Abstraction Recovery for Scalable Static Binary Analysis

Edward J. Schwartz
Software Engineering Institute
Carnegie Mellon University

The Gap Between Binary and Source Code

```
push    %ebp
mov     %esp,%ebp
sub     $0x10,%esp
movl    $0x1,-0x4(%ebp)
jmp     1d <f+0x1d>
mov     -0x4(%ebp),%eax
imul    0x8(%ebp),%eax
mov     %eax,-0x4(%ebp)
subl    $0x1,0x8(%ebp)
cmpl    $0x1,0x8(%ebp)
jg      f <f+0xf>
mov     -0x4(%ebp),%eax
leave
ret
```

Functions

Types

Variables

Control Flow

```
int f(int c) {
  int accum = 1;
  for (; c > 1; c--) {
    accum = accum * c;
  }
  return accum;
}
```

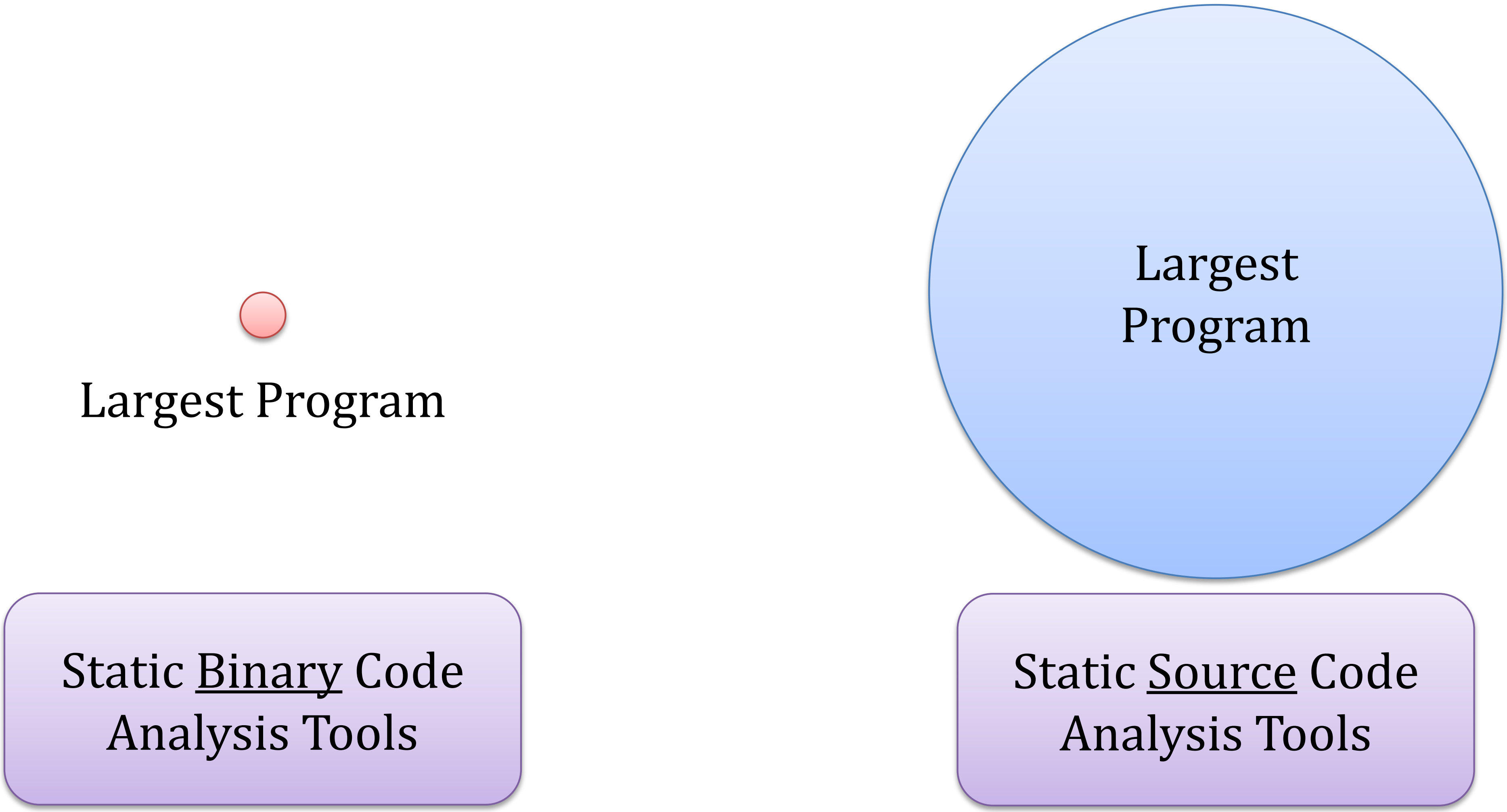
Static Binary Analysis

Automatic extraction of facts about binary programs without executing them

Static Binary Analysis Strengths

- **High Coverage**
  - Reason about most or all possible executions
- **Safe**
  - Does not execute (possibly unsafe) code
- **Widely Applicable**
  - Source code not needed
  - Useful for end-users, researchers, sysadmins

Primary Challenge: Scalability



The Gap Between Binary and Source Code

```
push    %ebp
mov     %esp,%ebp
sub     $0x10,%esp
movl    $0x1,-0x4(%ebp)
jmp     1d <f+0x1d>
mov     -0x4(%ebp),%eax
imul    0x8(%ebp),%eax
mov     %eax,-0x4(%ebp)
subl    $0x1,0x8(%ebp)
cmpl    $0x1,0x8(%ebp)
jg      f <f+0xf>
mov     -0x4(%ebp),%eax
leave
ret
```

Functions

Types

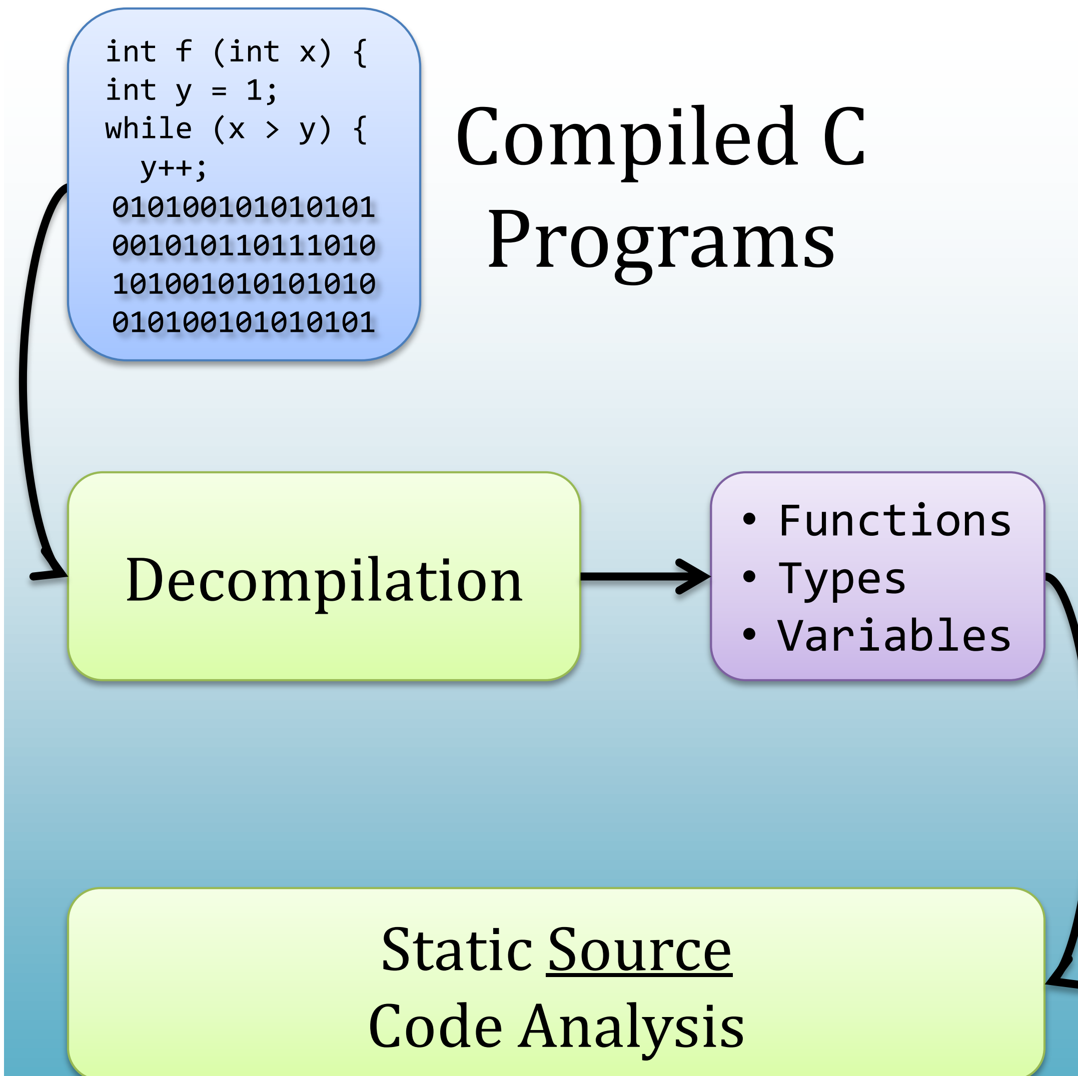
Variables

Control Flow

```
int f(int c) {
  int accum = 1;
  for (; c > 1; c--) {
    accum = accum * c;
  }
  return accum;
}
```

Abstraction Recovery

1. Choose abstractions
2. Recover abstractions
3. Scalable, high-level reasoning

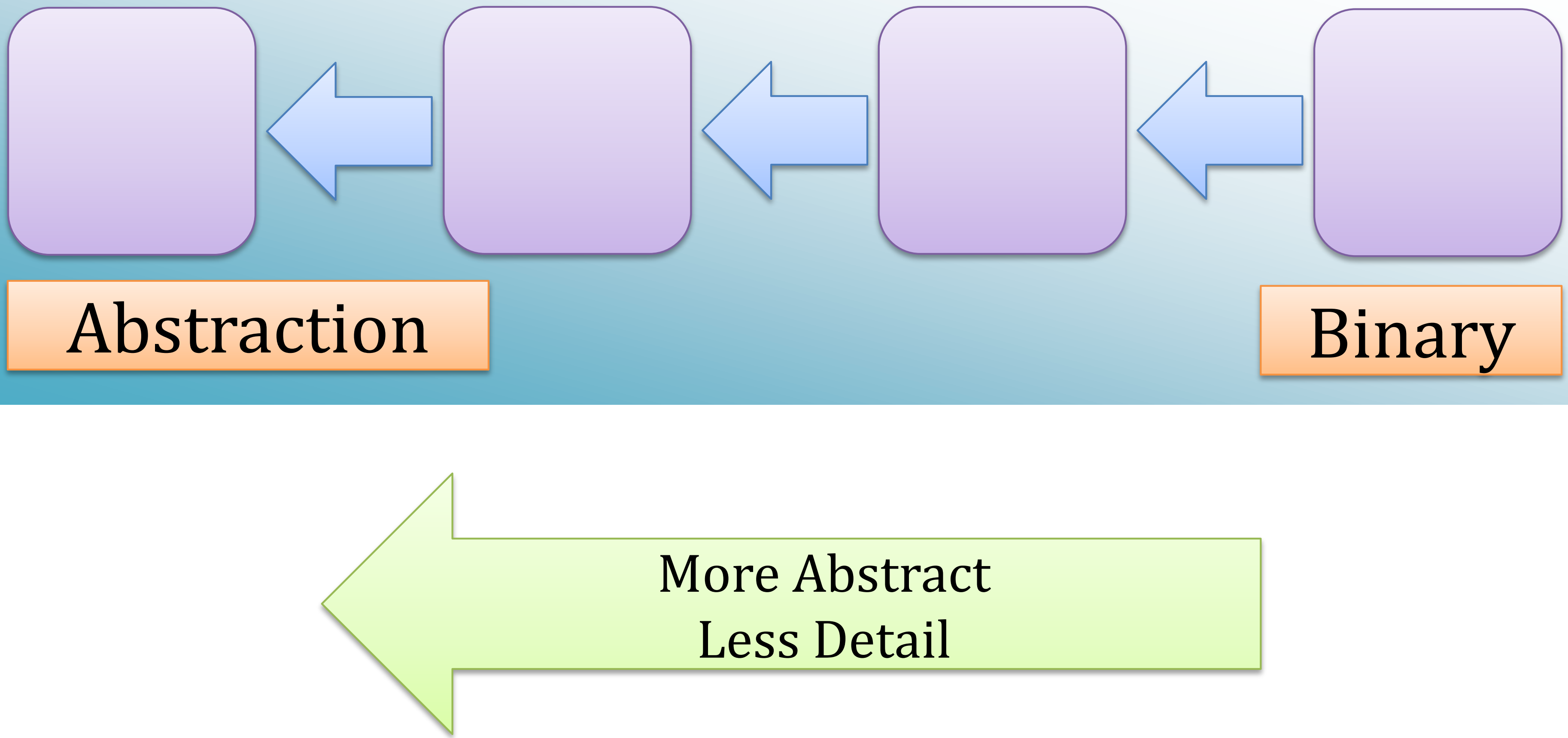


Reverse Engineering

"Reverse engineering is the process of analyzing a subject system to create representations of the system at a higher level of abstraction."

