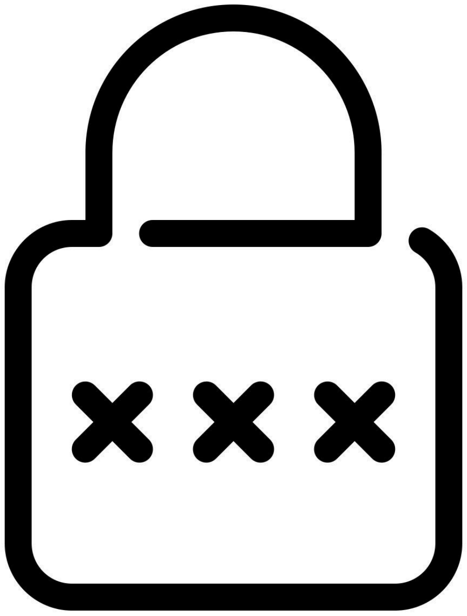
Tweakable AES (T-AES)

Criptografia Aplicada



Pedro Costa 130720

Henrique Dias 131144

Índice

1. Introduction and Objectives

The security of modern data storage and transmission relies fundamentally on block ciphers like the Advanced Encryption Standard (AES). While AES provides strong confidentiality, certain applications, most notably full disk encryption, require a more robust security model that can handle repeated encryption of different data blocks under the same key without introducing vulnerabilities.

This necessity gave rise to tweakable ciphers. These are a specialized class of encryption modes that introduce a non-secret, publicly known input, the tweak (T), alongside the usual key (K) . The tweak ensures that even if the same key encrypts two identical blocks, the resulting ciphertexts are entirely different, drastically increasing resistance against sophisticated cryptanalytic attacks.

This project aimed to implement a tweakable version of AES, referred to as T-AES. The core objective was to implement T-AES by modifying only a single Round Key (RK) during the encryption and decryption process. This modification involves an unsigned arithmetic addition of the 128-bit tweak to a chosen 128-bit Round Key:

$$RK_{modified} = RK[N] + T \left(mod\ 2^{128}\right)$$

Where N is selected based on the key size (RK5 for 128-bit keys, RK6 for 192-bit keys, or RK7 for 256-bit keys).

In addition to implementing the T-AES core in both software (SW) and using AES-NI hardware intrinsics, the project included:

1. Implementation of an ECB-based block counter mode that leverages the T-AES modification by incrementing the tweak for each processed data block.
2. Development of the required applications, including:
    a. encrypt / decrypt: For secure data stream processing, utilizing Ciphertext Stealing (CTS).
    b. speed: To benchmark the performance of the implemented T-AES modes (SW and AES-NI) against standard XTS-AES.

c. stat: To verify the cryptographic robustness of the T-AES modification by calculating the Hamming Distance and assessing adherence to the Strict Avalanche Criterion (SAC).

2. T-AES Implementation

The core task involves integrating the Tweak (T) into the cipher's structure and applying it in an ECB-based counter mode. The implementation uses the Main AES Key (K) and the Tweak (T) derived from two textual passwords using SHA-256 hashing.

2.1 Tweak Modification Logic (Software Implementation)

The T-AES modification requires performing an unsigned 128-bit arithmetic addition of the Tweak (T) with the selected 128-bit Round Key (RK[N]). Since the AES Round Keys are handled as an array of 32-bit words, we implemented the 128-bit addition through a manual 32-bit ripple-carry addition in the add_128_bit function, explicitly managing the carry bit propagation.

The index N of the Round Key to be modified is selected dynamically: RK5 for 128-bit keys, RK6 for 192-bit keys, and RK7 for 256-bit keys. The identical tweak-modified round key is used for both encryption and decryption paths.

