Get to Know Your Decompiler

0x41con 2023

@jonpalmisc

About me

@jonpalmisc

- Florida man
- Long-time programmer
- Low-level & security enjoyer
- Present: iOS stuff at <private>
- Past: Binary Ninja developer at Vector 35 for 2 years

History of decompilers

A concise, biased perspective

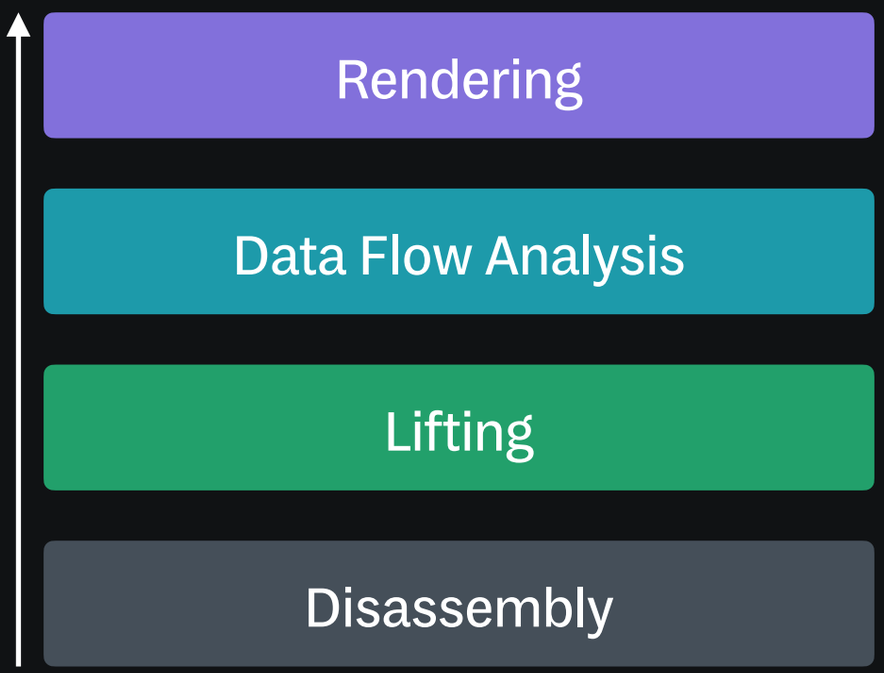
- 1994: “Reverse Compilation Techniques”
- 1995–2006: The Dark Ages
- 2007: Hex-Rays Decompiler
- 2019: Ghidra
- 2020: Binary Ninja HLIL

Decompilation stages

Overview

Decompilation stages

Overview



Decompilation stages

Disassembly

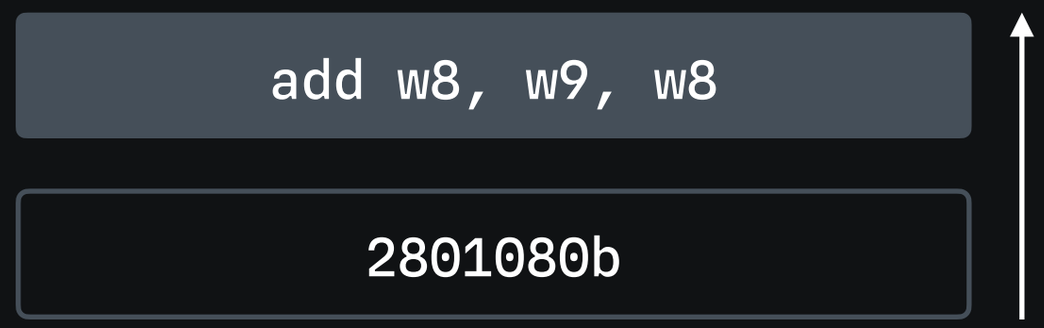
- Natural prerequisite for decompilation
- More than just producing text

```
2801080b
```

Decompilation stages

Disassembly

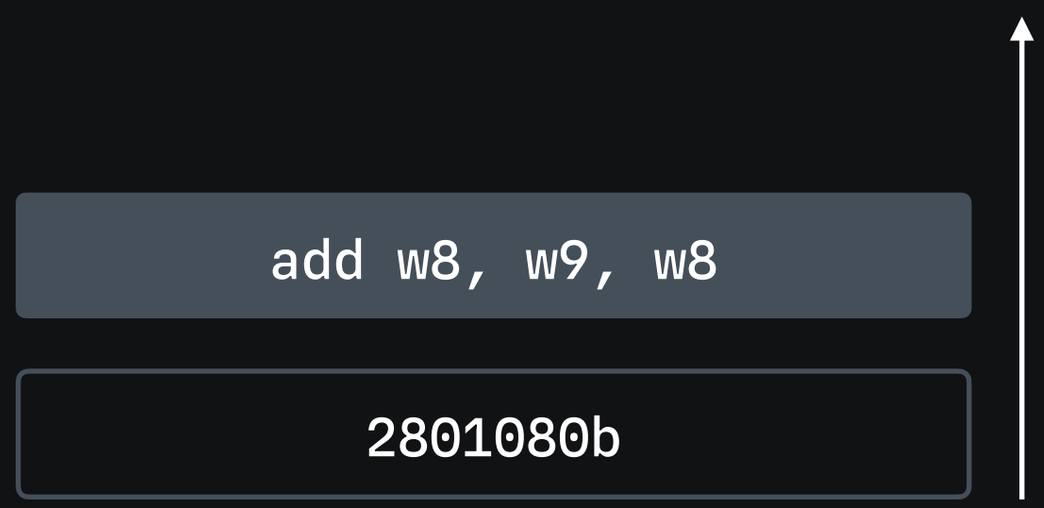
- Natural prerequisite for decompilation
- More than just producing text



Decompilation stages

Lifting

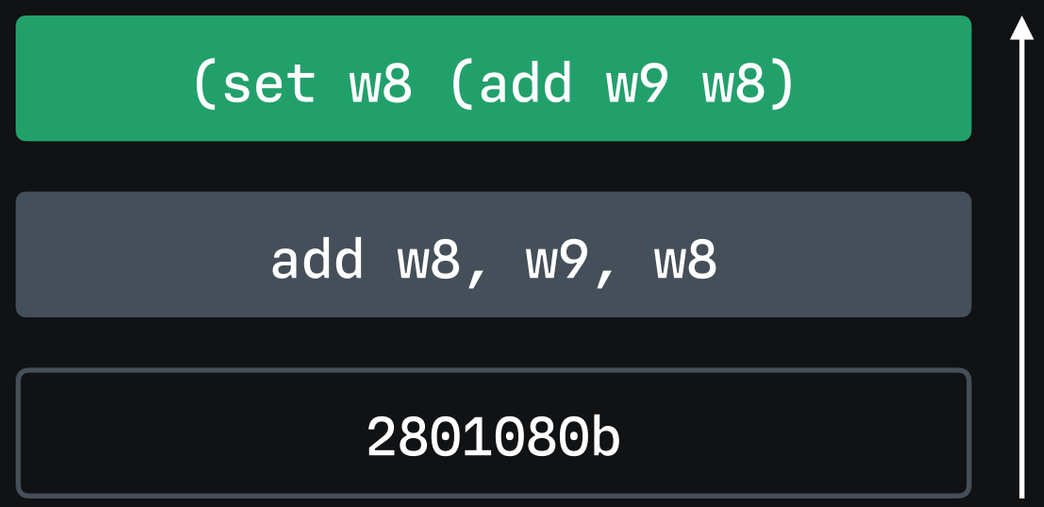
- Start abstracting machine code
- Need some intermediate representation
  - Numerous different approaches exist



Decompilation stages

Lifting

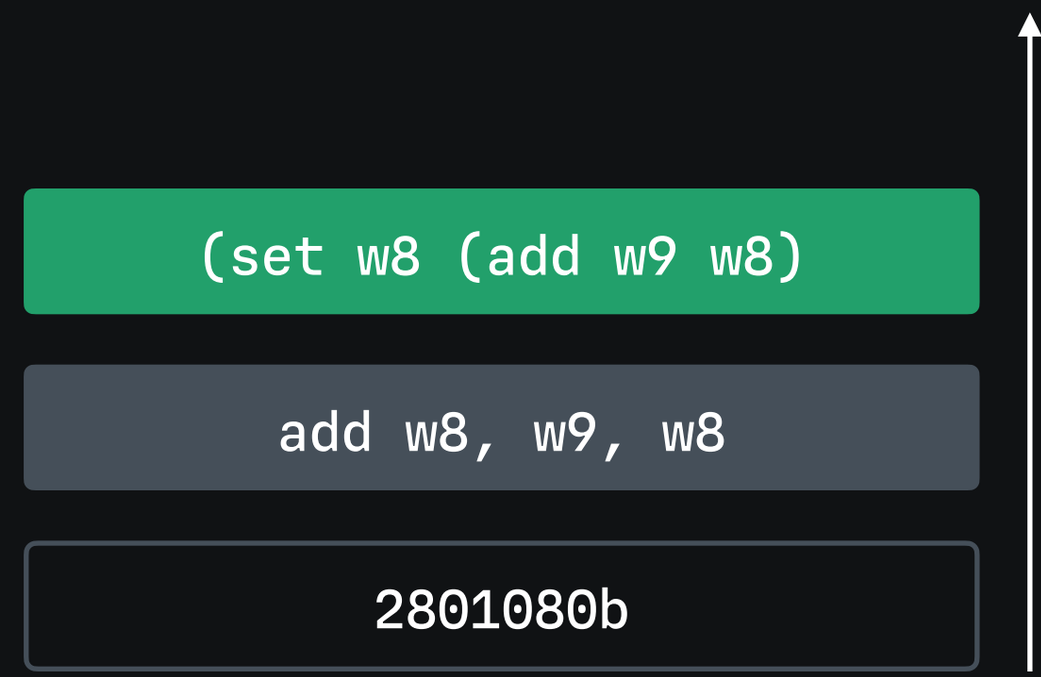
- Start abstracting machine code
- Need some intermediate representation
  - Numerous different approaches exist



