Compilation using LLVM

Juan Manuel Martinez Caamaño (@jmmartinez)

Quarkslab

Objective of the whole series

The objective of this course is to learn pratcical compilation

- Focused on **Clang/LLVM**
  - Widespread industrial toolchain
  - Most of the concepts are still valid for other toolchains
- Focused on security
  - Attack-Defense cycle

Today's objective

Introduction to a modern compiler toolchain

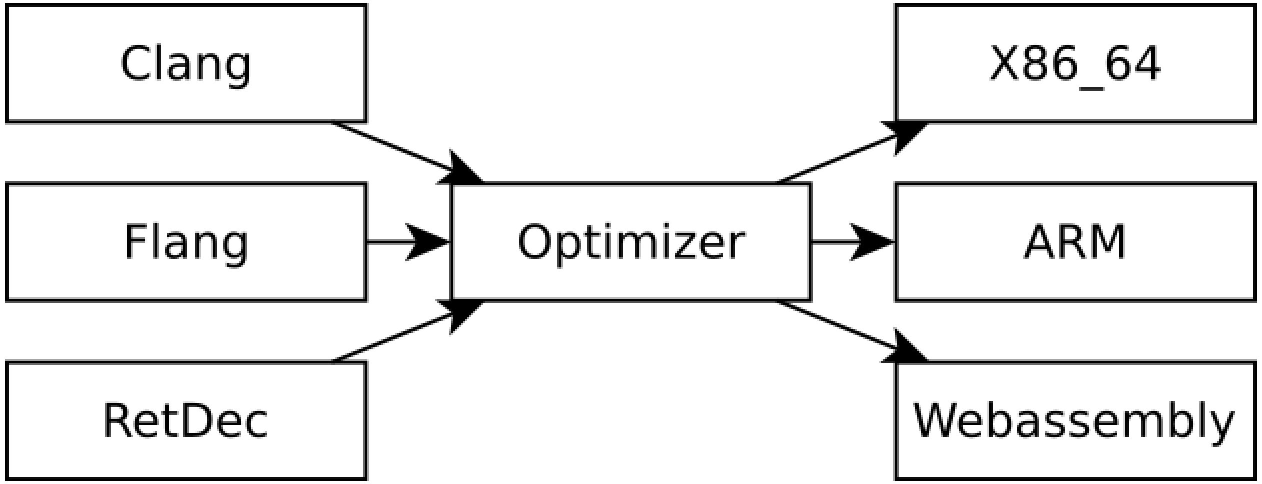
- 3 stage compiler
- First LLVM passes: Mixed-Boolean-Arithmetic

Overview of the 3 stage compiler

The 3 stage compiler

- Frontend: Syntactic and semantic analysis. Few transformations are done here
- Middle-end: Performs most of the code optimizations
- Backend: Generate assembly code. Performs some target dependant optimizations

The LLVM 3 stage compiler




```
ccache clang -S -emit-llvm -Xclang -disable-O0-optnone main.c -o - | opt -S -O2 | llc -o main.s
```

The Frontend

```
ccache clang -Xclang -ast-dump -fsyntax-only main.c
ccache clang -S -emit-llvm -Xclang -disable-O0-optnone main.c -o -
```

The AST

```
`-FunctionDecl 0x55aab514dec8 </home/jmmartinez/Downloads/uba/course0/main.c:3:1, line:6:1> line:3:5 main 'int (int, char **)'
  |-ParmVarDecl 0x55aab514dd78 <col:10, col:14> col:14 argc 'int'
  |-ParmVarDecl 0x55aab514ddf0 <col:20, col:27> col:27 argv 'char **'
  `-CompoundStmt 0x55aab514e0b8 <col:33, line:6:1>
    |-CallExpr 0x55aab514e020 <line:4:3, col:25> 'int'
    | |-ImplicitCastExpr 0x55aab514e008 <col:3> 'int (*)(const char *)' <FunctionToPointerDecay>
    | | `-DeclRefExpr 0x55aab514df78 <col:3> 'int (const char *)' Function 0x55aab5149dc0 'puts' 'int (const char *)'
    | `-ImplicitCastExpr 0x55aab514e068 <col:8> 'const char *' <BitCast>
    |   `-ImplicitCastExpr 0x55aab514e050 <col:8> 'char *' <ArrayToPointerDecay>
    |     `-StringLiteral 0x55aab514dfa0 <col:8> 'char [15]' lvalue "Hello, world!\n"
    `-ReturnStmt 0x55aab514e0a0 <line:5:3, col:10>
      `-IntegerLiteral 0x55aab514e080 <col:10> 'int' 0
```

