L1 Cache misses Attack ~ Cache Side- Channel Attack ~ Bruno Lopes

Index

General Attack's Idea 2

Type of Attack 2

CPU details 2

AES T-box Implementation & Encryption Modes Notes - (Background) 3

Measurement Phase 3

- Technology & Results 4
- File Structure - Fig. 1 5

Crypto-analysis Phase 5

- Round 1 attack sub-phase 6
- Round 2 attack sub-phase 7
- Results & Technology 8

Possible Optimizations & Notes 8

Github Repository:https://github.com/Gil96/cache_side_channel_attack

- General Attack's Idea

Our attack's purpose is to uncover the whole secret key that supports AES encryptions by a victim on a system, extracting and analysing side-channel information from L1 cache.
This attack is divided in 2 different phases: Measurement phase and Crypto-analysis phase, where the first phase serves to extract side-channel information to be consumed by the last one fulfilling our goal.

- Type of Attack

This attack exploits Aes 128-bit encryptions from OpenSSL 0.9.8a using T-Box tables i.e.: 5 1KB sized encryption tables, each containing 256 4B elements.
The targeted encryption mode is the Electronic CodeBook using the Full Unrolled Loop technique.
Our side-channel information is the number of L1 cache misses per L1 line for a different 16B plaintext values. PAPI 6.0.0 is the chosen profiler to register the number of PAPI_L1_DCM events which represent the number of L1 cache misses.
The attacker is supposed to choose the plaintext to be encrypted and know at least one T-box table memory address. The same way T-box tables are expected to be aligned on memory block boundaries.
Beforehand the attack also needs to know some CPU micro-architectural details, such as the L1 size, L1 associativity nº and L1 cache line size[1] for accomplishing the measurement phase.

- CPU details

  The cpu used for this attack and respective micro-architectural details are:

  Cpu Model name: Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz
  HyperThreading Tech.: yes
  Hardware Threads: 8
  Number of Cores: 4
  L1-d/i cache line size: 64B
  L1-d/i Associativity nº: 8
  L1-d/i lines nº: 64
  L1-d/i size: 32KB
  Write-policy: Write-Back
  Replacement-policy: Pseudo-LRU

[1] These can easily be extracted from a program accessing and timing each different sized array with a different sized stride, and comparing results - (such as the OC lab2).

- AES T-box Implementation & Encryption Modes Notes - (Background)

AES-128bit encryptions is a cryptographic process that receives a 16B plaintext, apply a set of secret key based transformations for 10 consecutive rounds, outputting a 16B ciphertext.
Since the same set of transformations are performed in the first 9 rounds (i.e.: SubBytes, ShiftRows, MixColumn, AddRoundKey), a AES T-box based implementation will use T0, T1, T2, T3 tables for the mentioned rounds. On the other side, the last round is computed differently from others (absence of MixColumn), and for that reason it uses a distinct table named T4.

Electronic CodeBook mode assures each encryption ciphertext depends solely on the received plaintext and the used secret key. Contrary to other modes, such as the Cipher-Block Chaining, where each encryption no longer depends on the plaintext and secret key but also the ciphertext from the last encryption.

- Measurement Phase

The general idea behind our measurement phase relies on having 2 different processes running in the same core at the same time. 1st process dutty is to perform some accesses to its data and registering the total number of cache misses. 2nd process calls victim's AES encryptions interfering with the measurements that are being performed by the 1st program. In other words, 2nd program by loading its data will evict or not the data in L1 previously loaded by the 1st program, which may or may not cause a cache miss when the same data is accessed by 1st program.

In detail, we have a process running in user mode called atk.c, that will perform a constant number of measurements. For each measurement atk.c will fork itself having its forked child run the code on vic.c, passing a random 16B plaintext. While vic.c is triggering several victim's encryptions inputting the same received plaintext, atk.c is filling[2] multiple times a specific L1 line with its own data for a constant number of times registering the respective the total L1 cache misses detected.
In the end of a single measurement, atk.c detains the number of L1 cache misses per L1 line and respective plaintext used on the vic.c. This information resulted from a several different plaintexts consists on the side-channel information produced on this phase. See Fig.1.

