| **Course Name:** | **Applied Cryptography (**116U01E628**)** | **Semester:** | **VI** |
| --- | --- | --- | --- |
| **Date of Performance:** | **26/01/2025** | **DIV/ Batch No:** | **C - 3** |
| **Student Name:** | **Romil Lodaya** | **Roll No:** | **16010122096** |

**Experiment No: 3**

**Title: Cryptographic Arithmetic**

| **Aim and Objective of the Experiment:** |
| --- |
| To learn about Cryptographic Arithmetic and its implementation   1. Prime factor generation 2. Extended Euclidian algorithm |

| **COs to be achieved:** |
| --- |
| **CO2: Demonstrate and implement various Cryptographic Algorithms for securing systems** |

| **Books/ Journals/ Websites referred:** |
| --- |
| 1. Stallings, W., Cryptography and Network Security: Principles and Practice, Second edition, Person Education 2. “Caesar Cipher in cryptography”, <https://www.geeksforgeeks.org/caesar-cipher-in-cryptography/>, last retrieved on Aug 01, 2023 3. “PlayFair Cipher in cryptography”: <https://www.geeksforgeeks.org/playfair-cipher-with-examples/>, last retrieved on Aug 01, 2023 4. “Transposition cipher in cryptology,  ”<https://www.britannica.com/topic/transposition-cipher>, last retrieved on Aug 01, 2023 |

| **Theory:** |
| --- |
| **Abstract**:-  Cryptographic arithmetic involves using mathematical operations and techniques within cryptography to secure communication and data. It includes concepts like modular arithmetic, one-way functions, prime numbers, exponentiation, and elliptic curve cryptography, all of which are crucial for ensuring the confidentiality, integrity, and authenticity of data in cryptographic systems.  **Related Theory:**  **1. Number Theory:** Number theory forms the foundation of many cryptographic algorithms. Concepts such as prime numbers, modular arithmetic, greatest common divisors, and Euler's totient function are central to understanding and implementing cryptographic arithmetic.  **2. Modular Arithmetic:** Modular arithmetic is fundamental in cryptographic operations. It involves performing arithmetic operations within a finite set, called a modulus. Theorems related to modular arithmetic, like Euler's Totient Theorem, are critical for cryptographic algorithms like RSA and Diffie-Hellman.  **3. Additive Inverse:** Two numbers a and b are additive inverses of each other if a + b ≡ 0 (mod n). The additive inverse of a can be calculated as b = n − a.  **4. Multiplicative Inverse:** Two numbers a and b are the multiplicative inverse of each other if a × b ≡ 1 (mod n) |

| **Algorithm :** |
| --- |
| **1. Prime Factors**  **Algorithm:**   1. **Function checkPrime(n)**:    * Initialize cnt = 0.    * Iterate i from 1 to sqrt(n):      + If n % i == 0:        - Increment cnt.        - If n / i != i, increment cnt again.    * If cnt == 2, return true (prime); else, return false. 2. **Function getPrimeFactors(n)**:    * Initialize an empty vector v.    * Iterate i from 2 to n:      + If n % i == 0 and checkPrime(i) is true, add i to v.    * Return v. 3. **Main**:    * Take input n.    * Call getPrimeFactors(n) and store the result in primeFactors.    * Print the prime factors.   **2. Extended GCD**  **Algorithm:**   1. **Main**:    * Take input a and b.    * Initialize:      + r1 = a, r2 = b.      + s1 = 1, s2 = 0.      + t1 = 0, t2 = 1.    * While r2 > 0:      + Compute q = r1 / r2.      + Update:        - r = r1 % r2.        - s = s1 - (s2 \* q).        - t = t1 - (t2 \* q).      + Shift values:        - r1 = r2, r2 = r.        - s1 = s2, s2 = s.        - t1 = t2, t2 = t.    * Print gcd = r1, s = s1, t = t1. |

