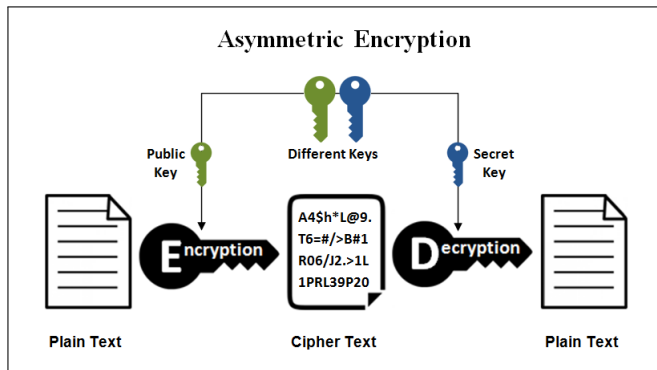


Attaques sur RSA

12345 – Prénom NOM

2022 / 2023 – *La ville*

RSA



Utilisation : HTTPS, cartes bancaires, ...

Plan

- 1 Introduction
- 2 Bases de RSA
- 3 Attaques
 - Premières attaques
 - Attaque de Wiener
 - Attaque de Håstad
- 4 Schéma de remplissage
- 5 Conclusion
- 6 Annexes
 - Correction de RSA
 - Preuve de l'attaque de Håstad
 - Preuve de l'attaque de Wiener
 - Structure du code
 - Code

Problématique / Objectifs

Problématique

Pourquoi l'algorithme RSA est-il utilisé alors qu'il existe des attaques sur cet algorithme ?

Problématique / Objectifs

Problématique

Pourquoi l'algorithme RSA est-il utilisé alors qu'il existe des attaques sur cet algorithme ?

Objectifs

- 1 Implémentation de l'algorithme RSA (version naïve, et avec un schéma de remplissage) ;

Problématique / Objectifs

Problématique

Pourquoi l'algorithme RSA est-il utilisé alors qu'il existe des attaques sur cet algorithme ?

Objectifs

- 1 Implémentation de l'algorithme RSA (version naïve, et avec un schéma de remplissage) ;
- 2 Étude et implémentation d'attaques sur RSA ;

Problématique / Objectifs

Problématique

Pourquoi l'algorithme RSA est-il utilisé alors qu'il existe des attaques sur cet algorithme ?

Objectifs

- 1 Implémentation de l'algorithme RSA (version naïve, et avec un schéma de remplissage) ;
- 2 Étude et implémentation d'attaques sur RSA ;
- 3 Montrer que l'implémentation de RSA doit être faite avec précaution.

Bases de RSA

Étapes de l'utilisation de RSA

- 1 Génération des clés ;

Bases de RSA

Étapes de l'utilisation de RSA

- 1 Génération des clés ;
- 2 Chiffrement ;

Bases de RSA

Étapes de l'utilisation de RSA

- 1 Génération des clés ;
- 2 Chiffrement ;
- 3 Déchiffrement.

Génération des clés

- Alice choisit p, q deux grands nombres premiers.

Génération des clés

- Alice choisit p, q deux grands nombres premiers.
- Le *module de chiffrement* est $n = pq$.

Génération des clés

- Alice choisit p, q deux grands nombres premiers.
- Le *module de chiffrement* est $n = pq$.
- On note $\varphi = \Phi(n) = (p - 1)(q - 1)$

Génération des clés

- Alice choisit p, q deux grands nombres premiers.
- Le *module de chiffrement* est $n = pq$.
- On note $\varphi = \Phi(n) = (p - 1)(q - 1)$
- Alice choisit $e \in \llbracket 3 ; \varphi - 1 \rrbracket$ premier avec φ , appelé *exposant de chiffrement*

Génération des clés

- Alice choisit p, q deux grands nombres premiers.
- Le *module de chiffrement* est $n = pq$.
- On note $\varphi = \Phi(n) = (p - 1)(q - 1)$
- Alice choisit $e \in \llbracket 3 ; \varphi - 1 \rrbracket$ premier avec φ , appelé *exposant de chiffrement*
- Alice calcule $d \equiv e^{-1} [\varphi]$, appelé *exposant de déchiffrement*

Génération des clés

- Alice choisit p, q deux grands nombres premiers.
- Le *module de chiffrement* est $n = pq$.
- On note $\varphi = \Phi(n) = (p - 1)(q - 1)$
- Alice choisit $e \in \llbracket 3 ; \varphi - 1 \rrbracket$ premier avec φ , appelé *exposant de chiffrement*
- Alice calcule $d \equiv e^{-1} [\varphi]$, appelé *exposant de déchiffrement*

- Clé publique : (e, n)
- Clé privée : (d, n)

Chiffrement / Déchiffrement

Soit $m \in \llbracket 0 ; n - 1 \rrbracket$.

Chiffrement / Déchiffrement

Soit $m \in \llbracket 0 ; n - 1 \rrbracket$.

Chiffrement

On chiffre m par

$$c \equiv m^e \pmod{n}$$

Chiffrement / Déchiffrement

Soit $m \in \llbracket 0 ; n - 1 \rrbracket$.

Chiffrement

On chiffre m par

$$c \equiv m^e [n]$$

Déchiffrement

Pour déchiffrer c :

$$m \equiv c^d [n]$$

Correction : $c^d \equiv m^{ed} \equiv m^{k\varphi+1} \equiv m [n]$ par le théorème d'Euler.

Plan

- 1 Introduction
- 2 Bases de RSA
- 3 Attaques**
 - Premières attaques
 - Attaque de Wiener
 - Attaque de Håstad
- 4 Schéma de remplissage
- 5 Conclusion
- 6 Annexes
 - Correction de RSA
 - Preuve de l'attaque de Håstad
 - Preuve de l'attaque de Wiener
 - Structure du code
 - Code

Propriété multiplicative

Propriété :

On a :

$$(m_1 m_2)^e \equiv m_1^e m_2^e \pmod{n}$$

Propriété multiplicative

Propriété :

On a :

$$(m_1 m_2)^e \equiv m_1^e m_2^e [n]$$

- Eve intercepte $c \equiv m^e [n]$.

Propriété multiplicative

Propriété :

On a :

$$(m_1 m_2)^e \equiv m_1^e m_2^e [n]$$

- Eve intercepte $c \equiv m^e [n]$.
- Soit $r \in \llbracket 1 ; n - 1 \rrbracket \mid r \wedge n = 1$
Soit $c' \equiv cr^e \equiv (mr)^e [n]$

Propriété multiplicative

Propriété :

On a :

$$(m_1 m_2)^e \equiv m_1^e m_2^e [n]$$

- Eve intercepte $c \equiv m^e [n]$.
- Soit $r \in \llbracket 1 ; n - 1 \rrbracket \mid r \wedge n = 1$
Soit $c' \equiv cr^e \equiv (mr)^e [n]$
- Elle demande à Alice de déchiffrer $c' \rightarrow m' \equiv c'^d [n]$

Propriété multiplicative

Propriété :

On a :

$$(m_1 m_2)^e \equiv m_1^e m_2^e [n]$$

- Eve intercepte $c \equiv m^e [n]$.
- Soit $r \in \llbracket 1 ; n - 1 \rrbracket \mid r \wedge n = 1$
Soit $c' \equiv cr^e \equiv (mr)^e [n]$
- Elle demande à Alice de déchiffrer $c' \rightarrow m' \equiv c'^d [n]$
- Eve récupère le message : $m \equiv m' r^{-1} [n]$

Factorisation de n avec φ

On a :

$$\begin{cases} n = pq \\ \varphi = (p-1)(q-1) \end{cases}$$

$$\Leftrightarrow \begin{cases} n = pq \\ \varphi = pq - p - q + 1 \end{cases}$$

$$\Leftrightarrow \begin{cases} pq = n \\ p + q = n - \varphi + 1 \end{cases}$$

$$\Leftrightarrow p, q \text{ solutions de } x^2 - (n - \varphi + 1)x + n = 0$$

Il suffit donc de calculer les racines de $x^2 - (n - \varphi + 1)x + n$

Plan

- 1 Introduction
- 2 Bases de RSA
- 3 Attaques**
 - Premières attaques
 - **Attaque de Wiener**
 - Attaque de Håstad
- 4 Schéma de remplissage
- 5 Conclusion
- 6 Annexes
 - Correction de RSA
 - Preuve de l'attaque de Håstad
 - Preuve de l'attaque de Wiener
 - Structure du code
 - Code

Attaque de Wiener – Notations

L'*attaque de Wiener* permet de récupérer la *clé privée* (d, n) à partir de la *clé publique* (e, n) sous certaines conditions :

Attaque de Wiener – Notations

L'*attaque de Wiener* permet de récupérer la *clé privée* (d, n) à partir de la *clé publique* (e, n) sous certaines conditions :

- $q < p < 2q$;

Attaque de Wiener – Notations

L'*attaque de Wiener* permet de récupérer la *clé privée* (d, n) à partir de la *clé publique* (e, n) sous certaines conditions :

- $q < p < 2q$;
- $d < \frac{1}{3}n^{\frac{1}{4}}$.

Idée

On peut montrer que sous ces conditions,

$$\left| \frac{e}{n} - \frac{k}{d} \right| \leq \frac{1}{2d^2} \quad (1)$$

Où $k = \frac{ed - 1}{\varphi}$.

- Donc $\frac{e}{n}$ est une *approximation* de $\frac{k}{d}$
- On va pouvoir retrouver k et d à l'aide des *fractions continues*.

Fractions continues I

Définition (*fraction continue*)

Une *fraction continue* est une fraction du type :

$$a_0 + \frac{1}{a_1 + \frac{1}{a_2 + \frac{1}{\ddots + \frac{1}{a_{n-1} + \frac{1}{a_n}}}}}$$

où $a_0 \in \mathbb{N}$, et $\forall k \in \llbracket 1 ; n \rrbracket$, $a_k \in \mathbb{N}^*$.

On la note $[a_0, \dots, a_n]$.

Fractions continues II

Définition (*réduites*)

Soit $f = [a_0, \dots, a_n]$ une fraction continue.