The Intermediate Representation

```
; ModuleID = '/home/jmmartinez/Downloads/uba/course0/main.c'
source_filename = "/home/jmmartinez/Downloads/uba/course0/main.c"
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

@.str = private unnamed_addr constant [15 x i8] c"Hello, world!\0A\00", align 1

; Function Attrs: noinline nounwind uwtable
define i32 @main(i32, i8**) #0 {
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  %5 = alloca i8**, align 8
  store i32 0, i32* %3, align 4
  store i32 %0, i32* %4, align 4
  store i8** %1, i8*** %5, align 8
  %6 = call i32 @puts(i8* getelementptr inbounds ([15 x i8], [15 x i8]* @.str, i32 0, i32 0))
  ret i32 0
}

declare i32 @puts(i8*) #1
```

The Optimizer

```
opt -S -O2 main.ll
```

The Intermediate Representation (Optimized)

```
; ModuleID = '<stdin>'
source_filename = "/home/jmmartinez/Downloads/uba/course0/main.c"
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

@.str = private unnamed_addr constant [15 x i8] c"Hello, world!\0A\00", align 1

; Function Attrs: noinline nounwind uwtable
define i32 @main(i32, i8** nocapture readnone) local_unnamed_addr #0 {
  %3 = tail call i32 @puts(i8* getelementptr inbounds ([15 x i8], [15 x i8]* @.str, i64 0, i64 0))
  ret i32 0
}

; Function Attrs: nounwind
declare i32 @puts(i8* nocapture readonly) local_unnamed_addr #1
```

The Backend

```
llc main.ll -o main.s
```

The Backend

```
main:                                   # @main
        .cfi_startproc
# %bb.0:
        push    rbp
        .cfi_def_cfa_offset 16
        .cfi_offset rbp, -16
        mov     rbp, rsp
        .cfi_def_cfa_register rbp
        mov     edi, offset .L.str
        call    puts
        xor     eax, eax
        pop     rbp
        ret
.Lfunc_end0:
        .size   main, .Lfunc_end0-main
        .cfi_endproc
                                        # -- End function
        .type   .L.str,@object          # @.str
        .section        .rodata.str1.1,"aMS",@progbits,1
.L.str:
        .asciz  "Hello, world!\n"
        .size   .L.str, 15
```

The Backend

The assembly is not directly executable as it is !

Functions or global variables that are imported from other modules

Linking

Object files contain the binary code and additional metadata

- Lists of symbols exported/imported
- How symbols map to sections
- Relocations

The linking stage combines multiple object files into a binary

- Associates symbols defined in an object file with symbols required by another
- Takes care of the layout of the binary

Linking

A more complex example

```c
int main(int argc, char** argv) {
  if(argc < 2)
    return -1;

  const char* user = argv[1];
  if(!validate(user)) // defined in another .c
    return -1;

  printf("Hello %s!\n", user);
  return 0;
}
```

```c
int validate(const char* user) {
  return strcmp(user, "juan") == 0;
}
```

Linking

```
clang link_main.c -c -o link_main.o
clang link_lib.c  -c -o link_lib.o
clang link_main.o link_lib.o -o lib
```

Linking

```
readelf -s link_main.o
```

```

Symbol table '.symtab' contains 7 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS link_main.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    2
     3: 0000000000000000     0 SECTION LOCAL  DEFAULT    4
     4: 0000000000000000   126 FUNC    GLOBAL DEFAULT    2 main
     5: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND printf
     6: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND validate
```

Linking

```
ldd ./lib
```

```
	linux-vdso.so.1 (0x00007ffcb97bb000)
	libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f3a3db69000)
	/lib64/ld-linux-x86-64.so.2 (0x00007f3a3df5a000)
```

