Lecture 3: Machine Code, Assembly, Disassembly, and Decompilation

Sanchuan Chen

schen@auburn.edu

8/23/2023

Hardware Organization

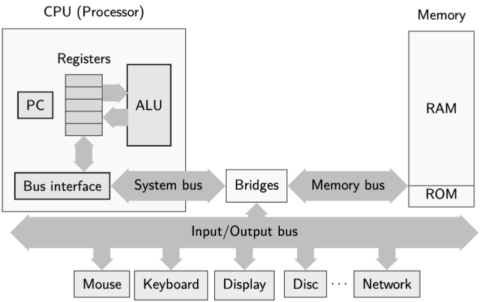


Figure: Hardware organization of a typical system

cpuinfo Example

```
schen@pc:~/comp6700/lec02$ cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 158
model name      : Intel(R) Core(TM) i7-7820HQ CPU @ 2.90GHz
stepping        : 9
microcode       : 0x8e
cpu MHz         : 2904.004
cache size      : 8192 KB
physical id     : 0
siblings        : 1
core id         : 0
cpu cores       : 1
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 22
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge ...
bugs            : cpu_meltdown spectre_v1 spectre_v2 spec_store_bypass ...
bogomips        : 5808.00
clflush size    : 64
...
```

Machine Code

Why Machine Code? (Offense and Defense)

1. **Code injection attack** requires injecting the shellcode. Without understanding the machine code, one cannot construct proper shellcode.
2. **Code reuse attack** does not directly inject machine code, but it injects addresses that point to existing code (e.g., ROP gadgets). It is important to understand where to find the proper machine code in the memory.

Simple Shellcode Test (shellcode.c)

```
/* call_shellcode.c */
/*A program that creates a file containing code for launching shell*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
const char code[] =
  "\x31\xc0" /* Line 1: xorl %eax,%eax */
  "\x50" /* Line 2: pushl %eax */
  "\x68""//sh" /* Line 3: pushl $0x68732f2f */
  "\x68""/bin" /* Line 4: pushl $0x6e69622f */
  "\x89\xe3" /* Line 5: movl %esp,%ebx */
  "\x50" /* Line 6: pushl %eax */
  "\x53" /* Line 7: pushl %ebx */
  "\x89\xe1" /* Line 8: movl %esp,%ecx */
  "\x99" /* Line 9: cdq */
  "\xb0\x0b" /* Line 10: movb $0x0b,%al */
  "\xcd\x80" /* Line 11: int $0x80 */
;

int main(int argc, char **argv)
{
  char buf[sizeof(code)];
  strcpy(buf, code);
  ((void(*)( ))buf)( );
}
```

X86 Instruction Encoding

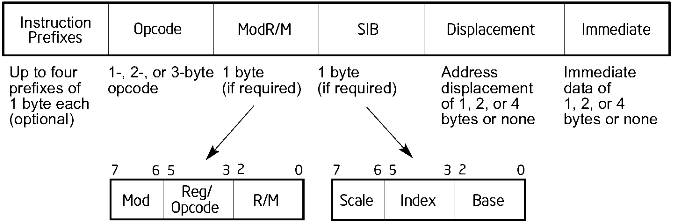


Figure: X86 Instruction Encoding

X86 Instruction Encoding

This course focuses on x86 architecture. x86 instructions can be anywhere between 1 and 15 bytes long. The length is defined separately for each instruction, depending on the available modes of operation of the instruction, the number of required operands and more.

We will never directly write machine code (though we may directly craft exploit sometimes), and instead we will write code in programming languages such as C, or **assembly**. However, often times, there is a need to decode machine code, particularly in **malware analysis**, **exploit reverse engineering**. More information about x86 machine code can be found in Intel Manuals or at `http://ref.x86asm.net/index.html`

How to Represent Data

C Data Type	Typical 32-bit	X86	X86-64
char	1	1	1
short	2	2	2
int	4	4	4
long	4	4	8
long long	8	8	8
float	4	4	4
double	8	8	8
long double	8	12	16
pointer	4	4	8

Note: On the x86 architecture, most C compilers implement `long double` as the 80-bit extended precision type supported by x86 hardware. With the GNU C Compiler, `long double` is 80-bit extended precision on x86 processors regardless of the physical storage used for the type (which can be either 96 or 128 bits).

