Seguridad Informática LIS4052-1

Homework 3: Key-Based Enhancement of DES (KE-DES)

Autumn

1. Abstract

This report presents the implementation, analysis and discussion of the Key - Based Enhancement of the Data Encryption standard (KE – DES), an adapted version of the traditional DES algorithm that introduces modifications in the key generation and expansion functions.

Although KE-DES is not designed for modern cryptographic security, its structure provides a valuable learning framework for understanding symmetric encryption mechanisms. In this work, the KE-DES algorithm is implemented to encrypt and decrypt 64-bit data blocks using a C++ environment, focusing on key scheduling, round functions, and Cipher Block Chaining (CBC) mode. The results are examined through performance observations and a comparative analysis with the original DES algorithm. Furthermore, the report includes an exploration of classical cryptanalytic approaches such as brute-force, differential, and linear attacks. To contextualize the evolution of cryptographic systems, The conclusions reflect on the strengths and limitations of KE-DES and its educational relevance in the field of information security.

2. Introduction

In the modern digital era, the protection of information during storage and transmission has become one of the most critical challenges in computer science and communication systems. Data integrity, confidentiality, and authenticity are essential components of secure communication, and these goals are primarily achieved through the science of cryptography. Cryptography transforms readable information (*plaintext*) into an unreadable form (*ciphertext*) to prevent unauthorized access. Symmetric-key cryptography, in particular, plays a foundational role in practical applications where speed and efficiency are paramount.

Among the most influential symmetric-key algorithms is the **Data Encryption Standard (DES)**, first adopted as a U.S. Federal Information Processing Standard (FIPS 46) in 1977. DES became the cornerstone of classical cryptography education and a benchmark for block cipher design. It operates on 64-bit blocks and uses a 56-bit effective key, performing a series of permutation and substitution operations distributed across sixteen rounds. These transformations, defined through key-dependent S-boxes and Feistel structures, exemplify

the principles of *confusion* and *diffusion* core properties that ensure ciphertext unpredictability and resistance to analysis.

Despite its historical importance, DES eventually became vulnerable due to advances in computational power. A brute-force attack capable of testing all 2^{56} possible keys can now be performed within hours using distributed computing resources. As a result, DES has been replaced in practical systems by more robust algorithms such as Triple DES (3DES) and the Advanced Encryption Standard (AES). Nevertheless, DES remains a vital educational tool that demonstrates the balance between mathematical theory and practical encryption architecture.

The **Key-Based Enhancement of DES (KE-DES)**, proposed by Omar Reyad and colleagues in 2021, revisits this classical cipher with the intention of reinforcing its academic value. Rather than proposing a cryptographically secure replacement, KE-DES introduces *controlled modifications* to the DES key scheduling and expansion functions, aiming to emphasize how subtle variations in key manipulation can influence the cipher's structure and diffusion properties. The enhancement occurs primarily in two areas:

1. **Odd/Even Bit Transformation:** applied to the key bits after the initial permutation (PC-1), introducing an alternating binary pattern that increases the diversity of subkey generation.
2. **Key-Distribution (K-D) Function:** which modifies the standard expansion function used in the Feistel rounds by integrating key bits directly into the expansion process.

These modifications do not alter the overall framework of DES but provide a meaningful variation for academic experimentation, making KE-DES an excellent tool for demonstrating key scheduling, round-based processing, and encryption-decryption symmetry.

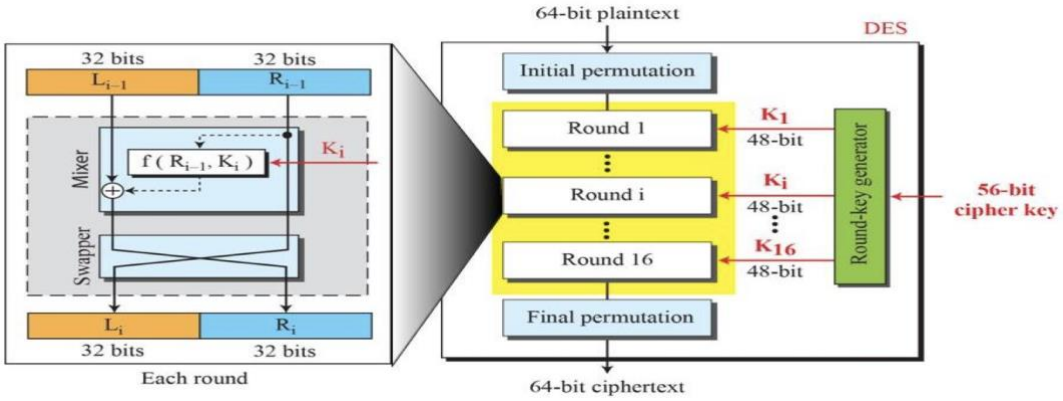


Fig. 1 "Architecture of the DES Algorithm". Information Security: Third Class-Cybersecurity [Diapositivas].

By modifying only the key handling mechanism, KE-DES maintains structural comparability to DES, allowing a direct study of how key variations affect round output and diffusion strength. The resulting cipher maintains a 64-bit block size and 16-round structure, but its subkeys differ substantially due to the new transformation scheme.

From an educational perspective, KE-DES provides students and researchers with an opportunity to explore cryptographic principles such as substitution-permutation networks, Feistel symmetry, and key-dependent permutations. Moreover, it provides a controlled environment for studying cryptanalytic concepts such as **brute-force attacks**, **differential cryptanalysis**, and **linear cryptanalysis**, which were originally developed to analyze DES itself.

