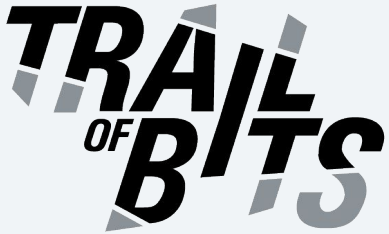
TRAIL OF BITS

UMD-CSEC

A mostly gentle introduction to LLVM

Hi

- **Me**
  - Previously: UMD-CSEC 2014-2018
  - Currently: Senior security engineer @ Trail of Bits
    - Program analysis research (LLVM, x86)
    - Open source engineering (package managers, Python, Rust)
- **Trail of Bits**
  - NYC-based security and research consultancy
  - ~100 people spread around the world, >50% remote
  - Research (DARPA), engineering (commercial, OSS), assurance (commercial)
  - Summer and winter internships



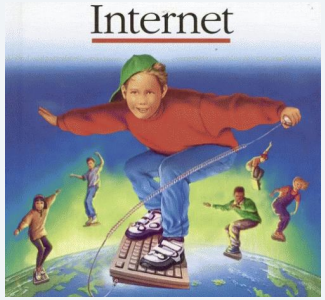
This talk

Goals: LLVM will no longer be just a C/C++ compiler to you

***Whirlwind* tour (that means you're *encouraged* to stop and ask questions):**

- **The Clang frontend (CFE) and what you can do with it**
- **LLVM IR and what's in it**
  - Writing static analyses using the pass APIs
- **How the sausage is made**
  - The LLVM middle end, target {in,}dependent instruction selection and lowering
- **Bonus: Using LLVM for things it wasn't designed for**

Why should you care?



You're in a cybersecurity club, compilers are just tools, right?

- **You want to find bugs in programs**
  - Virtually all static and dynamic analysis requires compiler fundamentals
    - Even fuzzing: AFL etc. use the compiler to instrument programs for coverage tracking
  - IRs and normalized forms, control and dataflow analyses are essential building blocks
- **You want to make software more secure**
  - The future of *unsafe* programming languages (C/C++) is safer compilers and more compiler-introduced/enforced mitigations
  - The future of programming languages *in general* is smarter compilers that can prove more properties about programs (both for optimization and safety)
- **You want to be 31337 and stunt on people with your CS skillz**

But what *is* LLVM?

Originally: ~~Low Level Virtual Machine~~: not low level, not really virtual

Actually: A *massive* compiler infrastructure project, encompassing:

- `clang`: **A GCC-compatible C/C++/Objective-C frontend (“CFE”)**
- **A flexible target independent* intermediate representation (LLVM IR)**
- `opt`: **an optimizer for LLVM IR**
- **Target dependent code generators, a modern linker, assembler, …**
- **…and so much more:**
  - A debugger (`lldb`)
  - Modern C++ runtime (`libc++`)
  - Symex engine (KLEE)

From the top: the frontend

You already know this part: it's the program that wraps all of LLVM's subcomponents into a single tool that produces binaries.

```
$ clang -g -O3 -o foo foo.c
```

"Compile `foo.c` into `foo`, embedding debug info (`-g`) and optimizing aggressively (`-O3`)"

CFE has two primary tasks:

- **Provide a familiar CLI to engineers (`clang` and `clang++`)**
  - Engineers and build systems know GCC's flags, so CFE attempts to be compatible with them
  - Translate flags and options into various compilation decisions
- **Lex C/C++ inputs, parse into abstract syntax trees, and then "lower" those ASTs into LLVM IR for subsequent optimization**

We can tell CFE to dump the generated AST:

```
$ clang -Xclang -ast-dump -fsyntax-only -o test.ast test.c
```

```
void foo(int, int, int);
void bar(const char *);

int main(void) {
  foo(1, 2, 3);
  bar("hello");

  return 1;
}
```

