T-AES: Software and Hardware-Accelerated AES with Ciphertext Stealing

Applied Cryptography Project
Computer Architecture
University of Aveiro
November 2025

***Abstract*—This report presents T-AES, a tweakable Advanced Encryption Standard implementation featuring both software and hardware-accelerated (AES-NI) versions. The tweak, a 128-bit value combined via arithmetic addition (mod 2^{128}) with the middle round key, enables deterministic variation across encrypted blocks without requiring secrecy. The implementation supports AES-128, AES-192, and AES-256 in an ECB-based counter mode where the tweak increments per block. Ciphertext stealing handles non-block-aligned data without padding. The project includes encrypt and decrypt command-line applications, a speed benchmark application (speed.cpp) comparing T-AES software, T-AES AES-NI, and OpenSSL XTS implementations, and a statistical analysis application (stat.cpp) that measures Hamming distance distributions to validate the cryptographic properties of the tweak. Both T-AES implementations are validated to produce identical outputs, demonstrating cryptographic equivalence.**

***Index Terms*—AES, encryption, hardware acceleration, AES-NI, ciphertext stealing, ECB mode, performance comparison**

I. Introduction

The Advanced Encryption Standard (AES) has been the widely adopted symmetric encryption algorithm since its standardization by NIST in 2001. Tweakable block ciphers extend the standard AES model by introducing a tweak parameter, a non-secret value that modifies the encryption without requiring a key change. A prominent example is XTS (XEX-based Tweaked-codebook mode with ciphertext Stealing), used in full-disk encryption, which applies the tweak at the input and output of AES-128 and includes ciphertext stealing to handle non-block-aligned data.

This project implements T-AES, a variant where a tweak is inserted in the middle of the AES substitution-permutation network (at the middle round key) rather than at input/output. Combined with an ECB-mode counter that increments the tweak per block and ciphertext stealing for final blocks, T-AES provides a complete tweakable encryption scheme. The implementation follows NIST FIPS 197 specifications and includes support for all three standard key sizes (128, 192, and 256 bits).

Modern processors have evolved to include specialized instructions for AES operations. Intel's AES New Instructions (AES-NI), introduced in 2010, provide hardware-level acceleration for AES encryption and decryption. These instructions offer substantial performance improvements while maintaining resistance to timing-based side-channel attacks that can affect software implementations. This project implements both software and hardware-accelerated versions to compare performance approaches and validate cryptographic equivalence across architectures. A notable feature is the ciphertext stealing technique, which eliminates the need for padding when processing data that doesn't align to 16-byte block boundaries.

II. Baseline Requirements and Specifications

The T-AES project was designed with clear functional and implementation requirements. The goal was to create a command-line tool capable of encrypting and decrypting data streams using both software and hardware-accelerated versions of AES, while maintaining identical cryptographic behavior in both implementations.

A. Functional Requirements

The system operates entirely through command-line interaction, using Unix-style pipelines to maximize flexibility and interoperability. Instead of relying on file-based input and output, all data is processed as binary streams through `stdin` and `stdout`. This approach allows the encryption and decryption of data of arbitrary size without the need to load entire files into memory, making the system suitable for stream-based processing and integration with other command-line utilities.

The encryption algorithm supports the three key sizes defined in the AES standard: 128, 192, and 256 bits. The block size is fixed at 128 bits (16 bytes), following the FIPS 197 specification. The tweak is combined with the round key via arithmetic addition (mod 2^{128}) at the middle round (RK5 for AES-128, RK6 for AES-192, RK7 for AES-256). A practical advantage of using arithmetic addition is that during decryption, the same tweak value is added to the round key without requiring subtraction. By default, T-AES operates in Electronic Codebook (ECB) mode, where each block is processed independently. An optional tweak parameter can be provided to enable counter-mode operation where the tweak increments per block, enabling deterministic variation between otherwise identical plaintext blocks. The applications detect this through argument count, as shown in Listing 1:

Listing 1. Tweak Detection via Argument Count (encrypt.cpp)

```
if (argc < 3) {
    cout << "Args: <aes_size> <password> [tweak_password]"
         << endl;
    return 1;
}
if (argc == 4) {
    TWEAK = true;
}
```

