Due date: Wednesday April 6th 11:59:59 PM ET

Points: This lab is worth 14 points (out of 100 points in 6.888).

Collaboration policy: Our full Academic Honesty policy can be found on the Course Information page of our website. As a reminder, **all 6.888 labs should be completed individually**. You may discuss the lab at a high level with a classmate, but you may not work on code together or share any of your code.

Getting started: You will conduct this lab assignment **on your own personal computer or laptop**. This attack should be micro-architecture agnostic. If you have difficulty getting access to an available device, please reach out to us.

We are using `git` for assignment version management and submission for this lab. Instructions for setting up git can be found on the course website. You will need to redo the first time setup from Lab 1 on your personal computer.

First time setup: Once you've logged into your computer, you will need to set up your Git repo. This needs to be redone as you are working on a different machine than the one we used for Lab 1. Once you've completed the instructions on the course website under "New Machine Setup," you can pull the lab 3 code with the following:

```
git fetch lab_release

git merge lab_release/lab3 --allow-unrelated-histories -m "Merging lab code"
```

Introduction

In this lab, you will implement side channel attacks from your web browser, using JavaScript. You will learn that attacks mounted from JavaScript are still effective even though the browser environment comes with several constraints, such as a coarse-grained timer, no access to `clflush`, and no effective way to manipulate addresses. You will implement the cache-occupancy side channel attack described in "*Robust Website Fingerprinting Through the Cache Occupancy Channel*" [1] or "*Prime+Probe 1, JavaScript 0: Overcoming Browser-based Side-Channel Defenses*" [2] to perform website fingerprinting attacks. You do not need to read these two papers, since we provide enough explanation of the papers and tips in the handout for you to complete the lab. If you get stuck, reading the papers might help you to figure out some implementation details.

As you will find, the attack developed in this lab works in any web browser, including Chrome, Firefox, Safari and even the secure Tor browser [3]. For parts 1-3, feel free to choose whichever browser you like. For part 4, you will have to use Chrome, because it requires the use of a Chrome extension.

Discussion Question 1: Report your browser and machine details (browser version, CPU type, cache sizes, memory, OS). We are gathering this information to test the attack's ability to work on various machine types. You can choose not to report this information if you are concerned about your privacy.

Website Fingerprinting: Website fingerprinting is a type of attack where an attacker tries to distinguish which website is visited by a victim. It makes it possible for an attacker to gather a lot of metadata regarding users. In the case of web browsing, simply knowing which website a user connects to is more than sufficient to obtain detailed information about a victim, such as political views, religious beliefs, and sexual orientation. There exist many variants of website fingerprinting attacks, which we can classify into two categories based on the resources that can be accessed by the attacker: *on-path attacks* and *co-located attacks*.

An on-path attacker executes on a different machine from the victim. The attacker observes all the network packets sent and received by the victim's machine and infers the website based on the timing and size of the observed network packets. A co-located attacker executes on the same machine as the victim and shares multiple micro-architectural resources with the victim, including caches, DRAM, and GPUs. In the case of a low-privileged attacker, the co-location can be achieved by observing the victim accessing a malicious website running attacker-controlled JavaScript code. We focus on co-located attacks in this lab.

Machine Learning in Side-channel Attacks: We will use machine learning techniques to analyze the collected traces. You are not required to understand the internal mechanisms of machine learning techniques in-depth. Instead, the goal is to allow you to use ML as a black-box tool for signal processing. You will only need to know how to use standard tools and python APIs to train models and perform website predictions.

1 Warmup (15%)

In part 1, you will familiarize yourself with the development environment and find the resolution of the timer offered by JavaScript.

Hello world: As a warm-up exercise, we will guide you to get familiar with the JavaScript development environment by writing a simple hello-world program.

We have provided the following files:

1. `warmup.js`: A JavaScript file with two functions, `measureOneLine` and `measureNLines`, which you will complete.
2. `warmup.html`: A webpage that displays the return values of the two functions.

If you're already familiar with JavaScript, feel free to skip to the subsection labeled "**Time measurement**" below. Otherwise, here we provide a brief overview of JavaScript and the developer tools you will need for this lab.