Decompilation stages

Data flow analysis

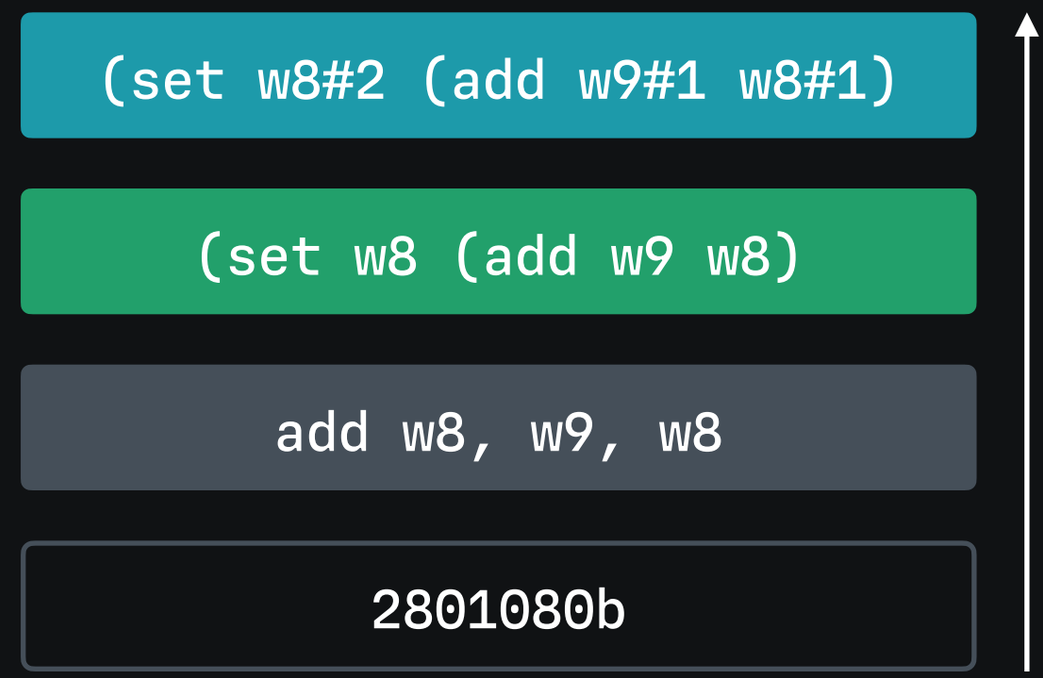
- Where are values defined and used?
- What are the possible values of X at address Y?
- Inherent part of decompilation internally



Decompilation stages

Data flow analysis

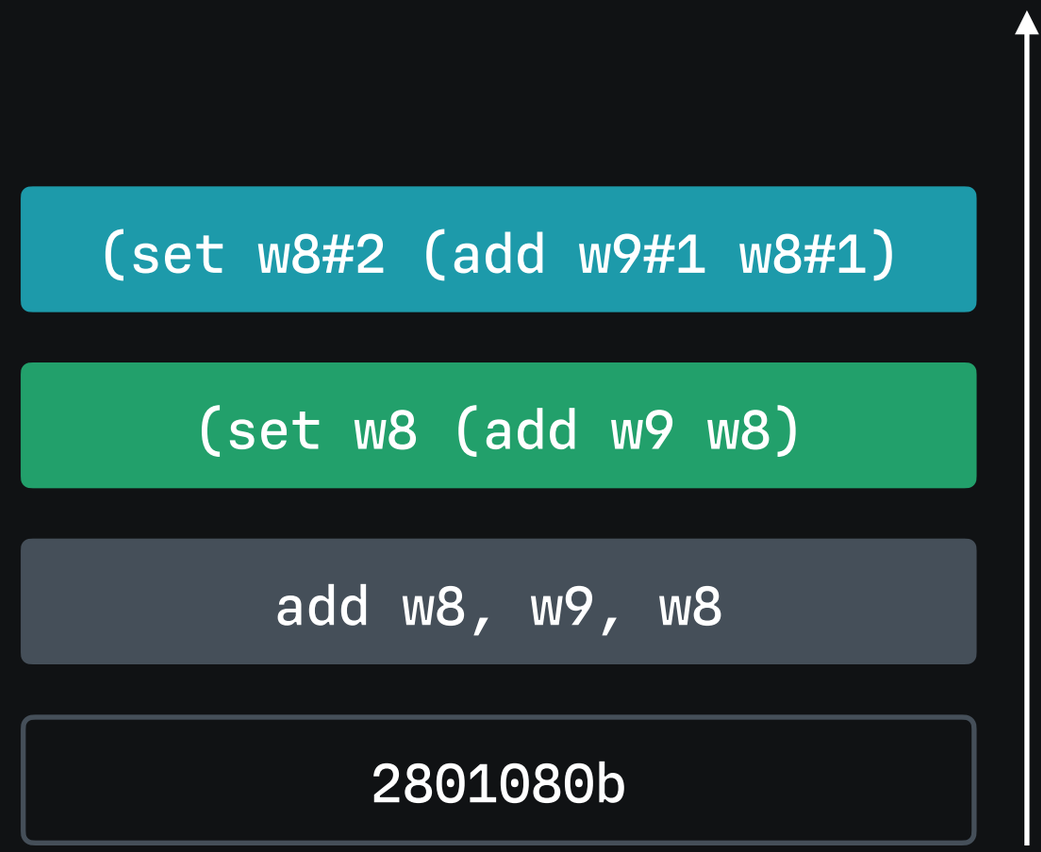
- Where are values defined and used?
- What are the possible values of X at address Y?
- Inherent part of decompilation internally



Decompilation stages

Rendering

- Need to represent the abstracted code somehow
  - C-flavored syntax has become the norm
- Not “trivial”, but less difficult



Decompilation stages

Rendering

- Need to represent the abstracted code somehow
  - C-flavored syntax has become the norm
- Not “trivial”, but less difficult

IDA’s microcode

Overview

- Machine code transformed into “microinstructions”
  - 72 unique operations
  - Single-purpose, simple in nature
  - No side effects
- Inspired by ideas from “Reverse Compilation Techniques”

IDA’s microcode

Microinstruction anatomy

Microinstruction

```
add x0.8, t2.8, t2.8
```

IDA’s microcode

Microinstruction anatomy

Microinstruction

```
add x0.8, t2.8, t2.8
```

Operation

IDA’s microcode

A basic example

AArch64 assembly

```
ldr x0, [x0, #8]
```

```
ret
```

IDA’s microcode

A basic example

AArch64 assembly

```
ldr x0, [x0, #8]
```

IDA’s microcode

A basic example

AArch64 assembly

```
ldr x0, [x0, #8]
```

Generated microcode

```
mov #8.8, t2.8
```

IDA’s microcode

A basic example

AArch64 assembly

```
ldr x0, [x0, #8]
```

Generated microcode

```
mov #8.8, t2.8
```

```
add x0.8, t2.8, t2.8
```

IDA’s microcode

A basic example

AArch64 assembly

```
ldr x0, [x0, #8]
```

Generated microcode

```
mov #8.8, t2.8
```

```
add x0.8, t2.8, t2.8
```

```
ldx t2.8, t1.8
```

IDA’s microcode

A basic example

AArch64 assembly

```
ldr x0, [x0, #8]
```

Generated microcode

```
mov #8.8, t2.8
add x0.8, t2.8, t2.8
ldx t2.8, t1.8
mov t1.8, x0.8
```

IDA’s microcode

Microcode lifecycle

- Unoptimized, initial lift (Level 1)
- Easy wins (Level 2)
- Local optimizations, function calls (Levels 3, 4)
- Global optimizations (Levels 5–7)
- Local variable analysis (Level 8)

IDA’s microcode

Optimizations visualized

AArch64 assembly

```
ldr x0, [x0, #8]
```

Generated microcode

```
mov #8.8, t2.8
add x0.8, t2.8, t2.8
ldx t2.8, t1.8
mov t1.8, x0.8
```

IDA’s microcode

Optimizations visualized

AArch64 assembly

```
ldr x0, [x0, #8]
```

```
ldx (x0.8 + #8.8), t1.8
```

```
mov t1.8, x0.8
```

