Formally Verified Lifting of C-Compiled x86-64 Binaries

Freek Verbeek, Joshua Bockenek, Zhoulai Fu, Binoy Ravindran

Virginia Tech

Open University, The Netherlands

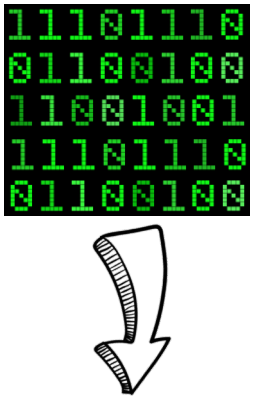
State University of New York, Korea

freek@vt.edu

PLDI'22

Disassembly

the start of binary analysis.



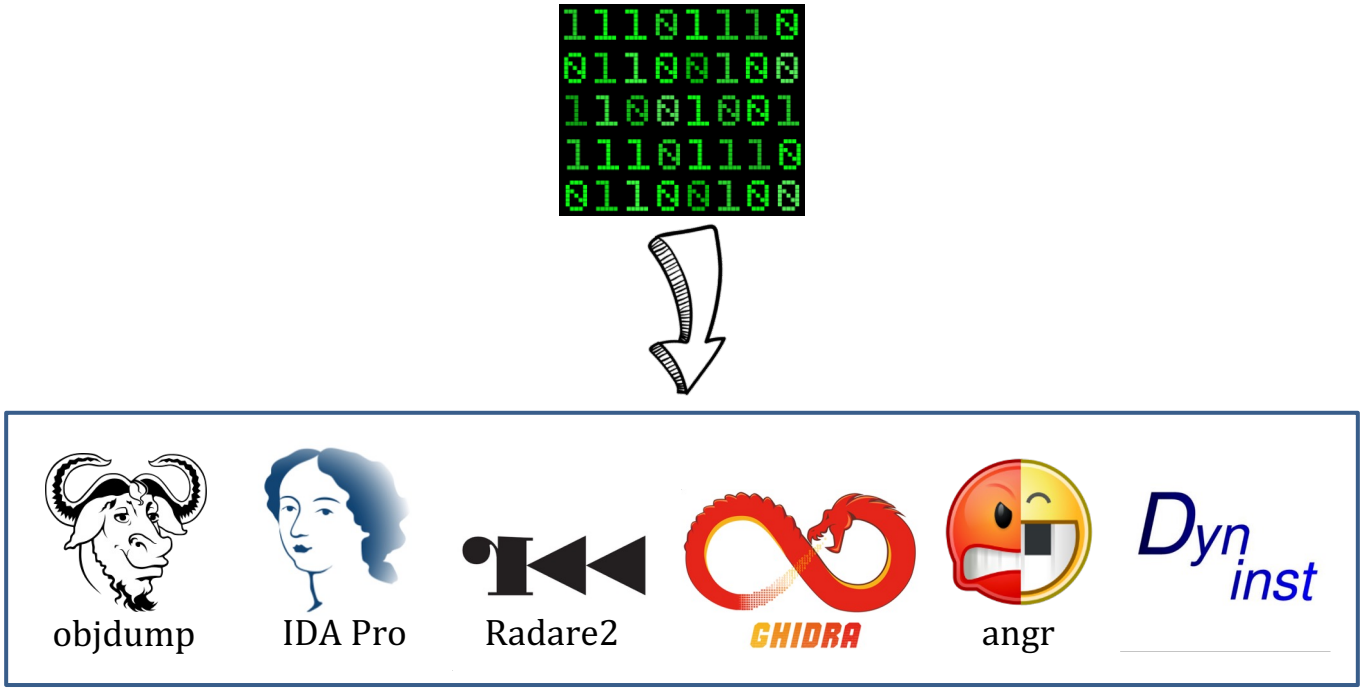

```
100000910: PUSH RBP
100000911: MOV RBP, RSP
100000914: PUSH RBX
100000915: PUSH RAX
100000916: CMP EDI, 2
100000919: JNE 0x10000094A
10000091b: MOV RDI, QWORD PTR [RSI + 8]
```

Disassembly

the start of binary analysis.

Disassembly

is an unsolved problem.



Problem Statement

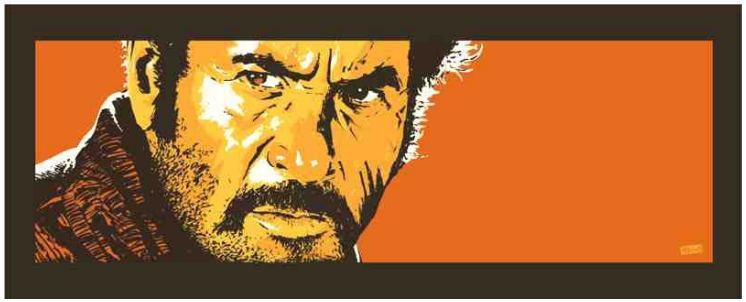
Can we build a formally verified disassembler?

