ICR Labo 2

Leo Pellandini

March 2025

1 Introduction

Ce travail pratique porte sur l'analyse de la sécurité de l'algorithme de signature ECDSA face à différentes vulnérabilités liées à la génération des nonces. Il est structuré en quatre challenges, chacun illustrant un type de faiblesse exploitable. L'objectif est de comprendre comment ces erreurs permettent, dans certains cas, de retrouver la clé privée à l'aide d'outils mathématiques (avec SageMath).

2 Questions

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

Par définition dans une *lattice*, toute combinaison linéaire entière des vecteurs d'une base appartient à celle-ci. Réciproquement, tout vecteur de la lattice peut s'écrire comme une telle combinaison linéaire des vecteurs de base.

Dans notre cas, les lignes de la matrice M forment une base de la lattice $\mathcal{L}$. Cela signifie que tout vecteur appartenant à ce réseau est nécessairement une combinaison linéaire entière des lignes de M.

Par conséquent, si un vecteur, tel que $(b_1, \ldots, b_m, -\frac{B\alpha}{p}, B)$ — appartient au réseau, cela signifie qu'il peut être obtenu par combinaison linéaire entière des lignes de M.

C'est cette propriété est exploitée par l'algorithme de réduction de LLL pour retrouver une solution candidate.

2.2 Explain why the vector $(b_1, b_2, \ldots, b_m, -B\alpha/p, B)$ is small compared to p.

Le vecteur $(b_1, b_2, \ldots, b_m, -\frac{B\alpha}{p}, B)$ est considéré comme de petite norme par rapport à p, car ses composantes sont toutes de taille au plus B, une valeur choisie de manière à être inférieure à p.

En effet :

- Les b_i correspondent à des erreurs sur les nonces, supposées petites.
- Le terme $-\frac{B\alpha}{p}$ reste lui aussi borné par B, puisque $\alpha < p$.
- Enfin, la dernière composante est simplement B.

Cela signifie que la norme de ce vecteur est globalement de l'ordre de $\sqrt{m+2} \cdot B$, ce qui est bien inférieur à celle des autres vecteurs de la base du réseau, qui eux contiennent typiquement des multiples de p ou n.

Cette différence de taille permet à l'algorithme de réduction de LLL de distinguer ce vecteur particulier en le plaçant parmi les premiers vecteurs de la base réduite, ce qui facilite l'extraction de la clé secrète.

2.3 Explain what the LLL algorithm does.

LLL est un algorithme de réduction de réseau. Il prend comme entrée une base de vecteurs d'un réseau (ou *lattice*) et renvoie une nouvelle base composée de vecteurs plus courts et plus proches les uns des autres.

Dans le contexte du problème du Hidden Number, la base initiale du réseau est construite de manière à contenir un vecteur particulier — celui qui encode la clé privée et les erreurs sur les nonces. Ce vecteur a la propriété d'être très court par rapport aux autres vecteurs de la base.

L'objectif de LLL est de transformer la base de départ pour que les vecteurs courts — comme celui contenant la clé — apparaissent dans les premières lignes de la base réduite. Cela permet ensuite de les tester directement et, dans le cas d'une attaque réussie, de retrouver la clé secrète.

3 Résolution

3.1 Challenge 1

Dans ce challenge, l'implémentation est vulnérable car dans la fonction `sign1` on génère les signatures de la manière suivante :

```
def sign1(G, m, n, a):
    F = Integers(n)
    n2 = n // 2^32
    k = F(ZZ.random_element(n2))  # nonce trop petit
    (x1, y1) = (k*G).xy()
    r = F(x1)
    return (r, (F(h(m)) + a * r) / F(k))
```

Le problème principal de cette fonction réside dans la génération du nonce k. Au lieu d'être choisi aléatoirement dans l'ensemble $\{1, \ldots, n-1\}$, il est généré dans un intervalle restreint de taille $n/2^{32}$. Cela signifie que k possède environ 32 bits de moins que la taille de n, le rendant significativement plus petit et donc vulnérable à une attaque par LLL.

