THALIUM

Beyond Assembly: IRs, SLEIGH, P-code, and Lifting in Ghidra

Jack Royer

21/04/2026 Bière et Sécu

jack (dot) royer (at) centralesupelec (dot) com

Motivation

Making sense of binaries

- Sometimes we only have a binary
  - > Malware analysis
  - > Proprietary software
  - > CTF challenges
- Raw hex dumps are difficult to interpret
  - > We need tools that recover semantics from bytes

What is this?

```
01001000
10000011
01110000
00010000
00101010
```

```
48 83 70 10 2a
```

Disassembly

- **Compilers** translate instructions into bytes
- **Disassemblers** try to recover instructions from bytes
- We'll discuss one way of doing this later..
- This is not an easy task...
  - > Code and data may be mixed together
  - > Indirect jumps make control flow unclear
  - > Variable-length instructions complicate parsing

It was mov

```
01001000
10000011
01110000
00010000
00101010
```

```
mov    QWORD PTR [rax+0x10], 42
```

Perfect disassembly is undecidable

Separating code from data in a binary (perfect disassembly) is **undecidable**.

```
jmp label
db 0x90, 0x90, 0x90   ; data? or instructions?
label:
...
```

If you could perfectly distinguish code from data:

- You would know exactly which bytes are ever executed
  - > You would know all possible execution paths
  - > You could decide whether a program halts
- But the **Halting Problem** is undecidable
  - > Perfect disassembly is **undecidable**

```
call foo
db 0xC3  ; executed as code iff foo returns
```

Disassembly is only the first step

We also want to:

- Identify functions
- Recover control flow
  - > `if else` is better than `goto`
- Perform decompilation
  - > pseudo-C is better than assembly
- *Perform deobfuscation? (my research interest :))*

We need to manipulate the semantics of the program

Semantics: how an instruction changes the program state.

Decompile: entry - (hello)

```
1
2  /* WARNING: Control flow encountered bad instruction data */
3
4  void processEntry entry(void)
5
6  {
7    long lVar1;
8    byte in_AF;
9
10   syscall();
11   do {
12     syscall();
13     syscall();
14     for (lVar1 = 0; (&input_buf)[lVar1] != '\n'; lVar1 = lVar1 + 1) {
15       if (9 < (byte)((&input_buf)[lVar1] - 0x30)) goto _start.invalid_input;
16     }
17     if (((in_AF & 1) == 0) || ((in_AF & 1) != 0)) {
18 _start.invalid_input:
19       syscall();
20     }
21     else {
22       if (((in_AF & 1) == 0) && ((in_AF & 1) != 0)) {
23         syscall();
24         syscall();
25                     /* WARNING: Bad instruction - Truncating control flow here */
26         halt_baddata();
27       }
28       syscall();
29     }
30   } while( true );
31 }
32
```

Why we need an abstraction

- x86 instructions often:
  - Modify flags implicitly
  - Use multiple operand sizes
  - Combine memory access and arithmetic

- And what if one day we want to use ARM? or RISC-V? or a custom architecture?

What does this instruction do?

```
xor QWORD PTR [rax+0x10],0x2a
```

- Read the value of `RAX`
- Performs an addition
- Reads a value from memory
- Performs a XOR
- Writes a value to memory
- Zeroes `OF`
- Writes a deterministic value to `ZF`
- Writes an **undefined** value to `AF`

Intermediate Representations (IRs)

IRs: The abstraction

- Create a meta-assembly that can be used in all our analysis
- Lifting: the process of translating an assembly language to an IR

A **good IR** should:

- Have few instruction types
- Use simple and explicit operations
- Stay close enough to machine behavior
- Be easy to analyze
- Still be feasible to lift into
- Easy analysis vs easy lifting is a real tradeoff

Famous IRs

LLVM IR
- Rich type system
- Excellent for optimization passes
- Higher level than machine semantics

VEX
- Used by angr and Valgrind
- Normalized operations for dynamic analysis
- Good for instrumentation and emulation

P-code
- Ghidra's internal IR
- Explicit side effects and temporaries
- Designed for lifting many architectures

Design tension Easy analysis usually wants a cleaner IR.