GOOD

- A disassembler whose output can be **formally verified**.
- Applicable to COTS x86-64 binaries.
- No debugging information or source code required.

BAD

- It may **fail.**
- Only small parts of the binary may be reached.



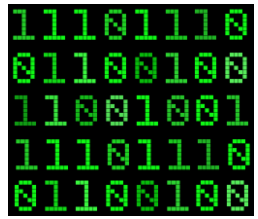
UGLY

- Undecidability necessitates low-level **assumptions**.
- Trustworthiness depends on validity of assumptions.

Disassembly

is more than just disassembly.

Solution:
Implement a monolithic approach.



bytes@0x4000 == FF E0 ...
disasm(0xFFE0) == "jmp rax"

Disassembler

$$\begin{aligned} \texttt{rax} &== *[\texttt{0x6000} + 4 \cdot \texttt{edi}, 4] \\ *[\texttt{0x6000}, 4] &== \texttt{0x5000} \\ *[\texttt{0x6004}, 4] &== \texttt{0x5008} \\ *[\texttt{0x6008}, 4] &== \texttt{0x5010} \\ *[\texttt{0x6010}, 4] &== \texttt{0x5000} \\ \texttt{edi} &< 5 \end{aligned}$$

Invariants

Reachable instruction addresses

$0x4000 \rightarrow \{0x5000, 0x5008, 0x5010\}$

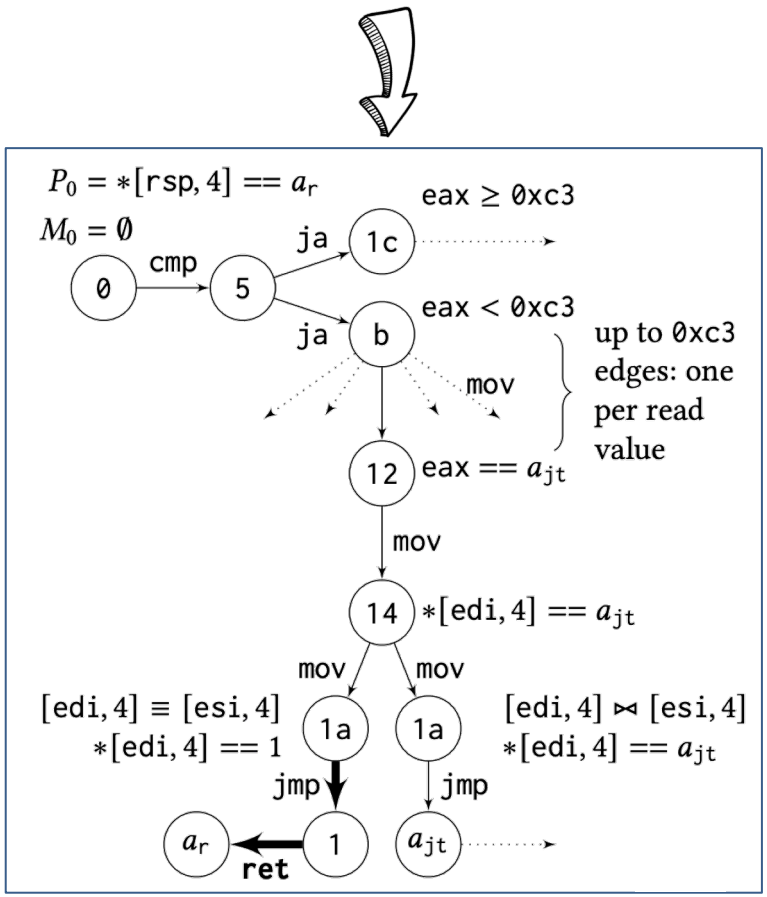
Resolved indirections

0x4000: jmp rax

GOOD

A disassembler whose **output** can be formally verified.

```
0x0:  3dc3000000    cmp eax,c3
0x5:  0f8718000000  ja  1c
0xb:  8b0485__a___  mov eax,DWORD PTR [eax*4+a]
0x12: 8907          mov DWORD PTR [edi],eax
0x14: c70601000000  mov DWORD PTR [esi],1
0x1a: ff27          jmp DWORD PTR [edi]
```



The output is a **Hoare Graph**:

- an overapproximation of the binary
- nodes are state predicates and memory models
- each edge is 1-step inductive
- overapproximation exposes **weird** edges