IDA’s microcode

Optimizations visualized

AArch64 assembly

```
ldr x0, [x0, #8]
```

Optimized microcode

```
ldx (x0.8 + #8.8), x0.8
```

IDA’s microcode

Optimizations visualized

AArch64 assembly

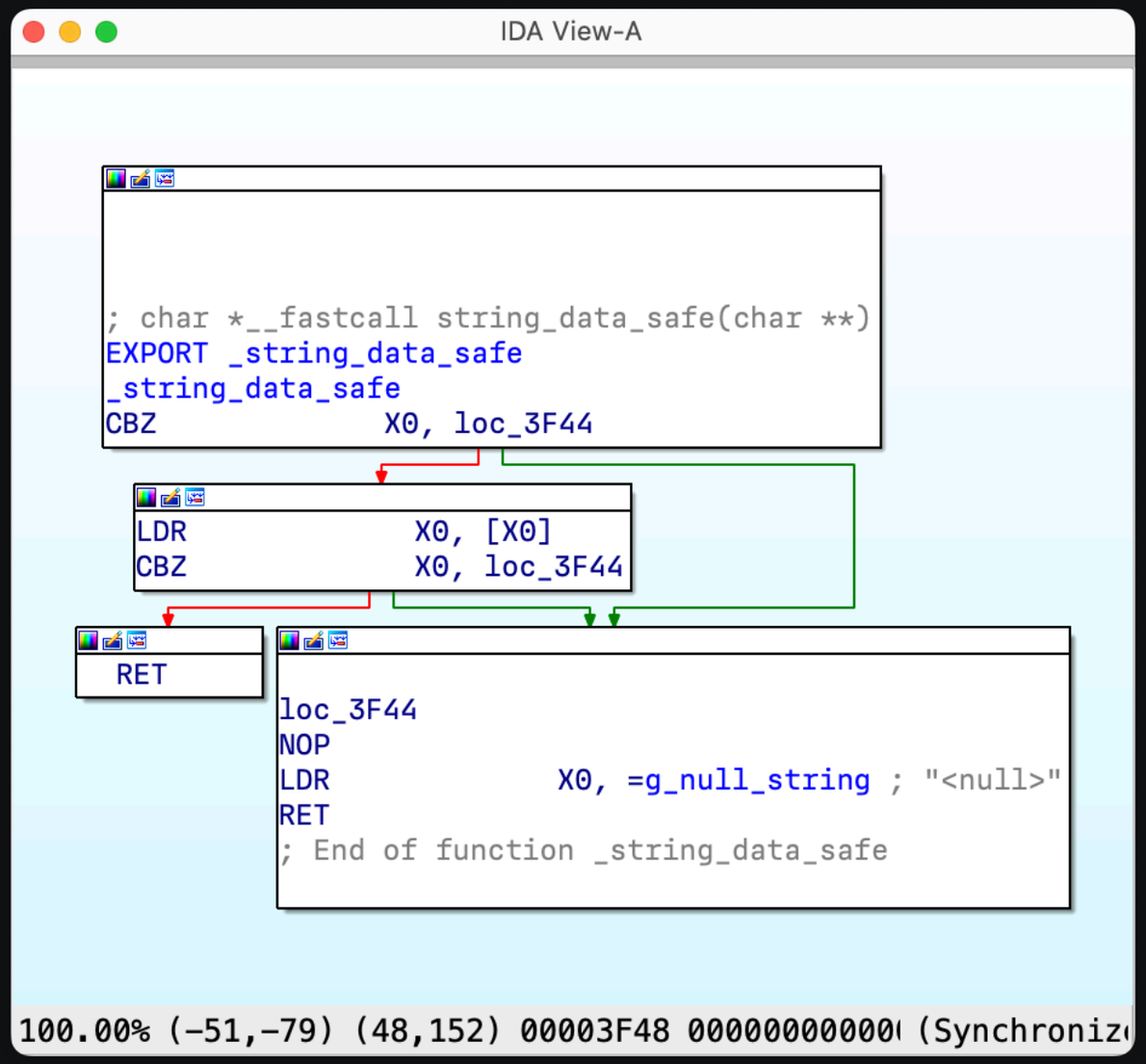
```
ldr x0, [x0, #8]
```

Optimized microcode with local variables

```
ldx (a1.8 + #8.8), result.8
```

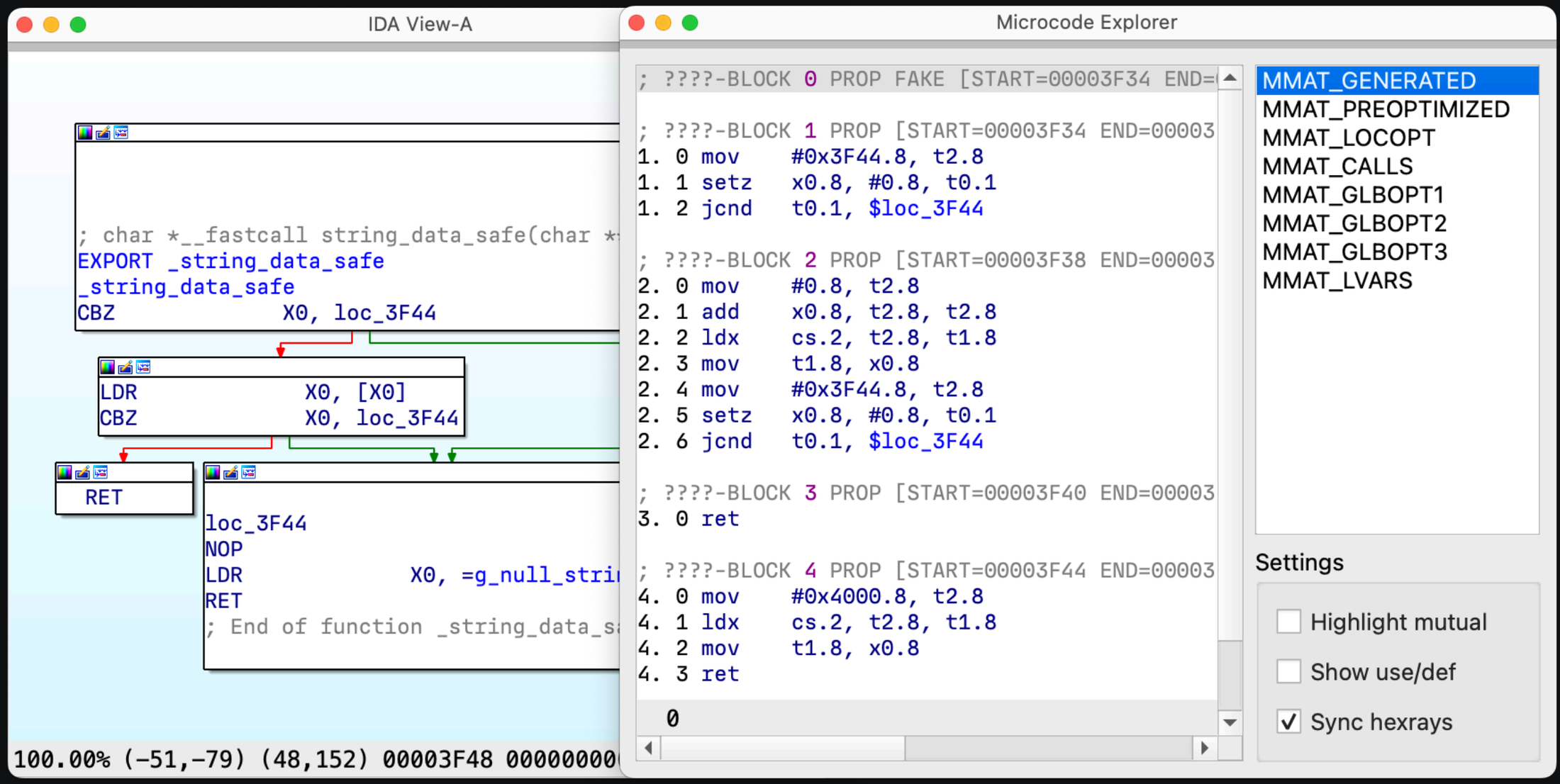
IDA’s microcode

A real-world example



IDA's microcode

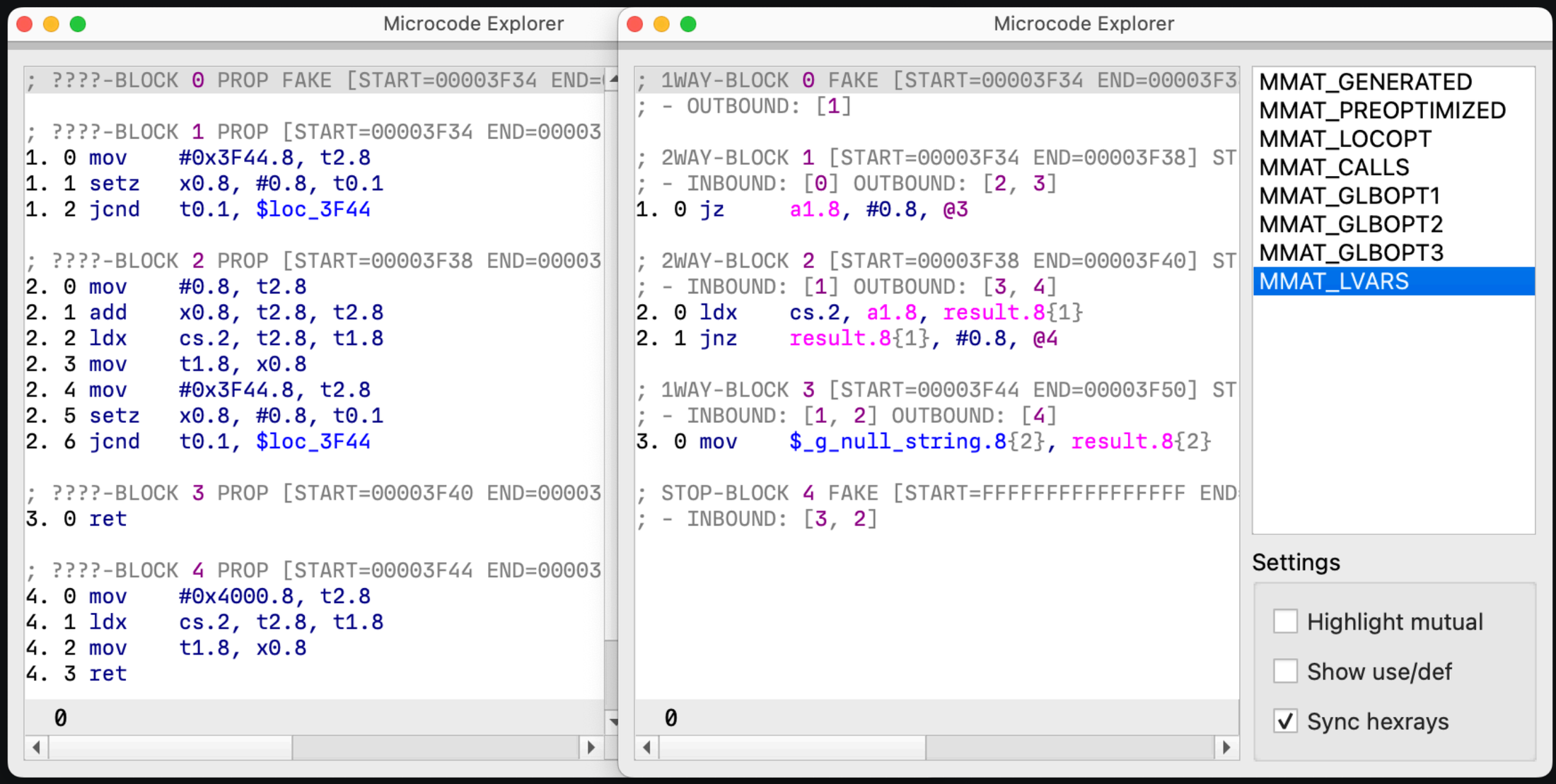
A real-world example



Microcode shown with the "Lucid" plugin by Markus Gassedelen. (https://github.com/gaasedelen/lucid)