Easy lifting usually wants an IR that stays close to machine behavior.

Case study: Ghidra and PCode

Why Ghidra?

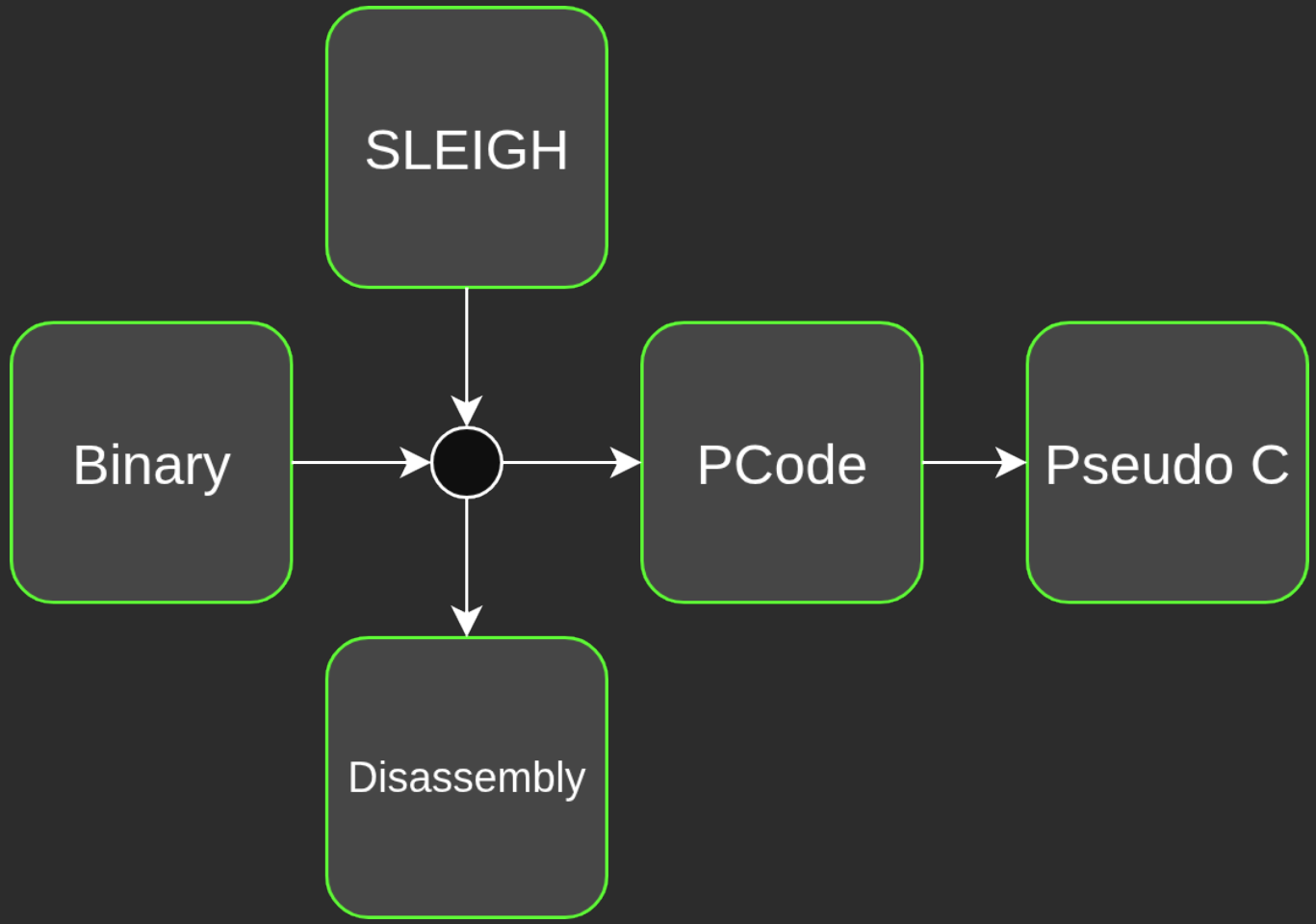
- Widely used reverse engineering framework
- Supports many architectures
- Uses P-code as a common intermediate representation
- SLEIGH makes architecture support extensible

Architecture coverage x86, ARM, MIPS, RISC-V and more are described with SLEIGH.

