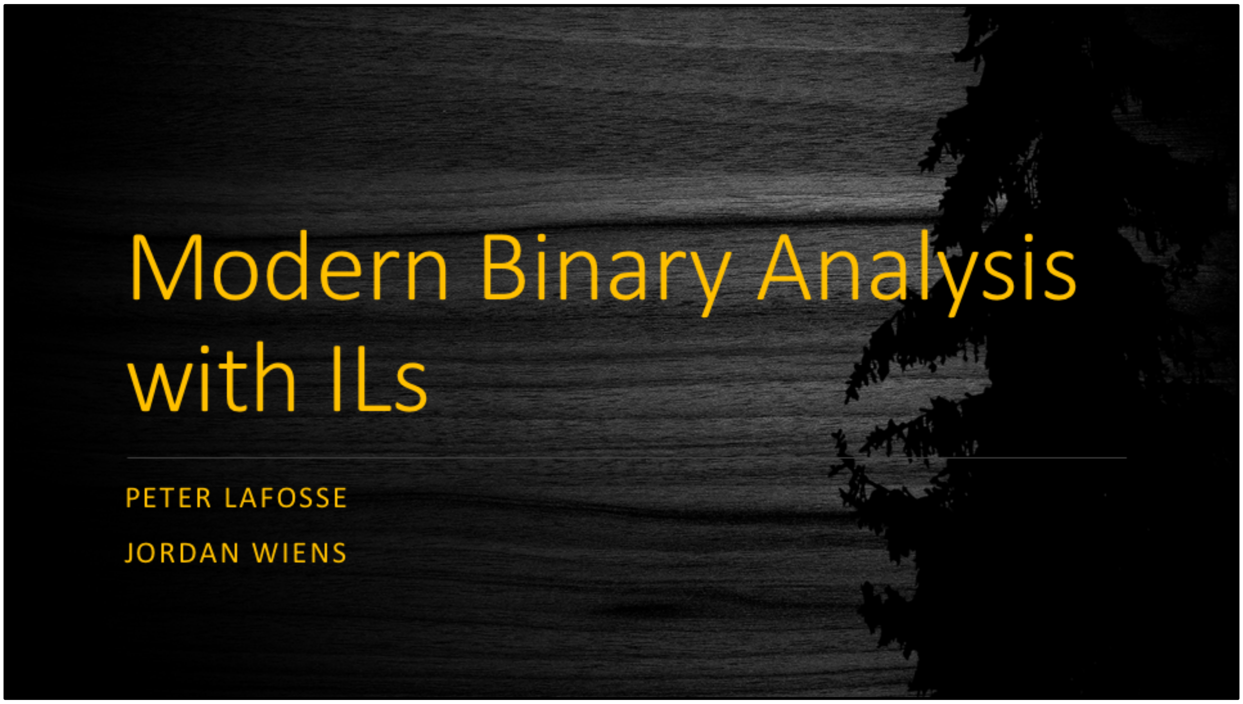
BLUEHAT

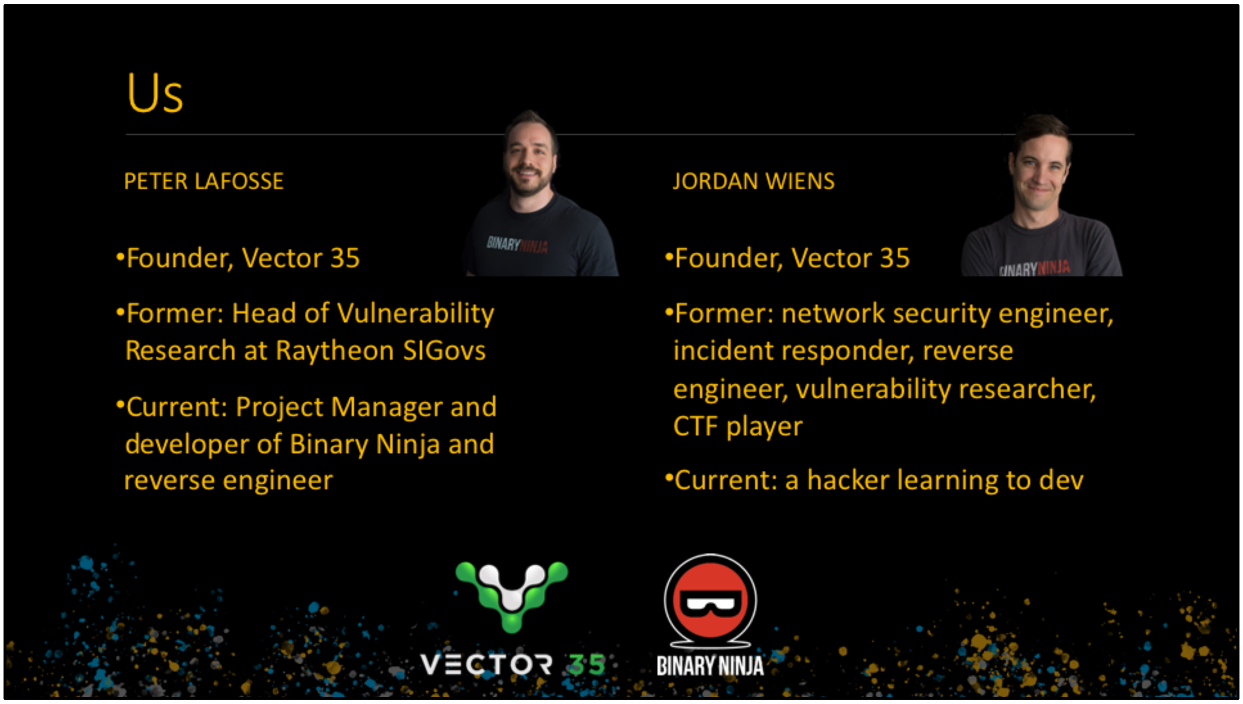
SEATTLE 2019



Modern Binary Analysis with ILs

PETER LAFOSSE

JORDAN WIENS



Us

PETER LAFOSSE

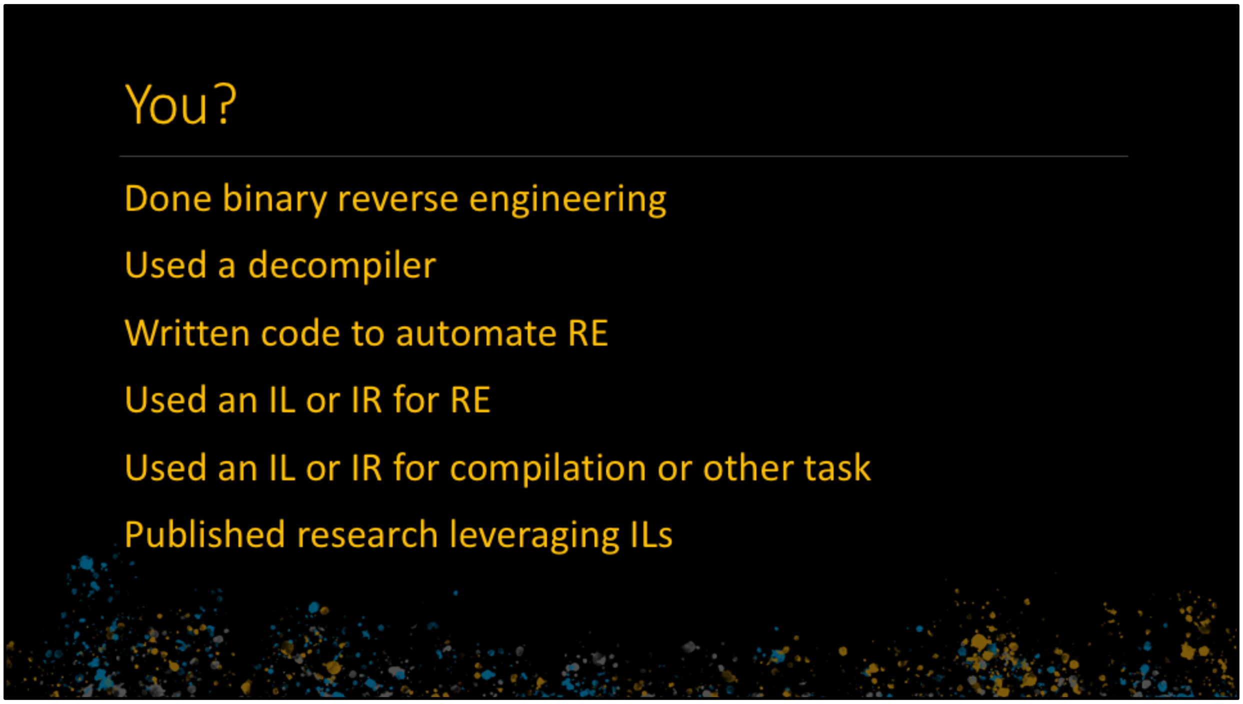
- Founder, Vector 35
- Former: Head of Vulnerability Research at Raytheon SIGovs
- Current: Project Manager and developer of Binary Ninja and reverse engineer

JORDAN WIENS

- Founder, Vector 35
- Former: network security engineer, incident responder, reverse engineer, vulnerability researcher, CTF player
- Current: a hacker learning to dev

VECTOR 35
BINARY NINJA

Clarify: We are NOT academic researchers who have studied intermediate language and compiler design. This is not meant as a slight by any stretch – we are merely acknowledging our own bias as we come at this from the perspective of practioners who try to learn from research what we can but realize we don't know everything coming out of the research community.



You?

Done binary reverse engineering

Used a decompiler

Written code to automate RE

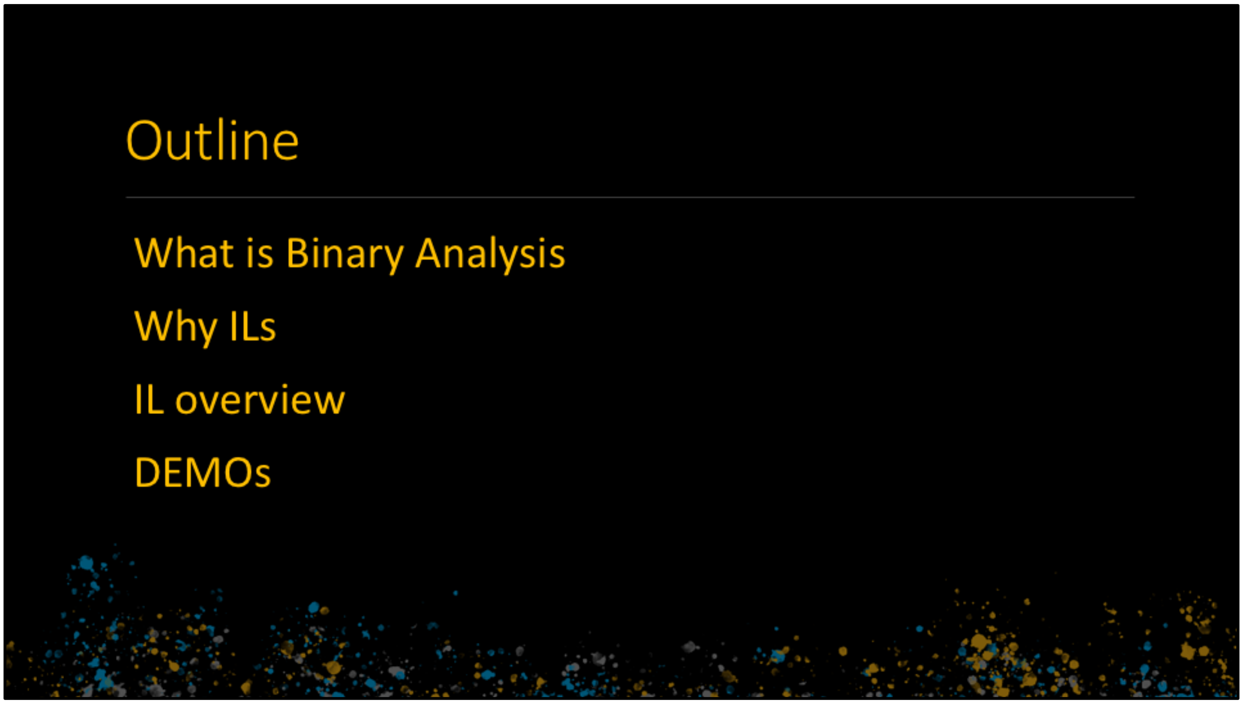
Used an IL or IR for RE

Used an IL or IR for compilation or other task

Published research leveraging ILs

To better help us understand our audience, we'd love to get a feel for the room so we can know how much time to spend on each section. It also makes sure everyone is awake since I know we're almost done with the conference, you just need to stay alert for a few more hours!

So first, everyone in the audience put your hand up to make sure you're awake.
Next, keep your hands up if you have done binary reverse engineering
…



Outline

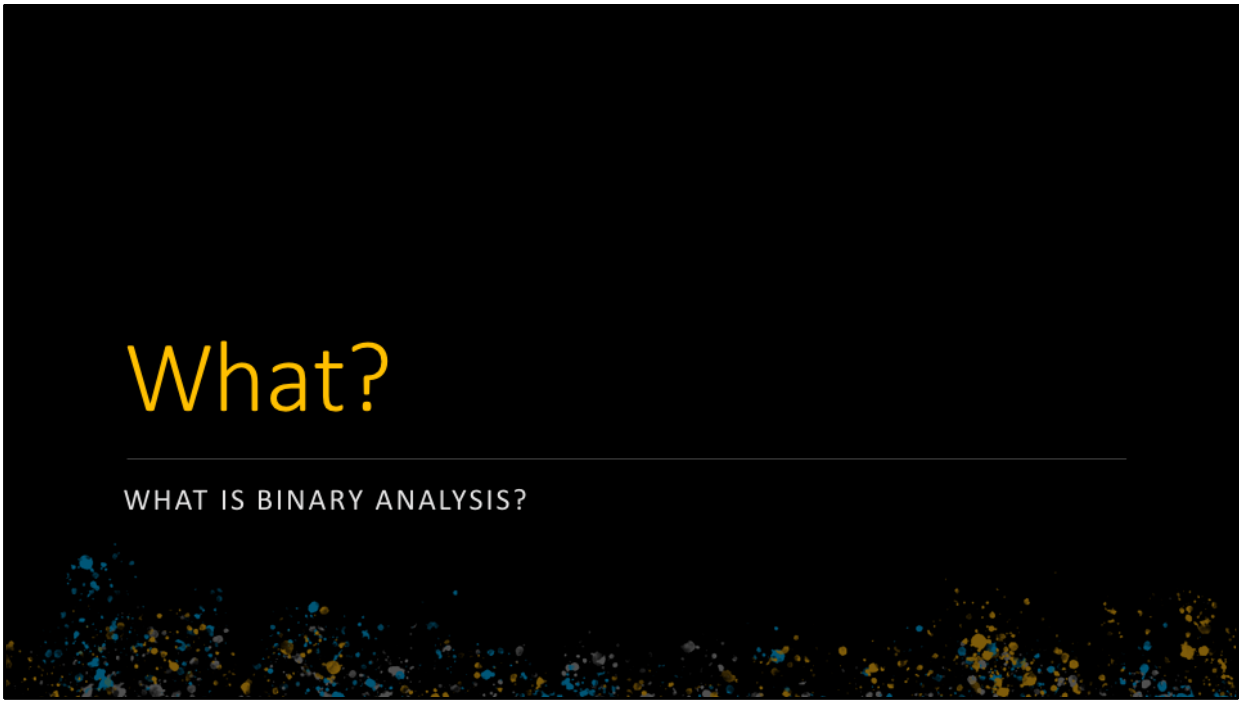
What is Binary Analysis

Why ILs

IL overview

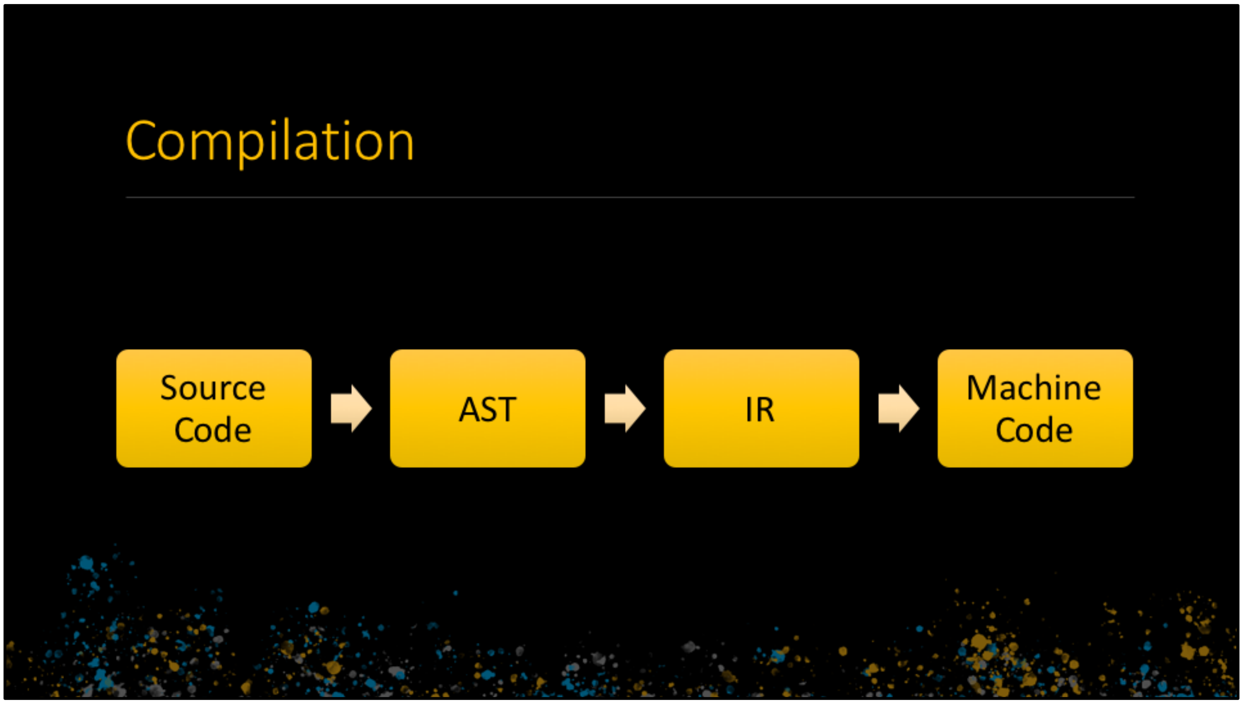
DEMOs

Justification – WHY you should be using Intermediate Languages
Introduction – Showing examples used in reverse engineering and the differences between them
Working with ILs – Some notes on how to best leverage ILs
DEMOs -- showing how to solve some common reverse engineering problems using ILs instead of raw assembly



What?

WHAT IS BINARY ANALYSIS?



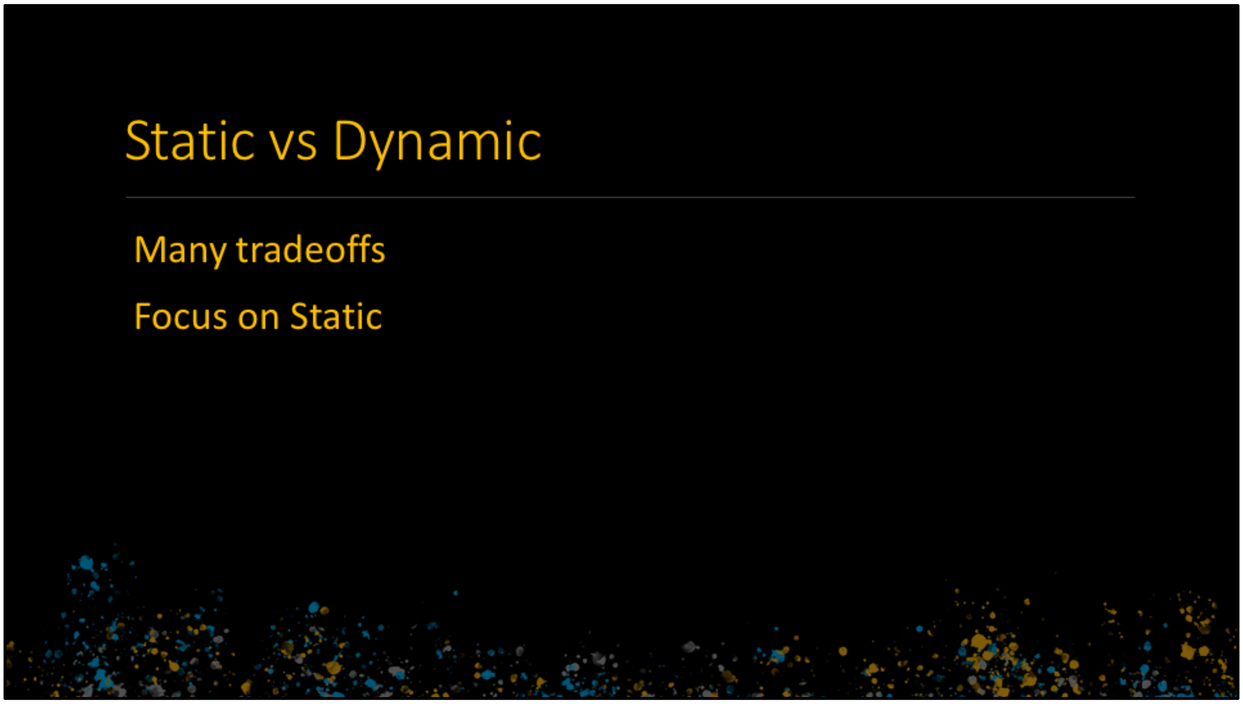
Compilation

Source Code → AST → IR → Machine Code

Very brief reminder from your introduction to compilers class.

Decompilation/Lifting

Lifting Is another



Static vs Dynamic

Many tradeoffs

Focus on Static

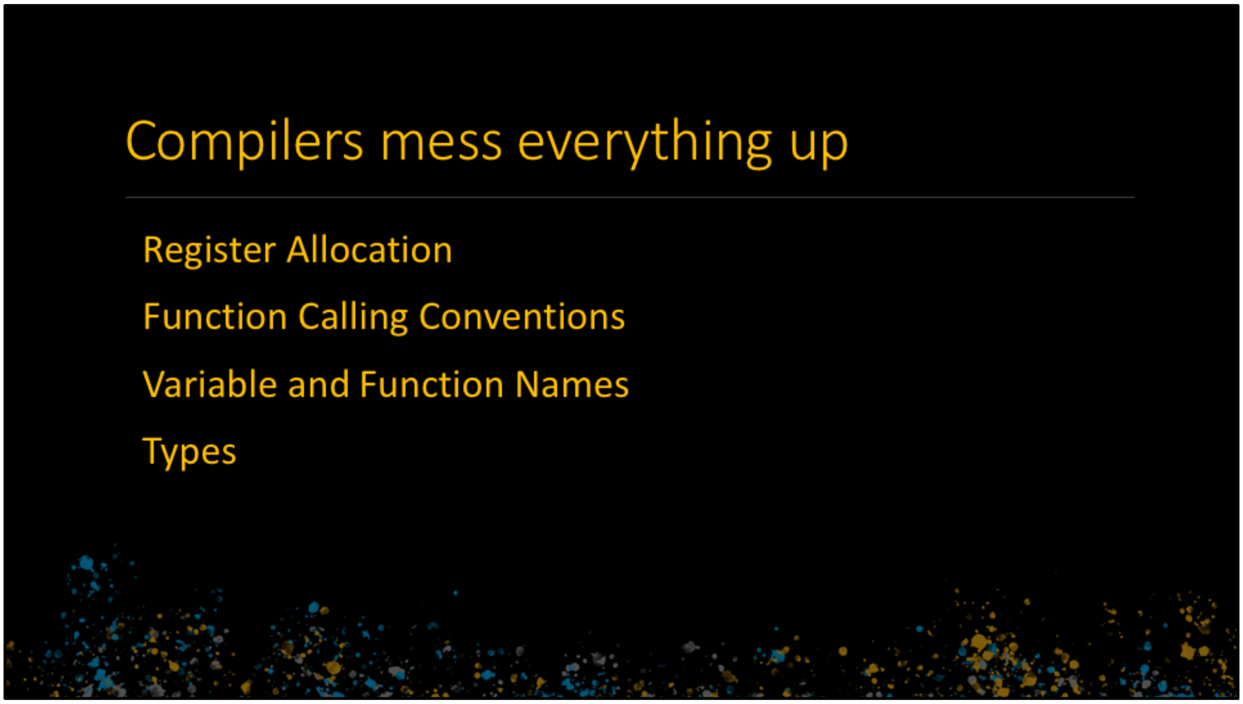
For the purposes of this talk we are going to stick with Static Binary Analysis

Binary Analysis != Source Analysis

All we have to do is decompile the binary… Right?