Cette réduction de l'espace de recherche rend possible une attaque par réduction de réseau (LLL), car les valeurs de k sont suffisamment petites pour qu'on puisse approximer les relations de signature par des équations linéaires et appliquer une attaque sur le *Hidden Number Problem*.

3.1.1 Conversion en Hidden Number Problem

Lorsque les τ bits de poids fort du nonce k sont à zéro, le nonce devient *petit* par rapport à l'ordre n du groupe. Cela permet de transformer le problème de récupération de la clé privée en une instance du **Hidden Number Problem**.

Relation ECDSA :

La signature (r, s) satisfait :

$$s = \frac{h(m) + a \cdot r}{k} \mod n \Rightarrow s \cdot k = h(m) + a \cdot r \mod n$$

On réécrit :

$$a \cdot r - s \cdot k \equiv -h(m) \mod n$$

Si k est petit (par exemple $k < 2^{n-\tau}$), alors $s \cdot k$ est aussi petit. On peut approximer :

$$a \approx \frac{s \cdot k + h(m)}{r} \mod n$$

Ce type de relation linéaire entre a et des valeurs bruitées est typique du **Hidden Number Problem**. On peut alors modéliser le problème sous forme de système linéaire et le résoudre via **LLL**.

Construction de la matrice et réduction LLL :

On construit une matrice M de taille $(m+2) \times (m+2)$ où m est le nombre de signatures. Les m premières lignes forment une matrice identité multipliée par n, puis on ajoute les lignes contenant les t_i et a_i calculés précédemment.

```
space = MatrixSpace(QQ, m+2, m+2)
matrice = space.identity_matrix() * n

for i, (msg, (r, s)) in enumerate(zip(messages, signatures)):
    s_inv = inverse_mod(s, n)
    t_i = r * s_inv
    a_i = -h(msg) * s_inv
    matrice[m, i] = t_i
    matrice[m+1, i] = a_i

matrice[m, m] = B / n
matrice[m+1, m+1] = B

matrice_reduite = matrice.LLL()
```

Cette matrice encode les relations linéaires entre les différents termes de l'HNP. On la réduit ensuite avec LLL.

La réduction LLL permet de retrouver dans les plus courts vecteurs de la base une approximation du vecteur contenant la clé privée.

3.1.2 Retrouver la clé privée

Après avoir appliqué l'algorithme LLL sur la matrice construite, on obtient une base réduite contenant des vecteurs courts. On suppose que l'un de ces vecteurs correspond (ou est proche) du vecteur

$$V_b = (b_1, b_2, \ldots, b_m, -B\alpha/n, B)$$

où les b_i sont les erreurs de bruit introduites dans les approximations du problème HNP, B est une borne supérieure du nonce k, et α est une estimation de la clé privée multipliée par n/B.

Lorsqu'on construit la matrice initiale, on encode les relations :

$$a \cdot r_i - s_i \cdot k \equiv -h(m_i) \mod n$$

puis on introduit deux lignes supplémentaires dans la matrice :

- Une ligne contenant les $t_i = r_i/s_i$ et un coefficient $\frac{B}{n}$ à la colonne m
- Une ligne contenant les $a_i = -h(m_i)/s_i$ et un coefficient B à la colonne $m+1$

Cela donne à la fin une matrice dont une des lignes réduites par LLL approxime le vecteur :

$$(x_1, x_2, \ldots, x_m, x_{m+1}, x_{m+2}) \approx (b_1, \ldots, b_m, -B\alpha/n, B)$$

Dans le code, on récupère la valeur x_{m+1} (avant-dernière composante de chaque vecteur) qui correspond à une approximation de $-B \cdot a/n$. On en déduit un candidat pour la clé privée a via l'inversion de cette relation :

```
for ligne in matrice_reduite.rows():
    valeur_x = ligne[-2]
    try:
        candidat_a = Integer(-valeur_x * n * inverse_mod(B, n))
        cles_candidates.append(candidat_a)
    except ZeroDivisionError:
        continue
```