sizes.c (from CSAPP)

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    printf("sizeof(unsigned) = %ld\n", sizeof(unsigned));
    printf("sizeof(size_t) = %ld\n", sizeof(size_t));
    printf("sizeof(ssize_t) = %ld\n", sizeof(ssize_t));
    printf("sizeof(int) = %ld\n", sizeof(int));
    printf("sizeof(long int) = %ld\n", sizeof(long int));
    printf("sizeof(char) = %ld\n", sizeof(char));
    printf("sizeof(short) = %ld\n", sizeof(short));
    printf("sizeof(float) = %ld\n", sizeof(float));
    printf("sizeof(double) = %ld\n", sizeof(double));
    printf("sizeof(long double) = %ld\n", sizeof(long double));
    printf("sizeof(char *) = %ld\n", sizeof(char *));
    exit(0);
}
```

Big Endian vs. Little Endian

In computer systems, **endianness** concerns how to order or store the sequence of bytes of digital data in memory. There are two endianness systems:

- Big-endian (BE): A BE system stores the most significant byte of a word at the smallest memory address, and the least significant byte at the largest.
- Little-endian (LE): A little-endian system, in contrast, stores the least-significant byte at the smallest address.

Solely LE systems: X86, X86-64; and solely BE systems: SPARC, OpenRISC. ARM and MIPS can be configured to support either.

Big Endian vs. Little Endian

Addr	0x100	0x101	0x102	0x103
LE	0x78	0x56	0x34	0x12
BE	0x12	0x34	0x56	0x78

Table: How to Store 0x12345678

endian.c

```
/*
* https://stackoverflow.com/questions/12791864/c-program-to-check-little-vs-big-endian
*/
#include <stdio.h>
#include <stdint.h>

int is_big_endian(void)
{
    union {
        uint32_t i;
        char c[4];
    } e = { 0x01000000 };

    return e.c[0];
}

int main(void)
{
    printf("System is %s-endian.\n",
        is_big_endian() ? "big" : "little");

    return 0;
}
```

hello32.s

```
# ----------------------------------------------------------------------------------------
#
#     gcc -c -m32 hello32.s && ld -m elf_i386 hello32.o -o a.out32 && ./a.out32
#
# ----------------------------------------------------------------------------------------

        .global _start

        .text
_start:
        # write(1, message, 13)
        mov     $4, %eax                # system call 4 is write
        mov     $1, %ebx                # file handle 1 is stdout
        mov     $message, %ecx          # address of string to output
        mov     $14, %edx               # number of bytes
        int     $0x80                   # invoke operating system to do the write

        # exit(0)
        mov     $1, %eax                # system call 60 is exit
        xor     %ebx, %ebx              # we want return code 0
        int     $0x80                   # invoke operating system to exit

message:
        .ascii  "Hello, world!\n"
```

hello64.s

```
# ---------------------------------------------------------------------------------------
#
#     gcc -c hello64.s && ld hello64.o -o a.out64 && ./a.out64
#
# ---------------------------------------------------------------------------------------

        .global _start

        .text
_start:
        # write(1, message, 13)
        mov     $1, %rax                # system call 1 is write
        mov     $1, %rdi                # file handle 1 is stdout
        mov     $message, %rsi          # address of string to output
        mov     $14, %rdx               # number of bytes
        syscall                         # invoke operating system to do the write

