Submitted by Germain Mucyo
Email: mucyo.g@northeastern.edu
NUID: 002301781

Lab 5 - Flush+Reload Attack against AES

1 Overview

In this lab, I learnt a widely used cache side-channel attack: Flush+Reload attack.

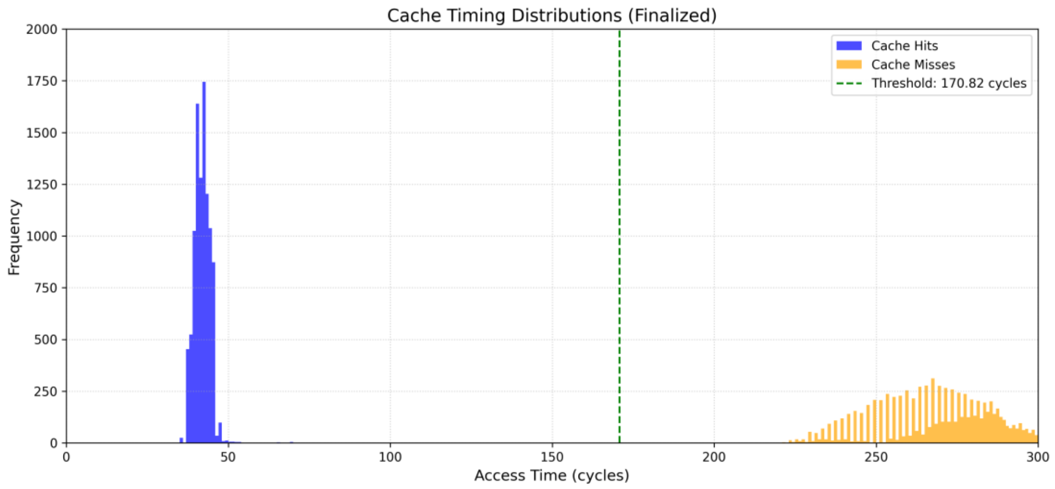
The objective of this assignment was to:

1. System Profiling, Benchmark and analyze cache timing behavior to differentiate between cache hits and misses.
2. Implement a Flush+Reload attack to extract cryptographic keys from a victim AES binary.
3. Visualize and validate the results of both the benchmark and the attack process.

2 System Profiling

Cache hits and misses were measured using a custom benchmark program, **see cashe_benchmark.c**

- **Hits:** Access times were measured for data preloaded into the L3 cache. **See hits_data.txt**
- **Misses:** Access times were measured after flushing cache lines to force main memory access. **misses_data.txt.**



This graph depicts the cache timing distributions for cache hits and misses, which are used to analyze the behavior of the cache hierarchy. The threshold value of 170.82 cycles is drawn to separate cache hits and misses. Access times below the threshold are classified as hits, while those above are classified as misses.

3 Attack - Online Phase: Collect Time Samples

Question 1. For your samples, are the outliers in the same column of subplots (on four cipher/key bytes), or actually in two columns? If you observe the latter case, explain it and describe how you can utilize such information leakage to recover additional key bytes (refer to slide 61 of Lecture 5)?

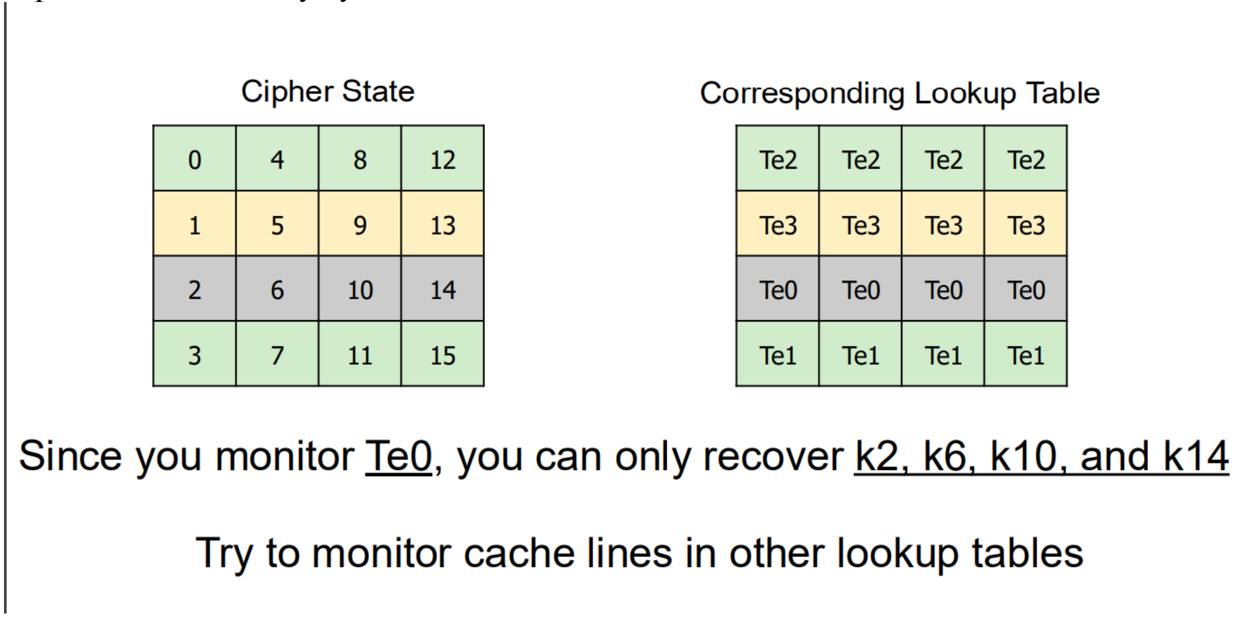
Ans:

For the samples collected during the Flush+Reload attack, the observed outliers appear in **two subplot columns**. This indicates that the cache timing behavior reveals information not only about the targeted T-table (e.g., Te0) but also leaks information about other key bytes due to the AES mix-column transformation and interdependencies between cipher state bytes.

By observing outliers in multiple columns, we can recover additional key bytes by monitoring **different T-tables** (Te1, Te2, Te3). Each T-table corresponds to specific cipher state positions (Kj), as illustrated in the "Cipher State to T-Table Mapping" (slide 61 of Lecture 5). Using this information leakage across columns, I implemented code to iterate over all T-tables, allowing the recovery of all 16 last-round key bytes.

4 Attack - Offline Analysis: Recover Last Round Key Bytes

In this case as the slides suggests I used Te0 to recover k2,k6,k10,k14. Below the screenshot depicts 4 recovered Key bytes.



Cipher State

0	4	8	12
1	5	9	13
2	6	10	14
3	7	11	15

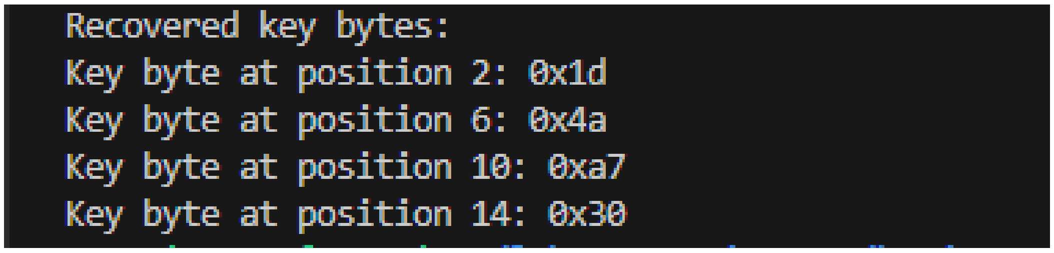
Corresponding Lookup Table

Te2	Te2	Te2	Te2
Te3	Te3	Te3	Te3
Te0	Te0	Te0	Te0
Te1	Te1	Te1	Te1

Since you monitor Te0, you can only recover k2, k6, k10, and k14

Try to monitor cache lines in other lookup tables

Figure 1 T-tables with their Kj(cipher state)



```
Recovered key bytes:
Key byte at position 2: 0x1d
Key byte at position 6: 0x4a
Key byte at position 10: 0xa7
Key byte at position 14: 0x30
```

