Modelling machine code semantics in C++

The life of an instruction in Remill

Peter Goodman
peter@trailofbits.com

TRAIL OF BITS

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);
}
```