Shuffled AES

1st Dinis Cruz
DETI
Universidade de aveiro
Aveiro,Portugal
cruzdinis@ua.pt

***Abstract*—This document follow the implementation and results of the first project for the course of Applied Cryptography, where the objective was to implement a version of AES with a special round.**

***Index Terms*—component, formatting, style, styling, insert**

I. Introduction

This work revolves around the construction of an encryption algorithm based on AES but with a special round where the AES operations of Substitute bytes, Shift Rows, Mix Columns and Add Round Key are altered by a shuffle key passed as an argument. The implementation follows a normal AES algorithm with a key of 128.

II. Program Structure and Usage

A. Usage

Both python scripts for aes encryption take the same flags and arguments (-n , -s , -t) :

- n refers to a plaintext word that will be hashed using MD5 for the AES encryption
- s refers to a plaintext word that will be hashed using MD5 to create variable values for the shuffle rounds
- t makes the program time the encryption/decryption part (Key generation and pre-computed values not included) and writes the time in seconds to the file passed as argument

B. Important Files

encrypt.py reads input from stdin and encrypts this using SAES and outputs the result to stdout.

decrypt.py reads input from stdin and decrypts this using SAES and outputs the result to stdout.

saes_module.py keeps the functions that both encrypt and decrypt use, this file makes the code more organized with precomputed variables and functions that are used both in encrypt and decrypt being stored in it.

III. Implementation

This section follows the most important functions and implementations steps that were taken to achive the end result.

A. Making the Keys

Both arguments passed as keys are hashed and passed to a list of integers. The process of Key Expansion follows the AES Key Expansion with 44 words being created, the first 4 come from the normal key and a process starts for the creation of all other words, the last word generated is transformed,if the number corresponding to that word is a multiple of 4 then a shift of 1 is used, every byte is subbed by the corresponding one in the sbox (more on this later but the main point is that 1 bytes becomes another in a deterministic way) and a xor is made with a round constant (pre-computed), either way the result is xord with the word created 4 steps before. At the end we get an array of 44 words, this 44 words will generate 11 keys being used in order in groups of 4 for each key. The shuffle key ,if passed, is transformed into an array of integers.

B. Getting equally distributed numbers from the shuffle key

The project has a need for number generation from the shuffled key, considering a number generated from the key is used for the shuffling of the sbox, the permutation used for shifting rows, the offset for shifting the columns and for the offset used to rotate the encryption key.

This process must take a number n and from the key generate an equally distributed result from 1 to n.

Considering that the key (a 32 bytes array) comes from an hash, and the definition of a hash functions is that each hash is equally provable than each byte in that 16 bytes array is equally likely to range from 0-15. The sum of all bytes will range from 0 to 512, again, with equally provability. Using the classic cryptography function of the modular division we can divide the sum with our number n.

```
def get_offset_from_key(n):
    if sk == []: return -1
    return (sum(sk) % n) +1
```

This was tested in the generation_from_hash.py script.

C. SAES Encryption and Decryption functions

1) Encryption flow: Exactly the same flow as AES but when the special round is reached the shuffled versions of the normal functions are used encrypting one ”special round”.

```
def saes_encrypt(block):
    sr = get_offset_from_key(9)
```

```
    block = xor_round_key(block,keys[0])
    for i in range(1,10):
        if i == sr:
            block = shuffle_sub_bytes(block)
            block = shuffle_shift_rows(block
            block = shuffle_mix_cols(block)
            block = shuffle_xor_round_key(
            block,keys[i])
        else:
            block = sub_bytes(block)
            block = shift_rows(block)
            block = mix_columns(block)
            block = xor_round_key(block,keys
    block = sub_bytes(block)
    block = shift_rows(block)
    block = xor_round_key(block,keys[10])
```

