Masters in Cybersecurity

Applied Cryptography

Shuffled AES (S-AES)

1st Project

João Luís - 107403

Ricardo Quintaneiro - 110056

November 10, 2024

1 Introduction

The Advanced Encryption Standard (AES) was derived from the Rijndael algorithm, which offered more flexibility in terms of block sizes and key sizes than what was ultimately standardized as AES. It was also possible to replace the S-Box for another ones without compromising the algorithm.

This project implements a variant of AES-128 called Shuffled AES (S-AES), which introduces an additional 128-bit shuffling key (SK) for a total key length of 256 bits.

S-AES modifies the standard AES algorithm in two key ways:

- It deterministically shuffles the round keys based on 64 bits of the SK, affecting both the order of the keys and the arrangement of bytes within each key;
- It modifies one of the first nine encryption rounds (selected using SK) by:
  - XORing the round key with the remaining 64 bits of SK used twice;
  - Using a shuffled variant of the original S-Box in the SubBytes step;

The implementation includes both a Python version and a hardware-accelerated version using Intel's AES-NI instructions in C and Assembly, allowing us to compare performance across different approaches. We'll explore the technical details of both implementations and analyze their relative speeds against standard library implementations of AES.

2 S-AES

Our implementation of S-AES follows work done by ipfans/pyAES [1] modified to run on Python3 and handle our specific use case. For that we made the modifications necessary to allow both AES-128 and S-AES, with 128 bits (16 bytes) of key and another 128 bits of shuffle-key, in ECB (Electronic Code Book) mode with PKCS#7 padding.

```
block_size = 16
key_size = 16

def new(key, skey=None):
    if skey is None:
        return ECBMode(AES(key))
    else:
        return ECBMode(AES(key, skey))
```

In our key setup, we also set the shuffle key, which if it's normal AES defaults to *None* and it's handled accordingly. If it's S-AES it will do the necessary operations to the shuffle key. We select the permutation subkey and the modified round subkey using an interleaving pattern so that the alternating byte distribution ensures that the two subkeys contain non-adjacent bytes from SK, which can add complexity to the key structure and make it harder to reverse-engineer the original SK based on either subkey alone, as well as introducing very different modifications to the encryption process if adjacent bytes of the key are changed.

```
class AES(object):
    block_size = 16

    def __init__(self, key, skey=None):
        self.setkey(key)
        self.setshufflekey(skey)
        self.expand_key()

    def setkey(self, key: bytes | str):
        """Sets the key and performs key expansion."""

        self.key: bytes = key if isinstance(key, bytes) else key.encode()
        self.key_size = len(key)
        self.rounds = 10

    def setshufflekey(self, skey: bytes | str | None):
        """Sets the shuffle key and performs key expansion."""

        if skey is None:
            self.skey = None
            self.permutation_indices = None
```

```
            self.round_key_order = None
            self.modified_round_number = None
            return None

        self.skey = skey if isinstance(skey, bytes) else skey.encode()
        self.skey_size = len(skey)

        self.permutation_skey = self.skey[:16:2]
        self.modified_round_skey = self.skey[1:16:2]

        self.permutation_indices =
        ↪  self.generate_bytes_permutation_indices(self.permutation_skey)
        self.round_key_order = self.round_key_order_permutation(self.permutation_skey)
        self.modified_round_number = self.select_modified_round_number(self.skey)
        self.saes_sbox = self.create_saes_sbox(self.modified_round_skey)
        self.saes_inv_sbox = self.create_saes_inverse_sbox()
```

The strategy we use to generate the indices to permutate bytes within each round key is to hash the permutation subkey added with a byte containing the round number using the digest function SHA-256. This introduces as much diffusion as possible to each of the outputs, which is essential to generate pseudo-random but deterministic permutations, as it ensures that each permutation is sensitive to changes in the permutation subkey and round number, creating unique permutations across rounds.

```
    def generate_bytes_permutation_indices(self, permutation_skey):
        """Generates a list of deterministic permutation indices for round keys."""
        num_round_keys = self.rounds + 1
        permutation_indices = []

        for round_idx in range(num_round_keys):
            # Hash the skey with the round index for deterministic bytes
            hash_input = permutation_skey + round_idx.to_bytes(1, 'big')
            hash_output = sha256(hash_input).digest()

            # Use the hash output to generate a permutation for this round key
            indices = list(range(16))
            for i in range(15, 0, -1):
                swap_idx = hash_output[i] % (i + 1)
                indices[i], indices[swap_idx] = indices[swap_idx], indices[i]

            permutation_indices.append(indices)

        return permutation_indices
```

