Kriptoloji Notlar
(www.cyberdatascience.com.tr)

Alper Karaca

January 2024

Contents

1 ———— KRİPTOLOJİ ————

2 Kriptoloji, Kriptografi, Kriptoanaliz

2.1 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.

2.2 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.

2.3 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.

3 Number Theory

Sayılar teorisi, matematiğin pozitif tam sayılar ve onların özellikleri ile ilgilenen bir alt dalıdır. Bu teori, asal sayılar, modüler aritmetik, çarpanlara ayırma, Diophantine denklemleri gibi konuları kapsar. Eski Yunanlılardan günümüze kadar uzanan tarihsel bir geçmişi olan Sayılar Teorisi, soyut matematik alanında temel bir rol oynar ve birçok pratik uygulama bulur. Sayılar Teorisi, özellikle modern kriptosistemlerin temelini oluşturur.

3.0.1 Asal Sayılar

Asal sayılar, yalnızca 1 ve kendisine bölünebilen pozitif tam sayılardır. Sayılar Teorisi'nde asal sayılarla ilgili derinlemesine araştırmalar yapılır. Kriptolojide ise asal sayılar:

- **RSA Algoritması**: RSA, büyük asal sayıların çarpımına dayanır. Bu sistemin güvenliği, büyük sayıları çarpanlarına ayırmanın zorluğuna bağlıdır.
- **Anahtar Üretimi**: Asal sayılar, kriptografik anahtarların üretilmesinde kullanılır.

3.0.2 Modüler Aritmetik

Modüler aritmetik, bir sayının bir modulo (bölümden kalan) içinde nasıl davrandığını inceler. Şifreleme sistemlerinde sıkça kullanılır:

- **Diffie-Hellman Anahtar Değişimi**: Modüler üs alma işlemi sayesinde, iki tarafın ortak bir gizli anahtar oluşturmasına olanak tanır.
- **Eliptik Eğri Kriptografisi**: Modüler aritmetik, eliptik eğrilerin üzerinde yapılan işlemlerde temel bir rol oynar.

3.1 Modular Arithmetic

Modüler aritmetik, sayılar üzerinde işlem yaparken belirli bir modulo (bölümden kalan) kullanarak işlemleri sınırlandırma yöntemidir. Matematikte, bir tam sayının bir modulo'ya göre kalanı incelenir. Bu, sayıları bir çember üzerinde tekrar eden bir sistem gibi düşünülmesini sağlar. a mod n ifadesi, a sayısının n sayısına bölündüğünde kalanı ifade eder. Modüler aritmetik, modern kriptolojinin temelini oluşturur:

- **RSA Algoritması**: Modüler üs alma, RSA'nın hem şifreleme hem de çözme süreçlerinde kullanılır.
- **Diffie-Hellman Anahtar Değişimi**: Güvenli bir şekilde ortak anahtar oluşturmak için modüler aritmetik kullanılır.
- **Eliptik Eğri Kriptografisi (ECC)**: Eliptik eğriler üzerinde yapılan işlemler modüler aritmetiğe dayanır.

3.1.1 Modüler Aritmetiğin İşleyişi

- **Toplama**: $(a + b) \bmod n = ((a \bmod n) + (b \bmod n)) \bmod n$.
- **Çıkarma**: $(a - b) \bmod n = ((a \bmod n) - (b \bmod n)) \bmod n$.
- **Çarpma**: $(a \cdot b) \bmod n = ((a \bmod n) \cdot (b \bmod n)) \bmod n$.
- **Üs Alma**: $a^b \bmod n$, büyük sayılarla çalışırken modüler üs alma yöntemleri kullanılır.

Örneğin $(17 \cdot 13 + 5^3) \bmod 7$ şöyle ifade edilebilir:

$$(17 \cdot 13 + 5^3) \bmod 7 = ((17 \cdot 13 \bmod 7) + (5^3 \bmod 7)) \bmod 7$$

$$(17 \cdot 13 + 5^3) \bmod 7 = ((221 \bmod 7) + (125 \bmod 7)) \bmod 7$$

Buradanda sonuç $(4 + 6) \bmod 7 = 3$ çıkar.

3.1.2 Python Kodu

```python
def modular_arithmetic(a, b, mod, base, exponent):
    multiply_result = a * b
    mod_multiply = multiply_result % mod
    power_result = base ** exponent
    mod_power = power_result % mod
    total = mod_multiply + mod_power
    result = total % mod
    return result
```

3.2 Modular Exponentiation

Modüler üs alma, bir sayının büyük üslerini bir modül üzerinde hesaplama yöntemidir. Amaç, sayıyı doğrudan üs alarak hesaplamak yerine, hesaplama süresini azaltmak ve büyük sayılarla çalışırken performans artışı sağlamaktır. RSA, Diffie-Hellman Anahtar Değişimi, Eliptik Eğri Kriptografisi (ECC) gibi algoritmalarda kritik bir rol oynar.

$$C = (A^B) \bmod N$$

Burada, A taban (base), B üs (exponent), N modulo (bölüm), C kalan temsil eder.

Kriptolojide modüler üs alma:

- **RSA Şifreleme**: Mesajı şifreli hale getirirken kullanılır: $C = M^e \bmod N$.
- **Diffie-Hellman Anahtar Değişimi**: Paylaşılan gizli anahtar, modüler üs alma ile hesaplanır.
- **Eliptik Eğriler**: Eliptik eğri tabanlı kriptografik algoritmalar modüler üs alma prensiplerini kullanır.

Örneğin, $5^{117} \bmod 13$ işlemini hesaplayalım. 5^{117} çok büyük bir sayı olduğundan, doğrudan üs alma verimsiz olur. Bunun yerine modüler üs alma yöntemi kullanılır. Modüler aritmetik kuralına göre:

$$5^{117} \bmod 13 = ((5 \bmod 13)^{117}) \bmod 13$$

$5 \bmod 13 = 5$. Bu yüzden:

$$5^{117} \bmod 13 = 5^{117} \bmod 13$$

Modüler üs alma yöntemi kullanılarak büyük üsler adım adım azaltılır:

- $5^2 = 25; 25 \bmod 13 = 12$.
- $5^4 = (5^2)^2 = 12^2 = 144; 144 \bmod 13 = 1$.
- $5^{117} = (5^4)^2 9 \cdot 5 = 1^{29} \cdot 5 = 5$.
- Sonuç: $5^{117} \bmod 13 = 5$.

3.2.1 Python Kodu

```
def modular_exponentiation(base, exponent, mod):
    result = 1
    base = base % mod
    while exponent > 0:
        if exponent % 2 == 1:
            result = (result * base) % mod
```

```
        exponent = exponent // 2
        base = (base * base) % mod

    return result
```

3.3 GCD and Euclidean Algorithm

GCD (En büyük ortak bölen), iki tam sayının ortak bölenlerinin en büyüğüdür. Öklid algoritması, GCD'yi hesaplamak için kullanılan verimli bir algoritmadır. Algoritma, tekrar eden modüler işlemleri kullanılarak GCD'yi bulur. Şu prensibe dayanır.

$$\text{GCD}(a, b) = \text{GCD}(b, a \bmod b)$$

Burada $a > b$ kabul edilir. İşlemler b sıfıra eşit olana kadar tekrarlanır. Son kalan a, GCD değeridir. Kriptolojide RSA algoritmasında, modüler ters eleman bulma işleminde ve diffie-hellman anahtar değişminde kullanılır.

- **RSA Algoritması**: RSA'da, açık anahtar üssü e seçilirken $\text{GCD}(e, \phi(n)) = 1$ şartı aranır. Burada $\phi(n)$, Euler'in Totient Fonksiyonudur. Bu şart, e'nin $\phi(n)$ ile aralarında asal olmasını sağlar.
- **Modüler Ters Eleman Bulma**: Euclidean Algorithm, Extended Euclidean Algorithm ile birlikte, modüler ters eleman hesaplamak için kullanılır. Modüler ters, kriptografik algoritmalarda şifre çözme anahtarını üretmek için gereklidir.
- **Anahtar Değişimi ve Dijital İmza**: Diffie-Hellman ve dijital imza protokollerinde sayılar arasındaki asal ilişkiler hesaplanırken kullanılır.

Örneğin, $\text{GCD}(48, 18)$ bulalım.

1. $a = 48$ ve $b = 18$'dir.
2. Birinci adım $\text{GCD}(48, 18) = \text{GCD}(18, 48 \bmod 18)$ buradan 12 elde edilir, $a = 18$ ve $b = 12$.
3. İkinci adım $\text{GCD}(18, 12) = \text{GCD}(12, 18 \bmod 12)$ buradan 6 elde edilir, $a = 12$ ve $b = 6$.
4. Üçüncü adım $\text{GCD}(12, 6) = \text{GCD}(6, 12 \bmod 6)$ buradan 0 elde edilir, $a = 6$ ve $b = 0$ ve bitirilir.
5. Sonuç: $\text{GCD}(48, 18) = 6$.

3.3.1 Python Kodu

```python
def gcd(a, b):
    while b != 0:
        a, b = b, a % b

    return a
```

3.4 Relatively Prime (Co-Prime) Numbers

Aralarında asal sayılar, iki sayının ortak bir böleninin sadece 1 olduğu durumları ifade eder. Yani $\text{GCD}(a, b) = 1$ olan iki sayı aralarında asaldır. Relatively Prime Numbers, kriptografide ve sayılar teorisinde şu alanlarda önemli bir rol oynar:

Örneğin 35 ve 64'ün aralarında asal olup olmadığına bakalım.

1. $a = 35$ ve $b = 64$'dir.
2. Birinci adım $\text{GCD}(35, 64) = \text{GCD}(64, 35 \bmod 64)$ buradan 35 elde edilir, $a = 64$ ve $b = 35$.
3. İkinci adım $\text{GCD}(64, 35) = \text{GCD}(35, 65 \bmod 35)$ buradan 29 elde edilir, $a = 35$ ve $b = 29$.
4. Üçüncü adım $\text{GCD}(35, 29) = \text{GCD}(29, 35 \bmod 29)$ buradan 6 elde edilir, $a = 29$ ve $b = 6$.
5. Dördüncü adım $\text{GCD}(29, 6) = \text{GCD}(6, 29 \bmod 6)$ buradan 5 elde edilir, $a = 6$ ve $b = 5$.
6. Dördüncü adım $\text{GCD}(6, 5) = \text{GCD}(5, 6 \bmod 5)$ buradan 1 elde edilir, $a = 5$ ve $b = 1$.
7. Beşinci adım $\text{GCD}(5, 1) = \text{GCD}(1, 5 \bmod 1)$ buradan 1 elde edilir ve bitirilir.
8. Sonuç: $\text{GCD}(35, 64) = 1$ olduğu için 35 ve 64 aralarında asaldır..

3.4.1 Python Kodu

```
def relatively_prime(a, b):
    while b != 0:
        a, b = b, a % b

    return a == 1
```

3.5 Euler's Totient Function (Phi Function)

Euler's Totient Function ($\phi(n)$), pozitif tam sayı n ile aralarında asal olan 1'den n'e kadar olan pozitif tam sayıların sayısını verir.

$$\phi(n) = |\{k \in Z^+ \| 1 \leq k \leq n, \text{GCD}(k,n) = 1\}$$

Yani $\phi(n)$, n'den küçük ve n ile aralarında asal sayıların toplamıdır. $\phi(n)$, modüler sistemlerinde davranışını anlamak için kullanılır. RSA şifrelemede $\phi(n)$, şifreleme ve çözme anahtarlarının üretiminde kullanılır.

Eğer n bir asal sayıysa, $\phi(n) = n-1$, çünkü asal sayının kendisi hariç tüm sayılar onunla aralarında asaldır.

Eğer $n = p_1^{e^1} \cdot p_2^{e^2} ... p_k^{e^k}$ şeklinde asal çarpanlarına ayrılmışsa:

$$\phi(n) = n \cdot (1 - \frac{1}{p_1}) \cdot (1 - \frac{1}{p_2}) ... (1 - \frac{1}{p_k})$$

Örneğin, $n = 36$ için $\phi(36)$'yı hesaplayalım.
İlk olarak 36'yı asal çarpanlarına ayıralım.

$$36 = 2^2 \cdot 3^2$$

Totient fonksiyonuna göre;

$$\phi(n) = n \cdot (1 - \frac{1}{p_1}) \cdot (1 - \frac{1}{p_2})$$

Burada $p_1 = 2$ ve $p_2 = 3$ olur. Hesaplarsak:

$$\phi(36) = 36 \cdot (1 - \frac{1}{2}) \cdot (1 - \frac{1}{3})$$

$$\phi(36) = 36 \cdot \frac{1}{2} \cdot \frac{2}{3}$$

$$\phi(36) = 36 \cdot \frac{1}{3} = 12$$

$\phi(36) = 12$ bulunur. Bu, 36 ile aralarında asal olan 12 pozitif sayı bulunduğunu gösterir.

3.5.1 Python Kodu

```
def phi_function(n):
    result = n
    p = 2
    while p * p <= n:
        if n % p == 0:
            while n % p == 0:
                n //= p

            result -= result // p
```

```
        p += 1

    if n > 1:
        result -= result // n

    return result
```

3.6 Fermat's Little Theorem

Fermat'ın küçük teoremi, asal sayıların modüler aritmetik üzerindeki özel ifadelerini ifade eden bir matematik teoremidir. Eğer p bir asal sayı ve a bir tam sayıysa ve p'nin katı değilse;

$$a^{p-1} \bmod p = 1$$

Eğer p, a'yı bölüyorsa $(p|a)$, bu durumda eşitlik geçerli değildir.

Örneğin, $a = 3$, $p = 7$ için $3^6 \bmod 7$'yi Fermat's Little Theorem ile hesaplayalım. Bu durumda:

$$3^6 \bmod 7 = 1$$

$$(3^3 \bmod 7)^2 \bmod 7$$

Önce $3^3 \bmod 7$'yi hesaplayalım.

$$3^3 = 27$$

$$27 \bmod 7 = 6$$

Daha sonra $(3^3 \bmod 7)^2 \bmod 7$'yi hesaplayalım.

$$6^2 = 36$$

$$36 \bmod 7 = 1$$

Sonuç $3^6 \bmod 7 = 1$'dir.

3.6.1 Python Kodu

```python
def format_mod_exp(a, n, p):
    n = n % (p - 1)
    result = pow(a, n, p)
    return result
```

3.7 Euler's Theorem

Euler Teoremi, modüler aritmetik üzerine kurulmuş bir teoremdir. Asal olmayan modüllerle çalışan problemlerde kullanılır ve Fermat'ın Küçük Teoremi'nin bir genellemesi olarak kabul edilir. Eğer n ve a aralarında asal $\text{GCD}(a, n) = 1$ ise:

$$a^{\phi(n)} \bmod n = 1$$

Burada $\phi(n)$, Euler'in Totient fonksiyonudur ve n'den küçük n ile aralarında asal olan sayıların sayısını ifade eder. Euler Teoremi, asimetrik kriptografide anahtar oluşturma ve şifreleme işlemlerinde kullanılır. Büyük modüllerle çalışmayı kolaylaştırır ve performansı artırır.

Örneğin, $a = 7$, $n = 12$ için $7^{\phi(12)} \bmod 12$'yi Euler's Theorem ile hesaplayalım.

$n = 12$ için Totient fonksiyonu ($\phi(n)$):

$$\phi(n) = n \cdot \prod_{p|n} (1 - \frac{1}{p})$$

Burada p, n'in asal çarpanlarıdır. 12'nin asal çarpanları 2 ve 3'tür:

$$\phi(12) = 12 \cdot (1 - \frac{1}{2}) \cdot (1 - \frac{1}{3})$$

$$\phi(12) = 12 \cdot \frac{1}{2} \cdot \frac{2}{3} = 4$$

$\phi(12) = 4$ kullanarak Euler's Theorem formülünü uygulayalım.

$$7^4 \bmod 12 = 1$$

$$7^2 = 49$$

$$49 \bmod 12 = 1$$

$$(7^2)^2 \bmod 12 = 1$$

3.7.1 Python Kodu

```python
def gcd(a, b):
    while b != 0:
        a, b = b, a % b

    return a

def phi_function(n):
    result = n
    p = 2
    while p * p <= n:
        if n % p == 0:
```

```
            while n % p == 0:
                n //= p

            result -= result // p

        p += 1

    if n > 1:
        result -= result // n

    return result

def euler_mod_exp(a, n):
    phi_n = phi_function(n)
    if gcd(a, n) != 1:
        raise ValueError("")

    return pow(a, phi_n, n)
```

3.8 Primitive Root

İlkel kök, modüler aritmetikte bir tam sayıdır ve belirli bir modül n'ye göre tüm birim grubunu oluşturur. İlkel kök, mod n'ye göre, modüler üs alma yoluyla $\phi(n)$ farklı kalanı oluşturabilen bir sayıdır. Bir tam sayı g, mod n'ye göre bir primitive root (ilkel kök) ise:

$$\{g^1 \bmod n, g^2 \bmod n, ..., g^{\phi(n)} \bmod n\}$$

kümesi, $\phi(n)$'nin tamamını oluşturur. Burada $\phi(n)$, Euler's Totient Function'dır. Primitive Roots, Diffie-Hellman gibi şifreleme protokollerinde kullanılır. Bu protokollerde bir primitive root seçilerek güvenli anahtar alışverişi yapılır. Primitive Roots, Discrete Logarithm probleminin temelini oluşturur. Bu problem, günümüz şifreleme sistemlerinin güvenlik yapısını destekler.

Örneğin, $n = 7$ için tüm ilkel kökleri bulalım. Euler's Totient Function'a göre, $n = 7$ bir asal sayı olduğundan:

$$\phi(7) = 7 - 1 = 6$$

İlkel kök, mod 7'ye göre $\phi(7) = 6$ farklı kalan oluşturmalıdır. Bir g sayısı, mod 7'ye göre bir ilkel kök olabilmek için $g^k \bmod 7 (1 \leq k \leq \phi(7))$ değerlerinin hepsinin farklı ve $\{1, 2, 3, 4, 5, 6\}$'yi oluşturması gerekir. Burda işlemler tek tek yapılmamıştır fakat tek tek denenirse 7'nin ilkel köklerinin 3 ve 5 olduğu bulunur. Yani;

$g = 3$ için:

1. $3^1 \bmod 7 = 3$.
2. $3^2 \bmod 7 = 9 \bmod 7 = 2$.
3. $3^3 \bmod 7 = 27 \bmod 7 = 6$.
4. $3^4 \bmod 7 = 81 \bmod 7 = 4$.
5. $3^5 \bmod 7 = 243 \bmod 7 = 5$.
6. $3^6 \bmod 7 = 729 \bmod 7 = 1$.

$g = 3$ için $\{3, 2, 6, 4, 5, 1\}$ kümesi oluşturulur. Bu küme, $\{1, 2, 3, 4, 5, 6\}$'yı oluşturur. Dolayısıyla $g = 3$ bir ilkel köktür.

3.8.1 Python Kodu

```
def gcd(a, b):
    while b != 0:
        a, b = b, a % b

    return a
```

```
def phi_function(n):
    result = n
    p = 2
    while p * p <= n:
        if n % p == 0:
            while n % p == 0:
                n //= p

            result -= result // p

        p += 1

    if n > 1:
        result -= result // n

    return result

def is_primitive_root(g, n):
    phi_n = phi_function(n)
    residues = set()
    for k in range(1, phi_n + 1):
        residues.add(pow(g, k, n))
    return len(residues) == phi_n

def find_primitive_roots(n):
    phi_n = phi_function(n)
    primitive_roots = []
    for g in range(1, n):
        if gcd(g, n) == 1 and is_primitive_root(g, n):
            primitive_roots.append(g)
    return primitive_roots
```

3.9 Multiplicative Inverse (Çarpımsal Ters)

Çarpımsal ters, modüler aritmetikte bir sayının, belirli bir modül n'ye göre tersidir. Bir tam sayı a, mod n'ye göre bir çarpımsal tersi x'e sahiptir, eğer:

$$a \cdot x \equiv 1 (\bmod n)$$

şartını sağlayan bir x varsa. RSA, ElGamal gibi asimetrik şifreleme yöntemlerinde modüler çarpımsal tersler, anahtar oluşturma ve şifre çözme işlemlerinde kullanılır. Modüler tersler, özel anahtarların üretilmesinde kullanılır.

Örneğin, $a = 3$ ve $n = 7$ için mod 7'ye göre a'nın çarpımsal tersini bulun.

Bir sayının çarpımsal tersi yalnızca a ve n aralarında asal (relatively prime) ise vardır. Yani:

$$\text{GCD}(a, n) = 1$$

3 ve 7 aralarında asal olduğu için çarpımsal ters vardır.

$$3 \cdot x \equiv 1 (\bmod 7)$$

Deneme yanılma yöntemiyle hesaplayalım:

- $x = 1$ için, $3 \cdot 1 \bmod 7 \neq 1$.
- $x = 2$ için, $3 \cdot 2 \bmod 7 \neq 1$.
- $x = 3$ için, $3 \cdot 3 \bmod 7 \neq 1$.
- $x = 5$ için, $3 \cdot 5 \bmod 7 = 1$.

Sonuç $x = 5$'tir. Extended Euclidean Algorithm ile bu soru daha hızlı çözülür.

$$7 = 2 \cdot 3 + 1$$
$$1 = 7 - 2 \cdot 3$$
$$-2 \bmod 7 = 5$$

3.9.1 Python Kodu

```python
def extended_gcd(a, n):
    if n == 0:
        return a, 1, 0

    gcd_, x1, y1 = extended_gcd(n, a % n)
    x = y1
```

```
    y = x1 - (a // n) * y1
    return gcd_, x, y

def modular_inverse(a, n):
    gcd_, x, _ = extended_gcd(a, n)
    if gcd_ != 1:
        raise ValueError("")

    return x % n
```

3.10 Extended Euclidean Algorithm

Genişletilmiş öklid algoritması, iki sayı a ve b için ortak bölenlerin en büyüğünü bulur ve şu denklemi çözer:

$$ax + by = \text{GCD}(a, b)$$

Burada x ve y, $\text{GCD}(a, b)$ için bulunan Diophantine katsayılarıdır. Bu katsayılar sayesinde modüler aritmetikte çarpımsal ters bulma gibi işlemler yapılır. Asimetrik şifreleme algoritmalarında anahtar oluşturmada kullanılır.

Örneğin, $a = 30$ ve $b = 50$ için hesaplayalım. Öncelikle, öklid algoritması ile iki sayının en büyük ortak böleni bulunur:

$$\text{GCD}(a, b) = \text{GCD}(b, a \bmod b)$$

Hesaplamalar:

1. $a = 30$ ve $b = 50$ için $30 \bmod 50 = 30$.
2. $a = 50$ ve $b = 30$ için $50 \bmod 30 = 20$.
3. $a = 30$ ve $b = 20$ için $30 \bmod 20 = 10$.
4. $a = 20$ ve $b = 10$ için $20 \bmod 10 = 0$.

$\text{GCD}(30, 50) = 10$ için geriye doğru çalışarak x ve y'yi buluruz.

- $10 = 30 - 1 \cdot 20$
- $10 = 30 - 1 \cdot (50 - 1 \cdot 30)$
- $10 = 30 - 1 \cdot 50 + 1 \cdot 30$
- $10 = 2 \cdot 30 - 1 \cdot 50$

Sonuç, $x = 2$ ve $y = -1$ bulunur.

3.10.1 Python Kodu

```python
def extended_gcd(a, n):
    if n == 0:
        return a, 1, 0

    gcd_, x1, y1 = extended_gcd(n, a % n)
    x = y1
    y = x1 - (a // n) * y1
    return gcd_, x, y
```

3.11 Chinese Remainder Problem

Chinese Remainder Problem, modüler aritmetik alanıda bir problem çözme yeteneğidir. CRT, birbirinden ayrık (coprime) modüllerle verilen birden fazla denklem sistemi için bir çözüm bulmayı sağlar.

$$x \equiv a_1 \bmod m_1$$

$$x \equiv a_2 \bmod m_2$$

$$x \equiv a_n \bmod m_n$$

Burada:

- $m_1, m_2, ..., m_n$ modülleri çift çift coprime (birbirleriyle asal) olmalıdır.
- CRT, bu denklem sisteminin tekil bir çözümünü mod $M = m_1 \cdot m_2 \cdot ... \cdot m_n$ içinde bulur.

Büyük sayılarla yapılan modüler hesaplamaları daha küçük alt problemlere ayırır, bu da verimlilik sağlar. RSA şifreleme sisteminde hızlı modüler üs alma ve deşifre işlemlerinde kullanılır. Asimetrik şifreleme algoritmalarında modüller arasındaki ilişkileri yönetmek için kullanılır.
Örneğin, aşağıdaki denklem sistemini çözelim.

$$x \equiv 2 \bmod 3$$

$$x \equiv 3 \bmod 5$$

$$x \equiv 2 \bmod 7$$

$M = 3 \cdot 5 \cdot 7 = 105$'tir. Her modül için M_i hesaplanır.

$$M_i = \frac{M}{m_i}$$

- $M_1 = \frac{105}{3} = 35$
- $M_2 = \frac{105}{5} = 21$
- $M_3 = \frac{105}{7} = 15$

Her M_i için modüler ters N_i:

$$M_i \cdot N_i \equiv 1 \bmod m_i$$

N_1: $35 \cdot N_1 \equiv 1 \bmod 3$ için:

- $35 \bmod 3 = 2$
- $2 \cdot N_1 \equiv 1 \bmod 3$
- $N_1 = 2$ (modüler ters).

N_2: $21 \cdot N_2 \equiv 1 \bmod 5$ için:

- $21 \bmod 5 = 1$
- $1 \cdot N_2 \equiv 1 \bmod 5$
- $N_2 = 1$ (modüler ters).

N_3: $15 \cdot N_3 \equiv 1 \bmod 7$ için:

- $15 \bmod 7 = 1$
- $1 \cdot N_3 \equiv 1 \bmod 7$
- $N_3 = 1$ (modüler ters).

