Compilers (LLVM)

Martin Fink

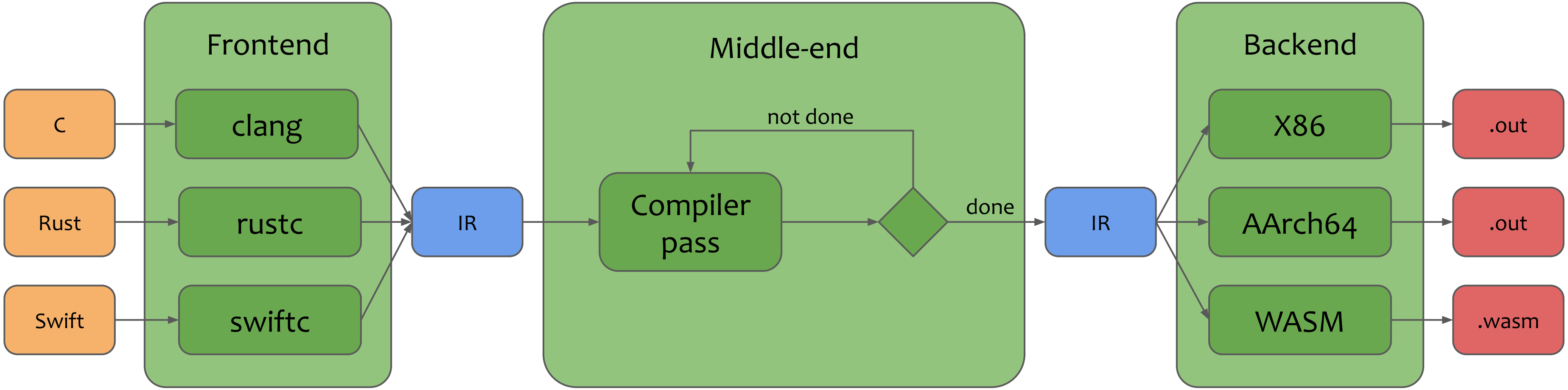
Overview

- What you will learn
  - Quick overview over LLVM
  - Writing your own compiler passes
  - Implementing simple optimizations
  - Implementing a simplified AddressSantizier
- What you will **not** learn
  - Implementing a Compiler yourself
  - Parsing
  - Code generation
  - Hacking on the internals of LLVM

