TIE: Principled Reverse Engineering of Types in Binary Programs

JongHyup Lee, Thanassis Avgerinos, and David Brumley

Reverse engineering on binary programs

1. Code structure
2. Data abstractions

TIE

Goal:

Reconstruct
data abstractions
conservatively
and
accurately

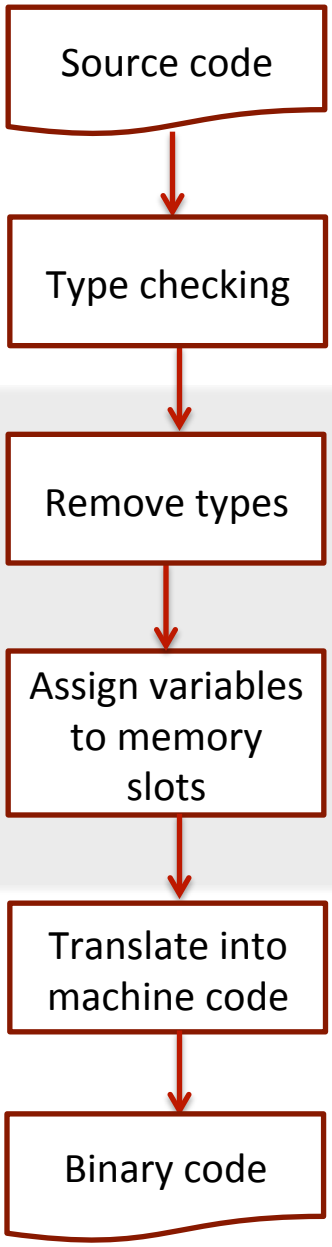
Goal:

Reconstruct
data abstractions
conservatively
and
accurately


All types are lost.

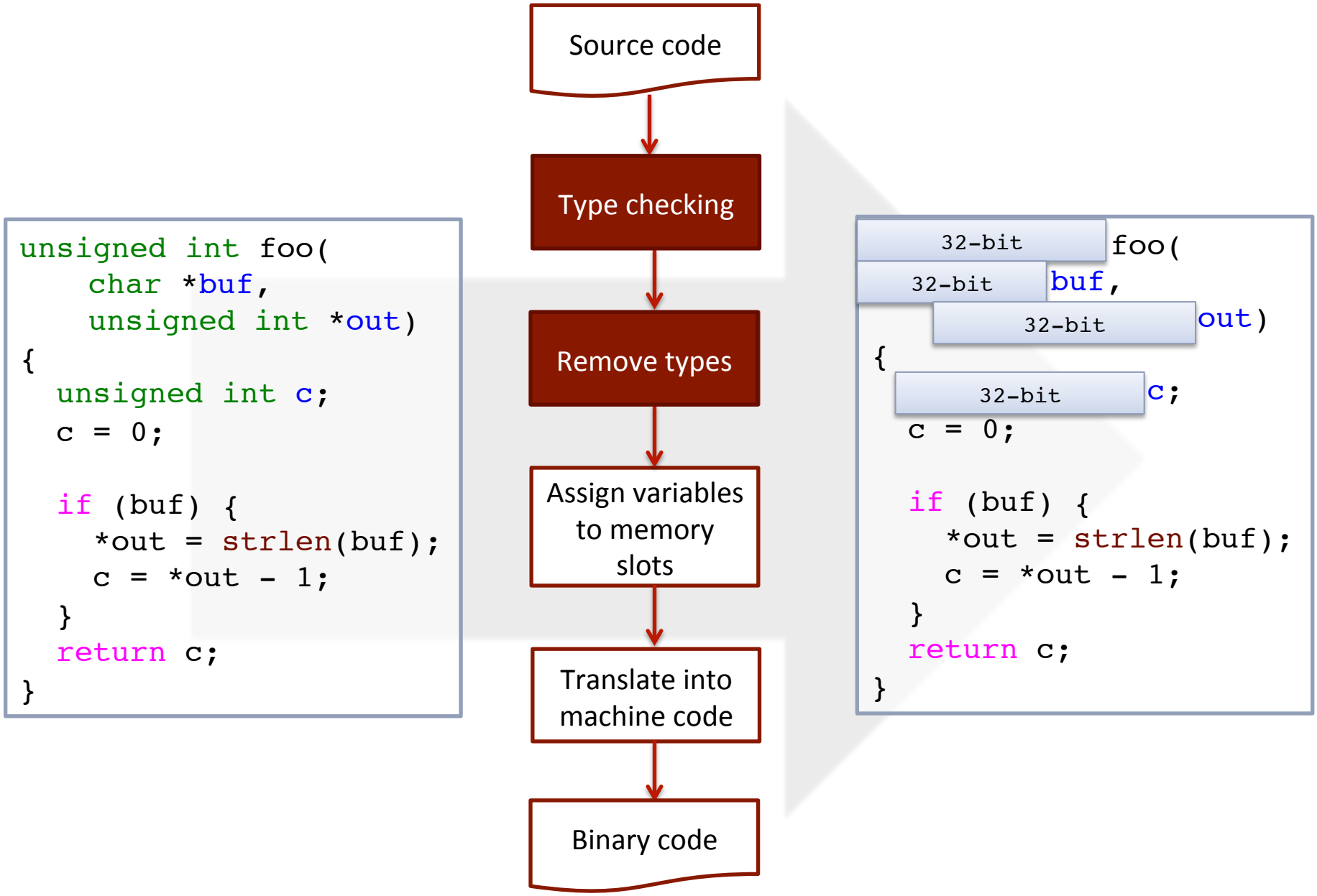
Compilation

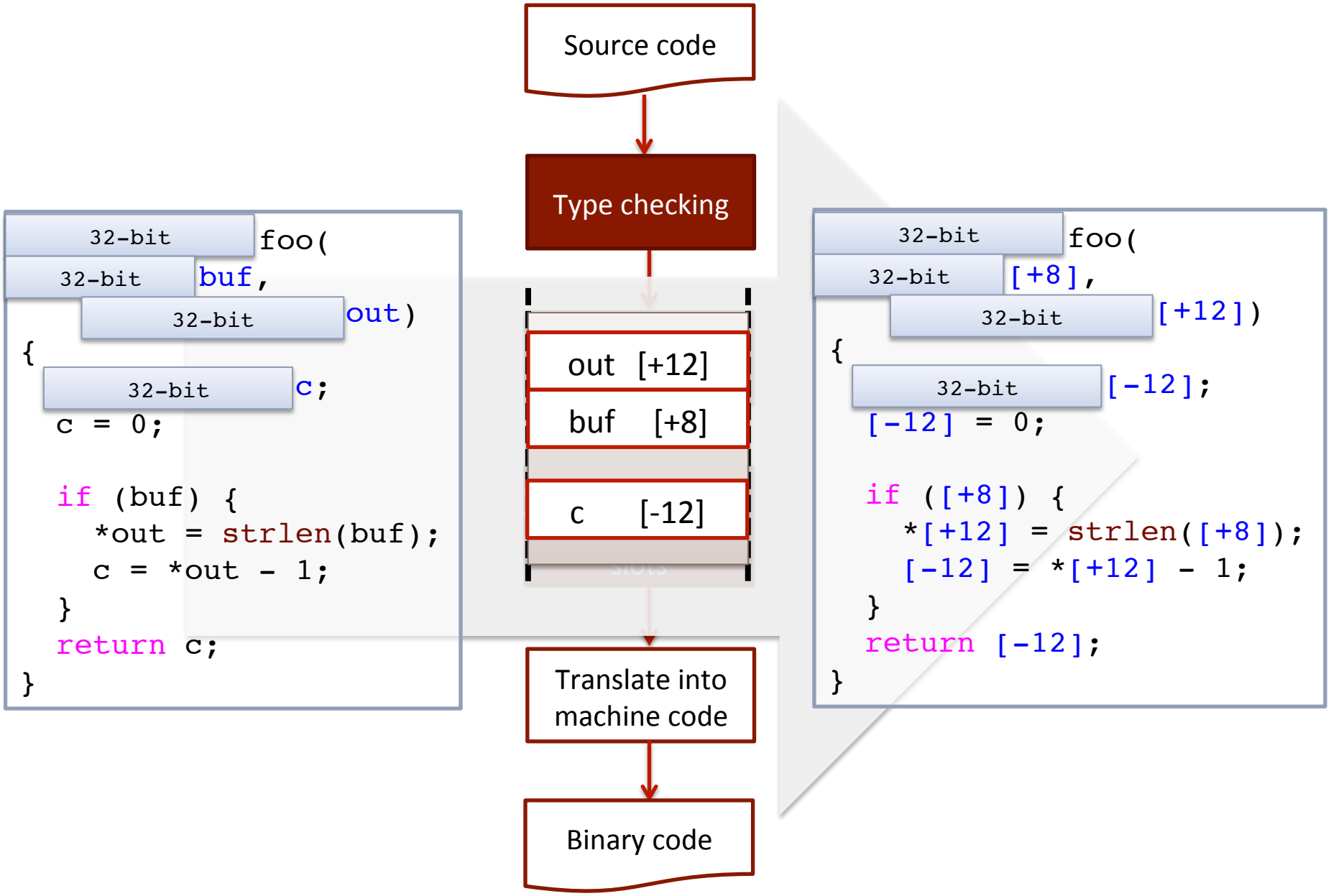
```
unsigned int foo(
    char *buf,
    unsigned int *out)
{
  unsigned int c;
  c = 0;

  if (buf) {
    *out = strlen(buf);
    c = *out - 1;
  }
  return c;
}
```

