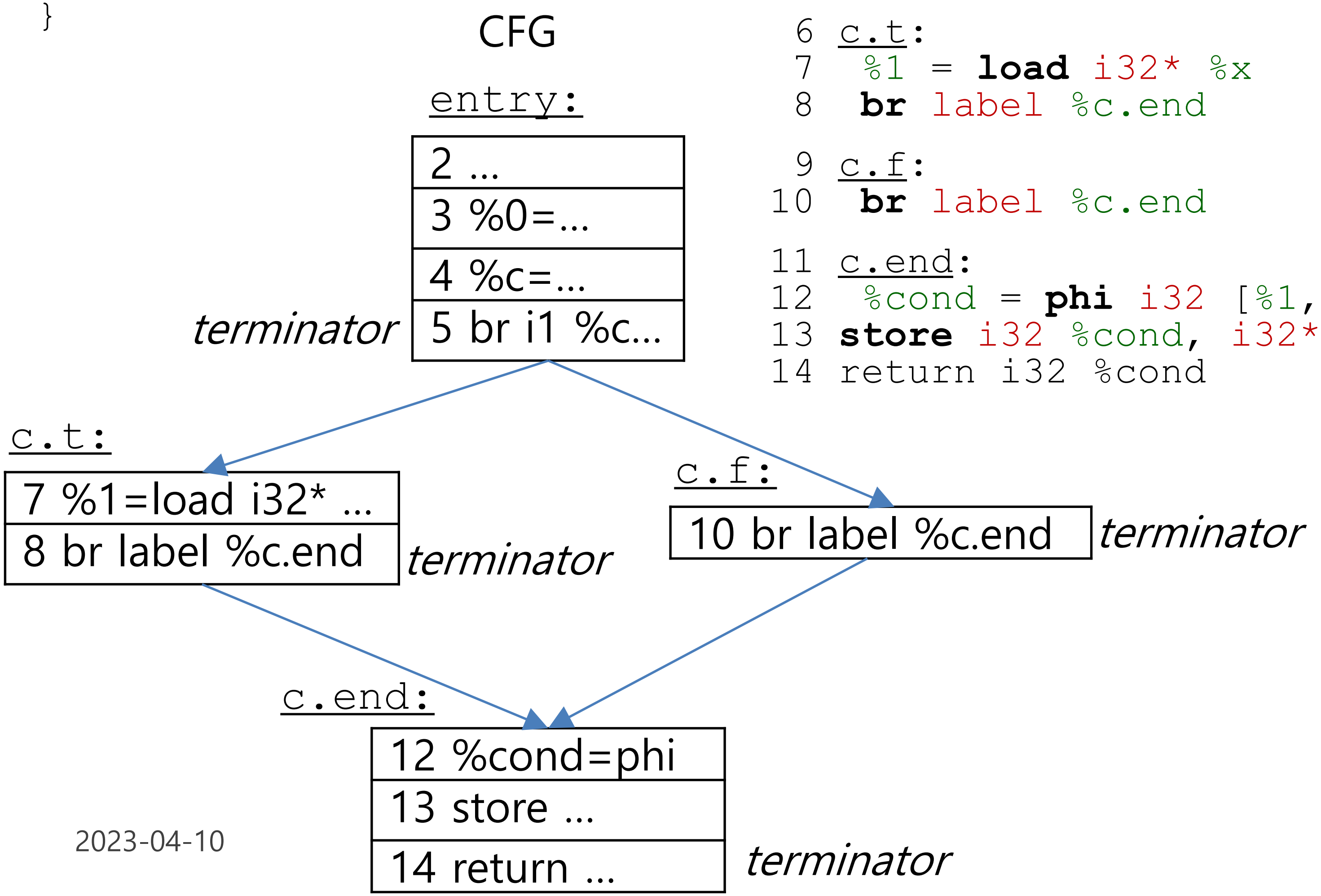
```
int f() {
  int y;
  y = (x > 0) ? x : 0 ;
  return y;
}
```

```
 1 entry:
 2 …
 3 %0 = load i32* %x
 4 %c = icmp sgt i32 %0 0
 5 br i1 %c, label %c.t, %c.f

 6 c.t:
 7  %1 = load i32* %x
 8  br label %c.end

 9 c.f:
10  br label %c.end

11 c.end:
12  %cond = phi i32 [%1,%c.t],[0,%c.f]
13 store i32 %cond, i32* %y
14 return i32 %cond
```