GOOD

A disassembler whose output can be **formally verified**.


```
(*
###################################
## Entry = 1000037cc, blockId == 0 ##
###################################
*)

htriple "ht_1000037cc"
 Separations "((RSP_0 -64 8),8) SEP ([0x100004008,8]_0,8); ((RSP_0 -64 8),8) SEP (0x100004008,8); ((RSP_0 -64 8),8) SEP ((RDI_
 Assertions  ""
 Pre   "RAX = RAX_0 ; RBX = RBX_0 ; RCX = RCX_0 ; RDX = RDX_0 ; RDI = RDI_0 ; RSI = RSI_0 ; RSP = RSP_0 ; RBP = RBP_0 ; R14 =
 Instruction "1000037cc: PUSH RBP 1"
 Post  "RAX = RAX_0 ; RBX = RBX_0 ; RCX = RCX_0 ; RDX = RDX_0 ; RDI = RDI_0 ; RSI = RSI_0 ; RSP = (RSP_0 -64 8) ; RBP = RBP_0
  by (htriple_solver seps: conjI[OF seps asserts,simplified] assms: assms)


htriple "ht_1000037cd"
 Separations ""
 Assertions  ""
 Pre   "RAX = RAX_0 ; RBX = RBX_0 ; RCX = RCX_0 ; RDX = RDX_0 ; RDI = RDI_0 ; RSI = RSI_0 ; RSP = (RSP_0 -64 8) ; RBP = RBP_0
 Instruction "1000037cd: MOV RBP, RSP 3"
 Post  "RAX = RAX_0 ; RBX = RBX_0 ; RCX = RCX_0 ; RDX = RDX_0 ; RDI = RDI_0 ; RSI = RSI_0 ; RSP = (RSP_0 -64 8) ; RBP = (RSP_0
  by (htriple_solver seps: conjI[OF seps asserts,simplified] assms: assms)


htriple "ht_1000037d0"
 Separations "((RSP_0 -64 16),8) SEP ([0x100004008,8]_0,8); ((RSP_0 -64 16),8) SEP (0x100004008,8); ((RSP_0 -64 16),8) SEP ((R
 Assertions  ""
 Pre   "RAX = RAX_0 ; RBX = RBX_0 ; RCX = RCX_0 ; RDX = RDX_0 ; RDI = RDI_0 ; RSI = RSI_0 ; RSP = (RSP_0 -64 8) ; RBP = (RSP_0
 Instruction "1000037d0: PUSH R14 2"
 Post  "RAX = RAX_0 ; RBX = RBX_0 ; RCX = RCX_0 ; RDX = RDX_0 ; RDI = RDI_0 ; RSI = RSI_0 ; RSP = (RSP_0 -64 16) ; RBP = (RSP_
  by (htriple_solver seps: conjI[OF seps asserts,simplified] assms: assms)
```

GOOD

Applicable to x86-64 COTS binaries.

Directory		Instrs.	Symbolic States	A	B	C	Time (h:m:s)
	Binaries						
. . . /bin	15 = 12 + 2 + 1 +0	6751	6829	21	19	0	0:15:54
. . . /xen/bin	17 = 7 + 1 + 8 +1	2433	2468	8	3	3	0:01:17
. . . /libexec	1 = 1 + 0 + 0 +0	82	87	1	0	0	0:00:10
. . . /sbin	30 = 25 + 1 + 4 +0	8858	9178	26	4	8	0:18:39
Total	63 = 45 + 3 +13+1	18 124	18 562	56	26	11	0:35:59
	Library functions						
. . . /lib	1907=1874+29+ 0 +4	353 433	362 635	1	244	600	15:28:17
. . . /xenfsimage	109 = 106 + 3 + 0 +0	17 184	17 683	0	0	27	1:58:36
. . . /dist-packages	16 = 16 + 0 + 0 +0	379	407	0	0	3	0:00:06
. . . /lowlevel	119 = 119 + 0 + 0 +0	10 651	10 799	0	0	90	0:08:43
Total	2151=2115+32+ 0 +4	381 647	391 524	1	244	720	17:35:42

$w + x + y + z$: w lifted, x unprovable return address, y concurrency, z timeout

A = Resolved indirection B = Unresolved jump(s) C = Unresolved call(s)

BAD

It may **fail**.

Functions are not adhering to the calling convention:

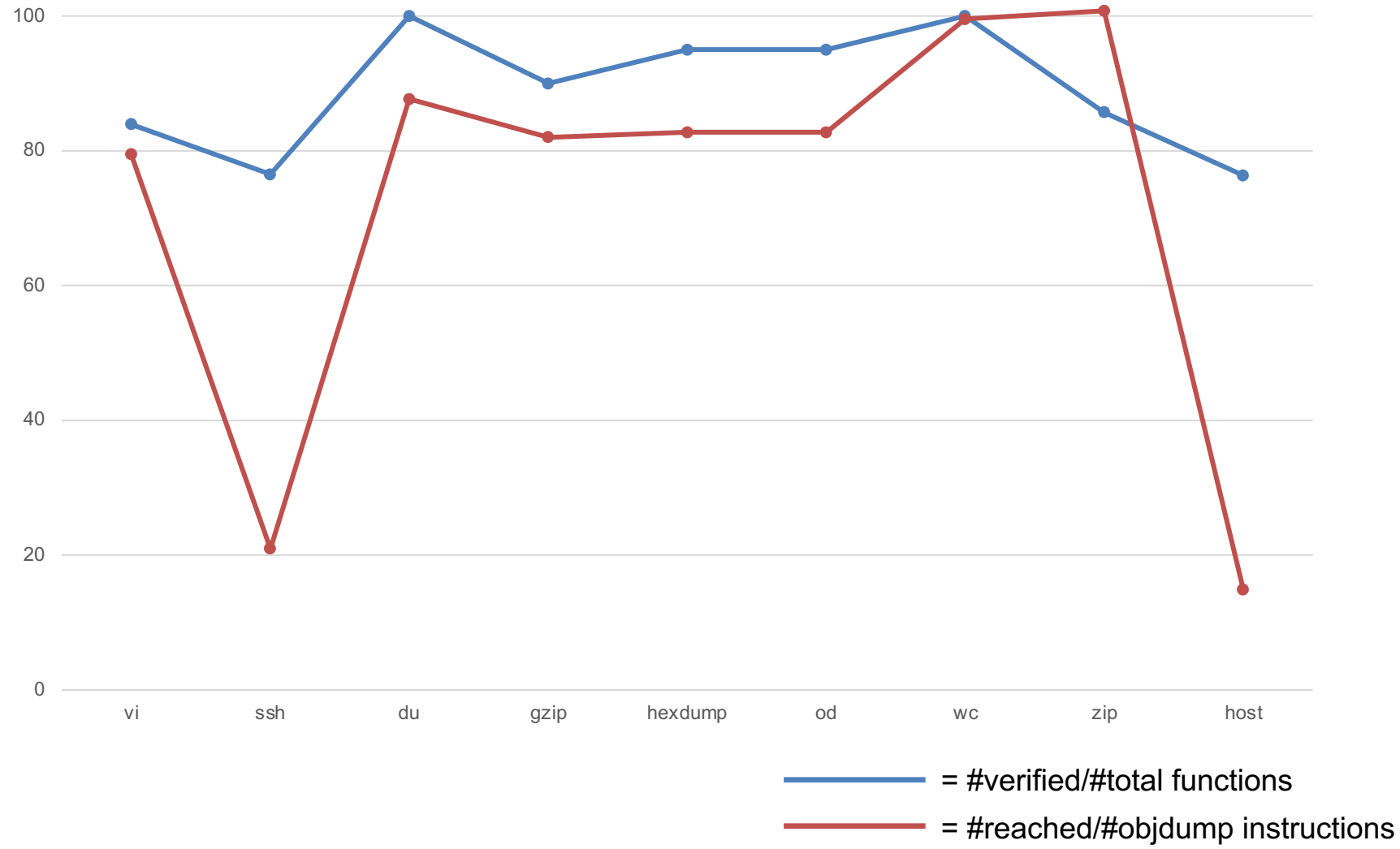
```
100009fe6:  mov eax, 0x1400
100009feb:  call 0x10000a6a0
100009ff0:  sub rsp, rax
```

After a function return, the symbolic value of the stackpointer is:

```
RSP ==
  DEREF (
    ((RSP_0 - (48 - ((0xfffffffffffffffc - b32(R9_0)) * 8))) & 0xfffffffffffffc00)
     +
    (((udiv64(b32(R9_0),4) * 4) * 8) + 8)
  ) + 56
```

BAD

Only small parts of the binary may be reached.



UGLY

Undecidability necessitates low-level **assumptions**.

Consider a function call to memset:

```
400701: call memset
```

Does this call overwrite register **RSP**?

UGLY

Undecidability necessitates low-level **assumptions**.

Consider a function call to memset:

```
400701: call memset
```

Does this call overwrite register **RBP**?

```
@400701: memset MUST PRESERVE rbp
```

UGLY

Undecidability necessitates low-level **assumptions**.

Consider a function call to memset:

```
400701: call memset
```

Does this call overwrite the return address of the caller?

```
@400701: memset(RDI := RSP_0 - 40) MUST PRESERVE [RSP_0 - 8 to RSP_0 + 8]
```

UGLY

Undecidability necessitates low-level **assumptions**.

Consider a function call to memset:

```
400701: call memset
```

Does this call overwrite the return address of the caller?