Chaque candidat est ensuite vérifié à l'aide de la fonction `correspondance_cle`, qui teste si le point $a \cdot G$ correspond à la clé publique A. La première valeur correcte permet de retrouver la clé privée exacte.

```
def correspondance_cle(k):
    point = k * G
    return point[0] == A1[0] and point[1] == A1[1]
```

Le premier candidat validant cette condition est considéré comme la bonne clé privée a, retrouvée directement à partir de la composante $-B \cdot a/n$ présente dans la base réduite par LLL. Aucune étape intermédiaire de calcul du nonce k n'est nécessaire ici.

3.1.3 Résultats obtenus

Une fois l'attaque effectuée à l'aide de la réduction LLL, le script a extrait 22 candidats potentiels pour le nonce k. Chacun a été testé via la fonction `correspondance_cle` afin de vérifier s'il permettait de retrouver la clé privée correcte.

Voici la clé privée trouvée avec le script :

```
237833619022848696639005404355272004209706566700873127236229246158457058933444093254637035700909651031571163274736 02
```

3.2 Challenge 2

Dans ce second challenge, la fonction de signature `sign2` implémente ECDSA avec un nonce déterministe, dérivé d'un chiffrement symétrique ChaCha20, comme ci-dessous :

```
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)
    size_n = ceil(RR(log(n,2))/8)
    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))
```

Ici, le nonce k est généré de manière **déterministe** à partir du message m seul, via le flux de chiffrement de l'algorithme `ChaCha20`.

Conséquences :

- Deux signatures sur des messages identiques produiront toujours le même nonce k.
- Un attaquant ayant accès à plusieurs signatures peut exploiter cette déterminisme pour retrouver la clé privée a.
- Contrairement au Challenge 1, ici k n'est pas trop petit, mais **complètement prévisible** si l'on connaît le mécanisme de génération.

Dans un scénario réel, la connaissance du nonce k suffit à remonter directement à la clé privée à l'aide de la formule de signature inversée :

$$a = \frac{s \cdot k - h(m)}{r} \mod n$$

Le challenge consiste donc ici à **reproduire la génération du nonce ChaCha20** et à exploiter sa nature déterministe pour retrouver la clé privée a.

3.2.1 Résolution

Pour résoudre ce challenge, le script va suivre ces étapes :

Étapes du script :

1. Pour chaque message m_i :
   - Le script régénère le même **nonce** k_i en utilisant `ChaCha20` avec :
     - Clé : `sha256(m)`
     - Nonce : une chaîne de 24 zéros (`b"\x00"*24`)
2. Il applique ensuite la formule :
   $$a_i = \frac{s_i \cdot k_i - h(m_i)}{r_i} \mod n$$
   pour reconstruire un candidat à la clé privée.
3. Tous les candidats a_i sont stockés dans une liste, puis comparés entre eux.

Vérification finale :

Si tous les candidats sont identiques, alors la valeur obtenue est probablement la clé privée. Le script vérifie ensuite si la clé publique calculée $a \cdot G$ correspond à la clé publique fournie A :

```
A_calcule = (a_recuperee * G).xy()
A == A_calcule  # vérifie la correspondance
```

3.2.2 Résultats obtenus

Voici la clé privée trouvée avec le script pour le challenge 2 :

```
820682598933094467007983093230781755859680299797213088595406947066147898645298712319314649272905128284180061720302 1
```

3.3 Challenge 3

Dans ce troisième challenge, la signature ECDSA est générée de manière déterministe, mais cette fois le nonce k dépend directement de la **clé privée** a. Voici l'implémentation :

```
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)
    size_n = ceil(RR(log(n,2))/8)
    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))
```

Dans cette version, le nonce k est généré de manière déterministe à partir de la **clé privée elle-même**. Concrètement :

Conséquence directe :

