Lec 20: Binary Analysis

CS492E: Introduction to Software Security

Sang Kil Cha
Binary Analysis is Difficult

Not only automated analysis, but manual analysis is difficult.

What’s the problem?
No Program Abstraction!

```plaintext
4C 8B 47 08  mov r8, qword ptr [rdi+8]
BA 02 00 00 00  mov edx, 2
48 8B 4F 20  mov rcx, qword ptr [rdi+20h]
45 0F B7 08  movzx r9d, word ptr [r8]
E8 54 16 00 00     call 00000001400026BC
48 8B 74 24 38  mov rsi, qword ptr [rsp+38h]
8B C3                mov eax, ebx
48 8B 5C 24 30  mov rbx, qword ptr [rsp+30h]
48 83 C4 20               add rsp, 20h
5F                  pop rdi
C3               ret
48 8B C4               mov rax, rsp
48 89 58 08               mov qword ptr [rax+8], rbx
```

No types
No variables
No functions
...

No Program Abstraction!
Binary Analysis (= Reverse Engineering)
Diassembly
First Step: Disassembling Binary Code

Disassemble

- Idea
- Source Code
- Intermediate Representation
- Assembly Code
- Binary Code

Reverse Engineering
Recursive Descent Disassembly

1. Disassemble instruction one by one until reaching branch instructions

2. When there is a branch instruction, we examine the target address(es) of the branch instruction, and recursively disassemble from there.
Figuring out Branch Target(s)

JMP EAX
CALL [EAX]

Can we statically decide what kind of values EAX can have?
Simplest Example

```c
int main(int c, char** argv)
{
    switch (c)
    {
    case 1: counter += 20; break;
    case 2: counter += 33; break;
    case 3: counter += 62; break;
    case 4: counter += 15; break;
    case 5: counter += 416; break;
    case 6: counter += 3545; break;
    case 7: counter += 23; break;
    case 8: counter += 81; break;
    }
    return counter;
}
```

```asm
000000000001130 <main>:
1130: push rbp
1131: mov rbp,rsp
1134: mov DWORD PTR [rbp-0x4],0x0
113b: mov DWORD PTR [rbp-0x8],edi
113e: mov QWORD PTR [rbp-0x10],rsi
1142: mov eax,DWORD PTR [rbp-0x8]
1145: add eax,0xffffffff
1148: mov ecx,eax
114a: sub eax,0x7
114d: mov QWORD PTR [rbp-0x18],rcx
1151: ja 122e <main+0xFE>
1157: lea rax,[rip+0xea6]
1162: movsx rdx,DWORD PTR [rax+rcx*4]
1166: add rdx,rax
1169: jmp rdx
116b: lea rax,[rip+0xe2be]
1172: mov rcx,QWORD PTR [rax]
1175: add rcx,0x14
...
Lifting
Second Step: Lifting

Lifting

- Idea
- Source Code
- Intermediate Representation
- Assembly Code
- Binary Code

Reverse Engineering
Why IR?

• Platform-neutral representation

• IR represents explicit semantics
Lifting Example

add dword ptr [ecx], eax

\[
\begin{align*}
T_0:\text{i32} &:= \text{EAX} \\
T_1:\text{i32} &:= \text{[ECX]} \\
T_2:\text{i32} &:= (T_0:\text{i32} + T_1:\text{i32}) \\
\text{[ECX]} &:= T_2:\text{i32} \\
\text{CF} &:= (T_2:\text{i32} < T_0:\text{i32}) \\
\text{OF} &:= ((\text{high:i1}(T_0:\text{i32}) = \text{high:i1}(T_1:\text{i32})) \& (\text{high:i1}(T_0:\text{i32}) \^ \text{high:i1}(T_2:\text{i32}))) \\
\text{AF} &:= (((T_2:\text{i32} \^ T_0:\text{i32}) \^ T_1:\text{i32}) \& (0x1:\text{i32} << 0x4:\text{i32})) = (0x1:\text{i32} << 0x4:\text{i32})) \\
\text{SF} &:= \text{high:i1}(T_2:\text{i32}) \\
\text{ZF} &:= (T_2:\text{i32} = 0x0:\text{i32}) \\
T_3:\text{i32} &:= (T_2:\text{i32} \^ (T_2:\text{i32} \gg \text{zext:i32}(0x4:i8))) \\
T_4:\text{i32} &:= ((T_2:\text{i32} \^ (T_2:\text{i32} \gg \text{zext:i32}(0x4:i8))) \^ (T_3:\text{i32} \gg \text{zext:i32}(0x2:i8)))) \\
\text{PF} &:= (~\text{low:i1}(((T_2:\text{i32} \^ (T_2:\text{i32} \gg \text{zext:i32}(0x4:i8))) \^ (T_3:\text{i32} \gg \text{zext:i32}(0x2:i8)))) \^ (T_4:\text{i32} \gg \text{zext:i32}(0x1:i8))))
\end{align*}
\]
Example: PUSH

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
</table>
| IF StackAddrSize = 64 | (* See Description section for possible sign-extension or zero-extension of source operand and fix *)
| THEN | (* a case in which the size of the memory store may be smaller than the Instruction's operand size *) |
| IF OperandSize = 64 | (* push word *) |
| THEN | |
| RSP ← RSP - 8 | (* push pushword *) |
| ELSE IF OperandSize = 32 | |
| THEN | |
| RSP ← RSP - 4 | (* push dword *) |
| Memory(SS,ESP) ← SRC | |
| ELSE (* OperandSize = 16 *) | |
| RSP ← RSP - 2 | (* push word *) |
| Memory(SS,ESP) ← SRC | |
| ELSE IF StackAddrSize = 32 | (* push word *) |
| THEN | |
| RSP ← RSP - 8 | |
| Memory(SS,ESP) ← SRC | |
| ELSE (* OperandSize = 16 *) | |
| RSP ← RSP - 4 | |
| Memory(SS,ESP) ← SRC | |
| ELSE | |
| (* push word *) |
| | |

Decimals:

- **Example:** PUSH
- **Address size**:
  - The D flag in the current code-segment descriptor determines the default address size; it may be overridden by an instruction prefix (67H).
  - The address size is used only when referencing a source operand in memory.
- **Operand size**:
  - The D flag in the current code-segment descriptor determines the default operand size; it may be overridden by instruction prefixes (66H or 67H).
  - The operand size (16, 32, or 64 bits) determines the amount by which the stack pointer is decremented (2, 4, or 8).

**Description**:

- Decrément the stack pointer and then stores the source operand on the top of the stack. Address and operand sizes are determined and used as follows:
  - **Address size**: The D flag in the current code segment descriptor determines the default address size; it may be overridden by an instruction prefix (67H).
  - The address size is used only when referencing a source operand in memory.
  - **Operand size**: The D flag in the current code segment descriptor determines the default operand size; it may be overridden by instruction prefixes (66H or 67H).

If the source operand is an immediate of size less than the operand size, a sign-extended value is pushed on the stack. If the source operand is a segment register (16 bits) and the operand size is 64-bits, a zero-extended value is pushed on the stack; if the operand size is 32-bits, either a zero-extended value is pushed on the stack or the segment selector is written on the stack using a 16-bit move. For the last case, all recent Core and Atom processors perform a 16-bit move, leaving the upper portion of the stack location unmodified.
Example: BSF (Pseudo Code)