Linking

```
readelf -s /lib/x86_64-linux-gnu/libc.so.6 | grep ' printf'
```

```
   627: 0000000000064e80   195 FUNC    GLOBAL DEFAULT   13 printf@@GLIBC_2.2.5
  1559: 0000000000064da0    28 FUNC    GLOBAL DEFAULT   13 printf_size_info@@GLIBC_2.2.5
  1983: 00000000000642c0  2770 FUNC    GLOBAL DEFAULT   13 printf_size@@GLIBC_2.2.5
```

Linking

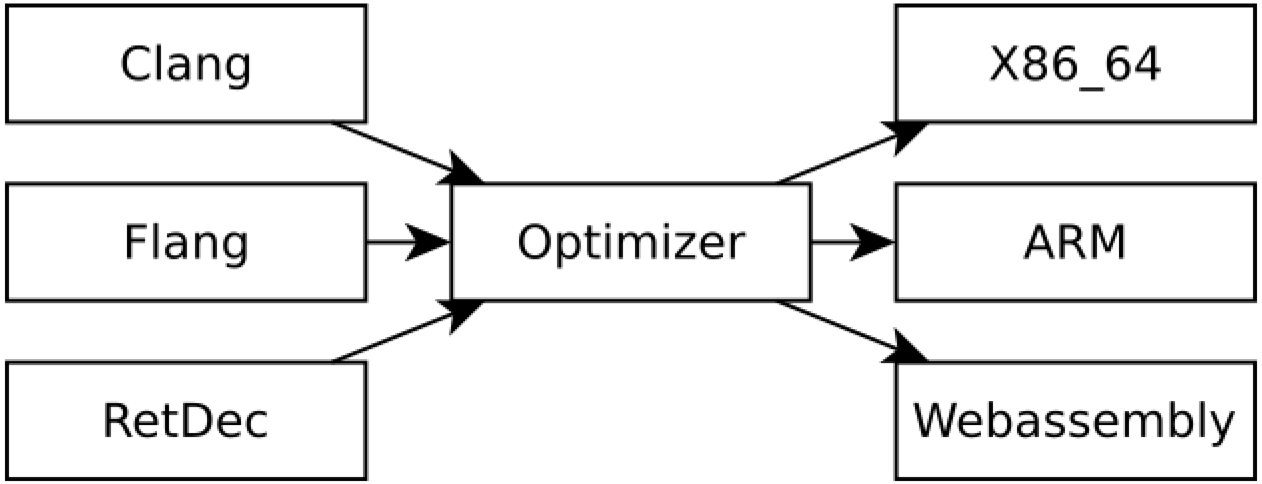
We're not going to focus much more on this stage

The LLVM

What is the LLVM ?

It's the specification of an **intermediate representation** (LLVM-IR) and an umbrella of projects that communicate using the IR.

llvm.org/docs/LangRef.html



The Module

```
; ModuleID = '<stdin>'
source_filename = "/home/jmmartinez/Downloads/uba/course0/main.c"
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

@.str = private unnamed_addr constant [15 x i8] c"Hello, world!\0A\00", align 1

; Function Attrs: noinline nounwind uwtable
define i32 @main(i32, i8** nocapture readnone) local_unnamed_addr #0 {
  %3 = tail call i32 @puts(i8* getelementptr inbounds ([15 x i8], [15 x i8]* @.str, i64 0, i64 0))
  ret i32 0
}

; Function Attrs: nounwind
declare i32 @puts(i8* nocapture readonly) local_unnamed_addr #1
```

The Module

Why the *target triple* and *data layout* if the IR is portable across targets ?