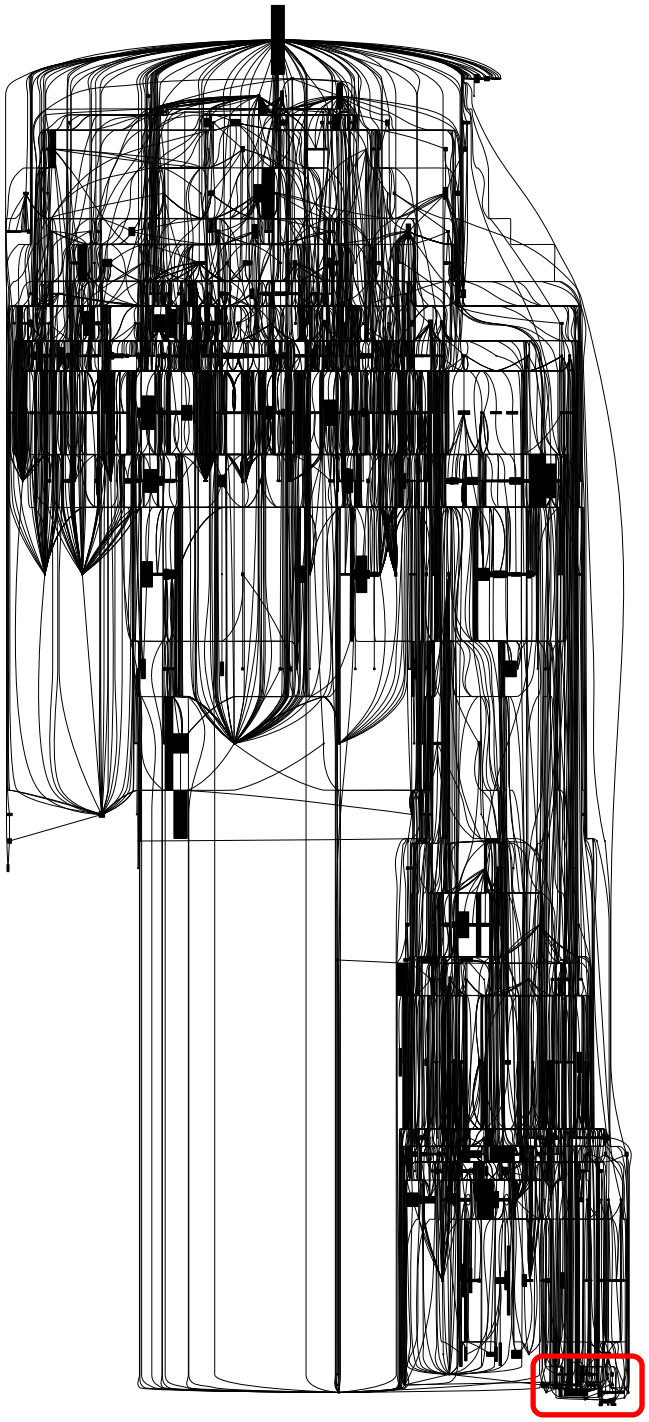
```
@400701: memset(RDI := RSP_0 - 40) MUST PRESERVE [RSP_0 - 8 to RSP_0 + 8]
```

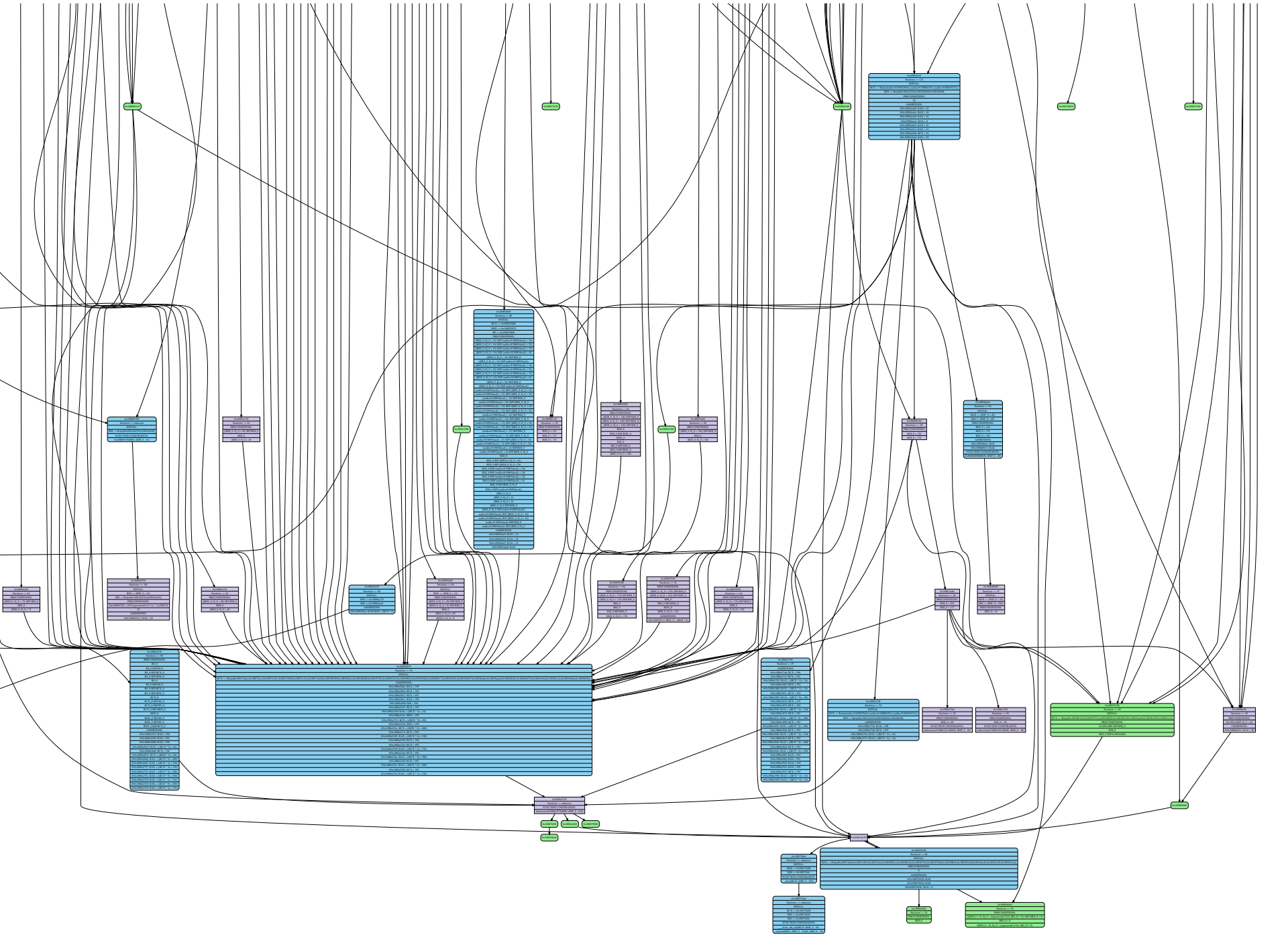
Is function provided a pointer to callers' stackframe?

