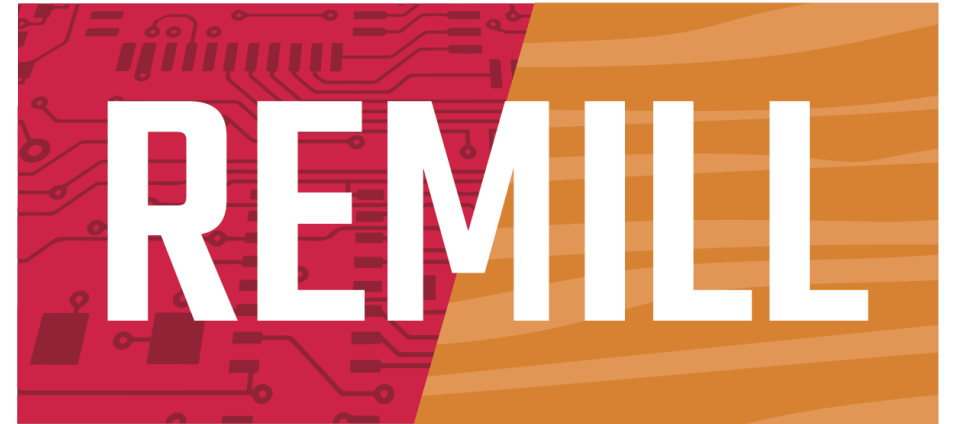
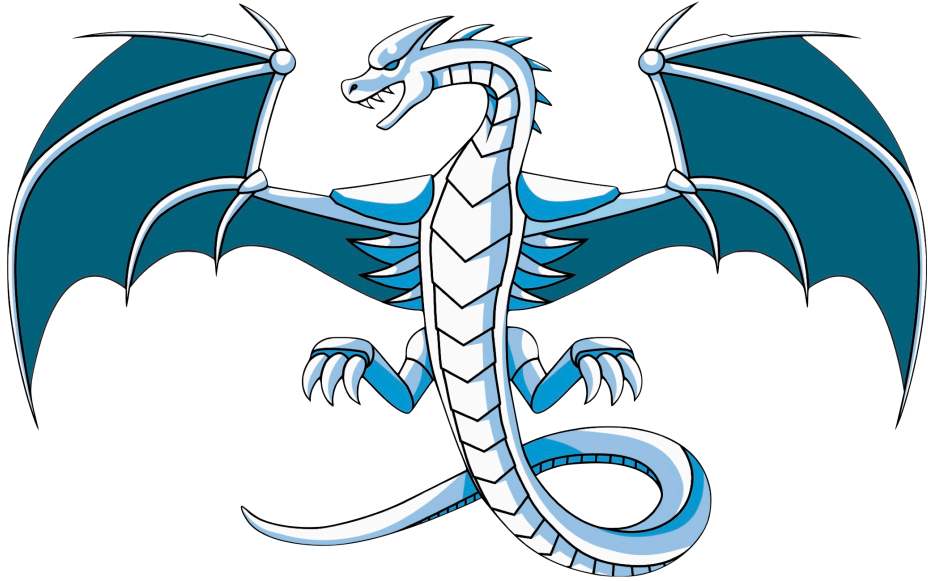



Workshop: LLVM for Reverse Engineers



Duncan Ogilvie

Setup: workshop.ogilvie.pl

1. Login to your GitHub account
2. [Fork the repository](#)
3. Click the green `<>` Code button
4. Press `...` and then `New with options...`
5. Change `Machine type` to `4-core`
6. Then `Create codespace`
7. Wait a ~3 minutes while the image is loading 
 - Press `Show log` to see progress

