Operating Systems

Assignment #4

담당교수 : 김태석

강의 시간 : 수2

학부 : 컴퓨터정보공학부

학번 : 2017202088

이름 : 신해담

1. Introduction

4-1. Files located in VM Area

입력 pid에 대해서, 해당하는 프로세스의 가상 메모리 주소, code, data, heap 주소, 정보 원본 파일의 전체 경로를 출력하는 module을 작성한다.

4-2. Dynamic Recompilation

컴파일된 코드를 최적화하는 D_compile을 작성한다. add, sub, imul, div 명령어가 중복으로 수행되면 해당 코드를 하나로 합쳐서 최적화한다.

2. 실행결과 및 분석

4-1. Files located in VM Area

- **task_struct**

  커널에서 유저 프로세스에 가상 메모리 공간을 할당하며, 가상 메모리 덕분에 각 프로세스들은 메모리 공간을 독점해서 사용하는 것처럼 사용할 수 있다. 프로세스의 정보가 담겨있는 task_struct 구조체에 mm_struct가 포함되어 프로세스의 메모리를 관리한다.

- **mm_struct**

  메모리 관리를 위한 구조체이다. mm_struct의 주소공간을 사용하는 task를 나타내는 mm_user, mm_struct의 참조 카운트를 나타내는 mm_count와 가상 메모리 영역의 리스트를 가지는 vm_area_struct 구조체인 mmap이 있다. 또한 code, data, heap 영역의 시작/끝 주소를 각각 start_code/end_code, start_data/end_data, start_brk/brk가 들어있다.

- **vm_area_struct**

  가상 메모리 영역을 나타내는 구조체이다. vm_start/vm_end에 VMA 영역의 시작, 끝 주소가 들어있다. vm_prev/vm_next이 다음과 이전 VMA 구조체를 나타낸다. vm_mm은 VMA가 속해있는 mm_struct를 가리킨다. vm_file은 정보 원본 파일을 가리킨다. 만약 vm_file이 NULL이면 이 VMA는 mmap으로 사용되고 있지 않으며, 따라서 원본 파일 경로를 추적할 수 없다.

- 주요 코드

```
// for each vm struct
while(vm) {
        // get file struct from vm
        file = vm->vm_file;

        // vm is used for mmap
        if(file) {
                // print mem, code, data, heap address and full path
                memset(buf, '\0', BUFFER_LENGTH);
                file_path = dentry_path_raw(vm->vm_file->f_path.dentry, buf, BUFFER_LENGTH-1);
                printk(" mem(%x~%x) code(%x, %x) data(%x, %x) heap(%x, %x) %s\n",
                        vm->vm_start, vm->vm_end, mm->start_code, mm->end_code, mm->start_data,
                        mm->end_data, mm->start_brk, mm->brk, file_path);
        }

        // get next vm
        vm = vm->vm_next;
}
```

vm_area_struct 구조체로부터 가상 메모리 주소를, mm_struct 구조체로부터 code, data, heap 주소를, file 구조체로부터 원본 파일 경로를 얻어서 출력한다.

- **결과화면**

```
[50569.914340] ######## Loaded files of a process 'a.out(19373)' in VM ########
[50569.914345]  mem(400000~401000) code(400000, 40074c) data(600e10, 601040) heap(1f5a000, 1f5a000) /home/os2017202088/working/Assignment4/4-1/a.out
[50569.914347]  mem(600000~601000) code(400000, 40074c) data(600e10, 601040) heap(1f5a000, 1f5a000) /home/os2017202088/working/Assignment4/4-1/a.out
[50569.914348]  mem(601000~602000) code(400000, 40074c) data(600e10, 601040) heap(1f5a000, 1f5a000) /home/os2017202088/working/Assignment4/4-1/a.out
[50569.914349]  mem(fe10a000~fe2ca000) code(400000, 40074c) data(600e10, 601040) heap(1f5a000, 1f5a000) /lib/x86_64-linux-gnu/libc-2.23.so
[50569.914350]  mem(fe2ca000~fe4ca000) code(400000, 40074c) data(600e10, 601040) heap(1f5a000, 1f5a000) /lib/x86_64-linux-gnu/libc-2.23.so
[50569.914351]  mem(fe4ca000~fe4ce000) code(400000, 40074c) data(600e10, 601040) heap(1f5a000, 1f5a000) /lib/x86_64-linux-gnu/libc-2.23.so
[50569.914352]  mem(fe4ce000~fe4d0000) code(400000, 40074c) data(600e10, 601040) heap(1f5a000, 1f5a000) /lib/x86_64-linux-gnu/libc-2.23.so
[50569.914353]  mem(fe4d4000~fe4fa000) code(400000, 40074c) data(600e10, 601040) heap(1f5a000, 1f5a000) /lib/x86_64-linux-gnu/ld-2.23.so
[50569.914354]  mem(fe6f9000~fe6fa000) code(400000, 40074c) data(600e10, 601040) heap(1f5a000, 1f5a000) /lib/x86_64-linux-gnu/ld-2.23.so
[50569.914355]  mem(fe6fa000~fe6fb000) code(400000, 40074c) data(600e10, 601040) heap(1f5a000, 1f5a000) /lib/x86_64-linux-gnu/ld-2.23.so
[50569.914356] ################################################################
```