no → **Assume** callers' stackframe is unmodified.

yes → **Assert** critical parts callers' stackframe are unmodified.

ssh





Binary verification is:

"Under the following 142.518 assumptions can we verify that memory region `[0x410320,8]` *holds value* `*[RSI_0 + 8,8]` *at address* `0x400960`*"*

Conclusion

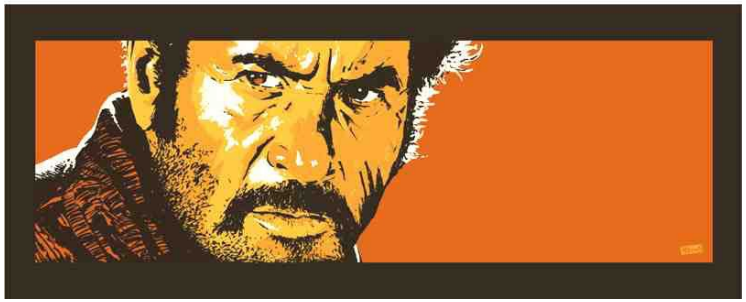
Can we build a formally verified disassembler?

GOOD

- A disassembler whose output can be **formally verified**.
- Applicable to x86-64 COTS binaries.
- No debugging information or source code required.

BAD

- It may **fail.**
- Only small parts of the binary may be reached.



UGLY

- Undecidability necessitates low-level **assumptions**.
- Trustworthiness depends on validity of assumptions.

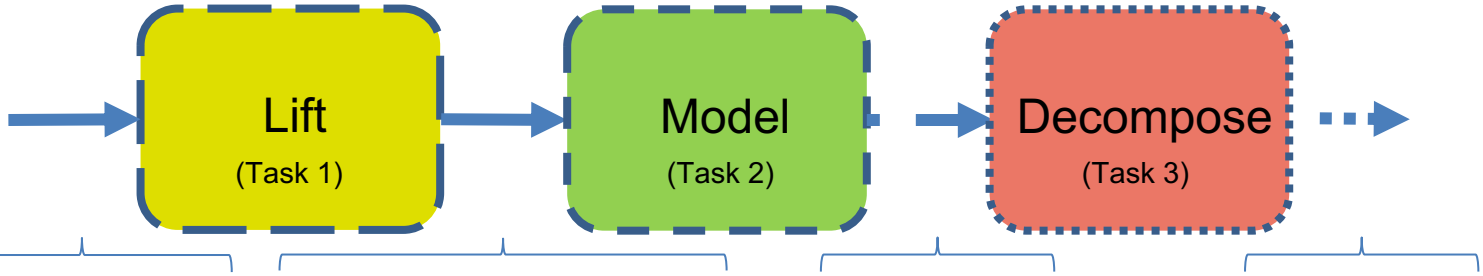

THANK YOU

FALCON

- Disassembly
- Control Flow
- Invariants
- Function Detection
- Verification of sanity properties

- Pointer Analysis
- Variable Recovery
- Control Flow Recovery
- Symbolization

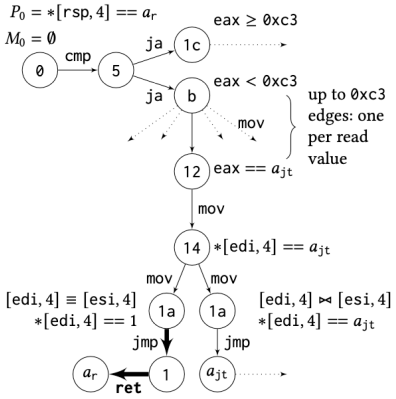
- Dataflow analysis
- Component Recovery
- Exploit generation



Binary:

```
0x0:  3dc3000000
0x5:  0f8718000000
0xb:  8b0485__a___
0x12: 8907
0x14: c70601000000
0x1a: ff27
```

SSM:



IR:

PIE NASM
LLVM?
C Code?

