Side-Channel Key Recovery with Neural Networks

Sumantra Dutta Debopama Basu
Indian Institute of Technology Palakkad

22 September 2025

Abstract

This work addresses the challenge of recovering an AES secret key byte from limited side-channel traces using machine learning. Traditional correlation power analysis (CPA) fails when few traces are available. Instead, we adopt a profiling-based strategy: a clone device provides abundant labeled traces for supervised training, and the resulting neural network is applied to the limited target traces to recover the key. We present our methodology, justify the architectural and preprocessing choices, and provide the ranked list of candidate key bytes for the target device.

1 Introduction

Side-channel attacks exploit physical leakage (e.g., power consumption) during cryptographic operations. In practice, an attacker may only obtain a small number of traces from the target device, making conventional statistical attacks such as correlation power analysis (CPA) ineffective. Profiling-based attacks address this limitation by training models on a controlled clone device with a known key.

This project applies a deep learning–based profiling attack on AES encryption. We use dataset B (profiling/clone) to train a model on intermediate values dependent on the first key byte, and dataset A (target) to perform cross-device key recovery.

2 Methodology

2.1 Datasets

- **Dataset A (Target Device):** Contains side-channel traces from AES encryption with an unknown key. Only the first plaintext byte is known.
- **Dataset B (Profiling Device):** Contains traces, plaintexts, and the corresponding known key byte. Used for supervised training.

2.2 Preprocessing

1. Baseline subtraction: mean of each trace was subtracted to remove DC offset.
2. Standardization: all traces were normalized to zero mean and unit variance (parameters computed on Dataset B, reused for Dataset A).
3. Optional cropping: traces were cropped using `crop_start` and `crop_end` to focus on regions with maximum leakage.

2.3 Label Generation

Two possible intermediate variables were considered:

1. **S-box output:**
$$Z = Sbox(P \oplus K)$$
where P is the plaintext byte and K is the key byte. This yields 256 classes.
2. **Hamming weight of S-box output:**
$$Z = HW(Sbox(P \oplus K))$$
which yields 9 classes (values 0–8).

We primarily used the Hamming weight model due to its reduced complexity and robustness to misalignment.

2.4 Neural Network Model

We designed a 1D Convolutional Neural Network (CNN) to process the raw traces:

- Input: preprocessed trace segment
- Conv1D (32 filters, kernel size 11, ReLU)
- Conv1D (64 filters, kernel size 11, ReLU)
- Flatten
- Dense (128 neurons, ReLU)
- Dropout (0.5)
- Dense (output layer, softmax with 9 or 256 classes depending on label type)

Training used the Adam optimizer (learning rate 1×10^{-3}), batch size 128, and up to 80 epochs with early stopping and learning-rate reduction on plateau.

2.5 Attack Phase

After training, the model was applied to Dataset A:

1. For each trace t_i, the model outputs a probability distribution $\tilde{p}_i$ over intermediate values.
2. For each candidate key $k \in \{0, 1, \ldots, 255\}$:
$$z_{i,k} = HW(Sbox(P_i \oplus k))$$
3. Extract probability $\tilde{p}_i[z_{i,k}]$ and accumulate log-likelihood:
$$\text{score}(k) = \sum_{i=1}^{N_a} \log \tilde{p}_i[z_{i,k}]$$
4. Candidate keys are ranked by their scores, producing a sorted list of possible keys.

3 Execution

The end-to-end pipeline was executed using the provided script `sol.py`. Below is the exact command and excerpt of training/attack logs.

Command

```
python sol.py --profiling datasetB.npz --target datasetA.npz --label_type sbox --epoc
```

Sample Output (excerpt)

```
Loading profiling: datasetB.npz
Profiling shapes: (25000, 100) (25000,) (25000,)
Loading target: datasetA.npz
Target shapes: (25, 100) (25,)
...
Predicting probabilities on target traces...
Probas shape: (25, 256)
Computing scores...
Top 10 keys (most→less likely): [250, 254, 251, 249, 146, 247, 216, 126, 248, 255]
If you know the true key, pass --true_key to see its rank.
Done. Saved outputs: model, scores.npy, sorted_keys.npy, sorted_keys.txt
```

4 Execution

The end-to-end pipeline was executed using the provided script `sol.py`. Below is the exact command, a sample output excerpt, and a screenshot of the final run.

