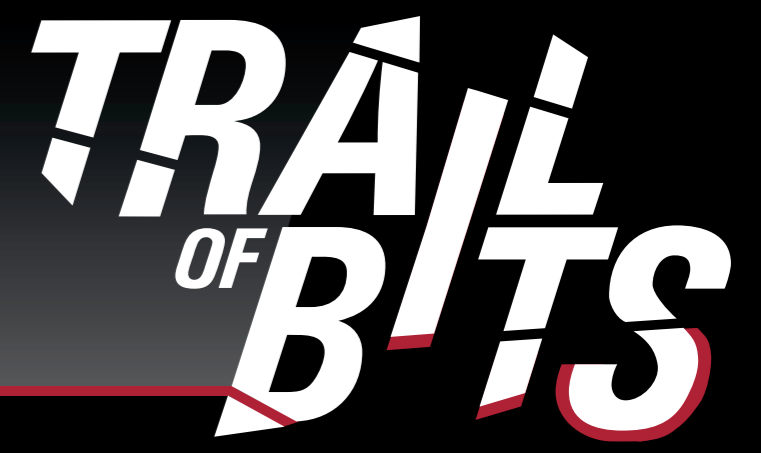


Lifting machine code instructions to LLVM bitcode

How to use the Remill C++ library

Peter Goodman
peter@trailofbits.com



Remill produces bitcode that emulates the operations of machine code instructions

Mindset: Lifted bitcode is like a machine code emulator that has been specialized to one or more instructions, and the LLVM bitcode instructions are the CPU μ -ops

Challenge: Some low-level operations don't map to higher-level constructs, e.g. memory access and indirect control-flows are modelled using "intrinsic" functions

Result: Lifted bitcode is not "executable" out-of-the-box. Users must define or replace intrinsic functions in order to implement memory access, control-flow, and architecture-specific "hyper calls"

How to lift machine code instructions with the Remill C++ library

1

Load instruction semantics bitcode module

```
llvm::LLVMContext C;  
auto A = remill::GetTargetArch();  
auto M = remill::LoadArchSemantics(A, &C);
```

2

Decode instruction bytes

```
auto bytes = "\xc7\x04\xba\x01...";  
remill::Instruction I;  
A->DecodeInstruction(0, bytes, I);
```

3

Create a "basic block" function to hold lifted instructions

```
auto F = remill::DeclareLiftedFunction(M, "F");  
remill::CloneBlockFunctionInto(F);
```

4

Lift instruction into block function

```
remill::IntrinsicTable it(M);  
remill::InstructionLifter lifter(A, it);  
lifter.LiftIntoBlock(I, &(F->getEntryBlock()));  
remill::AddTerminatingTailCall(F, it.jump);
```

5

Optimize and dump the bitcode

```
remill::OptimizeModule(M, {F});  
F->dump();
```

6

Result

```
$ remill-lift-6.0 --arch amd64 --bytes c704ba01000000 --ir_out /dev/stdout
```

```
define %struct.Memory* @F(%struct.State* noalias, i64, %struct.Memory* noalias) {  
  %3 = getelementptr inbounds %struct.State, %struct.State* %0, i64 0, i32 6, i32 33, i32 0, i32 0  
  %4 = getelementptr inbounds %struct.State, %struct.State* %0, i64 0, i32 6, i32 7, i32 0, i32 0  
  %5 = getelementptr inbounds %struct.State, %struct.State* %0, i64 0, i32 6, i32 11, i32 0, i32 0  
  %6 = load i64, i64* %4, align 8  
  %7 = load i64, i64* %5, align 8  
  %8 = shl i64 %7, 2  
  %9 = add i64 %8, %6  
  %10 = add i64 %1, 7  
  store i64 %10, i64* %3, align 8  
  %11 = tail call %struct.Memory* @__remill_write_memory_32(%struct.Memory* %2, i64 %9, i32 1) #3  
  %12 = tail call %struct.Memory* @__remill_jump(%struct.State* nonnull %0, i64 %10, %struct.Memory* %11)  
  ret %struct.Memory* %12  
}
```

What is this??



- Machine code instructions operate on registers and memory
- `__remill_basic_block` function in target machine semantics module `M` defines register variables for use by lifted instructions
- `CloneBlockFunctionInto` copies vars from `__remill_basic_block` into `F`

- Instruction `I` names semantics function in `M`, register vars in `F`
- `lifter` reads `I` and looks up semantics function in `M`, and register vars in `F`, then passes them as arguments to semantics function

Bytes representing instruction...

```
mov dword ptr [RDX + RDI * 4], 0x1
```



Where Remill is used

McSema:

- Lifts off-the-shelf program binaries into LLVM bitcode, using Remill to lift program instructions
- Lifted programs can be analyzed symbolically using KLEE, or compiled into new executables
- Cross platform (Linux, macOS, Windows), works on x86, amd64, and AArch64 programs (ELF, PE)
- Extends Remill's InstructionLifter API, injects "cross-references" between instructions/data
- Open-source, find it on GitHub at github.com/trailofbits/mcsema



REVEN-Axion:

- Commercial reverse-engineering product: full-system emulator, symbolic execution, reverse execution
- Uses Remill in experimental new feature