Binary Analysis I e Analysis

All we hav do is mpile the ary… Right?



Compilers mess everything up

Register Allocation

Function Calling Conventions

Variable and Function Names

Types

How Code generation works and why it makes Binary Analysis hard

Mapping between “infinite” set of variables, and finite set of registers

Process called “register allocation”

What happens when you have more variables than available registers?

Variables get “Spilled” on to the stack.

Function calls need to be made concrete.

Set of parameters being passed to a function need to be placed in specific registers

(or on the stack) given a predetermined “calling convention”

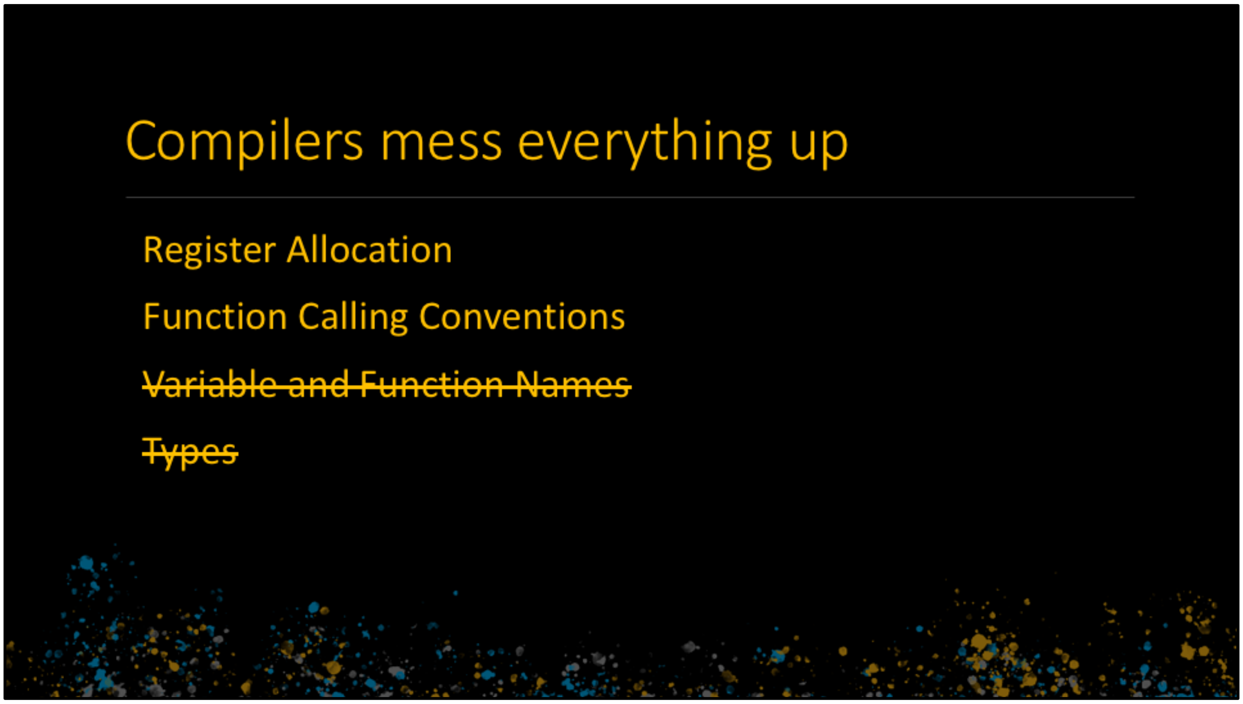
Variable and Functions Names are discarded

Types are discarded

There aren’t special instructions that let you know you’re working with a given type

This has to be determined indirectly

“Automatic structure/array recovery” is not a genericlly solvable problem



Compilers mess everything up

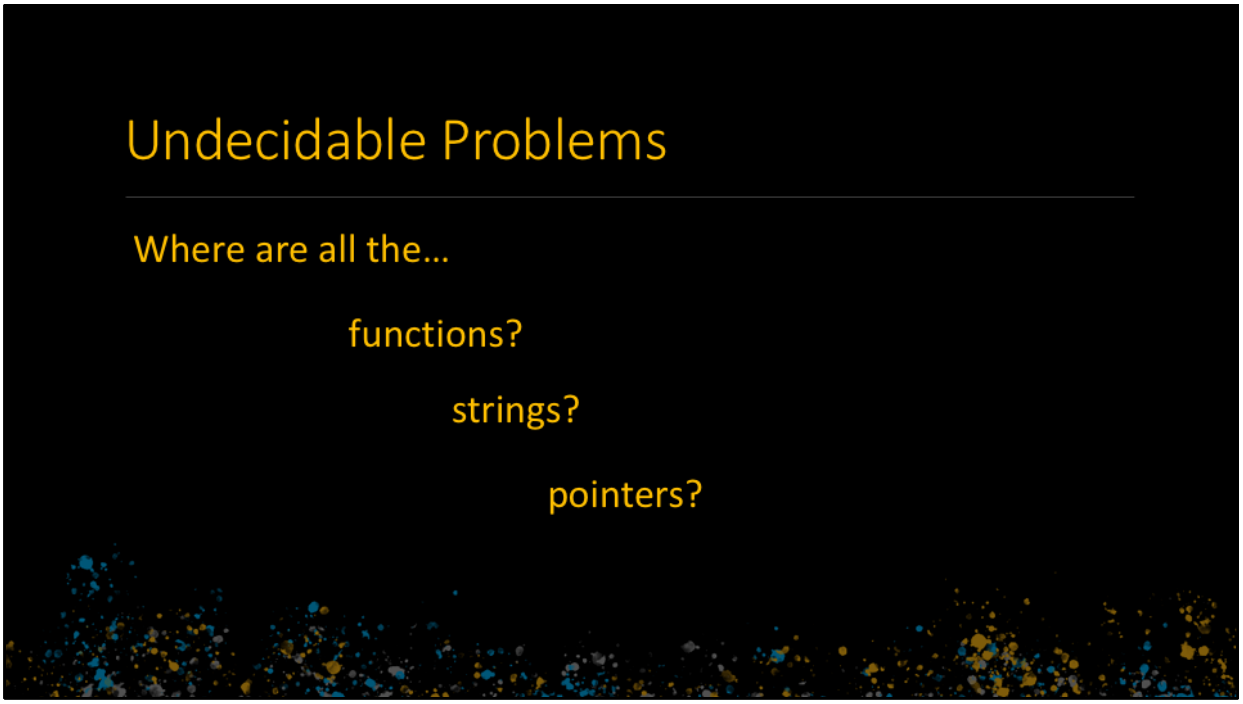
Register Allocation

Function Calling Conventions

~~Variable and Function Names~~

~~Types~~

How Code generation works and why it makes Binary Analysis hard
Mapping between “infinite” set of variables, and finite set of registers
Process called “register allocation”
What happens when you have more variables than available registers?
Variables get “Spilled” on to the stack.
Function calls need to be made concrete.
Set of parameters being passed to a function need to be placed in specific registers (or on the stack) given a predetermined “calling convention”
Types are discarded
There aren’t special instructions that let you know you’re working with a given type
This has to be determined indirectly
“Automatic structure/array recovery” is not a generally solvable problem



Undecidable Problems

Where are all the…

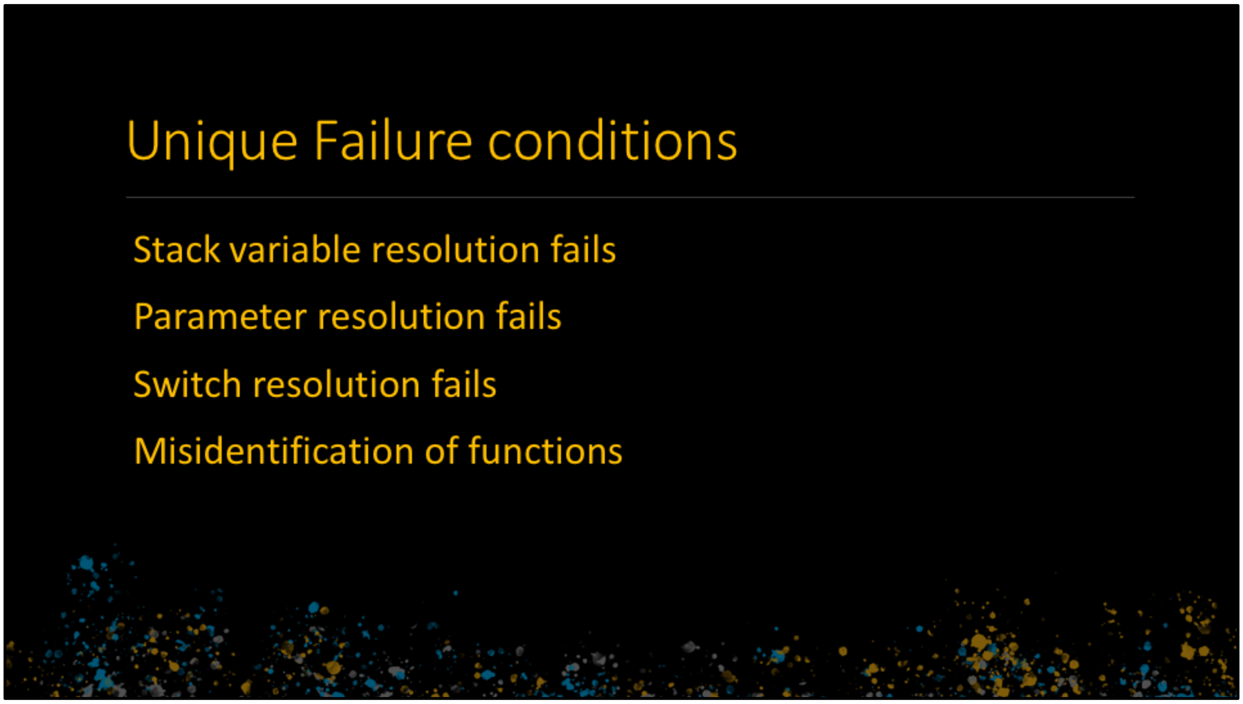
functions?

strings?

pointers?

To those not familiar:

“an undecidable problem is a problem that **requires a yes/no answer**, but where there cannot possibly be any computer program that always gives the correct answer”



Unique Failure conditions

Stack variable resolution fails

Parameter resolution fails

Switch resolution fails

Misidentification of functions

These undecidable problems lead to a set of failure conditions unique to static Binary Analysis

Thus you need to plan for dealing with issues like

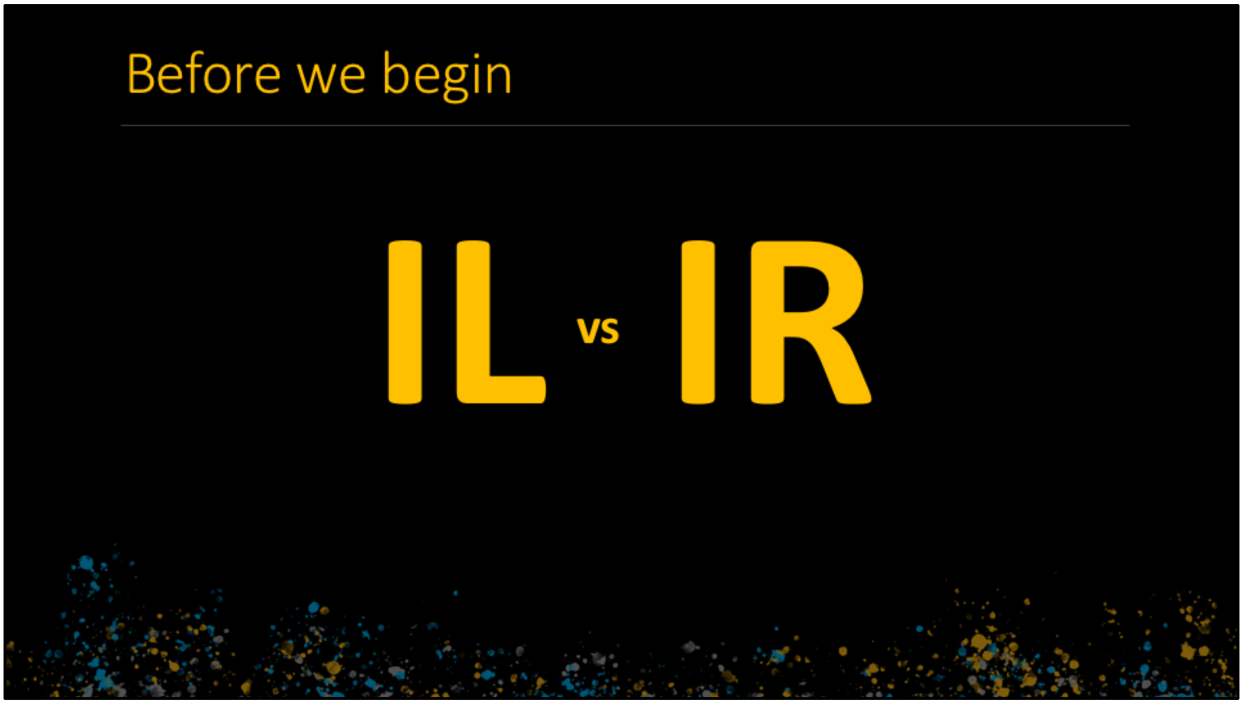
Stack variables can’t be resolved
Parameters can’t be determined
Indirect switch targets fail to be determined
False positive or false negative during Function identification

Reason discuss difficulties understand -> differences between Source & Binary

Unique requirements which need to be taken into account

Why?

WHY ILS?

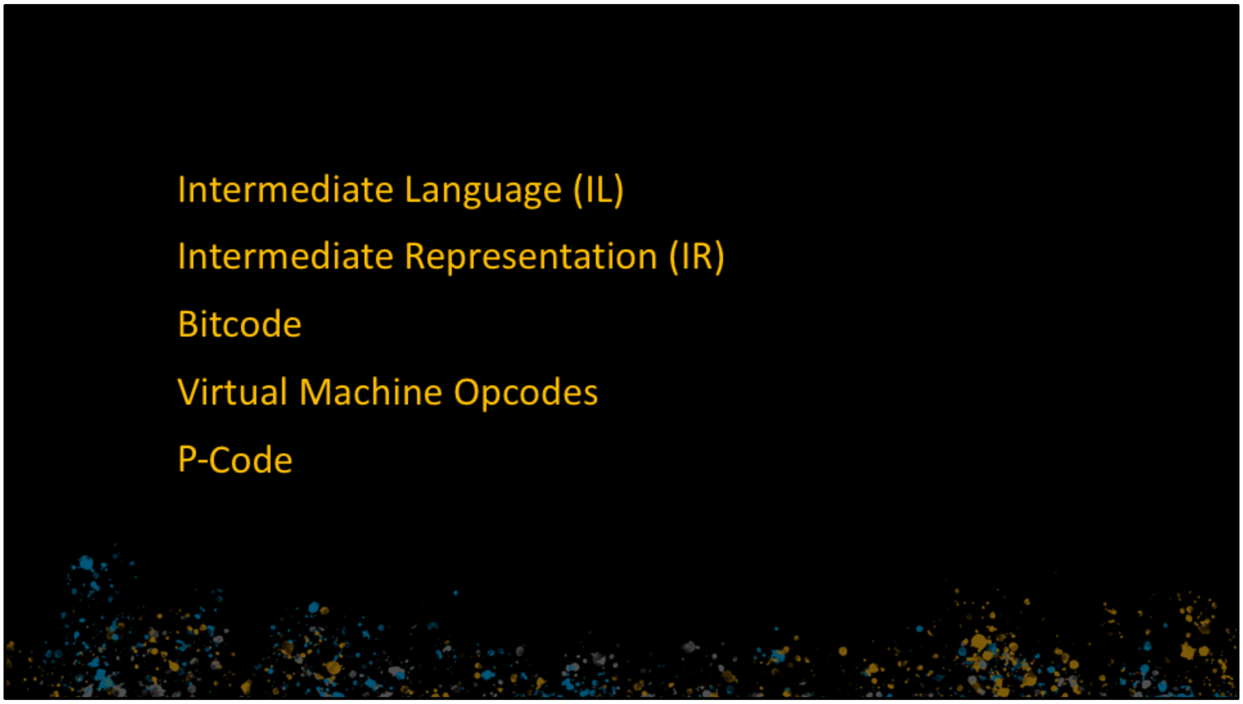


Before we begin

IL vs IR

Before begin, we wanted to have a quick note on the differences between an intermediate language and intermediate representation. They're often used interchangeably but there's one distinction that sometimes is important. An intermediate representation is not necessarily code. For example, you might choose to represent the data flows throughout a program via a graph showing sources and sinks. That would be an IR, but NOT an IL.

That said, we (well, Jordan in particular) has a bad habit of using the terms interchangeable so don't be surprised if we use them that way during this talk)



Intermediate Language (IL)

Intermediate Representation (IR)

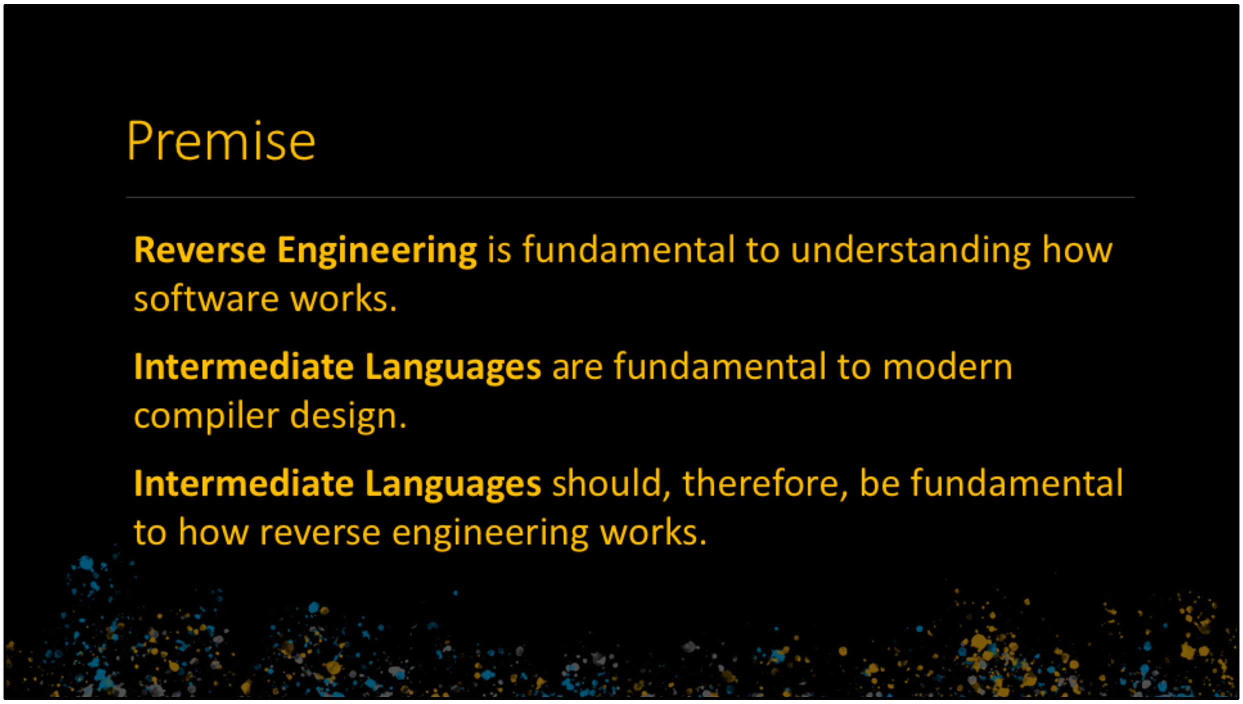
Bitcode

Virtual Machine Opcodes

P-Code

There are many different related terms for IRs or ILs. IR and IL are usually used interchangeably

P-Code is both the name of a specific implementation of an IL as well as a generic name for a portable machine code, another synonym for ILs.



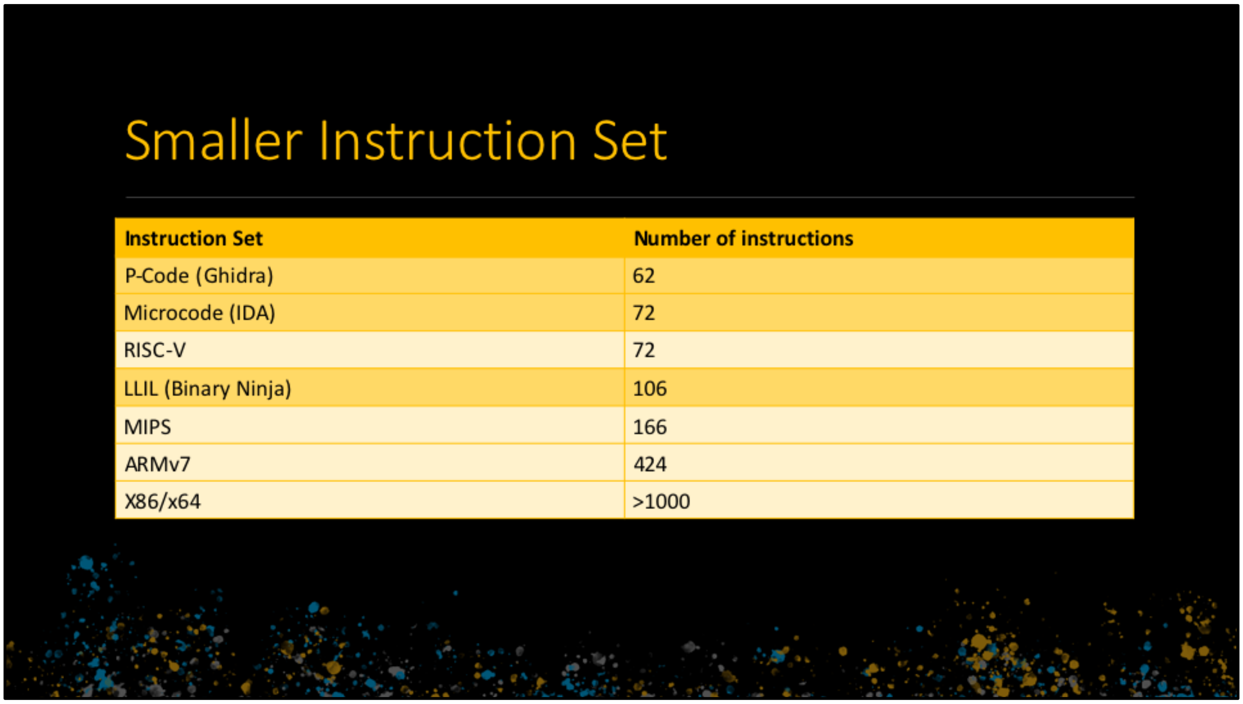
Premise

Reverse Engineering is fundamental to understanding how software works.

