rapport_labo03_fullemann

Nathan Füllemann

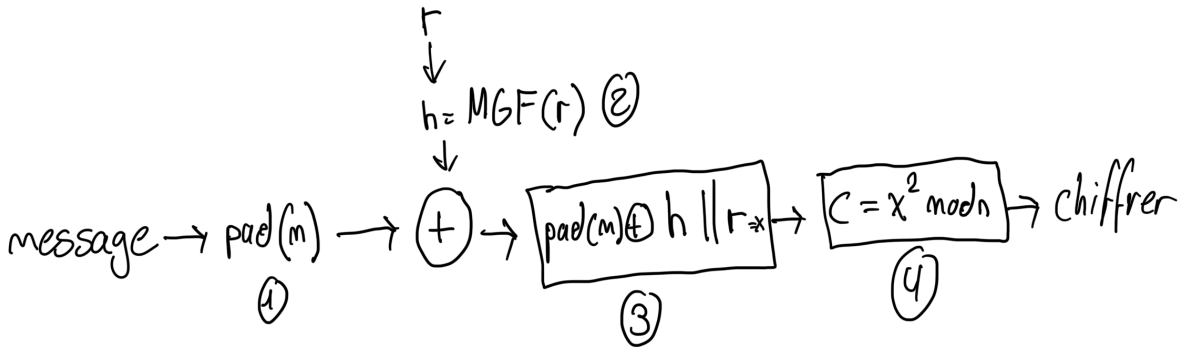
June 1, 2025

1 Rapport_CRY_labo03 - Nathan Füllemann

1.1 Exercice 1 - "Encryption" (2 pts)

1.1.1 Donnée

Le fichier encryption.sage contient un algorithme de chiffrement asymétrique. 1. Dessinez un schéma correspondant à cet algorithme. (0.2 pts) 2. Donnez une description mathématique du chiffrement et du déchiffrement correspondant. (0.2 pts) Indice : Regardez bien ce que fait la fonction pad. 3. Implémentez le déchiffrement. Testez votre code et montrez votre test dans votre rapport. (0.3 pts) 4. On suppose que les quatre racines carrées d'un texte chiffré ont fuité lors du déchiffrement. Vous trouverez dans votre fichier de paramètres la clef publique n, un message clair m, son texte chiffré correspondant c, les quatre racines carrées de c. Vous trouverez aussi un texte chiffré "challenge" chiffré avec la même clef. Décryptez-le. Donnez dans votre rapport le message récupéré ainsi qu'une explication mathématique expliquant votre attaque. (1 pt) Indice : Visualisez ces racines dans Zp × Zq et effectuez des opérations sur ces dernières afin d'obtenir un multiple de p ou de q. 5. Sur quel problème difficile est basé cette construction ? (0.1 pt) 6. A quoi sert la redondance dans la construction (variable REDUNDANCY) ? (0.2 pts) ### Réponses 1) Schema de l'algorithme



① padding du message

② génération du masque (h)

③ XOr du message paddéavec le masque et concaténation avec l'aléa

④ élévation au carré et modulo n

2) Description mathématique * $n = pq$, où $p \equiv q \equiv 3 \mod 4$ sont deux grands nombres premiers, * $m \in \{0,1\}^*$ le message en clair, * $r \in_R \{0,1\}^{128 \cdot 8}$ une chaîne aléatoire de 128 octets, * $\mathrm{MGF}(r, \ell)$ une fonction de masquage déterministe basée sur r et de longueur ℓ, * $\mathrm{pad}(m)$ le message avec un padding qui ajoute un octet `0x80` suivi de `0x00` jusqu'à atteindre la taille cible.