IDA's microcode

A real-world example



Microcode shown with the "Lucid" plugin by Markus Gassedelen. (https://github.com/gaasedelen/lucid)

Ghidra’s p-code

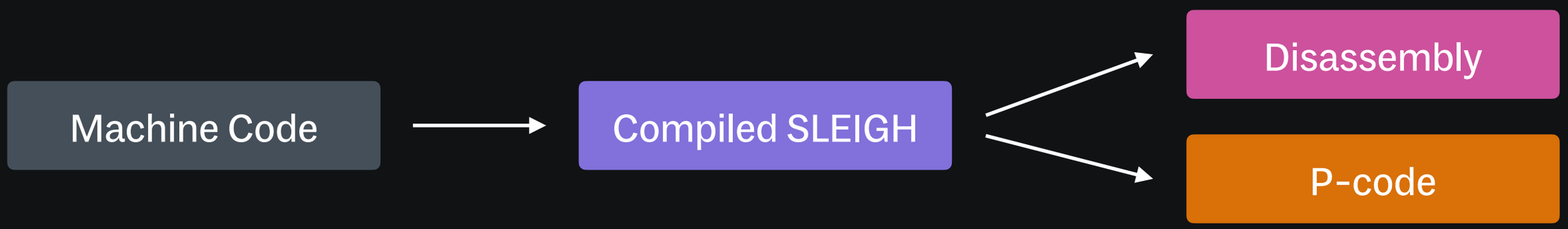
Overview

- Self-described register transfer language
- Similar to IDA’s microcode
- 62 unique operations
- Mostly free of side-effects
- etc...

Ghidra’s p-code

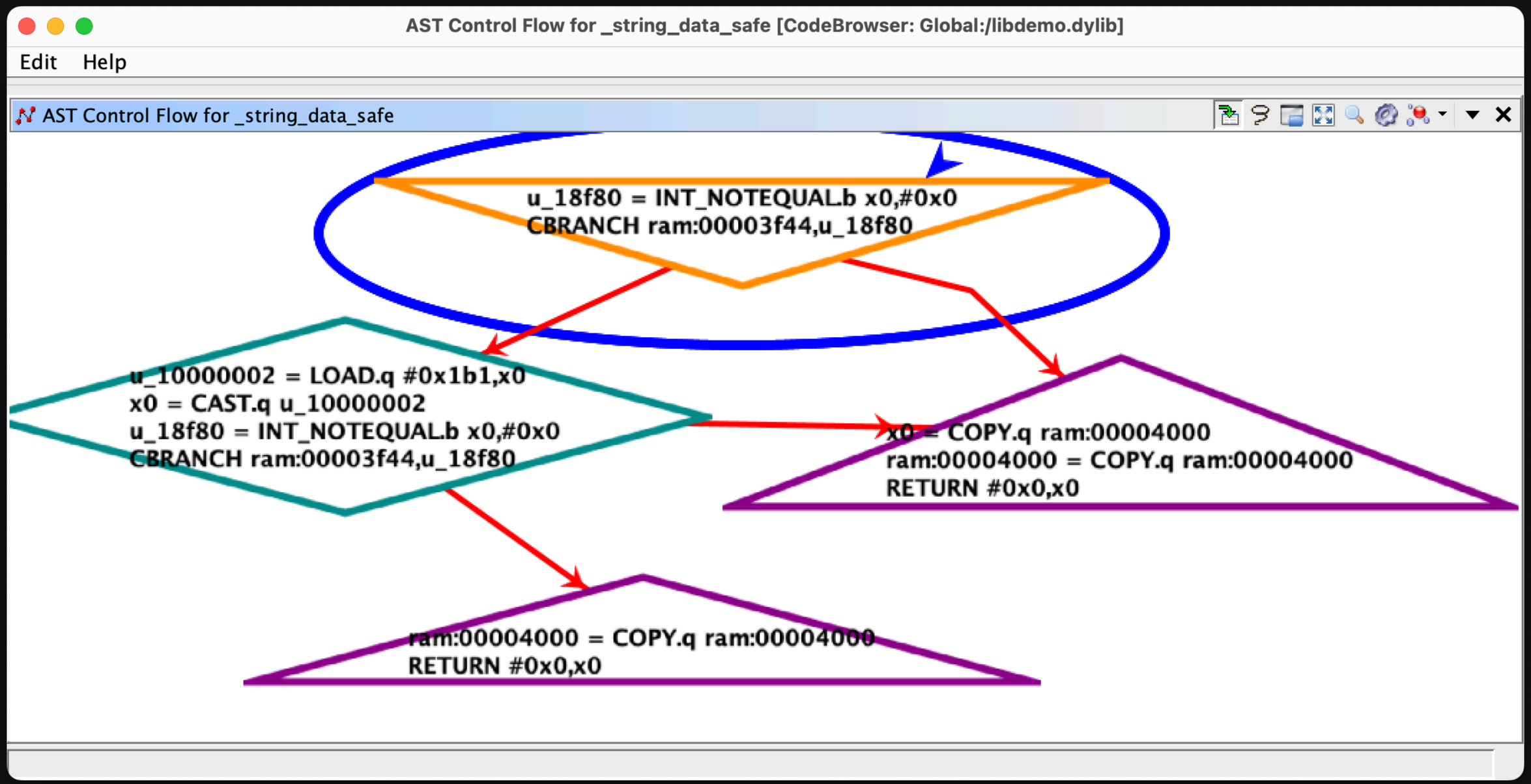
SLEIGH

- Expressive language for describing ISAs
  - Does both disassembly and lifting
- Inspired by SLED (AT&T Labs, 1997)
- Lots of moving parts



Ghidra's p-code

A real-world example



Binary Ninja’s ILs

Overview

- Stack of three intermediate languages (ILs)
  - Low-level IL (LLIL)
  - Medium-level IL (MLIL)
  - High-level IL (HLIL)
- Tree/expression-based

Binary Ninja’s ILs

Low-level IL

- 108 unique operations
- Most primitive user-facing IL, lifted from machine code
    - Internally, “Lifted IL” technically comes first
- No concept of memory
- Resolved flags & stack pointer

Binary Ninja's ILs

Medium-level IL

- Much more expressive than LLIL
  - Introduces variables, types, calls/parameters
- Acts as the medium for lots of important analysis
  - Type propagation
  - Data flow & value set analysis
  - Other important things

Binary Ninja’s ILs

High-level IL

- Looks closer to C/pseudocode
- Refined through numerous simplification passes
- Introduces semantic control flow
- Adds variable merging
- Dead code elimination
- Etc...

Binary Ninja’s ILs

Anatomy of an IL instruction

- Text form
- Tree form
  - Operation
  - Variation of operands
    - Source & destination
    - Left & right
    - Etc...

