Kriptoloji Notlar
(www.cyberdatascience.com.tr)

Alper Karaca

January 2024

Contents

1 Kriptoloji, Kriptografi, Kriptoanaliz

1.1 Kriptoloji

Kriptoloji, bilgi güvenliği ile ilgilenen bilim dalıdır. Kriptoloji, sadece bilgilerin gizlenmesini değil, aynı zamanda bu bilgilerin bütünşüğünü, kimlik doğrulamasını ve reddedilemezliğini de ele alır. Kriptoloji terimi, Yunanca "kryptos" (gizli) ve "logos" (bilim) kelimelerinden türetilmiştir. Kriptoloji, iki temel alt disiplini kapsar:

- **Kriptografi**: Verilerin gizliliğini sağlamak ve korumak için kullanılan yöntemleri inceleyen bilim dalıdır.
- **Kriptoanaliz**: Şifrelenmiş mesajların güvenliğini analiz eden ve bu mesajları kırmaya veya çözmeye çalışan bilim dalıdır.

1.2 Kriptografi

Kriptografi, verilerin şifrelenmesi ve şifre çözme yöntemlerini geliştirir. Bilgiyi yetkisiz erişime karşı korumak için matematiksel algoritmalar ve teknikler kullanır. Amacı, bir mesajı sadece belirli kişilerin anlayabileceği şekilde dönüştürmek (şifrelemek) ve daha sonra bu mesajın orijinal haline geri getirilmesini (şifre çözme) sağlamaktır.

- **Gizlilik (Confidentiality)**: Bilgilerin sadece yetkili kişilerce okunabilmesini sağlamak.
- **Bütünlük (Integrity)**: Bilgilerin iletim sırasında değiştirilmediğinden emin olmak.
- **Kimlik Doğrulama (Authentication)**: Mesajın kimden geldiğini doğrulamak.
- **Reddedilemezlik (Non-repudiation)**: Bir işlemi veya mesajı gönderen kişinin, bu işlemi gerçekleştirdiğini inkar edememesini sağlamak.

1.3 Kriptoanaliz

Kriptoanaliz, kriptografik sistemlerin güvenliğini test etme sürecidir. Kriptoanalistler, şifrelenmiş bilgileri çözmek veya zayıf yönleri bulmak için çalışırlar. Bu alan, güvenlik sistemlerinin ne kadar dayanıklı olduğunu test eder ve olası saldırılara karşı dayanıklılıklarını değerlendirir. Başarılı bir kriptoanaliz, şifrelenmiş bir mesajı şifreleme anahtarını bilmeden çözebilir.

2 Hash Functions (Özetleme Fonksiyonları)

Hash fonksiyonları, verileri belirli bir uzunlukta sabit bir çıktıya dönüştüren fonksiyonlardır. Bu çıktılar hash değeri veya hash kodu olarak adlandırılır. Verilerin değişmeden kaldığını doğrulamak için, veri tabanlarında hızlı arama ve veri eşleşmesini sağlamak için, parolaların güvenli bir şekilde saklanmasını sağlamak için, belirli bir verinin ya da mesajin kimliğini doğrulamak için, verilerin doğruluğunu ve bütünlüğünü korumak için kullanılır.

2.1 Types of Hash Functions

2.1.1 Division Method (Bölme Yöntemi)

Division Method, bir anahtarı belirli bir tam sayıya bölerek kalanını hesaplar ve bu kalan, hash değeri olarak kullanılır. Basit ve hızlıdır. Uygun bir mod seçilmediğinde çakışma oranı yüksektir. Bazı değerlerle belirli desenler oluşabilir.

$$h(k) = k \bmod m$$

Burada, k anahtar ve m bir mod değeridir.

2.1.2 Multiplication Method (Çarpma Yöntemi)

Bu yöntemde, anahtar, 0 ile 1 arasında bir sabit sayıyla çarpılır. Çarpım sonucunda kesirli kısım, bir mod değeriyle çarpılarak hash değeri elde edilir. Çakışma olasılığı düşüktür. Uygulaması, bölme yöntemine göre daha yavaştır.

$$h(k) = \lfloor m(kA \bmod 1) \rfloor$$

Burada, k anahtar, A sabit bir sayı (0 ile 1 arasında) ve m mod değeridir.

2.1.3 Mid-Square Method (Orta Kare Yöntemi)

Bu yöntemde, anahtar önce kendisiyle çarpılır ve ardından orta kısmı hash değeri olarak kullanılır. Anahtarın tüm basamaklarına duyarlıdır. Sabit sayı gerektirmez. Çakışma olasılığı yüksektir. Küçük anahtarlar için hash değeri iyi dağılmayabilir.

2.1.4 Folding Method (Katlama Yöntemi)

Anahtar, gruplara ayrılır (örneğin, dört basamaklı sayılar halinde). Bu gruplar toplanır ve bu toplam hash değeri olarak kullanılır. Dağınık verilerde bile oldukça iyi çalışır. Sayılar gruplara ayrıldığından, anahtarın tüm bölümlerini dikkate alır. Çakışma olasılığı yüksektir.

2.1.5 Cryptographic Hash Functions (Kriptografik Hash Fonksiyonları)

Kriptografik hash fonksiyonları, belirli güvenlik gereksinimlerini karşılaşmak için tasarlanmış hash fonksiyonlarıdır. Bunlar tek yönlü fonksiyonlardır, yani hash değerinden orijinal veriyi elde etmek imkansızdır. Parola saklama, dijital imza ve veri bütünlüğü doğrulama gibi alanlarda yaygın olarak kullanılır. Örneğin, MD5, SHA-1, SHA-256. Fakat MD5 ve SHA-1 gibi eski algoritmalar artık güvensiz olarak kabul edilmektedir.

2.1.6 Universal Hashing (Evrensel Hashleme)

Evrensel hashleme, birden çok hash fonksiyonu ailesi arasından rastgele bir fonksiyon seçilerek çakışma olasılığını minimize etmeyi amaçlar. Her bir giriş için farklı bir hash fonksiyonu kullanılır. Çakışma olasılığı son derece düşüktür. Güvenlik açısından avantajlar sunar.

2.1.7 Perfect Hashing (Mükemmel Hashleme)

Mükemmel hashleme, çakışma olmadan verilerin hashlenmesini sağlar. Yani, her anahtar benzersiz bir hash değerine sahiptir. Hiçbir çakışma olmaması, arama işlemlerini hızlandırır. Veri yapılarında verimli alan kullanımı sağlar.

2.2 Cyclic Redundancy Check (CRC)

CRC, veri iletimi sırasında ortaya çıkabilecek hataları tespit etmek için kullanılır. Belirli bir uzunluktaki veri bloklarının bir matematiksel özetini üretir ve bu özet, verinin bir hata içerip içermediğini kontrol etmek için kullanılır. Ağ iletişim protokollerinde ve dosya bütünlüğünü kontrol etmek için yaygın olarak kullanılır.

- **CRC-8**: 8 bitlik bir CRC kodu üretir.
- **CRC-16**: 16 bitlik bir CRC kodu üretir.
- **CRC-32**: 32 bitlik bir CRC kodu üretir. Ethernet ve ZIP dosyalarında kullanılır.
- **CRC-64**: 64 bitlik bir CRC kodu üretir.

2.2.1 Çalışma Adımları

1. Veri, belirli bir şekilde bit dizisi olarak temsil edilir.
2. CRC hesaplaması, belirli bir polinomla modüler bölme işlemi kullanılarak yapılır. Bu polinomlar belirli CRC standartlarına karşılık gelir.
3. Verinin sonuna CRC bitlerinin yerleştirileceği kadar 0 eklenir. Örneğin, CRC-16 için 16 sıfır eklenir.
4. Veriyi genişletildikten sonra, seçilen polinomla mod 2 bölme işlemi yapılır, yani bitler üzerinde xor işlemi uygulanır. Bu işlem sonucunda bir kalan elde edilir.
5. Son kalan (CRC Kodu), verinin sonuna eklenir. Bu eklenen bitler, veri ile birlikte gönderilir.
6. Alıcı, bu bitleri kontrol ederek hata tespiti yapar.

2.2.2 Python Kodu

```
import binascii
data = b"Hello, World!"
mask32 = 0xffffffff
crc_value = binascii.crc32(data) & mask32
print(f"CRC-32 Value (Hex): f{crc_value:#010x}")
print(f"CRC-32 Value (Dec): f{crc_value}")
```

2.3 Fletcher

Fletcher, veri doğrulama ve hata tespiti için kullanılır. CRC'den daha hızlı çalışır. Fletcher algoritması, iki ayrı toplama işlemi gerçekleştirerek verinin özetini oluşturur. CRC'nin aksine, bit seviyesinde değil, byte seviyesinde çalışır.

- **Fletcher-4**: 4 bitlik özet üretir.
- **Fletcher-8**: 8 bitlik özet üretir.
- **Fletcher-16**: 16 bitlik özet üretir.
- **Fletcher-32**: 32 bitlik özet üretir.
- **Fletcher-64**: 64 bitlik özet üretir.

2.3.1 Çalışma Adımları

1. Veri, byte dizisi (8-bit parçalar) olarak alınır.
2. İki toplam değeri başlatılır. Bu iki toplam, verinin tüm elemanları üzerinden iteratif bir şekilde güncellenir.
3. Veri, byte'ler halinde okunur ve her bir byte için: $\text{sum1} = (\text{sum1} + \text{byte}) \bmod 255$ ve $\text{sum2} = (\text{sum2} + \text{sum1}) \bmod 255$.
4. Hesaplama tamamlandıktan sonra, iki toplam değeri birleştirilir. Fletcher-16 için $\text{checksum} = (\text{sum2} << 8) | \text{sum1}$.

2.3.2 Python Kodu

```python
def fletcher16(data: bytes) -> int:
    sum1 = sum2 = 0
    for byte in data:
        sum1 = (sum1 + byte) % 255
        sum2 = (sum2 + sum1) % 255

    checksum = (sum2 << 8) | sum1
    return checksum

data = b"Hello, World!"
checksum = fletcher16(data)
print(f"Fletcher-16 Value: {checksum:#06x}")
```

2.4 Adler-32

Adler-32, CRC'ye benzer, veri doğrulama ve hata tespiti için kullanılır. 1995 yılında Mark Adler tarafından geliştirilmiştir. Adler-32, 32 bit uzunluğunda bir özet üretir ve Fletcher'a dayanır ancak bazı iyileştirmeler içerir. Zlib sıkıştırma algoritmasında kullanılır.

2.4.1 Çalışma Adımları

1. İki toplam değeri başlatılır: $A = 1$ ve $B = 0$.
2. Veri, byte'ler halinde okunur ve her byte için: $A = (A + \text{byte}) \bmod 65521$ (65521, Adler-32 için bir asal sayıdır.) ve $B = (B + A) \bmod 65521$. A, her byte'ı ekleyerek güncellenirken, B ise A değerini sürekli toplar.
3. Hesaplama tamamlandıktan sonra, A ve B değerleri birleştirilir: $\text{Adler32} = (B << 16) | A$.

2.4.2 Python Kodu

```python
def adler32(data: bytes) -> int:
    MOD_ADLER = 65521
    A = 1
    B = 0
    for byte in data:
        A = (A + byte) % MOD_ADLER
        B = (B + A) % MOD_ADLER

    checksum = (B << 16) | A
    return checksum

data = b"Hello, World!"
checksum = adler32(data)
print(f"Adler-32 Value: {checksum:#010x}")
```

```python
import zlib
data = b"Hello, World!"
checksum_zlib = zlib.adler32(data)
print(f"Adler-32 Value (Zlib): {checksum_zlib:#010x}")
```

2.5 XOR8

XOR8, 8 bitlik bir özet oluşturur. Verinin her bir byte'ı üzerinde XOR işlemi yaparak bir özet değeri oluşturur. Hata tespiti için kullanılır.

2.5.1 Çalışma Adımları

1. Bir başlangıç değeri seçilir.
2. Veri, byte'ler halinde okunur ve her byte için: $\text{checksum} = \text{checksum}^{\text{byte}}$. Her iki bitin aynı olduğu durumlarda 0, farklı olduğu durumlarda 1 verir.

2.5.2 Python Kodu

```python
def xor8(data: bytes) -> int:
    checksum = 0
    for byte in data:
        checksum ^= byte

    return checksum

data = b"Hello, World!"
checksum = xor8(data)
print(f"XOR8 Value: {checksum:#04x}")
```

2.6 Luhn Algorithm

Luhn Algoritması, bir doğrulama algoritmasıdır. Kredi kartı numaraları, IMEI numarları gibi kimlik belirleyici sayıların doğruluğunu kontrol etmek için kullanılır. Algoritma, bir sayının geçerli olup olmadığını belirlemek için basamaklar üzerinde matemtiksel işlemler yapar. Luhn Algoritması bit uzunluğuna göre sınırlanmış bir algoritma değildir; bunun yerine sayıların doğruluğunu kontrol eder.

2.6.1 Çalışma Adımları

1. Sayı ters çevrilir.
2. Ters çevrilen sayıda, 0'dan başlayarak çift indeksteki sayılar iki ile çarpılır. Eğer iki ile çarpım sonucu elde edilen değer 9'dan büyükse; elde edilen değerin rakamları toplamı yazılır.
3. Tüm sayılar toplanır.
4. Eğer sonuç'un 10'a modu 0 ise geçerlidir. Aksi halde geçerli değildir.

2.6.2 Python Kodu

```python
def luhn_algorithm(card_number: str) -> bool:
    total_sum = 0
    reverse_digits = card_number[::-1]
    for i, digit in enumerate(reverse_digits):
        n = int(digit)
        if i % 2 == 1:
            n *= 2
            if n > 9:
                n -= 9

        total_sum += n

    return total_sum % 10 == 0

card_number = ""
is_valid = luhn_algorithm(card_number)
print(f"Is Valid: {is_valid}")
```

2.7 Verhoeff Algorithm

Verhoeff algoritması, doğrulama işlemleri için kullanılan hata tespit algoritmasıdır. Bu algoritma, sayılar üzerindeki küçük hataları tespit edebilmek için tasarlanmıştır. 10 basamaktan oluşan bir doğrulama kodu üretir ve her bir basamağı 0-9 arası bir değer alır. D-algorithm adı verilen matematiksel bir yapı kullanarak çalışır.

2.7.1 Çalışma Adımları

1. Algoritma, verilen sayıyı soldan sağa doğru işler, yani sayı ters çevrilir.
2. Algoritmanın çalışma mekanizması için belirli bir çarpanlar ve diziler kullanılır. Bu diziler, D ve P tablosu olarak bilinir. D tablosu her basamağa bir işlem değeri atar. P tablosu ise sayının her basamağı için bir modül değeri sağlar.
3. Sayı soldan sağa doğru işlenir. Başlangıç değeri sıfırdır. Her bir basamağa karşılık gelen çarpanlar ve modül hesaplamaları yapılır. Bu işlem sonucu, son basamağa kadar devam edilir.
4. Eğer sonuç sıfır ise, sayı geçerli kabul edilir.

2.7.2 Python Kodu

```
D = [
    [0, 1, 2, 3, 4, 5, 6, 7, 8, 9],
    [1, 2, 3, 4, 5, 6, 7, 8, 9, 0],
    [2, 3, 4, 5, 6, 7, 8, 9, 0, 1],
    [3, 4, 5, 6, 7, 8, 9, 0, 1, 2],
    [4, 5, 6, 7, 8, 9, 0, 1, 2, 3],
    [5, 6, 7, 8, 9, 0, 1, 2, 3, 4],
    [6, 7, 8, 9, 0, 1, 2, 3, 4, 5],
    [7, 8, 9, 0, 1, 2, 3, 4, 5, 6],
    [8, 9, 0, 1, 2, 3, 4, 5, 6, 7],
    [9, 0, 1, 2, 3, 4, 5, 6, 7, 8]
]

P = [
    [0, 1, 2, 3, 4, 5, 6, 7, 8, 9],
    [1, 2, 3, 4, 5, 6, 7, 8, 9, 0],
    [2, 3, 4, 5, 6, 7, 8, 9, 0, 1],
    [3, 4, 5, 6, 7, 8, 9, 0, 1, 2],
    [4, 5, 6, 7, 8, 9, 0, 1, 2, 3],
    [5, 6, 7, 8, 9, 0, 1, 2, 3, 4],
    [6, 7, 8, 9, 0, 1, 2, 3, 4, 5],
    [7, 8, 9, 0, 1, 2, 3, 4, 5, 6],
    [8, 9, 0, 1, 2, 3, 4, 5, 6, 7],
```

```
    [9, 0, 1, 2, 3, 4, 5, 6, 7, 8]
]

def verhoeff_algorithm(number: str) -> bool:
    number = number[::-1]
    c = 0
    for i in range(len(number)):
        c = D[c][P[(i + 1) % 8][int(number[i])]]

    return c == 0

number = "1256849376"
is_valid = verhoeff_algorithm(number)
print(f"Is Valid: {is_valid}")
```

2.8 Damm Algorithm

Damm algoritması, veri doğrulama ve hata tespiti için kullanılır. Bir sayıya ek bir kontrol basamağı ekler ve bu kontrol basamağı, sayının geçerli olup olmadığını doğrulamak için kullanılır. Modüler aritmetik ve tablolama kullanarak çalışır.