To scramble the order of the round keys, we basically follow the same strategy as explained above, but now for a single array of indices only and without the byte addition to the permutation subkey.

```python
def round_key_order_permutation(self, permutation_skey):
    """Generates a deterministic permutation of round key order based on permutation
    ↪  sub_key."""
    num_round_keys = self.rounds + 1
    order_indices = list(range(num_round_keys))

    # Generate pseudo-random bytes for shuffling
    hash_output = sha256(permutation_skey).digest()
    for i in range(num_round_keys - 1, 0, -1):
        swap_idx = hash_output[i % len(hash_output)] % (i + 1)
        order_indices[i], order_indices[swap_idx] = order_indices[swap_idx],
        ↪  order_indices[i]

    return order_indices
```

Then we use both of these indices in the last steps of the function to expand the AES key. This allows us to shuffle the order of the bytes of each key as well as their order.

```python
def expand_key(self):
    """Performs AES key expansion on self.key and stores in self.exkey"""

    ...

    # Shuffle round keys if `self.permutation_indices` is set
    if self.permutation_indices and self.round_key_order:
        shuffled_exkey = array('B')
        for round_idx in range(self.rounds + 1):
            start = round_idx * 16
            end = start + 16
            round_key = exkey[start:end]
            permuted_round_key = array('B', [round_key[i] for i in
            ↪  self.permutation_indices[round_idx]])
            exkey[start:end] = permuted_round_key

        for round_idx in self.round_key_order:
            start = round_idx * 16
            end = start + 16
            shuffled_exkey.extend(exkey[start:end])
        self.exkey = shuffled_exkey
    else:
        self.exkey = exkey
```

For selecting the number of the modified round we also do a digest using SHA-256, this time of the whole SK as asked in the exercise, and use the integer value of the digest to do the remainder with the 9 rounds of key expansion available to be the modified round (can't happen on the last round). Then we add 1 because the way we have the code structured, the first round (index 0) is the round used to do a single AddRoundKey. This will return a number in between 1 and 9, and it will be selected as randomly as possible because of this process.

```python
def select_modified_round_number(self, skey):
    # Create a hash of the secret key
    hashed_key = sha256(skey).hexdigest()  # Using SHA-256

    # Convert the hash to an integer
    hash_value = int(hashed_key, 16)  # Convert hex to integer

    # Generate a round number based on the hash value
    round_number = (hash_value % (self.rounds - 1)) + 1  # Can't be on last round

    return round_number
```

Then we check in the AddRoundKey function if the round number matches the modified round number. If so, we perform the XOR operation to the round key with the modified round subkey, duplicating it as we need 128 bits and the subkey is only 64. Then the function resumes as normal, by XOR'ing the block with the round key.

```python
def add_round_key(self, block, round):
    """AddRoundKey step. This is where the key is mixed into plaintext"""
    offset = round * 16

    if round == self.modified_round_number:
        for i in range(16):
            self.exkey[offset+i] ^= self.modified_round_skey[i % 8]

    for i in range(16):
        block[i] ^= self.exkey[offset + i]
```