Genel çözüm hesaplanır:

$$x = \sum_{i=1}^{n} a_i \cdot M_i \cdot N_i \bmod M$$

$$x = (2 \cdot 35 \cdot 2) + (3 \cdot 21 \cdot 1) + (2 \cdot 15 \cdot 1) \bmod 105$$

$$x = (140) + (63) + (30) \bmod 105$$

$$x = 233 \bmod 105 = 23$$

Sonuç $x = 23$'tür.

3.11.1 Python Kodu

```python
def extended_gcd(a, n):
    if n == 0:
        return a, 1, 0

    gcd_, x1, y1 = extended_gcd(n, a % n)
    x = y1
    y = x1 - (a // n) * y1
    return gcd_, x, y

def modular_inverse(a, n):
    gcd_, x, _ = extended_gcd(a, n)
    if gcd_ != 1:
        raise ValueError("")

    return x % n

def chinese_remainder_problem(a, m):
    M = 1
    for mod in m:
        M *= mod
```

```
    x = 0
    for i in range(len(m)):
        Mi = M // m[i]
        Ni = modular_inverse(Mi, m[i])
        x += a[i] * Mi * Ni

    return x % M
```

3.12 Discrete Logarithm Problem

DLP, bir gruptaki modüler üs alma işleminin tersini bulma problemidir.

$$g^x \equiv y \bmod p$$

Burada:

- g: bir taban (generator veya primitive root).
- p: bir asal sayı (modülüs).
- x: bilinmeyen.
- y: verilen sonuç.

DLP, asimetrik kriptografide gizliliği korumak için kullanılır. DLP'nin çözümü büyük sayılar için çok zor olduğundan, birçok modern kriptografik algoritmada güvenlik temeli olarak kullanılır.

Örneğin, $2^x = 11 \bmod 19$ problemini çözelim.

- $x = 1$ için $2^1 \bmod 19 = 2, 2 \neq 11$
- $x = 2$ için $2^2 \bmod 19 = 4, 4 \neq 11$
- $x = 3$ için $2^3 \bmod 19 = 8, 8 \neq 11$
- $x = 4$ için $2^4 \bmod 19 = 16, 16 \neq 11$
- $x = 5$ için $2^5 \bmod 19 = 13, 13 \neq 11$
- $x = 6$ için $2^6 \bmod 19 = 7, 7 \neq 11$
- $x = 7$ için $2^7 \bmod 19 = 14, 14 \neq 11$
- $x = 8$ için $2^8 \bmod 19 = 9, 9 \neq 11$
- $x = 9$ için $2^9 \bmod 19 = 18, 18 \neq 11$
- $x = 10$ için $2^10 \bmod 19 = 11$

Sonuç $x = 10$'dur.

3.12.1 Python Kodu

```python
def discrete_logarithm_problem(base, result, modulus):
    for x in range(modulus):
        if pow(base, x, modulus) == result:
            return x

    return None
```

3.13 Fermat's Factoring Method

Fermat'ın çarpanlara ayırma yöntemi, bir sayıyı asal çarpanlarına ayırmak için kullanılır. n sayısının iki kare farkı olarak yazılmasını temel alır. Eğer n tek bir sayı ise, bu sayı iki kare farkı olarak yazılabilir:

$$n = a^2 - b^2 = (a - b)(a + b)$$

n sayısını asal çarpanlarına ayırmak için, ilk olarak n sayısına en yakın kare bulunur. Bu kare a^2 olarak alınır ve $b^2 = a^2 - n$ için b değeri bulunur. Yani, çarpanlar $a - b$ ve $a + b$ olur.

RSA şifreleme algoritmasında temel olarak büyük sayılar kullanılır. RSA'nın güvenliği, büyük bir sayıyı asal çarpanlarına ayırmanın zor olmasına dayanır. Fermat'ın yönteminin uygulanabilirliği, sayının asal çarpanlarını bulmanın zorluğunu gözler önüne serer.

Örneğin, $n = 5959$ sayısının asal çarpanlarını bulalım.

İlk adımda $n = 5959$ sayısına en yakın olan kareyi bulmamız gerekir.

$$\sqrt{5959} \approx 77.19$$

Yani, $a = 78$ seçiyoruz. Şimdi $a = 78$'i kullanarak b^2 bulunur:

$$b^2 = a^2 - n = 78^2 - 5959 = 6084 - 5959 = 125$$

$b^2 = 125$, bu bir tam kare değildir, bu yüzden a'yı bir artırarak bu işlemler tekrarlanır.

3.13.1 Python Kodu

```
import math

def fermat_factoring_method(n):
    a = math.isqrt(n) + 1
    b2 = a * a - n
    while not (int(math.isqrt(b2)) ** 2) == b2:
        a += 1
        b2 = a * a - n

    b = int(math.isqrt(b2))
    factor1 = a - b
    factor2 = a + b
    return factor1, factor2
```

3.14 Fermat's Primality Test

Fermat'ın asal sayı testi, bir sayının asal olup olmadığını kontrol etmek için kullanılan bir algoritmadır. Bu test, Fermat'ın Küçük Teoremi üzerine inşa edilmiştir. Fermat'ın Küçük Teoremi, p sayısının asal bir sayı olduğunu bildiğimizde, her a için şu eşitliğin doğru olduğunu söyler:

$$a^{p-1} \equiv 1(\bmod p)$$

Burada p asal bir sayı ve a ise p'den küçük bir tam sayıdır. Fermat Primality Testi, bu teoreme dayanarak bir sayının asal olup olmadığını test eder. Verilen n sayısının asal olup olmadığını test etmek için, rastgele bir a değeri seçilir. Eğer $a^{n-1} \equiv 1(\bmod n)$ ise, n sayısının asal olma ihtimali yüksektir. Ancak bu test her zaman doğru sonu. vermez, çünkü bazı bileşik sayılar, testten geçebilen Fermat sahte asal sayıları olabilir.

Fermat'ın Asal Testi, kriptografi alanında büyük öneme sahiptir, özellikle şifreleme algoritmalarında kullanılır. RSA gibi algoritmalar büyük asal sayılar kullanarak güvenlik sağlar. Fermat'ın testi, sayının asal olup olmadığını hızlı bir şekilde kontrol etmek için kullanılan ilk adımlardan birisidir.

Örneğin, $n = 11$ için Fermat'ın Asal Testi'ni uygulayalım. Test için n'den küçük $a = 2$ sayısını seçelim. Fermat'a göre:

$$a^{n-1} \equiv 1(\bmod n)$$

$$2^10 \bmod 11 = 1$$

Bu sonnuç, Fermat'ın asal testine göre $n = 11$'in asal olduğuna karar verir.

3.14.1 Python Kodu

```python
import random

def fermat_primality_test(n, k=5):
    for _ in range(k):
        a = random.randint(2, n - 1)
        if pow(a, n - 1, n) != 1:
            return False

    return True
```

3.15 Miller-Rabin Primality Test

Miller-Rabin testi, büyük sayıların asal olup olmadığını kontrol etmek için kullanılan olasılıksal bir algoritmadır. Fermat asal testinin geliştirilmiş bir versiyonudur. Fermat testinde, sahte asal sayılar yanlış sonuç verebilir fakat Miller-Rabin testinde bu tür hatalar önemli ölçüde azaltılmıştır. Miller-Rabin testine göre; eğer bir sayı asal ise, belirli bir eşitlik ve işlem kümesini her zaman sağlamalıdır. Eğer bir sayı bileşik ise, bu eşitlikler genellikle sağlanmaz, fakat nadiren de olsa sağlayabilir. Bu yüzden bu test probabilistik yani olasılıksal bir sonuç verir. Miller-Rabin testi, modüler üssün özelliklerini kullanarak bir sayının asal olup olmadığını test eder.

1. $n-1$ sayısı asal çarpanlarına ayrılır. $n-1 = 2^s \times d$ şeklinde yazılır, burada d tek bir sayı ve s bir pozitif tam sayıdır.
2. Eğer $a^d \equiv 1(\bmod n)$ veya $a^{2^r \times d} \equiv n-1 \ (\bmod n)$ sağlanıyorsa, n asal olabilir. Eğer bu eşitlik sağlanmazsa, n kesinlikle bileşiktir.
3. Test birkaç kez tekrar edilir, her seferinde farklı bir a seçilir.

RSA Algoritması, büyük asal sayılarla çalışır. Bu asal sayılar, şifreleme anahtarları oluşturmak için gereklidir. Miller-Rabin testi, bu asal sayıları hızlı bir şekilde test eder ve güvenli anahtarlar oluşturulmasına olanak tanır.

Örneğin, $n = 13$ sayısının asal olup olmadığını kontrol edelim. $n-1 = 12$ için 12'yi asal çarpanlarına ayıralım:

$$12 = 2^2 \times 3$$

Burada $s = 2$ ve $d = 3$ elde edilir. Rastgele bir a seçilir: $a = 2$. Miller-Rabin testi uygulayalım. İlk olarak a^d mod n'yi hesaplayalım:

$$2^3 \bmod 13 = 8, 8 \neq 1$$

Şimdi $a^{2^r \times d} \equiv n-1 \ (\bmod n)$ kontrol edelim (r = 1):

$$2^{2\times 3} \bmod \ = 13 = 2^6 \bmod \ = 13 = 64 \bmod 13 = 12$$

$12 = n-1$ eşit olduğu için, $n = 13$ asal olabilir. Eğer bu işlemler birçok a için geçerliyse, n'in asal olma olasılığı yüksektir.

3.15.1 Python Kodu

```
import random

def miller_rabin_test(n, k=5):
    if n <= 1 or n == 4:
        return False
```

```
    if n <= 3:
        return True

    s, d = 0, n - 1
    while d % 2 == 0:
        s += 1
        d //= 2

    for _ in range(k):
        a = random.randint(2, n - 2)
        x = pow(a, d, n)
        if x == 1 or x == n - 1:
            continue

        for _ in range(s - 1):
            x = pow(x, 2, n)
            if x == n - 1:
                break

        else:
          return False

    return True
```

3.16 Group

Grup, bir matematiksel yapıdır ve üzerinde belirli bir işlem tanımlanmamış olan bir kümedir. Bir grup, dört özelliğe sahip olmalıdır:

- **Kapalı Olma**: Eğer a ve b grup elemanları ise, $a * b$ işlemi de grup elemanı olmalıdır.
- **Birlik Elemanı (Identity Element)**: Bir grup elemanının üzerinde işlem uygulandığında, bu eleman kendisiyle aynı kalmalıdır. Bu, grup işlemi ile birleştirilmiş bir elemandır. Örneğin, sayıların toplama işlemi ile birleştirildiğinde sıfır bir birlik elemanıdır, $0 + 0 = 0$.
- **Ters Eleman (Inverse Element)**: Her grup elemanının tersine sahip olması gerekir. Yani, $a * a^{-1} = e$ olmalıdır. Burada a^{-1} elemanı a ile birleştirildiğinde birlik elemanı (e) verir.
- **İşlem Özelliği (Associativity)**: Grup elemanları arasındaki işlem, birleştirme özelliğine sahip olmalıdır. Yani, $(a * b) * c = a * (b * c)$ olmalıdır.

3.17 Abelian Group

Bir grup, üzerinde tanımlı işlem komütatif (sırasız) bir özellik gösteriyorsa o grup Abelian grup olarak adlandırılır. Yani, $a * b = b * a$ olmalıdır. Abel gruplarındaki işlemler sırasızdır. Tam sayılar kümesi, toplama işlemi altında bir abel grubu oluşturur çünkü toplama işlemi sırasızdır.

3.18 Cyclic Group

Bir grup, grubun elemanları bir tek elemanın (g) üssü olarak oluşturulabiliyorsa o grup döngüsel gruptur. Yani, $G = \{g^0, g^1, ..., g^{n-1}\}$ şeklinde yazılabilir. Bu durumda grup elemanları, grup içinde sürekli olarak bir "g" elemanın kuvvetleriyle ifade edilebilir.

3.19 Rings

Bir ring (halka), belirli bir küme ve bu küme üzerinde tanımlı iki işlem içeren bir matematiksel yapıdır. Halkaların tanımlanabilmesi için şu şartlar sağlanmalıdır:

- **Kapalı Olma (Closure)**: Eğer a ve b halkanın elemanları ise, $a + b$ ve $a \times b$ yine halkanın elemanlarıdır.
- **Toplama için Abel Grubu**: Toplama işlemi, Abel grubunun özelliklerini taşımalıdır. Yani toplama işlemi komütatif (sırasız) ve birleşmeli olmalıdır ve sıfır elemanı olmalıdır.
- **Çarpma için Dağılım Kuralı**: Çarpma işlemi, toplama işlemi üzerinde dağılım özelliğini sağlamalıdır. Yani, $a \times (b + c) = a \times b + a \times c$ olmalıdır.
- Çarpma işlemi için birim elemanı olması zorunlu değildir.

3.20 Fields

Bir field (cisim), bir ringin daha güçlü bir versiyonudur. Bir cisimde, çarpma işlemi için de ters elemanlar vardır. Yani, bir cisimde her elemanın bir çarpan tersi vardır. Cisim özellikleri:

- **Toplama ve Çıkarma İşlemi**: Her iki işlem de Abel grubunun özelliklerini taşır.
- **Çarpma için Ters Eleman**: Cisimde sıfır dışında her elemanın bir çarpan tersi vardır.
- Çarpma ve toplama için dağılım kuralı geçerlidir.

3.21 Finite Fields

Bir finite field (sonlu cisim), elemanlarının sayısı sonlu olan ve belirli özelliklere sahip bir cisimdir. Bu tür cisimler, kriptografide önemli bir rol oynar, çünkü sonlu sayılar üzerinde ilem yaparak güvenli anahtar değişimi ve şifreleme işlemleri gerçekleştirilir. Sonlu cisimlerin en bilinen örnekleri modüler aritmetik kullanılarak oluşturulan cisimlerdir. Sonlu alanların özellikleri:

- Eleman sayısı asal bir sayı olan bir cisim oluşturulabilir. Bu cisim Z_p olarak bilinir ve burada p bir asal sayıdır.
- Z_p üzerinde yapılan işlemler asal bir modül altında toplama ve çarpma işlemlerini içerir.

4 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.

4.1 Types of Hash Functions

4.1.1 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.

4.1.2 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.

4.1.3 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.

4.1.4 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.

4.1.5 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.

4.1.6 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.

4.1.7 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.

4.2 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.

4.2.1 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.

4.2.2 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}")
```

4.3 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.

4.3.1 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 $\text{checksum} = (\text{sum2} << 8) | \text{sum1}$.

4.3.2 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}")
```

4.4 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.

4.4.1 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: $\text{Adler32} = (B << 16)|A$.

4.4.2 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}")
```

4.5 XOR8

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

4.5.1 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.

4.5.2 Python Kodu

```
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}")
```

4.6 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.

4.6.1 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.

4.6.2 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}")
```

4.7 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.

4.7.1 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.

4.7.2 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}")
```

4.8 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.

4.8.1 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.

4.8.2 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}")
```

4.9 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.

4.9.1 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.

4.9.2 Python Kodu

```
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}")
```

4.10 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.

4.10.1 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.

4.10.2 Python Kodu

```
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}")
```

4.11 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.

4.11.1 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.

4.11.2 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}")
```

4.12 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.

4.12.1 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.

4.12.2 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}")
```

4.13 Bernstein's Hash (DJB2)

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

4.13.1 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.

4.13.2 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}")
```

4.14 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.

4.14.1 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.

4.14.2 Python Kodu

```
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}")
```

4.15 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.

4.15.1 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.

4.15.2 Python Kodu

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

4.16 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.

4.16.1 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.

4.16.2 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}")
```

4.17 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.

4.17.1 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.

4.17.2 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}")
```

4.18 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.

4.18.1 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}")
```

4.19 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.

4.19.1 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.

4.20 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.

4.20.1 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.

4.20.2 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}")
```

4.21 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.

4.21.1 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.

4.22 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.

4.22.1 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.

4.23 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.

4.23.1 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.

4.23.2 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}")
```

4.24 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.

4.24.1 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.

4.25 JH Hash

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

4.25.1 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.

4.26 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.

4.26.1 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.

4.27 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.

4.27.1 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.

4.27.2 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}")
```

4.28 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.

4.28.1 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.

4.28.2 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}")
```

4.29 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.

4.29.1 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.

4.29.2 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}")
```

4.30 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.

4.31 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.

4.31.1 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}")
```

4.32 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.

4.32.1 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.

4.32.2 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}")
```

4.33 Skein

2008 yılında geliştirilmiştir. Threefish blok şifreleme algoritmasını temel alır.

4.33.1 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.

4.34 Poly1305

Daniel J. Bernstein tarafından tasarlanmıştır. Poly1305, hızlı ve güvenli bir mesaj doğrulama kodu algoritmasıdır. Bir şireleme algoritmasıyla birlikte kullanılarak veri bütünlüğünü ve kimlik doğrulamasını sağlar. Anahtar uzunluğu 256 bit (32 bayt). Blok boyutu 16 bayttır. 128 bitlik nonce kullanır.

4.34.1 Çalışma Adımları

1. Anahtar iki parçaya ayrılır. 128-bit r, mesaj üzerinde matematiksel işlemler içinde kullanılır. r, bir sabit değerdir ve her zaman aynı kalır. 128-bit s, nihai MAC hesaplamasında kullanılan bir kayma değeridir.
2. Mesaj 16 baytlık bloklara bölünür. Son blok eksikse uygun şekilde doldurulur.
3. Her blok için modüler aritmetik kullanılarak bir toplama yapılır. $P = 2^{130} - 5$ modülü ile işlem yapılır.
4. Toplam hesaplama sonucunda, s ile birleştirilerek doğrulama kodu oluşturulur.
5. Alıcı, aynı anahtar ve nonce'u kullanarak doğrulama kodunu hesaplar ve gönderilen doğrulama kodu ile eşleşip eşleşmediğini kontrol eder.

4.34.2 Python Kodu

```
from Crypto.Hash import Poly1305
from Crypto.Random import get_random_bytes
from Crypto.Cipher import AES

def poly1305_generate_mac(message, key):
    mac = Poly1305.new(key=key, cipher=AES)
    mac.update(message.encode())
    return mac.hexdigest()

def poly1305_verify_mac(message, key, received_mac):
    mac = Poly1305.new(key=key, cipher=AES)
    mac.update(message.encode())
    try:
        mac.hexverify(received_mac)
        return True
    except:
        return False

key = get_random_bytes(32)
```

```
message = "Hello, World!"
mac = poly1305_generate_mac(message, key)
is_valid = poly1305_verify_mac(message, key, mac)
print("MAC:", mac)
print("Is Valid:", is_valid)
```

5 Tarihteki Şifreleme Yöntemleri

5.1 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.

5.1.1 Encryption

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

5.1.2 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.

5.1.3 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)
```

5.2 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.

5.2.1 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.

5.2.2 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.

5.2.3 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)
```

5.3 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.

5.3.1 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)
```

5.4 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.

5.4.1 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)
```

5.5 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.

5.5.1 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.

5.6 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.

5.6.1 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.

5.6.2 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)
```

5.7 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.

5.7.1 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.

5.8 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.

5.8.1 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)
```

5.9 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:

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

5.9.1 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.

5.9.2 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)
```

5.10 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.

5.10.1 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.

5.10.2 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)
```

5.11 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.

5.11.1 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.

5.11.2 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)
```

5.12 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.

5.12.1 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.

5.13 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.

5.13.1 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)
```

5.14 Linear Cipher

Linear Cipher, matematiksel bir doğrusal eşitlik kullanarak veriyi şifreler ve şifresini çözer. Şifreleme işlemi, bir matris çarpımı ve bir kaydırma gibi doğrusal işlemleri içerir. Şifreleme işlemi tersine çevrilebilir olmalıdır. Bu, matrisin tersinin alınabilmesi gerektiği anlamına gelir. Şifreleme için:

$$y = Ax + b$$

Burada, y şifreli metni, A şifrelemek için kullanılacak matris ve b kaydırma vektörünü temsil eder. Şifreyi çözmek için:

$$x = A^{-1}(y - b)$$

Şifreyi çözmek için şifreleme kullanılan matrisin tersi hesaplanır.

5.14.1 Encrpytion

Örneğin, "ALPER" kelimesini şifreleyelim. $A = 5$ ve $b = 9$ 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 "ALPER" kelimesinin her bir harfi için:

- A harfinin tablodaki değeri 0, $y = (5 \cdot 0 + 9) \bmod 26 = 9$, yani "J" harfine karşılıktır.
- L harfinin tablodaki değeri 11, $y = (5 \cdot 11 + 9) \bmod 26 = 12$, yani "M" harfine karşılıktır.
- P harfinin tablodaki değeri 15, $y = (5 \cdot 15 + 9) \bmod 26 = 6$, yani "G" harfine karşılıktır.
- E harfinin tablodaki değeri 4, $y = (5 \cdot 4 + 9) \bmod 26 = 3$, yani "D" harfine karşılıktır.
- R harfinin tablodaki değeri 17, $y = (5 \cdot 17 + 9) \bmod 26 = 16$, yani "Q" harfine karşılıktır.

"ALPER" kelimesinin şifrelenmiş hali "JMGDQ" dir. Şifreyi çözmek için ilk önce A'nın modüler tersi bulunmalıdır. 5'in 26'ya göre modüler tersi 21'dir, böylece $A^{-1} = 21$ olur. Eğer işlem negatif çıkarsa 26 ya göre modu eklenerek pozitif yapılır. Formülü uygularsak:

- J harfinin tablodaki değeri 9, $y = (21 \cdot (9 - 9)) \bmod 26 = 0$, yani "A" harfine karşılıktır.

- M harfinin tablodaki değeri 12, $y = (21 \cdot (12 - 9)) \bmod 26 = 11$, yani "L" harfine karşılıktır.
- G harfinin tablodaki değeri 6, $y = (21 \cdot (6 - 9)) \bmod 26 = 15$, yani "P" harfine karşılıktır.
- D harfinin tablodaki değeri 3, $y = (21 \cdot (3 - 9)) \bmod 26 = 4$, yani "E" harfine karşılıktır.
- Q harfinin tablodaki değeri 16, $y = (21 \cdot (16 - 9)) \bmod 26 = 17$, yani "R" harfine karşılıktır.

Şifreyi çözerek "JMGDQ" mesajından tekrar "ALPER" mesajını elde ettik.

5.14.2 Python Kodu

```
import numpy as np

def linear_encrypt(plaintext, A, b):
    numerical_text = [ord(char) - ord('A') for char in plaintext.upper()
                      if char.isalpha()]
    x = np.array(numerical_text)
    y = np.dot(A, x) + b
    return y

def linear_decrypt(ciphertext, A, b):
    A_inv = np.linalg.inv(A).astype(int)
    x = np.dot(A_inv, ciphertext - b).astype(int)
    plaintext = "".join(chr(num + ord("A")) for num in x)
    return plaintext

plaintext = "AK"
A = np.array([[3, 2], [2, 8]])
b = np.array([1, 4])
ciphertext = linear_encrypt(plaintext, A, b)
decrypted = linear_decrypt(ciphertext, A, b)
print("Encrypted:", ciphertext)
print("Decrypted:", decrypted)
```

5.15 Rail Fence Technique

Rail Fence, bir transpozisyon şifreleme tekniğidir. Veriyi belirli bir sayıda ray boyunca zikzak şeklinde yazmayı ve ardından bu raylarda yer alan karakterleri birleştirerek şifrelenmiş metni oluşturmayı içerir.

5.15.1 Encrpytion

1. Şifrelenecek metin bir dizi ray boyunca zikzak şeklinde yazılır.
2. Raylardaki karakterler sırayla birleştirilerek şifrelenmiş metin elde edilir.

Örneğin "ALPERKARACA" kelimesini şifreleyelim. Ray sayımız 2 olsun.

A		P		R		A		A		A
	L		E		K		R		C	

Şifreli metin, "APRAAALEKRC" elde edilmiştir.

5.15.2 Python Kodu

```
def rail_fence_encrypt(plaintext, num_rails):
    rails = ['' for _ in range(num_rails)]
    rail = 0
    direction = 1 # 1: Down, -1: Up

    for char in plaintext:
        rails[rail] += char
        rail += direction
        if rail == 0 or rail == num_rails - 1:
            direction *= -1

    return ''.join(rails)

def rail_fence_decrypt(ciphertext, num_rails):
    rail_lengths = [0] * num_rails
    rail = 0
    direction = 1

    for _ in ciphertext:
        rail_lengths[rail] += 1
        rail += direction
        if rail == 0 or rail == num_rails - 1:
            direction *= -1

    rails = []
```

```
    idx = 0
    for length in rail_lengths:
        rails.append(list(ciphertext[idx:idx + length]))
        idx += length

    result = []
    rail = 0
    direction = 1

    for _ in ciphertext:
        result.append(rails[rail].pop(0))
        rail += direction
        if rail == 0 or rail == num_rails - 1:
            direction *= -1

    return ''.join(result)

num_rails = 2
plaintext = "ALPERKARACA"
ciphertext = rail_fence_encrypt(plaintext, num_rails)
decrypted = rail_fence_decrypt(ciphertext, num_rails)
print("Encrypted:", ciphertext)
print("Decrypted:", decrypted)
```

5.16 Row-Column Transposition Cipher

Row-Column Transposition Cipher, metin şifrelemek için kullanılan bir transpozisyon şifreleme yöntemidir. Bu yöntemde metin, belirli bir satır ve sütun düzenine göre bir tabloya yerleştirilir. Şifreleme işlemi, tablodaki karakterleri sırasıyla sütunlar veya satırlar üzerinden okumayı içerir.

5.16.1 Encryption

1. Şifrelenecek metin uzunluğu, belirlenen tablo boyutlarına uygun olacak şekilde düzenlenir. Gerekirse metne dolgu karakterleri eklenir.
2. Metin, tabloya satır satır yerleştirilir.
3. Tablodaki metin sütun sütun okunarak şifrelenmiş metin oluşturulur.

Örneğin, "ALPERKARACA" mesajını şifreleyelim. 4 sütun ve 3 satır olsun.

A	L	P	E
R	K	A	R
A	C	A	X

Şifrelenmiş metin, "ARALKCPAAERX" elde edilmiştir.

5.16.2 Python Kodu