Binary Ninja’s ILs

A basic example

AArch64 assembly

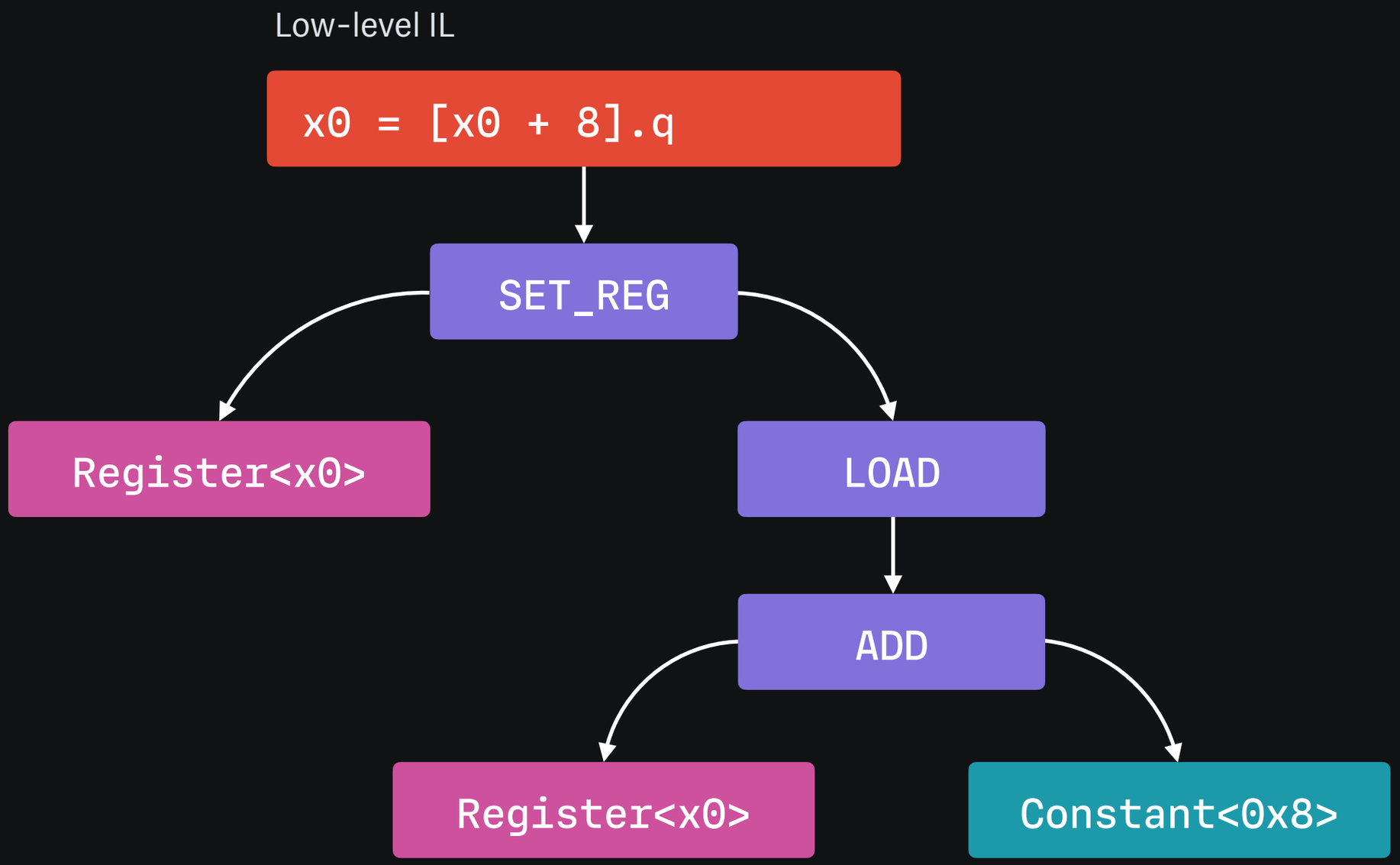
```
ldr x0, [x0, #8]
```

Low-level IL (text-form)

```
x0 = [x0 + 8].q
```

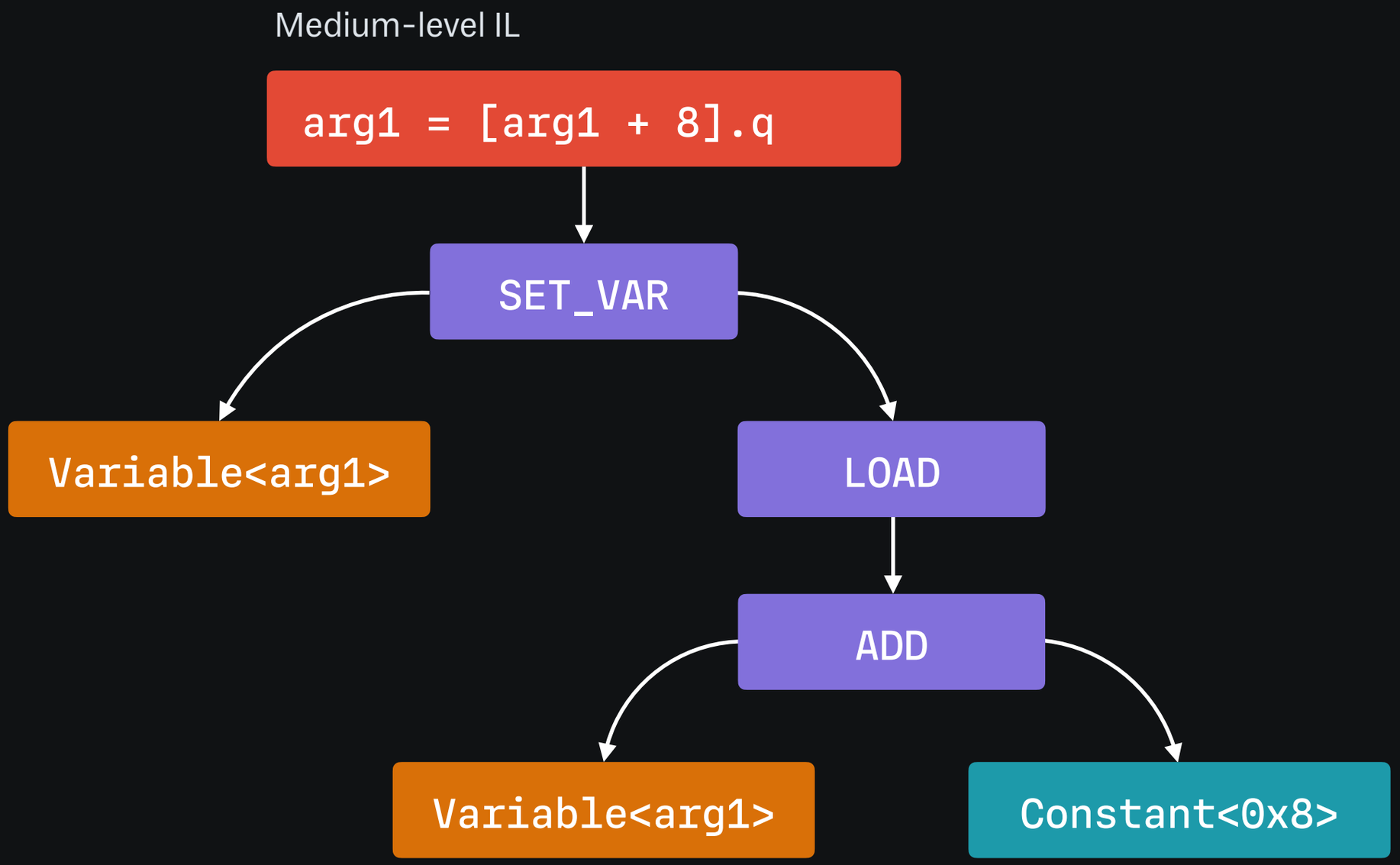
Binary Ninja's ILs

A basic example



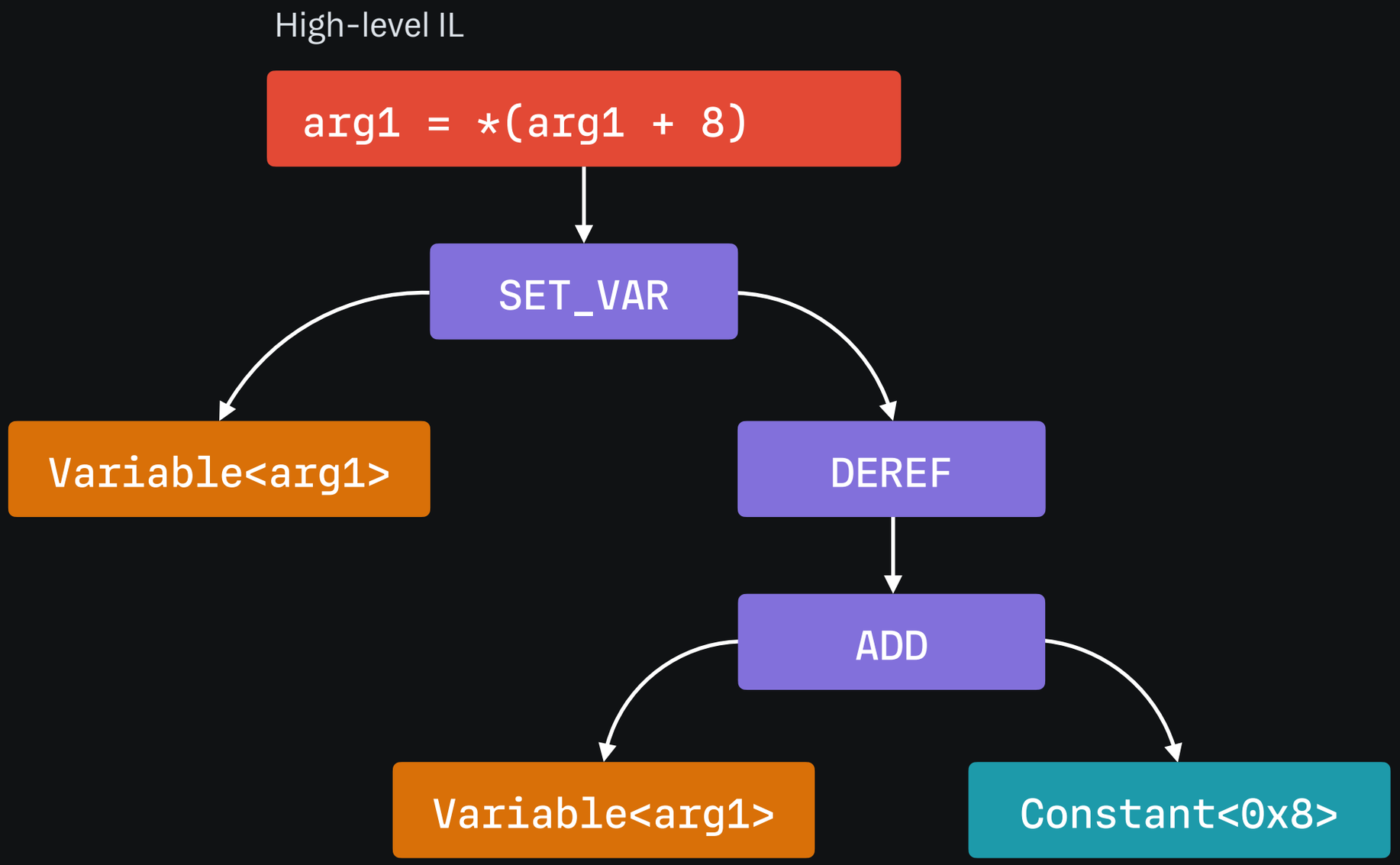
Binary Ninja's ILs

A basic example



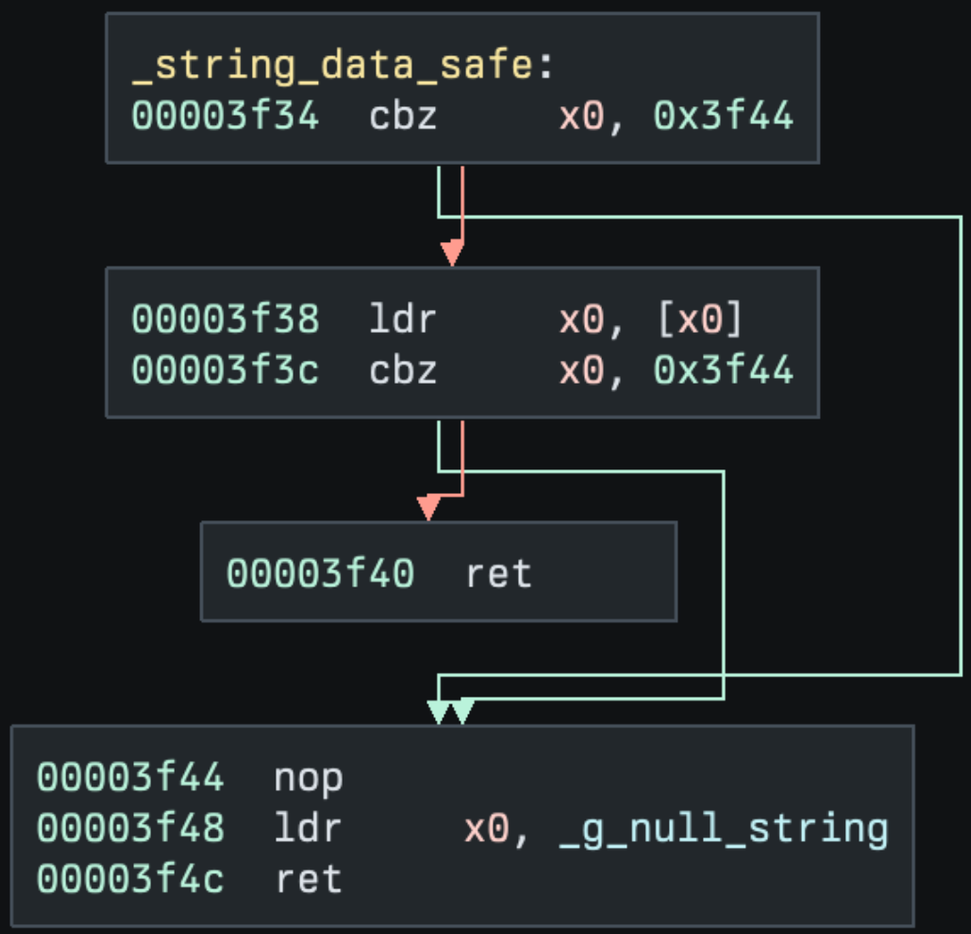
Binary Ninja's ILs

A basic example



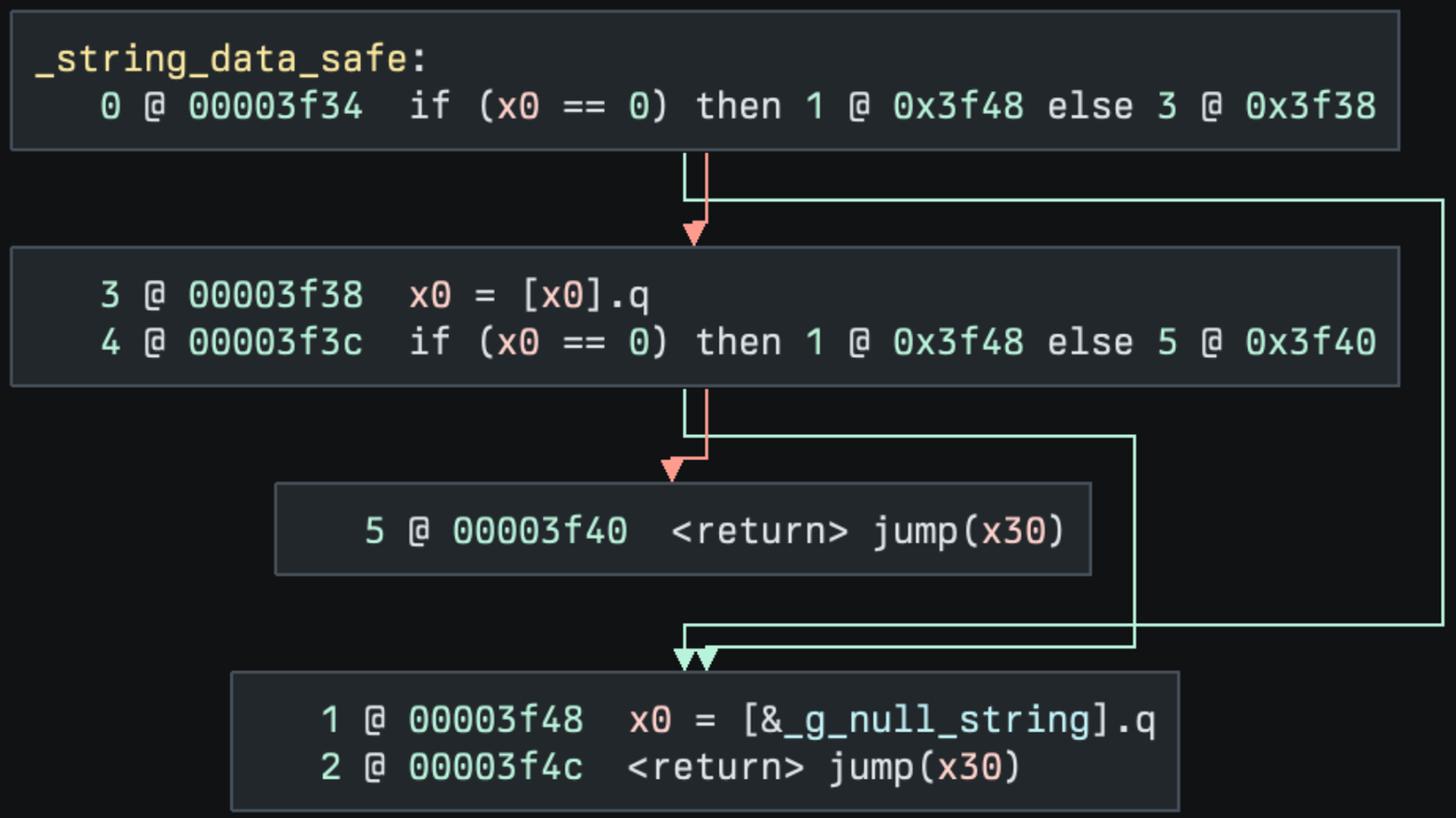
Binary Ninja's ILs

A real-world example: disassembly



