AES first round Prime+Probe attack

Filippo Pilutti - 12451650

May 24, 2025

1 Introduction

This document presents the methodology and analysis used to recover parts of the AES encryption key using a Prime+Probe side-channel attack targeting the first round of an AES implementation. The provided data, contained in the file `output.txt`, includes plaintext, ciphertext, and cache access timings across 64 sets, recorded for a large number of AES invocations ($\sim$ 1 million). The provided C code (`aesrun.c`) that generates the data, initializes the AES key and repeatedly encrypts randomly generated plaintexts, measuring the number of cycles needed to probe each cache set. The output logs each trace as a line:

```
Plaintext Ciphertext CL0 CL1 ... CL63
```

where each CLi indicates the access timing for cache set i.

2 Exploiting the first round of AES

To use the Prime+Probe side-channel attack, we exploit the structure of the first round of AES encryption, in particular its use of T-tables (T0, T1, T2, T3) for fast lookups during the SubBytes, ShiftRows and MixColumns operations. T-tables combine the AES operations into precomputed lookup tables to reduce the number of operations during encryption. The key property exploited is that each table spans exactly 16 cache lines: each table has 256 entries, each of size 4 bytes, assuming a 64-set cache with a 64-byte line size, a table occupies $(256 * 4)/64 = 16$ lines and each line contains 16 4-bytes table entries; therefore each T-table occupies contiguous cache sets and each table lookup maps a byte value to a specific offset in the T-table, and thus to one of the 16 cache sets. The main idea of the attack hence is: each cache set holds some lines of the table, then by using Prime+Probe an attacker can see which table entries are accessed during the first round of AES encryption.

2.1 Prime+Probe

1. **Prime**: attacker fills the cache sets corresponding to the tables with its own data
2. **AES encryption**: victim performs AES encryption on a known plaintext, and during the first round, it uses the tables to compute the round transformations. Specific table entries are accessed, which evict the attacker's data
3. **Probe**: attacker accesses the memory again and measures access times. If it records slow access times in particular sets, then his data was evicted, meaning that cache set was accessed during encryption, exposing which part of the table was used.

2.2 Key recovery

In the first round of AES, the encryption performs:

$$ciphertext[i] = Tj[plaintext[i] \oplus key[i]]$$

where $j = i \ mod \ 4$ selects which of the T-tables to use. This lookup uses the XOR between plaintext and key, producing a value $v = plaintext[i] \oplus key[i]$, which is used to determine which entry to use inside the corresponding T-table.

Since each table entry spans one cache line and the T-table spans 16 lines, the top 4 bits of v determine which cache set is accessed, but the bottom 4 bits only affect the offset within the line, which is not detectable using Prime+Probe, since it only has a resolution of cache set. Thus, when observing cache activity after encryption, we can only determine which cache set was accessed, and consequently, the top 4 bits of v (hence, of $key[i]$) can be computed by XORing the index of the table with the plaintext.

2.3 Statistical analysis

Since actual experiments have noise (like background processes accessing the cache), individual timing measurements are unreliable indicators of whether a particular cache set was accessed by the AES process. To mitigate this, a large number of traces is collected and grouped according to the 4 most significant bits (MSBs) of the corresponding plaintext byte. For each group, the average access time for every cache set is computed, and the global average is subtracted to highlight deviations indicative of cache misses. This statistical aggregation helps filter out the effects of random noise and transient disturbances, making genuine memory access patterns emerge more clearly. The larger the dataset, the more robust these average-based differences become, increasing the confidence in accurately identifying the cache set accessed and, consequently, the 4 MSBs of the key byte.

3 Recovery approach

3.1 Parsing

The `parser.py` module reads and converts each line of `output.txt` into an `AESData` object with plaintext, ciphertext and cache line timings

3.2 Averaging

The `compute_cache_line_averages` function inside the `utils.py` module computes the average access time for each cache set over all samples.

3.3 Grouping by the 4 MSBs (most significant bits)

For each plaintext byte index in (0-15), samples are grouped by the 4 most significant bits of the plaintext byte. For each group, the average cache line access time is calculated, with the function `compute_averages_for_plaintext_group`.

3.4 Data correction

The group averages are corrected by subtracting the global average to highlight relative differences (`averages_correction`).

3.5 Heatmap generation

The `generate_heatmap` function produces a visual heatmap for a specific byte index showing the timing per cache set for each 4 MSB group. This heatmap helps visually identify consistent cache misses correlated with specific bits values, and also to determine how the T-tables are mapped in the cache memory.

3.6 Cache miss detection

The cache set with maximum corrected access time for each 4 MSB group is considered the accessed set (`extract_cache_misses`).