a.out에서 본인의 pid를 넣어서 호출한 결과화면이다.

4-2. Dynamic Recompilation

- **objdump 분석**

```
os2017202088@ubuntu:~/working/Assignment4/4-2$ make objdump
gcc -c D_recompile_test.c
os2017202088@ubuntu:~/working/Assignment4/4-2$ ls
D_recompile  D_recompile.c  D_recompile_test.c  D_recompile_test.o
```

gcc 에 -c 옵션으로 오브젝트 파일을 생성한다.

```
objdump -d D_recompile_test.o
```

```
0000000000000000 <Operation>:
   0:   55                      push   %rbp
   1:   48 89 e5                mov    %rsp,%rbp
   4:   89 7d fc                mov    %edi,-0x4(%rbp)
   7:   8b 55 fc                mov    -0x4(%rbp),%edx
   a:   89 d0                   mov    %edx,%eax
   c:   83 c0 01                add    $0x1,%eax
   f:   83 c0 01                add    $0x1,%eax
  12:   6b c0 02                imul   $0x2,%eax,%eax
  15:   b2 02                   mov    $0x2,%dl
  17:   f6 f2                   div    %dl
  19:   83 e8 01                sub    $0x1,%eax
  1c:   83 c0 01                add    $0x1,%eax
  1f:   83 c0 02                add    $0x2,%eax
  22:   83 c0 03                add    $0x3,%eax
  25:   83 c0 01                add    $0x1,%eax
  28:   83 c0 02                add    $0x2,%eax
  2b:   83 c0 01                add    $0x1,%eax
  2e:   83 c0 01                add    $0x1,%eax
  31:   6b c0 02                imul   $0x2,%eax,%eax
  34:   6b c0 02                imul   $0x2,%eax,%eax
  37:   6b c0 02                imul   $0x2,%eax,%eax
```

objdump -d [object 파일명] 명령어로 파일을 dump 뜰 수 있다.

```
 f:   83 c0 01                add    $0x1,%eax
12:   6b c0 02                imul   $0x2,%eax,%eax
15:   b2 02                   mov    $0x2,%dl
17:   f6 f2                   div    %dl
19:   83 e8 01                sub    $0x1,%eax
```

최적화 대상인 add, sub, div, imul 을 분석한다.

add와 **sub**는 첫 byte가 83이고, 세번째 byte가 더하거나 빼는 값이 들어간다.

imul은 첫 byte가 6b이고, 세번째 byte가 곱할 값이 들어간다.

div 는 target register 에 값을 mov 하고 그 값으로 나눗셈하게 된다. mov 의 두번째 byte 에 나눌 값이 들어가고, div %dl 은 2 byte 의 f6 f2 이다.

각 명령어가 중복되어 나타날 때, 값을 합침으로써 하나의 명령어로 바꿀 수 있을 것이다.

- **주요 코드**

```
void sharedmem_init()
{
        segment_id = shmget(1234, PAGE_SIZE, 0);
        Operation = (uint8_t*)shmat(segment_id, NULL, 0);
}

void sharedmem_exit()
{
        shmdt(Operation);
        shmctl(segment_id, IPC_RMID, NULL);
}
```

sharedmem_init 함수는 D_recompile_test.c에서 생성한 shared memory의 공간에 접근해서, 들어있는 함수를 읽어온다.

sharedmem_exit 함수는 사용이 끝난 shared memory 공간을 분리해서 제거한다.

```
void drecompile_init(uint8_t *func)
{
        int fd;
        int i = 0;
        char temp[PAGE_SIZE];

        fd = open("Operation", O_RDWR|O_CREAT, S_IRUSR|S_IWUSR|S_IXUSR);
        for(i = 0; i < PAGE_SIZE; i++) temp[i] = '@';
        write(fd, temp, PAGE_SIZE);
        lseek(fd, 0, 0);

        // memory mapping
        compiled_code = mmap(0, PAGE_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);

        // copy memory
        memcpy(compiled_code, Operation, PAGE_SIZE);
}

void drecompile_exit()
{
        // unmap
        munmap(compiled_code, PAGE_SIZE);
}
```

drecompile_init 함수는 read, write 권한을 가진 메모리 공간을 할당한다. 공간

할당을 위해 임의로 파일을 생성한 다음 PAGE_SIZE만큼 write한다. 그리고 memcpy 함수를 사용해서 shared memory의 함수를 복사한다.

