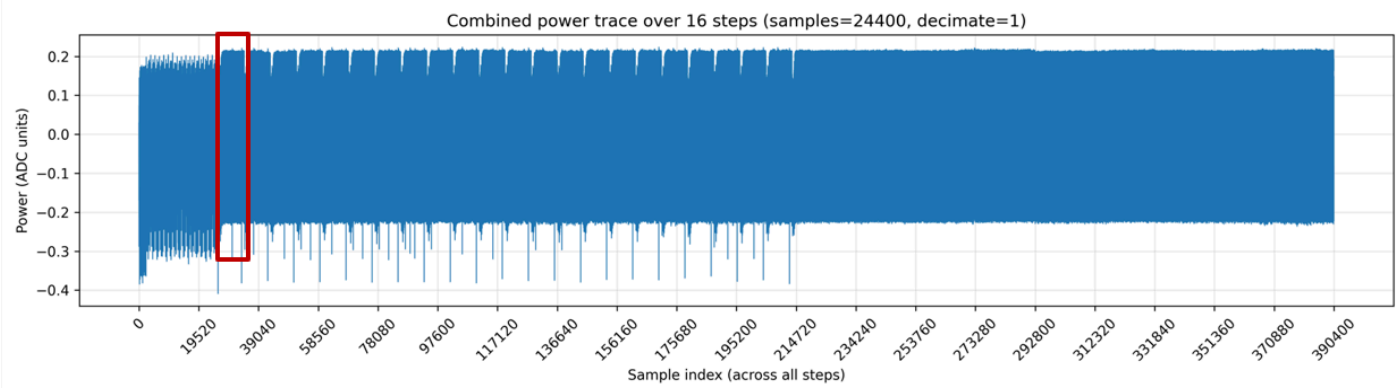
Task1

OVERVIEW

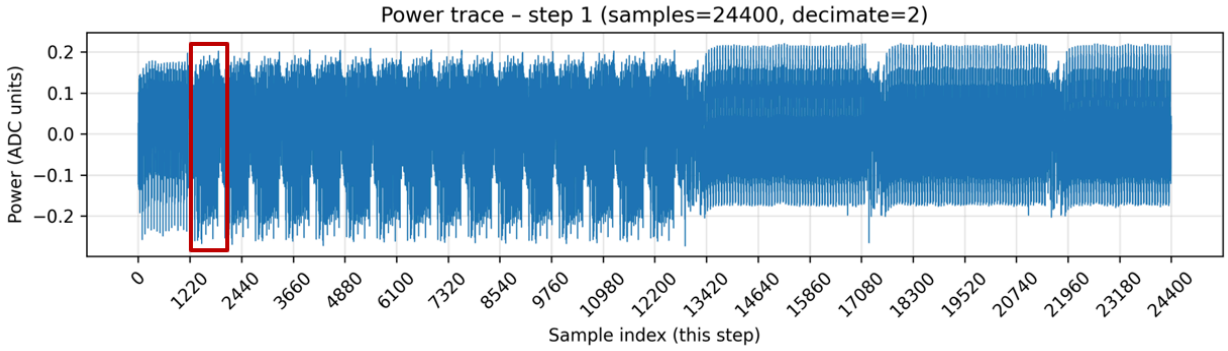
We know that on this board we have a buffer of 24,400 samples for measurement. To capture a longer execution, we can either lower the time resolution by increasing the decimation value, or keep a low decimation but repeat the measurement several times, each time increasing the offset to start recording from where the previous measurement ended. Then we concatenate all measurements to obtain a complete “helicopter view.”

You can check the `trace_steps_combined.png` file, which was created using this multi-step helicopter view method. In our Python code, we set the number of steps to 8 and the decimation to 2, and obtained the following result.

As shown in the figure, the red border highlights a repeating pattern that occurs 22 times. Since this pattern appears at the end of the trace, it cannot correspond to DES operations. Therefore, we interpret it as the UART transmission pattern.