Étapes mathématiques :

1. **Padding :** $m_{\text{pad}} = \mathrm{pad}(m) \in \{0,1\}^\ell$
2. **Masquage :** $h = \mathrm{MGF}(r, \ell)$ $m' = m_{\text{pad}} \oplus h$
3. **Concaténation :** $x = \text{bytes\_to\_int}(m' \| r) \in \mathbb{Z}_n$
4. **Chiffrement :** $c = x^2 \mod n$

3) Déchiffrement

```
message chiffrer 47971480592815291514956765060193364994647084367851137480635304507324144819100
Message original : b'Crypto'
Message déchiffré : b'Crypto'
Succès : True
```

1) Cassage

    1) L'algorithme

En connaissant les 4 racines carrées r_1, r_2, r_3, r_4 d'un même message chiffré c, on peut factoriser n en exploitant le fait que :

$$r_1^2 \equiv r_2^2 \mod n \Rightarrow (r_1 - r_2)(r_1 + r_2) \equiv 0 \mod n$$

Donc :

$$\gcd(r_1 - r_2, n) = p \quad \text{ou} \quad q$$

Mon attaque se base sur les 4 racines et une fois p et q obtenus, on peut déchiffrer n'importe quel message chiffré avec n.

2) Résultat

```
Facteurs récupérés : p = 165527578793874775288637020134360874828287960545808881015503547430743992512185022046151
q = 151454526570491640123745343356079546942489923225137567709201369042178064633128913941601
Facteurs récupérés : p = 151454526570491640123745343356079546942489923225137567709201369042178064633128913941601
q = 165527578793874775288637020134360874828287960545808881015503547430743992512185022046151
Facteurs récupérés : p = 165527578793874775288637020134360874828287960545808881015503547430743992512185022046151
q = 151454526570491640123745343356079546942489923225137567709201369042178064633128913941601
Message challenge déchiffré : b'Ni! Ni! Ni! We want a adenoidal'
```

2) Le problème difficile Le problème difficile est la difficulté de factoriser un grand entier n = p*q où p et q sont des très grand nombre premier et $p \equiv q \equiv 3 \mod 4$.

3) La redondance dans la construction Dans ce bout de code

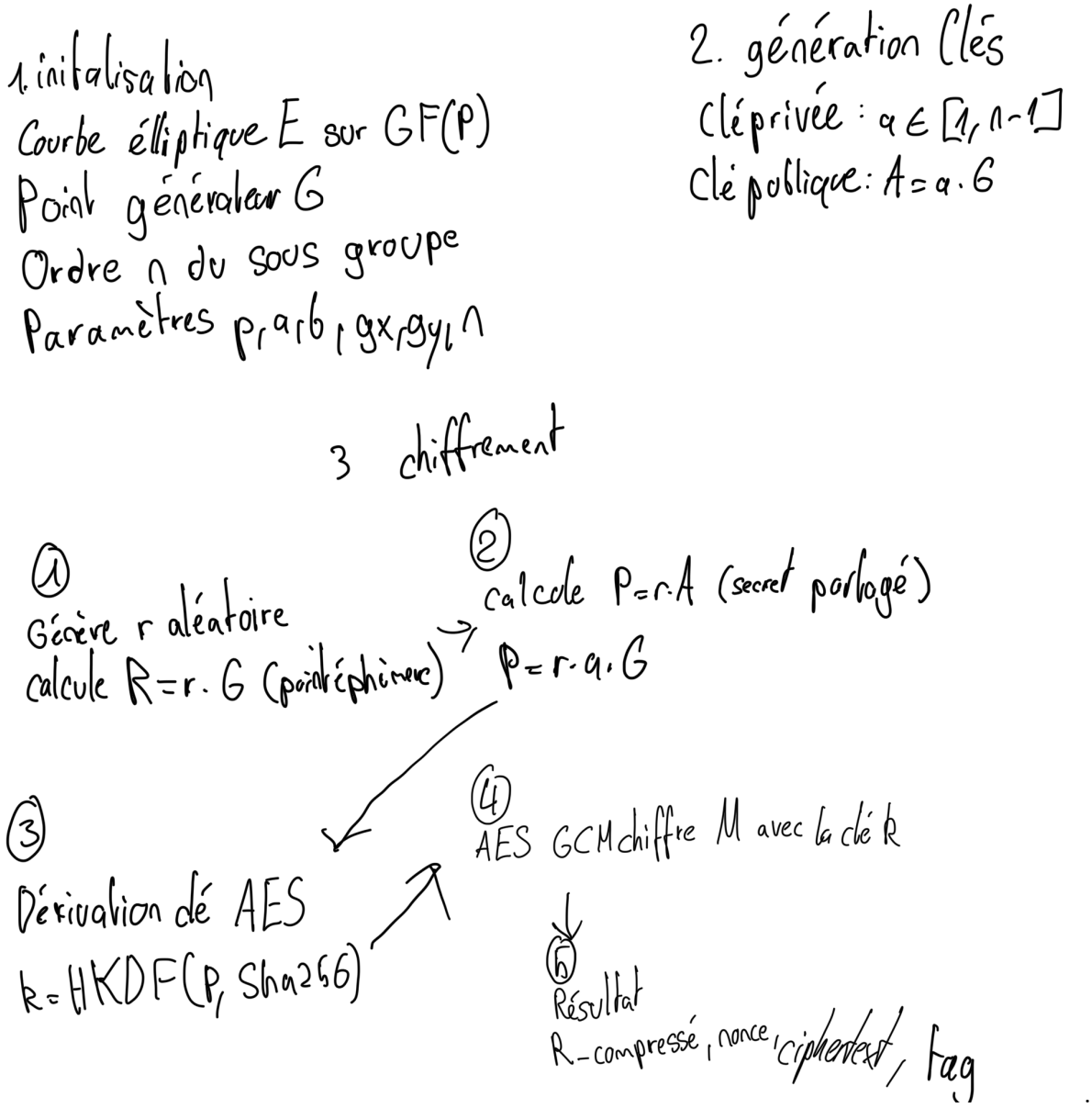
```
if len(m) > BYTE_LEN_MESSAGE_PART - REDUNDANCY - 1:
    raise Exception("Message too long.")
```

