Getting Started

This lab is done **on your own computer**, and should be (micro)architecture agnostic. If you have difficulty with accessing a device, please reach out to the Lab Instructor.

Introduction

In this lab, you will launch side-channel attacks from your web browser, using JavaScript. You will demonstrate that attacks mounted from restricted execution environments are still effective, despite having no browser access to fine-grained timers, cache flushing instructions, or low-level memory manipulation. You will implement the cache-occupancy side channel attack as originally described in Robust Website Fingerprinting Through the Cache Occupancy Channel or Prime+Probe 1, JavaScript 0: Overcoming Browser-based Side-Channel Defenses to perform a *website fingerprinting attack*. Reading these two papers is not required to complete the lab, however they may help you if you get stuck. The papers are available on the moodle website of the lab under background material for lab meeting #1.

The attack developed in this lab works in any web browser, including Chrome, Firefox, Safari, and even the Tor browser. You can use any browser for Parts 1-3, but for Part 4 you will need to use Chrome as it requires a Chrome extension.

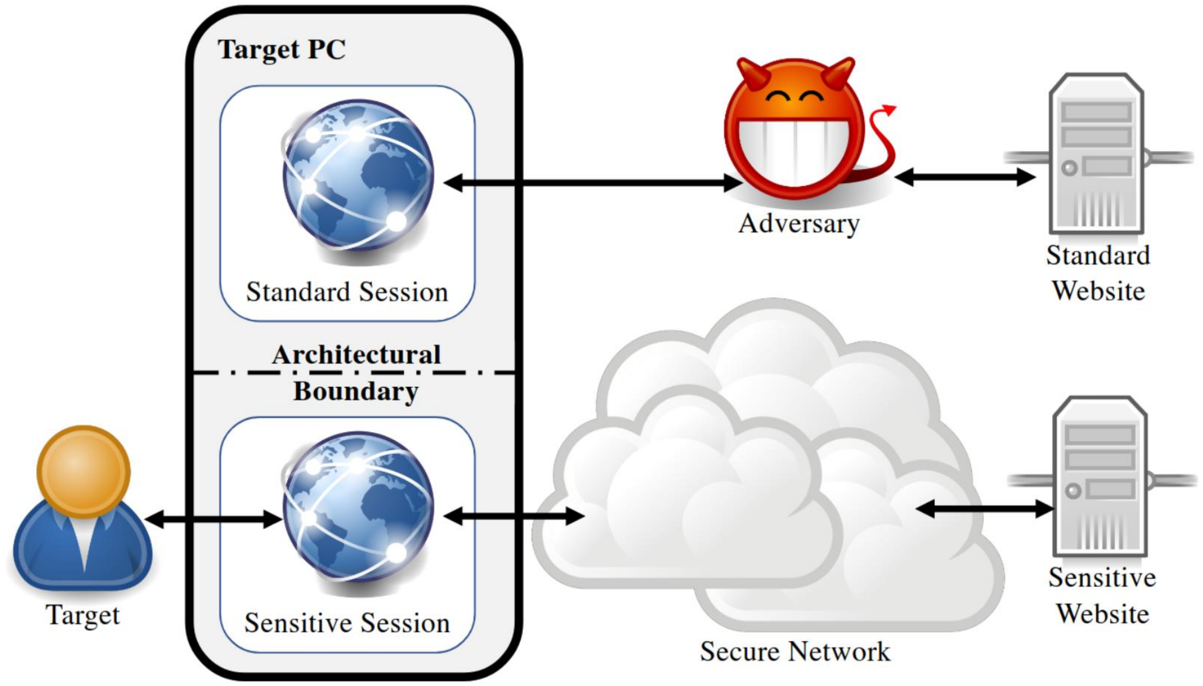
EXERCISE #1
Report your browser version, CPU type, cache size, RAM amount, and OS.

