MINISTERUL EDUCATIEI, CULTURII
ȘI CERCETĂRII AL REPUBLICII MOLDOVA
UNIVERSITATEA TEHNICA A MOLDOVEI
Facultatea Calculatoare, Informatică și Microelectronică

Lucrare de laborator nr.2

la disciplina: Securitatea Informațională

Tema: Algoritmi de criptare asimetrici (RSA)

A efectuat: Șaptefrați Victor
gr. TI-191 F/R

A verificat: Poștaru Andrei
lect. univ.

Chișinău 2024

Introducere

RSA (Rivest–Shamir–Adleman) este unul dintre primii algoritmi publici de criptografie cu cheie publică și este utilizat pentru securitatea datelor în rețelele de calculatoare. A fost dezvoltat în 1977 de către *Ron Rivest*, *Adi Shamir* și *Leonard Adleman* la MIT și rămâne unul dintre cele mai utilizate algoritme de acest gen.

RSA permite atât criptarea, cât și decriptarea datelor, oferind securitate în transmiterea informațiilor. Este bazat pe principiul matematic al dificultății de a factoriza numere mari compuse, un principiu considerat robust în fața atacurilor.

Principalele funcționalități ale RSA sunt:
- *Confidențialitate*: Prin criptarea datelor cu cheia publică, RSA asigură că numai destinatarul cu cheia privată corespunzătoare poate decripta și citi mesajul.
- *Autenticitate*: Poate fi folosit și pentru a semna digital un mesaj, asigurându-se că mesajul a fost emis de o anumită parte.
- *Integritate*: Asigură că datele nu au fost modificate în timpul transmiterii.
- *Non-repudiere*: Semnătura digitală RSA previne emitentul să nege trimiterea sau semnarea mesajului.

Implementare

Implementarea RSA constă din trei etape principale: generarea cheilor, criptarea și decriptarea.

1. Generarea cheilor

Generarea cheilor în RSA este procesul prin care se generează o pereche de chei, una publică și una privată, după cum urmează:

- Se aleg două numere prime distincte, p și q.
- Se calculează $n=p\cdot q$, unde n este modulul pentru ambele chei și lungimea sa determină rezistența cheii.
- Se calculează $\phi(n)=(p-1)\cdot(q-1)$, unde ϕ este funcția lui Euler.
- Se alege un număr întreg e, astfel încât $1<e<\phi(n)$ și e să fie coprim cu $\phi(n)$, adică e și $\phi(n)$ să aibă ca cel mai mare divizor comun numărul 1.
- Se calculează d, inversa modulară a lui e modulo $\phi(n)$, astfel încât $d\cdot e$ să fie congruent cu 1 modul $\phi(n)$.

Cheia publică e formată din perechea (e, n), iar cheia privată este formată din perechea (d, n).

2. Criptarea

Pentru a cripta un mesaj M, care este un număr întreg mai mic decât n, se calculează mesajul criptat C folosind formula mod $C=M^e \bmod n$.

3. Decriptarea

Pentru a decripta un mesaj criptat C, se calculează mesajul original M folosind cheia privată d prin formula $M=C^d \bmod n$.