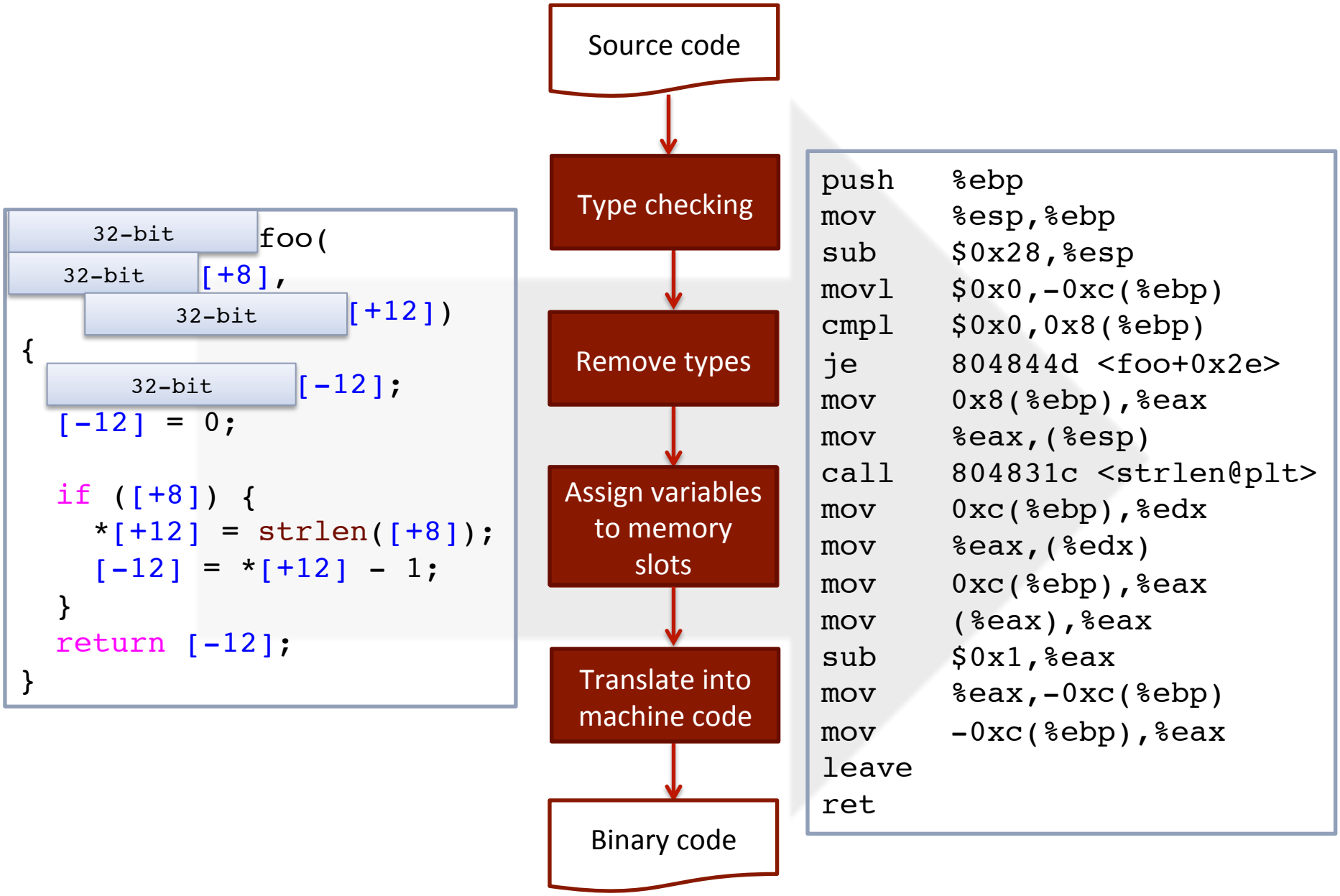


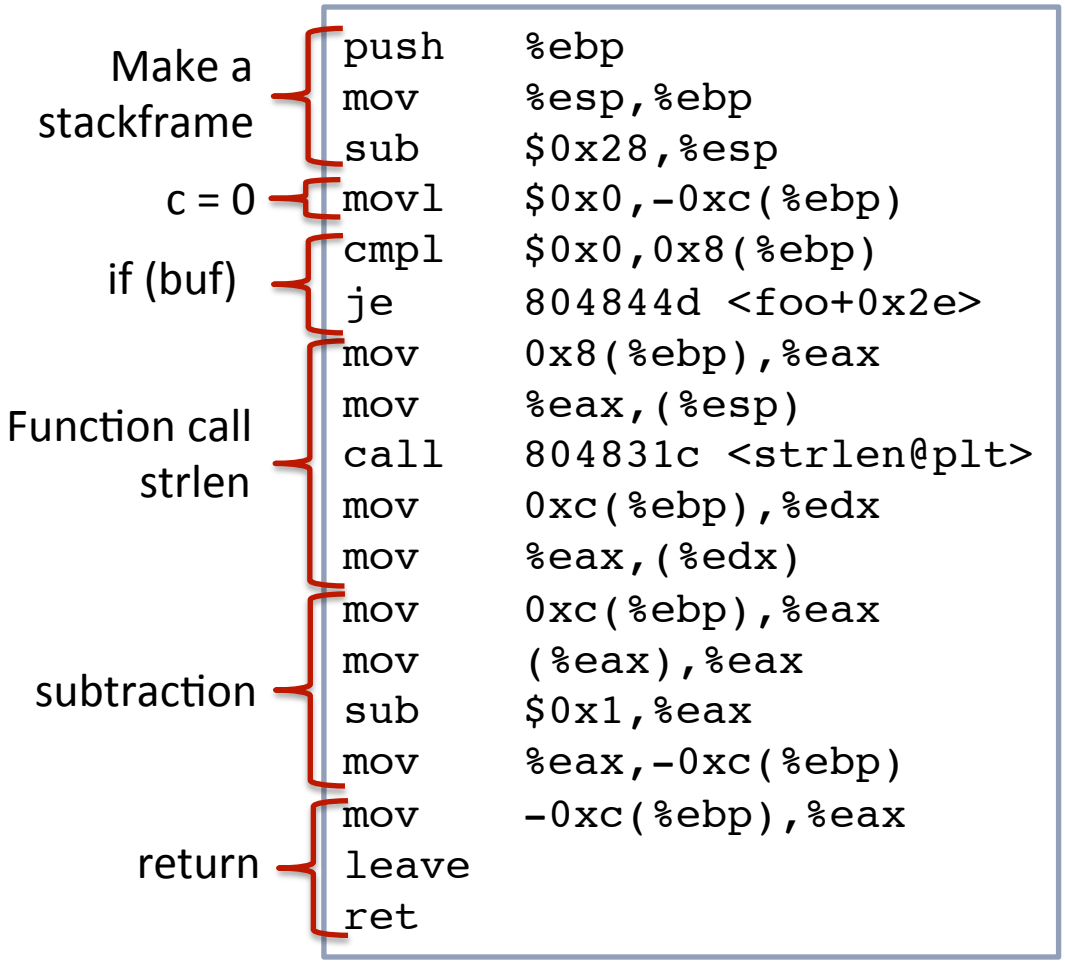

```
push   %ebp
mov    %esp,%ebp
sub    $0x28,%esp
movl   $0x0,-0xc(%ebp)
cmpl   $0x0,0x8(%ebp)
je     804844d <foo+0x2e>
mov    0x8(%ebp),%eax
mov    %eax,(%esp)
call   804831c <strlen@plt>
mov    0xc(%ebp),%edx
mov    %eax,(%edx)
mov    0xc(%ebp),%eax
mov    (%eax),%eax
sub    $0x1,%eax
mov    %eax,-0xc(%ebp)
mov    -0xc(%ebp),%eax
leave
ret
```

?


unsigned int foo(
char *buf,
unsigned int *out)
{
unsigned int c;
c = 0;
if (buf) {
*out = strlen(buf);
c = *out - 1;
}
return c;
}
Source code
Type checking
Remove types
Assign variables to memory slots
Translate into machine code
Binary code
32-bit
foo(
32-bit
buf,
32-bit
out)
{
32-bit
c;
c = 0;
if (buf) {
*out = strlen(buf);
c = *out - 1;
}
return c;
}

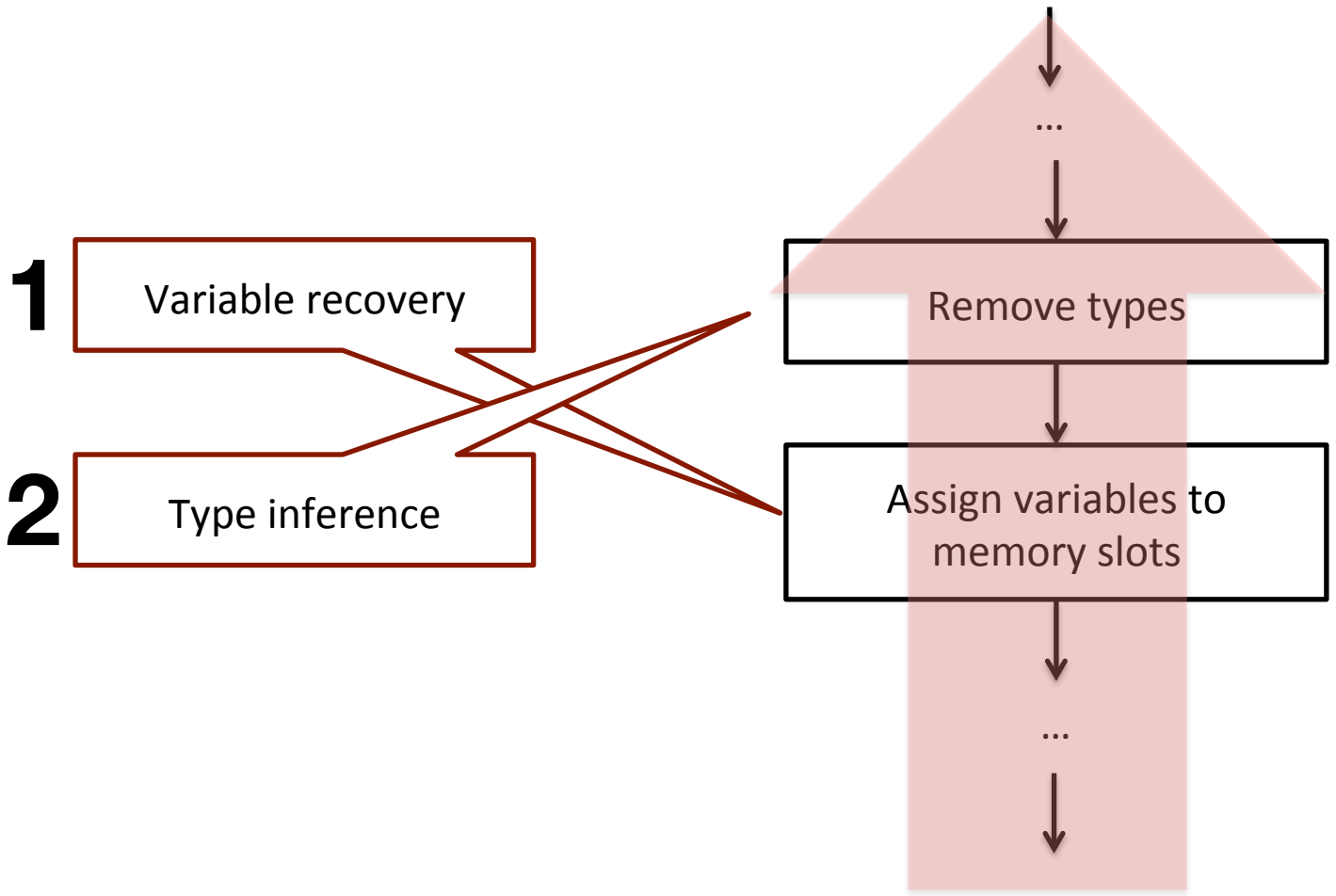

Source code
Type checking
32-bit foo(
32-bit buf,
32-bit out)
{
32-bit c;
c = 0;
if (buf) {
*out = strlen(buf);
c = *out - 1;
}
return c;
}
out [+12]
buf [+8]
c [-12]
Translate into machine code
Binary code
32-bit foo(
32-bit [+8],
32-bit [+12])
{
32-bit [-12];
[-12] = 0;
if ([+8]) {
*[+12] = strlen([+8]);
[-12] = *[+12] - 1;
}
return [-12];
}


Source code
Type checking
Remove types
Assign variables to memory slots
Translate into machine code
Binary code
32-bit foo(
32-bit [+8],
32-bit [+12])
{
32-bit [-12];
[-12] = 0;
if ([+8]) {
*[+12] = strlen([+8]);
[-12] = *[+12] - 1;
}
return [-12];
}
push %ebp
mov %esp,%ebp
sub $0x28,%esp
movl $0x0,-0xc(%ebp)
cmpl $0x0,0x8(%ebp)
je 804844d <foo+0x2e>
mov 0x8(%ebp),%eax
mov %eax,(%esp)
call 804831c <strlen@plt>
mov 0xc(%ebp),%edx
mov %eax,(%edx)
mov 0xc(%ebp),%eax
mov (%eax),%eax
sub $0x1,%eax
mov %eax,-0xc(%ebp)
mov -0xc(%ebp),%eax
leave
ret