Toutes les signatures générées auront le **même** r, car celui-ci dépend uniquement de k via la courbe elliptique :

$$r = (k \cdot G)_x \mod n$$

Cela implique que les signatures (r, s_1) et (r, s_2) sur deux messages différents m_1 et m_2 vérifient :

$$s_1 = \frac{h(m_1) + a \cdot r}{k} \quad \text{et} \quad s_2 = \frac{h(m_2) + a \cdot r}{k}$$

Donc :

$$s_1 - s_2 = \frac{h(m_1) - h(m_2)}{k} \Rightarrow k = \frac{h(m_1) - h(m_2)}{s_1 - s_2}$$

Une fois k calculé, on le remplace dans l'équation de s_1 pour retrouver la clé privée a :

$$a = \frac{s_1 \cdot k - h(m_1)}{r}$$

3.3.1 Résolution

Il suffit donc de deux signatures avec le même nonce k donc le même r pour remonter à la clé privée a de manière exacte. Comme ci-dessous :

```
h1 = h(messages3[0])
h2 = h(messages3[1])
r1, s1 = signatures3[0]
r2, s2 = signatures3[1]

assert r1 == r2
r = F(r1)
s1 = F(s1); s2 = F(s2)

k = (h1 - h2) / (s1 - s2)
a = (s1 * k - h1) / r
```

3.3.2 Résultats obtenus

Voici la clé privée trouvée avec le script pour le challenge 3 :

```
360501192232114648412411773098428472334892702465060332018757401094776609309432451876187775204294707797849235775436 47
```

3.4 Challenge 4

Dans ce dernier challenge, la fonction de signature `sign4` utilise un nonce déterministe mais pas constant dépendant à la fois de la **clé privée** a et du **message** m, généré via SHA256 :

```
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))
```

Contrairement aux challenges précédents :

- Le nonce k est **déterministe** mais **dépend du message et de la clé privée**, donc il varie pour chaque signature. Il est donc non-constant.
- Cependant, il est **faible en entropie** : il provient du hash SHA256, soit une valeur de seulement 256 bits.

Le nonce k est donc trop petit par rapport à la taille de l'ordre n de la courbe, qui est typiquement de 384 bits (puisque le hachage utilisé dans la signature est SHA384) :

$$k = \text{int}(\texttt{SHA256}(a||m)) \in [0, 2^{256}] \quad \text{alors que} \quad n \sim 2^{384}$$

Cela rend le système vulnérable à une attaque par **LLL**, comme dans le Challenge 1, à condition d'adapter correctement la borne B pour optimiser l'algorithme.

Le script utilise exactement la même méthode que dans le Challenge 1, mais avec une nouvelle borne $B = 2^{256}$ correspondant à la taille maximale du nonce SHA256 :

```
B = 2^256
...
matrice[m, m] = B / n
matrice[m+1, m+1] = B
matrice_reduite = matrice.LLL()
```

Après réduction, les vecteurs courts obtenus via LLL contiennent une approximation de la valeur $-B \cdot a/n$, ce qui permet de retrouver directement des candidats pour la clé privée a, sans passer par le nonce k.

Le script extrait plusieurs candidats à partir de la composante x_{m+1} des lignes de la matrice réduite :

```
for ligne in matrice_reduite.rows():
    valeur_x = ligne[-2]
    candidat_a = Integer(-valeur_x * n * inverse_mod(B, n))
```

Chaque candidat est ensuite testé à l'aide de la fonction `correspondance_cle`, qui vérifie si le point $a \cdot G$ correspond à la clé publique A fournie.

3.4.1 Résultats obtenus

Voici la clé privée trouvée avec le script pour le challenge 4 :

`64746123777786818153603731722647636707339986688573381997154323594844534456224187620174609121234940076799810884679 12`

4 Conclusion

Ces quatre challenges montrent que la sécurité d'ECDSA repose fortement sur la qualité du nonce. Qu'il soit trop petit, réutilisé ou déterministe de manière prévisible, la clé privée devient vulnérable.

Les scripts complets sont dans le .zip avec les délivrables.