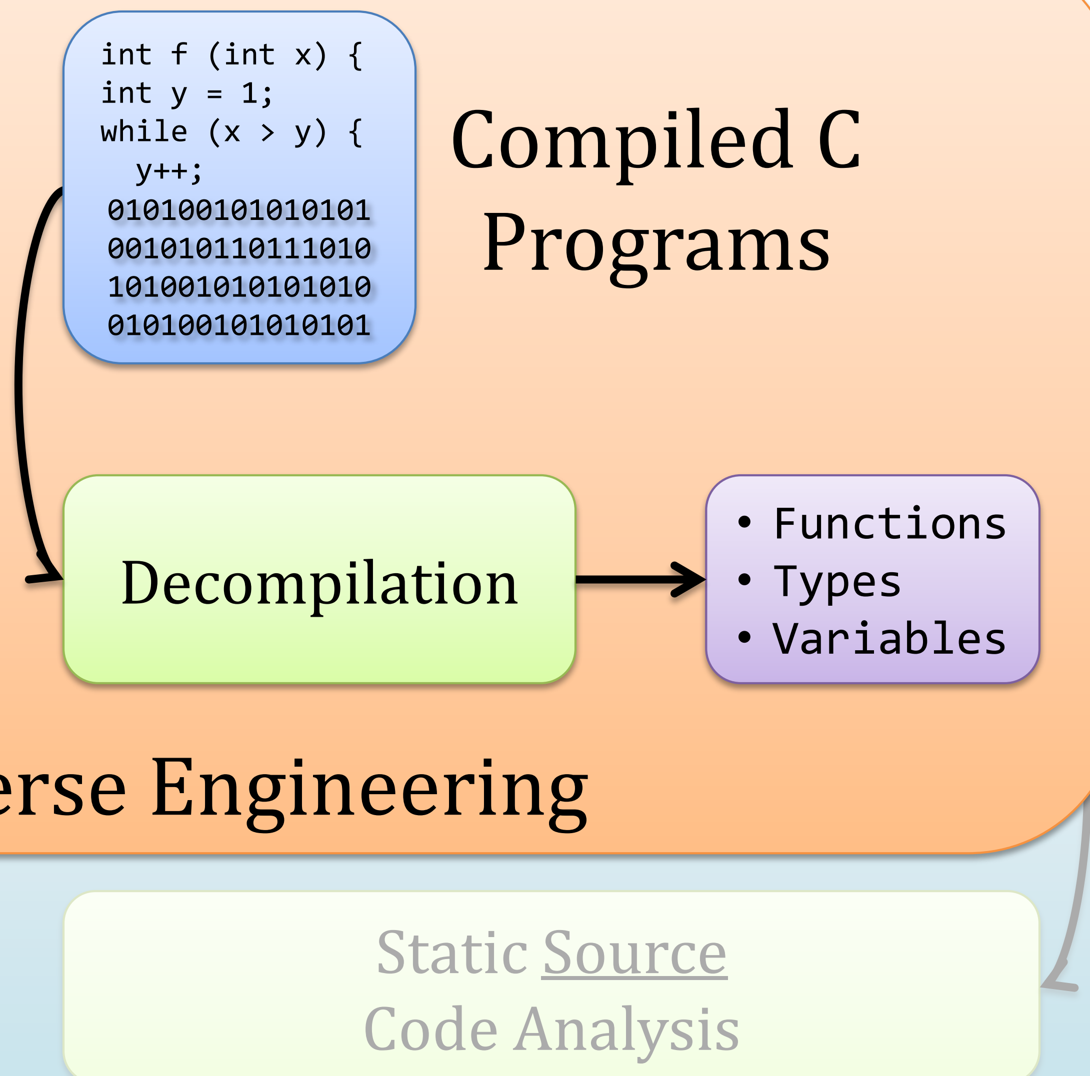
Chikofsky and Cross
Software Reuse and Reverse Engineering in Practice

Reverse Engineering



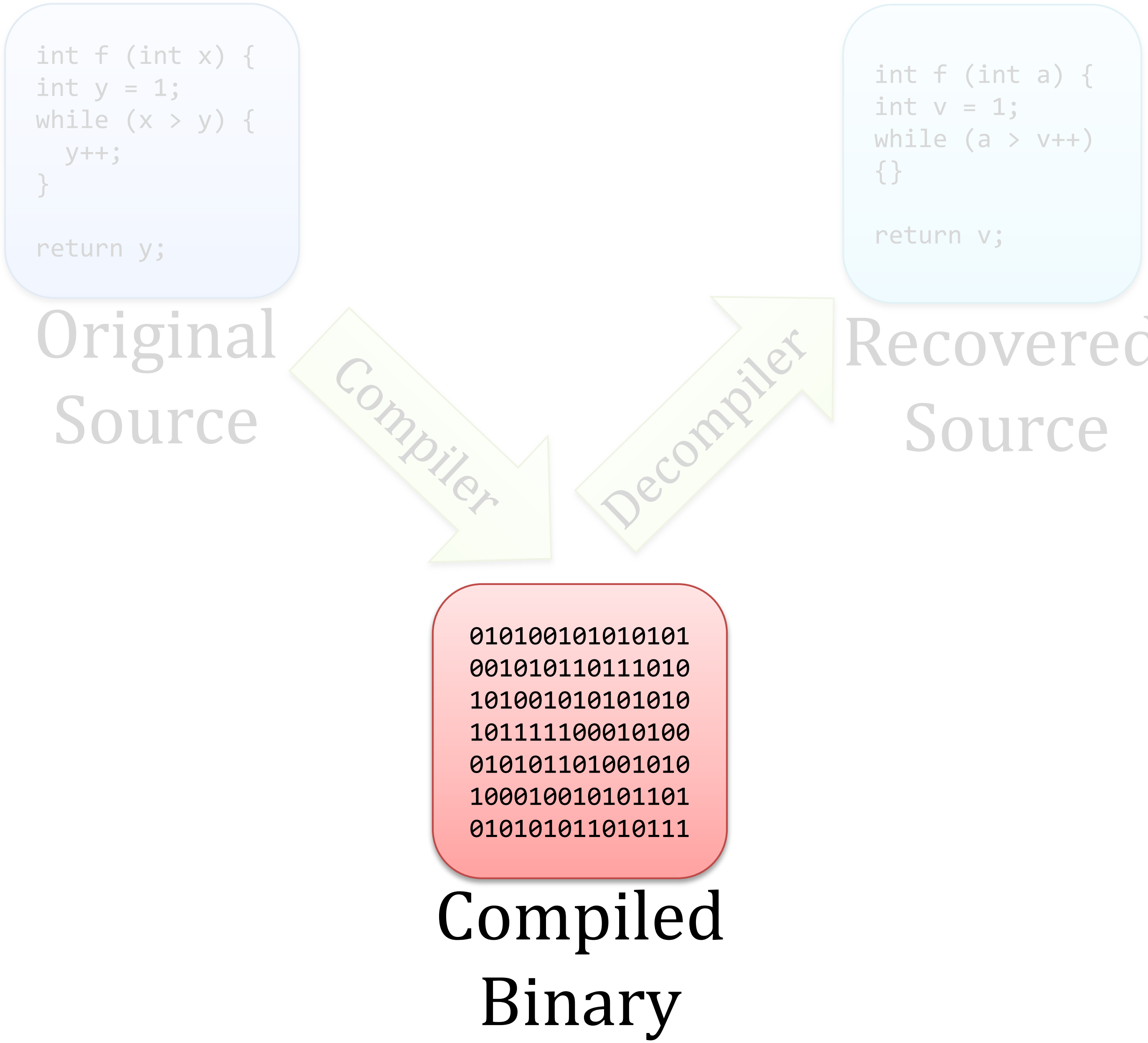
Abstraction Recovery

1. Choose abstractions
2. Recover abstractions
3. Scalable, high-level reasoning



Outline

- Introduction
- Recovering Abstractions
  - C abstractions (Phoenix Decompiler)
  - Gadget abstractions (Q ROP Compiler)
- Future Work and Conclusions

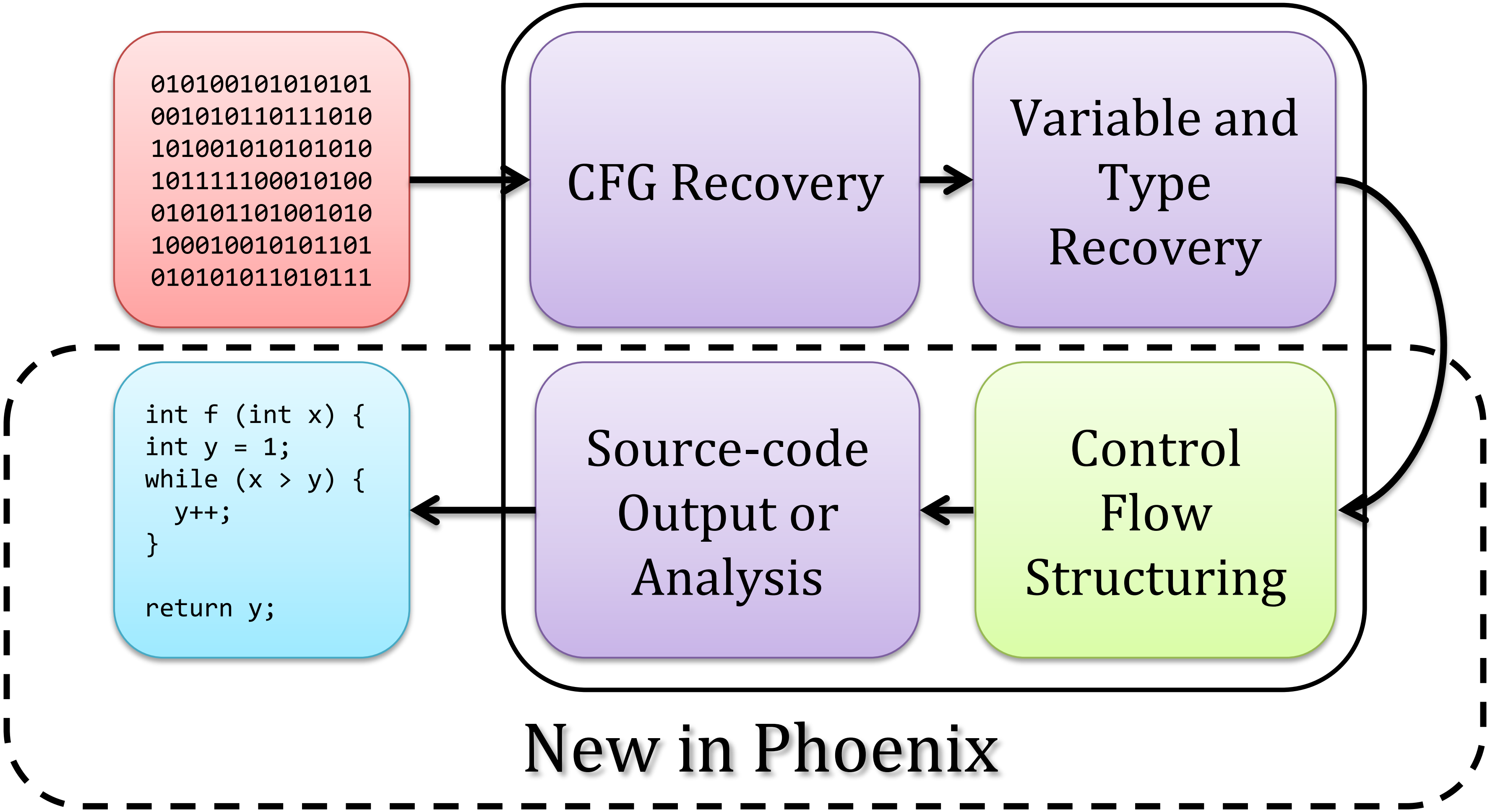

int f (int x) {
int y = 1;
while (x > y) {
y++;
}
return y;
Original Source
Compiler
Decompiler
int f (int a) {
int v = 1;
while (a > v++)
{}
return v;
Recovered Source
010100101010101
001010110111010
101001010101010
101111100010100
010101101001010
100010010101101
010101011010111
Compiled Binary