Codul programului

```
class RSA {
  /**
   * Această funcție calculează cel mai mare divizor comun (gcd)
   * între două numere, a și b, folosind algoritmul lui Euclid.
   *
   * Este folosită pentru a verifica dacă două numere sunt coprime,
   * adica să nu aibă alți divizori comuni în afara lui 1.
   *
   * @param a Primul număr
   * @param b Al doilea număr
   *
   * @returns Cel mai mare divizor comun
   */
  private static gcd(a: bigint, b: bigint): bigint {
    while (b !== 0n) {
      let t = b;
      b = a % b;
      a = t;
    }
    return a;
  }

  /**
   * Calculează inversul modular `d ≡ e^(-1) mod φ`,
   * folosind algoritmul extins al lui Euclid.
   *
   * @param e Exponentul public, reprezentat ca un bigint.
   * @param phi φ, totientul folosit pentru a calcula cheile, reprezentat ca
un bigint.
   * @returns Inversul modular al lui e modulo φ.
   */
  private static modInverse(e: bigint, phi: bigint) {
    // Salvăm valoarea inițială a lui φ pentru a ajusta rezultatul final dacă
este negativ.
    let m0 = phi;

    // Inițializăm y și x, care sunt folosite în algoritmul extins al lui
Euclid.
    // y va fi folosit pentru a stoca valorile intermediare ale inversului
modular,
    // în timp ce x va stoca valoarea anterioară a lui y în fiecare iterație.
    let y = BigInt(0);
    let x = BigInt(1);

    // Dacă φ este 1, inversul modular nu există, deci returnăm 0.
    if (phi === BigInt(1)) return BigInt(0);
```

```
    // Executăm algoritmul extins al lui Euclid.
    // Acesta funcționează prin găsirea coeficienților x și y astfel încât e
* x + φ * y = gcd(e, φ).
    while (e > 1) {
      // Calculăm partea întreagă a împărțirii lui e la φ.
      let q = e / phi;

      // Actualizăm valorile lui phi și e folosind algoritmul de schimbare
Euclidian.
      let t = phi;
      phi = e % phi;
      e = t;

      t = y; // Actualizăm valorile lui y și x.
      y = x - q * y; // Calculează noua valoare a lui y.
      x = t; // Actualizează x la valoarea anterioară a lui y.
    }

    // Dacă x este negativ, adăugăm m0 (valoarea originală a lui phi) pentru
a-l face pozitiv.
    // Inversul modular trebuie să fie întotdeauna un număr pozitiv.
    if (x < 0) x += m0;

    // Returnăm inversul modular calculat.
    return x;
  }

  /**
   * Această funcție generează cheile publice și private.
   *
   * Cheile sunt generate folosind două numere prime, `p` și `q`.
   *
   * Funcția calculează întâi valorile `n` și `phi` (φ, funcția lui Euler)
   * apoi găsește un exponent public `e` și calculează exponentul privat `d`.
   *
   * @param p Primul număr prim
   * @param q Al doilea număr prim
   *
   * @returns Cheile publice și private
   */
  public static generateKeys(p: bigint, q: bigint) {
    const n = p * q;

    // Totientul (sau Indicatorul lui Euler) reprezentează
    // numărul de numere mai mici decât n care sunt coprime cu `n`.
    const phi = (p - 1n) * (q - 1n);
```

```
    // Exponentul public `e` este ales astfel încât să fie coprim cu `phi`.
    // Valoarea inițială este 3, deoarece este cel mai mic număr coprim cu
`phi`.
    let e = 3n;
    while (this.gcd(e, phi) !== 1n) {
      e++;
    }

    // Calculează exponentul privat `d` folosind inversul modular.
    const d = this.modInverse(BigInt(e), BigInt(phi));

    return { publicKey: { e, n }, privateKey: { d, n } };
  }

  /**
   * Această funcție criptează o valoare folosind cheia publică.
   *
   * Criptarea se face conform formulei: `c = m^e mod n`
   */
  public static encrypt(
    value: bigint,
    publicKey: { e: bigint; n: bigint }
  ): BigInt {
    return this.modPow(value, BigInt(publicKey.e), BigInt(publicKey.n));
  }

  /**
   * Această funcție decriptează o valoare folosind cheia privată.
   *
   * Decriptarea se face conform formulei: `m = c^d mod n`
   */
  public static decrypt(
    cipherValue: bigint,
    privateKey: { d: bigint; n: bigint }
  ): BigInt {
    return this.modPow(cipherValue, BigInt(privateKey.d),
BigInt(privateKey.n));
  }

  /**
   * Calculează `base^exponent mod modulus` folosind exponentierea modulară.
   *
   * Este alternativa pentru expresia `Math.pow(base, exponent) % modulus`,
   * care permite calcularea rezultatului fără a depăși limitele numerice.
   */
  private static modPow(base: bigint, exponent: bigint, modulus: bigint) {
    let result = BigInt(1);
    base = base % modulus;
```

```
    while (exponent > 0) {
      if (exponent % BigInt(2) === BigInt(1)) {
        result = (result * base) % modulus;
      }
      exponent = exponent >> BigInt(1);
      base = (base * base) % modulus;
    }

    return result;
  }

  /**
   * Această funcție criptează un caracter folosind cheia publică.
   *
   * Caracterul este transformat în cod ASCII, apoi criptat.
   */
  private static encryptChar(
    char: string,
    publicKey: { e: bigint; n: bigint }
  ): BigInt {
    const charCode = char.charCodeAt(0);
    return this.encrypt(BigInt(charCode), publicKey);
  }

  /**
   * Această funcție decriptează un caracter folosind cheia privată.
   */
  private static decryptChar(
    ciphertext: bigint,
    privateKey: { d: bigint; n: bigint }
  ): string {
    const decryptedCode = this.decrypt(ciphertext, privateKey);
    return String.fromCharCode(Number(decryptedCode));
  }

  /**
   * Metodă ajutătoare pentru a cripta un text întreg.
   *
   * Textul este împărțit în caractere, apoi fiecare caracter este criptat.
   * La final, caracterele criptate sunt concatenate într-un șir de
caractere.
   *
   * @param text Textul de criptat
   * @param publicKey Cheia publică folosită pentru criptare
   *
   * @returns Textul criptat
   */
```

```
  public static encryptText(text: string, publicKey: { e: bigint; n: bigint
}) {
    const ciphertextParts: string[] = [];
    for (const char of text) {
      const encryptedChar = RSA.encryptChar(char, publicKey);
      ciphertextParts.push(encryptedChar.toString());
    }

    return ciphertextParts.join(" ");
  }

  /**
   * Metodă ajutătoare pentru a decripta un text întreg.
   *
   * @param text Textul de decriptat
   * @param privateKey Cheia privată folosită pentru decriptare
   *
   * @returns Textul decriptat (în clar)
   */
  public static decryptText(
    text: string,
    privateKey: { d: bigint; n: bigint }
  ) {
    const plaintextParts: string[] = [];
    for (const char of text.split(" ")) {
      const decryptedChar = RSA.decryptChar(BigInt(char), privateKey);
      plaintextParts.push(decryptedChar);
    }

    return plaintextParts.join("");
  }
}
```

Execuția programului

```
const { publicKey, privateKey } = RSA.generateKeys(977n, 9677n);
console.log("Public key:", publicKey);
console.log("Private key:", privateKey);

const plaintext = "RSA@vixeven"

const encrypted = RSA.encryptText(plaintext, publicKey);
console.log(`Encrypted: ${encrypted}`);

const decrypted = RSA.decryptText(encrypted, privateKey);
console.log(`Decrypted: ${decrypted}`);
```

```
Public key:                { e: 3n, n: 9454429n }
Private key:               { d: 6295851n, n: 9454429n }
Encrypted:                 551368 571787 274625 262144 1643032 1157625
1728000 1030301 1643032 1030301 1331000
Decrypted:                 RSA@vixeven
```