```
import math

def row_column_encrypt(plaintext, num_cols):
    num_rows = math.ceil(len(plaintext) / num_cols)
    padded_length = num_rows * num_cols
    plaintext = plaintext.ljust(padded_length, 'X')

    ciphertext = ""
    for col in range(num_cols):
```

```
        for row in range(num_rows):
            idx = row * num_cols + col
            ciphertext += plaintext[idx]

    return ciphertext

def row_column_decrypt(ciphertext, num_cols):
    num_rows = math.ceil(len(ciphertext) / num_cols)
    plaintext = [""] * len(ciphertext)
    idx = 0
    for col in range(num_cols):
        for row in range(num_rows):
            if idx < len(ciphertext):
                plaintext[row * num_cols + col] = ciphertext[idx]
                idx += 1

    return "".join(plaintext).rstrip('X')

num_cols = 4
plaintext = "ALPERKARACA"
ciphertext = row_column_encrypt(plaintext, num_cols)
decrypted = row_column_decrypt(ciphertext, num_cols)
print("Encrypted:", ciphertext)
print("Decrypted:", decrypted)
```

6 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.

Stream Cipher, veriyi bit bazında işler. Yani, her bir veri bitini (1 veya 0) tek tek şifreler. Veriyi sürekli (stream) olarak işler ve her bit için bir anahtar akışı oluşturur. Şifreleme ve şifre çözme işlemleri aynıdır. Şifreleme işlemi nasıl yapılırsa şifre çözme işlemi de aynı şekilde yapılır. Veri sürekli akış halinde olduğu için verilerin sırası önemli değildir. Anahtar akışı anahtarın bir parçası veya şifreleme algoritması tarafından üretilen bir sıralı sayı ile oluşuturulur. Aynı anahtar birden fazla bitin şifrelenmesinde kullanılabilir. Bit bitin hatalı olması sadece o bitin deşifre edilmesini etkiler. Hatalı bir bit şifreli veride yalnızca bir bitin yanlış çözülmesine yol açar. Hızlıdır. Veri uzunluğu belirsiz veya çok büyük olduğunda verimlidir. Ses ve video akışları, Wi-Fi, GSM şifrelemeleri gibi alanlarda kullanılır. Düşük donanım gereksinimlerine sahip cihazlarda tercih edilir.

Block Cipher, veriyi sabit uzunluktaki bloklar halinde işler. Veri birden fazla bitin bir bloğu olarak şifrelenir ve her blok, şifreleme işlemi sırsaında aynı anahtarı kullanır. Yavaştır. Şifreleme ve şifre çözme işlemleri ters sıralarda yapılır. Şifreleme ve şifre çözme algoritmaları birbirinden farklıdır. Sabit uzunluktaki veri bloklarını şifrelerken anahtar her blok için aynı kalır. Eğer bir blokta hata oluşursa, tüm blok etkilenir. Bu, hataların daha geniş bir alana yayılmasına yol açar. Dosya şifrelemeleri gibi alanlarda kullanılır. Daha güvenlidir.

6.1 Feistel Cipher

Feistel Yapısı, modern blok şifreleme algoritmalarının temelinde yatan bir yapıdır. Bu yapı, veriyi birden fazla turda işleyerek şifreleme veya

deşifreleme işlemi gerçekleştirir. Feistel Yapısı'nın en önemli özelliği, aynı algoritmanın hem şifreleme hem de deşifreleme için kullanılabilmesidir.

6.1.1 Encrpytion

$$L_{i+1} = R_i$$

$$R_{i+1} = L_i \oplus F(R_i, K_i)$$

Burada, F karmaşıklık ekleyen bir fonksiyondur ve K_i, tur anahtarıdır.

1. Açık metin eşit boyutta iki parçaya bölünür: Sol (L) ve Sağ (R).
2. Birden fazla tur uygulanır. Her turda, bir tur anahtarı kullanılır ve bu anahtar, her tur için farklı olabilir. Sol ve sağ parçalar birbiriyle değiştirilir ve bir fonksiyon (F) uygulanır. Bu fonksiyon, karmaşıklık katmak için tasarlanmıştır. Giriş verisi üzerinde matematiksel veya bit manipülasyonu işlemleri yapar.
3. Tüm turlar tamamlandıktan sonra R ve L birleştirilir.
4. Şifre çözmek için, şifreleme sırasında uygulanan işlemler ters sırada gerçekleştirilerek yapılır.

Örneğin, "00111010" anahtarını şifreleyelim. $K_0 = 0111$ ve $K_1 = 1001$ olsun. Fonksiyon olarak OR işlemini kullanalım. Tur sayımızı (round) 2 olsun.

1. Birinci turda;
   - Metin iki parçaya bölünür: "0011" ve "1010".
   - $L_0 = 0011$ ve $R_0 = 1010$ için $R_1 = L_0 \oplus F(R_0, K_0)$ işlemi yapılır. $F(R_0, K_0) = 1010 OR 0111 = 1111$ elde edilir. Şimdi bu değer ile L_0 XOR işlemine girer, $L_0 \oplus 1111 = 1010 \oplus 1111 = 1100$. Bu değer bizim yeni R_1 değerimizdir. Artık $L_1 = 1010$ ve $R_1 = 1100$ oldu.
2. İkinci turda;
   - $R_2 = L_1 \oplus F(R_1, K_1)$ için $F(R_1, K_1) = 1100 OR 1001 = 1101$ elde edilir. Bu değer ile L_1 XOR işlemine girer, $L_1 = \oplus 1101 = 1010 \oplus 1101 = 0111$. Bu değer bizim yeni R_2 değerimizdir. Artık $L_2 = 1100$ ve $R_2 = 0111$ oldu.
3. Swap işlemi yapılır, yani $L_2 = 0111$ ve $R_2 = 1100$ olur. Böylece şifrelenmiş veri "01111100" elde edilir.
4. Şifre çözme işlemi için de aynı işlemler yapılır. Fakat bu sefer anahtarlar tersten kullanılır yani ilk olarak K_1 anahtarı kullanılır.

6.1.2 Python Kodu

```
def feistel_encrypt(plaintext, keys):
    L = plaintext >> 32
    R = plaintext & 0xFFFFFFFF
    for key in keys:
        F_output = R ^ key
        new_L = R
        new_R = L ^ F_output
        L = new_L
        R = new_R

    ciphertext = (R << 32) | L
    return ciphertext

def feistel_decrypt(ciphertext, keys):
    L = ciphertext >> 32
    R = ciphertext & 0xFFFFFFFF
    for key in keys[::-1]:
        F_output = R ^ key
        new_L = R
        new_R = L ^ F_output
        L = new_L
        R = new_R

    decrypted = (R << 32) | L
    return decrypted

plaintext = 0x0000003A
keys = [0x00000007, 0x00000009]
ciphertext = feistel_encrypt(plaintext, keys)
decrypted = feistel_decrypt(ciphertext, keys)
print("Encrypted:", ciphertext)
print("Decrypted:", decrypted)
```

6.2 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.

6.2.1 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 ter-

sine göre düzenlenir.

4. Şifreleme işlemi tamamlanır.

6.2.2 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)
```

6.3 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.

6.3.1 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)
```

6.4 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.

6.4.1 Encryption

1. Düz metin, ilk olarak anahtar ile XOR işlemine girer.
2. Her tür şu 4 adımdan oluşur:
    - **SubBytes (Bayt Değiştirme)**: Her bayt, S-box adı verilen bir tablo yardımıyla bir başkasıyla değiştirilir. Doğrusal olmayan bir işlemdir. Durum matrisi üzerinde uygulanır. 128 bitlik bir anahtar için durum matrixi her elemanı 8 bit olan 4x4'lük bir matristir. Bu matrisin her bir elemanı 16'lık sayı tabanında yazılır. Durum matrisindeki her eleman, elemanın satır-sütun koordinatı ile S kutusu üzerindeki o koordinatta bulunan değerle değiştirilir.
    - **ShiftRows (Satır Kaydırma)**: Veri bloğundaki satırlar belirli bir düzene göre kaydırılır. Güncellenmiş durum matrisi üzerinde uygulanır. Her satır belirli bir oranda kaydırılır. Her n. satırda $n-1$'lik bir kaydırma yapılır, yani ilk satırda herhangi bir kaydırma olmazken, diğer satırlarda kendisinin bir eksiği kadar kaydırma yapılır.
    - **MixColumns (Sütun Karıştırma)**: Sütunlar matematiksel bir işlemle karıştırılır. Kaydırılan durum matrisi, sabit bir matris ile çarpılır.
    - **AddRoundKey (Tur Anahtarı Ekleme)**: Veri bloğu, o tura ait alt anahtar ile XOR yapılır. Toplama işlemi sütun bazlı yapılır.
3. Son turda, SubBytes, ShiftRows ve AddRoundKey işlemleri yapılır. MixColumns atlanır.
4. Tüm turlar tamamlandıktan sonra şifreli metin elde edilir.

6.4.2 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)
```

6.5 Salsa20

2005 yılında Daniel J. Bernstein tarafından tasarlananmış bir akış şifreleme algoritmasıdır. Yüksek hızda ve düşük donanım gereksinimleriyle çalışması için geliştirilmiştir. Salsa20, sabit zamanlı algoritma olarak tasarlanmıştır, bu da yan kanal saldırılarına karşı dayanıklılık sağlar. Anahtar uzunluğu 256 bittir. Temel olarak, XOR, ekleme, döndürme ve karıştırma işlemlerine dayanır. Bir anahtar ve 64 bitlik nonce (keyfi sayı) kullanarak bir keystream oluşturur ve bu keystream ile veriyi XOR işlemini sokarak şifreler. Nonce, aynı anahtarla şifreleme yapılırken keystream'lerin farklı olmasını sağlar. Blok sayacı, bloklar arasında tekrar eden keystream olmasını önler.

6.5.1 Encryption

1. 512 bitlik bir başlangıç durumu oluşturulur. Bu durum; sabit dizi, anahtar, nonce ve blok sayacından oluşur.
2. 20 tur boyunca durum üzerinde XOR, quarterround (çeyrek tur), rowround (satır turu) ve columnround (sütun turu) işlemleri yapılır.
3. Bu turlar sonucunda oluşan veri keystream olarak adlandırılır.
4. Düz metin bu keystream ile XOR işlemine sokularak şifrelenir. Aynı keystream ile şifrelenmiş metin tekrar XOR işlemine sokularak düz metin elde edilir.

6.5.2 Python Kodu

```
from Crypto.Cipher import Salsa20

def salsa20_encrypt(plaintext, key):
    cipher = Salsa20.new(key=key)
    encrypted = cipher.encrypt(plaintext.encode())
    ciphertext = cipher.nonce + encrypted
    return ciphertext

def salsa20_decrypt(ciphertext, key):
    nonce = ciphertext[:8]
    cipher = Salsa20.new(key=key, nonce=nonce)
    plaintext = cipher.decrypt(ciphertext[8:]).decode()
    return plaintext

key = b"secretsecretkey!"
plaintext = "Hello, World!"
ciphertext = salsa20_encrypt(plaintext, key)
decrypted = salsa20_decrypt(ciphertext, key)
print("Encrypted:", ciphertext)
print("Decrypted:", decrypted)
```

6.6 Blowfish

1993 yılında Bruce Schneier tarafından tasarlanmıştır. Blok boyutu 64 bittir, bu nedenle modern protokoller kadar güçlü değildir. 32 bit ile 448 bit arasında anahtar uzunluğuna sahip olabilir. Feistel ağına dayalıdır. Hem donanım hem de yazılımda hızlıdır. Patent veya lisans kısıtlaması yoktur.

6.6.1 Encryption

1. Kullanıcı tarafından verilen anahtar, P-dizisi ve dört S-box'ı başlatır. Toplamda yaklaşık 4168 baytlık bir yapı kullanılır.
2. Düz metin 64-bitlik iki parçaya bölünür: L (sol) ve R (sağ).
3. L ve R parçaları 16 tur boyunca işlem görür. Her turda, L ve R bir P-değeri ve S-box'lar kullanılarak değiştirilir. 16 tur sonunda L ve R yer değiştirir ve birleştirilir. XOR işlemleriyle şifreli metin elde edilir.

6.6.2 Python Kodu

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

def blowfish_encrypt(plaintext, key):
    cipher = Blowfish.new(key, Blowfish.MODE_CBC)
    iv = cipher.iv
    padded_text = pad(plaintext.encode(), Blowfish.block_size)
    ciphertext = cipher.encrypt(padded_text)
    return iv + ciphertext

def blowfish_decrypt(ciphertext, key):
    iv = ciphertext[:Blowfish.block_size]
    actual_ciphertext = ciphertext[Blowfish.block_size:]
    cipher = Blowfish.new(key, Blowfish.MODE_CBC, iv)
    padded_plaintext = cipher.decrypt(actual_ciphertext)
    plaintext = unpad(padded_plaintext, Blowfish.block_size)
    decrypted = plaintext.decode()
    return decrypted

key = b"secretsecretkey!"
plaintext = "Hello, World!"
ciphertext = blowfish_encrypt(plaintext, key)
decrypted = blowfish_decrypt(ciphertext, key)
print("Encrypted:", ciphertext)
print("Decrypted:", decrypted)
```

6.7 Twofish

Bruce Schneier tarafından tasarlanmıştır. Blok boyutu 128 bittir. Anahtar uzunluğu 128, 192, 256 bit olabilir. Feistel yapısına dayanır. Tescil ya da lisans kısıtlaması yoktur. Twofish, Feistel ağı ve MD5 matrisleri kullanır.

6.7.1 Encryption

1. Kullanıcının verdiği anahtar, anahtar programlama işlemi ile genişletilir. Subkey (alt anahtar) ve S-box tabloları oluşturulur.
2. Düz metin, 16 baytlık bloklara ayrılır. Her turda, düz metin bloğuna alt anahtarlarla XOR işlemi uygulanır. Veriler, S-boxlar ile karıştırılır. Pseudo Hamart Transform (PHT) ile veriler arasındaki karışım artırılır. Veriler, bit seviyesinde döndürülür.
3. Tüm şifreleme turları tamamlandıktan sonra son bir XOR işlemi yapılır ve şifreli metin elde edilir.

6.8 ChaCha20

Daniel J. Bernstein tarafından geliştirilmiştir. Hızlı, güvenli ve basit bir akış şifreleme algoritmasıdır. Salsa20 algoritmasının geliştirilmiş bir versiyonudur. Anahtar uzunluğu 256 bit (32 bayt). Nonce değeri, 96 bittir. Blok boyutu 512 bit (64 bayt).

6.8.1 Encryption

1. 256 bit uzunluğunda bir gizli anahtar (key) ve 96 bitlik bir nonce gerektirir. 32 bitlik bir sayaç ile çalışır.
2. 4 sabit değer, 256 bitlik anahtar, 96 bit nonce değeri ve 32 bit sayaç birleştirilerek 4x4 boyutunda durum matrisi oluşturulur.
3. Durum matrisi üzerinde bir dizi "Quarter Round" işlemi yapılır. Toplamda 20 tur round yapılır.
4. Durum matrisi kullanılarak bir anahtar akışı (key stream) oluşturulur. Şifreleme ve şifre çözme için bu akış kullanılır.
5. Anahtar akışı ile düz metin XOR işlemine girer böylece şifreli metin elde edilir.

6.8.2 Python Kodu

```
from Crypto.Cipher import ChaCha20
from Crypto.Random import get_random_bytes

def chacha20_encrypt(plaintext, key):
    nonce = get_random_bytes(12)
    cipher = ChaCha20.new(key=key, nonce=nonce)
    ciphertext = cipher.encrypt(plaintext.encode())
    return nonce, ciphertext

def chacha20_decrypt(ciphertext, key, nonce):
    cipher = ChaCha20.new(key=key, nonce=nonce)
    plaintext = cipher.decrypt(ciphertext).decode()
    return plaintext

key = get_random_bytes(32)
plaintext = "Hello, World!"
nonce, ciphertext = chacha20_encrypt(plaintext, key)
decrypted = chacha20_decrypt(ciphertext, key, nonce)
print("Encrypted:", ciphertext)
print("Decrypted:", decrypted)
```

6.9 RC2 (Rivest Cipher 2)

RC2, Ron Rivest tarafından 1987 yılında tasarlanmıştır. 64 bitlik blok boyutlarıyla çalışır. Anahtar boyutu 1-128 bit olabilir. ARC2 olarak da bilinir.

6.9.1 Encrpytion

1. Anahtar türetme algoritması, kullanıcının sağladığı anahtarı belirli bir uzunluğa genişletir.
2. Veri, 64 bitlik bloklara bölünür.
3. Her blok üzerinde RC2'nin belirlediği dönüşümler uygulanır.
4. Çıkış, şifrelenmiş veri bloklarıdır.

6.9.2 Python Kodu

```
from Crypto.Cipher import ARC2
from Crypto.Util.Padding import pad, unpad
from Crypto.Random import get_random_bytes

def rc2_encrypt(plaintext, key):
    cipher = ARC2.new(key, ARC2.MODE_CBC)
    ciphertext = cipher.encrypt(pad(plaintext.encode(), ARC2.block_size))
    return ciphertext, cipher.iv

def rc2_decrypt(ciphertext, key, iv):
    cipher = ARC2.new(key, ARC2.MODE_CBC, iv)
    plaintext = unpad(cipher.decrypt(ciphertext), ARC2.block_size)
    return plaintext.decode()

key = get_random_bytes(16)
plaintext = "Hello, World!"
ciphertext, iv = rc2_encrypt(plaintext, key)
decrypted = rc2_decrypt(ciphertext, key, iv)
print("Encrypted:", ciphertext)
print("Decrypted:", decrypted)
```

6.10 RC4 (Rivest Cipher 4)

RC4, Ron Rivest tarafından 1987 yılında tasarlanmıştır. Artık güvenli kabul edilmemektedir.

6.10.1 Encrpytion

1. Bir KSA (Key Scheduling Algorithm) ile, bir anahtar kullanılarak 256 baytlık bir S-box oluşturulur. S-box, anahtarın başlangıç durumunu temsil eder. S-box, 0'dan 255'e kadar tüm bayt değerlerini içerir.
2. Pseudo-random Generation Algorithm (PRGA) ile S-box kullanılarak bir rastgele bayt akışı üretilir. Bu rastgele bayt akışı, açık metinle XOR işlemine girerek şifreli metnini üretir.

6.10.2 Python Kodu

```
def ksa(key):
    S = list(range(256))
    j = 0
    for i in range(256):
        j = (j + S[i] + key[i % len(key)]) % 256
        S[i], S[j] = S[j], S[i]
    return S

def prga(S):
    i = 0
    j = 0
    while True:
        i = (i + 1) % 256
        j = (j + S[i]) % 256
        S[i], S[j] = S[j], S[i]
        yield S[(S[i] + S[j]) % 256]

def rc4_encrypt_decrypt(message, key):
    key = [ord(c) for c in key]
    S = ksa(key)
    keystream = prga(S)
    result = bytes([message[i] ^ next(keystream) for i in
        range(len(message))])
    return result

key = "secretkey"
plaintext = "Hello, World!"
ciphertext = rc4_encrypt_decrypt(plaintext.encode(), key)
decrypted = rc4_encrypt_decrypt(ciphertext, key)
print("Encrypted:", ciphertext)
print("Decrypted:", decrypted.decode())
```

7 Block Cipher Modes

Blok şifreleme modu, kriptografik blok şifreleme algoritmalarının nasıl kullanılacağını belirleyen bir yöntemdir. Tek başına bir blok şifreleme algoritması (AES, DES), sabit uzunlukta blokları şifreleyebilir. Ancak gerçekte veri genellikle sabit uzunlukta bloklara bölünemez. Blok şifreleme modları, bu algoritmaları verimli ve güvenli bir şekilde kullanmayı sağlar. Bu modlar, veri bütünlüğünü, gizliliğini ve gerektiğinde doğrulamayı garanti altına almak için tasarlanmıştır. Blok şifreleme şu algoritmalarla birlikte kullanılır:

- **AES**: CBC, CTR, GCM modlarıyla kullanılır.
- **DES**: Eski bir algoritma olup ECB ve CBC modlarında çalışır.
- **Blowfish ve Twofish**: CBC ve CTR modlarıyla çalışır.

7.1 Electronic Codebook Mode (ECB)

ECB, en basit şifreleme modlarından biridir. Bu modda, veri sabit boyutlu bloklara bölünür ve her blok bağımsız bir şekilde şifrelenir. Aynı giriş blokları, her zaman aynı çıkışı üretir. Her şifrelenmiş blok, aynı algoritma kullanılarak çözülür. Basit ve hızlıdır. Paralel işleme uygundur çünkü her blok bağımsızdır. Aynı giriş blokları, aynı şifreli blokları üretir (deterministic). Bu, desenlerin ortaya çıkmasına yol açar ve şifrelemeyi zayıf kılar. Güvenlik açısından kritik durumlarda önerilmez. CBC modu, sabit bir yapıya sahip dosyaların güvenliğini sağlamak için idealdir.

$$\text{Encryption } C_i = E_k[P_i]$$

$$\text{Decryption } P_i = D_k[C_i]$$

7.1.1 Python Kodu

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

key = b"secretsecretkey!"
plaintext = "Hello, World!"

cipher = AES.new(key, AES.MODE_ECB) # ECB Mode
padded_plaintext = pad(plaintext.encode("utf-8"), AES.block_size)
ciphertext = cipher.encrypt(padded_plaintext)
decrypted_data = cipher.decrypt(ciphertext)
decrypted = unpad(decrypted_data, AES.block_size).decode("utf-8")

print("Encrypted:", ciphertext)
print("Decrypted:", decrypted)
```

7.2 Cipher Block Chaining Mode (CBC)

CBC, her bir veri bloğunun şifrelenmesi sırasında önceki bloğun şifreli çıktısını kullanır. Bu yöntem, aynı düz metin bloklarının farklı şifreli metin blokları üretmesini sağlar. Desenlerin görünmesini engeller. İlk blok, bir başlangıç vektörü (Initialization Vector - IV) ile XOR yapılarak şifrelenir. Sonraki her blok, önceki şifrelenmiş blok ile XOR yapılarak şifrelenir. Aynı düz metin blokları farklı şifreli metinler üretir, desenleri gizler. Paralel şifrelemeye uygun değildir çünkü her blok bir öncekine bağlıdır. IV'nin güvenli bir şekilde iletilmesi gerekir.

Encryption için:

$$C_0 = \text{IV}$$

$$C_i = E_k[P_i \oplus C_{i-1}]$$

Decryption için:

$$C_0 = \text{IV}$$

$$P_i = C_{i-1} \oplus D_k[C_i]$$

7.2.1 Encryption

- İlk blok için bir başlangıç vektörü kullanılır. Bu, ilk bloğun şifreleme işlemini başlatmak için gereklidir. IV rastgele olmalıdır ve her şifreleme oturumu için farklı seçilmelidir.
- Her düz metin bloğu, şifreleme işleminden önce önceki bloğun şifreli metniyle XOR işlemine tabi tutulur. Elde edilen sonuç, blok şifreleme algoritması ile şifrelenir. İlk blok için IV kullanılır.
- Şifreli blok çözülür ve önceki şifreli blokla XOR işlemine tabi tutularak orijinal düz metin elde edilir.

7.2.2 Python Kodu

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

key = b"secretsecretkey!"
plaintext = "Hello, World!"

iv = get_random_bytes(AES.block_size)
cipher = AES.new(key, AES.MODE_CBC, iv) # CBC Mode
padded_plaintext = pad(plaintext.encode("utf-8"), AES.block_size)
ciphertext = cipher.encrypt(padded_plaintext)
cipher = AES.new(key, AES.MODE_CBC, iv)
decrypted_data = cipher.decrypt(ciphertext)
decrypted = unpad(decrypted_data, AES.block_size).decode("utf-8")
```

```
print("Encrypted:", ciphertext)
print("Decrypted:", decrypted)
```

7.3 Cipher Feedback Mode (CFB)

CFB, blok boyutunda veri işlemeye gerek duymadan bir akış şifresi gibi davranabilir. Bu, CFB'nin blok boyutundan bağımsız olarak bayt düzeyinde veri şifrelemesi yapmasını sağlar. IV veya önceki şifreli blok, şifreleme algoritması ile şifrelenir ve çıktısı düz metinle XOR yapılarak şifrelenmiş blok üretilir. Bir hata bir bloktan sonraki tüm blokları etkileyebilir (geribesleme nedeniyle). Paralel işleme uygun değildir. Blok boyutundan bağımsız olarak bayt düzeyinde şifreleme sağlar. Sadece şifreleme algoritması gerektiği için çözme işlemi de şifreleme algoritmasıyla yapılabilir.

Encryption için:

$$C_0 = \text{IV}$$

$$C_i = E_k[C_{i-1}] + P_i$$

Decryption için:

$$C_0 = \text{IV}$$

$$P_i = E_k[C_{i-1}] + C_i$$

7.3.1 Encryption

1. İlk işlem için bir başlangıç vektörü gereklidir. IV, rastgele bir değer olmalı ve şifreleme güvenliğini artırmak için kullanılmalıdır.
2. IV, blok şifreleme algoritması ile şifrelenir.
3. Şifreleme sonucunda elde edilen ilk blok, düz metinle XOR işlemine girer. XOR işleminin sonucu, şifreli metin bloğunu oluşturur.
4. Daha sonra, bu şifreli blok, sırada işlemler için geri besleme (feedback) olarak kullanılır.
5. Her bir şifreli metin bloğu, bir sonraki işlem için şifreleme algoritmasına girdi olarak verilir ve bu süreç devam eder.
6. Şifreli metin bloğu, önce blok şifreleme algoritmasının çıktısıyla XOR işlemine girer. Bu işlem düz metin bloğunu verir. Çözülen blok, sonraki işlemler için geri besleme olarak kullanılır.

7.3.2 Python Kodu

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

key = b"secretsecretkey!"
plaintext = "Hello, World!"
```

```
iv = get_random_bytes(AES.block_size)
cipher = AES.new(key, AES.MODE_CFB, iv) # CFB Mode
padded_plaintext = pad(plaintext.encode("utf-8"), AES.block_size)
ciphertext = cipher.encrypt(padded_plaintext)
cipher = AES.new(key, AES.MODE_CFB, iv)
decrypted_data = cipher.decrypt(ciphertext)
decrypted = unpad(decrypted_data, AES.block_size).decode("utf-8")

print("Encrypted:", ciphertext)
print("Decrypted:", decrypted)
```

7.4 Output Feedback Mode (OFB)

OFB, bir şifreleme algoritmasının çıktısını kullanarak düz metni şifreler veya şifreli metni çözer. Geri besleme (feedback) yöntemiyle çalışmasına rağmen, şifreleme işlemi sırasında düz metin veya şifreli metin şifreleme algoritmasına girdi olarak verilmez. IV, sürekli olarak şifreleme algoritmasından geçirilir. Her adımda elde edilen çıktı, düz metinle XOR yapılarak şifreli metin oluşturulur. Şifreleme ve çözme aynı IV ile başlamalıdır. Fakat aynı IV ve anahtar kullanılınca şifreleme kırılabilir. Bir blokta oluşan hata, sonraki blokları etkilemez. Bloklar bağımsız olduğundan hata yayılımı engellenir.