Handling plaintext that is not a multiple of 16 bytes (but made of 2 blocks, having as a requirement the guarantee that we weren't treating example cases with less than 2 blocks, although for good practices checks were still included) is achieved using ciphertext stealing (CTS). Unlike padding schemes such as PKCS#7, CTS preserves the exact length of the plaintext in the resulting ciphertext. When the final block is incomplete, the algorithm borrows bytes from the previous ciphertext block to construct a complete final block. The core encryption logic in encrypt.cpp implements this as follows:

Listing 2. Ciphertext Stealing Encryption (encrypt.cpp)

```
if (current_block.size() < 16) {
    // Incomplete final block: borrow from previous
        ciphertext
    size_t steal_size = 16 - current_block.size();
    vector<uint8_t> cipher_to_append(
        previous_encrypted_block.end() - steal_size,
        previous_encrypted_block.end());
    current_block.insert(current_block.end(),
        cipher_to_append.begin(), cipher_to_append.end());

    ciphertext_block = aes.encrypt_block(current_block);
    cipherBlocks[i - 1] = vector<uint8_t>(
        previous_encrypted_block.begin(),
        previous_encrypted_block.end() - steal_size);
    cipherBlocks.push_back(ciphertext_block);
}
```

This technique maintains proper block alignment without introducing artificial data, ensuring data integrity and enabling decryption to perfectly reconstruct the original plaintext.

B. Implementation Requirements

Two independent implementations were developed to provide a fair comparison between software and hardware encryption approaches. The software version is written entirely in C++, using lookup tables, bitwise operations, and standard arithmetic to implement the AES transformations. The hardware version, on the other hand, leverages Intel's AES-NI instruction set to perform most AES operations directly in the processor, significantly reducing instruction count and latency.

Both implementations were designed to be functionally equivalent. Given the same plaintext, key, and optional tweak, they must produce identical ciphertext. To ensure modularity and maintainability, the build system generates separate binaries for the software and hardware versions. This separation allows independent benchmarking and validation while keeping the codebase consistent across both modes.

Key derivation in T-AES is performed through a simple yet secure process. The user provides a password as a command-line argument, which is then hashed using SHA-256 to generate a fixed-length 256-bit key. Depending on the chosen AES mode, the appropriate number of bytes (16, 24, or 32) is extracted from the hash to serve as the encryption key. The same process can be repeated to derive the optional tweak value when specified, ensuring deterministic and reproducible behavior across executions.

1) Command-Line Interface: The encrypt and decrypt applications (encrypt.cpp/decrypt.cpp for software, encrypt_aesni.cpp/decrypt_aesni.cpp for hardware) accept arguments in the format:

```
./bin/encrypt <aes_size> <password> [tweak_password]
./bin/decrypt <aes_size> <password> [tweak_password]
./bin/encrypt_aesni <aes_size> <password> [tweak_password]
./bin/decrypt_aesni <aes_size> <password> [tweak_password]
```

Where `aes_size` is 128, 192, or 256. The third argument is optional. When provided, it enables counter-mode operation with tweak incrementing per block. Data flows through standard Unix pipes, allowing integration with other tools. Example usage for software AES without tweak:

```
./bin/encrypt 128 mypassword < plaintext.bin > ciphertext.bin
./bin/decrypt 128 mypassword < ciphertext.bin > decrypted.bin
```

For hardware-accelerated encryption with tweak:

```
./bin/encrypt_aesni 256 password1 password2 < plaintext.bin > cipher.bin
./bin/decrypt_aesni 256 password1 password2 < cipher.bin > plaintext.bin
```

2) Building: All applications are built automatically using the provided Makefile. Running `make` (or `make all`) compiles all six programs: encrypt, decrypt, encrypt_aesni, decrypt_aesni, speed, and stat. The build system automatically applies appropriate compiler flags: standard C++17 with optimizations (-O3) for software implementations, and additional AES-NI flags (-maes -msse4.1) for hardware-accelerated versions. OpenSSL libraries are linked automatically via `-lssl -lcrypto`. All binaries are placed in the `bin/` directory.

C. Validation Requirements

Validation was a central aspect of the project to guarantee correctness, equivalence, and robustness. The system was tested to ensure that encrypting and then decrypting any given input reproduces the original plaintext exactly, verifying the accuracy of both implementations. Cross-validation was also performed, where data encrypted with the software version was decrypted using the hardware version and vice versa, confirming complete compatibility.