[2] To fill a specific line, it was required to perform W (read + write) on the data from different cache blocks, where each block is the way's dimension apart from the next one. (W represents the associative number of L1 cache).

- atk.c

Atk.c starts by modifying the CPU affinity to make it run on the hardware thread 3[3], using sched_setaffinity() from <sched.h>. Then, it configures PAPI, which involves papi initialization and L1 cache miss event creation. Next it generates a random 16-B plaintext using the function srand() from <stdlib.h>. Afterwards atk.c forks its execution creating a child process using the function fork() from <unistd.h>.

This child process switch process image to vic.c invoquing execv() from <unistd.h> passing the previous plaintext generated as argument. Vic.c details are specify in the vic.c tab below. On the other side the parent atk.c continues normally its execution.

Then atk.c sleeps for 15'000 microseconds using usleep() from <unistd.h> with the purpose of waiting for vic.c to finish its configuration[4]. Afterwards, it starts to fill[5] each L1 line for 10'000 times[6], resorting on papi for recording the total nº of PAPI_L1_DCM events, (i.e.: the nº of cache misses) for that particular line. At the same time vic.c is performing encryptions, which will influence the results extracted by atk.c.

Each L1 line's nº of cache misses and respective measurement plaintext ends up being written in the meas.out file of that particular measurement. This process represents a single measurement, and I choose to be repeated for 500 times (i.e.: 500 measurements), with the purpose to gather 500 meas.out samples necessary for recovering the whole secret key on the crypto-analysis phase.

After atk.c writes on meas.out, it closes the respective file using fclose() from <stdio.h> and waits by victim to finish using wait() from <wait.h>.

- vic.c

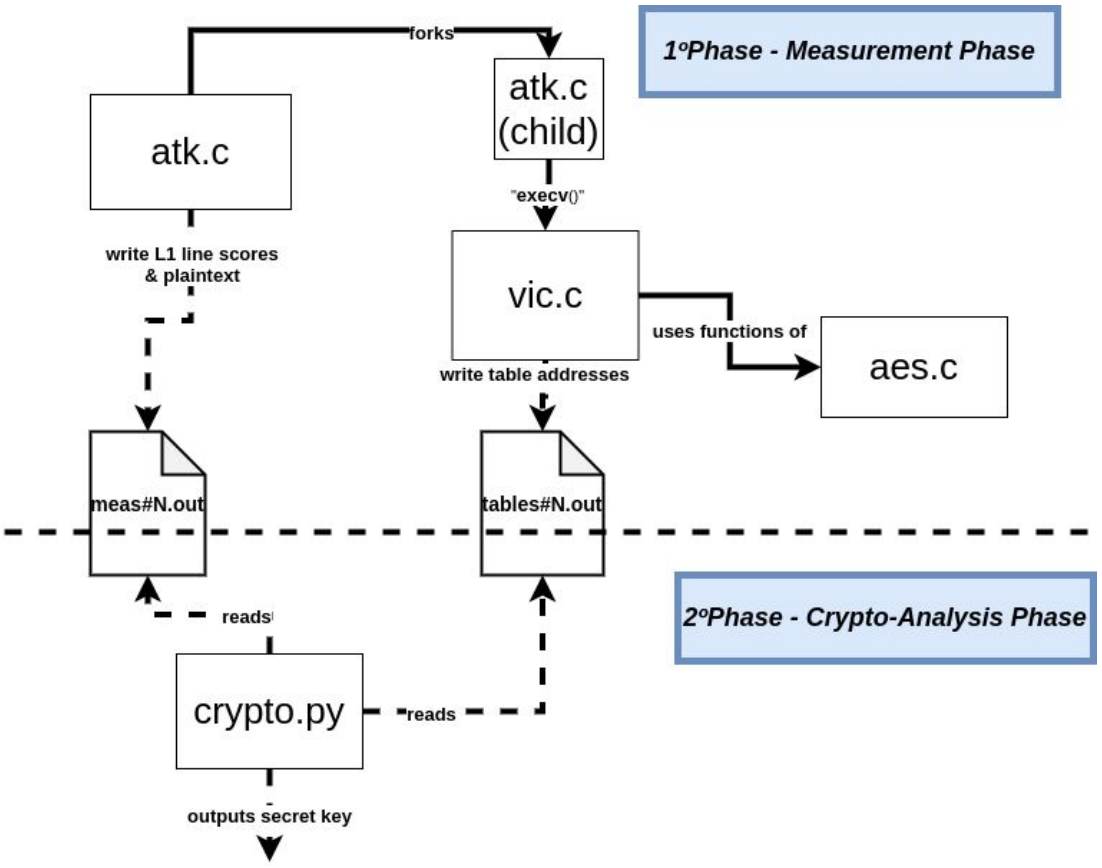
Likewise atk.c, vic.c modifies the CPU affinity to make it run on the hardware thread 7. It starts by receiving the plaintext from atk.c, creating an hardcoded aes key and consequently generate the respective round key using the AES_set_encrypt_key() function[7] . Afterwards, AES_print() - a function implemented by me, is called which creates a file called tables.out containing the 4 table addresses[8]. Keep in mind the, the attacker's knowledge of the table addresses information is an assumption of this attack, which justifies the existence of this function.