Encryption için:

$$X_0 = \text{IV}$$
$$X_i = E_k[X_{i-1}]$$
$$C_i = P_i + X_i$$

Decryption için:

$$X_0 = \text{IV}$$
$$X_i = E_k[X_{i-1}]$$
$$P_i = C_i + X_i$$

7.4.1 Encryption

1. OFB, rastgele bir IV kullanarak başlar. Bu vektör, ilk bloğun şifrelenmesi için gereklidir ve güvenli bir şekilde saklanmalıdır.
2. İlk olarak, IV, blok şifreleme algoritması ile şifrelenir ve bir çıktı üretir. Bu şifrelenmiş çıktı, düz metinle XOR işlemine girer. XOR işleminin sonucu, şifreli metin bloğunu oluşturur.
3. Her şifreleme adımında, şifreleme algoritamsının çıktısı bir sonraki bloğa girer. Şifreleme algoritmasının bu yeni çıktısı, sıradaki düz metin bloğunu XOR ile şifreler.
4. Şifreli metin, şifreleme algoritmasının çıktısı ile XOR işlemine tabi tutularak düz metin elde edilir. Şifreleme algoritmasının çıktısı, bir sonraki işlem için kullanılır.

7.4.2 Python Kodu

```
from Crypto.Cipher import AES
from Crypto.Random import get_random_bytes

key = b"secretsecretkey!"
plaintext = "Hello, World!"
```

```
iv = get_random_bytes(AES.block_size)
cipher = AES.new(key, AES.MODE_OFB, iv)
ciphertext = cipher.encrypt(plaintext.encode("utf-8"))
cipher = AES.new(key, AES.MODE_OFB, iv)
decrypted_data = cipher.decrypt(ciphertext)
decrypted = decrypted_data.decode("utf-8")

print("Encrypted:", ciphertext)
print("Decrypted:", decrypted)
```

7.5 Counter Mode (CTR)

CTR modunda, şifreleme ve çözme işlemleri için şifreleme algoritmasının bir sayacı (counter) girdisi olarak kullanılır. Her blok için bir sayaç değeri hesaplanır, bu değer şifreleme algoritmasıyla şifrelenir ve düz metin veya şifreli metin ile XOR işlemine tabi tutulur. Sayaç her blok için artırılır. Paralel işleme uygundur, hızlıdır. Sayaç değeri yeniden kullanılırsa güvenlik riski oluşturur.

$$X_i = E_k[\text{Counter} + i]$$
$$C_i = P_i \oplus X_i$$

7.5.1 Encryption

1. Rastgele bir başlangıç değeri (nonce) kullanılır. Bu değer her şifreleme işleminde farklı olmalıdır.
2. Her blok için bir sayaç değeri oluşturulur. Sayaç genellikle başlangıç değeri ve blok sırasına göre artar.
3. Nonce ve sayaç birleştirilerek bir giriş bloğu oluşturulur. Bu giriş bloğu şifreleme algoritmasıyla şifrelenir. Şifrelenmiş sayaç değeri, düz metin bloğu ile XOR işlemine girer. Bu işlem sonucunda şifreli metin bloğu oluşturulur.
4. Sayaç değeri bir artırılır ve süreç diğer bloklar için tekrar eder.
5. Şifreleme ile aynı sayaç ve nonce değerleri kullanılarak şifreleme algoritmasının çıktıları üretilir. Şifreli metin bloğu, şifreleme algoritmasından gelen değerle XOR işlemine tabi tutulur. Bu işlem sonucunda düz metin elde edilir.

7.5.2 Python Kodu

```
from Crypto.Cipher import AES
from Crypto.Util import Counter

key = b"secretsecretkey!"
plaintext = "Hello, World!"

nonce = b"12345678"
ctr = Counter.new(64, prefix=nonce, initial_value=0)
cipher = AES.new(key, AES.MODE_CTR, counter=ctr)
ciphertext = cipher.encrypt(plaintext.encode("utf-8"))
ctr = Counter.new(64, prefix=nonce, initial_value=0)
cipher = AES.new(key, AES.MODE_CTR, counter=ctr)
decrypted_data = cipher.decrypt(ciphertext)
decrypted = decrypted_data.decode("utf-8")
```

```
print("Encrypted:", ciphertext)
print("Decrypted:", decrypted)
```

8 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.

8.1 RSA (Rivest-Shamir-Adleman)

1977 yılında Ron Rivest, Adi Shamir ve Leonard Adleman tarafından geliştirilmiştir. İsmini kendisini geliştiren ekibin baş harflerinden alır. RSA, büyük asal sayıların çarpanlarına ayrılmasının zorluğuna dayanan bir algoritmadır. RSA, hem şifreleme hem de dijital imzalama için kullanılır. RSA'ya matematik ve yan kanal kriptanaliz saldırları yapılmıştır. Yan kanal saldırıları başarılı bir sonuç verirken, matematiksel saldırılar başarısız olmuştur. Algoritmanın gücü, asal sayıların büyüklüğüyle orantılıdır. Yeni saldırılar daha büyük bir asal sayı kullanır fakat büyük asal sayılar daha fazla matematiksel işlem doğurur.

8.1.1 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.

8.1.2 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)
```

8.2 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.

8.2.1 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: $M = C_2 - d \cdot C_1$.

8.3 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. Geliştirilirken, ElGamal algoritmasından yararlanılmıştır.

8.3.1 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.

8.3.2 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)
```

8.4 ElGamal

1984 yılında Taher ElGamal tarafından geliştirimiştir. ElGamal, açık anahtarlı şifreleme için kullanılan bir algoritmadır. Hem şifreleme hem de dijital imzalama işlemleri için kullanılabilir. RSA yöntemine benzerdir. ElGamal, Diffie-Hellman anahtar değişimi üzerine kuruludur ve büyük asal sayıların modüler aritmetiği üzerinde çalışır.

8.4.1 Encryption

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.

8.4.2 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)
```

8.5 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. Homomorfik şifreleme, şifreli mesajı çözmeden şifreli mesaj üzerinden işlemler yapılabilmesini sağlar. Şifreli mesaj üzerindeki işlemlerin sonucu, şifresiz mesaj üzerindeki işlemlerin sonuçlarıyla aynıdır.

8.5.1 Encryption

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. $m = L(u) \cdot \mu \bmod n$.

8.6 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.

8.6.1 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.

8.6.2 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.")
```

8.7 Merkle-Hellman Cipher

Merkle-Hellman şifreleme, 1978'de Ralph Merkle ve Martin Hellman tarafından önerilmiş olup, ilk açık anahtarlı şifreleme yöntemlerinden biridir. Bu yöntem, süper artan altküme toplamları kullanılarak verileri şifreler. Şifreleme, NP-Complete bir problem olan sırt çantası (knapsack) probleminin bir varyantına dayanır. NP-Complete, belirsiz Turing makineleri ile çok terimli (polinomsal) zamanda çözülebilen problemleri içerir.

8.7.1 Encryption

İlk olarak anahtar çifti oluşturulur.

1. Süper artan alt küme dizisi, özel bir dizi (W) seçilir. Bu dizide her sayı, kendinden önceki sayıların toplamından büyüktür.
2. Modül (m), diziden büyük bir asal sayı seçilir.
3. Çarpan (n), m ile aralarında asal bir sayı seçilir.
4. Açık anahtar (B): $B[i] = (W[i] \cdot n) \bmod m$ ile hesaplanır.

Daha sonra şifreleme işlemine geçilir.

1. Mesaj, binary hale getirilir.
2. Her bir bit, açık anahtar (B) ile çarpılır ve sonuçların toplamı (C) şifreli mesaj olarak elde edilir.

Şifre çözmek için:

1. Şifreli mesaj (C) ile $n^{-1} \bmod m$ ile hesaplanır.
2. Bu değer, özel anahtar dizisi (W) kullanılarak orijinal mesajın bitleri çözülür.

8.7.2 Python Kodu

```
W = [2, 3, 6, 14, 30]
m = 61
n = 17
B = [(w * n) % m for w in W]

def merkle_hellman_encrypt(plaintext, B):
    message_bits = [int(bit) for bit in message]
    C = sum([message_bits[i] * B[i] for i in range(len(message_bits))])
    return C

def merkle_hellman_decrypt(C, W, m, n):
```

```
    n_inv = pow(n, -1, m)
    C_prime = (C * n_inv) % m
    decrypted_bits = []
    for w in reversed(W):
        if C_prime >= w:
            decrypted_bits.insert(0, 1)
            C_prime -= w
        else:
            decrypted_bits.insert(0, 0)

    return "".join(map(str, decrypted_bits))

plaintext = "10101"
encrypted = merkle_hellman_encrypt(plaintext, B)
decrypted = merkle_hellman_decrypt(encrypted, W, m, n)
print("Encrypted:", encrypted)
print("Decrypted:", decrypted)
```

8.8 Okamoto-Uchiyama Cipher

Okamoto-Uchiyama şifreleme sistemi, 1998 yılında Tatsuaki Okamoto ve Shigenori Uchiyama tarafından önerilen bir açık kaynak anahtarlı şifreleme yöntemidir. Bu sistem, tam homomorfik şifreleme özelliklerine sahiptir ve sayı teorisine dayalıdır. Güvenliği, modüler üs alma ve modüler logaritma problemleri üzerine kuruludur.

8.8.1 Encryption

Önce anahtarlar oluşturulur.

1. Büyük asal sayılar p ve q seçilir. $n = p^2 \cdot q$ hesaplanır.
2. g, $Z^*_{n^2}$ kümesinden rastgele seçilir ve şu koşul sağlanır: $g^{p-1} \bmod p^2 \neq 1$.
3. Açık anahtar: (n, q).
4. Gizli anahtar: (p, q).

Mesaj m şifrelenir.

1. r, Z^*_n kümesinden rastgele seçilir.
2. Şifreli metin: $c = g^m \cdot r^n \bmod n^2$.

Şifre çözmek için:

1. Şifre çözmek için c, gizli anahtar p kullanılarak çözülür: $u = c^{p-1} \bmod p^2$.
2. $L(u) = \frac{u-1}{p}$.
3. $m = \frac{L(u)}{L(g^{p-1} \bmod p^2)} \bmod p$.

8.8.2 Python Kodu

```
import random
from math import gcd

def generate_keys():
    p = 23
    q = 29
    n = p**2 * q

    while True:
        g = random.randint(2, n**2 - 1)
        if pow(g, p - 1, p**2) != 1:
            break
```

```
    public_key = (n, g)
    private_key = (p, q)
    return public_key, private_key

def okamoto_uchiyama_encrypt(plaintext, public_key):
    n, g = public_key
    r = random.randint(2, n - 1)
    while gcd(r, n) != 1:
        r = random.randint(2, n - 1)

    c = (pow(g, plaintext, n ** 2) * pow(r, n, n ** 2)) % (n ** 2)
    return c

def okamoto_uchiyama_decrypt(ciphertext, public_key, private_key):
    n, g = public_key
    p, _ = private_key
    u = pow(ciphertext, p - 1, p ** 2)
    l_u = (u - 1) // p
    g_p = pow(g, p - 1, p ** 2)
    l_g_p = (g_p - 1) // p
    m = (l_u * pow(l_g_p, -1, p)) % p
    return m

public_key, private_key = generate_keys()
plaintext = 42
ciphertext = okamoto_uchiyama_encrypt(plaintext, public_key)
decrypted = okamoto_uchiyama_decrypt(ciphertext, public_key, private_key)
print("Encrypted:", ciphertext)
print("Decrypted:", decrypted)
```

8.9 Goldwasser-Micali Cipher

1982 yılında Shafi Goldwasser ve Silvio Michali tarafından geliştirimiştir. Olasılıklı açık anahtarlı şifreleme algoritmasıdır. Bu sistem, semantik güvenlik kavramının öncüsü olup, kriptolojik sağlamlık açısından önemli bir yere sahiptir. Goldwasser-Micali, yalnızca bit düzeyinde şifreleme yapar ve güvenliği, kuadratik kalıntı problemine dayanır.

8.9.1 Encryption

Anahtar çiftleri oluşturulur.

1. İki büyük asal sayı p ve q seçilir. $n = p \cdot q$ hesaplanır.
2. n'nin kuadratik kalıntısı olmayan bir y değeri rastgele seçilir: $y \in Z_n^*$ ve $(\frac{y}{n}) = -1$. Burada $(\frac{y}{n})$, Jacobi sembolünü ifade eder.
3. Açık anahtar: (n, y).
4. Gizli anahtar: (p, q).

Mesaj m şifrelenir.

1. r, Z_n^* kümesinden rastgele seçilir.
2. Şifreli metin: $c = r^2 \cdot y^m \bmod n$. Eğer $m = 0$ ise, $c = r^2 \bmod n$. Eğer $m = 1$ ise, $c = r^2 \cdot y \bmod n$.

Şifre çözmek için gizli anahtar kullanılarak c'nin kuadratik kalıntı olup olmadığı kontrol edilir. Eğer c, n modülünde kuadratik kalıntıysa $m = 0$, değilse $m = 1$.

8.9.2 Python Kodu

```
import random
from sympy import isprime, jacobi_symbol

def generate_keys(bit_length=16):
    while True:
        p = random.getrandbits(bit_length)
        if isprime(p):
            break
    while True:
        q = random.getrandbits(bit_length)
        if isprime(q):
            break

    n = p * q
    while True:
        y = random.randint(2, n - 1)
```

```
        if jacobi_symbol(y, n) == -1:
            break

    public_key = (n, y)
    private_key = (p, q)
    return public_key, private_key

def goldwasser_micali_encrypt(bit, public_key):
    n, y = public_key
    r = random.randint(1, n - 1)
    c = (pow(r, 2, n) * (y if bit == 1 else 1)) % n
    return c

def goldwasser_micali_decrypt(ciphertext, public_key, private_key):
    n, _ = public_key
    p, q = private_key
    if pow(ciphertext, (p - 1) // 2, p) == 1 and pow(ciphertext, (q - 1)
        // 2, q) == 1:
        return 0
    else:
        return 1

public_key, private_key = generate_keys()
message_bit = 1
ciphertext = goldwasser_micali_encrypt(message_bit, public_key)
decrypted = goldwasser_micali_decrypt(ciphertext, public_key,
    private_key)
print("Encrypted:", ciphertext)
print("Decrypted:", decrypted)
```

8.10 Blum-Goldwasser Cipher

Blum-Goldwasser şifreleme, olasılıklı bir açık anahtarlı şifreleme algoritmasıdır. Bu yöntem semantik güvenliğe sahip olup, verilerin şifrelenmesi sırasında bir rastgelelik unsuru kullanır. Blum Blum Shub (BBS) adı verilen bir rastgele sayı üreticisi üzerine kuruludur ve doğrusal olmayan bir kelebek yapısıyla çalışır. Güvenliği, kuadratik kalıntı probleminin çözümünün zorluğuna dayanır. Blum Blum Shub algoritması:

$$x_0 = \text{seed}$$
$$x_t = ((x * 2) \bmod n)$$
$$\text{random bit} = \bmod 2$$

Burada, x_0 başlangıçta seed değeri, x_t belirtilen uzunluk boyunca güncellenen x değeri, n modülüs değeri, random bit yapılan işlemler sonucunda oluşturulan rastgele bittir.

8.10.1 Encryption

Anahtar üretimi yapılır.

1. Büyük asal sayılar p ve q seçilir.
2. Modül n hesaplanır: $n = p \cdot q$
3. n açık anahtar olarak kullanılır.
4. p ve q gizli anahtar olarak kullanılır.

Mesaj şifrelenir.

1. Şifrelenecek mesaj ikili formata çevirilir.
2. Başlangıç değeri x_0, n'nin bir elemanı olacak şekilde rastgele seçilir.
3. Blum Blum Shub algoritması kullanılarak x_0'dan itibaren rastgele bitler üretilir.
4. Mesaj, üretilen bitlerle XOR işlemine girer ve şifrelenmiş mesaj elde edilir. $C = M \oplus$ random bits.
5. Son üretilen durum x_t ve C, alıcıya iletilir.

Şifre çözme için

1. Alıcı, gizli anahtarlar p ve q'yu kullanarak başlangıç değeri x_0'ı yeniden hesaplar.
2. Blum Blum Shub algoritması kullanılarak şifre çözmek için aynı rastgele bitler tekrar üretilir.
3. Şifre çözülür: $M = C \oplus$ random bits.

8.10.2 Python Kodu

```
import random
from sympy import isprime

def generate_keys(bit_length=16):
    while True:
        p = random.getrandbits(bit_length)
        if isprime(p) and p % 4 == 3:
            break
    while True:
        q = random.getrandbits(bit_length)
        if isprime(q) and q % 4 == 3:
            break

    n = p * q
    return (n, p, q)

def blum_blum_shub(seed, n, length):
    random_bits = []
    x = seed
    for _ in range(length):
        x = pow(x, 2, n)
        random_bits.append(x % 2)
    return random_bits, x

def blum_goldwasser_encrypt(message, public_key):
    n = public_key
    seed = random.randint(1, n - 1)
    message_bits = ''.join(format(ord(char), '08b') for char in message)
    random_bits, final_state = blum_blum_shub(seed, n, len(message_bits))
    cipher_bits = ''.join(str(int(b1) ^ int(b2)) for b1, b2 in
        zip(message_bits, random_bits))
    return cipher_bits, seed, final_state

def blum_goldwasser_decrypt(cipher_bits, private_key, seed):
    n, p, q = private_key
    message_length = len(cipher_bits)
    random_bits, _ = blum_blum_shub(seed, n, message_length)
    message_bits = ''.join(str(int(b1) ^ int(b2)) for b1, b2 in
        zip(cipher_bits, random_bits))
    chars = [chr(int(message_bits[i:i+8], 2)) for i in range(0,
        len(message_bits), 8)]
    decrypted = ''.join(chars)
    return decrypted

public_key, private_key = generate_keys(16)[0], generate_keys(16)[1:]
message = "karaca"
seed = 42
```

```
cipher_bits, seed, final_state = blum_goldwasser_encrypt(message,
    public_key)
decrypted = blum_goldwasser_decrypt(cipher_bits, (public_key,
    *private_key), seed)
print("Encrypted (Bits):", cipher_bits)
print("Decrypted:", decrypted)
```

9 Kriptoanaliz Yöntemleri

9.1 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.

9.1.1 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.

9.1.2 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)
```

9.2 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.

9.2.1 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.

9.2.2 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)
```

9.3 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.

9.3.1 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)
```

9.4 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.

9.4.1 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)
```

9.5 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.

9.5.1 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)
```

9.6 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.

9.7 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.

9.8 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.

9.8.1 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.

9.8.2 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.")
```

9.9 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.

9.10 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.