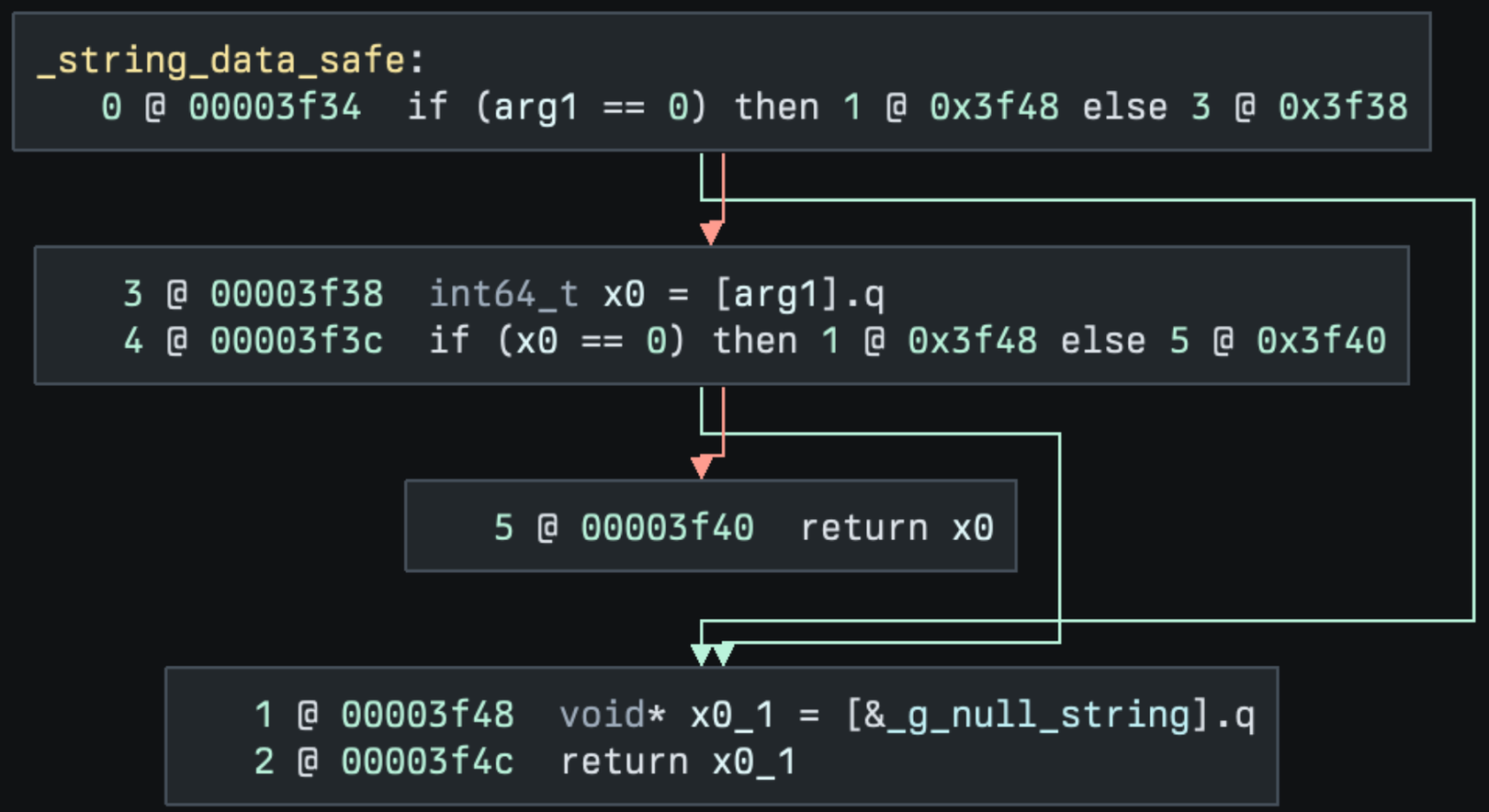
Binary Ninja's ILs

A real-world example: LLIL



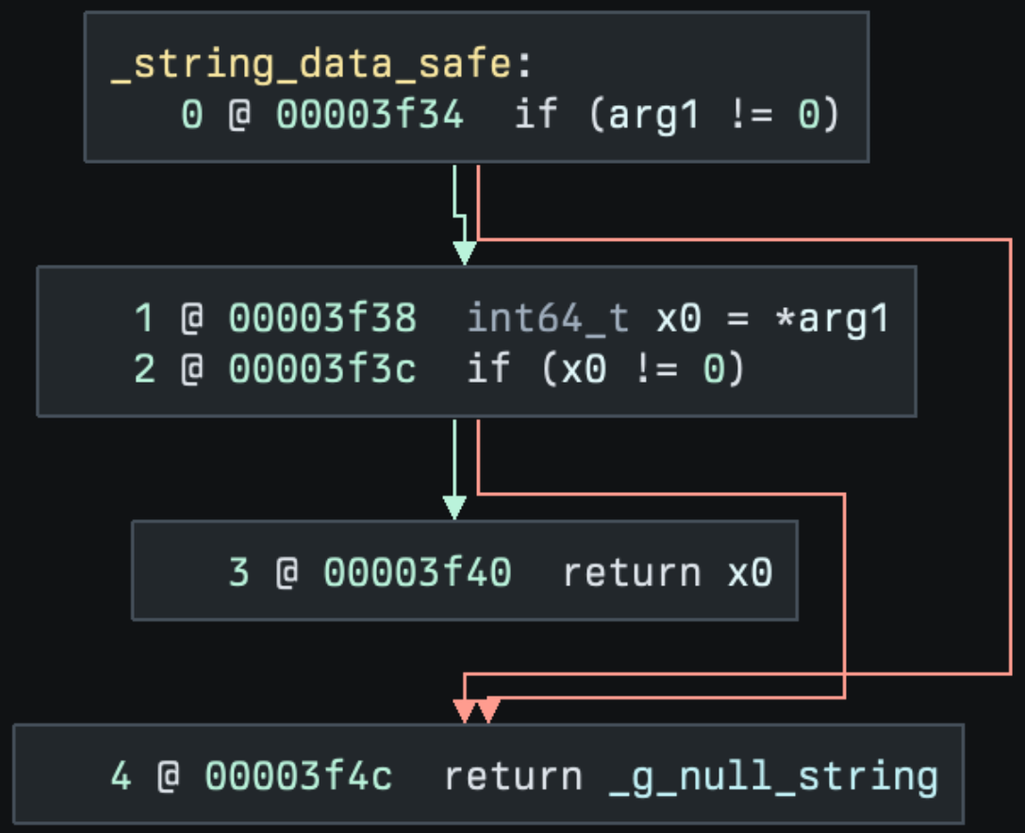
Binary Ninja's ILs

A real-world example: MLIL



Binary Ninja's ILs

A real-world example: HLIL



Aside

Immediate vs. deferred flags

- Immediate flags
  - IDA
  - Ghidra
- Deferred flags
  - Binary Ninja

Aside

What about LLVM IR..?

- Has some apparent advantages
  - Popularity, infrastructure, maturity, etc.
- Harder to lift to straight from assembly
- Missing important decompilation-oriented semantics

