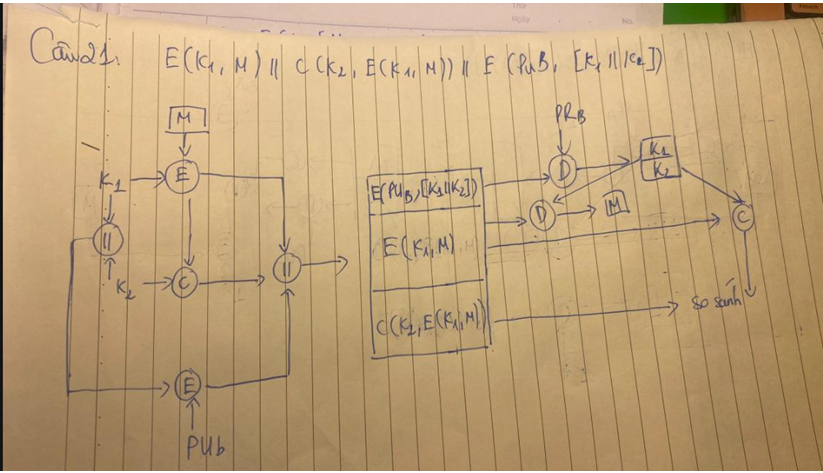
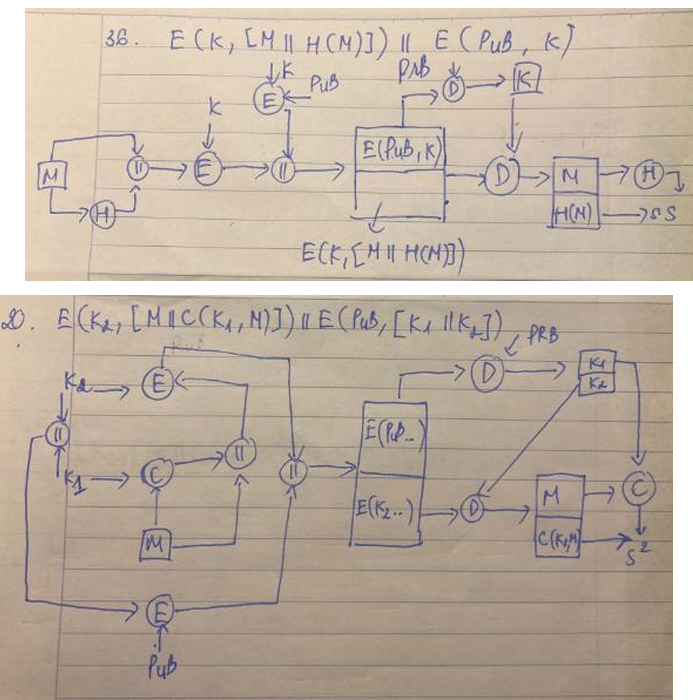
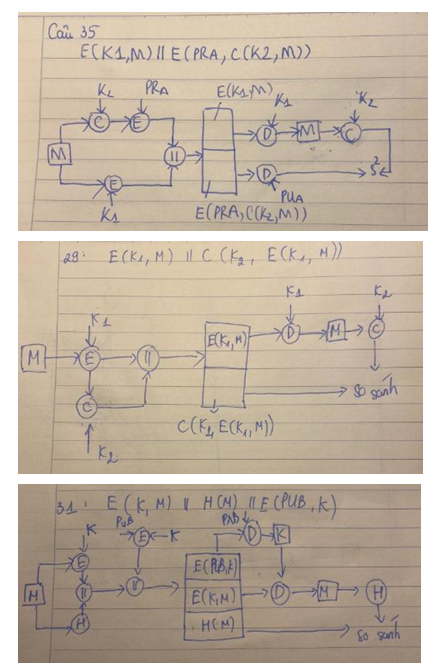
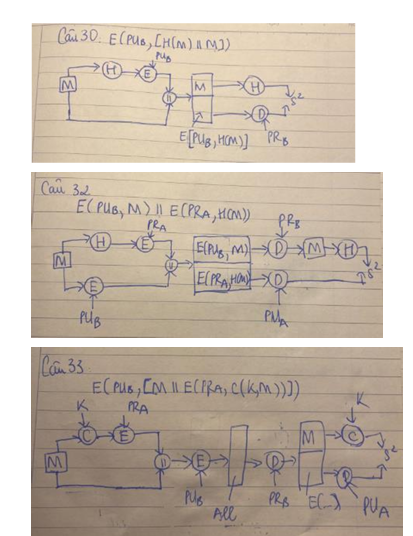
****