9.10.1 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)
```

9.11 Differential Cryptanalysis

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.

9.12 Integral Cryptanalysis

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.

9.13 Square Attack

Square Attack, simetrik blok şifrelemelerindeki zayıflıkları ortaya çıkarmak için kullanılır. Düşük tur sayısına sahip şifreleme sistemlerini hedef alır. İlk olarak, Rijndael şifreleme algoritmasını analiz etmek için geliştirilmiştir.

9.13.1 Çalışma Adımları

1. Bir blok şifreleme algoritması için aynı anahtar kullanılarak 256 farklı giriş değeri şifrelenir. Bu, hedeflenen şifrelme algoritmasının diferansiyel yapısını ortaya çıkarmaya yardımcı olur.
2. Giriş verileri algoritma üzerinden şifrelenir ve çıktı blokları kaydedilir.
3. XOR işlemi uygulayarak belirli desenler veya simetriler araştırılır. Bu analiz, algoritmanın belirli tur işlemleri sonnucunda farklılıkları nasıl işlediğini anlamayı sağlar.
4. Herhangi bir tur işleminden sonra, algoritmanın bazı belirli yapıları koruyup korumadığı incelenir. Örneğin XOR sonucunun 0 olduğu bayt pozisyonları aranır.
5. Algoritmanın bazı turlarını analiz ederek, kullanılan anahtarın parçaları hakkında bilgi elde edilir.

9.14 Davies Attack

Davies Attack, simetrik şifreleme algoritmalarında kullanılan bir kriptoanaliz yöntemidir. DES algoritmasına yönelik geliştirilmiştir. Algoritmanın zayıf alt-anahtarlar (subkeys) ürettiği durumları analiz eder. Bu saldırı, şifreleme işlemi sırasında anahtarların oluşturulma şekline ve şifreleme turlarındaki belirli desenlere dayanarak, şifreleme anahtarının bir kısmını veya tamamını tahmin etmeye çalışır.

9.15 Linear Cryptanalysis

Linear Cryptanalysis, blok şifreleme algoritmalarını çözmek için kullanılan bir kriptanaliz tekniğidir. Bu yöntem, şifreleme algoritmasının lineer bir modelle yaklaşık olarak temsil edilebileceğini varsayar. Amaç, algoritmanın belirli tur fonksiyonları arasında doğrusal bir ilişki bulmaktır ve bu ilişkilerden anahtar bilgisi çıkarmaktır. DES ve diğer simetrik şifreleme algoritmalarına uygulanmıştır.

9.16 Impossible Differential Cryptoanalysis

Impossible Differential Cryptoanalysis, modern blok şifreleme algoritmalarına yönelik bir kriptoanaliz tekniğidir. Bu yöntemde, belirli bir şifreleme algoritmasında mümkün olmayan diferansiyeller tespit edilir ve bu bilgi kullanılarak anahtar uzayı daraltılır. Yani, belirli bir giriş ve çıkış farkı kombinasyonunun hiçbir şekilde oluşamayacağı kanıtlanır ve bu bilgi, şifreleme sisteminin zayıflıklarını keşfetmek için kullanılır.

10 Zero-Knowledge Proof

Zero-Knowledge Proofs (Sıfır Bilgi İspatları), bir tarafın bir başka tarafa, belirli bir bilgiyi ifşa etmeden, o bilgiye sahip olduğunu kanıtlamasını sağlayan bir kriptografik yöntemdir. Bu süreçte kanıt sunucu, doğrulayıcıyı, gerçeği bildiğine ikna eder ancak bu gerçek hakkında hiçbir ek bilgi paylaşmaz. Bilgi sızdırmadan doğrualama yapılmasını sağlar. Şifre veya biyometrik veri paylaşmadan kimlik doğrulama yapılabilir. Kripto para işlemlerinde gizlilik odaklı protokoller için kullanılır.

- **Tamlık (Completeness)**: Eğer kanıt sunucu doğru bilgiye sahipse, doğrulayıcı bunu kabul eder.
- **Sağlamlık (Soundness)**: Kanıt sunucu yanlış bilgiye sahipse, doğrulayıcı kandırılamaz.
- **Sıfır Bilgi (Zero-Knowledge)**: Doğrulayıcı, bilgiye dair hiçbir şey öğrenemez.

Zero-Knowledge Proof türleri:

1. **Interactive Zero-Knowledge Proof**: Kanıt sunucu ile doğrulayıcı arasında interaktif (karşılıklı) bir iletişim vardır. Kanıt sunucu, doğrulayıcı tarafından sorulan sorulara yanıt verir.
2. **Non-Interactive Zero-Knowledge Proof**: Kanıt süreci interaktif değildir. Bir defa oluşturulan kanıt herkes tarafından doğrulanabilir. Kriptografik protokollerde daha yaygındır.
3. **Perfect Zero-Knowledge Proof**: Doğrulayıcı, kanıtın geçerliliğini yüzde yüz kesinlikle anlar.
4. **Statistical Zero-Knowledge Proof**: Kanıt, doğrulayıcıyı neredeyse kesin olarak ikna eder.
5. **Computational Zero-Knowledge Proof**: Kanıt, doğrulayıcıyı yalnızca belirli bir hesaplama gücüyle ikna eder.

10.0.1 "Ali Baba Mağarası" Problemi

Bir mağara, iki yola ayrılıyor (A ve B). Yolun sonunda bir kapı var ve bu kapıyı açmak için bir şifre gerekiyor. Kanıt sunucu, bu kapıyı açabildiğini doğrulayıcıya ispatlamak istiyor, ancak şifreyi açıklamak istemiyor. Doğrulayıcı, kanıt sunucunun gerçekten şifreyi bildiğinden emin olmak ister. Kanıt sunucu, mağaraya (A veya B yoluna) girer ve kapıya ulaşır. Doğrulayıcı, rastgele bir yol seçer ve kanıt sunucusundan o yoldan çıkmasını ister. Eğer kanıt sunucu şifreyi biliyorsa, kapıyı açarak doğru yolu takip eder ve çıkışı sağlar. Eğer kanıt sunucu şifreyi bilmiyorsa, her seferinde doğru yolu tahmin etmesi gerekecektir.

Buradaki ZKP; kanıt sunucu, doğrulayıcıya şifreyi bildiğini ispat eder, ancak şifre hakkında hiçbir bilgi vermez. Bu işlem birkaç kez tekrarlandığında, kanıt sunucunun gerçekten şifreyi bildiği doğrulayıcı tarafından güvenle kabul edilebilir. Zero-Knowledge özelliği, doğrulayıcının sadece doğru bilgiye sahip olunduğunu öğrenmesi, ancak şifreyi asla öğrenmemesidir.

10.1 "Renk Körü Arkadaş ve İki Top" Problemi

Bu problemde, bir kişi renk körüdür ve iki topu (biri kırmızı, diğeri mavi) ayırt edememektedir. Diğer kişi ise renk körü değildir ve iki topu birbirinden ayırabilmektedir. Kanıt sunucu, doğrulayıcıya bu iki topun rengini bildiğini ispatlamak ister. Kanıt sunucu, topu gizlice bir torbaya koyar ve doğrulayıcıya hangi topun hangi torbada olduğunu gösterir. Doğrulayıcı, rastgele bir top seçer ve kanıt sunucusundan bu topu torbadan çıkarıp hangi renk olduğunu söylemesini ister. Eğer kanıt sunucu doğru yanıt verirse, doğrulayıcı ona güvenebilir ve kanıt sunucunun topun rengini bildiğini kabul edebilir.

Buradaki ZKP; doğrulayıcı, kanıt sunucunun rengin doğru olduğunu bildiğini ispatlayabilir, ancak kanıt sunucu, rengin ne olduğunu doğrudan göstermez. Eğer bu işlem birkaç kez yapılırsa, doğrulayıcı kanıt sunucunun doğru bilgiye sahip olduğuna ikna olabilir, ancak rengin ne olduğunu öğrenmez.

10.2 "Waldo Nerede ?" Problemi

Bu problemde, bir kişi "Where's Waldo?" adlı bir çizgi romanda "Waldo"yu bulduğunu iddia eder ve bunu doğrulamak ister. Kanıt sunucu, çizgi romanda "Waldo"yu bulduğunu iddia eder. Doğrulayıcı, kanıt sunucusunun Waldo'yu gerçekten bulduğunu, ancak bu bilgiyi ifşa etmeden doğrulamak ister. Kanıt sunucu, çizgi romanın bir kısmını gösterir, ancak sadece Waldo'yu değil, aynı zamanda etrafındaki diğer öğeleri gizler. Doğrulayıcı, Waldo'yu görmeden sadece doğru yerin göstergelerini (örneğin, diğer karakterler veya çevresel unsurlar) izleyerek Waldo'nun bulunduğuna dair güvence alır.

Buradaki ZKP; kanıt sunucu, doğrulayıcıyı Waldo'nun yerini bulduğuna ikna eder, ancak doğrulayıcı, Waldo'nun tam olarak nerede olduğunu öğrenmez. Bu durum, doğrulayıcının kanıt sunucunun bilgiyi ifşa etmeden. doğru bilgiyi bildiğini kabul etmesini sağlar.

10.3 Interactive Zero-Knowledge Proof

Bu türde kanıt sunucu ve doğrulayıcı arasında bir dizi interaktif (karşılıklı) adım gerçekleşir. Yani, kanıt sunucu ile doğrulayıcı arasındaki etkileşim, kanıtın geçerliliğini doğrulamak için gereklidir. Kanıt sunucu ve doğrulayıcı arasındaki iletişim birden fazla adımda gerçekleşir. Kanıt sunucu, doğrulayıcı tarafından yapılan rastgele sorulara cevap verir. Doğrulayıcı, kanıt sunucusunun doğru bilgiye sahip olduğunu doğrular, ancak kanıt sunucusunun bildiği gerçek bilgi hakkında hiçbir şey öğrenmez. Birçok etkileşimden sonra doğrulayıcı, kanıt sunucusunun gerçekten doğru bilgiye sahip olduğuna ikna olur. IZKP'lerde her etkileşimde doğrulayıcı, kanıt sunucusunun doğru bilgiye sahip olduğunu %100 kesinlikle öğrenmez. Bunun yerine, belirli bir olasılıkla doğrulayıcı kanıt sunucusunun doğru bilgiye sahip olduğuna kanaat getirir ancak bu olasılık çok sayıda etkileşim ile sıfıra yaklaşır.

1. Kanıt sunucu, doğrulayıcıya bir bilgi verir ancak bu bilgi doğrudan açıklanmaz.
2. Doğrulayıcı, rastgele sorular sorar veya belirli bir bilgi hakkında daha fazla ayrıntı ister. Bu sorular, kanıt sunucusunun gerçek bilgiye sahip olup olmadığını anlamaya yöneliktir.
3. Kanıt sunucu, doğrulayıcının sorularına doğru yanıtlar verir. Bu yanıtlar, kanıt sunucusunun belirli bir bilgiye sahip olduğunu ispatlar.
4. Bu süreç birkaç kez tekrarlanabilir. Kanıt sunucu, her seferinde doğru yanıtlar vererek doğrulayıcıyı ikna eder.

10.4 Non-Interactive Zero-Knowledge Proof

Non-Interactive ZKP, kanıtın yalnızca tek bir mesajla doğrulayıcıya sunulması prensibine dayanır. Bu türde, kanıt sunucu ve doğrulayıcı arasında karşılıklı soru-cevap aşamaları yoktur. Bunun yerine, kanıt sunucu tek bir mesajla kanıtı iletir ve doğrulayıcı bu kanıtı kullanarak doğrulama işlemini gerçekleştirir. Kanıt sunucu, tek bir mesaj ile kanıtı sunar ve doğrulayıcı bu kanıtı inceleyerek geçerliliği doğrular. Bir kez oluşturulmuş bir kanıt, sınırsız sayıda doğrulayıcı tarafından incelenebilir. Bu da sistemin verimliliğini artırır. Bu, zaman tasarrufu sağlar ve işlemi hızlandırır. Tıpkı IZKP'lerde olduğu gibi, kanıt sunucu doğru bilgiye sahip olduğunu ispatlar, ancak doğrulayıcı kanıt sunucusunun bilgilerini öğrenmez. Non-interactive ZKP'ler, özel matematiksel yapılar ve kriptografik teknikler kullanılarak oluşturulur. Fiat-Shamir Yöntemi, bu tür ZKP'lerin en yaygın kullanılan yapılarından biridir.

Fiat-Shamir Heuristic (FSH), interaktif ZKP'yi non-interaktif hale getirmek için kullanılan bir tekniktir. Bu yöntem, etkileşimi ortadan kaldırmak için kriptografik hash fonksiyonları kullanır. Fiat-Shamir Heuristic, doğrulayıcıyı simüle eder ve kanıt sunucu, doğrulayıcıya gerekli bilgileri gönderecek şekilde tek bir mesaj oluşturur. Doğrulayıcı bu mesajı doğrulamak için önceden belirlenmiş bir kriptografik hash fonksiyonunu kullanarak doğrulamayı yapar. FSH, rastgele soruları doğrudan hash fonksiyonu ile belirler ve bu, etkileşim gerektirmeyen bir doğrulama süreci oluşturur. Bu sayede kanıt sunucu, doğrulayıcıya bir mesaj gönderir ve doğrulayıcı, bu mesajı doğrulamak için önceden belirlenmiş bir kuralı (örneğin hash fonksiyonu) kullanır.

1. Kanıt sunucu, doğrulayıcıya bir bilgi kanıtı gönderir. Bu kanıt, yalnızca tek bir mesajda sunulur.

2. Kanıt sunucu, doğrulayıcıya bir kanıt gönderir. Bu kanıt bir kriptografik imza veya hash içerebilir.

3. Doğrulayıcı, kanıtı alır ve önceden belirlenen matematiksel yapıları kullanarak bu kanıtın geçerliliğini kontrol eder. Eğer kanıt geçerliyse, doğrulayıcı kanıt sunucusunun doğru bilgiye sahip olduğunu kabul eder.

10.5 Perfect Zero-Knowledge Proof

Perfect ZKP, doğrulayıcıya, kanıt sunucusunun doğru bilgiye sahip olduğuna dair hiçbir bilgi vermezken aynı zamand tam gizlilik (perfect privacy) ve tam güvenlik (perfect soundness) özelliklerini sağlar. Bu tür bir kanıtın doğru olduğuna dair hiçbir şüpheye yer bırakmaz ve doğrulayıcı hiçbir bilgi elde etmeden doğrulama işlemini tamamlar. Diğer ZKP türlerinde probabilistic (olasılıksal) güvenlik sağlanırken, PZKP'de güvenlik ve gizlilik kesin olarak sağlanır.

- **Tam Gizlilik (Perfect Privacy)**: Kanıt sunucusunun sahip olduğu bilginin doğrulayıcıya tamamen gizli kalması gerektiğini ifade eder. Doğrulayıcı, kanıtı incelediğinde, kanıtın içeriğini asla öğrenemez. Yani doğrulayıcı yalnızca doğru bilgiye sahip olduğunun doğruluğunu onaylar, ancak bu bilgiyi öğrenmez.
- **Tam Güvenlik (Perfect Soundness)**: Kanıt sunucusunun yanlış bilgi sunması, yani yalan söylemesi, sıfır olasılıkla mümkün olmalıdır. Bu, kanıtın geçerliliğini bozan hiçbir hata veya çelişki olmaması anlamına gelir. Eğer kanıt sunucu doğru bilgiye sahip değilse, doğrulayıcı bunun farkına varır. Yani, geçerli olmayan bir kanıtın doğrulayıcıya sunulması imkansızdır.

10.6 Statistical Zero-Knowledge Proof

Statistical ZKP, doğrulama sürecinde doğrulayıcıya hiçbir bilgi vermeden doğrulayıcının belirli bir iddianın doğru olduğuna ikna edilmesini sağlar. Ancak, tam gizlilik ve tam güvenlik yerine olasılıksal güvenlik sağlar. Yani, doğrulayıcının doğru bilgiye sahip olduğuna dair kesin bir garanti vermez, ancak verilen iddianın doğru olduğu konusunda yüksek olasılıkla güvenilir bir sonuç elde eder. Yani doğrulayıcıya verilen kanıt doğru olabilir, ancak bu doğruyu bulma olasılığı bir denemede %100 değildir. Ancak, genellikle birkaç deneme ile doğrulayıcı doğruya ulaşabilir. Bu, kanıt sunucusunun yanlış bilgi verdiği durumların düşük olasılıkla gerçekleşmesini sağlar. Ancak yanlış bir kanıtın doğrulayıcıya kabul edilmesi tamamen imkansız değildir.

10.7 Computational Zero-Knowledge Proof

Computational ZKP, hesaplamalı güvenlik sağlar, yani doğrulayıcılar doğru bilgiye sahip olma konusunda yüksek bir güvence alırken, bu doğrulama işlemi belirli bir hesaplama gücüne dayanır. Bu tür kanıtlar, doğrulayıcıyı, belirli bir iddianın doğruluğuna ikna etmek için kullanılan hesaplama süreçlerinin güvenliğini ve gizliliğini sağlamak amacıyla tasarlanmıştır. Computational ZKP tam gizlilik sağlamaz. Bunun yerine, doğrulayıcıya belirli bir iddianın doğruluğunu sağlamanın yollarını sunar. Ancak doğrulayıcı, kanıtın doğruluğuna tam güven duyarken, kanıt sunucusunun iddia ettiği hakkında bilgi edinmez.

11 Quantum Cryptography

Kuantum kriptografi, kuantum mekaniğinin temel ilkelerine dayanan bir şifreleme yöntemidir. Diğer yöntemlerden farklı olarak, veri güvenliğini matematiksel hesaplamaların karmaşıklığından ziyade fiziksel prensiplere dayandırır. Bilgiler, kuantum bitleri (qubit) ile temsil edilir. Qubit'ler 0, 1 veya bu ikisinin süperpozisyonu şeklinde olabilir. Kuantum Kriptografi, Heisenberg Belirsizlik İlkesini kullanır. Kuantum parçacığının özelliklerini ölçmek, o parçacığın durumunu değiştirir. Bu özellik, dinleme girişimlerini tespit etmeyi sağlar. Kuantum durumları, birbiriyle çakışmayan bazlarda ölçüldüğünde kesin bilgi vermez. Bu, iletişim güvenliğini artırır. Dolaşıklık (Entanglement), iki veya daha fazla parçacığın durumlarının birbirine bağlı olması durumudur. Bu bağlantı, fiziksel olarak ayrı yerlerde bile ölçümler arasında güçlü bir bağıntı sağlar.

11.1 BB84 Algorithm

BB84 Algoritması, 1984 yılında Charless Bennett ve Gilles Brassard tarafından geliştirilen ilk kuantum anahtar dağıtım protokolüdür.

1. Alice, rastgele bir bit dizisi ve baz dizisi seçer. Bitleri seçilen bazlara göre kuantum durumlarına dönüştürür ve Bob'a gönderir.
2. Bob, rastgele bazlar seçerek kuantum bitlerini ölçer. Bazlar doğru seçildiyse, Alice'in gönderdiği biti doğru şekilde ölçer; aksi takdirde rastgele bir sonuç elde eder.
3. Alice ve Bob, bir kanal üzerinden hangi bazları kullandıklarını paylaşır, bitlerin kendisini paylaşmazlar. Sadece aynı bazda ölçülen bitler korunur ve diğerleri atılır.
4. Kalan bitler, ortak bir gizli anahtar olarak kullanılır.
5. Alice ve Bob, anahtarın bir kısmını karşılaştırarak Eve'in dinleme girişimlerini tespit eder. Eğer müdahale varsa, kuantum durumlarındaki değişimden dolayı hata oranı artar.

11.2 EPR-Ekert Protocol

EPR-Ekert Protokolü, 1991 yılında Artur Ekert tarafından önerilen bir kuantum anahtar dağıtım protokolüdür. Bu protokol, Einstein-Podolsky-Rosen (EPR) dolaşıklığı üzerine kuruludur.

1. Alice ve Bob, dolaşık kuantum parçacıkları alır. Bu parçacıklar, birbirinden bağımsız iki uzak yerde ölçülür.
2. Alice ve Bob, ölçümlerini farklı yönlerde rastgele yapar. Ölçüm sonuçları, kuantum mekaniğinin yasaları nedeniyle güçlü bir şekilde korelasyonludur.

3. Eğer Eve, kuantum kanalı dinlemeye çalışırsa, dolaşıklık bozulur ve korelasyonlar değişir. Bu, dinleme girişimlerini tespit etmeyi sağlar.

4. Alice ve Bob, ölçüm sonuçlarının korelasyonuna dayanarak bir gizli anahtar oluşturur.

12 Post-Quantum Cryptography

Post-Kuantum Kriptografi, kuantum bilgisayarların geleneksel kriptografi yöntemlerini tehdit etmesinden dolayı ortaya çıkmıştır. Geleneksel şifreleme yöntemleri büyük ölçüde matematiksel problemlerin çözüm zorluğuna dayanır. Ancak kuantum bilgisayarlar, güçlü algoritmalar sayesinde bu matematiksel problemleri etkili bir şekilde çözebilir. Post-Kuantum Kriptografi, kuantum bilgisayarların bu potansiyel tehditlerine karşı dayanıklı algoritmalar sunmayı amaçlar. Bu algoritmalar, kuan tum bilgisayarların güçlü işlem kapasitesine dayanacak şekilde tasarlanmıştır. Sadece kuantum saldırıları değil, aynı zamanda klasik bilgisayarların saldırılarına karşı da dayanıklıdır. Henüz tam olarak benimsenmiş standartlar yoktur ve standartlaştırma süreci devam etmektedir. PQC, problem kategorileri;

- **Izgara (Lattice)**: Çok boyutlu geometrik ızgaralarda kısa vektörlerin bulunması zorluğuna dayanır. Kullanılan algoritmalar: Learning with Errors (LWE), NTRU.
- **Kodlama (Code)**: Hata düzeltme kodlarının çözümündeki zorluğa dayanır. Kullanılan algoritmalar: McEliece.
- **Çok Değişkenli Polinomlar (Multivariate Polynomials)**: Çok değişkenli polinomların sıfırlarının bulunmasındaki zorluğa dayanır. Kullanılan algoritmalar: Rainbow, UOV (Unbalanced Oil and Vinegar).
- **Hash**: Güvenli hash fonksiyonlarına dayanır. Kullanılan algoritmalar: SPHINCS+.
- **Diğer**: Farklı matematiksel zorluklardan yararlanır. Kullanılan algoritmalar: Supersingular Elliptic Curve Isogeny (SIEC).

13 ———— BLOCKCHAIN ————

14 Blockchain

Blockchain, dijital bilgilerin (verilerin) dağıtık bir yapıda saklandığı, değiştirilemez bir güvenli kayıt sistemidir. Bloklar halinde organize edilen bir veri tabanıdır. Her block, kendinden önceki bloğa bir kriptografik bağl ile bağlıdır ve bu yapı bir zincir oluşturur.

- **Dağıtık Defter (Distributed Ledger)**: Ağdaki tüm katılımcılar (node) aynı defterin bir kopyasını tutar.
- **Değiştirilemezlik**: Bir blok onaylandığında, içeriği değiştirilemez hale getlir. Bu, sistemi manipülasyona karşı koruma sağlar.
- **Şeffeflık**: Blokchain üzerindeki işlemler herkes tarafından görüntülenebilir.
- **Merkeziyetsizlik**: Herhangi bir merkezi otoriteye bağlı olmadan, ağdaki tüm katılımcılar birbirleri arasında eşit haklara sahiptir.

14.1 Merkezi ve Merkeziyetsiz Sistemler

Merkezi sistemde tüm işlemler ve veriler tek bir otorite (merkezi sunucu) tarafından kontrol edilir. Tüm kullanıcılar bu merkezi otoriteye bağlıdır. Örneğin bankalar, sosyal medya platformları, kimlik yönetim sistemleri. Merkeziyetsiz sistemde veriler ve işlemler, birden fazla katılımcının olduğu bir ağ üzerinde dağıtık bir şekilde yönetilir. Tek bir otoriteye ihtiyaç duyulmaz. Örneğin dağıtık dosya paylaşım sistemleri, bitcoin ve ethereum gibi blockchain ağları.

- **Kontrol ve Yönetim**: Merkezi sistem, tek bir otorite tarafından kontrol edilir. Merkeziyetsiz sistemde, ağa katılan her düğüm eşit haklara sahiptir; otorite yoktur.
- **Veri Depolama**: Merkezi sistemde veriler merkezi bir sunucuda depolanır. Merkeziyetsiz sistemde veriler, ağdaki tüm kullanıcılar arasında dağıtılmış şekilde saklanır.
- **Güvenlik**: Merkezi sistemde, merkezi sunucu saldırıya uğradığında tüm sistem riske girer. Merkeziyetsiz sistemde, güvenlik ağdaki tüm düğümler tarafından sağlanır. Merkezi bir zayıf nokta yoktur.
- **Şeffaflık**: Merkezi sistemde, şeffaflık sınırlıdır; veriler otoritenin kontrolündedir. Merkeziyetsiz sistemde, işlemler herkese açık veya kısmen açık olabilir.

- **Kapsam ve Ölçek**: Merkezi sistemde, merkezi yapı ölçeklenebilir, ancak yoğun trafik durumunda darboğaz oluşabilir. Merkeziyetsiz yapılar daha dayanıklıdır ancak ölçeklenebilirlik için optimizasyon gerektirir.
- **Aracılar**: Merkezi sistemde işlemler genellikle bir aracı üzerinden gerçekleşir. Merkeziyetsiz sistemde işlemler doğrudan iki taraf arasında gerçekleşir.
- **Performans**: Merkezi sistemde işlemler hızlıdır, çünkü tek bir otorite işlemleri doğrular. Merkeziyetsiz sistemde doğrulama işlemleri daha yavaştır.
- **Giderler**: Merkezi sistemde yönetim ve bakım maliyetlidir. Merkeziyetsiz sistemde düşüktür.
- **Arıza Dayanıklılığı**: Merkezi sistemde, bir sunucu çöktüğünde sistem tamamen durabilir. Merkeziyetsiz sistemde, dağıtık yapı sayesinde tek bir düğümün çökmesi sistemi etkilemez.
- **Veri Manipülasyonu**: Merkezi sistemde, veri merkezi otorite tarafından değiştirilebilir. Merkeziyetsiz sistemde, bloklar onayladıktan sonra veri değiştirilemez, manipülasyona kapalıdır.

14.2 Blockchain Yapısının Bileşenleri

14.2.1 Blok

Blockchain'in temel yapı taşıdır. Her blok, belirli bir veriyi veya işlemleri saklar. Zincirdeki her blok, bir önceki bloğa bağlıdır. Bu bağlantı, sistemin güvenliğini artırır. Her blok, blok başlığı ve blok verisi olmak üzere iki kısımdan oluşur:

- **Başlık (Block Header)**:
  - **Önceki Blok Hash'i**: Zincirdeki önceki bloğun benzersiz hash değeri.
  - **Merkle Ağacı Kökü (Merkle Root)**: Bloğun içindeki tüm işlemleri özetleyen bir hash değeri.
  - **Zaman Damgası (Timestamp)**: Bloğun oluşturulduğu zamanı belirtir.
  - **Nonce**: Hash oluşturma sürecinde kullanılan rastgele bir sayı.
- **Veri (Block Data)**: İşlem kayıtları veya saklanması gereken diğer veriler.

14.2.2 Zincir

Zincir, blokların ardışık ve kronolojik bir şekilde birbirine bağlanarak oluşturduğu yapıdır. Her blok, önceki bloğun hash değerini içerir. Bu, bağlantı zincirini oluşturur. Zincirdeki bir bloğun değiştirilmesi, sonraki tüm blokların değiştirilmesini gerektirir, bu da neredeyse imkansızdır. Tüm zincir, ağdaki tüm düğümlerde saklanır.

14.2.3 Nonce

Nonce, "Number Only Used Once" kelimelerinin kısaltmasıdır. Bir blok hash'ini belirli bir zorluk seviyesine uydurmak için kullanılan rastgele bir sayıdır. Nonce, madencilik sürecinde hash değerini istenen zorluk seviyesine getirmek için sürekli değiştirilir. Doğru nonce bulunduğunda blok onaylanır ve zincire eklenir. Nonce değerini bulmak zaman alıcıdır ve hesaplama gücü gerektirir.

14.2.4 Hash