Intermediate Languages are fundamental to modern compiler design.

Intermediate Languages should, therefore, be fundamental to how reverse engineering works.

Intermediate Languages represent a mid-point between many modern source code languages and many modern architecutres, allowing optimizations and analysis to be shared amongst platforms.

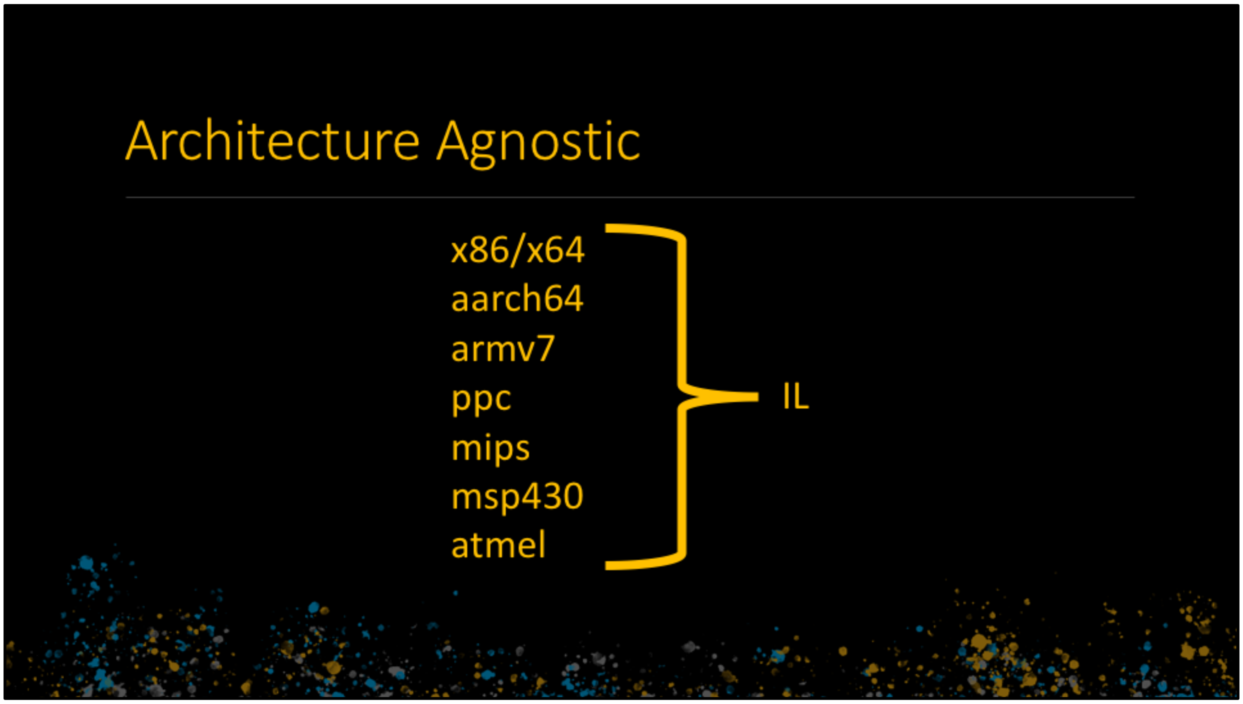


Smaller Instruction Set

Instruction Set	Number of instructions
P-Code (Ghidra)	62
Microcode (IDA)	72
RISC-V	72
LLIL (Binary Ninja)	106
MIPS	166
ARMv7	424
X86/x64	>1000

Given the option, would you want to write code that had to handle 1000 different unique instructions, or 100? It’s worth noting that there’s a LOT of ways you could change these numbers. X64 alone could be as many as almost 4000 instructions if you

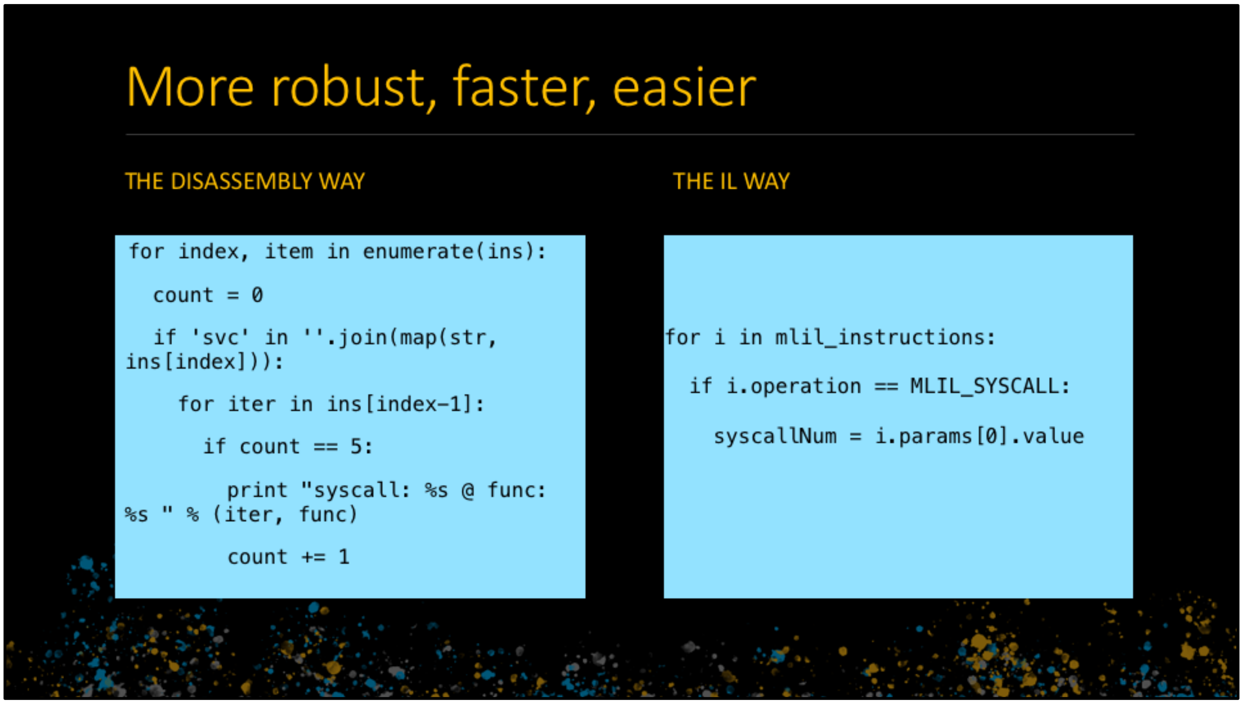
Data sources available from: https://docs.google.com/spreadsheets/d/15-GlRhASzk-I2vzqjIJs9brl-kkwohYdeg5-voMQS3o/edit?usp=sharing



Architecture Agnostic

x86/x64
aarch64
armv7
ppc
mips
msp430
atmel

IL



More robust, faster, easier

THE DISASSEMBLY WAY

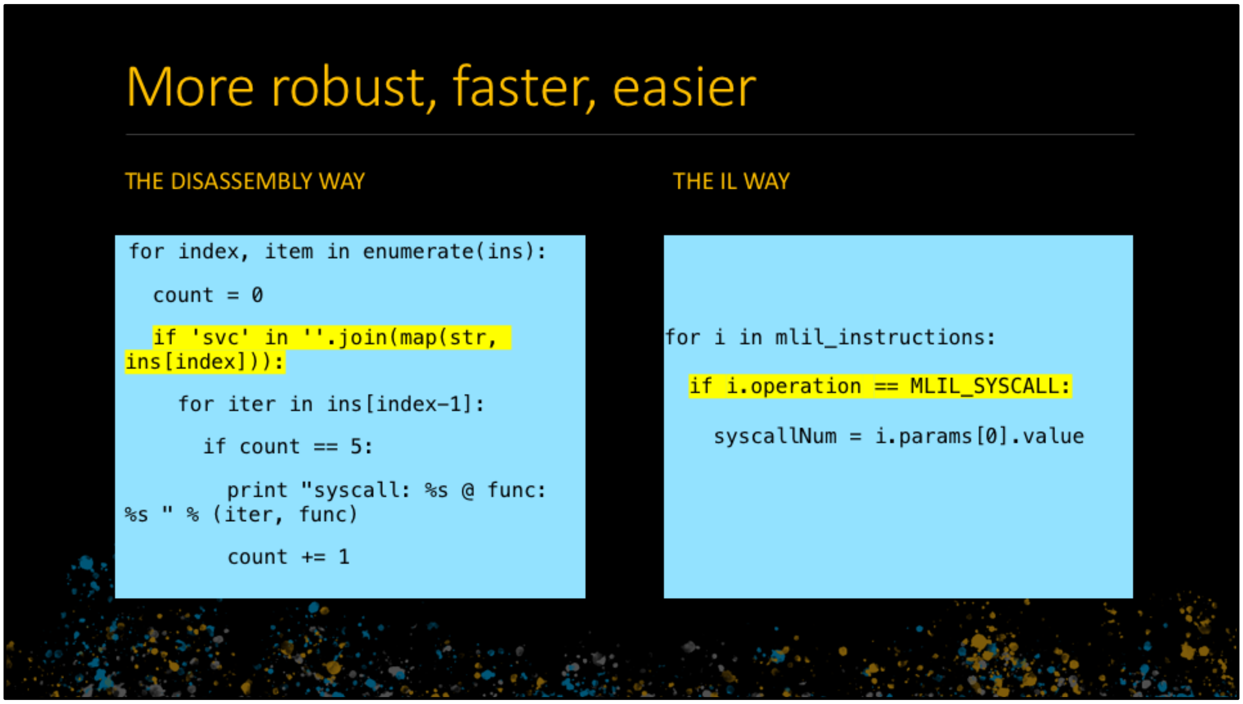
```
for index, item in enumerate(ins):
  count = 0
  if 'svc' in ''.join(map(str,
ins[index])):
    for iter in ins[index-1]:
      if count == 5:
        print "syscall: %s @ func:
%s " % (iter, func)
        count += 1
```

THE IL WAY

```
for i in mlil_instructions:
  if i.operation == MLIL_SYSCALL:
    syscallNum = i.params[0].value
```

Taken from: http://arm.ninja/2016/03/08/intro-to-binary-ninja-api/

Note that the assembly code is significantly more brittle, it will stop working if the compiler ever created even a slightly modified sequence of instructions such as re-using an existing constant value from another register, assigning to x8 anywhere except the prior instruction, etc. So not only is the IL implementation easier to write, but it's more robust, requires you to know less about the specific platform implementation, but it will also work out of the box with multiple architectures.



More robust, faster, easier

THE DISASSEMBLY WAY

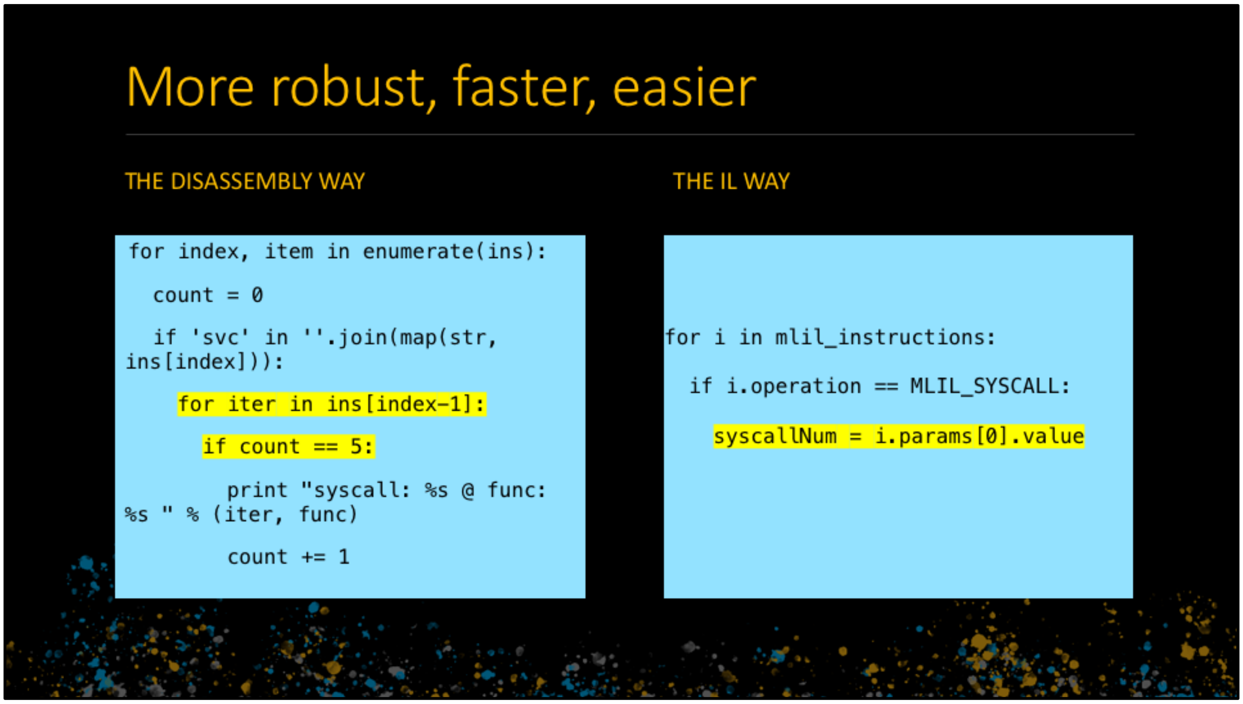
```
for index, item in enumerate(ins):
  count = 0
  if 'svc' in ''.join(map(str,
ins[index])):
    for iter in ins[index-1]:
      if count == 5:
        print "syscall: %s @ func:
%s " % (iter, func)
        count += 1
```

THE IL WAY

```
for i in mlil_instructions:
  if i.operation == MLIL_SYSCALL:
    syscallNum = i.params[0].value
```

Taken from: http://arm.ninja/2016/03/08/intro-to-binary-ninja-api/

Note that the assembly code is significantly more brittle, it will stop working if the compiler ever created even a slightly modified sequence of instructions such as re-using an existing constant value from another register, assigning to x8 anywhere except the prior instruction, etc. So not only is the IL implementation easier to write, but it's more robust, requires you to know less about the specific platform implementation, but it will also work out of the box with multiple architectures.



More robust, faster, easier

THE DISASSEMBLY WAY

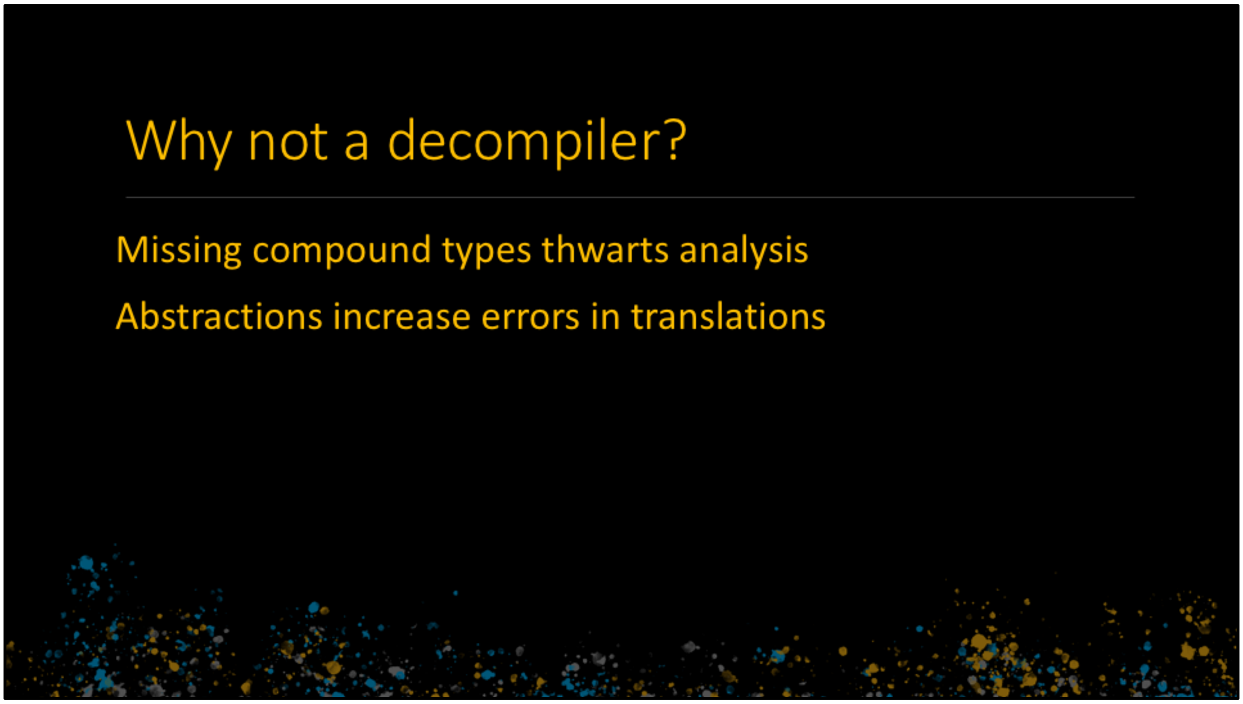
```
for index, item in enumerate(ins):
  count = 0
  if 'svc' in ''.join(map(str,
ins[index])):
    for iter in ins[index-1]:
      if count == 5:
        print "syscall: %s @ func:
%s " % (iter, func)
        count += 1
```

THE IL WAY

```
for i in mlil_instructions:
  if i.operation == MLIL_SYSCALL:
    syscallNum = i.params[0].value
```

Taken from: http://arm.ninja/2016/03/08/intro-to-binary-ninja-api/

Note that the assembly code is significantly more brittle, it will stop working if the compiler ever created even a slightly modified sequence of instructions such as re-using an existing constant value from another register, assigning to x8 anywhere except the prior instruction, etc. So not only is the IL implementation easier to write, but it’s more robust, requires you to know less about the specific platform implementation, but it will also work out of the box with multiple architectures.

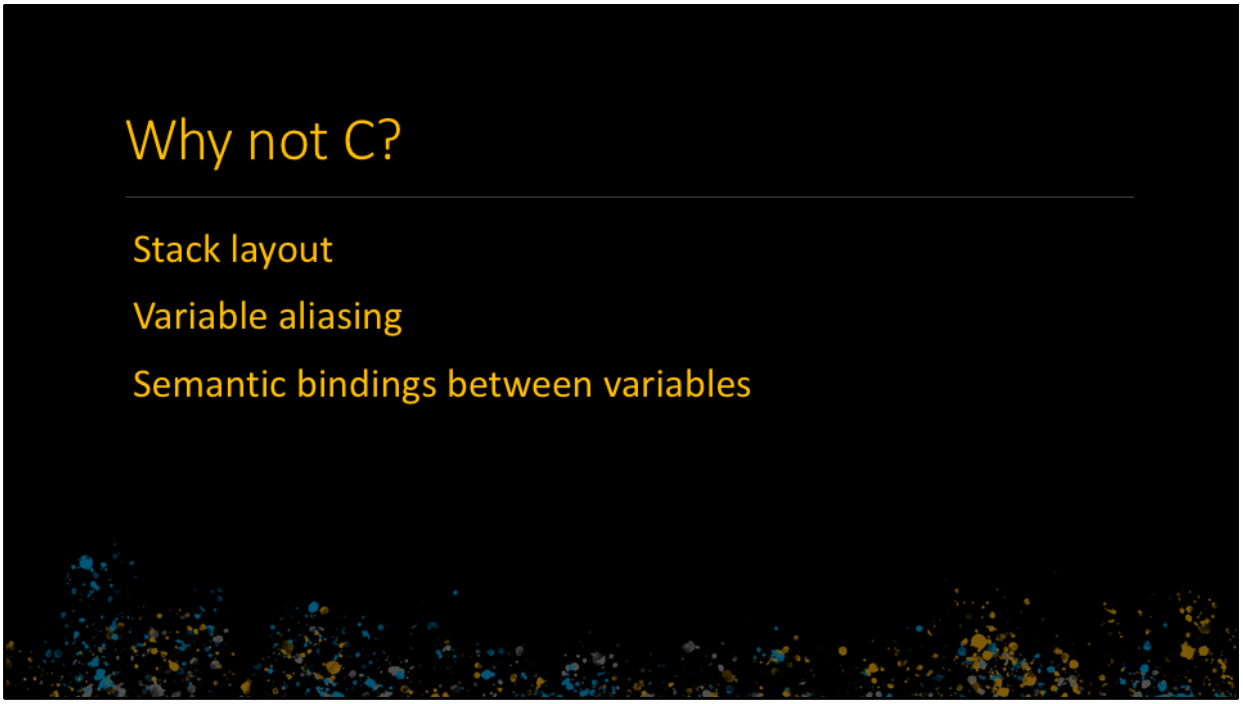


Why not a decompiler?

Missing compound types thwarts analysis

Abstractions increase errors in translations

Decompile so we can analyze with existing source analysis tools.
Existing source analysis tools don’t work well on “just a bunch of pointers”



Why not C?