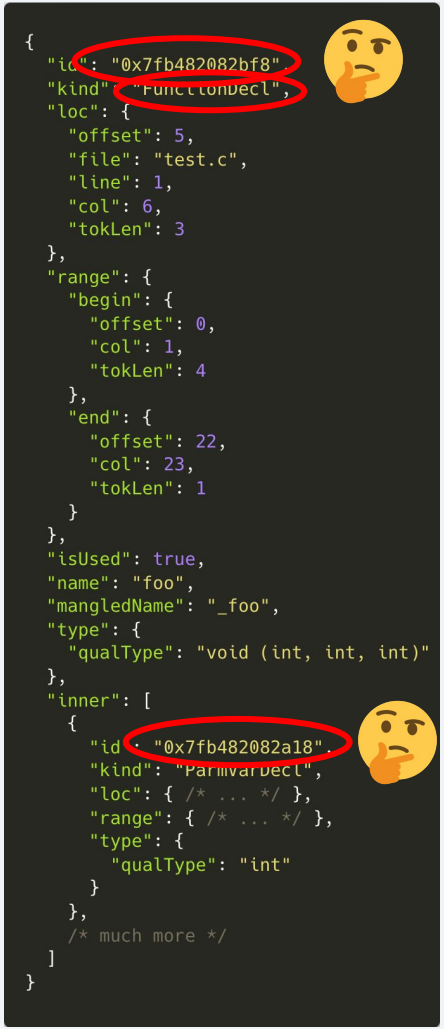
```
TranslationUnitDecl 0x7f7c3183ae08 <<invalid sloc>> <invalid sloc>
|-TypedefDecl 0x7f7c3183b6a0 <<invalid sloc>> <invalid sloc> implicit __int128_t '__int128'
| `-BuiltinType 0x7f7c3183b3a0 '__int128'
|-TypedefDecl 0x7f7c3183b710 <<invalid sloc>> <invalid sloc> implicit __uint128_t 'unsigned __int128'
| `-BuiltinType 0x7f7c3183b3c0 'unsigned __int128'
|-TypedefDecl 0x7f7c3183ba18 <<invalid sloc>> <invalid sloc> implicit __NSConstantString 'struct __NSConstantString_tag'
| `-RecordType 0x7f7c3183b7f0 'struct __NSConstantString_tag'
|   `-Record 0x7f7c3183b768 '__NSConstantString_tag'
|-TypedefDecl 0x7f7c3183bab0 <<invalid sloc>> <invalid sloc> implicit __builtin_ms_va_list 'char *'
| `-PointerType 0x7f7c3183ba70 'char *'
|   `-BuiltinType 0x7f7c3183aea0 'char'
|-TypedefDecl 0x7f7c3183bda8 <<invalid sloc>> <invalid sloc> implicit __builtin_va_list 'struct __va_list_tag [1]'
| `-ConstantArrayType 0x7f7c3183bd50 'struct __va_list_tag [1]' 1
|   `-RecordType 0x7f7c3183bb90 'struct __va_list_tag'
|     `-Record 0x7f7c3183bb08 '__va_list_tag'
|-FunctionDecl 0x7f7c31882bf8 <test.c:1:1, col:23> col:6 used foo 'void (int, int, int)'
| |-ParmVarDecl 0x7f7c31882a18 <col:10> col:13 'int'
| |-ParmVarDecl 0x7f7c31882a98 <col:15> col:18 'int'
| `-ParmVarDecl 0x7f7c31882b18 <col:20> col:23 'int'
|-FunctionDecl 0x7f7c31882dd8 <line:2:1, col:22> col:6 used bar 'void (const char *)'
| `-ParmVarDecl 0x7f7c31882d10 <col:10, col:21> col:22 'const char *'
`-FunctionDecl 0x7f7c31882f50 <line:4:1, line:9:1> line:4:5 main 'int (void)'
  `-CompoundStmt 0x7f7c31883268 <col:16, line:9:1>
    |-CallExpr 0x7f7c318830e0 <line:5:3, col:14> 'void'
    | |-ImplicitCastExpr 0x7f7c318830c8 <col:3> 'void (*)(int, int, int)' <FunctionToPointerDecay>
    | | `-DeclRefExpr 0x7f7c31883018 <col:3> 'void (int, int, int)' Function 0x7f7c31882bf8 'foo' 'void (int, int, int)'
    | |-IntegerLiteral 0x7f7c31883038 <col:7> 'int' 1
    | |-IntegerLiteral 0x7f7c31883058 <col:10> 'int' 2
    | `-IntegerLiteral 0x7f7c31883078 <col:13> 'int' 3
    |-CallExpr 0x7f7c318831e0 <line:6:3, col:14> 'void'
    | |-ImplicitCastExpr 0x7f7c318831c8 <col:3> 'void (*)(const char *)' <FunctionToPointerDecay>
    | | `-DeclRefExpr 0x7f7c31883118 <col:3> 'void (const char *)' Function 0x7f7c31882dd8 'bar' 'void (const char *)'
    | `-ImplicitCastExpr 0x7f7c31883220 <col:7> 'const char *' <NoOp>
    |   `-ImplicitCastExpr 0x7f7c31883208 <col:7> 'char *' <ArrayToPointerDecay>
    |     `-StringLiteral 0x7f7c31883178 <col:7> 'char [6]' lvalue "hello"
    `-ReturnStmt 0x7f7c31883258 <line:8:3, col:10>
      `-IntegerLiteral 0x7f7c31883238 <col:10> 'int' 1
```