        # exit(0)
        mov     $60, %rax               # system call 60 is exit
        xor     %rdi, %rdi              # we want return code 0
        syscall                         # invoke operating system to exit
message:
        .ascii  "Hello, world!\n"
```

Basic Instructions, Constants, Registers

Instructions: Three categories of instructions in general: (1) data movement, (2) arithmetic/logic, and (3) control-flow.
Constant: MUST be preceeded with "\$". "\$12" means decimal 12; "\$0xF0" is hex.
Registers: In x86, there are eight 32-bit general purpose registers (i.e., EAX, EBX, ECX, EDX, ESI, EDI, ESP, and EBP). MUST be preceded with "%". "%eax" means register eax.

Basic Instructions, Constants, Registers

Memory access:

1. **(Register as pointer)**: "(%esp)". Same as C "*esp".
2. **(Register + offset as pointer)**: "4(%esp)". Same as C "*(esp+4)".
3. **(Register + another register * scale)**: "(%eax, %ebx, 4)". Same as C "*(eax+ebx*4)".

Basic Instructions, Constants, Registers

Mnemonic	Purpose	Examples
`mov src,dest`	Move data between registers, load immediate data into registers, move data between registers and memory.	mov $4,%eax
`push src`	Insert a value onto the stack. Useful for passing arguments, saving registers, etc.	push %ebp
`pop dest`	Remove topmost value from the stack. Equivalent to "mov (%esp),dest; add $4,%esp"	pop %ebp
`call func`	Push the address of the next instruction and start executing func.	call print_int
`ret`	Pop the return program counter, and jump there. Ends a subroutine.	ret
`add src,dest`	dest=dest+src	add %ebx,%eax
`mul src`	Multiply eax and src as unsigned integers, and put the result in eax. High 32 bits of product go into eax.	mul %ebx #Multiply eax by ebx
`jmp label`	Goto the instruction label; Skips anything else in the way.	jmp label
`cmp a,b`	Compare two values. Sets flags that are used by the conditional jumps.	cmp $10,%eax
`jcc label`	Goto label if the condition code is satisified before jumping. Various conditions available are: jle (<=), je (==), jge (>=), jg (>), jne (!=), and many others.	jle loop_start

How to Calculate Memory Addresses

In x86 assembly, memory address is encoded in **D(Rb,Ri,S)**, where

- D: Constant “displacement”
- Rb: Base register: Any of the 8 general registers
- Ri: Index register: Any, except for %esp
- S: Scale: 1, 2, 4, or 8

and its memory address is calculated to:

$$\boxed{Mem[Reg[Rb] + S * Reg[Ri] + D]}$$

(Rb,Ri)	Mem[Reg[Rb]+Reg[Ri]]
D(Rb,Ri)	Mem[Reg[Rb]+Reg[Ri]+D]
(Rb,Ri,S)	Mem[Reg[Rb]+S*Reg[Ri]]

maddr.s (how to calculate memory address)

```
# ----------------------------------------------------------------------------------------
#
#     gcc -c -m32 maddr.s && ld -m elf_i386 maddr.o -o maddr.out
#
# ----------------------------------------------------------------------------------------

        .global _start

        .text
_start:

        mov $_start, %ebx
        mov (%ebx), %eax               /* Load 4 bytes from the memory address in EBX into EAX. */
        mov -4(%ebx), %eax             /* Move 4 bytes at memory address EBX + (-4) into EAX. */
        mov $1, %esi
        mov (%ebx,%esi,4), %edx        /* Move the 4 bytes of data at address EBX+4*ESI into EDX. */


        # exit(0)
        mov     $1, %eax               # system call 60 is exit
        xor     %ebx, %ebx             # we want return code 0
        int     $0x80                  # invoke operating system to exit
```

How to Make System Calls (/arch/x86/entry/syscalls/syscall_32.tbl)

```
# 32-bit system call numbers and entry vectors
#
# The format is:
# <number> <abi> <name> <entry point> <compat entry point>
#
# The __ia32_sys and __ia32_compat_sys stubs are created on-the-fly for
# sys_*() system calls and compat_sys_*() compat system calls if
# IA32_EMULATION is defined, and expect struct pt_regs *regs as their only
# parameter.
#
# The abi is always "i386" for this file.
#
0 i386 restart_syscall sys_restart_syscall
1 i386 exit sys_exit
2 i386 fork sys_fork
3 i386 read sys_read
4 i386 write sys_write
5 i386 open sys_open compat_sys_open
6 i386 close sys_close
7 i386 waitpid sys_waitpid
8 i386 creat sys_creat
9 i386 link sys_link
10 i386 unlink sys_unlink
11 i386 execve sys_execve compat_sys_execve
12 i386 chdir sys_chdir
13 i386 time sys_time32
14 i386 mknod sys_mknod
15 i386 chmod sys_chmod
16 i386 lchown sys_lchown16
17 i386 break
18 i386 oldstat sys_stat
```

How to Make System Calls (/arch/x86/entry/syscalls/syscall_32.tbl, continued)

