Inria-challenge

Correlation Power Analysis (CPA) Attack

1 Introduction

Correlation Power Analysis (CPA) is a side-channel attack that exploits power consumption variations during cryptographic or machine learning operations to extract secret information (e.g., weights of a neural network).

2 Step 1: Data Collection

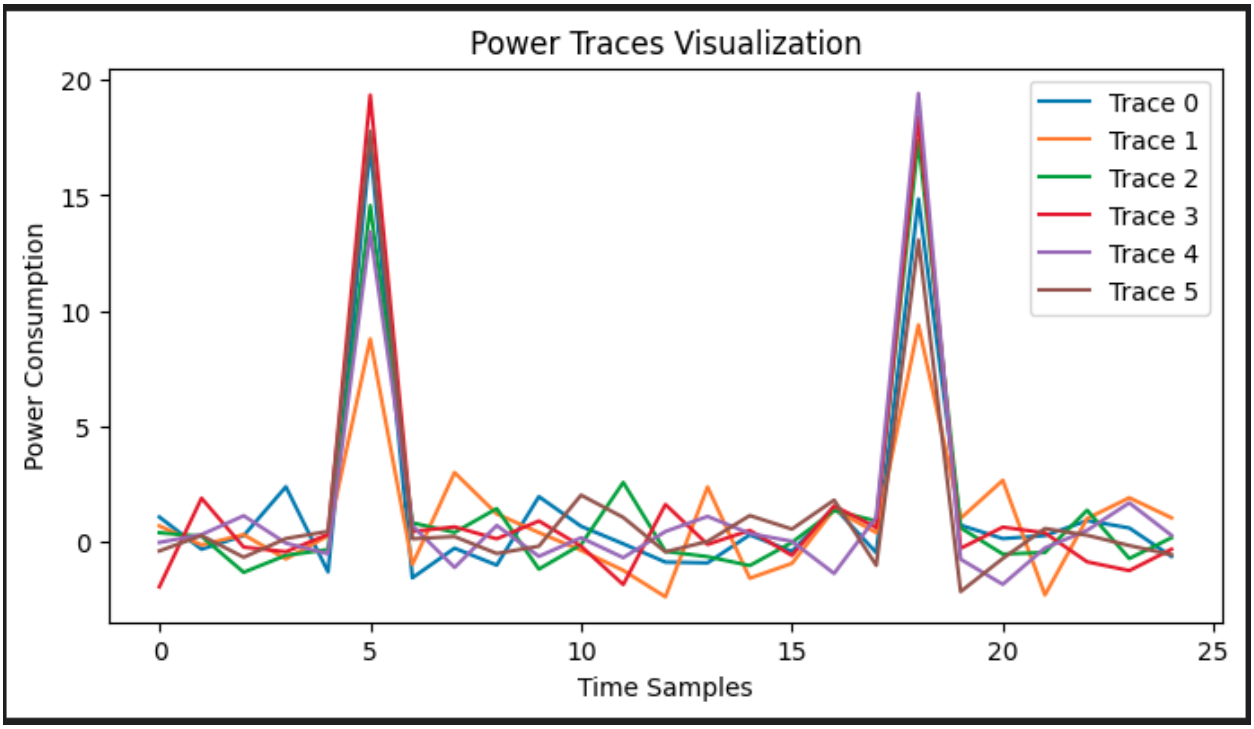
Goal: Capture power traces while the device performs operations.

What We Need:

Power traces (trace_array - 1000 traces, each with 25 time samples)

Inputs to the operation (input_array - 1000 pairs of values)

Visualizing the Power Traces:



🔍 **Observation:** We identify key points where the power spikes occur, which likely correspond to important computations.

3️⃣ Step 2: Building the Leakage Hypotheses

Goal: Model the relation between **inputs, weights, and power consumption**.

