# Reverse engineering through execution trace analysis

Gabriel Duque & Tanguy Dubroca

# Why are we doing this ?

- Reversing virtual machines is a hassle
- Their state is optimizable

## The lifetime of a program

- Capture traces with different inputs
- Analyze traces to build a CFG
- Lift our CFG to LLVM bytecode
- Run several optimization passes
- Rebuild an executable or analyze the LLVM IR

# What is a trace ?

# What is the purpose of a trace ?

# What is the purpose of a trace ? - Code Coverage

# What is the purpose of a trace ? - Reverse-Engineering

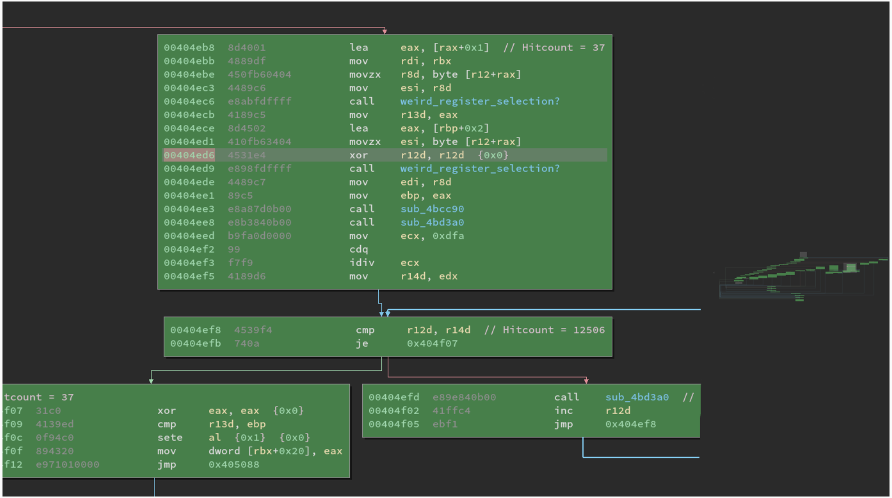


**Figure 1:** trace

```
0x1337a707: mov dword ptr [rbx - 4], eax
r14 : 0x0000000000000000 r15 : 0x000000009b7fdbf0 rcx : 0x000000001337a707 rsi : 0x00007f39fbaa4fb8
r10 : 0x0000000000000008 rbx : 0x00007f39fba9400c rsp : 0x00007ffeae7a3708 r11 : 0x0000000000000246
 r8 : 0x0000000000000000 rdx : 0x0000000000000020 rip : 0x000000001337a707  r9 : 0x00007f39fba72be0
r12 : 0x000000000000ff8c rbp : 0x00007f39fba95000 rdi : 0x0000000000000000 rax : 0x0000000000000020
----------------------------------------
0x1337a70a: sub rbx, 4
r14 : 0x0000000000000000 r15 : 0x000000009b7fdbf0 rcx : 0x000000001337a707 rsi : 0x00007f39fbaa4fb8
r10 : 0x0000000000000008 rbx : 0x00007f39fba9400c rsp : 0x00007ffeae7a3708 r11 : 0x0000000000000246
 r8 : 0x0000000000000000 rdx : 0x0000000000000020 rip : 0x000000001337a70a  r9 : 0x00007f39fba72be0
r12 : 0x000000000000ff8c rbp : 0x00007f39fba95000 rdi : 0x0000000000000000 rax : 0x0000000000000020
----------------------------------------
0x1337a70e: jmp 0x133707de
r14 : 0x0000000000000000 r15 : 0x000000009b7fdbf0 rcx : 0x000000001337a707 rsi : 0x00007f39fbaa4fb8
r10 : 0x0000000000000008 rbx : 0x00007f39fba94008 rsp : 0x00007ffeae7a3708 r11 : 0x0000000000000246
 r8 : 0x0000000000000000 rdx : 0x0000000000000020 rip : 0x000000001337a70e  r9 : 0x00007f39fba72be0
r12 : 0x000000000000ff8c rbp : 0x00007f39fba95000 rdi : 0x0000000000000000 rax : 0x0000000000000020
...
```