```
19 i386 lseek sys_lseek compat_sys_lseek
20 i386 getpid sys_getpid
21 i386 mount sys_mount
22 i386 umount sys_oldumount
23 i386 setuid sys_setuid16
24 i386 getuid sys_getuid16
25 i386 stime sys_stime32
26 i386 ptrace sys_ptrace compat_sys_ptrace
27 i386 alarm sys_alarm
28 i386 oldfstat sys_fstat
29 i386 pause sys_pause
30 i386 utime sys_utime32
31 i386 stty
32 i386 gtty
33 i386 access sys_access
34 i386 nice sys_nice
35 i386 ftime
36 i386 sync sys_sync
37 i386 kill sys_kill
38 i386 rename sys_rename
39 i386 mkdir sys_mkdir
40 i386 rmdir sys_rmdir
41 i386 dup sys_dup
...
```

How to Make System Calls (/arch/x86/entry/syscalls/syscall_64.tbl)

```
# 64-bit system call numbers and entry vectors
#
# The format is:
# <number> <abi> <name> <entry point>
#
# The abi is "common", "64" or "x32" for this file.
#
0 common read sys_read
1 common write sys_write
2 common open sys_open
3 common close sys_close
4 common stat sys_newstat
5 common fstat sys_newfstat
6 common lstat sys_newlstat
7 common poll sys_poll
8 common lseek sys_lseek
9 common mmap sys_mmap
10 common mprotect sys_mprotect
11 common munmap sys_munmap
12 common brk sys_brk
...
16 64 ioctl sys_ioctl
...
19 64 readv sys_readv
20 64 writev sys_writev
```

How to Make System Calls (/arch/x86/entry/syscalls/syscall_64.tbl, continued)

```
21 common access sys_access
22 common pipe sys_pipe
...
33 common dup2 sys_dup2
34 common pause sys_pause
35 common nanosleep sys_nanosleep
36 common getitimer sys_getitimer
37 common alarm sys_alarm
38 common setitimer sys_setitimer
39 common getpid sys_getpid
40 common sendfile sys_sendfile64
41 common socket sys_socket
42 common connect sys_connect
43 common accept sys_accept
44 common sendto sys_sendto
45 64 recvfrom sys_recvfrom
46 64 sendmsg sys_sendmsg
47 64 recvmsg sys_recvmsg
48 common shutdown sys_shutdown
49 common bind sys_bind
50 common listen sys_listen
51 common getsockname sys_getsockname
52 common getpeername sys_getpeername
53 common socketpair sys_socketpair
54 64 setsockopt sys_setsockopt
55 64 getsockopt sys_getsockopt
56 common clone sys_clone/ptregs
57 common fork sys_fork/ptregs
58 common vfork sys_vfork/ptregs
59 64 execve sys_execve/ptregs
60 common exit sys_exit
...
```

Table: The System Call Table in x86 (32-bit)

NR	Syscall name	%eax	arg0 (%ebx)	arg1 (%ecx)	arg2 (%edx)	arg3 (%esi)
0	restart_syscall	0x00				
1	exit	0x01	int error_code			
2	fork	0x02				
3	read	0x03	unsigned int fd	char *buf	size_t count	
4	write	0x04	unsigned int fd	const char *buf	size_t count	
5	open	0x05	const char *filename	int flags	umode_t mode	
6	close	0x06	unsigned int fd			
7	waitpid	0x07	pid_t pid	int *stat_addr	int options	
8	creat	0x08	const char *pathname	umode_t mode		
9	link	0x09	const char *oldname	const char *newname		
10	unlink	0x0a	const char *pathname			
11	execve	0x0b	const char *filename	const char *const *argv	const char *const *envp	
102	socketcall	0x66	int call	unsigned long *args		
187	sendfile	0xbb	int out_fd	int in_fd	off_t *offset	size_t count

https://chromium.googlesource.com/chromiumos/docs/+/master/constants/syscalls.md#x86-32_bit

How to Make Library Calls
helloha32.s (how to call libc code)

```
# ------------------------------------------------------------------------------------------
#
#  gcc -c -m32 helloha32.s
#  ld -m elf_i386 -dynamic-linker /lib/ld-linux.so.2 -lc  helloha32.o -o a.outlibc && ./a.outlibc
#
# ------------------------------------------------------------------------------------------

