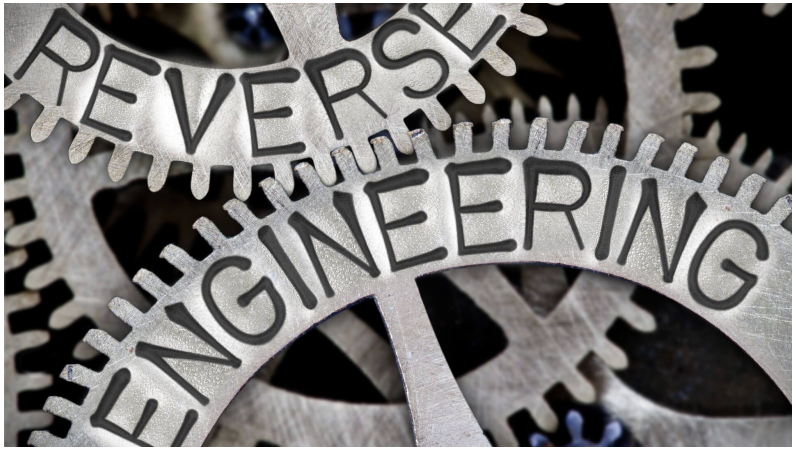
Binary analysis: Reverse Engineering

Yue Duan

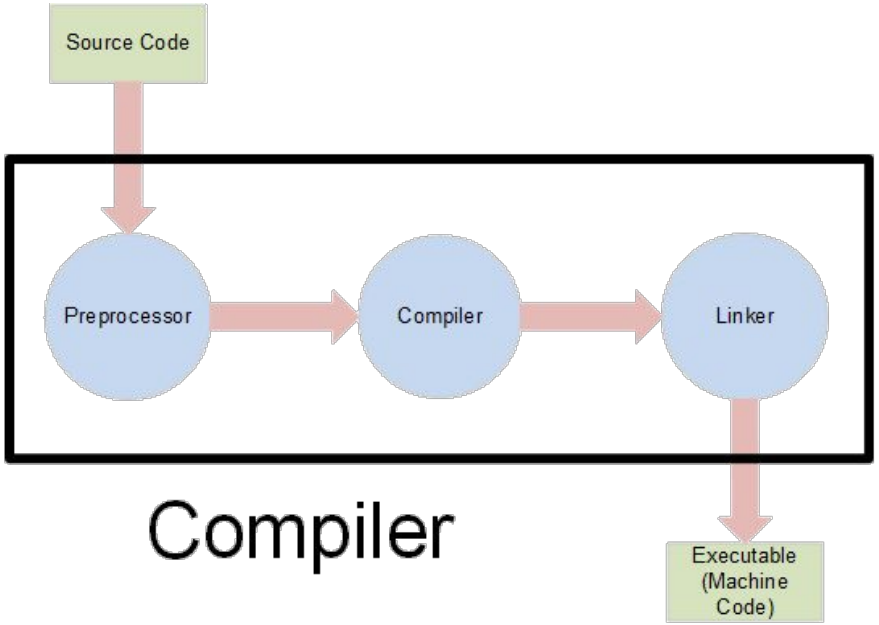
Outline

- Reverse engineering basics
- Research papers:
  - Automatic Reverse Engineering of Program Data Structures from Binary Execution
  - Howard: A Dynamic Excavator for Reverse Engineering Data Structures



Reverse Engineering Basics

- Compilation process:
  - source code to machine code



Reverse Engineering Basics

- Differences between source code and machine code
  - Source code
    - Human readable
    - Contain high-level semantics (e.g., data structure info, function boundary info)
  - Machine code (stripped binary)
    - Impossible to read
    - High-level semantic information is gone

Reverse Engineering Basics

- Reverse engineering
  - To recover missing information
    - code info
      - disassemble => assembly code
      - decompile = > source code
    - function info
      - function recovery => function boundary
    - data info
      - data structure recovery => data structures

Reverse Engineering Basics

- Ultimate goal

```
push   %ebp
mov    %esp,%ebp
sub    $0xa8,%esp
mov    0x8(%ebp),%eax
lea    -0x98(%ebp),%ecx
mov    %eax,%edx
mov    $0x8c,%eax
mov    %eax,0x8(%esp)
mov    %edx,0x4(%esp)
mov    %ecx,(%esp)
call   0x29
mov    0x8(%ebp),%eax
leave
ret
nop
nop
```

```
struct employee {
   char name [128];
   int year;
   int month;
   int day;
};
struct employee*
foo (struct employee* src)
{
   struct employee dst;
   // init dst

}
```

