

CS240 Laboratory 6

Disassembly and Reverse Engineering

Memory Layout

Kernel above 0x7fffffff	Addr	Perm	Contents	Managed by	Initialized
Stack below 0x7fffffff grows down	2 ^{N-1}	RW	Procedure context	Compiler	Run-time
Heap above Data segment		RW	Dynamic data structures	Programmer, malloc/free, new/GC	Run-time
Data segment statics and literals		RW	Global variables/ static data structures	Compiler/ Assembler/Linker	Startup
		R	String literals	Compiler/ Assembler/Linker	Startup
Text segment starts at 0x400000	0	X	Instructions	Compiler/ Assembler/Linker	Startup

The diagram illustrates the memory layout with the following components:

- Stack:** Located at the top, growing downwards. It contains "Procedure context" and is managed by the "Compiler". It has "RW" permissions.
- Heap:** Located below the Stack. It contains "Dynamic data structures" and is managed by the "Programmer, malloc/free, new/GC". It has "RW" permissions.
- Statics:** Located below the Heap. It contains "Global variables/static data structures" and is managed by the "Compiler/Assembler/Linker". It has "RW" permissions.
- Literals:** Located below Statics. It contains "String literals" and is managed by the "Compiler/Assembler/Linker". It has "R" permissions.
- Text:** Located at the bottom. It contains "Instructions" and is managed by the "Compiler/Assembler/Linker". It has "X" permissions.

The vertical axis represents memory addresses, ranging from 0 at the bottom to 2^N-1 at the top.

Instructions

Moving Data

mov Src,Dest

Note: the size of the data being referenced is often specified with an additional character:

b (byte)
w (2 bytes)
l (4 bytes), or
q (8 bytes).

Arithmetic/Logical operations – 2 operands

add *Src,Dest*

sub *Src,Dest*

imul *Src,Dest*

shr *Src,Dest*

sar *Src, Dest*

shl *Src,Dest*

sal *Src, Dest*

shr *Src,Dest*

xor *Src,Dest*

and *Src,Dest*

or *Src,Dest*

<i>mul</i>	<i>Src,Dest</i>
<i>imul</i>	<i>Src,Dest</i>
<i>div</i>	<i>Src,Dest</i>
<i>idiv</i>	<i>Src,Dest</i>

Arithmetic/Logical operations – 1 operand

<i>inc</i>	<i>Dest</i>
<i>dec</i>	<i>Dest</i>
<i>neg</i>	<i>Dest</i>
<i>not</i>	<i>Dest</i>

Setting Condition Codes Explicitly – used for control flow

<i>cmp</i>	<i>Src2,Src1</i>	sets flags based on value of Src2 – Src1, discards result
<i>test</i>	<i>Src2,Src1</i>	sets flags based on a & b, discards result

Operand Types

Immediate \$0x400, \$-533

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

some have special purpose: **%rsp** is stack pointer, **%rax** always used to return value from functions

Memory -0x18(%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 %rsp 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

JX	Condition	Description
<code>jmp</code>	1	Unconditional
<code>je</code>	<code>ZF</code>	Equal / Zero
<code>jne</code>	$\sim ZF$	Not Equal / Not Zero
<code>ja</code>	<code>SF</code>	Negative
<code>jns</code>	$\sim SF$	Nonnegative
<code>jg</code>	$\sim (SF \wedge OF) \wedge \sim ZF$	Greater (Signed)
<code>jge</code>	$\sim (SF \wedge OF)$	Greater or Equal (Signed)
<code>jl</code>	$(SF \wedge OF)$	Less (Signed)
<code>jle</code>	$(SF \wedge OF) \mid ZF$	Less or Equal (Signed)
<code>ja</code>	$\sim CF \wedge \sim ZF$	Above (unsigned)
<code>jb</code>	<code>CF</code>	Below (unsigned)

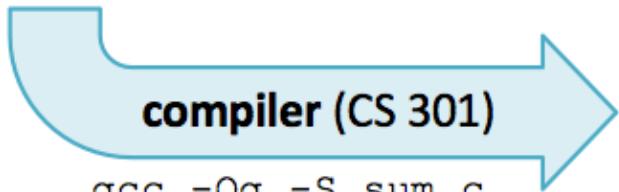
Jump instructions encode the offset from next instruction to destination PC, instead of the absolute address of the destination (makes it easier to relocate the code)

Turning C into Machine Code

C Code

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

sum.c



gcc -Og -S sum.c

Generated x86 Assembly Code

Human-readable language close to machine code.

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

sum.s

assembler

Object Code

```
01010101100010011110010110  
00101101000101000011000000  
00110100010100001000100010  
01111011000101110111000011
```

sum.o

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
- If there are more than 6 parameters for a function, the rest of the arguments are stored on the stack before the function is called
- Return value from function is always in %rax

The compiler will use only part of a register if the value stored there will fit in less than 64 bits (8 bytes). This is an optimization that makes instructions a bit shorter.

So, in the code, you may see register names of the following form, all of which refer to %rax:

%rax = 8 byte value

%eax = 4 byte value

%ax = 2 byte value

%al = 1 byte value

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.

Object	Disassembled by GDB
0x00400536:	0x0000000000400536 <+0>: add %rdi,%rsi
0x48	0x0000000000400539 <+3>: mov %rsi,(%rdx)
0x01	0x000000000040053c <+6>: retq
0xfe	
0x48	
0x89	
0x32	
0xc3	

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

objdump

can also be used to disassemble and display information

\$ *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  
0x45  
0x0c  
0x03  
0x45  
0x08  
0x89  
0xec  
0x5d  
0xc3
```

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.