DSA/ECDSA and biased-nonce attack

CS409: Introduction to Cryptography

Chalk and Talk Report

K Ashvanth

(22B1289)

Yashaswini K

(22B3911)

Under the guidance of

Prof. Sruthi Sekar

Indian Institute of Technology, Bombay

Contents

1 Introduction **4**
1.1 Digital Signature Algorithm (DSA) . . . 4
1.1.1 Key Features of DSA . . . 4
1.1.2 Steps in DSA . . . 4
1.1.3 Limitations of DSA . . . 5
1.2 Elliptic Curve Digital Signature Algorithm (ECDSA) . . . 5
1.2.1 Key Features of ECDSA . . . 5
1.2.2 Mathematics Behind Key Generation in ECDSA . . . 5
1.2.3 Steps in ECDSA . . . 7
1.2.4 Applications of ECDSA . . . 7
1.3 Comparison of DSA and ECDSA . . . 7

2 Attacks on ECDSA **8**
2.1 Pitfall #1 . . . 8
2.2 Pitfall #2 . . . 8
2.3 Pitfall #3 . . . 9

3 Solving the Hidden Number Problem (HNP) and Attacks on ECDSA Using Lattice-Based Techniques **11**
3.1 Lattice Basis Reduction . . . 11

4 The LLL Algorithm **12**
4.1 Steps and Description . . . 12
4.2 Key Properties of LLL: . . . 12
4.3 Applications in Cryptography: . . . 13
4.4 Python Implementation: . . . 13

5 Solving the Hidden Number Problem (HNP) **14**
5.1 Lattice Construction . . . 14
5.2 Short Vector Search Criterion . . . 14
5.3 Recovering α . . . 14

6 Lattice-Based Attacks on ECDSA **15**
6.1 ECDSA Signature Generation . . . 15
6.2 Lattice-Based Attack on ECDSA . . . 15
6.3 Comparison of LLL and CVP Approaches . . . 15
6.4 Conclusion . . . 15

7 LLL Attack on ECDSA **16**
7.1 Libraries Used 17
7.2 Ensuring Biased Nonces 18
7.3 Steps in the Attack 18
7.4 Verification 18

1 Introduction

Digital signatures play a critical role in modern cryptography, ensuring the authenticity, integrity, and non-repudiation of digital communications.
Two widely used algorithms for generating digital signatures are the **Digital Signature Algorithm (DSA)** and its elliptic curve variant, the **Elliptic Curve Digital Signature Algorithm (ECDSA)**. Both are fundamental to secure protocols, including SSL/TLS, blockchain technology, and secure email systems.

1.1 Digital Signature Algorithm (DSA)

The Digital Signature Algorithm (DSA) is a Federal Information Processing Standard (FIPS) for digital signatures. It was proposed by the National Institute of Standards and Technology (NIST) in 1991 as part of the Digital Signature Standard (DSS).

1.1.1 Key Features of DSA

- **Asymmetric Cryptography**: DSA relies on a pair of keys—a private key for signing and a public key for verification.
- **Mathematical Basis**: The algorithm is based on modular arithmetic and the discrete logarithm problem, which ensures its security.
- **Security Strength**: The security of DSA depends on the key size. Commonly used key sizes are 1024 bits, 2048 bits, and 3072 bits.

1.1.2 Steps in DSA

1. **Key Generation**:
   - Choose a large prime p and a prime divisor q such that $q \mid (p-1)$.
   - Select a generator $g = h^{(p-1)/q} \mod p$, where $1 < h < p-1$.
   - Choose a private key x, where $0 < x < q$.
   - Compute the public key $y = g^x \mod p$.
2. **Signing**:
   - Generate a random ephemeral key k, where $0 < k < q$.
   - Compute $r = (g^k \mod p) \mod q$. If $r = 0$, choose a new k.
   - Compute $s = k^{-1}(H(m) + x \cdot r) \mod q$, where $H(m)$ is the hash of the message and k^{-1} is the modular inverse of $k \mod q$.
   - If $s = 0$, choose a new k.
   - The signature is the pair (r, s).
3. **Verification**:
   - Verify that $0 < r < q$ and $0 < s < q$. If not, the signature is invalid.

