Clayton Petty
Bradley Swain

Assignment 2: Task 3 & 4 Report

Task 3

Problem:

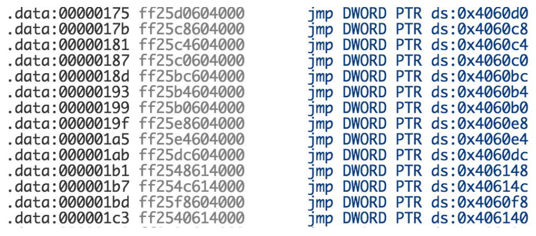
When we look at a random piece of binary data, we would like to disassemble it into its original instructions and data. This presents a practical "disassembly problem" because we do not know whether or not the piece of binary data we would like to disassemble contains instructions, data, or some combination thereof. This makes the code difficult to disassemble, as the binary data could begin on an instruction, in the middle of an instruction, or even in the middle of program data. For task 3, we tackled this problem to see if we could intelligently classify a piece of binary data as instructions or data.

Initial Identification:

The first thing we did was look at each of the five matching substrings between all four files in a hex editor and disassembler. We then looked for trends or patterns that we thought might indicate if we were looking at data or instructions. For instance, when we disassembled the longest substring, we got the following snippet of instructions:

```
0:   00 02                  add    BYTE PTR [edx],al
2:   00 00                  add    BYTE PTR [eax],al
4:   00 b0 6c 40 00 e8      add    BYTE PTR [eax-0x17ffbf94],dh
a:   6c                     ins    BYTE PTR es:[edi],dx
b:   40                     inc    eax
c:   00 00                  add    BYTE PTR [eax],al
e:   00 00                  add    BYTE PTR [eax],al
10:  00 f0                  add    al,dh
12:  12 40 00               adc    al,BYTE PTR [eax+0x0]
15:  00 00                  add    BYTE PTR [eax],al
17:  00 00                  add    BYTE PTR [eax],al
19:  6c                     ins    BYTE PTR es:[edi],dx
1a:  40                     inc    eax
1b:  00 00                  add    BYTE PTR [eax],al
1d:  00 00                  add    BYTE PTR [eax],al
1f:  00 90 80 40 00 00      add    BYTE PTR [eax+0x4080],dl
25:  00 00                  add    BYTE PTR [eax],al
27:  00 ff                  add    bh,bh
29:  ff                     (bad)
2a:  ff                     (bad)
```

In our opinion, this indicates that we are not looking at instructions, or we are starting in the middle of an instruction. This is because of the "(bad) instructions at 29 and 2a, and the strange add operations that occur frequently. In a different common substring we also observed an unlikely series of instructions, in this case a long series of unconditional jumps to roughly the same area in memory. This may be another indication that this is either data or the string starts in the middle of an instruction.



```
.data:00000175 ff25d0604000        jmp DWORD PTR ds:0x4060d0
.data:0000017b ff25c8604000        jmp DWORD PTR ds:0x4060c8
.data:00000181 ff25c4604000        jmp DWORD PTR ds:0x4060c4
.data:00000187 ff25c0604000        jmp DWORD PTR ds:0x4060c0
.data:0000018d ff25bc604000        jmp DWORD PTR ds:0x4060bc
.data:00000193 ff25b4604000        jmp DWORD PTR ds:0x4060b4
.data:00000199 ff25b0604000        jmp DWORD PTR ds:0x4060b0
.data:0000019f ff25e8604000        jmp DWORD PTR ds:0x4060e8
.data:000001a5 ff25e4604000        jmp DWORD PTR ds:0x4060e4
.data:000001ab ff25dc604000        jmp DWORD PTR ds:0x4060dc
.data:000001b1 ff2548614000        jmp DWORD PTR ds:0x406148
.data:000001b7 ff254c614000        jmp DWORD PTR ds:0x40614c
.data:000001bd ff25f8604000        jmp DWORD PTR ds:0x4060f8
.data:000001c3 ff2540614000        jmp DWORD PTR ds:0x406140
```

To see if we were starting in the middle of an instruction, we then deleted bytes off of the front of the substring and re-disassembled the code, to see if we ever got a realistic set of disassembled instructions. We deleted a max of 14 bytes, as the longest possible instruction is 15 bytes. We repeated this process for each of the five substrings.