An Overview of LLVM

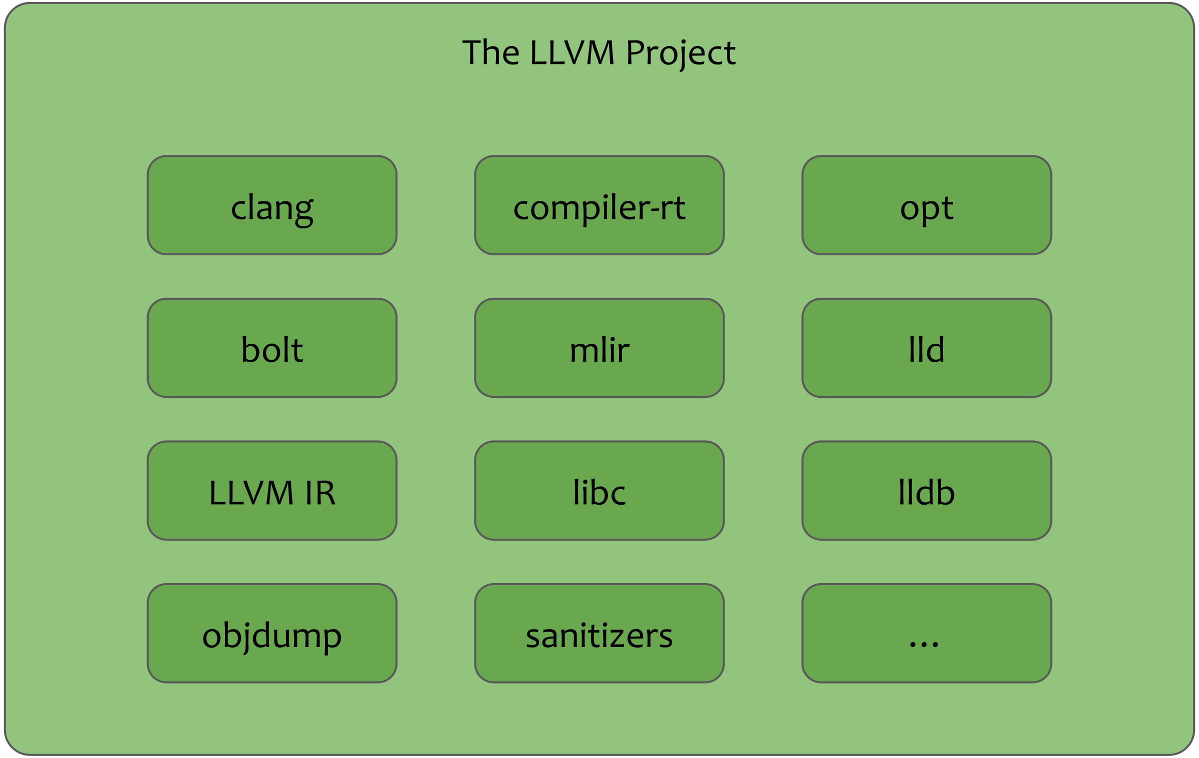
LLVM

- Developed at UIUC by Chris Lattner [1]
- A collection of modular and reusable compiler and toolchain technologies
- Wide variety of use cases across industry



[1]: https://llvm.org/pubs/2004-01-30-CGO-LLVM.pdf

What's part of LLVM?



LLVM IR

- A strongly-typed intermediate representation (IR) with defined semantics
- Static single assignment form (SSA) for local registers
  - Each register is written exactly once
- Designed to host a range of mid-level analyses, transformations/optimizations

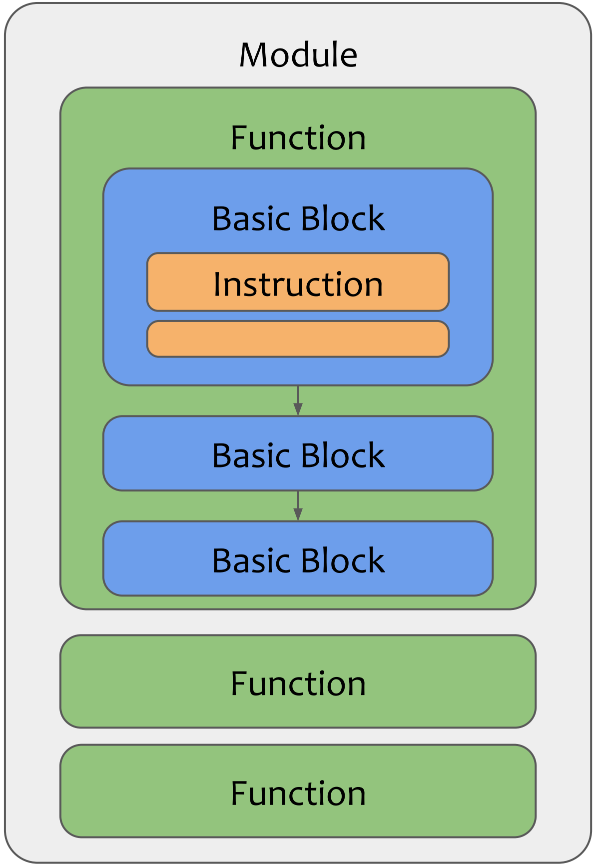
```c
int add1(int a, int b) {
    return a + b;
}
```

```llvm
define dso_local i32 @add1(i32 %0, i32 %1) {
  %2 = add nsw i32 %1, %0
  ret i32 %2
}
```

LLVM IR

- Module: A translation unit
- Function: A Module consists of functions
- Basic block: Function consists of basic blocks, which form a control flow graph. A basic block consists of a sequence of instructions
- Instruction: Instructions with one or more operands

- See LLVM IR: Compiler Explorer [2] or with clang:
  `$ clang -emit-llvm -o - -S main.c`



[2]: https://godbolt.org/z/zoxsqEbM1

LLVM IR (Instruction)

Type

Metadata

Virtual Register

```
%1 = add nsw i32 %0, 42, !dbg !1
```

Instruction

Operands

LLVM Instruction reference: https://llvm.org/docs/LangRef.html#instruction-reference
Instruction documentation: https://llvm.org/doxygen/classllvm_1_1Instruction.html

LLVM IR (Control Flow)

```c
if (cond > 0) {
  val += 1;
} else {
  val += 42;
}
printf("%d\n", val);
```

```llvm
  %cmp = icmp sgt i32 %cond, 0
  br i1 %cmp, label %if.then, label %if.else

if.then:
  %0 = add nsw i32 %val, 1
  br label %if.end

if.else:
  %1 = add nsw i32 %val, 42
  br label %if.end

if.end:
  %val2 = phi i32 [ %0, %if.then ], [ %1, %if.else ]
  %2 = call i32 (ptr, ... ) @printf(ptr @.str.1, i32 %val2)
```