2.8.1 Çalışma Adımları

1. Algoritmanın temelini, 10x10 boyutunda bir doğrulama tablosu oluşturur. Bu tablo, her bir sayı için bir diğer sayı ile yapılan işlemi gösterir.
2. Verilen sayıya ek olarak bir kontrol basamağı hesaplanır. Bu işlem, sayının her bir basamağı için tablodaki uygun değeri kullanarak yapılır.
3. Verilen sayının sonuna eklenen kontrol basamağı, sayının geçerliliğini kontrol eder. Eğer sayı doğru ise, kontrol basamağı ve hesaplanan sonuç sıfır olur.

2.8.2 Python Kodu

```
D = [
    [0, 1, 2, 3, 4, 5, 6, 7, 8, 9],
    [1, 2, 3, 4, 5, 6, 7, 8, 9, 0],
    [2, 3, 4, 5, 6, 7, 8, 9, 0, 1],
    [3, 4, 5, 6, 7, 8, 9, 0, 1, 2],
    [4, 5, 6, 7, 8, 9, 0, 1, 2, 3],
    [5, 6, 7, 8, 9, 0, 1, 2, 3, 4],
    [6, 7, 8, 9, 0, 1, 2, 3, 4, 5],
    [7, 8, 9, 0, 1, 2, 3, 4, 5, 6],
    [8, 9, 0, 1, 2, 3, 4, 5, 6, 7],
    [9, 0, 1, 2, 3, 4, 5, 6, 7, 8]
]

def damm_algorithm(number: str) -> bool:
    number = number[::-1]
    c = 0
    for i in range(len(number)):
        c = D[c][int(number[i])]

    return c == 0

number = "1357"
is_valid = verhoeff_algorithm(number)
print(f"Is Valid: {is_valid}")
```

2.9 Rabin Fingerprint

Michael O. Rabin tarafından geliştirilmiştir. Veri bloklarının hızlı bir şekilde karşılaştırılması ve tespit edilmesi için kullanılır. Algoritma, bir polinomin veriye karşı modüler aritmetik işlemi ile hesaplanır.

2.9.1 Çalışma Adımları

1. İlk adım, veriyi bir polinom şeklinde temsil etmektir. Her bir veri parçası, bir polinomun katsayısı olarak kabul edilir.
2. Veriye uygulanan polinom, belirlenen bir sabit polinom ile mod alınır. Bu mod işlemi, sonuç polinomunun sabit uzunlukta kalmasını sağlar.
3. Mod işlemi sonrası kalan değer, verinin özetini (fingerprint) verir.

2.9.2 Python Kodu

```python
def rabin_fingerprint(data: bytes, poly=0x000000000000001b) -> int:
    fp = 0
    for byte in data:
        value = (fp >> 56) ^ byte
        for i in range(8):
            if value & (1 << (63 - i)):
                value ^= poly << (7 - i)

        fp = (fp << 8) ^ value

    return fp

data = b"Hello, World!"
fingerprint = rabin_fingerprint(data)
print(f"Rabin Fingerprint: {fingerprint:#x}")
```

2.10 Tabulation Hashing

Tabulation Hashing, bir dizi tablo kullanarak veriyi küçük alt parçalara böler ve bu parçalar için tabloya dayalı rastgele değerler belirler. Yöntem, birden fazla tablo kullanarak rastgele tablolarla veriyi karıştırır, bu sayede düşük çakışma oranına sahip bir özet elde eder.

2.10.1 Çalışma Adımları

1. İlk adımda, belirli boyutta bir tablo rastgele sayılarla doldurulur. Her bir tabloa, veri parçalarının hash işlemi için kullanılacak rastgele değerlerini içerir.
2. Hashlenecek veri, belirli birimlere bölünür.
3. Her veri parçası, farklı tablolardan karşılık gelen rastgele değerlerle eşleştirilir. Bu eşleşen rastgele değerler, bit düzeyinde toplanarak birleştirilir.
4. Tüm veri parçalarının tablo değerleriyle işlenmesinden sonra, elde edilen sonuç hash değeri olur.

2.10.2 Python Kodu

```python
import random
def tabulation_hashing(data: bytes) -> int:
    table = [[random.getrandbits(32) for _ in range(256)]
                                     for _ in range(4)]
    hash_value = 0
    for i, byte in enumerate(data):
        hash_value ^= table[i % 4][byte]

    return hash_value
data = b"Hello, World!"
hash_value = tabulation_hashing(data)
print(f"Tabulation Hashing: {hash_value:#x}")
```

2.11 Zobrist Hashing

Zobrist Hashing, oyun tahtaları veya tablolar gibi çok boyutlu veri yapılarını hızlıca karşılaştırmak ve hash değerini almak için geliştirilmiştir. 1969 yılında Albert Zobrist tarafından geliştirilmiştir. Satranç, Go gibi oyunlarda tahtadaki mevcut pozisyonun temsil edilmesinde kullanılır. Her oyun tahtası karesi ve her oyun taşı için rastgele bir değer belirler. Her taş tahtaya yerleştirildiğinde veya kaldırıldığında bu rastgele değerler XOR işlemi ile hash hesaplamasına dahil edilir. Bu sayede tahtadaki dürüm sürekli olark güncellenerek hash değeri yeniden hesaplanabilir.

2.11.1 Çalışma Adımları

1. Her oyun tahtası karesi ve her bir oyun taşı tipine rastgele bir 64-bit veya 128-bit değer atanır. Eğer bir oyun taşının tahtada olup olmaması durumu da hesaba katılıyorsa, her taş için var-yok durumu da rastgele bir değerle temsil edilir.
2. Boş tahtanın hash değeri sıfırdır, çünkü herhangi bir taş yoktur ve XOR işlemleri sonucunda değişiklik yoktur.
3. Tahta üzerinde bir taş yerleştirildiğinde, ilgili karenin rastgele değeri mevcut hash değerine XOR ile eklenir.
4. Aynı taş kaldırıldığında tekrar XOR yapılarak, bu taşın etkisi hash değerinden çıkarılır. XOR işlemi aynı değeri iki kez eklediğinde sıfırlar.
5. Her oyun hamlesiyle birlikte taşların hareketiyle hash değeri de güncellenir. Yani bir taş eklendiğinde veya bir taş çıkarıldığında, o taşın temsil ettiği değer XOR ile hash'e eklenir veya çıkarılır.

2.11.2 Python Kodu

```python
import random

pieces = ["pawn", "knight", "bishop", "rook", "queen", "king"]
colors = ["white", "black"]

zobrist_table = {}
for piece in pieces:
    for color in colors:
        for row in range(8):
            for col in range(8):
                zobrist_table[(piece, color, row, col)] =
                    random.getrandbits(64)

board = [[0 for _ in range(8)] for _ in range(8)]
board[6][0] = ("pawn", "white")
```

```
board[1][0] = ("pawn", "black")
board[7][1] = ("knight", "white")

def zobrist_hashing(board, zobrist_table):
    hash_value = 0
    for row in range(8):
        for col in range(8):
            if board[row][col] != 0:
                piece, color = board[row][col]
                hash_value ^= zobrist_table[(piece, color, row, col)]

    return hash_value

hash_value = zobrist_hashing(board, zobrist_table)
print(f"Zobrist Hash: {hash_value:#x}")
```

2.12 Pearson Hashing

Pearson hashing, 8-bitlik bir hash fonksiyonudur. İlk olarak 1990 yılında Peter K. Pearson tarafından önerilen bu algoritma, küçük hash tabloları oluşturmak ve dizeleri hızlı bir şekilde özetlemek için kullanılır.

2.12.1 Çalışma Adımları

1. Pearson hashing, 256 elemanlı bir rastgele tablo kullanır. Bu tablo sabit olmalı ve her eleman, 0 ile 255 arasında bir değere sahip olmalıdır. Tablo, algoritmanın temelidir ve her baytın hash değerini belirlemede kullanılır.
2. Hash değeri başlangıçta sıfır olarak başlar. Algoritma boyunca bu değer her bayta göre güncellenir.
3. Hashlenecek veri bayt bayt işlenir. Her bayt için, hash değeri ilgili bayt ile XOR işlemine girer. Ortaya çıkan değer, 256 elemanlı tabloda indeks olarak kullanılır ve tabloda o indeksin karşılığı olan değer hash'e atanır.

2.12.2 Python Kodu

```
import numpy as np
pearson_table = np.arange(256)
np.random.shuffle(pearson_table)

def pearson_hash(data):
    data = data.encode("utf-8")
    hash_value = 0
    for byte in data:
        hash_value = pearson_table[hash_value ^ byte]
    return hash_value

data = "Hello, World!"
hash_result = pearson_hash(data)
print(f"Pearson Hash: {hash_result}")
```

2.13 Bernstein's Hash (DJB2)

Daniel J. Bernstein tarafından geliştirilmiştir. Metin verileri üzerinde hash hesaplamak için kullanılır.

2.13.1 Çalışma Adımları

1. Hash değeri başlangıçta 5381 olarak belirlenir.
2. Hashlenecek veri byte'ler halinde işlenir. Her bayt için; hash değeri, önce 33 ile çarpılır. Sonra, ilgili baytın ASCII değeri ile toplanır.
3. Verinin tüm baytları işlendikten sonra, 32-bit uzunluğunda bir hash elde edilir.

2.13.2 Python Kodu

```
def djb2_hash(data: str) -> int:
    hash_value = 5381
    for char in data:
        hash_value = (hash_value * 33) + ord(char)

    mask = 0XFFFFFFFF
    checksum = hash_value & mask
    return checksum

data = "Hello, World!"
hash_result = djb2_hash(data)
print(f"DJB2 Hash: {hash_result}")
```

2.14 Elf Hash

Elf Hash, bazı işletim sistemlerinde ve derleyicilerde kullanılan bir özetleme fonksiyonudur. Sembolik adlar veya dizeler için kullanılır. Bu algoritma, veri seti içerisindeki verileri hızlı bir şekilde işleyip hash tablolarında kullanılabilir hale getirmeyi amaçlar. 32-bit uzunluğunda bir hash değeri üretir.

2.14.1 Çalışma Adımları

1. Hash değeri başlangıçta sıfırdır.
2. Her bayt için hash değeri güncellenir. Öncelikle hash değeri, 4 bit kaydırılarak güncellenir, yani 16 ile çarpılır. Hash'e bayt değeri eklenir. Hash'in en üst 4'biti (28-31. bitler) maskelenir. Bu en üst 4 bit sıfırlanır ve gerekli durumlarda geri kalan hash değeriyle XOR işlemi yapılır.
3. Verinin tüm baytları işlendikten sonra, son hash değeri döndürülür.

2.14.2 Python Kodu

```python
def elf_hash(data: str) -> int:
    hash_value = 0
    for char in data:
        hash_value = (hash_value << 4) + ord(char)
        high_bits = hash_value & 0xF0000000

        if high_bits != 0:
            hash_value ^= (high_bits >> 24)

        hash_value &= ~high_bits

    mask = 0xFFFFFFFF
    checksum = hash_value & mask
    return checksum

data = "Hello, World!"
hash_result = elf_hash(data)
print(f"Elf Hash: {hash_value}")
```

2.15 Murmur Hash

Murmur Hash, kısmi olarak rastgeleleştirilmiş bir özet fonksiyonudur. Adını, bitlerin "homurdanması" olarak ifade edilen bir algoritmadan alır. Hsah tabloları, veri yapıları ve veritabanları gibi veri yönetim sistemlerinde yaygın olarak kullanılır. Deterministik bir fonksiyondur, yani aynı giriş için her zaman aynı hash değerini üretir. Ancak bazı varyasyonlarında rastgele bir başlangıç değeri (seed) kullanılarak daha güvenli hale getirilmiştir.

2.15.1 Çalışma Adımları

1. Girdi verisi sabit büyüklükte bloklara ayrılır. Eğer giriş verisinin uzunluğu bu blok boyutunun tam katı değilse, son blokta kalan veriler uygun şekilde doldurulur.
2. Her blok, sabit sayılarla çarpılır ve bit kaydırma işlemleriyle karşılaştırılır. Veriler üzerinde modüler aritmetik işlemler uygulanır.
3. Tüm bloklar işlendiğinde, kalan veriler eklenir ve son bir karıştırma işlemi yapılır. Bu aşama, küçük boyutlu girdilerde bile hash'in iyi dağılması sağlanır.

2.15.2 Python Kodu

```
import mmh3
data = "Hello, World!"
hash_value = mmh3.hash(data)
print(f"MurmurHash3 (32-bit): {hash_value}")
```

2.16 BLAKE2

BLAKE2, modern kriptografik özetleme fonksiyonudur. Bu algoritma, mesajların özetini hesaplar ve dijital imza, kimlik doğrulama gibi uygulamalarda kullanır. BLAKE2b, 64-bit platformlar için optimize edilmiştir ve 512-bit'e kadar özetler üretebilir. BLAKE2s, 32-bit platformlar için optimize edilmiştir ve 256-bit'e kadar özetler üretebilir.

2.16.1 Çalışma Adımları

1. BLAKE algoritmalarında, başlangıç vektörleri belirli sabit değerler olarak tanımlanır.
2. Girdi verisi sabit büyüklükte bloklara ayrılır. Her blok, belirli sayısal işlemlerle ve karıştırma fonksiyonlarıyla işlenir.
3. Veriler, sabit sayılarla çapılır ve modüler aritmetik işlemler uygulanarak karıştırılır. Bu işlem, verinin hash değerine düzgün ve güvenli bir şekilde dağılmasını sağlar.
4. Tüm bloklar işlendiğinde, son bir karıştırma işlemi yapılır ve hash fonksiyonunun ürettiği sabit bit uzunluğundaki özet değer elde edilir.

2.16.2 Python Kodu

```
import hashlib
data = b"Hello, World!"
blake2b_hash = hashlib.blake2b(data).hexdigest()
blake2s_hash = hashlib.blake2s(data).hexdigest()
print(f"BLAKE2b hash: {blake2b_hash}")
print(f"BLAKE2s hash: {blake2s_hash}")
```

2.17 HMAC (Hash-based Message Authentication Code)

HMAC (Hash-based Message Authentication Code), bir mesajın doğruluğunu ve bütünlüğünü garanti etmek için kullanılan bir özetleme fonksiyonudur. HMAC, bir kriptografik hash fonksiyonu ve bir gizli anahtar kullanarak mesajların doğrulanmasını sağlar. Bu, hem mesajın değiştirilip değiştirilmediğini kontrol eder hem de mesajın bir kaynaktan geldiğini doğrular. Anahtar, gizli bir anahtardır ve sadece iki taraf arasında paylaşılır. Mesajın doğrulaması bu anahtarla yapılır. HMAC, ağ güvenliği protokollerinde (TLS, IPsec) kullanılır.

2.17.1 Çalışma Adımları

1. Anahtar, hash fonksiyonunun blok boyutundan uzun ise, anahtar bir kez hash edilir ve daha kısa hale getirilir. Anahtar, hash fonksiyonunun blok boyutundan kısa ise, eksik kısımlar sıfırla doldurulur.
2. İki sabit byte dizisi kullanılır: ipad (iç dolgu) ve opad (dış dolgu). Anahtar ile ipad ve opad byte'leri XOR işlemi ile birleştirilir.
3. Mesaj, anahtar ve ipad ile birleştirilip bir kriptografik hash fonksiyonu ile hash edilir (iç hash).
4. İç hash sonucu, anahtar ve opad ile birleştirilir ve tekrar hash fonksiyonuna sokulur.
5. Elde edilen hash değeri, mesajın doğrulama kodudur.

2.17.2 Python Kodu

```
import hmac
import hashlib
message = b"Hello, World!"
key = b"secretkey"
hmac_hash = hmac.new(key, message, hashlib.sha256).hexdigest()
print(f"HMAC (SHA-256): {hmac_hash}")
```

2.18 Keccak (Keccak Message Authentication Code)

KMAC, Keccak (SHA-3) tabanlı bir mesaj doğrulama kodudur. SHA-3'ün kriptografik sünger fonksiyonlarını (sponge function) kullanarak bir mesajın bütünlüğünü ve doğruluğunu kontrol eder. KMAC, hem gizli bir anahtarı hem de bir veri girdisini kullanarak bir özet (hash) oluşturur. KMAC'in temel çalışma prensibi, Keccak sünger fonksiyonunun sağladığı esneklik ve güçlü güvenlik özelliklerini kullanarak hem uzunluk genişletme saldırılarına karşı güvenlik sağlar hem de esnek parametreler sunar.

2.18.1 Python Kodu

```
from Crypto.Hash import KMAC128
message = b"Hello, World!"
key = b"secretkey" * 8
kmac = KMAC128.new(key=key, mac_len=32)
kmac.update(message)
kmac_hash = kmac.hexdigest()
print(f"KMAC (256-bit): {kmac_hash}")
```

2.19 ECOH (Elliptic Curve Only Hash)

ECOH, kriptografik hash fonksiyonları için güçlü bir güvenlik seviyesi sağlamak amacıyla eliptik eğrilerden faydalanır. Eliptik eğriler, daha küçük anahtar boyutlarında yüksek güvenlik sağlayarak verimli kriptografik işlemler sunar.