Data flow analysis

Classic techniques (IDA)

- Tried-and-true compiler techniques
  - Value set analysis
  - Use-def chains
  - (Global) copy propagation
  - Etc...

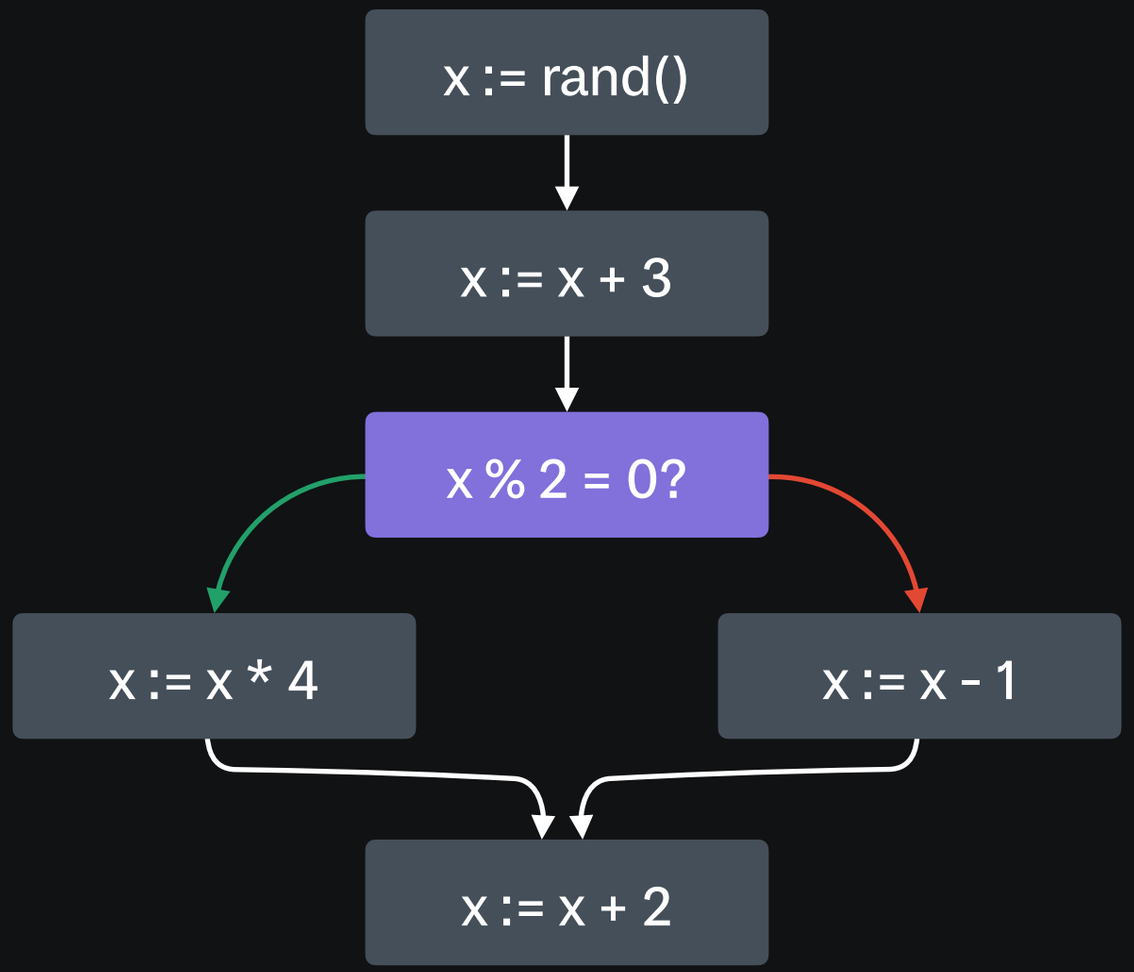
Data flow analysis

Static single assignment form (Binary Ninja & Ghidra)

- Every variable has exactly one definition
    - Subsequent assignments labeled with subscripts
        - e.g. x_2
- Joining of two live variables denoted with Phi (Φ)
    - e.g. $\Phi(x_2, x_3)$
- It’s easier to understand visually...