On peut trouver le paramètre **REDUNDANCY** qui est set a 10 au début du code. Ici on nous indique qu'il y a 10 octets de redondance qui permette au padding de créer une structure reconnaissable qui élimine les "fausses" racines. Car par exemple si c'était pas implémenté on pourrait choisir un chiffré **c=x^2 mod n** où il connaît x puis obtenir les 4 racines possibles via la méthode de dechiffrement et ainsi exploiter cette information pour factoriser n. Donc les 10 octets de redondance permette que la probabilité qu'une racine incorrecte passe la validation soit négligeable.

1.2 Exercice 2 - Courbes Elliptiques (2 pts)

1.2.1 Donnée

Vous trouverez dans le fichier elliptic.sage un algorithme de chiffrement asymétrique basé sur les courbes elliptiques. 1. Dessinez un schéma correspondant à cet algorithme. (0.2 pts) 2. Donnez une description mathématique du chiffrement et du déchiffrement correspondant. (0.2 pts) 3. Implémentez le déchiffrement. Testez votre code et montrez votre test dans votre rapport. (0.3 pts) 4. Il y a un problème dans cet algorithme. Lequel ? (0.1 pts) 5. Cassez la construction. Vous trouverez dans votre fichier de paramètres la clef publique et un texte chiffré complet (avec nonce et tag). Tout est en base64. Donnez dans votre rapport le message récupéré ainsi qu'une explication de votre attaque. (1 pt) 6. Corrigez l'erreur dans la construction. (0.2 pts) ### Réponses 1) Schéma de la construction



2) Description mathématique 1) **Paramètres publics et secrets** On utilise une **courbe elliptique** $E/\mathbb{F}_p$, définie par cette équation:

$$E : y^2 = x^3 + ax + b \quad \text{sur} \quad \mathbb{F}_p$$

Avec : * $p \in \mathbb{N}$: un nombre premier * $a, b \in \mathbb{F}_p$ * $G \in E(\mathbb{F}_p)$: un générateur d'ordre n * $n \in \mathbb{N}$: l'ordre du point G

2) **Génération de clé** Chaque utilisateur choisit :

- Une **clé privée** $a \in \mathbb{Z}_n$
- Une **clé publique** $A = aG \in E(\mathbb{F}_p)$

3) **Chiffrement** Pour chiffrer un message $M \in \{0,1\}^*$ (des octets), l'algorithme suis ses étapes :

1. **Choix aléatoire** : $r \in \mathbb{Z}_n$

2. **Calcul du point partagé (ECDH)** :

$$P = rA = r(aG) = arG \in E(\mathbb{F}_p)$$

3. **Compression du point** P, puis dérivation d'une **clé symétrique** k à partir de P :

$$k = \text{HKDF}(\text{compress}(P), \text{len} = 32)$$

4. **Chiffrement AES-GCM** :

$$\text{AES-GCM}_k(M) = (\text{nonce}, C, \text{tag})$$

5. L'expéditeur envoie :

$$\text{C} = (R, \text{nonce}, C, \text{tag})$$

où $R = rG \in E(\mathbb{F}_p)$ est la **clé publique éphémère** (compressée).

