(**To graders**: This is an UNOFFICIAL version of the readme. Please make sure to read the Compass submissions INSTEAD OF this one)

Our project is a chrome plugin that scans every tab and decides whether it is a malicious webminer, a safe non-mining webpage, or a suspicious webpage using too much CPU.

Eventually we decided that users may not want our plugin to scan their webpages locallys, because it requires their Chrome to run in debugger mode. So we moved the scanner and analyzer to a remote server and decided the plugin will fetch analysis results from the server instead of doing analysis locally.

Modularization:

1. A program to scan webpages and determine whether they are miners
2. A frontend for chrome plugin.
3. A simple Node.js server that accepts requests from chrome plugins, send them to my algorithm, and return the result.

How to the code:
Our team didn't finish the plugin but the scanner is finished. It is a command-line program that tells whether a webpage is mining. Here is how:

1. Please use commit cea3d7901513da346424b304bcddee540876fef5
2. Please use node version v9.2.0.
3. Please install latest version of chrome on your machine.
4. Two ways to test my code:
    1. [Recommended] Go to 'scanner'. Run 'npm install'. Run 'node analyzer.js [url] [profiling duration in milliseconds, usually 5000]'. Example urls can be found in 'scanner/data-sources.sh'
        1. Output labels:
        2. Malicious = 1
        3. Suspicious = 2
        4. Safe = 3
    2. This one may not work because it communicates with a temporary server instead of directly communicating with my algorithm. Go to cacheServer, run "npm install". Run "node testClient.js [url]".

How the scanner was developed:

1. I decided to analyze webpages using a Profiler, which samples JS function calls and aggregate them into either a CPU timeline or total CPU usages.

2. I found the correct way to invoke the Profiling tools from a headless chrome using Node.JS.
    1. This was not easy. See below:
    2. Latest chromes come with two profilers in the developer tools window. One is "Javascript Profiler" tab. The other is "Performance" tab.
    3. I've tried using both tools to manually analyze webpages, and found analysis is more straightforward when using "Performance", because it provides a CPU timeline, and data are not compressed into a tree structure.
    4. However the "performance" tab is a new feature, and there are very few documents available about how to invoke it from Node using Chrome's remote debugging protocol.
    5. There is even less documentations about the meaning and structure of the output data.
    6. I had to read Chrome Developer Tools' frontend source code to understand how to analyze this profiler's output and how to aggregate it to show the timeline of JS functions running on different threads.
3. After this, I tried many ways to aggregate the data, and to classify webpages based on the aggregated result.
    1. Firstly, I found out that, if a webpage runs a lot of JS function on a window frame whose id is "None" or "undefined", then it is highly likely a crytominer.
    2. This is because None frames are the only places where CPU usages are seen above 100%. In other frames, there seems to be only one JS process so cryptominers cannot utilize dual core CPUs to full potential unless they use None frames.
    3. I used this to find obvious malicious pages (if they use more than 100% cpu on average). But if a page passes this check, it goes on to the next one:
    4. As a backup detection method, I aggregate the CPU usages by function names and urls instead, and mark webpages as suspicous as soon as they use more than a threshold of CPU on average.
    5. One may worry that webpages like Youtube will be flagged in this way but Chrome doesn't recording video decoders because they are not JS functions (on most machines they are natively implemented). So this turns out to be already very useful.
4. But I later found out that, aggregating CPU timelines into an average CPU percentage can cause more false alarms. Additionally, if I divide timeline by current CPU usage (which is also not aggregated), the result is more interesting.
5. So I removed some aggregations. In "scanner/Analysis3.ipynb" you can see a very preliminary version of the timeline (with bugs later fixed in "scanner/FinalAlgo.ipynb")

    1. The algorithm still looks the same. It's just using the entire timeline instead of an averaged CPU percentage.
    2. I still check for None frames first.
    3. Then check whether the webpage is using too much "relative" CPU for too long. (Instead of just checking the average "absolute" CPU percentage)
        1. Absolute CPU usage means the percentage of time spent on JS function divided by total time
        2. Relative CPU usage is Absolute CPU usage divided by current CPU usage of the chrome process. This is only achievable if I run only one webpage in an entire chrome process, which is guaranteed by using headless chromes.
        3. This is very useful on actual servers because the chrome process performance may change dramatically when multiplers are querying the server. Calculating relative cpu usages before aggregations, cancels it out.
    4. (New functionality) I check the urls of javascripts that used the CPU, and do not count its CPU usage if it's whitelisted on the server side.
    5. Eventually if the webpage is using any CPU on None frames or using too much CPU, I say the webpage is suspicious. If the webpage is using too much CPU on None frames, I say it's malicious. Otherwise, I say it's safe.
6. Then I translated "FinalAlgo.ipynb" to javascript so the Node.JS server can easily invoke it.
7. Eventually I ran my Node.JS program on profiling data from multiple machines and configurations. I tweaked parameters to make sure it gives stable outputs.
8. The eventual algorithm is more prone to generating false alarms than false negatives. The only cases it'll create false negatives are
    1. when the crytojacking program waits until the profiler stops running before mining. This happens because I ran analysis on the server side and I can only run for a short period of time. Otherwise the server can be jammed. It'd be better to run the profiler locally if weren't for privacy concerns.
    2. When the malicious webpage can only be seen if I have the same cookies and credentials as the user. Again, for privacy concerns, this is hard to implement. But the coding isn't difficult if I simply run the algorithm locally.
    3. When the server is heavily overloaded and chrome ran so slowly that we can't tell things apart anymore. This doesn't happen now because we controlled total number of profilers and chromes running in the system. And I've tested this on the actual server. It never happened after I used a reasonable threshold -- no more than 5 webpages at the same time, even if all 5 are cryptojacking.