Reverse Engineering Basics

- Example 6de: 48 83 ec 30
  - 6de: code address
  - 48: long mode
  - 83:

83 /5 ib	SUB r/m16,imm8
83 /5 ib	SUB r/m32,imm8

  - ec:
    - could mean various things depending on the opcode
    - here it means %rsp
  - 30: 0x30
- Problems?
  - 55 48 89 e5 48 83 ec 30 89 7d dc

```
00000000000006da <main>:
 6da:   55                      push   %rbp
 6db:   48 89 e5                mov    %rsp,%rbp
 6de:   48 83 ec 30             sub    $0x30,%rsp
 6e2:   89 7d dc                mov    %edi,-0x24(%rbp)
 6e5:   48 89 75 d0             mov    %rsi,-0x30(%rbp)
 6e9:   bf f8 00 00 00          mov    $0xf8,%edi
 6ee:   e8 bd fe ff ff          callq  5b0 <malloc@plt>
 6f3:   48 89 45 f8             mov    %rax,-0x8(%rbp)
 6f7:   bf f8 00 00 00          mov    $0xf8,%edi
 6fc:   e8 af fe ff ff          callq  5b0 <malloc@plt>
 701:   48 89 45 f0             mov    %rax,-0x10(%rbp)
 705:   48 8b 45 f8             mov    -0x8(%rbp),%rax
 709:   48 89 c7                mov    %rax,%rdi
 70c:   e8 7f fe ff ff          callq  590 <free@plt>
 711:   48 8b 45 f0             mov    -0x10(%rbp),%rax
 715:   48 89 c7                mov    %rax,%rdi
 718:   e8 73 fe ff ff          callq  590 <free@plt>
 71d:   bf 00 02 00 00          mov    $0x200,%edi
 722:   e8 89 fe ff ff          callq  5b0 <malloc@plt>
 727:   48 89 45 e8             mov    %rax,-0x18(%rbp)
 72b:   48 8b 45 d0             mov    -0x30(%rbp),%rax
 72f:   48 83 c0 08             add    $0x8,%rax
 733:   48 8b 10                mov    (%rax),%rdx
 736:   48 8b 45 e8             mov    -0x18(%rbp),%rax
 73a:   48 89 d6                mov    %rdx,%rsi
 73d:   48 89 c7                mov    %rax,%rdi
 740:   e8 5b fe ff ff          callq  5a0 <strcpy@plt>
 745:   48 8b 45 f0             mov    -0x10(%rbp),%rax
 749:   48 89 c7                mov    %rax,%rdi
 74c:   e8 3f fe ff ff          callq  590 <free@plt>
 751:   48 8b 45 e8             mov    -0x18(%rbp),%rax
 755:   48 89 c7                mov    %rax,%rdi
 758:   e8 33 fe ff ff          callq  590 <free@plt>
 75d:   b8 00 00 00 00          mov    $0x0,%eax
 762:   c9                      leaveq
 763:   c3                      retq
 764:   66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
 76b:   00 00 00
 76e:   66 90                   xchg   %ax,%ax
```

Reverse Engineering Basics

```
int main(int argc, char **argv) {
  char *buf1R1;
  char *buf2R1;
  char *buf1R2;
  char buf3[100];

  memset(buf3, 0, 100);
  buf1R1 = (char *) malloc(BUFSIZE2);
  buf2R1 = (char *) malloc(BUFSIZE2);

  free(buf1R1);
  free(buf2R1);

  buf1R2 = (char *) malloc(BUFSIZE1);
  strcpy(buf1R2, argv[1]);

  free(buf2R1);
  free(buf1R2);
}
```

