An introduction to LLVM

Béatrice Creusillet
bcreusillet@quarkslab.com

Quarkslab

LLVM is not (just) a Compiler!

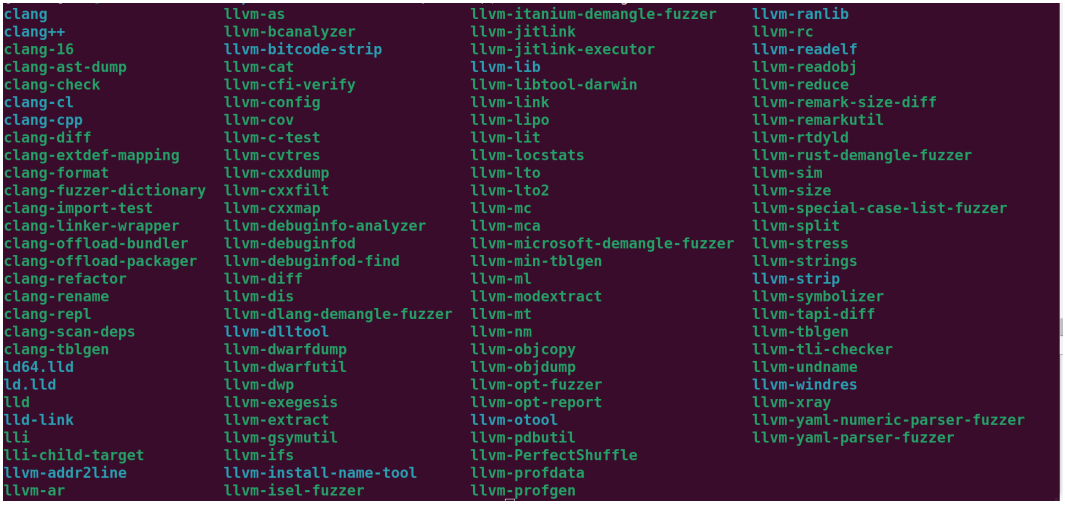
An introduction to LLVM

Béatrice Creusillet
bcreusillet@quarkslab.com

Quarkslab

Table of Contents

ls 'llvm/build/bin'



A *de facto* standard compiler infrastructure

The LLVM Project is a collection of **modular** *and* **reusable compiler** *and* **toolchain** *technologies.*

(llvm.org)

Main sub-projects:

- The **LLVM Core** library
- **Clang**
- **LLDB**
- **LLD**
- **libc**, **libc++** and **libc++ ABI**
- **compiler-rt**
- **MLIR**
- ...