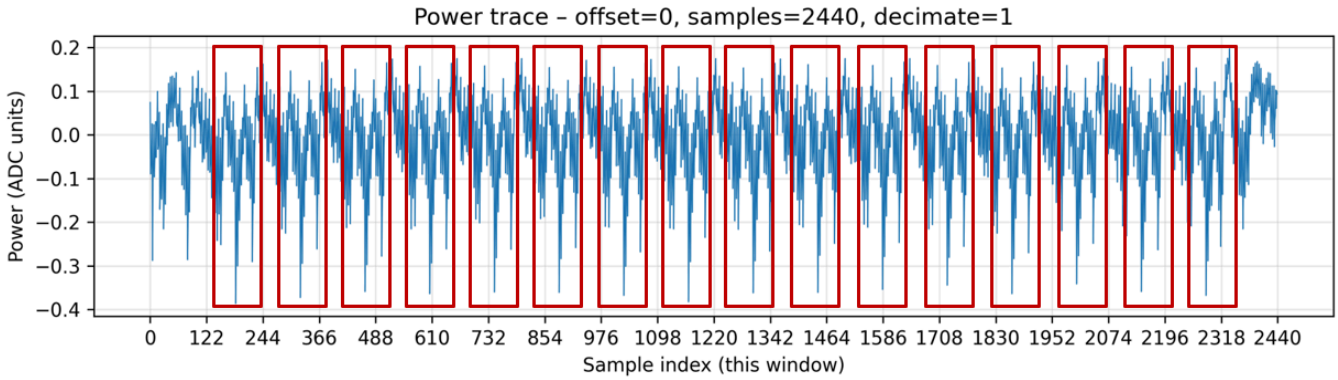


To identify the DES key generation and Feistel patterns, we zoomed in on the first part of the output. In the next figure, the red border highlights a pattern repeated 16 times. This likely corresponds to the Feistel steps in DES. Before this part of the measurement, we should find another 16 repeating patterns representing the key generation phase which is shown in the last figure.



In the next zoomed view, shown in the following figure, we can see another 16 repeated patterns that are shorter than the main encryption part. These shorter patterns correspond to the key generation of DES. This last image was captured using the `capture_window.py` script, which focuses on the early portion of the measurement.

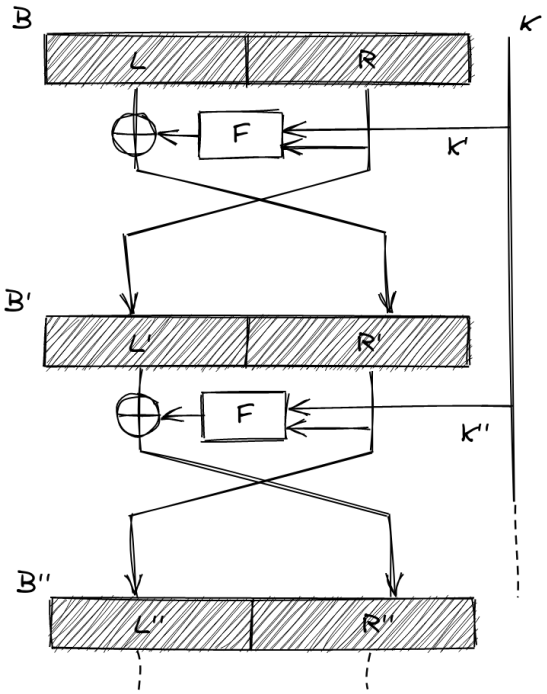
Using this approach, we were able to identify all the patterns needed for key leakage analysis.



Task2

In Task 2, we generated two sets of 100 plaintexts, where set A had values of the form 0x00000000XXXXXXXX and set B had 0xFFFFFFFFYYYYYYYY, with random 32-bit lower halves. Using the ChipWhisperer-Lite and STM32 target, we captured one power trace per plaintext with 13420 samples, decimate = 4, and offset = 0. Each trace was saved as a .npy file in set_A and set_B directories. We then averaged all traces in each set to obtain set_A_average.npy and set_B_average.npy, computed abs(tAavg − tBavg), and plotted it together with one example trace. The resulting difference trace showed a clear spike at the boundary between the key generation and the start of the Feistel rounds, indicating where the plaintext is first processed.