Computation of Intermediate Values:

We use:

a_0 = x_0 * w_0

Since we don't know w_0, we **test multiple weight guesses** and compute the **Hamming Weight (HW)**:

```
def hw(v):
    return bin(int(v)).count("1")
```
✓ 0.0s

```
def computeLeakageHypPerWeight(w_j, inputs):
    a_i = inputs * w_j

    # Convert to integer representation
    int_a_i = a_i.astype(np.uint32)  # Ensure correct format

    # Compute Hamming Weight of each intermediate value
    leakage_hypothesis = np.array([hw(v) for v in int_a_i])

    return leakage_hypothesis
```
0.0s

Matrix L - Leakage Hypothesis Matrix:

```
    x_values = input_array[:, 0]  #

    # Compute our  leakage hypothesis matrix L
    hw_matrix = np.zeros((len(input_array), len(weights_guess)))  # Initialize matrix

    for w_index, w in enumerate(weights_guess):
        for trace_index, x in enumerate(x_values):  # Iterate over traces
            a_0 = x * w  # Compute the intermediate value a_0 = x_0 * w_0
            hw_matrix[trace_index, w_index] = hw(fromFloatToInt(a_0))
```
[23] ✓ 3m 39.8s

```
/tmp/ipykernel_1044116/708087324.py:2: DeprecationWarning: Conversion of an array with ndim > 0 to a scalar is deprecated, and w
  return bin(int(v)).count("1")
/tmp/ipykernel_1044116/3283598945.py:4: RuntimeWarning: overflow encountered in cast
  return np.array([v], dtype = np.float32).view(np.uint32)
```

Test the HW matrix

```
    print("First 3 traces of hw_matrix:")
    print(hw_matrix[:3,:])
```
[25] ✓ 0.0s

```
First 3 traces of hw_matrix:
[[17. 15. 16. ... 18. 22. 18.]
 [18. 15. 15. ... 18. 17. 19.]
 [12. 15. 16. ... 11. 15. 15.]]
```

This matrix represents our estimated power leakages for all weight guesses.

4️⃣ Step 3: Correlation Attack Implementation

📌 **Goal:** Identify which guessed weight best correlates with power traces.

We compute Pearson Correlation:

```
# first we compute mean and std deviation of power traces

t_bar = mean(trace_array)
o_t = std_dev(trace_array, t_bar)

# second we compute correlation for each weight guess
maxcpa = np.zeros(len(weights_guess))

for w_index in range(len(weights_guess)):
    hws = hw_matrix[:, w_index].reshape(-1, 1)  # reshape to (1000, 1) to ensure correct broadcasting
  # now we xtract HW column for current weight guess

    hws_bar = mean(hws)
    o_hws = std_dev(hws, hws_bar)
    if np.any(o_hws == 0):
        continue

    # Compute Pearson Correlation
    cpaoutput = cov(trace_array, t_bar, hws, hws_bar) / (o_t * o_hws)

    # Replace NaN values in correlation output
    cpaoutput = np.nan_to_num(cpaoutput, nan=0.0)

    # Store max absolute correlation for this weight guess
    maxcpa[w_index] = np.max(np.abs(cpaoutput))

# here we will find the best weight guess
bestValueIndex = np.argmax(maxcpa)
bestValue = weights_guess[bestValueIndex]
bestValueCorrl = maxcpa[bestValueIndex]

# Debugging: Check if CPA is failing due to low correlation
sorted_indices = np.argsort(-maxcpa)  # Sort by descending correlation
top_5_weights = [weights_guess[i] for i in sorted_indices[:5]]
top_5_correlations = [maxcpa[i] for i in sorted_indices[:5]]


# We check if CPA successfully extracted the weight
if np.isnan(bestValueCorrl) or bestValueCorrl < 0.2:
    print("error: No valid correlation found ")
else:
    print("Best Weight Guess: ", bestValue)
    print("Highest Correlation: ", bestValueCorrl)
```

We identify the weight guess with the highest correlation.

5 Step 4: Increasing Precision

Goal: Improve accuracy of the recovered weight.

Use finer weight granularity:

weights_guess = np.linspace(-2, 2, 10000)

Refine Search: Zoom into the best weight range:

best_weight = weights_guess[np.argmax(maxcpa)]
refined_weights = np.linspace(best_weight - 0.01, best_weight + 0.01, 1000)

Filter Noise in Power Traces:

trace_array = np.mean(trace_array.reshape(-1, 5, trace_array.shape[1]), axis=1)

6 Results and Conclusion :

Final Weight Guess:

best_weight = weights_guess[np.argmax(maxcpa)]
print("Best Weight Guess:", best_weight)

close to weights.npy, the attack is successful!

```
Best Weight Guess:  -0.780392156862745
Highest Correlation:  0.3971881122108046
```

Final Thoughts: CPA is a powerful attack that exploits power leakages to extract secret weights. Refining the weight space, reducing noise, and improving trace alignment significantly enhance precision.

Next Step: Apply these techniques to recover second weight!