data structure information

```
00000000000006da <main>:
 6da:   55                      push   %rbp
 6db:   48 89 e5                mov    %rsp,%rbp
 6de:   48 83 ec 30             sub    $0x30,%rsp
 6e2:   89 7d dc                mov    %edi,-0x24(%rbp)
 6e5:   48 89 75 d0             mov    %rsi,-0x30(%rbp)
 6e9:   bf f8 00 00 00          mov    f8,%edi
 6ee:   e8 bd fe ff ff          callq  malloc@plt>
 6f3:   48 89 45 f8             mov    x8(%rbp)
 6f7:   bf f8 00 00 00          %edi
 6fc:   e8 af fe ff ff          <malloc@plt>
        48 89 45 f0             x,-0x10(%rbp)
                                0x8(%rbp),%rax
 709:   48 89 c7                ax,%rdi
 70c:   e8 7f fe ff ff          <free@plt>
 711:   48 8b 45 f0             (%rbp),%rax
 715:   48 89 c7                di
 718:   e8 73 fe ff ff          ee@plt>
 71d:   bf 00 02 00 00          00,%edi
 722:   e8 89 fe ff ff          callq  0 <malloc@plt>
 727:   48 89 45 e8             mov    %rax,-0x18(%rbp)
 72b:   48 8b 45 d0             mov    -0x30(%rbp),%rax
 72f:   48 83 c0 08             add    $0x8,%rax
 733:   48 8b 10                mov    (%rax),%rdx
 736:   48 8b 45 e8             mov    -0x18(%rbp),%rax
 73a:   48 89 d6                mov    %rdx,%rsi
 73d:   48 89 c7                mov    %rax,%rdi
 740:   e8 5b fe ff ff          callq  5a0 <strcpy@plt>
 745:   48 8b 45 f0             mov    -0x10(%rbp),%rax
 749:   48 89 c7                mov    %rax,%rdi
 74c:   e8 3f fe ff ff          callq  590 <free@plt>
 751:   48 8b 45 e8             mov    -0x18(%rbp),%rax
 755:   48 89 c7                mov    %rax,%rdi
 758:   e8 33 fe ff ff          callq  590 <free@plt>
 75d:   b8 00 00 00 00          mov    $0x0,%eax
 762:   c9                      leaveq
 763:   c3                      retq
 764:   66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
 76b:   00 00 00
 76e:   66 90                   xchg   %ax,%ax
```

Automatic Reverse Engineering of Program Data Structures from Binary Execution

Zhiqiang Lin, Xiangyu Zhang, Dongyan Xu

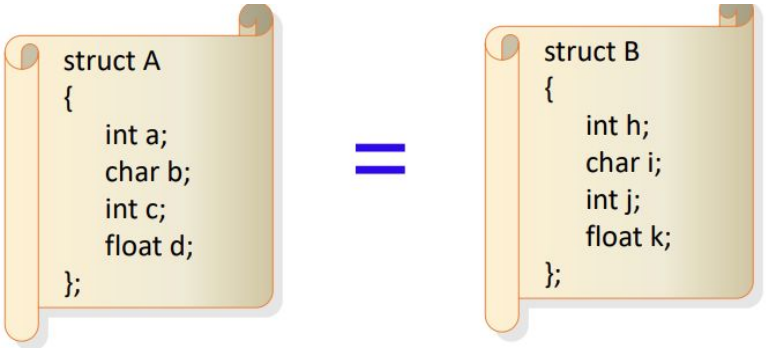
NDSS 2010

Problem definition

- Recover data structure specifications
  - syntactic
    - layout
    - offset
    - size
  - semantic
    - types

Motivation

- Security applications
  - vulnerability discovery
  - program signature
  - etc




```
1  void main(int argc, char* argv[])
2  {
3       char   tempname[1024];
4       strcpy(tempname, argv[1]);
5  }
```

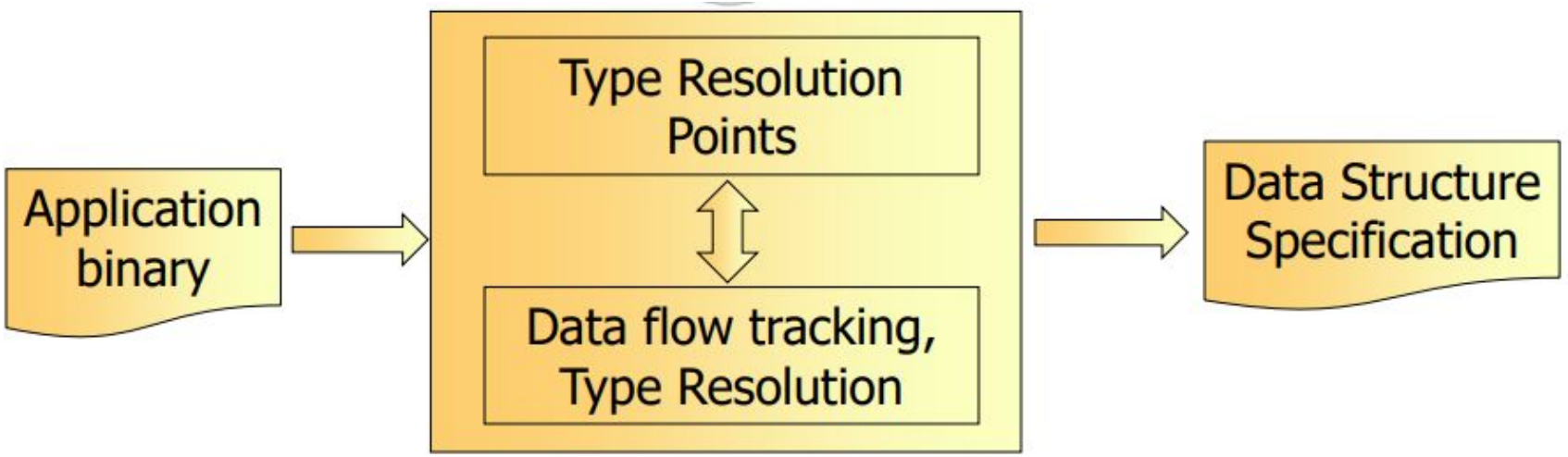
$ ./a.out aaa'\n'