Building compilers with LLVM

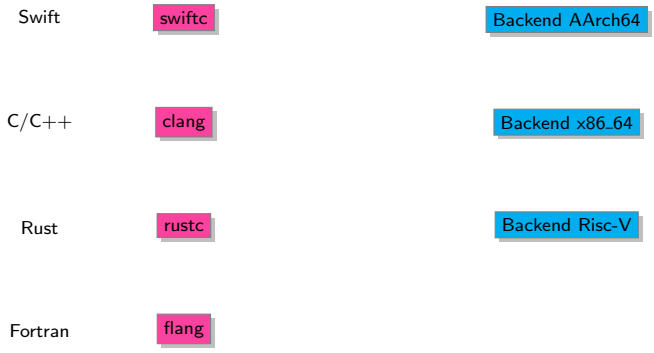
Swift

C/C++

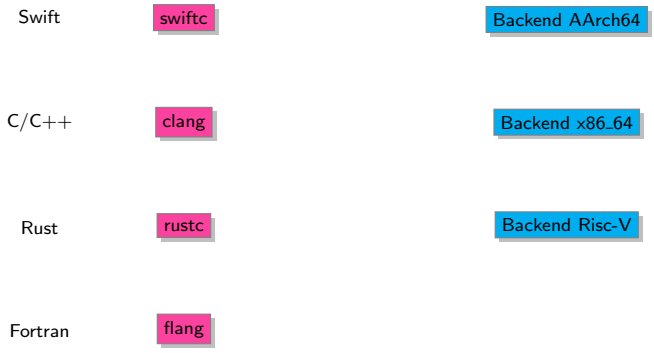
Rust

Fortran

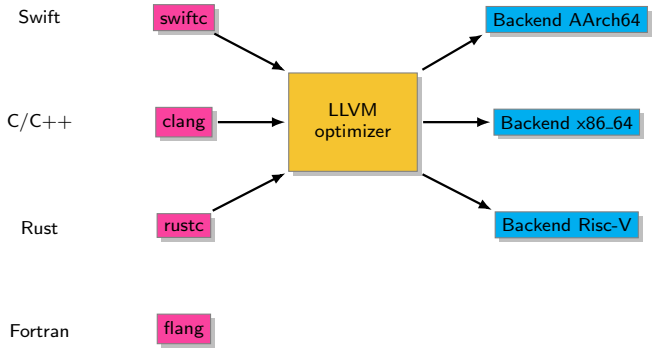
Building compilers with LLVM



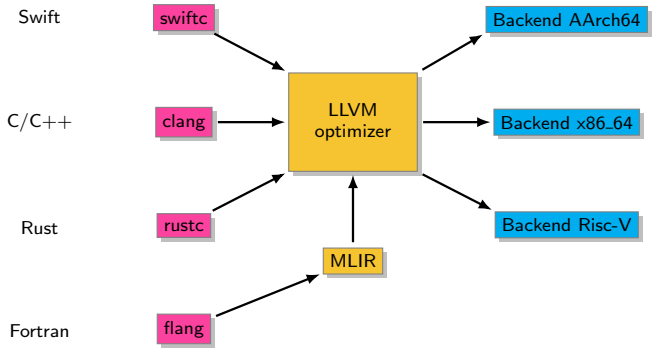
Building compilers with LLVM



Building compilers with LLVM



Building compilers with LLVM



Building compilers with LLVM

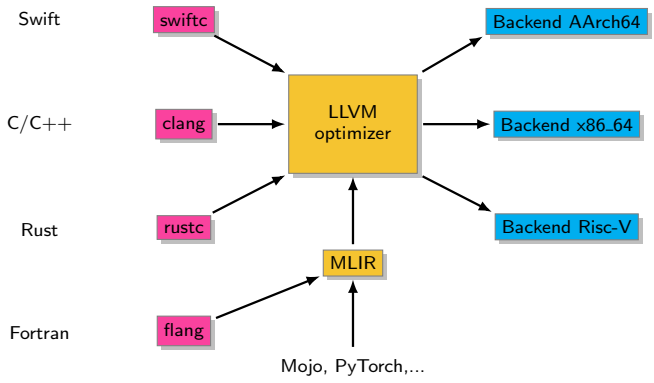
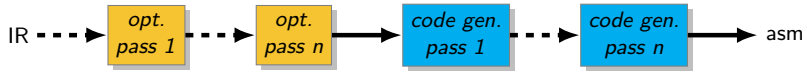
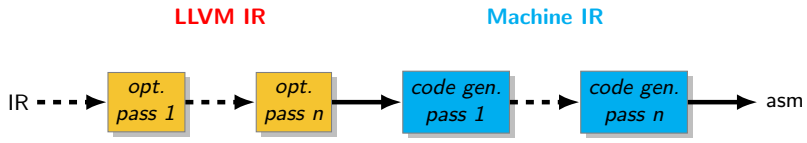