2) Decryption flow: The process is also similar to the decryption with AES but this time there is a need to decrypt the special round, with the added complexity of AES not having a symmetric flow with encryption and decryption, the decryption of the shuffled round must take place in the middle of one decryption round.

```
def saes_decrypt(block):
    sr = get_offset_from_key(9)

    block = xor_round_key(block,keys[10])
    for i in range(1,10):
        if i == 10-sr:
            block = inv_shift_rows(block)
            block = inv_sub_bytes(block)
            block = shuffle_xor_round_key(
            block,keys[10-i])
            block = shuffle_inv_mix_cols(
            block)
            block = shuffle_inv_shift_rows(
            block)
            block = shuffle_inv_sub_bytes(
            block)
            block = xor_round_key(block,
            keys[10-i-1])
            block = inv_mix_columns(block)
        elif i == 10 -sr +1:
            continue
        else:
            block = inv_shift_rows(block)
            block = inv_sub_bytes(block)
            block = xor_round_key(block,
            keys[10-i])
            block = inv_mix_columns(block)
    block = inv_shift_rows(block)
    block = inv_sub_bytes(block)
    block = xor_round_key(block,keys[0])
```

3) Simplistic View for better understanding:

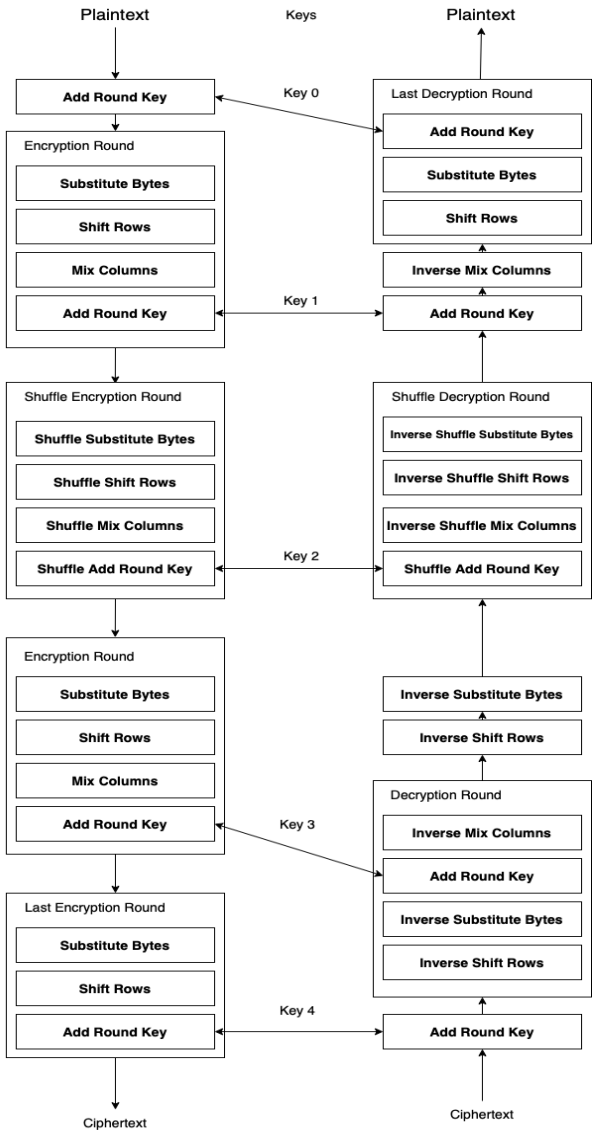

Fig. 1. Encryption and decryption flow where only 4 rounds are considered for a better view of all the process

D. Substitute Bytes

Substitute bytes follows a simple and yet efficient procedure where all bytes from a block are substituted by the sbox, this box has 256 entries where the numbers range from 0 to 256 which means that every byte subbed from the sbox will output a byte that is within the same range. This box is pre-stored in the python module I developed.

The inverse process follows the same logic, but this time the substitution comes from the inverse of the sbox (rsbox), this box is designed such that, with the byte x if we lookup the value at the position x of the sbox, take that value and look it up in the inverse sbox we end up with x.

