Secure Processor MicroArchitecture
Assignment 4

Team Members:
Surya Prasad S - EE19B121
Sai Dheeraj Ettamsetty - CS18B055

1. Working of the time-driven attack code
   Theory behind time-driven attacks on AES T-Table implementation:
   a. Plaintext is generated randomly with the first four bytes set to a fixed value.
   b. Before encrypting, clear the cache so that all the data accessed will be a miss.
   c. Each cache line contains 16 entries of the T-tables. If two requested memory accesses are in the same cache line then the total number of misses will be lesser.
   d. Time the encryption process and store it in an array.
   e. Repeat the process for multiple iterations and find the maximum deviation for each byte of the plaintext.

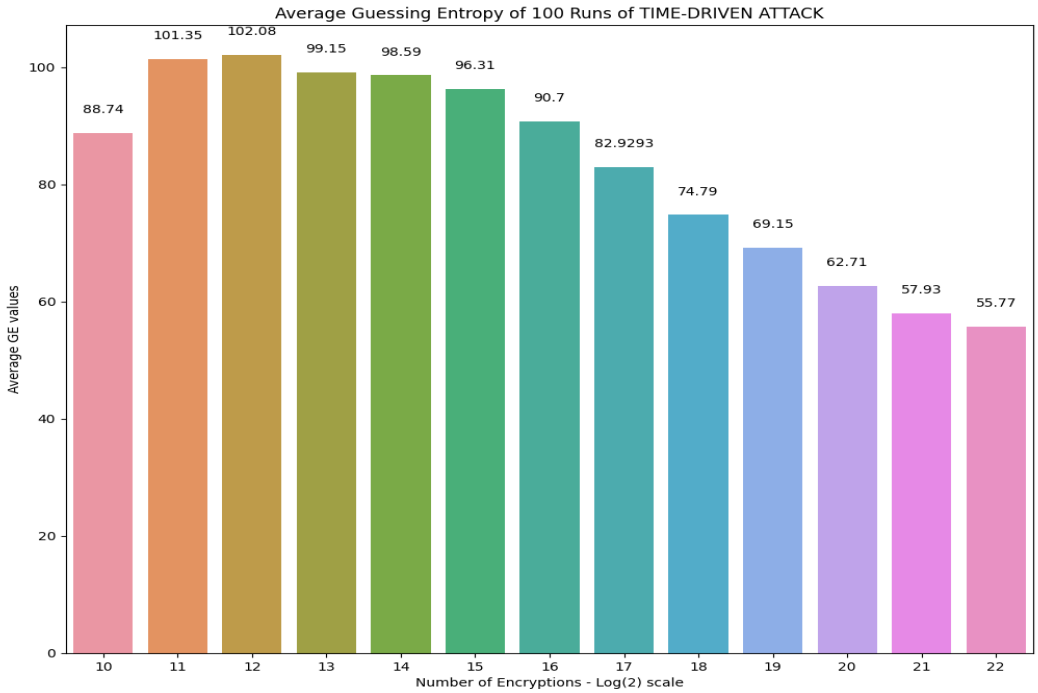
   Working of attack.c code;
   a. Plaintext is set randomly and the 4-MSB of the first four bytes are set to 0. We are also given that the first four bytes of the key are 0. So we can simply conclude in the end that the distinguished plaintext will be equal to the key (at least the first four bits).
   b. Cache is cleared of any entry of the T-tables.
   c. Encrypt and time the process.
   d. For each byte (from the fourth byte), we accumulate the time taken for encryption corresponding to each cache set (which is (PT_i ^ K_i) >> 4). This is not the best way to time as we update all bytes irrespective of the cache miss corresponding to that byte.
   e. In the previous step, we also don't include outliers in time taken due to context switching. "TIME_THRESHOLD" is set to remove this.
   f. We also count the number of times a cache set is accessed in each encryption so that the average time for access can be computed.
   g. Finally we compute the average time taken for accessing each cache set called `tavg[c][ ]` for byte `c` and for all accesses called `tavgavg` for the same byte `c` and we find the deviation for each cache set. If

2. CPU details
   Device: Lenovo Legion Y540
   Number of CPUs: 12
   Number of cores: 6
   L1 cache size: 32KB
   L1 associativity: 8

L1 sets: 64

CPUs 0,4 and 1,5 and 2,6 and 3,7 share the same core

3. Plot of average Guessing Entropy vs Number of encryptions for Time-Driven Attack



4. Converting Time-drive attack to Evict+Time

Theory of our approach:

   a. For evict+time, one cache set is targeted to fetch all the key bytes (the first four bits of each byte). It is given that all the T-tables are 12-bit aligned so all tables start from cache set 0. We basically target all those plaintext bytes which make the AES access that cache set.
   b. For each byte, we do an encryption then evict data of the corresponding T-table from that cache set and repeat the same encryption. If we evict the T-table data used by the encryption then it will be a miss and time taken will be higher.
   c. Repeat the above process for multiple iterations and time the second encryption. The time taken is again analyzed in the similar way as in the timing attack.