[3] Keep in mind CPU used on the experiments has Hyperthreading technology
[4] This sleep time has to be bigger enough to make sure atk.c starts to measures when vic.c is already triggering encryptions
[5] To fill a specific line, it is required to load data from W cache blocks, where each block is the way's dimension apart from the next one. (W represents the associative number of L1 cache).
[6] Around 5'120'000 Reads and Writes performed
[7] Function originally from openssl-0.9.8a/crypto/aes/aes_core.c
[8] The attack could also work with only 1 table address exposed

Finally, function AES_encrypt()[9] is going to be called 1'000'000 times[10] inputting the same plaintext. Note all these 3 AES functions are placed inside aes.c file.

During a single execution of atk.c, It has been created 500 executions of vic.c corresponding to a total of 500'000'000 encryptions. The measurement phase's (or atk.c) total elapsed time took 1.2min (this time result corresponds to the measurement average time of a total of 5 attacks against a different key victim).

- File Structure - Fig. 1



- Crypto-analysis Phase

This phase will take the data extracted from measurement phase, and use it to exploit some AES equations in order to uncover the secret key that supported the previous victim's encryptions. This phase is divided into a round-1 attack phase, that exploits the table indices equations used on 1-Round and a round-2 attack phase, exploiting the equations used on 2-Round.

[9] Function originally from openssl-0.9.8a/crypto/aes/aes_core.c
[10] Each AES encryption contains around 55 Writes + 240 Reads, totalizing in 55'000'000 Reads and Writes

1. Round 1 attack sub-phase

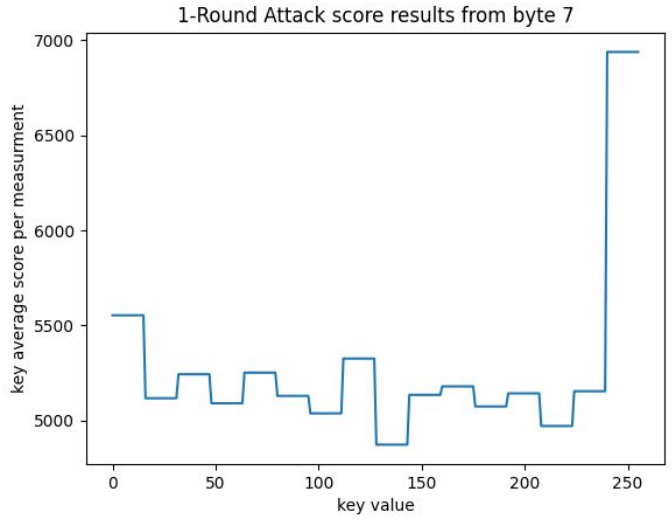
Let's represent each possible 256 key bytes value from each 16 key byte by the name of hypothetical key.
Thus, this strategy takes in consideration each hypothetical key, associating it to a respective single score value.
For each different measurement, the attacker knows the plaintext (from meas.out) and using the table indices equation for the 1st AES round is able to generate an hypothetical index. Remember that the 1º round table accesses depend solely on the particular key and plaintext byte, as demonstrated below:

$$x_i^{(0)} = p_i \oplus k_i \ (i = 0, \ldots, 15)$$

At this point, we have an hypothetical index and since we know the table addresses(from tables.out) we are able to extract the respective hypothetical L1 line. In other words, our hyp. index will access a certain table element that is mapped by the hypothetical line. Then we read our measurements meas.out file and retrieve the score of the L1 line given by the hypothetical line.
This score will be registered to be averaged with the previous scores from other measurements linked to that particular hypothetical key.