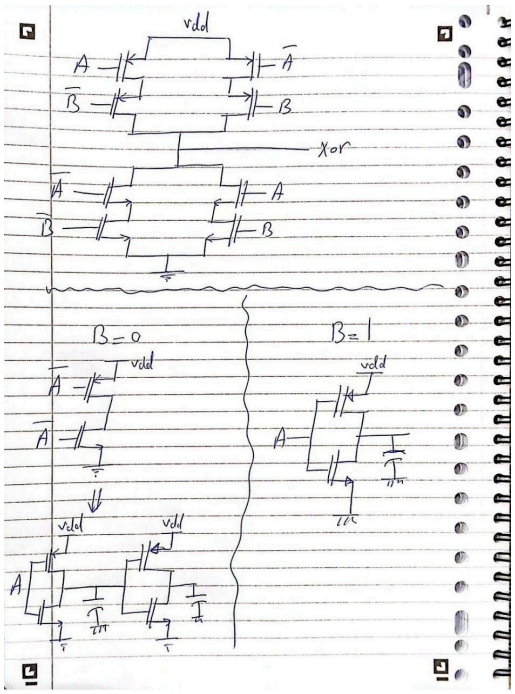
To explain why this happens, we need to look at the DES diagram and the Feistel structure. In the first round, the most significant 32 bits (MSB) of the data are XORed with the output of the F function.



In our experiment, the 32 MSB bits in set A are all zeros, while in set B they are all ones. This means that one of the inputs to the XOR gate during the first Feistel round is either all zeros or all ones, depending on the set.

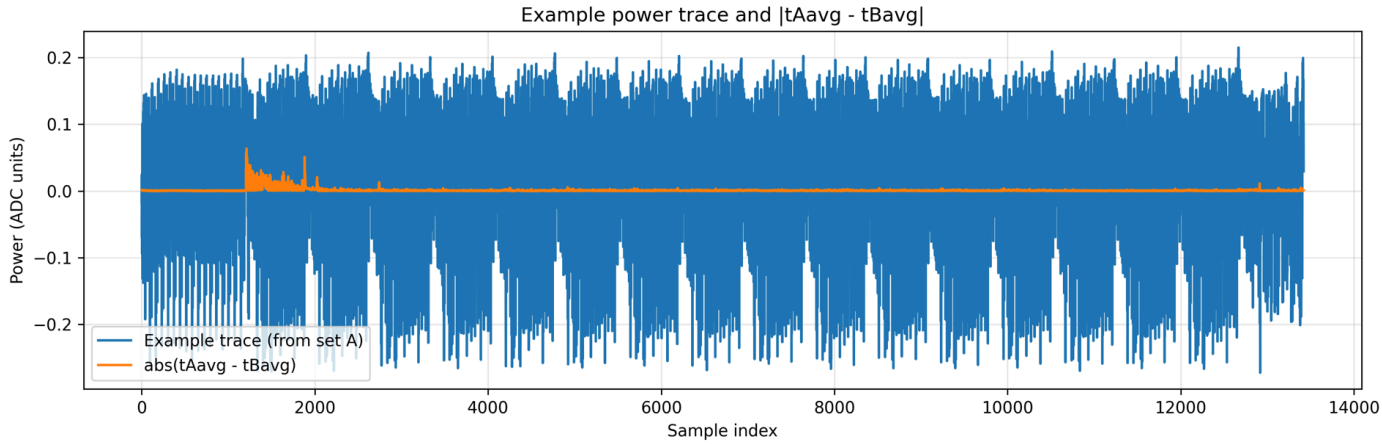
If we look at the CMOS-level implementation of the XOR gate, we can understand why this causes a difference in power consumption. In the diagram, the top part shows a standard CMOS XOR circuit. The lower part of the figure shows two simplified cases: on the left, the XOR when input B is zero, and on the right, when input B is one. When B = 0, the XOR gate behaves like two NOT gates in series. So, when the input changes, two gate capacitors must charge or discharge. However, when B = 1, the XOR behaves like a single NOT gate, meaning only one capacitor needs to charge or discharge. This results in a noticeable difference in dynamic power consumption between the two cases.

Therefore, since in set A the MSB bits are all zeros and in set B they are all ones, the CMOS switching activity is different when the first Feistel round starts. This difference creates the distinct spike we observed in the power trace, marking the beginning of the Feistel operation in DES.



