LLVM Tutorial

John Criswell, Ethan Johnson, and Colin Pronovost
LLVM Tutorial

John Criswell, Ethan Johnson, and Colin Pronovost

#IEEESecDev  https://secdev.ieee.org/2020
Goals
Today’s Goal

❖ Create LLVM pass that instruments store instructions
❖ Optimize checks on store instructions
Programming Background

❖ C++
❖ Other language bindings exist, but C++ is “native”
❖ Know how to use classes, pointers, and references
❖ Know how to use C++ iterators
❖ Know how to use Standard Template Library (STL)
Virtual Machine Image

https://www.cs.rochester.edu/u/ejohns48/secdev19/
Start Your Virtual Machines!

- Login name: user
- Login password: llvm is cool
- Source code is in /home/user/src/SecDev19
LLVM Compiler Structure
LLVM is a Compiler Infrastructure

❖ Set of libraries for building compiler tools
❖ Write tools to analyze software behavior
❖ Write tools that control program behavior
Ahead of Time (AOT) Compiler

Front End

- Clang AST
- Parsed Source Code Tree

Optimizer

- LLVM IR
- Architecture independent code in SSA form

Code Generator

- LLVM Machine IR
- Native Code Instructions
Ahead of Time (AOT) Compiler

Front End

- Clang AST
- Parsed Source Code Tree

Optimizer

- LLVM IR
- Architecture independent code in SSA form

Code Generator

- LLVM Machine IR
- Native Code Instructions
Optimizer Structure

Optimizer

LLVM IR → Pass 1 → LLVM IR → Pass 2 → LLVM IR
LLVM
Intermediate Representation
LLVM IR is a *language* into which programs are translated for analysis and transformation.
LLVM IR Forms

- LLVM Assembly Language
  - Text form saved on disk for humans to read
- LLVM Bitcode
  - Binary form saved on disk for programs to read
- LLVM In-Memory IR
  - Data structures used for analysis and optimization
LLVM IR Example

$ make  Compile SecDev Pass

$ sh compile-tests  Compile test programs with LLVM

$ cd run

$ llvm-dis global.bc  Disassemble and put output in global.ll
define dso_local void @set_foo(i32 %new_foo) #0 {  
entry:

%new_foo.addr = alloca i32, align 4

store i32 %new_foo, i32* new_foo.addr, align 4, !tbaa !2
%0 = load i32, i32* %new_foo.addr, align 4, !tbaa !2
store i32 %0, i32* @foo, align 4, !tbaa !4
ret void
}
Overview of LLVM IR

❖ Each assembly/bitcode file is a Module
❖ Each Module is comprised of
  ❖ Global variables
  ❖ A set of Functions which are comprised of
    ❖ A set of basic blocks which are comprised of
    ❖ A set of instructions
LLVM Bitcode File

Module

Global int[20];

Function: foo()

add
sub
br
add
ret

add
mult
br
add
ret

Function: bar()

add
div
br
add
ret

add
sub
br
add
ret
LLVM Instruction Set

❖ RISC-like architecture
❖ Virtual registers in SSA form
❖ Load instructions read from memory objects
❖ Store instructions write to memory objects
❖ All other instructions read or write virtual registers
Instructions for Computation

- Arithmetic and binary operators
- Two’s complement arithmetic (add, sub, multiply, etc)
- Bit-shifting and bit-masking
- Pointer arithmetic (getelementptr or “GEP”)
- Comparison instructions (icmp, fcmp)
- Generates a boolean result
Memory Access Instructions

- Load instruction reads memory
- Store instruction writes to memory
- Atomic compare and exchange
- Atomic read/modify/write
Control Flow Instructions

- Terminator instructions
  - Indicate which basic block(s) to jump to next
  - Conditional branch, unconditional branch, switch
  - Return instruction to return to caller
  - Call instruction calls a function
  - It can occur in the middle of a basic block
Memory Allocation Instructions

- Stack allocation (alloca)
  - Allocates memory on the stack
- Calls to heap-allocation functions (e.g., malloc())
  - Not a special instruction; just uses a call instruction
- Global variable declarations
  - Not really instructions, but allocate memory
- All globals are *pointers* to memory objects
Single Static Assignment (SSA)