| **Solve a small numerical for assigned algorithm(Paste photograph of same) :** |
| --- |
|  |

| **Code :** |
| --- |
| **1] Prime Factors:**  #include <bits/stdc++.h>  using namespace std;  bool checkPrime(int n)  {      int cnt = 0;      for(int i = 1; i <= sqrt(n); i++)      {          if(n % i == 0)          {              cnt++;              if(n / i != i)              {                  cnt++;              }          }      }      if(cnt == 2)return true;      else return false;  }  vector<int> getPrimeFactors(int n)  {      vector<int> v;      for(int i = 2; i <= n; i++)      {          if(n % i == 0)          {              if(checkPrime(i))              {                  v.push\_back(i);              }          }      }      return v;  }  int main()   {      int n;      cout << "Enter the number: ";      cin >> n;      vector<int> primeFactors = getPrimeFactors(n);      cout << "Prime Factors for " << n << ": ";      for(auto i: primeFactors)cout << i << " ";      cout << endl;  }  **2] Extended GCD:**  #include<bits/stdc++.h>  using namespace std;  int main()  {      int a, b;      cout << "Enter the numbers a & b: ";      cin >> a >> b;      int q = 0, r = 0, s = 0, t = 0, r1 = a, r2 = b, s1 = 1, s2 = 0, t1 = 0, t2 = 1;      while(r2 > 0)      {          q = r1 / r2;          r = r1 % r2;          s = s1 - (s2 \* q);          t = t1 - (t2 \* q);          r1 = r2;          r2 = r;          s1 = s2;          s2 = s;          t1 = t2;          t2 = t;      }      cout << "gcd: " << r1 << " s: " << s1 << " t: " << t1 << endl;      return 0;  } |

| **Output:** |
| --- |
| **1] Prime Factors:**    **2] Extended GCD:** |

| **Post Lab Subjective/Objective type Questions:** |
| --- |
| 1. What is the significance of finding the prime factorization of a number in cryptography?   **1. Basis for Security in RSA Cryptography:**   * **RSA Encryption** relies on the difficulty of factoring the product of two large prime numbers. * The public key is derived from the product of two large primes (n = p \* q), while the private key depends on knowing the prime factors (p and q). * If an attacker can efficiently factorize n, they can derive the private key and break the encryption. Thus, the security of RSA depends on the **hardness of prime factorization**.   **2. Computational Hardness:**   * Prime factorization is a **computationally hard problem** for large numbers, especially when the primes are very large (e.g., 1024-bit or 2048-bit numbers). * No known efficient algorithm can factorize large numbers in polynomial time, making it a **one-way function** for practical purposes.   **3. Key Generation:**   * In cryptographic systems, prime factorization is used during **key generation**:   + Two large prime numbers are randomly selected and multiplied to create a composite number (n).   + The difficulty of reversing this process (factoring n) ensures the security of the keys.   **4. Mathematical Foundations:**   * Prime factorization is fundamental to number theory, which underpins many cryptographic algorithms. * Concepts like **Euler’s Totient Function** (used in RSA) rely on knowing the prime factors of a number to compute values like φ(n) = (p-1)(q-1).   **5. Quantum Computing Threat:**   * Classical computers struggle with prime factorization, but **quantum computers** could potentially solve it efficiently using **Shor’s Algorithm**. * This poses a future threat to cryptographic systems like RSA, driving research into **post-quantum cryptography**.   **6. Digital Signatures and Authentication:**   * Prime factorization is also used in **digital signatures** and **authentication protocols**. * For example, RSA signatures rely on the private key (derived from prime factors) to sign messages, and the public key (derived from the product of primes) to verify them. |

| **Conclusion:** |
| --- |
| This experiment enhanced understanding of cryptographic arithmetic by implementing prime factor generation and the extended Euclidean algorithm, achieving CO2 by demonstrating foundational cryptographic techniques essential for securing systems. |