



Dynamic Data Structure Excavation

or “Gimme back my symbol table!”

Asia Slowinska, Traian Stancescu,
Herbert Bos
VU University Amsterdam

Anonymous bytes only...





Goals

- Long term: reverse engineer complex software

```
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
}
```



Goals

- Long term: reverse engineer complex software
- Short term: reverse engineer data structures

```
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 s1{
    char f1 [128];
    int f2;
    int f3;
    int f4;
};
struct s1*
fun1 (struct s1* a1)
{
    struct s1 l1;
}
```



WHY?

Application I: legacy binary protection



- Legacy binaries everywhere
- We suspect they are vulnerable

But...



How to protect legacy code from memory corruption?

Answer: find the buffers and make sure that all accesses to them do not stray beyond array bounds.

Application II: binary analysis



- We found a suspicious binary – is it malware?
- A program crashed... - let's investigate!

But...



Without symbols, what can we do?

Answer: generate the symbols ourselves!

(demo later)



```
(adb) print_variables_function0
$1 = {field_4_bytes_0 = 0, field_4_bytes_1 = 0, pointer_struct_hostent_0 = 0xbfffeaf0,
field_8_bytes_0_unused = 579558798248313200, pointer_char_0 = 0x2cfb14
"\274\t", field_in_addr_t_0 = -1073745296,
pointer_struct_1_0 = 0x0, field_1_byte_0_unused = 0 '000', field_1_byte_0 = 0 '000',
field_1_byte_1 = 0 '000', field_8_bytes_1_unused = -4611706891964220672,
inetaddr_string_0 = 0x80b0170 "www.google.com", field_4_bytes_2 = 0}
```



Why is it difficult?

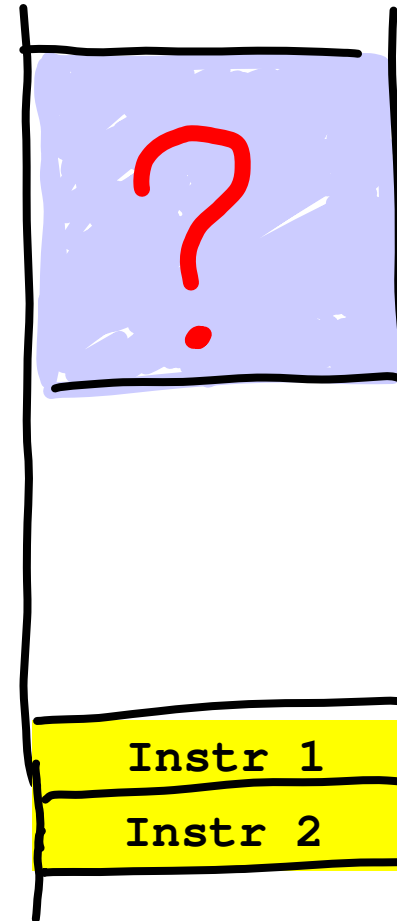


```
1. struct employee {  
2.     char name[128];  
3.     int year;  
4.     int month;  
5.     int day;  
6. };  
7.  
8. struct employee e;  
9. e.year = 2010;
```