Important Note:

We know that DES begins with an initial permutation that rearranges the input bits before the Feistel rounds start. This means the original 32 MSB bits of the plaintext are spread across the entire 64-bit block, but some of them always end up in the higher 32 bits used in the first round. As a result, even though the bits are permuted, the difference between all-zero MSBs in set A and all-one MSBs in set B still affects the data handled in the first Feistel round. When we run the experiment 100 times for both sets, this consistent difference in bit patterns leads to a measurable difference in power consumption, clearly visible in the averaged traces.



The three scripts in Task 2 work together to generate plaintexts, capture power traces, and process the results. `generate_64bit_sets.py` creates two sets of 100 random 64-bit values, set A with upper 32 bits as zeros and set B with upper 32 bits as ones. `task2_generate.py` reads these values, sends each to the STM32 target via ChipWhisperer, and captures one trace per input, saving them as `.npy` files in `set_A` and `set_B` folders. Finally, `task2_process.py` averages all traces in each set, computes the absolute difference between the averages, and plots it to highlight where the plaintext is processed in the DES operation.

Task3

Why finding K1 is enough to recover the full 64-bit DES key

In DES, the original 64-bit key includes **8 parity bits** at positions 1, 8, 16, 24, 32, 40, 48, and 64. These bits are not actually used in encryption. When we remove them, we get a **56-bit effective key**.

To generate the first round subkey **K1 (48 bits)**, DES performs these steps:

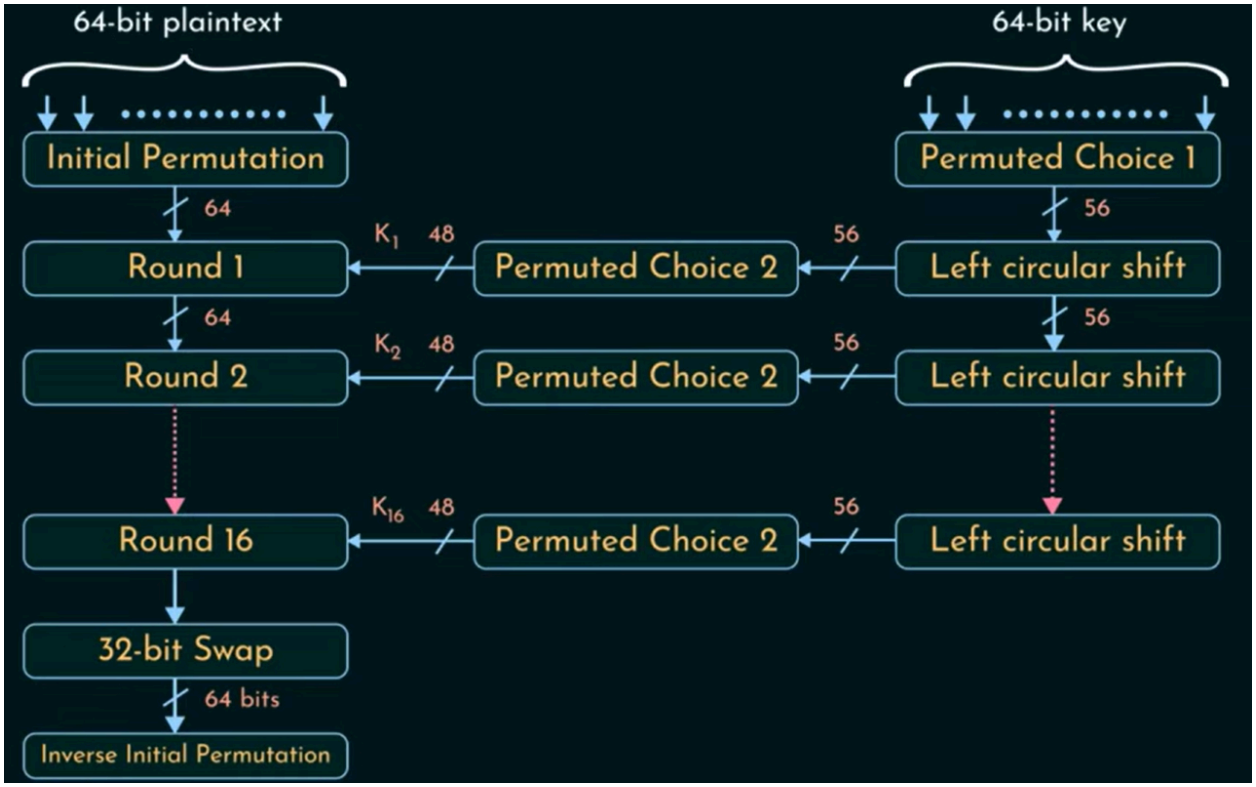
1. **Initial key permutation (PC-1):**
   This step rearranges the bits of the 64-bit key and discards the 8 parity bits, leaving a 56-bit result. This result is then split into two halves:
   - **C0:** the left 28 bits
   - **D0:** the right 28 bits
2. **Left circular shift:**
   For round 1, both halves **C0** and **D0** are rotated left by **1 bit** to produce **C1** and **D1**. This shifting changes the positions of bits slightly but does not lose any information, it's a reversible operation.
3. **Subkey permutation (PC-2):**
   The combined 56 bits (C1 + D1) are then permuted and reduced to **48 bits** using the PC-2 table. This step both reorders bits and removes 8 of them.