In addition to looking at the disassembled code, we looked at the binary in a hex editor. In the first substring we notice something that looks fairly "data-esque" in the form of actual words and human readable content. This seems to indicate that this substring is data, not instructions. We then repeated this process for each of the other 5 substrings.

```
........??1_Lot@
std@@QAE@XZ..m.?
?ot@std@@QAE@H@Z
...?cout@std@@$b
asistream@DU?$ch
ar_traits@D@std@
@@....?_Getgloba
lloe@loe@std@@CA
PAV_Lop@@XZ....?
uncaught_extion@
std@@YA_NXZ...?c
in@std@@$basistr
eam@DU?$char_tra
```

Our initial identification lead us to believe the following:

1. 1st longest substring is data
2. 2nd longest substring is Instructions
3. 3rd longest substring is data
4. 4th longest substring is data
5. 5th longest substring is data

Trends and Patterns:

Over static analysis of the five substrings we noticed a few trends that we would eventually implement in our model. These were things that we hypothesized would help identify a section of binary code as data or instructions. These include:

1. Large number of the most common instructions (and therefore bytes that indicate that instruction:
    a. mov, push, call, cmp, add, pop, lea, test
2. Large amounts of padding potentially indicate data
3. Look for unlikely combinations of instructions

We can break these down to explain why we hypothesize that they will have a positive effect on determining whether a block of binary is data or instructions.

First, we have "large number of common instructions". We can look at the block of binary data and look for common opcodes. 70% of total operations use only 8 different operators. These are mov, push, call, cmp, add, pop, lea, test. Based on the number of these opcodes we detect in the binary string. Based on how many possible operations there could be (using an average number of bytes per operation of 6), we can determine whether or not we think there are the appropriate number of possible operations in the binary string.

Secondly, we look for large amounts of padding. If there is a ton of padding, we found our analysis of the 5 common substrings that it generally look more like data than instructions. This was especially true when there were multiple sections of non-padding separated by multiple sections of high-quantity padding. This seemed to us like data.

Finally, we look for unlikely combinations of instructions. This means we look for things like back to back RET opcodes, or back to back unconditional JMP instructions (or any combination thereof). If we see these, then it is less likely that we are looking at instructions and more likely we are looking at data.

Our Model:

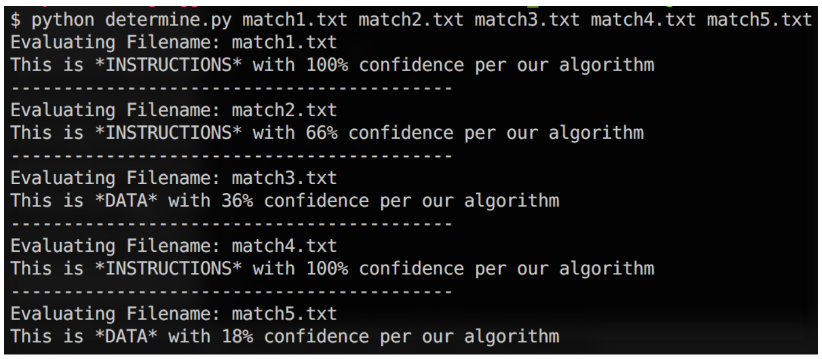
Our model implements what I described above. There is a class called HexString which has a number of functions to score a hex string. The HexString class contains most of the program logic. Each of the scoring functions awards either positive points for looking like instructions or negative points for looking like data. The main functions are as follows:

1. score_has_opcodes(): this function counts up the number of potential common opcodes and compares them to the number of possible opcodes based on average operation size. It then assigns a score accordingly. The score is determined by how close to the expected number of possible opcodes we see. If we are within 30% of the expected, we award a positive number of points, otherwise, negative.
2. score_padding(): This function looks at how much padding is in the hex string and gives a score accordingly. It creates a ratio of padding to non padding content in the hex string. If the ratio is above .5, we award negative points, otherwise, we award positive points.
3. score_has_unlikely_opcode_seq(): this function looks into whether or not there are unlikely combinations of opcodes back to back. It then assigns a score accordingly. The score begins at a positive amount of points and then slowly becomes negative with each unlikely opcode combination found.
4. score() and results_string(): These functions add up the scores and then calculate the end result and confidence level, and can output a string describing the end decision. This is mostly based off of a negative vs positive final score.

