**Elliptic Curve Digital Signature Algorithm (ECDSA)**

**Information Security IA - 1**

**Hiyaa Shah 16010121177**

**Vatsal Soni 16010121197**

**Raghav Srivastava 16010121198**

1. **Introduction**

The Elliptic Curve Digital Signature Algorithm is a Digital Signature Algorithm (DSA) that uses elliptic curve cryptography keys. It is a very efficient equation that is based on cryptography with public keys. ECDSA is utilized in many security systems, is popular in encrypted messaging apps, and is the foundation of Bitcoin security (with Bitcoin “addresses” serving as public keys).

It is a cryptographic algorithm used to create digital signatures. It is based on the mathematics of elliptic curves over finite fields. ECDSA offers a combination of security and efficiency, making it particularly suitable for resource-constrained environments like smart cards and mobile devices.

Elliptic Curve Digital Signature Algorithms (ECDSA) have recently received significant attention, particularly from standards developers, as alternatives to existing standard cryptosystems such as integer factorization cryptosystems and discrete logarithm problem cryptosystems.

1. **Features/Characteristics**

**Features**

1. **Security**: ECDSA provides a high level of security based on the elliptic curve discrete logarithm problem (ECDLP), making it resistant to various cryptographic attacks.
2. **Efficiency**: It offers strong security with relatively small key sizes, making it suitable for resource-constrained environments like mobile devices and embedded systems.
3. **Short Signatures**: ECDSA generates shorter signatures compared to other algorithms, reducing bandwidth and storage requirements.
4. **Fast Computation**: Operations such as key generation, signing, and verification are computationally efficient, enabling rapid processing.
5. **Key Size Flexibility**: ECDSA supports a range of key sizes, allowing users to balance security and performance according to their needs.
6. **Algorithmic Flexibility**: It can be implemented using various elliptic curves, providing flexibility in security parameters and performance.
7. **Standardization**: ECDSA is a standardized algorithm, ensuring interoperability and compatibility across different systems and platforms.
8. **Resistance to Quantum Attacks**: While vulnerable to quantum attacks, it offers better resistance compared to some other cryptographic algorithms.

**Characteristics:**

1. Mathematical Foundation: ECDSA is based on elliptic curve mathematics, specifically the discrete logarithm problem over elliptic curves.
2. Public-Key Cryptography: It is an asymmetric cryptographic algorithm, utilizing a pair of keys for signing and verification.
3. Digital Signature Scheme: ECDSA allows for the creation and verification of digital signatures, providing authenticity, integrity, and non-repudiation of messages.
4. Probabilistic Signature Scheme: Signatures generated by ECDSA are probabilistic, meaning that the same message may produce different signatures each time it's signed with the same private key.
5. Randomness Requirement: ECDSA requires the use of random numbers during signature generation to ensure security against replay attacks and prevent key exposure.
6. Hash Function Dependency: ECDSA relies on cryptographic hash functions to process messages before signing and verification, ensuring that signatures are based on the message content rather than the entire message itself.
7. Parameter Selection: Users need to select appropriate elliptic curves and key sizes based on security requirements and implementation constraints.
8. Interoperability: ECDSA implementations need to adhere to standardized algorithms and parameters to ensure interoperability across different systems and platforms.
9. Standardization: RSA is standardized in various cryptographic protocols and widely supported in cryptographic libraries, ensuring interoperability and ease of integration into diverse applications.
10. Legacy and Adoption: With decades of proven security and widespread adoption, RSA remains a cornerstone in cryptographic systems, trusted for securing sensitive data and communications across various domains.
11. **Methodology**

* Key Generation:

1. Choose an Elliptic Curve: Select a specific elliptic curve E defined over a finite field Fp or F2^m. The curve equation is usually in the form y^2 = x^3 + ax + b.

2. Generate a Private Key: Choose a random integer d such that 1 < d < n, where n is the order of the base point G on the chosen curve.