To create a shuffled sbox, we use a digest of the modified round subkey to again scramble indices. Then we run the whole list of indices and get the value of the original AES-SBox of each. After that we validate if at least 50% of the bytes in the shuffle differ from the original one. If that fails, we recursively call the function until we get 128 changed positions or more.

```python
def create_saes_sbox(self, modified_round_skey, sbox=None):
    """
    Creates a shuffled S-Box based on the provided modified round sub key.
    :param modified_round_skey: The modified sub key used for creating the S-Box.
    :return: A shuffled S-Box.
    """
    if sbox is None:
        sbox = aes_sbox

    def generate_shuffled_sbox():
        # Initial SHA-256 hash of the key
        hashed_key = sha256(modified_round_skey).digest()
```

```
            # Create a list of indices [0, 1, ..., 255]
            indices = list(range(256))

            # Use the hash to shuffle the indices
            for i in range(len(indices)):
                # The current byte from the hash to modify the index
                byte = hashed_key[i % len(hashed_key)]

                # Calculate the new position for this index
                swap_index = (i + byte) % 256

                # Swap the current index with the calculated index
                indices[i], indices[swap_index] = indices[swap_index], indices[i]

            # Create a shuffled S-Box using the shuffled indices
            shuffled_sbox = [aes_sbox[index] for index in indices]
            return shuffled_sbox

        def validate_and_shuffle(shuffled_sbox):
            changed_positions = 0
            # Ensure at least 50% of the S-Box bytes have changed their position
            for i in range(256):
                if shuffled_sbox[i] != aes_sbox[i]:
                    changed_positions += 1

            # If less than 50% have changed, recursively shuffle again
            if changed_positions < 128:  # Less than half
                return self.create_saes_sbox(modified_round_skey, shuffled_sbox)  # Recurse
                ↪  with the same key
            return shuffled_sbox

        shuffled_sbox = generate_shuffled_sbox()
        return array('B', validate_and_shuffle(shuffled_sbox))
```

After the setup, in the encryption process, each time the modified round number is the same as the current round, the SubBytes function will use the Shuffled SBox instead of the original one.

```
    def encrypt_block(self, block):
        """Encrypts a single block. This is the main AES function"""

        # For efficiency reasons, the state between steps is transmitted via a
        # mutable array, not returned
        self.add_round_key(block, 0)

        for round in range(1, self.rounds):
            if self.modified_round_number is None or round != self.modified_round_number:
                self.sub_bytes(block, aes_sbox)
            else:
                self.sub_bytes(block, self.saes_sbox)
            self.shift_rows(block)
```

```
            self.mix_columns(block)
            self.add_round_key(block, round)

        self.sub_bytes(block, aes_sbox)
        self.shift_rows(block)
        # no mix_columns step in the last round
        self.add_round_key(block, self.rounds)
```

Creating the inverse sbox for S-AES is just as easy as assigning the value of the current position to the index given by the value of the original sbox at that position.

```
    def create_saes_inverse_sbox(self):
        """
        Calculate the inverse S-box from the given S-box.
        The inverse S-box maps each substituted value back to its original value.

        Returns:
            array: Inverse S-box array
        """
        # Create an empty inverse s-box of the same size (256 bytes)
        saes_inv_sbox = array('B', [0] * 256)

        # For each position and value in the original s-box
        for position in range(256):
            value = self.saes_sbox[position]
            # In the inverse s-box, map the value back to its position
            saes_inv_sbox[value] = position
        return saes_inv_sbox
```

And to use it in the decryption process we do the same check as in the encryption process but we use the Inverse Shuffled SBox.

```
    def decrypt_block(self, block):
        """Decrypts a single block. This is the main AES decryption function"""
        self.add_round_key(block, self.rounds)
        for round in range(self.rounds - 1, 0, -1):
            self.shift_rows_inv(block)
            if self.modified_round_number is None or round != self.modified_round_number - 1:
                self.sub_bytes(block, aes_inv_sbox)
            else:
                self.sub_bytes(block, self.saes_inv_sbox)
            self.add_round_key(block, round)
            self.mix_columns_inv(block)

        self.shift_rows_inv(block)
        if self.modified_round_number is None or 0 != self.modified_round_number - 1:
            self.sub_bytes(block, aes_inv_sbox)
        else:
            self.sub_bytes(block, self.saes_inv_sbox)
        self.add_round_key(block, 0)
```