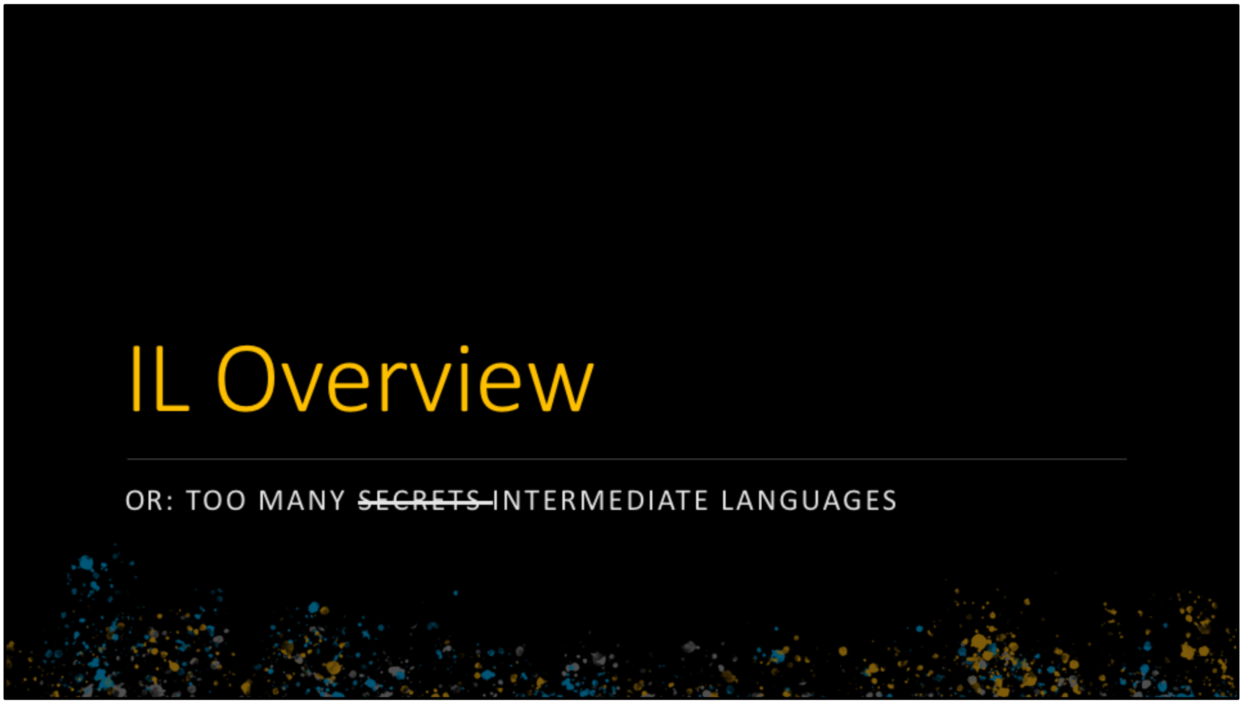
Stack layout

Variable aliasing

Semantic bindings between variables

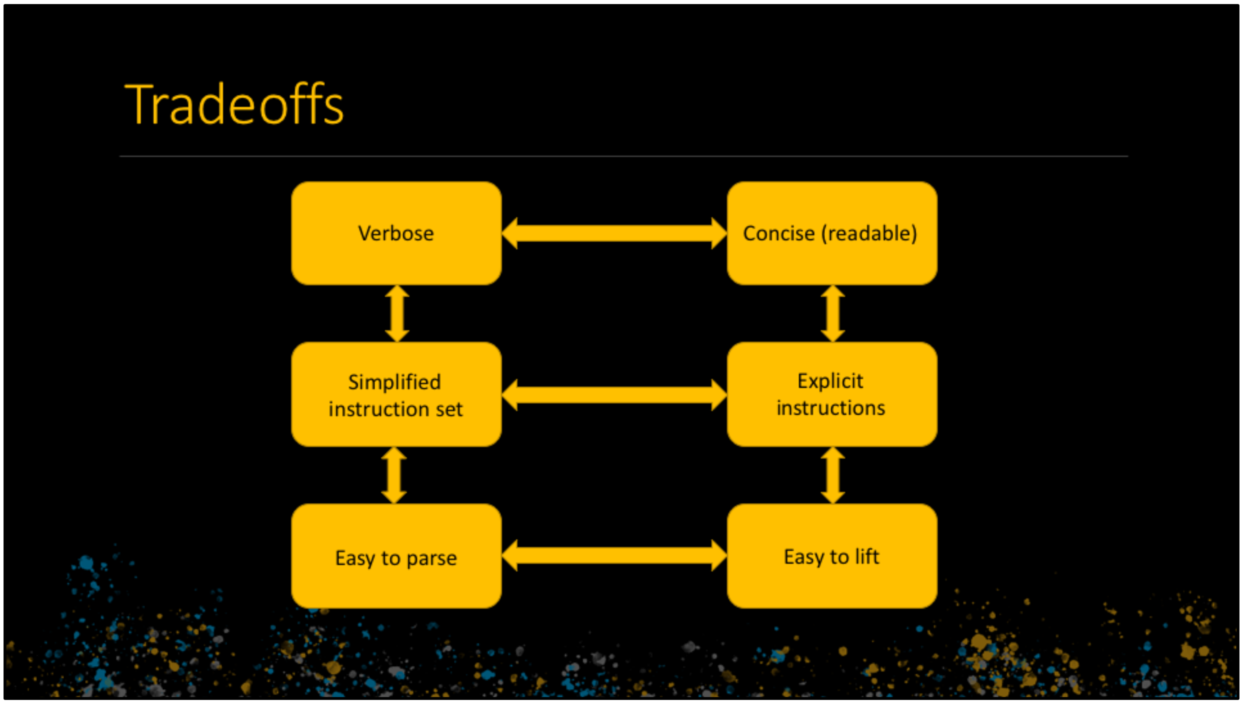
People think C is the ultimate goal of decompilation.

Many things that can be recovered from the binary don’t have C-language constructs



IL Overview

OR: TOO MANY ~~SECRETS~~ INTERMEDIATE LANGUAGES



Tradeoffs

Verbose

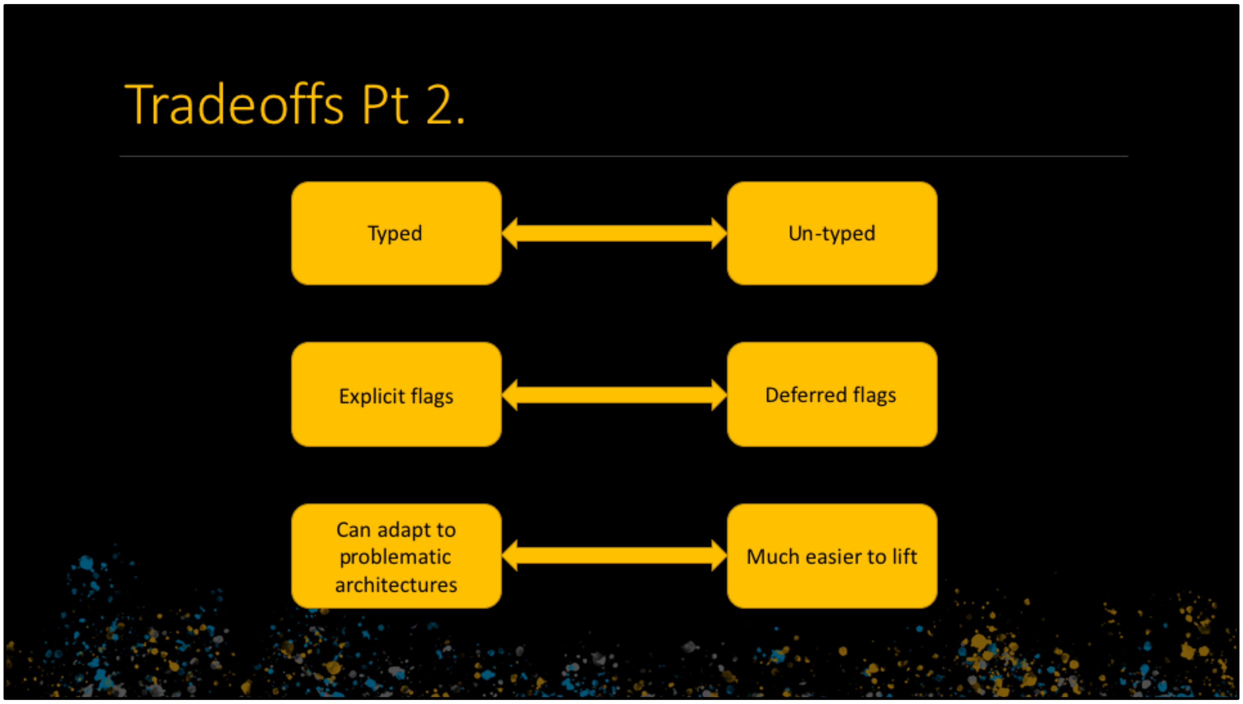
Concise (readable)

Simplified instruction set

Explicit instructions

Easy to parse

Easy to lift



Tradeoffs Pt 2.

Typed

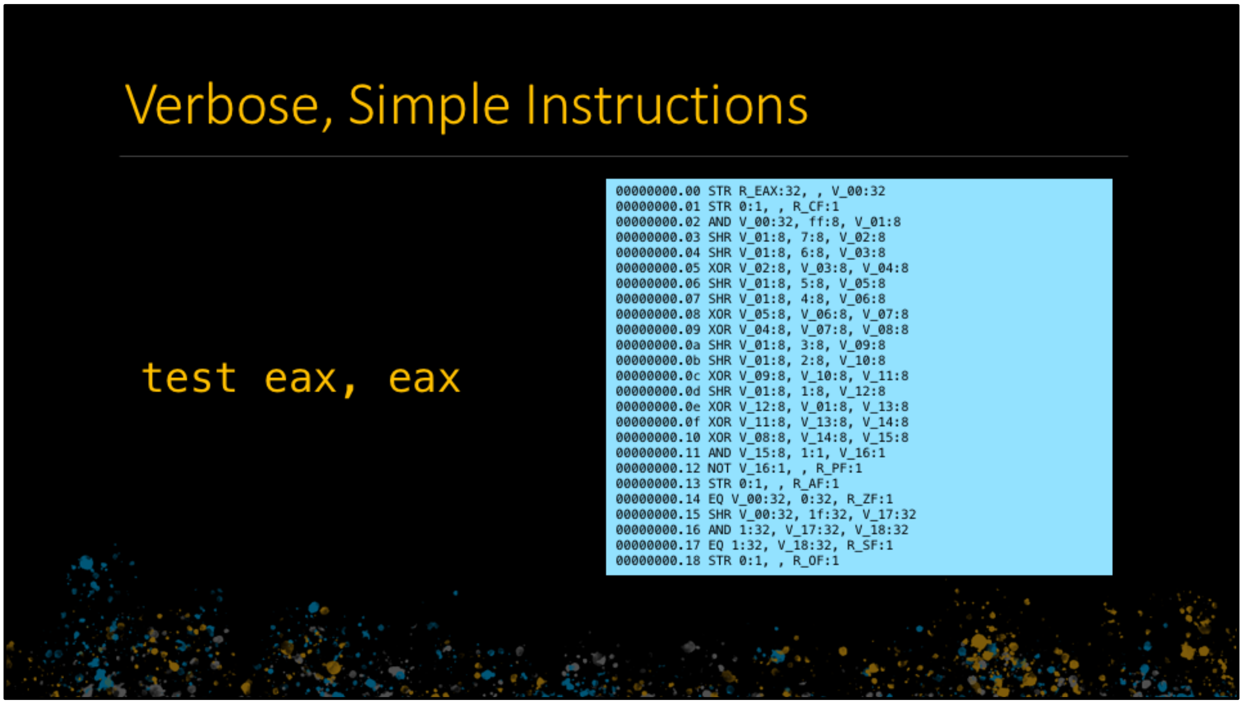
Un-typed

Explicit flags

Deferred flags

Can adapt to problematic architectures

Much easier to lift

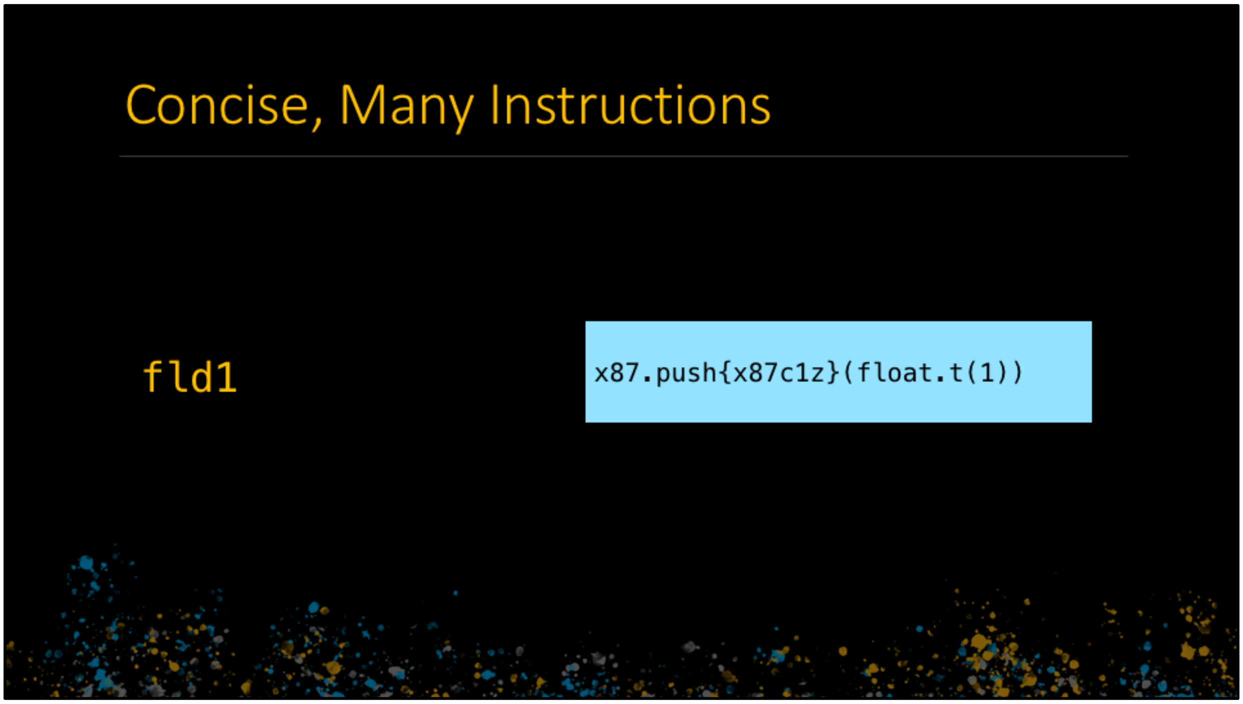


Verbose, Simple Instructions

```
test eax, eax
```

```
00000000.00 STR R_EAX:32, , V_00:32
00000000.01 STR 0:1, , R_CF:1
00000000.02 AND V_00:32, ff:8, V_01:8
00000000.03 SHR V_01:8, 7:8, V_02:8
00000000.04 SHR V_01:8, 6:8, V_03:8
00000000.05 XOR V_02:8, V_03:8, V_04:8
00000000.06 SHR V_01:8, 5:8, V_05:8
00000000.07 SHR V_01:8, 4:8, V_06:8
00000000.08 XOR V_05:8, V_06:8, V_07:8
00000000.09 XOR V_04:8, V_07:8, V_08:8
00000000.0a SHR V_01:8, 3:8, V_09:8
00000000.0b SHR V_01:8, 2:8, V_10:8
00000000.0c XOR V_09:8, V_10:8, V_11:8
00000000.0d SHR V_01:8, 1:8, V_12:8
00000000.0e XOR V_12:8, V_01:8, V_13:8
00000000.0f XOR V_11:8, V_13:8, V_14:8
00000000.10 XOR V_08:8, V_14:8, V_15:8
00000000.11 AND V_15:8, 1:1, V_16:1
00000000.12 NOT V_16:1, , R_PF:1
00000000.13 STR 0:1, , R_AF:1
00000000.14 EQ V_00:32, 0:32, R_ZF:1
00000000.15 SHR V_00:32, 1f:32, V_17:32
00000000.16 AND 1:32, V_17:32, V_18:32
00000000.17 EQ 1:32, V_18:32, R_SF:1
00000000.18 STR 0:1, , R_OF:1
```

REIL Zynamics



Concise, Many Instructions

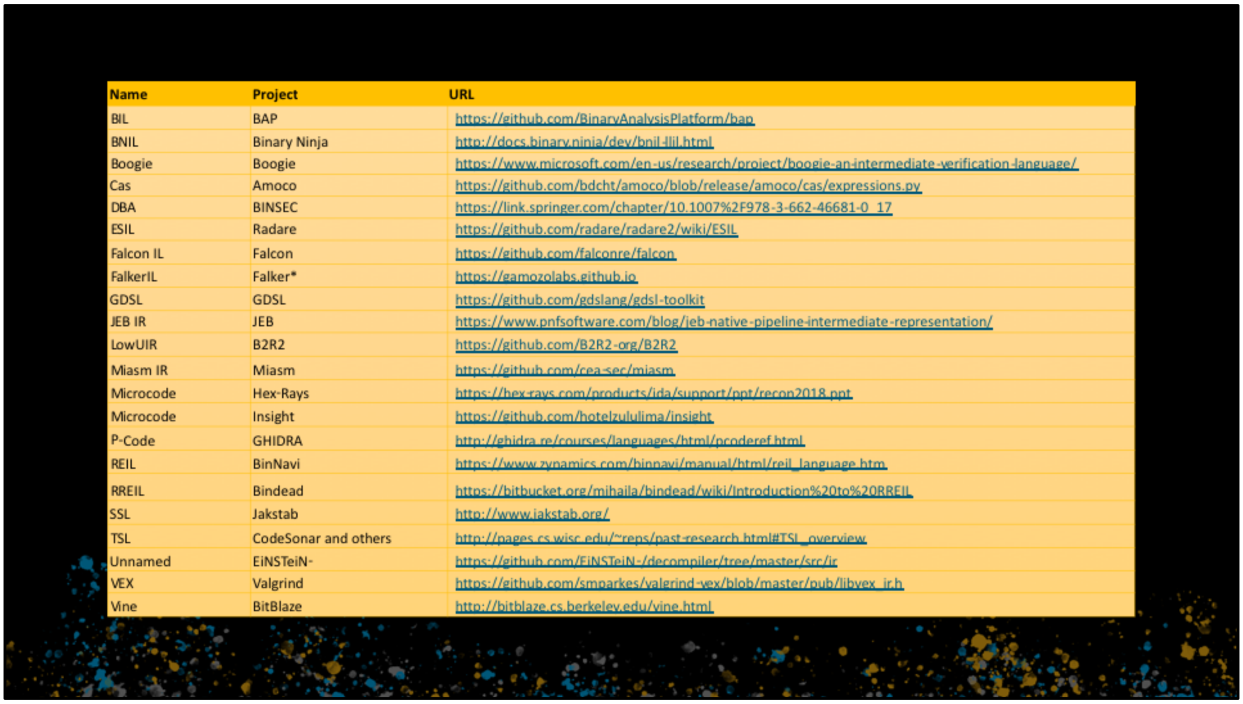
```
fld1
```

```
x87.push{x87c1z}(float.t(1))
```

BN LLIL

Landscape of ILs

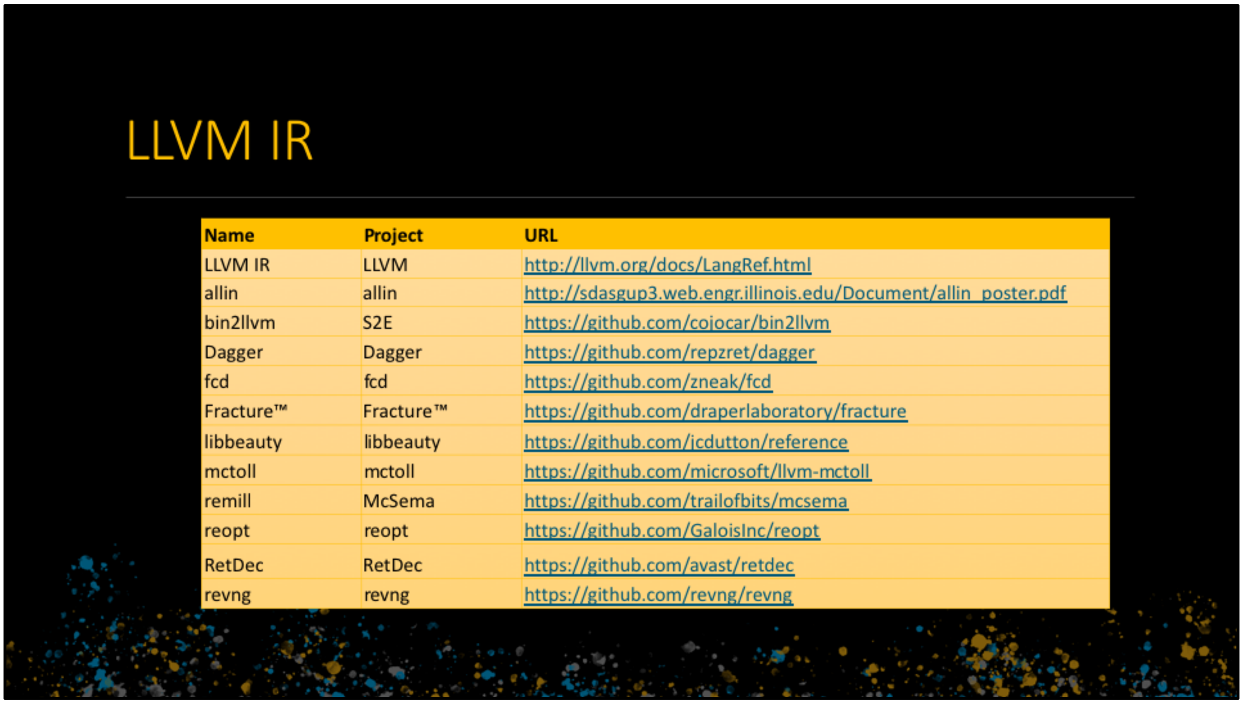
Just ones relevant to security/RE/binary lifting. The following slides are not meant to be read, they’re really just to emphasize that there are far too many options in this space.



Name	Project	URL
BIL	BAP	https://github.com/BinaryAnalysisPlatform/bap
BNIL	Binary Ninja	http://docs.binary.ninja/dev/bnil-llil.html
Boogie	Boogie	https://www.microsoft.com/en-us/research/project/boogie-an-intermediate-verification-language/
Cas	Amoco	https://github.com/bdcht/amoco/blob/release/amoco/cas/expressions.py
DBA	BINSEC	https://link.springer.com/chapter/10.1007%2F978-3-662-46681-0_17
ESIL	Radare	https://github.com/radare/radare2/wiki/ESIL
Falcon IL	Falcon	https://github.com/falconre/falcon
FalkerIL	Falker*	https://gamozolabs.github.io
GDSL	GDSL	https://github.com/gdslang/gdsl-toolkit
JEB IR	JEB	https://www.pnfsoftware.com/blog/jeb-native-pipeline-intermediate-representation/
LowUIR	B2R2	https://github.com/B2R2-org/B2R2
Miasm IR	Miasm	https://github.com/cea-sec/miasm
Microcode	Hex-Rays	https://hex-rays.com/products/ida/support/ppt/recon2018.ppt
Microcode	Insight	https://github.com/hotelzululima/insight
P-Code	GHIDRA	http://ghidra.re/courses/languages/html/pcoderef.html
REIL	BinNavi	https://www.zynamics.com/binnavi/manual/html/reil_language.htm
RREIL	Bindead	https://bitbucket.org/mihaila/bindead/wiki/Introduction%20to%20RREIL
SSL	Jakstab	http://www.jakstab.org/
TSL	CodeSonar and others	http://pages.cs.wisc.edu/~reps/past-research.html#TSL_overview
Unnamed	EiNSTeiN-	https://github.com/EiNSTeiN-/decompiler/tree/master/src/ir
VEX	Valgrind	https://github.com/smparkes/valgrind-vex/blob/master/pub/libvex_ir.h
Vine	BitBlaze	http://bitblaze.cs.berkeley.edu/vine.html

I don't expect anyone to read this now and I'm not going to cover all of these since there's a ton. Heck, there's even several with the same name! And these are just the ones that have been used for security analysis or reverse engineering. There are probably hundreds of total intermediate languages in total, with more growing by the minute.

