Reconstruction of Composite Types for Decompilation

K. Troshina
Y. Derevenets
A. Chernov
cher@unicorn.cmc.msu.ru
Moscow State University

Goals

- Explore possibilities of reconstructing “composite” data type (structures, arrays or their combinations) definitions from low-level representation:
  - Executable file
  - Assembly listing
  - Execution trace
  - Inside dynamic instrumentation framework (i.e. valgrind)

Assumptions

- The low-level input program was once written in a “good” C, and we deal with the result of its compilation
- No obfuscation, link-time optimizations

“Good” C

- No casts between pointers and integral values
- Limited casts between void*/char* and material pointers:
  - Memory allocation (malloc,free…)
  - Memory manipulation (memcpy, memset…)
- No casts between material pointers
- No unions

Decompilation stages

- Function interface reconstruction
- Structural analysis (reconstruction of control statements)
- Data type reconstruction
  - Basic data type reconstruction
  - Composite data type reconstruction
    - Structures
    - Arrays
    - Combinations of structures and arrays

Composite type compilation

```
struct s {
  int f1;
  char f2;
  void *f3;
    };
void f(struct s *p1, struct s *p2,
    struct s *p3) {
  p3->f1 = p1->f1 + p2->f1;
  p3->f2 = p1->f2;
  p3->f3 = p2->f3;
}
```

```
movl      8(%ebp), %ecx
movl      12(%ebp), %ebx
movl      16(%ebp), %edx
movl      0(%ebx), %eax
addl      0(%ecx), %eax
movl      %eax, 0(%edx)
movzbl    4(%ecx), %eax
movb      %al, 4(%edx)
movl      8(%ebx), %eax
movl      %eax, 8(%edx)
```

General observations

- Field offsets are computed at compile time
- Array indices are computed at runtime (mostly)
- Combination of both is used in address calculations for composite types
- Base addresses are copied from mem to regs, between regs, back to memory…

Algorithm sketch

- **Identify memory access operations corresponding to use of composite types**
- Group memory accesses by classes corresponding to different composite type in the source code
- Reconstruct composite type templates

Basic memory access instructions

- Direct memory access instructions
  movl	var, %eax		-> var – global
- Frame pointer-relative access
  movl	%eax, -8(%ebp) -> local vars
- Stack operations
  movl	%eax, 4(%esp) -> param push

Address expression

- Canonic form: $B + O + \sum_{j=1}^{n} C_j x_j$

- B – base expression
- O – constant numeric offset
- C – constant numeric multipliers
- x – values

Address expression recovery

- Backward slicing in the assembly code
- Each composite memory access instruction has a computed address expression

Algorithm sketch

- Identify memory access operations corresponding to use of composite types
- **Group memory accesses by classes corresponding to different composite type in the source code**
- Reconstruct composite type templates

Label propagation

- A unique label is assigned to each composite memory load instruction result
  - Some labels (non-pointer values) are later discarded
  - Other labels are grouped into equivalence classes
- Disjoint-set data structure is maintained

Label assignment

```
movl    8(%ebp), %ecx      ;; L1
movl    12(%ebp), %ebx     ;; L2
movl    16(%ebp), %edx     ;; L3
movl    0(%ebx), %eax      ;; L4
addl    0(%ecx), %eax      ;; L5
movl    %eax, 0(%edx)
movzbl  4(%ecx), %eax
movb    %al, 4(%edx)
movl    8(%ebx), %eax      ;; L6
movl    %eax, 8(%edx)
```

Label equiv. classes

- Forward iterative data-flow analysis
- Labels are grouped into equiv. classes
- Each equivalence class – future composite type
- Each equiv. class has an associated set of address expressions used to access the memory (offset and multiplicative component)

Equiv. class classification

- Offset = 0, no mult. part, matching size – a base type pointer (T*)
- Multiple offsets != 0, no mult. part – a structure type pointer
- Offset = 0, mult., matching size – a base type array
- Multiple offsets != 0, same mult. part, the least multiplier > the max offset – an array of structures

Algorithm sketch

- Identify memory access operations corresponding to use of composite types
- Group memory accesses by classes corresponding to different composite type in the source code
- **Reconstruct composite type templates**

Example

```
struct t {
int f1; int f2;
};
struct s {
  struct s *a;
  int b;
  char c;
  struct t d[4];
  char e[4];
};
```

- { e12 = addr(L11, 0, 0),
  e15 = addr(L11, 4, 0),
  e18 = addr(L11, 8, 0),
  e28 = addr(L11, 44,mul(1)),
  e20 = addr(L11,12,mul(8)),
  e22 = addr(L11,16,mul(8))}
- Induced size of L11 > 8
- S1 = addr(L11, 12,mul(8)) - new AE term
- e20 = addr(S1, 0, 0),
  e22 = addr(S1, 4, 0) – reduced AE terms
- S2 = addr(L11, 44, mul(1)) – another AE term

Structure skeleton

```
struct s2 {
    t1 f1; /* at offset 0 */
    t2 f2; /* at offset 4 */
    t3 f3; /* at offset 8 */
    struct s1 {/* sizeof(struct s1) == 8 */
      t4 f1; /* at offset 0 */
      t5 f2; /* at offset 4 */
    } f4[]; /* at offset 12 */
    t6 f5[]; /* at offset 44 */
    };
```

Results

Name	OrigStruct	RecStruct	Clones
38_day.s	2	2	3
59_lalr.s	1	0	-
83_retr.s	3	3	5
58_arithmetic.s	1	0	-
57_piano.s	1	1	0
56_warms.s	2	2	0
39_deflate.s	1	1	0
35_wc.s	1	1	0
36_cat.s	1	0	-

Thank You!

Controversial question

- Reverse engineering is diminished? Anybody may have as many virtual machines of different architectures with different setups as he/she wants?