Table of Contents

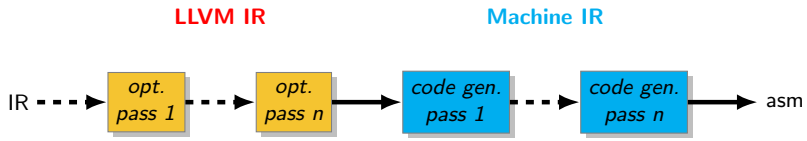
The LLVM Optimizer



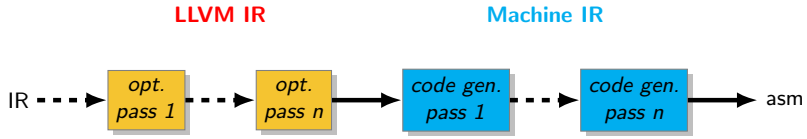
The LLVM Optimizer



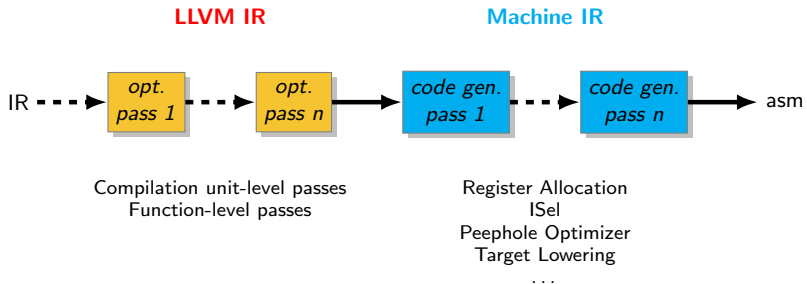
The LLVM Optimizer



The LLVM Optimizer



The LLVM Optimizer



- Pre-defined pass pipelines: '-O0', '-O1', '-O2', '-O3', '-Os', '-Oz'
- Passes can be modified by options
- Passes can be added to the pipelines with options
- Pass scheduling handled by pass managers

https://llvm.org/docs/Passes.html
https://llvm.org/doxygen/PassBuilderPipelines_8cpp_source.html
https://llvm.org/docs/WritingAnLLVMBackend.html

PassBuilderPipelines.cpp: A perpetual WIP

```
1  // TODO: Investigate the cost/benefit of tail call elimination on debugging.
2  FunctionPassManager
3  PassBuilder::buildO1FunctionSimplificationPipeline(OptimizationLevel Level,
4                                                      ThinOrFullLTOPhase Phase) {
5
6    FunctionPassManager FPM;
7
8    ...
9
10   // TODO: Investigate promotion cap for O1.
11   LPM1.addPass(LICMPass(PTO.LicmMssaOptCap, PTO.LicmMssaNoAccForPromotionCap,
12                         /*AllowSpeculation=*/false));
13
14   // Sparse conditional constant propagation.
15   // FIXME: It isn't clear why we do this *after* loop passes rather than
16   // before...
17   FPM.addPass(SCCPPass());
18
19   ...
20
21   // Finally, do an expensive DCE pass to catch all the dead code exposed by
22   // the simplifications and basic cleanup after all the simplifications.
23   // TODO: Investigate if this is too expensive.
24   FPM.addPass(ADCEPass());
25
26   ...
```

Compilation unit-level transformations

Interprocedural transformations

- Call Graph modifications
- Inlining, outlining, cloning, merging functions, global dce,...

Global variable modifications

- GlobalOpt, ConstantMerge, GlobalDCE,...

Function-level transformations

- Constant propagation, Common subexpressions elimination, loop transformations, ...
- Requires instructions to be moved relative to one another
- respecting their dependencies :
  - **spatial**
    (am I referencing the same memory area or register?)
  - **temporal**
    (is this store executed before this load?)

```
1        a = ...;
2        ... = a;
3        a = ...;
```

LLVM provides this information in various ways:

- its **internal representation** (LLVM IR)
  - Basic blocks and CFG
  - SSA form
    transforms register dependencies into value chains (use-defs chains)
    beware: does not represent memory dependencies!
- **analyses** that transformation passes can request/query
  - AliasAnalysis (AA)
  - MemorySSA
  - ...

Table of Contents

The LLVM Internal Representation

A **language**:

- assembly-like
- generic though very C-oriented
  - goal: represent *all* high-level languages
- strongly typed
  - except for pointers...
- in SSA form
- 3 forms:
  - in-memory
  - human readable (`.ll` files)
  - on-disk bitcode (`.bc` files)

See https://llvm.org/docs/LangRef.html for full description

Main syntax elements

Module

- compilation unit
- contains global variables and functions
- metadata

Functions

- arguments
- list of Basic Blocks

Basic Blocks

- consecutive instructions
- a terminator (branch instruction, return, invoke,...)
- successors, predecessors

Instructions

- operators, calls, return, branching, select,...
- PHINode
- **!!! no assignment !!!**