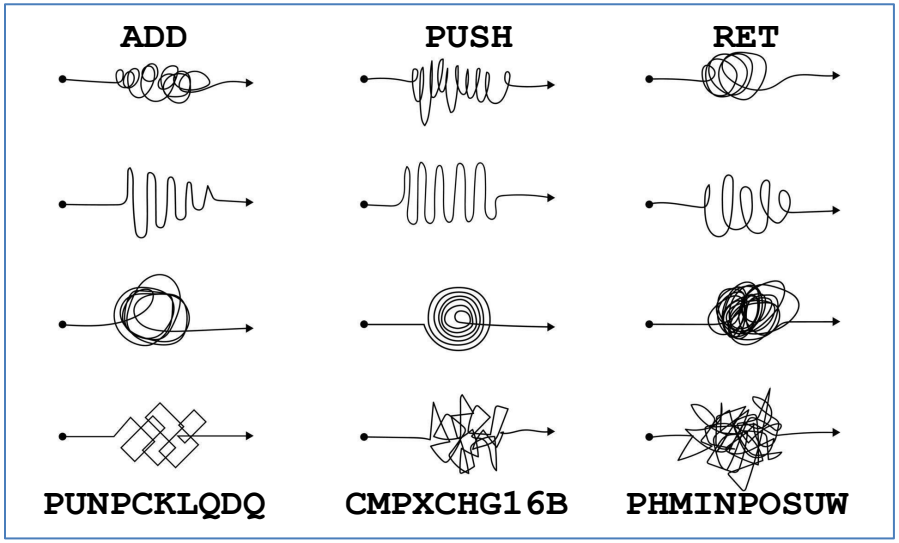
DSSM

Instruction Semantics

Jos Craaijo (OU)

- Overapproximative lifting requires formal semantics for the x86-64 instruction set.

Which instructions modify the instruction pointer non-trivially?
Which instructions modify parts of the state not explictely added as operands?
Which instructions follow a standard DST-SRCS patterns?

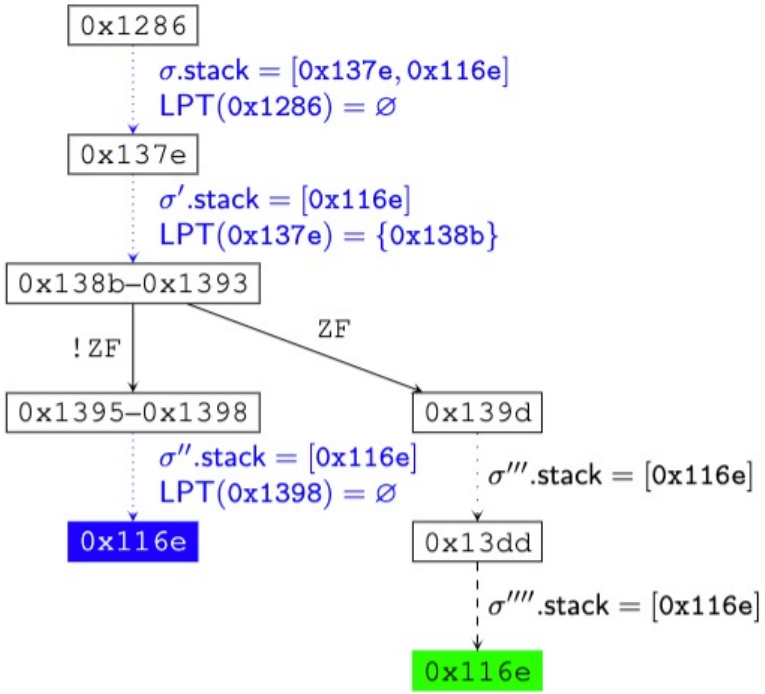


Solution:
Learn semantics automatically from a CPU.

Exceptional Control Flow

Joshua Bockenek (VT)