- Compute $w = s^{-1} \mod q$.
- Compute $u_1 = H(m) \cdot w \mod q$ and $u_2 = r \cdot w \mod q$.
- Compute $v = ((g^{u_1} \cdot y^{u_2}) \mod p) \mod q$.
- Verify that $v = r$. If true, the signature is valid; otherwise, it is invalid.

1.1.3 Limitations of DSA

- DSA requires relatively large keys to achieve high levels of security.
- It is computationally intensive compared to its elliptic curve counterpart.

1.2 Elliptic Curve Digital Signature Algorithm (ECDSA)

ECDSA is a variant of DSA that uses elliptic curve cryptography (ECC) to achieve the same objectives as DSA but with enhanced efficiency and security. It was standardized in 2000 by the American National Standards Institute (ANSI).

1.2.1 Key Features of ECDSA

- **Elliptic Curve Cryptography**: ECDSA relies on the mathematics of elliptic curves over finite fields. The elliptic curve discrete logarithm problem (ECDLP) forms the basis of its security.
- **Compact Key Size**: ECDSA achieves equivalent security to DSA with significantly smaller key sizes. For example:
  - A 256-bit ECDSA key provides security equivalent to a 3072-bit RSA key.
- **Efficiency**: ECDSA requires less computational power and memory, making it suitable for resource-constrained environments, such as IoT devices and mobile platforms.

1.2.2 Mathematics Behind Key Generation in ECDSA

The key generation step in the **Elliptic Curve Digital Signature Algorithm (ECDSA)** involves elliptic curve operations over finite fields. Below are the mathematical details:

1. **Define the Elliptic Curve:** An elliptic curve E is defined by the equation:

   $$y^2 = x^3 + ax + b \pmod{p},$$

   where:

   - p is a large prime defining the finite field $\mathbb{F}_p$,
   - a and b are constants such that the discriminant $\Delta = 4a^3 + 27b^2 \neq 0$ (ensures the curve is nonsingular),
   - E consists of all points (x, y) satisfying the equation, along with a special point at infinity $\mathcal{O}$, which serves as the identity element.

2. **Choose a Base Point G:**

- A specific point $G = (x_G, y_G)$ on the curve is selected as the *generator point.*
- G satisfies the curve equation $y^2 = x^3 + ax + b \pmod p$.
- G has a large prime order n, meaning $nG = \mathcal{O}$, where $\mathcal{O}$ is the point at infinity.

3. **Choose the Private Key d:**
   - The private key d is a randomly selected integer:
   $$d \in [1, n-1].$$
   - d must be kept secret.

4. **Compute the Public Key Q:**
   - The public key Q is computed as:
   $$Q = d \cdot G,$$
   where $d \cdot G$ denotes scalar multiplication of the point G by the scalar d.

Elliptic Curve Operations

To compute $Q = d \cdot G$, elliptic curve arithmetic is used:

1. **Point Addition:** For two points $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$, the sum $P_3 = P_1 + P_2$ is calculated as:
$$\lambda = \frac{y_2 - y_1}{x_2 - x_1} \pmod p,$$
$$x_3 = \lambda^2 - x_1 - x_2 \pmod p, \quad y_3 = \lambda(x_1 - x_3) - y_1 \pmod p.$$
The result is $P_3 = (x_3, y_3)$.

2. **Point Doubling:** For a point $P = (x, y)$, the result of doubling the point $2P$ is given by:
$$\lambda = \frac{3x^2 + a}{2y} \pmod p,$$
$$x_3 = \lambda^2 - 2x \pmod p, \quad y_3 = \lambda(x - x_3) - y \pmod p.$$
The result is $2P = (x_3, y_3)$.

3. **Scalar Multiplication:** Scalar multiplication $d \cdot G$ is computed efficiently using the *double-and-add algorithm*, which combines repeated point doubling and point addition.

