CSE 703 Seminar:
Advanced Software Security - Techniques and Tools

Instructor: Dr. Ziming Zhao

Location: Online
Time: Monday, 12:50 PM-2:55 PM

Writing LLVM Passes

Passes

- LLVM applies a chain of analyses and transformations to the target program
- Each of these analyses or transformations is called a **pass**

- Machine-independent passes are invoked by *opt*
- Machine-dependant passes are invoked by *llc*

- A pass may require information provided by other passes. Dependencies must be explicitly stated. A transformation pass may require an analysis pass.

Types of Passes

A pass is an instance of the LLVM class Pass.

LLVM Coding Basics

- Written in modern C++, uses the STL:
  - Particularly the vector, set, and map classes

- LLVM IR is almost all doubly-linked lists:
  - Module contains lists of Functions & GlobalVariables
  - Function contains lists of BasicBlocks & Arguments
  - BasicBlock contains list of Instructions

- Linked lists are traversed with iterators:

```
Function *M = …
for (Function::iterator I = M->begin(); I != M->end(); ++I) {
  BasicBlock &BB = *I;
  …
```

See also: docs/ProgrammersManual.html

LLVM Pass Manager

- Compiler is organized as a series of ‘passes’:
  - Each pass is one analysis or transformation
- Four types of Pass:
  - ModulePass: general interprocedural pass
  - CallGraphSCCPass: bottom-up on the call graph
  - FunctionPass: process a function at a time
  - BasicBlockPass: process a basic block at a time
- Constraints imposed (e.g. FunctionPass):
  - FunctionPass can only look at “current function”
  - Cannot maintain state across functions

See also: docs/WritingAnLLVMPass.html

Services provided by PassManager

- Optimization of pass execution:
  - Process a function at a time instead of a pass at a time
  - Example: If F, G, H are three functions in input pgm: “FFFFGGGGHHHH” not “FGHFGHFGHFGH”
  - Process functions in parallel on an SMP (future work)
- Declarative dependency management:
  - Automatically fulfill and manage analysis pass lifetimes
  - Share analyses between passes when safe:
    - e.g. “DominatorSet live unless pass modifies CFG”
- Avoid boilerplate for traversal of program

See also:
docs/WritingAnLLVMPass.html

HelloWorld Pass

```
#include "llvm/Pass.h"
#include "llvm/IR/Function.h"
#include "llvm/Support/raw_ostream.h"

#include "llvm/IR/LegacyPassManager.h"
#include "llvm/Transforms/IPO/PassManagerBuilder.h"

using namespace llvm;

namespace {
struct Hello : public FunctionPass {
  static char ID;
  Hello() : FunctionPass(ID) {}

  bool runOnFunction(Function &F) override {
    errs() << "Hello: ";
    errs().write_escaped(F.getName()) << '\n';
    return false;
  }
}; // end of struct Hello
}  // end of anonymous namespace

char Hello::ID = 0;
static RegisterPass<Hello> X("hello", "Hello World Pass",
                             false /* Only looks at CFG */,
                             false /* Analysis Pass */);

static RegisterStandardPasses Y(
    PassManagerBuilder::EP_EarlyAsPossible,
    [](const PassManagerBuilder &Builder,
       legacy::PassManagerBase &PM) { PM.add(new Hello()); });
```

```
43 namespace {
44   // Hello2 - The second pass in our tutorial
45   struct Hello2 : public FunctionPass {
46     static char ID; // Pass identification, replacement for typeid
47     Hello2() : FunctionPass(ID) {}
48
49     bool runOnFunction(Function &F) override {
50         unsigned int basicBlockCount  = 0;
51         unsigned int instructionCount = 0;
52         for(BasicBlock &bb : F){
53             ++basicBlockCount;
54             for(Instruction &i : bb){
55                 ++instructionCount;
56             }
57         }
58         errs() << "Hello2 is running: ";
59         errs().write_escaped(F.getName())   << "Basic Blocks:"  << basicBlockCount
60                                             << "Instruction:"   << instructionCount << "\n";
61     }
62
63     // We don't modify the program, so we preserve all analyses.
64     void getAnalysisUsage(AnalysisUsage &AU) const override {
65       AU.setPreservesAll();
66     }
67   };
68 }
69
70 char Hello2::ID = 0;
71 static RegisterPass<Hello2>
72 Y("hello2", "Hello World Pass (with getAnalysisUsage implemented)");
```

Here is where we will accumulate the basic blocks and instructions within our function

```
74 #include "llvm/IR/CallSite.h"
75 namespace {
76   // Hello3 - The third part of our tutorial
77   struct Hello3 : public FunctionPass {
78     static char ID; // Pass identification, replacement for typeid
79     Hello3() : FunctionPass(ID) {}
80
81     bool runOnFunction(Function &F) override {
82         for(BasicBlock &bb: F){
83             for(Instruction &i: bb){
84                 // Find where callsite is of our instruction
85                 CallSite cs(&i);
86                 if(!cs.getInstruction()){
87                     continue;
88                 }
89                 Value *called = cs.getCalledValue()->stripPointerCasts();
90                 if(Function* f = dyn_cast<Function>(called)){
91                     errs()  << "\tDirect call to function:" << f->getName()
92                             << " from "                     << F.getName() << "\n";
93                 }
94             }
95         }
96
97         return false;
98     }
99
100    // We don't modify the program, so we preserve all analyses.
101    void getAnalysisUsage(AnalysisUsage &AU) const override {
102      AU.setPreservesAll();
103    }
104  };
105 }
106
107 char Hello3::ID = 0;
108 static RegisterPass<Hello3>
109 Z("hello3", "Hello World Pass (Get direct calls)");
```

The Module pass

```
bool runOnModule(Module &M) override {
    // The setup hooks function creates
    // a 'stub' function for us to hook some source into.
    setupHooks("_Z10__initMaini",M);
    // We next loop through all our functions in the module
    // This is where you could instrument only a subset
    // of functions.
    // Be careful just not to modify instrumentation functi
    Module::FunctionListType &functions = M.getFunctionLi
    for(Module::FunctionListType::iterator FI = functio
        // Ignore our instrumentation function
        if("_Z10=initMaini"==FI->getName()){
            continue;
        }
        if("main"==FI->getName()){
            InstrumentEnterFunction("_Z10__initMaini",*FI,M);
        }
    }
    return true;
}
```

LLVM Dataflow Analysis

- LLVM IR is in SSA form:
  - use-def and def-use chains are always available
  - All objects have user/use info, even functions
- Control Flow Graph is always available:
  - Exposed as BasicBlock predecessor/successor lists
  - Many generic graph algorithms usable with the CFG
- Higher-level info implemented as passes:
  - Dominators, CallGraph, induction vars, aliasing, GVN, …

See also:
docs/ProgrammersManual.html

Homework

- Read “llvm: a compilation framework for lifelong program analysis & transformation” 2014
- Read “SoK: Sanitizing for Security”. Oakland 2019