The Phoenix Decompiler

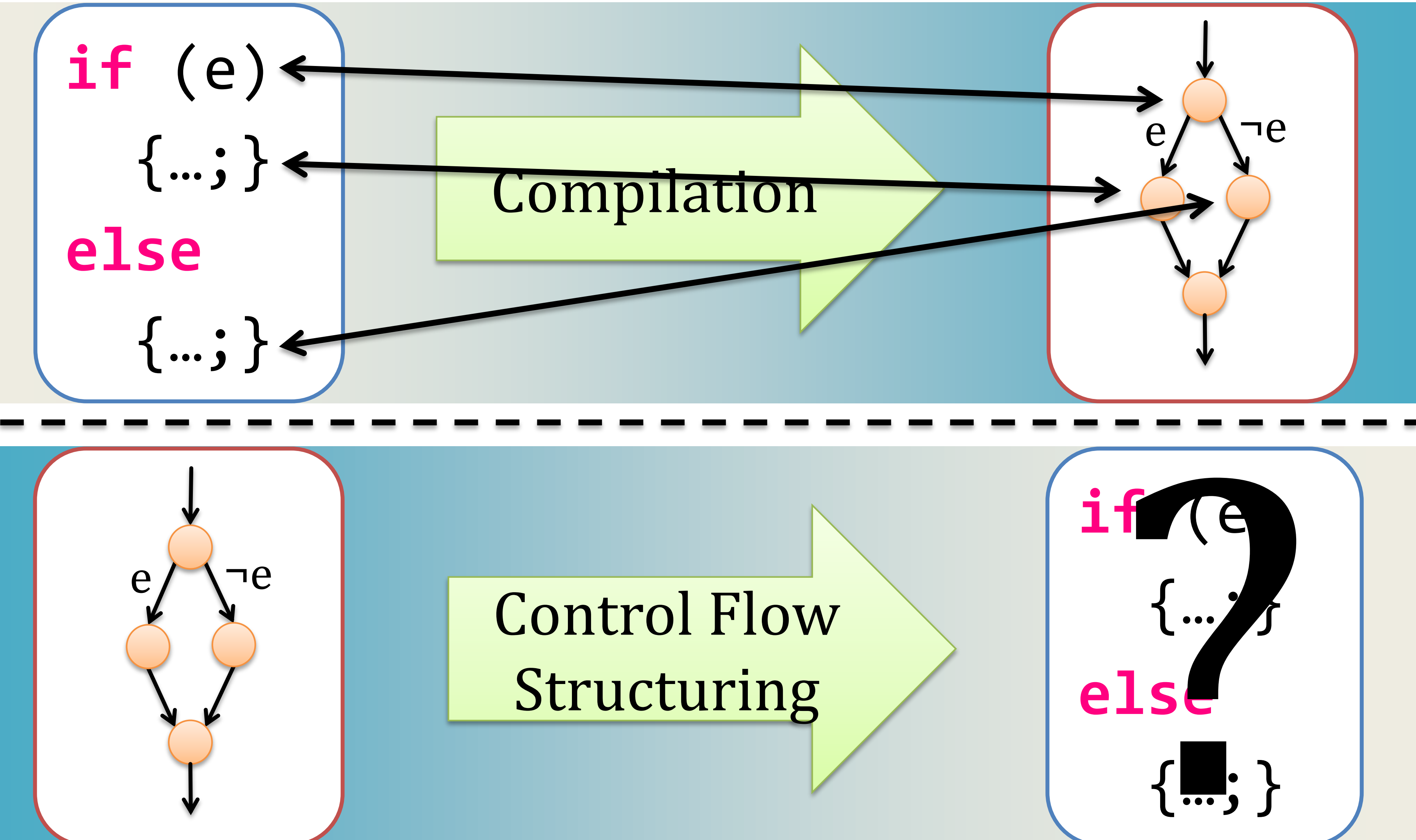
- Designed for abstraction recovery
  - Correctness (new)
    - Prior work: focus on manual reverse engineering
  - Effective abstraction recovery

- Design: series of stages
  - Each stage recovers a different abstraction
  - Some are new; some are not

Phoenix Overview

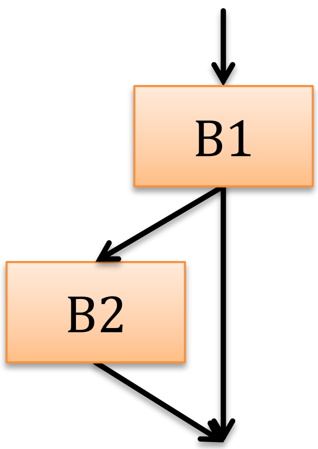


Control Flow Structuring

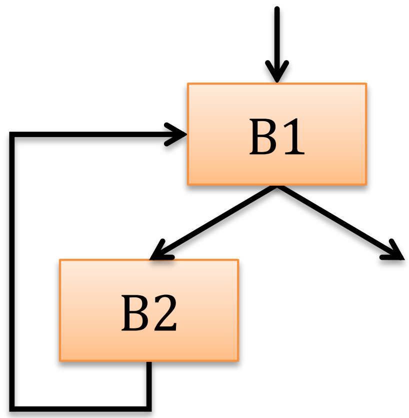


Structural Analysis

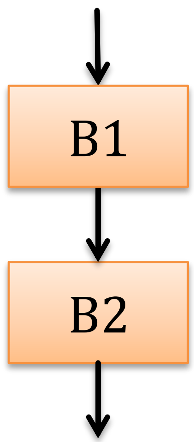
- Iteratively match patterns to CFG
  - Collapse matching regions



If-then

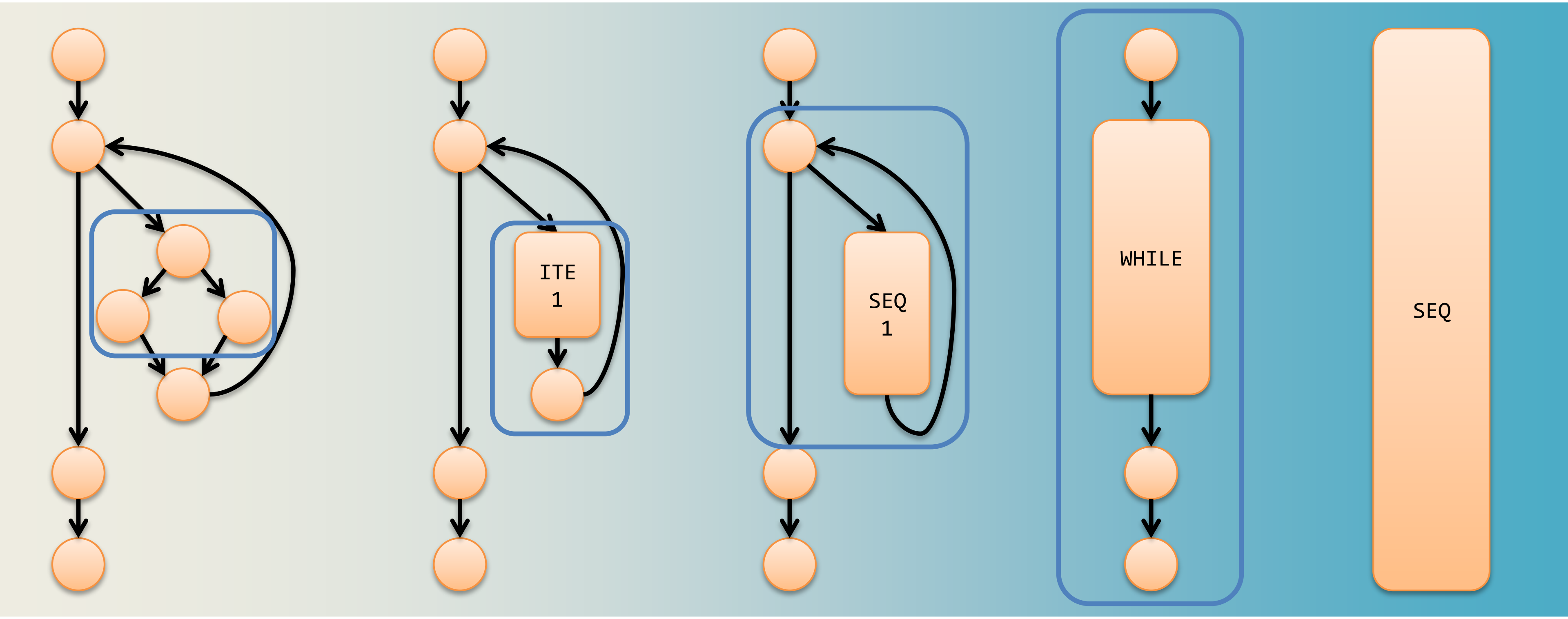


While



Sequence

Structural Analysis Example




```
...;
while (...) { if (...) {...} else {...} };
...; ...;
```

Structural Analysis Property Checklist

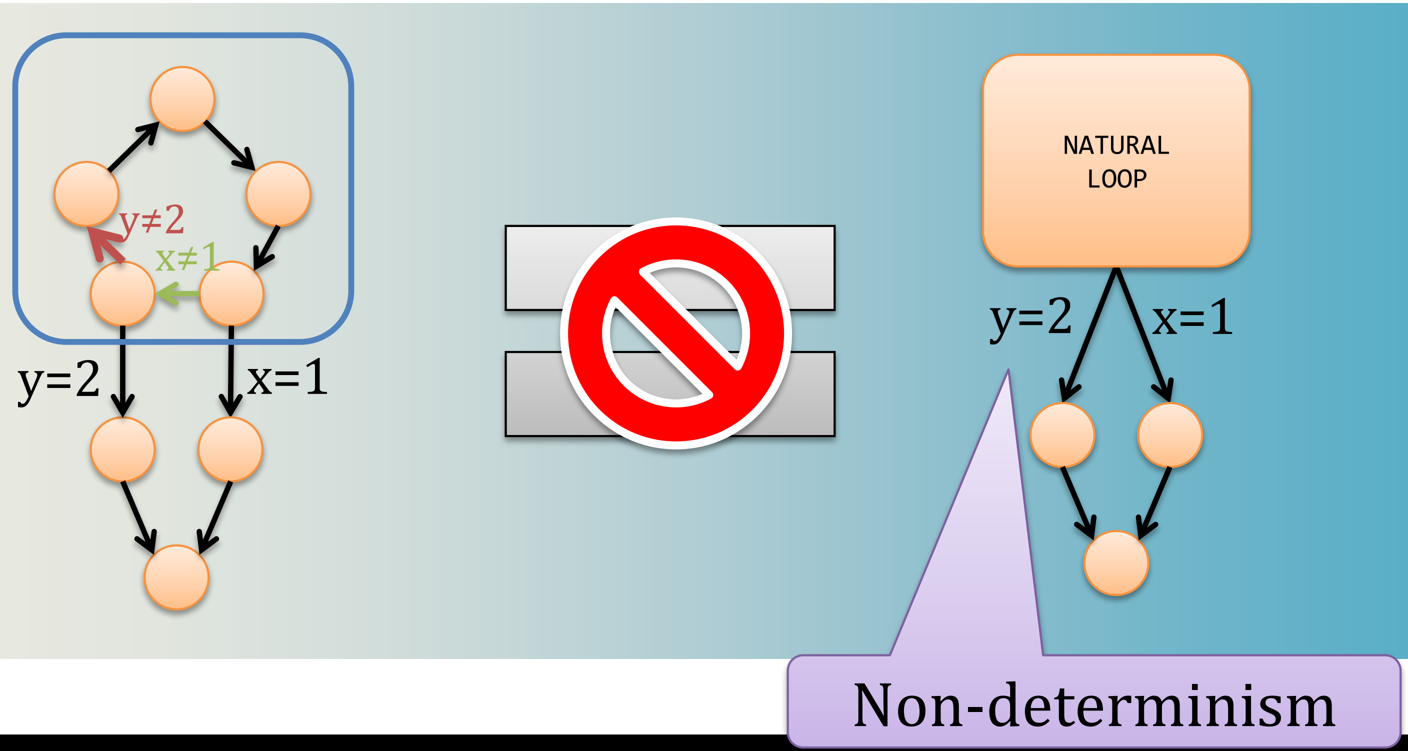
1. Correctness

Structural Analysis Property Checklist

1. ~~Correctness~~
   - Not originally intended for decompilation
   - Structure can be incorrect for decompilation

Semantics Preservation

- Reductions preserve meaning of program



Structural Analysis Property Checklist

1. ~~Correctness~~
   – Not originally intended for decompilation
   – Structure can be incorrect for decompilation

2. Effective abstraction recovery

Structural Analysis Property Checklist

1. ~~Correctness~~
   – Not originally intended for decompilation
   – Structure can be incorrect for decompilation

2. ~~Effective abstraction recovery~~
   – Grace<u>less</u> failures for unstructured programs
      • `break`, `continue`, and `goto`s
      • Failures cascade to large subgraphs

