AES Key Recovery Approach

Luka Kravos

Goal

The goal is to recover the high nibbles of an AES encryption key using cache timing data from a Prime+Probe side-channel attack. The attacker has access to plaintext, ciphertext inputs and cache access patterns. These access patterns come in the form of timing values for 4 AES lookup tables (each with 16 lines), resulting in 64 timing values per encryption.

The idea is that the key influences which table line is accessed, so if we can link a specific plaintext byte and cache line access with timing information, we can guess which key bytes are more likely. Since the accessed table line is determined by the upper 4 bits of the S-Box output (called the **high nibble**), we aim to recover only this part of each key byte. The high nibble is the most significant 4 bits of a byte. For example, in the byte `0x6f`, the high nibble is `0x60`.

Overview of Approaches

To solve this task, I implemented and tested three different methods:

1. **Sum-Max Approach:** This is a very simple method. It groups the timings for each guessed key byte and just sums them up per cache line. Then it takes the maximum value from those sums to decide the best guess.
2. **Sum-Avg Approach:** This version improves on the first one. It also groups timing values per guessed key byte and cache line, but instead of summing them only, it calculates the average access time for each line. This reduces the impact of outliers.
3. **CPA (Correlation Power Analysis):** This is the most advanced method I used. It builds a prediction matrix for each possible key byte and then uses correlation between the prediction and timing values. This helps to find the key with much more statistical confidence.

Each of these methods tries to guess the most likely high nibble of the AES key byte. I also generated heatmaps to visualize how well different key guesses correlate with timing measurements.

In the next sections, I explain each method in more detail, including how they process the input and how the key is recovered or visualized using heatmaps.

Sum-Max Approach

This was the first method I implemented. The goal was to find a simple way to recover the key using the cache timing data. For each key byte (there are 16 in total), I tested all 256 possible values from `0x00` to `0xFF`.

For each guessed key byte value:

- I XOR the guess with the plaintext byte to simulate what the AES would do.
- Then I take the top 4 bits of this result (this is the high nibble) and use it to select the expected cache line.
- I sum up the timing values for each cache line access.

The result is a 256x16 matrix: each row is a key guess, and each column is one of the 16 cache lines for the corresponding AES lookup table.

After that, for each row (key guess), I take the **maximum** timing value across all cache lines. The idea is: if a certain key guess causes access to one specific line more often, the timing sum for that line should be high.

I select the key guess that had the highest maximum timing as the best guess. This is done for every byte of the key. At the end, I keep only the **high nibble** of each guessed byte, which is the part we are interested in.

To better understand how good each guess is, I also generate **heatmaps**. In the heatmaps:

- The x-axis shows the cache lines (from L0 to L15).
- The y-axis shows only the 16 possible high nibble values (from 0x00 to 0xF0).
- The color shows how large the summed timing value is.

This method is very fast and simple, but not always very accurate. Looking at the generated heatmaps, I wasn't able to tell most of the time which key is the exact one, as multiple would have a very similar color. It does not average the results and does not consider how well the access pattern correlates with the prediction.

The key recovered with this one (only 64 bit of the 128 bit key / high nibbles only) is:
30 f0 10 90 f0 e0 30 30 60 50 d0 50 90 a0 50 80

Sum-Avg Approach

After trying the sum-max version, I realized that just summing up the timing values can be influenced by a few large values (outliers). So, I tried a better method that uses the average access time per cache line instead of only the sum.

Just like before, for each key byte (0 to 15), I go through all possible 256 guesses from `0x00` to `0xFF`. For every guessed value:

- I XOR it with the plaintext byte to simulate how AES would calculate the S-Box input.
- I take the high nibble (top 4 bits) of the result to predict which cache line will be accessed.
- I collect the timings for that line across all measurements and calculate the average access time.

This gives a 256x16 matrix like before, where each row is a key guess, and each column is one of the 16 cache lines. But this time, the values represent **average timing** instead of just the sum.

For each row (key guess), I take the maximum of the average timings across all lines. The idea is: if a certain guess causes consistent timing differences, the average will be higher. I then

select the key guess with the highest maximum average timing.

At first, I expected this method to give much better and clearer visual results than sum-max. But when I looked at the heatmaps, they were much harder to read. In many cases, the colors across all high nibble guesses looked almost the same. It was difficult to say which high nibble was the best one. As one would have the same color accross all high nibble key guesses for the specific cache line.