3. Compute the Public Key: Compute the corresponding public key Q = d \* G, where G is a predefined base point on the chosen elliptic curve, and \* denote elliptic curve point multiplication.

* Signing:

1. Compute the Digest: Compute a cryptographic hash of the message m using a secure hash function (e.g., SHA-256). Let z be the resulting hash value.

2. Generate a Random Value: Choose a random integer k such that 1 < k < n, where n is the order of the base point G.

3. Compute the Signature: Calculate r and s as follows:

- Calculate the point P = k \* G = (x1, y1).

- Compute r = x1 / n. If r = 0, choose another k and start over.

- Compute s = [ (z + rd) / k ] / n. If s = 0, choose another k and start over.

* Verification:

1. Compute the Digest: Recompute the hash of the received message m to obtain z.

2. Verify the Signature: Compute u1 = zs^{-1} / n and u2 = rs^{-1} / n.

3. Compute the Verification Point: Compute the point X = u1 \* G + u2 \* Q = (x1, y1).

4. Check the Signature: If r = x1 / n, the signature is valid.

* Security Considerations:

1. ECDSA security relies on the difficulty of solving the elliptic curve discrete logarithm problem (ECDLP).
2. Proper key management is essential to ensure the security of ECDSA encryption and digital signatures.
3. Key lengths should be chosen carefully to resist cryptographic attacks, with longer keys providing higher levels of security.
4. Protection against side-channel attacks, such as timing attacks and power analysis, is necessary to safeguard ECDSA implementations.
5. **Results**

**Code:**

**from collections import namedtuple**

**from hashlib import sha256**

**from math import ceil, log**

**from random import randint**

**from typing import NamedTuple**

***# Bitcoin ECDSA curve***

**secp256k1\_data = *dict*(**

***p*=*0x*FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFC2F, *# Field characteristic***

***a*=*0x*0, *# Curve param a***

***b*=*0x*7, *# Curve param b***

***r*=*0x*FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6AF48A03BBFD25E8CD0364141, *# Order n of basepoint G. Cofactor is 1 so it's ommited.***