Security of ECDSA Key Generation

The security of ECDSA relies on the **Elliptic Curve Discrete Logarithm Problem (ECDLP)**:

- Given the base point G and public key $Q = d \cdot G$, it is computationally infeasible to determine the private key d.
- This provides strong security even with smaller key sizes compared to other cryptographic systems like RSA.

1.2.3 Steps in ECDSA

1. **Key Generation**: Choose a private key d and compute the public key $Q = d \cdot G$, where G is a point on the elliptic curve.
2. **Signing**:
   - Generate a random integer k (ephemeral key).
   - Compute $R = (x_1, y_1) = k \cdot G$ and use $r = x_1 \mod n$.
   - Compute the signature $s = k^{-1}(H(m) + r \cdot d) \mod n$, where $H(m)$ is the hash of the message.
3. **Verification**:
   - Compute $u_1 = H(m) \cdot s^{-1} \mod n$ and $u_2 = r \cdot s^{-1} \mod n$.
   - Verify that $R' = u_1 \cdot G + u_2 \cdot Q$ satisfies $r = x_1 \mod n$, where x_1 is the x-coordinate of R'.

1.2.4 Applications of ECDSA

- Blockchain and Cryptocurrencies (e.g., Bitcoin, Ethereum)
- Secure Web Communication (SSL/TLS)
- Digital Certificates
- Internet of Things (IoT) Devices

1.3 Comparison of DSA and ECDSA

Feature	**DSA**	**ECDSA**
Key Size	Larger keys required	Smaller keys for equivalent security
Efficiency	Less efficient	Highly efficient
Security Basis	Discrete Logarithm Problem	Elliptic Curve Discrete Logarithm Problem
Applications	General-purpose cryptography	Resource-constrained environments

DSA and ECDSA are cornerstone algorithms in cryptography, enabling secure digital signatures for a wide range of applications. While DSA is foundational, ECDSA offers significant advantages in terms of efficiency and compactness, making it the preferred choice in modern systems. The adoption of ECDSA in blockchain technology and IoT highlights its importance in securing the future of digital communication.

2 Attacks on ECDSA

The Elliptic Curve Digital Signature Algorithm (ECDSA) is widely regarded as a secure and efficient cryptographic algorithm, but its security depends on proper implementation and usage. At its core, ECDSA derives its strength from the **Elliptic Curve Discrete Logarithm Problem (ECDLP)**, which is computationally infeasible to solve given sufficiently large key sizes and well-chosen elliptic curve parameters. However, ECDSA's reliance on ephemeral keys (k) for generating signatures introduces a significant vulnerability if k is reused, predictable, or insufficiently random. A predictable k can lead to private key recovery, as demonstrated in real-world attacks such as Bitcoin nonce reuse incidents.

Recall, the signature in ECDSA is computed as:

$$s = k^{-1}(h + r \cdot d) \mod n,$$

where:

- s: Signature component.
- k: Ephemeral nonce used in signing.
- h: Hash of the signed message $H(m)$.
- r: Signature component derived from the elliptic curve point $R = k \cdot G$.
- n: Order of the elliptic curve group.

2.1 Pitfall #1

Nonce k must remain secret, or else the secret key d is revealed.
The security of ECDSA relies heavily on the ephemeral nonce k. If k is reused, predictable, or exposed, the private key d can be calculated as:

$$d = (s \cdot k - h)r^{-1} \mod n,$$

An obvious solution to avoid this pitfall is to ensure the secrecy of the nonce k.

2.2 Pitfall #2

If k is ever re-used to sign distinct messages h_1, h_2, it is revealed

$$k = (h_1 - h_2)(s_1 - s_2)^{-1} \mod n$$

and thus the long-term private key d is revealed.

