CSCE 451
Abnob Doss and Matthew Bruns
04/01/18

HW 3 – Disassembly Assignment

Introduction:

Initial development took place with the use of the Capstone library, and the base design was made with that library and its variables in mind.

Abnob wrote the original linear assembler as well as the main function, which handles arguments as described below. He adapted the program to the second library, as we began in Capstone. Doss also wrote two other functions: ‘get_elf_entry_point’ to determine where the disassembler would begin in a recursive disassembly, and ‘if_elf’ to note whether a file was made in the ELF format or not.

Matthew was the main contributer to the recursive disassembler (‘recursive_disasm’) algorithm and design, also designing its helper functions ‘unchecked’, ‘update_list’, and ‘addpoints’ functions. He performed the experiments and wrote the report. He also modified ‘get_elf_entry_point’ into ‘get_text’ to draw data on the offset of the text section and its length if need be.

Running the Program:

This program requires the most current edition of python, as well as the capstone library and (TODO: List every other library we’re using). Executed from Linux terminal within the directory where the program and your target binary is being held, execute:

```
bounce.py [-h] --file f [--linear | --recursive] [--capstone | --pydasm] [--64 | --32]
```

[-h]: Help (Optional flag)

--file f: Substitute f for the filename.

[--linear | --recursive]: linear or recursive disassembly.

[--capstone | --pydasm]: Select library to translate with.

[--64 | --32]: x64 or x32 format.

Example (binary filename ‘standard’, using recursive dissassembly, recursive, Capstone, x64 format): python bounce.py --file standard --recursive --capstone --64

Pseudocode:

```
linear_disasm:
    0: Take the filename, the proper label, and the literal offset in bytes
    1: Seek the proper location the .text section begins at to properly disassemble
    2: For every line in the binary file:
        2.1: For every set of commands in the line
            2.1.1: Disassemble the code
            2.1.2: Print the useful information (address, command, target(s))
recursive_disasm:
    0: Take the start point, the file we are considering, and the offset in bytes
    1: Seek the proper location the .text section begins at to properly disassemble
        1.1: If we can’t go to the offset for some reason report error and return
    2: For every line in the binary file:
        2.1: For every set of commands in the line
            2.1.1: Disassemble the code
            2.1.2: Print the useful information (address, command, target(s))
            2.1.3: Return on a return command
            2.1.4: If there is an unconditional jump:
                2.1.4.1: Save the range of addresses we have considered
                2.1.4.2: Try to check if the target jump is a literal int
                    2.1.4.2.1: If we can recurse from the new position
                    2.1.4.2.2: If we can’t don’t try to, report error
                2.1.4.3: Return- an unconditional jump is the end of this line
            2.1.5: If there is a conditional jump:
                2.1.5.1: Save the range of addresses we have considered
                2.1.5.2: Try to check if the target jump is a literal int
                    2.1.5.2.1: If we can recurse from the new position
                    2.1.5.2.2: If we can’t don’t try to, report error
```

Experiment 1:

Referencing files rand1 and rand3, which were each created with a byte offset slightly different from the regular one through psuedo-random integer generation, we can see that where rand3.txt shows the disassembler starting off in the middle of an instruction relative to the OBJDUMP and recursive standard code. However, on the very next instruction, it returns to its regular results, showing that linear recovery will generally correct itself quickly. However, in a situation where the results were not so forgiving (IE: the results of an accidental offset are very similar to another program), the disassembler may accidentally take several instructions before correcting. However, this outcome is unlikely.

Experiment 2:

Linear to OBJDUMP:

(Files of reference: experiment_2_standard_linear.txt, experiment_2_hello_linear.txt, experiment_2_mainreturn_linear.txt, experiment_2_standard_objdump.txt, experiment_2_hello_objdump.txt, experiment_2_mainreturn_objdump.txt)

standard- The discovered functions were start, deregister_tm_clones, register_tm_clones, __do_global_dtors_aux, frame_dummy, main, __libc_csu_init, __libc_csu_fini, and _fini. This meant all the functions within .text were discovered.