Soient :

$$\begin{cases} p_{-2} = 0 \\ p_{-1} = 1 \\ p_k = a_k p_{k-1} + p_{k-2} \end{cases} \quad \begin{cases} q_{-2} = 1 \\ q_{-1} = 0 \\ q_k = a_k q_{k-1} + q_{k-2} \end{cases}$$

Alors les *réduites* de f sont les fractions ($k \in \llbracket 0 ; n \rrbracket$) :

$$\frac{p_k}{q_k}$$

Fractions continues III

Fraction continue d'un rationnel

Soient $(a, b) \in \mathbb{Z} \times \mathbb{N}^*$.

On peut calculer la fraction continue de $\frac{a}{b}$ avec l'algorithme suivant :

```
1 def get_continued_fraction_rec(a, b, f=[]):
2     '''Return a ContinuedFraction object, the continued
3     fraction of a/b. This is a recursive function.'''
4
5     # euclidean division : a = bq + r
6     q = a // b
7     r = a % b
8
9     if r == 0:
10         return ContinuedFraction(f + [q])
11
12     return get_continued_fraction_rec(b, r, f + [q])
```

Fractions continues IV

Théorème

Soient $a, a' \in \mathbb{Z}$, et $b, b' \in \mathbb{Z}^*$ tels que

$$\left| \frac{a}{b} - \frac{a'}{b'} \right| < \frac{1}{2b^2}$$

Alors $\frac{a}{b}$ est une réduite de $\frac{a'}{b'}$.

→ On déduit donc de (1) que $\frac{k}{d}$ est une réduite de $\frac{e}{n}$.

Attaque de Wiener

- On calcule les réduites de $\frac{e}{n}$, que l'on note $\frac{k_i}{d_i}$;

Attaque de Wiener

- On calcule les réduites de $\frac{e}{n}$, que l'on note $\frac{k_i}{d_i}$;
- On calcule $\varphi_i = \frac{e \cdot d_i - 1}{k_i}$;

Attaque de Wiener

- On calcule les réduites de $\frac{e}{n}$, que l'on note $\frac{k_i}{d_i}$;
- On calcule $\varphi_i = \frac{e \cdot d_i - 1}{k_i}$;
- On essaye de factoriser n avec φ_i .

Extension de l'attaque de Wiener

- On considère un d très grand : il va être "petit", négatif modulo φ .

Extension de l'attaque de Wiener

- On considère un d très grand : il va être "petit", négatif modulo φ .
- On prend d tel que

$$\varphi - d < \frac{1}{3}n^{\frac{1}{4}}$$

Extension de l'attaque de Wiener

- On considère un d très grand : il va être "petit", négatif modulo φ .
- On prend d tel que

$$\varphi - d < \frac{1}{3}n^{\frac{1}{4}}$$

- On pose $D = \varphi - d \equiv -d \pmod{\varphi}$

Extension de l'attaque de Wiener

- On considère un d très grand : il va être "petit", négatif modulo φ .
- On prend d tel que

$$\varphi - d < \frac{1}{3}n^{\frac{1}{4}}$$

- On pose $D = \varphi - d \equiv -d \pmod{\varphi}$
- D satisfait les propriétés précédentes : on va pouvoir de nouveau réaliser l'attaque, avec

$$\varphi_i = \frac{e \cdot d_i + 1}{k_i}$$

Résultats

```
6. Testing Wiener's attack :  
Key generation for Wiener's attack (2048 bits) ...  
Key generated in 0:00:15.661398s.  
Wiener's attack finished in 0:00:00.077236s.  
Correct result !  
-----  
7. Testing Wiener's attack with a large private exponent :  
Key generation for Wiener's attack (2048 bits) ...  
Key generated in 0:00:09.696836s.  
Wiener's attack finished in 0:00:00.077087s.  
Correct result !
```

Plan

- 1 Introduction
- 2 Bases de RSA
- 3 Attaques**
 - Premières attaques
 - Attaque de Wiener
 - **Attaque de Håstad**
- 4 Schéma de remplissage
- 5 Conclusion
- 6 Annexes
 - Correction de RSA
 - Preuve de l'attaque de Håstad
 - Preuve de l'attaque de Wiener
 - Structure du code
 - Code

Attaque de Håstad – Notations

Alice envoie un même message m à p destinataires ayant le même exposant de chiffrement :

$$(S) \quad \begin{cases} c_1 \equiv m^e [n_1] \\ \vdots \\ c_p \equiv m^e [n_p] \end{cases}$$

On suppose que tous les modules ont la même taille s
($\forall k \in \llbracket 1 ; p \rrbracket$, $\log_2(n_k) \approx s$). Généralement, $s = 2048$.

L'**attaque de Håstad** va permettre de récupérer le message m sous certaines conditions.

Solution au système

Par le théorème des restes chinois,

$$(S) \Leftrightarrow m^e \equiv \sum_{k=1}^p c_k N_k M_k [N]$$

où $\forall k \in \llbracket 1 ; p \rrbracket$:

- $N = \prod_{k=1}^p n_k$

- $N_k = \frac{N}{n_k}$

- $M_k \equiv N_k^{-1} [n_k]$

Condition sur le nombre d'équations p

Pour retrouver le message, on a besoin que $m^e < N$.

$$\text{Or } N = \prod_{k=1}^p n_k \approx 2^{sp}$$

Donc il faut que

$$p > \frac{e}{s} \log_2(m)$$

$m \leq 2^s - 1$, donc dans le cas général, $p > e$.

Résultats

```
4. Testing Hastad's attack (e = 5) :  
Number of equations actually needed to recover the message : 2.
```

```
Key generation for Hastad's attack (2048 bits, 2 keys) ...  
1/2 generated in 0:00:07.160342s.  
2/2 generated in 0:00:02.721468s.  
Done in 0:00:09.881926s.
```

```
Hastad attack ...  
Attack done in 0:00:00.113410s.  
Input and output are identical.
```

```
5. Testing Hastad's attack, testing the limit number of equations needed (e = 5, random  
message of length 100 characters) :  
Number of equations theoretically needed to recover the message : 3.
```

```
Key generation for Hastad's attack (2048 bits) ...  
1/3 generated in 0:00:06.413951s.  
2/3 generated in 0:00:07.811984s.  
3/3 generated in 0:00:19.650791s.  
Done in 0:00:33.876816s.
```

```
Hastad attack with 3 equations ...  
Attack done in 0:00:00.350084s.  
Attack succeeded : message correctly recovered.
```

```
Hastad attack with 2 equations ...  
Attack done in 0:00:00.203013s.  
Attack failed : message not correctly recovered. So the limit is correct.
```


Nécessité d'un schéma de remplissage (*padding scheme*)

Problèmes :

- L'algorithme RSA est **déterministe** : tel quel, il n'est donc pas *sémantiquement sûr*.

Nécessité d'un schéma de remplissage (*padding scheme*)

Problèmes :

- L'algorithme RSA est **déterministe** : tel quel, il n'est donc pas *sémantiquement sûr*.
- Si e et m sont trop petits, on peut retrouver le message clair sans la clé privée (si $m^e < n$).

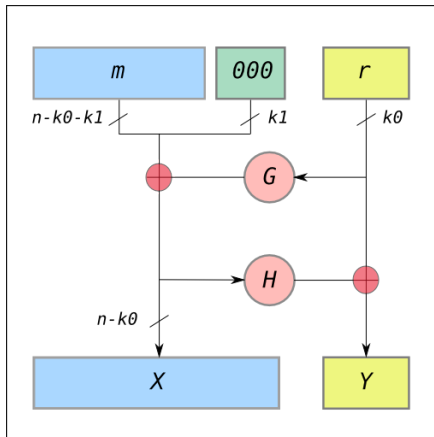
Nécessité d'un schéma de remplissage (*padding scheme*)

Problèmes :

- L'algorithme RSA est **déterministe** : tel quel, il n'est donc pas *sémantiquement sûr*.
- Si e et m sont trop petits, on peut retrouver le message clair sans la clé privée (si $m^e < n$).

→ Il faut donc utiliser un schéma de remplissage.

Le padding OAEP



Encodage :

$$X = m \overbrace{0 \dots 0}^{k_1} \oplus G(r)$$

$$Y = r \oplus H(X)$$

Décodage :

$$r = Y \oplus H(X)$$

$$m 0 \dots 0 = X \oplus G(r)$$

https://fr.wikipedia.org/wiki/Optimal_Asymmetric_Encryption_Padding

Conclusion

- L'utilisation d'un *padding* randomisé permet de se prémunir de l'attaque de Håstad ;
- Dans l'implémentation de la génération des clés, il faut vérifier que d n'est pas dans les conditions de l'attaque de Wiener.

Merci pour votre attention

Plan

- 1 Introduction
- 2 Bases de RSA
- 3 Attaques
 - Premières attaques
 - Attaque de Wiener
 - Attaque de Håstad
- 4 Schéma de remplissage
- 5 Conclusion
- 6 Annexes
 - Correction de RSA
 - Preuve de l'attaque de Håstad
 - Preuve de l'attaque de Wiener
 - Structure du code
 - Code

Preuve de correction de l'algorithme RSA

Théorème d'Euler

$\forall n \in \mathbb{N}^*, \forall a \in \llbracket 1 ; n \rrbracket \mid a \wedge n = 1$, on a :

$$a^{\phi(n)} \equiv 1 [n]$$

Comme $ed \equiv 1 [\varphi]$, $\exists k \in \mathbb{N} \mid ed = k\varphi + 1$.

