CAA 24-25: Lab #2

Nathan Rayburn

November 11, 2024

Abstract

This report analyzes potential vulnerabilities in ECDSA and demonstrates how lattice reduction techniques, specifically using the Lenstra–Lenstra–Lovász basis reduction algorithm, can compromise security when certain nonces are known. The lab covers deterministic-ECDSA weaknesses, focusing on nonce exposure and lattice attacks.

1 Introduction

In this lab, we examine common pitfalls in ECDSA implementations, analyzing both deterministic and random algorithms. The lattice-reduction attack applied here allows private key recovery if a portion of the nonce is known. An additional resource used: https://eprint.iacr.org/2019/023.pdf.

2 Lattice Attack on the Hidden Number Problem

In this section, we describe how one can solve the *hidden number problem* using lattice attacks. The hidden number problem is defined as follows:

Let $\alpha \in \mathbb{Z}_p$ be a secret. Let B be a known bound with B much smaller than p. Given pairs (t_i, a_i) such that $t_i\alpha - a_i \mod p = b_i$, with $b_i < B$, find the secret α.

To solve this problem, we employ lattice reduction, specifically using the LLL algorithm. The steps are as follows:

- Create the following matrix M over the rational numbers $\mathbb{Q}$:

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

  where the empty elements are zero. The dimensions of the matrix are $(m+2) \times (m+2)$.

- To construct this matrix in Sage, use the command `MatrixSpace(QQ, rows, cols)` and then obtain an identity matrix within this space with the method `identity_matrix()`. Assign this identity matrix to A. Use `A[i,j] = new_val` to modify the (i, j)-th element in the matrix.
- Run the LLL algorithm using the method `LLL()` on this matrix M to obtain an equivalent basis with shorter vectors.
- If successful, the vector $(b_1, b_2, \ldots, b_m, B\alpha/p, B)$ is a short vector in this matrix, allowing you to recover α.

3 Questions

3.1 1.1 Explanation of Vector Linear Combination

Discuss why the vector $(b_1, b_2, \ldots, b_m, B\alpha/p, B)$ is a linear combination of the matrix rows.

The primary objective of lattice reduction techniques, such as the LLL algorithm, is to find shorter vectors within a lattice that retains the same structure but with a reduced basis. In this case, the vector $(b_1, b_2, \ldots, b_m, B\alpha/p, B)$ is of particular interest because it represents a "short vector" within the lattice spanned by the rows of the matrix M.

Explanation

1. **Matrix Construction and Lattice Basis**:
   The matrix M is constructed in such a way that each row of M represents a basis vector in a lattice defined over the rational numbers $\mathbb{Q}$. Each of these basis vectors is designed to encode a relationship between the known values t_i, a_i, and the unknown value α. This setup is intended to capture the structure of the hidden number problem in a lattice form.

2. **Target Vector as a Linear Combination**:
   Since the target vector $(b_1, b_2, \ldots, b_m, B\alpha/p, B)$ is a short vector in the lattice spanned by the rows of M, it must be expressible as a linear combination of the matrix's row vectors. In other words, there exists a set of coefficients $(c_1, c_2, \ldots, c_{m+2})$ such that:

   $$(b_1, b_2, \ldots, b_m, B\alpha/p, B) = c_1 \cdot \text{row}_1 + c_2 \cdot \text{row}_2 + \ldots + c_{m+2} \cdot \text{row}_{m+2}$$

   where row_i denotes the i-th row of matrix M.

3. **Why This Vector is a "Short Vector"**:
   The LLL algorithm aims to find short. For a set of vectors, the goal is to find a shorter vector keeping the same Lattice by constructing a new set of basis using the orthogonal projection.

   For an example :

   $$\vec{b_1} = (0, 4)$$

$$\vec{b_2} = (1, 3)$$

Our Lattice would be the linear combinations of all integers in $\mathbb{Z}$ for our defined set of vectors of some basis B:

$$L = \sum_{i=1}^{m} \mathbb{Z} \cdot b_i = \left(\sum_{i=1}^{m} z_i \cdot b_i, \texttt{where } z_i \in \mathbb{Z}, b_i \in B, m \in \mathbb{N} \right)$$