2.19.1 Çalışma Adımları

1. ECOH, belirli bir eliptik eğri üzerinde çalışır. Bu eğri, mesajın matematiksel olarak işlenmesini sağlar.
2. Girdi mesajı, belirli boyutlardaki bloklara bölünür.
3. Mesaj blokları, seçilen eliptik eğri üzerinde matemtiksel işlemlerden geçer. Mesaj blokları, eğri üzerindeki noktalarla temsil edilir.
4. Bu eğri noktaları üzerinde işlemler devam eder ve bu noktalar birleştirilir.
5. Eliptik eğri üzerindeki noktalar işlenerek hash fonksiyonu tamamlanır.

2.20 Fast Syndrome-based Hash Function (FSB)

FSB, kuantum bilgisayarlara dayanıklı bir yapı sunmak için tasarlanmıştır. Doğrusal kodlar ve sendromlar üzerine kurulu olup, kriptografik özet üretirken matematiksel olarak güvenli bir temel sağlar. FSB algoritmasının temeli, doğrusal kodlama teorisindeki "sendrom" kavramına dayanır. FSB, bir doğrusal kodlama matrisiyle çalışır ve mesajın bu matris ile işlenmesi sonucunda bir özet (hash) üretir.

2.20.1 Çalışma Adımları

1. Veriyi işlemek için bir doğrusal kodlama matrisi seçilir.
2. Girdi mesajı, binary (ikili) bir bit dizisine dönüştürülür.
3. Mesaj bitleri, kodlama matrisi ile çarpılır. Bu çarpım sonucunda "sendrom" adı verilen bir bit dizisi elde edilir. Sendrom, matris ve mesajın ilişkisini tanımlayan bir çıktıdır.
4. Sendrom, belirli matematiksel işlemlerden geçirilerek nihai hash sonucu elde edilir.

2.20.2 Python Kodu

```
import numpy as np

matrix = np.array([
    [1, 0, 1, 1],
    [0, 1, 1, 0],
    [1, 1, 0, 1],
    [0, 0, 1, 1]
])

message = np.array([1, 0, 1, 0])
syndrome = np.dot(matrix, message) % 2
hash_value = ''.join(str(x) for x in syndrome)
print(f"Sendrom (Syndrome): {syndrome}")
print(f"FSB Hash: {hash_value}")
```

2.21 GOST

GOST, Rusya tarafından geliştirilmiştir. Belirli adımların sonucunda bir veri parçası için 256-bitlik bir özet üretir. Bu adımlar, girdiyi parçalar ve her bir parça üzerinde karmaşık bit manipülasyonları yaparak sonucu hesaplar.

2.21.1 Çalışma Adımları

1. Girdi veri uzunluğu sınırlandırılmaksızın alınır, fakat veri 256-bit bloklara bölünür. Girdi verisi birden fazla blok içeriyorsa, bloklar halinde işlenir. Bir başlangıç durumu belirlenir. Bu durum, ilk başta tüm sıfırlardan oluşan bir vektördür.

2. Girdi verisi 256-bit (32-byte) bloklar halinde işlenir. Eğer son blok tam olarak 256-bit değilse, boş kalan alanlar doldurulur.

3. Algoritma, veri üzerinde 32 turdan oluşan bit-manipülasyonları yapar. Her turda çeşitli aritmetik ve mantıksal işlemler yapılır. Bu işlemler sırasında kullanılan bir S-Box tablosu bulunur. Bu tablo, her bir giriş için karmaşık ve önceden tanımlanmış çıkış değerlerini sağlar.

4. Bütün veriler işlendiğinde, 256-bitlik (32-byte) nihai özet değeri elde edilir.

2.22 SipHash

SipHash, anahtarlı hash fonksiyonu olarak tasarlanmıştır. Kısa veri parçalarını güvenli bir şekilde özetlemek için geliştirilmiştir. DoS (Denial of Service) saldırılarına karşı korunmak için tasarlanmıştır. Temel özelliği, anahtarlı olmasıdır, yani güvenli bir hash hesaplaması yapabilmek için iki anahtar kullanılır. Bu, fonksiyonun rastgele veri ile üretilen sahte özetlere karşı korumalı olmasını sağlar. Aynı veriye her seferinde aynı anahtar ile hash hesaplandığında aynı sonuç elde edilir, ancak anahtar değiştirilirse hash sonucu da değişir.

2.22.1 Çalışma Adımları

1. SipHash, 128-bitlik bir gizli anahtar kullanır.
2. Girdi verisi parçalara ayrılır ve her bir parça, 64-bitlik bloklar halinde işlenir. Eğer veri son bloğa sığmazsa, boş kalan kısımlar sıfırla doldurulur.
3. SsipHash, iç durumda 4 adet 64-bitlik başlangıç değerini tutar. Bu durum, daha sonra hash hesaplaması sırasında güncellenir. Başlangıç durumu, gizli anahtara dayanarak oluşturulur.
4. Her veri bloğu, dört farklı turda işlenir. Her bir tur, veri ve anahtar üzerinde çeşitli bit manipülasyonları yapar. Her turda belirli sayıda döndürme işlemi gerçekleştirilir. SipHash-2-4, bu işlemi 2 iç ve 4 dış turla yapar.
5. Bütün veri işlendiğinde, 64-bit veya 128-bit (versiyona bağlı olarak) nihai hash sonucu üretilir.

2.23 Grostl

Grostl, 2008 yılında geliştirilen bir kriptografik özetleme (hash) fonksiyonu olup, NIST'in SHA-3 yarışmasında finalist olmuştur. İki temel bileşenden oluşur: P ve Q fonksiyonları. Her iki fonksiyon da AES benzeri blok şifreleme prensiplerine dayanır ve iç içe geçmiştir.

2.23.1 Çalışma Adımları

1. Hash işlemi başlamadan önce, algoritma, başlangıç değeri olarak sabit bir bit dizisi kullanır. Bu başlangıç değeri hash uzunluğuna göre değişir.
2. Girdi verisi belirli boyutlardaki bloklara bölünür. Grostl'de blok boyutu kullanılan hash fonksiyonunun uzunluğuna bağlıdır.
3. Her bir blok, AES benzeri P ve Q fonksiyonlarıyla işlenir. P ve Q fonksiyonları, çeşitli matris işlemleri ve bit manipülasyonları yapar: P fonksiyonu veriyi bir kez şifreler. Q fonksiyonu veriyi ters bir sırayla şifreler.
4. Son blok işlendikten sonra, verinin uzunluğu da son bloğa eklenir. Bu, özetin doğruluğunu sağlamak için kullanılır.
5. Tüm bloklar işlendiğinde, algoritma nihai hash (özet) sonucunu üretir.

2.23.2 Python Kodu

```
from Crypto.Hash import SHA3_256
data = b'Hello, World!'
hash_obj = SHA3_256.new()
hash_obj.update(data)
hash_value = hash_obj.hexdigest()
print(f"Grostl-256: {hash_value}")
```

2.24 HAVAL (Hash of Variable Length)

1992 yılında Peter Y. Yin, Zhengjun Yin ve Yuliang Zheng tarafından geliştirilmiştir. HAVAL, esnekliği ile öne çıkar ve değişken uzunluklu özetler üretir. Kullanıcı, hem özetin uzunluğunu hem de işlem turlarının sayısını seçebilir.

2.24.1 Çalışma Adımları

1. Algoritma, başlangıçta belirli sabit bir bit dizisi ile başlar. Bu başlangıç değeri, işlenecek verinin boyutuna ve tur sayısına göre değişmez.

2. Girdi verisi, 1024 bitlik (128 byte) bloklar halinde bölünür. Eğer son blok 1024 bitten kısa ise veri, belirli bir dolgu ile tamamlanır.

3. HAVAL, seçilen tur sayısına göre (3, 4 veya 5 tur) veri üzerinde karmaşık işlemler yapar. Her turda, veriyi manipüle eden matematiksel fonksiyonlar kullanılır. Her turda beş farklı komut kullanılır: F1, F2, F3, F4, ve F5. Bu fonksiyonlar, veriyi farklı yöntemlerle işleyip bir sonraki tura hazırlar.

4. Son blok işlendiğinde, verinin toplam uzunluğu da dikkate alınır ve özet sonuç hesaplanır.

2.25 JH Hash

Joan Daemen ve Gilles Van Assche tarafından geliştirilmiştir. JH, geniş bir iç durum ve sıkıştırma fonksiyonu tasarımına dayanır.

2.25.1 Çalışma Adımları

1. Hash fonksiyonu, başlangıçta belirlenmiş sabit bir değer ile başlar. Bu başlangıç durumu, hash'in bit uzunluğuna bağlıdır.
2. Girdi verisi, 512 bitlik bloklar halinde bölünür. Eğer son blok 512 bitten kısa ise, veri dolgu (padding) ile tamamlanır.
3. Her 512 bitlik blok üzerinde sıkıştırma fonksiyonu çalıştırılır. JH'nin sıkıştırma fonksiyonu, bir iç durum üzerinde çalışarak veriyi bir sonuca indirger. Bu aşamada, JH permütasyonu (iç durum) sürekli olarak güncellenir ve her blok için aynı algoritma tekrarlanır.
4. Tüm bloklar işlendikten sonra, son blok üzerinde bir son işlem yapılır ve elde edilen iç durumdan nihai özet (hash) değeri hesaplanır.

2.26 Locality-Sensitive Hash (LSH)

LSH, veri benzerliklerini hızlı bir şekilde bulmak için kullanılan bir tekniktir. Veri noktalarının birbirine yakın olanlarını gruplamak veya benzer ögeleri bulmak için kullanılır. LSH, belirli veri ögelerini özetler ve benzer veri noktalarının aynı özetleme sonucuna sahip olmasını sağlamak amacıyla tasarlanmıştır.

2.26.1 Çalışma Adımları

1. İlk adımda, özetlenmek istenen veri, vektör formuna dönüştürülür. Bu vektör, girdinin sayısal bir temsilidir.

2. Verilen vektörlerin yerel benzerliğini ölçmek için uzayda rastgele hiper düzlemler oluşturulur. Hiper düzlem, vektörlerin hangi tarafta olduğunu belirlemek için kullanılır.

3. Vektörler hiper düzlemler aracılığıyla işlenir ve benzer verilerin aynı hash fonksiyonuyla eşleştirilmesi sağlanır. Burada her bir vektör, hangi hiper düzlemde bulunduğuna bağlı olarak 1 veya 0 ile kodlanır.

4. Son aşamada, vektörlerin hiper düzlem karşılaştırmaları sonucunda elde edilen ikilik (binary) değerler birleştirilir ve özetleme sonucu ortaya çıkar.

2.27 MD2 (Message Digest 2)

1989 yılında Ronald Rivest tarafından geliştirilmiştir. Özetleme işlemi, değişen uzunluktaki bir veri girdisini sabit uzunlukta bir hash değerine (mesaj özeti) dönüştürür. MD2, modern standartlarla karşılaştırıldığında eski bir algoritmadır ve günümüzde kriptografik olarak güvenli kabul edilmez. MD2, her zaman 128 bitlik (16 byte) bir özet değer üretir. Çakışmaların bulunma riski yüksektir.

2.27.1 Çalışma Adımları

1. Girdi mesajının uzunluğu 16 byte'ın bir katı olacak şekilde doldurulur (padding). Doldurma, eksik byte sayısı kadar değer eklenerek yapılır.
2. MD2, veri bütünlüğünü sağlamak için bir checksum değeri hesaplar. Bu checksum, mesajın her byte'ını işler ve belirli bir tabloya göre güncellenir.
3. Girdi mesajı 16 byte'lık bloklara bölünür. Her blok, belirli dönüşüm kurallarına göre işlenir. Bu dönüşümler sabit bir s-dizisi (s-table) kullanılarak gerçekleştirilir.
4. Tüm bloklar işlendiğinde, elde edilen sonuç 128 bitlik (16 byte) özet değeridir.

2.27.2 Python Kodu

```
from Crypto.Hash import MD2
data = b'Hello, World!'
hash_obj = MD2.new()
hash_obj.update(data)
hash_value = hash_obj.hexdigest()
print(f"MD2: {hash_value}")
```

2.28 MD4 (Message Digest 4)

1990 yılında Ronald Rivest tarafından geliştirilmiştir. MD4, değişken uzunluktaki bir veri girdisini sabit uzunlukta bir özet değeri (hash) üretmek için kullanılır. Ancak MD4, artık kriptografik olarak güvenli kabul edilmez ve modern sistemlerde nadiren kullanılır. MD4, her zaman 128 bitlik (16 byte) bir özet değeri üretir. MD4, güvenlik açıkları nedeniyle kullanılmaması gereken bir algoritma olarak kabul edilmektedir. Çakışmaların (collision) kolayca bulunabildiği kanıtlanmıştır.

2.28.1 Çalışma Adımları

1. Veri, 512 bitlik (64 byte) bloklara bölünür. Mesajın uzunluğu 512'nin katı değilse, veri padding ile tamamlanır. İlk olarak, bir 1 biti eklenir, ardından 0 bitleri eklenir. Mesajın toplam uzunluğu (bit cinsinden) son 64 bitlik bir alana yerleştirilir.
2. MD4, dört adet 32-bitlik değişken (A, B, C, D) kullanır. Bu değişkenler sabit başlangıç değerlerine sahiptir:
   - **A**: 0x67452301
   - **B**: 0xEFCDAB89
   - **C**: 0x98BADCFE
   - **D**: 0x10325476
3. Her 512 bitlik blok, 48 turdan oluşan 3 farklı işlem aşamasından geçirilir:
   - **İlk Aşama (F)**: Modifiye edilmiş bir mantıksal AND ve OR işlemi kullanır.
   - **İkinci Aşama (G)**: Çıkış için daha karmaşık bir işlem uygular.
   - **Üçüncü Aşama (H)**: XOR ve rotasyon işlemleriyle çıktıyı işler.
4. Her turda, mesaj blokları belirli bir şekilde karıştırılır ve işlem sonuçları A, B, C, D değişkenlerine eklenir.
5. Tüm bloklar işlendiğinde, A, B, C, D değişkenleri birleştirilerek 128 bitlik (16 byte) özet değeri üretilir.

2.28.2 Python Kodu

```
from Crypto.Hash import MD4
data = b'Hello, World!'
hash_obj = MD4.new()
hash_obj.update(data)
hash_value = hash_obj.hexdigest()
print(f"MD2: {hash_value}")
```

2.29 MD5 (Message Digest 5)

1992 yılında Ronald Rivest tarafından geliştirilmiştir. MD5, bir mesajın değişken uzunluktaki içeriğini sabit uzunlukta 128 bitlik (16 byte) bir özet değerine dönüştürür. Veri bütünlüğünü doğrulamak için kullanılır. Web üzerinden bir dosya indirirken, dosyanın tam olarak indirilip indirilmediğini kontrol etme işlemlerinde MD5 kullanılır. Şifre çözülmesi teorik olarak imkansız olmasına rağmen geçmişte bazı saldırı girişleri yapılmıştır. 1993 yılında Antoon Bossalaers ve Bert den Boer, iki farklı girdi için MD5 algoritmasında çakışma bulmuşlardır. Bu olay, MD5'e olan güveni sarsmıştır. 2004 yılında MD5CRK isimli bir projede, 1 saat içinde MD5 algoritmasına yapılan bazı saldırıların başarılı sonuçlar verdiğini gözlemlemişlerdir. RainbrowCrack isimli bir projede, büyük-küçük harf, rakam gibi tek karakterden başlayıp sonsuz karakter kadar değerlerin MD5 özeti hesaplanarak bir tabloda (rainbow table) saklanmıştır. Daha sonra bu tablo kullanılarak, brute force saldırıları yapılmıştır. Bu tür brute force saldırılarından korunmak için MD5 özeti çıkarılan bir bilginin tekrar tekrar MD5 özetleri çıkarılmıştır. 2008'de bir grup sahte SSL sertifikasını doğrulamak için MD5 algoritmasını kullanmıştır.

2.29.1 Çalışma Adımları

1. Veri, 512 bitlik (64 byte) bloklara bölünür. Mesajın uzunluğu 512'nin katı değilse, veri padding ile tamamlanır. İlk olarak, bir 1 biti eklenir, ardından 0 bitleri eklenir. Mesajın toplam uzunluğu (bit cinsinden) son 64 bitlik bir alana yerleştirilir.
2. MD5, dört adet 32-bitlik değişken (A, B, C, D) kullanır. Bu değişkenler sabit başlangıç değerlerine sahiptir:
   - **A**: 0x67452301
   - **B**: 0xEFCDAB89
   - **C**: 0x98BADCFE
   - **D**: 0x10325476
3. Her blok üzerinde 64 turdan oluşan bir işlem yapılır.
4. Sonuç olarak, başlangıç durumları (A, B, C, D) birleştirilir ve 128 bitlik özet değeri oluşturulur.