RAW data (comments welcome)
https://docs.google.com/spreadsheets/d/1XPTe5sj1Vx9O40HuKLadU-pwit91Hzk_YdrV8wcFllQ

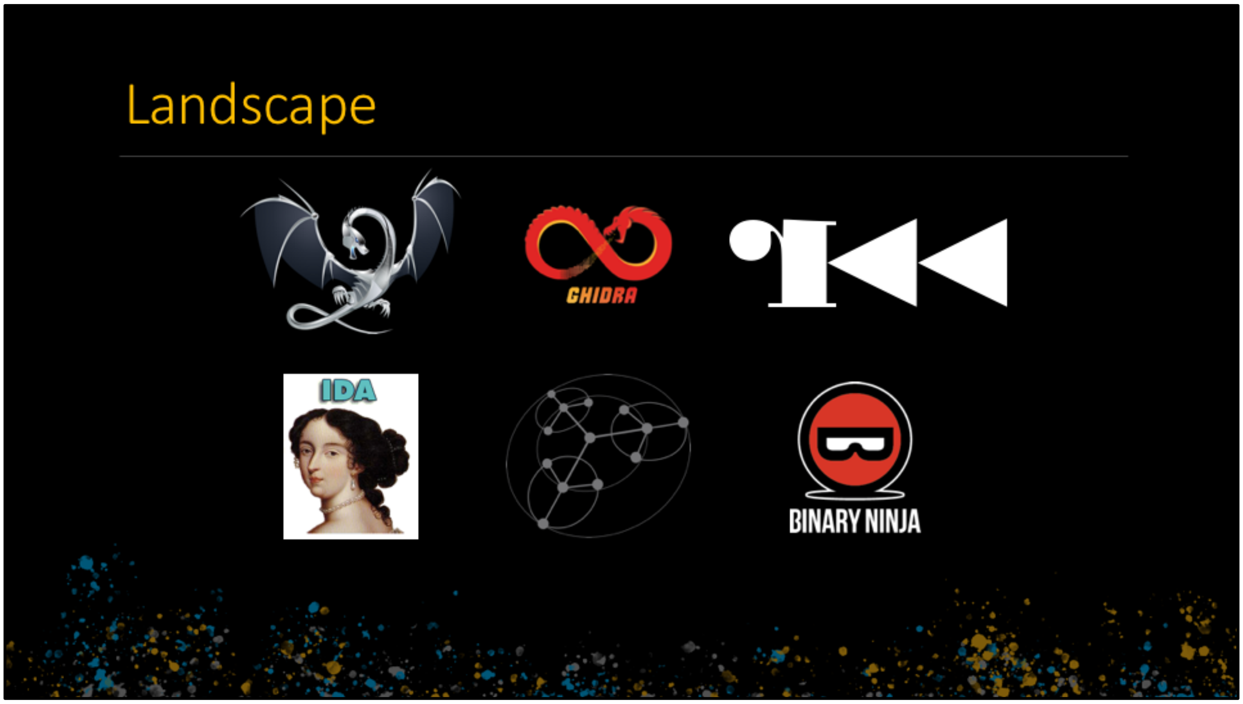


LLVM IR

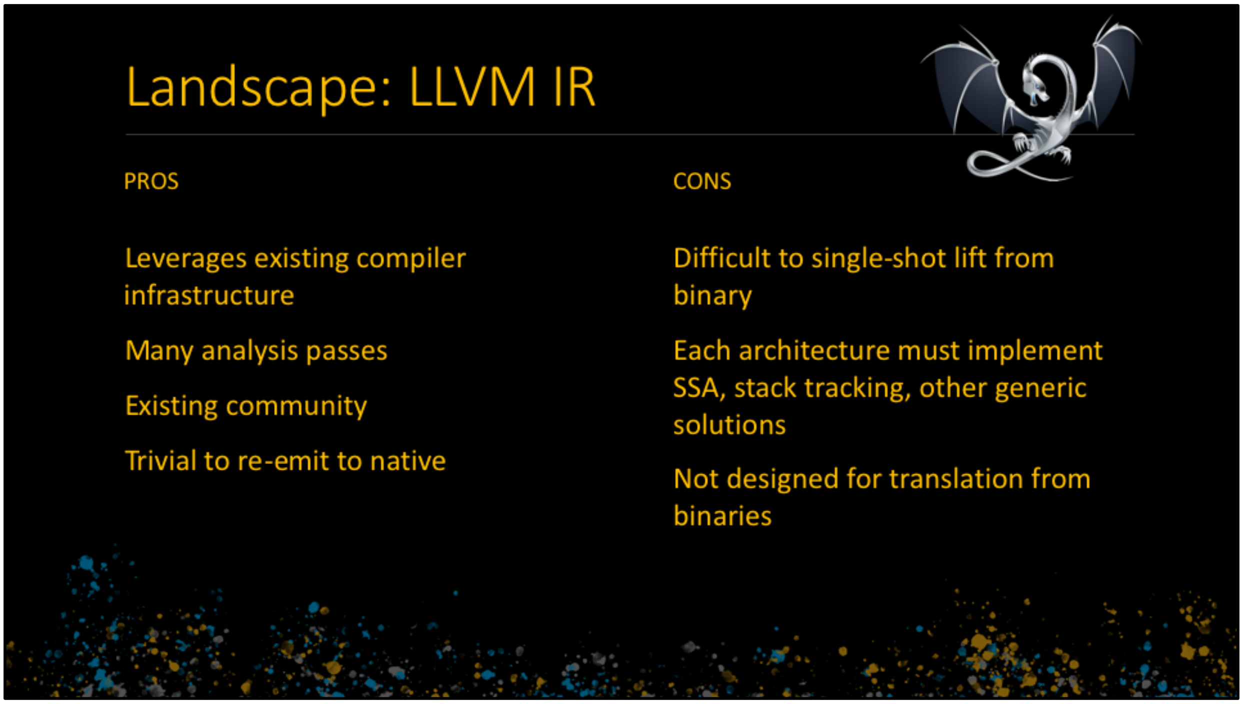
Name	Project	URL
LLVM IR	LLVM	http://llvm.org/docs/LangRef.html
allin	allin	http://sdasgup3.web.engr.illinois.edu/Document/allin_poster.pdf
bin2llvm	S2E	https://github.com/cojocar/bin2llvm
Dagger	Dagger	https://github.com/repzret/dagger
fcd	fcd	https://github.com/zneak/fcd
Fracture™	Fracture™	https://github.com/draperlaboratory/fracture
libbeauty	libbeauty	https://github.com/jcdutton/reference
mctoll	mctoll	https://github.com/microsoft/llvm-mctoll
remill	McSema	https://github.com/trailofbits/mcsema
reopt	reopt	https://github.com/GaloisInc/reopt
RetDec	RetDec	https://github.com/avast/retdec
revng	revng	https://github.com/revng/revng

And then there's the entire family of systems that just translate to LLVM IR. Sometimes the goal is to just re-emit the binary for a different architecture, but given the prevalence of a number of LLVM IR security analysis passes, that's often a common reason as well.

A great overview table is maintained by Trail of Bits on their McSemo project page: https://github.com/trailofbits/mcsema#comparison-with-other-machine-code-to-llvm-bitcode-lifters



Landscape



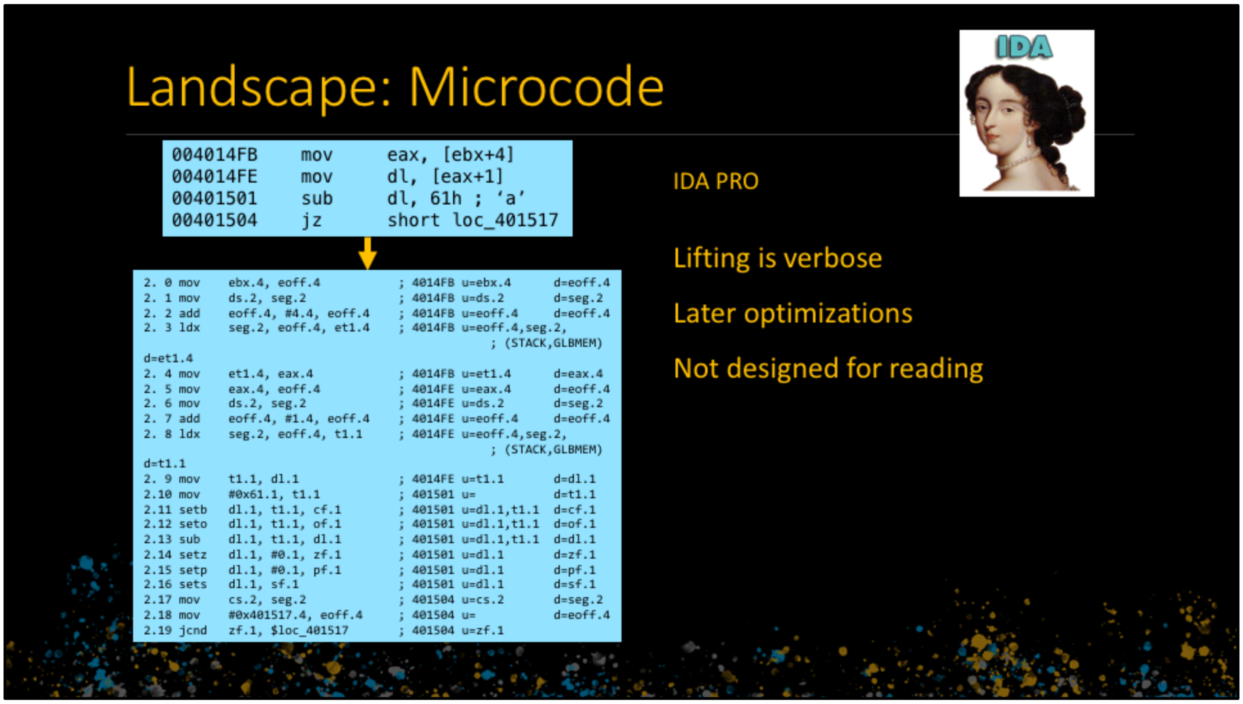
Landscape: LLVM IR

PROS

- Leverages existing compiler infrastructure
- Many analysis passes
- Existing community
- Trivial to re-emit to native

CONS

- Difficult to single-shot lift from binary
- Each architecture must implement SSA, stack tracking, other generic solutions
- Not designed for translation from binaries



Landscape: Microcode

```
004014FB     mov       eax, [ebx+4]
004014FE     mov       dl, [eax+1]
00401501     sub       dl, 61h ; ‘a’
00401504     jz        short loc_401517
```

```
2. 0 mov    ebx.4, eoff.4           ; 4014FB u=ebx.4       d=eoff.4
2. 1 mov    ds.2, seg.2             ; 4014FB u=ds.2        d=seg.2
2. 2 add    eoff.4, #4.4, eoff.4    ; 4014FB u=eoff.4      d=eoff.4
2. 3 ldx    seg.2, eoff.4, et1.4    ; 4014FB u=eoff.4,seg.2,
                                                ; (STACK,GLBMEM)
d=et1.4
2. 4 mov    et1.4, eax.4            ; 4014FB u=et1.4       d=eax.4
2. 5 mov    eax.4, eoff.4           ; 4014FE u=eax.4       d=eoff.4
2. 6 mov    ds.2, seg.2             ; 4014FE u=ds.2        d=seg.2
2. 7 add    eoff.4, #1.4, eoff.4    ; 4014FE u=eoff.4      d=eoff.4
2. 8 ldx    seg.2, eoff.4, t1.1     ; 4014FE u=eoff.4,seg.2,
                                                ; (STACK,GLBMEM)
d=t1.1
2. 9 mov    t1.1, dl.1              ; 4014FE u=t1.1        d=dl.1
2.10 mov    #0x61.1, t1.1           ; 401501 u=            d=t1.1
2.11 setb   dl.1, t1.1, cf.1        ; 401501 u=dl.1,t1.1   d=cf.1
2.12 seto   dl.1, t1.1, of.1        ; 401501 u=dl.1,t1.1   d=of.1
2.13 sub    dl.1, t1.1, dl.1        ; 401501 u=dl.1,t1.1   d=dl.1
2.14 setz   dl.1, #0.1, zf.1        ; 401501 u=dl.1        d=zf.1
2.15 setp   dl.1, #0.1, pf.1        ; 401501 u=dl.1        d=pf.1
2.16 sets   dl.1, sf.1              ; 401501 u=dl.1        d=sf.1
2.17 mov    cs.2, seg.2             ; 401504 u=cs.2        d=seg.2
2.18 mov    #0x401517.4, eoff.4     ; 401504 u=            d=eoff.4
2.19 jcnd   zf.1, $loc_401517       ; 401504 u=zf.1
```

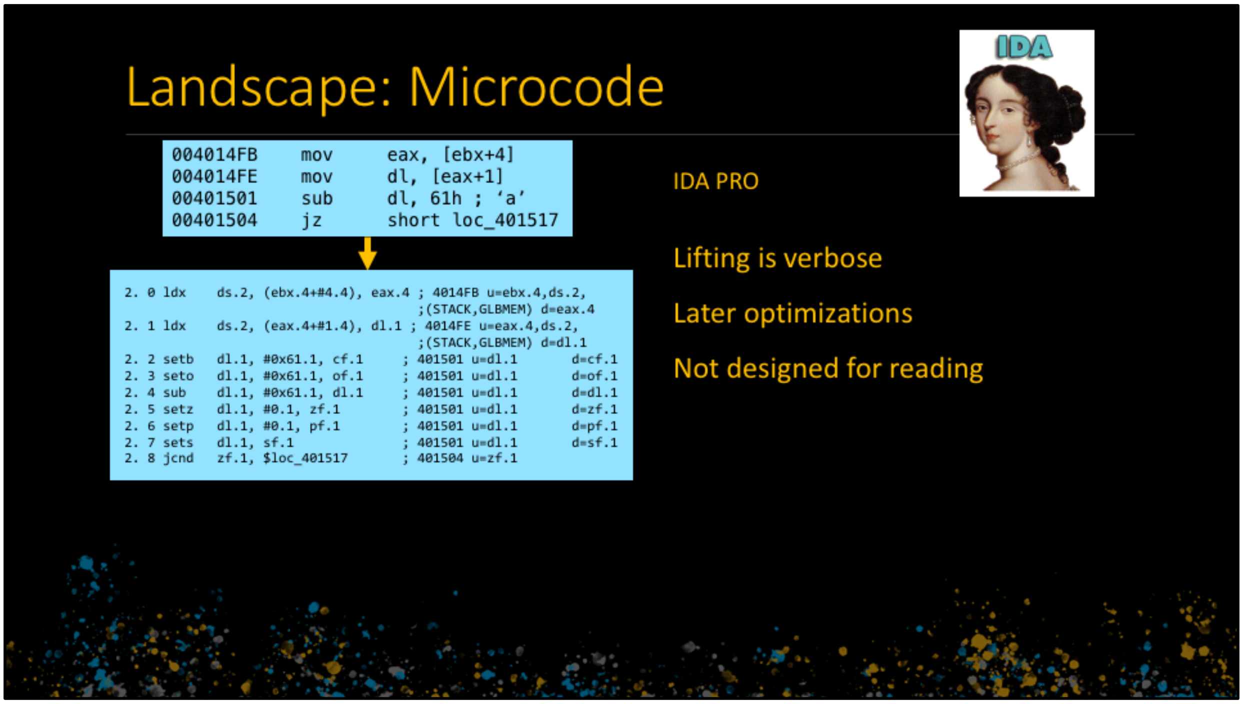
IDA PRO

Lifting is verbose

Later optimizations

Not designed for reading

Credit: Ilfak Guilfanov: https://hex-rays.com/products/ida/support/ppt/recon2018.ppt



Landscape: Microcode

```
004014FB    mov     eax, [ebx+4]
004014FE    mov     dl, [eax+1]
00401501    sub     dl, 61h ; ‘a’
00401504    jz      short loc_401517
```

```
2. 0 ldx    ds.2, (ebx.4+#4.4), eax.4 ; 4014FB u=ebx.4,ds.2,
                                      ;(STACK,GLBMEM) d=eax.4
2. 1 ldx    ds.2, (eax.4+#1.4), dl.1 ; 4014FE u=eax.4,ds.2,
                                      ;(STACK,GLBMEM) d=dl.1
2. 2 setb   dl.1, #0x61.1, cf.1      ; 401501 u=dl.1       d=cf.1
2. 3 seto   dl.1, #0x61.1, of.1      ; 401501 u=dl.1       d=of.1
2. 4 sub    dl.1, #0x61.1, dl.1      ; 401501 u=dl.1       d=dl.1
2. 5 setz   dl.1, #0.1, zf.1         ; 401501 u=dl.1       d=zf.1
2. 6 setp   dl.1, #0.1, pf.1         ; 401501 u=dl.1       d=pf.1
2. 7 sets   dl.1, sf.1               ; 401501 u=dl.1       d=sf.1
2. 8 jcnd   zf.1, $loc_401517        ; 401504 u=zf.1
```

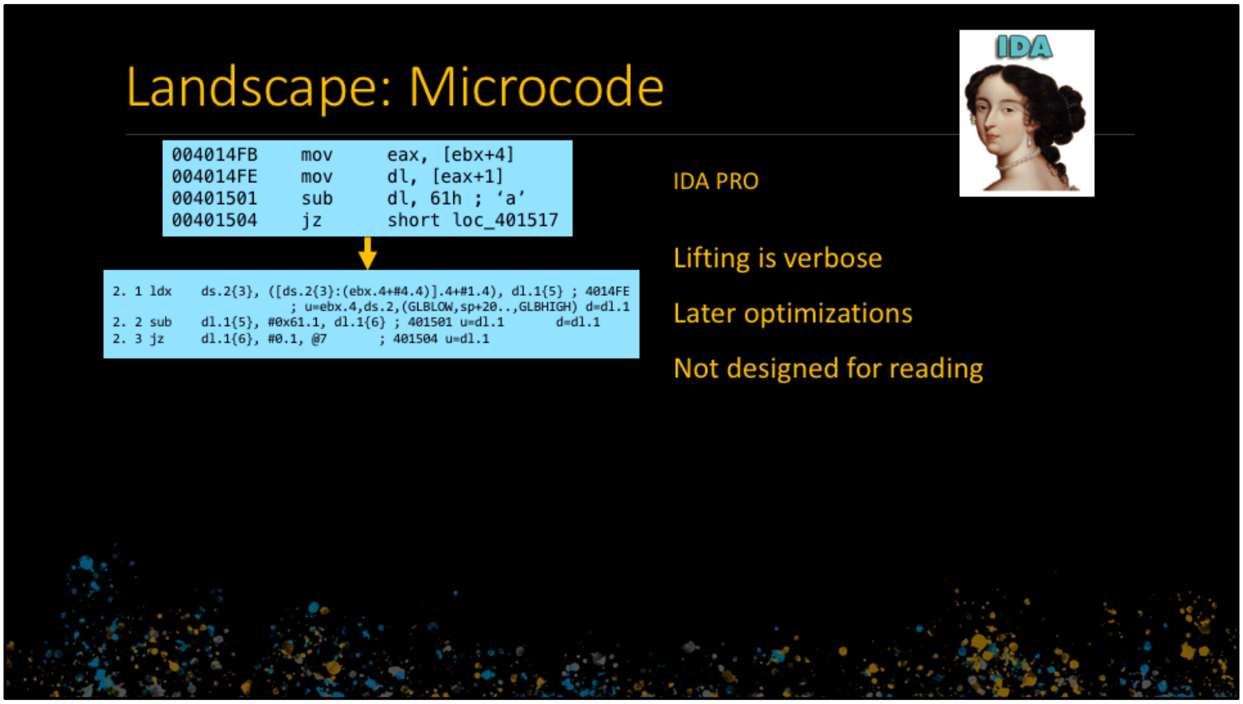
IDA PRO

Lifting is verbose

Later optimizations

Not designed for reading

Credit: Ilfak Guilfanov: https://hex-rays.com/products/ida/support/ppt/recon2018.ppt



Landscape: Microcode

```
004014FB    mov     eax, [ebx+4]
004014FE    mov     dl, [eax+1]
00401501    sub     dl, 61h ; ‘a’
00401504    jz      short loc_401517
```

```
2. 1 ldx    ds.2{3}, ([ds.2{3}:(ebx.4+#4.4)].4+#1.4), dl.1{5} ; 4014FE
                  ; u=ebx.4,ds.2,(GLBLOW,sp+20..,GLBHIGH) d=dl.1
2. 2 sub    dl.1{5}, #0x61.1, dl.1{6} ; 401501 u=dl.1       d=dl.1
2. 3 jz     dl.1{6}, #0.1, @7         ; 401504 u=dl.1
```

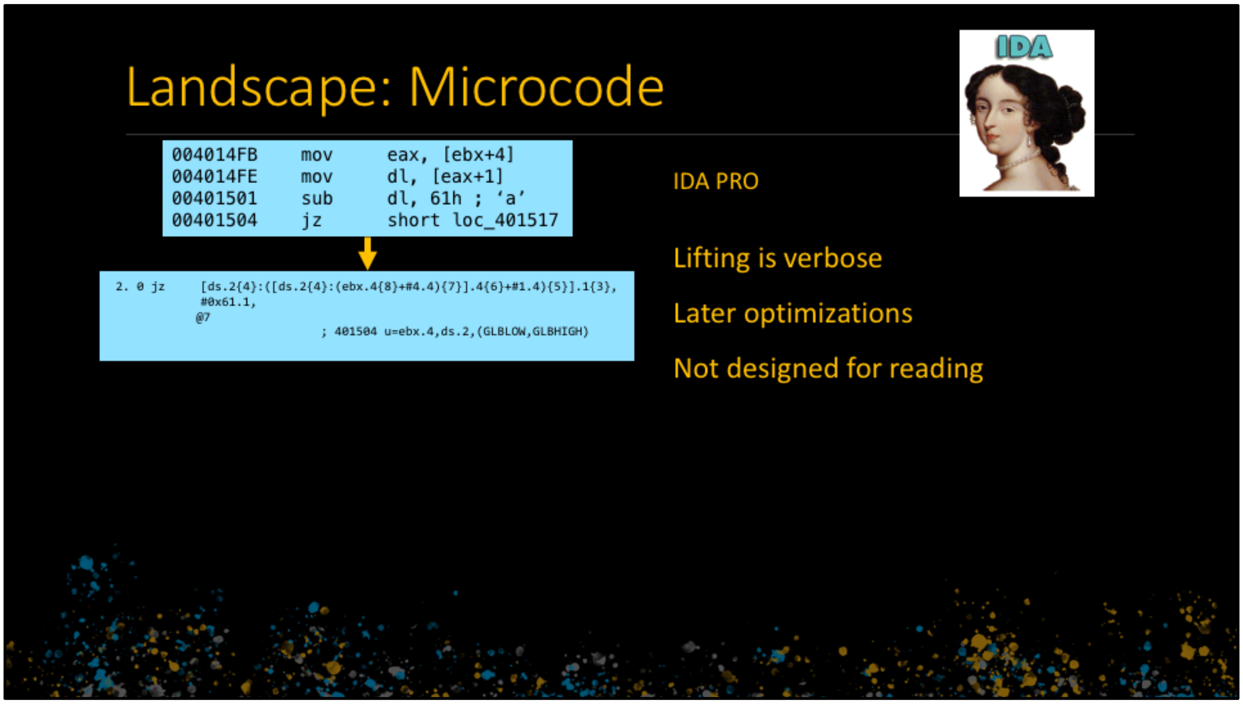
IDA PRO

Lifting is verbose

Later optimizations

Not designed for reading

Credit: Ilfak Guilfanov: https://hex-rays.com/products/ida/support/ppt/recon2018.ppt



Landscape: Microcode

```
004014FB    mov      eax, [ebx+4]
004014FE    mov      dl, [eax+1]
00401501    sub      dl, 61h ; ‘a’
00401504    jz       short loc_401517
```

```
 2. 0 jz     [ds.2{4}:([ds.2{4}:(ebx.4{8}+#4.4){7}].4{6}+#1.4){5}].1{3},
             #0x61.1,
            @7
                             ; 401504 u=ebx.4,ds.2,(GLBLOW,GLBHIGH)
```