My first function: types, instructions, values

```
1  int foo(int a, int b) {
2    return a + b;
3  }
```

Simplified llvm-ir:

```
1 define i32 @foo(i32 noundef %a, i32 noundef %b) {
2   %c = add nsw i32 %b, %a
3   ret i32 %c
4 }
```

```
$ clang-18 -O3 add.c -S -emit-llvm fno-discard-value-names -o -
```

```
; ModuleID = 'add.c'
source_filename = "add.c"
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-i128:128-f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

; Function Attrs: mustprogress nofree norecurse nosync nounwind willreturn memory(none) uwtable
define dso_local i32 @foo(i32 noundef %0, i32 noundef %1) local_unnamed_addr #0 {
  %3 = add nsw i32 %1, %0
  ret i32 %3
}

attributes #0 = { mustprogress nofree norecurse nosync nounwind willreturn memory(none) uwtable "min-legal-vector-width"="0" "no-trapping-math"="true" "stack-protector-buffer-size"="8" "target-cpu"="x86-64" "target-features"="+cmov,+cx8,+fxsr,+mmx,+sse,+sse2,+x87" "tune-cpu"="generic" }

!llvm.module.flags = !{!0, !1, !2, !3}
!llvm.ident = !{!4}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{i32 8, !"PIC Level", i32 2}
!2 = !{i32 7, !"PIE Level", i32 2}
!3 = !{i32 7, !"uwtable", i32 2}
!4 = !{!"Ubuntu clang version 18.1.3 (1ubuntu1)"}
```

```
$ clang-18 -O0 add.c -S -emit-llvm fno-discard-value-names -o -
```

```
; ModuleID = 'add.c'
source_filename = "add.c"
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-i128:128-f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @foo(i32 noundef %0, i32 noundef %1) #0 {
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  store i32 %0, ptr %3, align 4
  store i32 %1, ptr %4, align 4
  %5 = load i32, ptr %3, align 4
  %6 = load i32, ptr %4, align 4
  %7 = add nsw i32 %5, %6
  ret i32 %7
}

attributes #0 = { noinline nounwind optnone uwtable "frame-pointer"="all" "min-legal-vector-width"="0" "no-trapping-math"="true" "stack-protector-buffer-size"="8" "target-cpu"="x86-64" "target-features"="+cmov,+cx8,+fxsr,+mmx,+sse,+sse2,+x87" "tune-cpu"="generic" }

!llvm.module.flags = !{!0, !1, !2, !3, !4}
!llvm.ident = !{!5}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{i32 8, !"PIC Level", i32 2}
!2 = !{i32 7, !"PIE Level", i32 2}
!3 = !{i32 7, !"uwtable", i32 2}
```

Control flow: basic blocks, branches, predecessors, successors

```
1  extern void bar(int a);
2  extern void foo(int a);
3
4  void foobar(int a, int b,
5              int cond) {
6    if (a)
7      bar(a);
8    else
9      foo(b);
10 }
```

Control flow: basic blocks, branches, predecessors, successors

```c
extern void bar(int a);
extern void foo(int a);

void foobar(int a, int b,
            int cond) {
  if (a)
    bar(a);
  else
    foo(b);
}
```

```llvm
define void @foobar(i32 %a, i32 %b, i32 %cond) {

entry:
  %tobool.not = icmp eq i32 %a, 0
  br i1 %tobool.not, label %if.else, label %if.then

if.then:                ; preds = %entry

  tail call void @bar(i32 %a) #2
  br label %if.end

if.else:                ; preds = %entry

  tail call void @foo(i32 %b) #2
  br label %if.end

if.end:                 ; preds = %if.else, %if.then

  ret void
}
```

SSA: Static Single Assignment form

```
1  extern int foo(int in);
2  extern int bar(int in);
3
4  int foo_or_bar(int a,
5                 int cond){
6    if (cond) {
7      a = foo(a);
8    } else {
9      a = bar(a);
10   }
11   return a;
12 }
```