        .text
        .globl main
main:                                   # This is called by C library's startup code
        push    %ebp
        mov     %esp, %ebp
        push    $message
        call    puts                    # puts(message)

        # exit(0)
        mov     $1, %eax                # system call 60 is exit
        xor     %ebx, %ebx              # we want return code 0
        int     $0x80                   # invoke operating system to exit
message:
        .asciz "Helloha!"                # asciz puts a 0 byte at the end
```

About the Calling Convention

1. `__cdecl`: which stands for C declaration. Function arguments are pushed on the stack in the right-to-left order, i.e., the last argument is pushed first. Integer values and memory addresses are returned in the EAX. Caller cleans up stack

cdecl.c

```
#include <stdio.h>

int callee(int a, int b, int c)
{
    return a + b + c;
}

int caller(void)
{
    return callee(1, 2, 3) + 5;
}

int main()
{
    printf("The return value from the caller %d\n",caller());
    return 0;
}
```

About the Calling Convention

cdecl.asm

```
...
0000119d <callee>:
    119d: 55                   push   %ebp
    119e: 89 e5                mov    %esp,%ebp
    11a0: e8 75 00 00 00       call   121a <__x86.get_pc_thunk.ax>
    11a5: 05 33 2e 00 00       add    $0x2e33,%eax
    11aa: 8b 55 08             mov    0x8(%ebp),%edx
    11ad: 8b 45 0c             mov    0xc(%ebp),%eax
    11b0: 01 c2                add    %eax,%edx
    11b2: 8b 45 10             mov    0x10(%ebp),%eax
    11b5: 01 d0                add    %edx,%eax
    11b7: 5d                   pop    %ebp
    11b8: c3                   ret

000011b9 <caller>:
    11b9: 55                   push   %ebp
    11ba: 89 e5                mov    %esp,%ebp
    11bc: e8 59 00 00 00       call   121a <__x86.get_pc_thunk.ax>
    11c1: 05 17 2e 00 00       add    $0x2e17,%eax
    11c6: 6a 03                push   $0x3
    11c8: 6a 02                push   $0x2
    11ca: 6a 01                push   $0x1
    11cc: e8 cc ff ff ff       call   119d <callee>
    11d1: 83 c4 0c             add    $0xc,%esp
    11d4: 83 c0 05             add    $0x5,%eax
    11d7: c9                   leave
    11d8: c3                   ret
...
```

About the Calling Convention

2. `__fastcall`: Conventions entitled fastcall have not been standardized, and have been implemented differently, depending on the compiler vendor. It is fast in that the calling convention specifies that arguments to functions are to be passed in registers, when possible. Callee cleans up the stack.

fastcall.c

```
#include <stdio.h>

__attribute__((fastcall)) int fastcallee(int a, int b, int c)
{
    return a + b + c;
}

int caller(void)
{
    return fastcallee(1, 2, 3) + 5;
}

int main()
{
    printf("The return value from the caller %d\n",caller());
    return 0;
}
```

About the Calling Convention

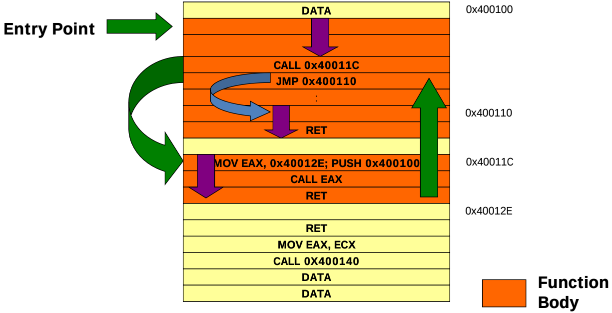
fastcall.asm

```
...
0000119d <fastcallee>:
    119d: 55                   push   %ebp
    119e: 89 e5                mov    %esp,%ebp
    11a0: 83 ec 08             sub    $0x8,%esp
    11a3: e8 80 00 00 00       call   1228 <__x86.get_pc_thunk.ax>
    11a8: 05 30 2e 00 00       add    $0x2e30,%eax
    11ad: 89 4d fc             mov    %ecx,-0x4(%ebp)
    11b0: 89 55 f8             mov    %edx,-0x8(%ebp)
    11b3: 8b 55 fc             mov    -0x4(%ebp),%edx
    11b6: 8b 45 f8             mov    -0x8(%ebp),%eax
    11b9: 01 c2                add    %eax,%edx
    11bb: 8b 45 08             mov    0x8(%ebp),%eax
    11be: 01 d0                add    %edx,%eax
    11c0: c9                   leave
    11c1: c2 04 00             ret    $0x4