Website Fingerprinting

In a website fingerprinting attack, an attacker tries to distinguish which website a victim accesses. Website fingerprinting attacks provide an opportunity for an attacker to gather a lot of user information, such as political views, religious belies, and sexual orientation. There exists 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 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.

In contrast, a *co-located* attacker executes on the same machine and shares microarchitectural resources with the victim, including caches, DRAM, and GPUs. In the case of a low-privileged attacker, this co-location can be achieved by running attacker-controlled JavaScript code in a different browser tab.



We focus on co-located attacks in this lab.

Machine Learning in Side-Channel Attacks

We will use machine learning techniques to analyze traces we collect. Knowledge of the internal machine learning techniques and mechanisms is not required, instead, the goal is to allow you to use ML as a black-box tool.

Part 1: Warm-up (15%)

In this part, you will familiarize yourself with the development environment, and determine the timer resolution offered by JavaScript.

Hello World

As a warm-up exercise, we will guide you through JavaScript development by writing a simple Hello World program. We have provided the following files:

- `warmup.js`: A JavaScript file with two functions, `measureOneLine` and `measureNLines`, which you will complete.
- `warmup.html`: A webpage that displays the return values of the two functions.
  If you're familiar with JavaScript, feel free to skip to **Timing Measurement**. Otherwise, here we provide a brief overview of JavaScript and the developer tools you'll need for this lab.

By including `<script src="warmup.js"></script>` on line 30 of `warmup.html`, your browser downloads, runs, and executes the script's contents immediately upon loading `warmup.html`. 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 the page and selecting `Inspect` (Cmd-Opt-I on MacOS, or F12 on Windows/Linux), and then select `Console`. In Safari, you may need to unlock inspect mode with `Safari > Preferences > Advanced > Show Develop Menu`.

EXERCISE #2
Add a `console.log` statement with a message of your choice, anywhere in `warmup.js`. Then, open `warmup.html` in your web browser and check the console to ensure that your message is displayed.

You can use `console.log` to debug your JavaScript code. JavaScript's basic syntax is fairly similar to other languages you might be familiar with, such as C or Java. If you need to review JavaScript's syntax while completing this lab, feel free to refer to various online resources.

Timing Measurement

Before we can execute a timing side channel attack, we need to determine the quality (i.e. *resolution*) of the timer. In later labs we'll use the `rdtscp` instruction in C to measure the number of cycles elapsed (at the granularity of 0.5ns assuming a 2GHz machine). JavaScript does not offer such a fine-grained timer, however. The JavaScript timer `performance.now()` yields different resolutions depending on the browser – in our experiments the resolution is 0.1 ms in Chrome, and 1ms in Firefox. In `warmup.js` we provide `measureOneLine()`, which is an example of how we can measure the access latency of a single memory access using `performance.now()`. You should see the following output when you open `warmup.html` in a browser (you may occasionally see some non-zero entries).

```
Warm-Up

1 Cache Line: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
N Cache Lines: []
```

Your first task is to determine the timing resolution of `performance.now()` by measuring the latency of accessing multiple cache lines. Report the observed value for accessing *N* cache lines, where *N* ranges from 1 to 10,000,000. Perform the measurement multiple times and report the median access latency. Feel free to access the memory in sequential order, you can ignore any potential effects of hardware prefetching.

Hint: The cache line size of your machine is likely 64 Bytes. If you are not sure, you can use `getconf -a | grep CACHE` if you are running Linux. Once you figure out the cache line size, you will want to access an array with a specified stride to make sure each access targets a different cache line.

EXERCISE #1.3B

Complete `measureNLines()` such that it measures the access time of *N* cache lines 10 times and pushes each measurement to the end of the `result` array. These values will be displayed on `warmup.html` when you refresh the page. Use these values to find the median access time of *N* cache lines, and report your results as follows:

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

DISCUSSION QUESTION #1.3B

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

For Part 1, you will need to submit two files:

- **Code.** You will need to submit `part1/warmup.js` You should not modify any other files.
- **Discussion Questions.** Submit a PDF in moodle. You are free to format this document as you wish (using LaTeX, Word, etc.).

