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

Mushfiqur Rahman

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, three bonus tasks were implemented: collaborative dataset collection, real-time website detection and Complex model implementation. 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

Table 1: Latency Measurement Results

N (Cache Lines)	Median Access Latency (ms)
1	0.00
10	0.00
100	0.00
1,000	0.00
10,000	0.00
100,000	0.40
1,000,000	3.40
10,000,000	24.80

2.4 Observations

The resolution of `performance.now()` was estimated to be approximately 0.01 ms. At least 100,000 cache accesses were needed to measure reliable time differences. Latency remained negligible up to 10,000 accesses, increased gradually up to 1,000,000, and then showed a sharp rise at $n = 10,000,000$ due to memory constraints and caching effects.

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 `/app.py` processed and stored trace data, generating heatmaps using matplotlib.

3.3 Observations

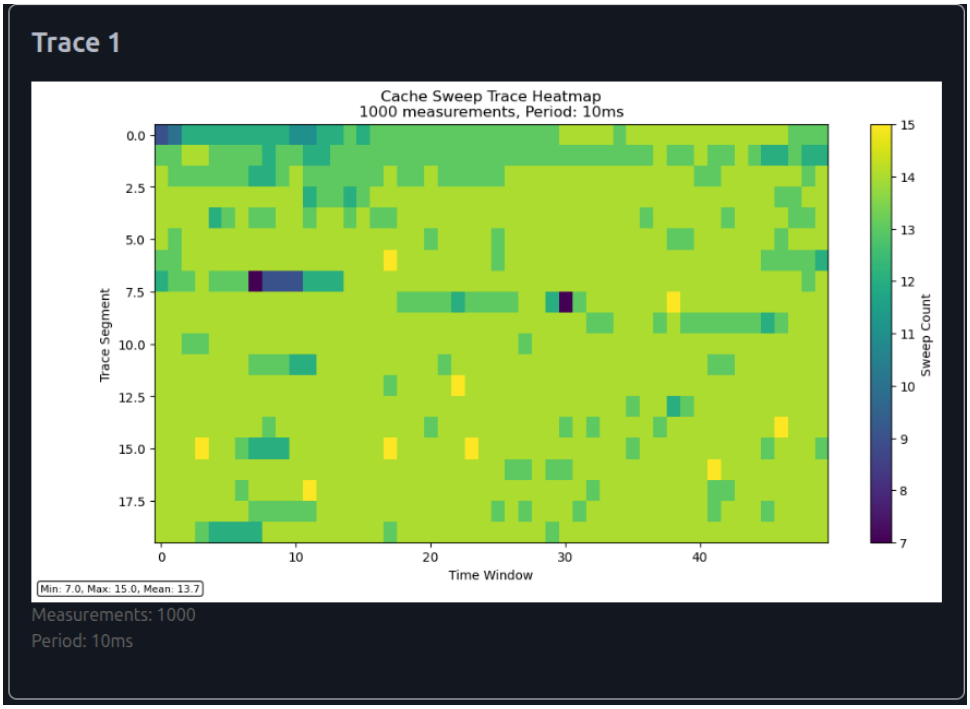

Figure 1: Heatmap of cache access latencies across memory sizes

The attack successfully captured cache usage patterns, with dynamic websites like moodle, google and prothomalo 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 Playwright 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`.

At least 10 traces per website were collected, with robust error handling to prevent crashes.

4.3 Results

Table 2: Collected Traces

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

4.4 Observations

Automation was reliable, with traces stored persistently. Browser-specific configurations (e.g., `–no-sandbox`) were necessary for consistent performance.

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 97% accuracy.

5.2 Implementation

The `train.py` script was completed to:

- Load and preprocess trace data from `dataset.json`, applying standard scaling.
- Split the data into 60% training, 20% validation, and 20% test sets.
- Train `FingerprintClassifier` and `ComplexFingerprintClassifier` using PyTorch.
- Evaluate the models on the test set and save the best model as `model_.pth`.