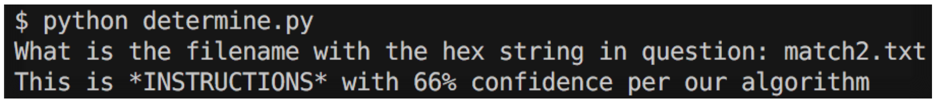
The scores assigned via our model are negative if the scoring function believes the string looks more like data, or positive if the string looks more like instructions. We then sum up all the scores at the end and based on whether that number is positive or negative, we know if the string has been predicted data or instructions. Based on how extremely positive or extremely negative the number is, we can get the percentage confidence of our model.

How to Use Our Model:

It is easy to test our model. All you need to do is run the program "determine.py" with python 2.7. You can either provide the filenames as command line parameters, or if you run the program with no parameters, it will prompt you for a filename. It then automatically scores and outputs the result. The files should be a single line string in hex. Below are two example runs:



```
$ python determine.py match1.txt match2.txt match3.txt match4.txt match5.txt
Evaluating Filename: match1.txt
This is *INSTRUCTIONS* with 100% confidence per our algorithm
----------------------------------------
Evaluating Filename: match2.txt
This is *INSTRUCTIONS* with 66% confidence per our algorithm
----------------------------------------
Evaluating Filename: match3.txt
This is *DATA* with 36% confidence per our algorithm
----------------------------------------
Evaluating Filename: match4.txt
This is *INSTRUCTIONS* with 100% confidence per our algorithm
----------------------------------------
Evaluating Filename: match5.txt
This is *DATA* with 18% confidence per our algorithm
```



```
$ python determine.py
What is the filename with the hex string in question: match2.txt
This is *INSTRUCTIONS* with 66% confidence per our algorithm
```

Issues with Our Model:

Our model is far from perfect and has many notable issues. One major issue is that valid data could potentially look like valid instructions. Given only a small subset of the total binary it would be essentially impossible to tell if the subset represents instructions or data, especially if that subset could be interpreted both as a valid data and as valid instructions. Our method is particularly prone to this situation as it does not check for instructions vs. data, but more closely resembles checking for instructions vs. not instructions. While there are plenty of ways to score positively for looking like valid instructions and negatively for not looking like valid instructions, there are not any checks that something looks like valid data or does not look like valid data. Because of this, in the case that something is both valid data and valid instructions, our method would only score it positively for recognizing valid instructions and not realize that it is also potentially valid data.

Another potential issue with our approach is the assumption that padding is associated with data and not instructions. In our limited research, it appears that padding is significantly more likely to occur in data sections although we are not certain that it is impossible for padding to occur in instructions. If there were padding included in instructions this portion of our metric would potentially fail and falsely identify the instructions as data. It is also possible that padding could occur between instructions and data, which leads to another shortcoming of our method.

Given a large enough string the information contained within could be a mix of both instructions and data. Our method does not account for this at all. Our method assumes the entire string is either all data or all instructions and assign a single score to the entire string. A potential solution to this problem was discussed where the string would be broken up into smaller sections and scored individually, then put back together one by one and rescored as a whole to see if any patterns could be seen within different areas of the string. This approach would, in theory, work similar to merge sort using a data vs. instruction heuristic instead of regular sorting.

Lastly, a potential but unlikely failure case for our method is that code could be crafted to purposely make our system fail. A nonsensical but valid set of instructions filled with nop slides would be classified as data using our system as it attempts to check for the exact opposite in a real set of instructions.