IDA PRO

Lifting is verbose

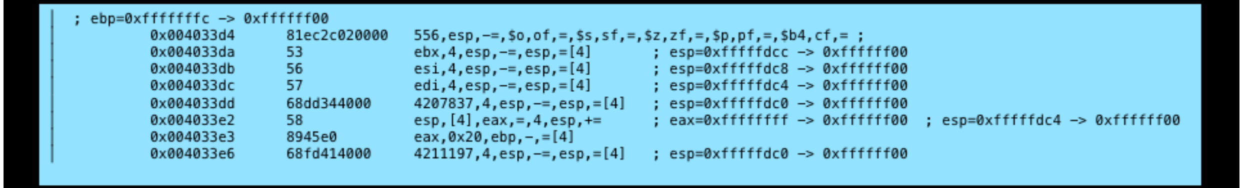
Later optimizations

Not designed for reading

Credit: Ilfak Guilfanov: https://hex-rays.com/products/ida/support/ppt/recon2018.ppt

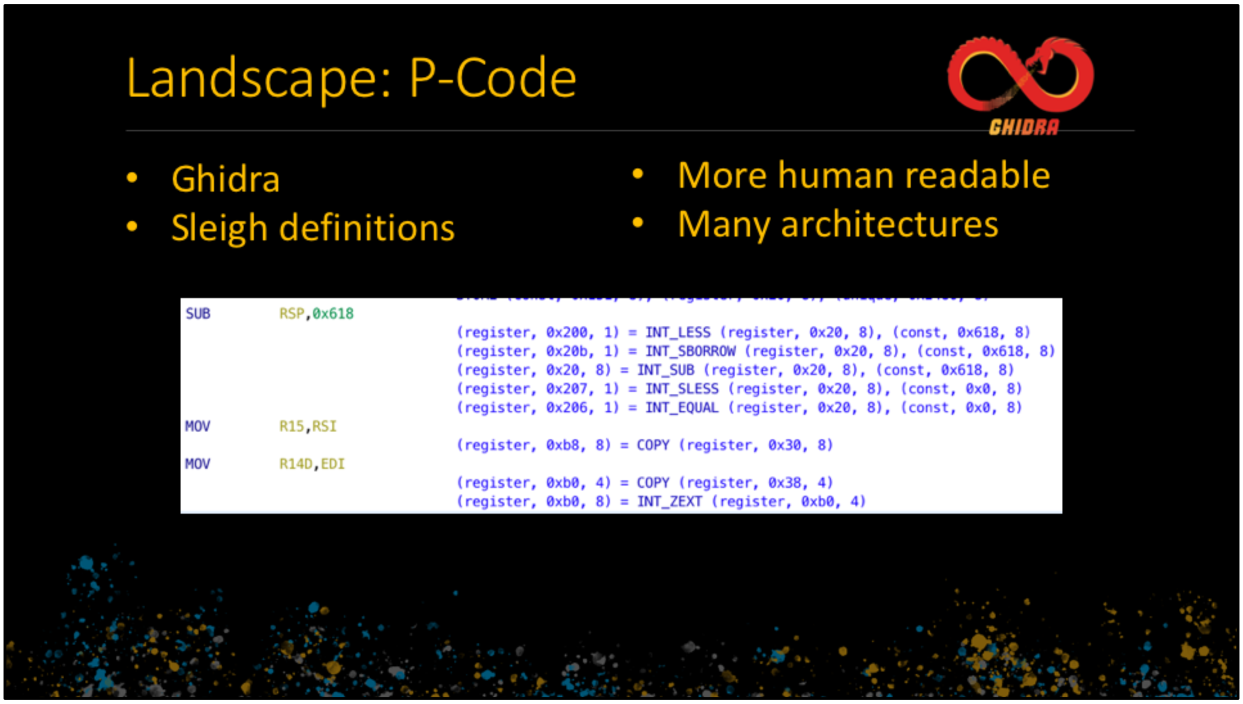
Landscape: ESIL

- Radare
- String based
- Post-fix notation
- Concise



```
; ebp=0xfffffffc -> 0xffffff00
        0x004033d4      81ec2c020000   556,esp,-=,$o,of,=,$s,sf,=,$z,zf,=,$p,pf,=,$b4,cf,= ;
        0x004033da      53             ebx,4,esp,-=,esp,=[4]        ; esp=0xfffffdcc -> 0xffffff00
        0x004033db      56             esi,4,esp,-=,esp,=[4]        ; esp=0xfffffdc8 -> 0xffffff00
        0x004033dc      57             edi,4,esp,-=,esp,=[4]        ; esp=0xfffffdc4 -> 0xffffff00
        0x004033dd      68dd344000     4207837,4,esp,-=,esp,=[4]    ; esp=0xfffffdc0 -> 0xffffff00
        0x004033e2      58             esp,[4],eax,=,4,esp,+=       ; eax=0xffffffff -> 0xffffff00  ; esp=0xfffffdc4 -> 0xffffff00
        0x004033e3      8945e0         eax,0x20,ebp,-,=[4]
        0x004033e6      68fd414000     4211197,4,esp,-=,esp,=[4]    ; esp=0xfffffdc0 -> 0xffffff00
```

Part of the rationale is that basically everything in radare is shuttled across a text-based interface, the API itself is just a pipe with text input and output, so this means that most things end up being textual which hurts efficiency though makes it much easier to use in a very unix-like way that is core to radare's architecture where anything can be piped into anything else.

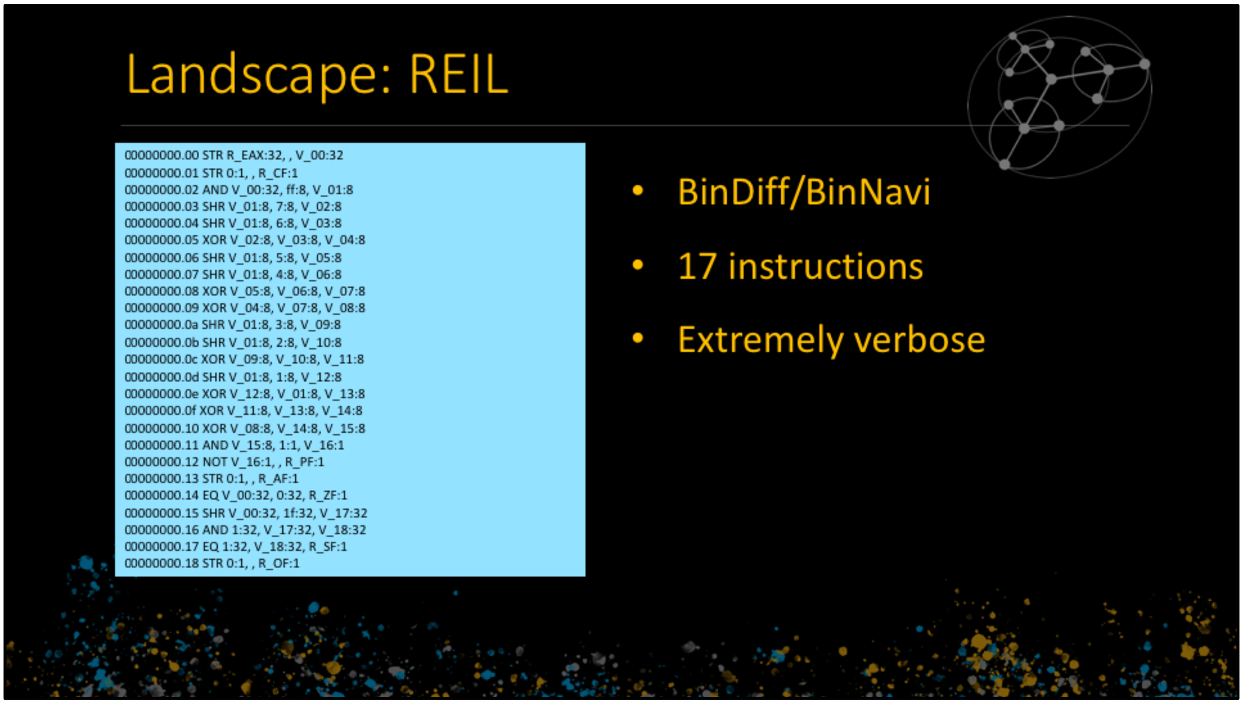


Landscape: P-Code

- Ghidra
- Sleigh definitions
- More human readable
- Many architectures

```
SUB         RSP,0x618
                                (register, 0x200, 1) = INT_LESS (register, 0x20, 8), (const, 0x618, 8)
                                (register, 0x20b, 1) = INT_SBORROW (register, 0x20, 8), (const, 0x618, 8)
                                (register, 0x20, 8) = INT_SUB (register, 0x20, 8), (const, 0x618, 8)
                                (register, 0x207, 1) = INT_SLESS (register, 0x20, 8), (const, 0x0, 8)
                                (register, 0x206, 1) = INT_EQUAL (register, 0x20, 8), (const, 0x0, 8)
MOV         R15,RSI
                                (register, 0xb8, 8) = COPY (register, 0x30, 8)
MOV         R14D,EDI
                                (register, 0xb0, 4) = COPY (register, 0x38, 4)
                                (register, 0xb0, 8) = INT_ZEXT (register, 0xb0, 4)
```

P-Code is both a generic term used as short-hand for “portable code machine” and used across many systems as well as the specific name of Ghidra’s intermediate language.



Landscape: REIL

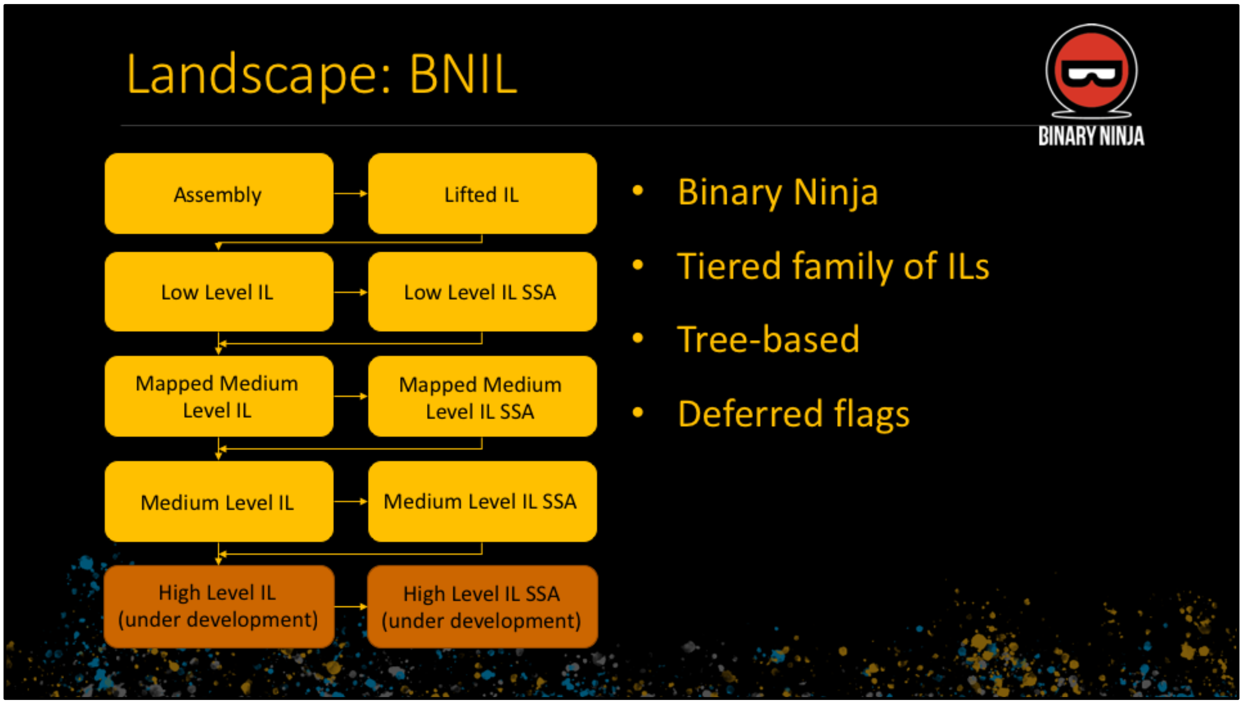
```
00000000.00 STR R_EAX:32, , V_00:32
00000000.01 STR 0:1, , R_CF:1
00000000.02 AND V_00:32, ff:8, V_01:8
00000000.03 SHR V_01:8, 7:8, V_02:8
00000000.04 SHR V_01:8, 6:8, V_03:8
00000000.05 XOR V_02:8, V_03:8, V_04:8
00000000.06 SHR V_01:8, 5:8, V_05:8
00000000.07 SHR V_01:8, 4:8, V_06:8
00000000.08 XOR V_05:8, V_06:8, V_07:8
00000000.09 XOR V_04:8, V_07:8, V_08:8
00000000.0a SHR V_01:8, 3:8, V_09:8
00000000.0b SHR V_01:8, 2:8, V_10:8
00000000.0c XOR V_09:8, V_10:8, V_11:8
00000000.0d SHR V_01:8, 1:8, V_12:8
00000000.0e XOR V_12:8, V_01:8, V_13:8
00000000.0f XOR V_11:8, V_13:8, V_14:8
00000000.10 XOR V_08:8, V_14:8, V_15:8
00000000.11 AND V_15:8, 1:1, V_16:1
00000000.12 NOT V_16:1, , R_PF:1
00000000.13 STR 0:1, , R_AF:1
00000000.14 EQ V_00:32, 0:32, R_ZF:1
00000000.15 SHR V_00:32, 1f:32, V_17:32
00000000.16 AND 1:32, V_17:32, V_18:32
00000000.17 EQ 1:32, V_18:32, R_SF:1
00000000.18 STR 0:1, , R_OF:1
```

- BinDiff/BinNavi
- 17 instructions
- Extremely verbose

REIL from Zynamics powers Bindiff and Binnavi

Didn’t need to recover types at all, was purpose built.

https://www.zynamics.com/downloads/csw09.pdf
https://github.com/Cr4sh/openreil



Landscape: BNIL

BINARY NINJA

Assembly → Lifted IL

Low Level IL → Low Level IL SSA

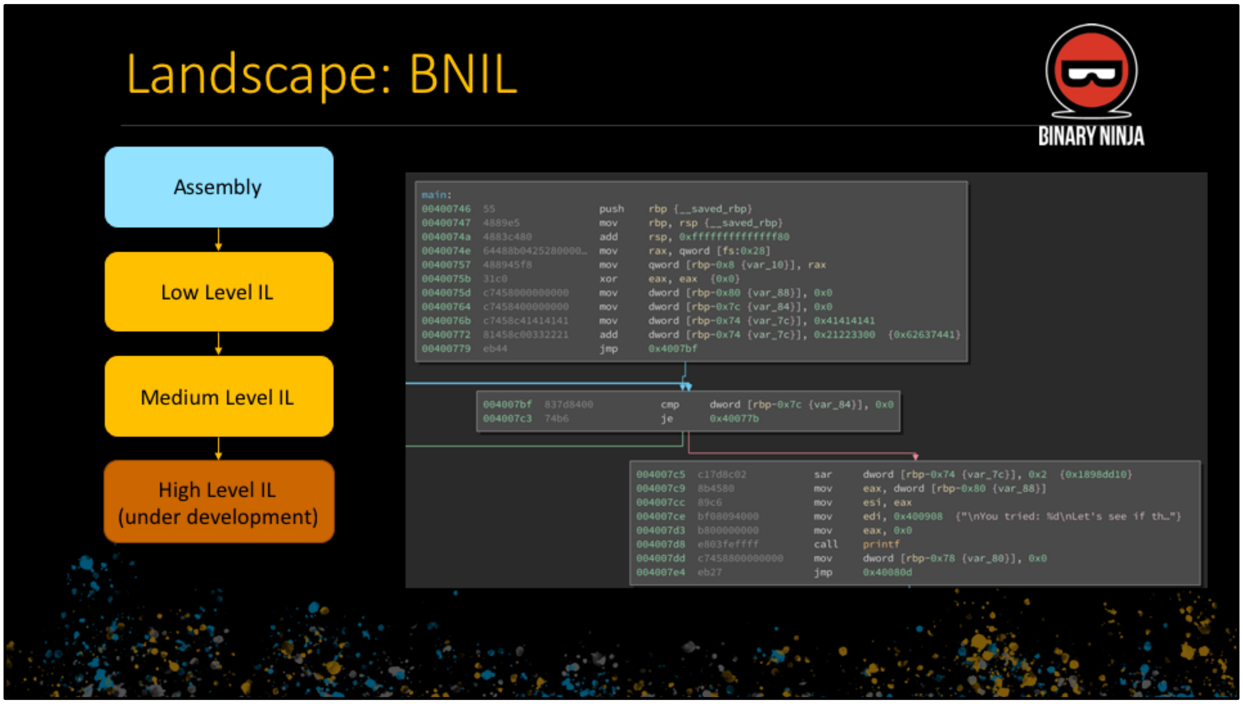
Mapped Medium Level IL → Mapped Medium Level IL SSA

Medium Level IL → Medium Level IL SSA

High Level IL (under development) → High Level IL SSA (under development)

- Binary Ninja
- Tiered family of ILs
- Tree-based
- Deferred flags

http://docs.binary.ninja/dev/bnil-llil.html
https://vimeo.com/215511922



Landscape: BNIL

Assembly

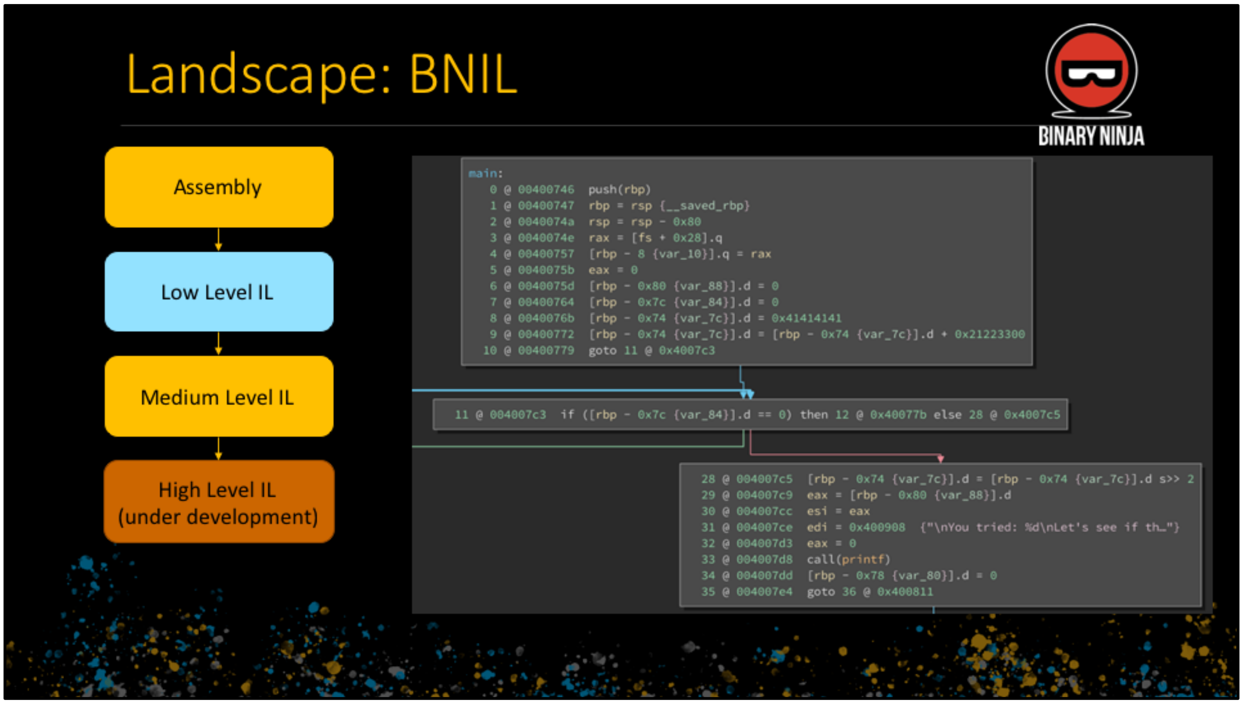
Low Level IL

Medium Level IL

High Level IL
(under development)

```
main:
00400746  55                   push    rbp {__saved_rbp}
00400747  4889e5               mov     rbp, rsp {__saved_rbp}
0040074a  4883c480             add     rsp, 0xffffffffffffff80
0040074e  64488b0425280000…    mov     rax, qword [fs:0x28]
00400757  488945f8             mov     qword [rbp-0x8 {var_10}], rax
0040075b  31c0                 xor     eax, eax  {0x0}
0040075d  c7458000000000       mov     dword [rbp-0x80 {var_88}], 0x0
00400764  c7458400000000       mov     dword [rbp-0x7c {var_84}], 0x0
0040076b  c7458c41414141       mov     dword [rbp-0x74 {var_7c}], 0x41414141
00400772  81458c00332221       add     dword [rbp-0x74 {var_7c}], 0x21223300  {0x62637441}
00400779  eb44                 jmp     0x4007bf

004007bf  837d8400             cmp     dword [rbp-0x7c {var_84}], 0x0
004007c3  74b6                 je      0x40077b

004007c5  c17d8c02             sar     dword [rbp-0x74 {var_7c}], 0x2  {0x1898dd10}
004007c9  8b4580               mov     eax, dword [rbp-0x80 {var_88}]
004007cc  89c6                 mov     esi, eax
004007ce  bf08094000           mov     edi, 0x400908  {"\nYou tried: %d\nLet's see if th…"}
004007d3  b800000000           mov     eax, 0x0
004007d8  e803feffff           call    printf
004007dd  c7458800000000       mov     dword [rbp-0x78 {var_80}], 0x0
004007e4  eb27                 jmp     0x40080d
```