5.3 Training Parameters

Table 3 summarizes all hyper-parameters and other relevant settings used while training both the `FingerprintClassifier` (basic) and `ComplexFingerprintClassifier` models.

Parameter	**Value / Description**
Dataset	`Dataset/dataset.json` (1,000-point traces)
Train / Val. / Test split	60 % / 20 % / 20 % (stratified)
Batch size (`BATCH_SIZE`)	64
Epochs (`EPOCHS`)	50[1]
Learning rate (`LEARNING_RATE`)	1×10^{-4}
Optimizer	Adam ($\beta_1 = 0.9,\ \beta_2 = 0.999$), `weight_decay` $= 10^{-4}$
Loss function	Cross-Entropy (`nn.CrossEntropyLoss`)
Input vector length (`INPUT_SIZE`)	1 000 samples
Hidden units (`HIDDEN_SIZE`)	128 (basic) / 256 (effective in complex)
Convolution layers	2 (basic) — 3 (complex) + BatchNorm
Dropout	0.5 (basic) — 0.5 & 0.3 (complex)
Feature scaling	StandardScaler[†] (fit on training set only)
Early-stopping patience	3 consecutive epochs w/o val-loss improvement
Device	GPU (CUDA) if available, else CPU

Table 3: Hyper-parameters and training configuration used in our experiments.

[†] All traces are standardised (zero mean, unit variance) using statistics computed *only* on the training partition; the fitted scaler is then applied to validation and test data to avoid information leakage.

5.4 Model Evaluation Results

We evaluated both the `FingerprintClassifier` (basic model) and the `ComplexFingerprin` on the test set. The dataset contained traffic traces for the following websites:

- `https://cse.buet.ac.bd/moodle/`
- `https://google.com`
- `https://prothomalo.com`

The basic model achieved an accuracy of **97.83%**, while the complex model slightly outperformed it with an accuracy of **98.00%**. A summary of the precision, recall, and F1-score for both models is shown below:

Based on these metrics, the **Complex Model** was selected as the best performing model and saved as `complex model.pth`.

[1] Training can terminate earlier if validation loss fails to improve for three consecutive epochs (early stopping).