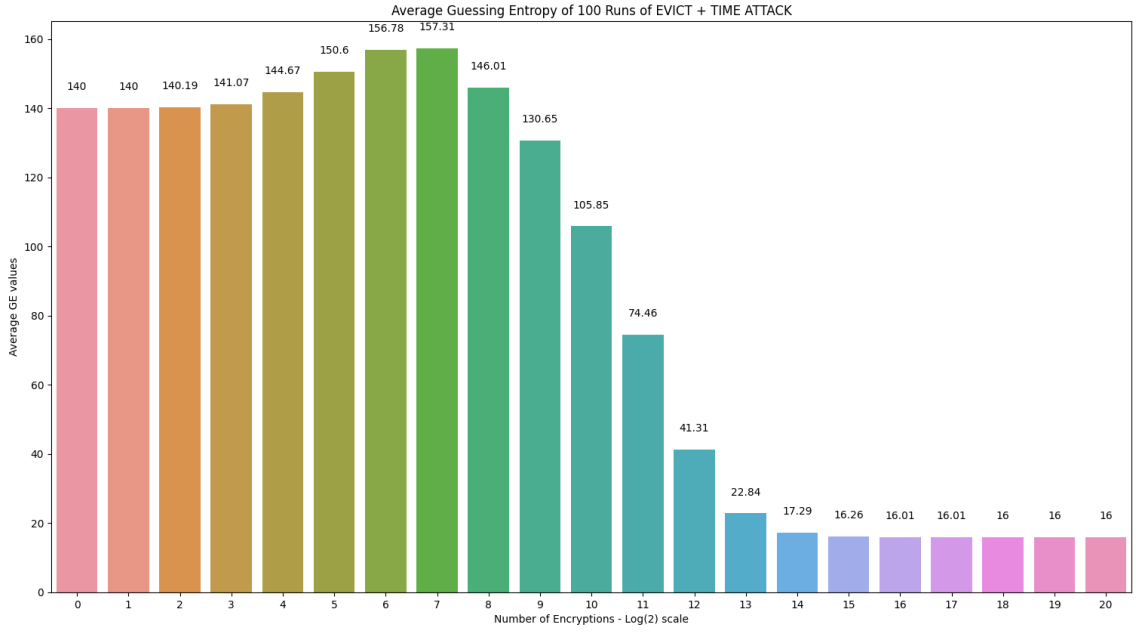
Codes which were modified: A new function, void clean_table_idx(int idx, int table), was added to the “aes_1024.c” file and also declared in “common.c”. “Attack.c” was modified to “evict.c” with the following snippet showing some of the changes.

Snippet of the code to perform evict+time attack vs time-driven attack:

```
for(int bytenum = 0; bytenum < 16; bytenum++) {
    int tablenum = bytenum%4;
    fprintf(f, "\n\n*******Attacking byte %d of Rnd1 key*******\n", bytenum);
    printf("Byte-%d\n", bytenum);

    int ii=0, idx=0;    // Idx need not be a range as plaintext is random - Any value
    from 0 to 255 will work. idx is T-table index

    while(ii++ <= (ITERATIONS)){
        /* Set a random plaintext */
        for(i=0; i<16; ++i) pt[i] = random() & 0xff;

        // Begin one encryption to put data into cache
        AES_encrypt(pt, ct, &expanded);
        asm volatile("mfence");

        clean_cache_table_idx(idx, tablenum);

        // In this encr, if data got removed then it will take more time
        start = timestamp();
        asm volatile("mfence");
        AES_encrypt(pt, ct, &expanded);
        asm volatile("mfence");
        end = timestamp();

        timing = end - start;

        if(timing < TIME_THRESHOLD){            // Removing outliers due to context
        switch
            /* Record the timings */
            // idx = pt[0] ^ kg[0]
            ttime[bytenum][(pt[bytenum] ^ idx) >> 4] += timing;
            tcount[bytenum][(pt[bytenum] ^ idx) >> 4] += 1;
        }

        /* print if its time */
        if (!(ii & (ii - 1))) {
            printf("%08x\t", ii);
            findkeys(bytenum);
        }

        if (ii == ITERATIONS) {
            for(int i=0; i<16; ++i) {
                fprintf(f, "%d  %.3f  %.4f\n", i, tavg[bytenum][i], deviations
                [bytenum][i]);
            }
        }
    }
}
int GE = 0;
```

```
while(ii++ <= (ITERATIONS)){
    /* Set a random plaintext */
    for(i=0; i<16; ++i) pt[i] = random() & 0xff;
    /* Fix a few plaintext bits of some plaintext bytes */
    pt[0] = pt[0] & 0x0f;
    pt[1] = pt[1] & 0x0f;
    pt[2] = pt[2] & 0x0f;
    pt[3] = pt[3] & 0x0f;

    /* clean the cache memory of any AES data */
    cleancache();

    /* Make the encryption */
    start = timestamp();
    AES_encrypt(pt, ct, &expanded);
    end = timestamp();

    timing = end - start;

    if(ii > 1000 && timing < TIME_THRESHOLD){
        /* Record the timings */
        for(i=4; i<16; ++i){
            ttime[i][pt[i] >> 4] += timing;
            tcount[i][pt[i] >> 4] += 1;
        }

        /* print if its time */
        if (!(ii & (ii - 1))){
            printf("%08x\t", ii);
            findkeys();
            printtime(4);
        }
    }
}
```