RISC like ISA

- Strongly typed (no implicit type casts)
- No signed types. Signed operations when needed: for example `div` and `sdiv`
- Basic atomic types like : `i32`, `i1`, `float`, `double`, `i128`, `i8*`, `void`
- Some special types: `token`

RISC like ISA

```
long polynome(int x) {
  return 2*x*x*x + 7*x*x + 9*x + 1234;
}
```

```
; Function Attrs: norecurse nounwind readnone uwtable
define i64 @polynome(i32) local_unnamed_addr #0 {
  %2 = shl i32 %0, 1
  %3 = add i32 %2, 7
  %4 = mul i32 %3, %0
  %5 = add i32 %4, 9
  %6 = mul i32 %5, %0
  %7 = add nsw i32 %6, 1234
  %8 = sext i32 %7 to i64
  ret i64 %8
```

RISC like ISA

```c
double polynome(float x) {
  return 2*x*x*x + 7*x*x + 9*x + 1234;
}
```

```llvm
; Function Attrs: norecurse nounwind readnone uwtable
define double @polynome(float) local_unnamed_addr #0 {
  %2 = fmul float %0, 2.000000e+00
  %3 = fmul float %2, %0
  %4 = fmul float %3, %0
  %5 = fmul float %0, 7.000000e+00
  %6 = fmul float %5, %0
  %7 = fadd float %6, %4
  %8 = fmul float %0, 9.000000e+00
  %9 = fadd float %8, %7
  %10 = fadd float %9, 1.234000e+03
  %11 = fpext float %10 to double
  ret double %11
```

Control-Flow

A basic block is a list of instructions that execute sequentially until a terminator is found

Very few basic terminators:

- `br` , `ret` , `switch`
- `invoke` , `resume` , `catchswitch` , `catchret` , `cleanupret`
- `unreachable`
- `indirectbr`

Control-Flow

Call instructions:

- `call`, `callbr`
- `invoke` again

Control-Flow

```
void then_(int);
void else_(int);
void if_then_else(int a, int b, int c) {
  if(a) then_(b);
  else else_(c);
}
```

```
  %4 = icmp eq i32 %0, 0
  br i1 %4, label %6, label %5

; <label>:5:                                      ; preds = %3
  tail call void @then_(i32 %1) #2
  br label %7

; <label>:6:                                      ; preds = %3
  tail call void @else_(i32 %2) #2
  br label %7
```

PHI-Nodes

The LLVM-IR is a Static Single Assignment (SSA) intermediate representation

```
int then_(int);
int else_(int);
int if_then_else(int a, int b, int c) {
  int y; // y_0
  if(a)
    y = then_(b); // y_1
  else
    y = else_(c); // y_2
  return y; // ?
}
```

PHI-Nodes

```
define i32 @if_then_else(i32, i32, i32) local_unnamed_addr #0 {
  %4 = icmp eq i32 %0, 0
  br i1 %4, label %7, label %5

; <label>:5:                                      ; preds = %3
  %6 = tail call i32 @then_(i32 %1) #2
  br label %9

; <label>:7:                                      ; preds = %3
  %8 = tail call i32 @else_(i32 %2) #2
  br label %9

; <label>:9:                                      ; preds = %7, %5
  %10 = phi i32 [ %6, %5 ], [ %8, %7 ]
  ret i32 %10
}
```

Memory

Very few instructions to access memory

- `load`
- `store`
- `cmpxchg`
- `atomicrmw [add, sub, xor, max, ...]`

```
; Function Attrs: norecurse nounwind uwtable
define void @inc(i64, i32) local_unnamed_addr #0 {
  %3 = getelementptr inbounds [500 x i32], [500 x i32]* @a, i64 0, i64 %0
  %4 = load i32, i32* %3, align 4, !tbaa !2
  %5 = add nsw i32 %4, %1
  store i32 %5, i32* %3, align 4, !tbaa !2
  ret void
}
```

Complex Types

- vector types like: `<4 x i32>`
- array types like: `i8[256]`
- structs: `%pair_of_ints = type { i32 , i32 }`

Exception Handling

A total mess and very dependant of the traget.

The basis is the `invoke` instruction.