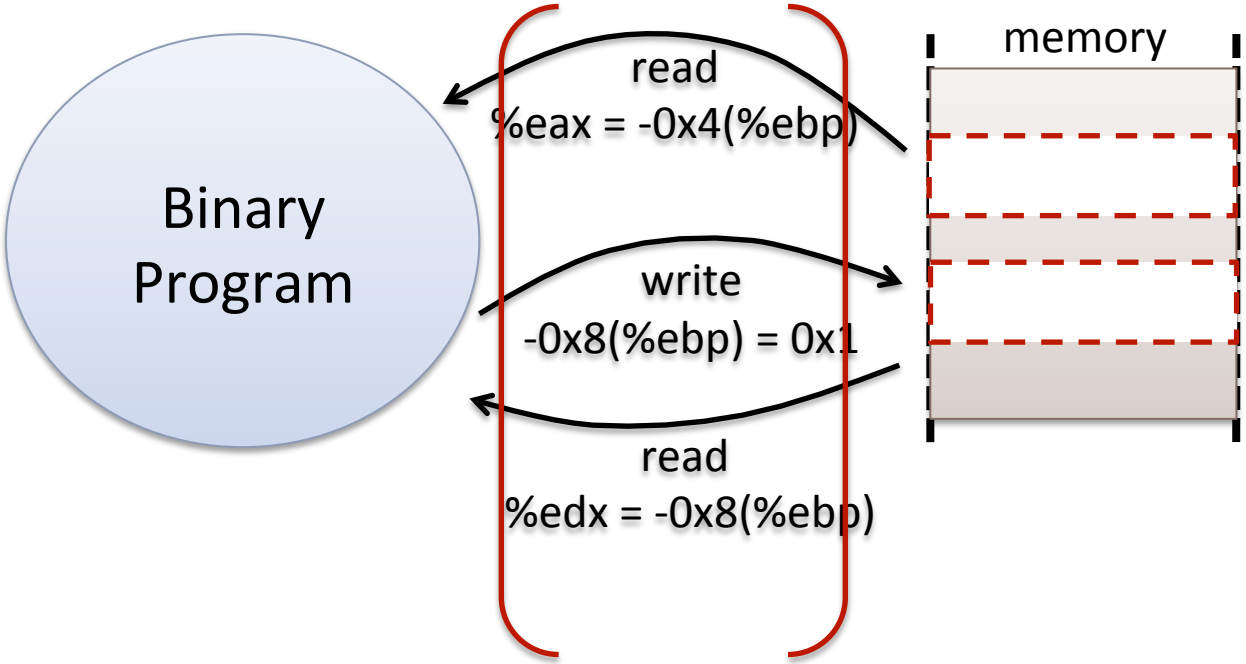


No types, no variables

TIE


1
Variable recovery
2
Type inference
...
Remove types
Assign variables to memory slots
...

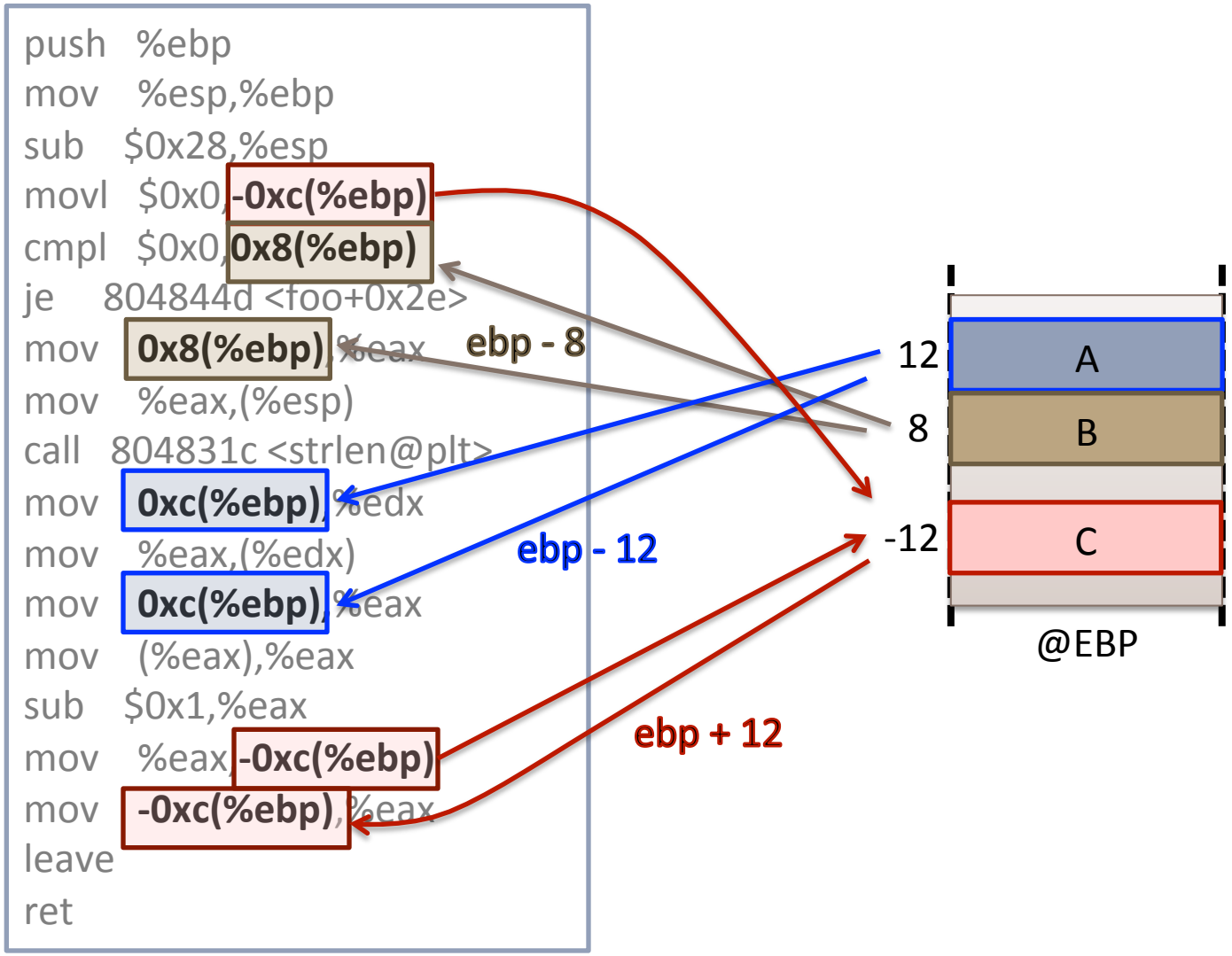
1. Variable Recovery



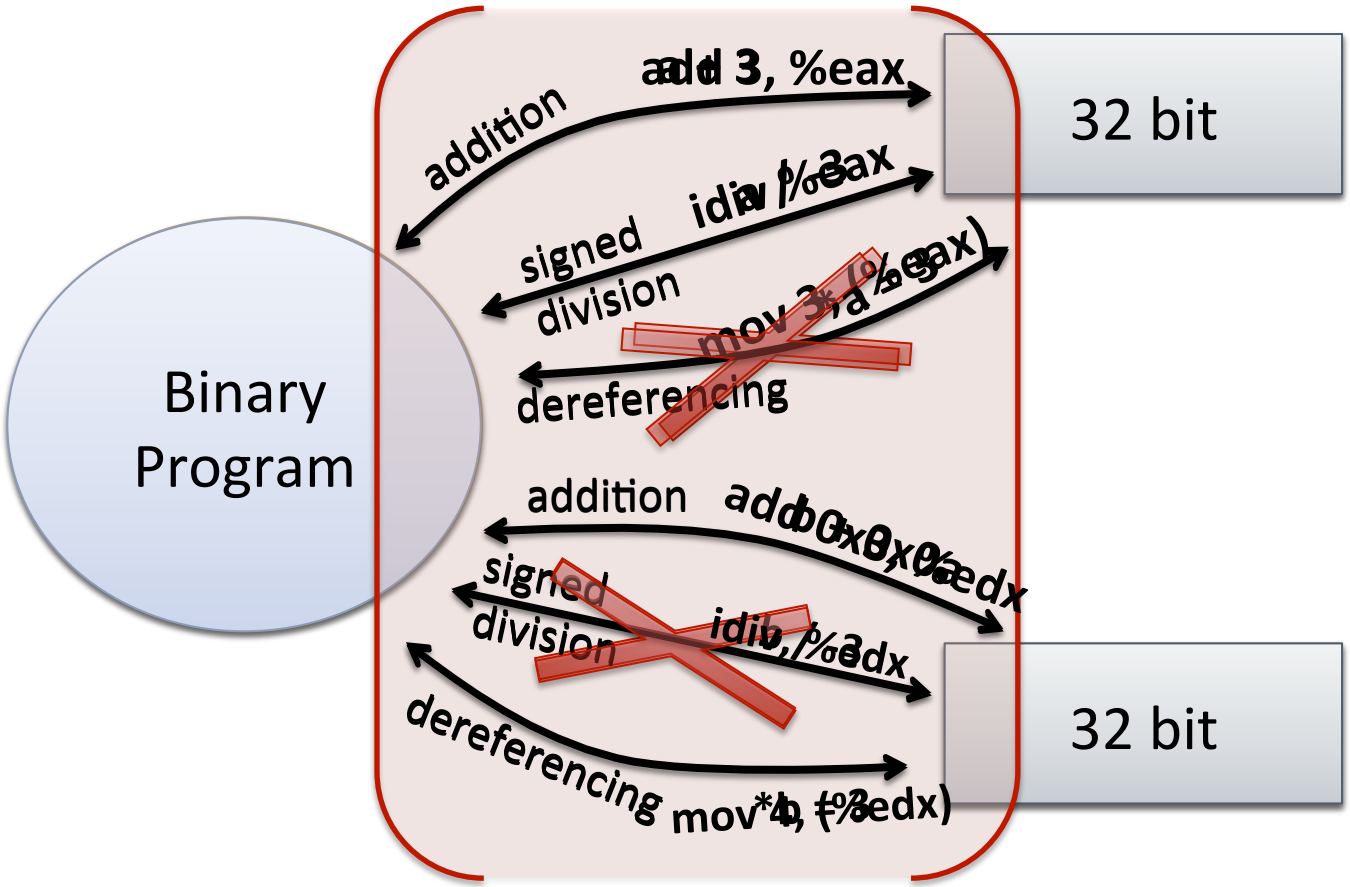
Analyze the value of "addresses"

A variant of VSA (Value Set Analysis) [Balakrishnan, 2007]

1. Variable Recovery



2. Type Inference



Behavior has not changed!

ANALYZE the behavior on variables

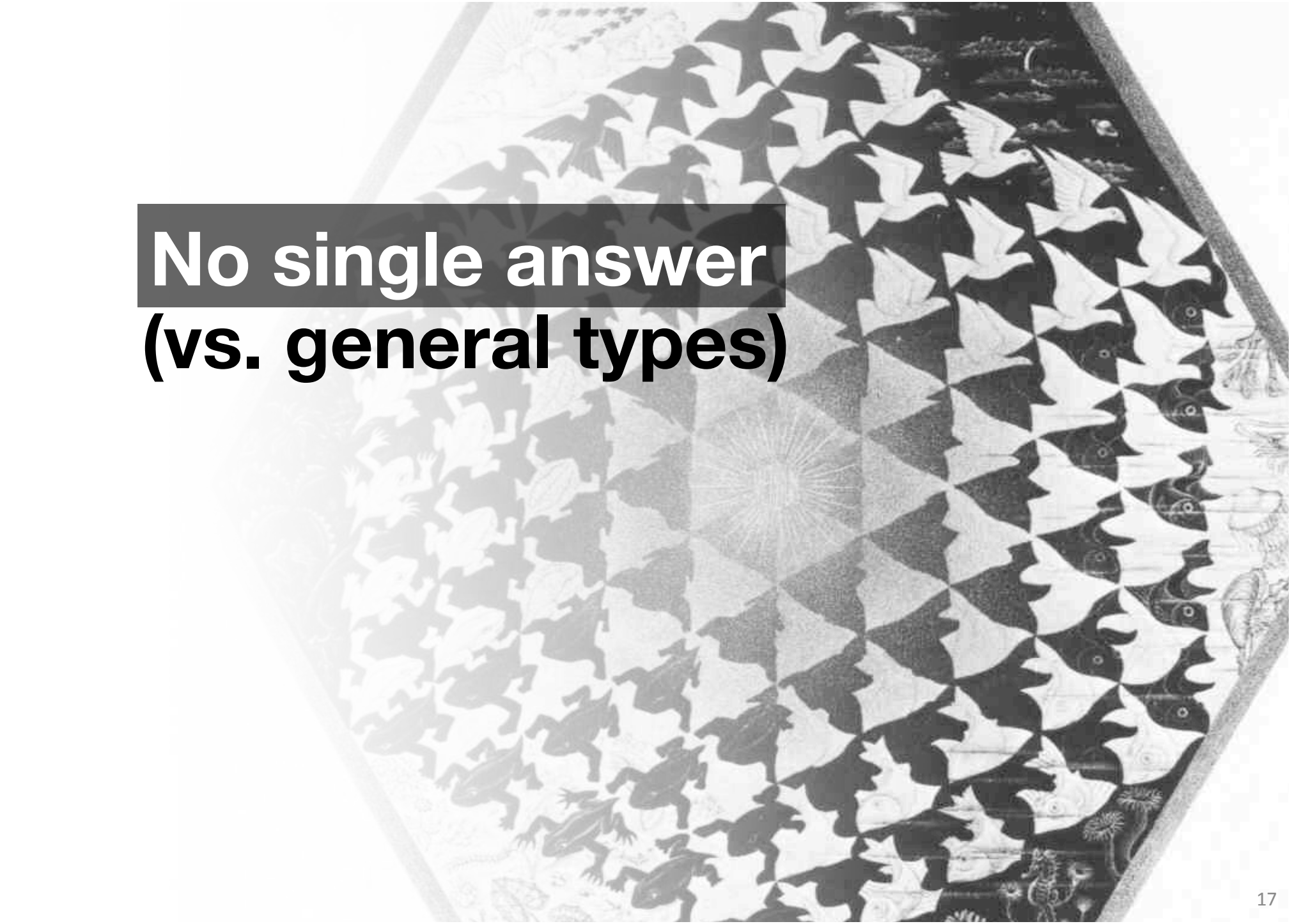
COLLECT the clues

INFER the type of variables

Goal:

Reconstruct
data abstractions
conservatively
and
accurately

No single answer
(vs. general types)



Multiple types are possible

```
int sum(int a, int b)
{
  int c;
  c = a + b;
  return c;
}
```

≠

```
char * advance(char * str,
    unsigned int m)
{
  char * tmp;
  tmp = str + m;
  return tmp;
}
```