Introduction

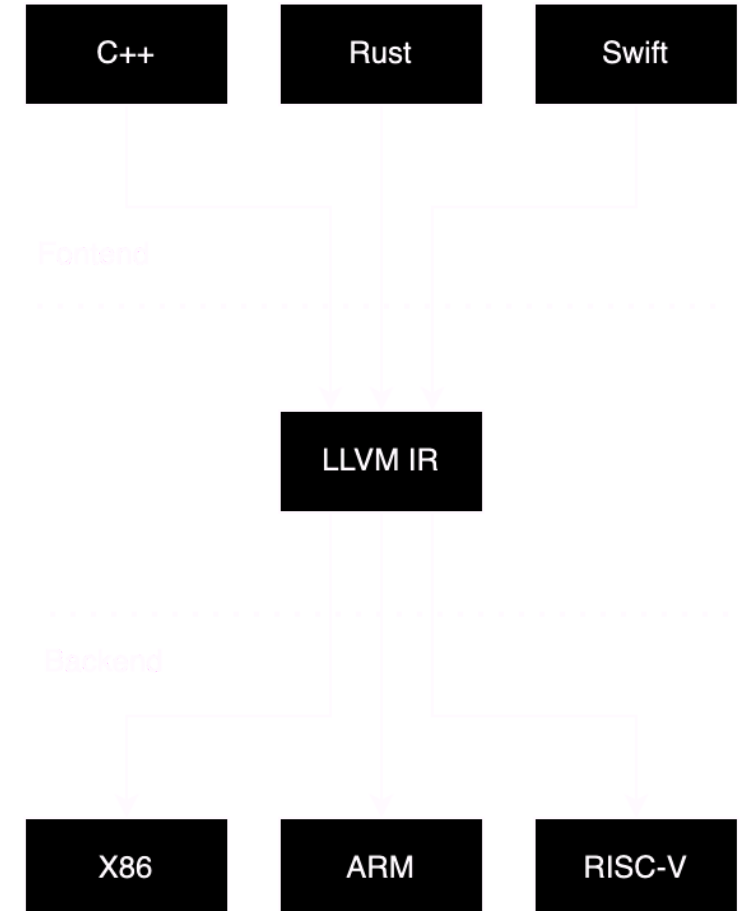
- Meant for absolute (LLVM) beginners
 - **C(++) programming experience absolutely required!**
 - Additionally you need basic reverse engineering knowledge
- Format: hands-on workshop
- Interactive
- Available for on-site training: training@ogilvie.pl

Outline

- LLVM IR (1h30m)
- Coffee break (15m)
- LLVM API (1h30m)
- Remill (time permitting)

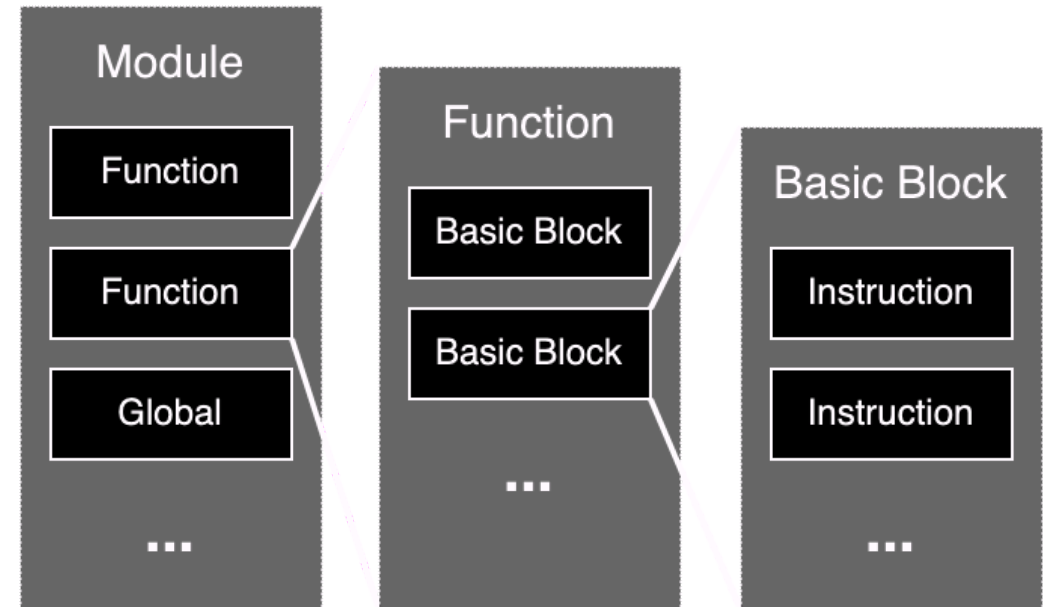
What is LLVM IR?