Part 2: Side Channel Attacks with JavaScript (45%)

In Part 2, you will implement a cache-occupancy attack, which measures contention within your computer's cache. Specifically, you will write JavaScript code to allocate a Last Level Cache (LLC) sized buffer, and then measure how frequently you can access the whole buffer in a given amount of time. The victim's accesses to memory will evict the contents of your buffer from the cache, and introduce delays for your own accesses. Thus, the number of times you can access the buffer in a given amount of time is roughly (inversely) proportional to the number of cache lines the victim accesses.

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

Part 2.1: Cache Trace Collection + Processing

You will implement trace collection functionality as shown below. In this attack, the victim code and attacker code resides in two separate JavaScript environments. They can be within two different browser tabs, or entirely separate web browsers on the same machine. The attacker tab will create two threads: a main thread that handles user interactions (e.g. clicking website buttons) and a worker thread that executes your provided code in the background. Note that the worker thread runs even if the attacker tab is not in the foreground.

Setting Up The Web Server

To run the worker thread, modern web browsers require that you load the page from a server (rather than simply opening `index.html` as a file). To get around this issue, you can develop your code by running a simple web server using the following commands, run from the `part2/` folder. Make sure you're using Python3 for this step (and the rest of the lab).

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

Web browsers typically cache the worker thread upon loading the page, so you will need to change your browser's settings to load updates you make to `worker.js`. Follow the instructions here in order to do this. If this doesn't work for you, you can force a refresh of the service worker 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 checking that the file's contents match what you expect.

Trace Collection

Open http://localhost:8000 in your preferred web browser. Pressing `Collect Trace` will begin a countdown, which you can use to prepare your experiment (i.e. switching to a new window). At the end of the countdown, the worker will trigger `record()`, which will be written by you in `worker.js`. 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 `Download Traces` will allow you to download all of the traces collected in a JSON format.

To implement the cache-occupancy attack in the `record()` function, **we recommend doing the following**. Divide your trace into a sequence of short intervals of time such that each element corresponds with activity recorded over an interval of length P (on the order of a few milliseconds). Within each interval, repeatedly access all N addresses (where you choose N to cover your entire last level cache, depending on your system configuration), 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. This is the approach used in Prime+Probe 1, JavaScript 0: Overcoming Browser-based Side-Channel Defenses. Feel free to experiment with different P values.

