CS451 – Software Analysis

Lecture 10

Custom Disassembly (Recursive)

Elias Athanasopoulos
elathan@cs.ucy.ac.cy

Linear vs recursive

- Linear disassembly has problems
  - Instructions can be hidden using obfuscation
  - The control flow of the program is not considered at al
- Recursive disassembly employs a different strategy
  - Use the control-flow of the program to discover basic blocks that are used by the program
  - Not always possible to find all destinations, since jump transfers may be entirely dynamic

Obfuscated code

```
int overlapping(int i) {
  int j = 0;

  __asm__ __volatile__(
  "cmp   $0x0,%1         ; "
  ".byte 0x0f,0x85       ; " /* relative jne */
  ".long 2               ; " /* jne offset   */
  "xorl  $0x04,%0        ; "
  ".byte 0x04,0x90       ; " /* add al,0x90  */
  : "=r" (j)
  : "r" (i)
  :
  );

  return j;
}

int main(int argc, char *argv[]) {
  srand(time(NULL));
  printf("%d\n", overlapping(rand() % 2));
  return 0;
}
```

Using objdump with obfuscated code

```
$ objdump --start-address=0x400666 --stop-address=0x40068c -d overlapping_bb
0000000000400666 <overlapping>:
  400666: 55                      push   %rbp
  400667: 48 89 e5                mov    %rsp,%rbp
  40066a: 89 7d ec                mov    %edi,-0x14(%rbp)
  40066d: c7 45 fc 00 00 00 00 movl   $0x0,-0x4(%rbp)
  400674: 8b 45 ec                mov    -0x14(%rbp),%eax
  400677: 83 f8 00                cmp    $0x0,%eax
  40067a: 0f 85 02 00 00 00       jne    400682 <overlapping+0x1c>
  400680: 83 f0 04                xor    $0x4,%eax
  400683: 04 90                   add    $0x90,%al
  400685: 89 45 fc                mov    %eax,-0x4(%rbp)
  400688: 8b 45 fc                mov    -0x4(%rbp),%eax
  40068b: 5d                      pop    %rbp
$ objdump --start-address=0x400682 --stop-address=0x40068c -d overlapping_bb
0000000000400682 <overlapping+0x1c>:
  400682: 04 04                   add    $0x4,%al
  400684: 90                      nop
  400685: 89 45 fc                mov    %eax,-0x4(%rbp)
  400688: 8b 45 fc                mov    -0x4(%rbp),%eax
  40068b: 5d                      pop    %rbp
```

Recursive approach

- Hold a queue with addresses that can be starting points of code
  - Initially, those addresses can be function-entrance points
- Process all addresses stored in the queue
  - Each time an address is dequeued for processing, update a map (hash) so that we are not processing the same address in the future
- We use C++ for the data structures *queue* and *map*

How disassembly proceeds

- We start at a given address and we decode each instruction
- Instead of blindly decoding and printing each instruction, we examine the instruction type
  - In contrast with linear disassembly, where only the end points matter, in recursive disassembly each instruction may be significant

Instruction grouping

- Capstone has many macros that assist in grouping instructions
- Recall that intel has several different opcodes for jumps, so we need to target the group of instructions

```
bool is_cs_cflow_group(uint8_t g) {
    return (g  == CS_GRP_JUMP) ||
           (g == CS_GRP_CALL)  ||
           (g == CS_GRP_RET)   ||
           (g == CS_GRP_IRET);
}
```

How to check for control-flow instructions

- We use the *detailed* mode of Capstone for inspecting each instruction
- Each instruction has a `detail` structure, where the groups field contains information about the instruction

```
bool is_cs_cflow_ins(cs_insn *ins) {
    for (size_t i = 0; i < ins->detail->groups_count; i++) {
        if (is_cs_cflow_group(ins->detail->groups[i])) {
            return true;
        }
    }
    return false;
}
```

How to handle control-flow instructions

- Once we reach a control-flow instruction we need to check if we can parse the target
  - For example, the target address of a jump
  - This is not always possible
- If the target is immediate and can be parsed, then the type of the instruction will be `X86_OP_IMM`
  - In such case, we put the target in the queue