- **Low Level Virtual Machine** ([paper](#))
 - Authors: Chris Lattner, Vikram Adve (2002)
 - Meant for compiler development
- **Intermediate Representation (IR)**
 - Platform agnostic (mostly)
 - Functions, basic blocks, control flow, etc.
 - **Reduced Instruction Set Computer** ('RISC')
 - **Single Static Assignment** (SSA)
- Reusable optimization/code generation pipeline



LLVM IR: Module

- Functions
 - Basic Blocks
 - Instructions ([reference](#))
- Globals (variables)
- Metadata



LLVM IR: Hello World

C:

```
int hello(int x) {  
    return x + 42;  
}
```

LLVM IR:

```
define i32 @hello(i32 %0) {  
1:  
    %2 = add i32 %0, 42  
    ret i32 %2  
}
```

- Identifiers: `@global`, `%local`
- No signed/unsigned number types
- **Implicit numbering** vs **explicit naming** of *values*

LLVM IR: Clang

hello.c:

```
define dso_local i32 @hello(i32 noundef %0) #0 {  
    %2 = alloca i32, align 4  
    store i32 %0, ptr %2, align 4  
    %3 = load i32, ptr %2, align 4  
    %4 = add i32 %3, 42  
    ret i32 %4  
}  
  
attributes #0 = { noinline nounwind optnone uwtable ... }
```

- `dso_local` : [Runtime Preemption Specifier](#) (always emitted by Clang)
- `noundef` : [Parameter attribute](#) to indicate the value is always defined
- `attributes` : [Group of function attributes](#)

LLVM IR: Single Static Assignment (SSA)

- Local *values* are defined, not assigned
 - *Variable* is a misnomer
 - Also called *registers*
- You cannot define the same value twice
- Does not apply to memory
 - The `load` and `store` instructions operate on `ptr` values

LLVM IR: Control Flow

cfgalloca.ll:

```
uint32_t cfg(uint32_t x) {  
    if (x > 10) return 123;  
    else      return 321;  
}
```

```
define i32 @cfg(i32 %x) {  
    %temp = alloca i32, align 4  
    %cond = icmp ugt i32 %x, 10  
    br i1 %cond, label %bb_if, label %bb_else  
bb_if:      ; preds = %entry, x > 10  
    store i32 123, ptr %temp, align 4  
    br label %bb_end  
bb_else:    ; preds = %entry, !(x > 10)  
    store i32 321, ptr %temp, align 4  
    br label %bb_end  
bb_end:     ; preds = %bb_if, %bb_else  
    %result = load i32, ptr %temp, align 4  
    ret i32 %result  
}
```

LLVM IR: **phi**

cfg_phi.ll:

```
define i32 @cfg(i32 %x) {  
    %cond = icmp ugt i32 %x, 10  
    br i1 %cond, label %bb_if, label %bb_else  
bb_if:      ; preds = %entry, x > 10  
    %result_if = add i32 0, 123  
    br label %bb_end  
bb_else:    ; preds = %entry, !(x > 10)  
    %result_else = add i32 0, 321  
    br label %bb_end  
bb_end:     ; preds = %bb_if, %bb_else  
    %result = phi i32 [%result_if, %bb_if], [%result_else, %bb_else]  
    ret i32 %result  
}
```

Simplifies analysis/optimization passes.

LLVM IR: **select** (ternary)

cfg_select.ll:

```
define i32 @cfg(i32 %x) {  
    %cond = icmp ugt i32 %x, 10  
    %result = select i1 %cond, i32 123, i32 321  
    ret i32 %result  
}
```

```
uint32_t cfg(uint32_t x) {  
    return (x > 10) // cond  
        ? 123  
        : 321  
}
```

LLVM IR: Exercises (part 1)

Instructions: `exercises/1_llvmir/README.md` (Exercise 1a-1d)

LLVM IR: `getelementptr`

- Pointer arithmetic
 - Arrays
 - Structs
- **Does not** read memory
- **Opaque Pointers**
 - Default since LLVM 15
 - Previously pointers had a type

LLVM IR: `getelementptr` (array)

C:

```
uint32_t arrayExample(uint32_t* arr) {  
    return arr[5];  
}
```

LLVM IR:

```
define i32 @arrayExample(ptr %arr) #0 {  
    %ptr_idx_5 = getelementptr i32, ptr %arr, i64 5  
    %result = load i32, ptr %ptr_idx_5  
    ret i32 %result  
}
```

- `ptr_idx_5 = (uintptr_t)arr + 5 * sizeof(i32)`

LLVM IR: `getelementptr` (member)

C:

```
typedef struct { uint64_t a[2]; uint32_t b; uint32_t c[5]; } MyStruct;

uint32_t structExample1(MyStruct* s) {
    return s->b; // s[0].b
}
```

LLVM IR:

```
%struct.MyStruct = type { [2 x i64], i32, [5 x i32] }

define i32 @structExample1(ptr %s) #0 {
    %ptr_b = getelementptr %struct.MyStruct, ptr %s, i32 0, i32 1
    %result = load i32, ptr %ptr_b
    ret i32 %result
}
```