Reusing the same nonce k for signing two different messages introduces a serious vulnerability. If two messages with hashes h_1 and h_2 are signed with the same k, their signatures (r, s_1) and (r, s_2) satisfy:

$$s_1 = k^{-1}(h_1 + r \cdot d) \mod n,$$

$$s_2 = k^{-1}(h_2 + r \cdot d) \mod n.$$

Subtracting these equations yields:

$$s_1 - s_2 = k^{-1}(h_1 - h_2) \mod n.$$

From this, k can be computed as:

$$k = (h_1 - h_2)(s_1 - s_2)^{-1} \mod n.$$

Once k is revealed, the private key d can be computed using the equation:

$$d = (s \cdot k - h)r^{-1} \mod n.$$

2.3 Pitfall #3

Biased Nonce Attack on ECDSA

In the Elliptic Curve Digital Signature Algorithm (ECDSA), the **nonce** k is critical for ensuring the security of the private key d. If k is not generated uniformly at random but is instead biased or reused, a **biased nonce attack** can recover d using lattice-based techniques or simple algebra.

ECDSA Signature Generation

1. **Key Pair:**
   - Private key: d
   - Public key: $Q = dG$, where G is the generator point of the elliptic curve.
2. **Signature Generation:**
   - Choose a random nonce $k \in [1, n-1]$, where n is the order of the elliptic curve.
   - Compute $r = (kG)_x \mod n$, the x-coordinate of the point kG.
   - Compute $s = k^{-1}(H(m) + dr) \mod n$, where $H(m)$ is the hash of the message m.

The signature is (r, s).

Attack Details

For m ECDSA signatures (r_i, s_i) corresponding to message hashes $H(m_i)$, the equations are:

$$k_1 - s_1^{-1}r_1d - s_1^{-1}H(m_1) \equiv 0 \pmod q,$$
$$k_2 - s_2^{-1}r_2d - s_2^{-1}H(m_2) \equiv 0 \pmod q,$$
$$k_3 - s_3^{-1}r_3d - s_3^{-1}H(m_3) \equiv 0 \pmod q,$$
$$\vdots$$
$$k_m - s_m^{-1}r_md - s_m^{-1}H(m_m) \equiv 0 \pmod q.$$

Matrix Representation

These equations can be expressed in matrix form for m signatures as:

$$\begin{bmatrix} 1 & -s_1^{-1}r_1 & -s_1^{-1} \\ 1 & -s_2^{-1}r_2 & -s_2^{-1} \\ 1 & -s_3^{-1}r_3 & -s_3^{-1} \\ \vdots & \vdots & \vdots \\ 1 & -s_m^{-1}r_m & -s_m^{-1} \end{bmatrix} \begin{bmatrix} k_1 \\ d \\ H(m_1) \end{bmatrix} \equiv \begin{bmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \pmod q.$$

Interpretation

If the nonces k_i are biased, such as being small or predictable, these equations can be used to construct a lattice and solve for the private key d using lattice-based techniques like the LLL algorithm.

Conditions for Attack

- The nonce k is small or drawn from a limited range, which biases the system.
- The system of equations is sufficiently overdetermined (i.e., there are more signatures than unknowns).
- Lattice techniques, leveraging the bias in k, recover k and d.

3 Solving the Hidden Number Problem (HNP) and Attacks on ECDSA Using Lattice-Based Techniques

Lattice-based techniques are powerful tools for solving the Hidden Number Problem (HNP) and attacking cryptographic schemes like ECDSA when biases or implementation flaws are present. Below, we detail the methods and their applications.

3.1 Lattice Basis Reduction

Lattice basis reduction simplifies a lattice basis to help identify short vectors corresponding to solutions of HNP or cryptographic vulnerabilities.

Lattice Definition. A lattice is a discrete subset of $\mathbb{R}^n$ consisting of all integer linear combinations of a set of linearly independent basis vectors:

$$\mathcal{L}(B) = \left\{ \sum_{i=1}^{n} c_i \mathbf{b}_i \mid c_i \in \mathbb{Z} \right\},$$

where $\mathbf{b}_i$ are the basis vectors of the matrix B.

Reduction Algorithms. Basis reduction transforms B into a "shorter" and "more orthogonal" basis. Key algorithms include:

- **LLL Algorithm:** Guarantees a good approximation by reducing the lengths of basis vectors.
- **BKZ Algorithm:** Enhances LLL with block-size optimizations for higher-quality reductions.

