The Decompilation Problem

CompSoc Talk

Louis-Emile Ploix — HT 2026

Problem Statement

- Objective is to reverse compilation
- Takes a compiled binary and produces equivalent, human readable, source code
- Incredibly useful for security research (both good and evil)
- Allows user to understand compiled code at a high level
- Metrics to optimise for:
  - Correctness (modulo some assumptions)
  - Readability / simplicity of the code
  - Wide architecture/binary support

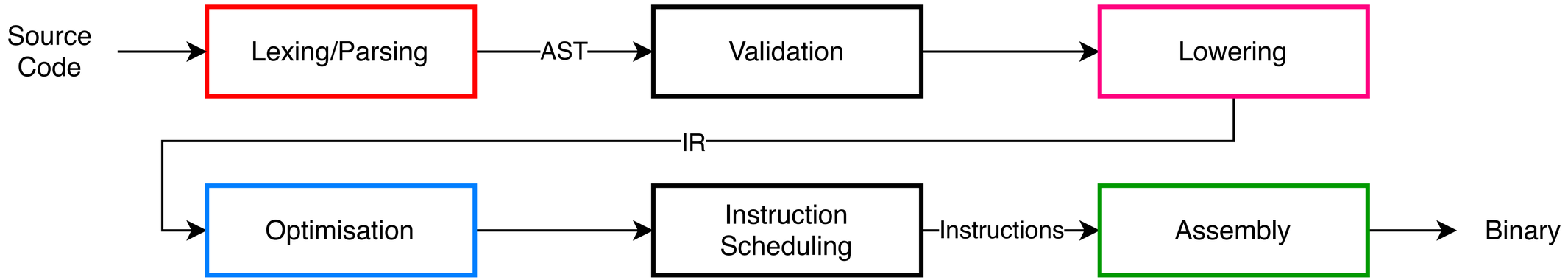
Existing Solutions

- GHIDRA - Open source, NSA, wide architecture support, ugly UI!
- BINARYNINJA - Closed source with free version, my personal favourite!
- Angr  - Open source, quite academic, but pretty interesting!
- Other good commercial solutions exist to…

Why is that difficult?

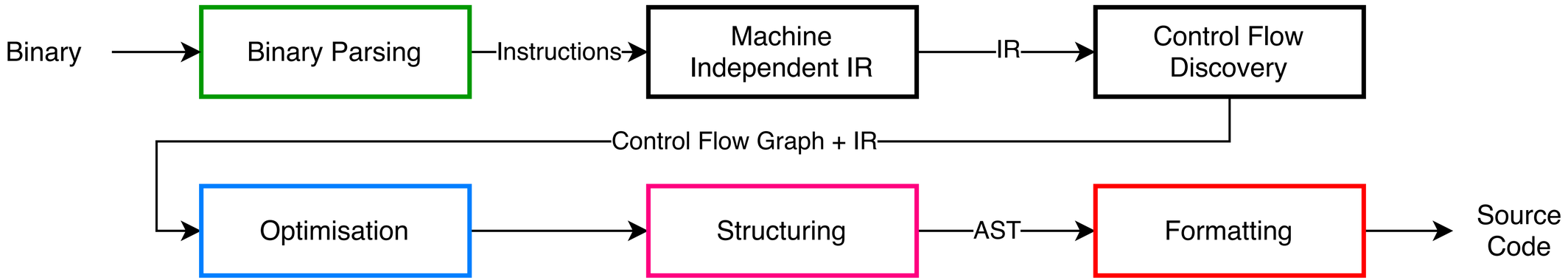
- Compilation loses a lot of information (exceptions exist, e.g. Java):
  - Variable names
  - Function names (for stripped binaries)
  - Types
  - Structure (will turn out to be particularly important, and difficult)
  - Optimisations inline definitions or compute at compile time
- We find ourselves doing a *lot* of program analysis, which can quickly turn into undecidability issues

Quick refresher on compilers



- Lexing/Parsing - Frontend, ingests source code and produces an AST
- Validation - Check types, lifetimes, static analysis based warnings, etc
- Lowering - Flatten control flow, and reduce to some kind of (mostly) machine independent internal representation
- Optimisation - Rules (called passes) applied iteratively to improve/speed/size etc
- Instruction scheduling - Lower from the machine independent internal representation to a machine dependent one (and maybe optimise more)
- Assembly - Produce machine code embedded in a binary for the relevant target triple

Decompilers have the reverse pipeline



- Binary Parsing - Translate directly to a low level, machine independent intermediate representation
- Optimisation - Rules (also called passes) applied iteratively to simplify (basically the same as the optimisation pass of a compiler!)
- Structuring - Translate goto/labels into structured ASTs (if statements, for loops, etc)
- Formatting - Takes the constructed AST and produces source code

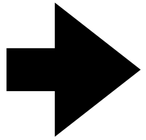
But it’s also kind of the same pipeline?