2.29.2 Python Kodu

```
from Crypto.Hash import MD5
data = b'Hello, World!'
hash_obj = MD5.new()
hash_obj.update(data)
hash_value = hash_obj.hexdigest()
```

```
print(f"MD2: {hash_value}")
```

2.30 MD6 (Message Digest 6)

2008 yılında Ronald Rivest ve ekibi tarafından geliştirilmiştir. MD6, bir ağacı andıran yapıya sahip, oldukça hızlı ve paralelleştirilebilir bir algoritmadır. Çok uzun verileri paralel olarak işleyebilmek için dörtlü Merkle ağacı yapısını kullanılır. Modern donanımlarda yüksek performans sağlar ve kullanıcı tarafından özelleştirilebilir bir güvenlik parametresi (derinlik) sunar. Çıkış uzunluğu esnektir. Çakışmaya, ön-görüntüye ve ikinci ön-görüntüye karşı güçlüdür. MD6'nın tasarımı, ağaca benzer bir yapı kullanır, bu da paralel işleme için uygundur. Algoritma içerisindeki çevrim sayısı r şöyle hesaplanır; $r = 40 + (\frac{d}{4})$. Burada d, özet uzunluğudur. Ağaç içerisindeki her bir düğümün dört çocuğu olmalıdır ve veri bu çocuklara yerleştirilir. Eğer çocuk sayısı az ise içeriği 0 olan düğümler eklenir. Yapraklarda veri saklanırken düğümlerde sıkıştırma işlemleri yapılır. İşlem yönü aşağıdan yukarıya yani çocuklardan düğümlere doğrudur.

2.31 SHA (Secure Hash Algorithm)

İlk olarak NSA tarafından tasarlanmış ve NIST tarafından standartlaştırılmıştır. SHA algoritması birkaç farklı versiyondan oluşur. Her biri farklı çıkış boyutlarına ve güvenlik seviyelerine sahiptir:

- **SHA-1**: 160 bit çıkışa sahiptir. Çakışma saldırılarına karşı zayıf olduğu için artık önerilmez. Çakışma bulabilmek için brute force saldırısıyla 2^{80} adet deneme yapmak gerekir. 2005 yılında Yiqun Lisa Yin ve ekibi, çalışmalarında 2^{69}'dan daha az işlem yaparak çakışma elde edilebileceğini söylemiştir. 2010 yılında Marc Steven tarafından yapılan çalışmalarda bu sayının 2^{61}'e düşürüldüğünü tahmin etmiştir. SHA-1 üzerinde birçok çalışma yapılmasına rağmen hepsi teorik olarak kalmıştır, bu yüzden algoritma hala güvenilir olarak kabul edilir.
- **SHA-2**: Çıkış 224, 256, 384 veya 512 bit olabilir. Güçlü ve modern uygulamalarda yaygın olarak kullanılır. SHA-1 hala güvenli olarak kabul edildiği için pek kullanım gerekliliği duyulmamıştır.
- **SHA-3**: Çıkış 224, 256, 384 veya 512 bit olabilir. SHA-2'nin daha da güçlendirilmiş bir versiyonu. Keccak algoritmasına dayanır. SHAKE128 ve SHAKE256 olmak üzere iki XOF (genişletilebilir çıktı fonksiyonu) bulunur. XOF, çıktıyı herhangi bir uzunlukta genişletebilen fonksiyondur. 128 ve 256 çıktının uzunluğunu değil, algoritmanın gücünü belirtir. SHA-3 fonksiyonları permütasyon tabanlıdır. İçerisinde bulunan sponge fonksiyonu, mesajı belirli bloklara ayırır ve blok bazı permütasyon işlemlerinin ardından özet blokları birleştirilir.

2.31.1 Python Kodu

```
import hashlib
data = "Hello, World!"
sha256_hash = hashlib.sha256()
sha256_hash.update(data.encode('utf-8'))
hash_value = sha256_hash.hexdigest()
print(f"SHA-256 Hash: {hash_value}")
```

2.32 SHA-3

SHA-3 (Secure Hash Algorithm 3), NIST tarafından 2015 yılında standartlaştırılmış bir özetleme algoritmasıdır. Keccak algoritmasına dayanır ve SHA-2'nin bir alternatifi olarak geliştirilmiştir. SHA-3, modern saldırılara karşı daha yüksek güvenlik sağlamak için tasarlanmıştır ve temel farkı sünger yapısı (sponge construction) kullanmasıdır. SHA-3, giriş verisini işleyip sabit uzunlukta bir özet üreten bir dizi matematiksel işlem gerçekleştirir.

2.32.1 Çalışma Adımları

1. Veri, Keccak algoritmasının gereksinimlerine göre bloklara ayrılır. Veriye padding (doldurma) eklenir, böylece blokların uzunluğu algoritmaya uyum sağlar.
2. SHA-3, absorbing (emilme) ve squeezing (sıkıştırma) aşamalarından oluşan sünger yapısını kullanır. Girdi verisi, bir iç durum (state) ile işlenir ve özet değeri bu durumdan türetilir.
3. Giriş blokları, algoritmanın iç durumuna sırayla emilir. Bu işlem sırasında bitwise XOR ve dönüşümler yapılır.
4. Sonuç bloğu (hash değeri), iç durumdan türetilir.

2.32.2 Python Kodu

```
import hashlib
message = "Hello, World!"
sha3_hash = hashlib.sha3_256()
sha3_hash.update(input_data.encode('utf-8'))
hash_value = sha3_hash.hexdigest()
print(f"SHA3-256 Hash: {hash_value}")
```

2.33 Skein

2008 yılında geliştirilmiştir. Threefish blok şifreleme algoritmasını temel alır.

2.33.1 Çalışma Adımları

1. Veriler, algoritmanın desteklediği boyutlara uygun şekilde bloklara ayrılır.
2. Her bir veri bloğu, Threefish blok şifresi ile işlenir. Bu şifreleme, belirli bir sayıda dönüşüm ve anahtar karıştırması içerir.
3. Skein, UBI işlemini kullanarak her bir bloğun farklı bağlamlarda işlendiğinden emin olur. Bu yapı, bloklar arasında bağlantıyı korurken esneklik sağlar.
4. Tüm bloklar işlendiğinde, sabit uzunlukta bir özet değeri üretilir.

3 Tarihteki Şifreleme Yöntemleri

3.1 Polybius Cipher

Polybius Cipher, eski Yunan filozofu Polybius tarafından önerilmiştir. Bu yöntem, bir tablo kullanılarak hashlerin şifrelenmesi prensibine dayanır. Her harfe tablo üzerinde bir koordinat atanır. İngiliz alfabesinin 26 harfinden biri eksiltilir ve 5x5 bir kare oluşturulur.

	1	2	3	4	5
1	A	B	C	D	E
2	F	G	H	I/J	K
3	L	M	N	O	P
4	Q	R	S	T	U
5	V	W	X	Y	Z

Şifreleme sırasında her harf, tablodaki satır ve sütun numarası ile temsil edilir. Şifre çözme sırasında bu sayılar tekrar harflere dönüştürülür. Örneğin "ALPER" kelimesinin şifrelenmiş hali "1131351542" dir.

3.1.1 Encryption

1. Bir Polybius tablosu oluşturulur.
2. Şifrelenecek metnin her harfi için koordinat bulunur.
3. Bu koordinatlar birleştirilerek şifreli metin elde edilir.

3.1.2 Decryption

1. Şifrelenmiş metin iki rakamlı gruplara ayrılır.
2. Her grubun satır ve sütun değerine karşılık gelen harf bulunur.
3. Harfler birleştirilerek orijinal metin elde edilir.

3.1.3 Python Kodu

```
def create_polybius_square():
    alphabet = "ABCDEFGHIKLMNOPQRSTUVWXYZ"
    square = {}
    index = 0
    for row in range(1, 6):
        for col in range(1, 6):
            square[alphabet[index]] = (row, col)
            index += 1
```

```
    coordinates = {v: k for k, v in square.items()}
    return square, coordinates

def encrypt_polybius(text):
    square, _ = create_polybius_square()
    text = text.upper().replace("J", "I")
    encrypted = ""
    for char in text:
        if char.isalpha():
            row, col = square[char]
            encrypted += f"{row}{col}"
        else:
            encrypted += char

    return encrypted

def decrypt_polybius(text):
    _, coordinates = create_polybius_square()
    decrypted = ""
    i = 0
    while i < len(text):
        if text[i].isdigit() and i + 1 < len(text) and text[i +
            1].isdigit():
            row = int(text[i])
            col = int(text[i + 1])
            decrypted += coordinates[(row, col)]
            i += 2
        else:
            decrypted += text[i]
            i += 1

    return decrypted

plaintext = "Hello World"
ciphertext = encrypt_polybius(plaintext)
decrypted_text = decrypt_polybius(ciphertext)
print("Ciphertext:", ciphertext)
print("Decrypted:", decrypted_text)
```

3.2 Caesar Cipher

Caesar Cipher, Julius Caesar tarafından kullanılan bir yer değiştirme (substitution) şifreleme yöntemidir. Bu yöntemde, alfabenin harfleri sabit bir sayı kadar kaydırılarak şifrelenir. Şifreleme ve deşifreleme işlemi, kaydırma miktarı (shift değeri) üzerinden yapılır. Eğer shift değeri 3 ise, A harfi D harfi olarak şifrelenir. Bu mantık şifreleme sırasında tüm harflere uygulanır. Deşifreleme ise tam tersine kaydırarak yapılır. Caesar algoritması oldukça güvensizdir çünkü brute force saldırıları ile tüm denemeler yapılarak düz metin kolayca elde edilir.

3.2.1 Encryption

1. Shift değeri belirlenir.
2. Şifrelenecek metindeki her harf için; harfin alfabe sırasındaki yerine shift değeri eklenir.
3. Eğer alfabe sonuna ulaşılırsa başa dönülür.

3.2.2 Decryption

1. Şifrelenmiş metindeki her harf için; harfin alfabe sıraısndaki yerinden shift değeri çıkarılır.
2. Eğer alfabe başından önceye gidilerse sona dönülür.

Örneğin "ALPER" kelimesinin 2 kelimelik shiftler ile şifrelenmiş hali "CNRGT" dir.

3.2.3 Python Kodu

```
def caesar_encrypt(plainttext, shift=3):
    encrypted = ""
    for char in plainttext:
        if char.isalpha():
            base = ord('A') if char.isupper() else ord('a')
            encrypted += chr((ord(char) - base + shift) % 26 + base)
        else:
            encrypted += char

    return encrypted

def caesar_decrypt(ciphertext, shift=3):
    decrypted = ""
    for char in ciphertext:
        if char.isalpha():
            base = ord('A') if char.isupper() else ord('a')
            decrypted += chr((ord(char) - base - shift) % 26 + base)
```

```
        else:
            decrypted += char

    return decrypted

plaintext = "Hello World"
ciphertext = caesar_encrypt(plaintext, shift=3)
decrypted_text = caesar_decrypt(ciphertext, shift=3)
print("Ciphertext:", ciphertext)
print("Decrypted:",decrypted_text)
```

3.3 Affine Cipher

Affine Cipher, bir yer değiştirme şifreleme yöntemidir. Her harf, bir matematiksel formül kullanılarak şifrelenir. Şifreleme işlemi, modüler aritmetiğe dayanır ve iki anahtar kullanır: a (çarpan) ve b (toplama sabiti). Bu yöntem, monoalfabetik bir şifreleme türüdür. Şifreleme için:

$$E(x) = (\alpha \cdot x + b) \bmod 26$$

Burada, x, harfin alfabe üzerindeki sırasını (0-25 arasında); α, çarpanı (mod 26 ile asal olmalıdır); b, toplama sabitini; mod 26, alfabe boyutunu temsil eder. Deşifreleme için:

$$D(y) = \alpha^{-1} \cdot (y - b) \bmod 26$$

Burada, y, şifreli harfin alfabe üzerindeki sırasını, α^{-1}, α'nın mod 26'ya göre ters çarpanını temsil eder.

Örneğin, "ALPER" kelimesini Affine Cipher ile şifreleyelim. $\alpha = 5$ ve $b = 8$ olsun.

A	B	C	D	E	F	G	H	I	J	K	L	M
0	1	2	3	4	5	6	7	8	9	10	11	12
N	O	P	Q	R	S	T	U	V	W	X	Y	Z
13	14	15	16	17	18	19	20	21	22	23	24	25

Tabloya göre şifrelenmiş mesaj şu şekildedir:

- A harfi için $y = (5 \cdot 0 + 8) \bmod 26 = \text{I}$
- L harfi için $y = (5 \cdot 9 + 8) \bmod 26 = \text{L}$
- P harfi için $y = (5 \cdot 15 + 8) \bmod 26 = \text{F}$
- E harfi için $y = (5 \cdot 4 + 8) \bmod 26 = \text{C}$
- R harfi için $y = (5 \cdot 16 + 8) \bmod 26 = \text{P}$

"ALPER" kelimesinin Affine Cipher ile şifrelenmiş hali "ILFCP" dir. Şimdi ise bu şifreyi çözelim. Her bir harf için:

- $1 = \alpha \cdot \alpha^{-1} \bmod 26$, buradan $\alpha_{\text{inv}} = 21$ olur.
- "I" karakteri için, tabloda değeri 8, $D(x) = 21 \cdot (8 - 8) \bmod 26 = 0$ olur. 0'ın tablodaki değeri "A"'dır.
- "L" karakteri için, tabloda değeri 11, $D(x) = 21 \cdot (11 - 8) \bmod 26 = 11$ olur. 11'in tablodaki değeri "L"'dır.
- "F" karakteri için, tabloda değeri 5, $D(x) = 21 \cdot (5 - 8) \bmod 26 = 15$ olur. 15'in tablodaki değeri "P"'dır.

- "C" karakteri için, tabloda değeri 2, $D(x) = 21 \cdot (2 - 8) \bmod 26 = 4$ olur. 4'ün tablodaki değeri "E"'dır.
- "P" karakteri için, tabloda değeri 15, $D(x) = 21 \cdot (15 - 8) \bmod 26 = 17$ olur. 17'nin tablodaki değeri "R"'dır.

Böylece, şifreli metin "ILFCP" den tekrar "ALPER" kelimesini elde etmiş olduk.

3.3.1 Python Kodu

```
import math

def mod_inverse(a, m):
    for x in range(1, m):
        if (a * x) % m == 1:
            return x

    return None

def affine_encrypt(plaintext, a=5, b=8):
    if math.gcd(a, 26) != 1:
        raise ValueError("")

    encrypted = ""
    for char in plaintext.upper():
        if char.isalpha():
            x = ord(char) - ord("A")
            encrypted += chr(((a * x + b) % 26) + ord("A"))
        else:
            encrypted += char

    return encrypted

def affine_decrypt(ciphertext, a=5, b=8):
    if math.gcd(a, 26) != 1:
        raise ValueError("")

    a_inv = mod_inverse(a, 26)
    if a_inv is None:
        raise ValueError

    decrypted = ""
    for char in ciphertext.upper():
        if char.isalpha():
            y = ord(char) - ord("A")
            decrypted += chr(((a_inv * (y - b)) % 26) + ord("A"))

        else:
```

```
            decrypted += char

    return decrypted

plaintext = "Hello World"
ciphertext = affine_encrypt(plaintext, a=5, b=8)
decrypted_text = affine_decrypt(ciphertext, a=5, b=8)
print("Ciphertext:", ciphertext)
print("Decrypted:", decrypted_text)
```

3.4 Vigenere Cipher

Vigenere Cipher, çok alfabeli yer değiştirme şifreleme (polyalphabetic substitution cipher) yöntemidir. 1553 yılında Giovan Battista Bellaso tarafından tanıtılmıştır ve 1586'da Fransız diplomat Blaise de Vigenere tarafından geliştirilmiştir. Şifreleme ve deşifreleme işlemi, bir anahtar kelime kullanılarak gerçekleştirilir. Anahtar kelime, şifrelenecek metnin uzunluğuna kadar tekrar eder. Bu yöntem, basit yer değiştirme şifrelemelerine göre daha güvenlidir, çünkü farklı alfabeler arasında geçiş yaparak daha karmaşık bir şifreleme sağlar. Şifreleme için, her harf bir anahtar kelime yardımıyla bir alfabe içinde kaydırılır. Caesar algoritmasının geliştirilmiş bir halidir.

$$C_i = (P_i + K_i) \bmod 26$$

Burada, P_i, düz metindeki harfin alfabe sırası (0-25 arasında); K_i, anahtar keliminin ilgili harfinin alfabe sırası (0-25 arasında); C_i, şifrelenmiş metindeki harfin alfabe sırasıdır. Deşifreleme işleminde, şifreleme işlerminin tersi yapılır.

$$P_i = (C_i - K_i) \bmod 26$$

Örneğin "ANAHTAR" kelimesini kullanarak "ALPERKARACA" ismini şifreleyelim.

A	B	C	D	E	F	G	H	I	J	K	L	M
0	1	2	3	4	5	6	7	8	9	10	11	12
N	O	P	Q	R	S	T	U	V	W	X	Y	Z
13	14	15	16	17	18	19	20	21	22	23	24	25

Tabloya göre şifrelenmiş mesaj şu şekildedir:

Anahtar Harf	Mesaj Harf	Index Toplamı	Mod	Şifreli Mesaj Index	Şifreli Harf
A	A	$0 + 0 = 0$	$0 \bmod 26 = 0$	0	A
N	L	$13 + 11 = 24$	$24 \bmod 26 = 24$	24	Y
A	P	$0 + 15 = 15$	$15 \bmod 26 = 15$	15	P
H	R	$7 + 4 = 11$	$11 \bmod 26 = 11$	11	L
T	T	$19 + 17 = 36$	$36 \bmod 26 = 10$	10	K
A	K	$0 + 10 = 10$	$10 \bmod 26 = 10$	10	K
R	A	$17 + 0 = 17$	$17 \bmod 26 = 17$	17	R
A	R	$0 + 17 = 17$	$17 \bmod 26 = 17$	17	R
N	A	$13 + 0 = 13$	$13 \bmod 26 = 13$	13	N
A	C	$0 + 2 = 2$	$2 \bmod 26 = 2$	2	C
H	A	$7 + 0 = 7$	$7 \bmod 26 = 7$	7	H

Buradan "ANAHTAR" kelimesi anahtarı ile "ALPERKARACA" mesajını şifrelenmiş hali "AYPLKKRRNCH" geliyor. Şimdi ise bunu aynı anahtar ile geri çözelim. Şifre çözme işlemi için şifreli mesaj harflerinin değerlerinden, anahtar kelimenin harflerinin değerleri çıkarılır. Eğer sonuç sıfırdan küçükse üzerine mod yani 26 eklenir.

