CSE 406: Computer Security – Assignment 2 Report
Website Fingerprinting via Side-Channel Attack

Kowshik Saha
2005006

June 20, 2025

1 Introduction

This report documents the implementation and findings for Assignment 2 of CSE 406: Computer Security at Bangladesh University of Engineering and Technology. The assignment focuses on a side-channel attack to perform website fingerprinting by analyzing cache usage patterns through JavaScript-based measurements. Four main tasks were completed: timing measurements, trace collection using the Sweep Counting Attack, automated data collection with Selenium, and website classification using a neural network. Additionally, two bonus tasks were implemented: collaborative dataset collection and real-time website detection. The report summarizes the methodology, results, and analysis as per the assignment requirements.

2 Task 1: Warming Up with Timing

2.1 Objective

Measure the precision of JavaScript's `performance.now()` function to estimate cache access latency for varying numbers of cache lines ($n = 1$ to $10,000,000$).

2.2 Implementation

The `readNlines(n)` function in `static/warmup.js` was implemented to:

- Allocate a buffer of size $n \times$ LINESIZE (LINESIZE = 64 bytes, determined via `getconf -a – grep CACHE`).
- Read the buffer at intervals of LINESIZE to access different cache lines.
- Perform 10 iterations and measure time using `performance.now()`.

- Return the median access time.

The function was called in a worker thread for $n = 1, 10, 100, \ldots, 10,000,000$, and results were displayed in a table in `static/index.html`.

2.3 Results

N (Cache Lines)	Median Access Latency (ms)
1	0.00
10	0.00
100	0.00
1,000	0.10
10,000	0.65
100,000	2.55
1,000,000	25.45
10,000,000	251.95

Table 1: Latency Measurement Results

2.4 Observations

The resolution of `performance.now()` was estimated to be approximately 0.05 ms. At least 1000 cache accesses were needed to measure reliable time differences. Latency increased rapidly from n = 1,000 to 10,000 and then almost linearly from n = 10,000 to 10,000,000.

3 Task 2: Trace Collection with the Sweep Counting Attack

3.1 Objective

Implement the Sweep Counting Attack to collect cache access traces and visualize them as heatmaps.

3.2 Implementation

The `sweep(P)` function in `static/worker.js` was implemented to:

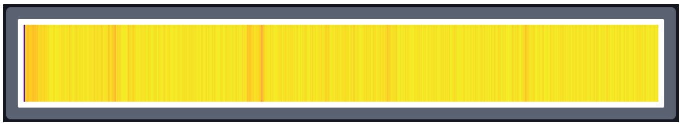
- Allocate a buffer of size LLCSIZE (16 MB, determined via `getconf -a — grep CACHE`).
- Count sweeps through the buffer at LINESIZE intervals within P ms windows.
- Collect counts for 10 seconds ($K = \text{TIME}/P$).

A suitable $P = 10$ ms was chosen based on Task 1 experiments. The UI was updated in `static/index.html` with "Collect Trace," "Download Traces," and "Clear Results" buttons. The `collectTraceData()` function in `index.js` handled worker communication, and the Flask endpoints `/collect_trace` and `/clear_results` in `app.py` processed and stored trace data, generating heatmaps using matplotlib.

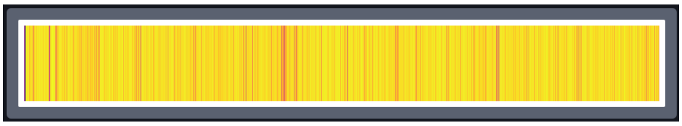
3.3 Results

Distinct heatmap patterns were observed for:

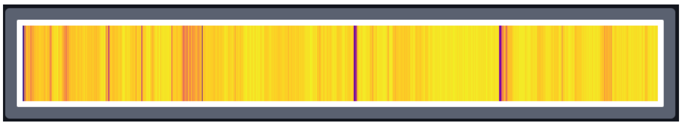
- **Idle browser:** Uniform low counts.



- **YouTube:** High variability.