000011c4 <caller>:
    11c4: 55                   push   %ebp
    11c5: 89 e5                mov    %esp,%ebp
    11c7: e8 5c 00 00 00       call   1228 <__x86.get_pc_thunk.ax>
    11cc: 05 0c 2e 00 00       add    $0x2e0c,%eax
    11d1: 6a 03                push   $0x3
    11d3: ba 02 00 00 00       mov    $0x2,%edx
    11d8: b9 01 00 00 00       mov    $0x1,%ecx
    11dd: e8 bb ff ff ff       call   119d <fastcallee>
    11e2: 83 c0 05             add    $0x5,%eax
    11e5: c9                   leave
    11e6: c3                   ret
...
```

Linear Sweep

Linear Sweep

1. Start with program entry point, proceed to disassemble instructions sequentially
2. Key assumption: all instructions appear one after the next, without any gaps



Linear Sweep

objdump -d a.out32

```
a.out32:     file format elf32-i386


Disassembly of section .text:

08049000 <_start>:
 8049000:       b8 04 00 00 00          mov    $0x4,%eax
 8049005:       bb 01 00 00 00          mov    $0x1,%ebx
 804900a:       b9 1f 90 04 08          mov    $0x804901f,%ecx
 804900f:       ba 0e 00 00 00          mov    $0xe,%edx
 8049014:       cd 80                   int    $0x80
 8049016:       b8 01 00 00 00          mov    $0x1,%eax
 804901b:       31 db                   xor    %ebx,%ebx
 804901d:       cd 80                   int    $0x80

0804901f <message>:
 804901f:       48                      dec    %eax
 8049020:       65 6c                   gs insb (%dx),%es:(%edi)
 8049022:       6c                      insb   (%dx),%es:(%edi)
 8049023:       6f                      outsl  %ds:(%esi),(%dx)
 8049024:       2c 20                   sub    $0x20,%al
 8049026:       77 6f                   ja     8049097 <message+0x78>
 8049028:       72 6c                   jb     8049096 <message+0x77>
 804902a:       64 21 0a                and    %ecx,%fs:(%edx)
```

Linear Sweep

objdump -d a.outlibc

```
a.outlibc:     file format elf32-i386

...

Disassembly of section .text:

08049020 <main>:
 8049020:       55                      push   %ebp
 8049021:       89 e5                   mov    %esp,%ebp
 8049023:       68 36 90 04 08         push   $0x8049036
 8049028:       e8 e3 ff ff ff          call   8049010 <puts@plt>
 804902d:       b8 01 00 00 00          mov    $0x1,%eax
 8049032:       31 db                   xor    %ebx,%ebx
 8049034:       cd 80                   int    $0x80