Donc on a :

$$\boxed{c^d} \equiv m^{ed} \equiv m^{k\varphi+1} \equiv \boxed{m} [n]$$

Plan

- 1 Introduction
- 2 Bases de RSA
- 3 Attaques
 - Premières attaques
 - Attaque de Wiener
 - Attaque de Håstad
- 4 Schéma de remplissage
- 5 Conclusion
- 6 Annexes**
 - Correction de RSA
 - Preuve de l'attaque de Håstad
 - Preuve de l'attaque de Wiener
 - Structure du code
 - Code

Théorème des restes chinois

Théorème des restes chinois :

Soient $p \in \mathbb{N}^*$ et $(n_k)_{k \in \llbracket 1 ; p \rrbracket} \in (\mathbb{N}^* \setminus \{1\})^p$ tels que

$$\forall i, j \in \llbracket 1 ; p \rrbracket, i \neq j \Rightarrow n_i \wedge n_j = 1$$

Avec $N = \prod_{k=1}^p n_k$, on a que :

$$\begin{aligned} \psi : \mathbb{Z}/N\mathbb{Z} &\longrightarrow \prod_{k=1}^p \mathbb{Z}/n_k\mathbb{Z} \\ cl_N(x) &\longmapsto (cl_{n_1}(x), \dots, cl_{n_p}(x)) \end{aligned}$$

est un isomorphisme d'anneaux.

Preuve de l'attaque de Håstad I

On détermine ψ^{-1} :

$$\begin{aligned}
 & \psi^{-1}((cl_{a_1}(c_1), \dots, cl_{a_n}(c_n))) \\
 = & \psi^{-1}\left(\sum_{k=1}^n c_k (cl_{a_1}(0), \dots, cl_{a_{k-1}}(0), cl_{a_k}(1), cl_{a_{k+1}}(0), \dots, cl_{a_n}(0))\right) \\
 = & \sum_{k=1}^n c_k \underbrace{\psi^{-1}(cl_{a_1}(0), \dots, cl_{a_{k-1}}(0), cl_{a_k}(1), cl_{a_{k+1}}(0), \dots, cl_{a_n}(0))}_{cl_a(m_k)} \\
 = & \sum_{k=0}^n c_k cl_a(m_k)
 \end{aligned}$$

Preuve de l'attaque de Håstad II

Il suffit de trouver des m_k qui conviennent, c'est à dire tels que :

$$\forall k \in \llbracket 1 ; n \rrbracket, \begin{cases} m_k \in \mathbb{Z} \\ \forall i \in \llbracket 1 ; n \rrbracket \setminus \{k\}, m_k \equiv 0 [a_i] \\ m_k \equiv 1 [a_k] \end{cases}$$

Soit $A = \prod_{k=1}^n a_k$, et $\forall k \in \llbracket 1 ; n \rrbracket, A_k = \frac{A}{a_k}$

Comme les a_k sont deux à deux premiers entre eux,

$\forall k \in \llbracket 1 ; n \rrbracket, A_k \wedge a_k = 1$, donc avec le théorème de Bézout :

$$\exists B_k, b_k \in \mathbb{Z} \mid A_k B_k + a_k b_k = 1$$

$$(B_k \equiv (A_k)^{-1} [a_k])$$

Preuve de l'attaque de Håstad III

Soient $\forall k \in \llbracket 1 ; n \rrbracket$, $m_k = A_k B_k \in \mathbb{Z}$.

On a, $\forall k \in \llbracket 1 ; n \rrbracket$:

$$m_k \equiv A_k B_k \equiv 1 - a_k b_k \equiv 1 \pmod{a_k}$$

et $\forall i \in \llbracket 1 ; n \rrbracket \setminus \{k\}$:

$$m_k \equiv A_k B_k \equiv 0 \pmod{a_i}$$

car $a_i | A_k$.

Donc finalement :

$$\psi^{-1}(cl_{a_1}(c_1), \dots, cl_{a_n}(c_n)) = \sum_{k=1}^n c_k cl_a(A_k B_k)$$

Plan

- 1 Introduction
- 2 Bases de RSA
- 3 Attaques
 - Premières attaques
 - Attaque de Wiener
 - Attaque de Håstad
- 4 Schéma de remplissage
- 5 Conclusion
- 6 Annexes**
 - Correction de RSA
 - Preuve de l'attaque de Håstad
 - Preuve de l'attaque de Wiener**
 - Structure du code
 - Code

Preuve de l'attaque de Wiener I

Comme $ed \equiv 1 \pmod{\varphi}$, $\exists k \in \mathbb{N} \mid ed - k\varphi = 1$, donc :

$$\begin{aligned}\frac{ed - k\varphi}{d\varphi} &= \frac{1}{d\varphi} \\ \Rightarrow \frac{e}{\varphi} - \frac{k}{d} &= \frac{1}{d\varphi} \\ \Rightarrow \left| \frac{e}{\varphi} - \frac{k}{d} \right| &= \frac{1}{d\varphi}\end{aligned}$$

Donc $\frac{k}{d}$ est une approximation de $\frac{e}{\varphi}$.

On peut essayer d'approximer φ avec n :

$$\varphi = \phi(n) = (p-1)(q-1) = n - p - q + 1$$

Preuve de l'attaque de Wiener II

Comme $\begin{cases} p < 2q \\ q < p \end{cases}$ (par hypothèse), on a :

$$\begin{cases} p + q < 3q \\ q^2 < pq = n \end{cases} \Rightarrow \begin{cases} p + q < 3q \\ q < \sqrt{n} \end{cases} \Rightarrow p + q < 3\sqrt{n} \Rightarrow p + q - 1 < 3\sqrt{n}$$

Donc $|n - \varphi| = |p + q - 1| < 3\sqrt{n}$.

On a ensuite :

Preuve de l'attaque de Wiener III

$$\begin{aligned} \left| \frac{e}{n} - \frac{k}{d} \right| &= \left| \frac{ed - nk}{nd} \right| \\ &= \left| \frac{ed - k\varphi + k\varphi - nk}{nd} \right| \\ &= \left| \frac{1 - k(n - \varphi)}{nd} \right| \\ &< \frac{1 + |k(n - \varphi)|}{|nd|} \\ &\ll \left| \frac{k(n - \varphi)}{nd} \right| \ll \left| \frac{3k\sqrt{n}}{nd} \right| = \frac{3k}{d\sqrt{n}}. \end{aligned}$$

Preuve de l'attaque de Wiener IV

Ensuite, $k\varphi = ed - 1 < ed$ et $e < \varphi$, donc $k < \frac{e}{\varphi}d < d$, donc :

$$k < d < \frac{1}{3}n^{\frac{1}{4}} \Rightarrow \frac{k}{d} < 1 < \frac{n^{\frac{1}{4}}}{3d}$$

Donc :

$$\begin{aligned} \left| \frac{e}{n} - \frac{k}{d} \right| &\leq \frac{k}{d} \frac{3}{\sqrt{n}} \\ &\leq \frac{n^{\frac{1}{4}}}{3d} \frac{3}{\sqrt{n}} \\ &= \frac{1}{dn^{\frac{1}{4}}} \end{aligned}$$

Preuve de l'attaque de Wiener V

Et :

$$2d^2 < \frac{2}{3}dn^{\frac{1}{4}} < dn^{\frac{1}{4}} \Rightarrow \frac{3}{2dn^{\frac{1}{4}}} < \frac{1}{2d^2}$$

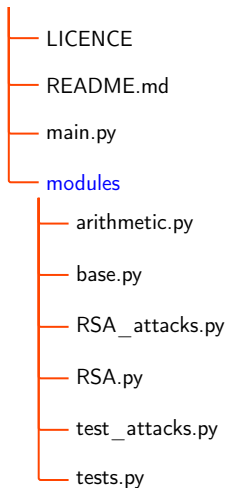
D'où :

$$\left| \frac{e}{n} - \frac{k}{d} \right| \leq \frac{1}{dn^{\frac{1}{4}}} \leq \frac{1}{2d^2}$$

Plan

- 1 Introduction
- 2 Bases de RSA
- 3 Attaques
 - Premières attaques
 - Attaque de Wiener
 - Attaque de Håstad
- 4 Schéma de remplissage
- 5 Conclusion
- 6 Annexes**
 - Correction de RSA
 - Preuve de l'attaque de Håstad
 - Preuve de l'attaque de Wiener
 - **Structure du code**
 - Code

Structure du code



Plan

- 1 Introduction
- 2 Bases de RSA
- 3 Attaques
 - Premières attaques
 - Attaque de Wiener
 - Attaque de Håstad
- 4 Schéma de remplissage
- 5 Conclusion
- 6 Annexes
 - Correction de RSA
 - Preuve de l'attaque de Håstad
 - Preuve de l'attaque de Wiener
 - Structure du code
 - Code

main.py |

```
1 | #!/bin/python3
2 | # -*- coding: utf-8 -*-
3 |
4 | '''Main file running tests on the attacks'''
5 |
6 | ##-Import
7 | from modules.test_attacks import *
8 |
9 | from datetime import datetime as dt
10 | from sys import argv
11 | from sys import exit as sysexit
12 |
13 | ##-Run tests function
14 | def run_tests(size=2048):
15 |     '''Run the tests defined in the file 'test_attacks.py'.
16 |     '''
17 |     passed = []
18 |
```

main.py II

```
19     t0 = dt.now()
20
21     try:
22         print('Launching tests...')
23         print('-' * 16)
24
25         print('1. Testing factorisation of the modulus with
the private exponent :')
26         passed.append(test_mod_fact(size))
27         print('-' * 16)
28
29         print('2. Testing common modulus (finding the
private exponent knowing a key set with the same
exponent) :')
30         passed.append(test_common_mod(size))
31         print('-'*16)
32
33         print('3. Testing multiplicative attack :')
34         passed.append(test_multiplicative_attack(size))
```


main.py III

```
35     print('-'*16)
36
37     print('4. Testing Hastad\'s attack (e = 5) :')
38     passed.append(test_hastad(msg='Testing this attack
with this message, because a message is needed.', size=
size, e=5))
39     print('-'*16)
40
41     print('5. Testing Hastad\'s attack, testing the
limit number of equations needed (e = 5, random message
of length 100 characters) :')
42     passed.append(test_hastad_message_size(size=size, e
=5))
43     print('-'*16)
44
45     print('6. Testing Wiener\'s attack :')
46     passed.append(test_wiener(size=size))
47     print('-'*16)
48
```