SSA: Static Single Assignment form

```
1  extern int foo(int in);
2  extern int bar(int in);
3
4  int foo_or_bar(int a,
5                 int cond){
6    if (cond) {
7      a = foo(a);
8    } else {
9      a = bar(a);
10   }
11   return a;
12 }
```

```
1  define i32 @foo_or_bar(i32 %a, i32 %cond) {
2  entry:
3    %res_cond = icmp eq i32 %cond, 0
4    br i1 %res_cond, label %then, label %else
5
6  then:                             ; preds = %entry
7
8    %a  = tail call i32 @foo(i32 %a) #2
9    br label %return
10
11 else:                             ; preds = %entry
12
13   %a  = tail call i32 @bar(i32 %a) #2
14   br label %return
15
16 return:                           ; preds = %then, %else
17
18
19   ret i32 %a
20 }
```

SSA: Static Single Assignment form

```
1  extern int foo(int in);
2  extern int bar(int in);
3
4  int foo_or_bar(int a,
5                 int cond){
6    if (cond) {
7      a = foo(a);
8    } else {
9      a = bar(a);
10   }
11   return a;
12 }
```

```
1  define i32 @foo_or_bar(i32 %a, i32 %cond) {
2  entry:
3    %res_cond = icmp eq i32 %cond, 0
4    br i1 %res_cond, label %then, label %else
5
6  then:                                ; preds = %entry
7
8    %a1 = tail call i32 @foo(i32 %a) #2
9    br label %return
10
11 else:                                ; preds = %entry
12
13   %a2 = tail call i32 @bar(i32 %a) #2
14   br label %return
15
16 return:                              ; preds = %then, %else
17
18   %a3 = phi i32 [ %a1, %then ], [ %a2, %else ]
19   ret i32 %a3
20 }
```

for loops

```
1  void loop(int* a, unsigned int n) {
2    for (int i = 0; i<n; ++i) {
3      a[i] = i;
4    }
5  }
```

for loops

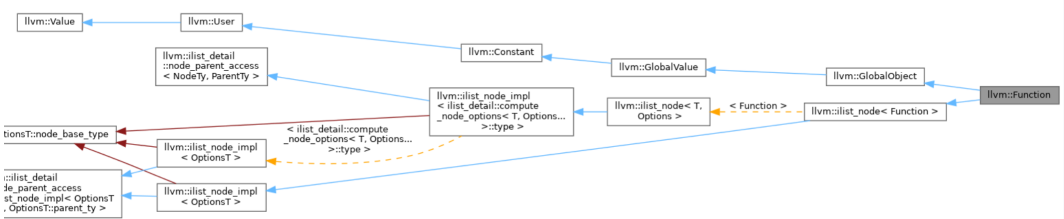
```
$ clang-18 -O1 loop.c -S -emit-llvm -o - -fno-discard-value-names
```

```
define void @loop(ptr %a, i32 %n) {
entry:
  %cmp4.not = icmp eq i32 %n, 0
  br i1 %cmp4.not, label %for.cond.cleanup, label %for.body.preheader

for.body.preheader:                               ; preds = %entry
  %wide.trip.count = zext i32 %n to i64
  br label %for.body

for.cond.cleanup:                                 ; preds = %for.body, %entry
  ret void

for.body:                                         ; preds = %for.body.preheader, %for.body
  %indvars.iv = phi i64 [ 0, %for.body.preheader ],
                        [ %indvars.iv.next, %for.body ]
  %arrayidx = getelementptr inbounds i32, ptr %a, i64 %indvars.iv
  %0 = trunc i64 %indvars.iv to i32
  store i32 %0, ptr %arrayidx, align 4, !tbaa !5
  %indvars.iv.next = add nuw nsw i64 %indvars.iv, 1
  %exitcond.not = icmp eq i64 %indvars.iv.next, %wide.trip.count
  br i1 %exitcond.not, label %for.cond.cleanup, label %for.body, !llvm.loop !9
}
```

Table of Contents

In memory LLVM Intermediate Representation (IR)

- *kind of* enriched Abstract Syntax Tree
- Implemented in C++
- Full doxygen documentation online
- ex: class `Function` (https://llvm.org/doxygen/classllvm_1_1Function.html)

