# Compilation using LLVM

**Juan Manuel Martinez Caamaño (@jmmartinez)**

**Quarkslab**

# Last course

- Introduction to static analysis: data-flow analysis
- Tainting and Backwards slice

# Today's objective

- Dissassembling
- Instrumentation
- Breaking OpaqueConstants
- Profile-Guided-Optimization

# Decompilation

# Decompilation

Moving from binary to a high level representaiton

# Decompilation

The Good:

- Useful for analysis and reverse engineering

# Decompilation

The Bad:

- The tools are very limited
  - Many architectures
  - Different ABIs
  - Information is lost during compilation
  - Different runtimes
  - Different object files formats (ELF, COFF, MachO)

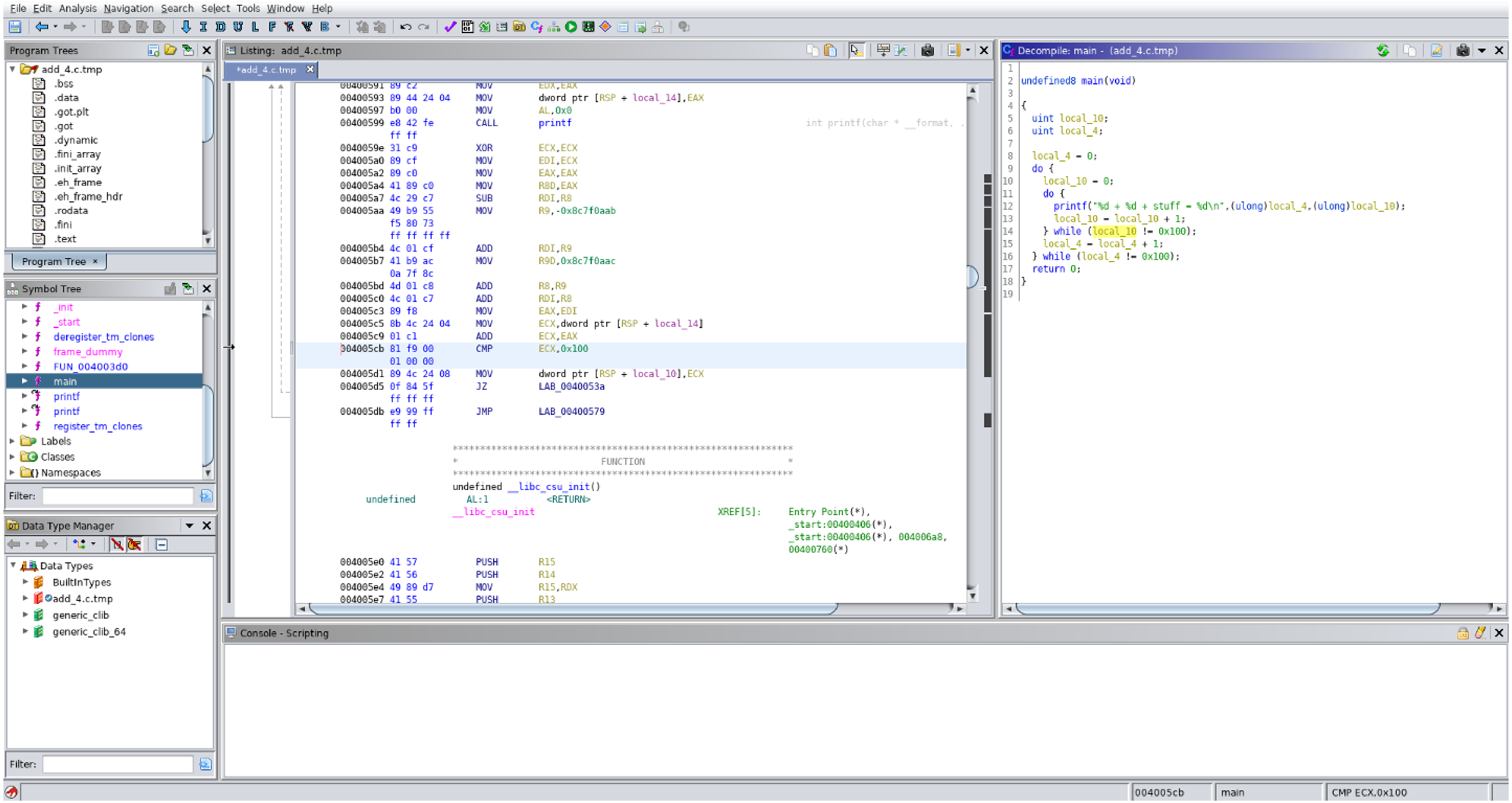
## How they work

- Relocations are evaluated up to a certain point
- Use symbol table to discover entry points to functions
- Start decoding instruction by instruction, map each instruction to it's LLVM equivalent
- When a jump/call is reached, continue dissasembling from the new offset