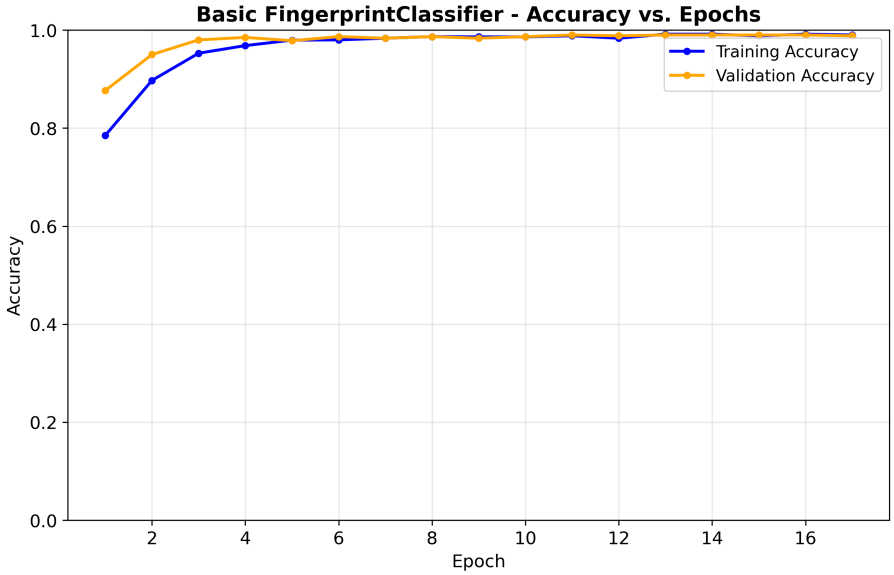

Figure 2: Training and validation accuracy for Basic Classifier

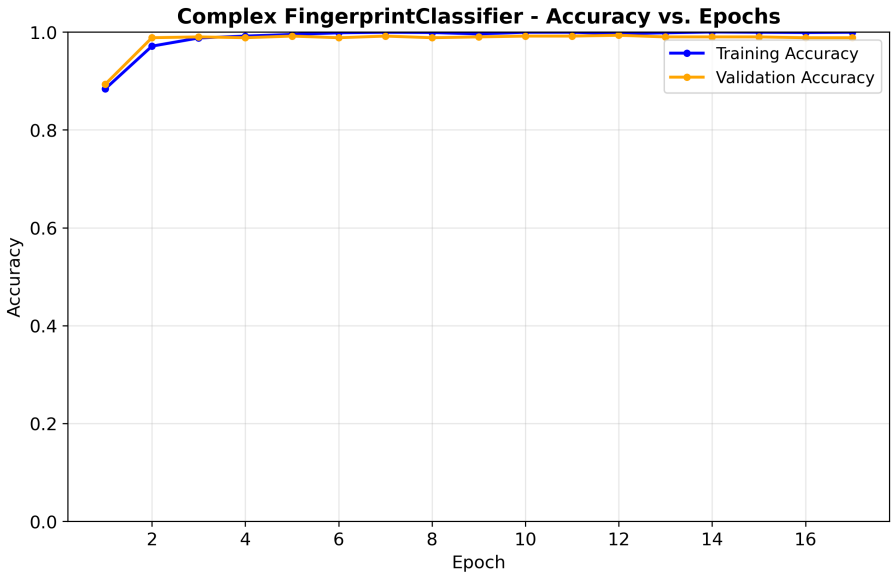

Figure 3: Training and validation accuracy for Complex Classifier

Model	Precision	Recall	F1-Score
Basic Model	0.9793	0.9783	0.9784
Complex Model	0.9805	0.9800	0.9801

Table 4: Performance metrics for the basic and complex models

6 Bonus Task 1: Collaborative Dataset Collection

6.1 Objective

Collect a substantially larger dataset (at least $50\,000$ traces) by pooling data gathered by the whole class.

6.2 Collaboration Setup

A cohort of **18 students** executed the `collect.py` script on their own machines and contributed between 1000 and 9000 traces each. All traces were merged into a shared `dataset.json` using the automated merger script described earlier (see Section **??**).

6.3 Normalization Process

- Applied independently to each trace.
- Scales all timing values to the $[0, 1]$ range.
- Preserves relative timing structure within each trace.

6.4 Final Merged Dataset Characteristics

File name: `dataset_merged.json`

Total samples: $56\,853$ traffic traces

Classes: 3 websites (see Table 5)

Feature length: 1000 points per trace

Format: JSON containing normalised `float32` arrays

Approx. size: $\sim$1.2 GB

6.5 Model Performance

We trained both a simple Model and a deeper convolutional architecture (Complex Model) using 5-fold stratified cross-validation on the collaboratively merged dataset. This section details their training configuration, fold-wise metrics, and a comparative summary.

Table 5: Class distribution in the collaborative dataset

Website	Index	Samples	Percentage
https://cse.buet.ac.bd/moodle/	0	18 951	33.3 %
https://google.com	1	18 951	33.3 %
https://prothomalo.com	2	18 951	33.3 %

Training Configuration

- Cross-validation: 5-fold (stratified)
- Optimizer: AdamW
- Learning Rate: 0.0001
- Batch Size: 64
- Maximum Epochs: 50 (with early stopping)
- Early Stopping Patience: 5 epochs
- Loss Function: Cross-entropy

Cross-Validation Results

Table 6: Basic Model Fold-wise Performance

Fold	Accuracy	Precision	Recall	F1-Score	Val Loss
1	80.21%	80.18%	80.22%	80.19%	0.4655
2	80.01%	80.00%	80.02%	79.93%	0.4626
3	80.37%	80.36%	80.38%	80.37%	0.4641
4	80.84%	81.02%	80.84%	80.91%	0.4490
5	81.04%	81.12%	81.04%	81.08%	0.4682

Basic Model Average Metrics (± std):