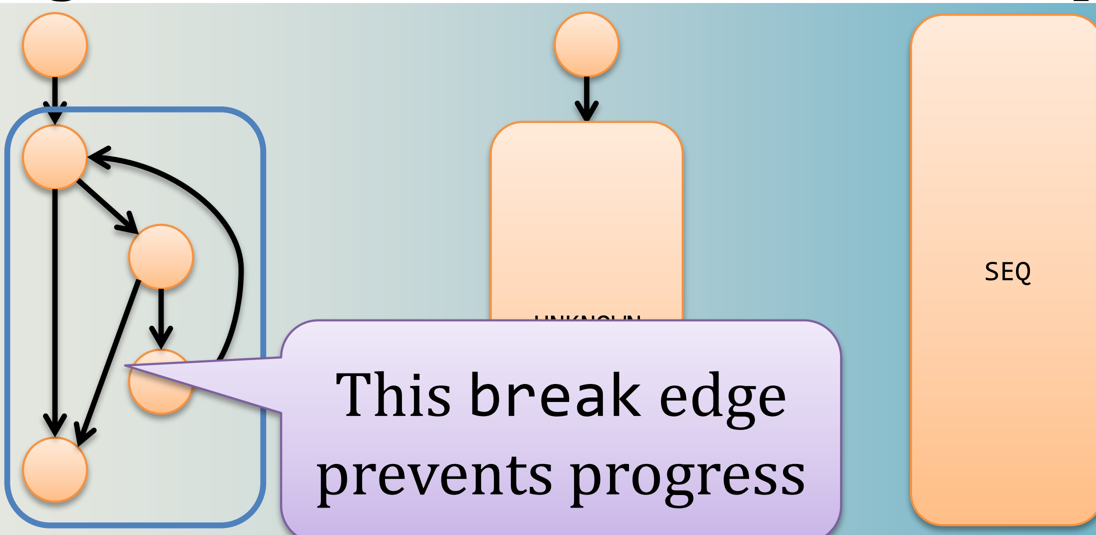
Unrecovered Structure

```
s1;
while (e1) {
  if (e2) { break; }
  s2;
}
s3;
```

Original

```
s1;
L1: if (e1) { goto L2; }
    else { goto L4; }
L2: if (e2) { goto L4; }
L3: s2; goto L1;
L4: s3;
```

Decompiled



Iterative Refinement

- Remove edges that are preventing a match
  - Represent in decompiled source as `break`, `goto`, `continue`
  - Run on remaining graph

Allows structuring algorithm to make more progress

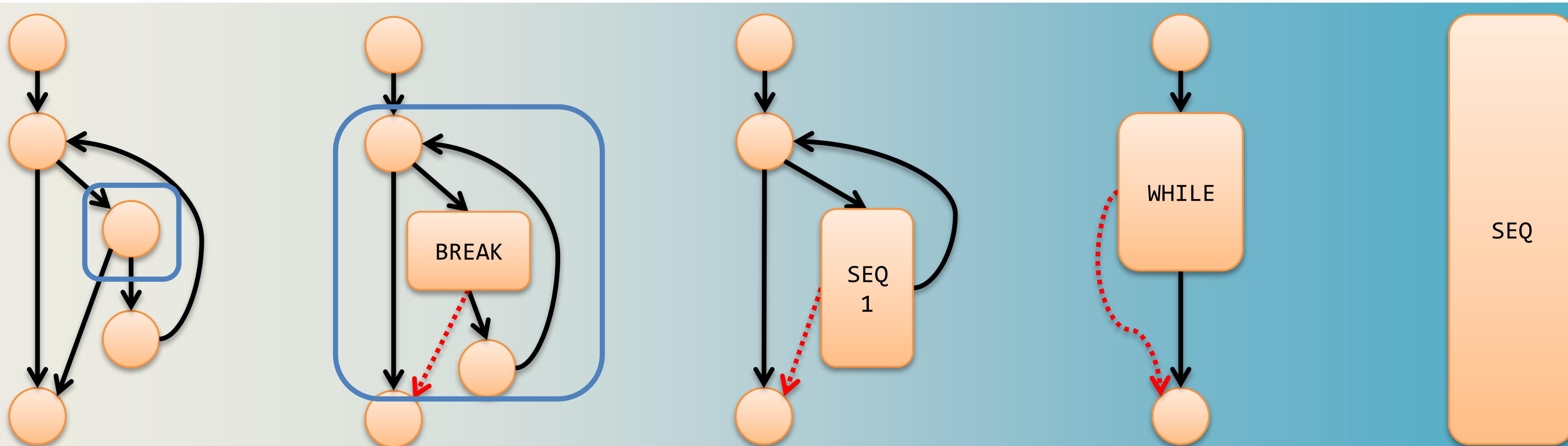
Iterative Refinement

```
s1;
while (e1) {
  if (e2) { break; }
  s2;
}
s3;
```

Original

```
s1;
while (e1) {
  if (e2) { break; }
  s2;
}
s3;
```

Decompiled



Large Scale Experiment Details

- How does Phoenix compare with state of the art?
- Measure impact of:
  - Semantics preservation
  - Iterative refinement
- Other decompilers
  - Hex-Rays (industry state of the art)
  - Boomerang (academic state of the art)

Large Scale Experiment Details

- How does Phoenix compare with state of the art?

- Measure impact of:
  - Semantics preservation
  - Iterative refinement

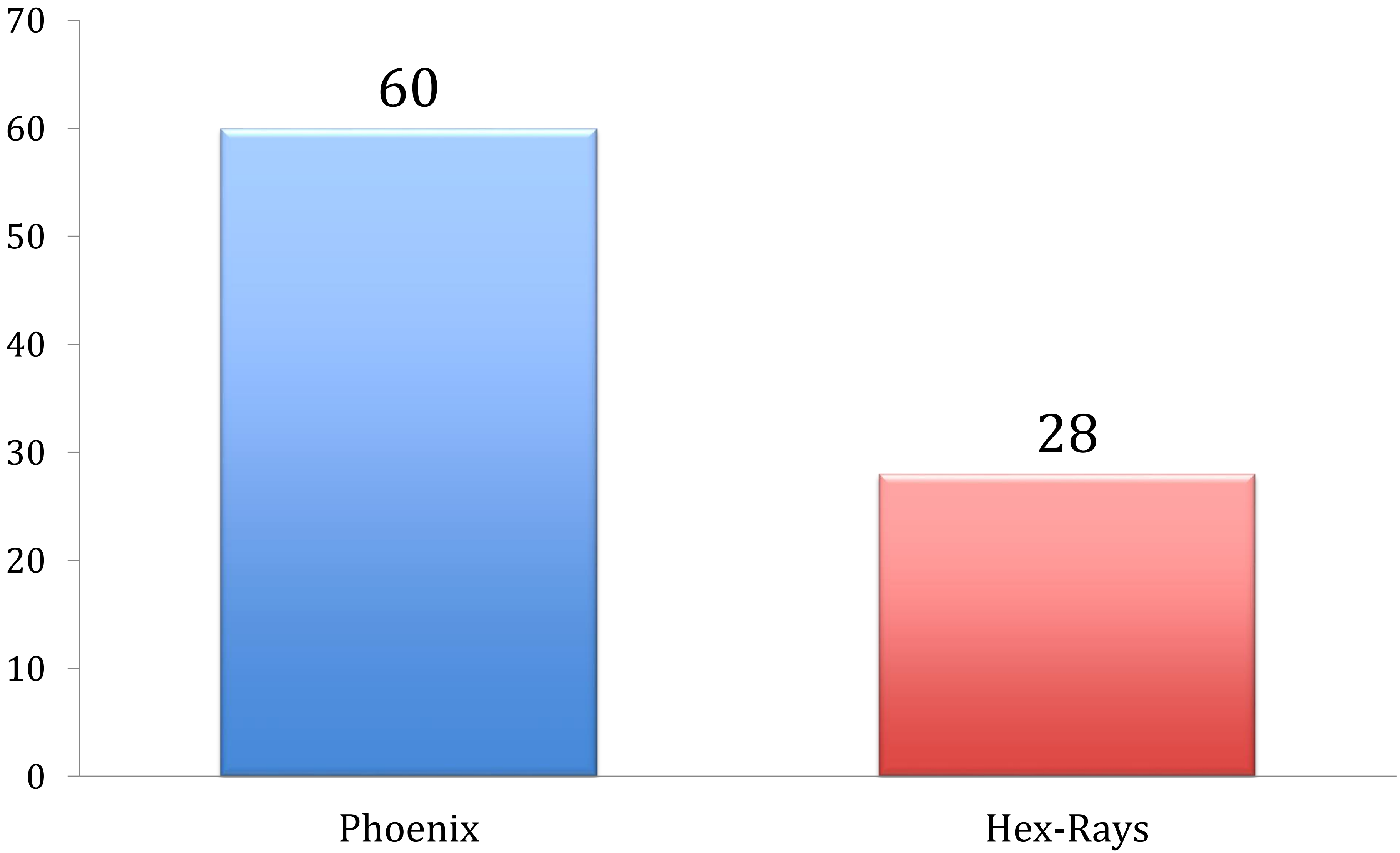
- Other decompilers
  - Hex-Rays (industry state of the art)
  - ~~Boomerang (academic state of the art)~~
    - Did not terminate in <1 hour for most programs

Large Scale Experiment Details

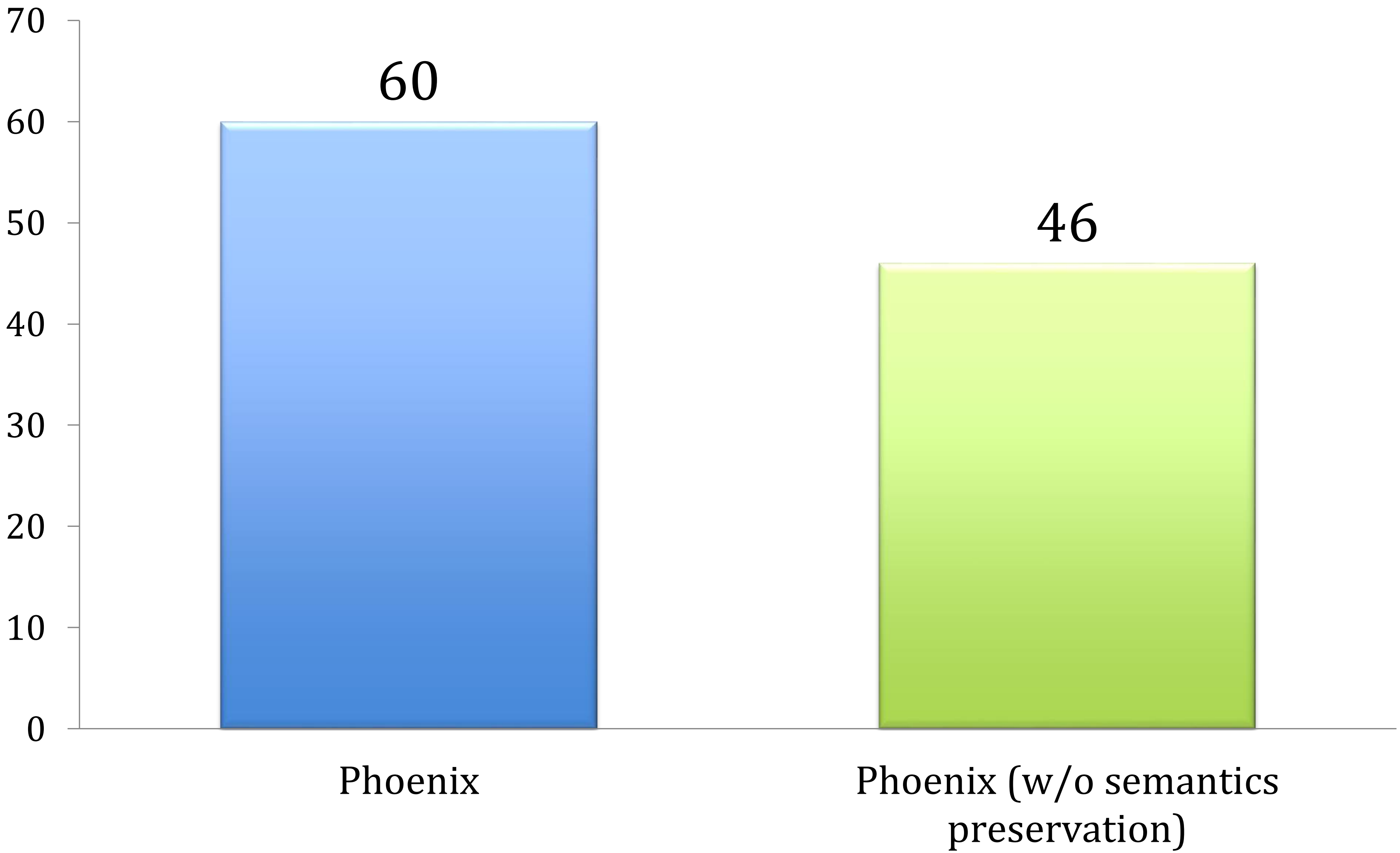
- GNU `coreutils` 8.17, compiled with `gcc`
  - Programs of varying complexity
  - Test suite