4 The LLL Algorithm

The Lenstra–Lenstra–Lovász (LLL) algorithm is a polynomial-time algorithm used for lattice basis reduction. It produces a basis that is approximately orthogonal and whose vectors are relatively short. This is crucial for solving problems like the Hidden Number Problem (HNP) or attacking cryptographic protocols.

4.1 Steps and Description

The LLL algorithm reduces a lattice basis $B = \{\mathbf{b}_1, \ldots, \mathbf{b}_n\}$ into a basis with shorter, nearly orthogonal vectors. The algorithm proceeds as follows:

1. Initialization: Compute the Gram-Schmidt orthogonalization (GSO) of the basis B:

$$\mathbf{b}_i^* = \mathbf{b}_i - \sum_{j=1}^{i-1} \mu_{i,j}\mathbf{b}_j^*, \quad \text{where } \mu_{i,j} = \frac{\mathbf{b}_i \cdot \mathbf{b}_j^*}{\|\mathbf{b}_j^*\|^2}.$$

2. Size Reduction: For $i > j$, reduce $\mathbf{b}_i$ using:

$$\mathbf{b}_i := \mathbf{b}_i - \lfloor \mu_{i,j} \rceil \cdot \mathbf{b}_j.$$

3. Check Lovász Condition: For $k = 2, \ldots, n$, verify:

$$\delta \cdot \|\mathbf{b}_{k-1}^*\|^2 \leq \|\mathbf{b}_k^*\|^2 + \mu_{k,k-1}^2 \cdot \|\mathbf{b}_{k-1}^*\|^2.$$

If violated, swap $\mathbf{b}_{k-1}$ and $\mathbf{b}_k$, recompute GSO, and repeat size reduction.

4. Termination: The algorithm terminates when all basis vectors satisfy both the size reduction condition and the Lovász condition.

4.2 Key Properties of LLL:

1. **Size-Reduced Basis:** Ensures the basis vectors satisfy the size-reduction condition:

   $$|\mu_{i,j}| \leq \frac{1}{2}, \quad \text{for } 1 \leq j < i \leq n,$$

   where $\mu_{i,j}$ are the Gram-Schmidt coefficients of the basis vectors.
2. **Lovász Condition:** Guarantees that the squared lengths of the basis vectors satisfy the inequality:

   $$\delta \cdot \|\mathbf{b}_i^*\|^2 \leq \|\mathbf{b}_{i+1}^*\|^2 + \mu_{i+1,i}^2 \cdot \|\mathbf{b}_i^*\|^2, \quad \text{for } i = 1, \ldots, n-1,$$

   where $0.25 \leq \delta < 1$ (commonly $\delta = 0.75$) and $\mathbf{b}_i^*$ are the Gram-Schmidt orthogonalized vectors.

4.3 Applications in Cryptography:

- Solving integer linear equations or approximations, like in the Hidden Number Problem (HNP).
- Breaking weak or biased implementations of cryptographic schemes (e.g., ECDSA or RSA with partial key exposure).

4.4 Python Implementation:

Below is an example implementation of the LLL algorithm using the `fpylll` library in Python:

```
from fpylll import IntegerMatrix, LLL

# Define the lattice basis as an integer matrix
basis = IntegerMatrix.from_matrix([
    [105, 821, 137],
    [492, 34, 621],
    [834, 345, 91]
])

# Apply LLL reduction
lll_reduced_basis = LLL.reduction(basis)

# Print the reduced basis
print("Reduced Basis:")
for vector in lll_reduced_basis:
    print(vector)
```

The code gives a reduced basis of the given matrix as an output.

5 Solving the Hidden Number Problem (HNP)

The Hidden Number Problem (HNP) can be solved using a lattice constructed from problem parameters, where the solution corresponds to a short vector in the lattice.