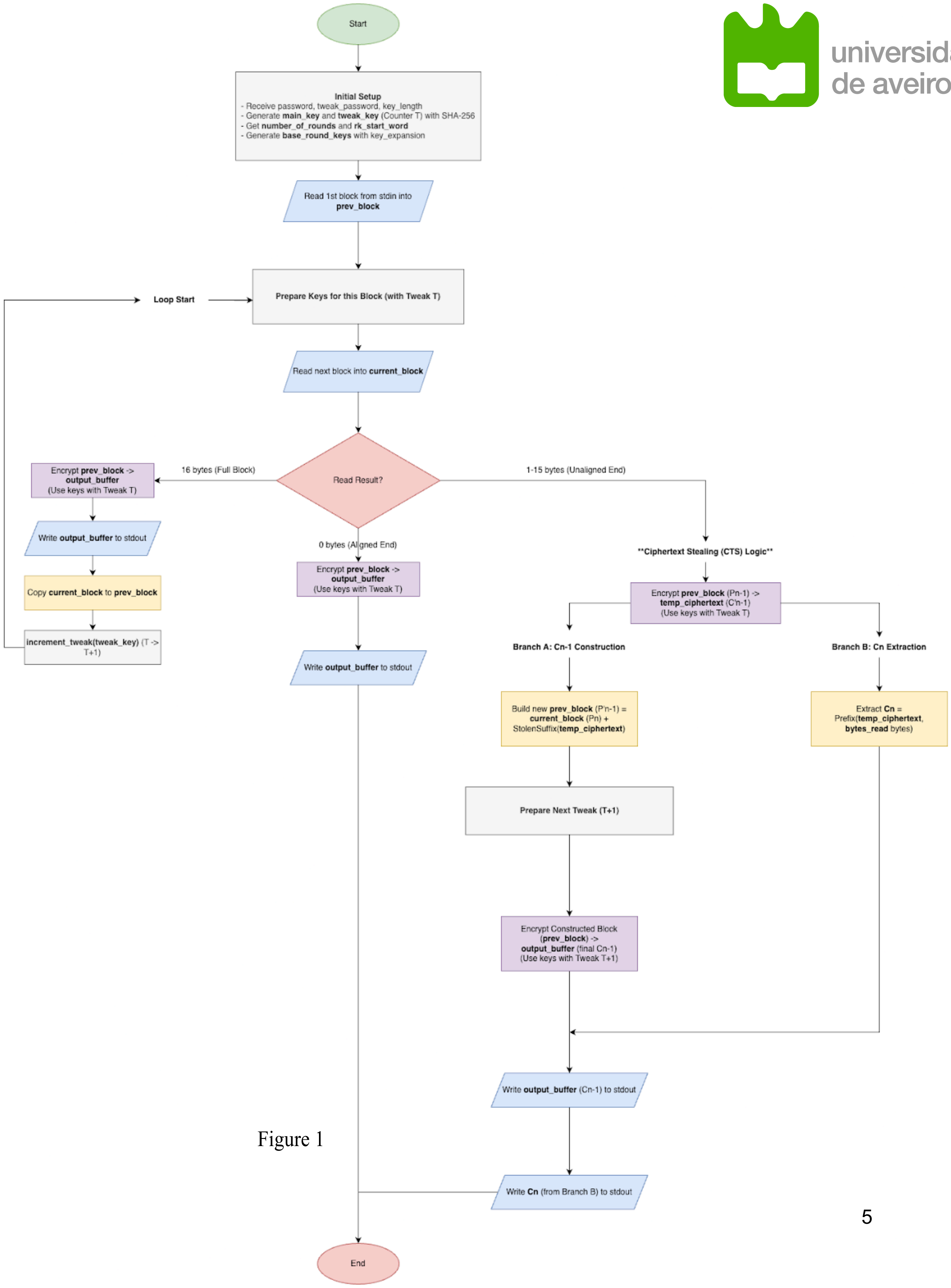
2.2 ECB-based Counter Mode and Ciphertext Stealing (CTS)

The T-AES counter mode operates in a loop, processing input blocks from stdin (as illustrated in Figure 1).

Counter Tweak: Before encrypting each block, the necessary round keys are prepared by applying the current Tweak T (Step: "Prepare Keys for this Block"). After a full block is encrypted, the Tweak is arithmetically incremented by one for the next block (Step: "Increment_tweak(tweak_key) (T -> T+1)").

Mode Selection: The applications operate in T-AES Counter Mode only if two passwords are provided (Key and Tweak). If only one password is provided, standard AES in ECB mode is used.

Ciphertext Stealing (CTS): The entire process handles the last two blocks using CTS logic for non-aligned input (1-15 bytes at the end). This involves encrypting the penultimate block (Pn-1) with Tweak T to obtain C'n-1, constructing P'n-1 by stealing bytes from C'n-1, preparing the next Tweak (T+1), and encrypting P'n-1 with T+1 to get the final Cn-1.