derecompile_exit 함수는 할당했던 메모리 공간을 unmap한다.

```
// for all function instruction
while(func[i] != 0xc3) {
        // operation, source register
        code[0] = func[i];

        // op is not target of optimization
        if(code[0] != 0x83 && code[0] != 0x6b && code[0] != 0xf6) {
                compiled_code[j++] = func[i++];
        }
```

drecompile 함수에서 복사한 함수를 최적화한다.

원본 함수의 명령어들을 읽어들인다. 명령어가 add, sub, imul, div가 아니라면 덮어쓴다.

```
else {
        // init value
        code[1] = func[i+1];
        code[2] = code[0] == 0x83 ? 0 : 1;

        if(code[0] == 0xf6) {   // div
                // same code[0-1]
                while(func[i] == code[0] && func[i+1] == code[1]) {
                        code[2] *= func[i-1];

                        // if repeated div, i=i+4
                        // else,            i=i+2
                        i = func[i+4] == code[0] ? i+4 : i+2;
                }
                compiled_code[j-1] = code[2];
                compiled_code[j++] = code[0];
                compiled_code[j++] = code[1];
        }
```

add와 sub는 0으로 초기화하고, imul과 div는 1로 초기화한다.

div 명령어가 반복해서 나타난다면 나눠지는 값들을 곱해서 하나의 값을 얻는다. 반복이 끝나면 얻은 값을 compiled_code에 반영한다.

```
else {                  // not div
        // same code[0-1]
        while(func[i] == code[0] && func[i+1] == code[1]) {
                if(code[0] == 0x83) // add suv
                        code[2] += func[i+2];
                else                // imul
                        code[2] *= func[i+2];
                i += 3;
        }
        compiled_code[j++] = code[0];
        compiled_code[j++] = code[1];
        compiled_code[j++] = code[2];
}
```

add, sub, imul은 모두 3 byte로 구성되므로 같이 묶을 수 있다. Add/sub는 반복되는 값을 더하고, imul은 반복되는 값을 곱한다. 명령어 반복이 끝나면 값을

compiled_code에 반영한다.

```
// chmod r-x
mprotect(compiled_code, PAGE_SIZE, PROT_READ | PROT_EXEC);
```

수정한 compiled_code에 실행권한을 부여한다.

- **실행결과**

```
start = clock();
for(int i = 0; i < 10000; i++) func(i);
end = clock();
```

D_recompile_test.c의 Operation 함수를 한번 수행하는 경우 시간이 너무 작아서 정확한 비교가 힘들기 때문에, 10000번 반복하면서 시간을 비교한다.

기존코드	0.000159	0.000160	0.000160
0.000261	0.000159	0.000160	0.000159
0.000169	0.000159	0.000159	0.000181
0.000159	0.000159	0.000160	0.000159
0.000170	0.000285	0.000159	0.000169
0.000159	0.000159	0.000169	0.000168
0.000160	0.000159	0.000160	0.000160
0.000159	0.000163	0.000160	0.000159
0.000159	0.000160	0.000159	0.000282
0.000159	0.000159	0.000158	0.000159
0.000169	0.000160	0.000169	0.000159
0.000159	0.000158	0.000158	0.000158
0.000171	0.000159	0.000159	0.000168
컴파일한 코드	0.000028	0.000029	0.000029
0.000028	0.000028	0.000028	0.000029
0.000029	0.000028	0.000029	0.000028
0.000028	0.000028	0.000029	0.000028
0.000028	0.000029	0.000029	0.000028
0.000029	0.000028	0.000028	0.000028
0.000028	0.000083	0.000028	0.000029
0.000029	0.000028	0.000028	0.000028
0.000029	0.000029	0.000043	0.000029
0.000031	0.000029	0.000029	0.000029
0.000029	0.000028	0.000029	0.000028
0.000028	0.000028	0.000028	0.000029
0.000029	0.000029	0.000028	0.000030

기존 코드의 경우 평균 0.000168초 걸리며, 컴파일한 코드는 평균 0.000030초 걸리는 것을 확인할 수 있다.

3. Reference

4-1. mm_struct, vm_area_struct / https://showx123.tistory.com/92