## What is the purpose of a trace ? - Be creative

- Traces are raw data
- They are not constrained to a specific use case
- Example projects:
    - Griffin: https://www.microsoft.com/en-us/research/wp-content/uploads/2017/01/griffin-asplos17.pdf
    - HeNet: https://arxiv.org/pdf/1801.02318

# How to capture a trace ?

# How to capture a trace ? - ptrace

- Syscall giving control over a process
- Can singlestep through the code
- Gives access to registers
- **But** painfully slow

# How to capture a trace ? - Emulators

- Allows to debug any kind of code
- Quite easy to implement
- Extensible
- **But** may not execute correctly the target executables

# How to capture a trace ? - Dynamic instrumentation

- Frameworks such as DynamoRIO, Pintools, Valgrind
- Very fast compared to previous methods
- **But** may add significant overhead during execution

# How to capture a trace ? - Hardware tracing

- In our case Intel PT
- Can trace anything (userland, kernel, hypervisor)
- Fastest method
- Low overhead (around 5% for intel pt)
- **But** difficult setup, trace loss and decoding overhead

# Methods used for the project

- ptrace
- intel PT

# Trace format

| size | Description |
|---|---|
| 8 | Magic (0xe9cae282c414b97d) |
| 8 | Edge count |
| edge count * sizeof(trace_entry) | Edge entries |
| 8 | Program mapping count |
| mapping count * sizeof(mapping_entry) | Memory mapping entries |

# Trace format

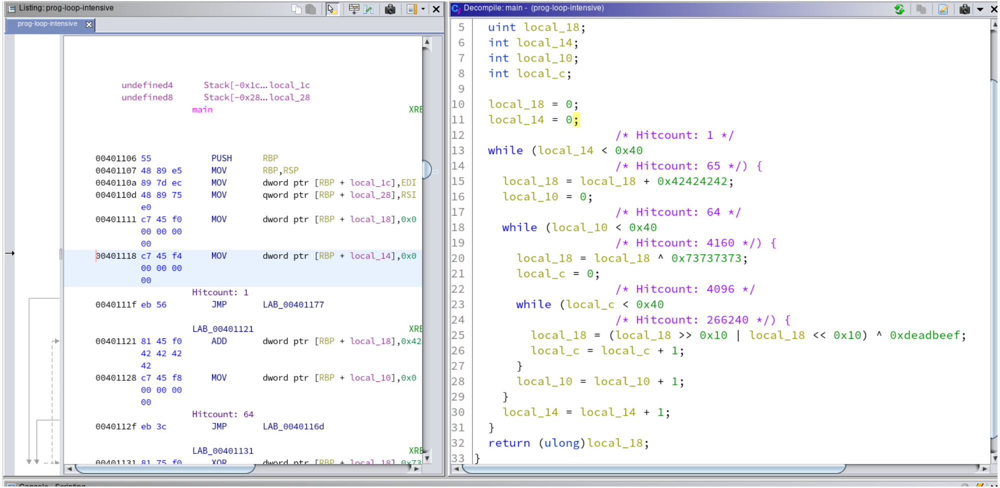


**Figure 2:** ghidra plugin

# What we need

Necessary information for native code lifting:

- Functions
- Basic blocks
  - Succesors
- Instructions
  - Cross-references
    - Code
    - Data

# Building the control flow graph

Two passes:

- First pass: build a linear CFG
  - Iterate through instructions
  - Add them to the current basic block
  - Start a new basic block when we find a control flow instruction
- Second pass: deduce the real CFG
  - Detect loops
  - Merge parents and children vectors