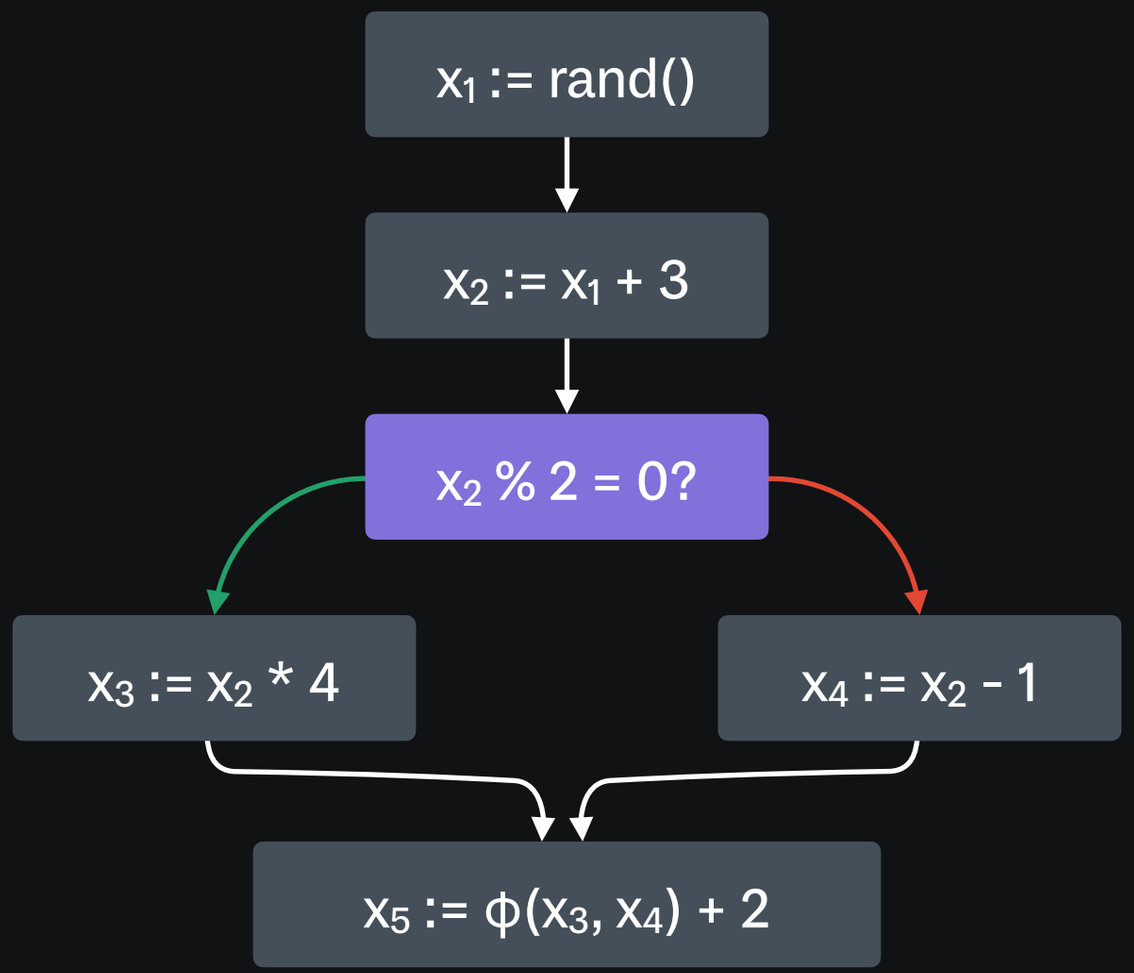
Data flow analysis

Non-SSA form example



Data flow analysis

SSA form example



Data flow analysis

Real-world SSA form example

```
_example_ssa_small:
00003f10  sub     sp, sp, #0x20
00003f14  stp     x29, x30, [sp, #0x10] {__saved_x29} {__saved_x30}
00003f18  add     x29, sp, #0x10 {__saved_x29}
00003f1c  bl      _rand
00003f20  stur    w0, [x29, #-0x4 {var_14}]
00003f24  ldur    w8, [x29, #-0x4 {var_14}]
00003f28  add     w8, w8, #0x3
00003f2c  stur    w8, [x29, #-0x4 {var_14_1}]
00003f30  ldur    w9, [x29, #-0x4 {var_14_1}]
00003f34  ldur    w8, [x29, #-0x4 {var_14_1}]
00003f38  tbnz    w9, #0, 0x3f44
```

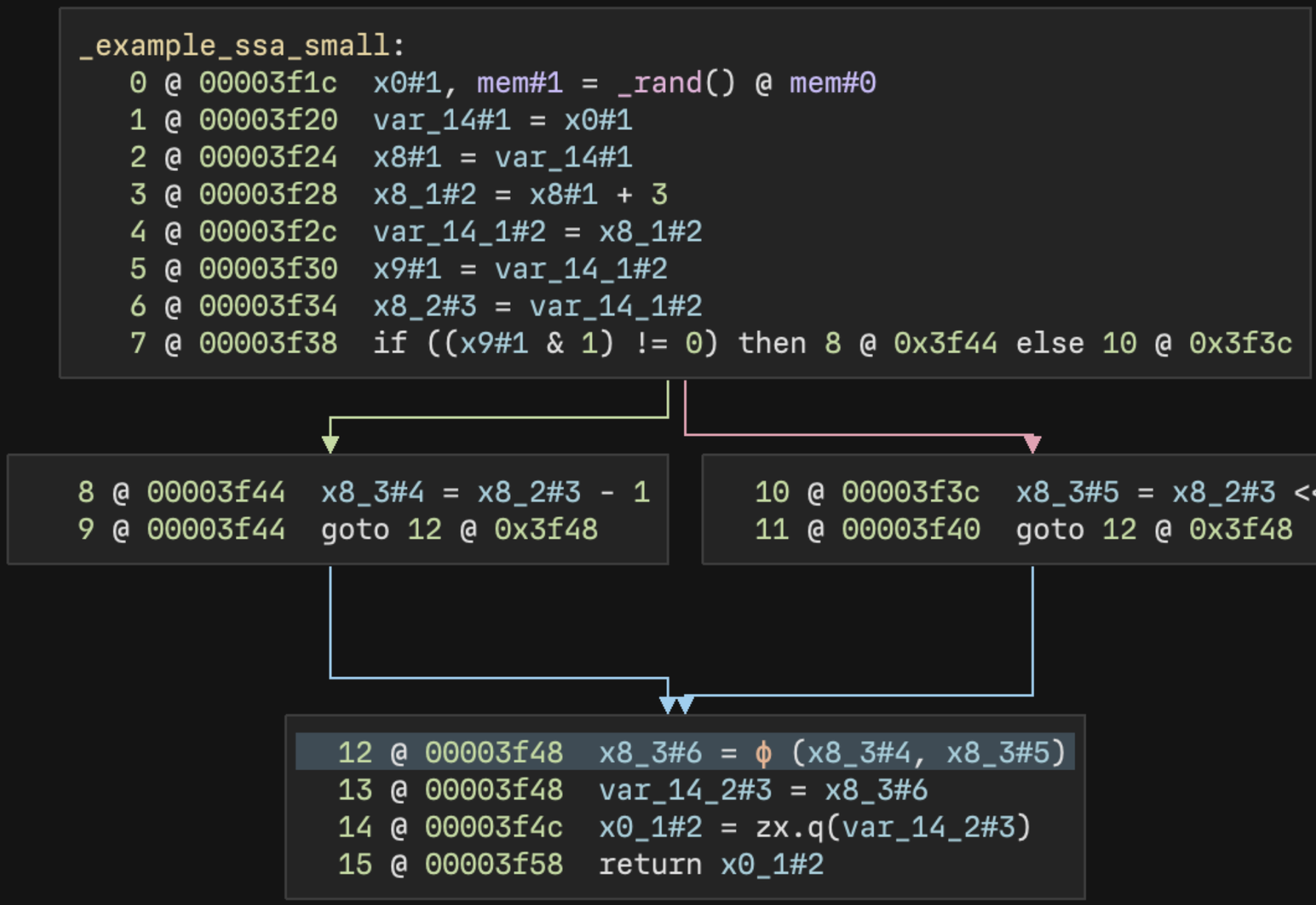
```
00003f44  sub     w8, w8, #0x1
```

```
00003f3c  lsl     w8, w8, #0x2
00003f40  b       0x3f48
```

```
00003f48  stur    w8, [x29, #-0x4 {var_14_2}]
00003f4c  ldur    w0, [x29, #-0x4 {var_14_2}]
00003f50  ldp     x29, x30, [sp, #0x10] {__saved_x29} {__saved_x30}
00003f54  add     sp, sp, #0x20
00003f58  ret
```

Data flow analysis

Real-world SSA form example



Closing thoughts

Modern decompilation challenges

- Unrepresentable instructions
- Inlining, outlining
- Pointer & length strings
- Pointer misconceptions
- Weird calling conventions
- Much more...

Closing thoughts

Modern language construct support

- Modern code generation is increasingly intricate
  - Swift
  - Rust
  - Firebloom (-fbounds-safety)
- Need to filter signal from noise

Closing thoughts

Decompile as X?

- Not everything is written in C anymore
  - Why do we try to decompile everything to C?
  - Should we still decompile everything as C?

Closing thoughts

Diverging goals in decompilation

- Literal decompilation?
- Semantic decompilation?
- Answer: Abstraction/optimization problem

Closing thoughts

Modularity & tuning

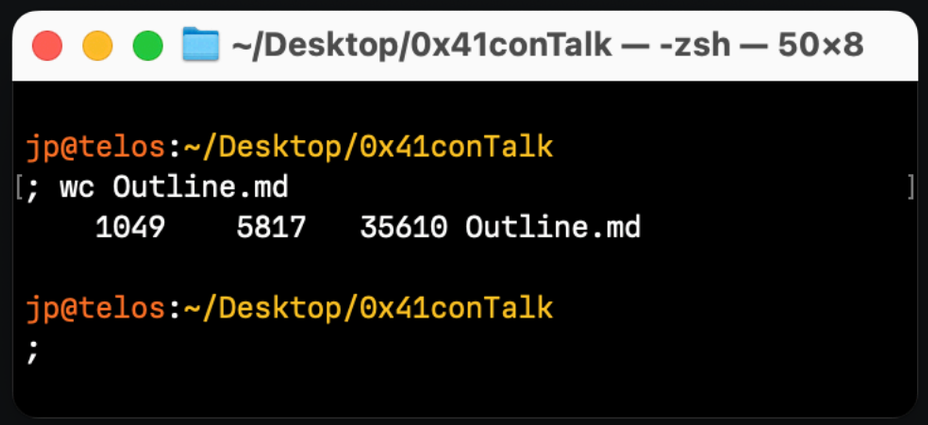
- Soon-to-be necessity
- “One size fits all” decompilation is aging poorly
- Need to be lean, adaptable
- What will the next ten years hold?

Demo time?

a.k.a “how long did those slides take?”

Thank you!

- Questions?
- Find me after if you want to chat more
  - Not everything made it into the slides :)



- Twitter: @jonpalmisc
- Mastodon: @jonpalmisc@infosec.exchange