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

  if (buf) {
    *out = strlen(buf);
    c = *out - 1;
  }
  return c;
}
8

Assign variables to memory slots

Remove types

Source code

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;
}

32-bit foo (32-bit [+8], 32-bit [+12]) {
    32-bit [-12];
    [-12] = 0;
    if ([+8]) {
        *[+12] = strlen([+8]);
        [-12] = *[+12] - 1;
    }
    return [-12];
}
unsigned int foo(unsigned char *[+8], unsigned int *[+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
push  %ebp
mov    %esp,%ebp
sub    $0x28,%esp
movl   $0x0,-0xc(%ebp)
cmpeq $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

Make a stackframe

\( c = 0 \)

if (buf)

Function call strlen

subtraction

return

No types, no variables
TIE

1. Variable recovery
2. Type inference

- Remove types
- Assign variables to memory slots

...
1. Variable Recovery

Binary Program

read
%eax = -0x4(%ebp)

write
-0x8(%ebp) = 0x1

read
%edx = -0x8(%ebp)

Analyze the value of "addresses"

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

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

```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**

```assembly
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
```
GUESS!

int  uint  pointer
Tell the type as it is

TIE

int
pointer
uint

Expressive type system
Type interval
TIE type system
allows us to express the type of variables as they are used

C-types
- int32
- uint32
- pointer

num32
reg32
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?”

T  -----------------------------

reg32  -------------------------

num32  ---------------

uint32  ---------------

⊥  -----------------------------
Type interval

refined
as we know more about the variable
Conservativeness of result

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

Conservative

Inferred type: \([\text{uint32}_t, \text{reg32}_t]\)

Real type: \(\text{uint32}_t\)
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

\[
\text{movl } (e_1), e_2
\]

<table>
<thead>
<tr>
<th>Usage clue</th>
<th>Type Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>value = load(index, s)</td>
<td>Memory load ‘index’ is a pointer of ‘value’ and the size of ‘value’ is ‘s’.</td>
</tr>
</tbody>
</table>

**Type variable for** \( e_1 \)

\[
e_2 = \text{load}(e_1, 32)
\]

\[
[e_1] = \text{ptr}([e_2]) \land [e_2] <: \text{reg32\_t}
\]

**Equality**

**Subtype relationship**

**conjunctive**
# Type Constraints from Usage Clues

<table>
<thead>
<tr>
<th>Usage clue</th>
<th>Type constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c = a +_{32} b$</td>
<td>32-bit addition&lt;br&gt;('&lt;a': number, 'b': number, 'c': number)&lt;br&gt; or&lt;br&gt;('&lt;a': pointer, 'b': number, 'c': pointer)&lt;br&gt; or&lt;br&gt;('&lt;a': number, 'b': pointer, 'c': pointer)</td>
</tr>
</tbody>
</table>
Inter-procedural Type Inference

- Type of passing arguments = Type of passed arguments

Function f1 ()
var2 = var3
...
call f2 (var1, var2)
...

Function f2 (arg1, arg2)
...
var4 = arg2

[var1] = [arg1]
[var2] = [arg2]
Type Constraints from Well-known Functions

\[ a = \text{strlen}(b) \]
\[ [a] = \text{uint32_t} \]
\[ [b] = \text{char *} \]

\[ \text{strcpy}(d, s) \]
\[ [d] = \text{char *} \]
\[ [s] = \text{char *} \]
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, $A \land B$
  – Solve all

• Disjunctive, $A \lor B$
  – Merge compatible terms
Equality constraints (unification)

\[ [a] = \text{ptr}([\text{uint32}]) \]

\[ \text{uint32} = \text{uint32} \]

\[ \text{uint32} = \text{uint32} \]
Subtype relationship constraints
(closure alg.)

\[ \text{int32}_t <: [a] <: [b] <: \text{num32}_t \]
Subtype relationship constraints (closure alg.)

\[
\text{int32}_t \lessdot [a] \lessdot [b] \lessdot \text{num32}_t
\]

\[
[a] \lessdot [b] \lessdot \text{num32}_t
\]
Subtype relationship constraints (closure alg.)

\[ \text{int32}_t \quad <: \quad [a] \quad <: \quad [b] \quad <: \quad \text{num32}_t \]

\[ [a] \quad <: \quad [b] \quad <: \quad \text{num32}_t \]

\[ \text{int32}_t \quad <: \quad [a] \quad <: \quad [b] \]
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

<table>
<thead>
<tr>
<th>Hex-Rays</th>
<th>TIE</th>
</tr>
</thead>
</table>
| ?        | Principled reverse engineering  
- Well defined process  
- Type theory |

<table>
<thead>
<tr>
<th>REWARDS</th>
<th>TIE</th>
</tr>
</thead>
</table>
| - 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

Make a stackframe

IF (buf)

c = 0

call strlen

subtraction

return

<table>
<thead>
<tr>
<th>Source types</th>
<th>Hex-Rays</th>
</tr>
</thead>
<tbody>
<tr>
<td>buf</td>
<td>char *</td>
</tr>
<tr>
<td>out</td>
<td>unsigned int *</td>
</tr>
<tr>
<td>c</td>
<td>unsigned int</td>
</tr>
</tbody>
</table>
REWARDS

Make a stackframe

BUF = 0

IF (buf)

Call strlen

Subtraction

C = 0

buf = 0

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 %eax,0xc(%ebp)
leave
ret

<table>
<thead>
<tr>
<th>Source types</th>
<th>REWARDS (buf=0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>buf</td>
<td>char *</td>
</tr>
<tr>
<td>out</td>
<td>unsigned int *</td>
</tr>
<tr>
<td>c</td>
<td>unsigned int</td>
</tr>
</tbody>
</table>
TIE

push %ebp
mov %esp,%ebp
sub $0x28,%esp
movl $0x0,-0xc(%ebp)
cmpl $0x0,0x8(%ebp)
je 804844d <foo+0x2e>
mov 0x8(%ebp),%eax
movl %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

Make a stackframe

IF (buf)
c = 0

Call strlen

Subtraction

TIE

<table>
<thead>
<tr>
<th>Source types</th>
<th>TIE</th>
</tr>
</thead>
<tbody>
<tr>
<td>buf</td>
<td>char *</td>
</tr>
<tr>
<td>out</td>
<td>unsigned int *</td>
</tr>
<tr>
<td>c</td>
<td>unsigned int</td>
</tr>
</tbody>
</table>
% of variables conservatively typed

Static

**Hex-Rays**
on 87 programs in coreutil 8.4

- **Hex-Rays**
- **TIE**

Dynamic

**REWARDS**
on single execute trace

- chroot
- df
- groups
- hostid
- users

(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

Static

Hex-Rays
on 87 programs in coreutil 8.4

0.25 0.50 0.75 1.00 1.25 1.50 1.75 2.00 2.25 2.50 2.75

Hex-Rays
TIE

(Difference in levels to real types)

Dynamic

REWARDS
on single execute trace

0.25 0.50 0.75 1.00 1.25 1.50 1.75 2.00 2.25 2.50 2.75

chroot df groups hostid users

(TIE)

(Lower is better)
REWARDS-c

(Special thanks to Zhiqiang Lin, Dongyan Xu)
Structural types

\[ \{ l_i : T_i \} \]

Record types

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

\{ 0 : int32, 4 : int32 *,
    8 : int16, 10 : int16 \}
Conclusion

**TIE: Principled Reverse Engineering of Types in Binary Programs**

- Type inference with a rich type system
- Well-defined process, Theoretical foundation
- Static and dynamic binary analysis
Thank you

Questions?
Email: jonglee@cmu.edu