Additional tests were conducted with incorrect keys and tweaks to verify that decryption fails gracefully without producing false positives. Edge cases, such as empty inputs, single-byte files, and non-block-aligned data, were thoroughly tested to confirm the proper functioning of ciphertext stealing. Finally, all combinations of key sizes, tweaks, and implementation modes were tested to ensure consistent behavior under all supported configurations.

Overall, these baseline requirements established a strong foundation for the development of T-AES, ensuring that both implementations would be secure, efficient, and directly comparable in terms of performance and correctness.

III. Software Implementation Details

The software implementation provides a pure C++ version of AES that operates independently of hardware acceleration features. This implementation uses lookup tables and bitwise operations to perform the AES transformations defined in NIST FIPS 197.

A. AES Core Algorithm

The AES algorithm operates on a 128-bit (16-byte) state represented as a 4×4 matrix of bytes. Encryption proceeds through multiple rounds (10, 12, or 14 depending on key size), with each round applying four transformations.

1) State Representation: The implementation stores the state as a linear array of 16 bytes that logically represents a column-major 4×4 matrix:

```
State[16] = {s0, s1, ..., s13, s14, s15}

Matrix form:  s0  s4  s8  s12
              s1  s5  s9  s13
              s2  s6  s10 s14
              s3  s7  s11 s15
```

This column-major layout aligns with AES specification and simplifies the MixColumns operation.

2) SubBytes Transformation: SubBytes provides non-linearity by substituting each byte using the AES S-box.

Implementation Approach:

- Pre-computed lookup table with 256 entries stored in memory
- Each byte b in the state is replaced with $S[b]$
- S-box constructed from multiplicative inverse in GF(2^8) plus affine transformation
- Simple array indexing: `state[i] = sbox[state[i]]` performs substitution

3) ShiftRows Transformation: ShiftRows performs cyclic shifts on each row of the state matrix to provide diffusion.

Operation:

- Row 0: No shift
- Row 1: Shift left by 1 byte (circular)
- Row 2: Shift left by 2 bytes
- Row 3: Shift left by 3 bytes

This is implemented using a temporary buffer or in-place permutation to avoid overwriting needed values.

4) MixColumns Transformation: MixColumns provides diffusion by mixing bytes within each column using Galois Field arithmetic.

Mathematical Foundation:

Each column is treated as a polynomial and multiplied by the fixed polynomial $c(x) = \{03\}x^3 + \{01\}x^2 + \{01\}x + \{02\}$ in GF(2^8).

For column $[a_0, a_1, a_2, a_3]^T$, the output is:

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

Where multiplication is in GF(2^8) and addition is XOR.

Implementation Strategy: Multiplication by 02 in GF(2^8):

```
uint8_t gmul2(uint8_t a) {
    return (a << 1) ^ (((a >> 7) & 1) * 0x1B);
}
```

Multiplication by 03: $\{03\} \cdot a = \{02\} \cdot a \oplus a$

Many implementations use pre-computed tables for efficiency:

- `mul2[256]`: Table for multiplication by 02
- `mul3[256]`: Table for multiplication by 03

Important: The last round of encryption doesn't apply MixColumns to make encryption and decryption structurally similar.

5) AddRoundKey Transformation: AddRoundKey incorporates key material by XORing the state with the round key.

Implementation:

```
for (int i = 0; i < 16; i++) {
    state[i] ^= roundKey[round * 16 + i];
}
```

This is the simplest and fastest operation for the software implementation, as XOR is a fundamental CPU instruction.

B. Key Expansion

Before encryption, the cipher key is expanded into a key schedule containing all round keys.

Expansion Process:

For AES-128 (128-bit key):

- Input: 4 words (16 bytes)
- Output: 44 words (11 round keys × 4 words)
- Rounds: 10 + initial AddRoundKey

Algorithm Steps:

1) Copy original key as first 4 words
2) For each subsequent word $w[i]$:
   - If $i \mod 4 = 0$:
     - Rotate previous word left by 1 byte
     - Apply SubBytes to each byte
     - XOR with round constant Rcon[i/4]
     - XOR with $w[i-4]$
   - Otherwise: $w[i] = w[i-1] \oplus w[i-4]$

Round Constants: Rcon values are powers of x in GF(2^8): Rcon[1] = 0x01, Rcon[2] = 0x02, Rcon[3] = 0x04, ..., Rcon[10] = 0x36

C. Full Encryption Process

Encryption Algorithm:

AddRoundKey (using key words 0-3)
for round = 1 to Nr-1 **do**
  SubBytes
  ShiftRows
  MixColumns
  AddRoundKey
end for
SubBytes (final round)
ShiftRows (final round)
AddRoundKey (final round, no MixColumns)

Where N_r = 10 for AES-128, 12 for AES-192, 14 for AES-256.