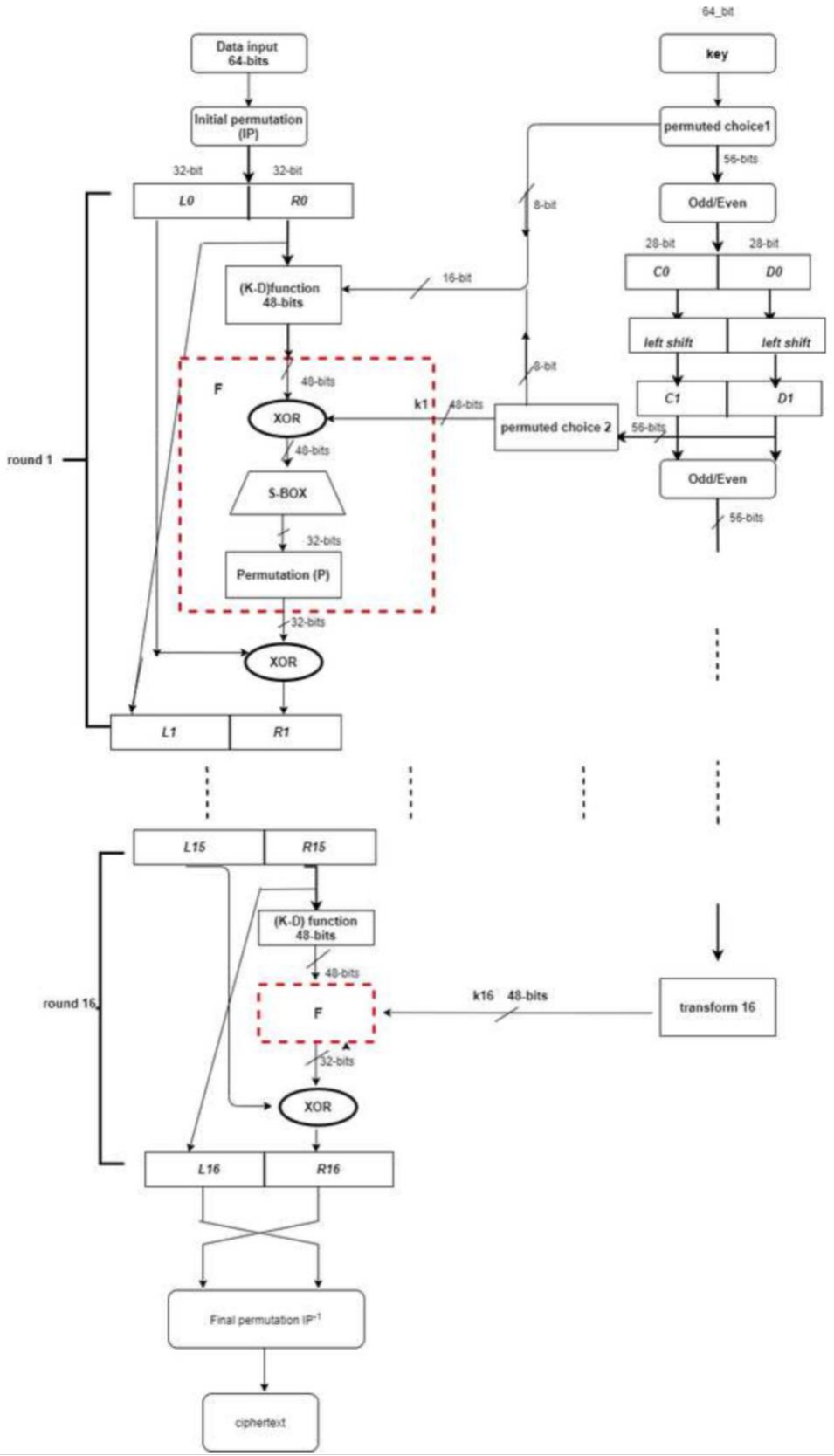


Fig. 2 "Architecture of Key based Enchancement of DES" from Reyad et al., 2021.

This project, developed as part of the Information Security course, aims to implement the KE-DES algorithm in the C++ programming language, validate its encryption and decryption functionality, and analyze its behavior under controlled test cases. The implementation focuses on text-based data, following the process described in the original IEEE paper by Reyad et al. (2021).

The report further includes a comparative evaluation between DES and KE-DES in terms of key generation time, operational structure, and theoretical resistance to cryptanalytic attacks. Additionally, a broader historical exploration is conducted by examining the **Enigma machine**, a World War II-era mechanical cipher that shares conceptual similarities with modern symmetric encryption. The Enigma machine's mechanical substitution and permutation mechanisms paved the way for the digital ciphers of today, including DES and its derivatives.

This study seeks to achieve three goals:

1. To implement and understand the detailed operations of KE-DES encryption and decryption.
2. To compare the modified algorithm's performance and design features with those of standard DES.
3. To connect modern cryptographic ideas with their historical roots through an analytical discussion of the Enigma machine and its relevance to symmetric-key encryption principles.

Through this exploration, KE-DES serves not only as a practical programming exercise but also as a bridge between historical cryptography and modern information security illustrating how fundamental design choices continue to shape the evolution of secure communication.

3. Methodology

3.1 Implementation Framework

The implementation of the Key – Based Enhancement of DES (KE – DES) was carried out in C++, a language chosen for its efficiency in low – level bit manipulation and deterministic control of memory structures. The goal was to replicate and replicate the architecture proposed in the 2021 "Key-Based Enhancement of Data Encryption Standard For Text Security" paper by Reyad et al., while maintaining modularity and readability in the code. Two core programs were developed:

1. **KE_DES.cpp** – responsible for key generation and encryption.
2. **KE_DES_decrypt.cpp** – responsible for decryption and validation.

Both programs were executed on a 64-bit operating system using the Visual Studio Code C++ compiler. The implementation follows the **Feistel cipher structure**, consisting of repeated rounds of permutation, substitution, and XOR operations between the data and key material.