One semantic core Disassembly, decompilation, and analysis all build on lifted P-code.

Hackable SLEIGH specs, scripts, and plugins make it practical to extend for research work.

Ghidra's architecture



PCode

PCode

- About 70 instructions
- Uses varnodes
  - > A memory space + size + address
- Close to machine semantics
  - > We can manipulate registers, memory, and flags directly
- No implicit behavior
  - > Everything needs to be explicited
  - > We translate one CPU instruction to many P-code instructions

P-Code Reference Manual

COPY	INT_ADD	BOOL_OR
LOAD	INT_SUB	FLOAT_EQUAL
STORE	INT_CARRY	FLOAT_NOTEQUAL
BRANCH	INT_SCARRY	FLOAT_LESS
CBRANCH	INT_SBORROW	FLOAT_LESSEQUAL
BRANCHIND	INT_2COMP	FLOAT_ADD
CALL	INT_NEGATE	FLOAT_SUB
CALLIND	INT_XOR	FLOAT_MULT
USERDEFINED	INT_AND	FLOAT_DIV
RETURN	INT_OR	FLOAT_NEG
PIECE	INT_LEFT	FLOAT_ABS
SUBPIECE	INT_RIGHT	FLOAT_SQRT
POPCOUNT	INT_SRIGHT	FLOAT_CEIL
LZCOUNT	INT_MULT	FLOAT_FLOOR
INT_EQUAL	INT_DIV	FLOAT_ROUND
INT_NOTEQUAL	INT_REM	FLOAT_NAN
INT_LESS	INT_SDIV	INT2FLOAT
INT_SLESS	INT_SREM	FLOAT2FLOAT
INT_LESSEQUAL	BOOL_NEGATE	TRUNC
INT_SLESSEQUAL	BOOL_XOR	CPOOLREF
INT_ZEXT	BOOL_AND	NEW
INT_SEXT		

Cross architecture semantics

amd64

```
00400456 01 d0  ADD   EAX,EDX
  CF = INT_CARRY EAX, EDX
  OF = INT_SCARRY EAX, EDX
  EAX = INT_ADD EAX, EDX
  RAX = INT_ZEXT EAX
  SF = INT_SLESS EAX, 0:4
  ZF = INT_EQUAL EAX, 0:4
  $U28080:4 = INT_AND EAX, 0xff:4
  $U28100:1 = POPCOUNT $U28080:4
  $U28180:1 = INT_AND $U28100:1, 1:1
  PF = INT_EQUAL $U28180:1, 0:1
```

aarch64

```
0010077c 20 00 00 0b  add   w0,w1,w0
  $U12180:4 = COPY w0
  tmpCY = INT_CARRY w1, $U12180:4
  tmpOV = INT_SCARRY w1, $U12180:4
  $U12280:4 = INT_ADD w1, $U12180:4
  tmpNG = INT_SLESS $U12280:4, 0:4
  tmpZR = INT_EQUAL $U12280:4, 0:4
  x0 = INT_ZEXT $U12280:4
```