MISSING

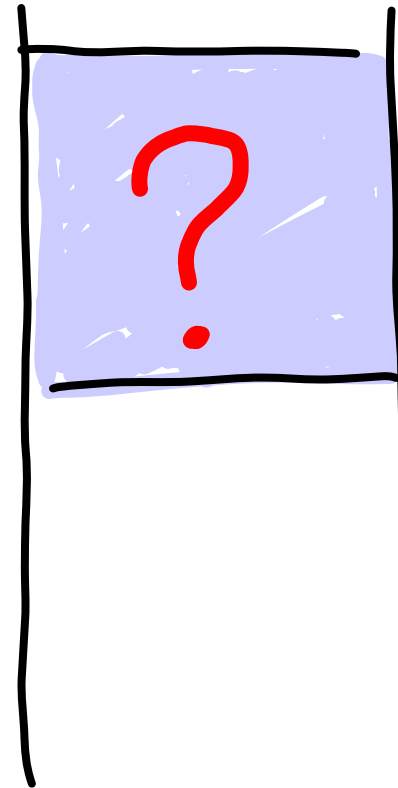
- Data structures
- Semantics



Data structures: 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;
```

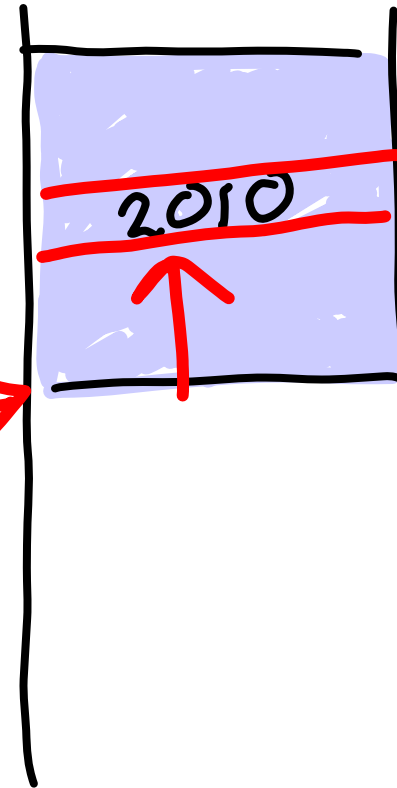


Yes, data is unstructured...
But – usage is NOT!

Data structures: key insight

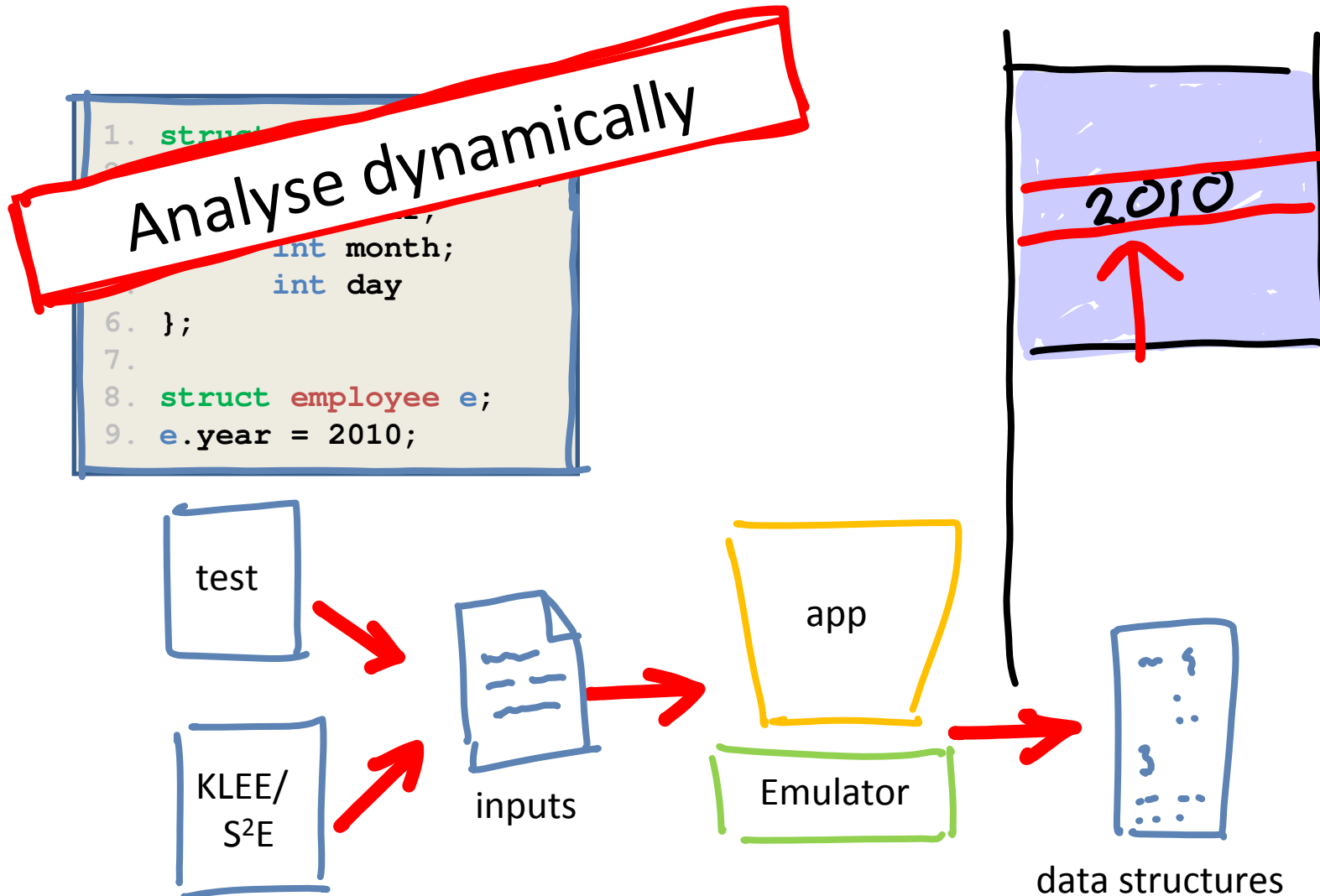