So, if we already know **K1 (48 bits)**, we can reverse these steps to find the original 64-bit key:

- **Step 1:** Reverse the PC-2 permutation. Since only 8 bits were dropped, we can brute-force those missing 8 bits. That means only $\mathbf{2^8 = 256}$ possibilities which is small and easily doable.
- **Step 2:** Reverse the 1-bit left shift to recover **C0** and **D0**. Because the rotation is circular, undoing it is straightforward.
- **Step 3:** Combine C0 and D0 to get the original 56-bit key, and then reinsert the 8 parity bits (which are unused) in positions 1, 8, 16, 24, 32, 40, 48, and 64.

That gives back the full 64-bit DES key.

In summary, knowing **K1** gives almost all the information about the key the remaining uncertainty is only 8 bits, which can be brute-forced easily.


64-bit plaintext
64-bit key
Initial Permutation
Permuted Choice 1
64
56
Round 1
K1
48
Permuted Choice 2
56
Left circular shift
64
56
Round 2
K2
48
Permuted Choice 2
56
Left circular shift
Round 16
K16
48
Permuted Choice 2
56
Left circular shift
32-bit Swap
64 bits
Inverse Initial Permutation

Task4

DPA on DES S-boxes

I ran a Differential Power Analysis (DPA) on the first round of DES using power traces captured from a ChipWhisperer board. For each of the eight S-boxes the script tested all 64 possible 6-bit subkey guesses. For every guess it split the traces into two groups (LSB of the S-box output = 0 or 1), averaged each group, and measured the difference. The script reports the five key guesses with largest difference peaks for each S-box and the sample index where the peak occurs. Example output (partial) showed keys such as `0x27`, `0x31`, `0x04` for S-box 1 and similar top-5 lists for S-boxes 2..8. These top guesses indicate which 6-bit values are most likely for each S-box.

How the attack works (simple)

1. Generate N random 64-bit plaintexts.
2. For each plaintext capture one power trace while the device encrypts it. Save all traces in memory and reuse them.
3. For each S-box (1..8) and each 6-bit key guess (0..63):
    - Use `sbox_out(sbox, plaintext, guess)` to compute the S-box output for every plaintext.
    - Group traces by the least-significant bit (LSB) of that output into `zero` and `one`.
    - Compute the average trace of each group (sample-wise).
    - Compute the absolute difference of these two average traces.
    - Find the maximum peak in the difference and its sample index. This peak value measures how strongly the guessed key explains power differences.
4. For each S-box sort the 64 guesses by peak and report the top five guesses and their sample indices.

This method finds small data-dependent leakage in the power traces; correct key guesses tend to produce larger peaks at the time samples where the device processes the S-box.

How to run the script (exact)

1. Requirements: Python 3, `numpy`, `chipwhisperer` Python package, and `sbox_out.py` in the same folder. Make sure ChipWhisperer hardware is connected.