PCode in action

```
                        undefined main()
     undefined          <UNASSIGNED>   <RETURN>
     undefined4           Stack[-0xc]:4  local_c                                XREF[2]:     00400471(W),
                                                                                              00400474(R)
                        main                                                   XREF[4]:     Entry Point(*),
                                                                                            _start:00400378(*), 0040114c,
                                                                                            004011c0(*)
0040045a 55             PUSH        RBP
                                    $U23c00:8 = COPY RBP
                                    RSP = INT_SUB RSP, 8:8
                                    STORE ram(RSP), $U23c00:8
0040045b 48 89 e5       MOV         RBP,RSP
                                    RBP = COPY RSP
0040045e 48 83 ec 10    SUB         RSP,0x10
                                    CF = INT_LESS RSP, 16:8
                                    OF = INT_SBORROW RSP, 16:8
                                    RSP = INT_SUB RSP, 16:8
                                    SF = INT_SLESS RSP, 0:8
                                    ZF = INT_EQUAL RSP, 0:8
                                    $U28080:8 = INT_AND RSP, 0xff:8
                                    $U28100:1 = POPCOUNT $U28080:8
                                    $U28180:1 = INT_AND $U28100:1, 1:1
                                    PF = INT_EQUAL $U28180:1, 0:1
00400462 be 03 00       MOV         ESI,0x3
         00 00
                                    RSI = COPY 3:8
00400467 bf 05 00       MOV         EDI,0x5
         00 00
                                    RDI = COPY 5:8
0040046c e8 d5 ff       CALL        add                                                undefined add()
         ff ff
                                    RSP = INT_SUB RSP, 8:8
                                    STORE ram(RSP), 0x400471:8
                                    CALL *[ram]0x400446:8
00400471 89 45 fc       MOV         dword ptr [RBP + local_c],EAX
                                    $U4700:8 = INT_ADD RBP, -4:8
                                    $U6a80:4 = COPY EAX
                                    STORE ram($U4700:8), $U6a80:4
00400474 8b 45 fc       MOV         EAX,dword ptr [RBP + local_c]
                                    $U4700:8 = INT_ADD RBP, -4:8
                                    $Ude00:4 = LOAD ram($U4700:8)
                                    EAX = COPY $Ude00:4
```

SLEIGH

SLEIGH: a brief overview

A Domain Specific Language (DSL) to describe a CPU architecture.

Machine model Defines endianness, memory spaces, and registers.

Token Describes instruction encoding as bit fields.

Constructor Maps bit patterns to disassembly and P-code semantics.

```
define endian=little;
define space ram type=ram_space size=4 default;
define space register type=register_space size=4;
define register offset=0 size=4 [ r0 r1 r2 r3 r4 r5 r6 r7 ];

define token instr(16)
  op=(0,5)
  mode=(6,9)
  reg1=(10,12)
  imm=(13,15)
  reg2=(13,15)
;
attach variables [ reg1 reg2 ] [ r0 r1 r2 r3 r4 r5 r6 r7 ];

op2: reg2 is mode=0xa & reg2   { export reg2; }
op2: imm is mode=0xb & imm     { export *[const]:4 imm; }
op2: [reg2] is mode=0xc & reg2 { tmp = *:4 reg2; export tmp;}

:and reg1,op2 is op=0xa & reg1 & op2 { reg1 = reg1 & op2; }
:xor reg1,op2 is op=0xb & reg1 & op2 { reg1 = reg1 ^ op2; }
:or  reg1,op2 is op=0xc & reg1 & op2 { reg1 = reg1 | op2; }
```

Reading SLEIGH

Machine definition

We define two memory spaces:

- **ram**: The RAM
- **registers**: A virtual CPU memory space

We define eight 4-byte registers in that space: `r0` to `r7`.

```
define endian=little;
define space ram type=ram_space size=4 default;
define space register type=register_space size=4;
define register offset=0 size=4 [ r0 r1 r2 r3 r4 r5 r6 r7 ];

define token instr(16)
  op=(0,5)
  mode=(6,9)
  reg1=(10,12)
  imm=(13,15)
  reg2=(13,15)
;
attach variables [ reg1 reg2 ] [ r0 r1 r2 r3 r4 r5 r6 r7 ];

op2: reg2 is mode=0xa & reg2   { export reg2; }
op2: imm is mode=0xb & imm     { export *[const]:4 imm; }
op2: [reg2] is mode=0xc & reg2 { tmp = *:4 reg2; export tmp;}

:and reg1,op2 is op=0xa & reg1 & op2 { reg1 = reg1 & op2; }
:xor reg1,op2 is op=0xb & reg1 & op2 { reg1 = reg1 ^ op2; }
:or  reg1,op2 is op=0xc & reg1 & op2 { reg1 = reg1 | op2; }
```

Tokens

A **token** is a range of bytes. It can hold multiple **fields**, each representing a sub-range of bits.