- Metrics
  - Correctness
    - Number of decompiled utilities that pass unit tests
    - <u>Has not been done before</u> on large scale!
  - Control-flow structure recovery
    - Count number of `goto` statements

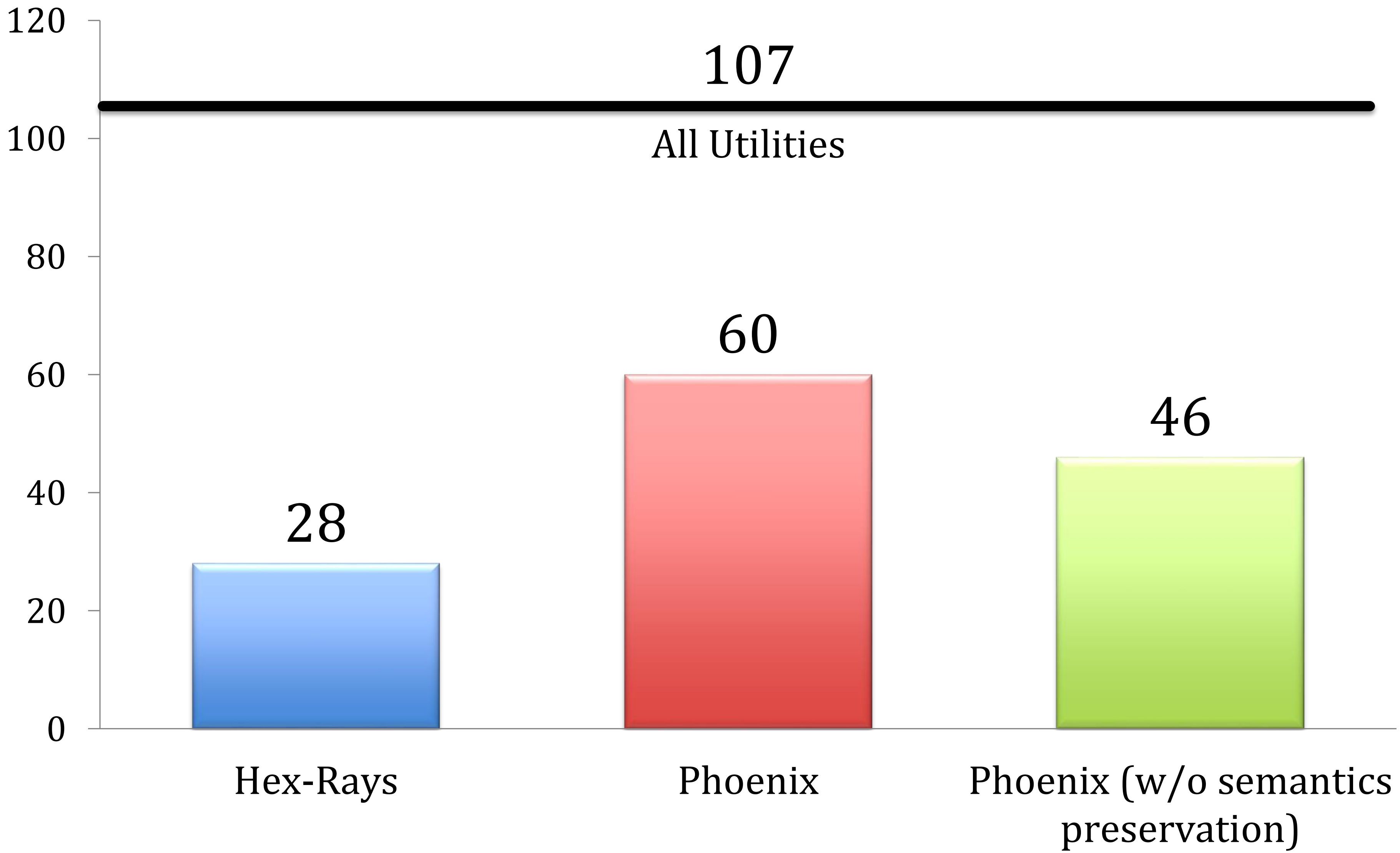
Number of Correct Utilities



Number of Correct Utilities



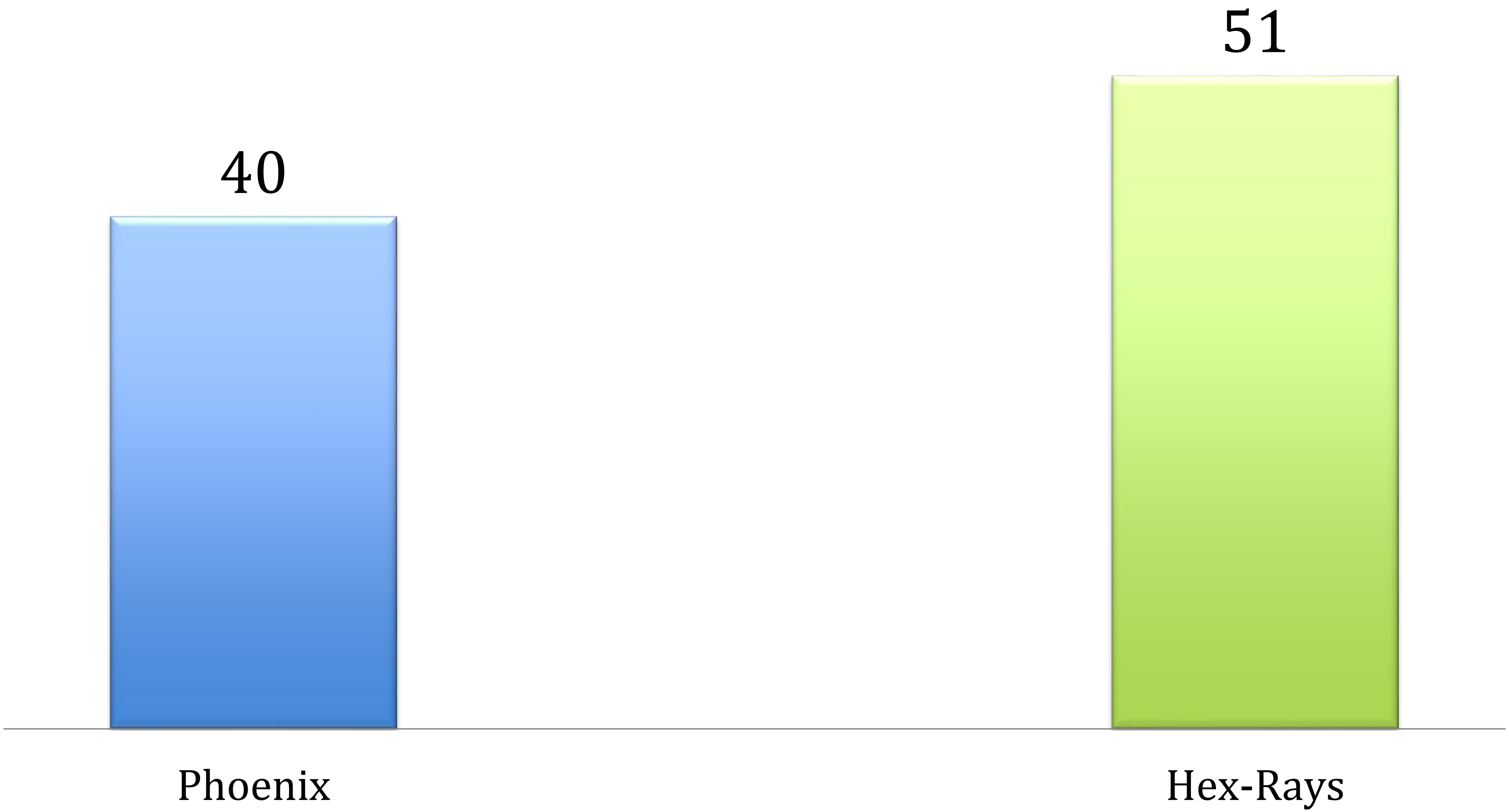
Number of Correct Utilities



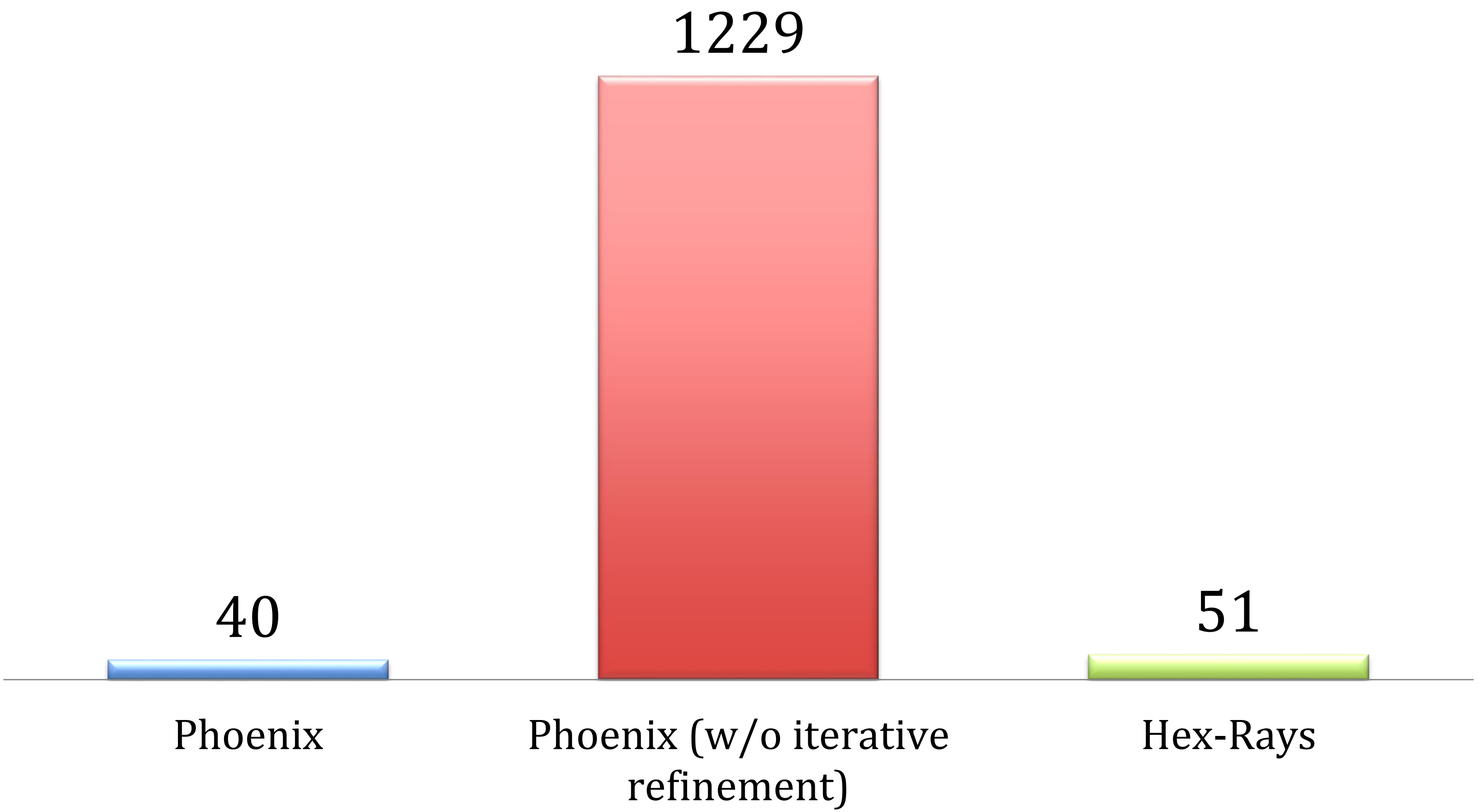
Correctness

- Any incorrect abstraction can cause incorrect decompilation
  - Hex Rays
    - ?
  - Phoenix
    - All (known) correctness errors attributed to type recovery
      - Undiscovered variables
    - No known problems in control flow structuring

Control Flow Structure: Gotos Emitted (Fewer is Better)



Control Flow Structure: Gotos Emitted (Fewer is Better)



Outline

- Introduction

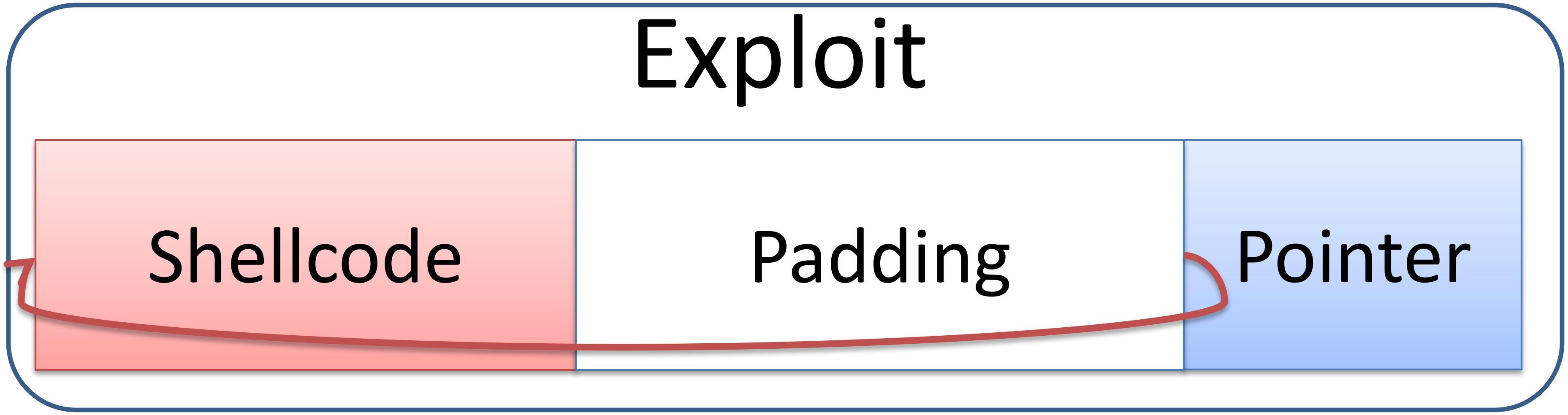
- Recovering Abstractions
  - C abstractions (Phoenix Decompiler)
  - Gadget abstractions (Q ROP Compiler)

- Future Work and Conclusions

OS Defenses

- All major operating systems employ defenses
  - DEP: Data Execution Prevention
  - ASLR: Address Space Layout Randomization
- Make reliable exploitation difficult
  - How difficult?