Almost every line was either identical or an equivalent. Several lines were processed in similar terms, such as

Linear-

```
0x400576:     nop     word ptr cs:[rax + rax]
```

compared to

OBJDUMP:

```
400576:         66 2e 0f 1f 84 00 00   nopw   %cs:0x0(%rax,%rax,1)
```

The one outstanding translation error between Linear to OBJDUMP is that linear includes an add at 0x400582 that is not present in OBJDUMP.

```
0x400582:     add     byte ptr [rax], al
```

hello- The discovered functions were start, deregister_tm_clones, register_tm_clones, __do_global_dtors_aux, frame_dummy, main, __libc_csu_init, __libc_csu_fini, and _fini. This meant all the functions within .text were discovered.

The previous translation errors persist here, as well as an additional set of lines beyond the final function, beginning at 0x400582 and at several times printing out nonexistent instructions from erroneously-labeled lines, starting at the base label passed in at the start. Most of the instructions are 'add   byte ptr [rax], al'. This is likely the linear disassembler handling junk data while OBJUMP does not consider it at all. Otherwise, the translation is correct.

mainreturn- The discovered functions were start, deregister_tm_clones, register_tm_clones, and __do_global_dtors_aux. __do_global_dtors_aux was not completely translated.

This program produced the most outstanding errors out of any of the ones tested. While OBJDUMP had no issues with line 400543, the linear decomplier basically collapsed as a result. It translated the previous line correctly, but when it came upon 'c6 05 f6 0a 20 00 01', translated by OBJDUMP below-

OBJDUMP:

```
c6 05 f6 0a 20 00 01   movb   $0x1,0x200af6(%rip)        # 601040 <__TMC_END__>
```

it proceeded to consider the following lines as beginning from 0x400480, and continued translating, albeit incorrectly. It seemed to be considering legitimate information, but the bytes it took in weren't correctly handled by the library, and the results were not the same as the same-numbered lines. This was essentially the same error as occurred in hello, except it occurred before all functions had been completed, in the middle of __do_global_dtors_aux. Evidently, this is the result that occurs when the program notices that it is handling data that it cannot properly process.

Recursive to OBJDUMP:

(Files of reference: experiment_2_standard_recursive.txt, experiment_2_hello_recursive.txt, experiment_2_mainreturn_recursive.txt, experiment_2_standard_objdump.txt, experiment_2_hello_objdump.txt, experiment_2_mainreturn_objdump.txt)

standard- the discovered functions were start and __libc_start_main@plt

While the program performed efficient and accurate translation, due to the nature of the unconditional jump at 0x400404 followed by another non-int jump at 0x4003c0, there's no way for the disassembler to continue without real-time analysis, which is not currently possible.

hello- the discovered functions were start and __libc_start_main@plt.

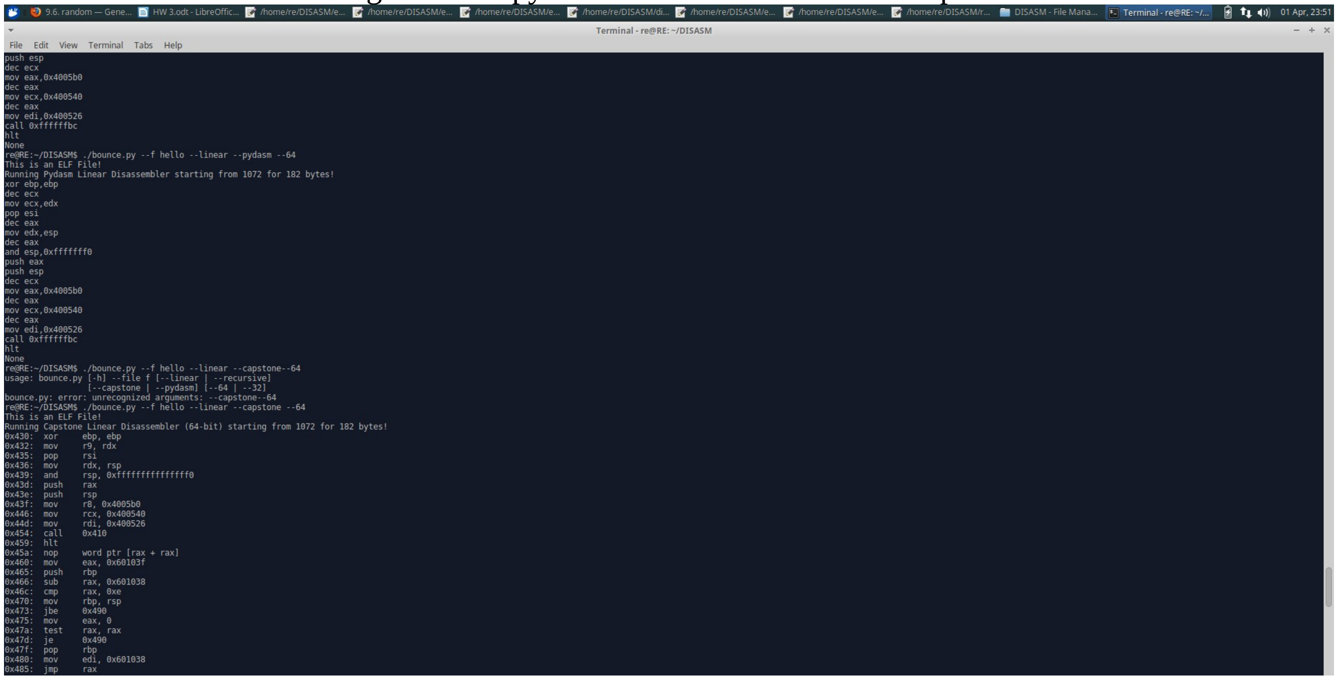
While the program performed correct translation for start, when it moved to __libc__start___main, it ended up mistranslating the jump taking place at line 0x400410, turning it into an or followed by three adds, a sar, and then another series of adds similar to the previous example of what happens when junk data is processed by the disassembler.