Here, the last three bits are interpreted differently: as `imm` or as `reg2` depending on the constructor.

```
define endian=little;
define space ram type=ram_space size=4 default;
define space register type=register_space size=4;
define register offset=0 size=4 [ r0 r1 r2 r3 r4 r5 r6 r7 ];

define token instr(16)
  op=(0,5)
  mode=(6,9)
  reg1=(10,12)
  imm=(13,15)
  reg2=(13,15)
;
attach variables [ reg1 reg2 ] [ r0 r1 r2 r3 r4 r5 r6 r7 ];

op2: reg2 is mode=0xa & reg2   { export reg2; }
op2: imm is mode=0xb & imm     { export *[const]:4 imm; }
op2: [reg2] is mode=0xc & reg2 { tmp = *:4 reg2; export tmp;}

:and reg1,op2 is op=0xa & reg1 & op2 { reg1 = reg1 & op2; }
:xor reg1,op2 is op=0xb & reg1 & op2 { reg1 = reg1 ^ op2; }
:or  reg1,op2 is op=0xc & reg1 & op2 { reg1 = reg1 | op2; }
```

Giving meaning to a field

Given the bytes: `0xcafe`

op	mode	reg1	imm
0b001010 = 0xa	0b1011 = 0xb	0b111 = 7	0b111 = 7

Simple numeric: `op = 0xa`, `imm = 7`

We also can `attach` to map field values to registers, names or other values. In our example, `reg1 = 7` becomes `reg1 = r7`.

```
define endian=little;
define space ram type=ram_space size=4 default;
define space register type=register_space size=4;
define register offset=0 size=4 [ r0 r1 r2 r3 r4 r5 r6 r7 ];

define token instr(16)
  op=(0,5)
  mode=(6,9)
  reg1=(10,12)
  imm=(13,15)
  reg2=(13,15)
;
attach variables [ reg1 reg2 ] [ r0 r1 r2 r3 r4 r5 r6 r7 ];

op2: reg2 is mode=0xa & reg2   { export reg2; }
op2: imm is mode=0xb & imm     { export *[const]:4 imm; }
op2: [reg2] is mode=0xc & reg2 { tmp = *:4 reg2; export tmp;}

:and reg1,op2 is op=0xa & reg1 & op2 { reg1 = reg1 & op2; }
:xor reg1,op2 is op=0xb & reg1 & op2 { reg1 = reg1 ^ op2; }
:or  reg1,op2 is op=0xc & reg1 & op2 { reg1 = reg1 | op2; }
```

A constructor

Display section — disassembly print:

```
and reg1, op2
```

Constraint section — when does it match:

```
op=0xa & reg1 & op2
```

Match `op=0xa` and any valid values for `reg1` and `op2`.

Semantic section — what does it mean:

```
{ reg1 = reg1 & op2; }
```

`reg1` and `op2` are replaced with the semantics of their values.

```
define endian=little;
define space ram type=ram_space size=4 default;
define space register type=register_space size=4;
define register offset=0 size=4 [ r0 r1 r2 r3 r4 r5 r6 r7 ];

define token instr(16)
  op=(0,5)
  mode=(6,9)
  reg1=(10,12)
  imm=(13,15)
  reg2=(13,15)
;
attach variables [ reg1 reg2 ] [ r0 r1 r2 r3 r4 r5 r6 r7 ];

op2: reg2 is mode=0xa & reg2   { export reg2; }
op2: imm is mode=0xb & imm     { export *[const]:4 imm; }
op2: [reg2] is mode=0xc & reg2 { tmp = *:4 reg2; export tmp;}

:and reg1,op2 is op=0xa & reg1 & op2 { reg1 = reg1 & op2; }
:xor reg1,op2 is op=0xb & reg1 & op2 { reg1 = reg1 ^ op2; }
:or  reg1,op2 is op=0xc & reg1 & op2 { reg1 = reg1 | op2; }
```

A table

- Often, we want to define the same operation on **registers**, **immediate values**, and **memory locations**.