D. Decryption Implementation

Decryption uses inverse transformations in reverse order:

- InvSubBytes: Uses inverse S-box
- InvShiftRows: Shifts right instead of left
- InvMixColumns: Uses different matrix for GF multiplication
- AddRoundKey: Same as encryption (XOR is its own inverse)

E. Ciphertext Stealing Implementation

Ciphertext stealing (CTS) enables secure encryption of data that does not align to block boundaries without padding. The encrypt.cpp and decrypt.cpp applications implement the ECB-based CTS approach described in NIST standards.

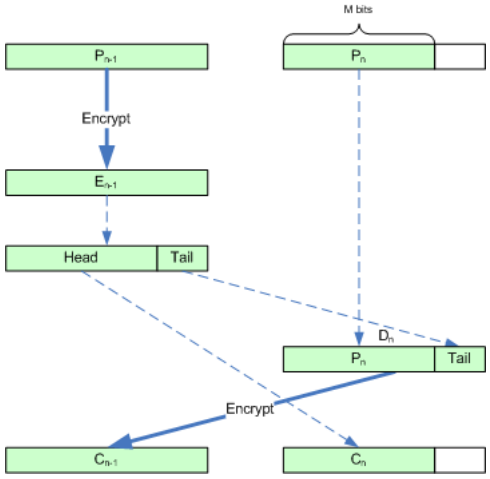

Fig. 1. Ciphertext Stealing

Encryption Algorithm: Given plaintext blocks $P_0, P_1, ..., P_{n-1}, P_n$ where P_n is partial (M ¡ 16 bytes), encryption proceeds as:

Encrypt all full blocks normally: $C_0, C_1, ..., C_{n-1}$
$steal\_size \leftarrow 16 - M$
Concatenate P_n with last $steal\_size$ bytes of C_{n-1} to form 16-byte block
$C_n \leftarrow$ Encrypt($P_n \parallel$ tail of C_{n-1})
$C'_{n-1} \leftarrow$ first M bytes of C_{n-1} (truncated)
Output: $C_0, C_1, ..., C'_{n-1}, C_n$

Decryption Algorithm: The symmetric reverse process is implemented in decrypt.cpp. Upon detecting the partial final block (size < 16 bytes), the decryption reconstructs the full ciphertext block by combining the truncated previous block with the current bytes, decrypts it to recover both plaintext and stolen bytes, reconstructs the penultimate ciphertext, and decrypts it to recover the previous plaintext block. This process, shown in decrypt.cpp lines 119-161, ensures the original plaintext is perfectly reconstructed without data loss.

F. Counter-Based Tweak Increment

When the optional tweak password is provided, the applications activate counter mode where the tweak increments per encrypted block. Both encrypt.cpp and encrypt_aesni.cpp implement this mechanism identically. After each block encryption, the tweak is incremented as a 128-bit big-endian integer using the utility function from utils.hpp:

Listing 3. Counter Mode Tweak Increment

```
vector<uint8_t> tweak_for_block = tweak;

for (size_t i = 0; i < all_blocks.size(); i++) {
    // Encrypt current block with current tweak
    ciphertext_block = aes.encrypt_block(current_block);
    cipherBlocks.push_back(ciphertext_block);

    // Increment tweak for next block
    if (TWEAK) {
        utils::increment_tweak(tweak_for_block);
    }
}
```

The increment_tweak function (utils.hpp) treats the tweak as a 128-bit big-endian integer with carry propagation, ensuring unique tweak values across blocks.

IV. Hardware Implementation Details

The hardware-accelerated implementation leverages Intel AES-NI (AES New Instructions) introduced in 2010. These specialized CPU instructions perform AES operations directly in hardware, providing significant performance and security advantages. The implementations (encrypt_aesni.cpp and decrypt_aesni.cpp) use the AESNI class from include/AESNI.hpp, which wraps the AES-NI intrinsics while maintaining identical encryption semantics to the software version.

A. AES-NI Instruction Set

Intel AES-NI provides six instructions specifically for AES operations, operating on 128-bit XMM registers.