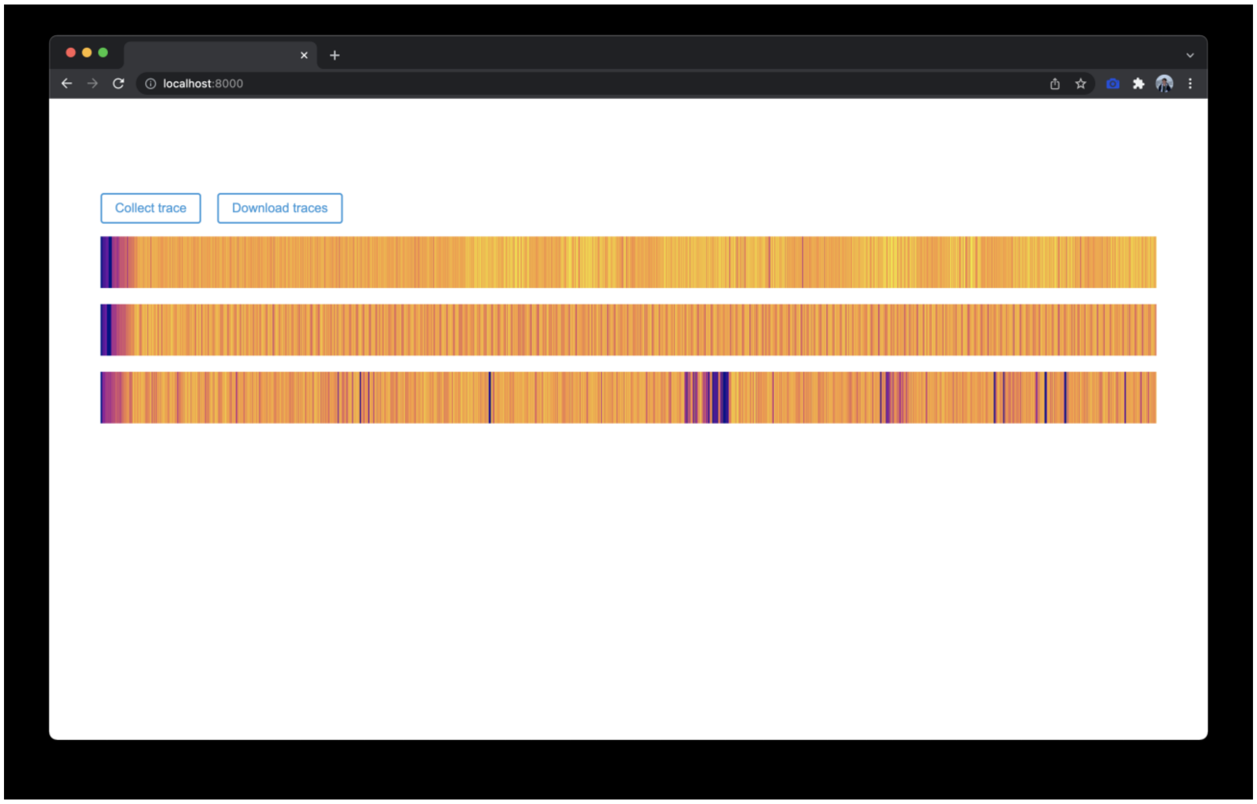
Trace Processing

You can process these downloaded 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. The below image shows three traces which were collected under the following circumstances:

- Do nothing while the trace is collected
- Add random system activity, moving the mouse during trace collection
- Open nytimes.com in a new window during trace collection



DISCUSSION QUESTION #2.1

Describe your trace collection method and important parameters, such as the number of addresses being accessed N and trace length K.

EXERCISE #2.2

Complete the `record()` function in `worker.js`. Collect traces for the three scenarios described above, and take a screenshot of these three traces. Your traces will not exactly match those in the provided example, but they should be visually distinguishable from one another.

Part 2.2: Automated Attacks with Machine Learning

It is tedious and unreliable to launch the victim website manually. To automate the attack process, we provide an automation script (`automate.py`) based on the Selenium browser automation framework for you to use.

Installing Drivers

To complete this section, you will need to install Flask, Selenium, and SciKit-Learn. Making sure you are using Python 3, install these modules with `pip install flask selenium scikit-learn`.

> Selenium should automatically install the latest drivers for the browser(s) you have installed. If you are encountering running Selenium, try manually installing the driver using option 3.

Using 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
```

Detailed descriptions of the arguments used by the automation script can be found by executing `python automate.py --help`:

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

optional arguments:
  -h, --help            show this help message and exit
  --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.

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.
```

We recommend starting with a few traces from google.com and nytimes.com. Google is a lightweight website with mostly static content, while NYTimes is a heavyweight website that loads many assets, making them easy to distinguish.

EXERCISE #2.3
Use the Python code we provided in Part 2.1 to analyze simple statistics (mean, median, etc.) on the traces from google.com and nytimes.com.