By including `<script src="warmup.js"></script>` on line 30 of `warmup.html`, when your browser loads `warmup.html`, it downloads `warmup.js` and evaluates the script's contents immediately. You can test this by adding a simple print function, such as `console.log("hello world")` You can then view the console by right clicking on the page and clicking "Inspect" (or Cmd-Opt-I on macOS, or F12 on Windows or Linux), and then selecting "Console" in the panel that opens up. In Safari, you'll need to first enable inspect mode with Safari > Preferences > Advanced > Show Develop menu in menu bar.

Exercise 1 (Optional): Add a `console.log` statement with a message of your choice, anywhere in `warmup.js`. Then, open `warmup.html` in your web browser and open the console to ensure your message is displayed. Feel free to skip this exercise if you are already familiar with web development.

You can use the above approach to debug your JavaScript code. JavaScript's basic syntax is fairly similar to other languages, such as C or Java. If you need to review JavaScript's syntax while completing this lab, feel free to refer to resources such as `https://learnxinyminutes.com/docs/javascript/`.

Time measurement: Before you can perform a timing side-channel attack, you need to figure out the quality of the timer, i.e. the resolution of the timer. In Lab 1 and Lab 2, we use the `rdtsc` instruction to measure the number of cycles, which is at the granularity of 0.5ns if assuming a 2GHz machine. However, the timer function `performance.now()` in JavaScript is much more coarse grained. Its resolution depends on which browser you're using. In Chrome, `performance.now()` yields a resolution of 0.1ms. In Firefox its resolution is 1ms. Function `measureOneLine()` gives an example of measuring the access latency of a single memory access. You should see the following output when you open the `warmup.html` in a browser. You may occasionally see a 1, but you may ignore these, as they would average out over many accesses.

```
6.888 Lab 3 Warmup
1 Cache Line: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
N Cache Lines: []
```

Your task is to figure out the time resolution of `performance.now()` by measuring the latency of accessing multiple cache lines. Please report the observed value for accessing N cache lines, where N ranges from 1 to $10,000,000$. You should perform the measurement multiple times and report the median access latency. Feel free to access each cache line sequentially – ignore the effects of the hardware prefetcher for this exercise.

Hint: The cache line size is usually 64 Bytes. If you are not sure, you can use `getconf -a | grep CACHE` if you are running Linux.

Exercise 2: Complete `measureNLines()` so that it measures the access time of N cache lines 10 times and pushes each measurement to the `result` array. These values will then get displayed on the webpage you opened earlier when you refresh it. Use these values to find the median access time of N cache lines, and report your results in Table 1.

Number of Cache Lines	Median Access Latency
1	
10	
100	
1,000	
10,000	
100,000	
1,000,000	
10,000,000	

Table 1: Median access latency reported by `performance.now()`.

Discussion Question 2: According to your measurement results, in order to measure differences in time with `performance.now()`, approximately how many cache accesses need to be performed?

Submission and Grading: There is no checkoff for Part 1. This part is graded manually based on the following submitted materials: code and a pdf file.

- Code. You will need to submit `part1/warmup.js` to your assigned Github repository. You should not modify other files.
- Discussion Questions. A pdf named as `solutions.pdf` at the root of your assigned Github repository. You will create a pdf file to provide your answers to the discussion questions above. Note that this pdf file will also be used by you to provide answers for the questions in the other parts.

2 Side Channel Attacks with JavaScript (45%)

In part 2, you will implement a "cache-occupancy attack". The main difference between a Prime+Probe attack and a cache-occupancy attack is that Prime+Probe measures contentions in specific cache sets, whereas cache-occupancy attack measures contention over the whole cache. Specifically, you will write JavaScript code to allocate an LLC-sized buffer and measure the time to access the entire buffer. The victim's access to memory evicts the contents of your buffer from the cache, introducing delays for your access. Thus, the time to access the buffer is roughly proportional to the number of cache lines that the victim uses.

You will first implement the cache trace collection functionality in JavaScript, then collect traces for different websites using the provided automation script. Finally, you will use machine learning to train a model that classifies websites based on the collected cache-occupancy traces.