4) **Déchiffrement** Le destinataire, possédant la clé privée a, reçoit $R = rG$ et :

1. **Recalcule le point partagé** :

$$P = aR = a(rG) = arG$$

2. **Dérive la clé symétrique** :

$$k = \text{HKDF}(\text{compress}(P), \text{len} = 32)$$

3. **Déchiffre avec AES-GCM** :

$$M = \text{AES-GCM}_k^{-1}(\text{nonce}, C, \text{tag})$$

5) Implémentation du déchiffrement

```
Paramètres de la courbe elliptique
 - Générateur G : (34736706601617260336801089627448256371787243214661931571076381713565253
 - Ordre n      : 2550513000803
 - Equation     : Elliptic Curve defined by y^2 = x^3 + 433278833198111994429967057323651

 Clés de Bob
 - Clé privée a  : 963178213981
 - Clé publique A: (139760302346054137356436370331653815244326149842086108057566374856475O

Message clair à chiffrer : Crypto

 Chiffrement par Alice
 - Point éphémère R : 0224f3b266d0b8f75a2db2dfd7b07b92ed5d8bf089a7080de13222d7f69db25c59
 - Nonce AES                   : f2583519a4ed55affd03e5195006959d
 - Ciphertext                  : fdbef7e54cf0
 - Tag                         : ec216a1b8127eb3d9ee16b6a5c405357
 - Message déchiffré : Crypto

Le message envoyé de Bob et celui d'Alice sont les même
```

6) Problème de l'algorithme Ici le problème est que n est un nombre beaucoup trop petit

$$(2550513000803 \approx 2^{42})$$

Donc grace au logarithme discret on pourra casser très facillement la construction.

7) Cassage de la construction

   1) Explication Le code commence par vérifier que les points A (clé publique) et rG (point éphémère) appartiennent bien à la courbe elliptique. Ensuite la fonction *solve_discrete_log()* calcule la clé privée a en utilisant le fait que n est trop petit et en utilisant les logarithme discret. Puis on calcul le secret partagé

$$rA = a * rG$$

   Puis on dérivation de la clé AES avec HKDF et on fini par décrypter avec AES-GCM avec tous les paramètres qu'on possède.

   2) Résultat Clé privée récupérée: **2248123502064** Message déchiffré : **Nobody expects the spanish inquisition ! Our chief weapon is rechecked**

8) Correction de l'erreur En mettant simple un n beaucoup plus grand. Donc prendre une courbe avec 256 bits.

1.3 Exercice 3 - RSA (1pt)

1.3.1 Donnée

Vous trouverez dans le fichier rsa.sage une implémentation de textbook RSA. 1. Implémentez le déchiffrement. Testez votre code et montrez votre test dans votre rapport. (0.5 pts) 1. Cassez la construction. Vous trouverez dans votre fichier de paramètres la clef publique et un texte chiffré complet. Le texte chiffré est en base64. Donnez dans votre rapport le message récupéré ainsi qu'une explication de votre attaque. (0.5 pts)

1.3.2 Réponses

1) Implementation du decode RSA

```
Message original: b'Crypto'
Message chiffré: 0e3efc03db1255b4619b7160994c39581fa270ce33a2316a671a3bc0c299eb1f...
Message déchiffré: b'Crypto'
```

2) Cassage de la construction #### Cassage de la construction On peut voir que cette ligne est problematique `q = next_prime(p + ZZ.random_element(2^15))`. Car en temps normal dans la génération d'une clés RSA **p** et **q** ne sont pas en relation. Ici **q** depend de **p** et donc cela réduis l'espace de recherche pour un attaquant qui tenterait de factoriser n. Car au lieu de chercher parmi tous les nombres premiers possibles, il va le faire uniquement sur ceux qui sont proches de la racine carrée de n dans l'intervalle 2^15.

1.3.2.1 Résultat

Message déchiffré : **What is your quest? To seek the holy grail. What is your favorite color? waiving**