2. Make the script executable and run with the number of traces you want. Example:

```
chmod +x dpa.py
./dpa.py 2000
# or
python3 dpa.py 2000
```

This will generate 200 random plaintexts in memory, capture 2000 traces, then run DPA and print the top 5 key guesses per S-box. The script saves traces to `traces/` and also to `traces_all.npy` / `plaintexts_all.npy`.

```
=== DPA top 5 keys per S-box (reuse traces) ===

S-box 1:
  #1: key= (0x27), max_peak=0.004486, sample=3853 (original ≈ 3853)
  #2: key= (0x31), max_peak=0.004429, sample=3893 (original ≈ 3893)
  #3: key= (0x04), max_peak=0.004363, sample=3853 (original ≈ 3853)
  #4: key= (0x10), max_peak=0.004010, sample=3893 (original ≈ 3893)
  #5: key= (0x19), max_peak=0.003898, sample=3829 (original ≈ 3829)

S-box 2:
  #1: key= (0x2F), max_peak=0.004184, sample=3829 (original ≈ 3829)
  #2: key= (0x1A), max_peak=0.003911, sample=3829 (original ≈ 3829)
  #3: key= (0x07), max_peak=0.003668, sample=3801 (original ≈ 3801)
  #4: key= (0x0E), max_peak=0.003474, sample=3829 (original ≈ 3829)
  #5: key= (0x04), max_peak=0.003029, sample=3765 (original ≈ 3765)

S-box 3:
  #1: key= (0x19), max_peak=0.005610, sample=3841 (original ≈ 3841)
  #2: key= (0x18), max_peak=0.004555, sample=3841 (original ≈ 3841)
  #3: key= (0x2C), max_peak=0.004093, sample=3841 (original ≈ 3841)
  #4: key= (0x09), max_peak=0.004092, sample=3865 (original ≈ 3865)
  #5: key= (0x08), max_peak=0.003659, sample=3841 (original ≈ 3841)

S-box 4:
  #1: key= (0x0E), max_peak=0.005261, sample=3961 (original ≈ 3961)
  #2: key= (0x21), max_peak=0.004470, sample=3829 (original ≈ 3829)
  #3: key= (0x36), max_peak=0.004063, sample=3829 (original ≈ 3829)
  #4: key= (0x23), max_peak=0.003676, sample=3765 (original ≈ 3765)
  #5: key= (0x1E), max_peak=0.003477, sample=3829 (original ≈ 3829)

S-box 5:
  #1: key= (0x13), max_peak=0.004476, sample=3841 (original ≈ 3841)
  #2: key= (0x0A), max_peak=0.003871, sample=4097 (original ≈ 4097)
```

```
  #3: key= (0x3B), max_peak=0.003777, sample=4097 (original ≈ 4097)
  #4: key= (0x07), max_peak=0.003605, sample=4097 (original ≈ 4097)
  #5: key= (0x2F), max_peak=0.003398, sample=3269 (original ≈ 3269)

S-box 6:
  #1: key= (0x2E), max_peak=0.005319, sample=3829 (original ≈ 3829)
  #2: key= (0x03), max_peak=0.004487, sample=3801 (original ≈ 3801)
  #3: key= (0x13), max_peak=0.003974, sample=3801 (original ≈ 3801)
  #4: key= (0x31), max_peak=0.003672, sample=3829 (original ≈ 3829)
  #5: key= (0x12), max_peak=0.003596, sample=3801 (original ≈ 3801)

S-box 7:
  #1: key= (0x21), max_peak=0.004604, sample=3829 (original ≈ 3829)
  #2: key= (0x01), max_peak=0.003844, sample=3829 (original ≈ 3829)
  #3: key= (0x05), max_peak=0.003239, sample=3829 (original ≈ 3829)
  #4: key= (0x03), max_peak=0.003016, sample=3829 (original ≈ 3829)
  #5: key= (0x00), max_peak=0.002896, sample=3829 (original ≈ 3829)

S-box 8:
  #1: key= (0x03), max_peak=0.004860, sample=3917 (original ≈ 3917)
  #2: key= (0x1D), max_peak=0.003704, sample=3829 (original ≈ 3829)
  #3: key= (0x04), max_peak=0.003440, sample=2689 (original ≈ 2689)
  #4: key= (0x08), max_peak=0.003340, sample=3829 (original ≈ 3829)
  #5: key= (0x28), max_peak=0.003251, sample=3801 (original ≈ 3801)
```

Task5

Correlation Power Analysis (CPA) Implementation

In this step, I implemented a Correlation Power Analysis (CPA) attack on the first round of the DES cipher to recover the 6-bit subkey candidates for all eight S-boxes. The script starts by generating *N* random 64-bit plaintexts and capturing *N* corresponding power traces from the ChipWhisperer board, using fixed ADC settings (offset = 3800, samples = 400, decimate = 1). Each plaintext is sent once to the target device, and its corresponding power trace is stored. These traces are later reused for all key guesses and S-boxes to avoid repeated captures.