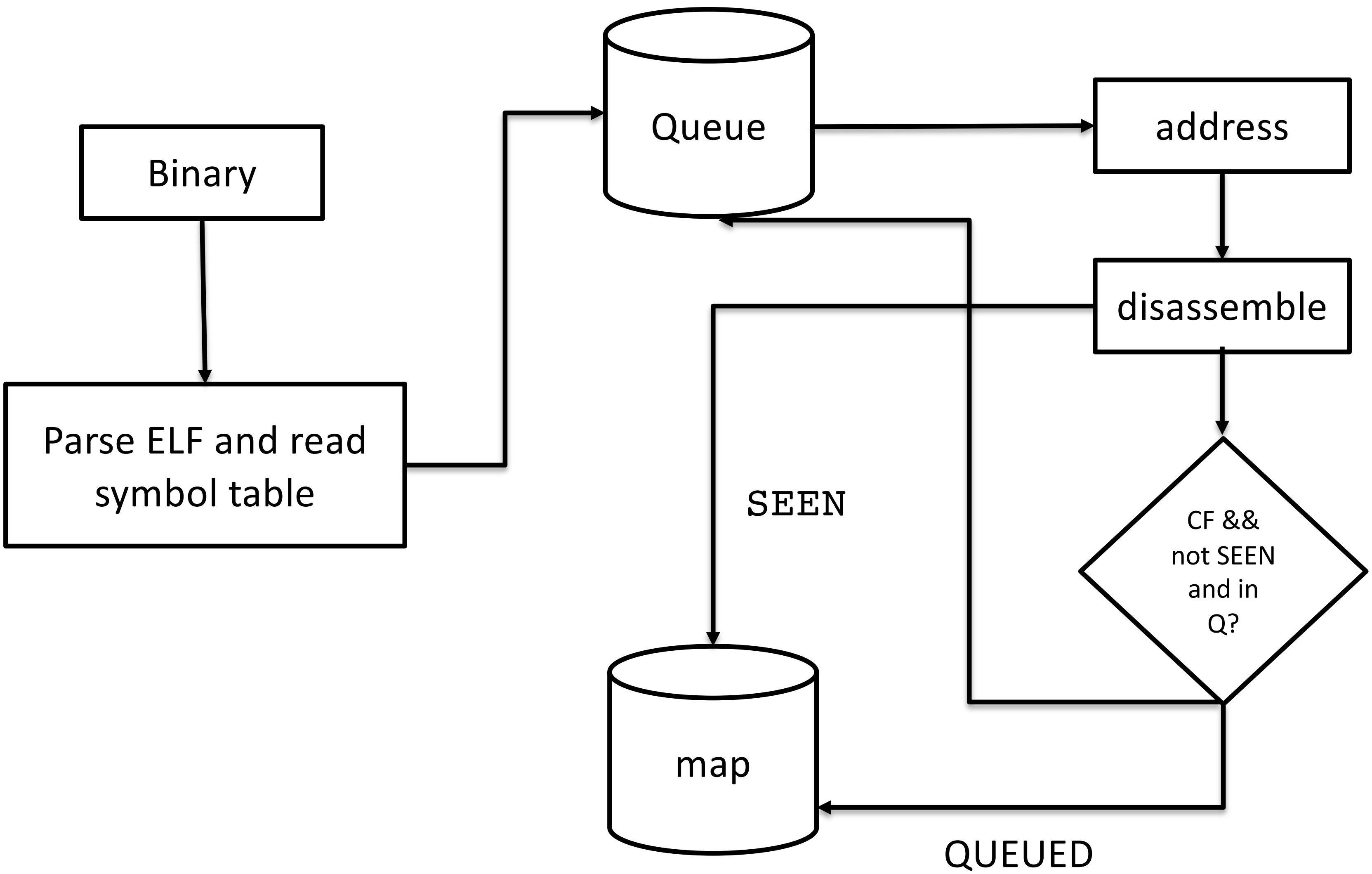
Discovered addresses `bb_status`

- `UNSEEN`
  - This is a new address that has not been seen in the past
- `ENQUEUED`
  - This is an address that has been enqueued, but has not been processed, yet
- `SEEN`
  - This is an address that has been processed

Main loop

- Start disassembling the next address in the queue
- For each disassembled instruction, update the map
- If we find a branch
  - Get the target
  - Check the map and if this address has not been processed nor queued, already, store it in the queue
- Check if the instruction is a ret, which means we reached the end of the function
  - This is not always accurate (check homework)

High-level idea



Homework

- Refactor the recursive disassembler
  - Avoid the use of global `g_text_start` and `g_text_end`
  - The initial addresses are pushed in the queue but are not updated in the map
  - Disassembly of a basic block stops at the end of the function, which is not checked that accurately
  - What happens with stripped binaries?