CS 6V81-05
Data Structure Reverse Engineering

Zhiqiang Lin

Department of Computer Science
The University of Texas at Dallas

September 2nd, 2011
What is Data Structure Reverse Engineering

Data structure reverse engineering aims to infer both the Syntax (layout, size, offset) of variables.
Data structure reverse engineering aims to infer both the
1. Syntax (layout, size, offset) of variables
2. Semantics (meaningful use, e.g., ip_addr, pid_t) of Types
Data structure reverse engineering aims to infer both the

1. **Syntax** (layout, size, offset) of **variables**

2. **Semantics** (meaningful use, e.g., `ip_addr`, `pid_t`) of **Types**

when only given a binary code.
Who cares

1. Memory Forensics
Who cares

1. Memory Forensics
2. Vulnerability Discovery
Who cares

1. Memory Forensics
2. Vulnerability Discovery
3. Protocol Reverse Engineering
Who cares

1. Memory Forensics
2. Vulnerability Discovery
3. Protocol Reverse Engineering
4. Program signature
Who cares

1. Memory Forensics
2. Vulnerability Discovery
3. Protocol Reverse Engineering
4. Program signature
5. Virtual machine introspection
Who cares

1. Memory Forensics
2. Vulnerability Discovery
3. Protocol Reverse Engineering
4. Program signature
5. Virtual machine introspection
6. ...

...
Understanding the Compilation

Credit: figure is from Jonghyup Lee's NDSS'11 presentation
Understanding the Compilation

```c
unsigned int foo(
    char *buf,
    unsigned int *out)
{
    unsigned int c;
    c = 0;
    if (buf) {
        *out = strlen(buf);
        c = *out - 1;
    }
    return c;
}
```

Credit: figure is from Jonghyup Lee’s NDSS’11 presentation
Understanding the Compilation

Credit: figure is from Jonghyup Lee's NDSS'11 presentation
Understanding the Compilation

```
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
```
Again, from binary code, infer

1. **Syntax** (layout, size, offset) of **variables**
2. **Semantics** (meaningful use, e.g., `ip_addr`, `pid_t`) of **Types**
Overview

REWARDS: Automatic Reverse Engineering of Data Structures from Binary Execution. NDSS 2010

TIE: Principled Reverse Engineering of Types in Binary Programs. NDSS 2011

HOWARD: A Dynamic Excavator for Reverse Engineering Data Structures. NDSS 2011

Three papers
Three papers

- **REWARDS**: Automatic Reverse Engineering of Data Structures from Binary Execution. *NDSS 2010*

- **TIE**: Principled Reverse Engineering of Types in Binary Programs. *NDSS 2011*
Three papers

- **REWARDS**: Automatic Reverse Engineering of Data Structures from Binary Execution. *NDSS 2010*
- **TIE**: Principled Reverse Engineering of Types in Binary Programs. *NDSS 2011*
- **Howard**: A Dynamic Excavator for Reverse Engineering Data Structures. *NDSS 2011*
Overview

REWARDS

TIE

HOWARD

Summary

Observation

(a) Source code of function `foo` and the `_start` assembly code

```c
1 struct {
2     unsigned int pid;
3     char data[16];
4 } test;
5
6 void foo()
7 {
8     char *p = "hello world";
9     test.pid = my_getpid();
10     strcpy(test.data, p);
11 }
```

(b) Disassembly code of the example binary

```assembly
rodata_0x08048118{
+00:    char[12],
+04:    char[12],
+16:    unused[4]
}
fun_0x080480b4{
-28:    unused[20],
-08:    char *,
-04:    stack_frame_t,
+00:    ret_addr_t
}
fun_0x08048110{
+00:    ret_addr_t
}
```

(c) Section map of the example binary

```
Nr Name Type Addr Off Size
1 .text PROGBITS 080480a0 0000a0 000078
2 .rodata PROGBITS 08048118 000118 00000c
3 .bss NOBITS 08049124 000124 000014
```

