From execution traces to binary reconstruction: A tale of CFGs and LLVM IR

Gabriel Duque

LSE - EPITA

gabriel.duque@lse.epita.fr

April 9, 2019

Overview

1. Susanoo
   - Trace capture
   - CFG recovery
   - Function detection
2. MCSema
   - General Flow
   - Interface
   - The Remill library

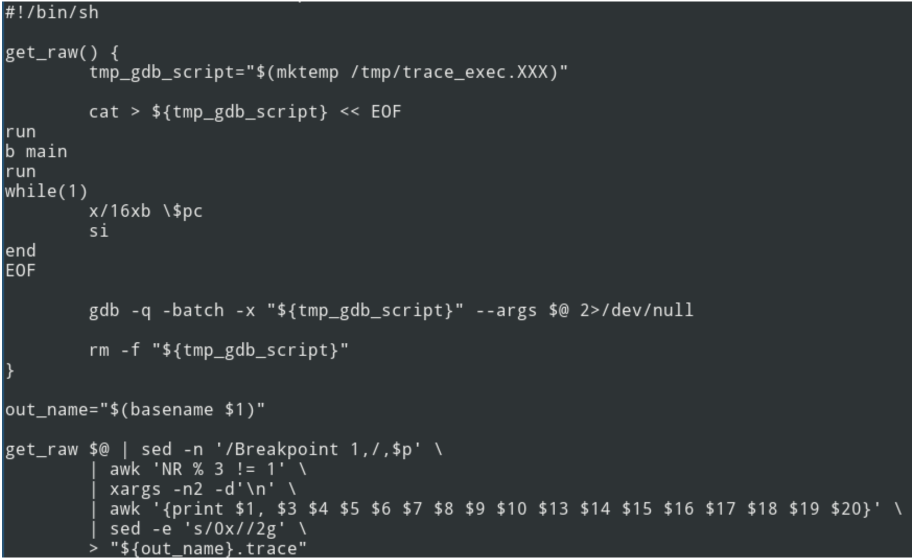
Introduction

- Obtaining execution traces
- Analyzing the flow of the binary
- Dumping the CFG in MCSema's protobuf format
- Lifting the CFG to LLVM IR
- Optimizing out the noise
- Analyzing IR or regenerating an executable file

Why use execution traces?

- We want to work on obfuscated files
- Simplify voluntarily complicated code
- Only the code that is really executed
- Avoid the hassle of indirect jumps

Hacky method



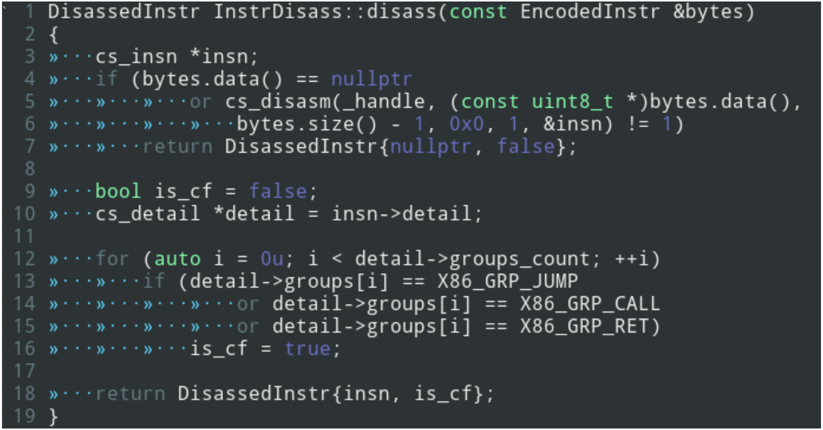
```
#!/bin/sh

get_raw() {
        tmp_gdb_script="$(mktemp /tmp/trace_exec.XXX)"

        cat > ${tmp_gdb_script} << EOF
run
b main
run
while(1)
        x/16xb \$pc
        si
end
EOF

        gdb -q -batch -x "${tmp_gdb_script}" --args $@ 2>/dev/null

        rm -f "${tmp_gdb_script}"
}

out_name="$(basename $1)"

get_raw $@ | sed -n '/Breakpoint 1,/,$p' \
        | awk 'NR % 3 != 1' \
        | xargs -n2 -d'\n' \
        | awk '{print $1, $3 $4 $5 $6 $7 $8 $9 $10 $13 $14 $15 $16 $17 $18 $19 $20}' \
        | sed -e 's/0x//2g' \
        > "${out_name}.trace"
```

Using ptrace

ptrace exposes an interface for observing and controlling the execution of another process:

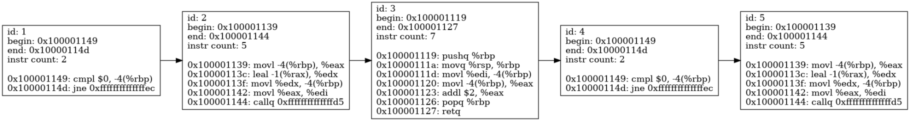
- PTRACE_GETREGS: get the value of %rip
- PTRACE_PEEKTEXT: get bytes at %rip
- PTRACE_SINGLESTEP: single step to next instruction

Instruction disassembly



```
DisassedInstr InstrDisass::disass(const EncodedInstr &bytes)
{
»···cs_insn *insn;
»···if (bytes.data() == nullptr
»···»···»···or cs_disasm(_handle, (const uint8_t *)bytes.data(),
»···»···»···»···bytes.size() - 1, 0x0, 1, &insn) != 1)
»···»···return DisassedInstr{nullptr, false};

»···bool is_cf = false;
»···cs_detail *detail = insn->detail;

»···for (auto i = 0u; i < detail->groups_count; ++i)
»···»···if (detail->groups[i] == X86_GRP_JUMP
»···»···»···»···or detail->groups[i] == X86_GRP_CALL
»···»···»···»···or detail->groups[i] == X86_GRP_RET)
»···»···»···is_cf = true;

»···return DisassedInstr{insn, is_cf};
}
```

Linear CFG



```
id: 1
begin: 0x100001149
end: 0x10000114d
instr count: 2

0x100001149: cmpl $0, -4(%rbp)
0x10000114d: jne 0xffffffffffffffec
```

```
id: 2
begin: 0x100001139
end: 0x100001144
instr count: 5

0x100001139: movl -4(%rbp), %eax
0x10000113c: leal -1(%rax), %edx
0x10000113f: movl %edx, -4(%rbp)
0x100001142: movl %eax, %edi
0x100001144: callq 0xffffffffffffffd5
```

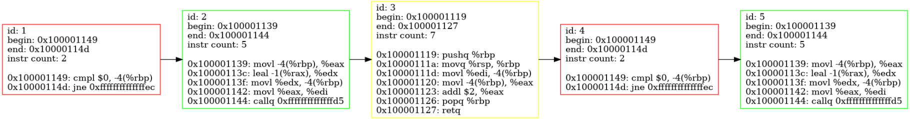
```
id: 3
begin: 0x100001119
end: 0x100001127
instr count: 7

0x100001119: pushq %rbp
0x10000111a: movq %rsp, %rbp
0x10000111d: movl %edi, -4(%rbp)
0x100001120: movl -4(%rbp), %eax
0x100001123: addl $2, %eax
0x100001126: popq %rbp
0x100001127: retq
```

```
id: 4
begin: 0x100001149
end: 0x10000114d
instr count: 2

0x100001149: cmpl $0, -4(%rbp)
0x10000114d: jne 0xffffffffffffffec
```

```
id: 5
begin: 0x100001139
end: 0x100001144
instr count: 5

0x100001139: movl -4(%rbp), %eax
0x10000113c: leal -1(%rax), %edx
0x10000113f: movl %edx, -4(%rbp)
0x100001142: movl %eax, %edi
0x100001144: callq 0xffffffffffffffd5
```

The linear basic blocks detected, one after another.

Linear CFG



```
id: 1
begin: 0x100001149
end: 0x10000114d
instr count: 2

0x100001149: cmpl $0, -4(%rbp)
0x10000114d: jne 0xffffffffffffffec
```

```
id: 2
begin: 0x100001139
end: 0x100001144
instr count: 5

0x100001139: movl -4(%rbp), %eax
0x10000113c: leal -1(%rax), %edx
0x10000113f: movl %edx, -4(%rbp)
0x100001142: movl %eax, %edi
0x100001144: callq 0xffffffffffffffd5
```

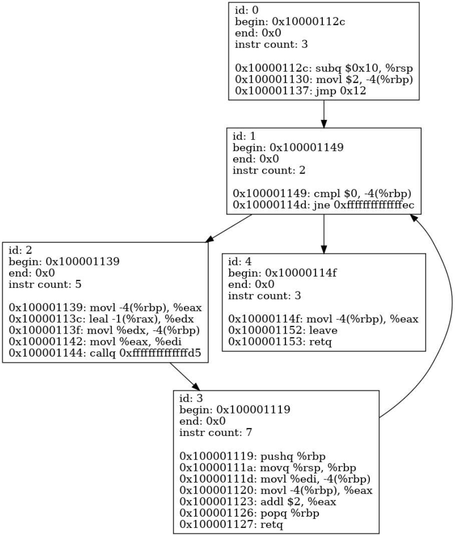
```
id: 3
begin: 0x100001119
end: 0x100001127
instr count: 7

0x100001119: pushq %rbp
0x10000111a: movq %rsp, %rbp
0x10000111d: movl %edi, -4(%rbp)
0x100001120: movl -4(%rbp), %eax
0x100001123: addl $2, %eax
0x100001126: popq %rbp
0x100001127: retq
```

```
id: 4
begin: 0x100001149
end: 0x10000114d
instr count: 2

0x100001149: cmpl $0, -4(%rbp)
0x10000114d: jne 0xffffffffffffffec
```

```
id: 5
begin: 0x100001139
end: 0x100001144
instr count: 5

0x100001139: movl -4(%rbp), %eax
0x10000113c: leal -1(%rax), %edx
0x10000113f: movl %edx, -4(%rbp)
0x100001142: movl %eax, %edi
0x100001144: callq 0xffffffffffffffd5
```