$ ./a.out aaaaaaaaaaaaaaa…a'\n'

System overview

- Recover data structure specifications
  - syntactic
    - layout
    - offset
    - size
  - semantic
    - types

System overview



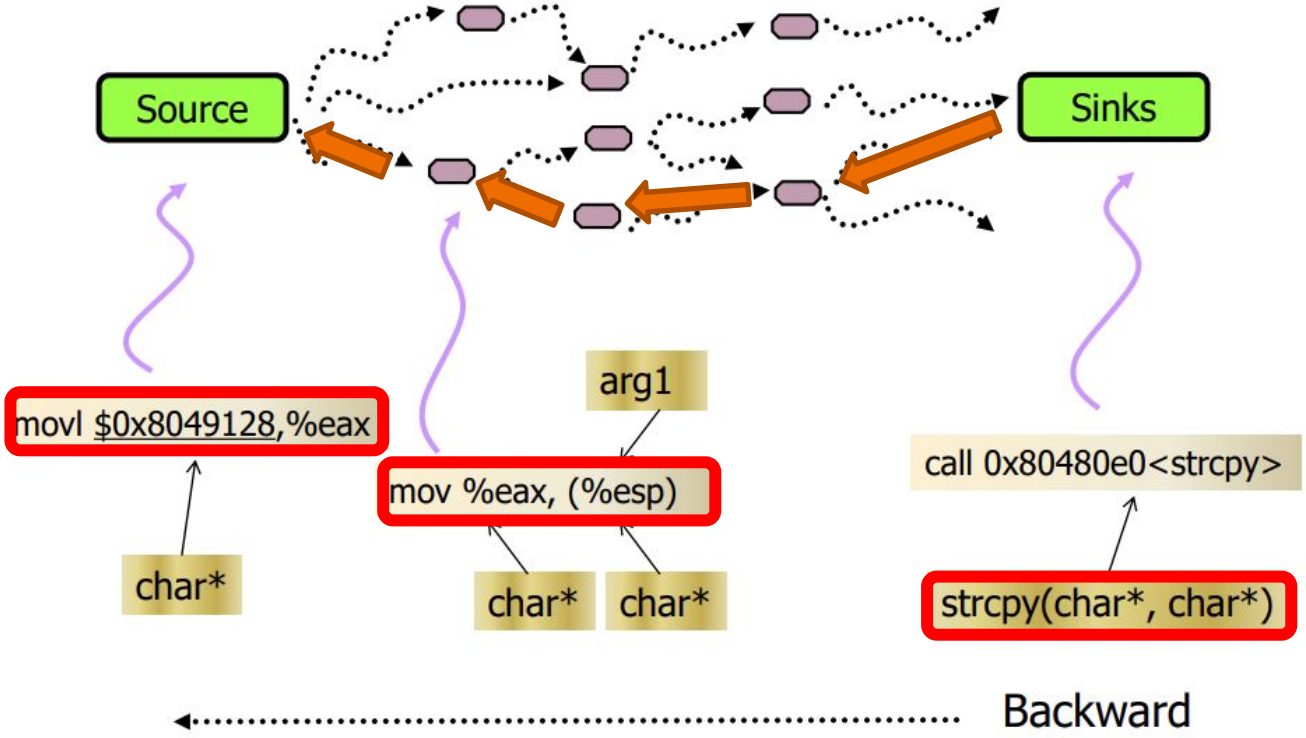
REWARDS

Reverse **E**ngineering **W**ork for **A**utomatic **R**evelation of **D**ata **S**tructures