LLVM Passes

LLVM Passes

- Run on different IR constructs, such as functions, modules, loops, and more
  - Can run in the middle- or back-end part of LLVM
- Self-contained
- May depend on some analysis done by other passes
- Take valid IR as an input, transform/analyze it, produce valid IR
- Add external passes to LLVM in order to implement different use cases
  - Security: https://github.com/tudinfse/sgxbounds
  - Reliability: https://github.com/tudinfse/elzar
- Run by the "opt" tool

Hello World Pass

Print the amount of instructions in a function

```
PreservedAnalyses run(Function &F, FunctionAnalysisManager &AM) {
  uint64_t numInstructions = 0;
  for (BasicBlock &BB : F) {
    for (Instruction &I : BB) {
      numInstructions++;
    }
  }

  errs() << F.getName() << ": " << numInstructions << " instructions\n";

  return PreservedAnalyses::all();
}
```

https://github.com/llvm/llvm-project/blob/main/llvm/lib/Transforms/Utils/HelloWorld.cpp

Visitor Pattern

```cpp
struct HelloWorldPass : PassInfoMixin<HelloWorldPass>, public InstVisitor<HelloWorldPass> {
  PreservedAnalyses run(Function &F, FunctionAnalysisManager &AM) {
    this->visit(F);
    return PreservedAnalyses::all();
  }

  void visitInstruction(Instruction &I) {
    // called for every instruction not handled by more specific visit functions
  }

  void visitLoadInst(LoadInst &I) {
  }

  void visitUDiv(BinaryOperator &I) {
  }
};
```

https://llvm.org/doxygen/classllvm_1_1InstVisitor.html

Replacing instructions

- Let's take a look at a simple code transformation [3]
- $\lfloor (x+y)/2 \rfloor \rightarrow (x\&y)+((x \oplus y)>>1)$
- While this is not a useful optimization ($(x+y)>>1$ would be faster), it shows how to replace instructions

```c
int x = read_user_input();
int y = 10 * 1;
int avg = (x + y) / 2;
```

```rust
let mut x = if cond { MIN_VAL } else { arg + 10 };
x += some_function(x).unwrap_or_else(|| get_default_y_val());
return x / 2;
```

[3]: Hacker's delight by Henry S. Warren Jr.

Replacing instructions

```cpp
void visitUDiv(BinaryOperator &I) {
  auto Divisor = dyn_cast<ConstantInt>(I.getOperand(1));
  auto Dividend = dyn_cast<AddOperator>(I.getOperand(0));
  if (!Divisor || !Dividend || Divisor->getValue() != 2) {
    return;
  }

  auto X = Dividend->getOperand(0);
  auto Y = Dividend->getOperand(1);

  IRBuilder<> Builder(&I);
  auto BAnd = Builder.CreateAnd(X, Y);
  auto BXor = Builder.CreateXor(X, Y);
  auto BShr = Builder.CreateLShr(BXor, 1);
  auto Result = Builder.CreateAdd(BAnd, BShr);

  I.replaceAllUsesWith(Result);
}
```

```
(x&y)+((x⊕y)>>1)
```

```llvm
%1 = and i32 %x, %y
%2 = xor i32 %x, %y
%3 = lshr i32 %2, 1
%4 = add i32 %1, %3
```

Replacing instructions

- Let's see if it works!

```
define i32 @avg(i32 %x, i32 %y) {
entry:
  %0 = add i32 %x, %y
  %1 = udiv i32 %0, 2
  ret i32 %1
}
```

```
$ opt -load-pass-plugin ./build/libMyPass.dylib -passes=opt-avg -o - -S test.ll
```

```
define i32 @avg(i32 %x, i32 %y) {
entry:
  %0 = add i32 %x, %y
  %1 = and i32 %x, %y
  %2 = xor i32 %x, %y
  %3 = lshr i32 %2, 1
  %4 = add i32 %1, %3
  %5 = udiv i32 %0, 2
  ret i32 %4
}
```

Subtask 1

Dead Code Elimination (DCE)

Dead Code Elimination (1/3)

- Optimizations often leave unused instructions behind
- Instructions are considered "dead" if
  - Their result is not used by any other instruction
  - They do not produce observable side effects (e.g. writing to memory, returning, etc.)

```
define dso_local i32 @main(i32 %argc, ptr %argv) {
entry:
  %add = add nsw i32 %argc, 42
  %mul = mul nsw i32 %add, 2
  ret i32 0
}
```

↓

```
define dso_local i32 @main(i32 %argc, ptr %argv) {
entry:
  ret i32 0
}
```