This process is repeated over the existing measurements updating each key byte hypothetical value's score.
When all the measurements have been considered, we choose from each byte the key bytes values with the highest score. In our attack, for each key byte, 16 values[11] are identified with the highest score.

Fig.2



Take for instance the 1-Round Atack results for key byte 7, getting the high score on the key values between [240 - 255] for a secret key equals to [FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF].

[11] 16 is our delta value, i.e.: each cache block contains a maximum of 16 table elements.

2. Round 2 attack sub-phase

At this point only the less 4 significant bits of each key byte are unknown. As mentioned before, this sub-phase is going to take a different approach in comparison with the last one. Looking to all the 2nd round equations from the specification, we pick the ones that depend on the less number of unknown bits, which are:

$$
\begin{aligned}
x_2^{(1)} &= s(p_0 \oplus k_0) \oplus s(p_5 \oplus k_5) \oplus 2 \bullet s(p_{10} \oplus k_{10}) \oplus 3 \bullet s(p_{15} \oplus k_{15}) \oplus s(k_{15}) \oplus k_2 \\
x_5^{(1)} &= s(p_4 \oplus k_4) \oplus 2 \bullet s(p_9 \oplus k_9) \oplus 3 \bullet s(p_{14} \oplus k_{14}) \oplus s(p_3 \oplus k_3) \oplus s(k_{14}) \\
&\quad \oplus k_1 \oplus k_5 \\
x_8^{(1)} &= 2 \bullet (p_8 \oplus k_8) \oplus 3 \bullet s(p_{13} \oplus k_{13}) \oplus s(p_2 \oplus k_2) \oplus s(p_7 \oplus k_7) \oplus s(k_{13}) \\
&\quad \oplus k_0 \oplus k_4 \oplus k_8 \oplus 1 \\
x_{15}^{(1)} &= 3 \bullet s(p_{12} \oplus k_{12}) \oplus s(p_1 \oplus k_1) \oplus s(p_6 \oplus k_6) \oplus 2 \bullet s(p_{11} \oplus k_{11}) \oplus s(k_{12}) \\
&\quad \oplus k_{15} \oplus k_3 \oplus k_7 \oplus k_{11}
\end{aligned}
\qquad (2)
$$

Thus, for every possible combination of unknown bits from a given equation we are able to generate the respective index (x from the equations above), replacing the plaintext information, and the key bits discovered in the previous phase. Note that the unknown bits of the key bytes outside the s-box operations are fixed to an arbitrary value. The happens because they have no impact with the table cache block accessed by the resulted index[12]. Where, each key combination has 16 (=4*4) bits - (total of 65356 different combinations).

Then, the table address informations allow the attacker to know which L1 line maps the table access using the previous generated index.

At this moment there are only 2 possible outcomes, if in fact the combination used is the real, then the generated line score from the measurements would be always high. However, if the combination is not the real one, then measurement at some point will tell the specific line has a low score, which translated means that that specific L1 line was not used during that encryption, which refuses the veracity of the tested unknown bit key combination. All the existing measurements files are read, removing/excluding all possible combinations each time a low score associated is encounter. The last equation's combination that remains alive in the end of this process becomes the real combination. This means this combination in particular got high scores in every measurement - which only happens with the real combination for a considerable set of measurements.

[12] For instance in x2, the k2 low bits value will not impact with the table block accessed by x2, but only the cache block offset

❖ Results & Technology

Crypto-analysis phase is implemented on python program crypto.py.