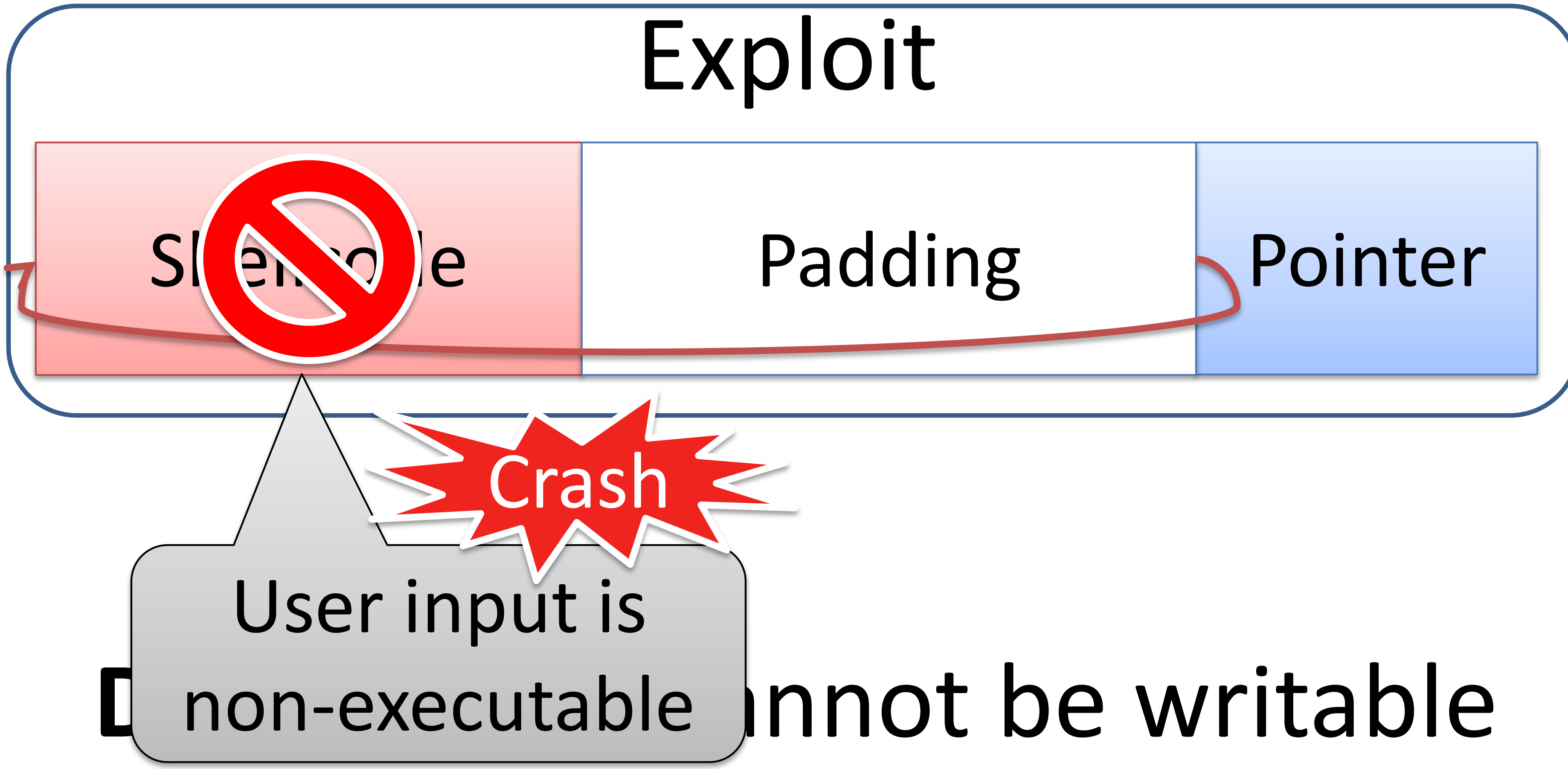
Simple Control-Flow Hijack Exploit



Computation

Control

Data Execution Prevention (DEP)



D nnot be writable
and executable

Bypassing DEP

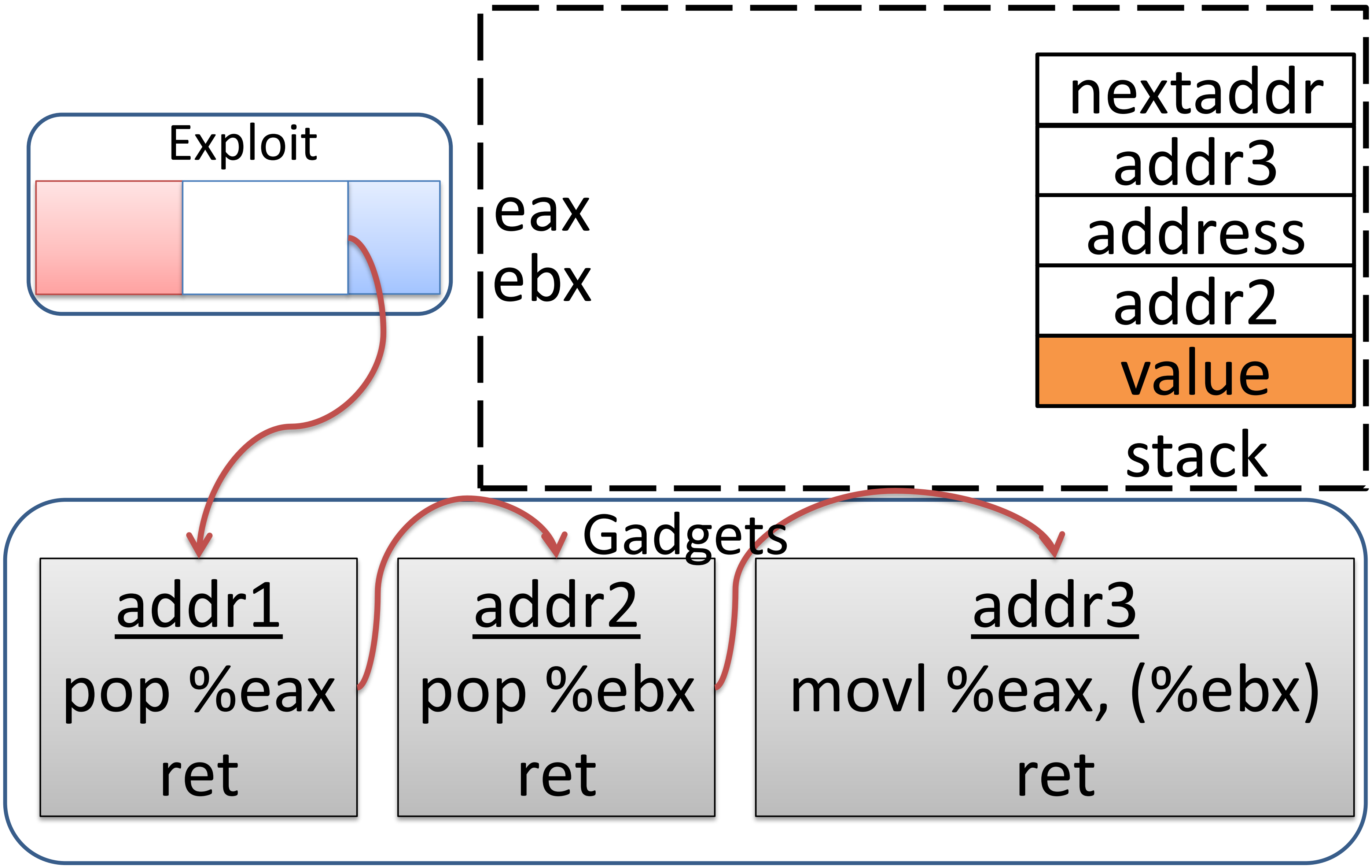
- Goal: Specify exploit computation even when DEP is enabled

- Return-oriented Programming [Shacham 2007]
  - Use existing instructions from program it to create self-contained gadgets
  - Chain gadgets together to encode computation

Return-oriented Programming

Example: How can we write to memory without shellcode?

Return-oriented Programming



Gadgets as Abstractions

- Gadgets are behavior specifications
  - Load constant
  - Store to memory
  - Don’t need to reason about low-level behavior to combine them

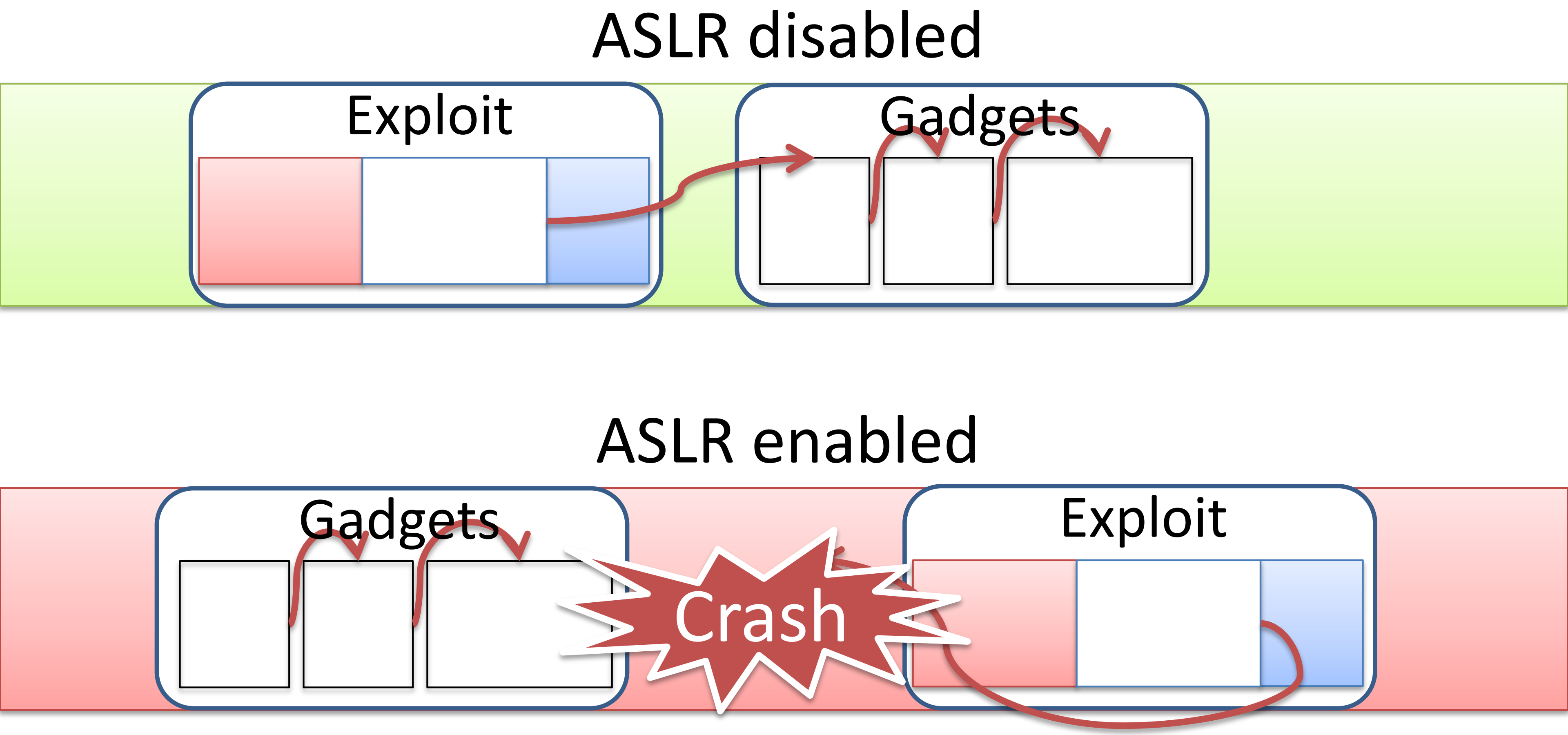
Load to register

Store to memory

Load from memory

Arithmetic

Address Space Layout Randomization (ASLR)