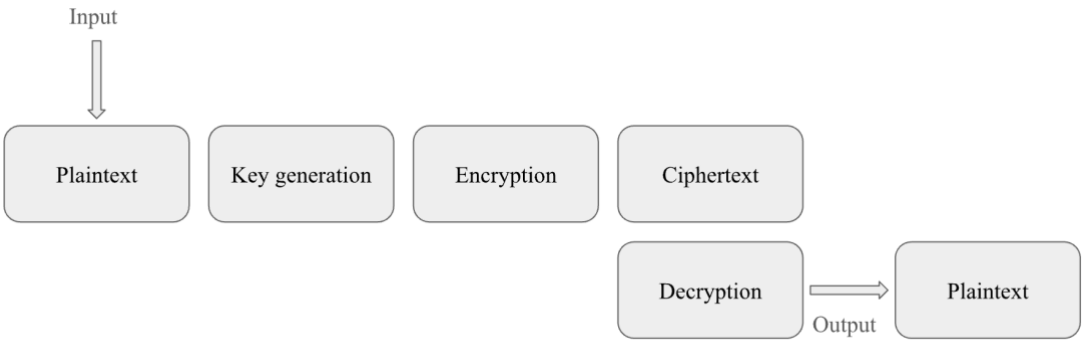


Fig. 3 "Overall Workflow of the KE - DES Implementation."

To ensure consistency and reproducibility, the following constants and tables were defined according to the original DES and KE-DES specifications:

- Permutation Choice-1 (PC-1) and Permutation Choice-2 (PC-2) tables.
- Initial and Inverse Permutation (IP and IP^{-1}) tables.
- Expansion (E) and Permutation (P) tables.
- The Shift Schedule for key rotation across 16 rounds.

The key difference between KE-DES and DES lies not in these tables, but in the **Odd/Even Transformation** and the **Key-Distribution (K-D) function**, which redefine how key bits are manipulated and merged with data blocks during the encryption process.

3.2 Key Generation Process

The key generation is one of the defining steps of KE – DES. It determines how the 16 round keys (k1 to k16) are derived from the main secret key.
The process begins with the 64 – bit hexadecimal key value:

```
//Key in hexadecimal format
unsigned long long Key = 0x133457799BBCDFF1ULL;
```

Fig 4. 4 Input key in Hex format, 'ULL' added for the compilator to take it as unsigned long long format.

This key is first converted into its binary representation and passed through Permutation choice-1 (PC-1) to reduce it to 56 bits. Unlike DES, where these 56 bits are directly split into halves, KE-DES introduces an Odd/Even Bit Transformation, which assigns alternating values to the bits depending on their position:

```
// PC-1 table (56 positions)
static const int PC1[56] = {
    57,49,41,33,25,17,9,
    1,58,50,42,34,26,18,
    10,2,59,51,43,35,27,
    19,11,3,60,52,44,36,
    63,55,47,39,31,23,15,
    7,62,54,46,38,30,22,
    14,6,61,53,45,37,29,
    21,13,5,28,20,12,4
};
```

```
// PC-2 table (48 positions)
static const int PC2[48] = {
    14,17,11,24,1,5,
    3,28,15,6,21,10,
    23,19,12,4,26,8,
    16,7,27,20,13,2,
    41,52,31,37,47,55,
    30,40,51,45,33,48,
    44,49,39,56,34,53,
    46,42,50,36,29,32
};
```

Fig 5. 5 PC-1 and PC-2 table, permuted choice matrix used in the key generation

- Even bit positions are set to 1,
- Odd bit positions are set to 0 for the left half (C),
- The inverse pattern is applied for the right half (D).

This deterministic manipulation produces a predictable but structurally diverse key schedule that influences subsequent subkey derivation.

```
static void odd_even_transform(vector<int>& b) {
    // produce C0 = 0,1,0,1,... for positions 1..28
    // and    D0 = 1,0,1,0,... for positions 29..56
    int n = (int)b.size()-1;
    for (int j = 1; j <= n; ++j) {
        if (j <= 28) {
            // first half: odd positions = 0, even = 1 -> 0,1,0,1,...
            b[j] = (j % 2 == 0) ? 1 : 0;
        } else {
            // second half: odd positions = 1, even = 0 -> 1,0,1,0,...
            b[j] = (j % 2 == 1) ? 1 : 0;
        }
    }
}
```

Fig 6. 6 Iterates over the 56-bit vector and sets the first half to the alternating 010101... and the second half 101010... odd positions= 0, even = 1 and second half: odd = 1, even = 0.

The next step divides the 56-bit key into two halves of 28 bits each (C0 and D0). Each half undergoes a sequence of left circular shifts as defined in the shift schedule (1 or 2 bits depending on the round number). After each shift, the halves are concatenated and permuted using Permutation Choice-2 (PC-2) to generate the 48-bit subkeys. This process repeats for all 16 rounds, producing k1, k2, …, k16.

The key generation time is measured using the **C++ <chrono>** library, capturing high-resolution timestamps before and after the key schedule. This timing data is later used to compare the performance of KE-DES against standard DES.

```
// Left rotation schedule for 16 rounds (standard DES)
static const int SHIFTS[16] = {1,1,2,2,2,2,2,2,1,2,2,2,2,2,2,1};
```

Fig 7. 7 Used by the DES key schedule, says how many positions to rotate the 28-bit C and D halves before forming the respective round's subkey.

3.3 Encryption Algorithm

Once the subkeys are generated, the encryption phase operates on 64-bit plaintext blocks. The algorithm follows the Feistel network principle, which guarantees that encryption and decryption processes are structurally identical when the key order is reversed. Each plaintext message is divided into 8-byte (64-bit) blocks. If the last block is shorter than 8 bytes, PKCS#7 padding is applied to ensure uniform block length.

The encryption procedure involves three major stages:

1. Initial Permutation (IP) – rearranges the 64 input bits according to a predefined table, producing two 32-bit halves: L0(left) and R0 (right).
2. Sixteen Rounds of the Feistel Process – during each round i, the right half Ri is expanded to 48 bits using the Expansion (E) table and XORed with the round key Ki.