Start
Initial Setup
- Receive password, tweak_password, key_length
- Generate main_key and tweak_key (Counter T) with SHA-256
- Get number_of_rounds and rk_start_word
- Generate base_round_keys with key_expansion
Read 1st block from stdin into prev_block
Loop Start
Prepare Keys for this Block (with Tweak T)
Read next block into current_block
Read Result?
16 bytes (Full Block)
Encrypt prev_block -> output_buffer (Use keys with Tweak T)
Write output_buffer to stdout
Copy current_block to prev_block
increment_tweak(tweak_key) (T -> T+1)
0 bytes (Aligned End)
Encrypt prev_block -> output_buffer (Use keys with Tweak T)
Write output_buffer to stdout
1-15 bytes (Unaligned End)
Ciphertext Stealing (CTS) Logic
Encrypt prev_block (Pn-1) -> temp_ciphertext (C'n-1) (Use keys with Tweak T)
Branch A: Cn-1 Construction
Branch B: Cn Extraction
Build new prev_block (P'n-1) = current_block (Pn) + StolenSuffix(temp_ciphertext)
Extract Cn = Prefix(temp_ciphertext, bytes_read bytes)
Prepare Next Tweak (T+1)
Encrypt Constructed Block (prev_block) -> output_buffer (final Cn-1) (Use keys with Tweak T+1)
Write output_buffer (Cn-1) to stdout
Write Cn (from Branch B) to stdout
End

Figure 1

3. T-AES with AES-NI instructions

The T-AES 128-bit arithmetic addition is performed manually using standard C types (uint64_t) in the add_128_bit_ni function, as AES-NI lacks a native intrinsic for this operation. Crucially, in the decryption path, the tweak must be correctly integrated with the Inverse Cipher Key Schedule. To ensure correctness, the Tweak arithmetic addition is applied to the base round key before it is transformed by the Inverse MixColumns intrinsic (_mm_aesimc_si128). This guarantees the modified key state is accurately inverted for the decryption process.

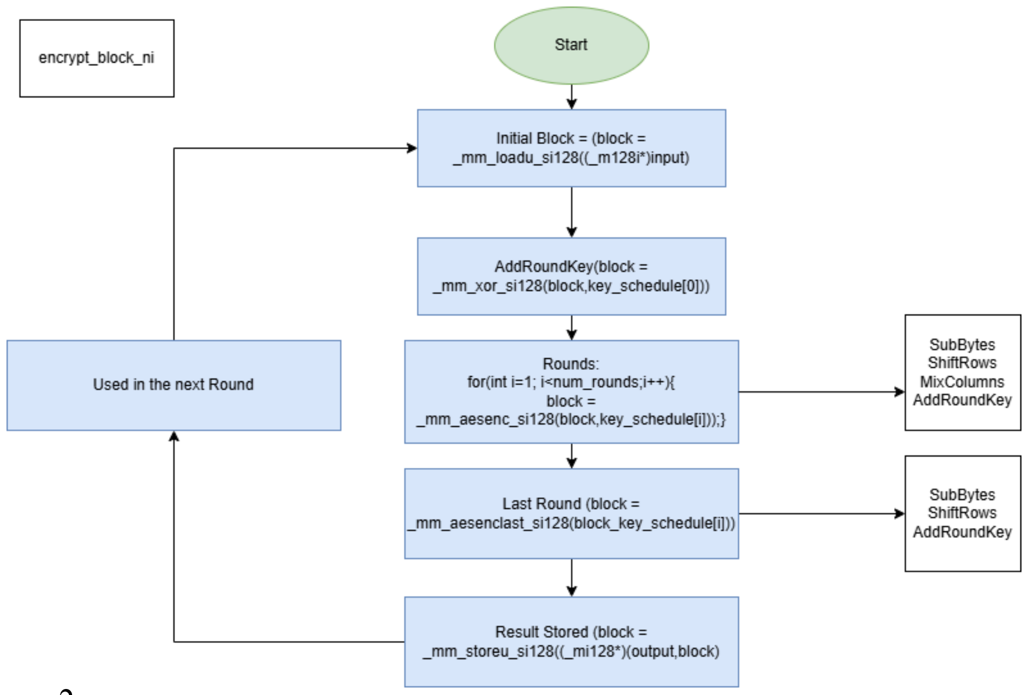

encrypt_block_ni
Start
Initial Block = (block = _mm_loadu_si128((_m128i*)input)
AddRoundKey(block = _mm_xor_si128(block,key_schedule[0]))
Rounds: for(int i=1; i<num_rounds;i++){ block = _mm_aesenc_si128(block,key_schedule[i]));}
SubBytes ShiftRows MixColumns AddRoundKey
Used in the next Round
Last Round (block = _mm_aesenclast_si128(block_key_schedule[i]))
SubBytes ShiftRows AddRoundKey
Result Stored (block = _mm_storeu_si128((_mi128*)(output,block)