2.1 Cache Trace Collection

You will implement your trace collection function in the attack infrastructure as shown in Figure 1. In this attack, the victim code and the attacker code reside in two separate JavaScript environments. They can

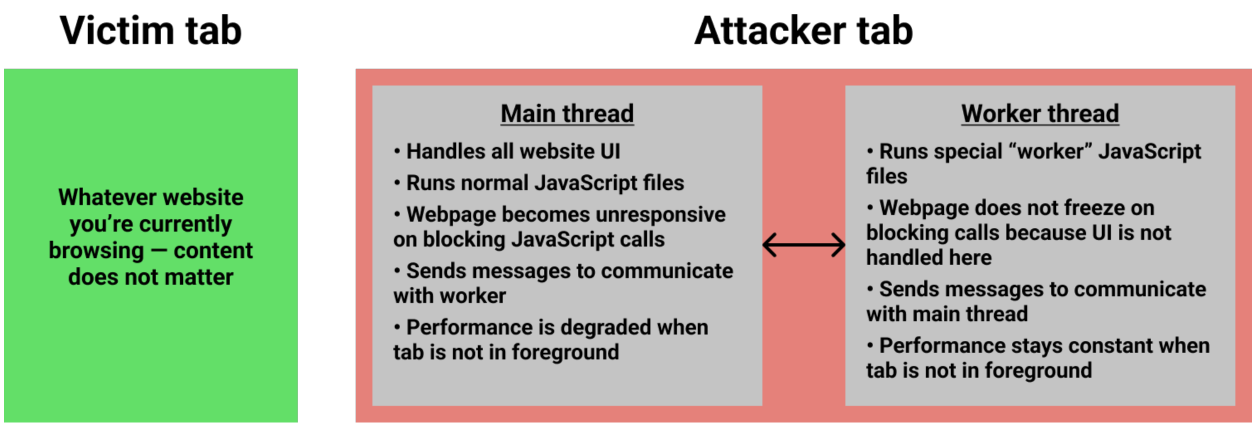
Victim tab
Whatever website you're currently browsing — content does not matter
Attacker tab
Main thread
• Handles all website UI
• Runs normal JavaScript files
• Webpage becomes unresponsive on blocking JavaScript calls
• Sends messages to communicate with worker
• Performance is degraded when tab is not in foreground
Worker thread
• Runs special "worker" JavaScript files
• Webpage does not freeze on blocking calls because UI is not handled here
• Sends messages to communicate with main thread
• Performance stays constant when tab is not in foreground

Figure 1: Attack infrastructure overview.

be in two different tabs within the same browser, or entirely separate web browsers on the same machine. The attacker tab will create two threads, a main thread that handles website user interactions (e.g., clicking buttons) and a worker thread that executes your provided code in the background without blocking user interactions. Note that, the worker thread runs even if the corresponding tab is not in foreground.

Setting Up Web Server: Modern web browsers require that websites be hosted on a web server in order to use the worker thread. Simply opening `index.html` as a file will not work, and a security warning will be displayed in the browser's console. To get around this issue, you can develop your code by running a simple web server using the following commands. *Note:* Make sure you're using ***Python 3*** for this step and for the rest of the lab.

```
$ cd part2
$ python -m http.server
Serving HTTP on :: port 8000 (http://[::]:8000/) ...
```

Web browsers typically cache the service worker upon loading the page, which means you will need to change your browser's settings in order to load updates you make to `worker.js`. Follow the instructions at Force update a service worker on Chrome, Firefox and Safari in order to do this. If this doesn't work for you, you can force a refresh by opening your worker script at `http://localhost:8000/worker.js`, holding down shift while clicking the refresh button in your browser's toolbar, and manually verifying that the file's contents match what you expect.

Trace Collection: You can now open `http://localhost:8000` in your preferred web browser, and you will see two buttons. Clicking the button labeled "Collect trace" will begin a countdown, which you can use to prepare your experiment (e.g. switching to a new window), and then trigger the `record()` function in `worker.js`, which will be completed by you. The output of this function is displayed as a heatmap for convenience. You can click this button multiple times without refreshing the window in order to collect multiple traces. Clicking the button labeled "Download traces" will allow you to download all the traces that have been collected in a JSON format.

To implement the cache-occupancy attack in the `record()` function, you may choose between one of the three approaches below, or come up with your own. In any of the cases, be careful with the cache line size.

(1) Measure the latency of accessing N addresses, such that the N addresses span your entire last-level cache. Repeat the measurement K times to form a trace.

(2) Slightly modify the previous approach to better synchronize traces. Divide your trace into a sequence of short intervals of time such that each element corresponds with activity recorded over an interval length P, on the order of a few milliseconds. Within each interval, measure the latency of accessing

N addresses. If the latency is smaller than the interval length P, record it. Otherwise, if the latency is longer than P, it means one or multiple intervals have been missed, then mark the missed intervals with a special symbol, e.g., `-1`. Repeat the latency measurement multiple times to form a trace for K intervals. This is similar to the approach used in [1] and they set the value of P to be 2ms. Feel free to experiment with different P values.