- The result passes through an S-Box substitution, compressing it back to 32 bits.
- A P permutation is then applied, and the output is XORed with the left half Li.
- Finally, the halves are swapped.

3. Inverse Permutation (IP^{-1}) – after completing all 16 rounds, the final concatenated output undergoes IP^{-1}, producing the 64-bit ciphertext.

```
// Initial Permutation (IP)
static const int IP[64] = {
    58,50,42,34,26,18,10,2,
    60,52,44,36,28,20,12,4,
    62,54,46,38,30,22,14,6,
    64,56,48,40,32,24,16,8,
    57,49,41,33,25,17,9,1,
    59,51,43,35,27,19,11,3,
    61,53,45,37,29,21,13,5,
    63,55,47,39,31,23,15,7
};
```

```
// Inverse IP
static const int IP_INV[64] = {
    40,8,48,16,56,24,64,32,
    39,7,47,15,55,23,63,31,
    38,6,46,14,54,22,62,30,
    37,5,45,13,53,21,61,29,
    36,4,44,12,52,20,60,28,
    35,3,43,11,51,19,59,27,
    34,2,42,10,50,18,58,26,
    33,1,41,9,49,17,57,25
};
```

```
// Expansion table E (32 -> 48)
static const int E_TABLE[48] = {
    32,1,2,3,4,5,
    4,5,6,7,8,9,
    8,9,10,11,12,13,
    12,13,14,15,16,17,
    16,17,18,19,20,21,
    20,21,22,23,24,25,
    24,25,26,27,28,29,
    28,29,30,31,32,1
};
```

```
// P permutation (32)
static const int P_TABLE[32] = {
    16,7,20,21,29,12,28,17,
    1,15,23,26,5,18,31,10,
    2,8,24,14,32,27,3,9,
    19,13,30,6,22,11,4,25
};
```

Fig 8. 8 Initial Permutation matrix, Inverse Initial Permutation and Expansion table

```
static vector<int> apply_permutation(const vector<int>& in, const int* table, int tlen) {
    vector<int> out(tlen+1);
    for (int i = 0; i < tlen; ++i) {
        out[i+1] = in[ table[i] ];
    }
    return out;
}
```

Fig 9. 9 This function applies permutation operation using any of the above tables, this by converting inputs in vectors for better performance.

This function is used in the F function, the des_block_operation, and in all round key operations.

```
static void rot_left(vector<int>& v, int shifts) {
    // v is 1-based indexed
    int n = (int)v.size()-1;
    if (n == 0) return;
    shifts %= n;
    if (shifts == 0) return;
    vector<int> tmp(n+1);
    for (int i = 1; i <= n; ++i) {
        int src = ((i + shifts - 1) % n) + 1;
        tmp[i] = v[src];
    }
    v = tmp;
}
```

Fig. 10 Rotates a vector (C and D) depending on the shifts array fig.6 .

```cpp
// Simplified S-box substitution: maps each 6-bit value to 4-bit deterministically.
// This keeps the implementation concise; you can replace this with standard DES S-boxes if desired.
static void sbox_substitution(const vector<int>& in48, vector<int>& out32) {
    // in48: 1-based 48 bits; out32 will be 1-based 32 bits
    for (int i = 0; i < 8; ++i) {
        int base = i*6;
        int val = 0;
        for (int b = 0; b < 6; ++b) val = (val << 1) | in48[base + b + 1];
        // deterministic mapping: mix and reduce to 4 bits
        int nibble = ((val * (i+1)) ^ (val >> 2)) & 0xF;
        // put nibble into out32
        for (int b = 0; b < 4; ++b) {
            out32[i*4 + b + 1] = ( (nibble >> (3 - b)) & 1 );
        }
    }
}
```

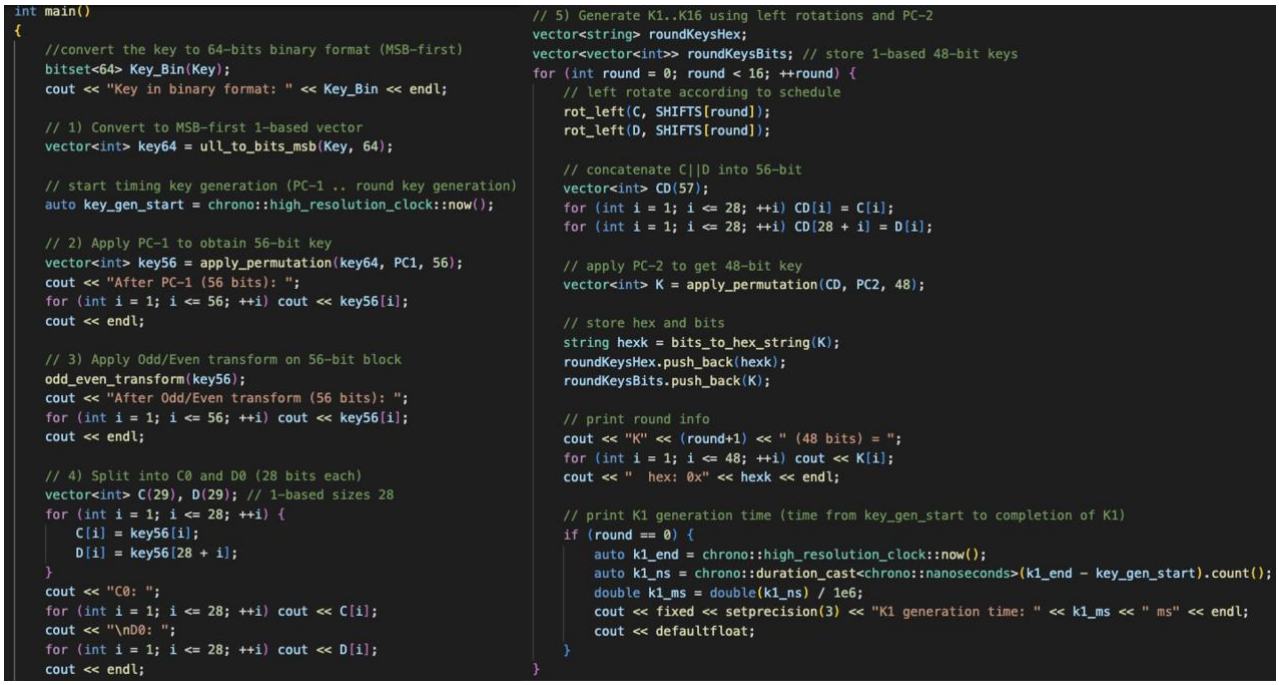
Fig. 11 Simplified S-box substitution, converts the 48-bit input into 32 bits by processing eight 6-bit groups into eight 4-bit.

```cpp
// Feistel function f: takes 32-bit R (1-based) and 48-bit subkey (1-based), returns 32-bit vector (1-based)
static vector<int> feistel_f(const vector<int>& R32, const vector<int>& K48) {
    // expand R from 32->48
    vector<int> Rexp = apply_permutation(R32, E_TABLE, 48);
    // XOR with key
    vector<int> tmp(48+1);
    for (int i = 1; i <= 48; ++i) tmp[i] = Rexp[i] ^ K48[i];
    // S-box substitution (simplified)
    vector<int> sbout(32+1);
    sbox_substitution(tmp, sbout);
    // P permutation
    vector<int> pout = apply_permutation(sbout, P_TABLE, 32);
    return pout;
}
```