****

**** ****

**Bài tập 16 (ChatGPT).** Trong giao dịch điện tử SET, thông tin đơn hàng OI cần được gửi cho người bán. OI thường được ghép với chữ ký ghép và PIMD trước khi gửi đi. Ghép PIMD vào để làm gì, bỏ đi có được không?

**Bài giải.**

* Order Information (OI): Thông tin về đơn hàng bao gồm mô tả chi tiết sản phẩm, số lượng, giá trị, địa chỉ giao hàng, và các thông tin khác liên quan đến đơn hàng. OI giúp xác định các chi tiết quan trọng liên quan đến giao dịch.
* Payment Instructions Message Digest (PIMD): Là một giá trị băm (hash) của các thông tin thanh toán như số thẻ tín dụng, ngày hết hạn, và một số thông tin thanh toán khác. PIMD được sử dụng để đảm bảo tính toàn vẹn của thông tin thanh toán trong quá trình chuyển trực tuyến.

Khi ghép PIMD vào OI và ký chữ ký ghép (digital signature), nó giúp đảm bảo rằng thông tin thanh toán không bị thay đổi trong quá trình truyền tải. Chữ ký số và giá trị băm giúp xác định tính toàn vẹn và nguồn gốc của thông tin, cũng như đảm bảo rằng người nhận có thể xác nhận rằng thông tin đã được gửi bởi người gửi chính xác và không bị thay đổi.

Nếu bạn bỏ PIMD khỏi quá trình này, điều này có thể làm giảm đi tính toàn vẹn và an toàn của giao dịch. Mất PIMD có thể làm tăng nguy cơ về mối quan tâm về bảo mật trong quá trình thanh toán trực tuyến, nơi tính toàn vẹn của thông tin thanh toán là một yếu tố quan trọng. Do đó, thường thì không nên bỏ PIMD ra khỏi quá trình này để đảm bảo tính an toàn và toàn vẹn trong SET.

**Bài tập 20 (ChatGPT).** Nêu các bước kỹ thuật brute-force

Brute-force là một phương pháp tấn công mật khẩu bằng cách thử nghiệm tất cả các khả năng có thể của mật khẩu cho đến khi tìm ra mật khẩu đúng. Dưới đây là các bước cơ bản của kỹ thuật brute-force:

1. Xác định mục tiêu: Xác định mục tiêu của tấn công brute-force, chẳng hạn như tài khoản người dùng, mật khẩu hoặc khóa mã hóa.
2. Xác định không gian khóa: Xác định không gian khóa, tức là tất cả các khả năng có thể của mật khẩu hoặc khóa mã hóa.
3. Chuẩn bị công cụ tấn công: Sử dụng phần mềm hoặc kịch bản tấn công để tự động thực hiện việc thử nghiệm mật khẩu.
4. Chọn chiến lược thử nghiệm: Chọn chiến lược thử nghiệm mật khẩu, có thể bao gồm thử tất cả các ký tự có thể, thử theo từ điển, hoặc thử theo mô hình có thể dự đoán được.
5. Thực hiện tấn công: Thực hiện tấn công brute-force bằng cách thử nghiệm mật khẩu từ không gian khóa đã xác định. Cứ mỗi mật khẩu được thử, hệ thống sẽ kiểm tra xem mật khẩu đó có đúng không.
6. Ghi lại kết quả: Ghi lại tất cả các mật khẩu đã thử và kết quả (đúng hay sai).
7. Kiểm tra kết quả: Kiểm tra kết quả để xem liệu tấn công đã thành công hay chưa. Nếu tìm thấy mật khẩu đúng, tấn công thành công.
8. Tối ưu hóa và điều chỉnh: Tùy thuộc vào kết quả, tối ưu hóa và điều chỉnh chiến lược thử nghiệm để làm cho tấn công hiệu quả hơn.

