Comp 360
Final project: Open-source Antivirus
Alex Piazza, Sam Javal, Jeremy Kattan, Kevin Koech

Introduction

- We build open-source anti-virus app for MacOS.
- How antiviruses work:
    - Check the file signature of each downloaded file against the file signatures of a database of known viruses or malwares. If any matches, it's a virus. Problem: What if a virus is slightly altered, or hidden inside a non-virus file?
    - Heuristic analysis - (for example) if first part returns false, run each downloaded file in a virtual environment, if it "behaves badly," it's probably a virus
- We can't (or don't know how, or don't want to?) do heuristic analysis, but we don't want to just use the signatures either, because then we only will catch viruses identical to those already known. A lot of them are known, but the problem is attackers can slightly modify an already known virus, thus changing the virus, and we can't catch this

- Idea: Check for viruses in a way that allows to catch these slightly altered viruses, or viruses hidden in non-virus files.
- So, here's what we do:

What we did:

- First step - get a list of known virus files from theZoo - A Live Malware Repository. Most of these files are executable files such as .apk, .exe, .bin, app. These files are stored in password-protected zipfiles, so they are safe to be stored if you're wondering about that.

- For each of these files, we use the hex() function in Python to get the hexadecimal version of each file, which is basically a very very long string of numbers/letters (0-9, a-f, of course) that displays the exact contents of the file at the very lowest level. For each virus file, we made a nested dictionary of length 1 of the form { "virusname" : {"md5hash" : checksum, "hexdump" : hex_value } }. We created a big dictionary that stores all of our viruses along with their hexes, and stored this in a JSON file, which we put on our github. This is basically our database of virus files.

- Then in our code, we obtain the nested dictionary from this json file.
- Set up an infinite loop in another process (using python's Multiprocess library, so we can run the user interface at the same time) to monitor the user's download folder (using OS library). Keep track of size and new size, check if new size bigger than size, if it is, get the most recently added item in the directory:
    - If it's a file, convert to the hexadecimal format, run the algorithm - if it returns true (virus), notify the user and delete the file if they want it deleted

    - If it’s a folder, convert each file in the folder to the hexadecimal format, run the algorithm on each in the folder and if we ever get a true, stop and alert the leader immediately, giving them the option to delete the entire folder
- We used Rumps and py2app to make this into a menu-bar app for MacOS

Algorithms:
We scan a file for viruses using two algorithms described below.

1. Algorithm 1:
    - Basic idea: check if any substring of size 200 (which corresponds to 8kb (200*20*2 bytes), which is the smallest possible size of an executable and that’s the type of file we’re targeting) in the hexadecimal of a downloaded file exists in any of the virus hexes
    - So, get a list of all 200-length substrings in the downloaded file, and also do this for each download files
    - Check for any overlap (any common element) between the list of the downloaded file’s substrings and the list of the first virus’s substrings. Then do this for the second virus, and so on... If at any point there’s an overlap, it’s a virus, and we stop and return true, otherwise we keep going - if we get to end, we return false
    - Optimize by first checking smaller (16) length substrings, and if there are no matches, immediately return false (if no matches for 16, certainly no matches for 200), then do 32, then 64,.... Until we get to a number bigger than 200. This helped a lot because for most files, it would stop at 16. If we get a match for the last step, return True and it’s a virus
    -

2. Algorithm 2
   This algorithm essentially performs sequence-based string similarity analysis. It checks for the frequency of similar substrings in the strings of the hexes of our file and the given list of viruses. We do so in the following steps:
    - We split the hex strings of file and virus into chunks of length 4. Why 4? Because I decided that 4 is an approximate length of a hex string that represents something meaningful in a file. For example, if hex string of our file was 'abcdeabcdabcdfgh', the resulting string would be 'abcd abcd efgh'
    - Get intersection of strings in both the file and a given virus file. That is, chunks of the hex strings in both virus and file. For example, if our file representation was 'abcd', 'abcd', 'efgh', and virus representation was ['abcd', 'abcd', 'abcd', 'efgh'], then: intersection(file: set, virus: set) -> ['abcd', 'efgh']. At this point, if the intersection is awfully similar to virus or file, then we flag the file as malicious.

- Next, we analyze the positions of each of the chunks in the intersection in our file and the virus. We initialize a counter to track strings that repeat in a contiguous manner in both the virus and file, i.e., a contiguous subsequence.
  One thing to note that the purpose of doing the intersection is to trim down the search for a contiguous subsequence by only looking for the positions of chunks that we have established exist in both the file and virus. This greatly improves the efficiency of our algorithm.

- We then calculate a virus metric v:
  $v = m / (((len(virus) + len(file)) / len(intersection)$
  $where\ v \in \{\mathbb{Z} : [0, ((len(virus) + len(file))/2],\ and\ m\ is\ the\ 'longest\ contiguous\ subsequence'$

- Currently, I set the virus threshold to be 1.0. So if the virus metric of a file is > 1.0, the file is flagged as a virus.
- Worst case complexity of our algorithm is O(n^2)

Problems:

- Speed - big problem - solutions:
  - We linearly compressed the hexadecimal string of each virus by only keeping every 20th character in the string. We do the same for each downloaded file. Since linear doesn't affect our pattern matching.
  - Multiprocessing - get the number of core's (say this is n) in the user's computer and create a new process for each one. Then divide the big dictionary into n (approximately) equal parts. Then, when a file is downloaded, we can simultaneously check it against each dictionary, which should make the overall process about n times faster.
- Space on github
  - Our json file was 40 Mb, and github won't store this unless we pay them to. So, zip the json file, making it only 9mb (25 mb is the max without paying). We now have to unzip it in our code, which luckily wasn't that hard (used python's zipfile library)

Open source part:

- People interested in uploading a virus easily create a pull request on our GitHub, and we will check the pull request and add the file to our database. To avoid cases of false positives, each file in our database has to be checked extensively to ensure that it's indeed a malicious file.