main.py IV

```
49         print('7. Testing Wiener\'s attack with a large
private exponent :')
50         passed.append(test_wiener(size=size, large=True))
51         print('-'*16)
52
53         print(f'\nDone in {dt.now() - t0}s.')
```

```
54
55     except KeyboardInterrupt:
56         print(f'\nStopped. Time elapsed : {dt.now() - t0}s.\
nNumber of tests done : {len(passed)}')
```

```
57
58     if not False in passed:
59         print('\nAll tests passed correctly !')
```

```
60
61     else:
62         print('\nThe following tests failed :')
```

```
63
64         for k, b in enumerate(passed):
65             if not b:
```

main.py ✓

```
66         print(f'\t{k + 1}')
```

```
67
```

```
68 ## - Run
```

```
69 if __name__ == '__main__':
```

```
70     if len(argv) == 1:
```

```
71         size = 2048
```

```
72
```

```
73     else:
```

```
74         try:
```

```
75             size = int(argv[1])
```

```
76
```

```
77         except:
```

```
78             print(f'Wrong argument at position 1 : should be  
the RSA key size (in bits).\nExample : "{argv[0]}  
2048".')
```

```
79             sys.exit()
```

```
80
```

```
81     run_tests(size)
```

arithmetic.py |

```
1 | #!/bin/python3
2 | # -*- coding: utf-8 -*-
3 |
4 | '''Useful arithmetic functions'''
5 |
6 | ##-Imports
7 | from random import randint
8 | from math import floor, ceil, sqrt, isqrt
9 | from fractions import Fraction
10 | from gmpy2 import is_square
11 |
12 |
13 | ##-Multiplicative inverse
14 | def mult_inverse(a: int, n: int) -> int:
15 |     '''
16 |         Return the multiplicative inverse u of a modulo n.
17 |         u*a = 1 modulo n
18 |     '''
19 |
```

arithmetic.py II

```
20     (old_r, r) = (a, n)
21     (old_u, u) = (1, 0)
22
23     while r != 0:
24         q = old_r // r
25         (old_r, r) = (r, old_r - q*r)
26         (old_u, u) = (u, old_u - q*u)
27
28     if old_r > 1:
29         raise ValueError(str(a) + ' is not inversible in the
30         ring Z/' + str(n) + 'Z.')
```

```
31     if old_u < 0:
32         return old_u + n
33
34     else:
35         return old_u
36
37
```

arithmetic.py III

```
38  ##-Max parity
39  def max_parity(n):
40      '''return (t, r) such that n = 2^t * r, where r is odd
41      '''
42
43      t = 0
44      r = int(n)
45      while r % 2 == 0 and r > 1:
46          r //= 2
47          t += 1
48
49      return (t, r)
50
51  ##-Probabilistic prime test
52  def isSurelyPrime(n):
53      '''Check if n is probably prime. Uses Miller Rabin test.
54      '''
```

arithmetic.py IV

```
55     if n == 2:
56         return True
57
58     elif n % 2 == 0:
59         return False
60
61     return miller_rabin(n, 15)
62
63
64 def miller_rabin_witness(a, d, s, n):
65     '''
66     Return True if a is a Miller-Rabin witness.
67
68     - a : the base ;
69     - d : odd integer verifying  $n - 1 = 2^s d$  ;
70     - s : positive integer verifying  $n - 1 = 2^s d$  ;
71     - n : the odd integer to test primality.
72     '''
73
```

arithmetic.py ✓

```
74     r = pow(a, d, n)
75
76     if r == 1 or r == n - 1:
77         return False
78
79     for k in range(s):
80         r = r**2 % n
81
82         if r == n - 1:
83             return False
84
85     return True
86
87
88 def miller_rabin(n, k=15) :
89     '''
90     Return the primality of n using Miller-Rabin
91     probabilistic primality test.
```


arithmetic.py VI

```
92 |     - n : odd integer to test the primality ;
93 |     - k : number of tests (Error =  $4^{-k}$ ).
94 |     '''
95 |
96 |     if n in (0, 1):
97 |         return False
98 |
99 |     if n == 2:
100 |         return True
101 |
102 |     s, d = max_parity(n - 1)
103 |
104 |     for i in range(k) :
105 |         a = randint(2, n - 1)
106 |
107 |         if miller_rabin_witness(a, d, s, n):
108 |             return False
109 |
110 |     return True
```

arithmetic.py VII

```
111 |
112 |
113 | ##-iroot
114 | def iroot(n, k):
115 |     '''
116 |         Newton's method to find the integer k-th root of n.
117 |
118 |         Return floor(n^(1/k))
119 |     '''
120 |
121 |     u, s = n, n + 1
122 |
123 |     while u < s:
124 |         s = u
125 |         t = (k - 1) * s + n // pow(s, k - 1)
126 |         u = t // k
127 |
128 |     return s
129 |
```

arithmetic.py VIII

```
130
131
132 ##-Fermat factorisation
133 def feramat_factor(n):
134     '''
135     Try to factor n using Fermat's factorisation.
136     For  $n = pq$ , works better if  $|q - p|$  is small, i.e if p
137     and q
138     are near  $\sqrt{n}$ .
139     '''
140     a = iroot(n, 2)
141
142     while not is_square(pow(a, 2) - n):
143         a += 1
144
145         if pow(a, 2) - n <= 0:
146             return False
147
```

arithmetic.py IX

```
148     b = isqrt(pow(a, 2) - n)
149     return (a - b, a + b)
150
151
152 ##-Continued fractions
153 class ContinuedFraction:
154     '''Class representing a continued fraction.'''
155
156     def __init__(self, f):
157         '''
158             Initialize the class
159
160             - f : the int array representing the continued
161                 fraction.
162         '''
163
164         if type(f) in (set, list):
165             self.f = list(f)
```

arithmetic.py X

```
166         else:
167             raise ValueError('ContinuedFraction: error: 'f'
should be a list')
168
169         if len(f) == 0:
170             raise ValueError('ContinuedFraction: error: 'f'
should not be empty')
171
172         for j, k in enumerate(f):
173             if type(k) != int:
174                 raise ValueError(f'ContinuedFraction: error:
'f' should be a list of int, but '{k}' found at
position {j}')
175
176
177     def __repr__(self):
178         '''Return a pretty string representing the fraction.
179         '''
```

arithmetic.py XI

```
180         ret = f'{self.f[-1]}'
181
182         for k in reversed(self.f[:-1]):
183             ret = f'{k} + 1/(' + ret + ')',
184
185         return ret
186
187
188     def __eq__(self, other):
189         '''Test the equality between self and the other.'''
190
191         return self.f == other.f
192
193
194     def eval_rec(self):
195         '''Return the evaluation of self.f via a recursive
196         function.'''
197
198         return self._eval_rec(self.f)
```

arithmetic.py XII

```
198
199
200     def _eval_rec(self, f_):
201         '''The recursive function.'''
202
203         if len(f_) == 1:
204             return f_[0]
205
206         return f_[0] + 1/(self._eval_rec(f_[1:]))
207
208
209     def truncate(self, pos):
210         '''
211         Return a ContinuedFraction truncated at position '
212         pos' from self.f.
213
214         - pos : the position of the truncation. The element
215         at position 'pos' is kept in the result.
216         '''
```

arithmetic.py XIII

```
215 |
216 |         return ContinuedFraction(self.f[:pos + 1])
217 |
218 |
219 | def get_convergents(self):
220 |     '''
221 |     Return two lists, p, q which represents the
222 |     convergents :
223 |     the n-th convergent is 'p[n] / q[n]'.
224 |     '''
225 |     p = [0]*(len(self.f) + 2)
226 |     q = [0]*(len(self.f) + 2)
227 |
228 |     p[-1] = 1
229 |     q[-2] = 1
230 |
231 |     for k in range(0, len(self.f)):
232 |         p[k] = self.f[k] * p[k - 1] + p[k - 2]
```


arithmetic.py XIV

```
233         q[k] = self.f[k] * q[k - 1] + q[k - 2]
234
235     return p, q
236
237
238     def eval_(self):
239         '''Return the evaluation of self.f.'''
240
241         p, q = self.get_convergents()
242
243         return p[len(self.f) - 1] / q[len(self.f) - 1]
244
245
246     def get_nth_convergent(self, n):
247         '''Return the convergent at the index n.'''
248
249         if n >= len(self.f):
```

arithmetic.py XV

```
250         raise ValueError(f'ContinuedFraction:
get_nth_convergent: n cannot be greater than {len(self.f
) - 1}')
251
252     p, q = self.get_convergents()
253
254     return p[n] / q[n]
255
256
257
258 def get_continued_fraction(a, b):
259     '''Return a ContinuedFraction object, the continued
fraction of a/b.'''
260
261     f = []
262     d = Fraction(a, b)
263     f.append(floor(d))
264
265     while d - floor(d) != 0:
```

arithmetic.py XVI

```
266         d = 1/(d - floor(d))
267         f.append(floor(d))
268
269     return ContinuedFraction(f)
270
271
272 def get_continued_fraction_real(x):
273     '''
274     Return a ContinuedFraction object, the continued
275     fraction of x.
276     Note that there can be errors because of the float
277     precision with this function.
278     '''
279
280     f = []
281
282     d = x
283     f.append(floor(x))
```

arithmetic.py XVII

```
283 while d - floor(d) != 0:
284     d = 1/(d - floor(d))
285     f.append(floor(d))
286
287 return ContinuedFraction(f)
288
289
290 def get_continued_fraction_rec(a, b, f=[]):
291     '''Return a ContinuedFraction object, the continued
292     fraction of a/b. This is a recursive function.'''
293
294     # euclidean division :  $a = bq + r$ 
295     q = a // b
296     r = a % b
297
298     if r == 0:
299         return ContinuedFraction(f + [q])
300
301     return get_continued_fraction_rec(b, r, f + [q])
```

arithmetic.py XVIII

```
301 |
302 |
303 | ## - Tests
304 | if __name__ == '__main__':
305 |     if False:
306 |         n = int(input('number :\n>'))
307 |         p, q = fermat_factor(n)
308 |         print('Result : {}\n p = {}'.format(p*q == n, p))
```