# Linear CFG

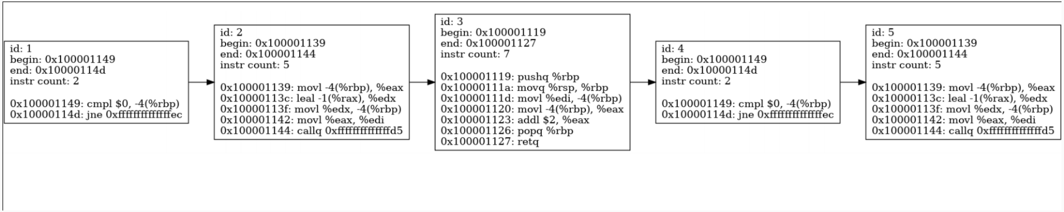


**Figure 3:** linear

# Duplicates

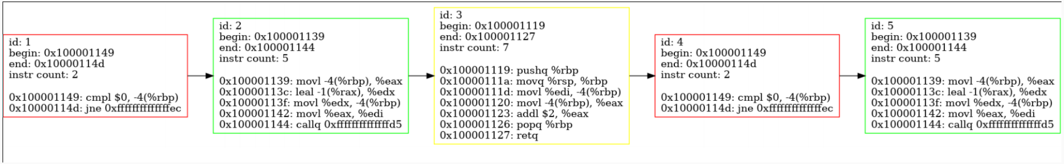


**Figure 4:** color

# The real loop

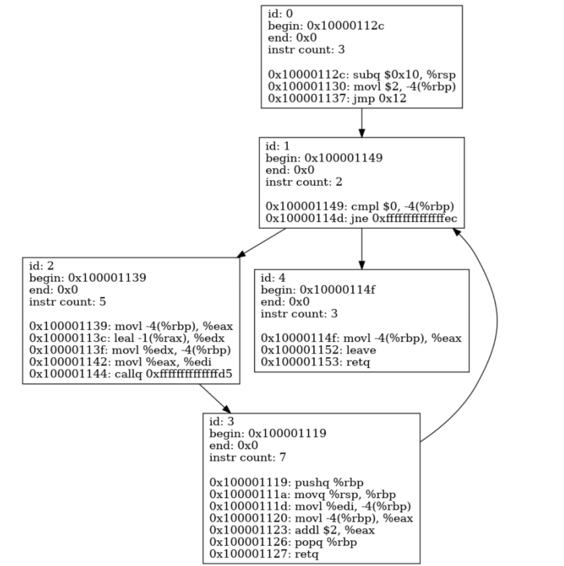


**Figure 5:** loop

# Where da function at ? (0)

- Disassemble with capstone
- Analyze the flow of the binary
- Generate the CFG

# Where da function at ? (0)

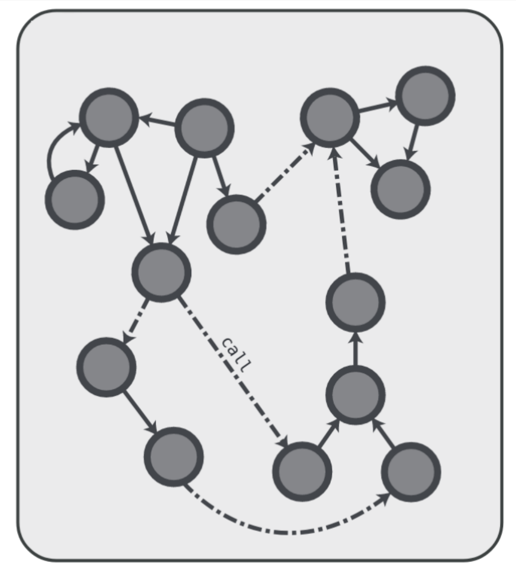


**Figure 6:** nucleus 0

## Where da function at ? (1)

- Ignore *call* edges
- Basic blocks are connected through intraprocedural edges
- Detect basic block clusters