```plaintext
if ( source == 0 ) {
    ZF = 0;
    destination = undefined;
}
else {
    ZF = 0;
    T = 0;
    while (Bit(source, T) == 0) {
        T = T + 1;
        destination = T;
    }
}
```
IR is Complex and Error-Prone!

Human is writing the lifter!
What Happens When IR is Incorrect?

• CVE-2009-2267, CVE-2009-1542
  – Security vulnerabilities

• QEMU failed to load a Linux kernel due to an IR bug
CFG Recovery & More
Third Step: CFG Recovery & More

CFG Recovery and Decompilation

Idea
Source Code
Intermediate Representation
Assembly Code
Binary Code

Reverse Engineering
Problem

• Recursive disassembly includes CFG recovery, but perfect disassembly is infeasible.

• Knowing the function entry points remains problematic.
Call Target = Function?

- **False positives**: call targets may not be a function entry point

- **False negatives**: regular jump targets can be a function entry point
Example: False Positives

<table>
<thead>
<tr>
<th>Address</th>
<th>Opcode</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>11a0:</td>
<td>55</td>
<td>push ebp</td>
</tr>
<tr>
<td>11a1:</td>
<td>89 e5</td>
<td>mov ebp, esp</td>
</tr>
<tr>
<td>11a3:</td>
<td>50</td>
<td>push eax</td>
</tr>
<tr>
<td>11a4:</td>
<td>e8 00 00 00 00 00</td>
<td>call 11a9</td>
</tr>
<tr>
<td>11a9:</td>
<td>58</td>
<td>pop eax</td>
</tr>
<tr>
<td>11aa:</td>
<td>81 c0 57 2e 00 00</td>
<td>add eax, 0x2e57</td>
</tr>
<tr>
<td>11b0:</td>
<td>31 c9</td>
<td>xor ecx, ecx</td>
</tr>
</tbody>
</table>

...
Example: False Negatives

... c30a0: 31 f6 xor esi,esi
c30a2: eb 0c jmp c30b0

000000000000c30b0 (!_bfd_generic_read_ar_hdr_mag>:
c30b0: 41 57 push r15
c30b2: 41 56 push r14
...
Any Solution?

• Function entry points often have specific patterns
  − But not all of them follow the patterns

• PC getters have specific patterns
  − Inlined assembly code?
Partitioned Functions

```plaintext
static void move_fd (int oldfd, int newfd)
{
    if (oldfd != newfd)
    {
        dup2 (oldfd, newfd);
        close (oldfd);
    }
}
```
Decompile?

• Value-Set Analysis (VSA)
  – Where are the variables?

• Type inference
  – Can we recover variable types?

• Structure Analysis
  – Can we recover high-level control flow structures?
B2R2: the Next Generation Binary Analysis Framework

- Binary analysis platform developed in KAIST
- Won the best paper award in NDSS BAR 2019

https://github.com/B2R2-org/B2R2
Conclusion

• Binary analysis is largely unsolved.

• There are many on-going research projects in every step of binary analysis.
Questions?