Lưu ý rằng brute-force thường là một phương pháp chậm và đòi hỏi nhiều tài nguyên tính toán. Các biện pháp bảo mật như việc sử dụng mật khẩu mạnh, khóa kép, và giới hạn số lần thử mật khẩu được áp dụng để ngăn chặn loại tấn công này.

**Bài tập 21 chương 5 (docs).** Các kỹ thuật lấy mật khẩu, xâm nhập?

Kỹ thuật lấy mật khẩu

1. Thử các mật khẩu từng dùng với các tài khoản chuẩn gắn liền với hệ thống.
2. Thử hết tất cả các mật khẩu ngắn
3. Thử các từ trong từ điển trực tuyến của hệ thống, hoặc danh sách mà rất có khả năng được làm mật khẩu
4. Thu thập thông tin người dùng như họ tên đầy đủ, tên vợ, chồng, con cái, …
5. Thử tất cả số điện thoại người dùng, các số an sinh xã hội, số nhà, phòng làm việc
6. Thử tất cả các biển xe hợp lý của địa phương
7. Dùng trojan horse để ăn cắp mật khẩu
8. Đấu nối một đường truyền giữa người dùng từ xa với host hệ thống.

Kỹ thuật xâm nhập

1. Tìm cách lấy được mật khẩu người dùng
2. Khai thác các lỗ hổng an ninh của hệ thống (ví dụ lỗi tràn bộ đệm..)
3. Mua chuộc, dụ dỗ…

**Bài tập 22 (ChatGPT).** Phân biệt 2 chuẫn mã hóa dữ liệu AES và DES.

**Bài giải.**

Mật mã AES (Advanced Encryption Standard) và DES (Data Encryption Standard) là hai thuật toán mã hóa khác nhau được sử dụng để bảo vệ thông tin và dữ liệu. Dưới đây là một số điểm khác biệt giữa chúng:

1. Chiều dài khóa:
   1. AES: Hỗ trợ các khóa có độ dài 128, 192, hoặc 256 bit.
   2. DES: Sử dụng khóa 56 bit.
2. Độ an toàn:
   1. AES: Được coi là mạnh và an toàn, được sử dụng rộng rãi trong nhiều ứng dụng bảo mật.
   2. DES: Đã bị coi là không an toàn do độ dài khóa ngắn và đã bị thay thế bởi các tiêu chuẩn mã hóa mạnh mẽ hơn.
3. Hiệu suất:
   1. AES: Thường có hiệu suất cao hơn so với DES, đặc biệt là với các khối dữ liệu lớn.
   2. DES: Có thể trở nên chậm đối với các tập tin hoặc dữ liệu lớn.
4. Kiểu thuật toán:
   1. AES: Sử dụng thuật toán thay thế và hoán vị để mã hóa dữ liệu.
   2. DES: Sử dụng mô hình Substitution-Permutation Network (SPN).
5. Tiêu chuẩn:
   1. AES: Được chấp nhận rộng rãi và được sử dụng trong nhiều ứng dụng và giao thức bảo mật.
   2. DES: Đã bị thay thế bởi các tiêu chuẩn mới hơn như AES và 3DES.

Tóm lại, AES thường được ưa chuộng hơn DES trong các ứng dụng hiện đại do khả năng an toàn và hiệu suất cao hơn.