Figure 2

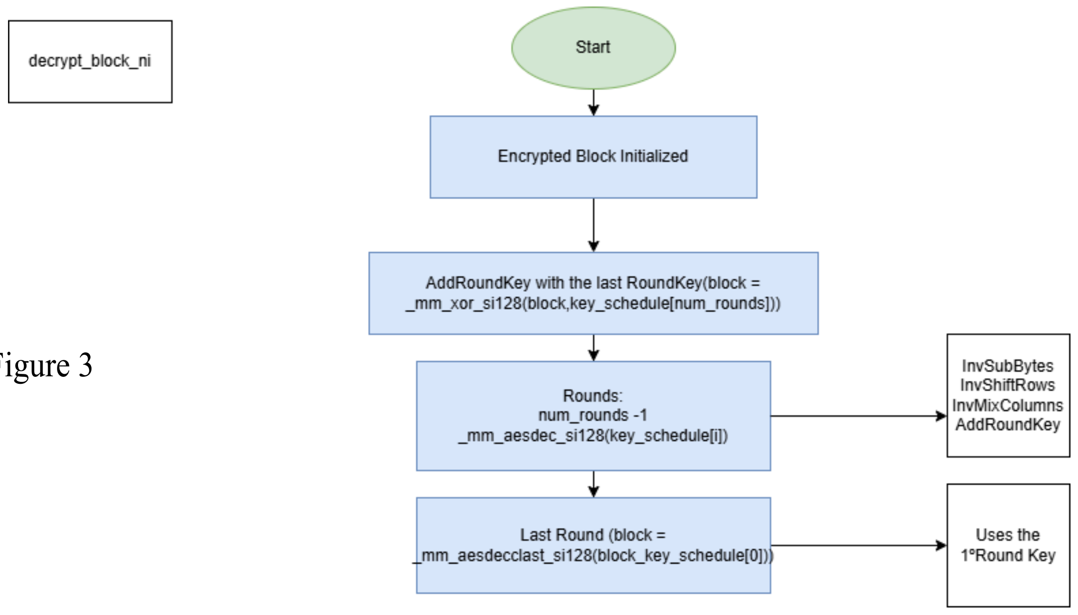
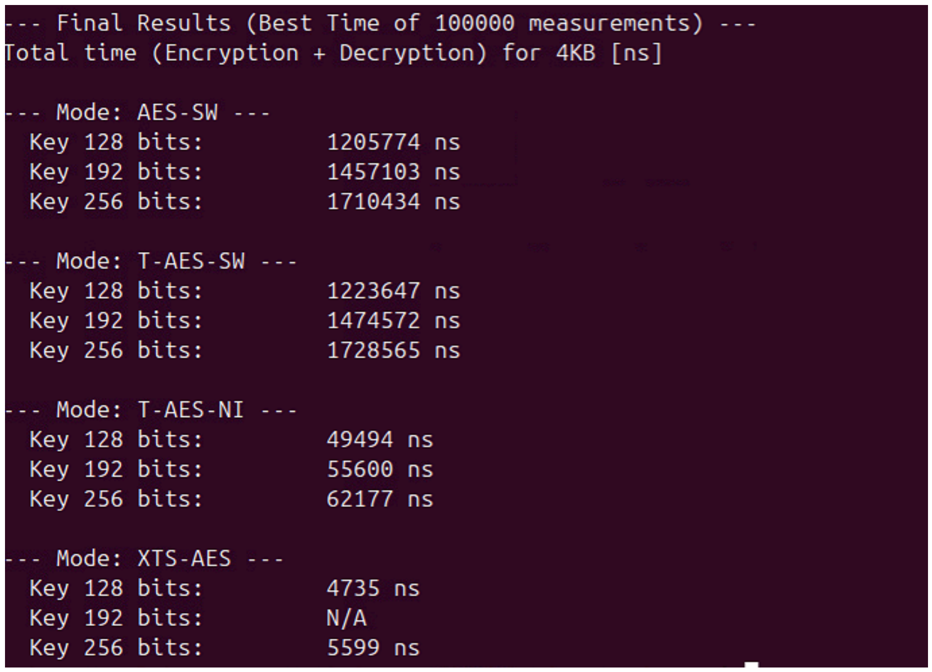

decrypt_block_ni
Start
Encrypted Block Initialized
AddRoundKey with the last RoundKey(block = _mm_xor_si128(block,key_schedule[num_rounds]))
Rounds: num_rounds -1 _mm_aesdec_si128(key_schedule[i])
InvSubBytes InvShiftRows InvMixColumns AddRoundKey
Last Round (block = _mm_aesdecclast_si128(block_key_schedule[0]))
Uses the 1ºRound Key