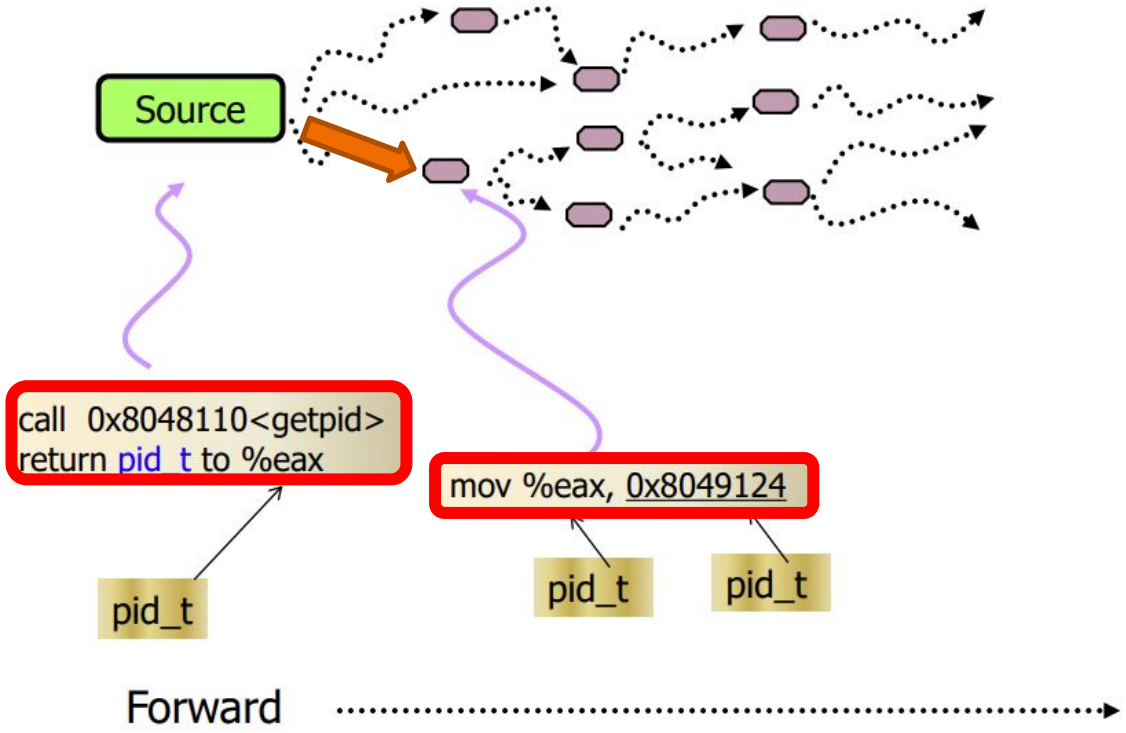
System design

- Key idea
  - find calls to a well-known function (like a system call)
  - types of all the arguments are known
  - label these memory locations accordingly
  - propagate this info backwards and forwards through the execution of the program
    - whenever labeled data is copied, the label is also assigned to the destination

System design



System design



Type resolution point 1

- Syscalls
  - syscall number
    - syscall_enter: type parameter passing registers (i.e., ebx, ecx, edx, esi)
    - syscall_exit: type return value (eax)

```
<getpid>
36  8048110:   mov    $0x14,%eax
37  8048115:   int    $0x80
38  8048117:   ret
```

Type resolution point 2

- Standard Library call
  - Types of arguments and return value
  - more useful than syscalls
    - 2016 APIs in Libc.so.6
    - 289 syscalls (2.6.15)

```
13  80480ce:    mov %eax,0x4(%esp)          <arg2>
14  80480d2:    movl $0x8049128, (%esp)     <arg1>
15  80480d9:    call 0x80480e0              <strcpy>
```

Evaluation

- Experiment setup
  - 10 utility binaries (e.g., ls, ps, ping)
- False negative (data structures missed)
  - global: 70%
  - heap: 55%
  - stack: 60%
  - reason
    - dynamic analysis
- False positive (get wrong data types)
  - global: 3%
  - heap: 0%
  - stack: 15%