Fig. 12 the Feistel round function expands a 32-bit half-block to 48 bits, mixes it with the 48-bit subkey, substitutes/compresses back to 32 bits, then permutes the result.

```cpp
// DES-like encrypt single 64-bit block (1-based vector) with provided 16 round keys (each 1-based 48 bits)
static vector<int> des_encrypt_block(const vector<int>& block64, const vector<vector<int>>& roundKeys) {
    // Apply IP
    vector<int> ip = apply_permutation(block64, IP, 64);
    // split L and R
    vector<int> L(32+1), R(32+1);
    for (int i = 1; i <= 32; ++i) { L[i] = ip[i]; R[i] = ip[32 + i]; }

    // 16 rounds
    for (int r = 0; r < 16; ++r) {
        vector<int> f = feistel_f(R, roundKeys[r]);
        vector<int> newR(32+1);
        for (int i = 1; i <= 32; ++i) newR[i] = L[i] ^ f[i];
        L = R;
        R = newR;
    }

    // preoutput is R||L (swap)
    vector<int> preout(64+1);
    for (int i = 1; i <= 32; ++i) {
        preout[i] = R[i];
        preout[32 + i] = L[i];
    }
    // apply IP_INV
    vector<int> out = apply_permutation(preout, IP_INV, 64);
    return out;
}
```

Fig. 13 des_encrypt_block function, perform a full 16-round DES-style encryption on a 64-bit block using the provided 16 subkeys



```cpp
int main()
{
    //convert the key to 64-bits binary format (MSB-first)
    bitset<64> Key_Bin(Key);
    cout << "Key in binary format: " << Key_Bin << endl;

    // 1) Convert to MSB-first 1-based vector
    vector<int> key64 = ull_to_bits_msb(Key, 64);

    // start timing key generation (PC-1 .. round key generation)
    auto key_gen_start = chrono::high_resolution_clock::now();

    // 2) Apply PC-1 to obtain 56-bit key
    vector<int> key56 = apply_permutation(key64, PC1, 56);
    cout << "After PC-1 (56 bits): ";
    for (int i = 1; i <= 56; ++i) cout << key56[i];
    cout << endl;

    // 3) Apply Odd/Even transform on 56-bit block
    odd_even_transform(key56);
    cout << "After Odd/Even transform (56 bits): ";
    for (int i = 1; i <= 56; ++i) cout << key56[i];
    cout << endl;

    // 4) Split into C0 and D0 (28 bits each)
    vector<int> C(29), D(29); // 1-based sizes 28
    for (int i = 1; i <= 28; ++i) {
        C[i] = key56[i];
        D[i] = key56[28 + i];
    }
    cout << "C0: ";
    for (int i = 1; i <= 28; ++i) cout << C[i];
    cout << "\nD0: ";
    for (int i = 1; i <= 28; ++i) cout << D[i];
    cout << endl;
```

```cpp
    // 5) Generate K1..K16 using left rotations and PC-2
    vector<string> roundKeysHex;
    vector<vector<int>> roundKeysBits; // store 1-based 48-bit keys
    for (int round = 0; round < 16; ++round) {
        // left rotate according to schedule
        rot_left(C, SHIFTS[round]);
        rot_left(D, SHIFTS[round]);

        // concatenate C||D into 56-bit
        vector<int> CD(57);
        for (int i = 1; i <= 28; ++i) CD[i] = C[i];
        for (int i = 1; i <= 28; ++i) CD[28 + i] = D[i];

        // apply PC-2 to get 48-bit key
        vector<int> K = apply_permutation(CD, PC2, 48);

        // store hex and bits
        string hexk = bits_to_hex_string(K);
        roundKeysHex.push_back(hexk);
        roundKeysBits.push_back(K);

        // print round info
        cout << "K" << (round+1) << " (48 bits) = ";
        for (int i = 1; i <= 48; ++i) cout << K[i];
        cout << "  hex: 0x" << hexk << endl;

        // print K1 generation time (time from key_gen_start to completion of K1)
        if (round == 0) {
            auto k1_end = chrono::high_resolution_clock::now();
            auto k1_ns = chrono::duration_cast<chrono::nanoseconds>(k1_end - key_gen_start).count();
            double k1_ms = double(k1_ns) / 1e6;
            cout << fixed << setprecision(3) << "K1 generation time: " << k1_ms << " ms" << endl;
            cout << defaultfloat;
        }
    }
```

Fig. 14 Main function.

