ENPM691 Homework Assignment 2

Hacking of C programs and Unix Binaries-Fall 2024 lcollier

Dipam Bhattacharyya, MEng Cybersecurity dipamb2@umd.edu

***Abstract*—The assignment aims to determine the logic behind different assembly instructions taken by the compiler for arithmetic operations. Depending on the constant values in the multiplication code, disassembled code varies.**

***Index Terms*—homework, hacking, C, coding**

I. Introduction

There are four steps in the C compilation process, preprocessing, compiling, assembling, and linking. Each of these is handled by separate units, and in C, different files are generated in each step. The C compiler, say GCC, calls these individual owners for the user's convenience. The assignment scope is to look at the second phase where the compiler generates a ".s" file containing assembly code from the ".i" file.

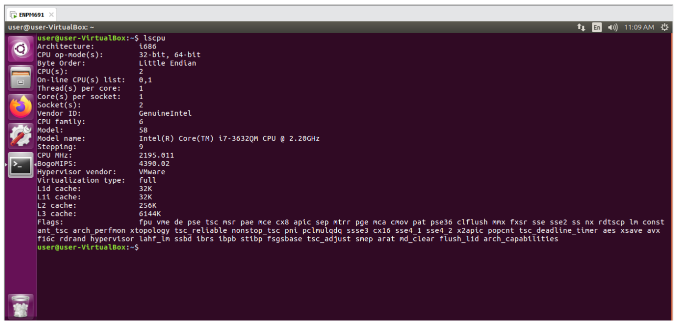
Although many debuggers like WinDbg, pwndbg exist, GNU Debugger (GDB) is basic and more popular, hence we will be using this in our experiment. The debugger helps break the executable into assembly instructions to understand memory assignments, add breakpoints, and inspect code.

The assignment's goal is to understand the assembly code generated by differing constant values and why certain operations are expensive. Along with that, explore the debugger and compiler options for processing the executable.

II. Methodology

A. Host environment and architecture

The experiment has been done on a Ubuntu v16.04 VM with Windows x64 being the host PC. All exploit protection settings in Windows are default, including ASLR(default ON). To check the VM architecture, "lscpu" terminal command was given:

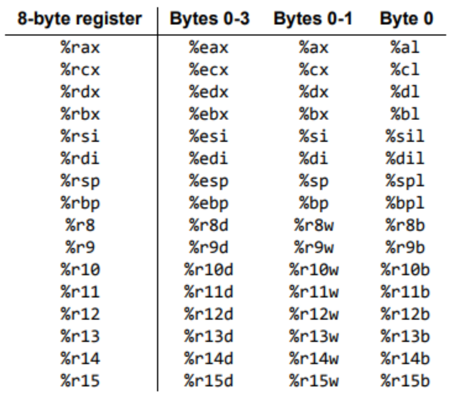


```
user@user-VirtualBox:~$ lscpu
Architecture:          i686
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                2
On-line CPU(s) list:   0,1
Thread(s) per core:    1
Core(s) per socket:    1
Socket(s):             2
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 58
Model name:            Intel(R) Core(TM) i7-3632QM CPU @ 2.20GHz
Stepping:              9
CPU MHz:               2195.011
BogoMIPS:              4390.02
Hypervisor vendor:     VMware
Virtualization type:   full
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              6144K
Flags:                 fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2 ss nx rdtscp lm const
ant_tsc arch_perfmon xtopology tsc_reliable nonstop_tsc pni pclmulqdq ssse3 cx16 sse4_1 sse4_2 x2apic popcnt tsc_deadline_timer aes xsave avx
f16c rdrand hypervisor lahf_lm ssbd ibrs ibpb stibp fsgsbase tsc_adjust smep arat md_clear flush_l1d arch_capabilities
user@user-VirtualBox:~$
```

Fig. 1. Ubuntu CPU information

B. Memory addressing with Registers and Stack

Since our underlying architecture is x64, there are sixteen 64-bit registers. Of these, depending on the code and compiler, each register can be logically divided into 8-bit, 16-bit, or 32-bit independent memory. The naming syntax of these registers can be "AT&T" or "Intel" format, but works the same beneath. The assembly code we are using will use registers of the earlier one due to the presence of "%".