Make a stackframe

1st arg + 2nd arg

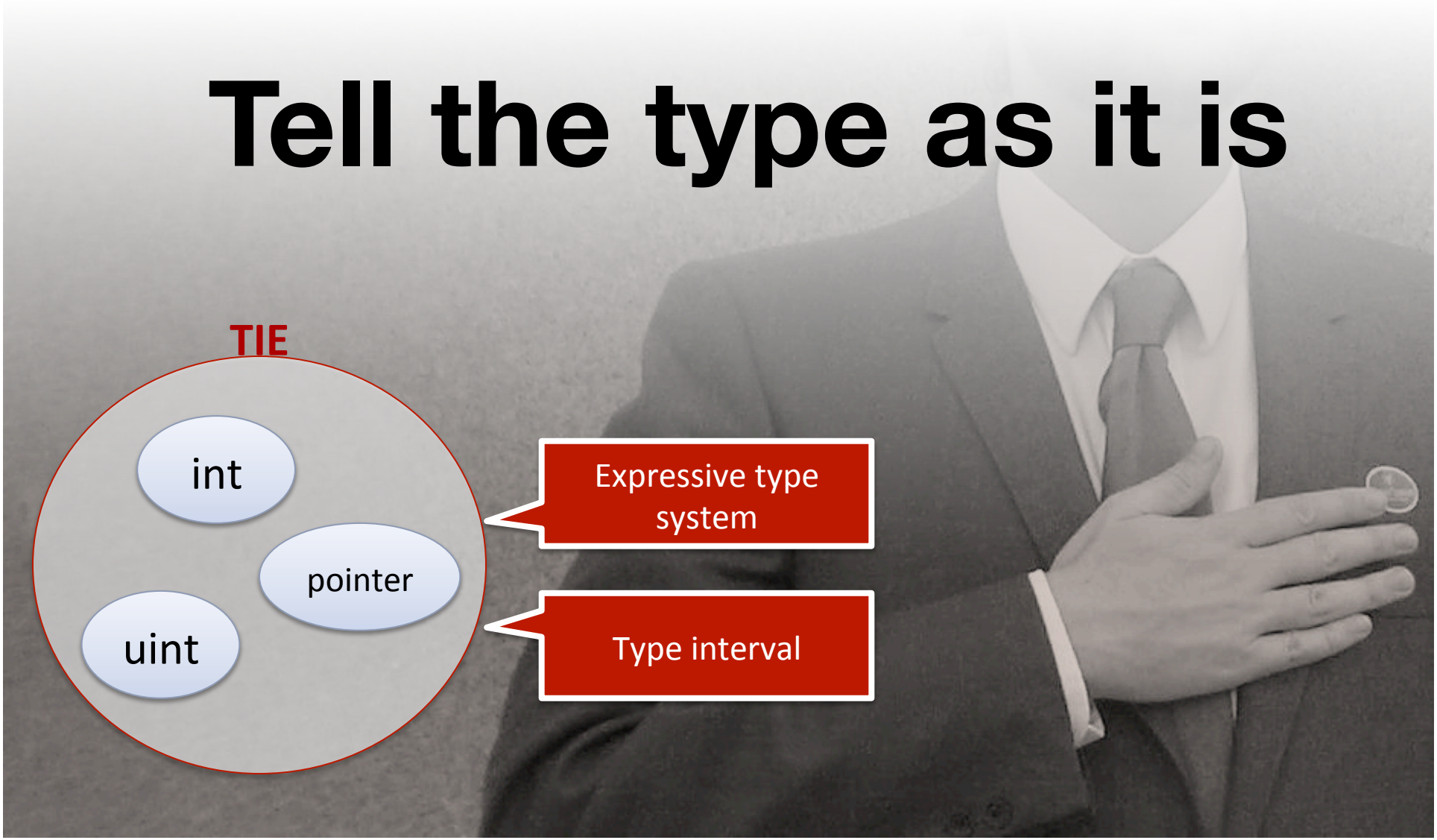
Return the result

```
push   %ebp
mov    %esp,%ebp
sub    $0x10,%esp
mov    0xc(%ebp),%eax
mov    0x8(%ebp),%edx
add    %edx,%eax
mov    %eax,-0x4(%ebp)
mov    -0x4(%ebp),%eax
leave
ret
```