CFG

entry:

2 …

3 %0=…

4 %c=…

5 br i1 %c…

terminator

c.t:

7 %1=load i32* …

8 br label %c.end

terminator

c.f:

10 br label %c.end

terminator

c.end:

12 %cond=phi

13 store …

14 return …

terminator

Example Pass

- Let’s create *IntWrite* that aim to monitor all history of 32-bit integer variable updates (definitions)
  - Implemented as a `FunctionPass`
  - Produces a text file where it records which variable is defined as which value at which code location.

- IntWrite instruments a target program to insert a *probe* before every integer writing operation, which extracts runtime information

```
10   y = x ;
11   z = y + x ;
```

```
     _probe_(10, “y”, x);
10   y = x ;
     _probe_(11, “z”, y+x);
11   z = y + x ;
     …
    void _probe_(int l,char *,int v){
      fprintf(fp, “%d %s %d\n”,…);}
```

`Module` Class

- A `Module` instance stores all information related to the LLVM IR created by a target program file (functions, global variables, etc.)
- APIs (public methods)
  - `getModuleIdentifier()`: return the name of the module
  - `getFunction(StringRef Name)`: return the `Function` instance whose identifier is `Name` in the module
  - `getOrInsertFunction(StringRef Name, Type *ReturnType,…)`: add a new `Function` instance whose identifier is `Name` to the module
  - `getGlobalVariable(StringRef Name)`: return the `GlobalVariable` instance whose identifier is `Name` in the module

`Type` Class

- A `Type` instance is used for representing the data type of registers, variables, and function arguments.

- Static members
  - `Type::getVoidTy(…)` : void type
  - `Type::getInt8Ty(…)` : 8-bit unsigned integer (char) type
  - `Type::getInt32Ty(…)` : 32-bit unsigned integer type
  - `Type::getInt8PtrTy(…)` : 8-bit pointer type
  - `Type::getDoubleTy(…)` : 64-bit IEEE floating pointer type

`FunctionPass` Class (1/2)

- `FunctionPass::doInitialization(Module &)`
  - Executed once for a module (file) before any visitor method execution
  - Do necessary initializations, and modify the given `Module` instances (e.g., add a new function declaration)

- `FunctionPass::doFinalization(Module &)`
  - Executed once for a module (file) before after all visitor method executions
  - Export the information obtained from the analysis or the transformation, any wrap-up

Example

- IntWrite should inserts a new function `_init_` at the beginning of the target program’s main function
  – _init_() is to open an output file

```
virtual bool doInitialization(Module & M)  {
  if(M.getFunction(StartingRef(“_init_”))!=NULL){
    errs() << “_init_() already exists.” ;
    exit(1) ;
  }
```

check if _init_() already exists

```
  FunctionType *fty =
    FunctionType::get(Type::getVoidTy(M.getContext()),false) ;
  fp_init_ = M.getOrInsertFunction(“_init_”, fty) ;
```

add a new declaration _init_()

```
  ...
  return true ;
}
```

`FunctionPass` Class (2/2)

- `runOnFunction(Function &)`
  - Executed once for every function defined in the module
  - Read and modify the target function definition

- `Function` Class
  - `getFunctionType()`: returns the `FunctionType` instance that contains the information on the types of function arguments.
  - `getEntryBlock()`: returns the `BasicBlock` instance of the entry basic block.
  - `begin()`: the head of the `BasicBlock` iterator
  - `end()`: the end of the `BasicBlock` iterator