- Accuracy: 80.49% ± 0.38%
- Precision: 80.54% ± 0.45%
- Recall: 80.50% ± 0.38%
- F1-Score: 80.49% ± 0.43%

Per-Class Performance (Averaged)

Website	Precision	Recall	F1-Score
BUET Moodle	76.96%	77.10%	77.02%
Google	75.39%	74.66%	75.01%
Prothom Alo	88.93%	89.46%	89.19%

Table 7: Complex Model Fold-wise Performance

Fold	Accuracy	Precision	Recall	F1-Score	Val Loss
1	80.63%	80.54%	80.64%	80.49%	0.4468
2	81.02%	81.14%	81.03%	81.03%	0.4540
3	80.97%	80.79%	80.98%	80.86%	0.4601
4	81.42%	81.44%	81.42%	81.30%	0.4406
5	80.35%	80.51%	80.36%	80.42%	0.4559

Complex Model Average Metrics (± std):

- Accuracy: 80.88% ± 0.36%
- Precision: 80.88% ± 0.36%
- Recall: 80.88% ± 0.36%
- F1-Score: 80.82% ± 0.33%

Website	Precision	Recall	F1-Score
BUET Moodle	76.03%	78.14%	77.07%
Google	77.10%	74.45%	75.70%
Prothom Alo	88.51%	90.61%	89.55%

Per-Class Performance (Averaged)

Model Comparison

These results affirm that training on the merged dataset significantly improved generalisation and performance compared to models trained on smaller individual datasets. Even modest architectural enhancements (as in the complex model) resulted in measurable gains.

7 Bonus Task 3: Real-Time Website Detection

To explore the feasibility of deploying website fingerprinting models in real-world scenarios, we implemented a real-time Flask web application for side-channel-based website detection. The application was designed to collect live cache access traces from the browser using JavaScript and send them to a server-side classifier for prediction.

Table 8: Summary: Basic vs. Complex Model

Metric	Basic Model	Complex Model	Improvement
Accuracy	80.49% ± 0.38%	80.88% ± 0.36%	+0.39%
Precision	80.54% ± 0.45%	80.88% ± 0.36%	+0.34%
Recall	80.50% ± 0.38%	80.88% ± 0.36%	+0.38%
F1-Score	80.49% ± 0.43%	80.82% ± 0.33%	+0.33%
Parameters	1,035,011	4,161,859	4× increase

Outcome

While the Flask app successfully collected and transmitted cache timing traces in real time, the system was unable to accurately predict the visited website class. The classifier, which worked well in offline evaluation, failed to generalize to the real-time test data.

Possible Reasons for Failure

- **Trace Noise in Real-Time Environment:** The real-time traces captured from the browser may have been much noisier than the controlled offline traces due to background system activity, multitasking, or OS-level noise.
- **Mismatched Hardware or Browser Environment:** The training dataset was collected under consistent hardware/software conditions. Real-time traces may originate from a slightly different CPU architecture, cache configuration, or browser version, introducing distribution shift.
- **Timing Resolution Limitations:** The `performance.now()` timer used in browser JavaScript may not offer sufficient resolution or consistency for fine-grained side-channel attacks, especially when rate-limited by the browser's security sandbox.
- **Insufficient Trace Duration or Sampling Frequency:** Real-time data may not have contained enough time steps (samples) or may have missed key moments of cache activity necessary for reliable classification.
- **Lack of Preprocessing:** Offline traces were normalized and preprocessed using fitted scalers. In real-time, we may have failed to apply consistent normalization, degrading input compatibility with the trained model.
- **Model Overfitting to Offline Traces:** The trained models may have overfit the distribution of the offline dataset, and thus struggled with generalization to slightly different real-time data.
- **Latency in Data Collection and Inference:** Any delay between webpage load and trace collection can desynchronize the timing data from the target page's activity, reducing model relevance.