Figure 3

4. Speed and Stat

For the speed file, the objective was to create a 4KB buffer and compare the speed and time that it takes to each of the different encryption methods to encrypt this 4KB file. In the speed_helper.c file, we have every single method that each of these encryption ways need to function, for example the t_aes_ni full encryption and decryption are defined in this file. These functions are called in the speed.c file, as well as the creation of the 4KB file, making it easier to see and analyze data. Notice that XTS-AES doesn’t show any result for 192-bits because for it to actually work, it needs two independent AES keys of equal length. With that said, with 128-bits key makes a total of 256 bits. With 256-bits key makes a total of 512 bits. If you used 192-bit AES keys, the total key length would be 384 bits, and no standard defines parameter handling or key schedule for 384-bit total keys in XTS mode. Not only that but, as we go forward, the time gets reduced, meaning that as days goes by, technology gets better, improving speed, space, computational power and the time to compute AES.

The speed test methodology was executed with rigor: 100,000 measurements were performed, generating new, random keys and tweaks for every single run. Timing was measured using a high-resolution clock, clock_gettime, to ensure nanosecond precision. The measurements excluded key setup and key expansion operations but included the full Encryption + Decryption pipeline for the 4KB buffer. The final results display the lowest time observed.



```
--- Final Results (Best Time of 100000 measurements) ---
Total time (Encryption + Decryption) for 4KB [ns]

--- Mode: AES-SW ---
  Key 128 bits:         1205774 ns
  Key 192 bits:         1457103 ns
  Key 256 bits:         1710434 ns

--- Mode: T-AES-SW ---
  Key 128 bits:         1223647 ns
  Key 192 bits:         1474572 ns
  Key 256 bits:         1728565 ns

--- Mode: T-AES-NI ---
  Key 128 bits:         49494 ns
  Key 192 bits:         55600 ns
  Key 256 bits:         62177 ns

--- Mode: XTS-AES ---
  Key 128 bits:         4735 ns
  Key 192 bits:         N/A
  Key 256 bits:         5599 ns
```

Figure 4

For the stat file, we are analyzing the effect of a tweak in T-AES. For this, we are calculating the Hamming distance of the output block relatively to the previous output block. Basically, the Hamming distance between two equal-length strings of symbols is the number of positions at which the corresponding symbols are different. For this implementation, we created the stat.c file.

5. Conclusion

First of all, to discuss results we must talk about the stat task and its results. The reason why we observe higher values in the middle of the Hamming distance distribution is due to principles of probability and randomness in cryptography, specifically the concept of the Strict Avalanche Criterion (SAC). For this to work it has two conditions: Flipping one input bit should cause every output bit to flip with a chance of 50% and this chance should be independent bit to bit. So, since each output has a 50% chance of flipping, the most likely outcome is that exactly half of the bits will flip. Not only that but it is extremely unlikely that a true random 128-bit output will have only 1 changed bit or 127 changed bits.

The T-AES avalanche effect was studied by performing 100,000 measurements of the Hamming distance between successive output blocks E(K, T) and E(K, T+1), ensuring the resulting probability distribution was statistically robust.