The CPA algorithm models how power consumption depends on the intermediate values of the DES round. For each S-box (from S1 to S8) and for every possible 6-bit subkey guess (k = 0...63), the script computes the S-box output of round 1 for all plaintexts (p_0 ... p_{n-1}). From each S-box output, it calculates the **Hamming weight**, which represents the number of bits equal to '1' and serves as a simple model of the device's power usage. This produces a data structure that combines all plaintext–key pairs:

```
(p0, k0)  → HW(...)
(p1, k0)  → HW(...)
...
(p(n-1), k0)  → HW(...)
...
(p0, k63) → HW(...)
...
(p(n-1), k63) → HW(...)
```

Each column of this structure corresponds to one key guess and contains *N* hypothetical power values (Hamming weights). The script then compares each key's Hamming weight vector with the real measured power traces. For every key guess, it calculates the **Pearson correlation coefficient** between the predicted

power values and each sample point in the traces. This measures how strongly the estimated power matches the actual hardware power consumption at that time. The sample position where this correlation is the highest in absolute value is considered the best point of leakage for that key guess.

Finally, the script stores the maximum absolute correlation and the corresponding sample index for all 64 key guesses of each S-box. After sorting by correlation strength, the top five keys with the highest correlation are reported as the best candidates for each S-box. These subkeys can later be combined to reconstruct the full DES round key.

```
chmod +x cpa.py
./dpa.py 2000
# or
python3 cpa.py 200
```

```
=== CPA top 5 keys per S-box ===

S-box 1:
  #1: key=0x27 (dec=39), max_abs_corr=0.551539, sample=49 (original ≈ 3849)
  #2: key=0x1D (dec=29), max_abs_corr=0.378878, sample=49 (original ≈ 3849)
  #3: key=0x0B (dec=11), max_abs_corr=0.322788, sample=51 (original ≈ 3851)
  #4: key=0x2A (dec=42), max_abs_corr=0.306056, sample=49 (original ≈ 3849)
  #5: key=0x22 (dec=34), max_abs_corr=0.300858, sample=77 (original ≈ 3877)

S-box 2:
  #1: key=0x2F (dec=47), max_abs_corr=0.605202, sample=49 (original ≈ 3849)
  #2: key=0x23 (dec=35), max_abs_corr=0.295200, sample=50 (original ≈ 3850)
  #3: key=0x20 (dec=32), max_abs_corr=0.294440, sample=49 (original ≈ 3849)
  #4: key=0x17 (dec=23), max_abs_corr=0.275846, sample=80 (original ≈ 3880)
  #5: key=0x1B (dec=27), max_abs_corr=0.272713, sample=6 (original ≈ 3806)

S-box 3:
  #1: key=0x19 (dec=25), max_abs_corr=0.469219, sample=89 (original ≈ 3889)
  #2: key=0x1D (dec=29), max_abs_corr=0.318470, sample=146 (original ≈ 3946)
  #3: key=0x09 (dec= 9), max_abs_corr=0.313989, sample=142 (original ≈ 3942)
  #4: key=0x1B (dec=27), max_abs_corr=0.305733, sample=142 (original ≈ 3942)
  #5: key=0x21 (dec=33), max_abs_corr=0.293713, sample=174 (original ≈ 3974)

S-box 4:
  #1: key=0x0E (dec=14), max_abs_corr=0.529196, sample=157 (original ≈ 3957)
  #2: key=0x1E (dec=30), max_abs_corr=0.467276, sample=157 (original ≈ 3957)
  #3: key=0x36 (dec=54), max_abs_corr=0.404159, sample=157 (original ≈ 3957)
```

```
  #4: key=0x26 (dec=38), max_abs_corr=0.396176, sample=157 (original ≈ 3957)
  #5: key=0x3A (dec=58), max_abs_corr=0.378984, sample=22 (original ≈ 3822)

S-box 5:
  #1: key=0x13 (dec=19), max_abs_corr=0.420292, sample=125 (original ≈ 3925)
  #2: key=0x1A (dec=26), max_abs_corr=0.328361, sample=83 (original ≈ 3883)
  #3: key=0x20 (dec=32), max_abs_corr=0.296806, sample=170 (original ≈ 3970)
  #4: key=0x3C (dec=60), max_abs_corr=0.293844, sample=0 (original ≈ 3800)
  #5: key=0x35 (dec=53), max_abs_corr=0.289183, sample=113 (original ≈ 3913)

S-box 6:
  #1: key=0x2E (dec=46), max_abs_corr=0.342465, sample=125 (original ≈ 3925)
  #2: key=0x13 (dec=19), max_abs_corr=0.337430, sample=160 (original ≈ 3960)
  #3: key=0x31 (dec=49), max_abs_corr=0.291988, sample=114 (original ≈ 3914)
  #4: key=0x3E (dec=62), max_abs_corr=0.281428, sample=33 (original ≈ 3833)
  #5: key=0x0D (dec=13), max_abs_corr=0.279607, sample=169 (original ≈ 3969)

S-box 7:
  #1: key=0x21 (dec=33), max_abs_corr=0.561832, sample=169 (original ≈ 3969)
  #2: key=0x0F (dec=15), max_abs_corr=0.329776, sample=169 (original ≈ 3969)
  #3: key=0x23 (dec=35), max_abs_corr=0.320745, sample=169 (original ≈ 3969)
  #4: key=0x14 (dec=20), max_abs_corr=0.299129, sample=62 (original ≈ 3862)
  #5: key=0x19 (dec=25), max_abs_corr=0.284162, sample=125 (original ≈ 3925)

S-box 8:
  #1: key=0x03 (dec= 3), max_abs_corr=0.467150, sample=101 (original ≈ 3901)
  #2: key=0x1D (dec=29), max_abs_corr=0.318814, sample=61 (original ≈ 3861)
  #3: key=0x1E (dec=30), max_abs_corr=0.312545, sample=127 (original ≈ 3927)
  #4: key=0x2C (dec=44), max_abs_corr=0.306080, sample=27 (original ≈ 3827)
  #5: key=0x07 (dec= 7), max_abs_corr=0.301414, sample=64 (original ≈ 3864)
```