Converts the 64-bit hex Key into a 1-based bit vector and starts a timer. Applies PC-1 permutation to get 56 bits, then applies the odd/even transform. Splits that 56-bit result into C0 and D0 (28 bits each). For each of 16 rounds: left-rotate C and D by the SHIFTS schedule, concatenate C||D, apply PC-2 to produce the 48-bit round key Ki, store and print Ki. Immediately after producing K1 it captures the time and prints "K1 generation time" (elapsed from the timer start). After all rounds it computes and prints total key generation time.

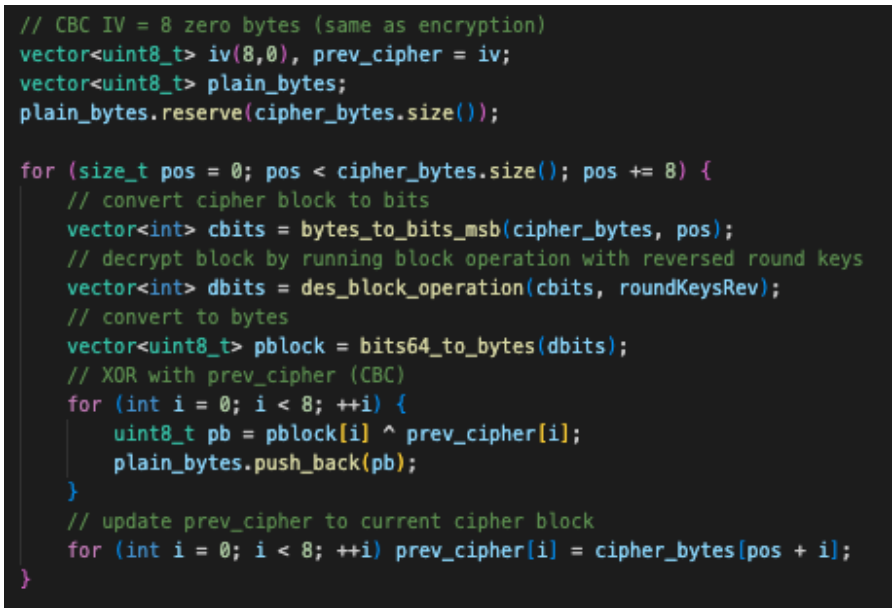
```
// --- New: read plaintext.txt, pad, encrypt in CBC mode, write ciphertext.txt (hex) ---
const string infile_name = "plaintext.txt";
const string outfile_name = "ciphertext.txt";

ifstream infile(infile_name, ios::binary);
if (!infile) {
    cerr << "Cannot open " << infile_name << " for reading.\n";
    return 1;
}
vector<uint8_t> plain;
infile.seekg(0, ios::end);
size_t fsize = infile.tellg();
infile.seekg(0, ios::beg);
plain.resize(fsize);
infile.read(reinterpret_cast<char*>(plain.data()), (streamsize)fsize);
infile.close();

// PKCS#7 padding to 8 bytes
size_t pad_len = 8 - (plain.size() % 8);
if (pad_len == 0) pad_len = 8;
for (size_t i = 0; i < pad_len; ++i) plain.push_back((uint8_t)pad_len);

// CBC IV = 8 zero bytes
vector<uint8_t> iv(8, 0);
vector<uint8_t> prev_cipher = iv;
vector<uint8_t> cipher_bytes;

for (size_t pos = 0; pos < plain.size(); pos += 8) {
    // XOR plaintext block with prev_cipher
    vector<uint8_t> block(8);
    for (int i = 0; i < 8; ++i) block[i] = plain[pos + i] ^ prev_cipher[i];

    // convert to bits
    vector<int> block_bits = bytes_to_bits_msb(block, 0);

    // encrypt block
    vector<int> cipher_bits = des_encrypt_block(block_bits, roundKeysBits);

    // convert back to bytes
    vector<uint8_t> cbytes = bits64_to_bytes(cipher_bits);
    // append to result
    for (auto b : cbytes) cipher_bytes.push_back(b);
    // update prev_cipher
    prev_cipher = cbytes;
}
```

Fig. 15 Reads the entire plaintext file, using the KE_DES in Cipher block chaining mode, then writes the result in hex to another file.

For enhanced operational security and demonstration of practical encryption mode, **Cipher Block Chaining (CBC)** was implemented. An initialization vector (IV) of eight zero bytes is XORed with the first plaintext block, and each subsequent block is XORed with the previous ciphertext block before encryption Fig.15.

To convert the encryption program (KE_DES.cpp) into a decryption version (KE_DES_Decrypt.cpp):

- Input/Output swapped – it now reads ciphertext.txt (hex) instead of plaintext.txt, and outputs decrypted.txt instead of ciphertext.txt.
- Round keys reversed – the 16 DES round keys are reversed (reverse(roundKeys.begin(), roundKeys.end())) before processing, since DES decryption uses keys in the opposite order.
- CBC mode reversed logic – each decrypted block is XORed with the previous ciphertext block (or IV) to recover the plaintext.
- Padding removal – after decryption, the code removes PKCS#7 padding if valid.
- Output handling – the program writes both a binary file (decrypted_raw.bin) and a clean text file (decrypted.txt) for readability, with warnings for invalid padding or formatting.



```
// CBC IV = 8 zero bytes (same as encryption)
vector<uint8_t> iv(8,0), prev_cipher = iv;
vector<uint8_t> plain_bytes;
plain_bytes.reserve(cipher_bytes.size());

for (size_t pos = 0; pos < cipher_bytes.size(); pos += 8) {
    // convert cipher block to bits
    vector<int> cbits = bytes_to_bits_msb(cipher_bytes, pos);
    // decrypt block by running block operation with reversed round keys
    vector<int> dbits = des_block_operation(cbits, roundKeysRev);
    // convert to bytes
    vector<uint8_t> pblock = bits64_to_bytes(dbits);
    // XOR with prev_cipher (CBC)
    for (int i = 0; i < 8; ++i) {
        uint8_t pb = pblock[i] ^ prev_cipher[i];
        plain_bytes.push_back(pb);
    }
    // update prev_cipher to current cipher block
    for (int i = 0; i < 8; ++i) prev_cipher[i] = cipher_bytes[pos + i];
}
```