1) Core Encryption Instructions: **AESENC** - AES Single Round Encryption:

- Intrinsic: `_mm_aesenc_si128(state, roundkey)`
- Operation: Performs SubBytes + ShiftRows + MixColumns + AddRoundKey
- Fuses four transformations into single hardware operation
- Throughput: 1-2 instructions per cycle with pipelining

AESENCLAST - AES Last Round Encryption:

- Intrinsic: `_mm_aesenclast_si128(state, roundkey)`
- Operation: SubBytes + ShiftRows + AddRoundKey (no MixColumns)
- Used for the final encryption round
- Same latency as AESENC

2) Core Decryption Instructions: **AESDEC** - AES Single Round Decryption:

- Intrinsic: `_mm_aesdec_si128(state, roundkey)`
- Operation: InvSubBytes + InvShiftRows + InvMixColumns + AddRoundKey
- Equivalent to AESENC but with inverse operations

AESDECLAST - AES Last Round Decryption:

- Intrinsic: `_mm_aesdeclast_si128(state, roundkey)`
- Operation: InvSubBytes + InvShiftRows + AddRoundKey

3) Key Expansion Instruction: **AESKEYGENASSIST** - AES Key Generation Assist:

- Intrinsic: `_mm_aeskeygenassist_si128(key, rcon)`
- Assists with key schedule generation
- Performs rotation, SubBytes application, and round constant XOR
- Software still handles remaining XOR operations

B. Hardware Encryption Implementation

Basic Encryption Flow:

```
__m128i aesni_encrypt_block(__m128i plaintext,
                            __m128i* key_schedule) {
    __m128i state = plaintext;
    state = _mm_xor_si128(state, key_schedule[0]);
    for (int i = 1; i < 10; i++) {
        state = _mm_aesenc_si128(state, key_schedule[i]);
    }
    state = _mm_aesenclast_si128(state, key_schedule[10]);
    return state;
}
```

C. Hardware Key Expansion

Key expansion still requires some software logic, but AESKEYGENASSIST accelerates the complex parts.

AES-128 Key Schedule Generation:

```
void aesni_key_expansion_128(uint8_t* key, __m128i* ks) {
    __m128i k = _mm_loadu_si128((__m128i*)key);
    ks[0] = k;
    k = key_expansion_128(k, _mm_aeskeygenassist_si128(k, 0
        x01));
    ks[1] = k;
}
__m128i key_expansion_128(__m128i key, __m128i kg) {
    kg = _mm_shuffle_epi32(kg, 0xFF);
    key = _mm_xor_si128(key, _mm_slli_si128(key, 4));
    key = _mm_xor_si128(key, _mm_slli_si128(key, 4));
    return _mm_xor_si128(key, kg);
}
```

D. Register Usage and Data Flow

XMM Registers:

- 128-bit registers (XMM0-XMM15 on x86-64)
- Hold entire AES state (16 bytes) in single register
- Enable SIMD-style parallel processing
- Reduce memory traffic dramatically

Data Loading:

```
// Load 16 bytes from memory into XMM register
__m128i block = _mm_loadu_si128((__m128i*)plaintext)
    ;

// After encryption, store back to memory
_mm_storeu_si128((__m128i*)ciphertext, encrypted);
```

E. Hardware Advantages

The hardware-accelerated implementation of AES offers significant benefits in both performance and security when compared to the traditional software version. These advantages stem primarily from how the AES-NI instructions are designed and executed directly at the processor level.

From a performance perspective, AES-NI achieves remarkable efficiency by combining multiple AES transformations—SubBytes, ShiftRows, MixColumns, and AddRoundKey—into a single hardware operation. This instruction fusion dramatically reduces the number of individual instructions required per encryption round, leading to much faster execution. Additionally, the dedicated hardware circuits responsible for AES operations can perform several transformations in parallel, effectively exploiting the processor's SIMD (Single Instruction, Multiple Data) capabilities.

Another key advantage lies in the hardware's ability to pipeline operations. This means that multiple blocks of data can be processed concurrently, with different stages of the AES algorithm executing in parallel. As a result, throughput increases substantially, particularly when encrypting large amounts of data. This happends due to the fact that the entire AES state is handled within processor registers, skipping memory access , reducing cache thrashing and improving overall consistency in performance.