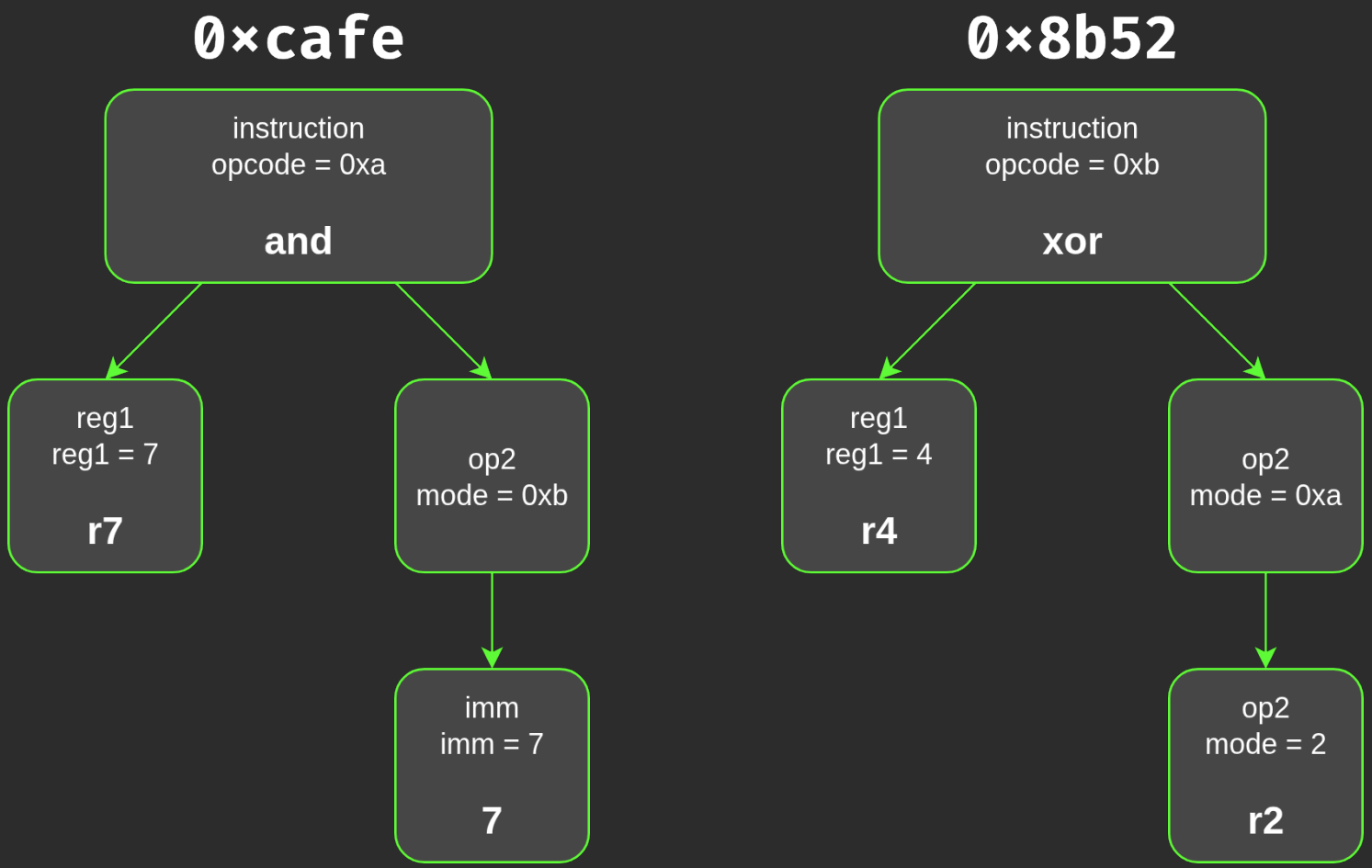
A **table** (this one named `op2`) contains multiple constructors.

When `op2` is used in another constructor, any of its constructors can match — like a big OR.