mainreturn- the discovered functions were start and __libc_start_main@plt.

While the program performed efficient and accurate translation, due to the nature of the unconditional jump at 0x4004a4 followed by another non-int jump at 0x400460, there's no way for the disassembler to continue without real-time analysis, which is not currently possible. This was virtually identical to the results gathered from the standard file's analysis.

<u>Experiment 3:</u>

Using Capstone as a base, we compared its output to pydasm and found that for the most part, the assembly codes were extremely similar. This was supported by the fact that they both used Intel formatting and a direct comparison. However, upon close analysis, the different disassemblers interpret the results under different registers and pydasm decrements ecx where capstone does not.



```
push esp
dec ecx
mov eax,0x4005b0
dec eax
mov ecx,0x400540
dec eax
mov edi,0x400526
call 0xffffffbc
hlt
None
re@RE:~/DISASM$ ./bounce.py --f hello --linear --pydasm --64
This is an ELF File!
Running Pydasm Linear Disassembler starting from 1072 for 182 bytes!
xor ebp,ebp
dec ecx
mov ecx,edx
pop esi
dec eax
mov edx,esp
dec eax
and esp,0xfffffff0
push eax
push esp
dec ecx
mov eax,0x4005b0
dec eax
mov ecx,0x400540
dec eax
mov edi,0x400526
call 0xffffffbc
hlt
None
re@RE:~/DISASM$ ./bounce.py --f hello --linear --capstone--64
usage: bounce.py [-h] --file f [--linear | --recursive]
                 [--capstone | --pydasm] [--64 | --32]
bounce.py: error: unrecognized arguments: --capstone--64
re@RE:~/DISASM$ ./bounce.py --f hello --linear --capstone --64
This is an ELF File!
Running Capstone Linear Disassembler (64-bit) starting from 1072 for 182 bytes!
0x430:  xor     ebp, ebp
0x432:  mov     r9, rdx
0x435:  pop     rsi
0x436:  mov     rdx, rsp
0x439:  and     rsp, 0xfffffffffffffff0
0x43d:  push    rax
0x43e:  push    rsp
0x43f:  mov     r8, 0x4005b0
0x446:  mov     rcx, 0x400540
0x44d:  mov     rdi, 0x400526
0x454:  call    0x410
0x459:  hlt
0x45a:  nop     word ptr [rax + rax]
0x460:  mov     eax, 0x60103f
0x465:  push    rbp
0x466:  sub     rax, 0x601038
0x46c:  cmp     rax, 0xe
0x470:  mov     rbp, rsp
0x473:  jbe     0x490
0x475:  mov     eax, 0
0x47a:  test    rax, rax
0x47d:  je      0x490
0x47f:  pop     rbp
0x480:  mov     edi, 0x601038
0x485:  jmp     rax
```