```
1. struct employee {  
2.     char name[128];  
3.     int year;  
4.     int month;  
5.     int day  
6. };  
7.  
8. struct employee e;  
9. e.year = 2010;
```



Yes, data is unstructured...
But – usage is NOT!

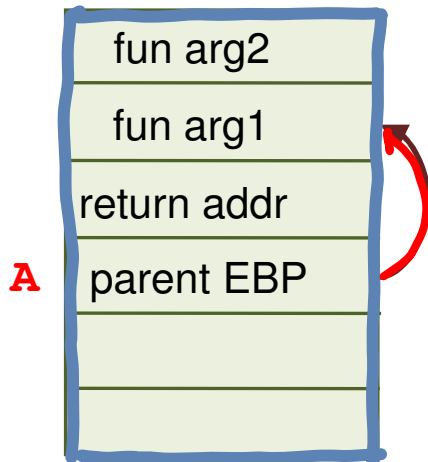
Data structures: key insight



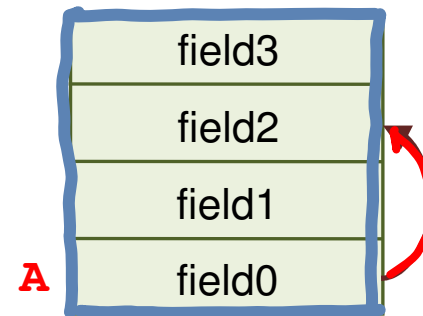
Intuition



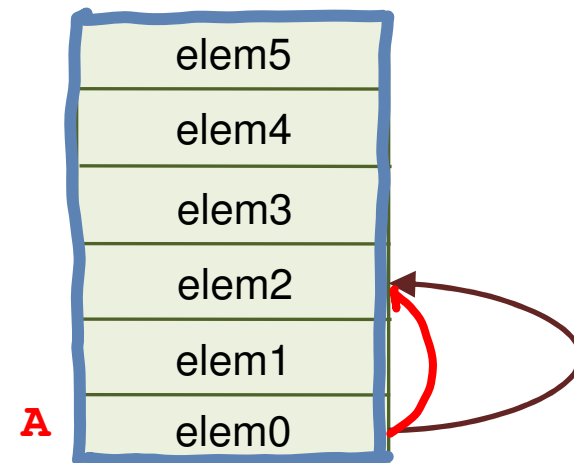
- Observe how memory is *used* at runtime to detect data structures
 - E.g., if **A** is a pointer...
1. and **A** is a function frame pointer, then $*(A + 8)$ is perhaps a function argument



