Cybersecurity MSc: Applied Cryptography 2025-26

1st Project: Tweakable AES (T-AES)

October 22, 2025 Due date: November 9, 2025

Changelog

- v1.0 - Initial version.
- v1.1 - The reference to a tweak subtraction was removed, because it was wrong.

1 Introduction

Tweakable cipher modes are encryption modes that introduce an additional input called a tweak, alongside the usual key and plaintext. The tweak functions like a nonce or IV but does not need to be kept secret. It allows for secure encryption of data blocks even when the same key and plaintext are reused, by varying the tweak. An example is XTS (XEX-based Tweaked-codebook mode with ciphertext Stealing), used in full disk encryption.

In XTS (or XEX) the tweak is a value that can vary per each encrypted block (e.g., it can be a counter), being added (usually XORed) with the input and output of the cipher algorithm. When this mode is used, a block of the cryptogram cannot be replaced by other blocks of the same cryptogram for changing deterministically the recovered plaintext block, since the tweak is not the same (it depends on the block position).

In this project we will develop a tweakable AES version, T-AES. This AES version will use a tweak only once per encryption and decryption, instead of twice like in XTS or XEX. The tweak is inserted in the middle of the AES substitution-permutation network, instead of at its input or output.

2 Homework

The work consists on implementing a tweakable version of AES (T-AES), which will be similar to AES but with an extra 128-bit tweak value. T-AES can work with any of the 3 possible key lengths, 128, 192 or 256, since it works on the data path (substitution-permutation pipeline). You have to create two implementations, both in C (or C++), but one of them using AES-NI instructions.

T-AES will operate as AES, but with one round key modified by a tweak. The round key should be RK5, RK6 or RK7 for AES with 128, 192 or 256 bit keys, respectively. The modification should be an arithmetic addition (not a XOR!) of the tweak with the original round key, considering that both values are unsigned, 128-bit integers[1]. Naturally, during decryption the tweak must be added the same way to the round key (but before its transformation when AES-NI instructions are used).

To complete the T-AES, implement a block counter mode using it, where the tweak used is incremented by one in each block (see Figure 1). In other words, implement an ECB mode where a different counter value is used as a tweak on the block cipher.

[1]Check the GCC `__int128` data type for this purpose.

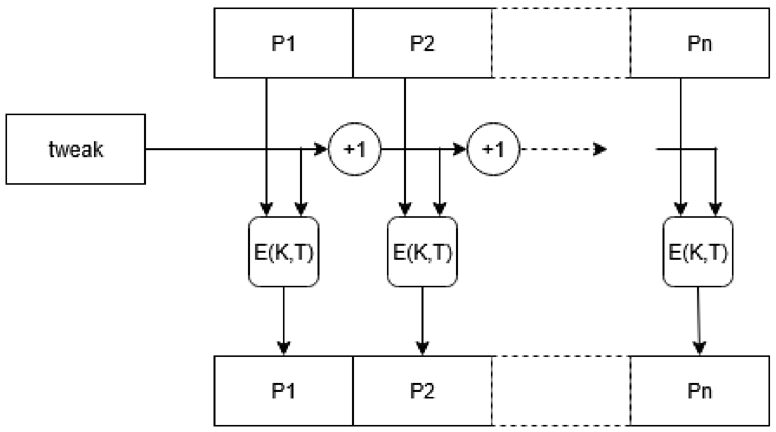

Figure 1: ECB-based cipher mode, with a different tweak per block. When the input is not block-aligned, the last two blocks are encrypted using the tweak as well but with the Ciphertext Stealing approach.

Besides implementing T-AES and its counter mode in a module (or library), you should implement two applications, `encrypt` and `decrypt`. These should receive one numerical parameter to select the AES key length (128, 192 or 256) and two textual passwords as arguments, which will be separately used to generate the two keys of T-AES – the normal AES key and tweak, in this order. In the absence of the second parameter, the program should operate with the normal AES. When the tweak is provided, the programs should use it with the counter mode previously referred; otherwise, should use AES in ECB mode.

The applications should process the input from `stdin` and produce a result to `stdout`. Use the ECB Ciphertext Stealing approach for the last two blocks if the input is not block aligned. Always consider the input a binary content (you cannot, in any case, use textual inputs).
Note: since you are reading from the `stdin`, you can only know if the input stream is not block-aligned when you reach the end of the stream. Therefore, you need to keep the last block on hold until reading the next one. Note: Ciphertext Stealing does not work with messages with one block or less. Do not consider that case, assume that messages are always longer than a block.

Create a third application, `speed`, to evaluate the relative performance of your T-AES implementation and one or more library implementation of XTS. For that, allocate a 4KB buffer (a memory page), fill it with random values (you can use `/dev/urandom` for that), and evaluate the time it takes to encrypt and decrypt the buffer with XTS (from the library) and T-AES (use your counter mode), both with and without AES-NI. Perform at least 100 000 measurements of each operation, and present the lowest values observed for each (i.e. the maximum achievable speed). For each measurement, use new, random keys. For accurate timing you can use the Linux system call `clock_gettime` function, which provides a nanosecond precision. Note that the measurements must encompass only encryptions and decryptions, and not AES or T-AES key-related set-up operations.

Create a fourth application, `stat`, that studies the effect of a tweak in T-AES. For that, successively encrypt the same value with an increasing tweak and compute the Hamming distance of the output block relatively to the previous output block (created with the tweak minus one). Then, produce a chart with the probability distribution of the observed Hamming distances. You can use a random value both for the input and the tweak for each application run. For getting relevant results, make a large number of measurements of the Hamming distance.