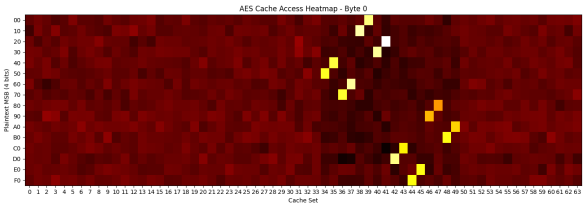

Figure 1: Example of generated heatmap for byte 0

3.7 Key recovery

Using the known structure of T-tables and their mapping to cache sets, we partially recover the key with the `recover_4msb_key_for_byte` function. By analyzing the generated heatmaps, the four T-tables are mapped to the cache sets in the following way:

- **T0**: sets [2 - 17]
- **T1**: sets [18 - 33]
- **T2**: sets [34 - 49]
- **T3**: sets [50 - 1]

so we are going to need to handle the wrap-around of the last table along the cache sets. For each byte of the plaintext, the key recovery proceeds following these steps:

1. Identify the specific T-table used by this key byte, considering the relative offset.

```
table_number = (byte_index % 4)
base_table = ((table_number + TTABLE_OFFSET) % 4) * 16 + TTABLE_OFFSET
```

2. For each group of 4 MSBs of the plaintext:

   - get the cache set index of the relative cache miss

   ```
   cache_miss_set = cache_misses[plaintext_msb]
   ```

   - infer the specific table-relative index used in the T-table

   ```
   table_set_index = (cache_miss_set - base_table) % SETS
   ```

   - recover the partial key candidate by XORing the T-table index with the 4 most significant bits of the plaintext byte for the key byte and add it to a list of candidates

   ```
   candidate_key_msb = table_set_index ^ plaintext_msb
   candidates.append(candidate_key_msb)
   ```

3. We now have 16 candidates for the 4 most significant bits of the key byte. The correct key should appear more often than the others, so we extract the most frequent candidate.

```
key_msb = Counter(candidates).most_common(1)[0][0]
```

4. We return the recovered partial key byte as an hex string.

```
return f"0x{key_msb:X}"
```

3.8 Result and recovered partial key

The script `extract.py` automates the above steps for each byte of the plaintexts and outputs the recovered 4 most significant bits of each key byte to `key.txt`.

Recovered key

`[0x5,0x5,0x6,0xB,0x4,0x4,0x9,0x2,0xE,0x5,0xD,0x2,0x0,0xB,0x5,0x6]`

Note: only the 4 most significant bits of each byte are recovered, as the Prime+Probe attack resolution is limited to cache sets.

3.9 Attack flow: summary

For each plaintext byte index `i`:

1. Group the measurements based on the 4 MSBs of `plaintext[i]`, resulting in 16 groups
2. For each group, compute the average access time per cache set, obtaining a 64-element vector
3. Normalize the data by subtracting the global average access time across all samples to highlight relative delays (probable cache misses)
4. Identify the cache set with the highest delay per group (the one that was probably accessed by the victim)
5. From the cace set index and known T-table structure, reverse the XOR and recover the 4 MSBs of `key[i]`

By doing so, we exploit the fact that **lookup index = plaintext** $\oplus$ **key**, and that the index determines the accessed cache set.

4 Recovering the 4 LSBs (least significant bits) of the key

To recover the 4 least significant bits of each key byte, an attacker should move beyond observing cache sets and instead aim to observe cache lines within those sets, or leverage additional side-channel information. Next are few viable options that could be used to achieve full-key recovery.

Brute-force approach using ciphertext

Since we have the plaintext-ciphertext pairs and we have already recovered the 4 MSBs of each key byte, for each byte k_i we could generate 16 candidate values by keeping the 4 found MSBs fixed and trying all 4-bit LSB combinations and use each of these candidate full key byte to simulate the AES encryption. By comparing the resulting output with the actual ciphertext, the correct LSBs should produce a match when combined with the known MSBs.

Attack further rounds

Another possibility is to extend the attack to later AES rounds: by analyzing access patterns and correlating them with intermediate states derived from partial key guesses, it's possible to recover additional bits.

Use higher resolution attacks

If the attacker shares the memory with the victim, a Flush+Reload attack can reveal which specific cache line was accessed, allowing to recover the full byte value that was used in the table lookup and thus allowing to infer the whole key.

5 Conclusion

This method successfully demonstrates a Prime+Probe side-channel cache attack targeting the first AES round. This attack recovered the upper 4 bits of each byte of the AES key, and suggests some further methods to achieve full-key recovery. The approach is systematic and reproducible, with heatmaps and statistical analysis supporting the key recovery.

A Program usage

Inputs

- `output.txt`: a file containing the plaintext, ciphertext, cache access timings data. Each line corresponds to one AES invocation and has the format

  ```
  Plaintext Ciphertext CL0 CL1 ... CL63
  ```

- (optional): the path to `output.txt` can be specified via command-line

```
python extract.py /path/to/output.txt
```

  If no path is given, the program defaults to `output.txt` in the current directory

Outputs

- `key.txt`: a list of 16 hexadecimal values corresponding to the 4 most significant bits of each key byte.
- `heatmaps`: a directory containing the 16 generated heatmaps in PNG format, one for each byte of the plaintexts. Each heatmap highlights which cache set was evicted for each group of 4 MSBs of the plaintext, aiding interpretation and verification.