Linear Algebra in Cryptography: Public-Key Cryptosystems and Elliptic Curves

Badger Code

July 22, 2023

Abstract

This paper explores the role of linear algebra in cryptography, focusing on public-key cryptosystems and elliptic curves. We provide an overview of cryptographic principles, discuss popular public-key algorithms like RSA and Diffie-Hellman, and explore the mathematics behind elliptic curves and their applications in cryptographic protocols. Additionally, we discuss cryptanalysis techniques and the impact of quantum computing on existing cryptosystems.

1 Introduction

Cryptography is the science of secure communication and data protection. Its primary goal is to provide a means for two parties to communicate securely even in the presence of potential adversaries. Cryptography plays a critical role in modern information security, ensuring the confidentiality, integrity, and authenticity of sensitive data. In this paper, we delve into the connection between linear algebra and cryptographic systems, with a specific focus on public-key cryptosystems and elliptic curves.

2 Overview of Cryptography

Cryptography can be classified into two main categories: symmetric-key cryptography and asymmetric-key cryptography (public-key cryptography).

2.1 Symmetric-Key Cryptography

In symmetric-key cryptography, a single secret key is used for both encryption and decryption. The same key is shared by both parties involved in the communication. Common symmetric-key algorithms include AES (Advanced Encryption Standard) and DES (Data Encryption Standard).

2.2 Asymmetric-Key Cryptography

Asymmetric-key cryptography uses a pair of keys: a public key and a private key. The public key is known to everyone, while the private key is kept secret. Messages encrypted with the public key can only be decrypted with the corresponding private key, and vice versa. This property allows secure communication even when the public key is openly distributed.

3 Public-Key Cryptosystems

Public-key cryptosystems are a fundamental aspect of modern cryptography, enabling secure communication over insecure channels. They were first introduced by Whitfield Diffie and Martin Hellman in their seminal 1976 paper "New Directions in Cryptography."

3.1 RSA (Rivest-Shamir-Adleman) Algorithm

RSA is one of the most widely used public-key cryptosystems, named after its inventors. It relies on the computational difficulty of factoring the product of two large prime numbers.

3.1.1 Key Generation

The RSA algorithm involves the generation of two large prime numbers, p and q. A modulus $n = pq$ is calculated, and an encryption exponent e and a decryption exponent d are derived.

```
# RSA key generation
def generate_rsa_key():
    # Step 1: Generate large prime numbers p and q
    p = generate_large_prime()
    q = generate_large_prime()

    # Step 2: Calculate modulus n
    n = p * q

    # Step 3: Calculate totient phi(n)
    phi_n = (p - 1) * (q - 1)

    # Step 4: Choose encryption exponent e
    e = choose_encryption_exponent(phi_n)

    # Step 5: Calculate decryption exponent d
    d = modular_inverse(e, phi_n)

    return (n, e, d)
```

3.1.2 Encryption and Decryption

The encryption process involves raising the plaintext message M to the power of the encryption exponent e modulo n. The resulting ciphertext C can be sent over the insecure channel.

```
# RSA encryption
def rsa_encrypt(message, n, e):
    return pow(message, e, n)
```

The decryption process involves raising the ciphertext C to the power of the decryption exponent d modulo n, yielding the original plaintext message M.

```
# RSA decryption
def rsa_decrypt(ciphertext, n, d):
    return pow(ciphertext, d, n)
```

3.2 Diffie-Hellman Key Exchange

The Diffie-Hellman key exchange algorithm enables two parties to establish a shared secret over an insecure channel without ever exchanging their private keys.