• Each function has infinite set of **virtual registers**
• Only **one** instruction assigns a value to a virtual register (called the **definition** of the register)
• An instruction and the register it defines are **synonymous**

\[ %z = \text{add} \ %x, \ %y \]
Single Static Assignment (SSA)

- Instruction and the register it defines are *synonymous*

%z = add %x, %y

**BinaryOperator**

- **Name**: %z
- **Operator**: add
- **Operands**: %x, %y
Writing an LLVM Pass
Optimizer Structure

Optimizer

LLVM IR → Pass 1 → LLVM IR → Pass 2 → LLVM IR
# Two Types of Passes

<table>
<thead>
<tr>
<th>Pass Type</th>
<th>Description</th>
</tr>
</thead>
</table>
| Analysis Pass                         | ❖ Does not modify program  
❖ Provides information “out of band” to other passes                      |
| Transform (Optimization) Pass         | ❖ Modifies the program  
❖ May call analysis pass methods to get “out of band” information          |
LLVM IR Pass Types

- ModulePass
- FunctionPass
- BasicBlockPass
- I recommend ignoring “funny” passes
  - LoopPass
  - RegionPass
Rules for LLVM Passes

❖ Only modify values and instructions at scope of pass
❖ ModulePass can modify anything
❖ FunctionPass should not modify anything outside of the function
❖ BasicBlockPass should not modify anything outside of the basic block
Let’s Write a Simple Pass
struct SecDev : public ModulePass {
    public:
    // Pass Identifier variable
    static char ID;

    // ModulePass public methods
    SecDev() : ModulePass(ID) {}
    StringRef getPassName() const { return "LLVM Tutorial Pass"; }
    bool runOnModule (Module & M);

    private:
    // Methods for transforming different instructions
    void visitLoadInst (LoadInst & LI);
    void visitStoreInst (StoreInst & SI);
};
runOnModule()

- Entry point for ModulePass
- Receives a reference to the Module as input
- Can locate functions, basic blocks, globals from Module
- Return true if the pass modifies the program
- An analysis pass *always* returns *false*.
- A transform pass can return either *true* or *false*.
SecDev.cpp Example

```cpp
static RegisterPass<SecDev> P ("secdev", "LLVM Tutorial Pass");

bool SecDev::runOnModule (Module & M) {
  return true;
}
```
In-Memory LLVM IR
There is a class for each type of IR object

- Module class
- Function class
- BasicBlock class
- Instruction class

Classes provide iterators for objects within them
LLVM In-Memory IR

- Module
  - Global int[20];
  - Global char[16];
- Function
- Function
- Function
- BasicBlock
  - add sub br
  - add mult br
  - add ret
Class Iterators

❖ Each class provides iterators for items it contains
  ❖ Module::iterator iterates over functions
  ❖ Function::iterator iterates over basic blocks
  ❖ BasicBlock::iterator iterates over instructions
Iterator Example

//
// Iterate over all instructions within a BasicBlock
//
BasicBlock * BB = ...;
BasicBlock::iterator it;
BasicBlock::iterator ie;

for (it = BB->begin(), ie = BB->end(); it != ie; ++it) {
    Instruction * I = &*it;
};
static RegisterPass<SecDev> P ("secdev", "LLVM Tutorial Pass");

bool SecDev::runOnModule (Module & M) {
    //
    // Iterate over all instructions within a Module
    //
    for (Module::iterator fi = M.begin(); fi != M.end(); ++fi) {
        for (Function::iterator bi = fi->begin(); bi != fi->end(); ++bi) {
            for (BasicBlock::iterator it = bi->begin(); it != bi->end(); ++it) {
                Instruction * I = &*it;
            }
        }
    }
    return true;
}
Everything that is an SSA value is a subclass of Value

All Instruction classes are a subclass of Instruction

Similar instructions share a common superclass
Simplified LLVM Class Hierarchy
Common Problem in Passes

Is this a LoadInst, a StoreInst, or something else?
# Casting to Subclass in LLVM