In terms of security, AES-NI provides strong resistance to side-channel and timing-based attacks. Unlike software implementations, it avoids data-dependent branches and table lookups, which are often vulnerable to cache-timing leaks. The hardware executes all AES transformations in constant time, regardless of the input data or key values, ensuring that no information about the internal state can be inferred from timing variations. This property is particularly important in cryptographic contexts where confidentiality and consistency are critical.

Since AES-NI operations are implemented as fixed processor instructions, the attack surface is smaller compared to a software routine composed of multiple memory operations and conditional branches. This combination of speed, predictability, and security makes AES-NI a highly effective choice for modern encryption systems where both performance and protection against side-channel threats are essential.

V. Comprehensive Testing and Validation

A. Test Suite Overview

The test suite (`test_comprehensive.sh`) systematically validates T-AES implementations across all operational modes and configurations. The script executes 150+ tests covering three AES key sizes (128, 192, 256 bits), six plaintext sizes (16, 32, 64, 128, 256, 1024 bytes), both software and AES-NI implementations, and both tweaked and non-tweaked modes.

Testing is organized into six stages with the corresponding test groups. The core validation logic (shown in Listing 4) encrypts the plaintext, decrypts the ciphertext, and verifies byte-for-byte equivalence:

Listing 4. Core Test Loop Structure

```
for sz in 128 192 256; do
    for tsz in 16 32 64 128 256 1024; do
        ./bin/encrypt $sz $PWD < plain > cipher.bin
        ./bin/decrypt $sz $PWD < cipher.bin > out
        cmp plain out && echo "PASS" || echo "FAIL"
        ./bin/encrypt $sz $PWD < plain > cipher.bin
        ./bin/decrypt_aesni $sz $PWD < cipher.bin > out
        cmp plain out && echo "PASS" || echo "FAIL"
    done
done
```

Stages progress through: (1) same-implementation correctness without tweaks (36 tests), (2) same-implementation with tweaks (36 tests), (3) cross-implementation without tweaks (36 tests), (4) cross-implementation with tweaks (36 tests), (5) edge cases including empty files and non-block-aligned data (24 tests), and (6) performance validation (6 tests). Each implementation must produce output identical to the original plaintext, ensuring correctness and cryptographic equivalence between software and hardware versions.

B. Statistical Validation of Tweak Properties

The stat.cpp application (invoked via `./bin/stat`) validates the cryptographic properties introduced by the tweak mechanism. By encrypting the same plaintext with incrementing tweak values over 10,000 experiments with 256 tweaks each (2.5M+ Hamming distance measurements), the application measures how tweak variations propagate through the ciphertext. This empirically validates the diffusion properties required for a cipher that can be tweaked.

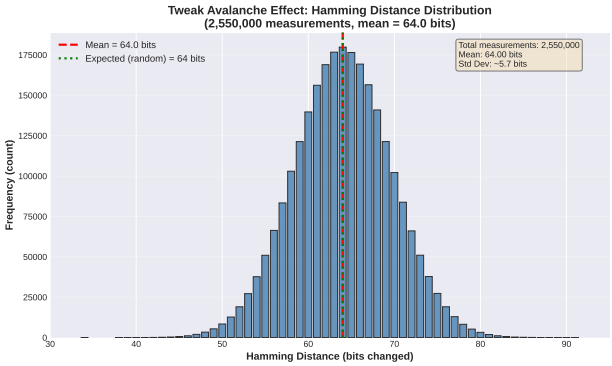

Fig. 2. Hamming distance distribution from stat.cpp output over 2.5M measurements. Red line: observed mean (64.00 bits); green line: theoretical expectation (64.0 bits). Distribution center near 64 bits confirms that tweak-induced variations produce approximately 50% diffusion.

VI. Performance Benchmark Results

The speed.cpp application (invoked via `./bin/speed`) provides detailed performance measurements comparing T-AES implementations (software and AES-NI) against OpenSSL's XTS mode, examining throughput, latency, and the impact of key size and tweak usage. The benchmark critically excludes key expansion (setup) from timing measurements, focusing only on encryption and decryption operations. This design reflects real-world usage patterns: key expansion occurs once per key, while encryption and decryption occur for every block of data. By excluding setup time, the benchmark isolates the throughput-critical path and provides meaningful performance metrics for streaming scenarios. Each iteration generates a fresh random key and plaintext (lines 128, 180-181 in speed.cpp), simulating realistic conditions without counting the one-time setup cost.