2.1 Library implementations of AES in Linux

In C, you can use these library implementations:

- The OpenSSL crypto library;
- The Nettle library.

In C++, you can use this library implementations:

- The Crypto++ library.

2.2 T-AES implementation with AES-NI

In order to have the minimum possible performance penalty when switching from an AES library version using AES-NI to T-AES, you can implement the latter using also AES-NI Intel assembly instructions, which are the following:

`AESKEYGENASSIST xmm1, xmm2/m128u, imm8`	Assist in round key generation using an 8-bit constant (`imm8`) with a 128-bit key specified in `xmm2/m128` and stores the result in `xmm1`. This instruction computes a value that needs to be combined with the key to generate the round key (presented below).
`AESENC xmm1, xmm2/m128`	Perform one encryption round (except the last one) over `xmm1` with the round key `xmm2/m128`.
`AESENCLAST xmm1, xmm2/m128`	Perform the last encryption round over `xmm1` with the round key `xmm2/m128`.
`AESDEC xmm1, xmm2/m128`	Perform one decryption round (except the last one) over `xmm1` with the round key `xmm2/m128`.
`AESDECLAST xmm1, xmm2/m128`	Perform the last decryption round over `xmm1` with the round key `xmm2/m128`.
`AESIMC xmm1, xmm2/m128`	Perform a transformation of a round key (1 to 9) to prepare it to be used in a decryption flow.

`xmm1` and `xmm2` are 128-bit registers, `m128` is the address of a 128-bit value and `imm8` is an 8-bit constant integer.

The GCC compiler has inline C functions for wrapping these AES-NI instructions. Those functions become available with the `-maes` compilation flag, and their prototype is the following:

```
// __m128i is a 128-bit integer type.

__m128i _mm_aeskeygenassist_si128 ( __m128i key, uint8_t const index );
__m128i _mm_aesenc_si128 ( __m128i input, __m128i rk );
__m128i _mm_aesenclast_si128 ( __m128i input, __m128i rk );
__m128i _mm_aesdec_si128 ( __m128i input, __m128i rk );
__m128i _mm_aesdeclast_si128 ( __m128i input, __m128i rk );
__m128i _mm_aesimc_si128 ( __m128i input );
```

The prototype of these functions is provided by the `wmmintrinc.h` file which exists under the GCC installation directory (at `/usr/lib/gcc/x86_64-linux-gnu/[GCC major version]/include`).

The AES encryption flow with AES-NI is as follows:

```
x = x ^ RK[0];
x = aesenc ( x, RK[1] );
x = aesenc ( x, RK[2] );
...
x = aesenc ( x, RK[9] );
x = aesenclast ( x, RK[10] );

// aesenc = AddRoundKey( MixColumns( ShiftRows ( SubBytes( x ) ) ), RK )
```

and the inverse, decryption flow is as follows:

```
// RK'[i] = aesimc( RK[i] ) for i in [1-9]

x = x ^ RK[10];
x = aesdec ( x, RK'[9] );
x = aesdec ( x, RK'[8] );
...
x = aesdec ( x, RK'1] );
x = aesdeclast ( x, RK[0] );

// aesdec = AddRoundKey( InvMixColumns( InvSubBytes( InvShiftRows( x ) ) ), RK' )
```

The setup of round keys is performed from the original key and the output of `AESKEYGENASSIST`, and can be implemented in C with intrinsic functions as follows:

```
// key is the main key
// index is the output of AESKEYGENASSIST

// _mm_shuffle_epi32, _mm_slli_si128 and _mm_xor_si128 are GCC internal, inline functions

__m128i get_round_key (__m128i key, __m128i index )
{
    __m128i tmp;

    index = _mm_shuffle_epi32( index, 0xff );
    tmp   = _mm_slli_si128( key, 0x4 );
    key   = _mm_xor_si128( key, tmp );
    tmp   = _mm_slli_si128( tmp, 0x4 );
    key   = _mm_xor_si128( key, tmp );
    tmp   = _mm_slli_si128( tmp, 0x4 );
    key   = _mm_xor_si128( key, tmp );
    key   = _mm_xor_si128( key, index );

    return key; // Round key
}
```

3 Evaluation

The project will be evaluated as follows:

- Implementation of T-AES (in C/C++, without AES-NI): 10%;
- Implementation of T-AES in C/C++ with AES-NI assembly instructions: 20%;
- Implementation of ECB-based cipher mode with a counter and T-AES: 10%;
- Implementation of the applications: 10% for `encrypt`, 10% for `decrypt`, 10% for `speed`, 10% for `stat`;
- Written report, with a complete explanations of the strategies followed and the results achieved: 20%

Bonus (10%): Implement the `speed` application using the `openssl` application (the source code is available). You have to add to it XTS, T-AES and your ECB-based cipher mode.

4 Homework delivery

Send your code to the course teachers through eLearning (a submission link will be provided). Include a small report, with no more than 10 pages, describing the implementation (not a complete copy of the code developed!). Code snippets may be used to illustrate your implementation.

Every piece of code imported from anywhere must be stated in the report and in the code itself. Failure to do so will be penalised.

5 References

- *Intel® Advanced Encryption Standard (AES) New Instructions Set*, Shay Gueron, White Paper, 2012, `https://www.intel.com/content/dam/develop/external/us/en/documents/aes-wp-2012-09-22-v01-165683.pdf`
- *Advanced Encryption Standard (AES)*, Morris J. Dworkin, Elaine B. Barker, James R. Nechvatal, James Foti, Lawrence E. Bassham, E. Roback, James F. Dray Jr., NIST FIPS 197, 2001, `https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf`
- *Ciphertext stealing*, `https://en.wikipedia.org/wiki/Ciphertext_stealing`