***Gx*=*0x*79BE667EF9DCBBAC55A06295CE870B07029BFCDB2DCE28D959F2815B16F81798, *# Base point x***

***Gy*=*0x*483ADA7726A3C4655DA4FBFC0E1108A8FD17B448A68554199C47D08FFB10D4B8, *# Base point y***

**)**

**secp256k1 = namedtuple("secp256k1", secp256k1\_data)(\*\*secp256k1\_data)**

**assert (secp256k1.Gy \*\* 2 - secp256k1.Gx \*\* 3 - 7) % secp256k1.p == 0**

***class* CurveFP(*NamedTuple*):**

**p: *int* *# Field characteristic***

**a: *int* *# Curve param a***

**b: *int* *# Curve param b***

***def* extended\_gcd(*aa*, *bb*):**

***# https://rosettacode.org/wiki/Modular\_inverse#Iteration\_and\_error-handling***

**lastremainder, remainder = abs(aa), abs(bb)**

**x, lastx, y, lasty = 0, 1, 1, 0**

**while remainder:**

**lastremainder, (quotient, remainder) = remainder, divmod(**

**lastremainder, remainder**

**)**

**x, lastx = lastx - quotient \* x, x**

**y, lasty = lasty - quotient \* y, y**

**return lastremainder, lastx \* (-1 if aa < 0 else 1), lasty \* (-1 if bb < 0 else 1)**

***def* modinv(*a*, *m*):**

***# https://rosettacode.org/wiki/Modular\_inverse#Iteration\_and\_error-handling***

**g, x, \_ = extended\_gcd(a, m)**

**if g != 1:**

**raise *ValueError***

**return x % m**

***class* PointEC(*NamedTuple*):**

**curve: CurveFP**

**x: *int***

**y: *int***

**@*classmethod***

***def* build(*cls*, *curve*, *x*, *y*):**

**x = x % curve.p**

**y = y % curve.p**

**rv = *cls*(curve, x, y)**

**if not rv.is\_identity():**

**assert rv.in\_curve()**

**return rv**

***def* get\_identity(*self*):**

**return PointEC.build(*self*.curve, 0, 0)**

***def* copy(*self*):**

**return PointEC.build(*self*.curve, *self*.x, *self*.y)**

***def* \_\_neg\_\_(*self*):**

**return PointEC.build(*self*.curve, *self*.x, -*self*.y)**

***def* \_\_sub\_\_(*self*, *Q*):**

**return *self* + (-Q)**

***def* \_\_equals\_\_(*self*, *Q*):**

***# TODO: Assert same curve or implement logic for that.***

**return *self*.x == Q.x and *self*.y == Q.y**

***def* is\_identity(*self*):**

**return *self*.x == 0 and *self*.y == 0**

***def* \_\_add\_\_(*self*, *Q*):**

***# TODO: Assert same curve or implement logic for that.***

**p = *self*.curve.p**

**if *self*.is\_identity():**

**return Q.copy()**

**if Q.is\_identity():**

**return *self*.copy()**

**if Q.x == *self*.x and (Q.y == (-*self*.y % p)):**

**return *self*.get\_identity()**

**if *self* != Q:**

**l = ((Q.y - *self*.y) \* modinv(Q.x - *self*.x, p)) % p**

**else:**

***# Point doubling.