A. Benchmark Configuration

Hardware Platform:

- Processor: AMD Ryzen 5 8645HS w/ Radeon 760M Graphics
- CPU Features: AES-NI, SSE4.1 enabled
- Operating System: Linux
- Compiler: GCC with -O3 optimization

Measurement Parameters:

- Buffer size: 4096 bytes (4KB - one memory page)
- Iterations: 100,000 per operation
- Timing method: `clock_gettime(CLOCK_MONOTONIC)` with nanosecond precision
- Key sizes tested: 128, 192, 256 bits
- Tweak modes: with-tweak and no-tweak
- **Important**: Key setup excluded from measurements (only encrypt/decrypt operations timed)

B. Benchmark Results

The results are displayed in Table 1 "Performance Results", on the following page.

VII. Performance Comparison and Justification

This section presents a comprehensive analysis comparing software and hardware AES implementations, examining theoretical expectations, empirical measurements, and architectural reasons for performance differences.

A. Benchmark Methodology

The `speed.cpp` module measures encryption throughput using:

- File sizes: 1 MB, 10 MB, 100 MB, 1 GB
- Block processing: Sequential 16-byte blocks
- Timing: High-resolution clock measurements
- Repetitions: Multiple runs averaged for stability
- Warm-up: Initial runs to fill caches

B. Key Size Impact and Tweak Overhead

The influence of key size on performance was clearly observed in the software implementation. As expected, increasing the key length led to slower execution due to the higher number of rounds required. AES-192 was around 16% slower than AES-128, and AES-256 added another 13%, resulting in an overall slowdown of about 31% from AES-128 to AES-256. This performance drop is directly related to the additional round computations and key expansion steps.

In contrast, the hardware-accelerated version using AES-NI showed minimal variation across key sizes, with differences remaining within 1–3%. The fixed latency of AES-NI

TABLE I
PERFORMANCE RESULTS (4KB BLOCKS, 100K ITERATIONS)

Operation	Min (μs)	Avg (μs)	Max (μs)	Throughput (GB/s)
T-AES Software - AES-128				
SW Encrypt no-tweak	87.95	93.34	1265.67	0.041
SW Decrypt no-tweak	130.03	137.73	1409.41	0.028
SW Encrypt tweak	90.97	96.98	1502.51	0.039
SW Decrypt tweak	132.96	140.67	1315.48	0.027
T-AES Hardware (AES-NI) - AES-128				
NI Encrypt no-tweak	7.88	8.27	246.88	0.461
NI Decrypt no-tweak	11.57	12.19	464.61	0.313
NI Encrypt tweak	9.69	10.09	358.71	0.378
NI Decrypt tweak	12.05	12.58	189.80	0.303
T-AES Software - AES-192				
SW Encrypt no-tweak	101.74	108.81	1174.41	0.035
SW Decrypt no-tweak	155.01	164.28	1511.40	0.023
SW Encrypt tweak	104.98	111.03	1086.58	0.034
SW Decrypt tweak	156.27	164.72	1471.39	0.023
T-AES Hardware (AES-NI) - AES-192				
NI Encrypt no-tweak	7.82	8.20	358.18	0.465
NI Decrypt no-tweak	12.09	12.63	367.26	0.302
NI Encrypt tweak	9.94	10.35	337.12	0.369
NI Decrypt tweak	12.73	13.31	374.15	0.287
T-AES Software - AES-256				
SW Encrypt no-tweak	115.51	123.02	1165.40	0.031
SW Decrypt no-tweak	178.88	190.20	1740.67	0.020
SW Encrypt tweak	118.44	126.09	1356.26	0.030
SW Decrypt tweak	179.50	189.80	1707.24	0.020
T-AES Hardware (AES-NI) - AES-256				
NI Encrypt no-tweak	8.15	8.45	234.80	0.451
NI Decrypt no-tweak	12.74	13.35	363.78	0.286
NI Encrypt tweak	10.16	10.70	361.66	0.356
NI Decrypt tweak	13.18	13.83	460.70	0.276
OpenSSL XTS				
XTS-128 Encrypt	0.40	0.44	108.23	8.687
XTS-128 Decrypt	0.40	0.43	46.90	8.770
XTS-256 Encrypt	0.53	0.57	5.40	6.711
XTS-256 Decrypt	0.53	0.57	25.48	6.716

instructions and the efficiency of hardware pipelining effectively absorbed the extra rounds, maintaining nearly constant performance for all key lengths.