For our set of our two vectors :

$$L = \sum_{i=1}^{2} z_i \cdot b_i$$

Our "Short vector" being orthogonal and keeping the same L would be :

$$\vec{b_1 *} = (-1, 1)$$

$$\vec{b_2 *} = (2, 2)$$

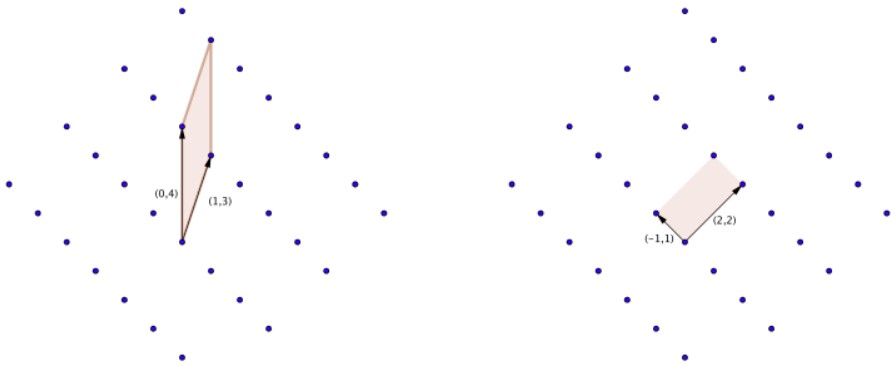

Figure 1: A lattice with two different basis. The right basis is reduced and orthogonal.

In a 2D reduced basis (b_1, b_2) is said to be reduced if it satisfies the following condition:

$$||b_1|| <= ||b_2||$$

u defined as our orthogonal projection coefficient.

$$u = \frac{b_1 \cdot b_2}{||b_1||} <= \frac{1}{2}$$

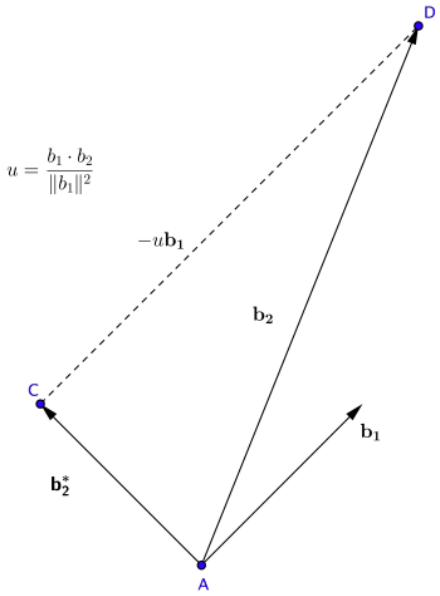

Figure 2: Orthogonal projection

In our contexte the vector $(b_1, b_2, \ldots, b_m, B\alpha/p, B)$ is chosen as it contains the small values $b_i < B$ along with terms proportional to $B\alpha/p$ and B. These values are small relative to the other possible combinations in the lattice.

4. **Recovering** α:
   If the LLL algorithm successfully identifies $(b_1, b_2, \ldots, b_m, B\alpha/p, B)$ as a short vector in the reduced basis, we can then isolate the term $B\alpha/p$ and recover the secret α. Specifically, since $B\alpha/p$ is known to be an integer, this value can be extracted from the short vector, enabling the recovery of α by rearranging terms. To recover the secret value α, we use the following formula:

$$\alpha = \mathbb{Z}\left(\frac{-v[-2] \cdot n}{B} \mod n\right)$$

   where:

   - α is the unknown value we are trying to recover.
   - $v[-2]$ represents the second-to-last element in the vector v, obtained from the LLL-reduced basis.
   - n is the modulus related to the hidden number problem.
   - B is a known bound that defines the problem constraints.
   - $\mathbb{Z}(\cdot)$ indicates the integer part of the expression within.

3.2 1.2 Vector Comparison to p

Explain why the vector $(b_1, b_2, \ldots, b_m, B\alpha/p, B)$ is relatively small compared to p. (In our case n)

In both challenges, our boundary B is chosen to be significantly smaller than p. The reason for this is that the lattice reduction algorithm, such as LLL, is more effective when applied to a lattice defined by shorter vectors with smaller magnitudes. By constructing the vector $(b_1, b_2, \ldots, b_m, B\alpha/p, B)$ with components scaled by B, we ensure that it is "short" relative to p. The b_i term represents a small error term. Specifically: $b_1, b_2, \ldots, b_m$ are the error terms in the hidden number equation, where each $b_i = t_i\alpha - a_i \mod p$ and $|b_i| < B$. These error terms are small values bounded by B, representing the discrepancies between $t_i\alpha$ and a_i in each instance of the modular equation.

3.3 1.3 Summarize what the LLL algorithm does and its application in this lab.

Thus, the vector $(b_1, b_2, \ldots, b_m, B\alpha/p, B)$ is a linear combination of the matrix rows because it lies within the lattice generated by these rows. By using lattice reduction to find this short vector, we exploit the properties of the lattice to uncover the hidden number α, achieving our goal in solving the hidden number problem.