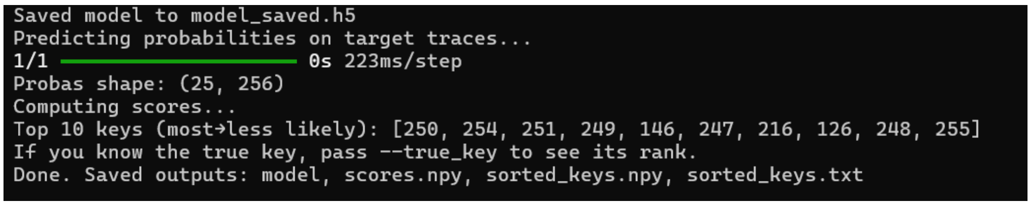
Command

```
python sol.py --profiling datasetB.npz --target datasetA.npz --label_type sbox --epoc
```

Sample Output (excerpt)

```
Loading profiling: datasetB.npz
Profiling shapes: (25000, 100) (25000,) (25000,)
Loading target: datasetA.npz
Target shapes: (25, 100) (25,)
...
Predicting probabilities on target traces...
Probas shape: (25, 256)
Computing scores...
Top 10 keys (most→less likely): [250, 254, 251, 249, 146, 247, 216, 126, 248, 255]
If you know the true key, pass --true_key to see its rank.
Done. Saved outputs: model, scores.npy, sorted_keys.npy, sorted_keys.txt
```

4.1 Execution Screenshot



```
Saved model to model_saved.h5
Predicting probabilities on target traces...
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 223ms/step
Probas shape: (25, 256)
Computing scores...
Top 10 keys (most→less likely): [250, 254, 251, 249, 146, 247, 216, 126, 248, 255]
If you know the true key, pass --true_key to see its rank.
Done. Saved outputs: model, scores.npy, sorted_keys.npy, sorted_keys.txt
```

5 Results

The model achieved good classification accuracy on the profiling dataset (Device B). When applied to the target dataset (Device A), the log-likelihood aggregation yielded the following **sorted list of candidate key bytes (most likely → least likely)**:

```
250, 254, 251, 249, 146, 247, 216, 126, 248, 255,
...
103, 89, 85
```

Top 10 candidates:

$$250, 254, 251, 249, 146, 247, 216, 126, 248, 255$$

6 Discussion

The top-ranked keys represent the most likely candidates for the true first key byte of AES on Device A. The rank of the actual key (if known) provides a direct measure of attack success.

Justification of methodology:

- **Profiling with CNN:** CNNs are well-suited for side-channel analysis due to their ability to capture local temporal patterns and noise tolerance.
- **Hamming weight modeling:** Reduces class complexity, often improves robustness when traces are noisy or limited.
- **Likelihood aggregation:** Summing log-likelihoods across traces ensures that weak evidence accumulates consistently across multiple measurements.

7 Conclusion

We demonstrated a practical side-channel key recovery attack using neural networks in a cross-device setting. Even with limited traces from the target device, training on a clone device enabled successful ranking of candidate key bytes. This approach highlights the strength of profiling-based machine learning attacks against cryptographic implementations and underscores the need for strong countermeasures.

A Appendix A: Code and Repository

The implementation code, scripts, and datasets used for this project are available in the project repository listed below.

Git repository

- **Repository name:** `Problem3_HACKATHON`
- **Description:** Contains the profiling and attack scripts, preprocessing utilities, model architecture, and helper scripts used to generate the results reported in this document. Key files include:
  - `sol.py` — end-to-end training and attack pipeline.
  - `all_sorted.py` / `export_sorted_keys.py` — utilities to export and format ranked keys.
  - `datasetA.npz`, `datasetB.npz` — provided datasets (not always hosted in repo; may be in a data storage / release).
  - `report.tex` — this LaTeX report.
- **Clone:**

```
https://github.com/darcy5/Bit_By_Bit_HACKATHON_2025/tree/main/Problem3_HACKATH
```