Exception Handling

```
#include <exception>
int maythrow(int);

int catchall(int a) {
  try {
    maythrow(a);
  }
  catch(std::exception &) { return 1; }
  catch(...) { return 2; }
  return 0;
}
```

Exception Handling

```
define i32 @_Z8catchalli(i32) local_unnamed_addr #0 personality i8* bitcast (i32 (...)* @__gxx_personality_v0 to i8*) {
  %2 = invoke i32 @_Z8maythrowi(i32 %0)
          to label %11 unwind label %3

; <label>:3:                                      ; preds = %1
  %4 = landingpad { i8*, i32 }
          catch i8* bitcast (i8** @_ZTISt9exception to i8*)
          catch i8* null
  %5 = extractvalue { i8*, i32 } %4, 0
  %6 = extractvalue { i8*, i32 } %4, 1
  %7 = tail call i32 @llvm.eh.typeid.for(i8* bitcast (i8** @_ZTISt9exception to i8*)) #3
  %8 = icmp eq i32 %6, %7
  %9 = tail call i8* @__cxa_begin_catch(i8* %5) #3
  tail call void @__cxa_end_catch()
  %10 = select i1 %8, i32 1, i32 2
  br label %11

; <label>:11:                                     ; preds = %1, %3
  %12 = phi i32 [ %10, %3 ], [ 0, %1 ]
  ret i32 %12
}
```

The LLVM Optimizer

Hooking in the LLVM optimizer

Transformations in the LLVM are implemented as compiler **passes**.

They define a `runOn` function and specify the analysis that the pass requires.

Passes are executed in a sequential order one after the other.

A First LLVM Pass

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

  bool runOnFunction(Function &F) override {
    errs() << "Hello: " << F.getName() << "\n";
    return false;
  }
};
char Hello::ID = 0;
}
```

A First LLVM Pass

```
// register in opt's command line
static RegisterPass<Hello> X("hello", "Hello World Pass",
                             false /* Only looks at CFG */,
                             false /* Analysis Pass */);

// register in the default pass pipeline
static RegisterStandardPasses Y(
    PassManagerBuilder::EP_EarlyAsPossible,
    [](const PassManagerBuilder &Builder,
       legacy::PassManagerBase &PM) { PM.add(new Hello()); });
```

Code Obfuscation

Reverse Engineering

The objective is to recover a certain property from a binary

- An equivalent version of an algorithm in a high level language
- A secret key
- An API protocol

Reverse Engineering

The attacker is in total control of the executable and it's environment.

He/she can dissasemble, run, and debug the application.

Code Obfuscation

The objective is:

- Generate a new program equivalent to the original
- Attacker cannot get more information about certain property from looking at the binary than by looking at its input / output.

Code Obfuscation

In practice:

- It's not a 100% guaranteed protection
- It won't fix vulnerabilities in the code
- It's not automatic

Code Obfuscation

What we pay:

- Slower execution
- Bigger binary
- Bigger memory consumption
- Bigger compile times

Code Obfuscation

What we win:

- Slow down reverse engineering

Mixed Boolean

Transform an operation into a sequence of logical and arithmetic operations

```
A+B == (A & B)<<1 + (A ^ B)
A-B == (A & -B)<<1 + (A ^ -B)
A^B == A + B - (A & B)<<1
```

Opaque Predicates

`X` is something random from the context.
`P(X)` is an expression that always yields `1` and that is difficult to analyze.

Opaque Predicates

Use a simple mathematical property: `log2(x) == log10(x) / log10(2)`

- Reverse engieering tools have poor support for floating point

Opaque Predicates

Or properties depending on series: approximate `pi` and check that the error is smaller than a certain bound.

- Reverse engieering tools have poor support for floating point
- Difficult to understand
- Can make tools timeout

Opaque Predicates

Use it to build opaque constants

```
pi = 4 * (1 - 1/3 + 1/5 - 1/7 + ... + 1/N)
```

After a certain amount of iterations `N`, the series approximates pi with an error smaller than 0.2

How to test obfuscations ?

- Unit tests
- Check that the obfuscation survives optimizations
- Fuzzing
- Reproductivility tests
- Apply on a real code base

For the curious

- From Clang's AST to LLVM-IR for a call: `CodeGenFunction::EmitCall` in `clang/lib/CodeGen/CGCall.cpp`
- LLVM optimization pipeline: `llvm/lib/Transforms/IPO/PassManagerBuilder.cpp` (this is not the full pipeline)

Conclusions

- Stable 3-stage architecture
- Perform target independant optimizations in the IR
- Code obfuscation slows reverse engineering
- Performance/Memory/Size tradeoff