Overview

• We’ve focused on building a compiler, end to end

• In practice, there are a lot of tools we can leverage

• Today we’ll discuss one of the most popular: LLVM
  ■ Introduction to the framework
  ■ Tour of the IR
  ■ Using command-line tools
  ■ Writing optimization passes
  ■ Using and extending the static analyzer
  ■ Symbolic execution with Klee
LLVM Overview

- From http://llvm.org/: “The LLVM Project is a collection of modular and reusable compiler and toolchain technologies.”

- Started in 2000 as a research project at the University of Illinois (Lattner and Adve)
  - Still actively used in compiler and PL research

- Has grown into an industrial scale collection of compilers, libraries, and tools
  - Used and supported by Apple, Adobe, Intel, etc.

- Written in C++, well-documented
Compiler architecture

• Specialized parsers (frontends) and code generators (backends), common optimizers

Getting LLVM

• The project changes frequently
  ▪ And contains a lot of code

• Typically, **build from source**
  ▪ But this can take a while…

• **Binary distributions** are also available

• Macs ship with a subset, installed with Xcode
  ▪ In particular, clang/clang++ (aliased as gcc)
LLVM IR

• Low-level, similar to RISC-like assembly
  ▪ With enough structure to see high-level features

• Strongly-typed: every value has a type
  ▪ includes support for structures

• Infinite temporary registers

• SSA -- static single assignment
  ▪ Can only assign to each variable once
  ▪ Simplifies program analysis

http://llvm.org/docs/LangRef.html
```c
int add(int a, int b)
{
    return a + b;
}
```

```sh
clang -S add.c -emit-llvm -o add.ll
```
LLLVM Tools

• Three IR formats: ASCII (.ll), Bitcode (.bc), and in-memory representation

• `clang/clang++`: compile C to LLVM IR (different frontends for other high-level languages)

• `llvm-as`: translate .ll into .bc

• `llvm-dis`: convert back from .bc to .ll

• `llvm-link`: combine multiple .bc files

• `lli`: interpreter and dynamic compiler

• `llc`: .bc to native assembly (.s)

• `opt`: LLVM optimizer/analyizer

[https://llvm.org/docs/CommandGuide/](https://llvm.org/docs/CommandGuide/)
**opt tool**

- `opt` can be used for both optimization and analysis
  - `loop.c` example: `-O3, -analyze -loops`

- Extensible via DLLs
  - Can write new analyses as “passes”
  - `opt -load LLVMHello.dylib -hello funcs.ll`

http://llvm.org/docs/WritingAnLLVMPass.html#quick-start-writing-hello-world
Static Analyzer

• LLVM can be used to build static analysis tools, e.g., http://clang-analyzer.llvm.org/

```c
void test(int z) {
    if (z == 0) {
        int x = 1 / z;
    }
}
```

```
$ scan-build clang -c div0.c
scan-build: Using 'clang-7' for static analysis
div0.c:3:9: warning: Value stored to 'x' during its initialization is never read
    int x = 1 / z;
        ^  ~~~~~
div0.c:3:15: warning: Division by zero
    int x = 1 / z;
         ~~~^~~
2 warnings generated.
scan-build: 2 bugs found.
```
Address Sanitizer

• LLVM/clang can be used to implement runtime instrumentation for safety, performance measurement, etc.

• https://clang.llvm.org/docs/AddressSanitizer.html

```cpp
int main(int argc, char **argv) {
    int *array = new int[100];
    delete [] array;
    return array[argc];  // BOOM
}
```

clang++ -O1 -g -fsanitize=address -fno-omit-frame-pointer UseAfterFree.cc

=================================================================
==65223==ERROR: AddressSanitizer: heap-use-after-free on address 0x614000000044 at pc…
READ of size 4 at 0x614000000044 thread T0
  #0 0x108d6af07 in main UseAfterFree.cc:4
  #1 0x7ffff67e3a14 in start (libdyld.dylib:dylib:x86_64+0x1014)
Klee: Symbolic Execution

http://klee.github.io/tutorials/testing-function/