The frontend

…or even dump and query as JSON!

```
$ clang -Xclang -ast-dump=json \
    -Xclang -ast-dump-filter \
    -Xclang foo \
    -fsyntax-only test.c -o-
```

“Dump the AST as JSON, reducing to only nodes that match `foo`**”**




```
$ clang -Xclang -ast-dump=json \
    -Xclang -ast-dump-filter \
    -Xclang foo -fsyntax-only test.c -o- \
  | sed 1d \
  | jq '.type'
```

```
{
  "qualType": "void (int, int, int)"
}
```

Frontend: wrapup

The Clang CLI is the most *limited* way to interact with the frontend's internals; we can use the CFE's C++ APIs to do much more powerful things.

Built-in examples:

- `clang-format`: **reformat code according to a standard style**
- `clang-tidy`: **lint C/C++ for common errors, like implicit conversions**
  - Automatically apply fixes to codebases!

External examples:

- `constexpr-everything`: **automatically rewrite C++ code to qualify as many things as `constexpr` as possible**

Key APIs: `RecursiveASTVisitor`, `ASTFrontendAction`, `FixItRewriter`

LLVM IR and the optimizer

The frontend's ultimate job is to translate the program's AST into LLVM's *Intermediate Representation* so that the *optimizer* can refine it.

LLVM IR looks a bit like funky C:

```
$ clang -c test.c -S -emit-llvm -o-
```

```
@.str = private unnamed_addr constant [6 x i8] c"hello\00", align 1

define dso_local i32 @main() #0 !dbg !8 {
  %1 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  call void @foo(i32 noundef 1, i32 noundef 2, i32 noundef 3), !dbg !14
  call void @bar(i8* noundef getelementptr inbounds ([6 x i8], [6 x i8]* @.str, i64 0, i64 0)), !dbg !15
  ret i32 1, !dbg !16
}

declare dso_local void @foo(i32 noundef, i32 noundef, i32 noundef) #1

declare dso_local void @bar(i8* noundef) #1
```

Stack allocations!

Function calls/returns!

Literals and constants!

Function declarations and definitions!

LLVM IR: key properties and semantics

Generally speaking, LLVM IR follows *C abstract machine* semantics: pointers, functions, etc. all behave the way they do in C.

Oversimplification: LLVM IR is a load-store architecture: there are two storage areas (registers and memory), with separate instructions to access each. Memory can be heap, stack, global, etc., and is accessed via `load` and `store`.