http://docs.binary.ninja/dev/bnil-llil.html
https://vimeo.com/215511922



Landscape: BNIL

Assembly

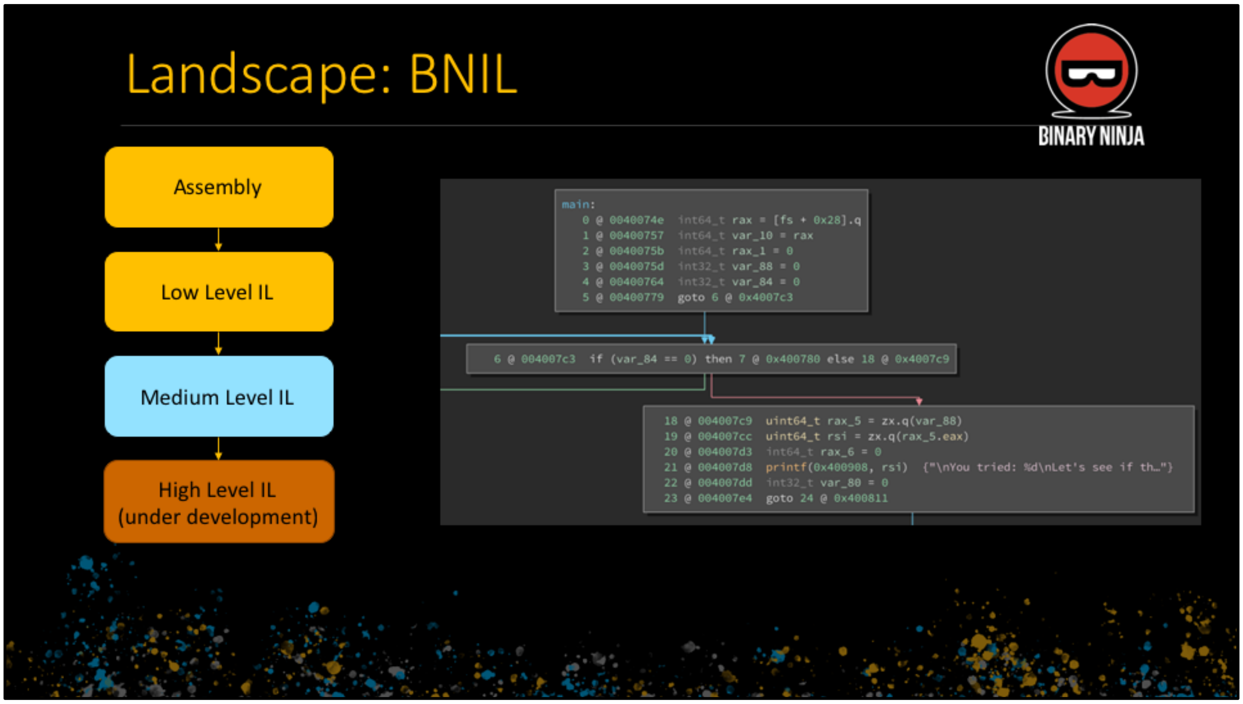
Low Level IL

Medium Level IL

High Level IL
(under development)

```
main:
   0 @ 00400746  push(rbp)
   1 @ 00400747  rbp = rsp {__saved_rbp}
   2 @ 0040074a  rsp = rsp - 0x80
   3 @ 0040074e  rax = [fs + 0x28].q
   4 @ 00400757  [rbp - 8 {var_10}].q = rax
   5 @ 0040075b  eax = 0
   6 @ 0040075d  [rbp - 0x80 {var_88}].d = 0
   7 @ 00400764  [rbp - 0x7c {var_84}].d = 0
   8 @ 0040076b  [rbp - 0x74 {var_7c}].d = 0x41414141
   9 @ 00400772  [rbp - 0x74 {var_7c}].d = [rbp - 0x74 {var_7c}].d + 0x21223300
  10 @ 00400779  goto 11 @ 0x4007c3

  11 @ 004007c3  if ([rbp - 0x7c {var_84}].d == 0) then 12 @ 0x40077b else 28 @ 0x4007c5

  28 @ 004007c5  [rbp - 0x74 {var_7c}].d = [rbp - 0x74 {var_7c}].d s>> 2
  29 @ 004007c9  eax = [rbp - 0x80 {var_88}].d
  30 @ 004007cc  esi = eax
  31 @ 004007ce  edi = 0x400908  {"\nYou tried: %d\nLet's see if th…"}
  32 @ 004007d3  eax = 0
  33 @ 004007d8  call(printf)
  34 @ 004007dd  [rbp - 0x78 {var_80}].d = 0
  35 @ 004007e4  goto 36 @ 0x400811
```

http://docs.binary.ninja/dev/bnil-llil.html
https://vimeo.com/215511922



Landscape: BNIL

BINARY NINJA

Assembly

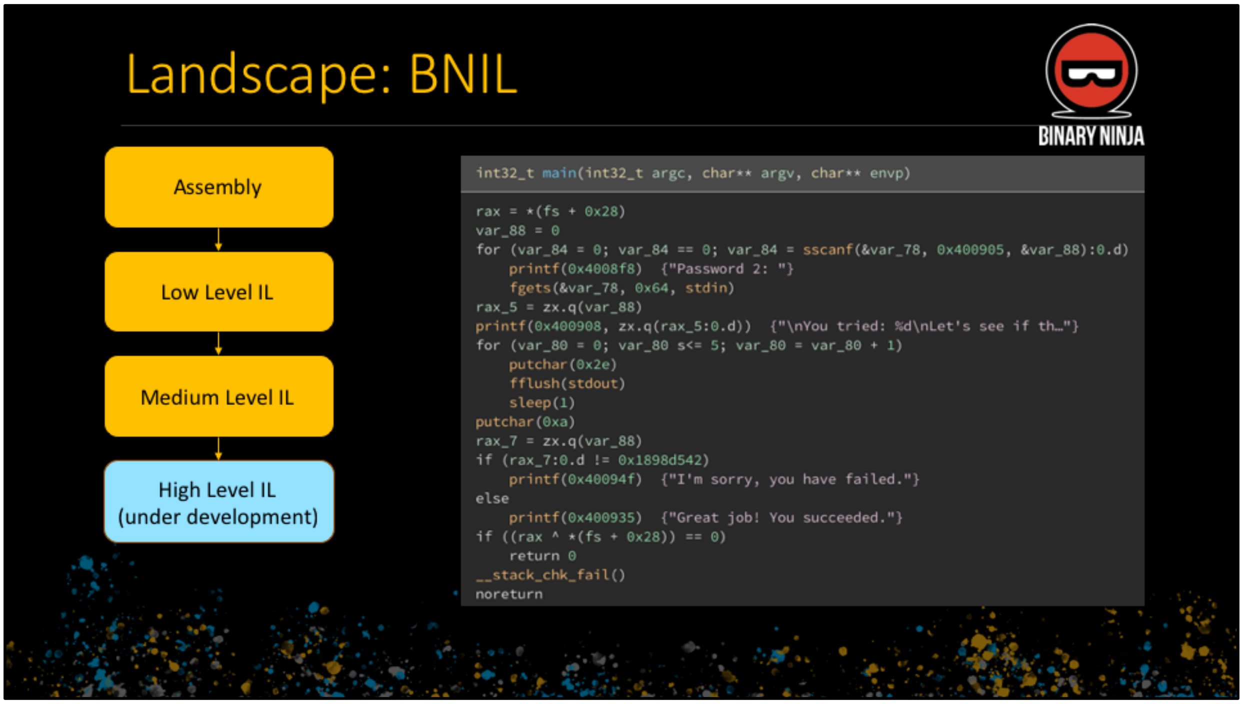
Low Level IL

Medium Level IL

High Level IL
(under development)

```
main:
  0 @ 0040074e  int64_t rax = [fs + 0x28].q
  1 @ 00400757  int64_t var_10 = rax
  2 @ 0040075b  int64_t rax_1 = 0
  3 @ 0040075d  int32_t var_88 = 0
  4 @ 00400764  int32_t var_84 = 0
  5 @ 00400779  goto 6 @ 0x4007c3

  6 @ 004007c3  if (var_84 == 0) then 7 @ 0x400780 else 18 @ 0x4007c9

 18 @ 004007c9  uint64_t rax_5 = zx.q(var_88)
 19 @ 004007cc  uint64_t rsi = zx.q(rax_5.eax)
 20 @ 004007d3  int64_t rax_6 = 0
 21 @ 004007d8  printf(0x400908, rsi)  {"\nYou tried: %d\nLet's see if th…"}
 22 @ 004007dd  int32_t var_80 = 0
 23 @ 004007e4  goto 24 @ 0x400811
```

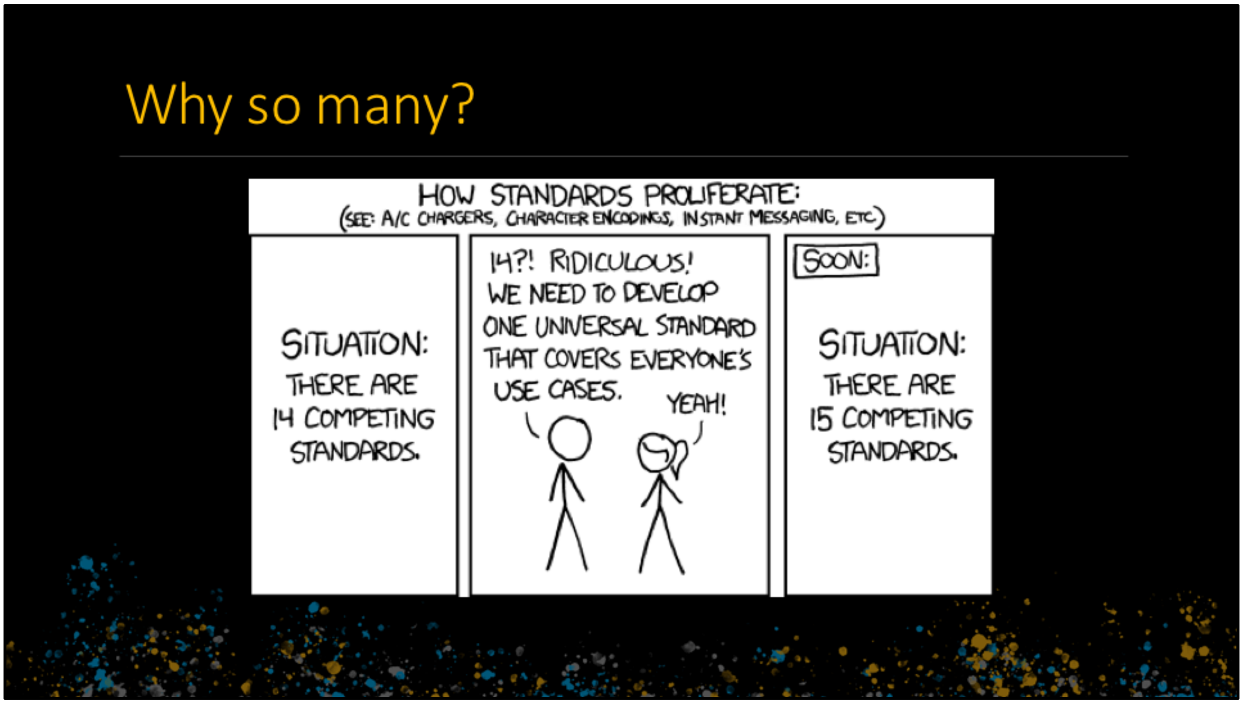
http://docs.binary.ninja/dev/bnil-llil.html
https://vimeo.com/215511922



Landscape: BNIL

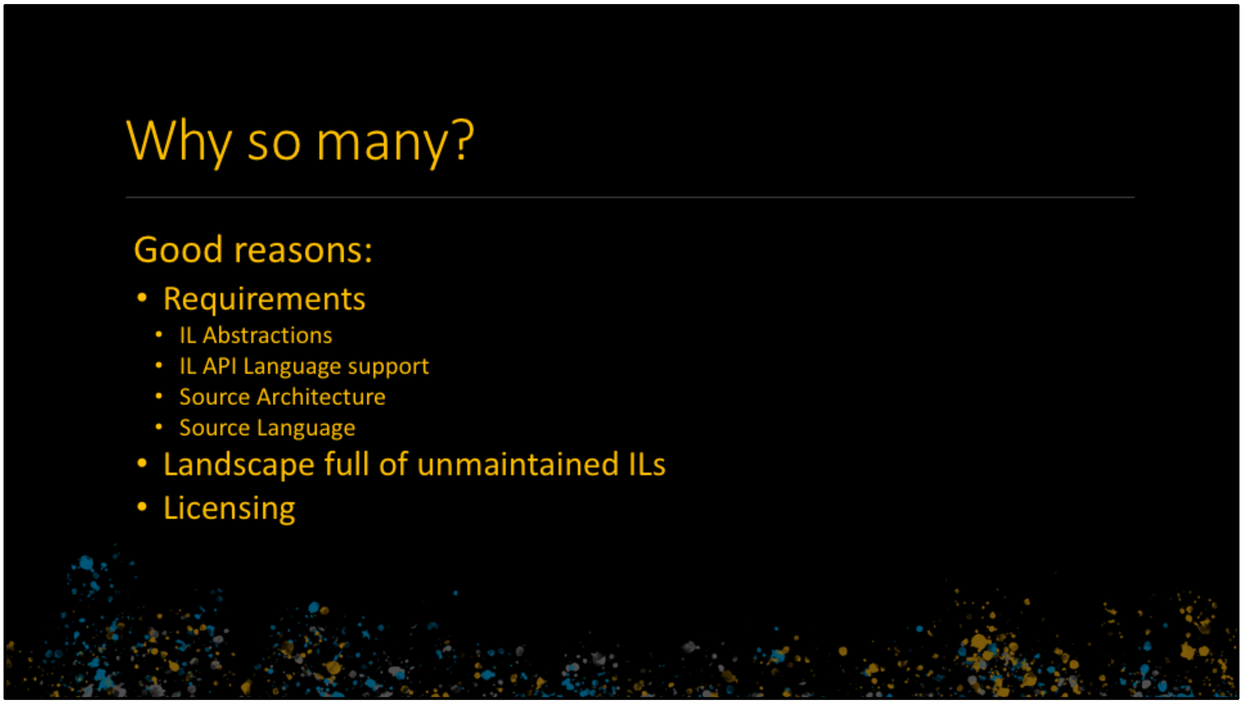
```
int32_t main(int32_t argc, char** argv, char** envp)

rax = *(fs + 0x28)
var_88 = 0
for (var_84 = 0; var_84 == 0; var_84 = sscanf(&var_78, 0x400905, &var_88):0.d)
    printf(0x4008f8)  {"Password 2: "}
    fgets(&var_78, 0x64, stdin)
rax_5 = zx.q(var_88)
printf(0x400908, zx.q(rax_5:0.d))  {"\nYou tried: %d\nLet's see if th…"}
for (var_80 = 0; var_80 s<= 5; var_80 = var_80 + 1)
    putchar(0x2e)
    fflush(stdout)
    sleep(1)
putchar(0xa)
rax_7 = zx.q(var_88)
if (rax_7:0.d != 0x1898d542)
    printf(0x40094f)  {"I'm sorry, you have failed."}
else
    printf(0x400935)  {"Great job! You succeeded."}
if ((rax ^ *(fs + 0x28)) == 0)
    return 0
__stack_chk_fail()
noreturn
```

http://docs.binary.ninja/dev/bnil-llil.html
https://vimeo.com/215511922



Why so many?

HOW STANDARDS PROLIFERATE:
(SEE: A/C CHARGERS, CHARACTER ENCODINGS, INSTANT MESSAGING, ETC.)
SITUATION: THERE ARE 14 COMPETING STANDARDS.
14?! RIDICULOUS! WE NEED TO DEVELOP ONE UNIVERSAL STANDARD THAT COVERS EVERYONE'S USE CASES.
YEAH!
SOON:
SITUATION: THERE ARE 15 COMPETING STANDARDS.



Why so many?

Good reasons:

- Requirements
  - IL Abstractions
  - IL API Language support
  - Source Architecture
  - Source Language
- Landscape full of unmaintained ILs
- Licensing

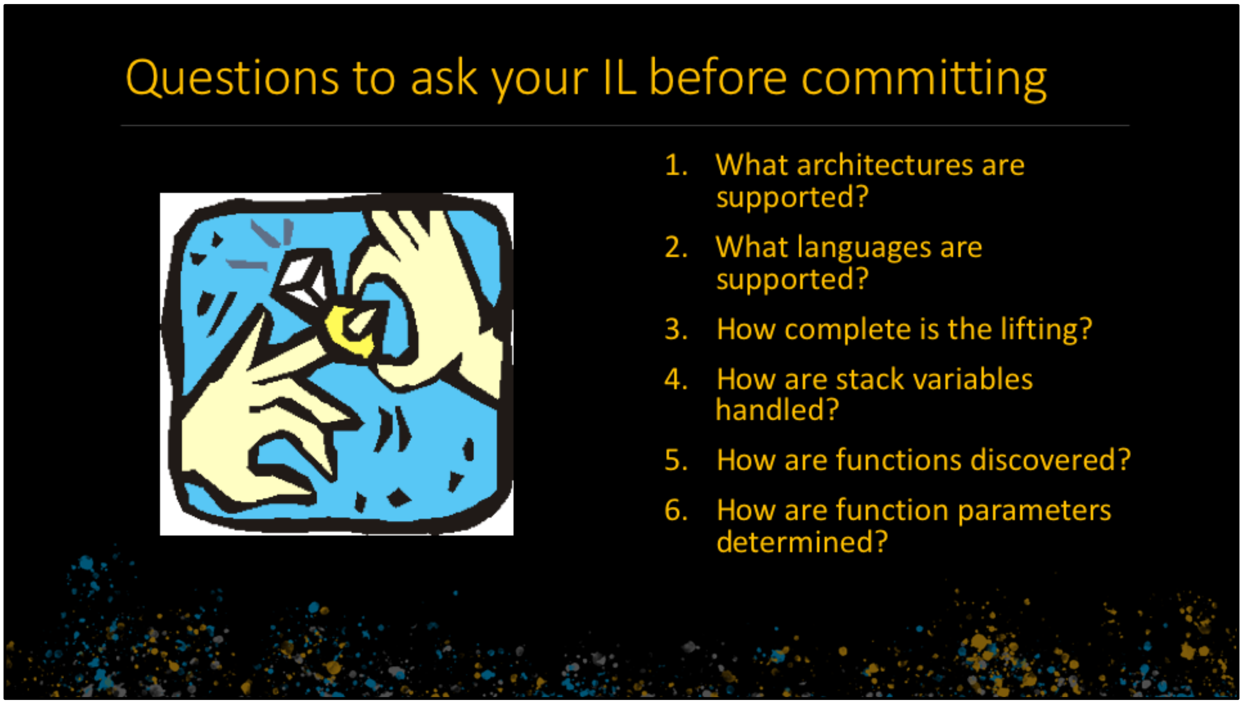
Different Purposes
Remember the tradeoffs above? Different ILs with different needs will choose one tradeoff versus another so it does make sense that there's a variety of ILs

- Ill suited for the task at hand. LLVM IR is great for going from Multiple source languages to binary but its too-close to source code to be a good choice for converting from binary. SSA and stack based means each lifter must understand the semantics of stack resolution and SSA generation making the extension to new architectures time consuming
- Binary and bitcode have very different semantics, thus different languages are needed for to ease initial translation
  - Binary uses address, implicit stack, no high-level control flow constructs, implicit parameter passing, etc
  - Bitcodes usually have explicit parameters, high-level control flow constructs, stack-based, sometimes SSA
    - This leads to very different sets of requirements for a target IL

Why so many?

Bad reasons:

- Not-Invented-Here
- Lack of awareness
- Publish or Perish

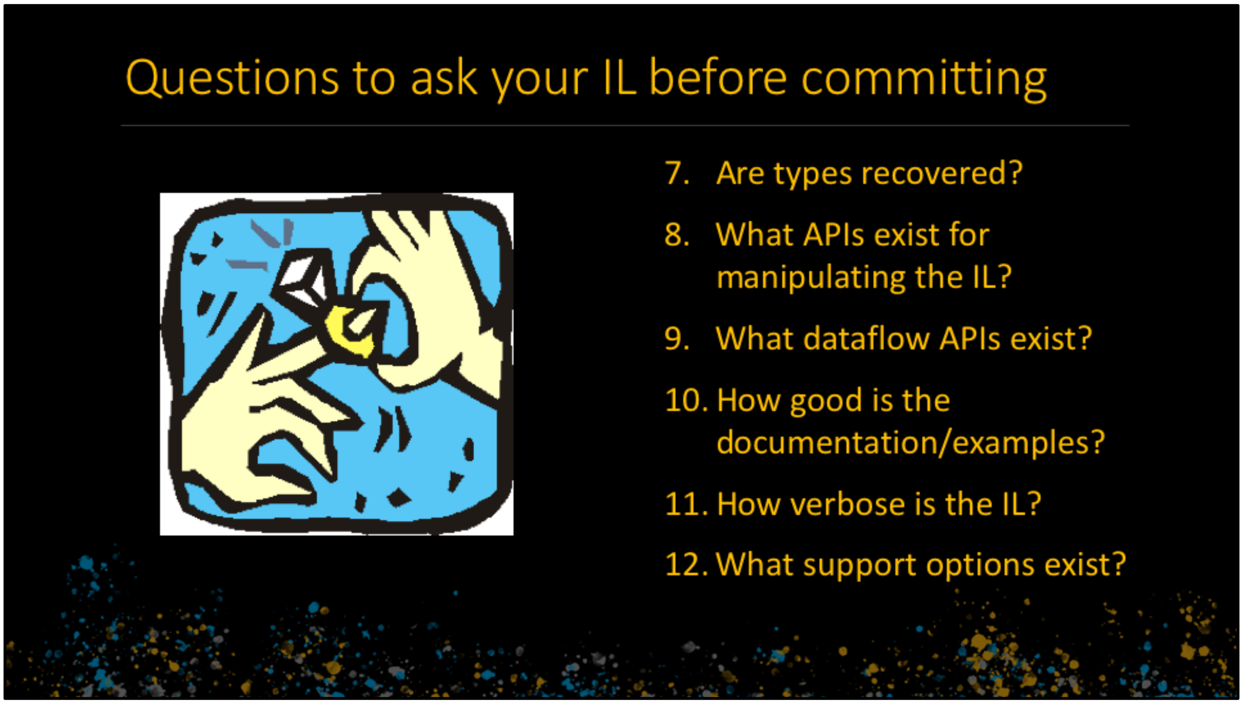


Questions to ask your IL before committing

1. What architectures are supported?
2. What languages are supported?
3. How complete is the lifting?
4. How are stack variables handled?
5. How are functions discovered?
6. How are function parameters determined?

You won’t believe number 10!

https://clipground.com/finger-ring-clipart.html



Questions to ask your IL before committing

7. Are types recovered?
8. What APIs exist for manipulating the IL?
9. What dataflow APIs exist?
10. How good is the documentation/examples?
11. How verbose is the IL?
12. What support options exist?