int,
uint,
pointer of …

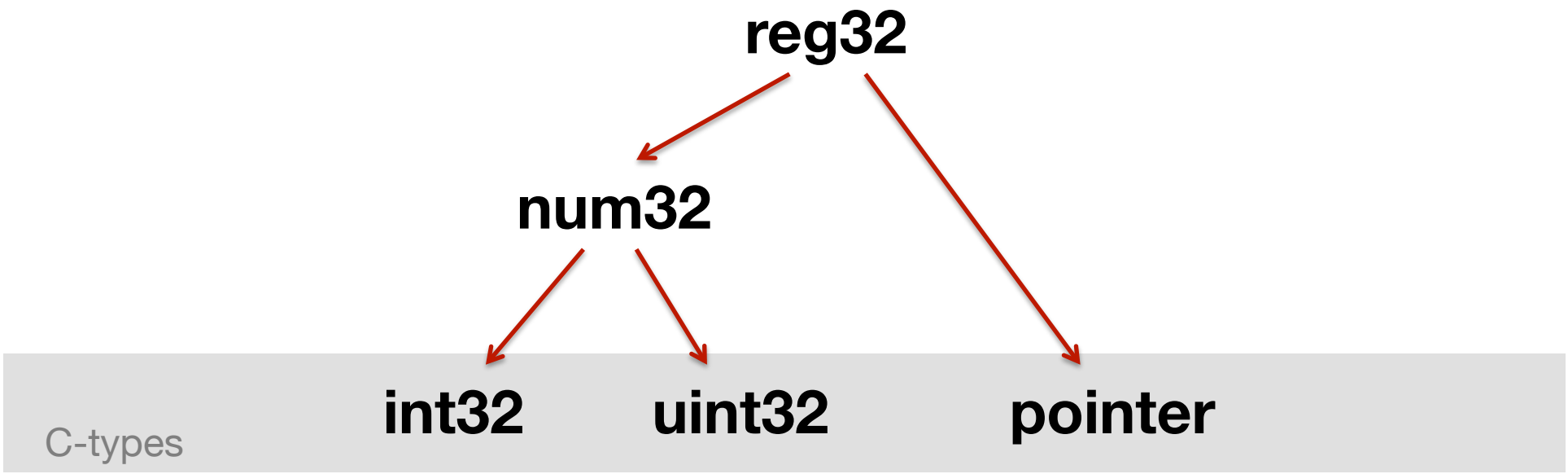

GUESS!
2
int
uint
pointer

Tell the type as it is



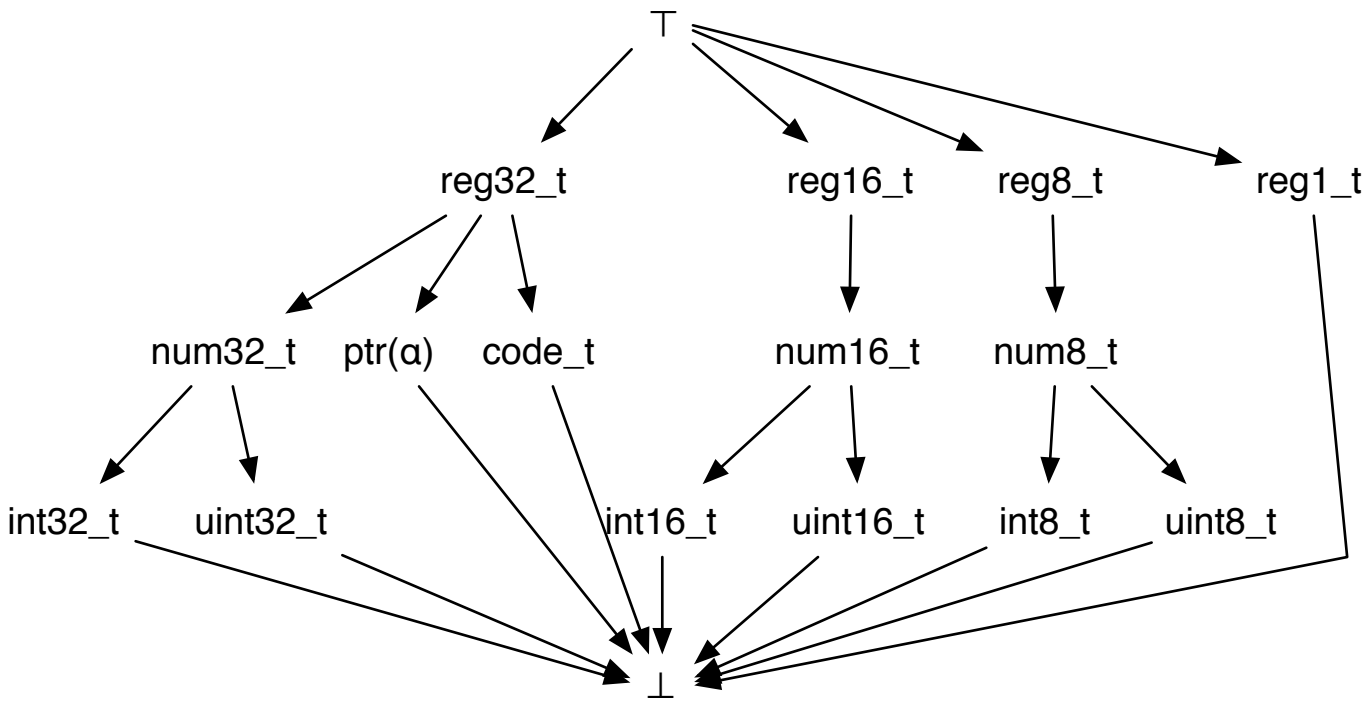
TIE type system

allows us to express the type of variables as they are used



Type lattice

- Basic types



Type specificity

A <: B

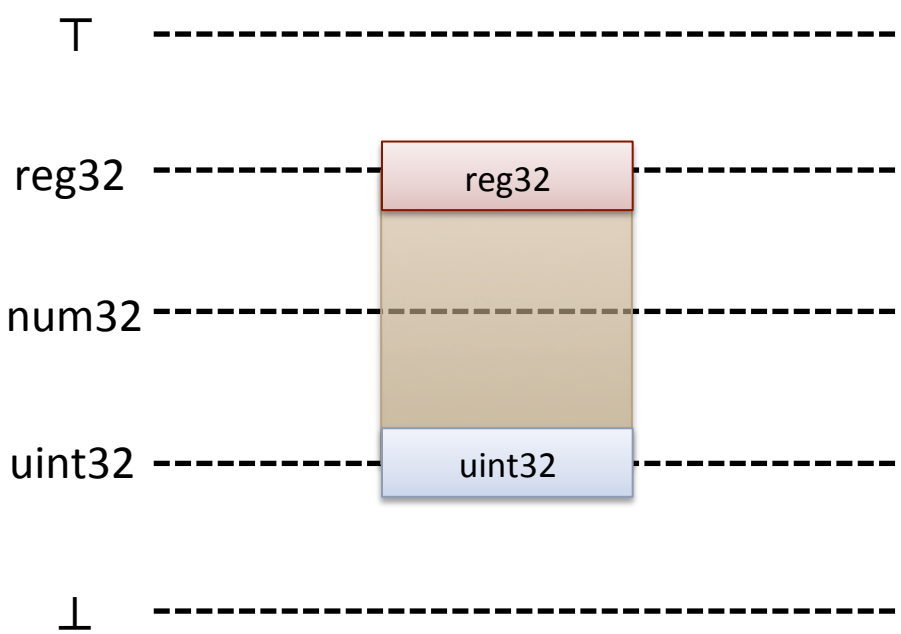
A is a subtype of B

A is more specific than B

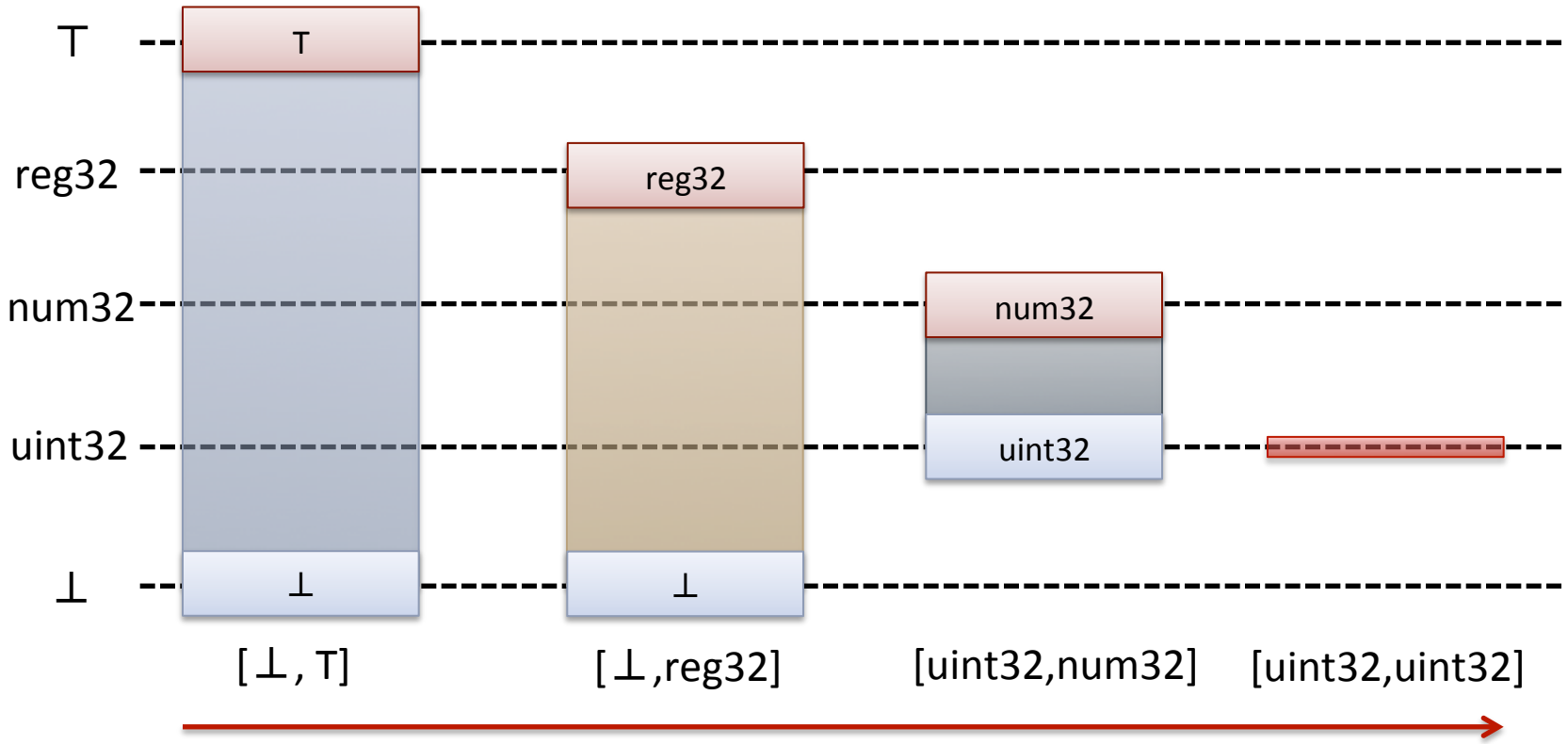
Ex) int32 <: num32

Type interval

“How much does a binary program tell us about a variable?”



Type interval

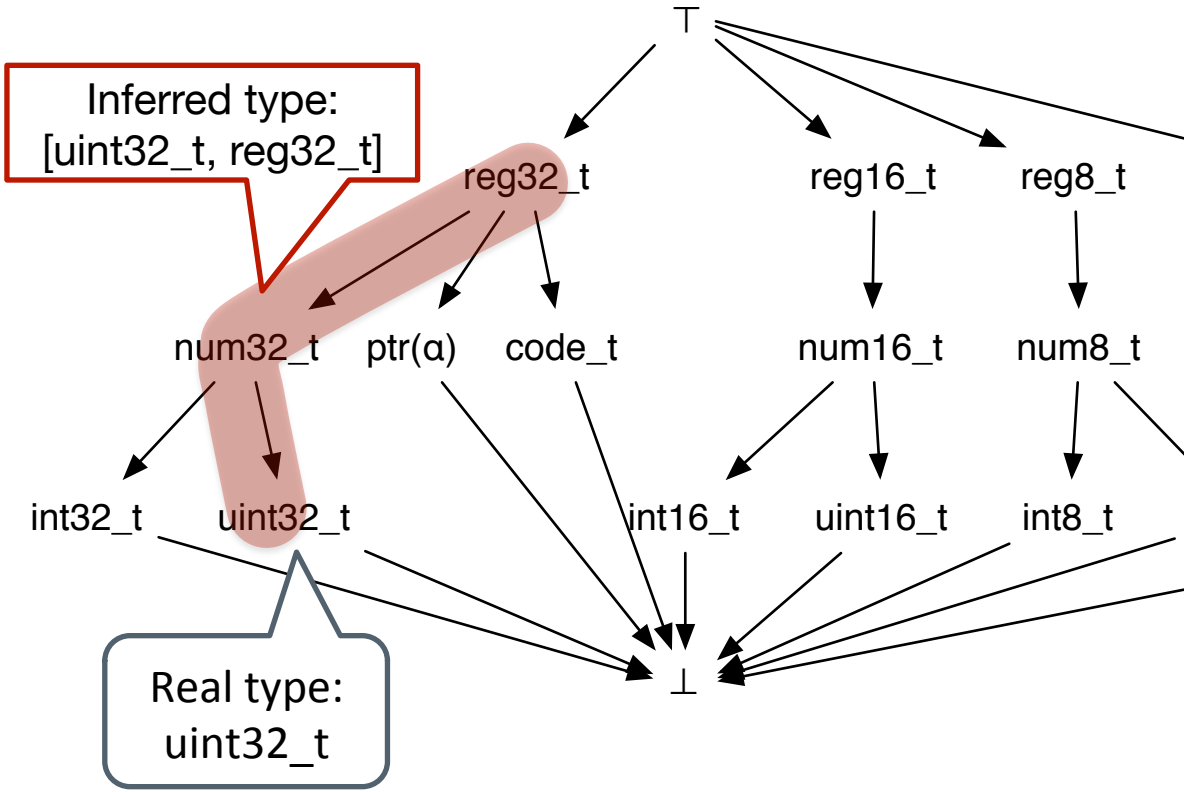


refined
as we know more about the variable

Conservativeness of result

“Is the real type within the inferred type interval?”≈

Conservative ←



Goal:

Reconstruct
data abstractions
conservatively
and
accurately

ANALYZE the behavior on variables

COLLECT every clue

INFER the type of variables

ANALYZE the behavior on variables

Type constraints generation rules

GENERATE type constraints

SOLVE type constraints

Type Constraints from Usage Clues

`movl` (e_1), e_2

Usage clue		Type Constraint
value = load(index,s)	Memory load	'index' is a pointer of 'value' and the size of 'value' is 's'.

e_2 = load(e_1 ,32)

$$[e_1] = \mathrm{ptr}([e_2]) \wedge [e_2] <: \mathrm{reg32\_t}$$

type variable for e_1

Equality

conjunctive

Subtype relationship

Type Constraints from Usage Clues

Usage clue		Type constraint
$c = a +_{32} b$	32-bit addition	('a': number, 'b': number, 'c':number) or ('a': pointer, 'b': number, 'c':pointer) or ('a': number, 'b': pointer, 'c':pointer)

$$e_3 = e_1 +_{32} e_2$$

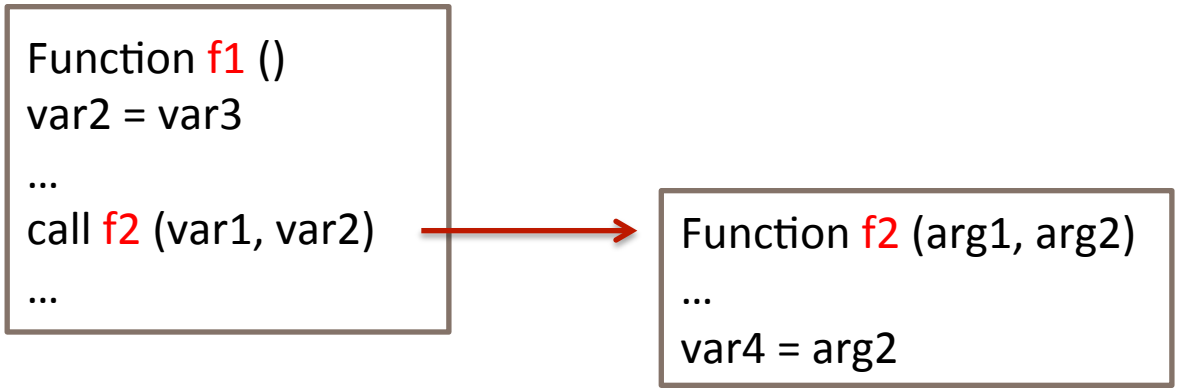
conjunctive

$$([e_1] <: \text{ptr}(\alpha) \wedge [e_2] <: \text{num32\_t} \wedge \text{ptr}(\beta) <: [e_3])$$
$$\vee\ ([e_1] <: \text{num32\_t} \wedge [e_2] <: \text{ptr}(\alpha) \wedge \text{ptr}(\beta) <: [e_3])$$
$$\vee \cdots$$

disjunctive

Inter-procedural Type Inference

- Type of passing arguments = Type of passed arguments



[var1] = [arg1]

[var2] = [arg2]

Type Constraints from Well-known Functions

a = `strlen`(b)

uint32_t ↑ a

char * ↑ b

[a] = uint32_t
[b] = char *

`strcpy`(d, s)

char * ↑ d, char * ↑ s

[d] = char *
[s] = char *

ANALYZE the behavior on variables

COLLECT every clue

GENERATE type constraints

SOLVE type constraints

Solving type constraints

- Equality, A = B
  - Unification
- Subtype relationship, A <: B
  - Closure algorithm
- Conjunctive, A ∧ B
  - Solve all
- Disjunctive, A ∨ B
  - Merge compatible terms

Equality constraints (unification)

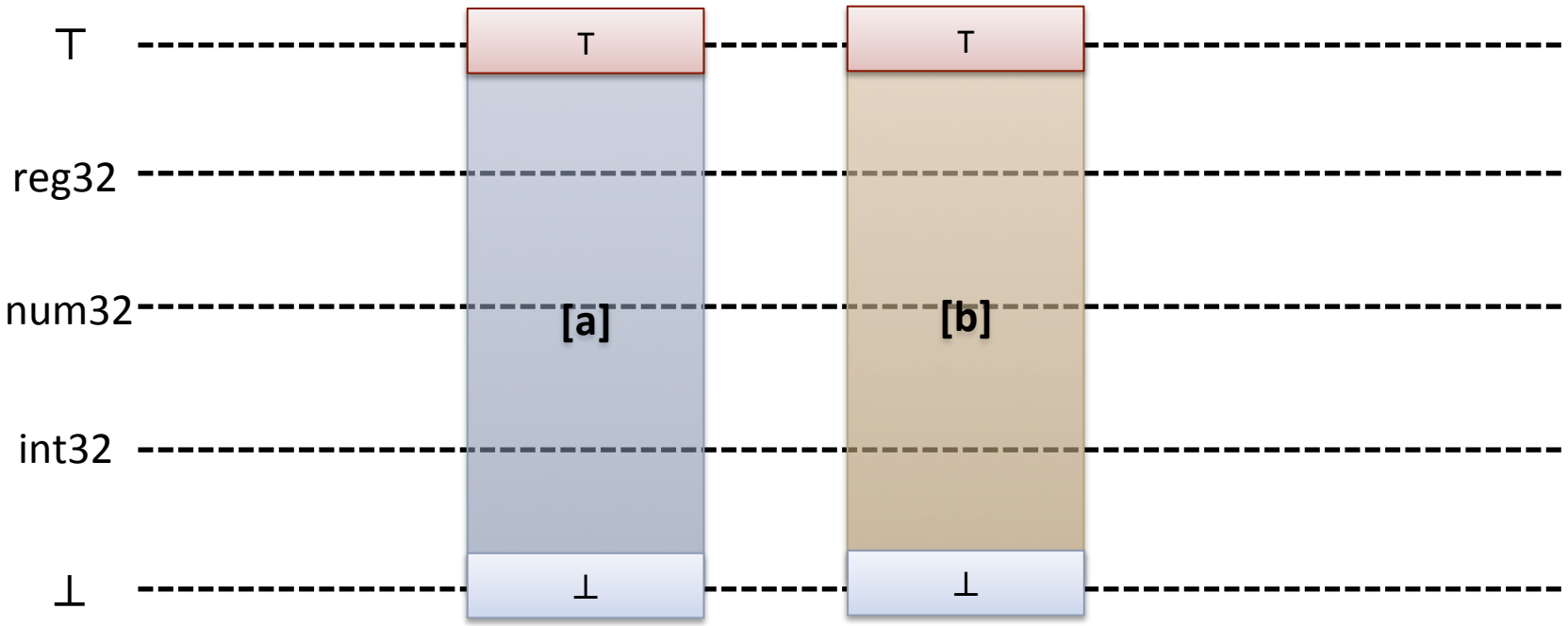
[a] = ptr(uint32)

uint32 = uint32

uint32 = uint32

Subtype relationship constraints (closure alg.)