# Where da function at ? (1)

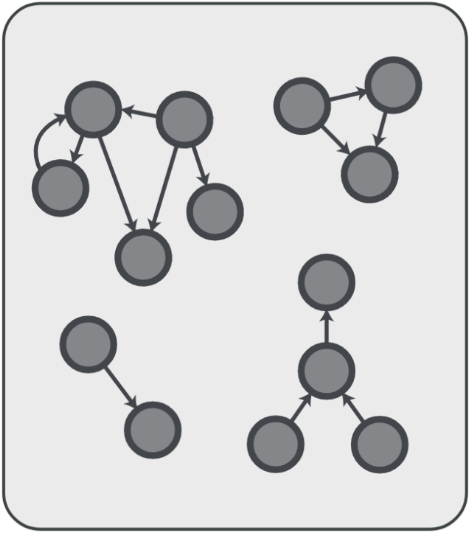

**Figure 7:** nucleus 1

# Where da function at ? (2)

- Reintroduce *call* edges
- Start at entrypoints
- Follow recursively until complete functions are formed

# Where da function at ? (2)

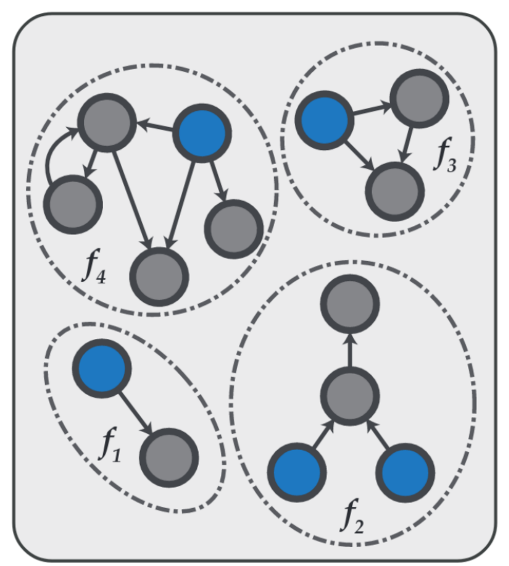


**Figure 8:** nucleus 2

We have:

- Functions

## What now ?

We have:

- Functions
- Blocks

## What now ?

We have:

- Functions
- Blocks
- Instructions

**Cross-references**

# Xrefs

- Internal or external ?
- Code or data ?

# Everything works

- Lifting works.
- Optimizing works.
- Rebuilding works.