Şifreli Mesaj Harf	Anahtar Harf	Index Toplamı	Mod	Mesaj Index	Mesaj Harf
A	A	$0-0=0$	0	0	A
Y	N	$24-13=11$	11	11	L
P	A	$15-0=15$	15	15	P
L	H	$11-7=4$	4	4	E
K	T	$10-19=-9$	$-9+26=17$	17	R
K	A	$10-0=10$	10	10	K
R	R	$17-17=0$	0	0	A
R	A	$17-0=17$	17	17	R
N	N	$13-13=0$	0	0	A
C	A	$2-0=2$	2	2	C
H	H	$7-7=0$	0	0	A

Görüldüğü gibi tekrardan "ALPERKARACA" kelimesini elde ettik.

3.4.1 Python Kodu

```
def vigenere_encrypt(plaintext, key):
    key = key.upper()
    encrypted = ""
    key_index = 0
    for char in plaintext.upper():
        if char.isalpha():
            p = ord(char) - ord("A")
            k = ord(key[key_index]) - ord("A")
            encrypted += chr((p + k) % 26 + ord("A"))
            key_index = (key_index + 1) % len(key)
        else:
            encrypted += char

    return encrypted

def vigenere_decrypt(ciphertext, key):
    key = key.upper()
    decrypted = ""
    key_index = 0
    for char in ciphertext.upper():
        if char.isalpha():
            c = ord(char) - ord('A')
            k = ord(key[key_index]) - ord('A')
            decrypted += chr((c - k + 26) % 26 + ord('A'))
            key_index = (key_index + 1) % len(key)
        else:
            decrypted += char
```

```
    return decrypted

plaintext = "Hello World"
key = "secretkey"
ciphertext = vigenere_encrypt(plaintext, key)
decrypted_text = vigenere_decrypt(ciphertext, key)
print("Ciphertext:", ciphertext)
print("Decrypted:", decrypted_text)
```

3.5 Playfair Cipher

1854 yılında Charles Wheatstone tarafından geliştirilmiştir. Carles Wheatsone tarafından geliştirilmiş olmasına rağmen adını bu algoritmanın kullanılmasını savunan Lord Playfair'den alır. Çift harfli (digraph) yer değiştirme şifreleme yöntemidir. Şifreleme işlemi 5x5 harf matrisinin kullanımıyla gerçekleştirilir. Bu yöntem, metni harf çiftleri (digraph) halinde işler ve bu çiftlerin matris üzerindeki pozisyonlarına göre şifreler. İlk yıllarda yöntemi karmaşık bulan İngiliz Dışişleri Bakanlığı daha sonra 1. ve 2. Dünya Savaşlarında bu yöntemi kullanmıştır. 2. Dünya Savaşında Avustral ve Yeni Zelanda da kullanmıştır.

3.5.1 Encryption

1. Anahtar kelime alınır ve tekrar eden harfler çıkarılarak matrisin ilk satırlarına yazılır. Matris, geriye kalan alfabe harfleriyle doldurulur.
2. Düz metin iki harfli gruplara bölünür. Aynı harfler bir çift içinde yer alırsa, araya "X" eklenir. Tek harfli kalan metinler için sona bir "X" eklenir.
3. Her harf çifti için matris üzerindeki konumlarına göre şifrelenir:
    - **Aynı Satır**: Harfler sağa doğru kaydırılır.
    - **Aynı Sütun**: Harfler aşağıya doğru kaydırılır.
    - **Farklı Satır ve Sütun**: Harfler dikdörtgenin karşı köşeleriyle değiştirilir.

Örneğin, "MONARCHY" anahtarı ile "ALPERKARACAD" kelimesini şifreleyelim. "MONARCHY" anahtarında tekrar eden bir harf olmadığı için anahtar aynı şekilde matrise eklenir. Matris:

M	O	N	A	R
C	H	Y	B	D
E	F	G	I/J	K
L	P	Q	S	T
U	V	W	X	Z

Şimdi "ALPERKARACA" kelimesini ikili gruplara ayıralım: "AL PE RK AR AC AD". Tabloya göre:

- Köşeleri A ve L olan karenin diğer köşeleri: MS.
- Köşeleri P ve E olan karenin diğer köşeleri: LF.

- R ve K aynı sütunda olduğu için birer altlarındaki değerler: DT.
- A ve R aynı satırda olduğu için birer sağındakiler: RM.
- Köşeleri A ve C olan karenin diğer köşeleri: MB.
- Köşeleri A ve D olan karenin diğer köşeleri: RB.

"MONARCHY" anahtarı ile "ALPERKARACAD" kelimesinin şifrelenmiş hali "MSLFDTRMMBRB" dir.

3.6 Bifid Cipher

1901 yılında Fransız kriptograf Felix Delastelle tarafından geliştirilmiştir. Felix Delastelle daha sonra Trifid ve Four-square şifreleme yöntemlerini de geliştirmiştir. Bu yöntem, Polybios karesi kullanarak hem yer değiştirme hem de transpozisyon şifreleme tekniklerini birleştirir. Amaç, bir mesajın harflerini hem satır hem de sütun koordinatları üzerinden şifrelemektir.

3.6.1 Encryption

1. Polybios karesini oluşturmak için alfabenin harfleri, bir 5x5 matris içinde yerleştirilir. Anahtar kelime, matrisin doldurulmasında ilk sırayı alır ve ardından geri kalan alfabe harfleri eklenir.
2. Düz metin (plaintext) yalnızca alfabe harflerinden oluşmalıdır ve büyük harflerle yazılmalıdır.
3. Her harf için matrisin satır ve sütun koordinatları alınır ve bir liste-eye yazılır.
4. Satır ve sütun koordinatları birleştirilerek yeni bir sıralama oluşturulur.
5. Oluşan yeni sıralama, her çift sayı için matristen harf seçilerek şifreli metin (ciphertext) elde edilir.

Örneğin, "ANAHTAR" kelimesi anahtarı ile "ALPERKARACA" kelimesini şifreleyelim. Bifid matrisi:

	1	2	3	4	5
1	A	N	H	T	R
2	B	C	D	E	F
3	G	I/J	K	L	M
4	O	P	Q	S	U
5	V	W	X	Y	Z

Tabloya göre "ALPERKARACA" kelimesinin şifrelenmiş hali:

Harf	A	L	P	E	R	K	A	R	A	C	A
Satır	1	3	4	2	1	3	1	1	1	2	1
Sütun	1	4	2	4	5	3	1	5	1	2	1

Şimdi bu tablodaki değerleri soldan sağa doğru ikişerli şekilde okuyarak Bifid tablosundaki değeriyle mesajı şifreleyelim:

- 13, 1. satır 3. sütun: H
- 42, 4. satır 2. sütun: P
- 13, 1. satır 3. sütun: H
- 11, 1. satır 1. sütun: A
- 12, 1. satır 2. sütun: N
- 11, 1. satır 1. sütun: A
- 42, 4. satır 2: sütun: P
- 45, 4. satır 5. sütun: U
- 31, 3. satır 1. sütun: G
- 51, 5. satır 1. sütun: V
- 21, 2. satır 1. sütun: B

"ANAHTAR" anahtarı ile "ALPERKARACA" mesajının şifrelenmiş hali "HPHANAPUGVB" dır.

3.6.2 Python Kodu

```
def create_polybius_square(key):
    key = key.upper().replace("J", "I")
    alphabet = "ABCDEFGHIKLMNOPQRSTUVWXYZ"
    key_square = []
    used_chars = set()
    for char in key:
        if char not in used_chars and char in alphabet:
            key_square.append(char)
            used_chars.add(char)

    for char in alphabet:
        if char not in used_chars:
            key_square.append(char)
```

```
    square = [key_square[i:i + 5] for i in range(0, 25, 5)]
    positions = {char: (i // 5, i % 5) for i, char in
        enumerate(key_square)}
    return square, positions

def bifid_encrypt(plaintext, key):
    square, positions = create_polybius_square(key)
    plaintext = plaintext.upper().replace('J', 'I')
    row_coords = []
    col_coords = []
    for char in plaintext:
        if char in positions:
            row, col = positions[char]
            row_coords.append(row)
            col_coords.append(col)

    combined_coords = row_coords + col_coords
    ciphertext = ""
    for i in range(0, len(combined_coords), 2):
        row = combined_coords[i]
        col = combined_coords[i + 1]
        ciphertext += square[row][col]

    return ciphertext

def bifid_decrypt(ciphertext, key):
    square, positions = create_polybius_square(key)
    reverse_positions = {v: k for k, v in positions.items()}
    coords = []
    for char in ciphertext:
        for item in positions[char]:
            coords.append(item)

    row_coords = coords[:len(coords) // 2]
    col_coords = coords[len(coords) // 2:]
    plaintext = ""
    for r, c in zip(row_coords, col_coords):
        plaintext += square[r][c]

    return plaintext

plaintext = "Hello World"
key = "secretkey"
ciphertext = bifid_encrypt(plaintext, key)
decrypted_text = bifid_decrypt(ciphertext, key)
print("Ciphertext:", ciphertext)
print("Decrypted:", decrypted_text)
```

3.7 Trifid Cipher

Trifid Cipher, Felix Delastelle tarafından geliştirilen çok boyutlu bir şifreleme yöntemidir. Bifid Şifreleme'nin bir uzantısı olarak, Trifid metodu metni şifrelemek için üç boyutlu bir matrisi (3x3x3) kullanır. Hem yer değiştirme hem de transpozisyon prensiplerini birleştirir, bu da daha güçlü bir şifreleme sağlar.

3.7.1 Encryption

1. Alfabenin harfleri, 27 hücrelik bir kübe yerleştirilir.
2. Şifrelenecek metin büyük harflere çevrilir. Harfler, kübteki pozisyonlarına göre (katman, satır, sütun) bir koordinat kümesine çevrilir.
3. Pozisyonlar üç ayrı gruba ayrılır: Katman, Satır, Sütun koordinatları. Koordinatlar birleştirilerek sıralama değiştirilir.
4. Yeni sıralamaya göre koordinatlar tekrar gruplanır ve kübteki harfler şifreli metni oluşturur.

Örneğin, "ANAHTAR" anahtarı ile "ALPER" mesajını şifreleyim. 3 adet tablo:

	1	2	3
1	A	N	H
2	T	R	B
3	C	D	E

	1	2	3
1	F	G	J
2	J	K	L
3	M	O	P

	1	2	3
1	Q	S	U
2	V	W	X
3	Y	Z	-

Tablodaki değerlere göre "ALPER" kelimesinin tablosu:

Harf	A	L	P	E	R
Katman	1	2	2	1	1
Sütun	1	3	3	3	2
Satır	1	2	3	3	2

Şimdi bu tablodaki değerleri soldan sağa doğru üçerli şekilde okuyarak Trifid tablosundaki değeriyle mesajı şifreleyelim:

- 122, 1. katman 2. sütun 2. satır: R
- 111, 1. katman 1. sütun 1. satır: A
- 333, 3. katman 3. sütun 3. satır: -
- 212, 2. katman 1. sütun 2. satır: J
- 332, 3. katman 3. sütun 2. satır: X

"ANAHTAR" anahtarı ile "ALPER" mesajının şifrelenmiş hali "RAJX" dır.

3.8 Vernam Cipher

Vernam Şifreleme, 1917'de Gilbert Vernam tarafından geliştirilen ve "One-Time Pad" olarak da bilinen bir şifreleme yöntemidir. 1. Dünya Savaşında Almanların çözemeyeceği bir metod geliştirilmesi için görevlendirilen mühendis Gilbert Vernam, Joseph Mauborgne adlı bir subay ile bu yöntemi geliştirdi. Bu yöntem, bir veri ve bir anahtar arasında bir mod-2 (XOR) işlemi gerçekleştirerek veri şifreler. Düz metin (plaintext) bir bit dizisine dönüştürülür. Anahtar da aynı uzunlukta rastgele bir bit dizisi olarak oluşturulur. Şifreleme işleminde, düz metnin her biti ile anahtarın karşılık gelen biti XOR işlemi ile işlenir:

$$C_i = P_i \oplus K_i$$

Burada, P_i, düz metinin i-inci biti; K_i, anahtarın i-inci biti; C_i, şifreli metnin i-inci biti.

Örneğin, "VERNAM" anahtarı ile "KARACA" kelimesini şifreleyelim. İlk olarak anahtar ve mesajdaki her bir harfin ASCII kodunun binary kodu elde edilir. Daha sonra bu anahtar ve mesajın binary kodları xor işlemine girer. ASCII tablosunda büyük-küçük harf duyarlıdır.

Anahtar Harf	ASCII Kodu	Binary Kodu	Mesaj Harf	ASCII Kodu	Binary Kodu
V	86	1010110	K	75	1001011
E	69	1000101	A	65	1000001
R	82	1010010	R	82	1010010
N	78	1001110	A	65	1000001
A	65	1000001	C	67	1000011
M	77	1001101	A	65	1000001

Anahtar Harf Binary	Mesaj Harf Binary	XOR İşlemi	ASCII Kodu
1010110	1001011	00011101	29
1000101	1000001	00000100	4
1010010	1010010	00000000	0
1001110	1000001	00001111	15
1000001	1000011	00000010	2
1001101	1000001	00001100	12

"KARACA" kelimesinin şifreli hali "00011101 00000100 00000000 00001111 00000010 00001100" dir. Aynı anahtarın binary kodunu ile şifreli mesajı xor işlemine sokarak şifrelenen metin çözülür.

3.8.1 Python Kodu

```
def vernam_encrypt(plaintext, key):
    binary_plaintext = ''.join(format(ord(char), '08b') for char in
        plaintext)
    binary_key = ''.join(format(ord(char), '08b') for char in key)
```

```
    if len(binary_plaintext) != len(binary_key):
        raise ValueError()

    ciphertext = ''.join('1' if p != k else '0' for p, k in
        zip(binary_plaintext, binary_key))
    return ciphertext

def vernam_decrypt(ciphertext, key):
    binary_key = ''.join(format(ord(char), '08b') for char in key)
    plaintext_binary = ''.join('1' if c != k else '0' for c, k in
        zip(ciphertext, binary_key))
    chars = [plaintext_binary[i:i+8] for i in range(0,
        len(plaintext_binary), 8)]
    decrypted = ''.join(chr(int(char, 2)) for char in chars)
    return decrypted

plaintext = "Hello World"
key = "secretkeyyy"
ciphertext = vernam_encrypt(plaintext, key)
decrypted_text = vernam_decrypt(ciphertext, key)
print("Ciphertext (Binary):", ciphertext)
print("Decrypted:", decrypted_text)
```

3.9 Hill Cipher

Hill Cipher, 1929 yılında Lester S. Hill tarafından geliştirilen bir şifreleme yöntemidir. Bu yöntem, doğrusal cebir kullanarak matris çarpımı prensibine dayalı bir şifreleme sağlar. Hill şifreleme, blok şifreleme yöntemleri arasında yer alır ve metni belirli boyutlardaki bloklara ayırarak işlem yapar.

Şifreleme için:

$$C = (K \times P) \bmod 26$$

Deşifreleme için:

$$P = (K^{-1} \times C) \bmod 26$$

3.9.1 Encryption

1. $n \times n$ boyutunda bir kare matris (anahtar) oluşturulur. Bu matris şifreleme ve deşifreleme işlemlerinde kullanılır. Anahtar matrisin determinantı 26 ile aralarında asal olmalıdır. Bu, matrisin tersinin alınabilir olmasını sağlar.

2. Düz metin (plaintext) harfler halinde sayılara dönüştürülür. Eğer metin uzunluğu matris boyutuna tam bölünmüyorsa, boşlukları doldurmak için dolgu karakteri eklenir.

3. Düz metin blokları, anahtar matrisi ile çarpılır ve mod 26 alınır.

Örneğin, "java" kelimesini 2x2'lik bir anahtar ile şifreleyelim. Anahtarımız $\begin{bmatrix} 6 & 2 \\ 1 & 4 \end{bmatrix}$ olsun. İlk olarak mesaj ikili bloklara bölünür. Böylece: $java = \begin{bmatrix} j \\ a \end{bmatrix}, \begin{bmatrix} v \\ a \end{bmatrix}$ olur.