base.py |

```
1 | #!/bin/python3
2 | # -*- coding: utf-8 -*-
3 |
4 | '''Miscellaneous and useful functions'''
5 |
6 | ##-Imports
7 | import hashlib
8 |
9 |
10 | ##-Split function
11 | def split(txt, size, pad_=None):
12 |     '''
13 |         Return a list representing txt by groups of size 'size'.
14 |
15 |         - txt : the text to split ;
16 |         - size : the block size ;
17 |         - pad_ : if not None, pad the last block with 'pad_' to
18 |         be 'size'length (adding to the end).
19 |     '''
```

base.py II

```
19
20     l = []
21
22     for k in range(len(txt) // size + 1):
23         p = txt[k*size : (k+1)*size]
24
25         if p in (',', b','):
26             break
27
28         if pad_ != None:
29             p = pad(p, size, pad_)
30
31         l.append(p)
32
33     return l
34
35
36 def pad(txt, size, pad=', ', end=True):
37     '''
```

base.py III

```
38 | Pad 'txt' to make it 'size' long.  
39 | If len(txt) > size, it just returns 'txt'.  
40 |  
41 | - txt : the string to pad ;  
42 | - size : the final wanted size ;  
43 | - pad : the character to use to pad ;  
44 | - end : if True, add to the end, otherwise add to the  
45 | beginning.  
46 | '''  
47 | while len(txt) < size:  
48 |     if end:  
49 |         txt += pad  
50 |  
51 |     else:  
52 |         txt = pad + txt  
53 |  
54 | return txt  
55 |
```


base.py IV

```
56 |
57 | ##-Mask generation function
58 | # From https://en.wikipedia.org/wiki/
   | Mask_generation_function
59 | def i2osp(integer: int, size: int = 4) -> str:
60 |     return int.to_bytes(integer % 256**size, size, 'big')
61 |
62 | def mgf1(input_str: bytes, length: int, hash_func=hashlib.
   | sha256) -> str:
63 |     '''Mask generation function.'''
64 |
65 |     counter = 0
66 |     output = b''
67 |     while len(output) < length:
68 |         C = i2osp(counter, 4)
69 |         output += hash_func(input_str + C).digest()
70 |         counter += 1
71 |
72 |     return output[:length]
```

base.py V

```
73
74
75 ##-Xor
76 def xor(s1, s2):
77     '''Return s1 xored with s2 bit per bit.'''
78
79     if (len(s1) != len(s2)):
80         raise ValueError('Strings are not of the same length
81         .')
82
83     if type(s1) != bytes:
84         s1 = s1.encode()
85
86     if type(s2) != bytes:
87         s2 = s2.encode()
88
89     l = [i ^ j for i, j in zip(list(s1), list(s2))]
90
91     return bytes(l)
```

base.py VI

```
91
92
93  ##-Int and bytes
94  def int_to_bytes(x: int) -> bytes:
95      return x.to_bytes((x.bit_length() + 7) // 8, 'little')
96
97  def bytes_to_int(xbytes: bytes) -> int:
98      return int.from_bytes(xbytes, 'little')
99
100
101  ##-Other
102  def str_diff(s1, s2, verbose=True, max_len=80):
103      '''
104      Show difference between strings (or numbers) s1 and s2.
105      Return s1 == s2.
106
107      - s1      : input string to compare ;
108      - s2      : output string to compare ;
```

base.py VII

```
108 |     - verbose : if True, show input and output message and
109 |     where they differ if so ;
110 |     - max_len : don't show messages if their length is more
111 |     than max_len. Default is 80. If negative, always show
112 |     them.
113 |     '''
114 |
115 |     s1 = str(s1)
116 |     s2 = str(s2)
117 |
118 |     if verbose:
119 |         if len(s1) <= max_len or max_len == -1:
120 |             print(f'\nEntry message : {s1}')
121 |             print(f'Output           : {s2}')
122 |
123 |         for k in range(len(s1)):
124 |             if s1[k] != s2[k]:
125 |                 if len(s1) <= max_len or max_len == -1:
```

base.py VIII

```
123         print(' '*(len('Output          : ') + k)
+ '~')
124
125         print('Input and output differ from position
{}.'.format(k))
126
127         return False
128
129         print('Input and output are identical.')
130
131     return s1 == s2
132
133
134 ##-Testing
135 if __name__ == '__main__':
136     msg = input('msg\n>').encode()
137
138     print(mgf1(msg, 10).hex())
139     print(xor('test', 'abcd'))
```

base.py IX

RSA.py |

```
1 | #!/bin/python3
2 | # -*- coding: utf-8 -*-
3 |
4 | '''Implementation of RSA cipher and key management'''
5 |
6 | ##-Imports
7 | try:
8 |     from arithmetic import *
9 |     from base import *
10 |
11 | except ModuleNotFoundError:
12 |     from modules.arithmetic import *
13 |     from modules.base import *
14 |
15 | from secrets import randbits
16 | from random import randint, randbytes
17 | import math
18 |
19 | import base64
```

RSA.py II

```
20 |
21 | ## - RsaKeys
22 | class RsaKey:
23 |     '''RSA key object'''
24 |
25 |     def __init__(self, e=None, d=None, n=None, phi=None, p=
26 |     None, q=None):
27 |         '''
28 |         - e : public exponent
29 |         - d : private exponent
30 |         - n : modulus
31 |         - p, q : primes that verify pq = n
32 |         - phi = (p - 1)(q - 1)
33 |         '''
34 |
35 |         self.e = e
36 |         self.d = d
37 |         self.n = n
38 |         self.phi = phi
```


RSA.py III

```
38         self.p = p
39         self.q = q
40
41         self.is_private = self.d != None
42
43         if self.is_private:
44             if self.q < self.p:
45                 self.p = q
46                 self.q = p
47
48         self.pb = (e, n)
49         if self.is_private:
50             self.pv = (d, n)
51
52         self.size = None
53
54     def __repr__(self):
55         if self.is_private:
```

RSA.py IV

```
56         return f'RsaKey private key :\n\tsize : {self.  
size}\n\te : {self.e}\n\td : {self.d}\n\tn : {self.n}\n\tphi : {self.phi}\n\tp : {self.p}\n\tq : {self.q}'  
57  
58     else:  
59         return f'RsaKey public key :\n\tsize : {self.  
size}\n\te : {self.e}\n\tn : {self.n}'  
60  
61  
62     def __eq__(self, other):  
63         '''Return True if the key are of the same type (  
public / private) and have the same values.'''  
64  
65         ret = self.is_private == other.is_private  
66  
67         if not ret:  
68             return False  
69  
70         if self.is_private:
```

RSA.py V

```
71         ret = ret and (  
72             self.e == other.e and  
73             self.d == other.d and  
74             self.n == other.n and  
75             self.phi == other.phi  
76         )  
77  
78         ret = ret and ((self.p == other.p and self.q ==  
other.q) or (self.q == other.p and self.p == other.q))  
79  
80     else:  
81         ret = ret and (  
82             self.e == other.e and  
83             self.n == other.d  
84         )  
85  
86     return ret  
87  
88
```

RSA.py VI

```
89     def public(self):
90         '''Return the public key associated to self in an
91         other RsaKey object.'''
92
93         k = RsaKey(e=self.e, n=self.n)
94         k.size = self.size
95         return k
96
97     def _gen_nb(self, size=2048, wiener=False):
98         '''
99         Generates p, q, and set attributes p, q, phi, n,
100         size.
101
102         - size      : the bit size of n ;
103         - wiener    : If True, generates p, q prime such that q
104         < p < 2q.
105         '''
```

RSA.py VII

```
105         self.p, self.q = 1, 1
106
107         while not isSurelyPrime(self.q):
108             self.q = randbits(size // 2)
109
110         while not (isSurelyPrime(self.p) and ((wiener and
self.q < self.p < 2 * self.q) or (not wiener))):
111             self.p = randbits(size // 2)
112
113         self.phi = (self.p - 1) * (self.q - 1)
114         self.n = self.p * self.q
115
116         self.size = size
117
118
119     def new(self, size=2048):
120         '''
121         Generate RSA keys of size 'size' bits.
```

RSA.py VIII

```
122     If self.e != None, it keeps it (and ensures that gcd  
123     (phi, e) = 1).  
124     - size : the key size, in bits.  
125     '''  
126  
127     self._gen_nb(size)  
128  
129     while self.e != None and math.gcd(self.e, self.phi)  
130     != 1:  
131         self._gen_nb(size)  
132  
133     if self.e == None:  
134         self.e = 0  
135         while math.gcd(self.e, self.phi) != 1:  
136             self.e = randint(max(self.p, self.q), self.  
phi)
```

RSA.py IX

```
137         elif math.gcd(self.e, self.phi) != 1: #Not possible
138             !
139             raise ValueError('RsaKey: new: error: gcd(self.e
140             , self.phi) != 1')
141
142         self.d = mult_inverse(self.e, self.phi)
143
144         self.is_private = True
145
146         self.pb = (self.e, self.n)
147         self.pv = (self.d, self.n)
148
149         self.size = size
150
151     def new_wiener(self, size=2048):
152         '''
153         Generate RSA keys of size 'size' bits.
154         If self.e != None, it does NOT keeps it.
```

RSA.py X

```
154 |         These key are generated so that the Wiener's attack  
    |         is possible on them.  
155 |  
156 |         - size : the key size, in bits.  
157 |         '''  
158 |  
159 |         self._gen_nb(size, wiener=True)  
160 |  
161 |         self.d = 0  
162 |         while math.gcd(self.d, self.phi) != 1:  
163 |             self.d = randint(1, math.floor(isqrt(isqrt(self.  
    | n)))/3))  
164 |  
165 |         self.e = mult_inverse(self.d, self.phi)  
166 |  
167 |         self.is_private = True  
168 |  
169 |         self.pb = (self.e, self.n)  
170 |         self.pv = (self.d, self.n)
```


RSA.py XI

```
171 |
172 |         self.size = size
173 |
174 |
175 |     def new_wiener_large(self, size=2048, only_large=True):
176 |         '''
177 |         Same as 'self.new_wiener', but 'd' can be very large
178 |         .
179 |         - size          : the RSA key size ;
180 |         - only_large   : if False, d can be small, or large,
181 |         and otherwise, d is large.
182 |         '''
183 |
184 |         self._gen_nb(size, wiener=True)
185 |
186 |         self.d = 0
187 |         while math.gcd(self.d, self.phi) != 1:
188 |             if only_large:
```