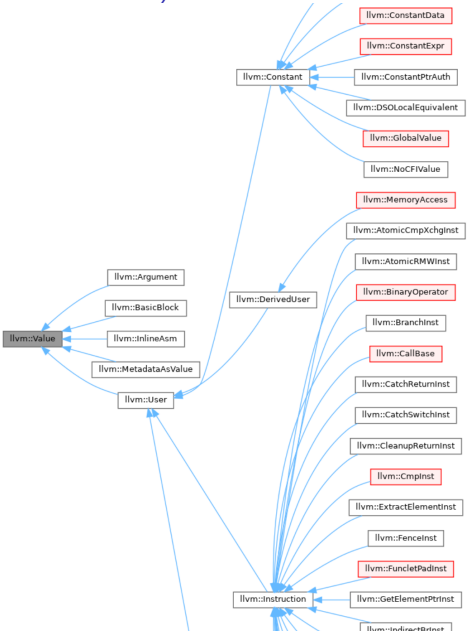


The `Value` class (https://llvm.org/doxygen/classllvm_1_1Value.html)

Almost everything is a Value!

base class of all values computed by a program that may be used as operands to other values.

- functions, global variables
- constants
- basic blocks
- instructions . . .



The `Value` class (https://llvm.org/doxygen/classllvm_1_1Value.html)

Almost everything is a Value!

base class of all values computed by a program that may be used as operands to other values.

- functions, global variables
- constants
- basic blocks
- instructions ...

A Value has Uses

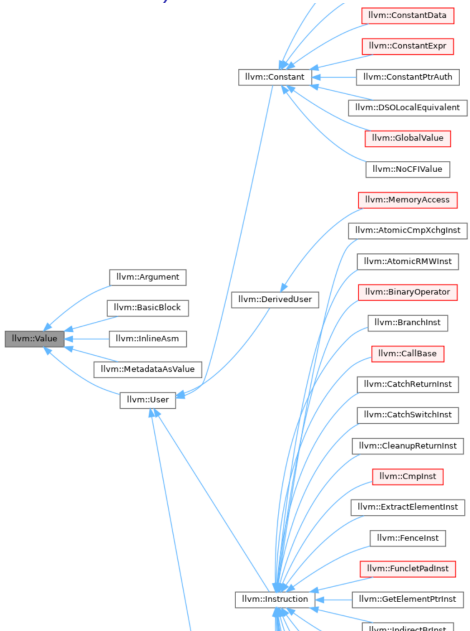
- implements a data flow graph
- can be iterated

  `iterator_range< use_iterator > uses ();`

  ex: `for (auto VUses: V.uses()){...}`
- can be replaced

  `void replaceAllUsesWith(Value *OtherValue);`

  ex: `myValue.replaceAllUsesWith(myNewValue);`



Adding a new function to a Module

```
1 define i32 @my_new_function(i32 %0, i32 %1) {
2 myBB:
3   %2 = add i32 %0, %1
4   ret i32 %2
5 }
```

Adding a new function to a Module

```
define i32 @my_new_function(i32 %0, i32 %1) {
myBB:
  %2 = add i32 %0, %1
  ret i32 %2
}
```

```
Function *createMyNewFunction(Module *M) {

  LLVMContext &Ctx = M.getContext();

  FunctionType *myNewFunctionTy
      = FunctionType::get(IntegerType::get(Ctx, 32),
                          {IntegerType::get(Ctx, 32), IntegerType::get(Ctx, 32)},
                               false);

  Function *myNewFunction = Function::Create(
      myNewFunctionTy, GlobalValue::InternalLinkage, "my_new_function", M);

  myNewFunction->addFnAttr(llvm::Attribute::NoInline);
  return myNewFunction;
}
```

Note: you can also use `Module::getOrInsertFunction(...)` or its variants

Adding a Basic Block to a function

```
1 define i32 @my_new_function(i32 %0, i32 %1) {
2 myBB:
3   %2 = add i32 %0, %1
4   ret i32 %2
5 }
```