Find FULL_KEY

We start from the S-box candidates found by the CPA attack and stored in sbox_out.txt. Each S-box gives us a few possible 6-bit key pieces. We combine one candidate from each of the eight S-boxes to build all possible 48-bit round-1 subkeys (K1). Then we reverse the DES key schedule: first, we use the inverse PC2 table to expand each 48-bit K1 into 56-bit candidates, filling in the dropped bits in every possible way. Next, we undo the round-1 shift to get the original 56-bit key before the first round. Finally, we apply the inverse PC1 table to make a full 64-bit key, setting the parity bits to zero.

The script then tests each generated key with the known plaintext 4142434445464748 and checks if it encrypts to ef770c97ad062c75. The key that matches this ciphertext is the correct full DES key.

To run it, install pycryptodome (pip3 install pycryptodome)

FULL_KEY IS: 0x5AE0F272B862DA58

```
$ python3 find_full_key.py
```

```
[INFO] Loaded CANDIDATES_HEX from sbox_out.txt:
  S-box 1: ['0x27', '0x38', '0x2C', '0x25', '0x39']
  S-box 2: ['0x2F', '0x1B', '0x23', '0x20', '0x2D']
  S-box 3: ['0x19', '0x11', '0x2C', '0x03', '0x0C']
  S-box 4: ['0x0E', '0x26', '0x1E', '0x36', '0x02']
  S-box 5: ['0x13', '0x35', '0x26', '0x20', '0x1F']
  S-box 6: ['0x2E', '0x0D', '0x3E', '0x13', '0x27']
  S-box 7: ['0x21', '0x0F', '0x2E', '0x14', '0x1C']
  S-box 8: ['0x03', '0x08', '0x32', '0x02', '0x2C']
[INFO] Using top 3 candidates per S-box.

[+] Found a matching key!
    Key (hex) = 0x5AE0F272B862DA58
[INFO] Tested 249 candidate 56-bit keys (from 1 K1s).
[RESULT] Full 64-bit key (parity bits = 0): 0x5AE0F272B862DA58
```