# Ghidra

- A reverse engineering suit by the NSA
- Open Source

# Ghidra - Use



File Edit Analysis Navigation Search Select Tools Window Help
Program Trees
Listing: add_4.c.tmp
Decompile: main - (add_4.c.tmp)
undefined8 main(void)
{
uint local_10;
uint local_4;
local_4 = 0;
do {
local_10 = 0;
do {
printf("%d + %d + stuff = %d\n",(ulong)local_4,(ulong)local_10);
local_10 = local_10 + 1;
} while (local_10 != 0x100);
local_4 = local_4 + 1;
} while (local_4 != 0x100);
return 0;
}
Symbol Tree
Data Type Manager
Console - Scripting
004005cb
main
CMP ECX,0x100

## Ghidra - Use

Be careful!

- The disassmbly view is not always reliable

## Retdec

- Open Source decompiler since 2017
- Goes from binary to LLVM-IR

# Retdec - Usage

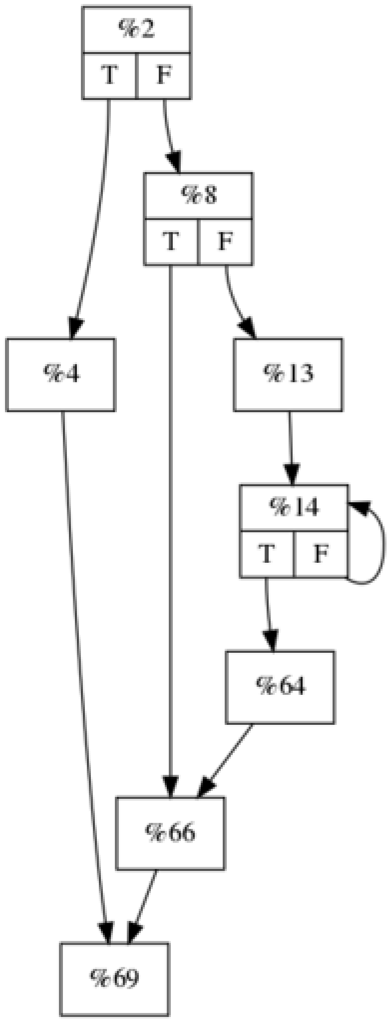
```
clang -O2 ./course2/ex/test/crc32.c -o ./course2/crc32
```

```
retdec-decompiler.py ./course2/crc32 -o ./course2/crc32.c
retdec-decompiler.py --stop-after=bin2llvmir ./course2/crc32 -o ./course2/crc32.dis.ll
opt -O2 -S ./course2/crc32.dis.ll -o ./course2/crc32.dis.opt.ll
```