### ✅ 1. ****Cộng****

char F\_add(char R, char K) {

return R + K; // Cộng đơn giản

}

### ✅ 2. ****AND****

char F\_and(char R, char K) {

return R & K; // Phép AND bitwise

}

### ✅ 3. ****NOT****

char F\_not(char R, char K) {

return ~(R ^ K); // Đảo bit của XOR, tăng độ rối

}

### ✅ 4. ****Nhân****

char F\_mul(char R, char K) {

return (R \* K) % 256; // Tránh tràn, giữ trong 8 bit

}

### ✅ 5. ****Modulo tổng****

char F\_mod(char R, char K) {

return (R + K) % 128; // Chỉ lấy kết quả trong khoảng 0–127

}

### ✅ 6. ****XOR kết hợp quay phải****

char F\_xor\_rotate(char R, char K, int round) {

return R ^ ((K >> round) | (K << (8 - round))); // Kết hợp xoay phải

}

### ✅ 7. ****Kết hợp nhiều phép****

char F\_mix(char R, char K, int round) {

char rotK = (K >> round) | (K << (8 - round)); // rotateRight

return ((R + rotK) ^ (R | K)) % 256;

}

### ✅ 8. ****Ngẫu nhiên có kiểm soát (pseudo-random)****

char F\_prng(char R, char K, int round) {

return (R ^ ((K \* round + 31) % 256)) + round;

}

**Câu 3 - Đề 5 – Trang 14:**

#**include** <iostream>

#**include** <string>

**using** **namespace** std;

char **F**(char R, char Ki) {

**return** R - Ki;

}

char **rotateRight**(char ch, int i) {

**return** (char)((ch >> i) | (ch << (8 - i)));

}

string **MHKhoi**(char P0, char P1, char k) {

char K[6], L[6], R[6];

string C = " "; // Độ dài 2 ký tự

R[0] = P0;

L[0] = P1;

K[0] = k;

**for** (int i = 1; i <= 5; i++) {

K[i] = rotateRight(K[0], i); // Quay phải i lần

R[i] = L[i - 1] ^ F(R[i - 1], K[i]); // Feistel function

L[i] = R[i - 1];

}

C[0] = L[5];

C[1] = R[5];

**return** C;

}

string **GiaiMaKhoi**(char C0, char C1, char k) {

char K[6], L[6], R[6];

string P = " ";

L[5] = C0;

R[5] = C1;

K[0] = k;

**for** (int i = 1; i <= 5; i++) {

K[i] = rotateRight(K[0], i); // Quay phải i lần

}

**for** (int i = 5; i >= 1; i--) {

R[i - 1] = L[i];

L[i - 1] = R[i] ^ F(R[i - 1], K[i]);

}

P[0] = R[0];

P[1] = L[0];

**return** P;

}

int **main**() {

string P, C, decryptedP;

char k;

cout << "Nhap chuoi plaintext: ";

getline(cin, P);

cout << "Nhap khoa K: ";

cin >> k;

**if** (P.size() % 2 == 1) P += 'X'; // Thêm 'X' nếu độ dài lẻ

**for** (int i = 0; i < P.size(); i += 2) {

C += MHKhoi(P[i], P[i + 1], k);

}

cout << "Chuoi ma hoa: " << C << endl;

**for** (int i = 0; i < C.size(); i += 2) {

decryptedP += GiaiMaKhoi(C[i], C[i + 1], k);

}

**if** (!decryptedP.empty() && decryptedP[decryptedP.size() - 1] == 'X') {

decryptedP.erase(decryptedP.size() - 1); // Bỏ 'X' cuối nếu có

}

cout << "Chuoi giai ma: " << decryptedP << endl;

**return** 0;

}

**Câu 3 – Đề 6 – T15:**

#**include** <iostream>

#**include** <string>

**using** **namespace** std;

// Hàm F đơn giản dùng phép XOR thay vì phép trừ

char **F**(char R, char Ki) {

**return** R ^ Ki; // XOR

}

// Hàm quay phải ký tự k đi n bit

char **rotateRight**(char k, int n) {

**return** (k >> n) | (k << (8 - n)); // Quay phải n lần

}

// Mã hóa một khối gồm 2 ký tự P0, P1 với khóa k

string **MHKhoi**(char P0, char P1, char k) {

char K[8], L[8], R[8];

string C = " "; // 2 ký tự

// Khởi tạo

R[0] = P0;