***

**l = ((3 \* *self*.x \*\* 2 + *self*.curve.a) \* modinv(2 \* *self*.y, p)) % p**

**l = *int*(l)**

**Rx = (l \*\* 2 - *self*.x - Q.x) % p**

**Ry = (l \* (*self*.x - Rx) - *self*.y) % p**

**rv = PointEC.build(*self*.curve, Rx, Ry)**

**return rv**

***def* in\_curve(*self*):**

**return ((*self*.y \*\* 2) % *self*.curve.p) == (**

**(*self*.x \*\* 3 + *self*.curve.a \* *self*.x + *self*.curve.b) % *self*.curve.p**

**)**

***def* \_\_mul\_\_(*self*, *s*):**

***# Naive method is exponential (due to invmod right?) so we use an alternative method:***

***# https://en.wikipedia.org/wiki/Elliptic\_curve\_point\_multiplication#Montgomery\_ladder***

**r0 = *self*.get\_identity()**

**r1 = *self*.copy()**

***# pdbsas***

**for i in range(ceil(log(s + 1, 2)) - 1, -1, -1):**

**if ((s & (1 << i)) >> i) == 0:**

**r1 = r0 + r1**

**r0 = r0 + r0**

**else:**

**r0 = r0 + r1**

**r1 = r1 + r1**

**return r0**

***def* \_\_rmul\_\_(*self*, *other*):**

**return *self*.\_\_mul\_\_(other)**

***class* ECCSetup(*NamedTuple*):**

**E: CurveFP**

**G: PointEC**

**r: *int***

**secp256k1\_curve = CurveFP(secp256k1.p, secp256k1.a, secp256k1.b)**

**secp256k1\_basepoint = PointEC(secp256k1\_curve, secp256k1.Gx, secp256k1.Gy)**

***class* ECDSAPrivKey(*NamedTuple*):**

**ecc\_setup: ECCSetup**

**secret: *int***

***def* get\_pubkey(*self*):**

***# Compute W = sG to get the pubkey***

**W = *self*.secret \* *self*.ecc\_setup.G**

**pub = ECDSAPubKey(*self*.ecc\_setup, W)**

**return pub**

***class* ECDSAPubKey(*NamedTuple*):**

**ecc\_setup: ECCSetup**

**W: PointEC**

***class* ECDSASignature(*NamedTuple*):**

**c: *int***

**d: *int***

***def* generate\_keypair(*ecc\_setup*, *s*=None):**

***# Select a random integer s in the interval [1, r - 1] for the secret.***

**if s is None:**

**s = randint(1, ecc\_setup.r - 1)**

**priv = ECDSAPrivKey(ecc\_setup, s)**

**pub = priv.get\_pubkey()**

**return priv, pub**

***def* get\_msg\_hash(*msg*):**

**return *int*.from\_bytes(sha256(msg).digest(), "big")**

***def* sign(*priv*, *msg*, *u*=None):**

**G = priv.ecc\_setup.G**

**r = priv.ecc\_setup.r**

***# 1. Compute message representative f = H(m), using a cryptographic hash function.***

***# Note that f can be greater than r but not longer (measuring bits).***

**msg\_hash = get\_msg\_hash(msg)**

**while True:**

***# 2. Select a random integer u in the interval [1, r - 1].***

**if u is None:**

**u = randint(1, r - 1)**

***# 3. Compute V = uG = (xV, yV) and c ≡ xV mod r (goto (2) if c = 0).