RSA.py XII

```
188         #ceil(sqrt(6)) = 3
189         self.d = randint(int(self.phi - iroot(self.n
, 4) // 3), self.phi)
190
191         else:
192             self.d = randint(1, self.phi)
193             if iroot(self.n, 4) / 3 < self.d or self.d <
self.phi - iroot(self.n, 4) / math.sqrt(6):
194                 self.d = 0 #go to the next iteration
195
196         self.e = mult_inverse(self.d, self.phi)
197         self.is_private = True
198         self.pb = (self.e, self.n)
199         self.pv = (self.d, self.n)
200
201         self.size = size
202
203
204
```

RSA.py XIII

```
205 ## - Padding
206 class OAEP:
207     '''Class implementing OAEP padding'''
208
209     def __init__(self, block_size, k0=None, k1=0):
210         '''
211         Initiate OAEP class.
212
213         - block_size      :      the bit size of each block ;
214         - k0                :      integer (number of bits in the
random part). If None, it is set to block_size // 8 ;
215         - k1                :      integer such that len(block) +
k0 + k1 = block_size. Default is 0.
216         '''
217
218         self.block_size = block_size #n
219
220         if k0 == None:
221             k0 = block_size // 8
```

RSA.py XIV

```
222
223     self.k0 = k0
224     self.k1 = k1
225
226
227     def _encode_block(self, block):
228         '''
229         Encode a block.
230
231         - block : an n - k0 - k1 long bytes string.
232         '''
233
234         #---Add k1 \0 to block
235         block += (b'\0')*self.k1
236
237         #---Generate r, a k0 bits random string
238         r = randbytes(self.k0)
239
240         X = xor(block, mgf1(r, self.block_size - self.k0))
```

RSA.py XV

```
241         Y = xor(r, mgf1(X, self.k0))
242
243     return X + Y
244
245
246
247 def encode(self, txt):
248     '''
249     Encode txt
250
251     Entry :
252         - txt : the string text to encode.
253
254     Output :
255         bytes list
256     '''
257
258     if type(txt) != bytes:
259         txt = txt.encode()
```

RSA.py XVI

```
260
261     #---Cut message in blocks of size n - k0 - k1
262     blocks = []
263     l = self.block_size - self.k0 - self.k1
264
265     blocks = split(txt, l, pad_=b'\0')
266
267     #---Encode blocks
268     enc = []
269     for k in blocks:
270         enc.append(self._encode_block(k))
271
272     return enc
273
274
275     def _decode_block(self, block):
276         '''Decode a block encoded with self._encode_block.
277         '''
```

RSA.py XVII

```
278     X = block[:self.block_size - self.k0]
279     Y = block[-self.k0:]
280
281     r = xor(Y, mgf1(X, self.k0))
282
283     txt = xor(X, mgf1(r, self.block_size - self.k0))
284
285     while txt[-1] == 0: #Remove padding
286         txt = txt[:-1]
287
288     return txt
289
290
291 def decode(self, enc):
292     '''
293     Decode a text encoded with self.encode.
294
295     - enc : a list of bytes encoded blocks.
296     '''
```

RSA.py XVIII

```
297
298     txt = b''
299
300     for k in enc:
301         txt += self._decode_block(k)
302
303     return txt
304
305
306
307 # - RSA
308 class RSA:
309     '''RSA cipher'''
310
311     def __init__(self, key, padding, block_size=None):
312         '''
313         - key          : a RsaKey object ;
314         - padding      : the padding to use. Possible values
315         are :
```


RSA.py XIX

```

315         'int' : msg is an int, return an int ;
316         'raw' : msg is a string, simply cut it in blocks
        ;
317         'oaep' : OAEP padding ;
318         - block_size : the size of encryption blocks. If
None, it is set to 'key.size // 8 - 1'.
319         '''
320
321     self.pb = key.pb
322     if key.is_private:
323         self.pv = key.pv
324
325     self.is_private = key.is_private
326
327     if padding.lower() not in ('int', 'raw', 'oaep'):
328         raise ValueError('RSA: padding not recognized.')
329
330     self.pad = padding.lower()
331

```

RSA.py XX

```
332         if block_size == None:
333             self.block_size = key.size // 8 - 1
334
335         else:
336             self.block_size = block_size
337
338
339     def encrypt(self, msg):
340         '''
341         Encrypt 'msg' using the key given in init.
342         Redirect toward the right method (using the good
343         padding).
344
345         - msg      : The string to encrypt.
346         '''
347
348         if self.pad == 'int':
349             return self._encrypt_int(msg)
```

RSA.py XXI

```
350         elif self.pad == 'raw':
351             return self._encrypt_raw(msg)
352
353         else:
354             return self._encrypt_oaep(msg)
355
356
357     def decrypt(self, msg):
358         '''
359         Decrypt 'msg' using the key given in init, if it is
360         a private one. Otherwise raise a TypeError.
361         Redirect toward the right method (using the good
362         padding).
363         '''
364         if not self.is_private:
365             raise TypeError('Can not decrypt using a public
key.')
```

RSA.py XXII

```
366         if self.pad == 'int':
367             return self._decrypt_int(msg)
368
369         elif self.pad == 'raw':
370             return self._decrypt_raw(msg)
371
372         else:
373             return self._decrypt_oaep(msg)
374
375     def _encrypt_int(self, msg):
376         '''
377         RSA encryption in its simplest form.
378
379         - msg : an integer to encrypt.
380         '''
381
382         e, n = self.pb
```

RSA.py XXIII

```
385         return pow(msg, e, n)
386
387
388     def _decrypt_int(self, msg):
389         '''
390         RSA decryption in its simplest form.
391         Decrypt 'msg' using the key given in init if
392         possible, using the 'int' padding.
393
394         - msg : an integer.
395         '''
396
397         d, n = self.pv
398
399         return pow(msg, d, n)
400
401     def _encrypt_raw(self, msg):
402         '''
```

RSA.py XXIV

```
403 |         Encrypt 'msg' using the key given in init, using the  
|         'raw' padding.  
404 |  
405 |         - msg : The string to encrypt  
406 |         '''  
407 |  
408 |         e, n = self.pb  
409 |  
410 |         #---Encode msg  
411 |         if type(msg) != bytes:  
412 |             msg = msg.encode()  
413 |  
414 |         #---Cut message in blocks  
415 |         m_lst = split(msg, self.block_size)  
416 |  
417 |         #---Encrypt message  
418 |         enc_lst = []  
419 |         for k in m_lst:  
420 |             enc_lst.append(pow(bytes_to_int(k), e, n))
```

RSA.py XXV

```
421
422         return b' '.join([base64.b64encode(int_to_bytes(k))
423 for k in enc_lst])
424
425 def _decrypt_raw(self, msg):
426     '''Decrypt 'msg' using the key given in init if
427 possible, using the 'raw' padding'''
428
429     d, n = self.pv
430
431     enc_lst = [base64.b64decode(k) for k in msg.split(b'
432 '),)]
433
434     c_lst = []
435     for k in enc_lst:
436         c_lst.append(pow(bytes_to_int(k), d, n))
437
438     txt = b''
```

RSA.py XXVI

```
437         for k in c_lst:
438             txt += int_to_bytes(k)
439
440         return txt.decode()
441
442
443     def _encrypt_oaep(self, msg):
444         '''Encrypt 'msg' using the key given in init, using
445         the 'oaep' padding.'''
446
447         e, n = self.pb
448
449         if type(msg) != bytes:
450             msg = msg.encode()
451
452         # ---Padding
453         E = OAEP(self.block_size)
454         m_lst = E.encode(msg)
```


RSA.py XXVII

```
455     #---Encrypt message
456     enc_lst = []
457     for k in m_lst:
458         enc_lst.append(pow(bytes_to_int(k), e, n))
459
460     return b' '.join([base64.b64encode(int_to_bytes(k))
461 for k in enc_lst])
462
463 def _decrypt_oaep(self, msg):
464     '''Decrypt 'msg' using the key given in init if
465 possible, using the 'oaep' padding.'''
466
467     d, n = self.pv
468
469     #---Decrypt
470     enc_lst = [base64.b64decode(k) for k in msg.split(b'
471 ')]
472     c_lst = []
```

RSA.py XXVIII

```
471
472     for k in enc_lst:
473         c_lst.append(pow(bytes_to_int(k), d, n))
474
475     #---Decode
476     encoded_lst = []
477     for k in c_lst:
478         encoded_lst.append(pad(int_to_bytes(k), self.
block_size, b'\0'))
479
480     E = OAEP(self.block_size)
481
482     return E.decode(encoded_lst)
483
484
485 ##-Testing
486 if __name__ == '__main__':
487     from tests import test_OAEP, test_RSA, dt
488     from sys import argv, exit as sysexit
```

RSA.py XXIX

```
489
490     if len(argv) == 1:
491         size = 2048
492
493     else:
494         try:
495             size = int(argv[1])
496
497         except:
498             print(f'Wrong argument at position 1 : should be
499                 the RSA key size (in bits).\nExample : "{argv[0]}
500                 2048".')
501             sys.exit()
502
503     t0 = dt.now()
504     print('Generating a key (for all the tests) ...')
505     k = RsaKey()
506     k.new(size)
507     print('Done.')
```

RSA.py XXX

```
506 |
507 |     test_OAEP(size // 8 - 1)
508 |     print(f'\n--- {dt.now() - t0}s elapsed.\n')
509 |     test_RSA(k, 'int', size)
510 |     print(f'\n--- {dt.now() - t0}s elapsed.\n')
511 |     test_RSA(k, 'raw', size)
512 |     print(f'\n--- {dt.now() - t0}s elapsed.\n')
513 |     test_RSA(k, 'oaep', size)
514 |     print(f'\n--- {dt.now() - t0}s elapsed.\n')
```