8-byte register	Bytes 0-3	Bytes 0-1	Byte 0
%rax	%eax	%ax	%al
%rcx	%ecx	%cx	%cl
%rdx	%edx	%dx	%dl
%rbx	%ebx	%bx	%bl
%rsi	%esi	%si	%sil
%rdi	%edi	%di	%dil
%rsp	%esp	%sp	%spl
%rbp	%ebp	%bp	%bpl
%r8	%r8d	%r8w	%r8b
%r9	%r9d	%r9w	%r9b
%r10	%r10d	%r10w	%r10b
%r11	%r11d	%r11w	%r11b
%r12	%r12d	%r12w	%r12b
%r13	%r13d	%r13w	%r13b
%r14	%r14d	%r14w	%r14b
%r15	%r15d	%r15w	%r15b

Fig. 2. Registers according to memory capacity

All the arithmetic operations and results generated are achieved using the registers, although function code is present in the stack. The stack grows downwards(towards lower memory addresses), and the top of the stack contains "%ebp"; other registers like "%esp" points at the top, and "%eax" stores the return value of the function.

C. Arithmetic operations in assembly

The assembly syntax of arithmetic operations involves keywords followed by register, memory, or constant values as operands. For example, to move value between locations, the "MOV S,D" instruction copies the content of source(S) to destination(D). Similarly, "ADD", "SUB", "SHL", "SHR" etc. exist, which are adding, subtracting, or bitwise shifting(left/right). These instructions are less expensive and hence compiler often tries to arrive at the result using such.

Of typical interest is the multiplication operation because of its complexity. There are two types, signed and unsigned multiply. Signed uses the "MUL" keyword and retains data on overflow, meaning the size of the result is twice that of the operand. For example, 32-bit multiply result, the upper half goes into ”edx” and lower to ”eax”. Unsigned uses the ”IMUL” wherein the upper half of the result is truncated. This means only absolute values until the bits fill in the lower register and hence lose negative values in case of overflow.

D. GCC compiler options

For this experiment, we will be using the GNU GCC compiler and with the following arguments to generate the executable:

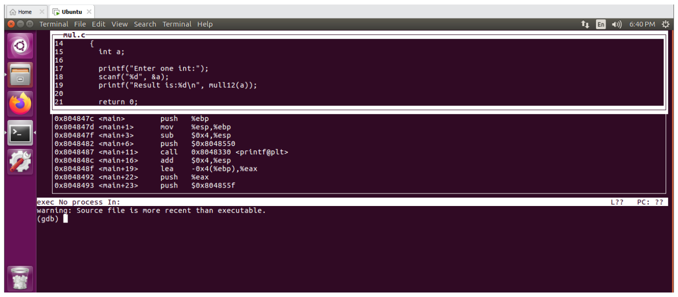
$$gcc \ -m32 \ -g \ mul.c \ -o \ mul \ -fno-stack-protector \\ -no-pie \ -mpreferred-stack-boundary=2 \\ -fno-pic \ -z \ execstack \tag{1}$$

- **-m32** : generates code for 32-bit mode, and sets int, long, pointer to 32 bits.
- **-g** : Provides meaningful symbols for debugging
- **-o** : Name of the executable
- **-fno-stack-protector** : Turns stacking smashing protector to off. The canary is not set next to the return address in the stack.
- **-no-pie, -fno-pic** : Both these options prevent dynamic memory allocation. Address randomization in virtual memory is not done and the acronym for Position Independent Executable/Code.
- **-mpreferred-stack-boundary=2** : Sets stack boundary to 4 (power of 2 or 2 square) bytes in memory resulting in smaller constraints for the code. The default boundary for an x64-based stack is 4 to 12.
- **-z execstack** : Makes the stack executable, meaning if malicious code is inserted, it can be run. In short, making the memory more vulnerable.

E. GDB debugging

After the generation of the executable, we are attaching gdb to it. Then, we are looking at both source and disassembled code together while also having terminal access.

$$(gdb) \ layout \ split \tag{2}$$



```
mul.c
14        {
15          int a;
16
17          printf("Enter one int:");
18          scanf("%d", &a);
19          printf("Result is:%d\n", mull12(a));
20
21          return 0;

0x804847c <main>        push   %ebp
0x804847d <main+1>      mov    %esp,%ebp
0x804847f <main+3>      sub    $0x4,%esp
0x8048482 <main+6>      push   $0x8048550
0x8048487 <main+11>     call   0x8048330 <printf@plt>
0x804848c <main+16>     add    $0x4,%esp
0x804848f <main+19>     lea    -0x4(%ebp),%eax
0x8048492 <main+22>     push   %eax
0x8048493 <main+23>     push   $0x804855f

exec No process In:                                              L??   PC: ??
warning: Source file is more recent than executable.
(gdb)
```