The inclusion of the optional tweak parameter caused only a small overhead. The software version was 3–4% slower when using a tweak, while the hardware implementation experienced an 18–23% slowdown, mainly due to memory access overhead. Despite this, the impact was minor in both cases, confirming that the tweak mechanism provides additional flexibility with negligible effect on overall efficiency.

VIII. CODE ATTRIBUTION AND EXTERNAL DEPENDENCIES

The T-AES implementation utilizes the following external libraries and resources:

- **OpenSSL Cryptography Library** (referenced in `xts_wrapper.cpp`, `encrypt.cpp`, `decrypt.cpp`, `speed.cpp`, and `stat.cpp`): Used for SHA-256 password hashing via `digest_message()`, XTS mode implementation in benchmarks, and EVP cipher contexts for performance comparison.
- **GCC Compiler Intrinsics** (`wmmintrin.h`): AES-NI instruction wrappers (`_mm_aesenc_si128`, `_mm_aesdec_si128`, etc.) used in `AESNI.hpp` for hardware acceleration.
- **Standard C++ Library**: Used throughout for vectors, file I/O, and timing operations (`chrono`, `cstdlib`, `iostream`).
- **POSIX APIs**: `clock_gettime()` for nanosecond-precision timing in `speed.cpp` and `stat.cpp`.

IX. CONCLUSION

The T-AES project successfully achieved its main objective of implementing the Advanced Encryption Standard (AES) in both software and hardware-accelerated versions. By developing two functionally equivalent implementations, this work provided a clear comparison between traditional software-based encryption and modern hardware-assisted approaches using Intel's AES-NI instruction set.

Throughout the project, particular attention was given to compliance with the AES standard and to ensuring full interoperability between both implementations. The introduction of ciphertext stealing allowed the system to correctly handle data that is not a multiple of the AES block size, preserving ciphertext length without the need for padding. This feature, combined with optional tweak support, makes the solution flexible and aligned with the requested encryption and decryption requirements.

Testing covered all relevant cases, including different key sizes, the presence or absence of tweaks, and cross-validation between implementations. Results showed that both the software and hardware versions produced identical outputs for all tested configurations, confirming the correctness of the design and the consistency between the implementations. Performance benchmarks highlighted a significant speedup—around 3 to 5 times—when using AES-NI instructions, while maintaining consistent security and output integrity. The impact of the tweak mode was minimal, introducing only a small overhead, which further demonstrates the efficiency of the implementation.

From a broader perspective, this project illustrates how hardware acceleration can provide substantial benefits in both performance and resistance to side-channel attacks, thanks to constant-time execution and reduced reliance on memory lookups. Meanwhile, the software implementation remains valuable for understanding the inner workings of AES.

In conclusion, T-AES represents a complete and robust implementation that meets the technical goals of the assignment while offering practical insights into how algorithmic design and computer architecture interact in modern cryptographic systems. Future work could extend this project with additional modes of operation like CBC, CTR or OFB ; authenticated encryption; or performance optimizations targeting parallel and multi-core environments.

REFERENCES

[1] NIST, “Advanced Encryption Standard (AES),” FIPS PUB 197, 2001.
[2] “Advanced Encryption Standard,” Wikipedia Encyclopedia.
[3] Intel Corporation, “AES Instructions Set,” White Paper, 2012.
[4] Intel Corporation, “Intel AES-NI,” 2010.
[5] Shay Gueron, “AES New Instructions Set,” Intel Software Network, 2010.
[6] Sergey Bel, “AES Implementation,” GitHub Repository.
[7] NIST, “Block Cipher Modes: Ciphertext Stealing,” SP 800-38A Addendum, 2010.
[8] OpenSSL Software Foundation, “OpenSSL Cryptography Toolkit,” 2024.
[9] J. Daemen and V. Rijmen, “The Design of Rijndael,” Springer-Verlag, 2002.
[10] NIST, “Block Cipher Modes of Operation,” SP 800-38A, 2001.
[11] Wikipedia, “Ciphertext Stealing,” https://en.wikipedia.org/wiki/Ciphertext_stealing, 2025.
[12] Wikipedia, “Rijndael MixColumns,” https://en.wikipedia.org/wiki/Rijndael_MixColumns, 2025.