RSA_attacks.py |

```
1 | #!/bin/python3
2 | # -*- coding: utf-8 -*-
3 |
4 | '''Implementation of RSA attacks'''
5 |
6 | ##-Imports
7 | try:
8 |     from arithmetic import *
9 |     import RSA
10 |
11 | except ModuleNotFoundError:
12 |     from modules.arithmetic import *
13 |     import modules.RSA as RSA
14 |
15 | import math
16 | from random import randint
17 |
18 | from datetime import datetime as dt
19 |
```

RSA_attacks.py II

```
20 |
21 | ##-Elementary attacks
22 | #-----Elementary attacks
23 | #---Factor modulus with private key
24 | def factor_with_private(e, d, n, max_tries=None):
25 |     '''
26 |     Factor modulus n using public and private exponent e and
27 |     d.
28 |
29 |     - max_tries : stop after 'max_tries' tries if not found
30 |     before. If None, don't stop until found.
31 |     '''
32 |
33 |     k = e*d - 1
34 |     t, r = max_parity(k) # k = 2^t * r, r is odd.
35 |
36 |     i = 0
37 |     while True:
38 |         g = 0
```

RSA_attacks.py III

```
37     while math.gcd(g, n) != 1: # find a g in  $(\mathbb{Z}/n\mathbb{Z})^*$ 
38         g = randint(2, n - 1)
39
40     for j in range(t, 1, -1): # Try with  $g^{(k / 2^j)}$ 
41         x = pow(g, k // (2**j), n)
42         y = math.gcd(x - 1, n)
43
44         if n % y == 0 and (y not in (1, n)):
45             return y, n//y
46
47     if max_tries != None:
48         i += 1
49         if i >= max_tries:
50             return None
51
52
53 #--- Common modulus
54 def common_modulus(N, e, d, e1):
55     '''
```

RSA_attacks.py IV

```
56     Entry :
57         - N : the common modulus ;
58         - e : the known public exponent ;
59         - d : the known private exponent ;
60         - e1 : public exponent associated to the wanted
           private exponent.
61
62     Calculate d1 the private exponent associated to e1.
63     '''
64
65     p, q = factor_with_private(e, d, N)
66     phi = (p - 1) * (q - 1)
67
68     return mult_inverse(e1, phi)
69
70
71 ---Multiplicative attack
72 def multiplicative_attack(m_, r, n):
73     '''
```


RSA_attacks.py V

```

74 | Uses the fact that the product of two ciphertexts is
75 | equal to the ciphertext of the product.
76 |
77 | We have  $c = m^e [n]$  and we want  $m$ .
78 | We ask for the decryption of  $c_ = c * r^e [n]$  ( $m_$ ).
79 |
80 | -  $m_$  : the decryption of  $c_ = c * r^e [n]$  ;
81 | -  $r$  : the number used to obfuscate the initial message
82 | ;
83 | -  $n$  : the modulus.
84 | '''
85 |
86 | inv_r = mult_inverse(r, n)
87 |
88 | return (m_ * inv_r) % n
89 |
90 | #-----Large message (close to n)
91 | def large_message(c, e, n):

```

RSA_attacks.py VI

```

91 |     '''
92 |     Return the decryption of c using the method from Hinek's
93 |     paper (cacr2004).
94 |     In order for this attack to work, we need to have
95 |          $n - n^{(1/e)} < m < n$ 
96 |     Then we have :
97 |          $m = n - (-c \% n)^{(1/e)}$ .
98 |
99 |     Arguments :
100 |         - c : the encryption of m :  $c = m^e [n]$  ;
101 |         - e : the public exponent ;
102 |         - n : the RSA modulus.
103 |     '''
104 |
105 |     return n - iroot(-c % n, e)
106 |
107 | ##-Hastad
108 | #---Hastad (same message)

```

RSA_attacks.py VII

```
109 | def _hastad(e, enc_msg_lst, mod_lst):
110 |     '''
111 |     Return (me, e, M). The decrypted message is 'iroot(me, e
112 |     )' or 'large_message(me, e, M)' (if the message was very
113 |     long).
114 |
115 |     - e           : the common public exponent ;
116 |     - enc_msg_lst : the list of the encrypted messages ;
117 |     - mod_lst     : the list of modulus.
118 |
119 |     The lists 'enc_msg_lst' and 'mod_lst' should have the
120 |     same length.
121 |     '''
122 |     M = 1
123 |     for k in mod_lst:
124 |         M *= k
```

RSA_attacks.py VIII

```
124     me = sum(enc_msg_lst[k] * (M // mod_lst[k]) *
125             mult_inverse(M // mod_lst[k], mod_lst[k]) for k in range
126             (len(mod_lst))) % M
127
128
129     return (me, e, M)
130
131 def hastad(e, enc_msg_lst, mod_lst):
132     '''
133     Return the decrypted message.
134
135     - e           : the common public exponent ;
136     - enc_msg_lst : the list of the encrypted messages ;
137     - mod_lst     : the list of modulus.
138
139     The lists 'enc_msg_lst' and 'mod_lst' should have the
140     same length.
141     '''
```

RSA_attacks.py IX

```
140     me, e = _hastad(e, enc_msg_lst, mod_lst)[: -1]
141
142     return iroot(me, e)
143
144
145 def hastad_large_message(e, enc_msg_lst, mod_lst):
146     '''
147     Return the decrypted message.
148
149     - e                : the common public exponent ;
150     - enc_msg_lst     : the list of the encrypted messages ;
151     - mod_lst         : the list of modulus.
152
153     The lists 'enc_msg_lst' and 'mod_lst' should have the
154     same length.
155     '''
156     me, e, M = _hastad(e, enc_msg_lst, mod_lst)
157
```

RSA_attacks.py X

```
158     return large_message(me, e, M)
159
160
161 ##-Wiener's attack
162 def factor_with_phi(n, phi):
163     '''
164     Return (p, q) such that  $n = pq$ , if possible. Otherwise,
165     raise a ValueError
166
167     - n : the RSA modulus ;
168     - phi : the Euler totien of n :  $\phi = (p - 1)(q - 1)$ .
169
170     It solve the quadratic
171          $x^2 - (n - \phi + 1)x + n = 0$ 
172     '''
173     delta = (n - phi + 1)**2 - 4*n
174
175     if delta < 0:
```

RSA_attacks.py XI

```
176         raise ValueError('Wrong modulus or wrong phi.')
177
178     p = (n - phi + 1 - isqrt(delta)) // 2
179     q = (n - phi + 1 + isqrt(delta)) // 2
180
181     if p * q != n:
182         raise ValueError('Wrong modulus or wrong phi.')
183
184     return p, q
185
186
187 def wiener(e, n):
188     '''
189     Run Wiener's attack on the public key (e, n).
190     Return a private RsaKey object.
191
192     Can factor the key if the private exponent d is such
193     that
194          $1 < d < n^{(1/4)} / 3$ 
```

RSA_attacks.py XII

```
194         or
195          $\phi - n^{(1/4)}/\text{sqrt}(6) < d < \phi$ 
196
197     - e : the public exponent ;
198     - n : the modulus.
199     '''
200
201     #---Calculate the continued fraction of e/n
202     e_n_frac = get_continued_fraction(e, n)
203
204     #---Calculate the convergents
205     k_, d_ = e_n_frac.get_convergents()
206
207     #---Compute phi to check correctness
208     for i in range(1, len(k_) - 2):
209         phi = (e * d_[i] - 1) // k_[i]
210         phi2 = (e * d_[i] + 1) // k_[i] #With large private
211         exponent.
```


RSA_attacks.py XIII

```
212         try:
213             p, q = factor_with_phi(n, phi)
214
215         except ValueError:
216             try:
217                 p2, q2 = factor_with_phi(n, phi2)
218
219             except ValueError:
220                 continue
221
222             else: #Correct factorisation with p2, q2
223                 key = RSA.RsaKey(e, phi2 - d_[i], n, phi2,
p2, q2)
224                 return key
225
226             else: #Correct factorisation with p, q
227                 key = RSA.RsaKey(e, d_[i], n, phi, p, q)
228                 return key
229
```

RSA_attacks.py XIV

```
230 ||     raise ValueError('The attack failed with this key')
```

test_attacks.py |

```
1 | #!/bin/python3
2 | # -*- coding: utf-8 -*-
3 |
4 | '''Tests for RSA attacks'''
5 |
6 | ##-imports
7 | try:
8 |     from RSA_attacks import *
9 |     from base import str_diff, int_to_bytes, bytes_to_int
10 |
11 | except ModuleNotFoundError:
12 |     from modules.RSA_attacks import *
13 |     from modules.base import str_diff, int_to_bytes,
14 |         bytes_to_int
15 |
16 | from secrets import randbits
17 | from random import randint
18 |
19 | ##-Fermat factorisation
```

test_attacks.py II

```
19 def test_fermat_factor(size=2048, dist=512):
20     '''
21     Tests the Fermat factorisation : generates two primes p,
22     q and test the algorithm on it.
23
24     - size : the size of the modulus to generate in bits, i.
25     e of p*q ;
26     - dist : the bit size of |p - q| ;
27     '''
28
29     print('Prime generation ...')
30     t0 = dt.now()
31
32     p = 1
33     while not isSurelyPrime(p):
34         p = randbits(size // 2)
35
36     q = p + 2**dist
37     while not isSurelyPrime(q):
```

test_attacks.py III

```
36         q += 1
37
38         print(f'Generation done in {dt.now() - t0}s.\nq : {round(
39             (math.log2(p), 2)} bits\nq : {round(math.log2(q), 2)}
40             bits\n|p - q| : {round(math.log2(q - p), 2)} bits\n2 * |
41             p - q|^(1/4) : {round(math.log2(2 * iroot(p * q, 4)), 2)
42             }')
43
44         b = q - p <= 2 * iroot(p * q, 4)
45
46         print('\nFactorisation ...')
47         t1 = dt.now()
48         a, b = ferat_factor(p * q)
49         print(f'Factorisation done in {dt.now() - t1}s.')
50
51         if a * b != p * q:
52             print('Factorisation failed : the product of the
53             result is not p * q.')
54             return False
```