# Retdec - Usage

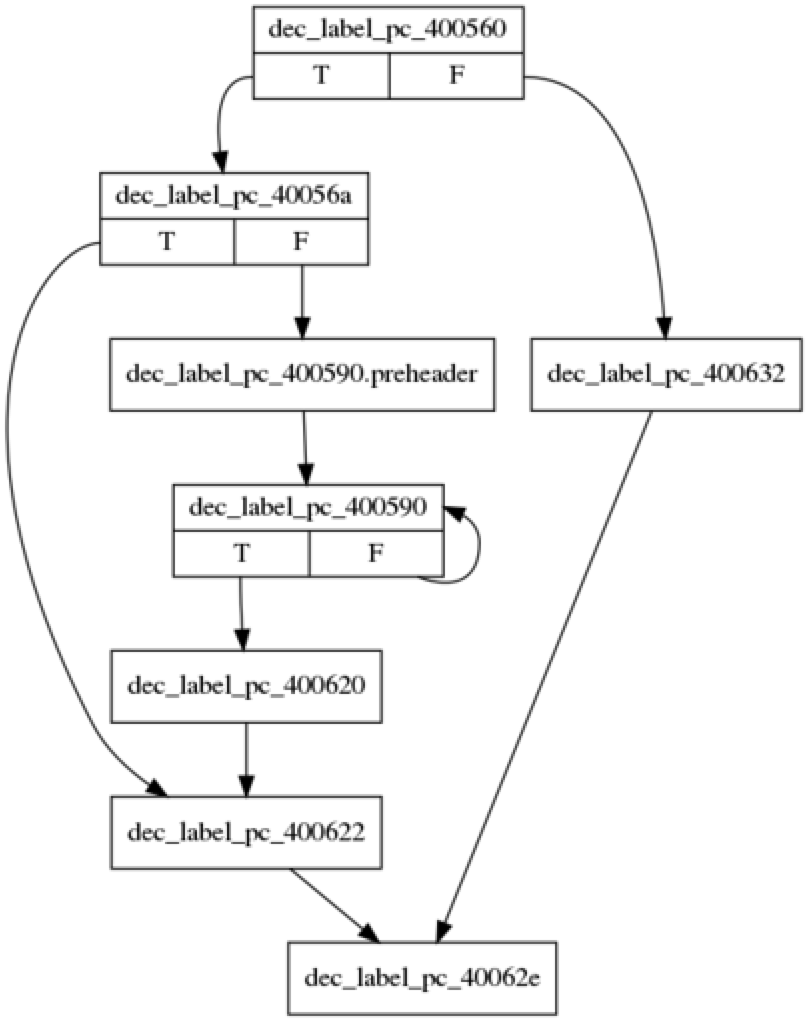
```c
static uint32_t crc32(const unsigned char *message) {
  int i, j;
  uint32_t byte, crc, mask;

  i = 0;
  crc = 0xFFFFFFFF;
  // main_loop
  while (message[i] != 0) {
    byte = message[i]; // Get next byte.
    crc = crc ^ byte;
    for (j = 7; j >= 0; j--) { // Do eight times.
      mask = -(crc & 1);
      crc = (crc >> 1) ^ (0xEDB88320 & mask);
    }
    i = i + 1;
  }
  return ~crc;
}

int main(int argc, char **argv) {
  if (argc < 2) {
    fprintf(stderr, "Usage: %s message\n", argv[0]);
    return 1;
  }

  const char *msg = argv[1];
  const unsigned int crc = crc32((const unsigned char *)msg);

  printf("0x%04x-%s\n", crc, msg);
  return 0;
}
```

# Retdec - Original



CFG for 'main' function

# Retdec - Recovered



CFG for 'main' function

## What now ?

We can start to write tools to transform the code and break obfuscations on a portable representation.

# Instrumentation

## Instrumentation

Introduce mechanisms to meassure, trace and control the execution of the program.

# Tools

- Debuggers: gdb, lldb, ptrace, Pin, Frida, ...
- Hooking: LD_PRELOAD, weak functions, ...
- Emulators: QEMU, ...
- Patchable functions: prologue to inject a jump to redirect the control