Fig. 3. gdb split layout output

To add a breakpoint according to the source file number(say line 7):

$$(gdb) \ b \ 7 \tag{3}$$

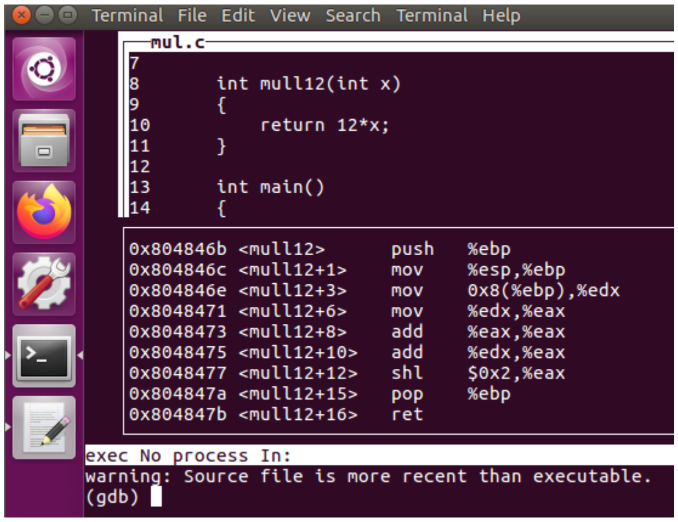
To check registers and their contents:

$$(gdb) \ info \ registers \tag{4}$$

Apart from these basic ones, to execute the program or step through it line by line, we use ”run” or ”nexti” commands.

F. Code Analysis and Verification

In the given ”mulby12.c” code, the main function takes an integer input and passes it to the multiply function. This function returns the output by multiplying with the constant in the code. In this experiment, we are varying these constants and checking the assembly code generated for the function.



```
mul.c
7
8         int mull12(int x)
9         {
10            return 12*x;
11        }
12
13        int main()
14        {

0x804846b <mull12>      push   %ebp
0x804846c <mull12+1>    mov    %esp,%ebp
0x804846e <mull12+3>    mov    0x8(%ebp),%edx
0x8048471 <mull12+6>    mov    %edx,%eax
0x8048473 <mull12+8>    add    %eax,%eax
0x8048475 <mull12+10>   add    %edx,%eax
0x8048477 <mull12+12>   shl    $0x2,%eax
0x804847a <mull12+15>   pop    %ebp
0x804847b <mull12+16>   ret

exec No process In:
warning: Source file is more recent than executable.
(gdb)
```

Fig. 4. disassembled code when the constant is 12

In the 4 above, the constant is 12 and there are nine lines of assembly code generated. The first and the last two lines are function initialization and return calls for the stack and will be same. In our experiment, we will be calculating the expensiveness(latency and throughput) of the remaining lines (five lines in the 4 above) of arithmetic operations.

III. Results

The following table captures the data derived from the disassembled code for varying constant values. The number of instructions, latency, and throughput have been calculated manually for the Intel i7 (Coffee Lake) processor.

TABLE I
CONSTANT VALUES AND THEIR MEMORY IMPLICATIONS

Sl. No.	Constant	No. of instructions	Latency	Throughput
1	12	5	6	1.75
2	0	1	0	0.25
3	22	2	5	1.5
4	29	2	5	1.5
5	64	2	3	1
6	21 Billion	2	5	1.5
7	22 Billion	2	5	1.5
8	210 Billion	2	5	1.5
9	-12	2	5	1.5
10	-29	2	5	1.5
11	-64	4	4	1.5
12	-21 Billion	2	5	1.5
13	-22 Billion	2	5	1.5
14	-210 Billion	2	5	1.5

IV. DISCUSSIONS

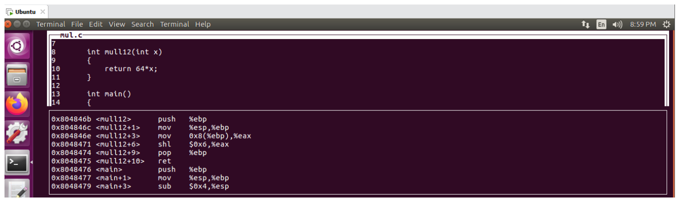
A. Calculation of latency and throughput

For current-generation CPUs, latency is the delay taken to arrive at a result whereas throughput is the number of machine cycles taken for a particular operation. In Table I above, both latency and throughput have been calculated manually for each assembly instruction from Instruction Tables (Coffee Lake). These columns represent the total added-up values for the total number of instructions in the adjacent column.

$$\begin{aligned} Latency &= latency\_ins1 + latency\_ins2 + ... \\ Throughput &= throughput\_ins1 + throughput\_ins2 + ... \end{aligned} \tag{5}$$

"**No. of instructions**" is the total lines of assembly code between the usual function call as mentioned in Section F of Methodology.