L[0] = P1;

K[0] = k;

// Tạo 7 khóa con và thực hiện 7 vòng Feistel

**for** (int i = 1; i <= 7; i++) {

K[i] = rotateRight(K[0], i); // Quay phải i lần tạo khóa con

R[i] = L[i - 1] ^ F(R[i - 1], K[i]); // R mới

L[i] = R[i - 1]; // L mới

}

// Kết quả sau 7 vòng: L7 và R7 là ciphertext

C[0] = L[7];

C[1] = R[7];

**return** C;

}

// Giải mã một khối gồm 2 ký tự C0, C1 với khóa k

string **GiaiMaKhoi**(char C0, char C1, char k) {

char K[8], L[8], R[8];

string P = " "; // plaintext 2 ký tự

// Gán lại giá trị L7, R7 từ ciphertext

L[7] = C0;

R[7] = C1;

K[0] = k;

// Tạo khóa con

**for** (int i = 1; i <= 7; i++) {

K[i] = rotateRight(K[0], i);

}

// Giải mã ngược từ vòng 7 → 1

**for** (int i = 7; i >= 1; i--) {

R[i - 1] = L[i]; // R trước = L sau

L[i - 1] = R[i] ^ F(R[i - 1], K[i]); // L trước

}

// Kết quả ban đầu

P[0] = R[0];

P[1] = L[0];

**return** P;

}

int **main**() {

string P, C, decryptedP;

char k;

cout << "Nhap chuoi plaintext: ";

getline(cin, P);

cout << "Nhap khoa K: ";

cin >> k;

// Nếu độ dài lẻ → thêm 'X'

**if** (P.size() % 2 == 1) P += 'X';

// Mã hóa từng cặp ký tự

**for** (int i = 0; i < P.size(); i += 2) {

C += MHKhoi(P[i], P[i + 1], k);

}

cout << "Chuoi ma hoa: " << C << endl;

// Giải mã từng cặp ký tự

**for** (int i = 0; i < C.size(); i += 2) {

decryptedP += GiaiMaKhoi(C[i], C[i + 1], k);

}

// Xoá ký tự 'X' nếu có (dùng để đệm)

**if** (!decryptedP.empty() && decryptedP[decryptedP.size() - 1] == 'X') {

decryptedP.erase(decryptedP.size() - 1);

}

cout << "Chuoi giai ma: " << decryptedP << endl;

**return** 0;

}

**Câu 2 – Đề 7: So sánh sự giống và khác nhau của Hash và MAC? Hàm nào sử dụng chữ ký số tốt hơn? Tại sao?**

**Giống:**

* **Tính toán giá trị cố định:** Cả hàm Hash và hàm MAC đều nhận đầu vào có độ dài bất kỳ và trả về một giá trị cố định đã định.
* **Đảm bảo tính toàn vẹn:** Cả hai đều dùng để đảm bảo tính toàn vẹn của dữ liệu, nghĩa là một thay đổi nhỏ trong đầu vào sẽ dẫn đến sự thay đổi lớn trong giá trị đầu ra.
* **Ứng dụng trong bảo mật:** Cả hai đều là công cụ mật mã học quan trọng và được sử dụng rộng rãi trong các ứng dụng bảo mật hiện đại.

|  | **Mac** | **Hash** |
| --- | --- | --- |
| **Khóa bí mật** | Sử dụng khóa bí mật cùng với đầu vào để tính toán giá trị MAC, đảm bảo cả tính toàn vẹn và xác thực nguồn gốc dữ liệu. | Không sử dụng khóa bí mật. Giá trị được tính toán chỉ dựa vào dữ liệu. |
| **Mục đích sử dụng** | Dùng để đảm bảo tính toàn vẹn và xác thực nguồn gốc dữ liệu, thường được sử dụng trong truyền thông mạng và các giao thức bảo mật. | Chủ yếu để kiểm tra tính toàn vẹn của dữ liệu. Dùng trong lưu trữ mật khẩu, chữ ký số, và kiểm tra tính toàn vẹn của các tập tin. |