A	B	C	D	E	F	G	H	I	J	K	L	M
0	1	2	3	4	5	6	7	8	9	10	11	12
N	O	P	Q	R	S	T	U	V	W	X	Y	Z
13	14	15	16	17	18	19	20	21	22	23	24	25

Sonra tabloya göre her bir harfe gelen değerlerle matris oluşturulur: $java = \begin{bmatrix}9\\0\end{bmatrix}, \begin{bmatrix}21\\0\end{bmatrix}$. Anahtar ile ikili bloklar çarpılır.

$$\begin{bmatrix}6 & 2\\1 & 4\end{bmatrix} \cdot \begin{bmatrix}9\\0\end{bmatrix} = \begin{bmatrix}54\\9\end{bmatrix}$$

$$\begin{bmatrix}6 & 2\\1 & 4\end{bmatrix} \cdot \begin{bmatrix}21\\0\end{bmatrix} = \begin{bmatrix}126\\21\end{bmatrix}$$

Çıkan sonuç mod değerinden büyük olduğu için 26'ya göre modu alınır.

$$\begin{bmatrix}54\\9\end{bmatrix} \bmod 26 = \begin{bmatrix}2\\9\end{bmatrix}$$

$$\begin{bmatrix}126\\21\end{bmatrix} \bmod 26 = \begin{bmatrix}22\\9\end{bmatrix}$$

Bulunan değerlerin tablodaki harf karşılığı bize şifreli mesajı verir.

$$\begin{bmatrix}2\\9\end{bmatrix} = \begin{bmatrix}c\\j\end{bmatrix}$$

$$\begin{bmatrix}22\\21\end{bmatrix} = \begin{bmatrix}w\\j\end{bmatrix}$$

Böylece "java" mesajının şifreli hali "cjwj" olur. Şimdi ise bu şifreli mesajı çözelim. Öncelikle anahtar matrisinin tersi alınır; anahtar matrisinin tersi ile kendisinin çarpımı bize birim matrisi verir. Anahtar matrisin tersinde negatif değer varsa mod yani 26 değeri eklenir. Anahtar matrisin tersini hesaplarsak:

$$A^{-1} = (ad - bc)^{-1} \begin{bmatrix}d & -b\\-c & a\end{bmatrix}$$

$$A^{-1} = (6 \cdot 4 - 2 \cdot 1)^{-1} \begin{bmatrix}4 & -2\\-1 & 6\end{bmatrix} \bmod 26$$

$$A^{-1} = \frac{1}{22} \cdot \begin{bmatrix}4 & -2\\-1 & 6\end{bmatrix} \bmod 26$$

Matrisin tersi bulunduktan sonra matris ile şifreli mesaj çarpılır. Bulunan değerlerin 26'ya göre modu alınır. Elde edilen değerlerin tablodaki harf karşılıkları alındığında metin çözülmüş olur.

3.9.2 Python Kodu

```
import numpy as np

def mod_inverse(a, m):
    a = a % m
    for x in range(1, m):
        if (a * x) % m == 1:
            return x
    return None

def hill_encrypt(plaintext, key_matrix):
    n = key_matrix.shape[0]
    plaintext = plaintext.upper().replace(" ", "")
    if len(plaintext) % n != 0:
        plaintext += 'X' * (n - len(plaintext) % n)

    plaintext_numbers = [ord(char) - ord('A') for char in plaintext]
    plaintext_blocks = np.array(plaintext_numbers).reshape(-1, n)

    ciphertext = []
    for block in plaintext_blocks:
        encrypted_block = np.dot(key_matrix, block) % 26
        ciphertext.extend(encrypted_block)

    encrypted = ''.join(chr(num + ord('A')) for num in ciphertext)
    return encrypted

def hill_decrypt(ciphertext, key_matrix):
    n = key_matrix.shape[0]
    ciphertext_numbers = [ord(char) - ord('A') for char in ciphertext]
    ciphertext_blocks = np.array(ciphertext_numbers).reshape(-1, n)

    det = int(round(np.linalg.det(key_matrix)))
    det_inv = mod_inverse(det, 26)
    key_matrix_inv = np.linalg.inv(key_matrix) * det
    key_matrix_inv = (key_matrix_inv * det_inv) % 26
    key_matrix_inv = np.round(key_matrix_inv).astype(int) % 26

    plaintext = []
    for block in ciphertext_blocks:
        decrypted_block = np.dot(key_matrix_inv, block) % 26
        plaintext.extend(decrypted_block)

    decrypted = ''.join(chr(num + ord('A')) for num in plaintext)
    return decrypted

key_matrix = np.array([[6, 24, 1], [13, 16, 10], [20, 17, 15]])
plaintext = "Hello World"
```

```
ciphertext = hill_encrypt(plaintext, key_matrix)
decrypted_text = hill_decrypt(ciphertext, key_matrix)
print("Ciphertext:", ciphertext)
print("Decrypted:", decrypted_text)
```

3.10 Bible Code

Bible Code (İncil Kodu), klasik bir şifreleme yöntemi olmaktan çok, belirli bir metin içerisinden gizli mesajlar ya da kodlar bulma amacıyla kullanılan bir yöntemdir. Tarihte, İncil gibi büyük metinlerde gizli mesajların bulunabileceği inancı üzerine kurgulanmıştır. Bu yöntem, metin içerisindeki harflerin belirli bir desenle seçilmesi ve bir mesaj oluşturulması üzerine dayanır.

3.10.1 Encryption

1. Şifreleme ve çözme işlemleri için bir kaynak metin seçilir. Bu genelde İncil gibi uzun bir metindir.
2. Şifrelenecek veriye göre bir desen (örneğin her 5. harfi seçmek gibi) belirlenir.
3. Şifrelenecek mesajın her bir harfi, kaynak metindeki bir pozisyonla eşleştirilir.
4. Kaynak metin üzerinden harflerin sıralı bir şekilde bulunması sağlanır.

3.10.2 Python Kodu

```
def bible_code_encrypt(message, text, step):
    message = message.upper().replace(" ", "")
    text = text.upper().replace(" ", "").replace("\n", "")

    indices = []
    current_index = 0

    for char in message:
        while current_index < len(text):
            if text[current_index] == char:
                indices.append(current_index)
                current_index += step
                break
            current_index += 1

    return indices

def bible_code_decrypt(indices, text):
    text = text.upper().replace(" ", "").replace("\n", "")
    message = ''.join([text[i] for i in indices])
    return message

source_text = ""
message_to_encrypt = "God"
step_size = 5
```

```
encrypted_indices = bible_code_encrypt(message_to_encrypt, source_text,
    step_size)
decrypted_message = bible_code_decrypt(encrypted_indices, source_text)
print("Encrypted (Indexes):", encrypted_indices)
print("Decrypted:", decrypted_message)
```

3.11 Base64

Base64, ikili verileri (binary data) ASCII formatına dönüştürmek için kullanılan bir kodlama yöntemidir. Şifreleme değil, bir kodlama yöntemidir ve esas amacı, veriyi taşınabilir ve okunabilir hale getirmektir. E-posta sistemlerinde, URL'lerde veya JSON formatında veriyi taşırken kullanılır. Veriyi 6 bitlik gruplara böler ve bu grupları bir tabloya göre ASCII karakterlerine dönüştürür. Alfabetik harfler (A-Z, a-z), rakamlar (0-9), +, / karakterlerini kullanır. 64 farklı karakter kullandığı için "Base64" adını almıştır. Eksik bitleri tamamlamak için "=" karakteri ile dolgu yapılır.

3.11.1 Encryption

1. Kodlanacak veri önce ASCII değerlerine, ardından ikili (binary) formatına dönüştürülür.
2. Oluşan binary veri 6 bitlik gruplara bölünür. Eğer toplam uzunluk 6'nın katı değilse, veri 0 eklenerek tamamlanır.
3. 6 bitlik gruplar, Base64 tablosundaki karakterlere dönüştürülür.
4. Eğer veri tam 3 byte (24 bit) değilse, eksik kısımlar eşitlik (=) karakteriyle doldurulur.

Base64 tablosunda, 0-25 arası indekslerde A-Z, 26-51 arası indekslerde a-z, 52-61 arası indekslerde 0-9, 62. indekste "+" ve 63. indiste "/" bulunur. Şifrelenecek mesaj önce üçerli gruplara bölünür. Bunun nedeni her karakterin 8 bit olması ve bu blokların 6 bitlik yeni bloklara bölünecek olmasıdır. 6 ve 8'ın EKOK'u 24'tür. Her üçerli blokta 24 bit bulunur. 24 bit ile 6 bitlik 4 blok oluşturulur. Örneğin, Base64 ile "KARACA" kelimesini şifreleyelim. "KARACA" mesajını "KAR" ve "ACA" olmak üzere ikiye ayıralım. Her parçada 3 karakter var, her karakter 8 bit ise 24 bit elde ettik. Bu 24 biti de 6 bitlik 4 gruba böleceğiz.

	ASCII Değeri	Binary Değeri
K	75	01001011
A	65	01000001
R	82	01010010
A	65	01000001
C	67	01000011
A	65	01000001

Böylece:

- KAR: 010010110100000101010010
- ACA: 010000010100001101000001

Elde edildi. Şimdi bu mesajları 6 bitlik gruplara bölelim

- KAR: 010010 110100 000101 010010
- ACA: 010000 010100 001101 000001

Bu değerlerin ASCII değerinin tablodaki karşılığını alarak mesajı şifreleyelim.

Binary Değeri	Decimal Değeri	Tablodaki Karakter
010010	18	S
110100	52	0
000101	5	F
010010	18	S
010000	16	Q
010100	20	U
001101	13	N
000001	1	B

"KARACA" kelimesinin base64 ile şifrelenmiş hali "S0FSQUNB" elde edildi. Şifre çözme aşamasında ise bu karakterlerin binary karşılıklarının 8 bitlik gruplar halinde bölünür ve bu grupların decimal değerinin tablodaki karşılığı elde edilerek mesaj çözülür.

3.11.2 Python Kodu

```
import base64

def base64_encode(data):
    byte_data = data.encode('utf-8')
    encoded_data = base64.b64encode(byte_data)
    return encoded_data.decode('utf-8')

def base64_decode(encoded_data):
    byte_data = encoded_data.encode('utf-8')
    decoded_data = base64.b64decode(byte_data)
    return decoded_data.decode('utf-8')

plaintext = "Hello World"
encoded = base64_encode(plaintext)
decoded = base64_decode(encoded)
print("Encoded:", encoded)
print("Decoded:", decoded)
```

3.12 ROT13 Cipher

ROT13 (Rotation by 13 places) basit bir şifreleme yöntemidir. Her harfi, alfabedeki 13. harfe kaydırarak şifreler. Eğer bir harf A ile M arasında ise, harf 13 kaydırılır; eğer N ile Z arasında ise yine 13 kaydırılır. Bu yöntem, şifreyi hem şifreler hem de çözer, çünkü alfabede 26 harf olduğundan, 13 kaydırma işleminden sonra aynı harfe geri dönülür. Bu özellik, ROT13'ün kolayca çözülmesini sağlar.

3.12.1 Python Kodu

```
import string

def rot13_encrypt_decrypt(text):
    rot13_table = str.maketrans(
        string.ascii_lowercase + string.ascii_uppercase,
        string.ascii_lowercase[13:] + string.ascii_lowercase[:13] +
        string.ascii_uppercase[13:] + string.ascii_uppercase[:13]
    )
    return text.translate(rot13_table)

plaintext = "Hello World"
encrypted_text = rot13_encrypt_decrypt(plaintext)
decrypted_text = rot13_encrypt_decrypt(encrypted_text)
print("Encrpyted:", encrypted_text)
print("Decrypted:", decrypted_text)
```

A	B	C	D	E	F	G	H	I	J	K	L	M
0	1	2	3	4	5	6	7	8	9	10	11	12
N	O	P	Q	R	S	T	U	V	W	X	Y	Z
13	14	15	16	17	18	19	20	21	22	23	24	25

Örneğin, "ALPER" mesajının ROT13 ile şifrelenmiş hali tabloya göre "NYCRE" dir.

3.13 Lehmer Code

Lehmer Code, sıralı bir küme içerisindeki öğelerin sırasını temsil etmek için kullanılan bir kodlama sistemidir. Sıralama ve permütasyonlarla ilişkili işlemlerde kullanılır. Lehmer kodu, bir permütasyonun sırasını ifade etmek için her öğe için "daha küçük" öğelerin sayısını belirler. Bu yöntem, belirli bir öğenin sırasını hesaplamak için kullanılır.

3.13.1 Python Kodu

```
def lehmer_code_encryption(perm):
    n = len(perm)
    lehmer = []
    for i in range(n):
        count = 0
        for j in range(i + 1, n):
            if perm[j] < perm[i]:
                count += 1
        lehmer.append(count)
    return lehmer

def lehmer_code_decryption(lehmer):
    n = len(lehmer)
    perm = []
    elements = list(range(1, n + 1))

    for i in range(n):
        index = lehmer[i]
        perm.append(elements.pop(index))
    return perm

perm = [3, 1, 2]
encrypted = lehmer_code_encryption(perm)
decrypted = lehmer_code_decryption(encrypted)
print("Encrypted:", encrypted)
print("Decrypted:", decrypted)
```

4 Gizli Anahtarlı Şifreleme (Symmetric Encryption)

Simetrik şifrelemede, aynı anahtar hem şifreleme hem de şifre çözme işlemleri için kullanılır. Veri, bir şifreleme algoritması ve bir gizli anahtar kullanılarak şifrelenir. Şifrelenmiş veri, aynı gizli anahtarı bilen bir alıcıya gönderilir. Alıcı, aynı anahtarı kullanarak veriyi çözer ve orijinal haline ulaşır. Çok hızlıdır. Anahtar paylaşımı güvenli bir şekilde yapılmazsa sistem kırılabilir Her iki taraf da aynı gizli anahtara sahip olmalıdır. VPN bağlantılarında kullanılır.

- **Blok Şifreleme**: Veriyi sabit uzunlukta bloklara bölerek şifreler. Örneğin; DES, 3DES, AES.
- **Akış Şifreleme**: Veriyi tek tek bitler veya baytlar halinde şifreler. Örneğin; RC4, Salsa20, ChaCha20.
- **Hybrid Yaklaşımlar**: Akış ve blok şifrelemenin avantajlarını birleştirir. Çoğu modern protokolde hibrit modeller tercih edilir.

4.1 DES (Data Encryption Standard)

DES, 1970'lerde IBM tarafından geliştirilmiş ve ABD hükümeti tarafından standart olarak kabul edilmiş bir blok şifreleme algoritmasıdır. Veriyi 64 bitlik bloklar halinde işler ve 56 bit bir anahtar kullanır. Her bloğun son bit parity için kullanıldığından şifreleme işleminde kullanılmaz, bu yüzden 56 bit anahtar kullanır. 56-bit anahtar uzunluğu kısa olduğundan brute-force saldırılarına karşı zayıftır. Şifreleme ve şifre çözme için Feistel yapısını kullanır ve şifreleme işlemi toplamda 16 tur gerçekleştirir. Her turda farklı bir alt anahtar kullanarak veriyi işler.

Açık anahtarlı şifrelemenin kurucularından Whitfield Diffie ve Martin Hellman, DES'in ticari amaçlar için güvenli olduğunu fakat istihbat için kullanılmaması gerektiğini, algoritmanın saldırılara karşı dayanıksız olduğunu ifade ettiler. Anahtar boyutunun kısa olması ve algoritma içerisindeki S kutularının güvenirliliğinin az olmasından söz ettiler. Buna rağmen DES algoritması kullanılmaya devam edildi. 1977 yılında Whitfield Diffie ve Martin Hellman, maliyeti 20 milyon dolar olan "DES-Crack" isimli DES algoritmasının anahtarını 1 günde bulabilecek bir makine önerdiler. 1993 yılında Michael Wiener, maliyeti 1 milyon dolar olan ve DES algoritmasının anahtarını 7 saatte bulabilecek bir makine önerdi. Her iki çalışma da yüksek maliyetten dolayı gerçekleşmedi. 1990 yılında Eli Bilham ve Adi Shamir diferensiyel kriptanaliz metodu ile DES üzerinde bir saldırı gerçekleştirdiler fakat bu girişim başarısız oldu. 1993 yılında Mitsuru Matsui DES algoritması üzerinde lineer kriptanaliz yöntemi tasarladı. Bu, DES üzerinde uygulanan ilk deneysel kriptanaliz yöntemi oldu. Daha sonra ise, DES algoritması için özel olarak Davies saldırısı yöntemi tasarlandı. Donald

Davies tarafından önerilen bu saldırı, Eli Bilham ve Biryukov tarafından geliştirildi. Saldırı, 2^{50} hesaplama maliyetine sahipti. 1998 yılında EFF, 250 bin dolar maliyetle "Deep Crack" isminde, DES algoritmasının tüm anahtar ihtimallerini deneyen bir makine üretti. Saniyede 90 milyar anahtarı test edebilen ve her biri 64 mikroçip içeren 27 karttan oluşan bu makine, 56 bitlik bir anahtar ile şifrelenmiş bir metni 56 saatte kırdı. Bu olay, DES algoritmasına olan güvenin sarsılmasına yol açtı. Bu olaydan sonra 1999 yılında Triple DES (3DES) yöntemi geliştirildi.