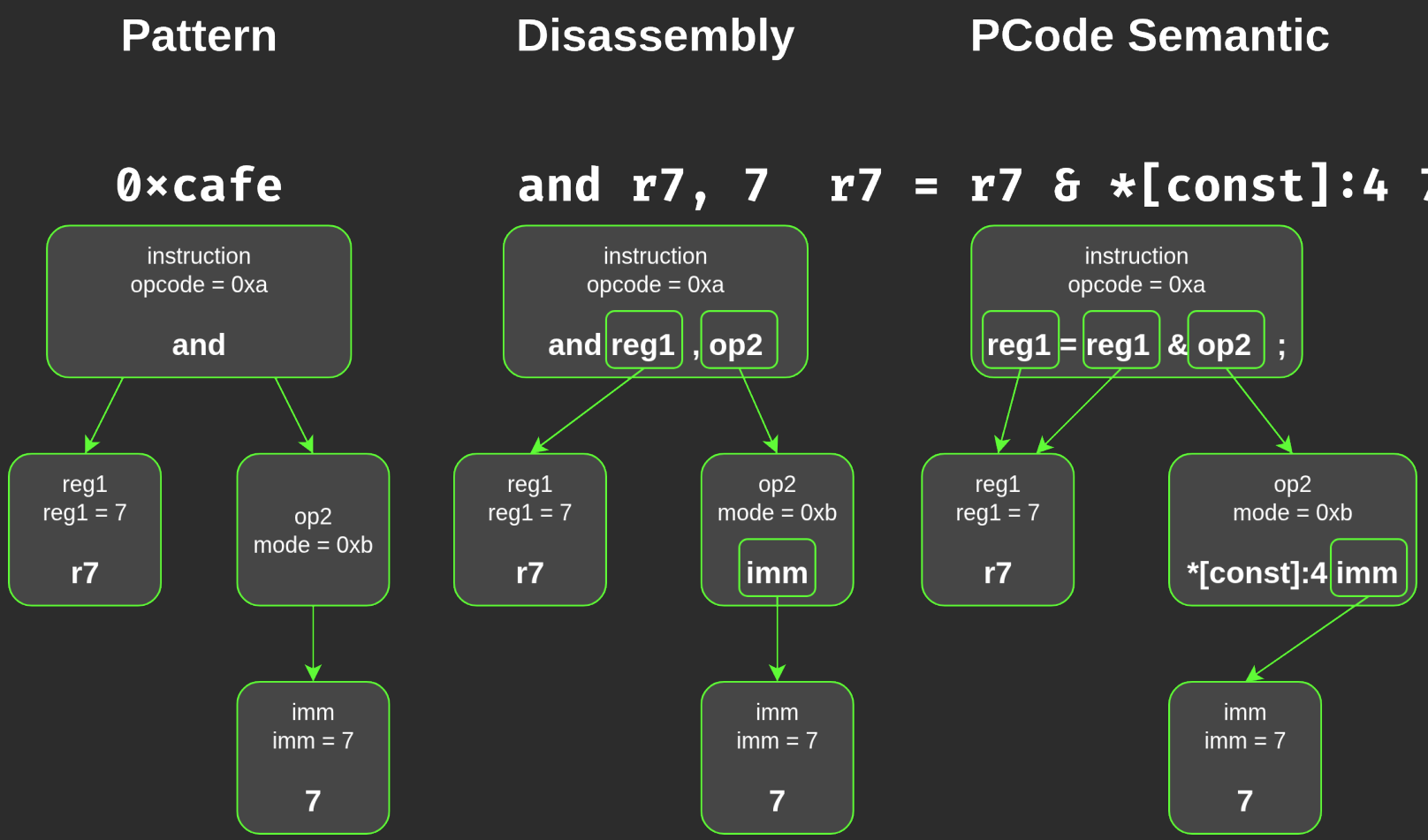
Each constructor **exports** a value that the parent constructor can use as `op2`.

```
define endian=little;
define space ram type=ram_space size=4 default;
define space register type=register_space size=4;
define register offset=0 size=4 [ r0 r1 r2 r3 r4 r5 r6 r7 ];

define token instr(16)
  op=(0,5)
  mode=(6,9)
  reg1=(10,12)
  imm=(13,15)
  reg2=(13,15)
;
attach variables [ reg1 reg2 ] [ r0 r1 r2 r3 r4 r5 r6 r7 ];

op2: reg2 is mode=0xa & reg2   { export reg2; }
op2: imm is mode=0xb & imm     { export *[const]:4 imm; }
op2: [reg2] is mode=0xc & reg2 { tmp = *:4 reg2; export tmp;}

:and reg1,op2 is op=0xa & reg1 & op2 { reg1 = reg1 & op2; }
:xor reg1,op2 is op=0xb & reg1 & op2 { reg1 = reg1 ^ op2; }
:or  reg1,op2 is op=0xc & reg1 & op2 { reg1 = reg1 | op2; }
```