# What could go wrong ?

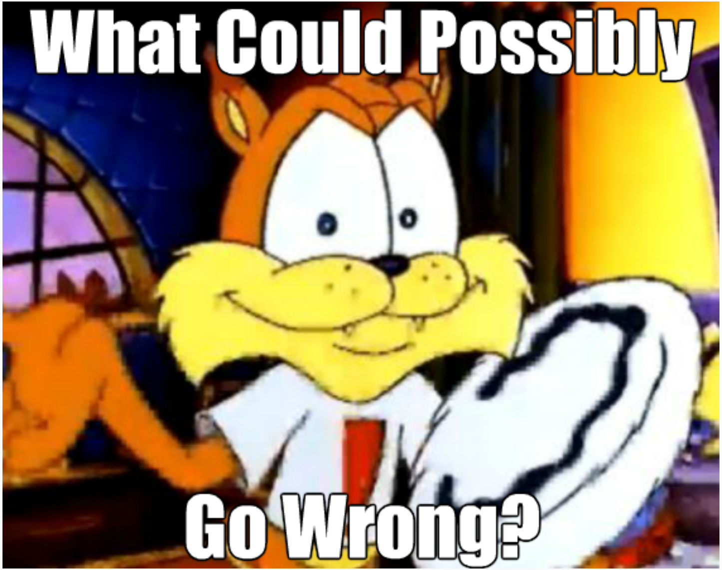


**Figure 9:** meme1

# What could go wrong ?

```
void sub_40076d_main(astruct *param_1,long param_2)

{
  int *piVar1;
  int *piVar2;
  int iVar3;
  long *plVar4;
  undefined8 uVar5;
  undefined *puVar6;
  int *piVar7;
  byte bVar8;
  uint uVar9;
  byte bVar10;
  uint uVar11;
  int iVar12;
  long lVar13;
  uint uVar14;
  uint uVar15;
  long lVar16;
  ulong uVar17;
  uint uVar18;
  long lVar19;
  bool bVar20;
  bool bVar21;

  lVar16 = param_1->field_0x908;
  *(long *)(lVar16 + -8) = param_1->field_0x918;
  param_1->field_0x918 = lVar16 + -8;
  param_1->field_0x908 = lVar16 + -0x68;
  *(undefined4 *)(lVar16 + -0x5c) = *(undefined4 *)&param_1->field_0x8f8;
  *(ulong *)(lVar16 + -0x68) = param_1->field_0x8e8;
  *(undefined8 *)(lVar16 + -0x10) = *(undefined8 *)(param_1->field_0x878 + 0x28);
  *(undefined4 *)(lVar16 + -0x44) = 0;
  *(undefined8 *)(lVar16 + -0x4c) = 0x100000001;
  iVar12 = *(int *)(lVar16 + -0x4c);
  param_1->field_0x8c8 = (long)iVar12;
  uVar17 = (long)*(int *)(lVar16 + -0x48) * 0xb + (long)iVar12;
  puVar6 = maze + uVar17;
  param_1->field_0x811 = (-(puVar6 < maze) | -(0xfffffffff9fbf7f < uVar17)) & 1;
  uVar14 = (int)((ulong)puVar6 & 0xff) - ((uint)(((ulong)puVar6 & 0xff) >> 1) & 0x55);
  uVar14 = (uVar14 >> 2 & 0x33333333) + (uVar14 & 0x33333333);
  param_1->field_0x813 = ~(byte)(((uVar14 >> 4) + uVar14 & 0x10f0f0f) * 0x1010101 >> 0x18) & 1;
  param_1->field_0x815 = ((byte)uVar17 ^ 0x80 ^ (byte)puVar6) >> 4 & 1;
  param_1->field_0x817 = puVar6 == (undefined *)0x0;
  param_1->field_0x819 = (byte)((ulong)puVar6 >> 0x3f);
  param_1->field_0x81d = ((ulong)puVar6 >> 0x3f) + ((uVar17 ^ (ulong)puVar6) >> 0x3f) == 2;
  *puVar6 = 0x58;
  param_1->field_0x8d8 = 7;
  param_1->field_0x8e8 = 0xb;
  param_1->field_0x8f8 = 0x403518;
  param_1->field_0x8a8 = 0;
  lVar16 = param_1->field_0x908;
  *(long *)(lVar16 + -8) = param_2 + 0x74;
  param_1->field_0x908 = lVar16 + -8;
  param_1->field_0x9a8 = param_2 + -0x1cd;
  uVar5 = ext_6010e8_printf(param_1);
  lVar16 = param_1->field_0x9a8;
  param_1->field_0x8d8 = (ulong)*(uint *)(param_1->field_0x918 + -0x40);
  param_1->field_0x8e8 = (ulong)*(uint *)(param_1->field_0x918 + -0x44);
  param_1->field_0x8f8 = 0x403530;
  param_1->field_0x8a8 = 0;
  lVar19 = param_1->field_0x908;
  *(long *)(lVar19 + -8) = lVar16 + 0x17;
  param_1->field_0x908 = lVar19 + -8;
  param_1->field_0x9a8 = lVar16 + -0x241;
  uVar5 = ext_6010e8_printf(param_1,uVar5);
  lVar16 = param_1->field_0x9a8;
  param_1->field_0x8e8 = (ulong)*(uint *)(param_1->field_0x918 + -0x3c);
  param_1->field_0x8f8 = 0x403548;
  param_1->field_0x8a8 = 0;
  lVar19 = param_1->field_0x908;
  *(long *)(lVar19 + -8) = lVar16 + 0x14;
  param_1->field_0x908 = lVar19 + -8;
  param_1->field_0x9a8 = lVar16 + -600;
  uVar5 = ext_6010e8_printf(param_1,uVar5);
  lVar16 = param_1->field_0x9a8;
  param_1->field_0x8f8 = 0x403560;
  lVar19 = param_1->field_0x908;
  *(long *)(lVar19 + -8) = lVar16 + 10;
  param_1->field_0x908 = lVar19 + -8;
  param_1->field_0x9a8 = lVar16 + -0x28c;
  uVar5 = ext_6010d8_puts(param_1,uVar5);
  lVar16 = param_1->field_0x9a8;
  param_1->field_0x8f8 = 0x4035a2;
  lVar19 = param_1->field_0x908;
  *(long *)(lVar19 + -8) = lVar16 + 10;
  param_1->field_0x908 = lVar19 + -8;
  param_1->field_0x9a8 = lVar16 + -0x296;
  uVar5 = ext_6010d8_puts(param_1,uVar5);
  lVar16 = param_1->field_0x908;
  lVar19 = param_1->field_0x9a8;
  *(long *)(lVar16 + -8) = lVar19 + 5;
  param_1->field_0x908 = lVar16 + -8;
  sub_4006f6_draw(param_1,lVar19 + -0x12a,uVar5);
  lVar16 = param_1->field_0x9a8;
  uVar17 = param_1->field_0x918 - 0x30;
  param_1->field_0x8a8 = uVar17;
  param_1->field_0x8d8 = 0x1c;
  param_1->field_0x8e8 = uVar17;
  param_1->field_0x8f8 = 0;
  lVar19 = param_1->field_0x908;
  *(long *)(lVar19 + -8) = lVar16 + 0x16;
  param_1->field_0x908 = lVar19 + -8;
  param_1->field_0x9a8 = lVar16 + -0x275;
  uVar5 = ext_6010f0_read(param_1);
  lVar16 = param_1->field_0x9a8 + 0x1ee;
  do {
    lVar19 = param_1->field_0x918;
    uVar15 = *(uint *)(lVar19 + -0x3c);
    uVar14 = uVar15 - 0x1b;
    *(bool *)&param_1->field_0x811 = uVar15 < 0x1b;
```