The shuffle sbox is based on the famous algorithm Fisher–Yates shuffle were a random permutation of that array is calculated, this algorithm was taken from a stack overflow thread specified in the code. I also created a very simple

pseudo random number generator that is embedded in the code after the professor explained why using random number generators can become a problem if used on different systems.

This random generator will be the one giving random numbers to the fisher-yates algorithm, the seed for the generator is the integer representation of the key, maximum value being 2 to the power of 128.

It is important to note that AES is a deterministic algorithm that is why even tho there is a generator of pseudo random number sin the code, with a seed coming from the shuffle key we can ensure the integrity of the cipher.

E. Shift Rows

Simply shifting every row a predetermined value ([0,1,2,3]), the reverse of this function was achieved using the "inverse" of all offsets, an offset of 0 makes no changes so no need to shift, all the other offsets are transformer using:

$$4 - offset \tag{1}$$

Example of this operation:

$$[1, 2, 3, 4] \tag{2}$$

With an offset of 3 becomes:

$$[4, 1, 2, 3] \tag{3}$$

With an offset of 1 becomes the original array

$$[1, 2, 3, 4] \tag{4}$$

The shuffled rounds use the same principle but the offsets are chosen from all 24 permutations (pre-computed) of the original tuple (0,1,2,3).

F. Mix Columns

The mix columns function follows the implementation taken from here where each column is multiplied by a pre-selected and pre-computed matrix, the matrix that inverses these operations is also pre-selected and pre-computed within the code. There is a catch, this multiplication must be done within the Galois field of 2 ^ 8 which means that addition becomes an xor and multiplication becomes a multiplication within this field, there are no overflows, all the values come to a value between 0-256.

Mix columns matrix:

$$\begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix}$$

Inverse mix columns matrix:

$$\begin{bmatrix} 14 & 11 & 13 & 9 \\ 9 & 14 & 11 & 13 \\ 13 & 9 & 14 & 11 \\ 11 & 13 & 9 & 14 \end{bmatrix}$$

Considering a columns with generic values like:

$$\begin{bmatrix} a1 \\ a2 \\ a3 \\ a4 \end{bmatrix}$$

The multiplication of the first row of the mix columns matrix would result in something like:

$$r1 = 2 * a1 + 3 * a2 + 1 * a3 + 1 * a4 \tag{5}$$

But within the Galois field we have

$$r1 = (2 * a1)\hat{}(3 * a2)\hat{}(1 * a3)\hat{}(1 * a4) \tag{6}$$

where * means multiplication in GF(28)

This is transformed into python code (using lookups as will be mentioned after)

```
block_res[0][0] = lg[2][a1] ^ lg[3][a2]
^ lg[1][a3] ^ lg[1][a4]
```

Note: lg is the function that does the look ups, it a dictionary where the keys are the number of the matrix and this returns a list where the index corresponds to the multiplication of said number with the values of the column.

The shuffle round makes the columns shift a certain value from 0 to 3.

It used the same logic has a normal shift rows but here all rows are shifted an equal amount, which translates to a shifting of columns.

G. Add Round Key

The add round key step basically does a bit wise xor of the bytes within the key with the bytes in block, this xor is simple in python using the operation " ^ ", for the shuffle add round key, an offset is computed from 0 to 15 to rotate the key and then the xor is made.

```
def shuffle_xor_round_key(block,k):
    offset_key = shift_list(k,
    get_offset_from_key(15))
    return xor_round_key(block,offset_key)
```

The function above chooses a number from the key and shifts the round key, after that each bytes is xord using the function bellow.

```
def xor_adr(s,h):
    return h ^ s
```

An example of a key rotated by 3

Original key

$$[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16] \tag{7}$$

After the rotation

$$[4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 1, 2, 3] \tag{8}$$

IV. Optimization

A big part in cryptography and one of the reasons hardware is made with special instruction is to make encryption and decryption a fast process. So an effort was made to make the code run faster after it's first conception. For faster implementations a language closer to machine code would make since or even C but due to my proficiency in C being low python was chosen for this project.