Hint: Take a look at `llvm/Transforms/Utils/Local.h`

Dead Code Elimination (2/3)

- Eliminate useless basic blocks, useless unconditional jumps
- This step might then allow for elimination of more variables

```
 br i1 %cmp, label %cond.true, label %cond.false

cond.true:
  br label %cond.end

cond.false:
  br label %cond.end

cond.end:
  ret i32 0
```

```
br i1 %cmp, label %cond.end, label %cond.end

cond.end:
  ret i32 0
```

```
br label %cond.end

cond.end:
  ret i32 0
```

Dead Code Elimination (3/3)

- Eliminate useless loads/stores to local stack slots
- Walk backwards in a function, check if the stored value is read again
  - Eliminate useless stores
  - This might allow to eliminate other instructions

```
entry:
  %x = alloca i32
  store i32 10, ptr %x
  %0 = load i32, ptr %x
  store i32 %0, ptr %x
  ret void
```

```
entry:
  %x = alloca i32
  store i32 10, ptr %x
  %0 = load i32, ptr %x
  ret void
```

```
entry:
  %x = alloca i32
  store i32 10, ptr %x
  ret void
```

```
entry:
  %x = alloca i32
  ret void
```

```
entry:
  ret void
```

Subtask 2

Memory Safety

Address Sanitizer

- We're building a basic version of Google's Address Sanitizer [3]
- Our Sanitizer should detect these bugs:
  - Spatial: buffer overflows
  - Temporal: use-after-free
- We build two components:
  - LLVM pass
  - Runtime library
- For simplicity:
  - You only need to detect buffer overflows of <= 16 bytes
  - You only need to detect use-after-frees until malloc reuses a memory block

[3]: https://github.com/google/sanitizers/wiki/AddressSanitizer

LLVM Pass

- Replace calls to malloc/free with calls to our runtime
- Detect memory accesses, insert calls to runtime library

```
%0 = call ptr @malloc(i64 16)
store i32 0, ptr %0
```

```
%0 = call ptr @__runtime_malloc(i64 16)
call void @__runtime_check_addr(ptr %0, i8 4)
store i32 0, ptr %0
```

Access size

Pointer to check

LLVM Pass

- Useful functions:
  - `Module::getOrInsertFunction`
    - Get a reference to your runtime function
  - `IRBuilder::createCall`
    - Create a call to the function you got with `getOrInsertFunction`
  - `Instruction::replaceAllUsesWith`
  - `Instruction::eraseFromParent`

Runtime library

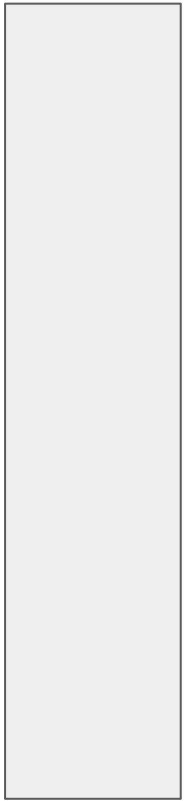
- Export functions that will be called from user code
- For simplicity, the runtime library will be linked against the test code
- You only need to check memory that has been allocated by your runtime library
- Provide functions such as
  - `__runtime_init`
  - `__runtime_check_addr`
  - `__runtime_malloc`
  - `__runtime_free`
  - ...
  - Feel free to add whatever functions your implementation requires
- Additional task
  - Check stack allocations (alloca; spatial checks only) as well as heap allocations

Shadow memory

- Start with a naive implementation
  - Allowed to be slow/inefficient
  - Will give you most of the points
  - Example: store memory regions in a list/map
- Optimization
  - Implement shadow memory
  - Use `mmap` to allocate memory
    - If an allocated shadow page is never used, no physical page will be allocated
    - You can allocate ⅛ of the whole userspace
  - Every byte of memory can be mapped to one bit of shadow memory