(d) Output of REWARDS

```
1 80480a0:    e8 0f 00 00 00 00    call    0x80480b4
2 80480a5:    b8 01 00 00 00 00    mov     $0x1,%eax
3 80480aa:    bb 00 00 00 00 00    mov     $0x0,%ebx
4 80480af:    cd 80                 int     $0x80
5 ...
6 80480b4:    55                   push    %ebp
7 80480b5:    89 e5                mov     %esp,%ebp
8 80480b7:    83 ec 18              sub     $0x18,%esp
9 80480ba:    c7 45 fc 18 81 04 08    movl    $0x8048118,0xfffffffc(%ebp)
10 80480c1:    e8 4a 00 00 00    call    0x8048110
11 80480c6:    a3 24 91 04 08    mov     %eax,0x8049124
12 80480c8:    8b 45 fc              mov     0xfffffffc(%ebp),%eax
13 80480ce:    89 44 24 18 91 04 08    mov     0xfffffffc(%ebp),%eax
14 80480d2:    c7 04 24 28 91 04 08    movl    $0x8049128,(%esp)
15 80480d9:    e8 02 00 00 00    call    0x80480e0
16 80480de:    c9                     leave
17 80480e0:    55                   push    %ebp
18 80480e1:    89 e5                mov     %esp,%ebp
19 80480e4:    8b 5d 08              mov     0x8(%ebp),%eax
20 80480e7:    8b 55 0c              mov     0xc(%ebp),%edx
21 80480ea:    89 d8                 mov     %eax,%ebx
22 80480ec:    29 d0                sub     %edx,%eax
23 80480ee:    8d 48 ff              lea     0xffffffff(%eax),%ecx
24 80480f0:    0f b6 b6 02           movzbl  (%edx),%eax
25 80480f4:    83 c2 01              add     $0x1,%edx
26 80480f7:    84 c0                test    %al, %al
27 80480f9:    84 04 0a              mov     %al, (%edx, %ecx,1)
28 80480fc:    75 f3                jne     0x80480f1
29 80480fe:    89 d8                mov     %ebx,%eax
30 80480f:    5b                   pop     %ebx
31 8048102:    e8 02 00 00 00    call    0x80480e0
32 8048107:    c3                     ret
33 ...
36 8048110:    b8 14 00 00 00 00    mov     $0x14,%eax
37 8048115:    cd 80                 int     $0x80
38 8048117:    c3                     ret
```
Key Ideas

1. **Identifying** type resolution points in binary code
Key Ideas

1. **Identifying** type resolution points in binary code
2. **Tracking** data flow to resolve other all involved memory/register types
Type Resolution points

<getpid>

8048110:  mov    $0x14,%eax
8048115:  int    $0x80
8048117:  ret

- **System calls**: syscall num (eax)
- **Syscall Enter**: Type parameter passing registers (i.e., ebx, ecx, edx, esi, edi, and ebp) if they involved
Type Resolution points

<getpid>

8048110:  mov    $0x14,%eax
8048115:  int    $0x80
8048117:  ret

- **System calls**: syscall num (eax)
  - **Syscall Enter**: Type parameter passing registers (i.e., ebx, ecx, edx, esi, edi, and ebp) if they involved
  - **Syscall Exit**: Type return value (eax)
Type Resolution points

80480ce: mov %eax, 0x4(%esp) <arg2>
80480d2: movl $0x8049128, (%esp) <arg1>
80480d9: call 0x80480e0 <strcpy>

- Standard Library Call (API)
  - Type corresponding argument and return value
Type Resolution points

80480ce:  mov %eax, 0x4(%esp)  <arg2>
80480d2:  movl $0x8049128, (%esp)  <arg1>
80480d9:  call 0x80480e0  <strcpy>

Standard Library Call (API)
- Type corresponding argument and return value
- More wealth than System call
  - 2016 APIs in Libc.so.6 vs. 289 sys call (2.6.15)
Type Resolution points

- Other type revealing instructions in binary code:
Other type revealing instructions in binary code:

- String related (e.g., MOVs/B/D/W, LOADs/STOS/B/D/W)
- Floating-point related (e.g., FADD, FABS, FST)
- Pointer-related (e.g., MOV (%edx), %ebx)
Data Flow Tracking

Standard technique:
Data Flow Tracking

**Standard technique:**
- Using shadow memory to keep the variable attributes and track the propagation
Standard technique:

- Using shadow memory to keep the variable attributes and track the propagation

New challenges
Data Flow Tracking

Standard technique:
- Using shadow memory to keep the variable attributes and track the propagation

New challenges
- Two-way type resolution
- Dynamic life time of stack and heap variables
- Memory locations will be reused
- Multiple instances of the same type
Implementations

Dynamic Binary Instrumentation: A pin-tool on top of Pin-2.6
http://www.pintool.org/
Implementations

Dynamic Binary Instrumentation: A pin-tool on top of Pin-2.6
http://www.pintool.org/

Data structures of interest
- File information
- Network communication
- Process information
- Input-influenced (for vulnerability discovery)
Limitation

- Loss of data structure hierarchy

There is no corresponding type resolution point with the same hierarchical type. Heuristics exist (e.g., AutoFormat [Lin et al, NDSS2008]).