Figure 2 Recovered Key bytes

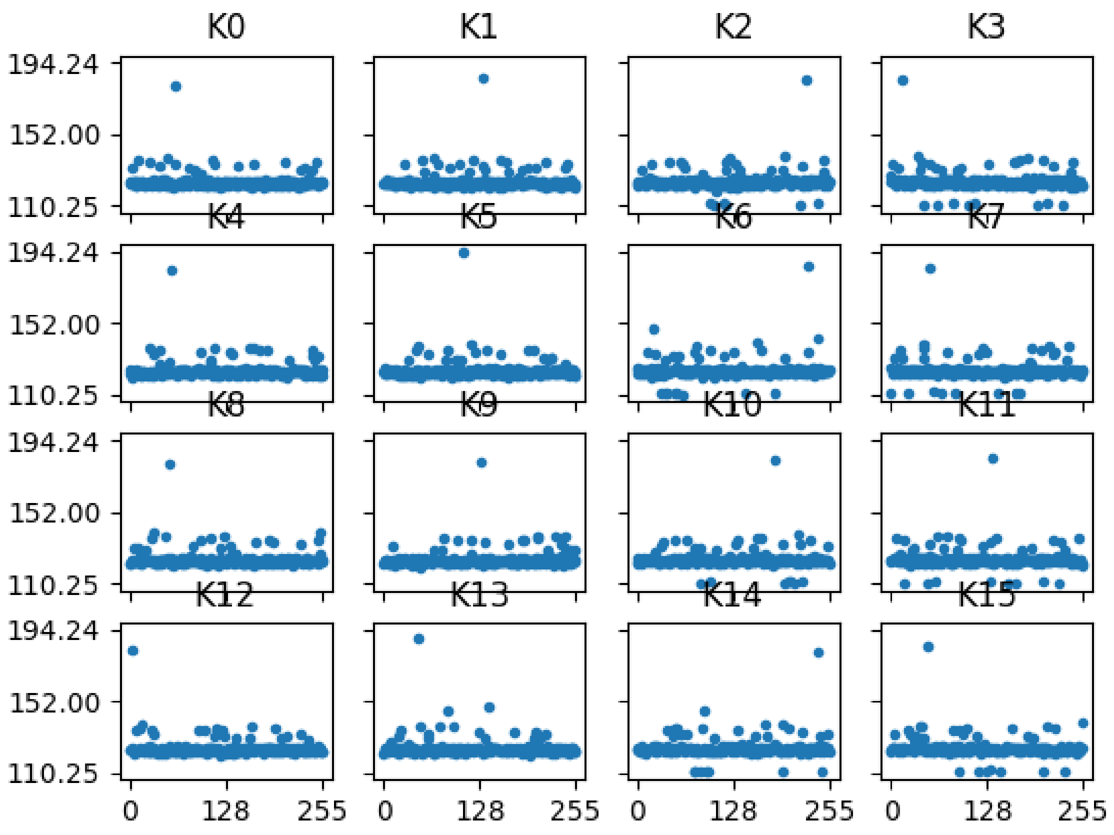
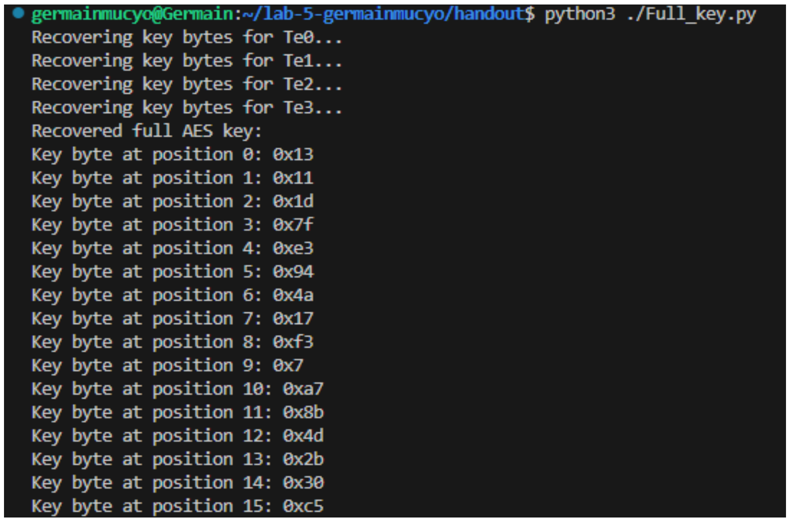


Figure 3, the corresponding plot of recovered key bytes

5 Extra Credits

To recover all 16 last-round key bytes, I implemented the code in **Full_key.py**, which loops through all four T-tables (Te0, Te1, Te2, and Te3). For example, in lab5.py, I focused on Te0 to recover only 4 key bytes (k2, k6, k10, and k14). Subsequently, I processed the other T-tables (Te1, Te2, Te3) to recover their corresponding key bytes (k3, k7, k11, k15, etc.), completing the full 16-byte key by combining all results.



```
germainmucyo@Germain:~/lab-5-germainmucyo/handout$ python3 ./Full_key.py
Recovering key bytes for Te0...
Recovering key bytes for Te1...
Recovering key bytes for Te2...
Recovering key bytes for Te3...
Recovered full AES key:
Key byte at position 0: 0x13
Key byte at position 1: 0x11
Key byte at position 2: 0x1d
Key byte at position 3: 0x7f
Key byte at position 4: 0xe3
Key byte at position 5: 0x94
Key byte at position 6: 0x4a
Key byte at position 7: 0x17
Key byte at position 8: 0xf3
Key byte at position 9: 0x7
Key byte at position 10: 0xa7
Key byte at position 11: 0x8b
Key byte at position 12: 0x4d
Key byte at position 13: 0x2b
Key byte at position 14: 0x30
Key byte at position 15: 0xc5
```

Figure 4 Results of recovered full AES key

1. (5 points) Implement a different attack strategy from the one you have chosen: keep samples with lower or higher timing, and use argmax or argmin for the counters. Discuss the pro and con of each strategy.

Ans:
For the alternate strategy, I implemented **argmin.py**, which focuses on samples with higher timing values (cache misses) and uses np.argmin to determine the most likely key byte by selecting the candidate with the minimum counter value. The recovered AES key was the same as the one obtained using the original strategy (argmax with lower timing). The argmin approach was faster because it processed a smaller dataset by filtering higher timing values. However, it is more susceptible to noise from unrelated delays, making it less reliable than the original approach. The argmax method is more accurate due to lower noise but slower with larger datasets.