### ****Hàm Hash sử dụng tốt hơn cho chữ ký số:****

1. **Tính toàn vẹn và hiệu quả:**  
   Hàm hash nhanh chóng tạo ra một giá trị băm nhỏ gọn đại diện cho dữ liệu, giúp phát hiện thay đổi trong dữ liệu một cách hiệu quả.
2. **Hệ thống khóa công khai:**  
   Chữ ký số dựa trên hàm băm giá trị băm bằng khóa riêng và xác minh bằng khóa công khai, phù hợp với cách hoạt động của hàm hash.
3. **Không yêu cầu khóa bí mật:**  
   Hàm hash không cần khóa bí mật để tính toán giá trị, đơn giản hóa quá trình tạo và xác minh chữ ký số.

**Câu 3 – Đề 7:** Giống 2 đề ở trên, chỉ khác ở chỗ: Số vòng xử lý là n=4. Hàm F thực hiện phép OR

// ----------------- feistel OR -------------------

#**include** <iostream>

#**include** <string>

**using** **namespace** std;

// Hàm F sử dụng phép OR giữa R và Ki

char **F**(char R, char Ki) {

**return** R | Ki; // OR logic

}

// Hàm quay phải 1 ký tự k đi n bit

char **rotateRight**(char k, int n) {

**return** (k >> n) | (k << (8 - n)); // Quay phải n lần

}

// Mã hóa một khối gồm 2 ký tự với khóa k

string **MHKhoi**(char P0, char P1, char k) {

char K[5], L[5], R[5]; // Dùng 4 vòng → mảng 5 phần tử

string C = " ";

R[0] = P0;

L[0] = P1;

K[0] = k;

// 4 vòng mã hóa

**for** (int i = 1; i <= 4; i++) {

K[i] = rotateRight(K[0], i); // Tạo khóa con bằng quay phải

R[i] = L[i - 1] ^ F(R[i - 1], K[i]); // R mới = L cũ ^ F(R cũ, K con)

L[i] = R[i - 1]; // L mới = R cũ

}

C[0] = L[4];

C[1] = R[4];

**return** C;

}

// Giải mã một khối gồm 2 ký tự với khóa k

string **GiaiMaKhoi**(char C0, char C1, char k) {

char K[5], L[5], R[5];

string P = " ";

L[4] = C0;

R[4] = C1;

K[0] = k;

// Tạo khóa con như khi mã hóa

**for** (int i = 1; i <= 4; i++) {

K[i] = rotateRight(K[0], i);

}

// Giải mã ngược từ vòng 4 về 1

**for** (int i = 4; i >= 1; i--) {

R[i - 1] = L[i];

L[i - 1] = R[i] ^ F(R[i - 1], K[i]);

}

P[0] = R[0];

P[1] = L[0];

**return** P;

}

int **main**() {

string P, C, decryptedP;

char k;

cout << "Nhap chuoi plaintext: ";

getline(cin, P);

cout << "Nhap khoa K: ";

cin >> k;

// Nếu độ dài lẻ → thêm 'X' để đủ cặp

**if** (P.size() % 2 == 1) P += 'X';

// Mã hóa từng cặp ký tự

**for** (int i = 0; i < P.size(); i += 2) {

C += MHKhoi(P[i], P[i + 1], k);

}

cout << "Chuoi ma hoa: " << C << endl;

// Giải mã

**for** (int i = 0; i < C.size(); i += 2) {

decryptedP += GiaiMaKhoi(C[i], C[i + 1], k);

}

// Bỏ 'X' nếu là ký tự đệm

**if** (!decryptedP.empty() && decryptedP[decryptedP.size() - 1] == 'X') {

decryptedP.erase(decryptedP.size() - 1);

}

cout << "Chuoi giai ma: " << decryptedP << endl;

**return** 0;

}