5.1 Lattice Construction

Given modular equations $t_i\alpha - a_i \equiv b_i \mod p$, the lattice basis is:

$$M = \begin{bmatrix} p & 0 & \cdots & 0 & 0 & 0 \\ 0 & p & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & p & 0 & 0 \\ t_1 & t_2 & \cdots & t_m & B/p & 0 \\ a_1 & a_2 & \cdots & a_m & 0 & B \end{bmatrix}.$$

A short vector corresponds to $\mathbf{v_b} = (b_1, b_2, \ldots, b_m, B\alpha/p, B)$, where the norm of $\mathbf{v_b}$ is small due to the bounded values of b_i and $B\alpha/p$.

5.2 Short Vector Search Criterion

From the provided modular equations and lattice properties, the following key points can be derived:

- **Dimension of the lattice:** $\dim \mathcal{L} = m + 2$.
- **Determinant of the lattice:** $\det \mathcal{L} = B^2 n^{m-1}$.
- Ignoring approximation factors, the LLL or BKZ algorithm will find a vector satisfying:

$$|\mathbf{v}| \leq (\det \mathcal{L})^{1/\dim \mathcal{L}}.$$

We aim to find a vector $\mathbf{v_k} = (k_1, k_2, \ldots, k_m, B\alpha/n, B)$ such that the length of $\mathbf{v_k}$ satisfies:

$$|\mathbf{v_k}| \leq \sqrt{m+2}B.$$

This search is successful when:

$$\log B \leq [\log n(m-1)/m - (\log m)/2].$$

5.3 Recovering α

The hidden number α is extracted from the component corresponding to $B\alpha/p$:

$$\alpha = \frac{\text{component corresponding to } B\alpha/p \times p}{B}.$$

6 Lattice-Based Attacks on ECDSA

ECDSA relies on the security of discrete logarithms over elliptic curves. However, implementation flaws, such as biased or reused nonces, expose vulnerabilities exploitable by lattice-based attacks.

6.1 ECDSA Signature Generation

ECDSA generates a signature (r, s) for a message m as follows:

1. Choose a random nonce k.
2. Compute $r = (kG)_x \mod q$.
3. Compute $s = k^{-1}(h(m) + dr) \mod q$, where d is the private key.

The private key d can be recovered if k is biased or reused across signatures.

6.2 Lattice-Based Attack on ECDSA

Given m signatures (r_i, s_i), the equations are reformulated as:

$$k_i - s_i^{-1} r_i d - s_i^{-1} h_i \equiv 0 \pmod q.$$

This system is embedded in a lattice with the basis:

$$M = \begin{bmatrix} q & 0 & \cdots & 0 & 0 \\ 0 & q & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ s_1^{-1} r_1 & s_2^{-1} r_2 & \cdots & s_m^{-1} r_m & 1 \end{bmatrix}.$$

6.3 Comparison of LLL and CVP Approaches

- **LLL Attack:** Identifies a short vector using lattice reduction. Effective for larger biases.
- **CVP Attack:** Finds the vector closest to the target. Suitable for smaller biases but computationally intensive.

6.4 Conclusion

Lattice-based techniques are effective for solving the Hidden Number Problem and exploiting ECDSA vulnerabilities when nonces are biased or reused. The success of these methods depends on the quality of lattice reduction, the size of biases, and the number of samples available. Proper nonce generation is crucial to maintain cryptographic security.