But, this behavior actually makes sense: the average timing does not depend on how often a line is accessed, only how slow the accesses are. So even if one key guess hits the correct line many times, its average can be the same as another guess that only hit it a few times. This "smooths out" the differences and makes the heatmap look more flat.

To better see which high nibble is most likely correct, I still generated heatmaps. The axis are the same as before, just now the color shows the average timing:

- The x-axis shows the cache lines (L0–L15).
- The y-axis shows the 16 possible high nibbles (0x00–0xF0).
- The color shows the strength of the average timing.

This approach gives more stable values than sum-max, but is not easy to read visually. In many heatmaps, multiple key guesses had similar average timing values, and it was not always clear which one is correct.

Even though the heatmaps from this approach looked more uniform and harder to interpret visually, the actual recovered key was in most cases more accurate than with the sum-max method. This is because averaging helps reduce the effect of noisy or extreme timing values. So in terms of key recovery performance, this method should have been better than the simpler sum-max version.

The key recovered with this one (only 64 bit of the 128 bit key / high nibbles only) is:
60 e0 80 10 00 90 00 50 60 00 b0 80 e0 70 b0 f0

CPA (Correlation Power Analysis) Approach

After searching on the web, how this is typically done, I came across this method, or at least this idea, which I then incorporated in my code. The last and most advanced method I used is based on CPA, which stands for Correlation Power Analysis. Unlike sum-max and sum-avg, this method uses statistical correlation to directly measure how well a guessed key explains the timing variations.

Again, for each key byte (0 to 15), I go through all possible 256 guesses from `0x00` to `0xFF`. For each guess:

- I XOR the guess with the plaintext byte to simulate the input to the S-Box.
- I then take the high nibble of the result to predict which cache line should be accessed.
- For each of the 16 possible cache lines, I create a binary mask. If a timing belongs to that line, the mask is 1, else 0.

- I then calculate the Pearson correlation between the mask and the actual timing values of that cache line.

This gives a 256x16 matrix for each key byte, where each cell contains the correlation between a guessed key and the access time for a specific line.

Then I take the **maximum correlation value** for each guessed key (max across the 16 lines), and the key guess with the highest such value is chosen as the best guess. I only keep the **high nibble** of the final guessed byte.

This method is much more reliable because correlation takes into account how closely the predicted access pattern matches the real timing data. It is not just based on sums or averages, but on statistical dependence.

As with the other approaches, I also generated heatmaps. But now they show the correlation strength for each guessed key and each line.

- The x-axis shows the cache lines (L0–L15).
- The y-axis shows the full 256 key guesses (0x00–0xFF).
- The color shows the absolute correlation value.

To make the results easier to read, I also generated a **combined heatmap** where I collapsed the guesses down to only 16 high nibbles. For each high nibble, I took the best correlation of all 16 values that share that nibble. This makes it easy to visually compare which high nibble is most likely for each byte. I also generated using the plotly library an interactive combined heatmap where one can hover over the fields and see the best guess per byte.

From the key guess, i extracted the correlation value for each of the 16 high nibble key parts in a csv file, to see how well each one scored.

The table below shows the final result of the CPA method for each key byte:

- The **Byte** column indicates the key byte index (0 to 15).
- The **HighNibble** shows the upper 4 bits of the guessed key.
- The **Correlation** value shows how well the prediction matches the observed timing behavior.

Byte	HighNibble	Correlation
0	10	0.00313
1	a0	0.00319
2	60	0.00344
3	10	0.00303
4	60	0.00281
5	c0	0.00307
6	60	0.00267
7	70	0.00261
8	30	0.00270
9	30	0.00276
10	30	0.00395
11	90	0.00368
12	40	0.00285
13	10	0.00334
14	d0	0.00296
15	f0	0.00298

Even though these correlation values may seem small (around 0.003), they are meaningful in the context of noisy cache timing data. The correct key byte often results in a consistently higher correlation than incorrect guesses.

This method clearly performed the best out of the three. The heatmaps showed strong contrast between the correct guess and the others, and the final key guessed (64 bit of the 128 bit key / high nibble part) should also be very accurate.

The key recovered with this one (only 64 bit of the 128 bit key / high nibbles only) is:
10 a0 60 10 60 c0 60 70 30 30 30 90 40 10 d0 f0