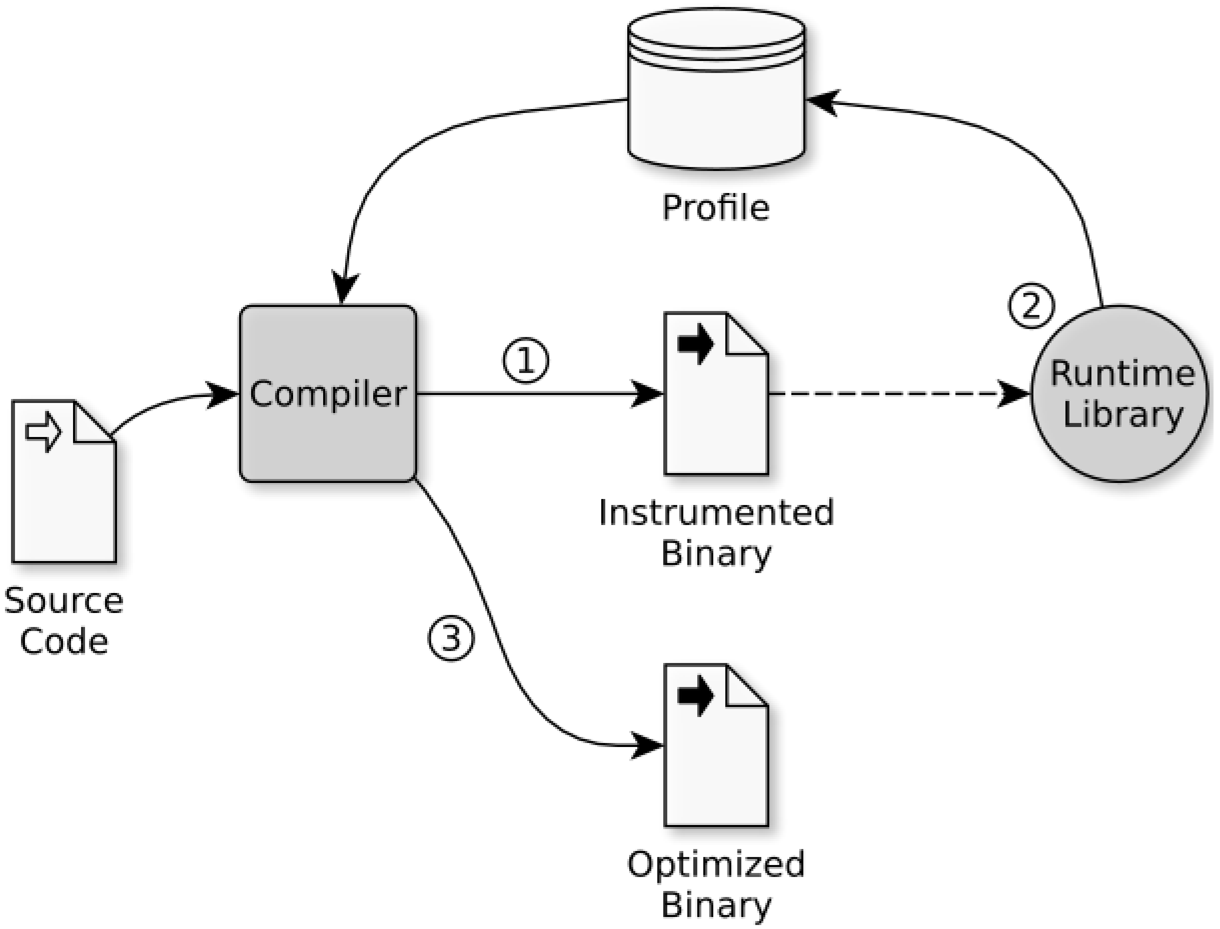
# Why write our own tools in LLVM ?

## Why write our own tools in LLVM ?

To have a 1x1 mapping between what is instrumented and our LLVM-IR.

We can use directly the analysis results in our LLVM code.

# Tool Implementation



## Breaking OpaqueConstants

OpaqueConstants replaces constants by complex expressions that depend on the contexts.

These expressions always yield a constant value: after 1 execution, after 10, after 1000, ...

## Breaking OpaqueConstants

The attack:

- Monitor the values given by every instruction in the llvm-ir and check for constants.
- Replace instructions in the llvm-ir by their constant value, and reoptimize the code.

# Profile-Guided-Optimization

Optimize the code based on a representative set of executions

Can recover information that is very difficult to deduce statically:

- Functions that are rarely executed
- Hot paths
- Dependencies

## Profile-Guided-Optimization

Classical usages:

- Used for code placement
- Used for aggresive loop optimizations and alias analysis

# Profile-Guided-Optimization

The LLVM by default provides two modes:

- Instrumentation: Clang generates calls in the IR that collect information about how many times functions are executed
- Sampling: An external profiler, like perf, collects stacktraces in intervals to know which functions are executed

## Profile-Guided Optimization

The set of inputs used to obtain the profile must be relevant !

# Conclusions

- Decompilation tools remain limited
- A reverse engineer does not try to retreive an exact answer
    - Good enough answers often work
- Dynamic analysis can retrive conclusions about a program from obserivng a few executions