5. (Full code is given along with the report)
6. Plot of average Guessing Entropy vs Number of encryptions for Evict+Time attack



7. Attack on round 2 - Two versions were made
   Version 1: 4-bit attack
   Approach:
   a. For reducing noise due to the first round, generate random PT such that all the table data used in round 1 are all in one cache set (idx). This can be easily done by generating the plaintext by XORing the correct key bits (from round 1 attack) and index.
   b. Set the target index of the cache set. Suppose the target is the first byte of the round 1 key, change the PT[0]'s last four bits randomly over multiple iterations and time the process. By changing the last four bits, we get that change to be visible in one value in round 1 and that affects all the values in round 2.
   c. Repeat over multiple iterations.
   d. Repeat this for all the bytes.

   Reasoning: This is a very weak attack and primary expectation is that for each target idx we identify the most deviating value and then map and identify the rest of the bits of the key (not done in the code)

   Results: The guessing entropy was around 120-140 which is similar to guessing the bytes randomly. Here the number on the right of the brackets is the guessing entropy. In this result two bytes were fetched accurately.

   RESULTS
   00(0) 13 01(7) 12 02(c) 9 03(0) 2 04(8) 5 05(7) 12 06(4) 7 07(8) 1 08(c) 7 09(0) 3
   10(c) 13 11(d) 4 12(d) 1 13(0) 14 14(5) 8 15(5) 14
   Net GE=125

   Version 2: 32-bit attack
   Approach:
   a. Set the plaintext such that round 1 table data goes to one cache set (idx)
   b. For each byte and target table, we find bits corresponding to two target indices of the cache
   c. To find the first key byte, we randomly change the lower bits of other bytes (which don't affect round 2 like in this case 0, 5, 10, 15 affect). We also change the lower bits for one of the bytes which affects round 2 (say 0).
   d. We iterate this multiple times and store the time against the 4 bits of Byte 0.
   e. We repeat the process with another target index and find the most deviating value
   f. We know the following:
      <Target index> = <t0> = <Te0[pt[0] ^ k[0]] ^ Te1[pt[5] ^ k[5]] ^...> is ideally true for the most deviating value
      The part from Te1 is a constant across both the iterations so we take XOR of the

two equations with the two target indices. In this we substitute different values for the LSB 4-bits of the target key byte.

g. Repeat for other key bytes and tables

Reasoning: On paper this is a much better attack but also requires a well deviated result in both the iterations for the key guess to match. It is also harder to tell if the deviating plaintext byte is a false deviation. One modification which was made for this was to get multiple deviating values and test the combinations.

Results: Only one byte and table was targeted as there were a lot of changes required for each byte and table. Deviation was not significant and changed with each execution and key.

```
1st Iteration log

0  850.414  0.2481
1  850.083  0.0821
2  850.194  0.0286
3  849.791  0.3747
4  850.788  0.6224
5  850.503  0.3372
6  850.308  0.1427
7  849.831  0.3343
8  850.526  0.3601
9  849.266  0.9000
10  850.423  0.2577
11  849.775  0.3902
12  850.239  0.0731
13  850.366  0.2001
14  849.843  0.3223
15  850.294  0.1284

2nd Iteration log
0  826.809  0.4204
1  824.868  1.5207
2  827.206  0.8169
3  827.298  0.9091
4  826.844  0.4553
```

```
5  826.453  0.0645
6  825.805  0.5842
7  826.445  0.0564
8  825.766  0.6228
9  826.591  0.2026
10  826.629  0.2400
11  826.166  0.2223
12  826.164  0.2244
13  826.601  0.2121
14  826.347  0.0415
15  826.221  0.1674
```

Multiple values of plaintext can match and give the right key but the pair identified above is not always giving the right key byte. So in the best scenario we get only 1 best key guess and in the worst case we could get 10 choices too.