ECED4406 Lab 2: AES Timings

Due Date: October 25th, 2023.

Groups:

Labs should be completed in groups of 2. Some groups of 1 or 3 will be accepted based on the class size.

Lab groups will be configured in Brightspace to allow students to submit labs online.

Marking:

The lab is worth a total of 26 points:

2 pts – Generate lab formatting, editorial, English etc.

24 pts - Answers to questions (detailed below)

Expected Report Format:

A lab report is required. This lab report for this lab should primarily answer the questions in the lab, it does not require extensive background information.

Pre-requisites: Using Google CoLab

There is an introductory video at https://www.youtube.com/watch?v=v5W8Uff4x0Q . You can freely use "raw" Python in this course, but you may find this useful.

You can find a notebook for this lab at this link:
https://colab.research.google.com/drive/1b6VVTbS_NVw0yyqCftkTEmz2fdrCUNPk?usp=sharing

NOTE: You will need to copy this to your own Google account for it to run. Or you can copy the Python code out and run locally.

Part 1: Measuring Execution Time with OpenSSL

Objective: Learn how to use OpenSSL to benchmark a cryptographic implementation, and compare the impact of different key sizes and modes on performance of AES.

Setup:

If you have OpenSSL installed, you can simply run the following at the command line:

```
openssl speed -elapsed -evp aes-128-ctr
```

This would run aes-ctr with 128-bit key size. By adjusting the mode names & key size you can see the impact of various changes. See slides from lecture 0x20B for more information.

If you *don't* have OpenSSL, you can run OpenSSL on the colab cloud server.

Part 2: Measuring Execution Time of AES Functions

Objective: See how different functions within AES have different overheads. Measure the execution time of the various functions with a simple Python implementation.

Setup:

See the link to the Co-lab, which has the required Python code.

Part 3: Measuring Constant or Non-Constant Execution Time

Objective: For this part you'll be comparing two functions within the AES operation: the xtime() function (part of MixColumns), and the S-Box (part of subbytes). This is split into "3-1" and "3-2", which are using xtime() and the s-box respectively.

Setup:

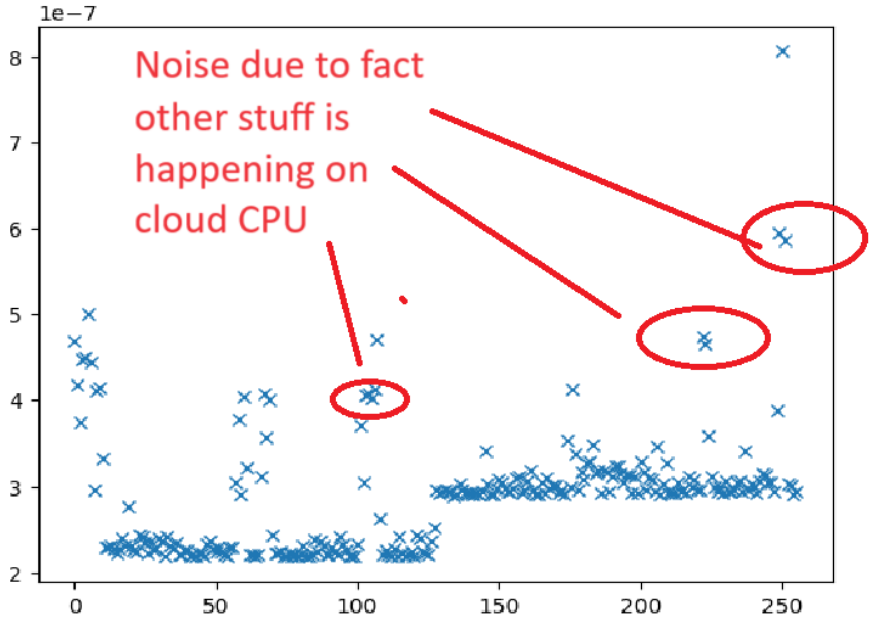
See the link to the Co-lab, which has the required Python code.

Part 3-1)

Run the test of the xtime() operation, which should generate a graph showing the execution time along with the input data. Note that there will be a lot of noise – you can see here a general flow where the

execution time suddenly jumps, but also lots of longer points. If you re-run this you'll find those random points move around, and they are just noise.

We care about the general "flow" of this, and where the sudden jump seems to consistently happen.



The xtime() function is used in the mixcolumn operation. The general flow of xtime can be seen here:

```
def xtime(a):
  if (a & 0x80):
    return ((a << 1) ^ 0x1B) & 0xFF
  else:
    return (a << 1)
```

If you prefer C code, see the link to the AES code present in the slides. This also includes an xtime example.

Part 3-2)

For this part, you'll be doing a similar task as above but with the S-Box. The S-Box is just a lookup table, the definition of it can be seen in the Python file.

Question 1: AES Mode & Key-Size Timing [8 pts]

A) [6pts]

Measure the through-put of the following configurations:

- aes-128-ctr at 16, 1024, 16384 byte chunks
- aes-256-ctr at 16, 1024, 16384 byte chunks
- aes-128-cbc at 16, 1024, 16384 byte chunks
- aes-256-cbc at 16, 1024, 16384 byte chunks
- aes-128-gcm at 16, 1024, 16384 byte chunks
- aes-256-gcm at 16, 1024, 16384 byte chunks

Report the results as a table such as the following. Be sure to provide reasonable units for the throughput (don't just copy the openssl output, but scale to a constant GBytes/second or similar)

Mode	**Key Size**	**Throughput**
AES-CTR	128	
AES-CTR	256	
AES-CBC	128	
etc	etc	etc

B) [2 pts] Which was the slowest mode/key-size combination. Suggest why this is the case, with reference to the AES algorithm itself.

Question 2: AES Individual Function Timing [6 pts]

A) [4 Pts] Measure each of the four functions and fill out a table showing the time it took on your test computer for **each** iteration. Ensure you include units and the table has reasonable units (e.g., displaying this in nanoseconds or microseconds may be more relevant).

Function	**Time per Iteration**
SubBytes	?
ShiftRows	?
MixColumns	?
AddRoundKey	?

B) [2 pts] What is the slowest function of those? Take a look at the Python implementation of that code, you'll likely find it's slow as contains additional functions within it. Measure the execution time of just the internal function and report this value.

Question 3: AES Constant-Time Operations [10 pts]

Part 3-1)

3-1-A) [2 pts] Include the graph as generated. Consider averaging several outputs to generate a smoother result, which will reduce the effect of the random outliers.

3-1-B) [2 pts] Does the execution time depend on the input data? If so, what is the cause of this? Look back at the code of **`xtime()`** to understand the root cause.

Part 3-2)

3-2-A) [2 pts] Include the graph as generated. Consider averaging several outputs to generate a smoother result, which will reduce the effect of the random outliers.

3-2-B) [2 pts] Does the execution time depend on the input data? How do you know it is or isn't constant-time?

3-2-C) [2 pts] For both 3-1 and 3-2 there is a lot of "noise" in the graph. Where does that noise come from? Note the timeit() is running each function multiple times, why is that insufficient to eliminate this noise?