- `ptr_b = (uintptr_t)s + 0 * sizeof(MyStruct) + offsetof(MyStruct, b)`

LLVM IR: `getelementptr` (member array)

C:

```
typedef struct { uint64_t a[2]; uint32_t b; uint32_t c[5]; } MyStruct;

uint32_t structExample2(MyStruct* s) {
    return s->c[3];
}
```

LLVM IR:

```
%struct.MyStruct = type { [2 x i64], i32, [5 x i32] }

define i32 @structExample2(ptr %s) #0 {
    %ptr_c = getelementptr %struct.MyStruct, ptr %s, i32 0, i32 2
    %ptr_c_3 = getelementptr [5 x i32], ptr %ptr_c, i32 0, i32 3
    %result = load i32, ptr %ptr_c_3
    ret i32 %result
}
```

LLVM IR: `getelementptr` (optimization)

```
%struct.MyStruct = type { [2 x i64], i32, [5 x i32] }

define i32 @structExample2(ptr %s) #0 {
    %ptr_c = getelementptr %struct.MyStruct, ptr %s, i32 0, i32 2
    %ptr_c_3 = getelementptr [5 x i32], ptr %ptr_c, i32 0, i32 3
    %result = load i32, ptr %ptr_c_3
    ret i32 %result
}

define i32 @structExample2_opt(ptr %s) #0 {
    %ptr_c_3 = getelementptr %struct.MyStruct, ptr %s, i32 0, i32 2, i32 3
    %result = load i32, ptr %ptr_c_3
    ret i32 %result
}
```

LLVM IR: `getelementptr` (flattening)

```
%struct.MyStruct = type { [2 x i64], i32, [5 x i32] }

define i64 @structExample3(ptr %s) #0 {
    %ptr_a = getelementptr %struct.MyStruct, ptr %s, i32 0, i32 0
    %ptr_a_1 = getelementptr [2 x i64], ptr %ptr_a, i32 0, i32 1
    %result = load i64, ptr %ptr_a_1
    ret i64 %result
}

define i64 @structExample3_opt(ptr %s) #0 {
    ; No reference to MyStruct at all anymore
    %ptr_a_1 = getelementptr [2 x i64], ptr %s, i64 0, i64 1
    %result = load i64, ptr %ptr_a_1
    ret i64 %result
}
```

- [LLVM IR Godbolt](#)
- [C Godbolt](#) (play with the optimization settings)




LLVM IR: Exercises (part 2)

Instructions: `exercises/1_llvmir/README.md` (Exercise 2a)

Quick break (15 min)



LLVM API

-  Difficult to navigate
-  Annoying to set up
-  Use Google/ChatGPT liberally

LLVM IR: Verification

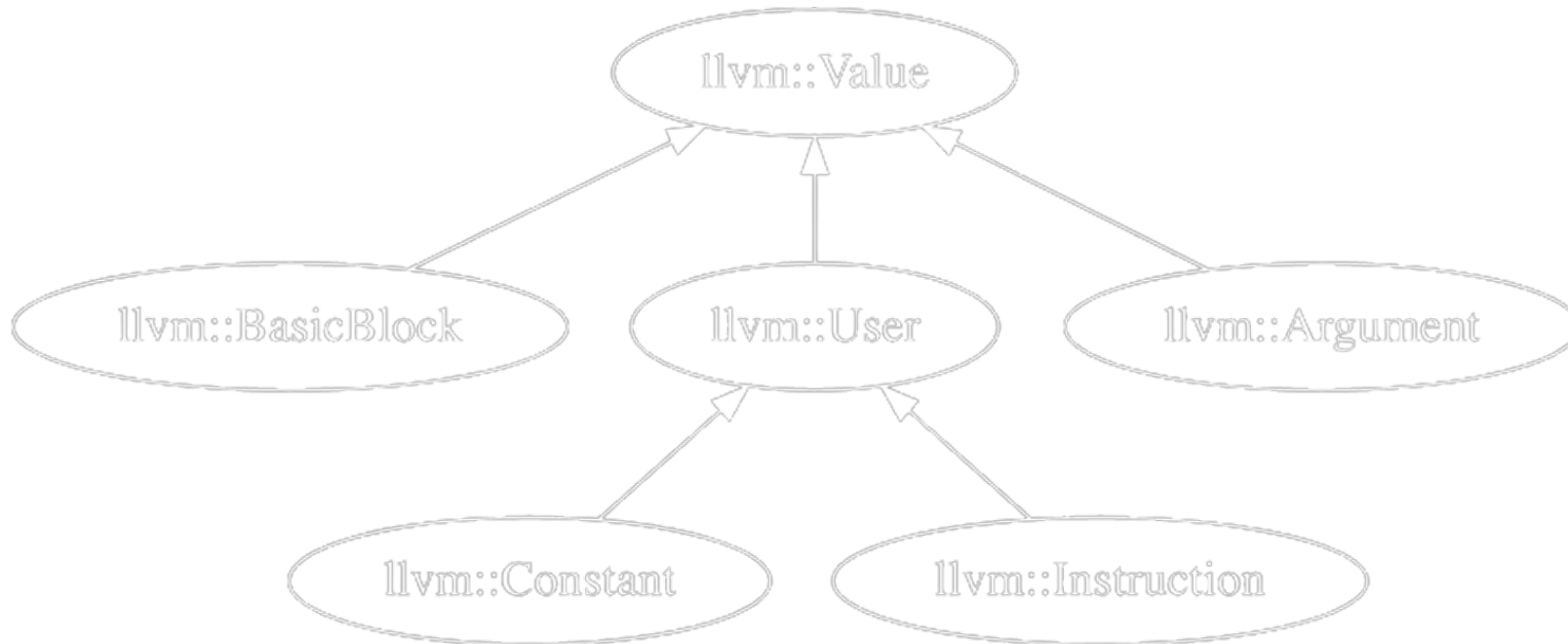
- Well-formedness
 - Type checking
 - Control Flow Graph (CFG) integrity
 - Terminator instructions
 - Entry block cannot have predecessors
 - `phi` / `alloca` at the start
 - Values defined before they are used
- `llvm::verifyModule` + `LLVM_ENABLE_ASSERTIONS`