In Fig 5 below, total latency and throughput are the summations of the individual two "mov" and "shl" instructions.



```
        int mull12(int x)
        {
            return 64*x;
        }

        int main()
        {
```

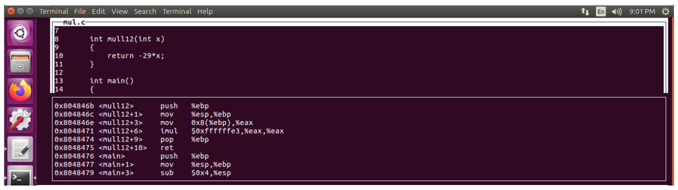
```
0x804846b <mull12>      push   %ebp
0x804846c <mull12+1>    mov    %esp,%ebp
0x804846e <mull12+3>    mov    0x8(%ebp),%eax
0x8048471 <mull12+6>    shl    $0x6,%eax
0x8048474 <mull12+9>    pop    %ebp
0x8048475 <mull12+10>   ret
0x8048476 <main>        push   %ebp
0x8048477 <main+1>      mov    %esp,%ebp
0x8048479 <main+3>      sub    $0x4,%esp
```

Fig. 5. Debugger output when the constant is 64. Add and Shift operations are used

B. Memory analysis of constants

Based on the figures above, we can conclude the following -

- Multiply instruction(like imul) takes more CPU resources than a single addition/shift operation
- Multiply by "0" is the fastest among all



```
        int mull12(int x)
        {
            return -29*x;
        }

        int main()
        {
```

```
0x804846b <mull12>      push   %ebp
0x804846c <mull12+1>    mov    %esp,%ebp
0x804846e <mull12+3>    mov    0x8(%ebp),%eax
0x8048471 <mull12+6>    imul   $0xffffffe3,%eax,%eax
0x8048474 <mull12+9>    pop    %ebp
0x8048475 <mull12+10>   ret
0x8048476 <main>        push   %ebp
0x8048477 <main+1>      mov    %esp,%ebp
0x8048479 <main+3>      sub    $0x4,%esp
```

Fig. 6. Debugger output when the constant is -24. "IMUL" operation is used

- Multiply by a prime number (29) versus any divisible number(22) may be equally expensive
- If any number is at the integer boundary (10 digits or 21 Billion) or beyond it (11 digits) is the same to the compiler
- A number that is the power of 2 (16,32,64 etc) is usually less heavy and takes the add/shift operation than the actual multiply
- For the same constant(64 vs -64), negative or signed multiplication takes same or more resources

C. Unexpected Results

Interestingly, multiplying by a constant 12 is nearly as heavy as multiplying by a 10-digit number. What is more surprising is that, -12 has fewer instructions in total with lower resource cost. Secondly, even though we are doing unsigned multiplication, the compiler uses "imul" instead of "mul". That is the reason why in case of overflow in multiplication result, the upper half is discarded.

D. Limitations and Considerations

The experiment is done on a Ubuntu 32-bit VM environment with a GCC compiler. These values will change if the compiler changes or we compile using a "-m64" flag. In a 64-bit world, 8-byte registers will come into play and signed values can be retained.

Another thing to notice is that we have static memory allocation by disabling the "PIC" flag and thus the address randomization is off. The latency and throughput calculation above varies from processor to processor and in fact in some cases multiply can be less expensive in contrast to add.

APPENDIX

C code
For this experiment, the following C code has been utilized and different executables are generated by changing the constant.

```
/* Program to demonstrate arithmetic  right shift */
//gcc -m32 -g mul.c -o mul -fno-stack-protector -no-pie -mprefe

//

#include <stdio.h>

int mull12(int x)
{
    return 12*x; /*vary the constant 12 here*/
}

int main()
{
  int a;

  printf("Enter one int:");
  scanf("%d", &a); /*user input*/

  /* function call and print result*/
  printf("Result is:%d\n", mul12(a));

  return 0;
}
```