- In both cases, we:
  1. Parse (source code / machine code)
  2. Transform to IR
  3. Optimise (for similar, but not identical metrics)
  4. Schedule (instruction selection / structuring)
  5. Output

Central Problem: Structure

- Machine code (or assembly) is flat. There are no if statements, while loops, for loops, etc, there are only branches to potentially arbitrary addresses

```
void g() {
    for (int i = 0; i < 10; i++) {
        // ... a ...
        if (c) break;
        // ... b ...
    }
    // ... d ...
}
```

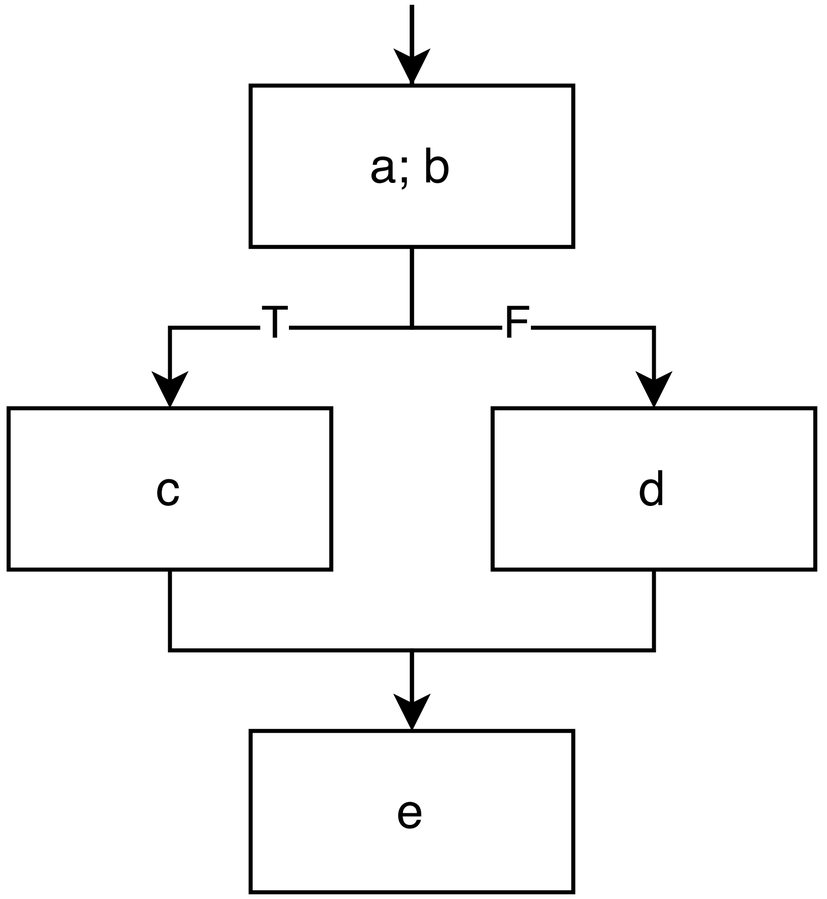


```
 0:   push   %rbp
 1:   mov    %rsp,%rbp
 4:   sub    $0x10,%rsp
 8:   movl   $0x0,-0x4(%rbp)
 f:   jmp    37 <g+0x37>
11:   mov    $0x0,%eax
16:   call   1b <g+0x1b>
1b:   mov    $0x0,%eax
20:   call   25 <g+0x25>
25:   test   %eax,%eax
27:   jne    3f <g+0x3f>
29:   mov    $0x0,%eax
2e:   call   33 <g+0x33>
33:   addl   $0x1,-0x4(%rbp)
37:   cmpl   $0x9,-0x4(%rbp)
3b:   jle    11 <g+0x11>
3d:   jmp    40 <g+0x40>
3f:   nop
40:   mov    $0x0,%eax
45:   call   4a <g+0x4a>
4a:   leave
4b:   ret
```

- In order to recover high level constructs like if statements, we need to first build some kind of control flow graph, and then analyse it.