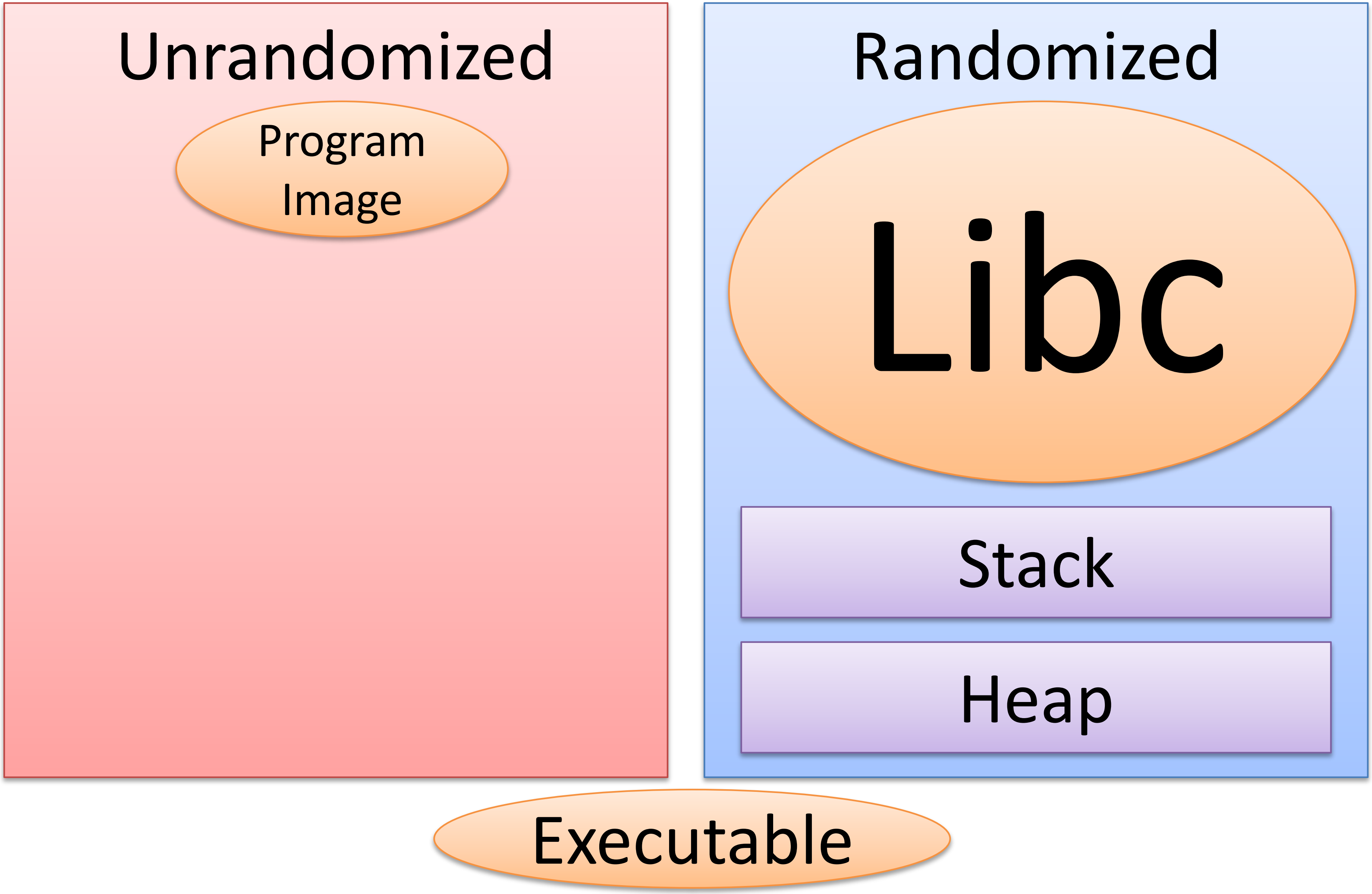
ASLR: Addresses are unpredictable

Return-oriented Programming + ASLR

- Randomized code can't be used for ROP

- But ASLR <u>implementations</u> do not randomize all code…

(Typical) Randomized Code in Linux



Modern Exploitation using ROP

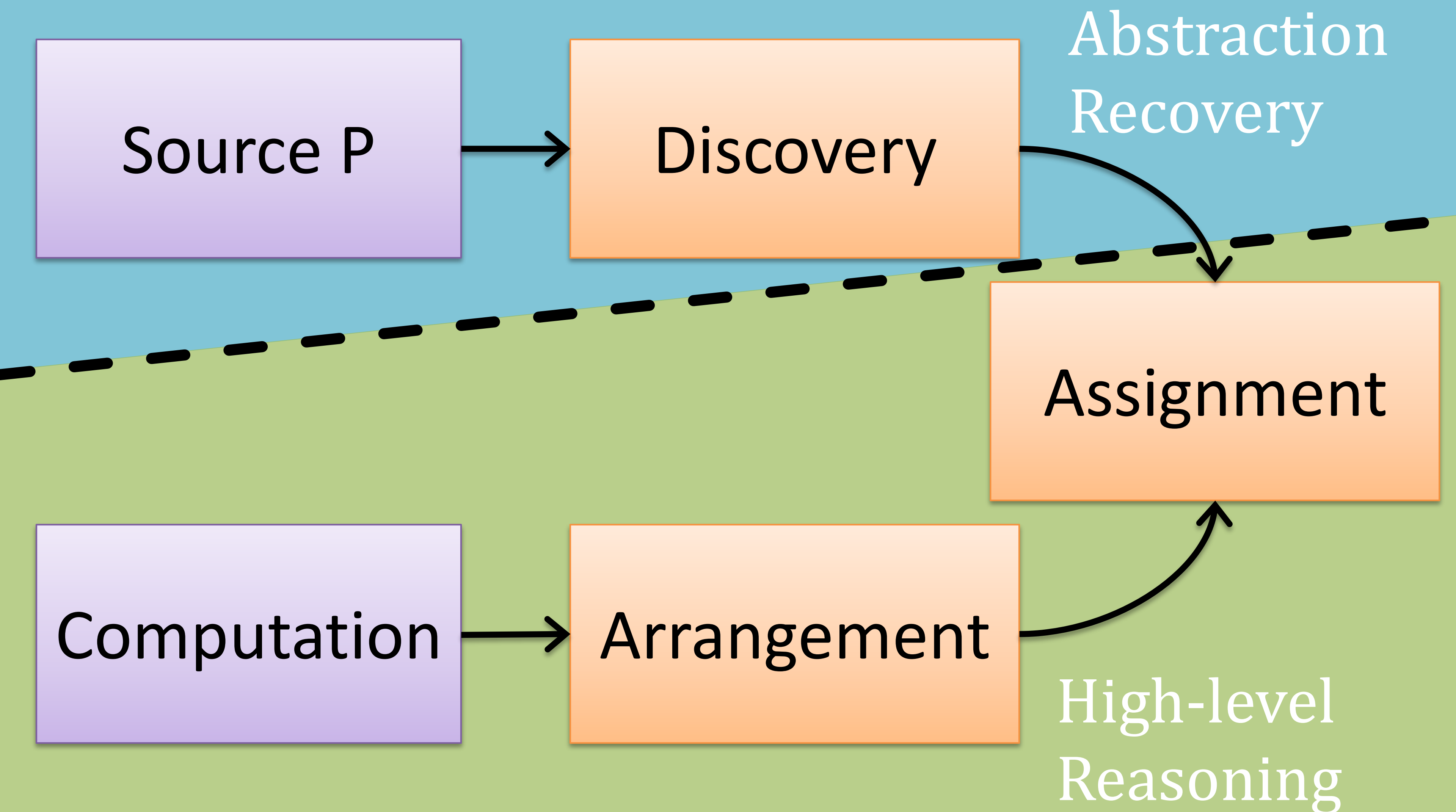
- Program image is often the only unrandomized code
  - Small
  - Program-specific

- How much unrandomized code does an attacker need to use ROP?

~~We need a graduate student with a lot of free time~~
We need automatic ROP techniques that can work with the program image

Q: Automatic ROP System

Q: ROP Overview



Gadget Discovery

- Discovery: Does instruction sequence do something we can use for our computation?
- Fast randomized test for every program location (thousands or millions)

```
sbb %eax, %eax;
neg %eax; ret
```

Randomized Testing

Before

EAX	0x0298a7bc
CF	0x1
ESP	0x81e4f104

```
sbb %eax, %eax;
neg %eax; ret
```

After

EAX	0x1
ESP	0x81e4f108
EBX	0x0298a7bc

OutReg <- InReg

Semantic Definition For Move

Q's Semantic Definitions/ Gadget Types

Gadget Type	Semantic Definition	Real World Example
MoveRegG	Out <- In	xchg %eax, %ebp; ret
LoadConstG	Out <- Constant	pop %ebp; ret
ArithmeticG	Out <- In1 + In2	add %edx, %eax; ret
LoadMemG	Out <- M[Addr + Offset]	movl 0x60(%eax), %eax; ret
StoreMemG	M[Addr + Offset] <- In	mov %dl, 0x13(%eax); ret
ArithmeticLoadG	Out +<- M[Addr + Offset]	add 0x1376dbe4(%ebx), %ecx; (…); ret
ArithmeticStoreG	M[Addr + Offset] +<- In	add %al, 0x5de474c0(%ebp); ret

Randomized Testing

- Randomized testing quickly rules out non-gadgets
  - Fast
  - Enables more expensive second stage
- Second stage: program verification

Connection to Program Verification

```
sum = 0
while (n > 0) {
  sum += n;
  n--;
}
```

$$\text{sum} = \sum_{i=0}^{n} i$$

```
sbb %eax, %eax
neg %eax; ret
```

EAX <- CF

Does the post-condition always hold after executing program?