4.1.1 Encryption

1. Veri, sabit bir permütasyon tablosu (initial permutation) kullanılarak yeniden düzenlenir.
2. Veri, Feistel yapısı ile 16 tur işlenir. Her turda, veri L_i (sol) ve R_i (sağ) olmak üzere 32-bitlik iki yarıya bölünür. Sağ taraf, bir fonksiyon ve bir alt anahtarla işlenir, ardından sol tarafla XOR yapılır. Sol ve sağ taraf yer değiştirir.
3. İşlenen veri (final permutation), başlangıç permütasyonunun tersine göre düzenlenir.
4. Şifreleme işlemi tamamlanır.

4.1.2 Python Kodu

```
from Crypto.Cipher import DES
from Crypto.Util.Padding import pad, unpad

def des_encrypt(plaintext, key):
    if len(key) != 8:
        raise ValueError("")

    cipher = DES.new(key, DES.MODE_ECB)
    padded_text = pad(plaintext.encode(), 8)
    encrypted = cipher.encrypt(padded_text)
    return encrypted

def des_decrypt(ciphertext, key):
    if len(key) != 8:
        raise ValueError("")

    cipher = DES.new(key, DES.MODE_ECB)
    decrypted_padded_text = cipher.decrypt(ciphertext)
    decrypted = unpad(decrypted_padded_text, 8)
    return decrypted.decode()

plaintext = "Hello, World!"
```

```
key = b"secretky"
encrypted = des_encrypt(plaintext, key)
decrypted = des_decrypt(encrypted, key)
print("Encrypted:", encrypted)
print("Decrypted:", decrypted)
```

4.2 3DES (Triple Data Encryption Standard)

3DES, DES algoritmasının güvenlik açıklarını kapatmak için geliştirilmiş bir genişletilmiş versiyonudur. 3 farklı anahtar veya aynı anahtarın farklı kombinasyonlarını kullanarak 3 adımda şifreleme ve şifre çözme işlemleri gerçekleştirir. Bir veri bloğu üzerinde DES algoritması şu sırayla üç kez uygulanır:

1. İlk anahtar ile DES şifrelemesi uygulanır.
2. İkinci anahtar ile DES şifre çözmesi uygulanır.
3. Üçüncü anahtar ile tekrar DES şifrelemesi uygulanır.

$$C = E_{K_3}(D_{K_2}(E_{K_1}(P)))$$

Burada, P, düz metin (plaintext); C, şifrelenmiş metin (ciphertext); E, DES şifreleme işlemi; D, DES şifre çözme işlemidir.

4.2.1 Python Kodu

```
from Crypto.Cipher import DES3
from Crypto.Util.Padding import pad, unpad
from Crypto.Random import get_random_bytes

def triple_des_encrypt(plaintext, key):
    cipher = DES3.new(key, DES3.MODE_ECB)
    padded_text = pad(plaintext.encode(), 8)
    encrypted = cipher.encrypt(padded_text)
    return encrypted

def triple_des_decrypt(encrypted, key):
    cipher = DES3.new(key, DES3.MODE_ECB)
    decrypted_padded_text = cipher.decrypt(encrypted)
    decrypted = unpad(decrypted_padded_text, 8)
    return decrypted.decode()

key = get_random_bytes(24)
plaintext = "Hello, World!"
DES3.adjust_key_parity(key)
encrypted = triple_des_encrypt(plaintext, key)
decrypted = triple_des_decrypt(encrypted, key)
print("Encrypted:", encrypted)
print("Decrypted:", decrypted)
```

4.3 AES (Advanced Encryption Standard)

AES (Advanced Encryption Standard), 2001 yılında Rijndael algoritması temel alınarak geliştirilmiş modern ve güvenli bir blok şifreleme algoritmasıdır. Veri güvenliği için dünya çapında yaygın olarak kullanılan bir standarttır. AES, sabit bir 128 bit blok boyutunda çalışır. Farklı güvenlik seviyeleri için 3 farklı anahtar uzunluğunu destekler: 128 bit (10 tur), 192 bit (12 tur) ve 256 bit (14 tur). AES, hem donanım hem de yazılım ortamlarında hızlıdır ve brute-force saldırılarına karşı oldukça dayanıklıdır.

4.3.1 Encryption

1. Düz metin, ilk olarak anahtar ile XOR işlemine girer.
2. Her tür şu 4 adımdan oluşur:
   - **SubBytes**: Her bayt, S-box adı verilern bir tablo yardımıyla bir başkasıyla değiştirilir.
   - **ShiftRows**: Veri bloğundaki satırlar belirli bir düzene göre kaydırılır.
   - **MixColumns**: Sütunlar matematiksel bir işlemle karıştırılır.
   - **AddRoundKey**: Veri bloğu, o tura ait alt anahtar ile XOR yapılır.
3. Son turda, SubBytes, ShiftRows ve AddRoundKey işlemleri yapılır. MixColumns atlanır.
4. Tüm turlar tamamlandıktan sonra şifreli metin elde edilir.

4.3.2 Python Kodu

```
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad, unpad
from Crypto.Random import get_random_bytes

def aes_encrypt(plaintext, key):
    cipher = AES.new(key, AES.MODE_CBC)
    iv = cipher.iv
    padded_text = pad(plaintext.encode(), AES.block_size)
    encrypted = cipher.encrypt(padded_text)
    return iv + encrypted

def aes_decrypt(encrypted, key):
    iv = encrypted[:AES.block_size]
    actual_encrypted_text = encrypted[AES.block_size:]
    cipher = AES.new(key, AES.MODE_CBC, iv)
    decrypted_padded_text = cipher.decrypt(actual_encrypted_text)
```

```
    decrypted = unpad(decrypted_padded_text, AES.block_size)
    return decrypted.decode()

key = get_random_bytes(24)
plaintext = "Hello, World!"
encrypted = triple_des_encrypt(plaintext, key)
decrypted = triple_des_decrypt(encrypted, key)
print("Encrypted:", encrypted)
print("Decrypted:", decrypted)
```

5 Açık Anahtarlı Şifreleme (Asymmetric Encryption)

Asimetrik şifrelemede, birbiriyle matematiksel olarak ilişkili iki farklı anahtar kullanılır: açık anahtar ve özel anahtar. Açık anahtar şifreleme için kullanılır. Özel anahtar şifre çözme için kullanılır. Gönderici, alıcının açık anahtarını kullanarak veriyi şifreler. Alıcı, yalnızca kendisinde bulunan özel anahtarını kullanarak şifreli veriyi çözer. Anahtar paylaşımı daha güvenlidir; açık anahtar herkese açık olabilir. Dijital imzalar ve kimlik doğrulama gibi güvenlik mekanizmalarında kullanılır. Kripto para cüzdanlarında vs ssl/tls protokollerinde anahtar değişiminde kullanılır. Daha yavaştır. Örneğin; RSA, ECC, DSA, ElGamal.

5.1 RSA (Rivest-Shamir-Adleman)

1977 yılında Ron Rivest, Adi Shamir ve Leonard Adleman tarafından geliştirilmiştir. RSA, büyük asal sayıların çarpanlarına ayrılmasının zorluğuna dayanan bir algoritmadır. RSA, hem şifreleme hem de dijital imzalama için kullanılır.

5.1.1 Encryption

İlk olarak anahtar çifti oluşturulur.

1. p ve q olmak üzere iki büyük asal sayı seçilir.
2. Modülüs hesaplanır: $n = p \cdot q$. Bu n, açık ve özel anahtar için ortak kullanılır.
3. Euler totient fonksiyonu hesaplanır: $\phi(n) = (p-1) \cdot (q-1)$.
4. Açık bir sayı (ϵ) seçilir, $1 < \epsilon < \phi(n)$ olacak şekilde, ϵ ve $\phi(n)$ aralarında asal olmalıdır.
5. Özel anahtar (d) hesaplanır: $(d \cdot \epsilon) \bmod \phi(n) = 1$ olacak şekilde bulunur. Bu işlem, modüler ters alma işlemidir.

Mesajı şifrelemek için: $C = M^{\epsilon} \bmod n$ formülü kullanılır. Burada, C şifrelenmiş metin, M orijinal metindir. Şifre çözmek için: $M = C^{d} \bmod n$ formülü kullanılır. Burada d, özel anahtardır.

5.1.2 Python Kodu

```
import random
import math
from sympy import mod_inverse, isprime
```

```
def random_prime(bit_size):
    while True:
        num = random.getrandbits(bit_size)
        if isprime(num):
            return num

def generate_keys(bit_size=16):
    p = random_prime(bit_size)
    q = random_price(bit_size)
    n = p * q
    phi = (p - 1) * (q - 1)

    e = random.randint(2, phi - 1)
    while math.gcd(e, phi) != 1:
        e = random.randint(2, phi - 1)

    d = mod_inverse(e, phi)
    return (e, n), (d, n)

def rsa_encrypt(plaintext, public_key):
    e, n = public_key
    cipher = [pow(ord(char), e, n) for char in plaintext]
    return cipher

def rsa_decrypt(cipher, private_key):
    d, n = private_key
    decrypted = ''.join([chr(pow(char, d, n)) for char in cipher])
    return decrypted

public_key, private_key = generate_keys()
plaintext = "Hello, World!"
cipher = rsa_encrypt(plaintext, public_key)
decrypted = rsa_decrypt(cipher, private_key)
print("Encrypted:", cipher)
print("Decrypted:", decrypted)
```

5.2 ECC (Elliptic Curve Cryptography)

ECC, eliptik eğri matematiğini kullanır. RSA'ya kıyasla çok daha küçük anahtar boyutlarıyla aynı güvenlik seviyesini sunar. Bu, ECC'yi sınırlı işlem gücü ve bellek gereksinimleri olan cihazlar için ideal hale getirir. Eliptik eğri denklemi:

$$y^2 = x^3 + ax + b \bmod p$$

Bu eğri, iki parametreye ve bir modülüse bağlıdır. Şifreleme işlemleri, bu eğri üzerindeki noktalar arasında yapılır.

5.2.1 Encryption

Parametreler ve anahtarlar oluşturulur.

1. Eliptik eğri denklemi parametreleri belirlenir.
2. Eğri üzerinde taban noktası (G) seçilir.
3. Özel anahtar (d) seçilir. Rastgele bir sayı seçilir: $1 \leq d < n$. Burada n, eğrinin düzenidir.
4. Açık anahtar hesaplanır: $Q = d \cdot G$.

Mesaj bir noktaya (M) dönüştürülür. Rastgele bir sayı (k) seçilir: $1 \leq k < n$. Şifrelenmiş mesaj çifti (C_1, C_2): $C_1 = k \cdot G$ ve $C_2 = M + k \cdot Q$. Şifre çözerken özel anahtar (d) şu şekilde hesaplanır: $M = C_2 - d \cdot C_1$.

5.3 DSA (Digital Signature Algorithm)

DSA, NIST tarafından dijital imzalar için geliştirilmiş bir açık anahtarlı şifreleme algoritmasıdır. DSA, dijital bir mesajın kaynağını doğrulamak ve mesajın değiştirilip değiştirilmediğini kontrol etmek için kullanılır. Bu algoritma, şifreleme için değil, dijital imzalama ve doğrulama işlemleri için tasarlanmıştır.

5.3.1 Encryption

Anahtarlar oluşturulur.

1. Büyük bir asal sayı (p) seçilir.
2. Bir asal sayı (q) seçilir. Bu, $p-1$'in çarpanı olan daha küçük bir asal sayıdır.
3. g tabanı hesaplanır: $g = h^{\frac{p-1}{q}} \bmod p$, burada h rastgele seçilen bir sayı ve $1 < h < p-1$.
4. Özel anahtar (x) seçilir: rastgele bir sayı seçilir $1 \leq x < q$.
5. Açık anahtar (y) hesaplanır: $y = g^x \bmod p$.

Mesaj m için dijital imza oluşturulur.

1. Rastgele bir sayı (k) seçilir: $1 \leq k < q$ ve k ile q aralarında asal olmalıdır.
2. r değeri hesaplanır: $r = (g^k \bmod p) \bmod q$.
3. s değeri hesaplanır: $s = k^{-1} \cdot (H(m) + x \cdot r) \bmod q$. Burada, $H(m)$ mesajın hash değeridir, k^{-1}, k'nın modüler tersidir.

İmza (r, s) çiftidir. Dijital imza doğrulanır.

1. w değeri hesaplanır: $w = s^{-1} \bmod q$.
2. u_1 ve u_2 değerleri hesaplanır: $u_1 = H(m) \cdot w \bmod q$ ve $u_2 = r \cdot w \bmod q$.
3. v değeri hesaplanır: $v = (g^{u_1} \cdot y^{u_2} \bmod p) \bmod q$.
4. Eğer $v = r$ ise imza geçerlidir, aksi halde geçersizdir.

5.3.2 Python Kodu

```
import hashlib
from sympy import mod_inverse

p = 23
x = 6
q = 11
g = 2
k = 7

def generate_keys(g, x, p):
    y = pow(g, x, p)
    return x, y

def sign_message(plaintext, private_key, k, g, p, q):
    r = pow(g, k, p) % q
    k_inv = mod_inverse(k, q)
    hash_value = int(hashlib.sha256(plaintext.encode()).hexdigest(), 16)
        % q
    s = (k_inv * (hash_value + private_key * r)) % q
    return r, s

def verify_signature(plaintext, signature, public_key, g, p, q):
    r, s = signature
    if not (0 < r < q and 0 < s < q):
        return False

    w = mod_inverse(s, q)
    hash_value = int(hashlib.sha256(plaintext.encode()).hexdigest(), 16)
        % q
    u1 = (hash_value * w) % q
    u2 = (r * w) % q
    v = ((pow(g, u1, p) * pow(public_key, u2, p)) % p) % q
    return v == r

private_key, public_key = generate_keys(g, x, p)
plaintext = "Hello, World!"
signature = sign_message(plaintext, private_key, k, g, p, q)
is_valid = verify_signature(plaintext, signature, public_key, g, p, q)
print("Digital Signature:", signature)
print("Is Valid:", is_valid)
```

5.4 ElGamal

ElGamal, açık anahtarlı şifreleme için kullanılan bir algoritmadır. Hem şifreleme hem de dijital imzalama işlemleri için kullanılabilir. ElGamal, Diffie-Hellman anahtar değişimi üzerine kuruludur ve büyük asal sayıların modüler aritmetiği üzerinde çalışır.

5.4.1 Encrpytion

Anahtarlar oluşturulur.

1. Büyük bir asal sayı (p) seçilir.
2. g (taban) seçilir: rastgele bir sayı olup $1 < g < p$.
3. Özel anahtar (x) seçilir: rastgele bir sayı olup $1 < x < p - 1$.
4. Açık anahtar (y) hesaplanır: $y = g^x \bmod p$.

Mesaj m için şifreleme yapılır.

1. m, p ile uyumlu olacak şekilde bir tam sayıya dönüştürülür.
2. Rastgele bir sayı (k) seçilir: $1 < k < p - 1$.
3. Şifrelenmiş mesaj: $c_1 = g^k \bmod p$ ve $c_2 = m \cdot y^k \bmod p$.

Şifre çözme ise

1. $s = c_1^x \bmod p$.
2. $m = c_2 \cdot s^{-1} \bmod p$, burada s^{-1}, s'nin modüler tersidir.

5.4.2 Python Kodu

```
import secrets
from sympy import mod_inverse

g = 5
p = 97

def generate_keys(g, p):
    x = secrets.randbelow(p - 1) + 1
    y = pow(g, x, p)
    return (p, g, y), x

def elgamal_encrypt(plaintext, public_key):
    p, g, y = public_key
    m = plaintext % p
    k = secrets.randbelow(p - 1) + 1
```

```
    c1 = pow(g, k, p)
    c2 = (m * pow(y, k, p)) % p
    return c1, c2

def elgamal_decrypt(ciphertext, private_key, public_key):
    p, _, _ = public_key
    c1, c2 = ciphertext
    s = pow(c1, private_key, p)
    s_inv = mod_inverse(s, p)
    m = (c2 * s_inv) % p
    return m

public_key, private_key = generate_keys(g, p)
plaintext = 42
ciphertext = elgamal_encrypt(message, public_key)
decrypted = elgamal_decrypt(ciphertext, private_key, public_key)
print("Encrypted:", ciphertext)
print("Decrypted:", decrypted_message)
```

5.5 Paillier Cipher

Paillier Cipher, 1999 yılında Pascal Paillier tarafından geliştirilmiştir. Bu algoritma, homomorfik şifreleme özelliği sayesinde, şifreli veriler üzerinde işlem yapmayı mümkün kılar. Bu özellik, gizli veriler üzerinde matematiksel işlemlerin yapılmasını gerektiren uygulamalarda kullanılır.

5.5.1 Encrpytion

Anahtarlar oluşturulur.

1. İki büyük asal sayı (p ve q) seçilir.
2. Modül (n) hesaplanır: $n = p \cdot q$.
3. Lambda (λ) değeri hesaplanır (carmichael fonksiyonu): $\lambda = \text{lcm}(p-1, q-1)$, burada lcm en küçük olan ortak kattır.
4. Bir yardımcı fonksiyon $L(u)$ hesaplanır: $L(u) = \frac{u-1}{n}$.
5. g, n^2 modülünde bir sayı seçilir: $\gcd(L(g^\lambda \bmod n^2), n) = 1$
6. Açık anahtar (n, g)'dir. Özel anahtar ise (λ, μ). Burada $\mu = L(g^\lambda \bmod n^2)^{-1} \bmod n$.

Mesaj şifrelenir.