***

**V = u \* G**

**c = V.x % r**

**if c == 0:**

**print(*f*"c={c}")**

**continue**

**d = (modinv(u, r) \* (msg\_hash + priv.secret \* c)) % r**

**if d == 0:**

**print(*f*"d={d}")**

**continue**

**break**

**signature = ECDSASignature(c, d)**

**return signature**

***def* verify\_signature(*pub*, *msg*, *signature*):**

**r = pub.ecc\_setup.r**

**G = pub.ecc\_setup.G**

**c = signature.c**

**d = signature.d**

***# Verify that c and d are integers in the interval [1, r - 1].***

***def* num\_ok(*n*):**

**return 1 < n < (r - 1)**

**if not num\_ok(c):**

**raise *ValueError*(*f*"Invalid signature value: c={c}")**

**if not num\_ok(d):**

**raise *ValueError*(*f*"Invalid signature value: d={d}")**

***# Compute f = H(m) and h ≡ d^-1 mod r.***

**msg\_hash = get\_msg\_hash(msg)**

**h = modinv(d, r)**

***# Compute h1 ≡ f·h mod r and h2 ≡ c·h mod r.***

**h1 = (msg\_hash \* h) % r**

**h2 = (c \* h) % r**

***# Compute h1G + h2W = (x1, y1) and c1 ≡ x1 mod r.***

***# Accept the signature if and only if c1 = c.***

**P = h1 \* G + h2 \* pub.W**

**c1 = P.x % r**

**rv = c1 == c**

**return rv**

***def* get\_ecc\_setup(*curve*=None, *basepoint*=None, *r*=None):**

**if curve is None:**

**curve = secp256k1\_curve**

**if basepoint is None:**

**basepoint = secp256k1\_basepoint**

**if r is None:**

**r = secp256k1.r**

***# 1. Select an elliptic curve E defined over ℤp.***

***# The number of points in E(ℤp) should be divisible by a large prime r.***

**E = CurveFP(curve.p, curve.a, curve.b)**

***# 2. Select a base point G ∈ E(ℤp) of order r (which means that rG = 𝒪).***

**G = PointEC(E, basepoint.x, basepoint.y)**

**assert (G \* r) == G.get\_identity()**

**ecc\_setup = ECCSetup(E, G, r)**

**return ecc\_setup**

***def* main():**

**ecc\_setup = get\_ecc\_setup()**

**print("\n")**

**print(*f*"E: y^2 = x^3 + {ecc\_setup.E.a}x + {ecc\_setup.E.b} (mod {ecc\_setup.E.p})")**

**print(*f*"base point G({ecc\_setup.G.x}, {ecc\_setup.G.y})")**

**print(*f*"order(G, E) = {ecc\_setup.r}")**

**print("Generating keys")**

**priv, pub = generate\_keypair(ecc\_setup)**

**print(*f*"private key s = {priv.secret}")**

**print(*f*"public key W = sG ({pub.W.x}, {pub.W.y})")**

**msg\_orig = *b*"hello world"**

**signature = sign(priv, msg\_orig)**

**print(*f*"signature ({msg\_orig}, priv) = (c,d) = {signature.c}, {signature.d}")**

**validation = verify\_signature(pub, msg\_orig, signature)**

**print(*f*"verify\_signature(pub, {msg\_orig}, signature) = {validation}")**

**msg\_bad = *b*"hello planet"**

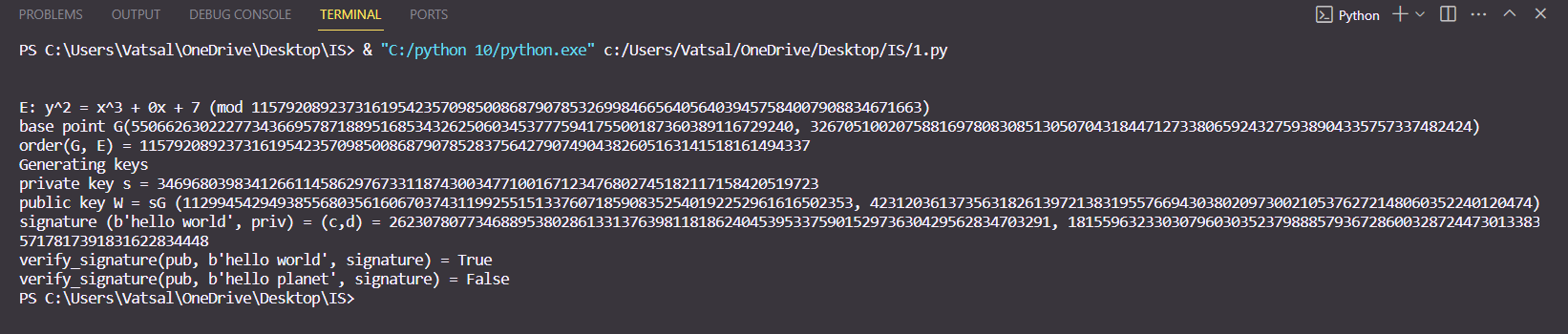
**validation = verify\_signature(pub, msg\_bad, signature)**

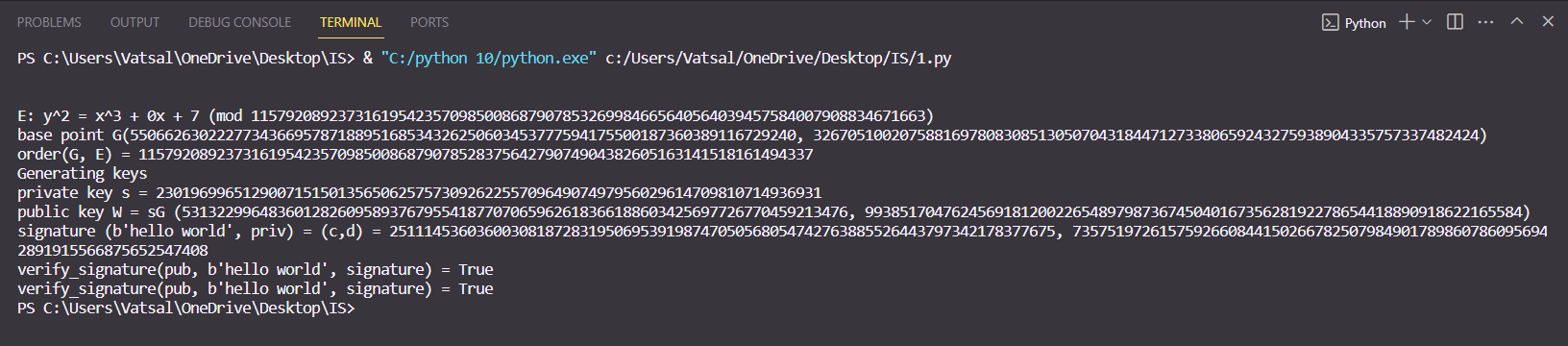
**print(*f*"verify\_signature(pub, {msg\_bad}, signature) = {validation}")**