08049036 <message>:
 8049036:       48                      dec    %eax
 8049037:       65 6c                   gs insb (%dx),%es:(%edi)
 8049039:       6c                      insb   (%dx),%es:(%edi)
 804903a:       6f                      outsl  %ds:(%esi),(%dx)
 804903b:       68                      .byte 0x68
 804903c:       61                      popa
 804903d:       21 00                   and    %eax,(%eax)
```

Recursive Traversal

Recursive Traversal

1. After a control-flow transfer instruction (CTI), proceed to disassemble target address
2. For conditional CTI and non-CTI, proceed to disassemble next instruction
3. Key problems
   - Code reached only through indirect CTIs
   - Functions that do not return in the usual way

See https://reverseengineering.stackexchange.com/questions/2347/what-is-the-algorithm-used-in-recursive-traversal-disassembly

Recursive Traversal

1. Udis86 is an easy-to-use, minimalistic disassembler library (libudis86) for the x86 class of instruction set architectures https://github.com/vmt/udis86. Manual https://www.cs.dartmouth.edu/~sergey/io/cs258/2009/udis86/udis86.pdf.
2. Capstone is a lightweight multi-platform, multi-architecture disassembly framework. http://www.capstone-engine.org/

Decompilation

A decompiler is a computer program that takes an **executable file** as input, and attempts to create a high level **source file** which can be recompiled successfully.

Decompilers are usually unable to perfectly reconstruct the original source code, due to the loss of rich-useful information (e.g., symbols, types) during compilation.

Decompilers still remain an important tool in the **reverse engineering** or even **patching** of binary code.

Ghidra

https://github.com/NationalSecurityAgency/ghidra

"
Ghidra is a software reverse engineering (SRE) framework created and maintained by the National Security Agency Research Directorate. This framework includes a suite of full-featured, high-end software analysis tools that enable users to analyze compiled code on a variety of platforms including Windows, macOS, and Linux. Capabilities include disassembly, assembly, decompilation, graphing, and scripting, along with hundreds of other features. Ghidra supports a wide variety of processor instruction sets and executable formats and can be run in both user-interactive and automated modes. Users may also develop their own Ghidra plug-in components and/or scripts using Java or Python.

In support of NSA's Cybersecurity mission, Ghidra was built to solve scaling and teaming problems on complex SRE efforts, and to provide a customizable and extensible SRE research platform. NSA has applied Ghidra SRE capabilities to a variety of problems that involve analyzing malicious code and generating deep insights for SRE analysts who seek a better understanding of potential vulnerabilities in networks and systems.

To start developing extensions and scripts, try out the GhidraDev plugin for Eclipse, which is part of the distribution package. The full release build can be downloaded from our project homepage.

This repository contains the source for the core framework, features, and extensions. If you would like to contribute, please take a look at our contributor guide to see how you can participate in this open source project.

If you are a U.S. citizen interested in projects like this, to develop Ghidra, and other cybersecurity tools, for NSA to help protect our nation and its allies, consider applying for a career with us.
"

Simple RE: pwdre0.c

```
/*
 * COMP 6700
 *
 * Simple RE Demo of how to break PWD
 *
 * Directly using strings
 *
 */

#include<stdio.h>
#include<string.h>

int main(int argc, char **argv){

    if(argc==1)
    {
        printf("Please provide the password\n");
        return 0;
    }

    if(argc>2)
    {
        printf("Please provide just the password, no other arguments");
        return 0;
    }

    if (!strcmp("comp6700",argv[1])){
        printf("Congratulations! You win\n");
    }
    else
    {
        printf("You loose. Please try again\n");
    }
}
```

objdump -d pwdre0

```
...
080491d6 <main>:
 80491d6:   f3 0f 1e fb             endbr32
 80491da:   8d 4c 24 04             lea    0x4(%esp),%ecx
 80491de:   83 e4 f0                and    $0xfffffff0,%esp
 80491e1:   ff 71 fc                pushl  -0x4(%ecx)
 80491e4:   55                      push   %ebp
 80491e5:   89 e5                   mov    %esp,%ebp
 80491e7:   51                      push   %ecx
 80491e8:   83 ec 04                sub    $0x4,%esp
 80491eb:   89 c8                   mov    %ecx,%eax
 80491ed:   83 38 01                cmpl   $0x1,(%eax)
 80491f0:   75 17                   jne    8049209 <main+0x33>
 80491f2:   83 ec 0c                sub    $0xc,%esp
 80491f5:   68 08 a0 04 08          push   $0x804a008
 80491fa:   e8 a1 fe ff ff          call   80490a0 <puts@plt>
 80491ff:   83 c4 10                add    $0x10,%esp
 8049202:   b8 00 00 00 00          mov    $0x0,%eax
 8049207:   eb 60                   jmp    8049269 <main+0x93>
 8049209:   83 38 02                cmpl   $0x2,(%eax)
...
```

Decompiled pwdre0.c w/ Symbol using Ghidra

```
int main(int argc,char **argv)
{
  int iVar1;

  if (argc == 1) {
    puts("Please provide the password");
  }
  else {
    if (argc < 3) {
      iVar1 = strcmp("comp6700",argv[1]);
      if (iVar1 == 0) {
        puts("Congratulations! You win");
      }
      else {
        puts("You loose. Please try again");
      }
    }
    else {
      printf("Please provide just the password, no other arguments");
    }
  }
  return 0;
}
```

Decompiled pwdre0.c w/o Symbol using Ghidra