Hash, giriş verilerini sabit uzunlukta benzersiz bir çıkışa dönüştüren matematiksel bir algoritmadır. Aynı veri için, her zaman aynı hash değeri üretilir (deterministik). Hash işlemleri hızlı yapılır. Veri küçük bir değişikliğe uğrasa bile hash tamamen farklı bir sonuç üretir. Hash'ten orijinal veriye geri dönüş yapılamaz. Her blok, önceki bloğun hash'ini içerir ve bu da zinciri oluşturur. Hash, verilerin değiştirilmediğini doğrulamak için kullanılır.

14.2.5 Konsensüs Mekanizması

Konsensüs mekanizması, ağdaki tüm düğümlerin bir bloğun geçerliliği üzerinde hemfikir olmasını sağlayan bir protokoldür. Blockchain'in güvenliğini salğar. Ağda sahte işlemlerin onaylanmasını engeller. Örneğin:

- **Proof of Work (PoW)**: Bitcoin'de kullanılır. Madencilik yoluyla zorlu matematiksel problemler çözülerek blok oluşturulur.
- **Proof of Stake (PoS)**: Ethereum'da kullanılır. Stake edilen varlık miktarına göre blok oluşturma yetkisi verilir.

14.2.6 Düğümler

Düğümler, blockchain ağına katılan cihazlardır. Bir düğümün görevi; blockchain verilerini saklamak, işlemleri doğrulamak ve yeni blokların onaylanmasına katılmaktır. Tam Düğüm ve Hafif Düğüm olmak üzere iki tiptir:

- **Tam Düğümler (Full Nodes)**: Blockchain'in tüm verisini saklar ve doğrulama yapar.

- **Hafif Düğümler (Light Nodes)**: Yalnızca işlem doğrulama için gerekli verileri saklar.

14.2.7 Akıllı Sözleşmeler

Akıllı sözleşmeler, blockchain üzerinde çalışan, önceden tanımlanmış kurallara göre otomatik olarak yürütülen kod parçalarıdır. İnsan müdahalesi olmadan otomatik işlem yapar. Ethereum gibi blockchain platformlarında yaygın olarak kullanılır.

14.3 Blockchain Türleri

14.3.1 Public Blockchain

Public Blockchain, herkesin katılabileceği, işlem yapabileceği ve blokları doğrulayabileceği tamamen merkeziyetsiz bir blockchain türüdür. Herhangi bir kullanıcı, kimlik bilgisi vermeden bu tür blockchain ağına katılabilir. Hiçbir merkezi otoriteye bağlı değildir. Tüm işlemler ve veriler herkese açıktır. Proof of Work veya Proof of Stake gibi güçlü konsensüs mekanizmaları ile güvenliği sağlanır. Büyük ölçekle ağlarda işlem süreleri uzun olabilir. Proof of Work kullanan sistemlerde enerji tüketimi yüksektir. Bitcoin, Ethereum, dApps'da kullanılır.

14.3.2 Private Blockchain

Private Blockchain, yalnızca belirli bir grup insanın erişim ve kullanım iznine sahip olduğu kapalı bir blockchain türüdür. Kontrol bir organizasyon veya kurum tarafından sağlanır. Ağdaki düğümler yalnızca davet ile katılabilir. Bir organizasyon veya şirket tarafından yönetilir. Daha az düğüm olduğu için işlemler daha hızlıdır. Yalnızca yetkilendirilmiş kişiler işlemleri görebilir. Tam merkeziyetsizlik sağlanamaz. Bankacılık, sağlık gibi alanlarda kullanılır.

14.3.3 Hybrid Blockchain

Hybrid Blockchain, public ve private blockchain'in bir kombinasyonudur. Belirli kısımlar halka açık olabilirken, diğer kısımlar yalnızca yetkilendirilmiş kullanıcılar tarafından erişilebilir. Hangi verilerin açık olacağı ve hangi verilerin gizli kalacağı kontrol edilebilir. Bazı kararlar merkeziyetsiz bir şekilde alınabilir. Kurulum ve yönetimi daha zordur.

14.3.4 Consortium Blockchain

Consortium Blockchain, birden fazla organizasyonun ortaklaşa işlettiği özel bir blockchain türüdür. Kontrol, konsorsiyumu oluşturan üyeler arasında paylaşılır. Yalnızca konsorsiyuma üye olan organizasyonlar ağa katılabilir. Merkezi kontrol yerine konsorsiyum üyeleri arasında

yetki dağıtılır. Konsorsiyum üyeleri arasında bir güven mekanizması oluşturur. Üyeler arasında işlem şeffaflığı sağlanır. Halka açık kullanım için uygun değildir.

14.4 Byzantine Generals Problem

Bizans Generali Problemi, dağıtık sistemlerde güvenilir bir konsensüs (uzlaşma) mekanizması oluşturmanın zorluğunu ifade eder. Bu problem, düğümlerin bir kısmının kötü niyetli olduğu bir ortamda, sistemin doğrulukla çalışmasını sağlama amacıyla ilgilidir. Senaryo şöyledir;

Bizans İmparatorluğu ordusunun generalleri bir şehri kuşatmıştır. Generaller farklı noktalarda kamp kurmuşlardır ve hepsi ya şehre saldırmalı ya da çekilmelidir. Ancak generaller yalnızca mesajlarla iletişim kurabilir. Bazı generaller ihanet edebilir ve yanlış mesajlar göndererek diğerlerini yanıltabilir. Amaç, ihanet eden generallerin varlığına rağmen, sadık generallerin hemfikir olduğu bir stratejide uzlaşmasıdır.

Bizans Generali Problemi'nin çözümü, sistemin kötü niyetli ya da hatalı aktörlerin (düğümlerin) varlığında doğru bir şekilde çalışabilmesini sağlamaktır. Bu sorunu çözmek için çeşitli konsensüs algoritmaları geliştirilmiştir.

- **Proof of Work (PoW)**: Madenciler, işlemleri doğrulamak için zor matematiksel problemleri çözer. Çözümü ilk bulan ödüllendirilir ve işlemi blok zincirine ekler. Problem çözmek enerji ve kaynak gerektirdiği için kötü niyetli düğümler sistemi ele geçirmek için büyük kaynak ayırmak zorunda kalır.
- **Proof of State (PoS)**: Kullanıcılar, sahip oldukları token miktarı oranında blok üretme hakkı kazanır. Düşük enerji tüketimiyle çalışır.
- **Byzantine Fault Tolerance (BFT)**: Sistem, kötü niyetli aktörlerin oranı toplamının üçte birinden fazla olmadığında doğru bir şekilde çalışabilir. Tüm düğümler mesajları doğrular ve çoğunluk oyuna göre konsensüse ulaşılır.
- **Delegated Proof of Stake (DPoS)**: Belirli düğümler (delegeler) oylama ile seçilir ve blokları doğrulamakla görevlendirilir. Sistem daha hızlı ve enerji verimlidir.

14.5 Madenciler ve Mining İşlemi

Madenci, bir blockchain ağında işlemleri doğrulayan, yeni blokları oluşturan ve bu sürecin sonunda ödüller kazanan katılımcıya denir. Madenciler, blockchain sisteminin güvenliğini sağlar ve merkeziyetsiz yapısını destekler. Madenciler, kazım işlemi (mining) adı verilen bir süreçle çalışır.

Bu süreç, işlemleri doğrulamak, blockchain'e yeni bloklar eklemek ve ağı saldırılara karşı korumak için gereklidir. Madencinin görevi;

Madencilerin temel amacı, hem blockchain ağının işleyişine katkıda bulunmak hem de bunun karşılığında ödül kazanmaktır. Bu ödüller:

- **Blok Ödülleri**: Yeni bir blok oluşturan madenciye verilen ödül.
- **İşlem Ücretleri**: Kullanıcılar tarafından işlemlerinin onaylanması için ödenen küçük ücretler.

Kazım işlemi, bir konsensüs mekanizması ile gerçekleştirilir. Proof of Work yöntemi için kazma işlemleri;

1. Ağdaki kullanıcılar işlemlerini gönderir. Bu işlemler madenciler tarafından alınır ve doğrulanır. Doğrulanan işlemler bir araya getirilerek bir kuyruk oluşturulur.
2. Madenci, işlemleri bir blok içine koyar ve bu bloğun ağın geri kalanıyla uyumlu olduğunu kanıtlamak için çalışmaya başlar.
3. Her blokta bir nonce adı verilen rastgele bir sayı bulunur. Madenci, bu nonce değerini değiştirerek, bloğun hash değerini belirli bir hedefin altına düşürmeye çalışır. Bu süreç, yoğun bir hesaplama gücü gerektirir çünkü hash değerini bulmak için birçok olasılık denenir.
4. Hash, bir bloğun kimlik kartıdır. Madenci, bloğun hash'ini doğru bir şekilde hesapladığında, bu hash diğer düğümler tarafından doğrulanır. Hash değeri, bloğun önceki blokla bağlantısını da içerir, bu da blockchain'in zincir yapısını oluşturur.
5. Madenci başarılı olduğunda, oluşturduğu blok tüm ağa gönderilir. Diğer düğümler bu bloğu doğrular ve zincire ekler.
6. Başarılı madenci, blok ödülü ve işlem ücretlerini alır.

14.6 Coin ve Token

Coin, bağımsız bir blockchain ağı üzerinde çalışan dijital bir para birimidir. Bitcoin gibi coin'ler blockchain ağlarının yerel para birimleridir. Coin'ler kendi blockchain ağları üzerinde çalışır. Coin'lerin işlemleri ve transferleri bu blockchain üzerinde gerçekleşir. Coin'ler bir değer saklama aracı, bir değişim aracı (transfer) veya bir birim olarak kullanılır. Coin'ler, blockchain'in kendisine entegre olan bir madencilik veya konsensüs mekanizması kullanır.

Token, bir blockchain ağı üzerinde çalışan dijital varlıklardır ancak bağımsız bir blockchain ağına sahip değildir. Bunun yerine, başka bir blockchain platformunu kullanır. Token'lar, akıllı sözleşmelerle oluşturulur ve bir blockchain ağına bağlıdır. Token'lar, yalnızca finansal

işlemler için değil, aynı zamanda bir hizmete erişim sağlamak, yönetim hakkı tanımak veya bir varlığın temsili olarak da kullanılabilir. Yeni bir token oluşturmak, bir coin'den çok daha kolaydır çünkü bağımsız bir blockchain geliştirmeye gerek yoktur.

15 NFT (Non-Fungible Token)

NFT, benzersiz token anlamına gelir. NFT'ler, bir blockchain üzerinde saklanan dijital varlıklardır ve her biri benzersizdir. Bu, onları diğer dijital varlıklardan ayıran en önemli özelliktir. NFT'ler değiştirilemez ve takas edilemez (non-fungible) bir yapıya sahiptir, bu da onları fiziksel koleksiyonlarının dijital eşdeğerleri haline getirir. Her NFT'nin kendine özgü bir kimliği ve meta verisi vardır. Örneğin, bir sanat eserinin dijital versiyonu gibi, NFT başka bir NFT ile birebir takas edilemez. NFT'ler blockchain üzerinde saklandığı için kimin sahibi olduğu tamamen şeffaf bir şekilde izlenebilir. Çoğu NFT, bir bütün olarak alınıp satılır ve Bitcoin gibi parçalara bölünemez. NFT'ler genellikle Ethereum üzerinde çalışır ve standartlar (örneğin, ERC-721 ve ERC-1155) kullanılarak oluşturulur. Bu, geliştiricilerin yeni NFT'ler üretmesini kolaylaştırır. NFT'ler, kripto cüzdanlarda saklanabilir ve blockchain üzerinde kolayca transfer edilebilir. NFT'ler, akıllı sözleşmeler aracılığıyla oluşturulur. Bu sözleşmeler, NFT'nin benzersiz kimliğini ve meta verilerini saklar. NFT'lerin içinde dosyalar (görseller, videolar, müzik vb.) saklanmaz. Bunun yerine, bu dosyaların bağlantıları saklanır. Minting (NFT Oluşturma), NFT'ler bir dijital varlığın blockchain üzerinde kayıtlı hale getirilmesiyle oluşturulur. Bu işlem NFT platformlarında yapılır.

15.1 Cryptopunks Token Sözleşmesi

Cryptopunks, 2017 yılında Larva Labs tarafından başlatılan ilk NFT projelerinden biridir. Cryptopunks, NFT'lerin potansiyelini gösteren ilk projelerden biridir.Ethereum blockchain üzerindeki akıllı sözleşmelerle yönetilir. 10.000 benzersiz piksel sanat eseri koleksiyonundan oluşur. Her Cryptopunk, ERC-721 standardına dayalı bir NFT'dir. Her punk benzersizdir ve belirli bir özellik setine sahiptir. Cryptopunks, kendi özel token sözleşmesine sahiptir. Bu sözleşme:

- Hangi adresin hangi punk'a sahip olduğunu kaydeder.
- Kullanıcıların Cryptopunks alıp satmasını sağlra.
- Her punk'un özelliklerini ve benzersiz kimliğini içerir.

Cryptopunks, NFT alanında devrim yaratmış ve NFT'lerin benimsenmesinde önemli bir rol oynamıştır. Birçok kişi için, NFT ekosisteminin başlangıcını temsil eder.

15.2 NFT Standartları

ERC (Ethereum Request for Comments), ethereum ekosisteminde, standartların ve protokollerin önerilmesi ve tartışılması için kullanılan bir süreçtir. ERC terimi, internet protokollerinin geliştirilmesinde kullanılan

"Request for Comments (RFC)" sürecinden esinlenmiştir. Bu süreçte, yeni bir standart önerisi topluluğun görüşüne sunulur ve tartışmalar sonucunda kabul edilir veya reddedilir. ERC, ethereum ağında, akıllı sözleşmeler ve token standardizasyonu için önerilerin sunulması ve uygulamaya geçirilmesi amacıyla kullanılır. Öneriler, Github üzerindeki "Ethereum IPs" reposunda önerilir.

EIP (Ethereum Improvement Proposal), ethereum ağında yapılacak geliştirme ve iyileştirme önerileri için kullanılan bir süreçtir. EIP terimi, yazılım geliştirme projelerinde kullanılan iyileştirme önerisi süreçlerinden türetilmiştir. Ethereum'un teknik altyapısına dair her türlü değişiklik veya geliştirme için kullanılır.

15.2.1 ERC-721 Standardı

ERC-721, Ethereum üzerindeki ilk NFT standardıdır. Her bir tokenin benzersiz olmasını sağlar. Her NFT'nin kendine özgü bir kimliği (token ID) vardır. Her token farklıdır ve bir diğer token ile takas edilemez. Her NFT'ye ait meta veriler, kimlik bilgileri ve varlık detayları bu standart üzerinde saklanır. Bir tokenin kime ait olduğu açık bir şekilde blockchain üzerinde saklanır. NFT'ler bir kullanıcıdan diğerine transfer edilebilir. CryptoKitties, benzersiz dijital kedi karakterleri oluşturup ticaret yapmayı sağlayan ilk popüler ERC-721 NFT uygulamasıdır.

15.2.2 ERC-1155 Standardı

ERC-1155, hem fungible (değiştirilebilir) hem de non-fungible (değiştirilemez) tokenleri destekleyen hibrit bir standarttır. Aynı sözleşme altında birden fazla token türünün oluşturulmasına olanak tanır. Transfer işlemleri optimize edilmiştir; birden fazla token tek bir işlemde transfer edilebilir. Meta veri ve sahiplik bilgileri daha az yer kalpar. Oyunlarda, hem benzersiz varlıklar (karakterler) hem de değiştirilebilir varlıklar (oyun içi altınlar) oluşturmak için kullanılır. Gods Unchained, ERC-1155 standardını kullanan bir dijital koleksiyon kart oyunudur.

15.2.3 ERC-998 Standardı

ERC-998, kombinasyonlu NFT'leri (Composable NFT) destekleyen bir standarttır. NFT'lerin diğer NFT'leri veya fungible tokenleri sahiplenmesine olanak tanır. Bir NFT başka bir NFT'nin veya tokenin sahibi olabilir. Dijital varlıklar, birden fazla bileşenden oluşan bir yapı oluşturabilir. Örneğin, bir karakterin kıyafetleri ve aksesuarları ayrı NFT'ler olarak tanımlanabilir ve bu karakterin sahibi bunları tek bir varlık olarak yönetebilir.

15.2.4 EIP-2309 Standardı

ERC-721'in bir uzantısı olarak tasarlanmıştır ve büyük miktarda NFT'lerin daha verimli bir şekilde mint edilmesine (oluşturulması) olanak tanır. Bir işlemde çok sayıda NFT oluşturulabilir. Çoklu NFT işlemlerini tek bir işlemde birleştirerek gaz ücretlerini düşürür.

16 Konsensüs Algoritmaları

16.1 Nakamoto Consensus

Nakamoto Konsensüsü, Bitcoin'in temelini oluşturan ve birçok blockchain ağına ilham veren bir konsensüs mekanizmasıdır. İlk kez Satoshi Nakamoto tarafından 2008 yılında Bitcoin whitepaper'ında tanımlanmıştır. Bu konsensüs mekanizması, dağıtık bir ağdaki düğümlerin (nodes), merkezi bir otorite olmadan üzerinde uzlaştıkları bir işlem sırasını sağlamasını hedefler. Merkezi bir otoriteye ihtiyaç duymadan, ağdaki tüm düğümlerin üzerinde anlaştığı bir defter oluşturur. Dijital para birimlerinde bir varlığın aynı anda birden fazla yerde harcanmasını önler (çifte harcama problemi). Ağın dürüst düğümleri tarafından desteklenen en uzun zincir (Longest Chain Rule) kabul edilir, bu da kötü niyetli saldırıları zorlaştırır. Her düğüm, tüm işlemleri doğrular ve blok zincirine eklenen her şey şeffaf bir şekilde kaydedilir.

16.1.1 Çalışma Adımları

1. Nakamoto Konsensüsü, Proof of Work (PoW) mekanizmasını kullanır. Madenciler, belirli bir zorluğu karşılayan bir hash değeri bulmak için kriptografik bir bulmacayı çözer.
2. Bir madenci, geçerli bir blok oluşturup bulmacayı çözdüğünde, bu bloğu ağa yayınlar. Diğer düğümler, bloğu doğrular ve kabul eder.
3. En Uzun Zincir kuralı sayesinde ağdaki tüm düğümler, en uzun ve en fazla iş gücünü temsil eden zinciri doğru zincir olarak kabul eder. Yeni bir blok üretildiğinde, bu blok yalznıca en uzun zincire ekler.
4. Eğer ağda aynı anda iki farklı blok üretilirse (fork), düğümler, sonunda en fazla işlem gücüyle desteklenen zinciri seçer.

16.1.2 Güvenlik

- **Proof of Work (PoW)**: Yeni bir blok eklemek için çok yüksek bir hesaplama gücü gerektirir. Bu, saldırganların zincire müdahale etmesini maliyetli ve zor hale getirir.
- **En Uzun Zincir Kuralı (Longest Chain Rule)**: Sistemin, en uzun ve en fazla işlem gücüyle desteklenen zinciri kabul etmesi, kötü niyetli bir aktörün alternatif bir zincir oluşturmasını çok zorlaştırır.
- **Zorluk Seviyesi**: Bitcoin gibi sistemlerde, blokların ortalama belirli bir sürede bir oluşturulmasını sağlamak için zorluk seviyesi düzenli olarak ayarlanır. Bu, saldırganların blok üretme hızını artırmasını zorlaştırır.

16.2 Proof of Work (Pow)

PoW, ağdaki katılımcıların, belirli bir matematiksel problemi çözmesi için yoğun hesaplama gücü harcamasını gerektirir. Bu problemin çözümü "iş kanıtı" olarak değerlendirilir ve çözen madenci, blok zincirine bir blok ekleyerek ödüllendirilir. Bu yöntem, ağdaki konsensüs mekanizmasını (uzlaşmayı) sağlar ve kötü niyetli aktörlerin sistemi manipüle etmesini zorlaştırır. Yüksek enerji tüketir. Yavaş ve düşük işlem gücüne sahiptir. Daha güçlü donanıma sahip madenciler daha avantajlı duruma geçer.

Bitcoin ağında PoW şu şekilde işler:

1. Bir kullanıcı, bir işlemi gönderir.
2. İşlemler bir blokta toplanır.
3. Madenciler, bu bloğu onaylamak için hash problemini çözmeye çalışır.
4. İlk çözen madenci, bloğu doğrular ve blockchain'e ekler.
5. Diğer madenciler bu bloğu kabul eder ve işlem geçerli hale gelir.

16.2.1 Çalışma Adımları

1. Bir madenci, yeni bir blok oluşturmak için işlemleri gruplar ve bunları hash fonksiyonları ile özetlemeye başlar. PoW'da çözülmesi gereken matematiksel problem, bir hash fonksiyonunun beilrli bir zorluğa (difficulty) uygun bir çıktı üretmesidir.
2. Madenciler, nonce (rastgele sayı) değerini değiştirerek bu hash'i bulmaya çalışır.
3. Ağa bağlı madencilerin sayısına ve hesaplama gücüne göre problem zorluğu ayarlanır. Bitcoin ağı, her 2016 blokta bir (yaklaşık 2 haftada bir) bu zorluğu otomatik olarak ayarlar.
4. İlk doğru hash'i bulan madenci, bloğunu ağdaki diğer düğümlere gönderir. Diğer düğümler, bloğun geçerli olup olmadığını kontrol eder. Eğer geçerliyse, blok zincirine eklenir ve madenci ödüllendirilir.

16.2.2 Güvenlik

PoW'un temel gücü, bir blok eklemenin veya geçmişteki bir bloğu değiştirmenin çok yüksek bir hesaplama maliyeti gerektirmesidir.

- **Enerji ve Zaman Maliyeti**: Bir bloğu değiştirmek veya sahte bir blok eklemek için, kötü niyetli bir aktörün mevcut tüm PoW hesaplamalarını yeniden yapması gerekir. Bu, büyük bir enerji ve zaman maliyetine yol açar.

- **Ağ Gücü (Hashrate)**: Manipülasyon yapmak isteyen bir aktör, ağdaki toplam hash gücünün %51'inden fazlasını kontrol etmek zorundadır (%51 saldırısı). Ancak, büyük bir blockchain ağında bu düzeyde güç toplamak ekonomik olarak imkansızdır.
- **Kriptografik Güvenlik**: Hash fonksiyonları tek yönlüdür ve geri dönüşü yoktur. Bu nedenle, bir hash değerine ulaşmak için rastgele denemeler yapılır. Bu sürecin tahmin edilememesi manipülasyonu zorlaştırır.

16.3 Proof of Stake (PoS)

PoS, blokların oluşturulması ve doğrulanmasını, madencilik hesaplama gücünden ziyade, kullanıcıların sahip oldukları varlık miktarına ve bu varlıkları "stake" (kilitleme) etmelerine dayanır. Kullanıcılar, belirli miktarda coin veya token'lerini bir süreliğine kilitler ve bu miktar, onların blok doğrulama hakkını kazanma olasılığını artırır. PoS, kripto para ağına katılan doğrulayıcıların, sahip oldukları kripto para miktarına ve ağda bu varlıkları ne kadar süre stake ettiklerine göre ödüllendirilmesini sağlar.

Ethereum, başlangıçta PoW kullanan bir ağdı ancak Ethereum 2.0 güncellemesiyle PoS'e geçti. Ethereum'da PoS şu şekilde işler:

1. Kullanıcılar, minimum 32 ETH stake ederek doğrulayıcı olabilir.
2. Doğrulayıcılar, blok üretmek veya doğrulamak için rastgele seçilir.
3. Doğru çalışan doğrulayıcılar ödüllendirilir, kötü niyetli doğrulayıcılar cezalandırılır.

16.3.1 Çalışma Adımları

1. Kullanıcılar ellerindeki coin'leri ağın belirlediği bir süre boyunca kilitler. Kilitlenen miktar, doğrulayıcı seçilme olasılığını artırır. Bu süreç, bir tür piyango veya rastgele seçim algoritması ile desteklenir.
2. Ağ, bir doğrulayıcı seçmek için şu kriterlere bakar:
    - **Stake Miktarı**: Daha fazla coin stake edenin seçilme olasılığı daha yüksektir.
    - **Stake Süresi**: Stake edilen varlıkların ağda ne kadar süre kilitli kaldığı göz önünde bulundurulur.
    - **Rastgelelik**: TAm merkeziyetsizlik sağlamak için rastgelelik unsurları eklenir.
3. Seçilen doğrulayıcı, yeni bloğu oluşturur ve ağa önerir. Diğer doğrulayıcılar, bloğun geçerliliğini kontrol eder. Eğer blok geçerliyse, blok zincire eklenir.
4. Doğrulayıcı, stake ödülü alır.

16.3.2 Güvenlik

- **Slashing Mekanizması**: Bir doğrulayıcı, blok oluşturma veya doğrulama sırasında hile yaparsa, bu mekanizma sayesinde varlıkların bir kısmını veya tamamını kaybedebilir. Bu durum, kötü niyetli davranışı finansal olarak cezalandırır ve ağın güvenliğini sağlar.

- **Ekonomik Teşvikler**: PoS, sisteminde bir saldırı yapmak için ağın büyük bir kısmını kontrol etmek gerekir. Bu çok maliyetlidir.
- **Konsesüs Çeşitliliği**: Bazı PoS sistemleri, kötü niyetli doğrulayıcıların işlemleri çift harcamasını veya zinciri çatal yapmasını önlemek için ilave mekanizmalar kullanılır.
- **Rastgelelik**: Blok doğrulayıcıların seçim süreci rastgele olduğu için kötü niyetli bir doğrulayıcının sürekli seçilmesi imkansız hale gelir.

16.4 Delegated Proof of Stake (DPoS)

Delegated Proof of Stake (DPoS), Proof of Stake (PoS) konsensüs algoritmasının daha hızlı, daha ölçeklenebilir ve topluluk odaklı bir versiyonudur. Bu sistemde, ağ katılımcıları (token sahipleri), "delegeler" olarak adlandırılan bir grup doğrulayıcı seçer. Delegeler, ağdaki işlemleri doğrulamak ve yeni bloklar oluşturmakla sorumludur. Delegelerin kötü niyetli davranışlarını cezalandıracak mekanizmalar içerir.

16.4.1 Delege Seçme

- Token sahipleri, ellerindeki token miktarına göre oy hakkına sahiptir. Oylama, ağı güvence altına almak için delegelerin seçilmesini sağlar.
- Delgeler, toplulukta güvenilirlik, şeffaflık ve sorumluluk sahibi olarak tanınırsa daha fazla oy alma şansına sahiptir.
- Token sahipleri, herhangi bir delegenin performansından memnun kalmazsa oylarını geri çekebilir ve başka bir delegeye oy verebilir.
- Bazı ağlar, oy gücünü sınırlandırarak büyük token sahiplerinin sistemi domine etmesini önlemek iççin mekanizmalar uygular.

16.4.2 Çalışma Adımları

1. Ağ katılımcıları, ellerindeki token miktarına bağlı olarak delegeleri seçmek için oy kullanır. Her token bir oy hakkı sağlar. Token sahipleri, oylarını istedikleri doğrulayıcıya verebilir ve dilerlerse oylarını başka birine devredebilir.