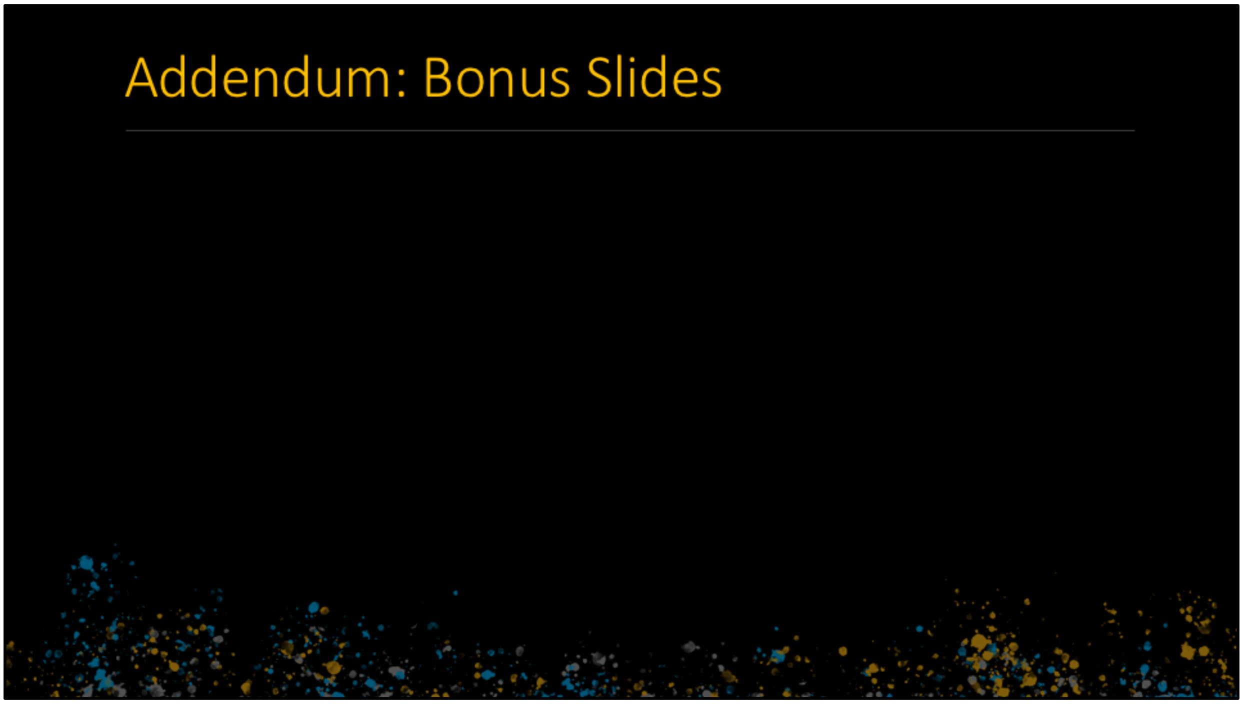
You won’t believe number 10!

https://clipground.com/finger-ring-clipart.html

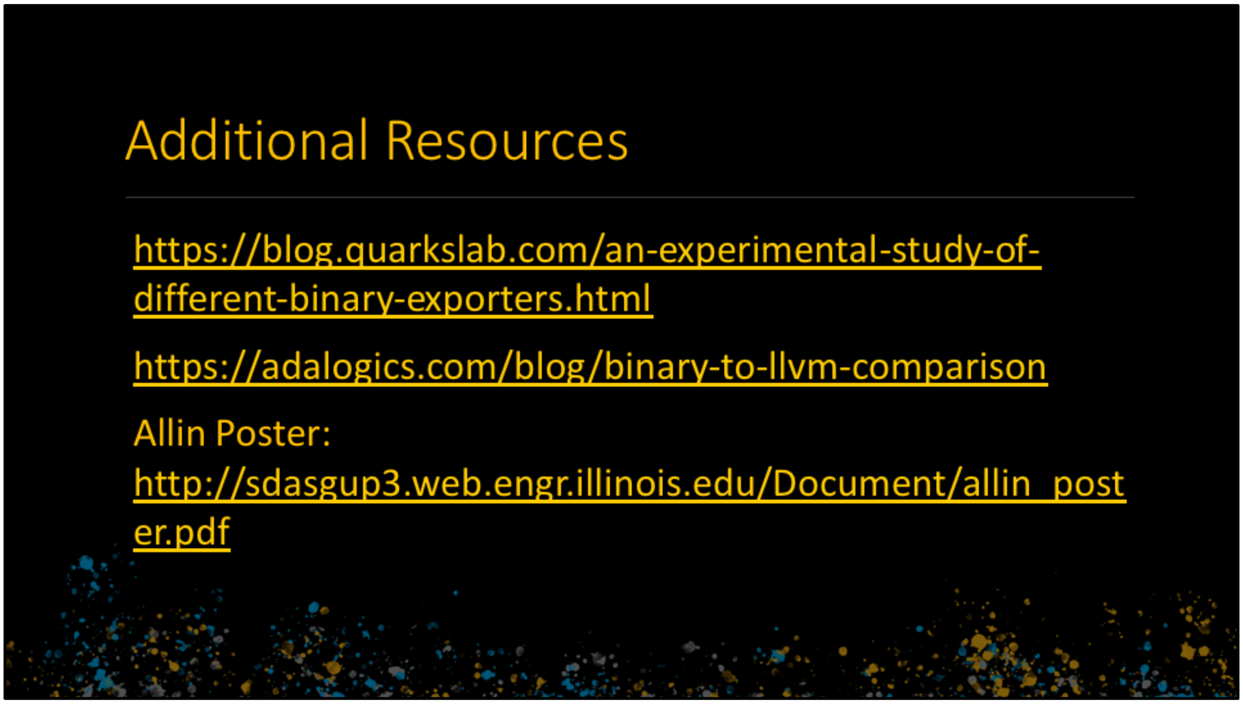
DEMOS

~~Questions?~~

NOT NOW, FIND US IN THE SPEAKER SPOT!



Addendum: Bonus Slides



Additional Resources

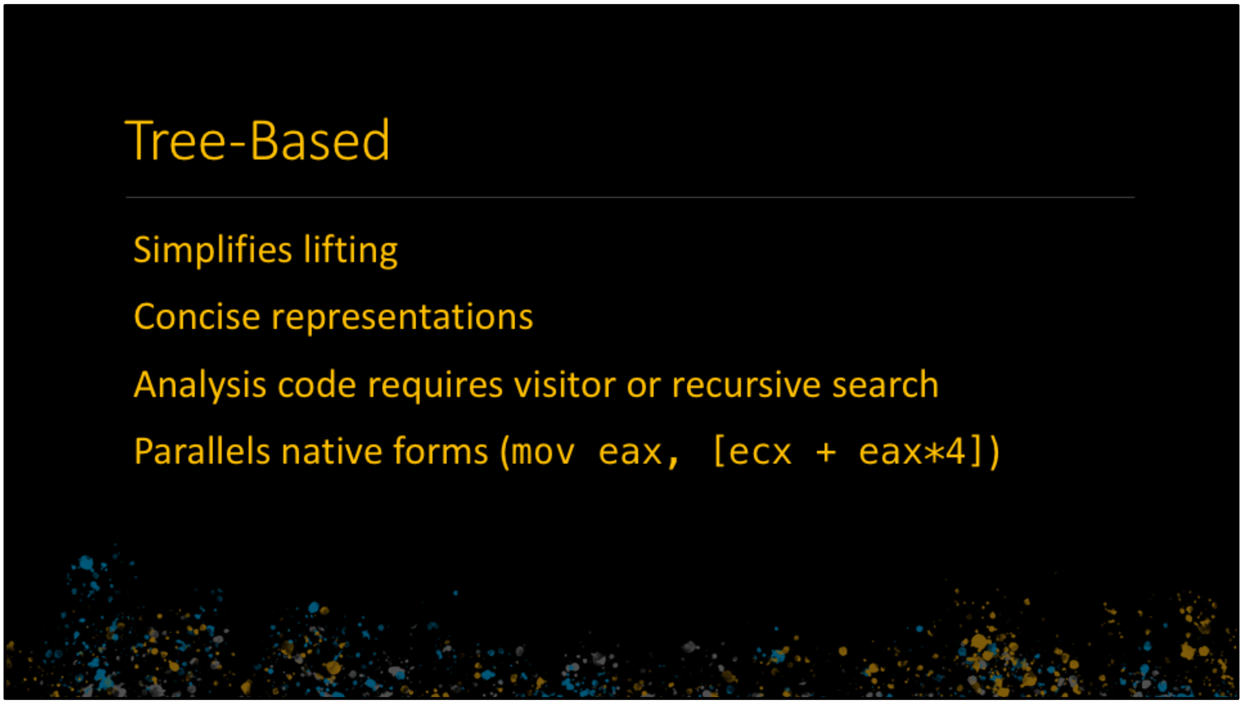
https://blog.quarkslab.com/an-experimental-study-of-different-binary-exporters.html

https://adalogics.com/blog/binary-to-llvm-comparison

Allin Poster: http://sdasgup3.web.engr.illinois.edu/Document/allin_poster.pdf

Working with ILs

GENERAL TECHNIQUES AND TIPS



Tree-Based

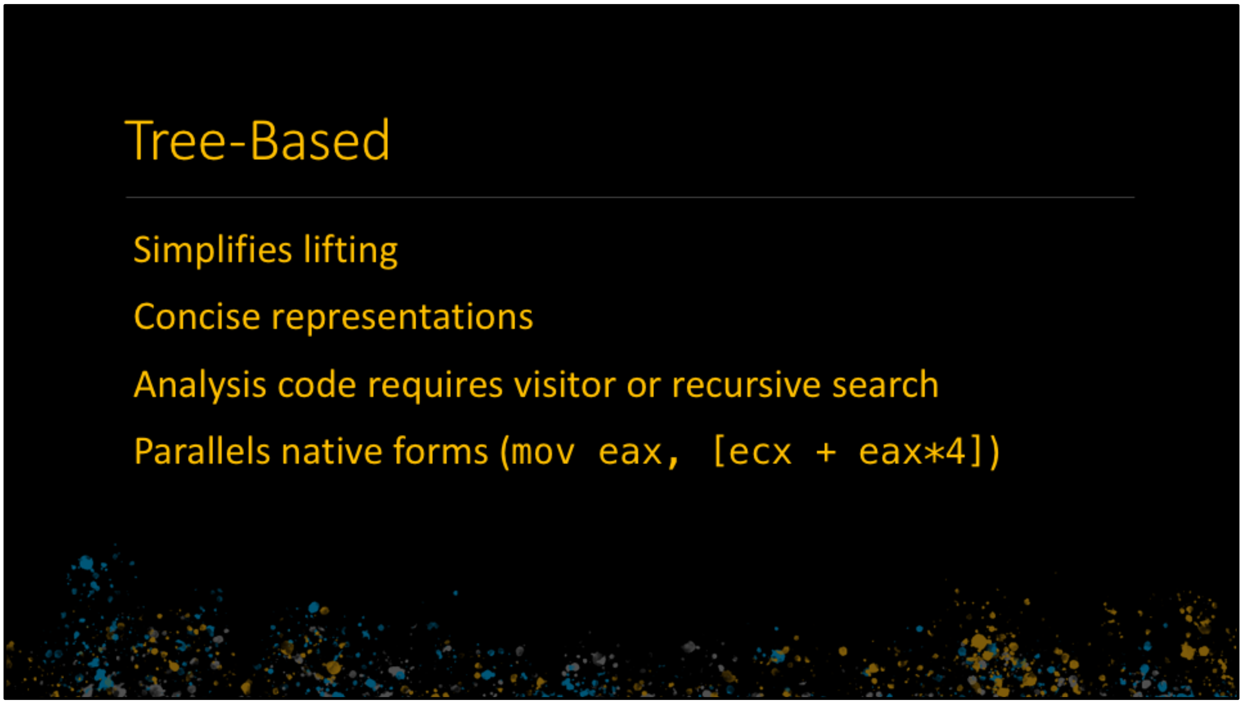
Simplifies lifting

Concise representations

Analysis code requires visitor or recursive search

Parallels native forms (`mov eax, [ecx + eax*4]`)

Adsfas fads fasdfs fd



Tree-Based

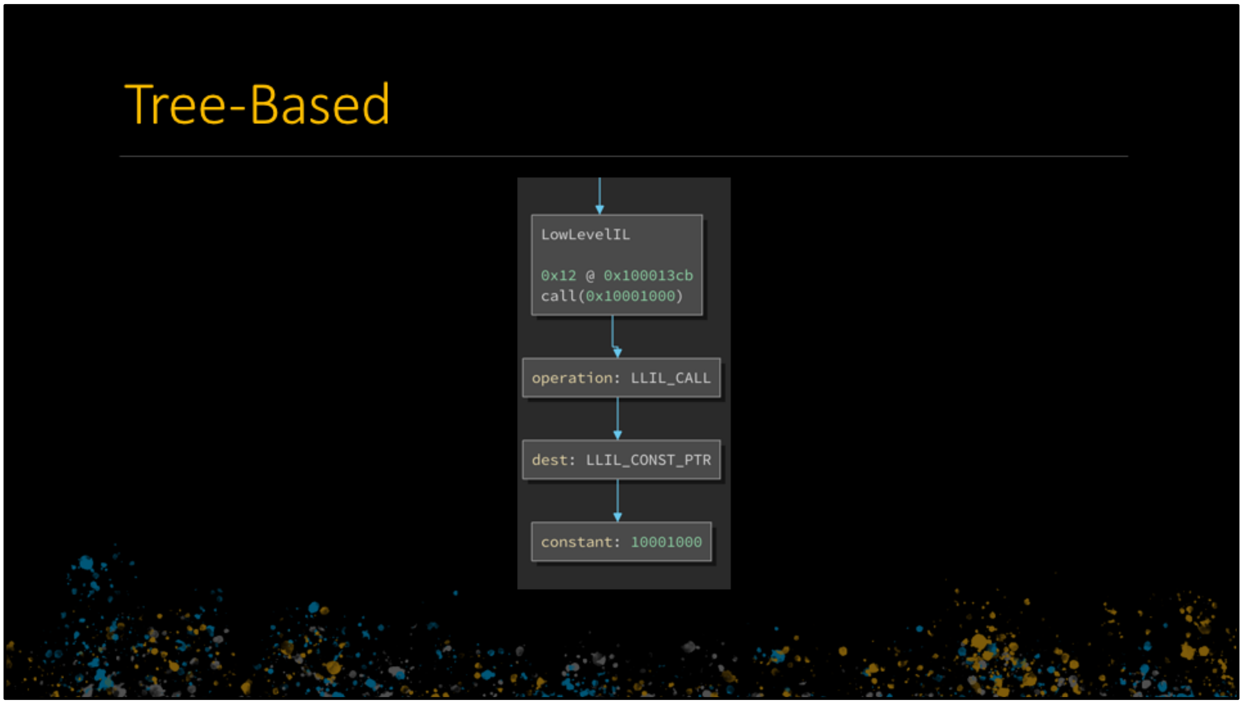
Simplifies lifting

Concise representations

Analysis code requires visitor or recursive search

Parallels native forms (`mov eax, [ecx + eax*4]`)

Adsfas fads fasdfs fd



Tree-Based

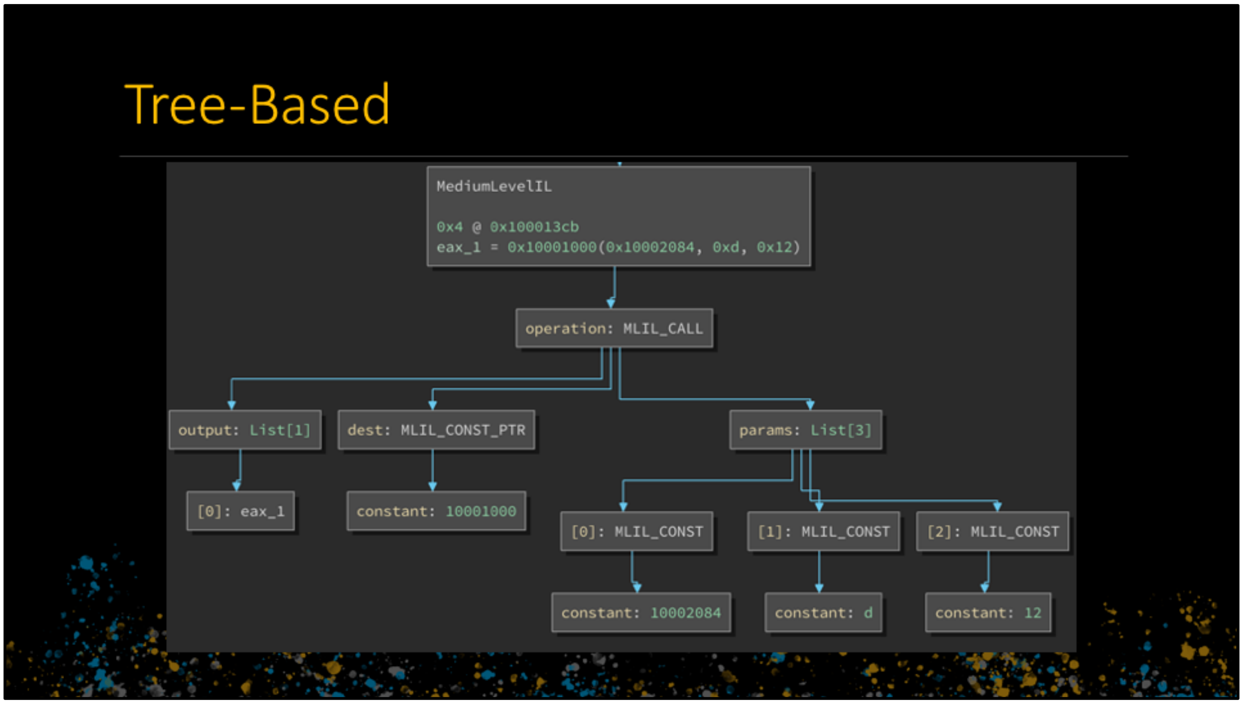
```
LowLevelIL

0x12 @ 0x100013cb
call(0x10001000)
```

```
operation: LLIL_CALL
```

```
dest: LLIL_CONST_PTR
```

```
constant: 10001000
```

LowLevelIL
“call(0x10001000)”
operation: LLIL_CALL

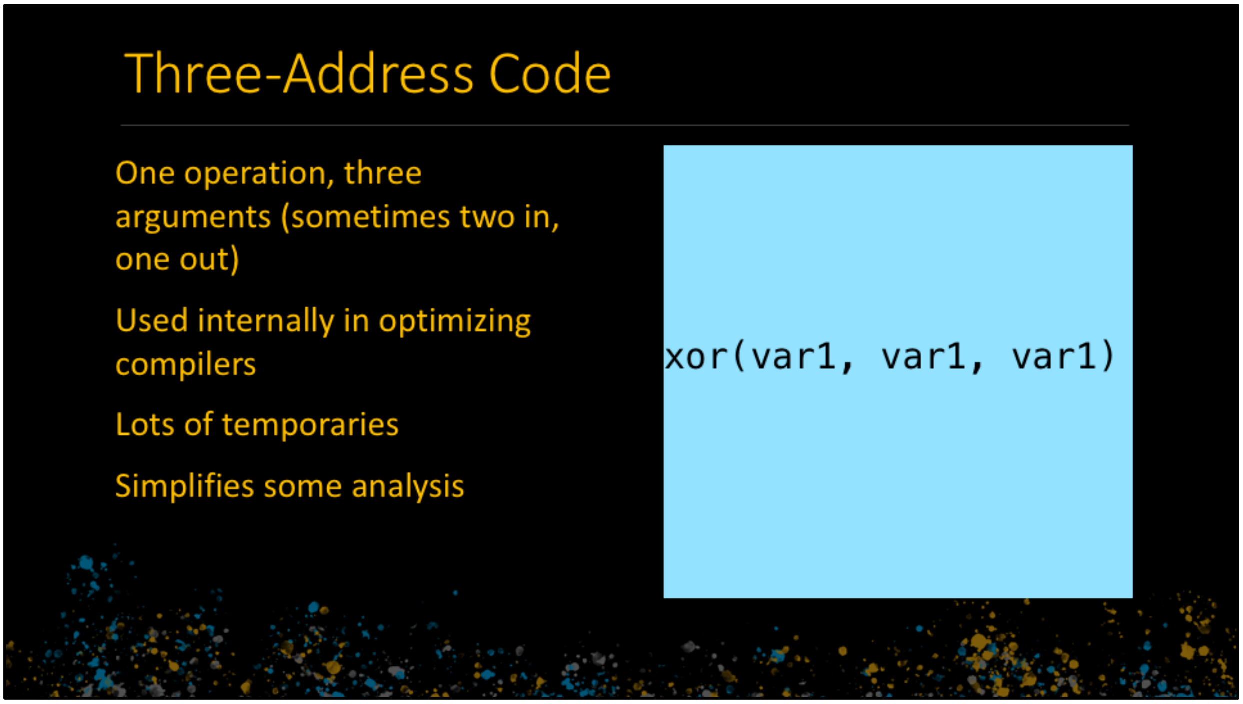


Tree-Based

```
MediumLevelIL

0x4 @ 0x100013cb
eax_1 = 0x10001000(0x10002084, 0xd, 0x12)
```

```
operation: MLIL_CALL
output: List[1]
[0]: eax_1
dest: MLIL_CONST_PTR
constant: 10001000
params: List[3]
[0]: MLIL_CONST
constant: 10002084
[1]: MLIL_CONST
constant: d
[2]: MLIL_CONST
constant: 12
```

TODO: better instruction with other architectures



Three-Address Code

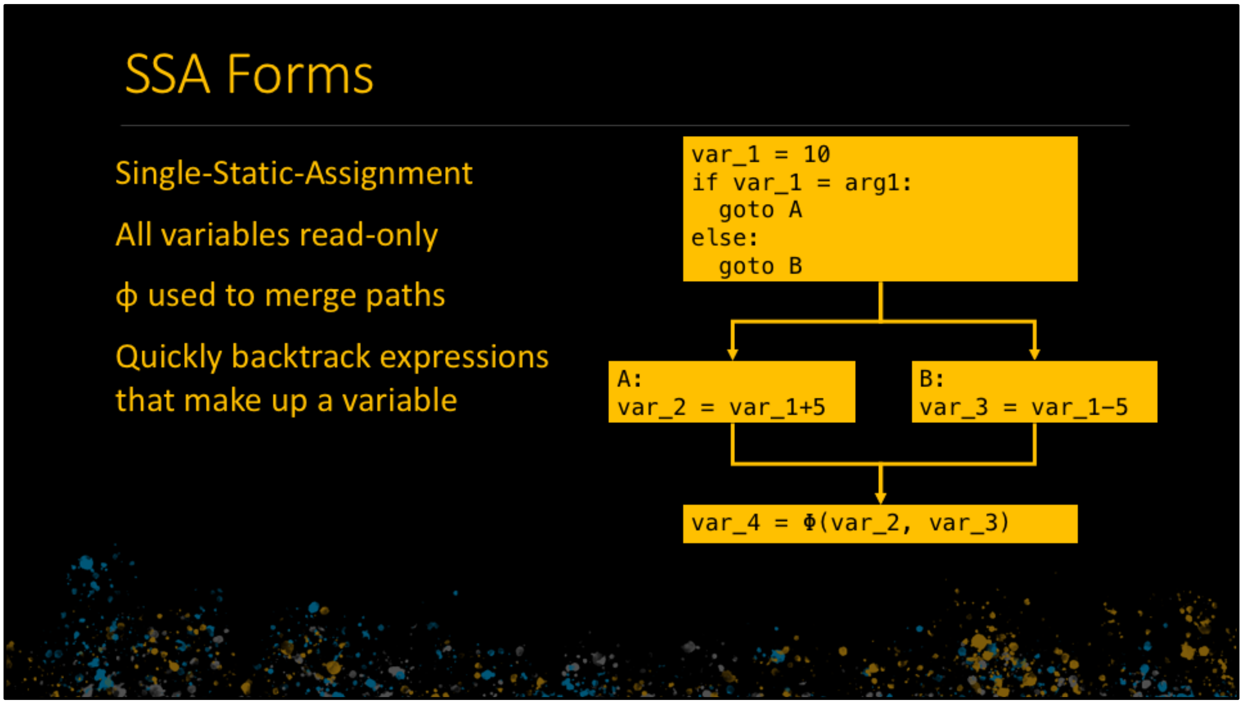
- One operation, three arguments (sometimes two in, one out)
- Used internally in optimizing compilers
- Lots of temporaries
- Simplifies some analysis

```
xor(var1, var1, var1)
```

Some analysis are much simpler – finding all add instructions for example is fast and easy.

But to find all adds that are a part of a pointer dereference means that you have to implement a dataflow system that can track through those temporary values.

TODO: code example



SSA Forms

Single-Static-Assignment

All variables read-only

ϕ used to merge paths

Quickly backtrack expressions that make up a variable

```
var_1 = 10
if var_1 = arg1:
  goto A
else:
  goto B
```

```
A:
var_2 = var_1+5
```

```
B:
var_3 = var_1-5
```

```
var_4 = Φ(var_2, var_3)
```

SSA forms can be one of the more intimidating features of intermediate languages but they're actually fairly simple and extremely useful. In short, in an SSA form, variables are immutable. You can create a variable with a value or from a change to a previous variable, but the previous variable never changes.

The only complication to this arises when you have multiple control flow paths that merge back. Because you do not have perfect knowledge of the path through a program during static analysis, a ϕ is introduced to indicate that a variable came from multiple sources. It's then up to whatever analysis is running across the SSA to determine what it wants to do with the fact that a variable comes from a ϕ.