Key property: LLVM IR registers ("variables") are in *single static assignment (SSA) form*: every variable is written to exactly once, and there are an infinite number of variables. This property forms the backbone of many of LLVM's optimizations.

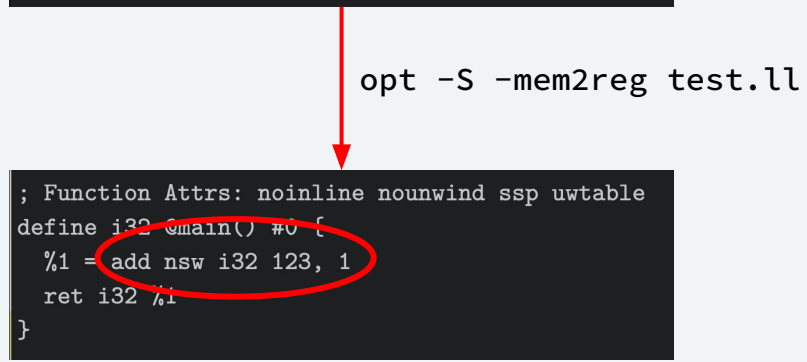
```
int main(void) {
  int x = 123;
  x += 1;
  return x;
}
```

```
define dso_local i32 @main() #0 !dbg !8 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  call void @llvm.dbg.declare(metadata i32* %2, metadata !14, metadata !DIExpress
  store i32 123, i32* %2, align 4, !dbg !15
  %3 = load i32, i32* %2, align 4, !dbg !16
  %4 = add nsw i32 %3, 1, !dbg !16
  store i32 %4, i32* %2, align 4, !dbg !16
  %5 = load i32, i32* %2, align 4, !dbg !17
  ret i32 %5, !dbg !18
}
```

LLVM IR: SSA construction

- The naive SSA form is *very* inefficient: lots of memory locations means lots of loads/stores that slow down execution.
- One of LLVM's earliest optimization passes is `mem2reg`, which "lifts" any `alloca` that has only `loads` and `stores` into one or more SSA variables.

```
; Function Attrs: noinline nounwind ssp uwtable
define i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  store i32 123, i32* %2, align 4
  %3 = load i32, i32* %2, align 4
  %4 = add nsw i32 %3, 1
  store i32 %4, i32* %2, align 4
  %5 = load i32, i32* %2, align 4
  ret i32 %5
}
```



Constant folding opportunity!

LLVM IR: Optimizing and optimizer passes

- At `-O1` and higher, the clang frontend will run `mem2reg` and a variety of other default optimizations: constant folding, dead code elimination, control flow graph simplification, etc.
- Each of these optimizations is written as LLVM *passes*, which visit the IR in different ways (entire program, per-function, per-loop, callgraph, etc.)
- The pass API is a public C++ API, and we can write our own!

```
$ cargo install llvm-passgen

# create a function pass named Test
$ llvm-passgen Test \
  --kind function

$ cd Test/build

$ cmake ..

$ cmake --build .

# run the Test pass on test.ll
$ opt -S \
  -load LLVMTest.so --Test \
  ~/test.ll
```

```cpp
#include "llvm/IR/Function.h"
#include "llvm/Pass.h"
#include "llvm/Support/raw_ostream.h"

using namespace llvm;

namespace {
struct Test : public FunctionPass {
  static char ID;

  Test() : FunctionPass(ID) {}

  bool runOnFunction(Function &F) override {
    errs() << "function pass: " << F.getName() << '\n';
    return false;
  }
};
} // namespace

char Test::ID = 0;
static RegisterPass<Test> X("Test", "Test pass", true, false);
```

- `runOnFunction` executes once for each `llvm::Function` in the `llvm::Module`, and returns `true` if it modifies the function.
- Lots of interesting program state: each function can iterate over its basic blocks (≈ control flow) and constituent instructions
- Passes can register dependencies on other passes, causing LLVM to run those passes first and collect their results
- Passes can invalidate the results of earlier passes (e.g. changing the call graph), requiring them to be re-run if needed later

From IR to machine code

Real computer architectures are not like LLVM IR:

- **Infinite, typed SSA registers → finite, untyped machine registers**
- **Modules, functions, basic blocks → translation units, stack frames**
- **Parallel operations (no data deps) → limited execution slots and units**
- **Not all LLVM instructions correspond to machine instructions**
  - One-to-many, many-to-one depending on ISA support for multiplication, vectorized ops, etc.