2. Topluluğun oyları sonucunda, sınırlı sayıda delege seçilir. Bu delegeler, ağın işlem doğrulama ve blok oluşturma görevini üstlenir.
3. Delegeler, sırayla blok oluşturur ve zincire ekler. Eğer bir delege, sırada görevini yerine getirmezse, bir sonraki delege devreye girer.
4. Blok oluştuma görevini başarıyla yerine getiren delegeler, işlem ücretlerinden ve blok ödüllerinden pay alır. Bazı ağlarda delegeler, kazançlarını oy veren token sahipleriyle paylaşabilir.
5. Kötü niyetli veya görevini yerine getiremeyen delegeler, topluluk oylarıyla görevden alınabilir ve yerlerine yeni delegeler seçilebilir.

16.4.3 Güvenlik

- **Delegelerin Sorumluluğu**: Delegeler, token sahipleri tarafından seçildiği için topluluğa hesap verme yükümlülüğü taşır. Görevlerini kötüye kullanan delegeler, oy kaybederek sistemden dışlanır.

- **Oylama Mekanizması**: Oylar sürekli olarak yeniden düzenlenebilir. Bu, bir delegenin uzun süre sistemde kötü niyetli davranmasını zorlaştırır.
- **Azınlık Koruma**: Delegelerin sayısı sınırlı olduğu için uzlaşma daha hızlı sağlanır ve bu durum saldırganların ağın %51'inde kontrol sağlamasını zorlaştırır.
- **Ekonomik Teşvikler**: Delegelerin kötü niyetli davranması, yalnızca görevlerini değil aynı zamanda gelecekteki ödülleri de kaybetmelerine neden olur. Ayrıca, delegeler sıklıkla ödüllerini toplulukla paylaştıkları için, ekonomik olarak dürüst kalmaya teşvik edilirler.

16.5 Proof of Authority (PoA)

Proof of Authority (PoA), izinli (permissioned) bir blockchain konsensüs algoritmasıdır. Bu mekanizma, blok oluşturma ve doğrulama yetkisini, güvenilir bir grup doğrulayıcıya (validator) devreder. Doğrulayıcılar, kimliklerini kanıtlar ve ağın güvenliğini sağlamak için güvenilirliklerini ortaya koyarlar. PoA, işlem doğrulama sürecini hızlandırır ve saniyede daha fazla işlem (TPS) sağlar. Özel (private) blockchain ağlarında tercih edilir, çünkü yetkilendirilmiş doğrulayıcılar kullanır. PoW gibi enerji tüketimi yüksek mekanizmalara ihtiyaç duymaz, bu da düşük maliyetli bir sistem sağlar. Kimlik temelli doğrulayıcı seçimi, ağın kötü niyetli kişiler tarafından ele geçirilmesini zorlaştırır.

16.5.1 Çalışma Adımları

1. Ağ, belirli kriterlere göre seçilen ve kimliklerini doğrulayan az sayıda doğrulayıcıyı yetkilendirir. Bu doğrulayıcılar, ağı işletmek ve işlemleri doğrulamakla sorumludur.
2. PoA, doğrulayıcıların kimliklerini kanıtlamalarını gerektirir. Bu, bir doğrulayıcının gerçek kişi veya kurum olması anlamına gelir. Kimlik doğrulama, ağın kötü niyetli kişilerden korunmasını sağlar.
3. Blok üretimi ve doğrulama sırası, önceden tanımlı bir algoritmayla belirlenir. Doğrulayıcılar sırayla blok oluşturur ve ağı senkronize eder.
4. PoA sistemlerinde doğrulayıcıların kazançları genellikle işlemlerden alınan işlem ücretlerine veya belirli bir ödül mekanizmasına dayanır.
5. Az sayıda doğrulayıcı kullanıldığı için saldırıların gerçekleşmesi zorlaşır. Manipülasyon riski doğrulayıcıların kimlikleri ve sorumluluklarıyla minimize edilir.

16.5.2 Güvenlik

- **Kimlik Tabanlı Güven**: Doğrulayıcıların gerçek kimlikleri bilinir ve güvenilirlikleri şeffaftır. Kötü niyetli davranış sergileyen doğrulayıcılar, kimliklerinin ifşa edilmesi nedeniyle itibarlarını kaybeder.
- **Sınırlı Doğrulayıcılar**: Az sayıda doğrulayıcı olduğundan ağın kontrolü kolaylaşır. Saldırı gerçekleştirmek için doğrulayıcıların büyük bir kısmını kontrol etmek gerekir ki bu oldukça zordur.
- **Ceza Mekanizmaları**: Kötü davranışta bulunan doğrulayıcılar, yetkilerini kaybedebilir.
- **Güçlü Merkeziyetçi Kontrol**: Manipülasyon ihtimali, güvenilir bir doğrulayıcı grubu seçilerek minimize edilir.

16.6 Proof of Elapsed Time (PoET)

Proof of Elapsed Time (PoET), Intel tarafından Hyperledger Sawtooth projesi kapsamında geliştirilen bir blockchain konsensüs algoritmasıdır. PoET, madencilik işlemlerinde enerji tüketimini minimuma indirerek, PoW (Proof of Work) gibi yüksek enerji tüketen mekanizmalara alternatif sunar. Bu mekanizma, blok oluşturma hakkını adil bir şekilde dağıtmayı hedefler ve bu süreci donanım tabanlı bir güven mekanizmasıyla sağlar. Her katılımcıya eşit şans tanıyarak, blok üretim sürecinde merkeziyetsizliği korur. Kurumsal ihtiyaçlara uygun, ölçeklenebilir bir çözüm sunar. PoW gibi yoğun hesaplama gücü gerektiren işlemlerden kaçınır, bu nedenle düşük maliyetli donanımlarla çalışabilir.

- **Hyperledger Sawtooth**: PoET'nin ilk uygulandığı blockchain platformudur. İzne dayalı blockchain projelerinde kullanılır.

16.6.1 Çalışma Adımları

1. Her node (düğüm), bir blok oluşturma hakkı kazanabilmek için rastgele bir bekleme süresi (elapsed time) seçer. Bu süre, ağdaki tüm düğümler tarafından belirlenen donanımın güvenilir bir ortamında üretilir.

2. Düğümler, rastgele bekleme sürelerini hesapladıktan sonra bu süre boyunca bekler. Süresi dolan ilk düğüm, blok oluşturma hakkı kazanır.

3. PoET, Intel'in Software Guard Extensions (SGX) teknolojisini kullanarak rastgele bekleme süresinin manipüle edilmesini engeller. SGX, düğümlerin güvenilir bir şekilde çalışmasını sağlayan bir donanım tabanlı güvenli işlem ortamıdır.

4. Rastgele bekleme süresi, tüm düğümler arasında eşit dağıtılır, bu da manipülasyonu zorlaştırır.

5. Blok oluşturulduktan sonra diğer düğümler, blok üretim sürecinin geçerliliğini doğrular. SGX, bu doğrulama işlemini güvenilir bir şekilde destekler.

16.6.2 Güvenlik

- **Intel SGX**: PoET, rastgele bekleme süresini Intel SGX gibi güvenilir donanım ortamlarında üretir. Bu sayede, düğümlerin süreci manipüle etmesi veya bekleme süresini kısaltması mümkün değildir.

- **Şeffaflık**:Rastgele bekleme süresi hesaplamaları, ağdaki diğer düğümler tarafından doğrulanabilir. Bu, blok oluşturma sürecinin güvenilirliğini artırır.

- **Koordinasyon**: PoET, diğer konsensüs algoritmalarında olduğu gibi düğümler arasında bir koordinasyon gerektirmez. Rastgele bekleme süresi mekanizması manipülasyon girişimlerini önler.
- **Adil Katılım**: PoET, bekleme sürelerini rastgele ve eşit şekilde dağıtarak blok oluşturma şansını tüm düğümlere eşit şekilde tanır. Bu, merkeziyetçiliği azaltır.
- **Güvenilir Donanım**: Manipülasyon girişiminde bulunan düğümler, güvenilir donanım ortamında tespit edilebilir ve ağdan dışlanabilir.

16.7 Byzantine Fault Tolerance (BFT)

Byzantine Fault Tolerance (BFT), bir dağıtık sistemde, ağdaki bazı düğümlerin kötü niyetli davranmasına veya hatalı çalışmasına rağmen sistemin doğru çalışmaya devam etmesini sağlayan bir konsensüs mekanizmasıdır. BFT, adını "Bizans Generalleri Problemi"nden alır ve dağıtık sistemlerde güvenilirlik sağlama sorununu çözmeye odaklanır. Elektrik kesintisi, ağ sorunları veya yazılım hataları nedeniyle bazı düğümler çalışamaz hale gelse bile sistemi işler durumda tutar. Ağdaki tüm doğrulayıcıların (veya düğümlerin) doğru bir şekilde aynı veriler üzerinde uzlaşmasını sağlar.

- **Dürüst Düğümler**: Sistemdeki çoğunluk dürüst olduğunda konsensüs sağlanabilir.
- **Kötü Niyetli Düğümler**: Kötü niyetli düğümler sahte bilgi yaysa bile dürüst düğümler doğru bir karar üzerinde uzlaşır.

$$\text{Fault Tolerance} = 3f + 1$$

Burada, f sistemde maksimum kötü niyetli düğüm sayısını ifade eder. Uzlaşma sağlanabilmesi için düğüm sayısı en az $3f + 1$ olmalıdır. Bu durumda, f sayıda kötü niyetli düğüm olsa bile sistem doğru karar verebilir.

16.7.1 Çalışma Adımları

1. Bir düğüm (lider) belirli bir işlem veya blok önerir.
2. Öneri, mesajlaşma ağdaki diğer düğümlere iletilir. Düğümler birbirleriyle bilgi paylaşır.
3. Her düğüm, aldığı bilgilere göre önerinin doğru olup olmadığına karar verir, oylama yapılır.
4. Dürüst düğümlerin büyük çoğunluğu (%66 veya daha fazla) aynı karara varırsa, öneri kabul edilir.

16.7.2 Güvenlik

- **Dürüst Çoğunluk İlkesi**: Sistemdeki düğümlerin büyük bir kısmı dürüst olduğu sürece konsensüs mekanizması manipülasyonu engeller.
- **İletişim Protokolleri**: BFT mekanizmaları, her düğümün diğer düğümlerle mesaj alışverişi yaparak bilgi doğrulamasını sağlar. Kötü niyetli düğümlerden gelen hatalı bilgiler bu süreçte filtrelenir.

- **Deterministik Karar Süreci**: Tüm düğümler aynı girdiye sahipse ve doğru protokolü izliyorsa, aynı çıktıyı üretir. Böylece, manipülasyon girişimleri etkisiz kalır.
- **Çoğunluk Kuralı**: Sistemin kararı, dürüst düğümlerin çoğunluk oyu ile belirlenir. Kötü niyetli düğümlerin oyları karar sürecinde dikkate alınmaz.
- **Lider Seçimi ve Değişimi**: Sistem, belirli bir süre lider düğümü kullanır. Eğer lider düğüm kötü niyetli veya işlevsiz hale gelirse, yeni bir lider seçilir.

17 Blockchain'de Veri Yapıları

17.1 Merkle Tree

Merkle Tree, bir tür ikili ağaç (binary tree) veri yapısıdır. Kriptografik hash fonksiyonlarını kullanarak verilerin bütünlüğünü ve doğruluğunu kontrol etmek için tasarlanmıştır. Her düğüm, alt düğümlerin hash değerini saklar. Ağaçtaki yaprak düğümler, ham verilerin hash değerlerini içerirken, diğer düğümler alt düğümlerin hash'lerinin birleşiminden oluşur. En üstteki düğüm, Merkle Root olarak adlandırılır ve tüm ağacın özetini temsil eder. Verilerin değişmediğini veya değiştirilmediğini doğrulamak için kullanılır. Blockchain sistemlerinde işlemlerin doğruluğunu kontrol eder. Merkle Tree sayesinde, kullanıcılar tüm blockchain'i indirmeden bir işlemi doğrulayabilir. Tüm veriyi doğrulamak yerine yalnızca ilgili hash değerleri kontrol edilerek işlem doğrulama yapılır. Blockchain'de, bir bloğun işlemlerinin bütünlüğü ve doğru zincirlenmesi Merkle Tree ile sağlanır. İşlemler arasında minimum bilgiyle maksimum doğrulama sağlar.

17.1.1 Çalışma Adımları

1. Ağaç yaprak düğümlerinden başlar. Her yaprak düğüm, bir işlem verisinin hash'ini saklar. Hash işlemi kriptografik hash fonksiyonlarıyla yapılır.

2. Alt düğümlerin hash'leri birleştirilir ve bir üst düğümde depolanır. Bu işlem, kök düğüme ulaşana kadar devam eder. Merkle Root, ağacın en üstündeki hash değeridir. Bu değer, tüm işlemlerin özetini temsil eder.

3. Belirli bir veri veya işlemin doğrulanması için yalnızca yol boyunca hash değerlerine ihtiyaç duyulur.

17.2 Trie

Trie, ön ek ağacı olarak adlandırılan bir veri yapısıdır. Verileri hiyerarşik bir şekilde depolamak için kullanılır. Trie, string işlemlerinde hızlı arama, ekleme ve silme işlemleri için tasarlanmıştır. Blockchain'de Trie, Ethereum gibi platformlarda state database'i ve işlemleri organize etmek için kullanılır. Ethereum'da, hesap bakiyelerini, kodlarını ve diğer verileri depolamak için Trie'nin geliştirilmiş bir hali olan Patricia Trie kullanılır. Trie, verilerdeki ortak ön ekleri paylaşarak bellek tüketimini azaltır. Bir string'in varlığını kontrol etmek veya bir ön ekle başlayan tüm stringleri bulmak çok hızlıdır. Bir Trie'de her düğüm bir harf içerir ve birden fazla çocuğu olaiblir. Yaprak düğümler, stringlerin sonunu temsil eder.

- **Ekleme İşlemi**: Bir kelime Trie'ye eklenirken, her harf hiyerarşik olarak bir düğüm olarak eklenir.
- **Arama İşlemi**: String'in harfleri sırayla takip edilerek Trie'de olup olmadığı kontrol edilir.
- **Silme İşlemi**: String'in sonunda ilgili düğüm kaldırılır ve eğer düğümün başka çocukları yoksa gereksiz düğümler silinir.

17.3 Patricia Tree

Patricia Tree, bir sıkıştırılmış Trie yapısıdır. Trie'nin optimize edilmiş bir versiyonudur ve benzer ortak ön eklere sahip olana anahtarları saklamak için kullanılır. Trie'deki her harf için ayrı düğüm oluşturulması yerine, benzer yollar birleştirilir. Patricia Trie'de anahtarlar, sıkıştırılmış yollar boyunca depolanır. Bu yapı hash değerleriyle birlikte birleştirilerek veri bütünlüğü sağlanır.

17.4 Merkle Patricia Tree

Merkle Patricia Trie (MPT), Ethereum gibi blokzincirlerinde kullanılan bir veri yapısıdır. Bu yapı, hem Merkle Tree'lerin güvenlik özelliklerini hem de Patricia Trie'lerin verimli depolama avantajlarını birleştirir. MPT, verilerin güvenli bir şekilde depolanmasını ve hızlı bir şekilde doğrulanmasını sağlar. Ethereum ve diğer blokzincir projeleri, her işlemdeki veriyi ve durumu (state) şifreli bir şekilde saklamak için Merkle Patricia Trie'yi kullanır. Bu, her işlemde yapılan değişikliklerin, ağın tüm katılımcıları tarafından doğrulanabilmesini sağlar. Patricia Trie, aynı ön eki paylaşan anahtarları daha verimli depolayarak veri belleği kullanımını optimize eder.

17.4.1 Çalışma Adımları

1. Merkle Patricia Trie, hem Trie'yi hem de Merkle Tree'yi içerir. Trie, her düğümün bir anahtar ve değer sakladığı bir yapıdır, ve Merkle Tree ise her düğümde bir hash değerine sahiptir.
2. Düğüm Tipleri;
   - **Branch Node**: Çocuk düğümlerin bağlantılarını ve bir hash değeri içerir.
   - **Extension Node**: Anahtarları uzatarak önceki düğüme bağlanır.
   - **Leaf Node**: Anahtar-değer çiftini içerir.
3. Her düğümün, altındaki verilerin hash'lerini içerdiği bir Merkle hash yapısı vardır. Bu sayede, herhangi bir veri değiştirilirse, sadece kök hash değeri değişir ve bu değişiklik tüm ağ tarafından kolayca doğrulanabilir.
4. MPT'ye yeni bir veri eklendiğinde, trie'nin kökünden başlayarak ilgili düğüme kadar hash hesaplamaları yapılır. Veriler silindiğinde, boş kalan düğümler temizlenir.

17.5 Directed Acyclic Graphs (DAG)

Directed Acyclic Graph (DAG), bir graf yapısında düğümler ve yönlendirilmiş bağlantılar (kenarlar) içeren, ancak döngü barındırmayan bir veri yapısıdır. Yönlendirilmiş bir graf olduğu için, her bir kenarın bir yönü vardır ve herhangi bir döngü bulunmadığından bir düğümden kendisine geri dönülemez. Bu yapı, blockchain alternatifleri ve dağıtılmış defter teknolojilerinde (DLT) yaygın olarak kullanılır. DAG, geleneksel blockchain yapılarında bulunan madencilik ve işlem sıralama süreçlerini ortadan kaldırarak daha hızlı ve ölçeklenebilir bir yapı sunar. Yüksek işlem hızlarına ve düşük işlem maliyetlerine izin verir. Blockchain'in aksine, tüm işlemler bir zincir yerine bir grafik üzerinde işlenir. Birden fazla işlem aynı anda işlenebilir ve bu durum sistemin hızını artırır.

17.5.1 Çalışma Adımları

1. Her bir düğüm, bir işlemi temsil eder.
2. Bir düğümden diğerine giden kenarlar, işlemlerin onay mekanizmasını temsil eder. Bir işlem, önceki işlemleri doğrulayarak ağı ilerletir.
3. Yapıdaki kenarlar, hiçbir zaman bir döngü oluşturmaz. Bir işlem sadece kendisinden önce gelen işlemleri referans alabilir.
4. Yeni bir işlem, iki veya daha fazla önceki işlemi onaylamak zorundadır. Bu onaylar, işlemin geçerli sayılması için gereklidir.

18 Layer-2

Layer-2, bir blockchain'in ana katmanı olan Layer-1 üzerine inşa edilen ikinci bir protokol veya ağ katmanıdır. Layer-2 çözümleri, temel blockchain'in işleyişini değiştirmeden ölçeklenebilirlik, işlem hızını artırma ve işlem maliyetlerini düşürme gibi sorunları çözmeyi hedefler. Layer-1 blockchain'ler düşük işlem kapasitesine sahiptir. Örneğin, Bitcoin yaklaşık 7 işlem/saniye (TPS), Ethereum ise 30 TPS civarında işlem yapabilir. Layer-2 çözümleri, bu kapasiteyi artırarak daha falza işlemi destekler. Layer-1 üzerinde her işlem için madencilik ücreti ödenir ve yoğun trafik sırasında bu ücretler çok artabilir. Layer-2, işlemleri daha ucuz hale getirir. Layer-1'in blok doğrulama süreleri, işlemleri yavaşlatabilir. Layer-2, bu işlemleri Layer-1'den bağımsız bir şekilde hızlandırır. Layer-2 çözümleri, Layer-1'in güvenlik protokollerini kullanmaya devam eder. Bu, Layer-2 ağlarının güvenli bir şekilde çalışmasını sağlar.

Layer-2 Çözümleri, işlemleri Layer-1'den alır ve bu işlemleri kendi sistemde işler. Daha sonra işlemlerin özetini veya sonuçlarını Layer-1'e kaydeder. Bu yöntem sayesinde Layer-1 üzerindeki yük azaltılır.

1. Kullanıcılar, Layer-1 üzerinde bir kanal veya bağlantı oluşturur ve işlemlerini Layer-2'ye yönlendirir.
2. Layer-2, işlemleri kendi ağı içinde gerçekleştirir. Bu, işlemlerin hızlı ve düşük maliyetle tamamlanmasını sağlar.
3. İşlemlerin toplam sonucu veya bir özeti, güvenlik sağlamak için Layer-1 blockchain'ine kaydedilir. Bu, Layer-2 üzerindeki işlemlerin Layer-1 ile aynı güvenlik standartlarına sahip olmasını sağlar.

18.1 State Channels

State Channels, iki veya daha fazla taraf arasında bir bağlantı (kanal) oluşturarak işlemlerin büyük çoğunluğunu off-chain (zincir dışı) gerçekleştirir. Bu yöntemle, işlem sonuçları yalnızca kanal kapatıldığında veya bir anlaşmazlık durumunda on-chain (zincir üzerinde) kaydedilir. İşlemler, blockchain ağının onaylama süreçlerinden bağımsız bir şekilde gerçekleştirilir. Her bir işlem için madencilik ücreti ödenmesi gerekmediği için maliyetler büyük ölçüde azalır. İşlemler zincir dışı gerçekleştiği için Layer-1 üzerindeki trafik azalır. Zincir dışı işlemler, blockchain üzerinde görünmez. Bu, taraflar arasında daha fazla gizlilik sağlar.

18.1.1 Çalışma Adımları

1. Kanal açılırken, bir çoklu imza hesabı oluşturulur. Bu hesap, yalnızca tarafların imzalarının birleştirilmesiyle işlem yapabilir.
2. İki taraf, blockchain üzerinde bir akıllı sözleşme (smart contract) aracılığıyla bir kanal oluşturur. Taraflar, kanalda kullanılacak bir

teminat (collateral) veya fonu akıllı sözleşmeyle kilitler. Bu aşamada kanalın başlangıç durumu blockchain'e kaydedilir.

3. Kanal açık olduğu sürece işlemler, taraflar arasında gerçekleşir. Her işlemde, taraflar yeni bir durum üzerinde anlaşır ve bu durum dijital olarak imzalanır. Bu, işlemlerin geçerliliğini ve üzerinde anlaşılan durumun değiştirilemeyeceğini garanti eder. İşlemler yalnızca taraflar arasında paylaşıldığı için blockchain'e kaydedilmez.
4. Kanal kapatılırken, taraflar son durum üzerinde anlaşır. Son durum blockchain'e kaydedilir ve kanal kapatılır.
5. Eğer bir anlaşmazlık olursa, akıllı sözleşme devreye girer ve önceki durumlardan hangisinin geçerli olduğunu belirler.

18.1.2 Bitcoin Lightning Network

Bitcoin işlemlerini hızlı ve ucuz hale getirmek için kullanılır. Kullanıcılar arasında sürekli ödeme yapılmasını sağlar.

18.1.3 Ethereum Raiden Network

Ethereum üzerinde token transferlerini hızlandırır ve maliyetleri düşürür.

18.2 Rollups

Rollups, çok sayıda işlemi zincir dışında birleştirir (roll-up) ve bu işlemlerin yalnızca özetini veya sıkıştırılmış halini ana zincire gönderir. Bu, hem Layer-1 güvenliğinden faydalanırken hem de Layer-2 üzerinde yüksek ölçeklenebilirlik sağlar. İşlemler Layer-2 üzerinde gerçekleştiği için Layer-1 üzerindeki gaz ücretleri büyük ölçüde azalır. Kullanıcılar çok daha düşük işlem ücretleri öder. Rollups, Layer-1'in güvenliğinden faydalanır. İşlemlerin geçerliliği ana zincir tarafından garanti edilir. Rollups, iki kategoriye ayrılır:

- **Optimistic Rollups**: İşlemlerin geçerli olduğu varsayılır (optimistic) ancak bir işlemde hata olduğunda veya kötü niyetli bir girişim olduğunda itiraz edebilir. İtiraz süresi 1-2 hafta sürebilmesi nedeniyle fon çekme işlemleri daha uzun sürebilir. Bu itiraz sürecinde, işlem geçerliliğini kanıtlamak için "Fraud Proof" adı verilen bir mekanizma devreye girer. Fraud Proof, sahte veya geçersiz bir işlemi kanıtlamak için kullanılır. Kullanıcılar, yanlış bir işlemi fark ettiklerinde sahtekarlık kanıtı sunabilirler.
- **Zero-Knowledge Rollups**: İşlemlerin geçerliliği, sıfır bilgi kanıtı adı verilen kriptografik bir teknikle kanıtlanır. Her bir batch'in geçerliliği, Layer-1 üzerinde bir "Validity Proof" ile doğrulanır. Bu

kanıtlar, işlemlerin doğru olduğunu Layer-1 üzerinde matematiksel olarak garanti eder. Daha hızlıdır çünkü itiraz sürecine gerek yoktur.

18.2.1 Çalışma Adımları

1. Kullanıcılar işlemlerini Layer-2 üzerindeki bir rollup operatörüne (veya doğrulayıcıya) gönderir. Operatör, bu işlemleri toplar ve tek bir işlem halinde "roll-up" eder.
2. Rollup, işlemlerin toplamını veya özeti (merkle kökü gibi) Layer-1 blockchain'ine gönderir. Bu özet, işlemlerin geçerliliğini kanıtlar ve ana zincir üzerinde doğrulanır.
3. Rollup operatörü, Layer-2 durumunu sürekli olarak günceller ve bu durum Layer-1 zincirine bağlı kalır.

18.3 Plasma

Plasma, Ethereum blockchain'i ölçeklendirmek için önerilmiş bir Layer-2 çözümüdür. Plasma, 2017 yılında Vitalik Buterin ve Joseph Poon tarafından tanıtılmış bir konsepttir. Tasarımı, Layer-1'in güvenliğini kullanırken Layer-2 üzerinde yüksek ölçeklenebilirlik ve işlem hızı sağlamayı hedefler. Plasma, Ethereum ana zincirine (Layer-1) bağlı alt zincirler oluşturur. Bu alt zincirler, işlemleri Layer-2 üzerinde işleyerek ağın işlem kapasitesini artırır ve maliyetleri düşürür. Her alt zincir kendi akıllı sözleşmeleri ve kuralları ile çalışır ancak güvenliği ana zincirden alır.

18.3.1 Çalışma Adımları

1. Ana zincir üzerinde bir Plasma akıllı sözleşmesi oluşturulur. Bu sözleşme, alt zincirlerin kurallarını ve durumlarını yönetir.
2. Kullanıcılar işlemlerini alt zincirlerde gerçekleştirir. Bu işlemler, alt zincirlerde birleştirilir ve sıkıştırılmış bir şekilde ana zincire gönderilir.
3. Alt zincirlerdeki işlemler, bleirli aralıklarla bir Merkle ağacı kökü olarak ana zincire kaydedilir (checkpoint).
4. Kullanıcılar, fonlarını alt zincirden ana zincire çekmek istediklerinde bir "exit" işlemi başlatır. Eğer alt zincirde sahte bir işlem olduğu düşünülüyorsa, kullanıcılar Fraud Proof mekanizması ile itiraz eder.

18.4 Sidechains

Sidechain (Yan Zincir), bir Layer-2 ölçeklendirme çözümü olarak kullanılan, ana blockchain'e bağlı ama ondan bağımsız bir blockchain ağını ifade eder. Sidechain'ler kendi konsensüs mekanizmalarına sahip ve ana zincirden (Layer-1) farklı kurallar çerçevesinde çalışan paralel ağlardır. Sidechain'ler, ana blockchain üzerinde işlem yapmadan, kullanıcıların daha hızlı ve düşük maliyetle işlem gerçekleştirmesini sağlar. Ana zincir ile sidechain arasındaki bağlantıyı bir iki yönlü köprü (two-way peg) kurar. Bu mekanizma, kullanıcıların token'larını ana zincirden sidechain'e ve geri taşımasına olanak tanır.

18.4.1 Çalışma Adımları

1. Sidechain, ana zincirden bağımsız olarak çalışır ancak bir "iki yönlü köprü" ile ana zincire bağlıdır. Köprü, kullanıcıların token'larını ana zincirden sidechain'e taşımasına olanak tanır.