2. and **A** is an address of a structure, then $*(A + 8)$ is perhaps a field in this structure



3. and **A** is an address of an array, then $*(A + 8)$ is perhaps an element of this array



Arrays are tricky

Access pattern & detection:

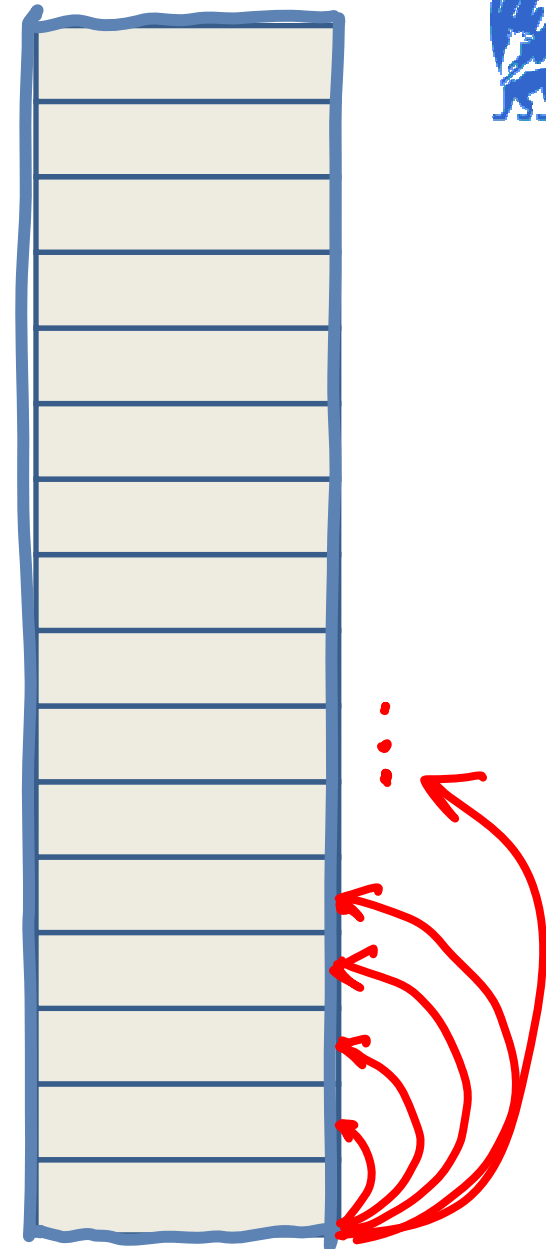
- `elem = next++;`
 - Look for chains of accesses in a loop



Arrays are tricky

Access pattern & detection:

- `elem = next++;`
 - Look for chains of accesses in a loop
- `elem = array[i];`
 - Look for sets of accesses with the same base in a linear space