# What could go wrong ?

```c
uVar14 = uVar15 - 0x1b;
*(bool *)&param_1->field_0x811 = uVar15 < 0x1b;
uVar11 = (uVar14 & 0xff) - (uVar14 >> 1 & 0x55);
uVar11 = (uVar11 >> 2 & 0x33333333) + (uVar11 & 0x33333333);
param_1->field_0x813 = ~(byte)(((uVar11 >> 4) + uVar11 & 0x10f0f0f) * 0x1010101 >> 0x18) & 1;
param_1->field_0x815 = ~(byte)(uVar14 ^ uVar15) >> 4 & 1;
bVar20 = uVar14 == 0;
param_1->field_0x817 = bVar20;
bVar8 = (byte)(uVar14 >> 0x1f);
param_1->field_0x819 = bVar8;
bVar21 = ((uVar14 ^ uVar15) >> 0x1f) + (uVar15 >> 0x1f) == 2;
param_1->field_0x81d = bVar21;
bVar21 = (bVar8 != 0) == bVar21;
lVar13 = 10;
if (bVar20 || !bVar21) {
  lVar13 = -0x1e9;
}
lVar13 = lVar13 + lVar16;
if (bVar21 && !bVar20) {
  param_1->field_0x8f8 = 0x403642;
  lVar16 = param_1->field_0x908;
  *(long *)(lVar16 + -8) = lVar13 + 10;
  param_1->field_0x908 = lVar16 + -8;
  param_1->field_0x9a8 = lVar13 + -0x4b3;
  uVar5 = ext_6010d8_puts(param_1,uVar5);
  param_1->field_0x8a8 = 1;
  plVar4 = (long *)param_1->field_0x918;
  uVar17 = *(ulong *)(param_1->field_0x878 + 0x28) ^ plVar4[-1];
  param_1->field_0x8e8 = uVar17;
  param_1->field_0x811 = 0;
  uVar14 = (int)(uVar17 & 0xff) - ((uint)((uVar17 & 0xff) >> 1) & 0x55);
  uVar14 = (uVar14 >> 2 & 0x33333333) + (uVar14 & 0x33333333);
  param_1->field_0x813 = ~(byte)(((uVar14 >> 4) + uVar14 & 0x10f0f0f) * 0x1010101 >> 0x18) & 1;
  bVar21 = uVar17 == 0;
  param_1->field_0x817 = bVar21;
  param_1->field_0x819 = (byte)(uVar17 >> 0x3f);
  param_1->field_0x81d = false;
  param_1->field_0x815 = 0;
  if (bVar21) {
    param_1->field_0x918 = *plVar4;
    param_1->field_0x9a8 = plVar4[1];
    *(long **)&param_1->field_0x908 = plVar4 + 2;
    return;
  }
  lVar16 = param_1->field_0x9a8 + 0x14 + (ulong)bVar21 * 5;
  lVar19 = param_1->field_0x908;
  *(long *)(lVar19 + -8) = lVar16 + 5;
  param_1->field_0x908 = lVar19 + -8;
  param_1->field_0x9a8 = lVar16 + -0x4c1;
  uVar5 = ext_6010e0___stack_chk_fail(param_1,uVar5);
  goto LAB_00401ad5;
}
*(undefined4 *)(lVar19 + -0x38) = *(undefined4 *)(lVar19 + -0x44);
*(undefined4 *)(lVar19 + -0x34) = *(undefined4 *)(lVar19 + -0x40);
bVar8 = *(byte *)(lVar19 + -0x30 + (long)*(int *)(lVar19 + -0x3c));
uVar14 = SEXT14((char)bVar8);
piVar1 = (int *)(lVar19 + -0x44);
```