LLVM goes through several phases to get to machine code; at a high level:

- **Instruction selection (ISel): LLVM IR → SelectionDAG**
- **Scheduling: SelectionDAG → MachineInstrs**
- **Register allocation: select concrete registers for MachineInstrs**
- **Prologue/Epilogue Insertion (PEI): MachineFunctions have concrete stack frames and setup/teardown code**
- **Emission: Conversion into actual target assembly/raw machine code**

Machine code generation: fiddly bits

- **How do you go from infinite registers to finite registers?**
  - Register allocation!
- **Naive: fill each GPR with an SSA variable until you run out, “spill” the rest onto the stack**
  - Problems with this? Can we do better?
  - Can the compiler/IR help us?
- **Different performance tradeoffs**
  - Normally we don’t care (much) about compilation time, but sometimes we do (JITs)
- **In the abstract: register allocation is reducible to k-coloring**
  - NP-hard!
  - Real world adds additional annoyances:
    - Some ISAs “pre-color” the graph (e.g. specifying registers for params, return)
    - Some ISAs have aliased registers (AMD64: al/ax/eax/rax)

Machine code generation: other fiddly bits

- **Middle and late optimization**
  - Most optimization is done on LLVM IR, but some things are better suited for lower representations: instruction fusion, peephole optimization, modulo scheduling
- **Prologue/epilogue insertion**
  - Need to compute stack frame size, inject exception handling code, lots of other small things
- **Many other things**
  - Call lowering (GOT/PLT, direct & indirect calls)
  - Hardening and mitigations (canaries, ASLR)
  - Debug information (DWARF, PDB)
  - Object file formats (PE, Mach-O, COFF, ELF)
  - Linking….
  - Recently: binary layout optimization (PGO on steroids): BOLT
- **There's a lot going on!**

What else can we do?

- **LLVM normally turns source code into machine code**
  - But LLVM IR is extremely useful…
  - What if we go in the other direction?
- **Binary lifting/translation: faithfully decompile (“lift”) machine instructions/functions/entire programs to LLVM IR**
  - McSema / Remill / Anvill
  - Iteratively “brighten” the IR so that it resembles what a frontend like Clang would produce
  - Why would we do this?
  - Challenges: CFG recovery, state decomposition (revert x86 context to SSA)
    - The opposite of register allocation!
- **We don’t always want to compile programs**
  - LLVM IR has a lot of detail in it; it’s a pretty good starting point for static analyses

Wrapup

- **LLVM is an extremely large project; we've only scratched the surface**
- **Not all is perfect in heaven:**
  - LLVM started as a C/C++ compiler; lots of leaky abstractions in the IR
    - We can talk about all the ways this causes problems for Rust and others
  - Designed in tandem with the CFE, so LLVM recognizes IR patterns that CFE uses
    - Deviate from those patterns, and the optimizer suffers
  - LLVM IR is *complex*, both structurally and semantically
    - For good reasons! But it's not the most straightforward example of an SSA IR

Resources

- **LLVM's documentation**
  - Writing an LLVM Pass
  - The LLVM Language Reference
  - LLVM Developers' Meeting Archives (lots of good talks)
- **Trail of Bits blog: https://blog.trailofbits.com/**
- **Trail of Bits GitHub: https://github.com/trailofbits**
  - Rellic, McSema, Remill, Anvill
- **Personal blog: https://blog.yossarian.net**
  - LLVM-specific posts: https://blog.yossarian.net/tags#llvm
- **Personal GitHub: https://github.com/woodruffw**
  - llvm-passgen, mollusc