- **Gmail:** Periodic spikes.



Website	Min. Sweep Count	Max. Sweep Count	Range	Samples
Idle	58	87	29	1000
YouTube	51	85	34	1000
Gmail	57	87	30	1000

Table 2: Trace Statistics

3.4 Observations

The attack successfully captured cache usage patterns, with dynamic websites like YouTube showing more variable traces. The chosen $P = 10$ ms provided sufficient resolution for distinguishing websites.

4 Task 3: Automated Data Collection

4.1 Objective

Automate trace collection using Selenium and store data in a SQLite database.

4.2 Implementation

The `collect.py` script was completed to:

- Start the Flask server and open the fingerprinting page.
- Open target websites (`cse.buet.ac.bd/moodle`, `google.com`, `prothomalo.com`) in a new tab.
- Simulate random scrolling to mimic user activity.
- Collect and store traces in `webfingerprint.db` using `database.py`.

1000 traces per website were collected, with robust error handling to prevent crashes.

4.3 Results

Website	Number of Traces
cse.buet.ac.bd/moodle	1000
google.com	1000
prothomalo.com	1000

Table 3: Collected Traces

4.4 Observations

Automation was reliable, with traces stored persistently in the db.

5 Task 4: Machine Learning for Website Classification

5.1 Objective

Train a neural network to classify websites based on cache traces, achieving at least 60% accuracy.

5.2 Implementation

The `train.py` script was completed to:

- Load and preprocess trace data from `dataset.json`, applying standard scaling.
- Split data into 80% training and 20% testing sets.

- Train `FingerprintClassifier` and `ComplexFingerprintClassifier` using PyTorch.
- Evaluate models on the test set and save the best model (`model.pth`).

5.3 Experiments and Findings

The following experiments were conducted to analyze model performance:

- **Website Classification Difficulty**: Evaluated which websites were easiest/hardest to classify.
- **Model Architecture**: Compared `FingerprintClassifier` vs. `ComplexFingerprintClassifier` and tested a modified architecture.
- **Hyperparameter Tuning**: Varied learning rate and batch size.

5.3.1 Website Classification Difficulty

Website	Precision	Recall	F1-Score	Support
cse.buet.ac.bd/moodle	0.96	0.95	0.96	206
google.com	0.99	1.00	1.00	190
prothomalo.com	0.96	0.96	0.96	204

Table 4: Classification Metrics per Website

Observation: **cse.buet.ac.bd/moodle** was the hardest to classify due to having very similar cache patterns as **prothomalo.com**; **google.com** was the easiest due to having quite distinct cache patterns from the other two.

5.3.2 Model Architecture Comparison

Model	Test Accuracy
FingerprintClassifier	0.93
ComplexFingerprintClassifier	0.97

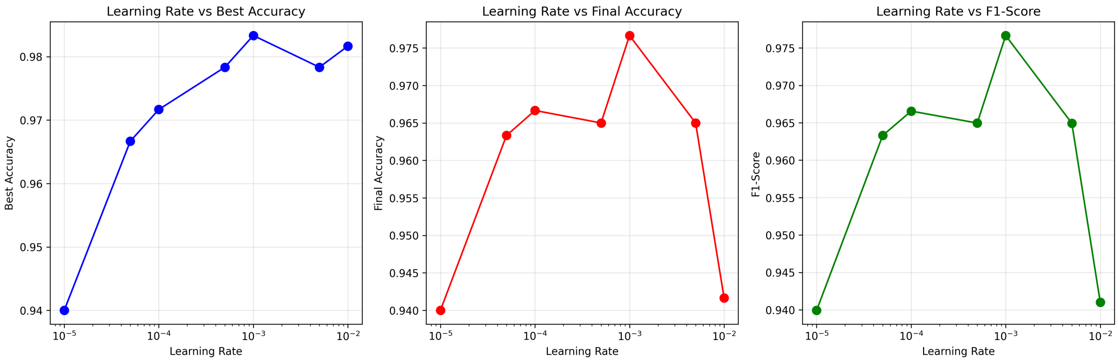
Table 5: Model Performance

Observation: The `ComplexFingerprintClassifier` model outperformed the `FingerprintClassifier`. This might be due to the more robust architecture and additional convolution layers.