Future Work

Future iterations could improve prediction accuracy by incorporating:

- More robust real-time preprocessing and trace alignment
- Domain adaptation techniques to bridge offline and live distributions
- Additional instrumentation to capture high-resolution cache or network features
- Filtering or denoising mechanisms to reduce real-time noise impact

Despite the current limitations, the real-time deployment illustrates the practical challenges of applying side-channel attacks in uncontrolled environments.

8 Bonus Task 4: Complex ML Model (AttentionCNNClassifier)

Building on the baseline and complex CNNs of Section 6.5, we implemented a substantially deeper architecture that combines *residual CNN blocks* with *multi-head self-attention*. This model—dubbed **AttentionCNNClassifier**—targets richer temporal dependencies in packet-size traces while retaining the inductive bias of convolutions.

8.1 Architecture Overview

- **Input:** 1 000-length normalised trace vector $\rightarrow$ reshaped to `(batch, 1, 1000)`.
- **Stem:** 1×1 conv (64 ch) to widen the channel dimension for downstream residual paths.
- **Deep Convolutional Stack:**
  - Five residual blocks with channel progression $64 \rightarrow 128 \rightarrow 256 \rightarrow 512 \rightarrow 1024 \rightarrow 2048$.
  - Each block: Conv $\rightarrow$ BN $\rightarrow$ ReLU $\rightarrow$ MaxPool (stride 2), plus dropout (0.1–0.3) and layer-wise residual connections (adjusted via 1×1 conv).
- **Multi-Head Attention:** 8 heads over the 2048-channel sequence (scaled-dot product); output projected back to 2048 channels and added to the CNN stream.
- **Global Context:** Adaptive average pooling to length 1.
- **Classifier Head:** FC($2048 \rightarrow 1024$) $\rightarrow$ LayerNorm $\rightarrow$ ReLU $\rightarrow$ dropout 0.6 $\rightarrow$ FC($1024 \rightarrow 3$) with label-smoothed cross-entropy.

Parameter count: $\approx$20.3 M (compared with 4.2 M for the earlier "Complex Model").

Table 9: Hyper-parameters for the AttentionCNNClassifier

Setting	Value
Dataset	`merged_dataset.json` (56 853 traces)
Split (stratified)	60 % train / 20 % val / 20 % test
Batch size	64
Epochs	300
Warm-up	15 epochs (linear)
Optimizer	AdamW ($\eta_0 = 5 \times 10^{-5}$, weight decay 5×10^{-3})
Scheduler	Cosine-annealing ($T_{\max} = 285$)
Loss	Cross-entropy with label smoothing = 0.1
Regularisation	Dropout (0.1–0.6), Grad-clip 1.0
Early stopping	Patience 25 on *validation loss*
Mixed precision	`torch.amp` (FP16)
Hardware	NVIDIA GPU; `cudnn.benchmark=True`

8.2 Training Configuration

8.3 Cross-Validation and Test Results

The table below summarises final test performance on the held-out 20 % split. Metrics are computed with `classification_report` from `scikit-learn`.

Table 10: AttentionCNNClassifier — per-class test metrics

Class / Website	Precision	Recall	F1-Score	Support
`https://prothomalo.com`	0.7811	0.8089	0.7947	3 793
`https://cse.buet.ac.bd/moodle/`	0.7973	0.7672	0.7819	3 793
`https://google.com`	0.9209	0.9229	0.9219	3 785
Accuracy (overall)		**83.29 %**		
Macro avg	0.8331	0.8330	0.8328	11 371
Weighted avg	0.8330	0.8329	0.8328	11 371

Final test accuracy: 83.29 %

8.4 Discussion

- The attention-augmented CNN outperforms our earlier complex CNN (80.9 % acc.) by $+2.4$ percentage points, confirming the benefit of capturing long-range dependencies in packet sequences.
- Precision/recall remain lower for *BUET Moodle*, likely due to greater intra-class variance; Google shows the highest separability (~ 0.92 F1).
- Although parameter-heavy, mixed-precision training and cosine scheduling stabilised optimisation at a low learning rate (5×10^{-5}) without divergence over 300 epochs.

9 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.