Arrays are tricky

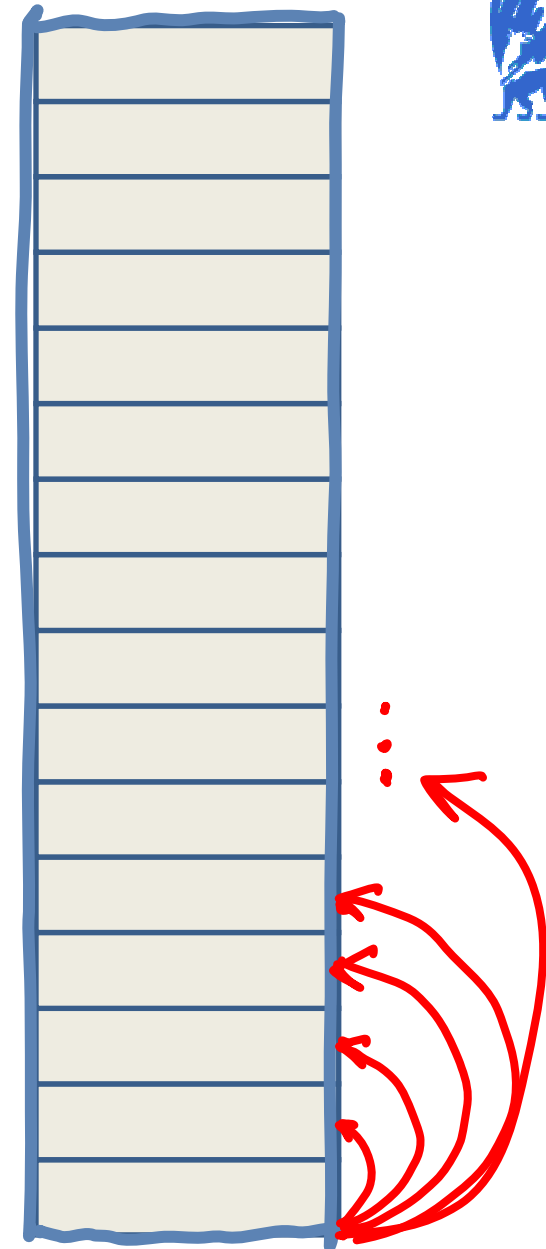


Access pattern & detection:

- `elem = next++;`
 - Look for chains of accesses in a loop
- `elem = array[i];`
 - Look for sets of accesses with the same base in a linear space

Challenges:

- Boundary elements accessed outside the loop
- Nested loops
- Multiple loops in sequence

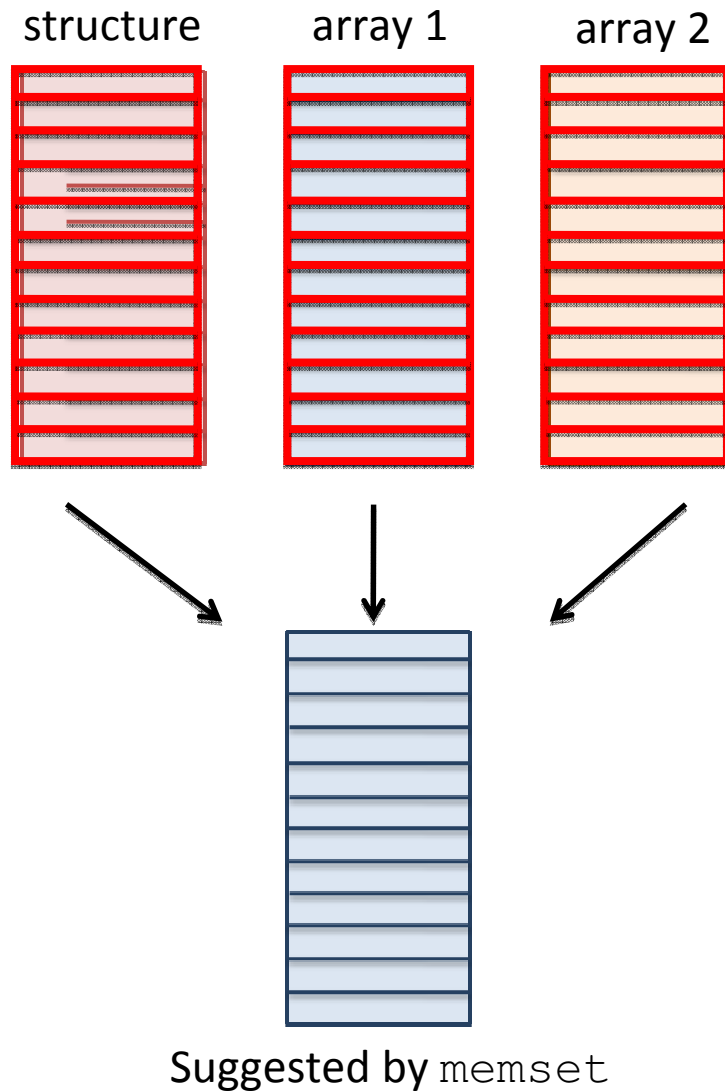


More challenges



Examples:

- Decide which memory accesses are relevant
 - Problems caused by e.g., `memset`-like functions