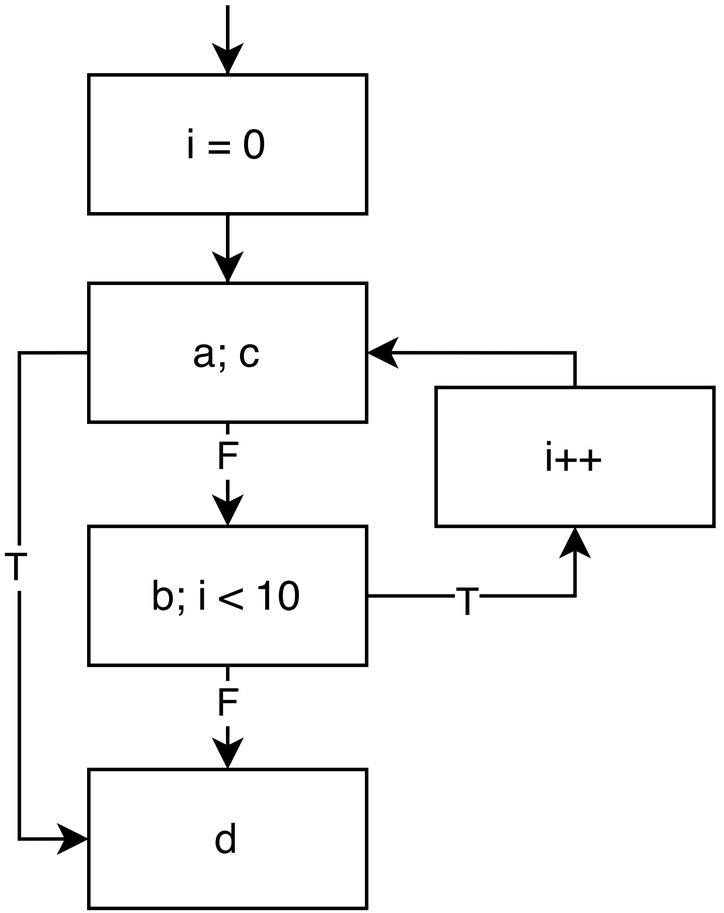
Control Flow Graph: If/Else

```
void f() {
    // ... a ...
    if (b) {
        // ... c ...
    } else {
        // ... d ...
    }
    // ... e ...
}
```



Control Flow Graph: For Loop

```
void g() {
    for (int i = 0; i < 10; i++) {
        // ... a ...
        if (c) break;
        // ... b ...
    }
    // ... d ...
}
```



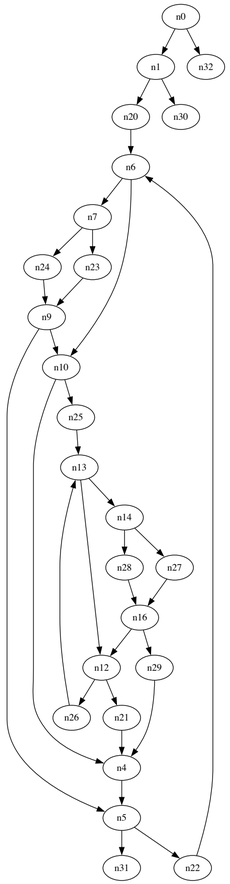
Control Flow Graph Construction

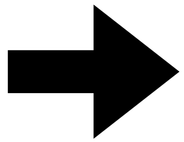
- Building such a control flow graph from the binary is already difficult (technically undecidable), since we need to:
  - Determine the target locations and conditions of every branch
  - Figure out boundaries of basic blocks
- We *can* do it fairly well, but it's heuristic, and only because compilers are predictable:

```
37      /// Construct a sorted list of all data block boundaries in memory.
38      /// This is done by finding all constants which can be created. Constants
39      /// may span multiple statements.
40      /// The results are quite heuristic in that there may be missing blocks,
41      /// and there may be too many blocks.
42      /// FIXME: This is quite slow! It tries to evaluate every expression in the
43      /// binary, which doesn't go great.
44      pub fn map_memory_regions(lir: &[Stmt<AddrLabel>], bin: &Binary) -> Vec<VAddr> {
```

Central Problem: Structuring

- Now we have a CFG, we need to do **graph structuring**. Probably the central problem of decompilation.





```
if (a) return
if (b) return
do {
  if (d) {
    if (e) // ...
    else // ...
    if (f) continue
  }
  // ... etc ...
} while (c)
```

Central Problem: Structuring

- Now we have a CFG, we need to do **graph structuring**. Probably the central problem of decompilation.
- Problem statement:
  - Find an equivalent AST to the input CFG, that is “easy to read”
    - Minimise gotos? Not quite! *Exercise: why?*
    - Minimise size? Not quite! *Exercise: why?*
    - Minimise duplication? Not quite! *Exercise: why?*
    - Actually quite difficult to pin down a metric.

Central Problem: Structuring

- It’s an open question!
- Phoenix (2013): Repeated pattern matching
  - + Simple, Fast, Good for small functions
  - - Not great for large functions, Lots of gotos
- DREAM (2015): Condition graph methods
  - + 0 gotos!
  - - Slow, requires SAT solving, doesn’t work well or look right in practice
- SAILR (2024): Compiler optimisation based pattern matching
  - + All benefits of Phoenix but better at large scale
  - - More complicated to implement (I guess?), still not perfect
- Or somewhere in-between!

Conclusions

- Decompilers have many technically interesting problems inside of them, and they are not solved!
- I do highly recommend looking into decompilers, especially if you want systems experience!

Thanks!