There are many duplicate blocks.

Final CFG



```
id: 0
begin: 0x10000112c
end: 0x0
instr count: 3

0x10000112c: subq $0x10, %rsp
0x100001130: movl $2, -4(%rbp)
0x100001137: jmp 0x12
```

```
id: 1
begin: 0x100001149
end: 0x0
instr count: 2

0x100001149: cmpl $0, -4(%rbp)
0x10000114d: jne 0xffffffffffffffec
```

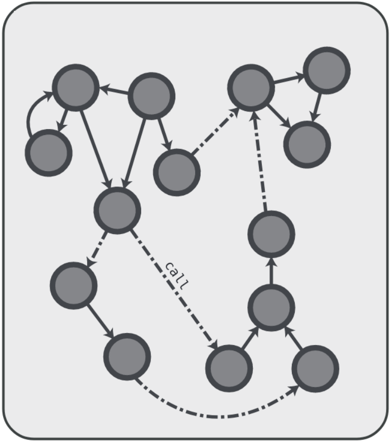
```
id: 2
begin: 0x100001139
end: 0x0
instr count: 5

0x100001139: movl -4(%rbp), %eax
0x10000113c: leal -1(%rax), %edx
0x10000113f: movl %edx, -4(%rbp)
0x100001142: movl %eax, %edi
0x100001144: callq 0xffffffffffffffd5
```

```
id: 4
begin: 0x10000114f
end: 0x0
instr count: 3

0x10000114f: movl -4(%rbp), %eax
0x100001152: leave
0x100001153: retq
```

```
id: 3
begin: 0x100001119
end: 0x0
instr count: 7

0x100001119: pushq %rbp
0x10000111a: movq %rsp, %rbp
0x10000111d: movl %edi, -4(%rbp)
0x100001120: movl -4(%rbp), %eax
0x100001123: addl $2, %eax
0x100001126: popq %rbp
0x100001127: retq
```

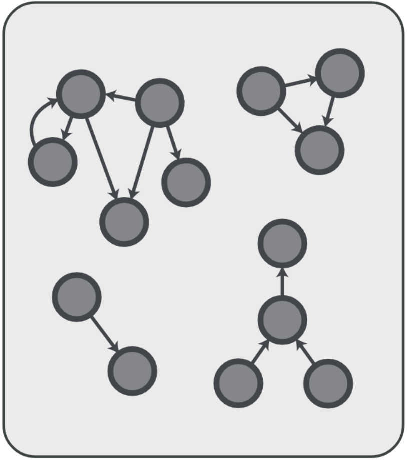
Function detection (1)

- Disassembling with capstone
- Analyzing the flow of the binary
- Generating the CFG



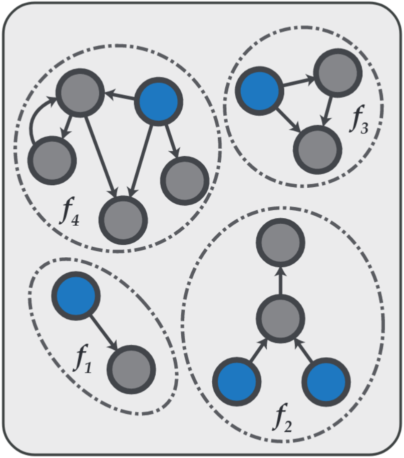
Function detection (2)

- Ignoring the *call* edges
- Basic blocks connected through intraprocedural edges
- Detecting the basic block clusters



Function detection (3)