```
undefined4 FUN_080491d6(int param_1,int param_2)
{
  int iVar1;

  if (param_1 == 1) {
    puts("Please provide the password");
  }
  else {
    if (param_1 < 3) {
      iVar1 = strcmp("comp6700",*(char **)(param_2 + 4));
      if (iVar1 == 0) {
        puts("Congratulations! You win");
      }
      else {
        puts("You loose. Please try again");
      }
    }
    else {
      printf("Please provide just the password, no other arguments");
    }
  }
  return 0;
}
```

Simple RE: pwdre1.c

```
/*
 * COMP6700
 * Simple RE Demo of how to break PWD
 * Manual inspection, or using symbolic execution (angr)
 */
#include<stdio.h>
#include <string.h>

int main(int argc, char **argv){
    if(argc==1){
        printf("Please provide the password\n");
        return 0;
    }
    if(argc>2){
        printf("Please provide just the password, no other arguments");
        return 0;
    }
    char buf[8];
    strncpy(buf,argv[1],8);
    if (buf[0] == 'C'){
        if (buf[1] == 'O'){
            if (buf[2] == 'M'){
                if (buf[3] == 'P'){
                    if (buf[4] == '6'){
                        if (buf[5] == '7'){
                            if (buf[6] == '0'){
                                if (buf[6] == '0'){
                                    printf("Congratulations! You win\n");
                                    return 1;
    }}}}}}}}
    printf("You loose. Please try again\n");
    return 0;
}
```

objdump -d pwdre1

```
000011cd <main>:
...
    1252: e8 29 fe ff ff        call   1080 <strncpy@plt>
    1257: 83 c4 10              add    $0x10,%esp
    125a: 0f b6 45 ec           movzbl -0x14(%ebp),%eax
    125e: 3c 43                 cmp    $0x43,%al
    1260: 75 51                 jne    12b3 <main+0xe6>
    1262: 0f b6 45 ed           movzbl -0x13(%ebp),%eax
    1266: 3c 4f                 cmp    $0x4f,%al
    1268: 75 49                 jne    12b3 <main+0xe6>
    126a: 0f b6 45 ee           movzbl -0x12(%ebp),%eax
    126e: 3c 4d                 cmp    $0x4d,%al
    1270: 75 41                 jne    12b3 <main+0xe6>
    1272: 0f b6 45 ef           movzbl -0x11(%ebp),%eax
    1276: 3c 50                 cmp    $0x50,%al
    1278: 75 39                 jne    12b3 <main+0xe6>
    127a: 0f b6 45 f0           movzbl -0x10(%ebp),%eax
    127e: 3c 36                 cmp    $0x36,%al
    1280: 75 31                 jne    12b3 <main+0xe6>
...
```

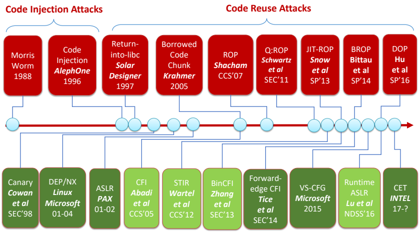
angr-decomp.py

```
import angr

p = angr.Project("./pwdre0", auto_load_libs=False);
p.analyses.CFG(normalize=True);
print(p.analyses.Decompiler(p.kb.functions['main']).codegen.text)
```

python angr-decomp.py

```
...
int main(){
    char *v0;  // [bp-0x20]
    char *v1;  // [bp-0x1c]
    unsigned int v2;  // [bp-0x10]
    unsigned int v3;  // [bp-0x4]
    unsigned int v4;  // [bp+0x0]
    unsigned int v5;  // [bp+0x4]
    struct_0 *v6;  // [bp+0x8]

    v3 = v4;
    v2 = stack_base + 4;
    if (v5 == 1){
        v0 = &g_402008;
        puts(v0);
        return;
    }
    if (v5 > 2){
        v0 = &g_402024;
        printf(v0);
        return;
    }
    v1 = v6->field_4;
    v0 = &g_402059;
    if (!strcmp(v0, v1)){
        v0 = &g_402062;
        puts(v0);
        return;
    }
    v0 = &g_40207b;
    puts(v0);
    return;
}
```

Thank You



[1]

Q&A

schen@auburn.edu
schuan.github.io

[1] Instructor appreciates the help from Prof. Zhiqiang Lin.