Example

```
virtual bool runOnFunction(Function &F) {
  cout << “Analyzing “ << F->getName() << “\n” ;
  for (Function::iterator i = F.begin(); i != F.end(); i++){
    runOnBasicBlock(*i) ;
  }
  return true;//You should return true if F was modified. False otherwise.
}
```

`BasicBlock` Class

- A `BasicBlock` instance contains a list of instructions
- APIs
  - `begin()`: return the iterator of the beginning of the basic block
  - `end()`: return the iterator of the end of the basic block
  - `getFirstInsertionPt()`: return the first iterator (i.e., the first instruction location) where a new instruction can be added safely (i.e., after phi instruction and debug intrinsic)
  - `getTerminator()`: return the terminator instruction
  - `splitBasicBlock(iterator I, …)`: split the basic block into two at the instruction of `I` by inserting an unconditional jump

`Instruction` Class

- An `Instruction` instance contains the information of an LLVM IR instruction.
- Each type of instruction has a subclass of `Instruction` (e.g. `LoadInst`, `BranchInst`)
- APIs
  - `getOpcode()`: returns the opcode which indicates the instruction type
  - `getOperand(unsigned i)`: return the i-th operand
  - `getDebugLoc()`: obtain the debugging data that contains the information on the corresponding code location
  - `isTerminator(),isBinaryOp(), isCast(), ….`

Example

```
bool runOnBasicBlock(BasicBlock &B) {
  for(BasicBlock::iterator i = B.begin(); i != B.end(); i++){
    if(i->getOpcode() == Instruction::Store &&
       i->getOperand(0)->getType() == Type::getInt32Ty(ctx)){
      StoreInst * st = dyn_cast<StoreInst>(i);
      int loc = st->getDebugLoc().getLine(); //code location
      Value * var = st->getPointerOperand(); //variable
      Value * val = st->getOperand(0); // value
      /* insert a function call */
    }
  }
  return true ;
}
```

How to Insert New Instructions

- `IBBuilder` class provides a uniform API for inserting instructions to a basic block.
  - `IRBuilder(Instruction *p)`: create an `IRBuilder` instance that can insert instructions right before `Instruction *p`
- APIs
  - `CreateAdd(Value *LHS, Value *RHS, …)`: create an add instruction whose operands are `LHS` and `RHS` at the predefined location, and then returns the `Value` instance of the target operand
  - `CreateCall(Value *Callee, Value *Arg,…)`: add a new call instruction to function `Callee` with the argument as `Arg`
  - `CreateSub(),CreateMul(),CreateAnd(),…`

`Value` Class

- A `Value` is a super class of all entities in LLVM IR such as a constant, a register, a variable, and a function.
- The register defined by an `Instruction` is represented as a `Value` instance.
- APIs
  - `getType()`: returns the `Type` instance of a `Value` instance.
  - `getName()`: return the name from the source code.

Example

```
if(i->getOpcode() == Instruction::Store &&
   i->getOperand(0)->getType() == Type::getInt32Ty(ctx) {

   StoreInst * st = dyn_cast<StoreInst>(i);

   int loc = st->getDebugLoc().getLine(); //code location
   Value * var = st->getPointerOperand(); //variable
   Value * val = st->getOperand(0); // target register

   IRBuilder<> builder(i) ;
   Value * args[3] ;
   args[0] = ConstantInt::get(intTy, loc, false) ;
   args[1] = builder.CreateGlobalStringPtr(var->getName(),"");
   args[2] = val ;
   builder.CreateCall(p_probe, args, Twine("")) ;
   // p_probe should be created before by using
   // getOrInsertFunction() and target code should be compiled
   // with the function definition pointed by p_probe.
   // See IntWrite.cpp and IntWrite.c which contains the
   // definition of probe function
}
```

More Information

- Writing an LLVM Pass
  - http:// llvm.org/docs/WritingAnLLVMPass.html
- LLVM API Documentation
  - http://llvm.org/doxygen/
- How to Build and Run an LLVM Pass for Homework#4
  - http://swtv.kaist.ac.kr/courses/s453-14fall/hw4-manual.pdf