3.2.1 Key Exchange

The algorithm involves the following steps:

1. Both parties agree on a large prime number p and a primitive root modulo p, denoted by g.
2. Each party generates a private key: a for Party A and b for Party B.
3. Party A calculates $A = g^a \mod p$, and Party B calculates $B = g^b \mod p$.
4. The public values A and B are exchanged.
5. Party A calculates the shared secret as $S = B^a \mod p$, and Party B calculates the shared secret as $S = A^b \mod p$. Both parties now share the same secret value S.

```
# Diffie-Hellman key exchange
def diffie_hellman(p, g):
    # Step 1: Parties agree on a large prime number p and a primitive root g mod

    # Step 2: Generate private keys a and b
    a = generate_private_key()
    b = generate_private_key()

    # Step 3: Calculate public values A and B
    A = pow(g, a, p)
    B = pow(g, b, p)

    # Step 4: Exchange public values A and B
```

```
    # Step 5: Calculate the shared secret S
    S_A = pow(B, a, p)
    S_B = pow(A, b, p)

    # Both parties now have the same shared secret S
    return S_A == S_B, S_A
```

4 Elliptic Curve Cryptography (ECC)

Elliptic Curve Cryptography is another widely used public-key cryptosystem that is based on the mathematical properties of elliptic curves.

4.1 Elliptic Curves and Points

An elliptic curve is defined by an equation of the form $y^2 = x^3 + ax + b$, where a and b are constants. The curve also includes a point at infinity $\mathcal{O}$, which serves as the identity element.

Points on an elliptic curve form a group, and addition of points on the curve is well-defined.

```
# Elliptic Curve Point Addition
def ec_point_add(p1, p2, a, p):
    if p1 == (0, 0): return p2
    if p2 == (0, 0): return p1
    if p1 == p2:
        m = (3 * p1[0]**2 + a) * modular_inverse(2 * p1[1], p) % p
    else:
        m = (p2[1] - p1[1]) * modular_inverse(p2[0] - p1[0], p) % p
    x = (m**2 - p1[0] - p2[0]) % p
    y = (m * (p1[0] - x) - p1[1]) % p
    return (x, y)
```

4.2 Elliptic Curve Digital Signature Algorithm (ECDSA)

ECDSA is an analog of the Digital Signature Algorithm (DSA) using elliptic curve cryptography.

4.2.1 Key Generation

The ECDSA algorithm involves generating a key pair: a private key d and a public key $Q = d \cdot G$, where G is a generator point on the elliptic curve.

```
# ECDSA Key Generation
def generate_ecdsa_key(a, b, p, G, n):
    # Step 1: Generate a private key d (random number in [1, n-1])
```

```
    d = generate_private_key(n)

    # Step 2: Calculate the public key Q = d * G
    Q = scalar_multiply(G, d, a, p)

    return d, Q
```

4.2.2 Signing and Verification

To sign a message, the signer generates a random number k and calculates the signature (r, s).

```
# ECDSA Signing
def ecdsa_sign(message, d, a, p, G, n):
    # Step 1: Generate a random number k (in [1, n-1])
    k = generate_random_k(n)

    # Step 2: Calculate the point (x1, y1) = k * G
    x1, y1 = scalar_multiply(G, k, a, p)

    # Step 3: Calculate r = x1 mod n
    r = x1 % n

    # Step 4: Calculate s = k^(-1) * (hash(message) + d * r) mod n
    s = (modular_inverse(k, n) * (hash(message) + d * r)) % n

    return r, s
```

The verification process involves checking whether the signature (r, s) is valid for the given message and public key.

```
# ECDSA Verification
def ecdsa_verify(message, r, s, Q, a, p, G, n):
    # Step 1: Calculate w = s^(-1) mod n
    w = modular_inverse(s, n)

    # Step 2: Calculate the hash of the message
    e = hash(message)

    # Step 3: Calculate u1 = (e * w) mod n
    u1 = (e * w) % n

    # Step 4: Calculate u2 = (r * w) mod n
    u2 = (r * w) % n

    # Step 5: Calculate the point (x1, y1) = u1 * G + u2 * Q
    x1, y1 = ec_point_add(scalar_multiply(G, u1, a, p), scalar_multiply(Q, u2, a
```

```
    # Step 6: If r = x1 mod n, the signature is valid
    return r == x1 % n
```