Disas Tree: Example

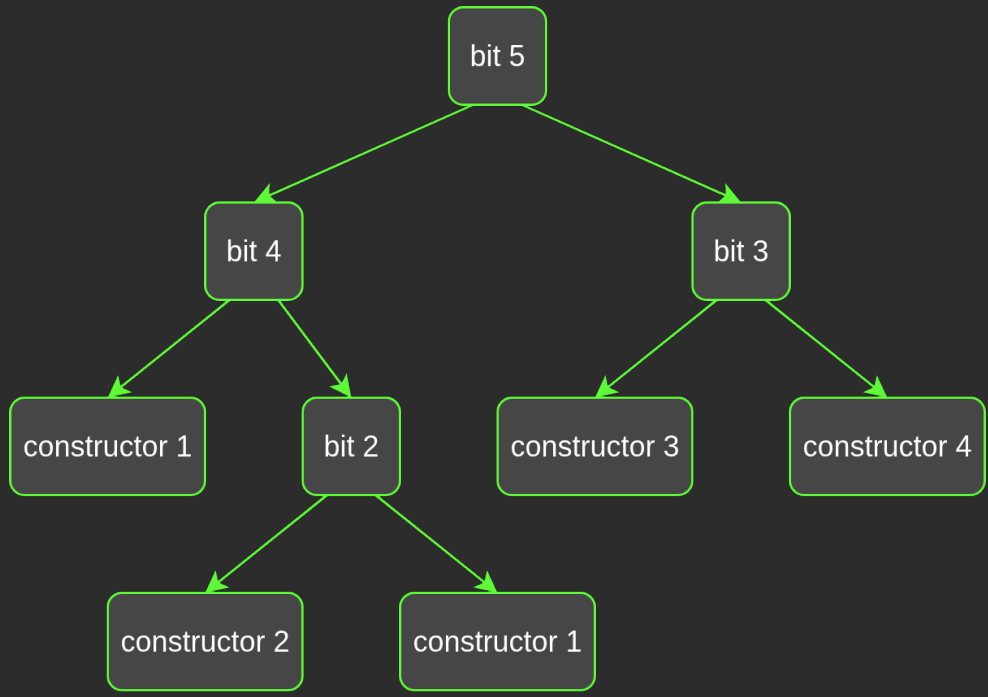


Extracting the disassembly or semantics



Actually matching

- Naive matching explodes in complexity
  - > Brute force means searching an exponential number of patterns
- Instead, we try to build a decision tree
  - > Find a bit or bit pattern that evenly splits possible instructions in the table
    - > This is made more complicated by any-matches
    - > Variable-size instructions make this even harder
- We try to build constraints for tables: what every constructor matches



Limitations of P-code and SLEIGH

Improving the IR

Limitations of raw P-code:

- Control flow is still implicit
  - > PCode mainly reasons with addresses
- Variables may be reassigned many times
  - > This makes data flow analysis difficult
- Constants are bizarre:
  - > `*[const]:4 7` why not `7` ?

I am currently finalizing a new IR, which addresses these issues and is still designed to be lifted into from SLEIGH.

Closing

Takeaway

Three takeaways:

1. IRs simplify binary analysis.
2. Ghidra lifts instructions into PCode, which powers the decompiler and analysis engine.
3. SLEIGH is used to describe an architecture in PCode.

> *“Reverse engineering tools do not reason directly about machine instructions — they reason about **lifted representations**.”*

Maybe I’ll do a part 2…

Thank you

References

1. https://ghidra.re/ghidra_docs/languages/html/sleigh_ref.html
2. https://ghidra.re/ghidra_docs/languages/html/pcoderef.html