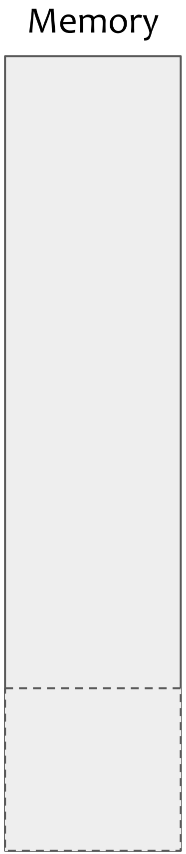
Shadow memory

Memory



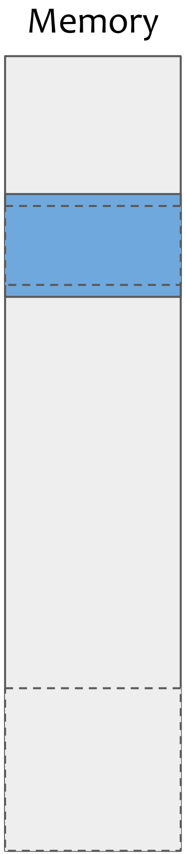
Shadow memory

- Initialize runtime



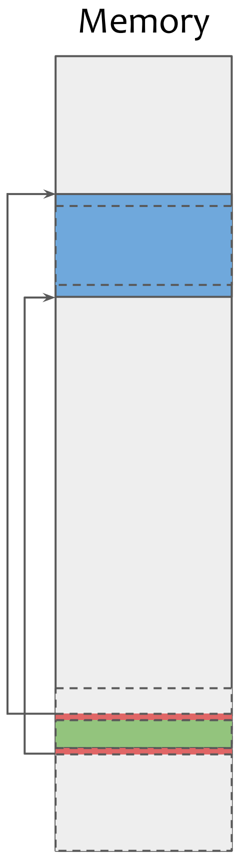
Shadow memory

- Initialize runtime
- Allocate memory (padded)



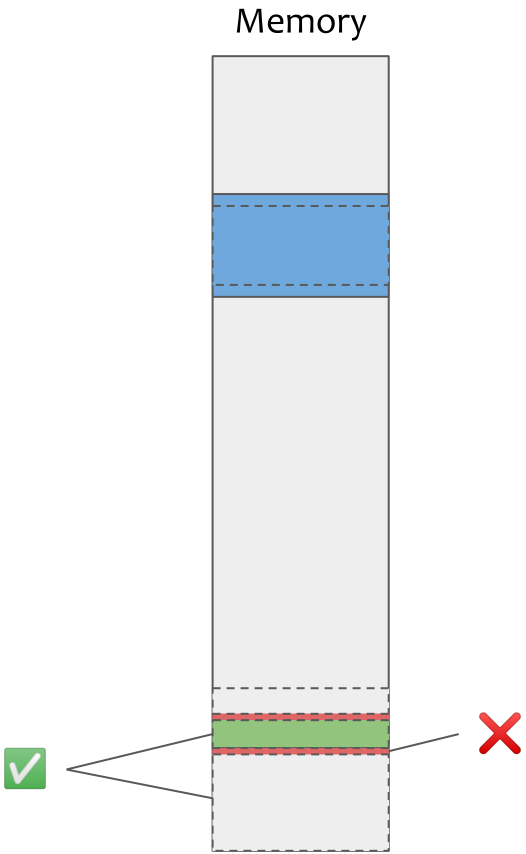
Shadow memory

- Initialize runtime
- Allocate memory (padded)
  - Mark as accessible in shadow memory
  - Mark bounds as inaccessible



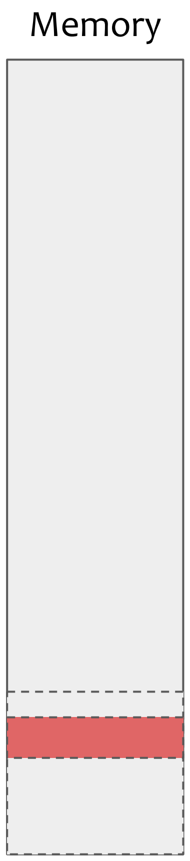
Shadow memory

- Initialize runtime
- Allocate memory (padded)
    - Mark as accessible in shadow memory
    - Mark bounds as inaccessible
- On access, check shadow memory
    - If accessible, allow
    - If inaccessible, print error message and `exit(1)`



Shadow memory

- Initialize runtime
- Allocate memory (padded)
  - Mark as accessible in shadow memory
  - Mark bounds as inaccessible
- On access, check shadow memory
  - If accessible, allow
  - If inaccessible, print error message and `exit(1)`
- On free, mark whole region as inaccessible
  - This allows to detect use-after-free errors



Useful tips

- The LLVM passes can only be implemented in C++, not Rust/C
  - The passes only require basic C++ knowledge
  - The runtime library can be implemented in C/C++/Rust
- Before starting on the task, skim through the documentation to see which methods are available in the `Instruction`, `IRBuilder`, `BasicBlock` classes
- Get familiar with LLVM IR (e.g. through Compiler Explorer)
- Use `errs() << I` to print instructions, blocks, functions
- Alternatively: Use your debugger to debug passes
- Have fun!

Thank you for listening!

See you in the Q&A session!