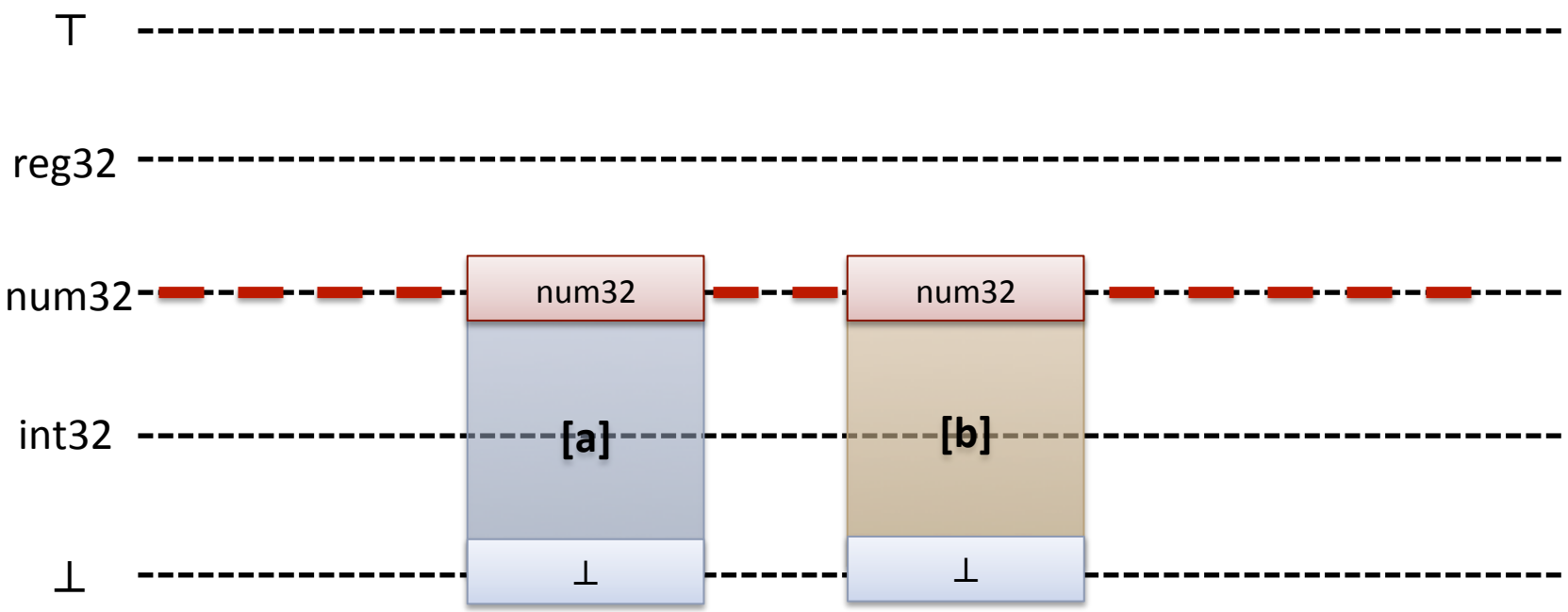
int32_t <: [a] <: [b] <: num32_t



Subtype relationship constraints (closure alg.)

int32_t <: [a] <: [b] <: num32_t

[a] <: [b] <: num32_t

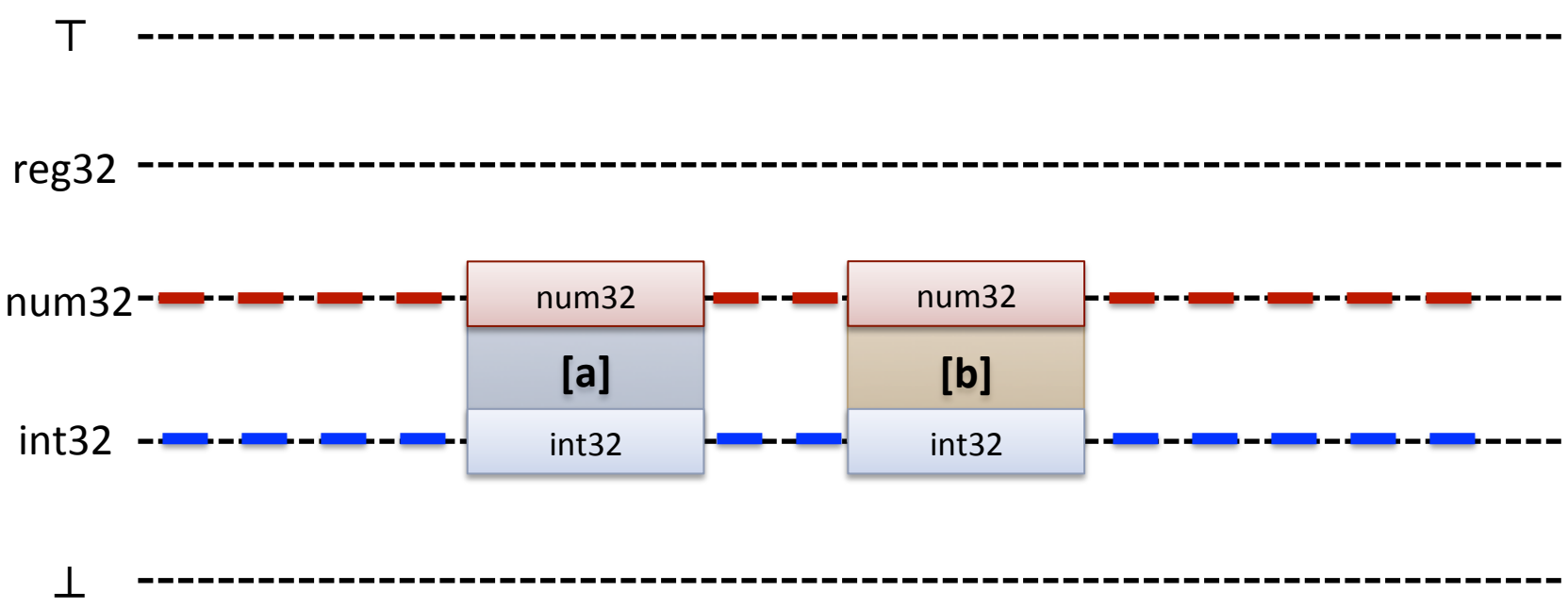


Subtype relationship constraints (closure alg.)

int32_t <: [a] <: [b] <: num32_t

[a] <: [b] <: num32_t

int32_t <: [a] <: [b]



Rest of the talk

- Limitations
- Related work
- Evaluation

Limitations

- Works on regular programs compiled from C code
  - Not very informative for irregular programs
- Infers types as what is in the TIE type system only
  - Extendable

Related work

Hex-Rays	TIE
?	Principled reverse engineering - Well defined process - Type theory

REWARDS	TIE
- Dynamic analysis only - Type propagation from type sinks (unification)	- Static + dynamic - Type inference with more expressive type system (unification + closure alg.)

Boomerang [RE2005] Laika[OSDI2008] Tupni[CCS2008]

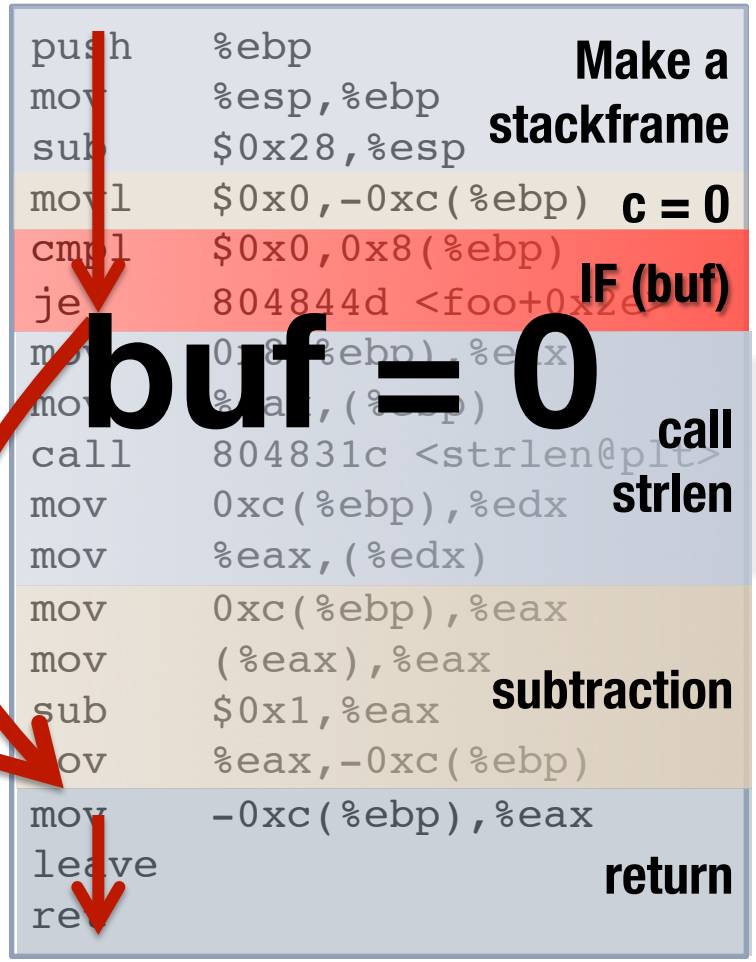
Evaluation

Hex-Rays

```
push   %ebp                          Make a
mov    %esp,%ebp                     stackframe
sub    $0x28,%esp
movl   $0x0,-0xc(%ebp)               c = 0
cmpl   $0x0,0x8(%ebp)                IF (buf)
je     804844d <foo+0x2e>
mov    0x8(%ebp),%eax
mov    %eax,(%esp)                   call
call   804831c <strlen@plt>          strlen
mov    0xc(%ebp),%edx
mov    %eax,(%edx)
mov    0xc(%ebp),%eax
mov    (%eax),%eax                   subtraction
sub    $0x1,%eax
mov    %eax,-0xc(%ebp)
mov    -0xc(%ebp),%eax
leave                                return
ret
```

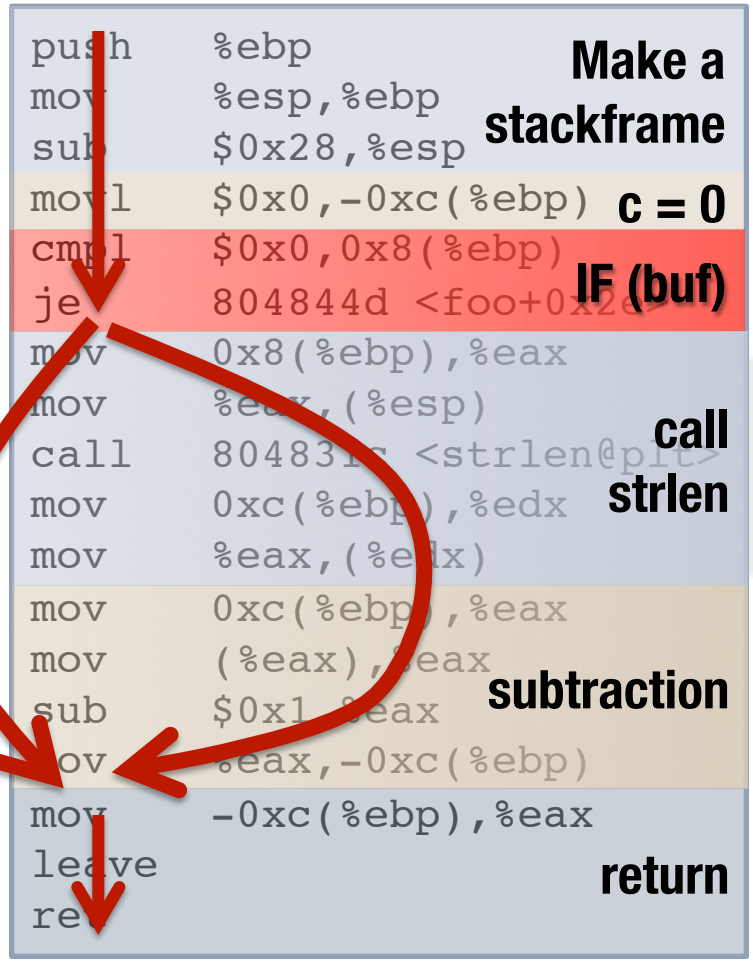
	Source types	Hex-Rays
buf	char *	char *
out	unsigned int *	int
c	unsigned int	int