Gadget Verification

```
sbb %eax, %eax
neg %eax; ret
```

EAX <- CF

Weakest Precondition

Θ

Θ

Validity Check

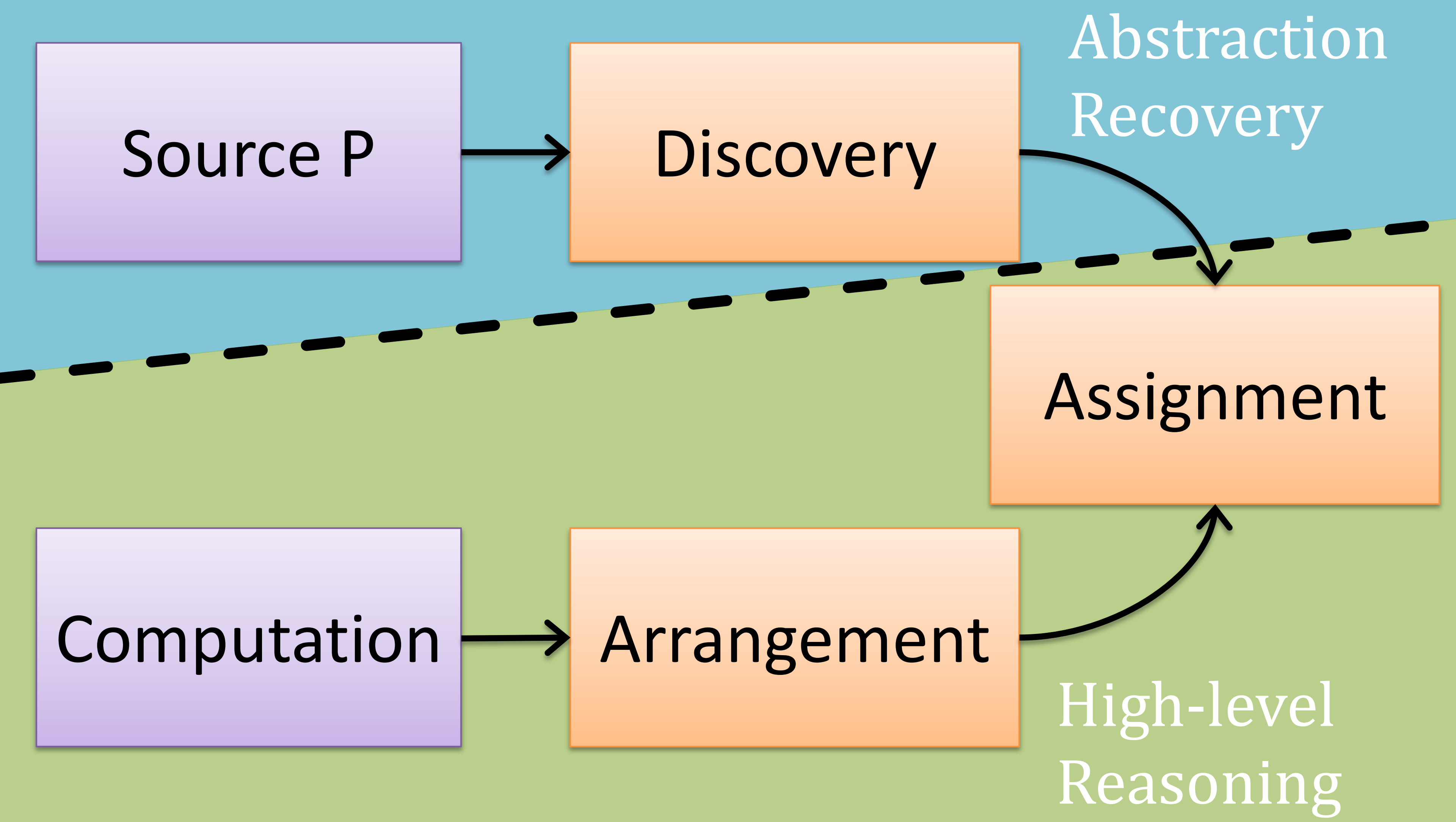
Valid (Gadget)
Invalid (not Gadget)

Semantic-based Gadget Discovery

- Q is better at finding gadgets than I am!

`imul $1, %eax, %ebx` `ret`	Move %eax to %ebx
`lea (%ebx,%ecx,1), %eax` `ret`	Store %ebx+%ecx in %eax
`sbb %eax, %eax; neg %eax` `ret`	Move carry flag to %eax

Q: ROP Overview

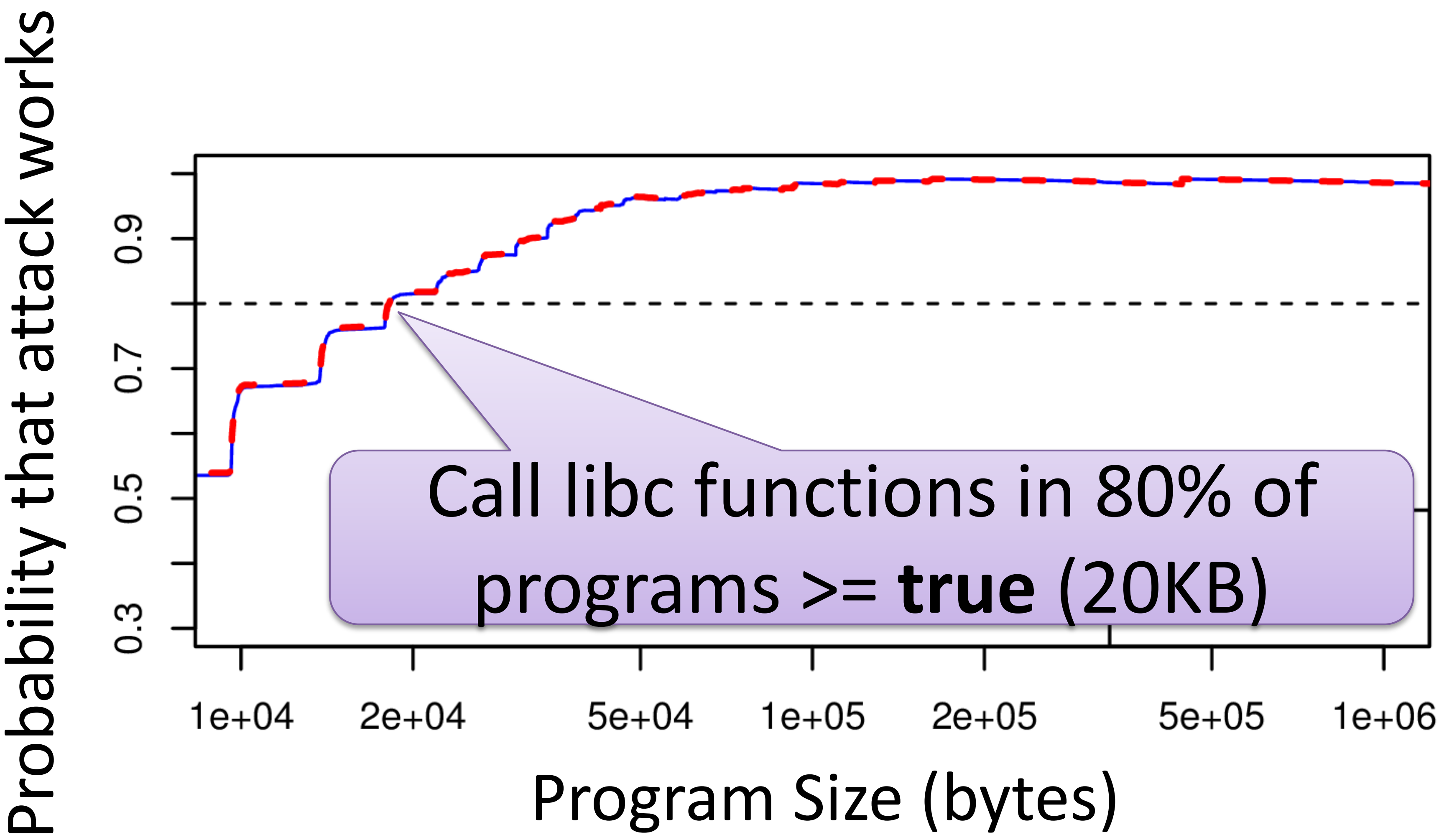


Research Questions

How much unrandomized code is sufficient to create ROP payloads?

- Detail: payloads call any functions in libc
  - `system`, `execv`, `connect`, `mprotect`

ROP Success Probability



Research Questions

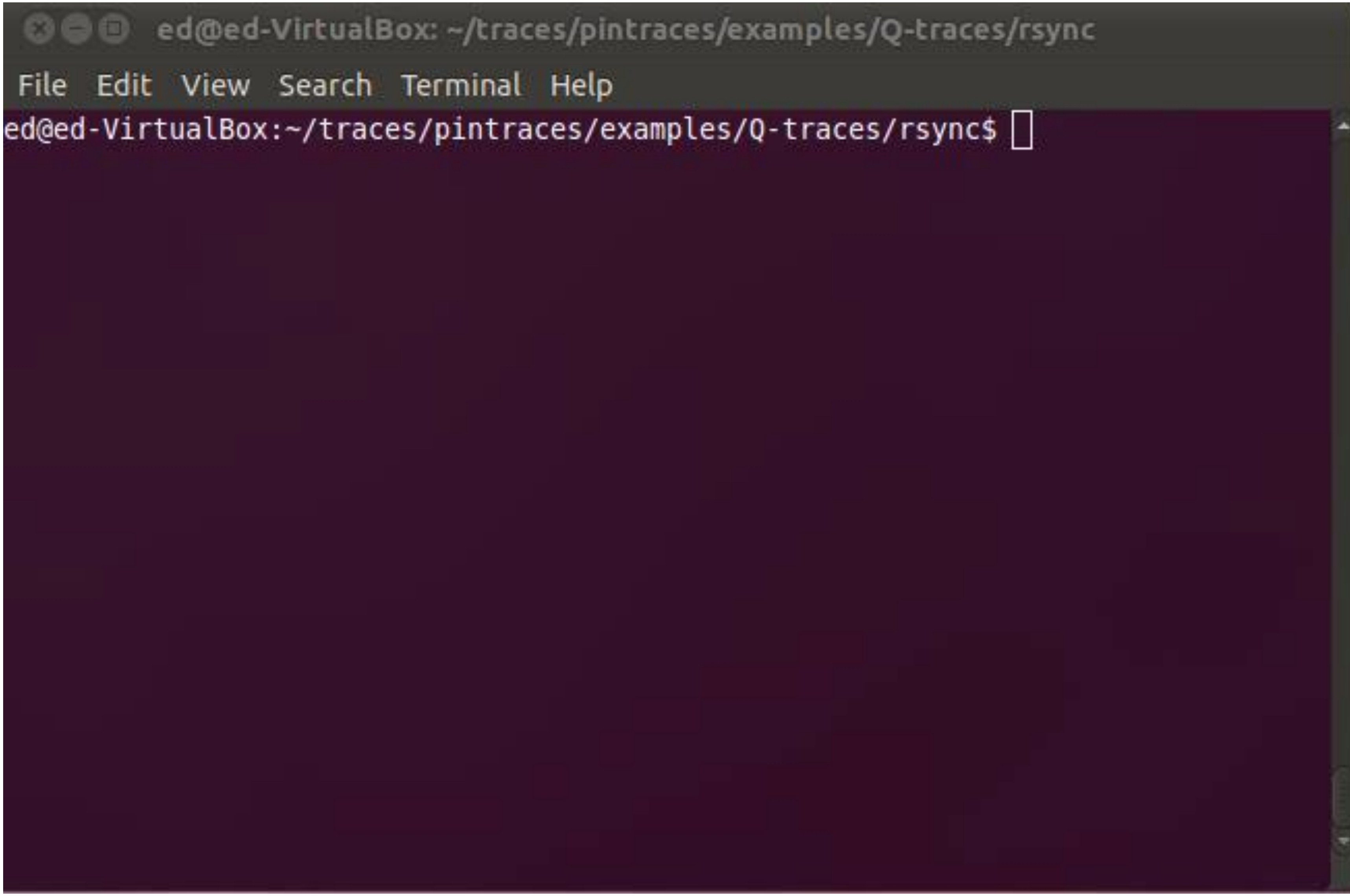
Can Q automatically add ROP payloads to existing exploits for real programs?

Real Exploits

- Q was able to automatically add ROP to nine exploits downloaded from exploit-db.com

Name	Total Time	OS
Free CD to MP3 Converter	130s	Windows 7
Fatplayer	133s	Windows 7
A-PDF Converter	378s	Windows 7
A-PDF Converter (SEH exploit)	357s	Windows 7
MP3 CD Converter Pro	158s	Windows 7
rsync	65s	Linux
opendchub	225s	Linux
gv	237s	Linux
Proftpd	44s	Linux

Demo!


ed@ed-VirtualBox: ~/traces/pintraces/examples/Q-traces/rsync
File Edit View Search Terminal Help
ed@ed-VirtualBox:~/traces/pintraces/examples/Q-traces/rsync$

Outline

- Introduction
- Recovering Abstractions
  - C abstractions (Phoenix Decompiler)
  - Gadget abstractions (Q ROP Compiler)
- Future Work and Conclusions

Abstraction Recovery Questions

- Systems: How do we build systems that
  - Recover abstractions?
  - Use abstractions?
- Theory: When is it possible to recover abstractions?
  - Observable behaviors preserved by compilation
- Scalability: How does recovering and utilizing abstractions improve scalability?
  - ROP (150x)
  - C verification (15x)

Future Work

- Certified decompilation
  - Prove that binary → C translation is correct

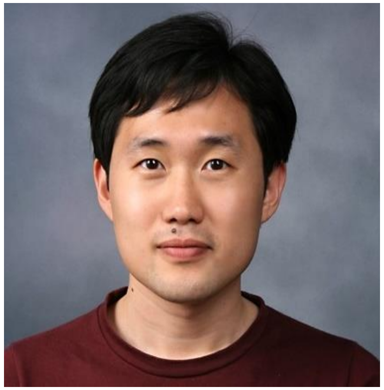
- Optimal abstraction recovery
  - Provably optimal algorithms (i.e., minimum `gotos`)

- Additional abstractions & architectures
  - C++, ARM, Dalvik

Thanks to My Great Co-authors

Thanassis Avgerinos

David Brumley



JongHyup Lee

Maverick Woo

Conclusion

- Abstraction Recovery
  - Recovering abstractions helps static binary analysis
- Phoenix decompiler
  - Goal: Correct, effective decompilation
  - New control-flow structuring algorithm
- Q ROP Compiler
  - Takeaway: Unrandomized code is dangerous
  - 20KB makes DEP+ASLR ineffective

Thanks ☺

- Questions?

Edward J. Schwartz

eschwartz@cert.org

http://www.ece.cmu.edu/~ejschwar