3 S-AES with AES-NI

In order to make an implementation of S-AES with AES-NI assembly instructions we made the decision of copying the example C code from the Intel specification [2] and modifying it to our needs. For that purpose we made the functions needed to set the shuffle key and derived subkeys, generate both the permutations indices, select the modified round number and created the Shuffled SBoxes. After that, we made the necessary changes to the encryption and decryption process, implementing the functions for SubBytes, ShiftRows and MixColumns and using them alongside the AES-NI XOR function for AddRoundKey.

```
void AES_ECB_encrypt(const unsigned char *in,
                     unsigned char *out,
                     unsigned long length,
                     const char *key,
                     int number_of_rounds,
                     int modified_round_number,
                     uint8_t *modified_round_skey,
                     uint8_t *saes_sbox)
{
    __m128i tmp;
    int i, j;
    if (length % 16)
        length = length / 16 + 1;
    else
        length = length / 16;
    for (i = 0; i < length; i++)
    {
        tmp = _mm_loadu_si128(&((__m128i *)in)[i]);
        tmp = _mm_xor_si128(tmp, ((__m128i *)key)[0]);
        for (j = 1; j < number_of_rounds; j++)
        {
            if (j == modified_round_number)
            {
                sub_bytes((uint8_t *)&tmp, saes_sbox);
                shift_rows((uint8_t *)&tmp);
                mix_columns((uint8_t *)&tmp);
                tmp = _mm_xor_si128(tmp, ((__m128i *)key)[j]);
            }
            else {
                tmp = _mm_aesenc_si128(tmp, ((__m128i *)key)[j]);
            }
        }
        tmp = _mm_aesenclast_si128(tmp, ((__m128i *)key)[j]);
        _mm_storeu_si128(&((__m128i *)out)[i], tmp);
    }
}
```

```
void AES_ECB_decrypt(const unsigned char *in,
                     unsigned char *out,
                     unsigned long length,
                     const char *key,
                     int number_of_rounds,
                     int modified_round_number,
                     uint8_t *modified_round_skey,
                     uint8_t *saes_inverse_sbox,
                     const char *old_key)
{
    __m128i tmp;
    int i, j;
    if (length % 16)
        length = length / 16 + 1;
    else
        length = length / 16;
    for (i = 0; i < length; i++)
    {
        tmp = _mm_loadu_si128(&((__m128i *)in)[i]);
        tmp = _mm_xor_si128(tmp, ((__m128i *)key)[0]);
        for (j = 1; j < number_of_rounds; j++)
        {
            if (j == number_of_rounds- modified_round_number +1 )
            {
                shift_rows_inv((uint8_t *)&tmp);
                sub_bytes((uint8_t *)&tmp, saes_inverse_sbox);
                tmp = _mm_xor_si128(tmp, ((__m128i *)key)[j]);
                mix_columns_inv((uint8_t *)&tmp);
            }
            else {
                tmp = _mm_aesdec_si128(tmp, ((__m128i *)key)[j]);
            }
        }
        tmp = _mm_aesdeclast_si128(tmp, ((__m128i *)key)[j]);
        _mm_storeu_si128(&((__m128i *)out)[i], tmp);
    }
}
```

The key expansion comes from the Assembly code of AES-NI provided by Intel and we leverage it by introducing it into the compilation process with a custom Makefile.

```
# Compiler
CC = gcc

# Compiler flags
CFLAGS = -maes -msse4 -Wno-error=implicit-function-declaration -I/usr/include/openssl
```

```
# Linker flags
LDFLAGS = -lssl -lcrypto -lgmp

# Source files
ASSEMBLY_SRCS = key_expansion.s
C_SRCS = ecb_main.c aes.c encrypt_decrypt.c

# Object files
ASSEMBLY_OBJS = $(ASSEMBLY_SRCS:.s=.o)
C_OBJS = $(C_SRCS:.c=.o)

# Executable
TARGET = ecb_exe

# Default rule
all: $(TARGET)

# Rule to compile assembly files
$(ASSEMBLY_OBJS): %.o : %.s
        $(CC) $(CFLAGS) -c $< -o $@

# Rule to link and build the final executable
$(TARGET): $(ASSEMBLY_OBJS) $(C_OBJS)
        $(CC) $(CFLAGS) $(C_SRCS) $(ASSEMBLY_OBJS) $(LDFLAGS) -o $(TARGET)

# Clean up object files and the executable
clean:
        rm -f $(ASSEMBLY_OBJS) $(C_OBJS) $(TARGET)
```