(3) Divide your trace into a sequence of short intervals of time such that each element corresponds with activity recorded over an interval length P, on the order of a few milliseconds. Within each interval, repeatedly access all N addresses and count the number of times that you can traverse the full buffer within the time interval P. This count represents the throughput of memory accesses for the given interval. In this case, you may want to set P to be slightly larger than the value of P in approach (2). This is the approach used in [2].

Trace Processing: You can later process these traces in Python, with code such as the following:

```python
import json
import numpy as np

with open("traces.json", "r") as f:
    # Load contents from file as JSON data
    data = json.loads(f.read())

    # Convert 2D array into Numpy for data processing
    traces = np.array(data["traces"])

    # Labels are only available with the automation script.
    # Use the line below in part 2.2 onward to access them.
    # labels = data["labels"]

# Example data analysis
print(traces.mean())
```

Such traces can be used to distinguish different system events. Figure 2 shows three traces which were collected under the following circumstances:

(1) Do nothing while trace is collected;

(2) Add random system activity; move the mouse during trace collection;

(3) Open a website in a new window during trace collection.

Discussion Question 3: Briefly describe your trace collection method and important parameters, such as number of addresses being accessed (N) and trace length (K).

Exercise 3: Complete the `record()` function in `worker.js`. Collect traces for the three scenarios described above and take a screenshot of the generated 3 traces. Note, your traces may not exactly match the examples in Figure 2, but they should be visually distinguishable.

2.2 Automated Attacks with Machine Learning

It is tedious to launch the victim websites manually. To automate the attack process, we wrote an automation script based on the Selenium browser automation framework [4] for you to use. The script is located in the root directory of your GitHub repository and it will be used in the remaining parts of the lab.

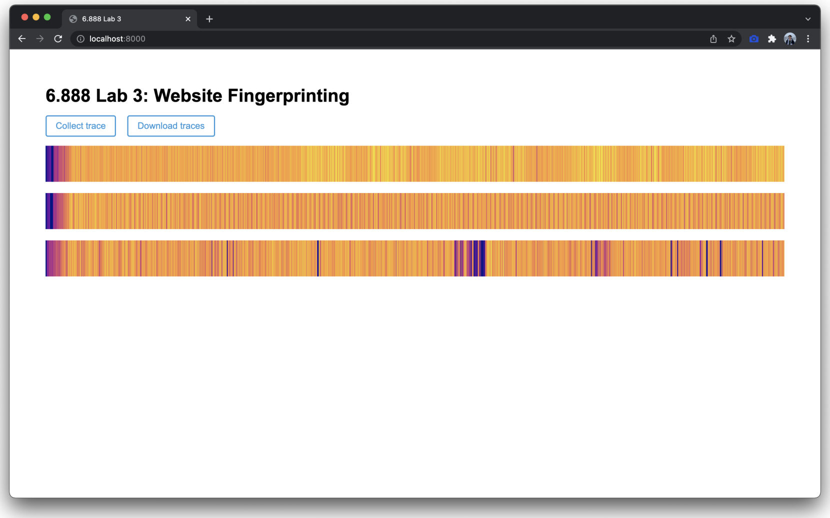

Figure 2: Example traces.

Install drivers: Before completing this section, you will need to install Flask and selenium. Make sure you're using Python 3, and then install these modules with

```
pip install flask selenium
```

You may additionally need `scikit-learn` in order to complete parts of this lab.

```
pip install scikit-learn
```

You should then download the latest driver for your browser and add it to your path. We recommend you use the driver download links at the top of Selenium: Install browser drivers, and then additionally follow option 2 under the "Three Ways to Use Drivers" header.

If you're using Chrome, you may check your Chrome version by opening `chrome://version`, as you will need this when you click on the Chrome download link at the link above. If you're using Firefox on a Linux machine, you may need to untar your driver download, which you can do with `tar -xzf filename.tar.gz`.