REWARDS



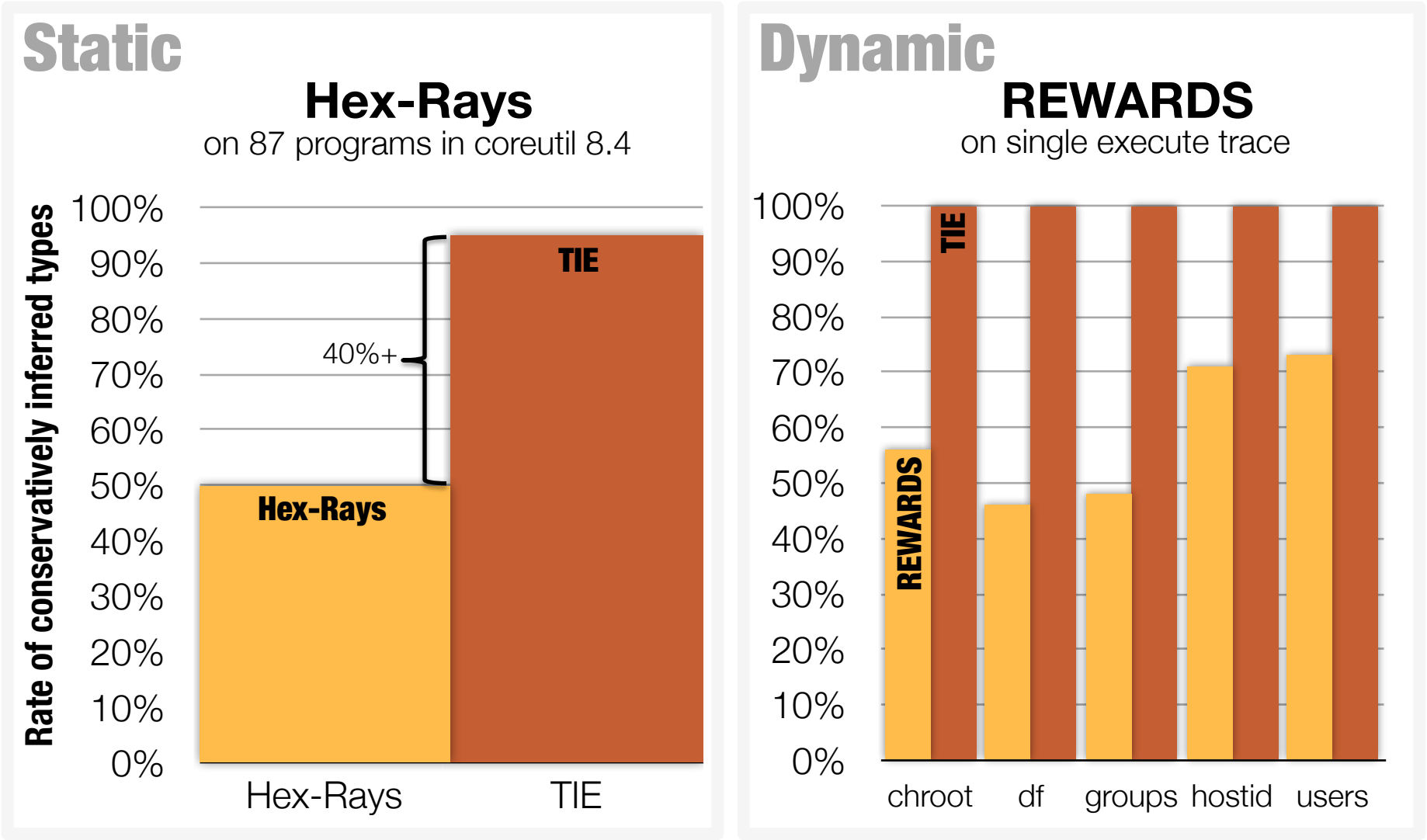
	Source types	REWARDS (buf=0)
buf	char *	unsigned int
out	unsigned int *	unsigned int
c	unsigned int	unsigned int

TIE



	Source types	TIE
buf	char *	char *
out	unsigned int *	unsigned int *
c	unsigned int	unsigned int

% of variables conservatively typed

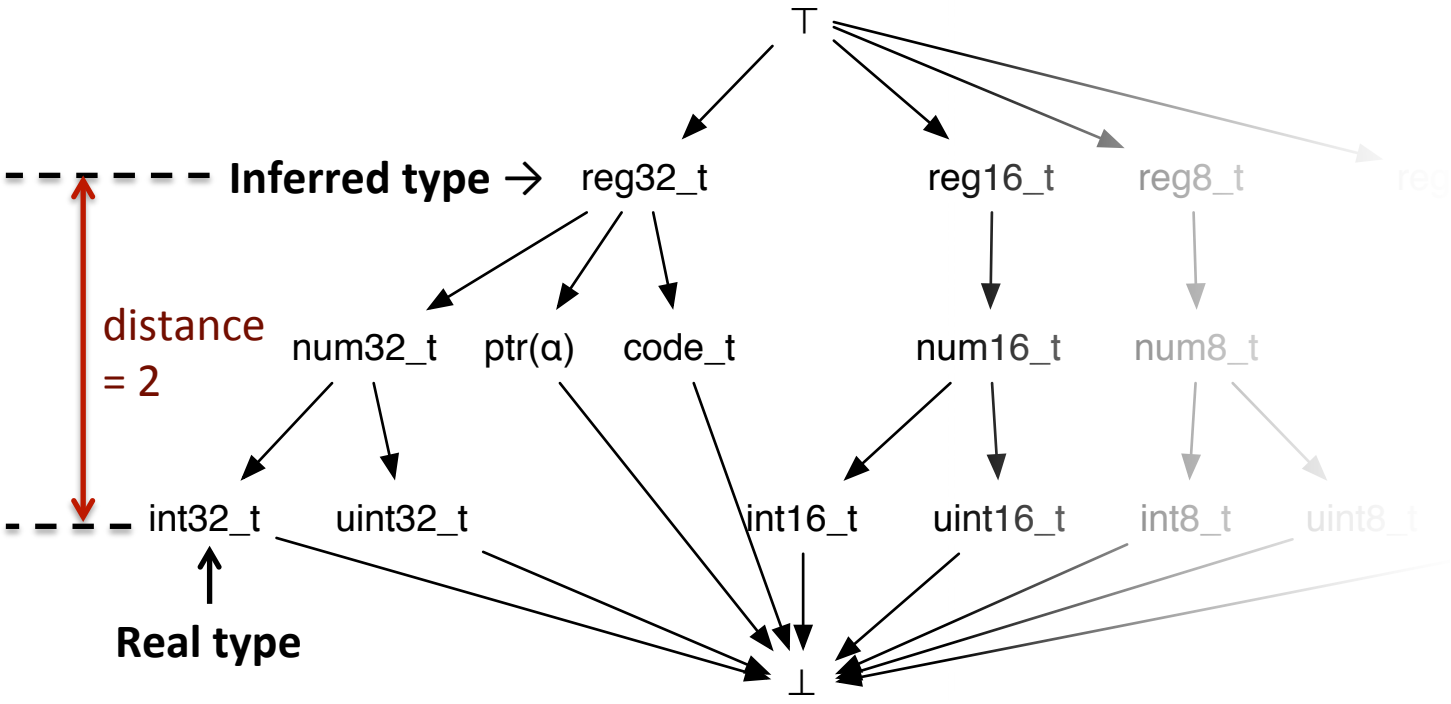


(Higher is better)

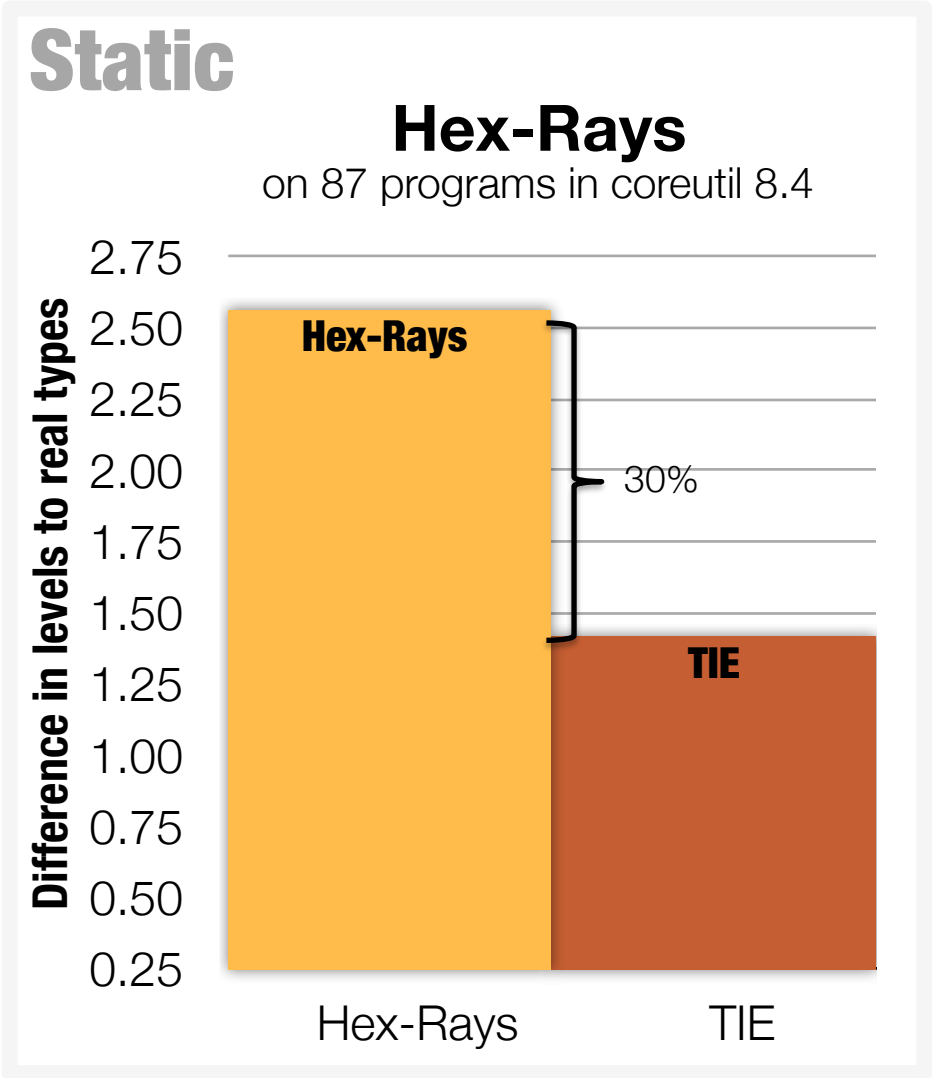
Accuracy

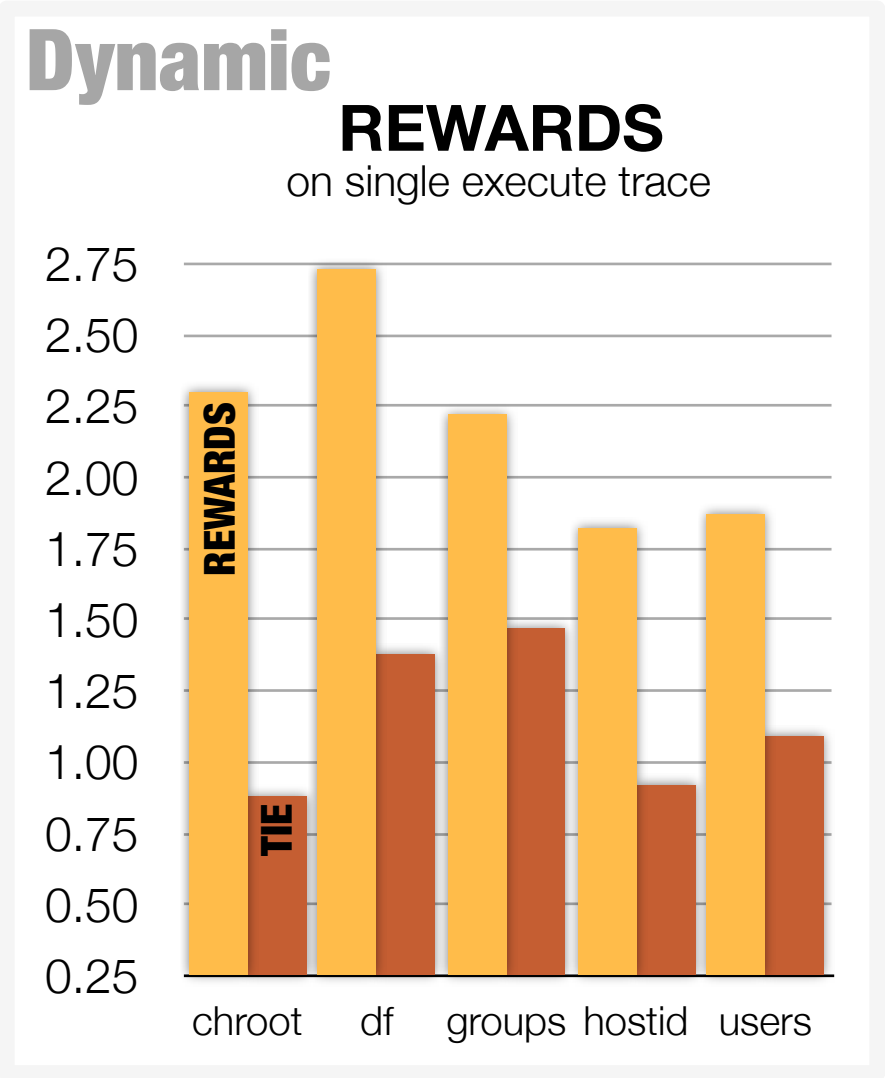
Distance to real types

Difference of level between a real type and an inferred type (selected form type interval)