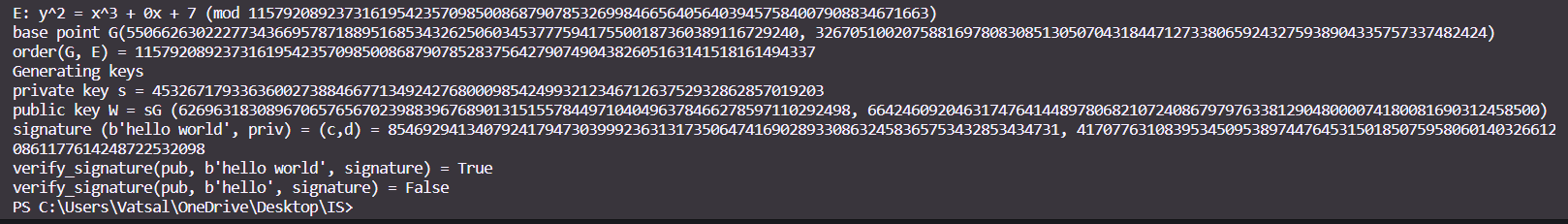
**if \_\_name\_\_ == "\_\_main\_\_":**

**main()**

**Output:**

****

****

****

**Similar ECDSA in java**

**Signature verified: true**

**import *java.security.\**;**

**import *java.security.spec.ECGenParameterSpec*;**

**import *java.util.Base64*;**

***public* *class* ECDSA1 {**

***public* *static* *void* main(*String*[] *args*) *throws* *Exception* {**

***// Generate ECDSA key pair***

***KeyPairGenerator* keyPairGenerator = KeyPairGenerator.getInstance("EC");**

***ECGenParameterSpec* ecSpec = new ECGenParameterSpec("secp256r1"); *// You can choose different curves***

**keyPairGenerator.initialize(ecSpec, new SecureRandom());**

***KeyPair* keyPair = keyPairGenerator.generateKeyPair();**

***// Print public key***

***PublicKey* publicKey = keyPair.getPublic();**

**System.out.println("Public Key: " + Base64.getEncoder().encodeToString(publicKey.getEncoded()));**

***// Print private key***

***PrivateKey* privateKey = keyPair.getPrivate();**

**System.out.println("Private Key: " + Base64.getEncoder().encodeToString(privateKey.getEncoded()));**

***// Signing***

***Signature* ecdsaSign = Signature.getInstance("SHA256withECDSA");**

**ecdsaSign.initSign(privateKey, new SecureRandom());**

***String* message = "Hiyaa";**

**ecdsaSign.update(message.getBytes());**

***byte*[] signature = ecdsaSign.sign();**

**System.out.println("Message: " + message);**

**System.out.println("ECDSA Signature: " + Base64.getEncoder().encodeToString(signature));**

***// Verification***

***Signature* ecdsaVerify = Signature.getInstance("SHA256withECDSA");**

**ecdsaVerify.initVerify(publicKey);**

**ecdsaVerify.update(message.getBytes());**

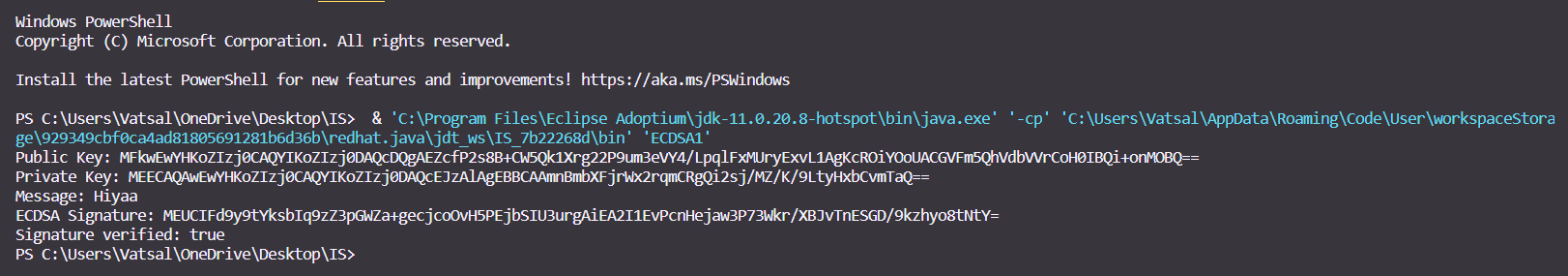
***boolean* verified = ecdsaVerify.verify(signature);**

**System.out.println("Signature verified: " + verified);**

**}**

**}**

**OUTPUT:**

****

**Signature verified: false**

**import *java.security.\**;**

**import *java.security.spec.ECGenParameterSpec*;**

**import *java.util.Base64*;**

***public* *class* ECDSA {**

***public* *static* *void* main(*String*[] *args*) *throws* *Exception* {**

***// Generate ECDSA key pair***

***KeyPairGenerator* keyPairGenerator = KeyPairGenerator.getInstance("EC");**

***ECGenParameterSpec* ecSpec = new