4 Challenge 1: Nonce Bits Known

Abstract

The first challenge was slightly guided to understand the fundamentals how we can detect and apply an Lenstra–Lenstra–Lovász basis reduction algorithm. The first challenge we can observe that our element k which is used as a random nonce for ECDSA signatures, is generated within 352 bits. This is relatively smaller than p (384 bits) which can lead us to being able to apply the LLL algorithm since part of the nonce is know.

4.1 Conversion to Hidden Number Problem

In this section, we convert the problem of recovering the ECDSA private key with known nonce bits into a hidden number problem. The hidden number problem is defined as follows:

Let $\alpha \in \mathbb{Z}_p$ be a secret. Let B be a known bound with $B \ll p$. Given pairs (t_i, a_i) such that:

$$t_i\alpha - a_i \mod p = b_i, \quad \text{with } b_i < B,$$

find the secret α. (In our context $n = p$)

In the context of ECDSA, the goal is to recover the private key α using multiple signatures (r_i, s_i) and corresponding messages m_i. For each signature,

the nonce k_i satisfies:

$$s_i = \frac{h(m_i) + \alpha \cdot r_i}{k_i} \mod n,$$

where $h(m_i)$ is the hash of the message, and n is the order of the curve. Rearranging this equation to isolate k_i, we obtain:

$$k_i = \frac{h(m_i) + \alpha \cdot r_i}{s_i} \mod n.$$

We can rewrite k_i as:

$$k_i = \frac{h(m_i)}{s_i} + \frac{\alpha \cdot r_i}{s_i} \mod n.$$

Let:

$$t_i \cdot \alpha = \frac{\alpha \cdot r_i}{s_i} \mod n \quad \text{and} \quad -a_i = \frac{h(m_i)}{s_i} \mod n.$$

Isolating :

$$t_i = \frac{r_i}{s_i} \mod n \quad \text{and} \quad a_i = -\frac{h(m_i)}{s_i} \mod n.$$

Why $k_i = b_i$:

The equivalence $k_i = b_i$ arises because b_i represents the residual error term in the modular equation:

$$b_i = t_i \cdot \alpha - a_i \mod n.$$

This error term is directly related to k_i through the relationship defined by the signature equation. Since the lattice reduction algorithm focuses on finding small values of b_i, and b_i captures the modular behavior of k_i, we can use them interchangeably within the lattice framework.

To summarize:

- k_i is the nonce in the signature equation.
- b_i is the modular error term in the hidden number problem formulation.
- The equivalence $k_i = b_i$ holds because both describe the same mathematical quantity under modular arithmetic.

By solving for b_i using lattice reduction, we recover k_i, enabling the determination of α, the private key.

4.2 Key Recovery

Implement and describe the approach used to recover the private key from ECDSA signatures with specific nonce bit settings.

```
    m1 = messages1
    s1 = signatures1

    nbCols = len(s1) + 2
    nbRows = nbCols
    Aspace = MatrixSpace(QQ, nbRows, nbCols)

    A = copy(Aspace.identity_matrix())

    A = A*n
    B = n // pow(2,32)

    A[ - 2,  - 2] = B / n
    A[ - 1,  - 1] = B

    for i in range(len(m1)):

        ai = ((-h(m1[i]))/s1[i][1]) % n
        ti = (s1[i][0]/s1[i][1]) % n
        A[-1, i] = ai
        A[-2, i] = ti

    M = A.LLL()

    for v in M:
        if v[-1] == B:
            alpha = ZZ((- v[-2] * n / B)%n)

    if (alpha * G == A1):
        print("Cracked private key")
        print(alpha)
```

Listing 1: Python Code for Lattice Attack Setup

5 Challenge 2: Deterministic ECDSA - Weakness Analysis

We observe that k can be fully determined without any unknown variables, which implies that the intended security of the discrete logarithm problem is no longer leveraged for $k \cdot G$. Additionally, as shown below, the key (computed with hash(h)) and the nonce (fixed as
x00) are not random in the ChaCha20 context, allowing us to easily derive k.

```
def sign2(G, m, n, a):
    F = Integers(n)
    key = hashlib.sha256(m).digest()
    nonce = b"\x00"*24
    cipher = ChaCha20.new(key=key, nonce = nonce)
    #ciphertext = cipher.encrypt(plaintext)
    size_n = ceil(RR(log(n,2))/8) #taille en bytes
    k = int.from_bytes(cipher.encrypt(b"\x00"*size_n))
    (x1,y1) = (k*G).xy()
    r = F(x1)
```

```
    return (r, (F(h(m)) +a * r) /F(k))
```