Howard: A Dynamic Excavator for Reverse Engineering Data Structures

Asia Slowinska , Traian Stancescu, Herbert Bos

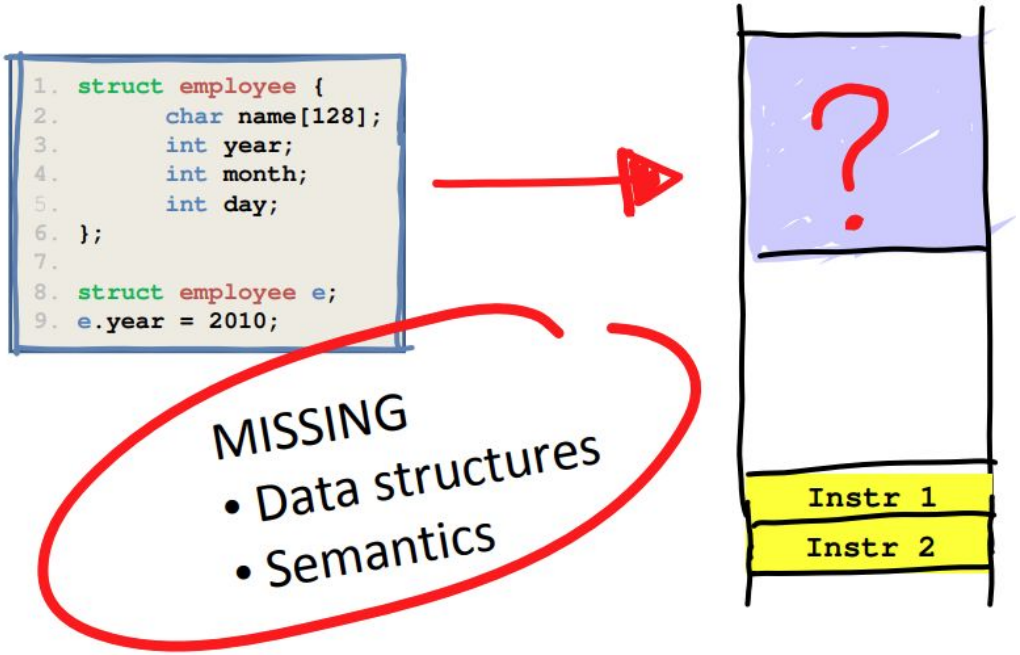
NDSS 2011

Motivation

- Rewards only
  - recovers those data structures
    - appear directly or indirectly in the arguments of well-known functions
  - only a very small portion of all data structures
  - example:
    - internal variables
    - data structures in the program
- Goal:
  - recover more data structures!

System Design

- Why is it so difficult?