And that expansion is called in the setup and after that we derive from it the needed subkeys. In the decryption process a check is made for the modified round, so that we can skip using the InverseMixColumns function of AES-NI as that would break the S-AES modified round process, because of the order and way of application of that function in AES-NI which is unconventional but probably more optimal.

```
int AES_set_encrypt_key(const unsigned char *userKey,
                        const int bits,
                        AES_KEY *key,
                        uint8_t *permutation_indices,
                        uint8_t *order_indices,
                        int modified_round_number,
                        uint8_t *modified_round_skey)
{
    if (!userKey || !key)
        return -1;
    if (bits == 128)
    {
        AES_128_Key_Expansion(userKey, key);
```

```
        permute_key(key, AES_128_KEY_SCHEDULES, permutation_indices);
        shuffle_key(key, AES_128_KEY_SCHEDULES, order_indices);

        __m128i *modified_key = (__m128i *)key->KEY;
        uint8_t expanded_skey[16];
        memcpy(expanded_skey, modified_round_skey, 8);
        memcpy(expanded_skey + 8, modified_round_skey, 8);
        __m128i expanded_skey_m128 = *(__m128i *)expanded_skey;
        modified_key[modified_round_number] =
        ↪  _mm_xor_si128(modified_key[modified_round_number], expanded_skey_m128);

        key->nr = 10;
        return 0;
    }
    return -2;
}

int AES_set_decrypt_key(const unsigned char *userKey,
                        const int bits,
                        AES_KEY *key,
                        uint8_t *permutation_indices,
                        uint8_t *order_indices,
                        int modified_round_number,
                        uint8_t *modified_round_skey)
{
    int i, nr;
    ;
    AES_KEY temp_key;
    __m128i *Key_Schedule = (__m128i *)key->KEY;
    __m128i *Temp_Key_Schedule = (__m128i *)temp_key.KEY;
    if (!userKey || !key)
        return -1;
    if (AES_set_encrypt_key(userKey, bits, &temp_key, permutation_indices, order_indices,
    ↪  modified_round_number, modified_round_skey) == -2)
        return -2;
    nr = temp_key.nr;
    key->nr = nr;
    Key_Schedule[nr] = Temp_Key_Schedule[0];
    for (int i = 1; i < nr; i++) {
        if (nr - modified_round_number + 1 == nr - i) {
            Key_Schedule[nr - i] = Temp_Key_Schedule[i];
        } else {
            Key_Schedule[nr - i] = _mm_aesimc_si128(Temp_Key_Schedule[i]);
        }
    }
    Key_Schedule[0] = Temp_Key_Schedule[nr];
    return 0;
}
```

4 Relative speeds

As instructed, we measured the relative lowest speeds of our two implementations as well as the speeds using a library across 100 000 4kB random plaintexts. The library we used to compare from was Python's Cryptography [3].

Here are the results in seconds:

	Encryption	Decryption
AES (Python)	0.019012144	0.021267187
S-AES (Python)	0.020028277	0.022383203
Cryptography (Python*)	0.000018369	0.000020882
S-AES-NI (C & Assembly)	0.000038484	0.000014666

Comparing S-AES to AES reveals a 1 microsecond loss, we think mainly due to the checks needed for the modified round.

Comparing our Python code with the Cryptography library reveals significant losses as was expected, because this library leverages OpenSSL's C implementation [4], which uses AES-NI instructions if possible by the CPU [5], making the whole process that much faster.

Our S-AES implementation in C using AES-NI instructions is very close to Python's Cryptography, being somewhat slower in encryption and faster in decryption.

5 References

[1] ipfans, *Pyaes*. [Online]. Available: https://github.com/ipfans/pyAES/blob/master/pyaes.py.

[2] Intel. [Online]. Available: https://www.intel.com/content/dam/develop/external/us/en/documents/aes-wp-2012-09-22-v01-165683.pdf.

[3] Cryptography. [Online]. Available: https://cryptography.io/en/latest/.

[4] Cryptography-OpenSSL. [Online]. Available: https://cryptography.io/en/latest/openssl/.

[5] OpenSSL-AES-NI. [Online]. Available: https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/6/html/security_guide/sect-security_guide-encryption-openssl_intel_aes-ni_engine#sect-Security_Guide-Encryption-OpenSSL_Intel_AES-NI_Engine.