- Reintroducing the *call* edges
- Following flow until complete block is formed



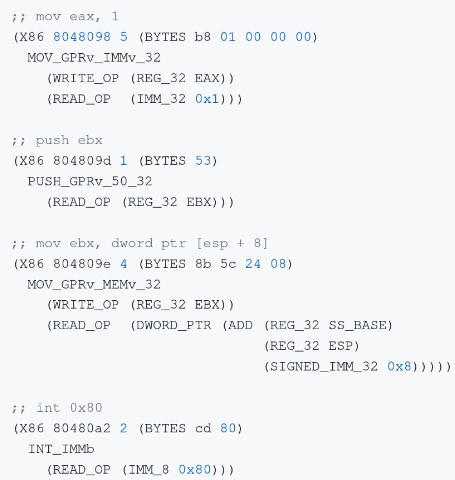
The life of a program

- Dump CFG in wanted protobuf format
- Use MCSema to get some LLVM bytecode with *mcsema-lift*
- Get LLVM assembly language representation with *llvm-dis*
- Optimize this with *opt*
- Rebuild an executable or analyze the optimized IR

How CFGs are lifted

- Declare the lifted functions
- Add segment information to handle cross-references
- **Lift instruction blocks**
- Handle exports if any
- Generate init and fini code
- Optimize to remove function calls at each instruction

Lifting Instructions (1)

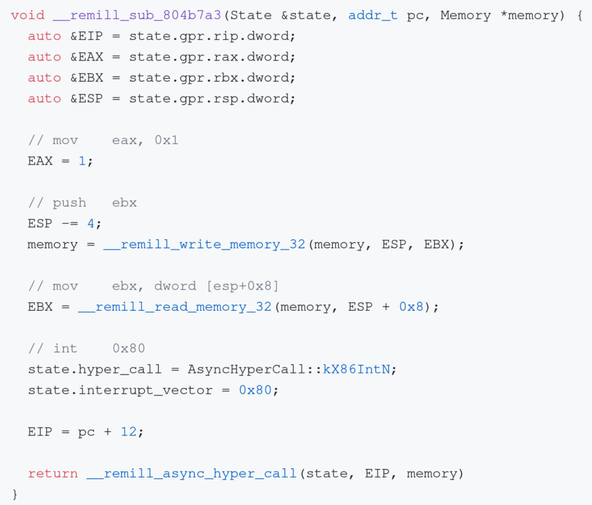


```
;; mov eax, 1
(X86 8048098 5 (BYTES b8 01 00 00 00)
  MOV_GPRv_IMMv_32
    (WRITE_OP (REG_32 EAX))
    (READ_OP  (IMM_32 0x1)))

;; push ebx
(X86 804809d 1 (BYTES 53)
  PUSH_GPRv_50_32
    (READ_OP (REG_32 EBX)))

;; mov ebx, dword ptr [esp + 8]
(X86 804809e 4 (BYTES 8b 5c 24 08)
  MOV_GPRv_MEMv_32
    (WRITE_OP (REG_32 EBX))
    (READ_OP  (DWORD_PTR (ADD (REG_32 SS_BASE)
                              (REG_32 ESP)
                              (SIGNED_IMM_32 0x8)))))

;; int 0x80
(X86 80480a2 2 (BYTES cd 80)
  INT_IMMb
    (READ_OP (IMM_8 0x80)))
```

First we decode the instruction into a higher level Instruction structure.

Lifting Instructions (2)

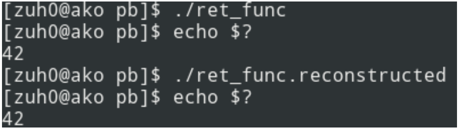
Once the block has been lifted it looks like this:



```
void __remill_sub_804b7a3(State &state, addr_t pc, Memory *memory) {
  auto &EIP = state.gpr.rip.dword;
  auto &EAX = state.gpr.rax.dword;
  auto &EBX = state.gpr.rbx.dword;
  auto &ESP = state.gpr.rsp.dword;

  // mov    eax, 0x1
  EAX = 1;

  // push   ebx
  ESP -= 4;
  memory = __remill_write_memory_32(memory, ESP, EBX);

  // mov    ebx, dword [esp+0x8]
  EBX = __remill_read_memory_32(memory, ESP + 0x8);

  // int    0x80
  state.hyper_call = AsyncHyperCall::kX86IntN;
  state.interrupt_vector = 0x80;

  EIP = pc + 12;

  return __remill_async_hyper_call(state, EIP, memory)
}
```

Rebuilding an executable file

An executable file can then be regenerated using remill's custom build of *clang* and mcsema's runtime static library *libmcsema_rt64*.



```
[zuh0@ako pb]$ ./ret_func
[zuh0@ako pb]$ echo $?
42
[zuh0@ako pb]$ ./ret_func.reconstructed
[zuh0@ako pb]$ echo $?
42
```

Conclusion

- Better traces
- Memory mappings
- MCSema is a hassle to build
- Ignoring libraries
- The process is hard to automate
- Will it be worth it? Currently testing on a Brainfuck interpreter

Links

https://bitbucket.org/vusec/nucleus
https://github.com/trailofbits/mcsema
https://github.com/trailofbits/remill