Adding a Basic Block to a function

```
define i32 @my_new_function(i32 %0, i32 %1) {
myBB:
  %2 = add i32 %0, %1
  ret i32 %2
}
```

```
void populateMyNewFunction(Function *myNewFunction) {
  LLVMContext &Ctx = myNewFunction->getContext();
  BasicBlock *myBB = BasicBlock::Create(Ctx, "myBB", myNewFunction);

  IRBuilder<> Builder(myBB);

  Argument *arg0 = myNewFunction->getArg(0);
  Argument *arg1 = myNewFunction->getArg(1);

  Value *result = Builder.CreateAdd(arg0, arg1);
  Builder.CreateRet(result);
}
```

Creating control flow

```
define i32 @foobar(i32 %0, i32 %1, i32 %2) {

header:
  br i1 %cond, label %then, label %else

then:                ; preds = %header

  %3 = add i32 %0, %1
  br label %footer

else:                ; preds = %header

  %4 = sub i32 %0, %1
  br label %footer

footer:              ; preds = %else, %then

  %5 = phi i32 [%3, %then], [%4, %else]
  ret %5
}
```

Creating control flow

```
1  Function *functionWithCF(Function *F) {
2    Context &Ctx = myNewFunction.getContext();
3    auto *headerBB = BasicBlock::Create(Ctx, "header", F);
4    auto *thenBB = BasicBlock::Create(Ctx, "then", F);
5    auto *elseBB = BasicBlock::Create(Ctx, "else", F);
6    auto *footerBB = BasicBlock::Create(Ctx, "footer", F);
7
8    Argument *arg0 = F->getArg(0);
9    Argument *arg1 = F->getArg(1);
10   Argument *cond = F->getArg(2);
11
12   IRBuilder<> B(headerBB);
13   B.createCondBr(cond, thenBB, elseBB);
14
15   B.setInsertPoint(thenBB);
16   Value *add = B.createAdd(arg0, arg1);
17   B.createBr(footerBB);
18
19   B.setInsertPoint(elseBB);
20   Value *sub = B.createSub(arg0, arg1);
21   B.createBr(footerBB);
22
23   B.setInsertPoint(footer);
24   PHINode *phi = B.createPHI(arg0.getType(), 2);
25   phi->addIncoming(add, thenBB); phi->addIncoming(sub, elseBB);
26   B.CreateRet(phi);
27 }
```

Creating a new LLVM transformation pass

Where?

- **in tree**
  - clone the LLVM code base
  - put your code deep inside
  - compile everything (might be looong :( )
  - use `clang`, `opt`,... as usual
  - fear LLVM upgrades...
- **out of tree, using pass plugins**
  - install LLVM on your laptop
  - put the code of your pass in a specific directory
  - with a appropriate `CMakeLists.txt` file
  - compile everything into a dynamic library (fast :))
  - use `opt` with `-load-pass-plugin` and `load` options

For more information: https://llvm.org/docs/WritingAnLLVMNewPMPass.html

```cpp
#include "llvm/IR/PassManager.h"
#include "llvm/IR/Function.h"
#include "llvm/Passes/PassBuilder.h"
#include "llvm/Passes/PassPlugin.h"
using namespace llvm;

struct MBA : public llvm::PassInfoMixin<MBA> {
  llvm::PreservedAnalyses run(Function &F, FunctionAnalysisManager &AM);
};

namespace {
  bool addPassToFPM(StringRef Name, FunctionPassManager &FPM,
                    ArrayRef<PassBuilder::PipelineElement>) {
    if (Name == "mypass") {
      FPM.addPass(MyPass());
      return true;
    }
    return false;
  }
}

PassPluginLibraryInfo getMyPassPluginInfo() {
  return {LLVM_PLUGIN_API_VERSION, "MyPass", LLVM_VERSION_STRING,
    [](PassBuilder &PB) { PB.registerPipelineParsingCallback(addPassToFPM); }};
}

extern "C" LLVM_ATTRIBUTE_WEAK ::llvm::PassPluginLibraryInfo
llvmGetPassPluginInfo() {
  return getMyPassPluginPluginInfo();
}
```