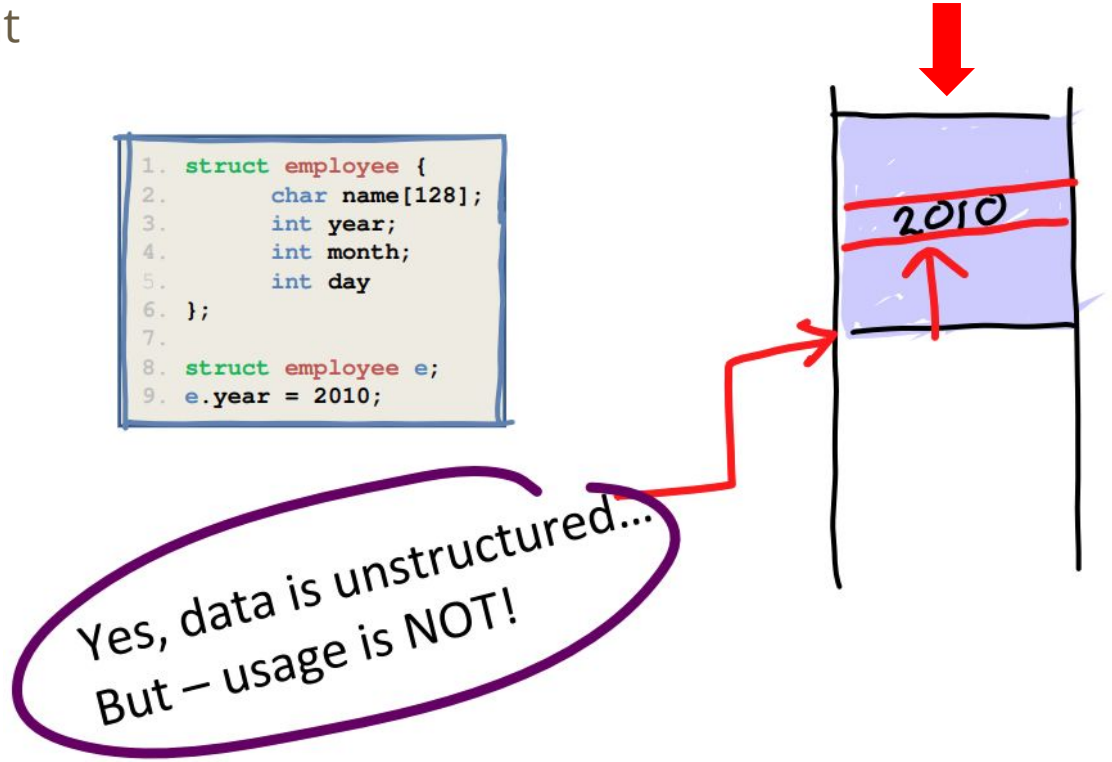


System Design

recovery by ***memory access patterns***

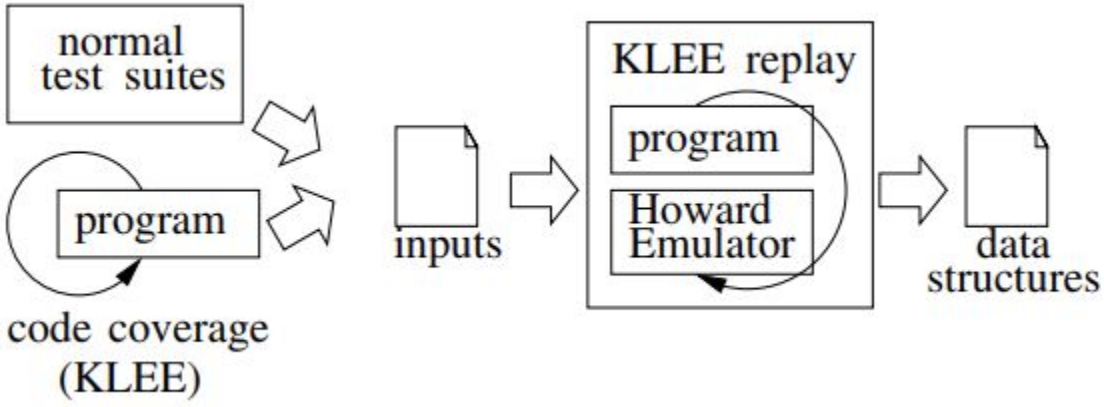
- Key insight

```
1. struct employee {
2.         char name[128];
3.         int year;
4.         int month;
5.         int day
6. };
7.
8. struct employee e;
9. e.year = 2010;
```



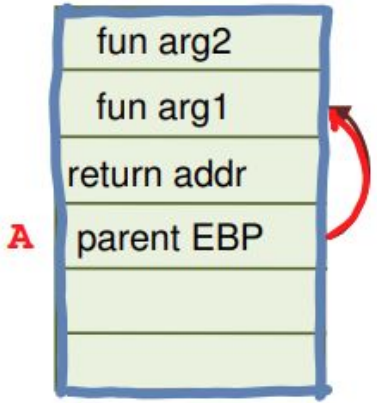
System Design

- System overview
  - dynamic approach
  - on top of KLEE - a symbolic execution engine



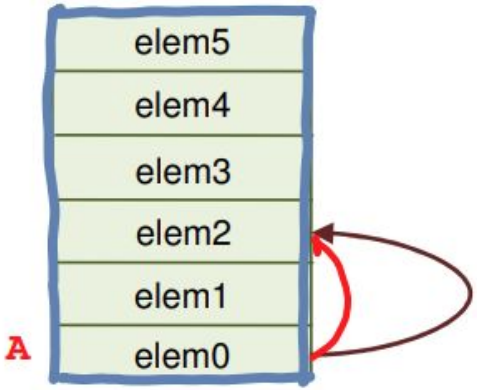
System Design

- Observe how memory is used at runtime
  - detect data structures based on access pattern
  - memory access patterns provide clues about the layout of data in memory
- if A is a function frame pointer
  - then *(A+8) is likely to point to a function argument passed via the stack