1. Rastgele bir sayı (r) seçilir: $1 < r < n$.
2. Şifrelenmiş mesaj (c): $c = (g^m \cdot r^n) \bmod n^2$.

Şifre çözme ise:

1. $u = c^\lambda \bmod n^2$.
2. $m = L(u) \cdot \mu \bmod n$.

5.6 Diffie-Hellman Key Exchange

Diffie-Hellman Anahtar Değişimi (DHKE), iki tarafın güvenli bir şekilde ortak bir şifreleme anahtarı oluşturmasını sağlayan açık anahtarlı bir protokoldür. Bu protokol, özellikle güvenli bir kanal üzerinden iletişim kurulmadan önce, şifreleme anahtarlarının değiştirilmesini mümkün kılar.

5.6.1 Anahtar Değişimi

1. Her iki taraf da büyük bir asal sayı p (modül) ve bir taban g seçer. Bu parametreler açıkça paylaşılır.
2. Alice ve Bob, gizli özel anahtarlar seçer. Alice için bu anahtar a, Bob içinse b olur.
3. Açık anahtarlar hesaplanır. Alice, $A = g^a \bmod p$ hesaplar ve Bob'a gönderir. Bob, $B = g^b \bmod p$ hesaplar ve Alice'e gönderir.
4. Ortak anahtarlar hesaplanır. Alice, $K_A = B^a \bmod p$ hesaplar. Bob, $K_B = A^b \bmod p$ hesaplar.
5. Her iki tarafta aynı ortak anahtarı elde eder. Bu anahtar daha sonra şifreleme için kullanılır.

5.6.2 Python Kodu

```
def diffie_hellman(p=23, g=5, a=6, b=15):
    A = pow(g, a, p)
    B = pow(g, b, p)
    print(f"Alice's public key: {A}")
    print(f"Bob's public key: {B}")

    K_A = pow(B, a, p)
    K_B = pow(A, b, p)

    print(f"Alice's shared key: {K_A}")
    print(f"Bob's shared key: {K_B}")

    if K_A == K_B:
        print("The shared key has been created successfully.")
    else:
        print("Shared key did not match.")
```

6 Kriptoanaliz Yöntemleri

6.1 Frekans Analizi

Frekans analizi, şifrelenmiş metindeki harflerin veya sembollerin ne sıklıkla tekrar ettiğini inceleyerek şifreyi çözmeyi amaçlar. Monoalfabetik şifreleme (örneğin caesar cipher) gibi basit şifreleme yöntemlerini kırmak için kullanılır. Temel varsayımı, doğal bir dilde belirli harflerin veya sembollerin diğerlerin daha sık kullanılmasıdır. Örneğin, İngilizce'de en sık kullanılan harfler "e,t,a,o,i,n" iken, Türkçe'de en sık kullanılan harfler "a,e,i,n,r" harfleridir.

6.1.1 Çalışma Adımları

1. Şifreli metindeki her harfin kaç kere geçtiği hesaplanır.
2. Harflerin frekansları metindeki toplam harf sayısına bölünerek yüzde oranları hesaplanır.
3. Şifreli metindeki frekans dağılımı bilinen bir dilin frekansları ile karşılaştırılır.
4. Harf eşleştirmeleri yapılarak metin çözülür.

6.1.2 Python Kodu

```
from collections import Counter

turkish_frequencies = {
}

encrypted = "bmrfs lbsbdb"

def frequency_analysis(text, language_frequencies):
    text = text.lower()
    letter_counts = Counter(filter(str.isalpha, text))
    total_letters = sum(letter_counts.values())
    encrypted_frequencies = {letter: (count / total_letters) * 100
                             for letter, count in letter_counts.items()}

    for letter, freq in sorted(encrypted_frequencies.items(), key=lambda
        x: x[1], reverse=True):
        match = sorted(language_frequencies.items(), key=lambda x:
            abs(x[1] - freq))[0]
        print(f"Encrypted Letter: {letter}, Frequency: {freq:.2f}%,
            Matched: {match[0]}")

frequency_analysis(encrypted, turkish_frequencies)
```

6.2 Kasiski Method

Kasiski yöntemi, polialfabetik şifreleme yöntemlerini (örneğin vigenere cipher) kırmak için kullanılır. Bu yöntem, şifreli metinde tekrar eden harf gruplarını analiz ederek şifreleme anahtarının uzunluğunu tahmin etmeye çalışır. Polialfabetik şifreleme yöntemlerinde, aynı düz metin harfi, farklı şifreleme anahtarlarına bağlı olarak farklı harflerle şifrelenir. Ancak, aynı anahtar tekrarlandığı için belirli harf grupları benzer şekilde şifrelenir. Kasiski yöntemi bu tekrarlardan yararlanır.

6.2.1 Çalışma Adımları

1. Şifreli metinde aynı olan 3 veya daha fazla harf uzunluğunda tekrar eden diziler belirlenir.
2. Tekrar eden diziler arasındaki mesafeler bulunur.
3. Bu mesafelerin ortak bölenleri, anahtar uzunluğu için aday değerleri verir.
4. En sık görülen ortak bölen, anahtar uzunluğunu tahmin etmek için kullanılır.

6.2.2 Python Kodu

```
from collections import defaultdict
from functools import reduce
from math import gcd

def kasiski_analysis(text, sequence_length=3):
    sequences = defaultdict(list)
    for i in range(len(text) - sequence_length + 1):
        seq = text[i:i + sequence_length]
        sequences[seq].append(i)

    repeating_sequences = {seq: indices for seq, indices in
        sequences.items()
                        if len(indices) > 1}

    distances = []
    for indices in repeating_sequences.values():
        for i in range(len(indices) - 1):
            distances.append(indices[i + 1] - indices[i])

    if distances:
        key_length = reduce(gcd, distances)
        print("Key Length:", key_length)
        return key_length
    else:
```

```
        return None

encrypted_text = "ABABXYZXYZABABXYZXYZ"
kasiski_analysis(encrypted_text)
```

6.3 Known-Plaintext Analysis (KPA)

KPA yönteminde saldırgan, şifrelenmiş metni (cipherText) ve buna karşılık gelen düz metni (plaintext) bilir. Bu bilgi, şifreleme algoritmasını veya anahtarı bulmak için kullanılır. Aynı şifreleme anahtarı kullanılarak şifrelenmiş diğer şifreli metinleri çözmek için etkili bir yöntemdir. Bu yöntem, blok şifreleme, akış şifreleme veya diğer şifreleme tekniklerinin analizinde kullanılır.

6.3.1 Python Kodu

```
plaintext = "HELLO"
xor_ciphertext = [75, 80, 93, 93, 82]

def extract_key(plaintext, ciphertext):
    key = []
    for p, c in zip(plaintext, ciphertext):
        key.append(ord(p) ^ c)

    return key

def kpa_analysis(ciphertext, key):
    decrypted = ""
    for c, k in zip(ciphertext, key):
        decrypted += chr(c ^ k)

    return decrypted

key = extract_key(plaintext, xor_ciphertext)
print("Key:", key)
decrypted = kpa_analysis(xor_ciphertext, key)
print("Decrypted:", decrypted)
new_ciphertext = [87, 82, 85, 85, 88]
decrypted_new_plaintext = kpa_analysis(new_ciphertext, key)
print("New Decrypted", decrypted_new_plaintext)
```

6.4 Chosen-Plaintext Analysis (CPA)

CPA, bir saldırganın düz metni seçebilmesine sahip olduğu ve seçtiği düz metne karşılık gelen şifreli metini elde edebildiği bir yöntemdir. Amaç, şifreleme algoritmasını veya anahtarı çözerek, başka metinleri çözmektir. Blok şifreleme ve akış şifreleme yöntemlerinin zayıflıklarını analiz etmek için kullanılır. Seçilen girdiye göre sistemin nasıl çıktı oluşturduğunu anlamaya dayanır.

6.4.1 Python Kodu

```
def xor_encrypt(plaintext, key):
    ciphertext = []
    for i, char in enumerate(plaintext):
        ciphertext.append(ord(char) ^ key[i % len(key)])

    return ciphertext

def xor_decrypt(ciphertext, key):
    plaintext = ""
    for i, char in enumerate(ciphertext):
        plaintext += chr(char ^ key[i % len(key)])

    return plaintext

def cpa_analysis(plaintext_list, ciphertext_list):
    key_guess = []
    for i in range(len(plaintext_list[0])):
        key_char = ord(plaintext_list[0][i]) ^ ciphertext_list[0][i]
        key_guess.append(key_char)

    return key_guess

key = [42, 17, 56]
plaintext_list = ["HELLO", "WORLD"]
ciphertext_list = [xor_encrypt(plaintext, key) for plaintext in
    plaintext_list]
guessed_key = cpa_analysis(plaintext_list, ciphertext_list)
print("Original Key:", key)
print("Guessed Key:", guessed_key)
for ciphertext in ciphertext_list:
    decrypted = xor_decrypt(ciphertext, guessed_key)
    print("Decrypted:", decrypted)
```

6.5 Ciphertext-Only Analysis (COA)

COA'da saldırgan, yalnızca şifreli metinlere (ciphertext) sahiptir, elinde başka bir bilgi yoktur. Amaç, şifreleme algoritmasını çözmek, düz metni (plaintext) geri elde etmek veya kullanılan anahtarı bulmaktır. COA, caesar cipher, substitution cipher gibi basit şifreleme algoritmalarında etkilidir. COA'nın başarılı olabilmesi için:

- Şifreleme algoritmasında zayıflıklar olması gerekir.
- Şifreli metinlerde istatistiksel düzenler (örneğin harf frekansı) bulunmalıdır.
- Uzun veya yinelenen metinler gibi analiz için kullanılabilecek özellikler içermelidir.

6.5.1 Python Kodu

```
from collections import Counter

english_letter_freq = {
    'E': 12.7, 'T': 9.1, 'A': 8.2, 'O': 7.5, 'I': 7.0,
    'N': 6.7, 'S': 6.3, 'H': 6.1, 'R': 6.0, 'D': 4.3,
    'L': 4.0, 'C': 2.8, 'U': 2.8, 'M': 2.4, 'W': 2.4,
    'F': 2.2, 'G': 2.0, 'Y': 2.0, 'P': 1.9, 'B': 1.5,
    'V': 1.0, 'K': 0.8, 'J': 0.2, 'X': 0.2, 'Q': 0.1, 'Z': 0.1
}

def coa_analysis(ciphertext, language_frequencies):
    ciphertext = ciphertext.upper()
    letter_counts = Counter(ciphertext)
    total_letters = sum(letter_counts.values())
    ciphertext_freq = {c: (count / total_letters) * 100 for c, count in
        letter_counts.items()
                    if c.isalpha()}
    most_common_letter = max(ciphertext_freq, key=ciphertext_freq.get)
    v = list(language_frequencies.values())
    k = list(language_frequencies.keys())
    assumed_most_common = k[v.index(max(v))]
    shift = (ord(most_common_letter) - ord(assumed_most_common)) % 26
    plaintext = ""
    for char in ciphertext:
        if char.isalpha():
            offset = 65 if char.isupper() else 97
            plaintext += chr((ord(char) - offset - shift) % 26 + offset)
        else:
            plaintext += char

    return plaintext, shift
```

```
plaintext = "HELLO WORLD"
ciphertext = "KHOOR ZRUOG" # Caesar (key = 3)
decrypted_text, guessed_key = caesar_coa(ciphertext)
print("Encrypted:", ciphertext)
print("Decrypted:", decrypted_text)
print("Guessed Key:", guessed_key)
```

6.6 Man-in-the-Middle (MITM) Attack

Man-in-the-Middle (MITM) saldırısı, iki taraf arasındaki iletişimin saldırgan tarafından gizlice dinlendiği, değiştirildiği veya yönlendirildiği bir tür kriptanaliz veya güvenlik ihlali yöntemidir. Saldırgan, iletişim hattına girerek, iki tarafın birbirlerine doğrudan bağlandığını düşünmesini sağlar. Bu sırada:

- Saldırgan, iki taraf arasındaki mesajları dinleyebilir.
- Mesajları değiştirebilir.
- İletişimi kesebilir veya sahte mesajlar ekleyebilir.

6.7 Adaptive Chosen-Plaintext Analysis (ACPA)

ACPA, Chosen-Plaintext Analysis (CPA) yönteminin bir uzantısıdır. Bu yöntemde saldırgan, belirli metinleri (plaintext) şifreleme algoritmasına gönderir ve şifrelenmiş sonuçları (ciphertext) alır. Şifreleme algoritması hakkında öğrendiklerine dayanarak, yeni metinler seçer ve bunların şifrelenmiş çıktısını incelemeye devam eder.

6.8 Birthday Attack

Birthday Attack, bir hash fonksiyonunda çakışmaları bulmaya çalışan bir saldırı yöntemidir. Bu yöntem, doğum günü paradoksundan yararlanır. Doğum günü paradoksu, bir grupta iki kişinin aynı doğum gününe sahip olma olasılığının beklenenden daha yüksek olduğunu belirtir. Eğer hash fonksiyonu yeterince güçlü değilse, bu saldırıyla çakışma bulunabilir.

6.8.1 Çalışma Adımları

1. Bir hash fonksiyonu, bir verinin özet değerini üretir. Bir hash fonksiyonunun çıkış aralığı N ise, iki farklı girdinin aynı hash değerine sahip olma olasılığı yaklaşık olarak $\sqrt{N}$ girişte ortaya çıkar.

2. Rastgele birçok giriş oluşturulur ve bu girişlerin hash değeri hesaplanır. Hash değerleri karşılaştırılarak çakışma aranır. Çakışma bulunduğunda saldırı başarılı olur.

6.8.2 Python Kodu

```
import hashlib
import random
import string

def birthday_attack(hash_function, num_attempts=10000, length=8):
    hashes = {}
    for _ in range(num_attempts):
        random_data = ''.join(random.choice(string.ascii_letters +
            string.digits) for _ in range(length))
        hash_value = hashlib.md5(random_data.encode()).hexdigest()

        if hash_value in hashes:
            print(f"Found!")
            print(f"1. Data: {hashes[hash_value]}")
            print(f"2. Data: {random_data}")
            print(f"Hash: {hash_value}")
        else:
            hashes[hash_value] = random_data
    print("Not found.")
```

6.9 Side-Channel Attack

Side-Channel Attack (Yan Kanal Saldırısı), bir kriptografik algoritmanın matematiksel veya mantıksal kusurlarını hedeflemek yerine, fiziksel uygulamasından elde edilen yan bilgileri kullanarak yapılan bir saldırı türüdür. Bu saldırılar, bir cihazın çalışması sırasında ortaya çıkan enerji tüketimi, elektromanyetik yayılım, işlem süresi veya akustik sinyaller gibi yan bilgileri analiz eder.

6.10 Brute-Force Attack

Brute-Force Attack (Kaba Kuvvet Saldırısı), bir şifreleme sistemini kırmak için olası tüm anahtar veya parola kombinasyonlarının sistematik olarak denenmesi yöntemidir. Saldırgan, doğru kombinasyonu bulana kadar tüm olası değerleri dener. Bu yöntem, şifreleme mekanizmalarının zayıflığından veya zayıf parolalardan faydalanır.

6.10.1 Python Kodu

```
def caesar_brute_force(ciphertext):
    for shift in range(1, 26):
        decrypted = ""
        for char in ciphertext:
            if char.isalpha():
                offset = 65 if char.isupper() else 97
                decrypted += chr((ord(char) - offset - shift) % 26 +
                    offset)
            else:
                decrypted += char

        print(f"Shift: {shift}, Decrypted: {decrypted}")

ciphertext = "bmqfs lbsbdb"
caesar_brute_force(ciphertext)
```

6.11 Differential Cryptoanalysis

Differential Cryptanalysis (Diferansiyel Kriptoanaliz), simetrik şifreleme algoritmalarını analiz etmek ve zayıflıklarını bulmak için kullanılan bir yöntemdir. Feistel şifreleme yapıları ve blok şifreleme algoritmaları üzerinde etkili bir analiz yöntemidir. Bu yöntem, şifreleme algoritmasındaki farklılıkların (differentials) şifreleme sürecinde nasıl yayıldığını inceleyerek anahtar bilgisine ulaşmayı amaçlar. İki farklı düz metin arasındaki farkların şifrelenmiş metne nasıl yansıdığını gözlemleyerek saldırgan, anahtar hakkında bilgi elde etmeye çalışır.

6.12 Integral Crytpoanalysis

Integral Cryptanalysis Attack, blok şifreleme algoritmalarında kullanılan bir kriptoanaliz yöntemidir. Bu saldırı türü, algoritmanın yapısındaki kısmi giriş/çıkış bağımsızlıklarını analiz eder. Feistel şifreleme yapıları ve SPN (Substitution-Permutation Network) tabanlı algoritmalar üzerinde kullanılır. Bu yöntem, algoritmanın iç durumlarının belirli bir kısmının değişmez kaldığı durumları tespit ederek şifreleme sürecini incelemeye odaklanır. Integral kriptoanaliz, özellikle algoritmanın birden fazla turuna yayılan aktif bitlerin (active bits) yayılımını analiz eder. Bu analiz sonucunda şifreleme algoritmasının zayıf noktalarını bulmak ve anahtarın bazı bölümlerini veya tamamını tahmin etmek mümkündür.