CS240 Laboratory 7
Disassembly and Reverse Engineering

Memory Layout

<table>
<thead>
<tr>
<th>Addr</th>
<th>Stack</th>
<th>Heap</th>
<th>Statics</th>
<th>Literals</th>
<th>Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{N-1}$</td>
<td>RW</td>
<td>RW</td>
<td>RW</td>
<td>R</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>Procedure context</td>
<td>Dynamic data structures</td>
<td>Global variables/static data structures</td>
<td>String literals</td>
<td>Instructions</td>
</tr>
<tr>
<td></td>
<td>Compiler</td>
<td>Programmer, malloc/free, new/GC</td>
<td>Compiler/Assembler/Linker</td>
<td>Compiler/Assembler/Linker</td>
<td>Compiler/Assembler/Linker</td>
</tr>
<tr>
<td></td>
<td>Run-time</td>
<td>Run-time</td>
<td>Startup</td>
<td>Startup</td>
<td>Startup</td>
</tr>
</tbody>
</table>

**Text Segment**
Program instructions can be stored starting at 0x400000 in memory.
Grows up into higher addresses in memory with longer programs.

**Stack Segment**
Top of stack is initially 0x7fffffffffffff ($2^{47} - 1$).
Grows down into lower addresses in memory as stack fills.

When examining X86 code, addresses or numbers used as displacements or pointers/addresses will have values in the range of the text or stack segments.
Instructions

Moving Data

\[ \text{movl Src, Dest} \]

Load Effective Address - compute address or arithmetic expression of the form \( x + k*I \) (does not set the condition flags!)

\[ \text{leal Src, Dest} \]

Arithmetic/Logical operations – 2 operands

\[ \text{addl Src, Dest} \]
\[ \text{subl Src, Dest} \]
\[ \text{imull Src, Dest} \]
\[ \text{shrl Src, Dest} \]
\[ \text{sarl Src, Dest} \]
\[ \text{shll Src, Dest} \]
\[ \text{sall Src, Dest} \]
\[ \text{shrl Src, Dest} \]
\[ \text{xorl Src, Dest} \]
\[ \text{andl Src, Dest} \]
\[ \text{orl Src, Dest} \]
\[ \text{mull Src, Dest} \]
\[ \text{imull Src, Dest} \]
\[ \text{divl Src, Dest} \]
\[ \text{idivl Src, Dest} \]
Arithmetic/Logical operations – 1 operand

incl Dest
decl Dest
negl Dest
notl Dest

Zero Extend from Byte to Quad Word

movzbq Src,Dest

Setting Condition Codes Explicitly – used for control flow

cmpl/cmpq Src2,Src1  sets flags based on value of Src2 – Src1, discards result
testl/testq Src2,Src1 sets flags based on a & b, discards result
Operand Types

Immediate

$0x400, -$533

**Register**: 16 general purpose

%rax, %rbx, %rcx, %rdx, %rsi, %rdi, %rbp, %rsp,
%r8, %r9, %r10, %r11, $r12, %r13, %r14, %r15

**Memory**: (%rsp)

---

**Most General Form:**

\[
D(Rb, Ri, S) \quad \text{Mem[Reg[Rb] + S*Reg[Ri] + D]}
\]

- **D**: Constant "displacement" value represented in 1, 2, or 4 bytes
- **Rb**: Base register: Any register
- **Ri**: Index register: Any except %esp (or %esp if 64-bit); %ebp unlikely
- **S**: Scale: 1, 2, 4, or 8 (*why these numbers?*

**Special Cases**: can use any combination of D, Rb, Ri and S

- \((Rb, Ri)\) \quad \text{Mem[Reg[Rb]+Reg[Ri]]} \quad (S=1, D=0)
- \(D(Rb, Ri)\) \quad \text{Mem[Reg[Rb]+Reg[Ri]+D]} \quad (S=1)
- \((Rb, Ri, S)\) \quad \text{Mem[Reg[Rb]+S*Reg[Ri]]} \quad (D=0)
Control Flow
Conditional jump instructions in X86 implement the following high-level constructs:

- if (condition) then {...} else {...}
- while (condition) {...}
- do {...} while (condition)
- for (initialization; condition; iterative) {...}