7 LLL Attack on ECDSA

Below is the Python code for implementing an LLL attack on ECDSA when biased nonces are used:

```python
from hashlib import sha256
from Crypto.Util.number import bytes_to_long, inverse
from ecdsa.ecdsa import curve_256, generator_256, Public_key, Private_key
from random import randint
from sage.all import matrix, QQ, log, sqrt

# Elliptic curve parameters and generator
G = generator_256
q = int(G.order())  # Order of the curve's generator
p = int(curve_256.p())  # Prime modulus of the field

# Generate a random private key d
d = randint(1, q - 1)
# Create public and private key pair
pubkey = Public_key(G, d * G)
privkey = Private_key(pubkey, d)
print(f"Generated private key d: {d}")

# List of messages to sign
m_list = [b'secret_message', b'really_secret', b'impossible_to_decrypt']
r_list, s_list, h_list, k_list = [], [], [], []

for msg in m_list:
    # Hash the message
    h = bytes_to_long(sha256(msg).digest())
    h_list.append(h)

    # Generate a biased nonce k (not completely secure)
    k = randint(1, 2**160)
    k_list.append(k)

    # Sign the message and append r, s values
    sig = privkey.sign(h, k)
    r_list.append(int(sig.r))
    s_list.append(int(sig.s))

print("Signed messages with biased nonces.")

# Function to recover the private key d from biased k values
def short_biased_k(r_list, s_list, h_list, q, p, B):
    # Compute intermediate values for lattice construction
    t_list = [inverse(s, q) * r % q for (s, r) in zip(s_list, r_list)]
    a_list = [inverse(s, q) * h % q for (s, h) in zip(s_list, h_list)]

    # Construct the lattice matrix M
    m = len(a_list)
    M = matrix(QQ, m + 2, m + 2)

    for ii in range(m):
        M[ii, ii] = q     # Set diagonal to q
        M[-2, ii] = t_list[ii] # Fill the second-last row with t values
        M[-1, ii] = a_list[ii] # Fill the last row with a values

    M[-2, -2] = QQ(B) / QQ(q)
    M[-1, -1] = QQ(B)

    # Apply LLL reduction to find the short vector
    M_lll = M.LLL(delta=0.75)

    # Identify the shortest vector with B as the last element
    v_short = None
    for v in M_lll:
        if v[-1] == B:
            v_short = v
            break

    if v_short is None:
        raise ValueError("Failed to find the expected short vector.")

    k0 = v_short[0] # Retrieve the guessed k value from the vector

    # Calculate d using the equation for the private key
    d = inverse(r_list[0], q) * (k0 * s_list[0] - h_list[0]) % q
    return d

# Define B as a bound close to the bit-length of q
B = randint(1, 2**174)
d_decrypted = short_biased_k(r_list, s_list, h_list, q, p, B)
print(f"Recovered private key d: {d_decrypted}")

# Check if recovered key matches the original private key
assert d == d_decrypted, "Failed to recover the correct private key!"
print("Successfully recovered the private key.")
```

Explanation of the Code

7.1 Libraries Used

- `hashlib`: Used to compute SHA-256 hash of the messages. This provides the message digest for signing.
- `Crypto.Util.number`: Provides utility functions like `bytes_to_long` (for converting byte data to long integers) and `inverse` (to compute modular inverses).
- `ecdsa.ecdsa`: Implements ECDSA over the SECP256k1 curve, providing utilities for key generation and message signing.
- `sage.all`: Provides mathematical tools such as lattice manipulations, rational field (`QQ`), and LLL reduction algorithm.

- `random.randint`: Generates random numbers. This is used to produce biased nonces.

7.2 Ensuring Biased Nonces

In ECDSA, the nonce k is required to be uniformly random. In this code, we deliberately generate nonces that are biased by restricting their range to $1 \leq k \leq 2^{160}$, which is much smaller than the full range $1 \leq k \leq q$ where q is the order of the curve's generator point G This bias reduces the entropy of the nonces, making them predictable enough for lattice-based attacks to succeed.

7.3 Steps in the Attack

1. **Key Pair Generation:** A random private key d is generated, and the corresponding public key is derived.

2. **Message Signing:** A set of messages are signed using biased nonces k. The r, s, and h (hash) values for each signature are stored.

3. **Lattice Construction:** Using the signature values, a lattice is constructed with:

$$t_i = s_i^{-1} r_i \mod q \quad \text{and} \quad a_i = s_i^{-1} h_i \mod q.$$

   These values form part of the lattice basis.

4. **LLL Reduction:** The LLL algorithm is applied to reduce the lattice basis, revealing short vectors.

5. **Key Recovery:** From the shortest vector, the guessed k_0 is used to compute the private key d using:

$$d = r_0^{-1}(k_0 s_0 - h_0) \mod q.$$

7.4 Verification

The recovered private key d is compared against the original key to verify the success of the attack. An assertion ensures that the attack is valid only if the keys match.