**Figure 10:** cancer 3


Figure 11: meme3

**Wait a minute... Was this supposed to be a deobfuscation or an obfuscation tool?**

# Custom LLVM IR wrapper

```c
#include <stddef.h>

struct State {
    size_t rdi;
    size_t rsi;
    size_t rax;
    size_t r11;
};
```

```c
static void sub_loop_0(struct State *s_ptr) {
    s_ptr->rax = s_ptr->rdi;
    for (s_ptr->r11 = 0; s_ptr->r11 != s_ptr->rsi; ++(s_ptr->r11))
        s_ptr->rax += s_ptr->rsi;
    return;
}
```

```c
size_t wrapper_loop(size_t a, size_t b) {
    struct State s;
    s.rdi = a;
    s.rsi = b;

    sub_loop_0(&s);
    return s.rax;
}
```

# Custom LLVM IR wrapper

```
define i64 @wrapper_loop(i64, i64) {
    %3 = alloca %struct.State
    ; set struct.State.rdi and struct.State.rsi before entering @sub_loop_0
    ; %4 is a pointer to struct.State.rdi
    ; %5 is a pointer to struct.State.rsi
    %4 = getelementptr %struct.State, %struct.State* %3, i32 0, i32 0
    store i64 %0, i64* %4
    %5 = getelementptr %struct.State, %struct.State* %3, i32 0, i32 1
    store i64 %1, i64* %5

    ; call our static sub function
    call void @sub_loop_0(%struct.State* %3)

    ; return the right value: struct.State.rax
    %6 = getelementptr %struct.State, %struct.State* %3, i32 0, i32 2
    %7 = load i64, i64* %6
    ret i64 %7
}
```


**Figure 12:** meme2

# sed -i -e ‘s/noinline/inline/g’ *.ll

A big problem still remains memory access generates function calls to intrinsic procedures.

Because of these generated calls to functions in *remill* functions are not optimizable.

We have to write an optimization pass to force-inline them.

# Conclusion

- Looked like a good idea.
- Was not a good idea.
- We would have to rewrite a big part of *mcsema*/*remill*.

# Questions ?

Do you have any questions?