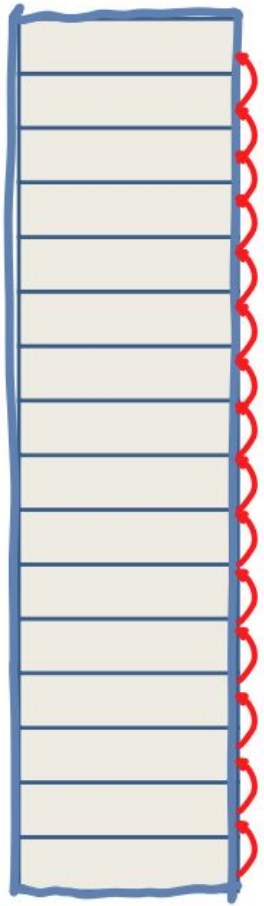
System Design

- if A is an address of an array
  - then *(A+8) is likely to point to an element of this array



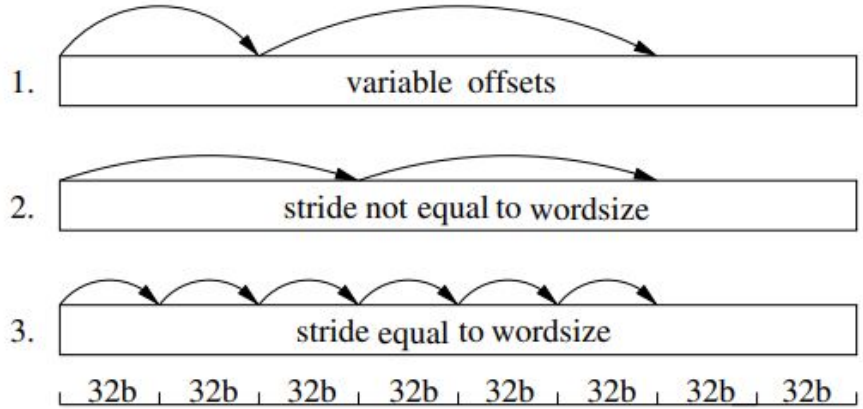
System Design

- Example: array recovery
  - access patterns
    - elem = *(next++);
      - looking for chains of accesses in a loop
    - elem = array[i];
      - looking for sets of accesses with the same base in a linear space



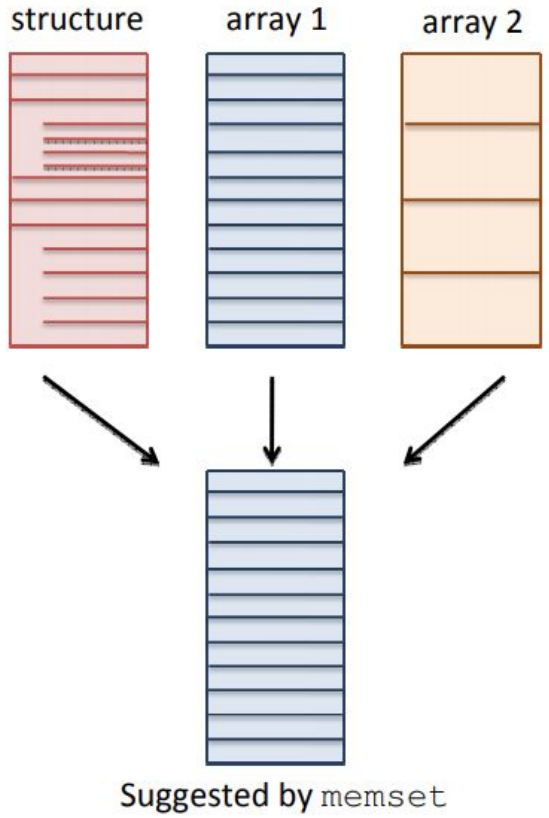
System Design

- Challenges 1:
  - Different memory access patterns within the same space
    - Solution:
      - Howard prefers pattern 1 over pattern 2 over pattern 3



System Design

- Challenges 2:
  - Decide which memory accesses are relevant
  - problems caused by *memset-like* functions
  - solution
    - heuristic preference for non-regular accesses



Evaluation

Prog	LoC	Size	Funcs%	Vars%	How tested?	KLEE%
wget	46K	200 KB	298/576 (51%)	1620/2905 (56%)	KLEE + test suite	24%
fortune	2K	15 KB	20/28 (71%)	87/113 (77%)	test suite	N/A
grep	24K	100 KB	89/179 (50%)	609/1082 (56%)	KLEE	46%
gzip	21K	40 KB	74/105 (70%)	352/436 (81%)	KLEE	54%
lighttpd	21K	130 KB	199/360 (55%)	883/1418 (62%)	test suite	N/A

Thank you!

Question?