Additional Approach:

A more robust method used by various disassemblers makes use of two concepts, *Linear Sweep* and *Recursive Traversal*. The end goal of these techniques is essentially to test whether or not the file can be interpreted as a logical set of instructions. The Linear Sweep method works best on sets of instructions that execute sequentially one after another. Linear Sweep steps through and converts everything into sequential instructions. If at any point Linear

Sweep encounters bad instructions or a set of instructions that do not make sense, a large number of nonsensical jumps being one example, it is likely that the data being viewed is not a linear set of instructions. Linear Sweep is less robust given a nonlinear set of instructions. Branching paths of execution are more difficult to evaluate with Linear Sweep. This is where Recursive Traversal becomes useful. Recursive Traversal follows the various branching paths of execution, where Linear Sweep would have essentially ignored them and continued on sequentially. Both systems break down when some aspect of execution is calculated at runtime. A ret call is one example that cannot be easily traced, as it depends on the address of the next instruction to be executed being taken from the stack which cannot be determined without actually running the program.

It would have been interesting to attempt to implement a system that made use of the Linear Sweep/Recursive Traversal techniques, but proved to be to difficult in the time given. Firstly, given only a small portion of a binary it can be difficult to determine where the next instruction starts. This is fairly trivial to solve and was addressed above. The largest possible instruction in x86 is 15 bytes long so by evaluating the data 15 times, shifting the starting byte over by one each time, you are guaranteed to have evaluated the the data at the start of the next instruction if it is indeed a set of instructions. A much larger problem, and the real reason we did not attempt this approach, is that these techniques would require the ability to parse a string of hex values into valid x86 instructions. While not exactly a difficult task, it would certainly take quite some time to complete and we only had one week within the scope of this assignment.

Why This is a Hard Problem:

Being given essentially a random hex string and then trying to identify it as either valid instructions or data is not an easy problem. Because everything is just ones and zeroes, the meaning of those ones and zeroes depends entirely on how you interpret them. Interpreting a set of ones and zeroes as data may be just as valid as interpreting them as instructions. Even if a header for the executable is available, it is not of any use unless the original location of the hex string is known. Were the original location of the hex string known, the header would tell what type of information is contained in that section and no sort of disassembly or guess work would be required to determine whether the hex string represents instruction or data.

Task 4

We had just barely started looking into creating a plugin for Ollydbg when we found out that this task had been made optional, and decided to focus our attention on Task 3 instead. However, prior to that decision some progress was made. We chose Ollydbg mostly because it seemed less complicated to write a plugin for than IDA Pro. In the real world, given more time, IDA Pro seems more powerful with more features, better documentation, and more thorough guides on how to interact with its various features. However, in the scope of this project, IDA Pro is significantly more complex than Ollydbg. When looking for guides to writing plugins for each, we found a 150 page document for IDA Pro and a roughly 10 paragraph document for

Ollydbg. Even though Ollydbg has spotty documentation at best, given one week, 10 paragraphs seemed much more approachable than 150 pages.

We found that Ollydbg recommended a specific, and old, C++ compiler, *Borland C++ 5.5*. We were able to acquire the Borland C++ 5.5 compiler and began to get it set up Ollydbg's Plugin Development Kit. We were also able to find the source code of a few Ollydbg plugins although many only gave the precompiled binary. We began looking into porting our logic over to a C++ program and how we would implement it into a Ollydbg plugin, but it was a t this stage that we found out the task was optional and began to focus our efforts elsewhere.

Resources

http://www.utd.edu/~kxh060100/wartell-pkdd11.pdf
http://unixwiz.net/techtips/win32-callconv-asm.html
https://www.strchr.com/x86_machine_code_statistics
http://sparksandflames.com/files/x86InstructionChart.html
http://resources.infosecinstitute.com/linear-sweep-vs-recursive-disassembling-algorithm/#gref
http://reverseengineering.stackexchange.com/questions/2347/what-is-the-algorithm-used-in-recursive-traversal-disassembly
http://blog.onlinedisassembler.com/blog/?p=23
http://www.ollydbg.de/Help/API.htm
http://www.ollydbg.de/pdk.htm
https://www.hex-rays.com/products/ida/support/idapython_docs/
http://www.binarypool.com/page_id=53/index.html