Path-sensitive memory reuse. The compiler might assign different local variables declared in different program paths to the same memory address. Path-sensitive analysis.

Type cast

```
int a=(float)b;
```
Limitation

- **Loss of data structure hierarchy**
  - There is no corresponding type resolution point with the same hierarchical type
  - Heuristics exist (e.g., AutoFormat [Lin et al, NDSS2008])
Limitation

- **Loss of data structure hierarchy**
  - There is no corresponding type resolution point with the same hierarchical type
  - Heuristics exist (e.g., AutoFormat [Lin et al, NDSS2008])

- **Path-sensitive memory reuse**
  - Compiler might assign different local variables declared in different program paths to the same memory address
  - Path-sensitive analysis
Limitation

- **Loss of data structure hierarchy**
  - There is no corresponding type resolution point with the same hierarchical type
  - Heuristics exist (e.g., AutoFormat [Lin et al, NDSS2008])

- **Path-sensitive memory reuse**
  - Compiler might assign different local variables declared in different program paths to the same memory address
  - Path-sensitive analysis

- **Type cast**
Limitation

- **Loss of data structure hierarchy**
  - There is no corresponding type resolution point with the same hierarchical type
  - Heuristics exist (e.g., AutoFormat [Lin et al, NDSS2008])

- **Path-sensitive memory reuse**
  - Compiler might assign different local variables declared in different program paths to the same memory address
  - Path-sensitive analysis

- **Type cast**
  - `int a=(float)b;`
Limitation

- Dynamic analysis
Limitation

- Dynamic analysis
  - Full coverage of data structures?
  - TIE [NDSS 2011]
Limitation

- **Dynamic analysis**
  - Full coverage of data structures?
  - TIE [NDSS 2011]
- **Only user-level data structure**
  - OS kernel?
Limitation

- **Dynamic analysis**
  - Full coverage of data structures?
  - TIE [NDSS 2011]
- **Only user-level data structure**
  - OS kernel?
- **Obfuscated binary can cheat REWARDS**
Limitation

- **Dynamic analysis**
  - Full coverage of data structures?
  - TIE [NDSS 2011]
- **Only user-level data structure**
  - OS kernel?
- **Obfuscated binary can cheat REWARDS**
  - No explicit library call
  - Memory cast
Conclusion

- REWARDS
Conclusion

- **REWARDS**
  - Binary only
  - Dynamic analysis
  - Data flow tracking
Conclusion

- **REWARDS**
  - Binary only
  - Dynamic analysis
  - Data flow tracking

- **Key insight**
Conclusion

- **REWARDS**
  - Binary only
  - Dynamic analysis
  - Data flow tracking

- **Key insight**
  - Using system call/API/Type revealing instruction as type resolution point
  - Two-way type resolution
  - Unique benefits to memory forensics and vulnerability discovery
Outline

1. Overview
2. REWARDS
3. TIE
4. HOWARD
5. Summary
Goal I: Variable Discovery

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

Credit: figure is from Jonghyup Lee’s NDSS’11 presentation
Goal II: Type Inference

Credit: figure is from Jonghyup Lee’s NDSS’11 presentation
Problems: multiple possible types

```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;
}
```

Credit: figure is from Jonghyup Lee's NDSS'11 presentation
TIE’s Contribution

```
reg32_t
num32_t
ptr(α)
⊥
int32_t
uint32_t
reg16_t
num16_t
int16_t
uint16_t
reg8_t
num8_t
int8_t
uint8_t
reg1_t
```
Key Idea

- Collecting Type Constraint from
Key Idea

- **Collecting Type Constraint from**
  - Standard Library Call
  - System Call
  - Type Revealing Instructions
Key Idea

- Collecting Type Constraint from
  - Standard Library Call
  - System Call
  - Type Revealing Instructions
- Solving Type Constraint based on the Type Lattice
Key Idea

- **Collecting Type Constraint from**
  - Standard Library Call
  - System Call
  - Type Revealing Instructions