Unconditional jumps are used for high-level constructs such as:

- break
- continue

<table>
<thead>
<tr>
<th>jX</th>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jmp</td>
<td>?</td>
<td>Unconditional</td>
</tr>
<tr>
<td>je</td>
<td>ZF</td>
<td>Equal / Zero</td>
</tr>
<tr>
<td>jne</td>
<td>~ZF</td>
<td>Not Equal / Not Zero</td>
</tr>
<tr>
<td>js</td>
<td>SF</td>
<td>Negative</td>
</tr>
<tr>
<td>jns</td>
<td>~SF</td>
<td>Nonnegative</td>
</tr>
<tr>
<td>jg</td>
<td>~ (SF^OF) &amp; ~ZF</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>jge</td>
<td>~ (SF^OF)</td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td>jl</td>
<td>(SF^OF)</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td>jle</td>
<td>(SF^OF)</td>
<td>Less or Equal (Signed)</td>
</tr>
<tr>
<td>ja</td>
<td>~CF &amp; ~ZF</td>
<td>Above (unsigned)</td>
</tr>
<tr>
<td>jb</td>
<td>CF</td>
<td>Below (unsigned)</td>
</tr>
</tbody>
</table>

PC-relative Addressing
Jump instructions encode the offset from next instruction to destination PC, instead of the absolute address of the destination (makes it easier to relocate the code)
Turning C into Machine Code

C Code

```c
void sumstore(long x, long y, long *dest) {
    long t = x + y;
    *dest = t;
}
```

Generated x86 Assembly Code

```
sum:
    addq %rdi,%rsi
    movq %rsi,(%rdx)
    retq
```

Executable: `sum`

Resolve references between object files, libraries, (re)locate data

- X86 instructions can be in different order from C code
- Some C expressions require multiple X86 instructions
- Some X86 instructions can cover multiple C expressions
- Compiler optimization can do some surprising things!
- Local or temporary variables can be stored in registers or on the stack
Function Calling Conventions

- Arguments for functions are stored in registers, in the following order: arg1 – arg6: %rdi,%rsi,%rdx,%rcx,%r8,%r9
- Return value from function always in %rax
Tools

Tools can be used to examine bytes of object code (executable program) and reconstruct (reverse engineer) the assembly source.

**gdb** – disassembles an executable file into the associated assembly language representation, and provides tools for memory and register examination, single step execution, breakpoints, etc.

<table>
<thead>
<tr>
<th>Object</th>
<th>Disassembled by GDB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00400536:</td>
<td>0x000000000000400536 &lt;+0&gt;: add %rdi,%rsi</td>
</tr>
<tr>
<td>0x48</td>
<td>0x000000000000400539 &lt;+3&gt;: mov %rsi,(%rdx)</td>
</tr>
<tr>
<td>0x01</td>
<td>0x00000000000040053c &lt;+6&gt;: retq</td>
</tr>
<tr>
<td>0xfe</td>
<td></td>
</tr>
<tr>
<td>0x48</td>
<td></td>
</tr>
<tr>
<td>0x89</td>
<td></td>
</tr>
<tr>
<td>0x32</td>
<td></td>
</tr>
<tr>
<td>0xc3</td>
<td></td>
</tr>
</tbody>
</table>

**objdump**

Can also be used to disassemble and display information:

```bash
$ gdb sum
(gdb) disassemble sumstore
  (disassemble function)
(gdb) x/7b sum
  (examine the 13 bytes starting at sum)
```

$ objdump -t p

Prints out the program’s symbol table. The symbol table includes the names of all functions and global variables, the names of all the functions the called, and their addresses.
$ objdump -d p

Object Code
0x401040 <sum>:
  0x55
  0x89
  0xe5
  0x8b 0x4 0x0c 0x03 0x45 0x08
  0x89 0xec
  0x5d
Disassembled version
00401040 <_sum>:
  0:  55     push %ebp
  1:  89 e5   mov %esp,%ebp
  3:  8b 45 0c mov 0xc(%ebp),%eax
  6:  03 45 08 add 0x8(%ebp),%eax
  9:  89 ec   mov %ebp,%esp
 b:  5d pop   %ebp
 c:  c3 ret

strings
$ strings –t x p
Displays the printable strings in your program.