Use the automation script: You can test the automation script by collecting a few traces while your victim opens different websites using the following commands.

```
$ python automate.py
      --part 2
      --domains google.com,nytimes.com
      --num_traces_per_domain 4
      --out_filename traces.out
```

You can get the detailed descriptions of the arguments used by the automation script by executing `python automate.py --help`.

```
usage: automate.py --part {2,3,4} --out_filename OUT_FILENAME
                   [--browser {chrome,firefox,safari}]
                   [--domains DOMAINS]
                   [--enable_countermeasure ENABLE_COUNTERMEASURE]
                   [--num_traces_per_domain NUM_TRACES_PER_DOMAIN]
                   [--trace_length TRACE_LENGTH]

required arguments:
  --out_filename OUT_FILENAME
                        Name of the output file to save traces to.
  --part {2,3,4}        Set to the part of the lab you're working on.

optional arguments:
  --browser {chrome,firefox,safari}
                        Browser to run automation in.
  --domains DOMAINS     Comma-separated list of domain names to collect traces
                        from. Defaults to
                        google.com,youtube.com,baidu.com,facebook.com
  --enable_countermeasure ENABLE_COUNTERMEASURE
                        Set to true to enable the countermeasure. Browser must
                        be set to Chrome. Defaults to false.
  --num_traces_per_domain NUM_TRACES_PER_DOMAIN
                        Number of traces to collect per domain.
  --trace_length TRACE_LENGTH
                        The length of each recorded trace, in milliseconds.
                        Defaults to 5000.
```

We recommend starting with a few traces from `google.com`, and a few traces from `nytimes.com`. The reason is that `google.com` is a lightweight website with mostly static content, while `nytimes.com` is a heavy-weight website that loads many assets, making them easy to distinguish. Feel free to substitute these for another pair of lightweight and heavy websites as desired.

Exercise 4: Use the Python code from Section 2.1 to analyze simple statistics (mean, median, etc.) on the traces from `google.com` and `nytimes.com`, or another two sites of your choosing.

Use Machine Learning for Classification: Now, we will create a more sophisticated attacker. Instead of collecting 4 traces on 2 websites, we're going to collect 20 traces on 4 different websites. When each trace is 5 seconds long, this will take about $5 * 20 * 4 = 400$ seconds, or about 7 minutes to run. Pick four of your favorite websites to classify between, pass them to the domains argument, and leave your computer alone for a few minutes. If you're on macOS, you can have your computer alert you when it's done by reading some text out loud with the `say` command or something similar:

```
$ python automate.py
      --part 2
      --domains website1.com,website2.com,website3.com,website4.com
      --num_traces_per_domain 20
      --out_filename traces.out
  && say "I'm done collecting traces"
```

Once the script has finished, you should divide your traces into a *training set* with 16 traces from each site, and a *testing set* with 4 traces from each site. The training set is used to train a machine learning model, and the testing set is used to evaluate its accuracy once training is complete. We recommend using the `train_test_split` function from the `scikit-learn` library, with `test_size=0.2`. Then, train a `RandomForestClassifier` or another classification model of your choice, also available from the `scikit-learn` library, on your training set from earlier. Finally, use your model to predict labels for the testing set, and check your model's accuracy with `scikit-learn`'s `classification_report` function. An example classification report is shown below with the top four websites in the world as of March 9, 2022, according to Alexa.

```
                          precision    recall  f1-score   support

   https://www.baidu.com       1.00      1.00      1.00         4
  https://www.google.com       1.00      1.00      1.00         4
https://www.facebook.com       1.00      0.75      0.86         4
 https://www.youtube.com       0.80      1.00      0.89         4

                accuracy                           0.94        16
               macro avg       0.95      0.94      0.94        16
            weighted avg       0.95      0.94      0.94        16
```

Exercise 5: Complete the `eval()` function in `eval.py`. In this function, you should (1) Load your traces into memory, (2) Split your data into a training and testing set, (3) Train a classification model on your training set, (4) Use your model to predict labels for your test set, and (5) Print out the model's accuracy with `classification_report`. Copy and paste the classification report into a file, `accuracy.txt`, and include it in the repo along with your traces file.

Exercise 6 (Optional): Try a different machine learning model to see whether you can improve on the accuracy in the previous exercise.

Submission and Grading: There is no checkoff for Part 2. You need to submit your code (`part2/worker.js`), traces (`part2/traces.out`), and classification report (`part2/accuracy.txt`), to the GitHub repository. Anything higher than 80% accuracy (the number on the "accuracy" line in the classification report) will receive full credit. However, you should easily be able to achieve upwards of 95% accuracy given the simplicity of this task. Note that Exercise 6 is optional.