<table>
<thead>
<tr>
<th>Casting Function</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>isa&lt;Class&gt;()</td>
<td>Return true or false if value is of that class.</td>
<td>isa&lt;BranchInst&gt;(I)</td>
</tr>
<tr>
<td>dyn_cast&lt;Class&gt;()</td>
<td>Returns pointer to object of type Class or NULL</td>
<td>dyn_cast&lt;BranchInst&gt;(I)</td>
</tr>
</tbody>
</table>
// Iterate over all instructions within a BasicBlock

BasicBlock::iterator it;
BasicBlock::iterator ie;

for (it = BB->begin(), end = BB->end(); it != end; ++it) {
  Instruction * I = *it;

  if (StoreInst * SI = dyn_cast<StoreInst>(I)) {
    // Do something with store instruction SI
    visitStoreInst(SI);
  }
}
Let’s Add a Run-time Check
void SecDev::visitStoreInst (StoreInst * SI) {
    //
    // Retrieve the pointer operand from the store instruction.
    //
    Value * Pointer = SI->getPointerOperand();

    //
    // Cast the pointer argument of the store instruction to our void pointer
    // type.
    //
    LLVMContext & C = SI->getContext();
    Pointer = new BitCastInst(Pointer, getVoidPtrType(C), Pointer->getName(), SI);

    //
    // Insert a call instruction to the check memory function before the store
    // instruction.
    //
    CallInst::Create(checkMemory, ArrayRef<Value *>(Pointer), "", SI);
}
Try Some Test Programs

$ cd run
$ ./global
$ ./int-sum

Watch calls to checkMemory() print out pointers used in loads and stores!
Can we optimize away unnecessary checks?
Conditions for Optimizing Store Instruction

❖ Pointer points to global variable
❖ Pointer points to result of an alloca
❖ In other words, pointer points to stack-allocated memory
void SecDev::visitStoreInst (StoreInst * SI) {
    //
    // Retrieve the pointer operand from the store instruction.
    //
    Value * Pointer = SI->getPointerOperand();
    
    // If the pointer points to a global variable or to stack-allocated
    // memory, forego adding the run-time check.
    //
    if (…) {
        return;
    } 
    
    // Cast the pointer argument of the store instruction to our void pointer
    // type.
    //
    LLVMContext & C = SI->getContext();
    Pointer = new BitCastInst(Pointer, getVoidPtrType(C), Pointer->getName(), SI);
    
    // Insert a call instruction to the check memory function before the store
    // instruction.
    //
    CallInst::Create(checkMemory, ArrayRef<Value *>(Pointer), "", SI);
}
Let’s Look at Class Hierarchy

Value

Instruction
- LoadInst
- StoreInst
- AllocaInst

GlobalValue
- GlobalVariable
void SecDev::visitStoreInst (StoreInst * SI) {
    //
    // Retrieve the pointer operand from the store instruction.
    //
    Value * Pointer = SI->getPointerOperand();
    //
    // If the pointer points to a global variable or to stack-allocated
    // memory, forego adding the run-time check.
    //
    if (isa<GlobalVariable>(Pointer) || isa<AllocaInst>(Pointer)) {
        return;
    }
    //
    // Cast the pointer argument of the store instruction to our void pointer
    // type.
    //
    LLVMContext & C = SI->getContext();
    Pointer = new BitCastInst(Pointer, getVoidPtrType(C), Pointer->getName(), SI);
    //
    // Insert a call instruction to the check memory function before the store
    // instruction.
    //
    CallInst::Create(checkMemory, ArrayRef<Value *>(Pointer), "", SI);
}
Beyond the Tutorial
Data Flow Analysis

❖ Books
❖ *Compilers: Principles, Techniques, and Tools*
   ❖ Aka “The Dragon Book”
❖ *Principles of Program Analysis*

❖ Papers
❖ Papers on SSA-based algorithms
❖ Kam-Ullman paper on iterative data-flow analysis
Helpful Documents

- LLVM Language Reference Manual
- LLVM Programmer’s Manual
- How to Write an LLVM Pass
- Online LLVM Doxygen pages
We’d love to hear your feedback!

https://forms.gle/xxFzXCkvHBCnSqHo9