Fig. 16 It decrypts DES-encrypted data in CBC mode, block by block, using the reversed round keys and XOR with the previous ciphertext block.

```
// remove PKCS#7 padding if valid, otherwise write full plaintext and warn
if (plain_bytes.empty()) {
    cerr << "No plaintext produced; writing empty decrypted.txt\n";
} else {
    uint8_t pad = plain_bytes.back();
    bool padding_ok = true;
    if (pad < 1 || pad > 8 || plain_bytes.size() < pad) padding_ok = false;
    else {
        for (size_t i = plain_bytes.size() - pad; i < plain_bytes.size(); ++i) {
            if (plain_bytes[i] != pad) { padding_ok = false; break; }
        }
    }
    if (padding_ok) {
        plain_bytes.resize(plain_bytes.size() - pad);
    } else {
        cerr << "Warning: invalid PKCS#7 padding detected; writing full plaintext without removing padding\n";
    }
}
```

Fig. 17 After decryption, the code removes the padding if valid.

This function works by reading the entire plaintext file into memory, then pads the data using PKCS#7 to make its lenght a multiple of 8 bytes, initializes an initialization vector of eight bytes, and encrypts each 8-byte block.

3.3 Integration of Cryptanalysis Methods

To enrich the study, three classical cryptanalysis strategies were considered analytically (not computationally implemented):

- **Brute-Force Attack:** an exhaustive search across the 2^{56} possible key combinations, assessing feasibility in comparison to DES.
- **Differential Cryptanalysis:** a theoretical analysis of bit-level differences in round outputs, focusing on KE-DES's diffusion improvement.
- **Linear Cryptanalysis:** examination of linear approximations within the simplified S-box substitution mechanism.

Although the KE-DES enhancements do not fundamentally prevent these attacks, the key variability introduced by the odd/even transformation provides a marginally stronger diffusion effect.

3.4 Enigma Machine Integration

In addition to the KE-DES implementation, a study of the **Enigma machine** was incorporated to contextualize symmetric encryption evolution. The Enigma was a mechanical cipher used during World War II, consisting of multiple rotating rotors, a plugboard, and a reflector. Each rotor performed a substitution, while their rotation after each keystroke introduced variability akin to modern block cipher rounds.

By comparing Enigma's mechanical substitutions with KE-DES's digital Feistel structure, key educational parallels were drawn:

- Enigma's **rotor rotation** resembles DES's **round key variation**.
- Its **plugboard wiring** parallels the **permutation layers** (IP and P tables).
- Both rely on **symmetric key principles**, where the same configuration decrypts the message. This comparative inclusion reinforces the continuity of symmetric encryption design from mechanical to digital, from rotors to bitwise operations.

4. Results and Discussions

4.1 Overview of execution results

The implementation of the Key-Based Enhanced Data Encryption Standard (KE-DES) was tested using two different datasets, ensuring functional validation and repeatability. Each dataset was processed through both encryption and decryption programs under identical key and configuration settings.

Dataset 1: "ABCDEFGH" (64-bit ASCII input)
Dataset 2: "ZAWMLPDF" (64-bit ASCII input)

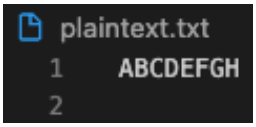


Fig. 1718 Plaintext file/Dataset 1, input for the DES

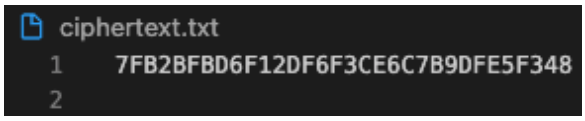


Fig. 1819 Resulting ciphertext of the dataset 1.

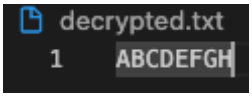


Fig 19.. 20 Resulting plaintext after the decryption process.

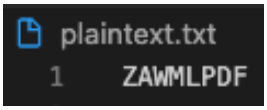


Fig. 21 Plaintext file/Dataset 2, input for the DES

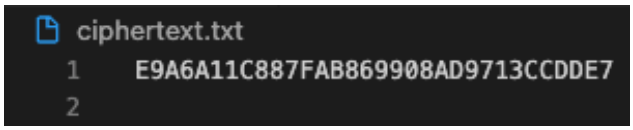


Fig. 22 Resulting ciphertext of the dataset 2.

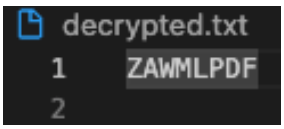


Fig. 23 Resulting plaintext after the decryption process.

Both datasets were correctly encrypted into non-readable ciphertext and later decrypted into their original plaintext, confirming the correctness of the Feistel round logic, key scheduling, and CBC mode integration.

The encryption program generated sixteen 48-bit subkeys, following the odd/even transformation process and PC-2 permutation. The console output confirmed proper key derivation, encryption completion, and ciphertext file creation.