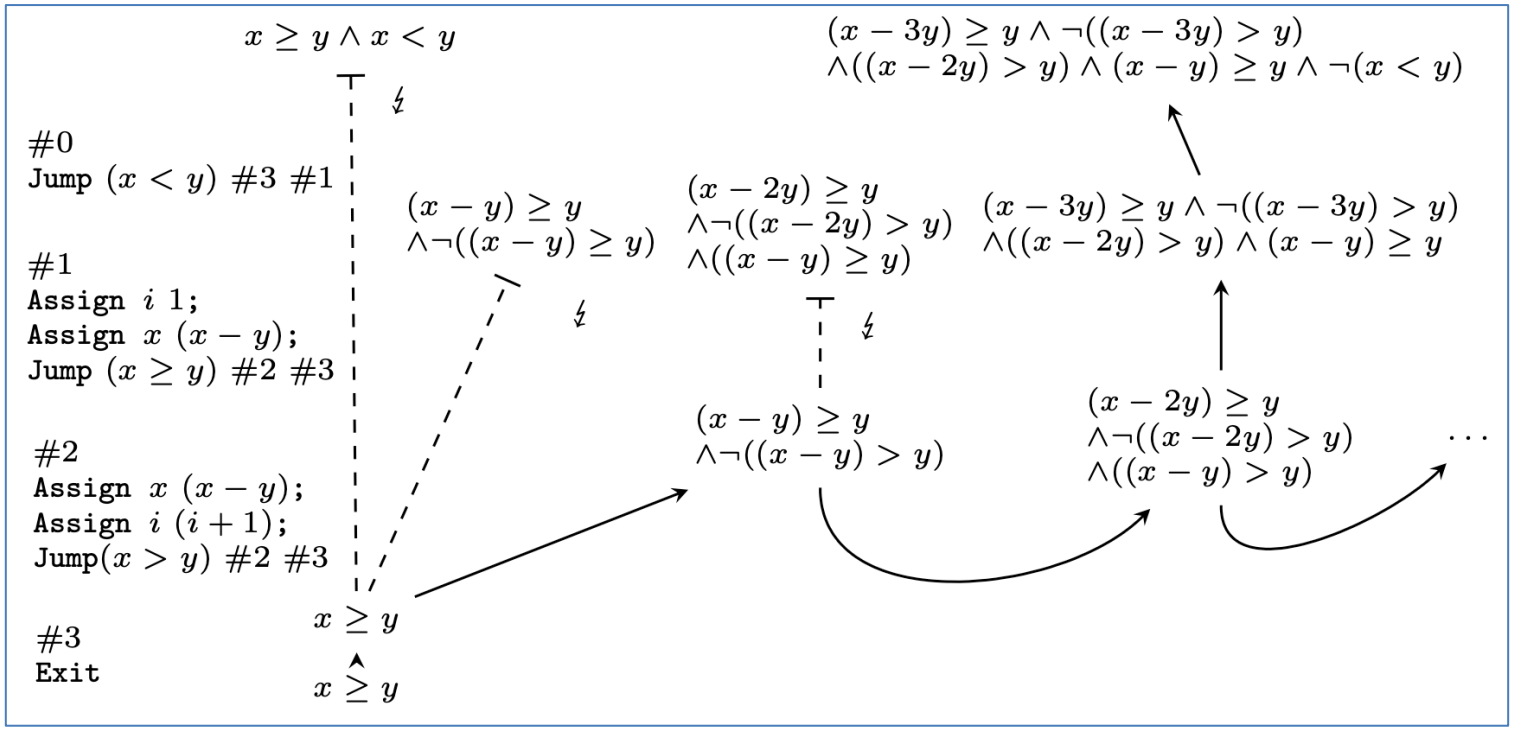
- Dealing with the binary-level behavior of C++ throw- and catch behavior.



Exploit Generation

Nico Naus (~~OU~~, VT (but also OU))

- Can we leverage the generated assumptions+graphs to make **weird** behavior?



Windows Executable Exploration

Xiaoxin An (VT) + Matthijs de Bruijn (OU)

- If we have recovered the control flow, can we explore paths in it?

Exec name	# of reached instrs	# of paths	# of negatives	# of unini-tialized	# of unresolved indirects	# of unreached instrs
ARP.EXE	2825	996	8	9	3	777
HOSTNAME.EXE	1037	241	0	10	3	110
clip.exe	3642	1078	0	13	10	1732
ftp.exe	3898	1423	0	9	3	6405
logman.exe	6351	2321	0	101	34	11507
msconfig.exe	570	174	0	20	3	16615
ndadmin.exe	1625	591	0	30	5	209
netsh.exe	5028	1696	0	206	5	6125
ping6.exe	3002	1443	0	194	2	4437
replace.exe	1438	245	0	14	18	1034

Formal Decompilation

Daniel Spaniol (OU)

- … because “formal disassembly” is such an easy problem.

```
factorial(int):
 push rbp
 mov rbp, rsp
 mov DWORD PTR [rbp-20], edi
 mov DWORD PTR [rbp-4], 1
 mov DWORD PTR [rbp-8], 0
 jmp .L2
.L3:
 mov eax, DWORD PTR [rbp-4]
 imul eax, DWORD PTR [rbp-8]
 mov DWORD PTR [rbp-4], eax
 add DWORD PTR [rbp-8], 1
.L2:
 mov eax, DWORD PTR [rbp-8]
 cmp eax, DWORD PTR [rbp-20]
 jl .L3
 mov eax, DWORD PTR [rbp-4]
 pop rbp
 ret
```

```
factorial(int):
%rsp1 ← push %rbp0
%rbp1 ← mov %rsp1
%n0   ← mov %edi0
%res0 ← mov 1
%i0   ← mov 0
        jmp .L2
.L3:
%eax1 ← mov %res2
%eax2 ← imul %eax1, %i2
%res1 ← mov %eax2
%i1   ← add  %i2, 1
.L2:
%i2   ← Φ(%i0, %i1)
%res2 ← Φ(%res0, %res1)
%eax3 ← mov %i2
        cmp %eax3, %n
        jl .L3
%eax4 ← mov %res2
%rbp2, %rsp2 ← pop
        ret
```


THANK YOU