- **Solving Type Constraint based on the Type Lattice**
  - Equality \( A = B \), essentially **Type Unification**
  - Subtype relation \( A <: B \), using unification closure algorithm
  - Conjunctive ( \( A \land B \) )
  - Disjunctive ( \( A \lor B \) )
## Limitations

- Works on regular programs compiled from C code
Limitations

- Works on regular programs compiled from C code
- Not very informative for irregular programs
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
Principled reverse engineering
Conclusion

- Principled reverse engineering
- Well defined process + Type theory
Principled reverse engineering
Well defined process + Type theory
Type inference with more expressive type system (unification closure algorithm)
Key Idea

Analyse dynamically

Credit: Picture from Asia Slowinska's NDSS’11 presentation
Observe how memory is used at runtime to detect data structures. For instance, if A is a pointer, then if
Observe how memory is used at runtime to detect data structures. For instance, if A is a pointer, then if

- A is a function frame pointer, then *(A + 8) is perhaps a function argument
Observe how memory is used at runtime to detect data structures. For instance, if A is a pointer, then if

- A is a function frame pointer, then *(A + 8) is perhaps a function argument
- A is an address of a structure, then *(A + 8) is perhaps a field in this structure
Intuition in HOWARD

Observe how memory is used at runtime to detect data structures. For instance, if A is a pointer, then if

- A is a function frame pointer, then *(A + 8) is perhaps a function argument
- A is an address of a structure, then *(A + 8) is perhaps a field in this structure
- A is an address of an array, then *(A + 8) is perhaps an element of this array
Contributions

Increased coverage

- Leverage S2E/Klee to increase the path coverage
Contributions

**Increased coverage**
- Leverage S2E/Klee to increase the path coverage

**Can discover array and more internal data structure**
Contributions

Increased coverage

- Leverage S2E/Klee to increase the path coverage

Can discover array and more internal data structure

- Look for chains of accesses in a loop (e.g., `elem = next++;`)

Look for sets of accesses with the same base in a linear space (e.g., `elem = array[i];`)

Look for chains of accesses in a loop

Boundary elements accessed outside the loop

Nested loops

Multiple loops in sequence
Contributions

**Increased coverage**
- Leverage S2E/Klee to increase the path coverage

**Can discover array and more internal data structure**
- Look for chains of accesses in a loop (e.g., `elem = next++;`)
- Look for sets of accesses with the same base in a linear space (e.g., `elem = array[i];`)
**Contributions**

**Increased coverage**
- Leverage S2E/Klee to increase the path coverage

**Can discover array and more internal data structure**
- Look for chains of accesses in a loop (e.g., `elem = next++;`)
- Look for sets of accesses with the same base in a linear space (e.g., `elem = array[i];`)
- Look for chains of accesses in a loop
Contributions

Increased coverage

- Leverage S2E/Klee to increase the path coverage

Can discover array and more internal data structure

- Look for chains of accesses in a loop (e.g., `elem = next++;`)
- Look for sets of accesses with the same base in a linear space (e.g., `elem = array[i];`)
- Look for chains of accesses in a loop
  - Boundary elements accessed outside the loop
  - Nested loops
  - Multiple loops in sequence
Outline

1. Overview
2. REWARDS
3. TIE
4. HOWARD
5. Summary
All three techniques: from data use to infer syntax (offset, size, layout) of the variables
Summary

All three techniques: from data use to infer syntax (offset, size, layout) of the variables

REWARDS

REWARDS makes a first step in reconstructing the syntax and semantics of data structure, and demonstrated the security benefits of such reversing.
All three techniques: from data use to infer syntax (offset, size, layout) of the variables

**REWARDS**

REWARDS makes a first step in reconstructing the syntax and semantics of data structure, and demonstrated the security benefits of such reversing.

**TIE**

TIE is able to statically analyze the binary code, and figure out the types for each individual variables
Summary

All three techniques: from data use to infer syntax (offset, size, layout) of the variables

- **REWARDS**
  REWARDS makes a first step in reconstructing the syntax and semantics of data structure, and demonstrated the security benefits of such reversing.

- **TIE**
  TIE is able to statically analyze the binary code, and figure out the types for each individual variables

- **HOWARD**
  HOWARD integrates symbolic execution to have a larger coverage than REWARDS, and also it focuses more on the internal data structures defined in the program.