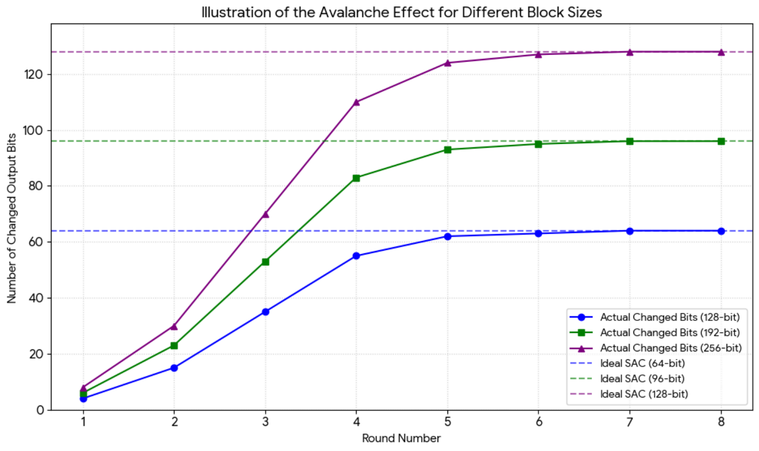


Figure 5

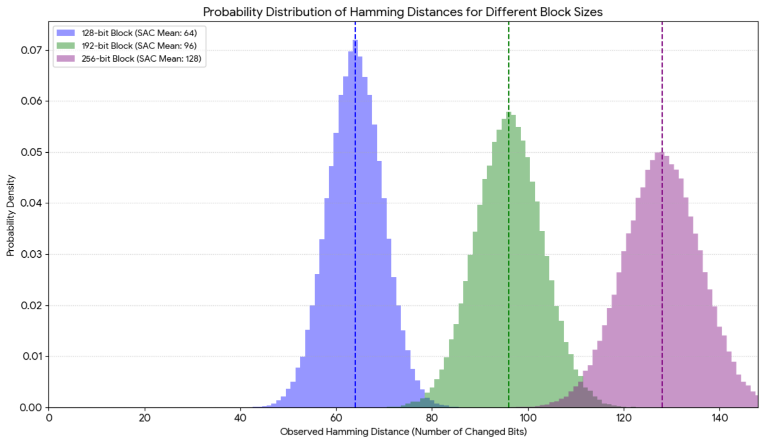


Figure 6

Secondly, about the speed and time that each encryption mode took we can see that XTS-AES was the fastest. But the transition from software(T-AES-SW) to hardware(T-AES-NI) can't go unmarked as it has shown to be almost x24 times faster than T-AES-SW. But nothing could actually beat XTS-AES, as its structure allows for the independent and parallel encryption/decryption of data blocks, minimizing dependencies and this way, maximizing efficiency, making it x302 times faster.

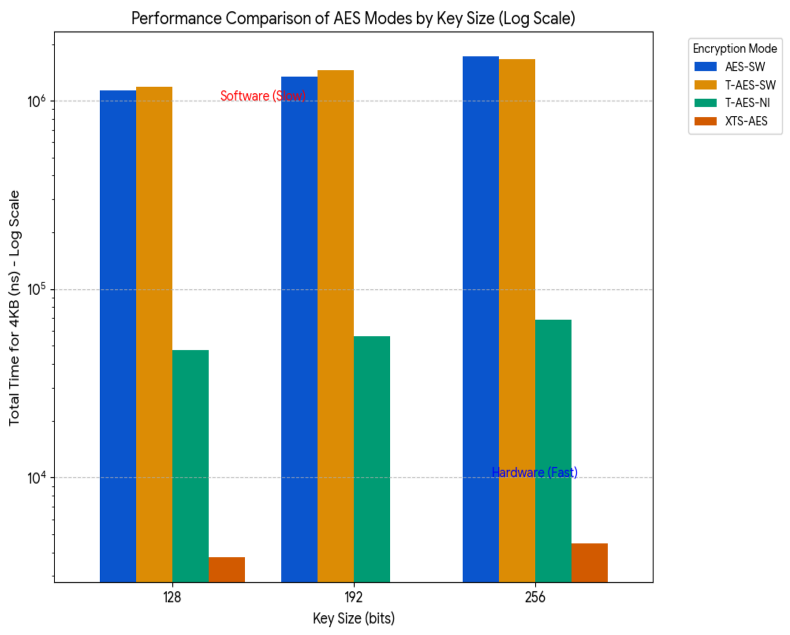


Figure 7

6. Webgraphy

[1] OpenSSL Project. (2025). *openssl/openssl: TLS/SSL and crypto library* [Repositório GitHub]. GitHub. https://github.com/openssl/openssl

[2] Computer Science for Engineers. (2018, maio 2). *How to solve AES example? | AES Encryption Example | How AES explained* [Vídeo]. YouTube. https://www.youtube.com/watch?v=I68uBhHdnM4

[3] Computerphile. (2019, novembro 22). *AES Explained (Advanced Encryption Standard)* [Vídeo]. YouTube. https://www.youtube.com/watch?v=O4xNJsjtN6E

[4] PakCrypt. (2024, janeiro 25). *Algo Optmz AES* [White paper]. https://pakcrypt.org/assets/whitepapers/algo_optmz_aes.pdf

[5] GeeksforGeeks. (s.d.). *Concepts of Hamming distance*. GeeksforGeeks. https://www.geeksforgeeks.org/dsa/concepts-of-hamming-distance/

[6] Wikipedia. (2025, 4 de novembro). *Hamming distance*. Wikipedia. https://en.wikipedia.org/wiki/Hamming_distance