Lifting machine code instructions to LLVM bitcode

How to use the Remill C++ library

Peter Goodman
peter@trailofbits.com

TRAIL OF BITS

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);
```

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

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
}
```

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

MCSEMA

REVEN-Axion:

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

TETRANE

Modelling machine code semantics in C++

The life of an instruction in Remill

Peter Goodman
peter@trailofbits.com

We want to analyze machine code

- Is this program vulnerable to memory corruption, return-oriented programming attacks, or other exploits?
- Are these two functions equivalent?

Machine code is hard to analyze

- Thousands of instructions, many with complex side-effects
- Legacy (e.g. x87) and modern (e.g. AVX) features
- Memory is flat and opaque, no high-level types

Remill translates x86/amd64 and AArch64 (ARMv8) instructions into LLVM bitcode

Motivation: LLVM bitcode is easier to analyze, and many analyses for LLVM bitcode already exist
Challenge: Need LLVM bitcode semantics for all machine code instructions
Solution: Implement instruction semantics with C++ functions, compile them to LLVM bitcode with Clang

Program source code is compiled into...

```
int *RDX = ...;
for (long RDI = 0; ...; ++RDI) {
  RDX[RDI] = 1;
}
```

Assembly, a textual representation of...

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

Machine code, which we want to analyze

```
c7 04 ba 01 00 00 00
```

github.com/trailofbits/remill

REMILL

1 **Instructions implemented as C++ functions...**

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

2 **Operating on registers in a C++ structure...**

```
struct State : public ArchState {
  ArithFlags     aflag;
  GPR            gpr;
  ...            ...
};
```

3 **And specialized by different instruction operand types**

```
DEF_ISEL_MnW_In(MOV_MEMv_IMMz, MOV);
// extern "C" constexpr auto MOV_MEMv_IMMz_32 = MOV<M32W, I32>;
```

4 **Remill uses instruction decoder information to select a C++ semantics function...**

```
(AMD64 100000fb1 7 (BYTES c7 04 ba 01 00 00 00) MOV_MEMv_IMMz_32
  (WRITE_OP (DWORD_PTR (ADD (REG_64 RDX) (MUL (REG_64 RDI) (IMM_64 0x4)))))
  (READ_OP (SIGNED_IMM_32 0x1))
```

5 **And calls the semantics within a "basic block" function with pre-defined "register" variables**

```
Memory *__remill_basic_block(State &state, addr_t pc, Memory *memory) {
  auto &RDX = state.gpr.rdx.qword;  // Pre-defined
  auto &RDI = state.gpr.rdi.qword;  // Pre-defined
  memory = MOV_MEMv_IMMz_32(memory, state, RDX + RDI * 0x4, 0x1);
  return memory;
}
```

6 **Remill aggressively optimizes the result into LLVM bitcode equivalent to the following**

```
Memory *__remill_basic_block(State &state, addr_t pc, Memory *memory) {
  return __remill_write_memory_32(
      memory, state.gpr.rdx.qword + state.gpr.rdi.qword * 4, 1);
}
```