ECGenParameterSpec("secp256r1"); *// You can choose different curves***

**keyPairGenerator.initialize(ecSpec, new SecureRandom());**

***KeyPair* keyPair = keyPairGenerator.generateKeyPair();**

***// Print public key***

***PublicKey* publicKey = keyPair.getPublic();**

**System.out.println("Public Key: " + Base64.getEncoder().encodeToString(publicKey.getEncoded()));**

***// Print private key***

***PrivateKey* privateKey = keyPair.getPrivate();**

**System.out.println("Private Key: " + Base64.getEncoder().encodeToString(privateKey.getEncoded()));**

***// Signing***

***Signature* ecdsaSign = Signature.getInstance("SHA256withECDSA");**

**ecdsaSign.initSign(privateKey, new SecureRandom());**

***String* message = "Hello, ECDSA!";**

**ecdsaSign.update(message.getBytes());**

***byte*[] signature = ecdsaSign.sign();**

**System.out.println("Message: " + message);**

**System.out.println("ECDSA Signature: " + Base64.getEncoder().encodeToString(signature));**

***// Tamper with the message or signature to simulate incorrect output***

**message = "Modified message"; *// Tampered message***

***// signature[0] = (byte) ~signature[0]; // Uncomment this line to tamper with the signature***

***// Verification***

***Signature* ecdsaVerify = Signature.getInstance("SHA256withECDSA");**

**ecdsaVerify.initVerify(publicKey);**

**ecdsaVerify.update(message.getBytes());**

***boolean* verified = ecdsaVerify.verify(signature);**

**System.out.println("Tampered Message: " + message);**

**System.out.println("Signature verified: " + verified);**

**}**

**}**

**OUTPUT:**

****

1. **Conclusion**

In summary, the Elliptic Curve Digital Signature Algorithm (ECDSA) stands as a robust cryptographic solution, leveraging the computational complexity of the elliptic curve discrete logarithm problem (ECDLP) to ensure secure digital signature generation and verification. Its efficiency, manifested through smaller signature sizes and rapid computational operations, makes it suitable for resource-constrained environments. ECDSA's algorithm flexibility, standardization, and proven reliability further cement its status as a widely adopted and trusted cryptographic algorithm, offering a balance of security, efficiency, and interoperability for diverse cryptographic applications.