Specifically on Round 1 attack sub-phase is implemented on the function round_1atk(). The number of measurements read are the same as the number of measurements generated in the previous phase, i.e.: 500.
Once again, for crypto.py to get the L1 line scores and plaintext used on a particular measurement *M*, it will have to scan the file *meas#M.out*. On the other side, to get the table addresses information it only has to read the file with the name *tables#M.out*.
Since the tables happen to be aligned the memory block boundaries, this is all of our tables are mapped in the position 0 of the respective cache line, each byte's 16 values previously picked allows the attacker to uncover the most significant 8-log2(16) = 4 bits of every key byte from the secret key. Totalizing in 64 out of 128 bits discovered.

Round 2 attack sub-phase is placed inside the round_2_atk() function. Likewise in the round 1 attack, it loops over the same 500 measurements, accessing meas.out and table.out files, uncovering the less significant log2(16) = 4 bits of each key byte, and consequently the whole 128-bit secret key. Low scores correspond to scores below 50 , and high scores to scores above 50 (L1 cache misses).
It was required the inclusion of the python module pyfinite for the GF(2^8) multiplications and an S-box 256-element-table, while the hypothetical index was being generated.

Overall crypto-analysis duration is 2.5min, which means this attack takes in average 3.7 min of a total of 5 attacks all against a different key victim.
All the attack's constants, (such as the number of measurements, the low score threshold, sleep time by atk.c, number of victim encrypt. by measurement) were extracted experimentally.
5 out of 5 attacks using the configuration described above recovers the whole 128-bit secret key. However attacks using different attack constants configurations, make the key bits recovered in average to drop. This problem is addressed in the section below.

- Possible Optimizations & Notes

The first goal of our attack is to uncover the secret key but after that there are plenty of other aspects to consider such as execution time, samples needed, or just the simplicity of the code that can be improved. In this section some of these aspects are approached:

Some notes and optimizations to consider on measurement phase:

1. The number of the iterations a given line is filled on atk.c or the number of encryptions triggered by vic.c were extracted experimentally. One thing to consider about these numbers would be to lower them, in an attempt to low the measurement time required. However if we perform this move of lowering these numbers, it starts

to appear at some point an unique false positive measurement, i.e.: a measurement completely filled with low scores, which is an very odd result. If we exclude this odd measurement from the rest, the attack is able to recover whole key.

2. Everytime we are filling a L1 line on atk.c, instead of doing a write and a read of the desired attacker data, we could only perform a read. Theoretically the papi should be able to detect the cache misses as we access attacker's data, however this not verifies experimentally. If we take a close look to the measurements performed using reads, we find out the number of L1 cache is residual (close to 0).

3. Another point is: right after atk.c finish his write on meas.out, it waits for the victim to finish its execution. Using kill() function from <signal.h> to stop victim's execution should be consider in an attempt to save some measurement time. However I'm a bit skeptical about the kill(), since it should be avoid for some reason.

Regarding the Crypto-Analysis phase, there are multiple aspects to consider to optimize:

1. Take for instance the number of measurement files required to extract the full secret key: in all my experiments, I was able to uncover the full key using a maximum of 500 measurements. If we wanted to low this number, one approach to consider should be to include in our attack the others 2-round equations and even the 3-round equations. The less measurements required would impact the measurement phase time by lowering it, however our crypto-analysis phase would increase complexity, and consequently last more time.

2. In terms of time optimizations, I think there are no major changes to make in order to decrease the this phase's time. The current crypto.c implementation corresponds to the shortest in time in my opinion. Of course I can still implement small time optimizations such as sporadic checks of the existence of one key value per key byte, instead of looping throughout all the existing measurements.

3. Another optimization to consider would be to use the same technique from the 1-Round attack on the 2-Round attack. However the score technique cannot be used on the 2-Round because some hypothetical combination might get L1 lines score in average higher than the real one, since there are high scores 3x or 4x higher than a normal high score - (I tested myself this).
   For instance having combination scores of combination *a* = *[5000,3000,4000,13]* and the real combination *r* = *[2000,2000,1000,1000]*, using the score technique the combination *a* would get an higher average score, with interpreter.c choosing *a* instead of *r*.

4. Another thing to consider would be to distribute crypto.py execution across all existing system's cores in order to short the time needed, however I don't know if that is our top of priorities right now.