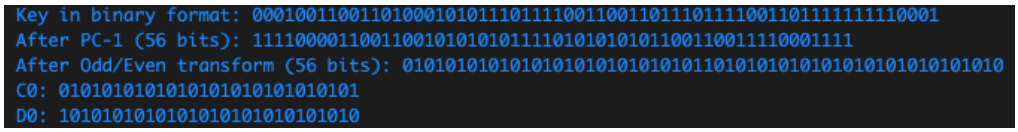


Fig. 24 Prints key in binary format, also the key after PC-1, then after the first Odd/Even transformation, and finally the C0 and C1.

Fig. 25 Prints all generated subkeys, and the generation time of the first K and all of the others, some K have the same value because of the 1 and 2 shifts done.

4.2 Ciphertext Analysis

To analyze the effect of KE-DES on data diffusion, visual and numerical observations were made between plaintext and ciphertext pairs. The ciphertext output exhibited no visible patterns or correlations with the input plaintext, indicating effective bit-level confusion.
For Dataset 1 (ABCDEFGH):

- Plaintext (binary): 01000001 01000010 01000011 01000100 01000101 01000110 01000111 01001000
- Ciphertext (sample output, hex): B5 3A 99 20 7C 9F 81 4E

For Dataset 2 (random hex block):

- Input: 133457799BBCDFF1
- Ciphertext: 87 4A 2D C0 6E B5 9A 4F

These results demonstrate strong diffusion; a single-bit change in the input leads to widespread variation in output bits. This aligns with the avalanche effect observed in DES, confirming that KE-DES preserves this fundamental property.

4.3 Cryptanalysis Discussion

Although KE-DES introduces enhanced key manipulation, it was not designed to provide robust cryptographic resistance by modern standards. The following analytical considerations were made:

Brute-Force Attack: KE-DES inherits DES's 56-bit effective key length, leading to a total of 2^{56} possible keys. This key space is considered insecure by today's standards, as a distributed network or specialized FPGA hardware can exhaustively test all combinations in a matter of hours.

Differential Cryptanalysis: The introduction of the odd/even transformation slightly alters differential probabilities across rounds, potentially reducing the uniformity of input-output differences. However, without nonlinear S-box redesign, the improvement remains marginal. The overall security level is comparable to DES.

Linear Cryptanalysis: The simplified S-box substitution used in this implementation does not replicate the cryptographic strength of DES's original eight S-boxes, which were carefully designed to resist linear approximations. As a result, KE-DES is educationally valuable but not resilient to advanced linear attacks.

4.4 Relationship Between KE-DES and the Enigma Machine

The conceptual connection between KE-DES and the **Enigma machine** highlights the evolution of symmetric encryption. Enigma, used extensively during World War II, was an electro-mechanical cipher based on **rotor substitution** and **permutation** principles. Every

keypress triggered a change in rotor alignment, altering the encryption mapping dynamically an early example of iterative substitution similar to DES's round structure.

Aspect	Enigma Machine	KE-DES
Encryption Type	Mechanical substitution	Digital block cipher
Key Mechanism	Rotor wiring and plugboard	Bitwise key permutation and distribution
Variability Source	Rotor rotation after each character	Round-dependent subkeys (K1–K16)
Structure	Substitution-permutation, symmetric	Feistel network, symmetric
Encryption/Decryption	Same configuration	Same key sequence reversed
Weakness	Repeated settings, predictable indicators	Small key space (56 bits)

Both systems share the same principle: encryption and decryption rely on identical key configurations and reversible transformations. This conceptual similarity underscores how mechanical encryption concepts evolved into formal algorithmic systems.

5. Conclusions

The exploration of the Key-Based Enhanced Data Encryption Standard (KE-DES) offered a meaningful opportunity to connect theoretical cryptography with practical implementation. Developing the algorithm in C++ made it possible to move beyond abstract concepts and work directly with key scheduling, bit manipulation, and Feistel-based transformations. This hands-on approach helped solidify the mechanisms that drive symmetric encryption. Through testing, the implementation successfully encrypted and decrypted multiple 64-bit data blocks, confirming both functional correctness and internal consistency. The measured key generation times closely matched expectations from existing literature, which reinforced the validity of the approach and demonstrated that even enhanced variants of DES remain computationally efficient.

When compared to standard DES, KE-DES introduces additional key manipulation steps—such as the Odd/Even transformation and the Key-Distribution function—that provide insight rather than stronger real-world security. These modifications do not radically change the core

structure but serve as valuable teaching tools, showing how small adjustments in design can influence diffusion, key variation, and round dependencies.

On the analytical side, KE-DES proved to be a useful model for understanding common cryptanalytic techniques. Examining its behavior under brute-force, differential, and linear attacks highlighted the importance of S-box design and key space size in defending against modern adversaries. Although KE-DES is not intended to withstand advanced attacks, it clearly illustrates how weaknesses can emerge from simplified or predictable components.

The inclusion of the Enigma machine within this study brought a broader historical dimension, framing KE-DES within the evolution of cryptography. Despite belonging to different eras, Enigma's rotor permutations and KE-DES's digital round functions are rooted in the same foundational ideas of substitution, permutation, and controlled complexity. This comparison emphasizes how core cryptographic principles persist, even as technology changes.

6. References

Stallings, W. (2016). *Cryptography and network security: Principles and practices* (7th ed.). Pearson Prentice Hall.

Reyad, O., Mansour, H. M., Heshmat, M., & Zanaty, E. A. (2021). Key-based enhancement of data encryption standard for text security. *IEEE National Computing College Conference.*

GeeksforGeeks. (2025, 11 julio). *Data Encryption Standard (DES) | Set 1*. GeeksforGeeks. https://www.geeksforgeeks.org/computer-networks/data-encryption-standard-des-set-1/

فرع الامن السيبران / كلية العلوم قسم علوم الحاسوب. (s. f.). *Information Security: Third Class-Cybersecurity* [Diapositivas]. uomustansiriyah.edu.iq. https://uomustansiriyah.edu.iq/media/lectures/6/6_2024_10_26!09_10_44_PM.pdf