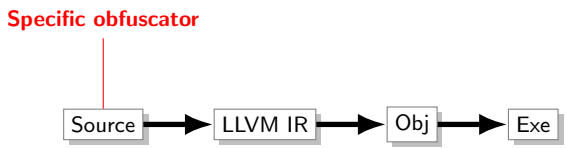
Table of Contents

Obfuscate = Transform = Compile

Obfuscate = Complexify but **Compile = Optimize**

Obfuscate = Transform = Compile

Obfuscate = Complexify but **Compile = Optimize**

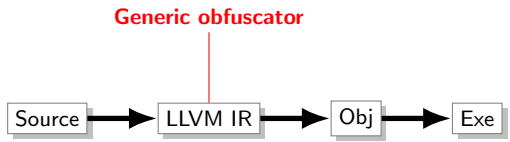


Obfuscate = Transform = Compile

Obfuscate = Complexify but **Compile = Optimize**

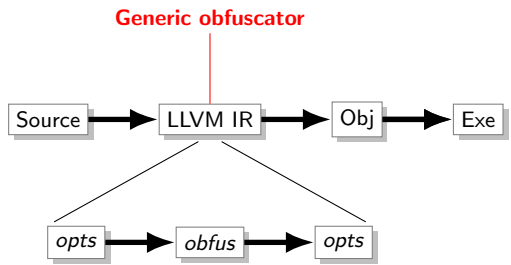
Obfuscate = Transform = Compile

Obfuscate = Complexify but **Compile = Optimize**



Obfuscate = Transform = Compile

Obfuscate = Complexify but **Compile = Optimize**



Protecting operations with MBAs

M

B

A

expressions

Information Hiding in Software with Mixed Boolean-Arithmetic Transforms, Zhou *et al.*, 2007

Protecting operations with MBAs

M

B

A rithmetic $+, -, \times$ $x + y$

expressions

Information Hiding in Software with Mixed Boolean-Arithmetic Transforms, Zhou *et al.*, 2007

Protecting operations with MBAs

M

Boolean	$\wedge$, $\vee$, $\oplus$, $\neg$	$a \vee b$
Arithmetic	$+$, $-$, $\times$	$x + y$

expressions

Information Hiding in Software with Mixed Boolean-Arithmetic Transforms, Zhou *et al.*, 2007

Protecting operations with MBAs

Mixed		$(x + y) \vee z$
Boolean	$\wedge$, $\vee$, $\oplus$, $\neg$	$a \vee b$
Arithmetic	$+$, $-$, $\times$	$x + y$

expressions

Information Hiding in Software with Mixed Boolean-Arithmetic Transforms, Zhou *et al.*, 2007

Protecting operations with MBAs

Mixed		$(x + y) \vee z$
Boolean	$\wedge$, $\vee$, $\oplus$, $\neg$	$a \vee b$
Arithmetic	$+$, $-$, $\times$	$x + y$

expressions over $Z/2^nZ$

Information Hiding in Software with Mixed Boolean-Arithmetic Transforms, Zhou *et al.*, 2007

Protecting operations with MBAs

Mixed		$(x + y) \vee z$	$(x \oplus y) + 2 \times (x \wedge y)$	$=$	$x + y$
Boolean	$\wedge$, $\vee$, $\oplus$, $\neg$	$a \vee b$	$(x \oplus y) + 2 \times (x \wedge y) - (x + y)$	$=$	0
Arithmetic	$+$, $-$, $\times$	$x + y$	$(x \oplus y) + 2 \times (x \wedge y) - (x + y) + 1$	$=$	1

expressions over $Z/2^nZ$

Information Hiding in Software with Mixed Boolean-Arithmetic Transforms, Zhou *et al.*, 2007

Protecting operations with MBAs

Mixed		$(x + y) \vee z$	$(x \oplus y) + 2 \times (x \wedge y)$	$=$	$x + y$
Boolean	$\wedge$, $\vee$, $\oplus$, $\neg$	$a \vee b$	$(x \oplus y) + 2 \times (x \wedge y) - (x + y)$	$=$	0
Arithmetic	$+$, $-$, $\times$	$x + y$	$(x \oplus y) + 2 \times (x \wedge y) - (x + y) + 1$	$=$	1

expressions over $Z/2^nZ$

Very hard to **prove** and/or **simplify** by *solvers*

$\longrightarrow$ Let's use them to **obfuscate** programs!

Information Hiding in Software with Mixed Boolean-Arithmetic Transforms, Zhou *et al.*, 2007