LLVM API: Basics

Walkthrough: `src/api-basics.cpp`

LLVM API: Memory Model

- `llvm::Value*` (same pointer -> same value)
- `llvm::isa<T>` / `llvm::dyn_cast<T>`



LLVM API: Tool template

Show: `src/bc-tool.cpp`

LLVM API: Exercises

Instructions: `exercises/2_api/README.md`

What is Remill?

- Authors: Trail of Bits (2015)
- *Lifts* native instructions to LLVM IR
 - Applications: binary analysis/instrumentation/emulation
 - Architectures: ARM, X86, PPC, SPARC, *Sleigh*
- Mild abuse of the IR, requires some tricks

Remill: Concepts

- Instruction semantics in C++
 - Easier to maintain
 - Compiled to LLVM IR
- `State*` structure -> CPU Registers
- `Memory*` pointer -> memory manager
 - Total ordering to preserve semantics
- 'Messaging' required

Remill: Instruction Semantics

Semantics of the x86 `mov` instruction:

```
template <typename D, typename S>
DEF_SEM(MOV, D dst, const S src) {
    WriteZExt(dst, Read(src));
    return memory;
}

DEF_ISEL(MOV_GPRv_MEMv_32) = MOV<R32W, M32>;
```



Remill: Lifting Basic Blocks

Basic Block Definition:

```
Memory *__remill_basic_block(State &state, Ptr block_addr, Memory *memory);
```

- Calls to the semantics are inserted here.
- State is fully symbolic
- Requires additional work to restore the calling convention

Remill: High level example

```
Memory *__remill_basic_block(State &state, Ptr block_addr, Memory *memory) {  
    // mov rax, rdi  
    state.rax = state.rdi;  
    state.rip += 3;  
    // ret  
    state.rip = *(Ptr*)state.rsp;  
    state.rsp += sizeof(Ptr);  
    return __remill_function_return(state, state.rip, memory);  
}
```


Remill: Helpers

```
Memory *__remill_write_memory_8(Memory *m, addr_t a, uint8_t v);  
Memory *__remill_write_memory_16(Memory *m, addr_t a, uint16_t v);  
Memory *__remill_write_memory_32(Memory *m, addr_t a, uint32_t v);  
Memory *__remill_write_memory_64(Memory *m, addr_t a, uint64_t v);
```

- Abstraction to represent interaction with the host CPU (memory, calls, indirect branches, syscalls, flag computations)
- Implementation varies depending on the purpose (emulation, symbolic execution, decompilation)
- Makes the lifted IR difficult to work with for humans (extremely verbose)

Remill: Exercises

Helpers: `helpers/x86_64/RemillHelpers.cpp`

Instructions: `exercises/3_lifting/README.md`

Note: Read the [SATURN](#) paper if you want to see where this can take you...

Closing Remarks

- Continue at home!
- Thanks: [Matteo Favaro](#)
- Get in touch: training@ogilvie.pl