Using Machine Learning for Classification

Let's now design a sophisticated attacker. Instead of collecting four traces on two websites, we're going to collect 20 traces on four different websites. As we're collecting five second traces, this will take about 7 minutes to run. Pick four of your favourite websites to classify between, pass them to the domains argument, and leave your computer alone until it's done.

```
$ python automate.py
    --part 2
    --domains website1.com,website2.com,website3.com,website4.com
    --num_traces_per_domain 20
    --out_filename traces.out
```

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 from scikit-learn) on your training set. 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.

```
                        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 #2.4

Complete the `eval()` function in eval.py. In this function you should:

- Load your traces into memory
- Split your data into a training and test set
- Train a classification model on your training set
- Use your model to predict labels for your test set
- Print out your model's accuracy using `classification_report`
  Include your classification report in your solutions pdf.

EXERCISE #2.5 - BONUS

Try different machine learning models to see whether you can improve on the accuracy of your previous scheme.

Submission and Grading

You neet to submit your the code you changed (mainly `part2/worker.js` and `part2/eval.py`) and traces (`part2/traces.out`) , and your classification reports. Anything higher than 80% accuracy (i.e. the accuracy f1-score reported in the classification report) will recieve full credit, however you should easily be able to achieve 95% accuracy.

Part 3: Root Cause Analysis (10%)

Machine learning side channel attacks are very powerful as they are able to find correlations across traces, and can tolerate medium to heavy amounts of noise. A key challenge with using machine learning, however, is that it doesn't provide us insight as to *why* an attack works. Given that JavaScript is a high-level language we do not have full control or knowledge of the instructions being executed on the processor, nor do we have a good idea of where our signal is actually coming from!

In this part you will try a slightly modified attack to learn about the pros and cons of ML-driven attacks.

EXERCISE #3.1

Modify `record()` by removing all memory accesses in your code, and re-collect the traces for the four sites you previously examined. With your new traces, repeat Exercise 2-4 and report the accuracy.

DISCUSSION QUESTION #3.2

Compare your accuracy numbers between Part 2 and 3. Does the accuracy decrease in Part 3? Do you think that our "cache-occupancy" attack actually exploits a cache side-channel? If not, take a guess as to possible root causes of the modified attack.

Note: Without detailed investigation, you will not be able to verify your answer to this question. We will give full credit as long as the reasoning behind your guess is logical.

If you're curious as to the reasons why, we recommend reading the paper There's Always a Bigger Fish: A Case Study of a Misunderstood Timing Side Channel. The paper is available on the moodle website under further reading.

Submission and Grading

You neet to submit your code (`part3/worker.js`) and traces (`part3/traces.out`) and your classification reports. Anything higher than 80% accuracy (i.e. the accuracy f1-score reported in the classification report) will recieve full credit, however you should easily be able to achieve 95% accuracy.

Part 4: Bypassing Mitigations (30%)

In this section, you will examine a proposed solution to such an attack. To protect against this attack, users could install a browser extension that adds noise to the system, making it more difficult for the attacker to recover the original signal. You can run an experiment with this mitigation enabled by setting `enable_countermeasure` to `true`.

```
$ 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
```

EXERCISE #4.1

Using your code from Part 2, run the attack with the countermeasure enabled (using the same four websites you previously used). Repeat the evaluation you performed in Exercise 2-4, reporting your accuracy.

DISCUSSION QUESTION #4.2

How much did the accuracy decrease compared to Part 2? Is noise injection an effective mitigation?

Finally, let's see how much you can do to defeat this countermeasure!

DISCUSSION QUESTION #4.3

Propose a change to the attacker - you may consider changing the attacker's algorithm or variable passed to the automation script (such as the trace length). Explain why your method should be effective in theory.

EXERCISE #4.4

Evaluate your proposed change by repeating the steps from Exercise 4-1. Did your accuracy increase?

Submission and Grading

Submit your traces (`part4/traces.out`) for Exercise 4.1 along with the classification report in moodle. If you made any changes to the attacker, submit your updated `part4/worker.js`, along with new traces (`part4/traces2.out`) and classification report for Exercise 4.2. If you did not modify the attacker, explain what other changes you made to defeat the countermeasure. Full credit for Exercise 4.1 will be given regardless of your accuracy. Full credit for Exercise 4.2 will be awarded if your accuracy increase by at least 10% (compared to Exercise 4.1), or if your accuracy from Exercise 4.1 is already above 80%.