A. Identification of bottle neck functions

The first step was identifying were the program was spending most of it's time, this was done using sneakviz and the following result shows that within the decryption process (It's the same for encryption) our bottle neck lies within the mix columns function.

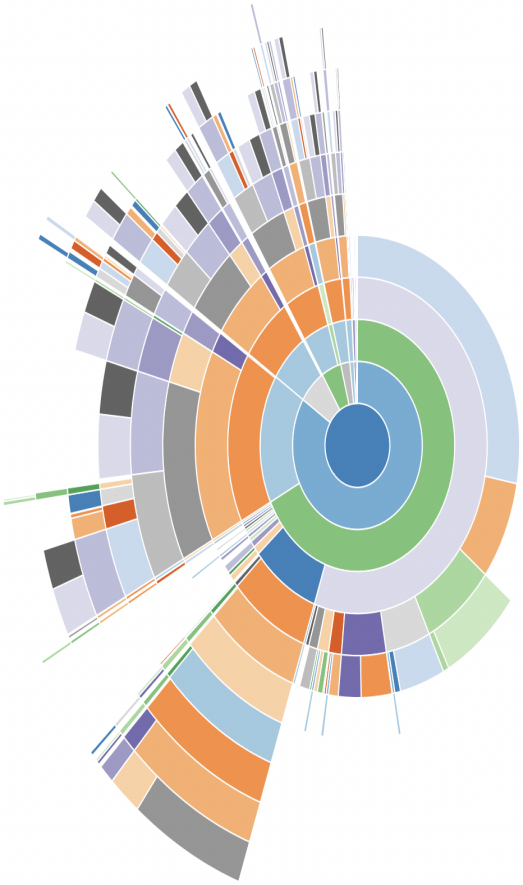

Fig. 2. Results from snakeviz

B. Precomputed Galois Field (28) Lookup

Before this, a function was being used to calculate the multiplications while keeping the values in the Galois field, this function was taking too much time and calls so all the values of multiplications within the Galois field by 2,3,9,11,13,14 were precomputed and stored so that the results of multiplication can be looked up instead of calculated with gives us a complexity of 0(1)

C. Usage of List Comprehensions

Some other problems were identified within some other functions (like sub_bytes), these problems revolved around the usage of for loops for iteration over lists, with the usage of list comprehensions the process is sped up since python optimizes these types of operations.

D. Removal of for loops

For better performance and in places where loops were being used to condense the code these loops were transformed in unique calls to that there isn't nearly as many jumps in the code. This was done for example in the printing of information from the blocks.

V. Results

The results follow the measurements of 4 different AES encryptions and decryptions of the same 4KB page for 100000 iterations. The following are the algorithms experimented

- My Shuffled AES
- My Normal AES
- Cryptography AES (From the cryptography library)
- PyCrypto AES (From the Pycrypto AES)

All libraries and programs were ran using the ECB mode of encryption, this takes into consideration the best measurements from the 100000 iterations, the program used for this computations was speed.py where sub processes are used to run bash code to execute the python scripts for encryption and decryption. Each time a new key is generated and the 4KB page is encrypted and the result of that is decrypted. Times do not take into consideration key setup and pre-computed values, only the encryption adn decryption phases. The results are as follows:

	SAES	AES	Cryptography	Pycrypto
Encryption	0.0221	0.0213	0.000461	$1.788e^{-5}$
Decryption	0.0218	0.0210	0.000429	$1.668e^{-5}$

Table 1: Results from speed

The shuffle round didn't become a bottle neck as can be seen in the results but my didn't perform the best. Cryptography is around 47 time faster than my algorithm and PyCrypto was around 1274 times faster than my algorithm.

All results can be found in results.txt

VI. Conclusion

This work proved to be very challenging considering there was a failed first attempt to solve it in C with AES-NI instructions. Not enough knowledge in C and it's memory instructions made it impossible with the estimated time I had. Changing to python made the process move faster at the cost of efficiency and 15% of the grade, never the less it lead to a better understanding of the AES encryption algorithm and how to work around its asymmetric encryption/decryption flow.