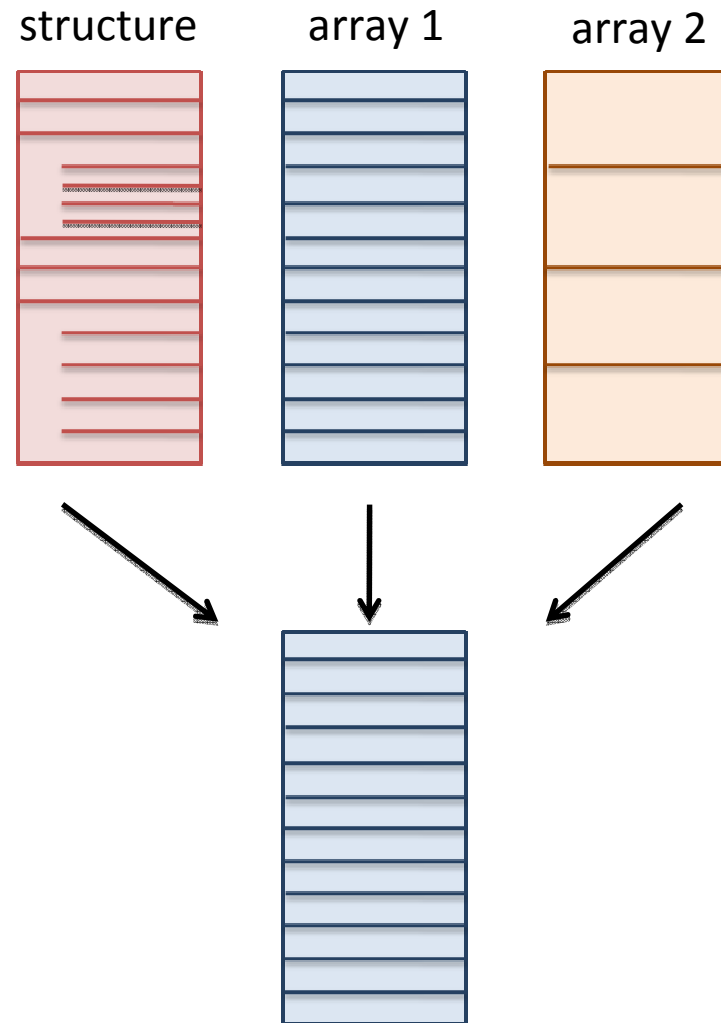


More challenges



Examples:

- Decide which memory accesses are relevant
 - Problems caused by e.g., `memset`-like functions
- Even more in the paper 😊



Suggested by `memset`

Results in terms of accuracy – heap memory

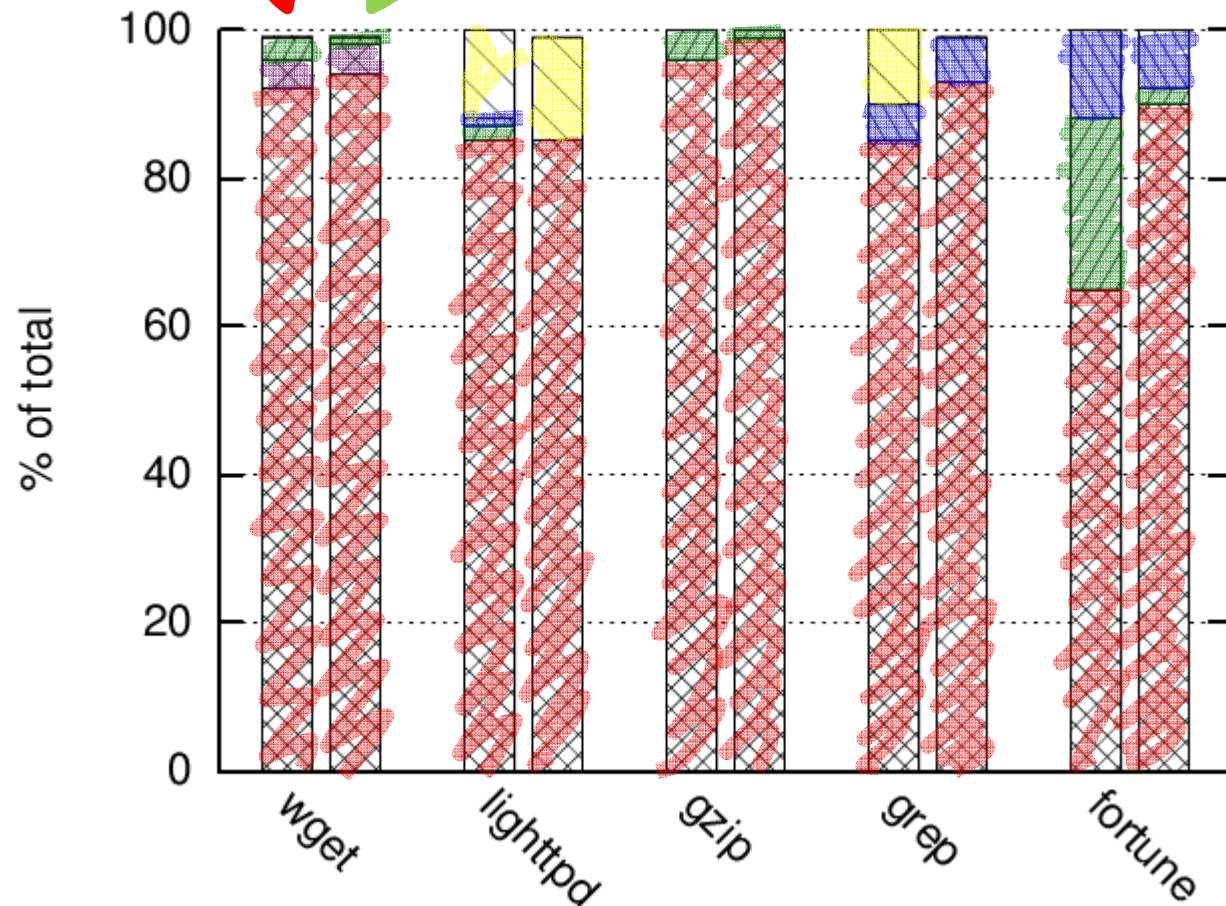


variables

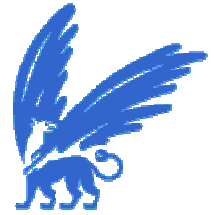
bytes

Heap Memory

Prog	LoC
wget	46K
fortune	2K
grep	24K
gzip	21K
lighttpd	21K



demo now



```
(adb) print_variables_function0  
$1 = {field_4_bytes_0 = 0, field_4_bytes_1 = 0, pointer_struct_hostent_0 = 0xbfffeaf0,  
field_8_bytes_0_unused = 579558798248313200, pointer_char_0 = 0x2cfb14  
"\274\t", field_in_addr_t_0 = -1073745296,  
pointer_struct_1_0 = 0x0, field_1_byte_0_unused = 0 '000', field_1_byte_0 = 0 '000',  
field_1_byte_1 = 0 '000', field_8_bytes_1_unused = -4611706891964220672,  
inetaddr_string_0 = 0x80b0170 "www.google.com", field_4_bytes_2 = 0}
```



Conclusions



- *We can* recover data structures by tracking memory accesses
- We believe we can protect legacy binaries
- We are working on data coverage