2. Ana zincirde bir token sidechain'e gönderildiğinde, bu token ana zincirde kilitlenir ve sidechain'de eşdeğer bir token oluşturulur. Benzer şekilde, sidechain'deki token ana zincire geri gönderildiğinde, ana zincirdeki kilitli token serbest bırakılır.

3. Sidechain'ler, kendi bağımsız konsensüs mekanizmalarına sahiptir.

4. Sidechain'deki işlemler, sidechain'in validator veya madencileri tarafından doğrulanır ve bloklar oluşturulur. Bu işlemler ana zincire doğrudan kaydedilmez.

19 Smart Contracts (Akıllı Sözleşmeler)

Akıllı Contract (Smart Contract), Blockchain teknolojisinin temel özelliklerinden biri olan, kendi kendine yürütülebilen ve belirli koşullar sağlandığında otomatik olarak çalışan dijital sözleşmelerdir. Akıllı Contract, merkezi bir otoriteye ihtiyaç duymadan, şeffaf, güvenli ve değiştirilemez bir şekilde işlem yürütülmesini sağlar. Bu kontratlar, belirli kurallar ve koşullar tanımlanarak yazılır ve bu koşullar karşılandığında otomatik olarak işlemleri gerçekleştirirler. Akıllı kontratların kodları ve işlemleri Blockchain üzerinde saklanır, bu da onları şeffaf ve değiştirilmesi zor hale getirir. Akıllı kontrat, bir programlama diliyle (örneğin Solidity) yazılır ve "eğer-şart" türü kurallar belirlenir. Kontrat, dış girdilerle (örneğin bir ödemeyle) tetiklenir. Tanımlanan koşullar karşılandığında kontrat, programlandığı gibi işlemi gerçekleştirir

19.1 Ethereum Virtual Machine (EVM)

Ethereum Virtual Machine (EVM), Ethereum Blockchain'i üzerinde çalışan, programlanabilir bir ortam sağlayan ve Ethereum ağı üzerinde işlem gören akıllı kontratları çalıştırmak için kullanılan sanal bir makinedir. Akıllı kontratlar, Ethereum üzerinde merkeziyetsiz uygulamalar (DApp'ler) yaratmak için kullanılır ve bu kontratlar EVM'de çalıştırılır. EVM, Ethereum ağı üzerindeki tüm işlemleri işlemek ve doğrulamak için hesaplamalar yapar. Her işlem, EVM tarafından işlenir ve her düğüm bu işlemi kendi EVM'sinde tekrar doğrular.

EVM'de her işlem belirli bir gas tüketir. Gas, işlem veya akıllı kontratın çalıştırılması için gereken hesaplama gücünü ifade eder. Gas ücretleri, işlemi gerçekleştiren kişiye Ethereum üzerinden ödenir. Gas ücretleri, EVM'nin işlem maliyetlerini dengelemek için kullanılır.

19.2 Gas Mekanizması

Gas Mekanizması, Ethereum ve diğer blockchain platformlarında, bir işlemi gerçekleştirmek için gereken hesaplama gücünün ve kaynakların fiyatını belirleyen bir sistemdir. Gas, Ethereum ağında her işlem veya akıllı kontrat çalıştırması için ödenmesi gereken ücret birimidir. Gas, ağın dengesini sağlamak, aşırı yüklenmeleri engellemek ve işlemlerin adil bir şekilde gerçekleşmesini sağlamak için kullanılır. Gas ücretlerinin, işlem türüne ve karmaşıklığına göre değişmesi, Ethereum ağının verimli çalışmasına yardımcı olur. Gas, işlem yaparken kullanılan hesaplama gücü, veri depolama, ve işlem doğrulama gibi kaynakları ölçer. Ethereum kullanıcıları, işlemleri gerçekleştirebilmek için yeterli gas sağlamak zorundadırlar. Bu ücretlerin hesaplanmasında iki bileşen bulunur:

$$\text{Gas Ücreti} = \text{Gas Limit} \times \text{Gas Price}$$

- **Gas Limit**: Bir işlem için harcanabilecek maksimum gas miktarıdır. Gas limit, kullanıcının işlem başına belirlediği sınırdır. Bu limit, işlemin ne kadar karmaşık olduğunu ve işlem için ayrılacak gas miktarını belirler.
- **Gas Price**: Gas başına ödenecek ücretin miktarıdır. Gas fiyatı, kullanıcıların ödemek istediği fiyatı belirlediği bir terimdir. Gwei cinsinden ifade edilir. (1 gwei = 0.000000001 ETH) Gas fiyatı, işlem hızını ve ağ yoğunluğunu etkiler. Gas fiyatı ne kadar yüksek olursa, işlem o kadar hızlı bir şekilde işlenir çünkü madenciler veya doğrulayıcılar, daha yüksek gas ücretine sahip işlemleri önceliklendirir.

Akıllı kontratların karmaşıklığına göre gas kullanımı artabilir. Ethereum ağı, her işlem için gereken gas miktarını belirlemek için işlem tipi ve kontratın içerdiği hesaplamaları göz önünde bulundurur. Gas miktarı, Ethereum ağı tarafından otomatik olarak hesaplanır. Gas fiyatını etkileyen faktörler:

- **Ağ Yoğunlupu**: Ethereum ağı daha yoğun olduğunda, işlem onay süreleri uzar. Bu durumda kullanıcılar, işlem hızlarını artırmak için daha yüksek gas fiyatları teklif ederler.
- **Madencilerin Tercihleri**: Madenciler veya doğrulayıcılar, gas fiyatı yüksek olan işlemleri daha hızlı işleyecek şekilde seçerler. Bu nedenle, bir işlemdeki gas fiyatı ne kadar yüksek olursa, işlem o kadar hızlı onaylanır.
- **Akıllı Kontratın Karmaşıklığı**: Daha karmaşık bir akıllı kontrat çağrısı, daha fazla hesaplama gücü gerektirir ve bu da daha fazla gas kullanımına neden olur.
- **Gas Savaşları (Gas Wars)**: Gas savaşları, işlem onayı için yüksek gas fiyatı teklif eden birden fazla kullanıcının olduğu durumlarda ortaya çıkar. Kullanıcılar daha hızlı işlem yapmak için gas fiyatlarını artırır ve bu da ağda rekabeti artırır.

20 Blockchain Attacks

20.1 Sybil Attack

Sybil Attack, bir blockchain ağında ya da merkezi olmayan bir sistemde, bir saldırganın birden fazla sahte kimlik (node) oluşturmasıyla gerçekleştirdiği bir saldırı türüdür. Bu sahte düğümler, ağın işleyişini bozabilir, konsensüs mekanizmasını manipüle edebilir veya ağın kaynaklarını kötüye kullanabilir.

20.1.1 Çalışma Adımları

1. Saldırgan, sistemde sahte kimliklerle temsil edilen birden fazla düğüm oluşturur. Bu düğümler, saldırganın kontrolünde çalışır ve ağın diğer gerçek düğümleriyle aynı haklara sahiptir.
2. Saldırgan, sahte düğümler ile ağda önemli bir pozisyon elde etmeye çalışır.
3. Saldırgan, çeşitli manipülasyonlar yapar:
    - **Veri Manipülasyonu**: Yanlış veri yayabilir veya işlemleri engelleyebilir.
    - **Konsensüs Bozma**: Sahte düğümlerle çoğunluğu elde ederek yanlış kararlar alabilir.
    - **İzleme ve Analiz**: Gerçek düğümlerin aktivitelerini izleyerek ağdaki işlemleri takip edebilir.
    - **Denial of Service**: Ağda hizmet reddi saldırısı oluşturabilir.
    - **Double Spending**: Çifte harcama saldırısı yapabilir.

20.1.2 Engelleme Yöntemleri

- **Konsensüs Mekanizmaları**: PoW, PoS gibi konsensüs mekanizmaları kullanılabilir.
- **Kimlik Doğrulama**: Düğümlerin kimliklerini doğrulamak için merkezi olmayan doğrulama mekanizmaları kullanılabilir.
- **Merkeziyetsiz Bağlantı Yapısı**: Ağa katılan düğümlerin rastgele seçilmesi, sahte düğümlerin ağın kritik bölgelerine erişimini zorlaştırır.
- **Kaynak Sınırlandırma**: Sahte kimliklerin oluşturulmasını zorlaştırmak iççin bant genişliği, işlem gücü gibi bazı kaynaklara sınırlama getirilebilir.
- **Karmaşık Topoloji**: Ağın daha karmaşık bir bağlantı yapısına sahip olması, sahte düğümlerin kritik pozisyonları ele geçirmesini zorlaştırır.

20.2 Eclipse Attack

Eclipse Attack saldırısında, bir düğümün diğer düğümlerle olan bağlantıları manipüle edilir ve saldırganın kontrol ettiği sahte düğümlerle sınırlanır. Saldırgan, hedef düğümü izole ederek yalznıca kendi kontrolündeki bilgiye erişmesini sağlar. Böylece, hedef düğüm yalnızca saldırganın belirlerdiği verileri alır. Ağa katılan diğer düğümlerle bilgi alışverişi yapamaz.

20.2.1 Çalışma Adımları

1. Saldırgan, hedef düğümün IP adresini, ağ bağlantı yapılandırmasını ve düğümün iletişim kurduğu diğer düğümleri analiz eder. Hedef düğümün sınırlı sayıda eş (peer) bağlantısı olması saldırıyı kolaylaştırır.

2. Saldırgan, birden fazla sahte düğüm oluşturur ve bu düğümleri hedef düğümle iletişim kuracak şekilde yapılandırır.

3. Saldırgan, hedef düğümün ağdaki diğer gerçek düğümlerle olan bağlantılarını kesmeye çalışır. Hedef düğüm yalnızca saldırganın kontrol ettiği düğümlerle bağlantı kurar.

4. Saldırgan, hedef düğüme yanlış veya eksik bilgi gönderir. Örneğin; yanlış bloklar sunulur, gerçek işlemler gizlenir veya saldırganın istediği işlemler doğru olarak gösterilir.

5. Saldırgan, izole edilen düğümden gelen yanlış bilgileri kullanarak çift harcama veya konsensüs manipülasyonu gibi saldırılar gerçekleştirir.

20.2.2 Engelleme Yöntemleri

- **Bağlantı Limitlerini Artırma**: Hedef düğümün daha fazla eş (peer) düğümle bağlantı kurması sağlanır. Bu, sahte düğümlerin hedef düğümü izole etmesini zorlaştırır.

- **IP Adresi Filtreleme**: Aynı IP adresinden gelen çoklu bağlantılar sınırlandırılır. Saldırganın birden fazla sahte düğümü aynı IP üzerinden kontrol etmesi engellenir.

- **Rastgele Bağlantı Kurma**: Hedef düğümün, her oturumda rastgele düğümlerle bağlantı kurması sağlanır. Böylece saldırganın, hedef düğümü tamamen izole etmesi zorlaşır.

20.3 Eavesdropping Attack

Eavesdropping Attack, bir blockchain ağı üzerindeki iletişim trafiğini gizlice dinleyerek bilgi elde etme amacı taşıyan bir saldırıdır. Bu saldırıda, saldırgan ağdaki düğümler arasında geçen mesajları veya işlemleri ilzer, veri akışını kaydeder ve bu verileri kötü niyetli amaçlarla kullanabilir. Örneğin, işlem bilgilerini çalabilir, kullanıcı kimliklerini açığa çıkarabilir veya ağın kullanım modelini analiz edebilir. Bu saldırı doğrudan ağın güvenliğine zarar vermez ancak kullanıcı gizliliğini tehlikeye atar ve daha karmaşık saldırılar için veri sağlar.

20.3.1 Çalışma Adımları

1. Saldırgan, hedef blockchain ağına bağlanır veya düğümlerin iletşim trafiğini ağ seviyesinde izler.
2. Düğümler arasındaki işlem mesajlarını, blok bilgilerini veya diğer ağ içi iletişim verilerini yakalar.
3. Toplanan veriler analiz edilir.

20.3.2 Engelleme Yöntemleri

- **TLS (Transport Layer Security)**: Düğümler arası iletişimde TLS gibi güvenli protokoller kullanılarak trafiğin şifrelenmesi sağlanır.
- **Meta Veri Gizliliği**: Blockchain protokolleri, IP adresleri gibi meta verilerin toplanmasını sınırlamalıdır.

20.4 Denial of Service (DoS) Attack

DoS Saldırısı, bir blockchain ağını veya belirli bir düğümü hedef alarak bu sistemlerin işleyişini bozmayı amaçlayan bir saldırı türüdür. Saldırgan, hedef sisteme aşırı miktarda sahte istek göndererek kaynaklarını tüketir ve sistemi kullanıcılar için erişilemez hale getirir. Ağda yavaşlamalara, tıkanıklıklara veya hizmet kesintilerine sebep olabilir. Sahte işlemlerle madencilerin veya doğrulayıcıların gereksiz kaynak harcamasını sağlayabilir. Türleri:

- **Flooding Attack**: Ağa aşırı miktarda sahte işlem gönderilir. Ağ bant genişliği veya düğüm işlem kapasitesi tüketilir.
- **Consensus-Level DoS**: Madencileri veya doğrulayıcıları yanıltmak için gereksiz veya yanlış işlemler gönderilir. Konsensüs mekanizmasını yavaşlatır.
- **Application-Level DoS**: Belirli bir blockchain uygulamasını (örneğin akıllı kontratlar) hedef alan saldırılardır.
- **Distributed DoS**: Birden fazla saldırgan kullanılarak hedefe eş zamanlı saldırılar gerçekleştirilir.

20.4.1 Çalışma Adımları

1. Saldırgan, blockchain ağında en savunmasız düğümleri veya hizmetleri belirler. Düşük kapasiteye sahip düğümler veya doğrulayıcılar seçilir.
2. Saldırgan, hedefe sürekli sahte istekler gönderir. Büyük miktarda sahte işlem göndererek blok doğrulama sürecini yavaşlatır. Düğümün kaynaklarını tüketmek için veri akışı başlatır.
3. Düğümün CPU, RAM veya depolama gibi kaynakları tükenir. Diğer düğümlerle iletişimi kesilir ve hedef düğüm işlevsiz hale gelir.
4. Hedeflenen düğüm ağdan koparılır veya işlevi tamamen durdurulur. Ağ performansı genel olarak düşer, işlemler gecikir veya doğrulanamaz hale gelir.

20.4.2 Engelleme Yöntemleri

- **Kapasite Artırımı**: Düğümlerin işlem gücü, bellek ve ağ bant genişliği artırılarak saldırılara dayanıklı hale getirilir.
- **Rate Limiting**: Bir düğümün belirli bir süre içinde kabul edebileceği işlem veya istek sayısını sınırlandırır.
- **Kara Liste**: Kötü niyetli davranış sergileyen düğümler kara listeye alınır.

20.5 Border Gateway Protocol (BGP) Hijack Attack

BGP Hijack saldırısı, BGP protokolündeki zayıflıkları hedef alır. Blockchain ağında, verilerin farklı düğümler arasında taşınması için internet altyapısı kullanılır. BGP, internet üzerindeki ağlar arasında veri yönlendirme için kullanılan bir protokoldür. Bu protokol, bir ağın hangi IP adreslerine sahip olduğunu ve bu adreslere nasıl ulaşılacağını belirler. Ancak BGP, doğrulama süreçlerinde güvenliğe çok dayanmaz. Bu durum, saldırganların sahte yönlendirme bilgileri göndererek internet trafiğini yanlış yönlendirmesine olanak tanır. BGP saldırısında saldırgan, ağ trafiğini yanlış yönlendirerek veri akışını kesintiye uğratır, gecikmeler yaratır veya trafik üzerinde casusluk yapar.

20.5.1 Çalışma Adımları

1. Saldırgan, blockchain ağına hizmet sağlayan düğümleri belirler.
2. Saldırgan, kontrol ettiği bir yönlendiriciden sahte BGP güncellemeleri gönderir. Sahte güncellemeler, saldırganın IP adres aralığına veya yönlendirme yollarına daha kısa yollar gösterir.
3. Blockchain ağına gönderilmesi gereken veri trafiği, saldırganın belirlediği yanlış IP adreslerine yönlendirilir. Saldırgan, bu veriyi inceleyebilir, manipüle edebilir veya tamamen kesebilir.
4. Veriler, blockchain ağına ulaşmadan saldırgan tarafından değiştirilir veya tamamen düşürülür. Madencilik havuzları birbirleriyle senkronize olamaz veya kullanıcı işlemleri gecikir.

20.6 Alien Attack

Alien saldırısı, blockchain sistemine yabancı (alien) unsurların dahil edilmesiyle gerçekleşir. Amaç, ağı manipüle etmek, hatalara neden olmak, işlemleri geciktirmek veya güvenlik açıklarını istismar etmektir. Alien Attack, doğrudan bir güvenlik açığından faydalanmak yerine, mevcut sistemlerin dışındaki unsurları kullanarak ağın işleyişini bozmayı hedefler.

20.6.1 Çalışma Adımları

1. Saldırgan, blockchain ağına bağlı harici unsurları analiz eder.
2. Saldırgan, sahte düğümler, yanıltıcı veri sağlayıcılar veya kötü niyetli yazılımlar oluşturur.
3. Harici unsurlar, blockchain ağına entegrasyon süreçlerinden geçer. Örneğin, bir oracle saldırısında, blockchain ağına yanlış veri gönderilir.
4. Saldırgan, sisteme entegre ettiği unsurlar üzerinden ağın işleyişini manipüle eder. Akıllı sözleşmeler, yanlış veri doğrultusunda yanlış sonuçlar üretir.
5. Yanıltıcı işlemler, veri kaybı veya sistem hataları yaratılır.

20.6.2 Engelleme Yöntemleri

- **Doğrulama Mekanizmaları**: Blockchain ağında kullanılan oracle'ların verileri birden fazla kaynaktan doğrulaması sağlanmalıdır. Çapraz doğrulama ve çoğunluk onayı mekanizmaları devreye sokulmalıdır.
- **Zincirler Arası Güvenlik**: Blockchainler arasında veri aktarımı sırasında kriptografik imzalar ve zincirleme doğrulama yöntemleri kullanılmalıdır. Güvenli köprü (secure bridge) protokolleri uygulanmalıdır.

20.7 Timejacking Attack

Timejacking Attack, blockchain ağlarında zaman damgalarını (timestamps) manipüle ederek ağın işleyişini bozmayı hedefleyen bir saldırı türüdür. Blockchain sistemleri, işlemleri ve blokları zaman sırasına göre organize eder. Bu süreçte, düğümler (nodes) birbirlerinin zaman bilgilerine güvenerek çalışır. Saldırganlar, bu zaman bilgilerinin hatalı veya manipüle edilmiş olmasını sağlayarak ağı yanıltabilir.

20.7.1 Çalışma Adımları

1. Saldırgan, ağa bağlanan düğümlere yanlış zaman bilgisi içeren mesajlar gönderir. Bu mesajlar, ağdaki diğer düğümlerin zaman bilgisiyle çelişir ve ağı yanıltarak düğümlerin zaman algılarını değiştirir.
2. Hedef düğümler, sahte zaman bilgisine göre kendi sistem saatlerini günceller. Zaman damgaları manipüle edildiğinde, düğümler yanlış blok zincirlerini doğrulamaya başlar.
3. Yanıltılan düğümler, hatalı blokları kabul eder ve bunları diğer düğümlere yayar. Sonuç olarak, ağda çatallanma (fork), işlemlerin doğrulanmasında gecikme veya çift harcama (double spending) gibi sorunlar ortaya çıkar.

20.7.2 Engelleme Yöntemleri

- **Çeşitli Zaman Kaynakları**: Blockchain ağları, merkezi olmayan ve dağıtık zaman kaynakları kullanmalıdır.
- **Zaman Bilgini Doğrulama**: Düğümler, yalnızca tek bir zaman kaynağına güvenmek yerine birden fazla güvenilir kaynaktan zaman bilgisi almalıdır.

20.8 The Ethereum Black Valentine's Day Vulnerability

14 Şubat 2019'da Ethereum platformunda keşfedilen bir güvenlik açığıdır. Bu açık, Ethereum istemci yazılımının belirli bir sürümünde ortaya çıkan bir kod hatasından kaynaklanıyordu. Bu açık, Ethereum ağındaki düğümlerin birbirinden farklı durumlara düşmesine neden olan bir hata içeriyordu. Bu durum, düğümlerin aynı işlemi farklı şekillerde işlemesine ve blok zincirinin geçici veya kalıcı olarak çatallanmasına (fork) neden olabiliyordu.

1. Ethereum istemci yazılımındaki hata, bazı işlemlerin belirli koşullarda yanlış bir şekilde işlenmesine neden oldu. Bu durum, düğümlerin aynı işlemi farklı sonuçlarla yorumlamasına yol açtı.
2. Düğümler arasında bir fikir birliği sağlanamaması, ağda çatallanma riskini artırdı. Farklı düğümler, farklı zincirleri geçerli olarak görmeye başladı.
3. Bu çatallanma, saldırganların aynı varlıkları farklı zincirlerde birden fazla kez harcama olasılığını doğurdu.
4. Saldırganlar, hatalı düğümleri manipüle ederek Ethereum ağına zarar verebilir ve işlemleri geciktirebilir veya durdurabilirdi.

20.9 Long Range Attack

Long Range saldırısı, Proof of Stake (PoS) tabanlı blockchain sistemlerinde görülen bir saldırı türüdür. Saldırganın geçmişte sahip olduğu ancak artık kontrol etmediği bir stake (hisse) üzerinden blockchain ağını manipüle etmesiyle gerçekleşir. Amaç, geçmiş bir noktadan itibaren alternatif bir zincir oluşturarak, mevcut blockchain ağına üstünlük sağlamaktır. Bu saldırı, düşük maliyetli zincir yeniden yapılandırma fırsatı sunduğu için PoS sistemlerine özgü bir tehdittir.

20.9.1 Çalışma Adımları

1. PoS sistemlerinde, blok üretimi kullanıcıların stake miktarına göre belirlenir. Saldırgan, geçmişte stake sahibi olduğu dönemi hedefler.

2. Saldırgan, geçmişteki bir bloktan itibaren yeni bir blockchain zinciri üretmeye başlar. Bu yeni zincir, mevcut zincirden farklı bir yön izler ve saldırganın istediği şekilde düzenlenir.

3. Saldırganın zinciri, mevcut blockchain ağında dolaşımda olan zincirle rekabet eder. Long Range Attack, PoW'deki (Proof of Work) gibi yoğun hesaplama gücü gerektirmez. Saldırgan, düşük maliyetle geçmiş zinciri yeniden oluşturabilir. Daha uzun ve daha geçerli bir zincir oluşturmayı başarırsa, mevcut blockchain ağı bu yeni zinciri kabul edebilir.

4. Saldırgan, kendi oluşturduğu zinciri ağa tanıtarak mevcut zinciri geçersiz kılmaya çalışır. Saldırgan, zinciri manipüle ederek sahte işlemleri veya çift harcama (double-spending) işlemlerini gerçekleştirebilir.

20.9.2 Engelleme Yöntemleri

- **Finality Mekanizmaları**: Finality, bir işlemin veya blok zincirinin belirli bir noktadan sonra geri alınamayacağını garanti eder. PoS sistemlerinde kullanılan bu mekanizmalar, zincir geçmişine dayalı saldırıları önlemek için etkilidir.

- **Stake Taşıma Süresi**: Kullanıcıların stake taşıma işlemleri kısıtlanarak, stake'i geçmişe dayalı olarak kullanmaları önlenir.

20.10 Bribery Attack

Bribery saldırısı, blockchain ağını manipüle etmek için saldırganın madencilere mali teşviklerle rüşvet vererek, saldırganın istediği işlemleri onaylamalarını veya belirli bir zincir versiyonunu desteklemeleri sağlanır. PoW ve PoS mekanizmalarında görülür. Saldırgan rüşvet vererek; belirli işlemleri bloke edebilir veya başka işlemlerin zincire öncelikli olarak eklenmesini sağlayabilir, bölünme (fork) yaratarak alternatif bir zincirin benimsenmesini sağlayabilir.

20.10.1 Çalışma Adımları

1. Saldırgan, blockchain ağına katılan madencilere veya doğrulayıcılara, belirli bir zinciri veya işlemi desteklemeleri için rüşvet teklif eder.
2. Rüşveti kabul eden madenciler/doğrulayıcılar, saldırganın belirttiği işlemleri onaylar veya blok zincirini yeniden düzenler.
3. Rüşvet kripto para birimi olarak ödenir ve gizli bir şekilde yapılır. Akıllı sözleşmeler kullanılarak ödemeler otomatikleştirilebilir ve saldırganın kimliği gizlenir.
4. Saldırganın zinciri veya işlemleri ağda geçerli kabul edilir. Bu durum ağın güvenilirliğini ve işleyişini zedeler.

20.10.2 Engelleme Yöntemleri

- **Slashing**: Saldırganın hedeflediği kazancı ekonomik olarak dezavantajlı hale getiren modeller kullanılabilir. Örneğin, akıllı sözleşmelerle rüşvet alanların cezalandırılması.
- **Ödül Artırma**: Madencilerin veya doğrulayıcıların dürüst davranışlarını desteklemek için daha cazip teşvikler sunulabilir. Örneğin, blok ödüllerinin artırılması.

20.11 Race Attack

Race saldırısı, bir işlemin farklı versiyonlarının blockchain ağına eş zamanlı olarak gönderilmesiyle gerçekleştirilir. Amaç, ağın bir işlemi geçerli kılmasını sağlarken diğerini iptal etmektir. Hızlı işlem onayı gerektiren durumlarda karşılaşılır. PoW tabanlı blockchain ağlarında görülür. Race Attack, iki işlemin aynı kripto parayı harcadığı bir durum yaratarak çalışır. Saldırgan, mağdurun ödemenin alındığına inanmasını sağlarken, diğer işlemin geçerli kılınmasını hedefler.

20.11.1 Çalışma Adımları

1. Saldırgan, iki çelişkili işlem oluşturur. İlk işlem, mağdurun cüzdanına bir miktar kripto para göndermek için oluşturulur. İkinci işlem, saldırganın kendi cüzdanına aynı kripto parayı geri göndermek için hazırlanır.
2. Saldırgan, iki işlemi eşzamanlı olarak blockchain ağına gönderir. Blockchain ağı, hangi işlemi önce onaylayacağı konusunda bir yarış başlatır.
3. Eğer saldırganın kendi lehine olan işlem madenciler tarafından onaylanırsa, mağdura yapılan ödeme geçersiz hale gelir.
4. Mağdur, saldırganın mal veya hizmeti alıp ödemenin aslında başarısız olduğunu fark ettiği bir durumla karşı karşıya kalır.

20.11.2 Engelleme Yöntemleri

- **İşlem Onayı**: Blockchain ağında bir işlemin tam olarak geçerli sayılması için birkaç blok onayının alınması beklenir.
- **Anlık Ödemelerden Kaçınma**: Anlık ödeme sistemlerinde yeterli blok onayı beklenmeden işlem tamamlanırsa, saldırıya açık hale gelinir. Bu nedenle, hızlı onay gerektiren durumlarda ek güvenlik önlemleri alınmalıdır.