Static Program Analysis

CS390R - UMass Amherst

Course Information

- Project 5 Out
- Homework 4 Out
- Presentation Checkpoint-2 Out

Today’s Content

- Homework Review
- Basic Ghidra Automated Analysis
- LLVM Introduction
- Why we care
- Source Layout
- Compiler Pipeline
- Demo 1 (C -> LLVM-IR)
- LLVM IR
- Modular Design
- Writing LLVM passes

Ghidra

- currentProgram - Class describing entire open program
  - currentProgram.getMemory().getBlocks() - Return memory mappings
  - currentProgram.getFunctionManager() - Can be used to eg. iterate through functions
- currentLocation - Class describing currently open program-location
- currentAddress - Class describing current location of cursor in window
- etc…

- API Documentation: https://ghidra.re/ghidra_docs/api/

What is LLVM

- Target-independent compiler backend that performs various analyses and transformations on provided LLVM-Intermediate Representation
- Clang is a popular compiler frontend that takes source code (Generally related to the C-family eg. C/C++) and transforms it into LLVM-IR
- LLVM can then transform this IR into machine code for the desired platform (eg. x86/arm)
- Many hobbyists that enjoy writing programming languages also use llvm to create an optimized backend for their language with minimal effort

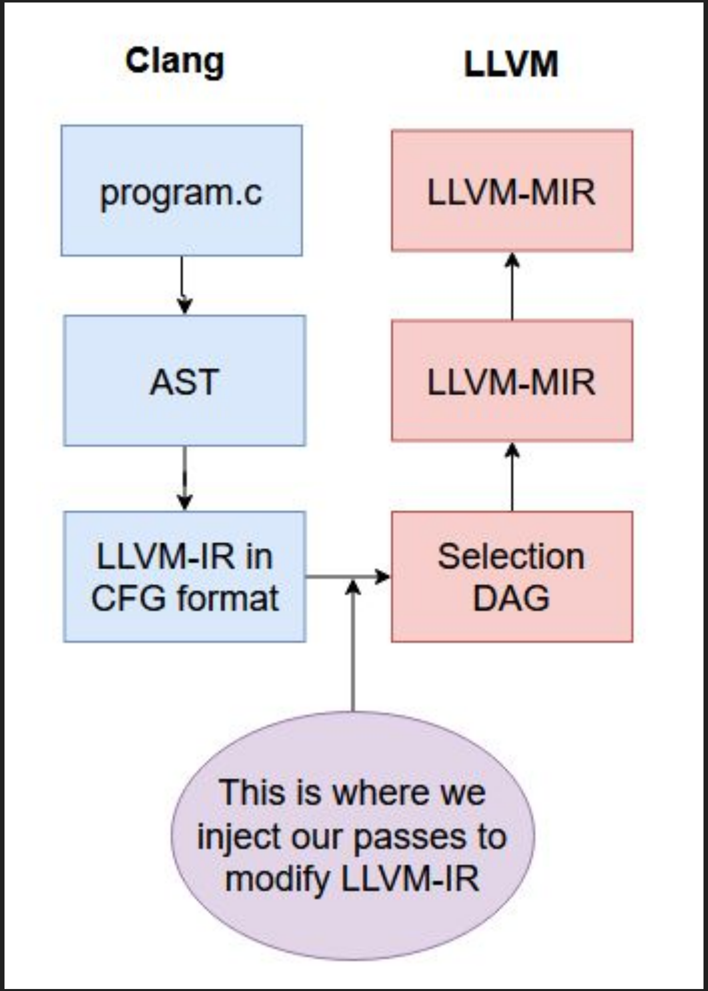
What will we be using it for

- LLVM is built in a modular fashion that allows us to write plugins that interact with the IR's and analysis passes
- SWE teams could use this to write custom optimization passes for their specialized workloads
- On the bug-finding side, llvm exposes its IR, so we can write various static analysis passes on top of this to perform eg. dataflow analysis
- We can also modify this IR before compiling to assembly to change the produced binary at compile time
- This is extremely powerful!
- Why not source? Parsing C/C++ syntax is messy, and performing analysis on it is horrible when compared to a solid IR

Source Introduction

- Source Overview
  - https://github.com/llvm/llvm-project
  - llvm-project/clang - Clang compiler frontend
  - llvm-project/libc   - C standard lib implementation
  - llvm-project/llvm   - LLVM source code
- Building LLVM
  - cmake -DLLVM_ENABLE_PROJECTS=”clang;lld” -DCMAKE_BUILD_TYPE=Release LLVM_ENABLE_ASSERTIONS=ON
  - make -j <num-threads>

Basic Compiler Pipeline



Demo - C -> LLVM-IR

LLVM IR /1

- The format on which llvm operates is called “bitcode” and uses the .bc extension
- Instructions are basically machine independent assembly that encodes enough information to support conversions to various architectures
- All variables are represented in SSA form
- Module - Roughly represents a single source file
    - Global Variable - Represents global variables in this source file
    - Function - Named chunks of callable code, functions/methods, CFG
        - BasicBlock - Sequence of non-branching instructions
            - Instruction - A single code operation, basically llvm-style assembly
- These are all represented as C++ classes in the LLVM source code

LLVM-IR /2

- Binary Operations
  - %sub = sub nsw i32 %0, 3 // Subtract 3 from signed 32-bit integer register %0
  - %cmp = icmp sgt i32 %1, 2 // Compare signed greater than register %1 with value 2
- Memory Operations
  - store i32 5, i32* %b, align 4 // Store value 5 to %b, 4-byte alignment
  - %2 = load i32, i32* %b, align 4 // %2 = i32-value from %b
- Control Flow
  - br i1 %cmp, label %if.then, label %if.else // Branch based on comparison value
  - %call = call i32 @f(i32 noundef 3) // Call function f with i32-argument 3
- Casting
  - %a = trunc i32 950 i8 // %a = 950 casted to an i8
- Other
  - %a = alloca i32, align 4 // %a is allocated 32-bits of memory on the stack

Modular Design

- Sequence of individual passes that run back to back to run various optimization passes on the LLVM-IR bitcode.
- The logic of these individual passes is kept relatively isolated, allowing LLVM to be built up in a modular fashion, thus making it easy to extend it with additional passes
- The individual passes take in IR, do their work/modifications on the IR, and then emit IR for the next pass to consume

Writing LLVM Passes

- 2 Types of passes, static & dynamic
- With static analysis we can implement various algorithms on the presented SSA IR such as gen-kill chains, liveness analysis, etc and determine various program properties based on them without ever modifying the IR
- With dynamic passes we actively modify the IR to observe state changes during execution
    - This is often coupled with other analysis techniques like fuzzing where an LLVM pass can be used to insert instructions at the start of every BasicBlock to track coverage-information or ASAN that can be used to find non-crashing bugs
- Passes can be implemented at the Module, Function, BasicBlock, and Instruction level (Eg. a Function-level pass would result in the pass being invoked on every function)

More Demos