Distance to real types

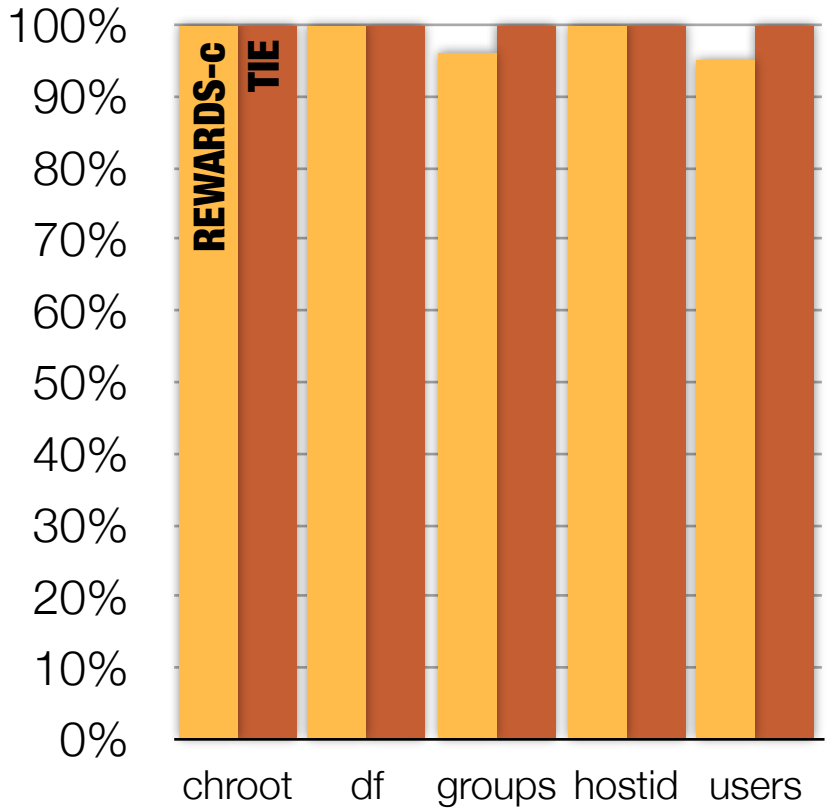




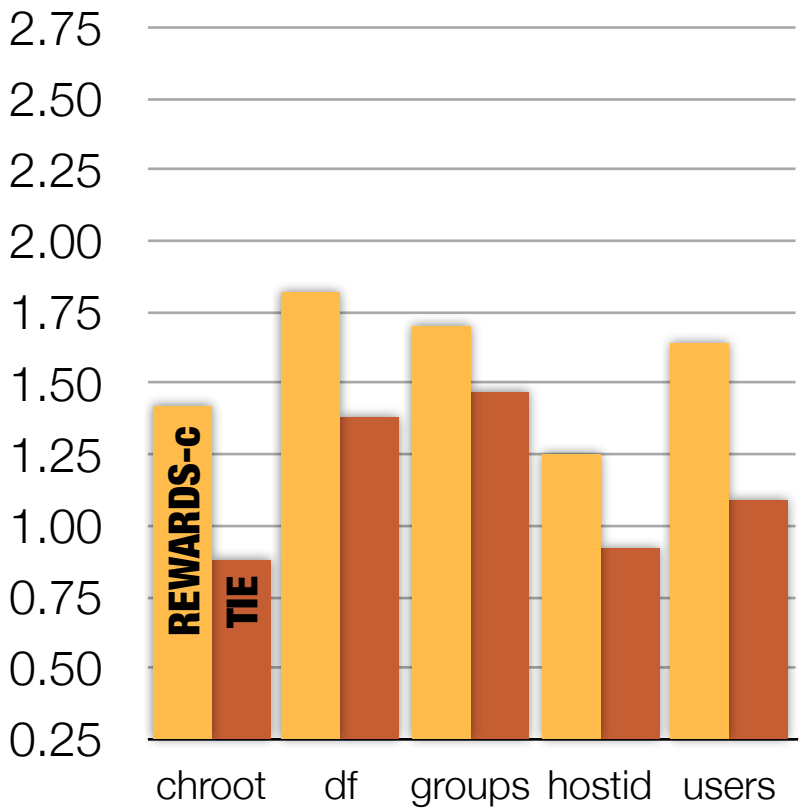
(Lower is better)

REWARDS-c

(Special thanks to Zhiqiang Lin, Dongyan Xu)



Conservativeness



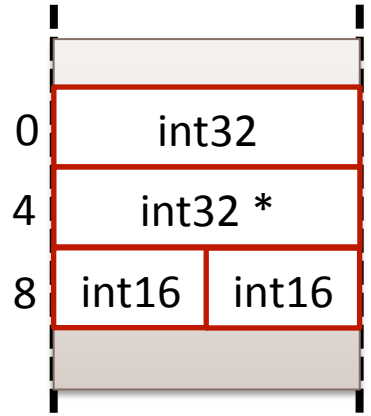
Distance

Structural types

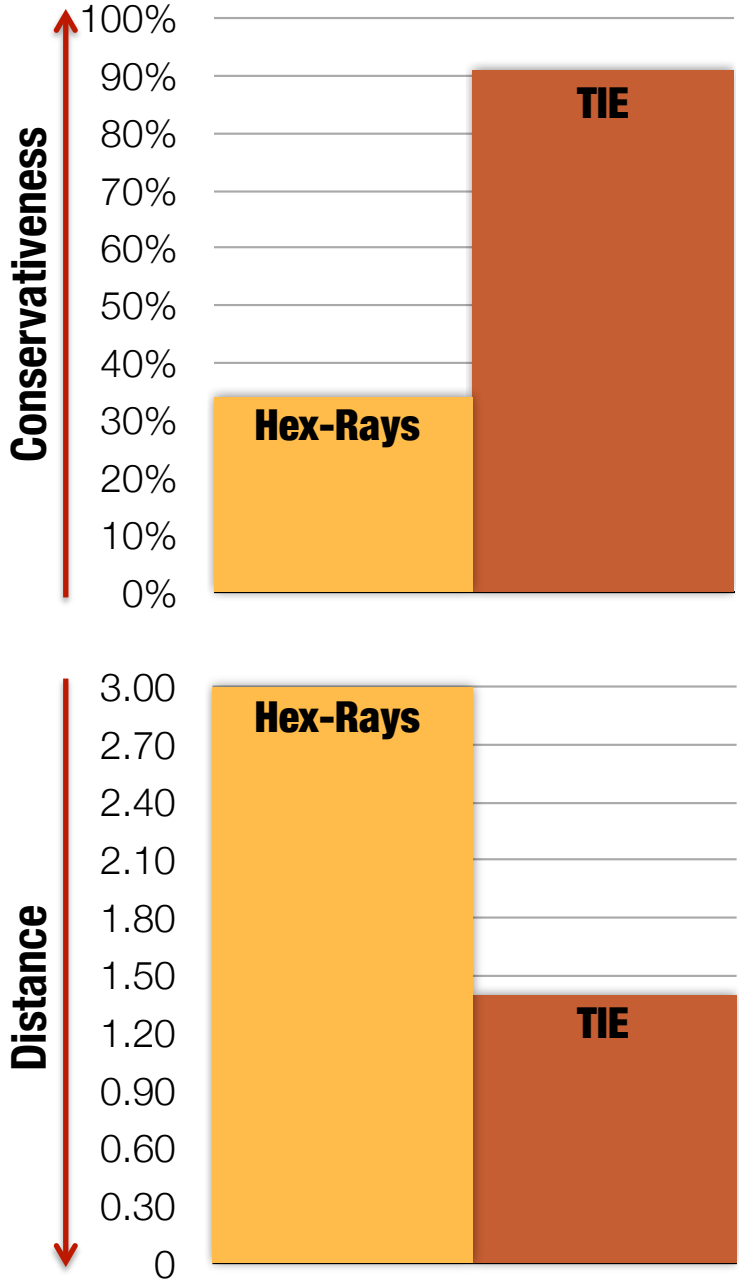
$$\{l_i : T_i\}$$

Record types

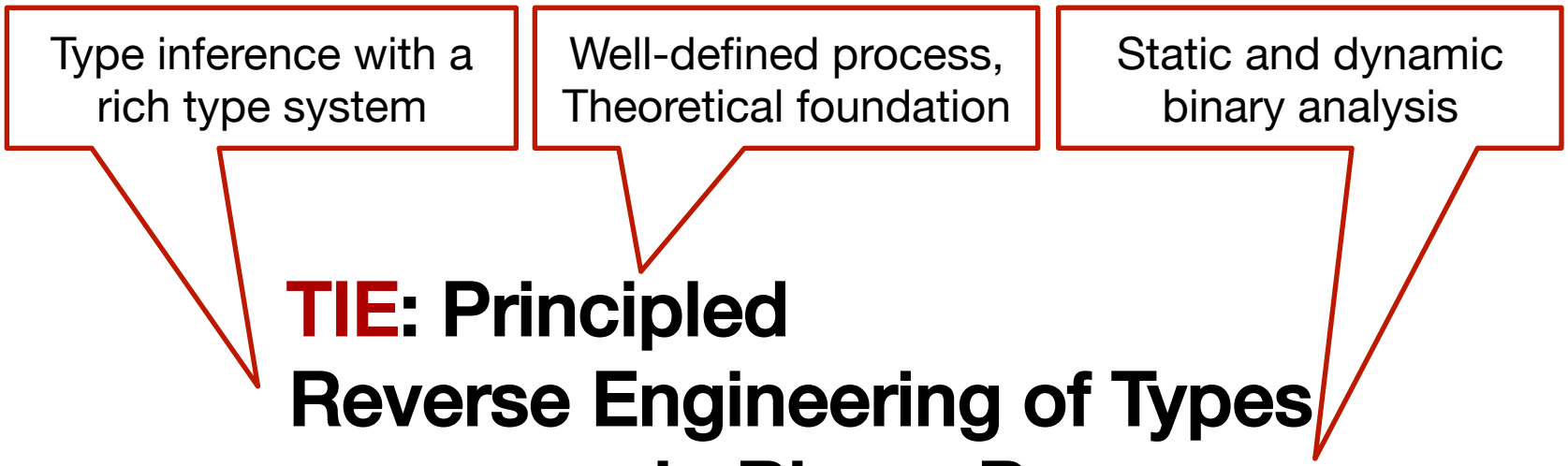
```
struct {
    int a;
    int* b;
    short c;
    short d;
}
```




```
{0 : int32, 4 : int32 *,
8 : int16, 10 : int16}
```



Conclusion


Type inference with a rich type system
Well-defined process, Theoretical foundation
Static and dynamic binary analysis
TIE: Principled Reverse Engineering of Types in Binary Programs

Thank you

Questions?

Email: jonglee@cmu.edu