3 Root Cause Analysis (10%)

Machine Learning-based side channel attacks are generally powerful since it is capable to find correlations across traces and can tolerate medium to heavy noise. However, one key problem is that, we are unable to analyze the root cause of the attack. In Lab 1 and Lab 2, we know for sure the attack exploit cache side channels, because we carefully design the attack, manipulate memory access patterns, and rely on direct correlation between the observed reslts and the secret to decode secrets. However, in this lab, given that the JavaScript is a high-level language, we do not have full control of the instructions being executed on the processor and we leverage Machine learning techniques which do not tell us where the signal actually comes from.

In this part, you will try a slightly modified version of the attack and you will have a better understanding of the pros and cons of ML-driven attacks. Remove the memory accesses in your code, collect new data, and repeat exercise 5.

Exercise 7: Try the modified attack, repeat the training and inference operations in part 2, and report the accuracy.

Discussion Question 4: Compare the accuracy numbers you get for part 2 and part 3. Does the accuracy increase or decrease. Do you think the "cache-occupancy" attack actually exploits a cache side channel or not? If not, take a guess of possible root cause of the modified attack.

Note: Without further investigation, you may be unable to verify your answers. We will give full credit as long as the reasoning behind your guess is logical. If you are really curious, we recommend reading this paper: There's Always a Bigger Fish: A Case Study of a Misunderstood Timing Side Channel.

Submission and Grading: You need to submit your code (`part3/worker.js`), traces (`part3/traces.out`), and classification report (`part3/accuracy.txt`), to the GitHub repository. Anything higher than 80% accuracy (the number on the "accuracy" line in the classification report) will receive full credit. However, you should again easily be able to achieve upwards of 95% accuracy given the simplicity of this task.

4 You Can Bypass Mitigations (30%)

In this section, you will explore one proposed mitigation to such an attack. The mitigation is relatively simple – users can install a browser extension that adds noise to the system, making it more difficult for the attacker to recover a signal. You can run an experiment with this mitigation enabled by setting `enable_countermeasure` to true. For this part, you must use Chrome. For example:

```
$ python automate.py
      --part 4
      --domains website1.com,website2.com,website3.com,website4.com
      --num_traces_per_domain 20
      --out_filename part4/traces.out
      --enable_countermeasure true
```

Run this experiment with the same four websites you've been using, and repeat the steps you took in part 2.

Exercise 8: Run the attack with the countermeasure enabled, repeat the training and inference operations from part 2, and report your accuracy.

Discussion Question 5: How much did the accuracy decrease? Is noise injection an effective mitigation?

Finally, see how much you can do to defeat this countermeasure.

Exercise 9: Propose a change to your attacker. You may consider changing your attacker's algorithm, or a variable passed to the automation script, such as the trace length. Explain why your method should be effective in theory, and then evaluate it by repeating the steps from part 2 with the countermeasure enabled. Did your accuracy increase?

Submission and Grading: Submit your traces (`part4/traces.out`) and classification report (`part4/accuracy.txt`) for exercise 8 to the GitHub repository. Additionally submit your updated attacker if you made any changes (`part4/worker.js`) along with new traces (`part4/traces2.out`) and a new classification report (`part4/accuracy2.txt`) for exercise 9. If you did not modify the attacker, be prepared to explain what other changes you made to defeat the countermeasure. Full credit for exercise 8 will be given regardless of your accuracy. Full credit for exercise 9 will be awarded if you increase accuracy from exercise 8 by at least 10%, or if your accuracy from exercise 8 is already above 80%.

Acknowledgments

Contributors: Jack Cook, Mengjia Yan.

References

[1] Anatoly Shusterman et al. "Robust Website Fingerprinting Through the Cache Occupancy Channel". In: 2019.

[2] Anatoly Shusterman et al. "Prime+ Probe 1, JavaScript 0: Overcoming Browser-based Side-Channel Defenses". In: *30th USENIX Security Symposium (USENIX Security 21)*. 2021, pp. 2863–2880.

[3] The Tor Project Inc. *Tor Browser*. https://www.torproject.org/. Accessed on 08.13.2021.

[4] Gunes Acar, Marc Juarez, and individual contributors. *tor-browser-selenium - Tor Browser automation with Selenium*. https://github.com/webfp/tor-browser-selenium. 2020.