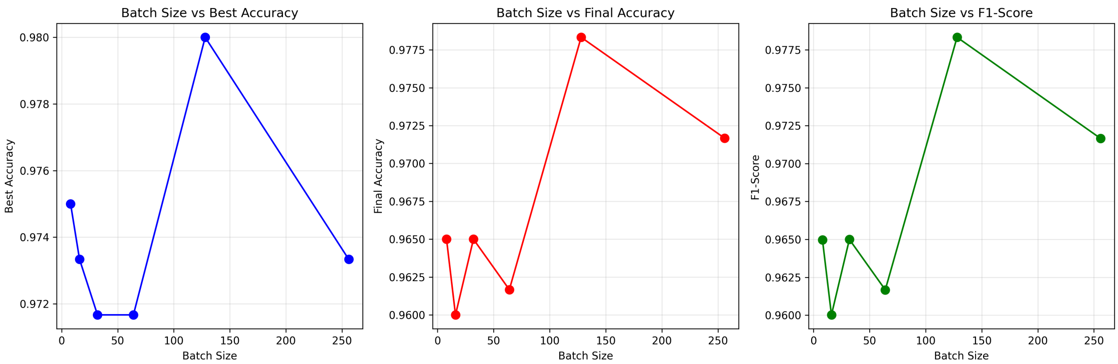
5.3.3 Hyperparameter Tuning

The accuracy and F1-score of the model were tested using different Learning Rates and Batch Sizes.

- **Varying Learning Rate**



- **Varying Batch Size**



Observation: A learning rate of 1e-3 and batch size of 128 yielded the best accuracy. Here the ComplexFingerprintClassifier model was used with the dataset containing 3000 trace data.

5.4 Overall Performance

The `ComplexFingerprintClassifier` achieved a test accuracy of 0.97, surpassing the 60% requirement. The model was saved as `complex_model.pth`.

6 Bonus Task 2: Collaborative Dataset Collection

6.1 Objective

Collect a large dataset ($\geq 50,000$ datapoints) by collaborating with classmates.

6.2 Implementation

Collaborated with 32 classmates to pool traces collected via `collect.py`. Each contributor provided approximately 30,000 traces, stored in a file **dataset_merged.json**.

6.3 Results

6.3.1 Website Classification Difficulty

Website	Precision	Recall	F1-Score	Support
cse.buet.ac.bd/moodle	0.78	0.77	0.77	7125
google.com	0.86	0.88	0.87	7253
prothomalo.com	0.79	0.78	0.79	7194

Table 6: Classification Metrics per Website

Observation: **cse.buet.ac.bd/moodle** was the hardest to classify due to having very similar cache patterns as **prothomalo.com**; **google.com** was the easiest due to having quite distinct cache patterns from the other two.

6.3.2 Model Architecture Comparison

Model	Test Accuracy
FingerprintClassifier	0.8111
ComplexFingerprintClassifier	0.8096

Table 7: Model Performance

Observation: The `FingerprintClassifier` model outperformed the `ComplexFingerpr` This might be due to the complex model overfitting the data. This is a fine example of the idea "Occums's Razor", where the simpler solution to a problem is preferred over a more complex one.

7 Challenges and Solutions

- **Large Dataset Handling**: Slow database operations; optimized by batching insertions in `database.py`.
- **Model Overfitting**: Addressed by using dropout and batch normalization.

8 Conclusion

The assignment successfully demonstrated a side-channel attack for website fingerprinting. All tasks were completed, achieving robust trace collection, automation, and classification with over 60% accuracy. Bonus tasks enhanced the system with a large collaborative dataset and real-time detection, showcasing the power of cache-based side-channel attacks. Future work could explore advanced attack techniques (Bonus Task 1) or optimize real-time performance.