Listing 2: Signature function Chall 2

Signature formula :

$$s_i = \left(\frac{h(m_i) + \alpha \cdot r_i}{k_i} \mod n \right)$$

Therefore isolating *alpha* :

$$alpha = \frac{s \cdot k - h(m)}{r}$$

Let's implement.

```
def challenge2(p,E,G,n):

    ...

    nonce = b"\x00"*24
    i = 0

    m = messages2[i]
    r = signatures2[i][0]
    s = signatures2[i][1]

    F = Integers(n)
    key = hashlib.sha256(m).digest()                    # can
    calculate

    cipher = ChaCha20.new(key=key, nonce = nonce)
    size_n = ceil(RR(log(n,2))/8)
    k = int.from_bytes(cipher.encrypt(b"\x00"*size_n))  # can
    calcualte

    a = F(s * k - h(m))/F(r)                            # Isolating
     the key

    if ( a * G == A2 ):
        print("Cracked private key")
        print(a)
```

Listing 3: Cracking the key for Chall 2

6 Challenge 3: Further Deterministic ECDSA Analysis

We can see that if all messages use the same key α, they will also share the same k and, consequently, the same r. By examining the signatures, we confirm that each message indeed has the same r, validating the hypothesis that only two messages are needed to recover the private key α.

```sage
def sign3(G, m, n, a):
    F = Integers(n)
    key = hashlib.sha256(str(a).encode()).digest()
    nonce = hashlib.sha256(str(a).encode()).digest()[:24]
    cipher = ChaCha20.new(key=key, nonce = nonce)
    #ciphertext = cipher.encrypt(plaintext)
    size_n = ceil(RR(log(n,2))/8) #taille en bytes
    k = int.from_bytes(cipher.encrypt(b"\x00"*size_n))
    (x1,y1) = (k*G).xy()
    r = F(x1)
    return (r, (F(h(m)) +a * r) /F(k))
```

Listing 4: Signature for Chall 3

We define an s and an s' :

$$s = \frac{h(m_1) + \alpha \cdot r}{k}$$

$$s' = \frac{h(m_2) + \alpha \cdot r}{k}$$

Isolate α :

$$\alpha = \frac{s \cdot k - h(m_1)}{r}$$

We remove α from our first equation and isolate k:

$$\frac{s \cdot k - h(m_1)}{r} = \frac{s' \cdot k - h(m_2)}{r}$$

$$s \cdot k - h(m_1) = s' \cdot k - h(m_2)$$

$$s \cdot k - s' \cdot k = h(m_1) - h(m_2)$$

$$k = \frac{h(m_1) - h(m_2)}{s - s'}$$

Let's implement.

```sage
def challenge3(p,E,G,n):

    ...

    m1 = messages3[0]
    m2 = messages3[1]

    F = Integers(n)

    r      = signatures3[0][0]
    s      = signatures3[0][1]

    sprime = signatures3[1][1]

```

```
    k = F((h(m1)-h(m2))/(s-sprime))

    alpha = F((s*k - h(m1))/r)
    if alpha * G == A3:
        print("Cracked private key")
        print(alpha)
```

Listing 5: Cracking private key Chall 3

7 Challenge 4: Final Deterministic ECDSA Case

In this signature we can see that our k is created predictably the same for each message using α concatenated with the message thrown into a hash function. Hash functions are predictable and that our boundary (256 bits) is significantly smaller than p, 384 bits (like said before in our case n). Therefore we also know part of the nonce and can apply LLL algorithm like the first challenge.

```
def sign4(G, m, n, a):
    F = Integers(n)
    k = int(hashlib.sha256(str(a).encode() + str(m).encode()).
    hexdigest(),16)
    (x1,y1) = (k*G).xy()
    r = F(x1)
    return (r, (F(h(m)) +a * r) /F(k))
```

Listing 6: Signature Chall 4

Let's implement.

```
    m4 = messages4
    s4 = signatures4
    nbCols = len(signatures4) + 2
    nbRows = nbCols
    Aspace = MatrixSpace(QQ, nbRows, nbCols)

    A = copy(Aspace.identity_matrix())

    A = A*n
    B = pow(2,256)

    A[ - 2,  - 2] = B / n
    A[ - 1,  - 1] = B

    for i in range(len(m4)):

        ai = ((-h(m4[i]))/s4[i][1]) % n
        ti = (s4[i][0]/s4[i][1]) % n
        A[-1, i] = ai
        A[-2, i] = ti

    M = A.LLL()

    for v in M:
        if v[-1] == B:
            alpha = ZZ((- v[-2] * n / B)%n)
```

```
27
28      if (alpha * G == A4):
29          print("Cracked private key")
30          print(alpha)
```

Listing 7: LLL Algorithm to crack private key chall 4