test_attacks.py IV

```
50
51     if not (p in (a, b) and q in (a, b)):
52         print('Factorisation failed : p or q not in the
53             result.')
```

```
54         return False
55
56     print('Good factorisation.')
```

```
57     return True
58
59
60 ##-Modulus factorisation
61 def test_mod_fact(size=2048):
62     print('Key generation ...')
```

```
63     t0 = dt.now()
64     key = RSA.RsaKey()
65     key.new(size)
66     print(f'Generation done in {dt.now() - t0}s.')
```

```
67
```

test_attacks.py ✓

```
68     t1 = dt.now()
69     try:
70         p, q = factor_with_private(key.e, key.d, key.n)
71
72     except TypeError:
73         print('not found !')
74         return False
75
76     else:
77         print('Found in {}. \nCorrect : n == pq : {}, key.p
78         in (p, q) : {}'.format(dt.now() - t1, key.n == p*q, key.
79         p in (p, q)))
80         return True
81
82         # for k in (key.p, key.q, p, q):
83         #     print(k)
84
85     ##- Common modulus
86 def test_common_mod(size=2048):
```

test_attacks.py VI

```
85     print('Key generation ...')
86     t0 = dt.now()
87     key = RSA.RsaKey()
88     key.new(size)
89     print(f'Generation done in {dt.now() - t0}s.')
90
91     t1 = dt.now()
92     e1 = 0
93     while math.gcd(e1, key.phi) != 1:
94         e1 = randint(max(key.p, key.q), key.phi)
95
96     print(f'Generation of e1 done in {dt.now() - t1}s.')
97
98     t2 = dt.now()
99     d1 = mult_inverse(e1, key.phi)
100    if common_modulus(key.n, key.e, key.d, e1) == d1:
101        print(f'Attack succeeded : private exposant
recovered.\nDone in {dt.now() - t2}s.')
102    return True
```


test_attacks.py VII

```
103     else:
104         print(f'Attack failed : private exposant NOT
105             recovered.\nDone in {dt.now() - t2}s.')
```

```
106
107
108 ##-Test Multiplicative attack
109 def test_multiplicative_attack_one_block(m=None, size=2048):
110     '''
111     Test multiplicative_attack.
112
113     - m      : the message (int). If None, generates a random
114     one ;
115     - size  : the RSA key size.
116     '''
117     t0 = dt.now()
118     print('Key generation ...')
```

```
119     key = RSA.RsaKey()
```

test_attacks.py VIII

```
120     key.new(size)
121
122     n = key.n
123     e = key.e
124     d = key.d
125
126     print(f'Done in {dt.now() - t0}s')
127
128     if m == None:
129         m = randint(1, n - 1)
130
131     c = pow(m, e, n)
132
133     t1 = dt.now()
134     print('Running the attack ...')
135     r = randint(2, n - 1)
136
137     if math.gcd(r, n) != 1: # To ensure that r is inversible
        modulo n
```

test_attacks.py IX

```
138     p = math.gcd(r, n)
139     q = n // p
140     print(f'We accidentally factorized n ...\nn = {n}\np
= {p}\nq = {q}.\nn == p*q : {n == p * q}.'))
141     return n == p * q
142
143     c_ = (c * pow(r, e, n)) % n #obfuscated encrypted
message
144     m_ = pow(c_, d, n) #The inoffensive looking message (
obfuscated) gently decrypted by Alice
145
146     recov_m = multiplicative_attack(m_, r, n)
147
148     print(f'Attack done in {dt.now() - t1}s.')
149
150     if recov_m == m:
151         print('Attack successful')
152         return True
153
```

test_attacks.py X

```
154     else:
155         print('Attack failed')
156         return False
157
158
159 def test_multiplicative_attack(m=None, size=2048):
160     '''
161     Test multiplicative_attack.
162
163     - m      : the message (int). If None, generates a random
164               one ;
165     - size  : the RSA key size.
166     '''
167     t0 = dt.now()
168     print('Key generation ...')
169     key = RSA.RsaKey()
170     key.new(size)
171
```

test_attacks.py XI

```
172 | n = key.n
173 | e = key.e
174 | d = key.d
175 |
176 | print(f'Done in {dt.now() - t0}s')
177 |
178 | if m == None:
179 |     m = randint(1, n - 1)
180 |
181 | E = RSA.OAEP(key.size // 8 - 1)
182 | m_e = [bytes_to_int(k) for k in E.encode(int_to_bytes(m)
183 | )] #message encoded in blocks
184 | enc_lst = [pow(k, e, n) for k in m_e] #The ciphertexts
185 |
186 | t1 = dt.now()
187 | print('Running the attack ...')
188 | r_lst = [randint(2, n - 1) for k in range(len(m_e))] #
189 |     choose one r per block
```

test_attacks.py XII

```
189     for r in r_lst:
190         if math.gcd(r, n) != 1: # To ensure that all r are
            inversible modulo n
191             p = math.gcd(r, n)
192             q = n // p
193
194             print(f'We accidentally factorized n ...\nn = {n
}\np = {p}\nq = {q}.\nn == p*q : {n == p * q}.'))
195
196             return n == p * q
197
198     enc_lst_r = [(c_k * pow(r_k, e, n)) % n for (c_k, r_k)
in zip(enc_lst, r_lst)] #List of obfuscated encrypted
messages
199
200     dec_lst = [pow(k, d, n) for k in enc_lst_r] #The
inoffensive looking messages (obfuscated) gently
decrypted by Alice
201
```

test_attacks.py XIII

```
202     recov_lst = [multiplicative_attack(m_k, r_k, n) for (m_k
    , r_k) in zip(dec_lst, r_lst)]
203
204     decoded = E.decode([int_to_bytes(k) for k in recov_lst])
205
206     print(f'Attack done in {dt.now() - t1}s.')
```

```
207
208     if bytes_to_int(decoded) == m:
209         print('Attack successful')
210         return True
211
212     else:
213         print('Attack failed')
214         return False
215
216
217 ##-Test large positive numbers
218 def test_large_message(e=3, size=2048, verbose=False):
219     '''
```

test_attacks.py XIV

```

220 Cf cacr2004 (Hinek) paper.
221 Generates an RSA key, and a message m such that  $n - n^{1/e} < m < n$ 
222 Then encrypt it :  $c = m^e \pmod n$ 
223 It is possible to recover the message :
224  $m = n - (-c \% n)^{1/e}$ 
225 '''
226
227 print('Generating RSA key ...')
228 t0 = dt.now()
229
230 k = RSA.RsaKey(e = e)
231 k.new(size)
232 print(f'Generation done in {dt.now() - t0}s.')
233
234 print('Generating message and encrypting it ...')
235 t1 = dt.now()
236 m = randint(k.n - iroot(k.n, e), k.n)
237 c = pow(m, e, k.n)

```


test_attacks.py XV

```
238     print(f'Done in {dt.now() - t1}s.')
```

```
239
```

```
240     print('Recovering the message ...')
```

```
241     t2 = dt.now()
```

```
242     m_recov = large_message(c, k.e, k.n)
```

```
243     print(f'Message recovered in {dt.now() - t2}s.')
```

```
244
```

```
245     if str_diff(str(m), str(m_recov), verbose=verbose,
246               max_len=-1):
247         print(f'Attack successful. Done in {dt.now() - t0}s.
248               ')
249         return True
250     else:
251         print(f'Attack failed. Time elapsed : {dt.now() - t0
252               }s.')
```

```
253         return False
```

test_attacks.py XVI

```

254  ## - Hastad
255  def test_hastad(msg = 'testing', e=3, size=2048, nb_eq=None,
256                try_large=False):
257      '''
258      Tests the 'hastad' function.
259
260      - msg          : the message that will be encrypted with
261                      RSA and be recovered ;
262      - e            : the public exponent used for all the keys
263                      ;
264      - size         : the size of the modulus ;
265      - nb_eq        : the number of equations. If None,
266                      calculate the right number using the message ;
267      - try_large    : bool indicating if trying to break the
268                      message using hastad_large_message.
269      '''
270
271      msg = int(''.join(format(ord(k), '03') for k in msg)) #
272      testing -> 116101115116105110103

```

test_attacks.py XVII

```
267
268     n = math.ceil(e * math.log2(msg) / size)
269     print(f'Number of equations actually needed to recover
the message : {n}.')
270
271     if nb_eq == None:
272         nb_eq = n
273
274     keys = [RSA.RsaKey(e=e) for k in range(nb_eq)]
275
276     print(f'\nKey generation for Hastad\'s attack ({size}
bits, {nb_eq} keys) ...')
277     t0 = dt.now()
278     for k in range(nb_eq):
279         t1 = dt.now()
280         keys[k].new(size)
281         print(f'{k + 1}/{nb_eq} generated in {dt.now() - t1}
s.')
```

test_attacks.py XVIII

```
283     print(f'Done in {dt.now() - t0}s.')
```

```
284
```

```
285     mod_lst = [keys[k].n for k in range(nb_eq)]
```

```
286     ciphers = [RSA.RSA(keys[k], 'int') for k in range(nb_eq)
```

```
                ]
```

```
287     enc_lst = [ciphers[k].encrypt(msg) for k in range(nb_eq)
```

```
                ]
```

```
288
```

```
289     print('\nHastad attack ...')
```

```
290     t2 = dt.now()
```

```
291     ret = hastad(e, enc_lst, mod_lst)
```

```
292     print(f'Attack done in {dt.now() - t2}s.')
```

```
293
```

```
294     dec_out = ''.join([chr(int(str(ret)[3*k : 3*k + 3])) for
```

```
                        k in range(len(str(ret)) // 3)])
```

```
295
```

```
296     if msg != ret and try_large:
```

test_attacks.py XIX

```
297     # This can't work because no message can fit in  $[M -$   
     $M^{(1/e)} ; M]$  : they would need to have exactly  $\text{len}(\text{str}(\text{M})) / k = \text{len}(\text{str}(M - \text{iroot}(M, e))) / k$  characters (where  $k$   
    is defined with the encoding used (here it is  $k = 3$ ) so  
    we need that  $k$  divide  $\text{len}(\text{str}(M))$  (thus that way it is  
    possible to find an int of this length that will thus  
    maybe correspond to an encoded message).  
298  
299     print('Attack failed, trying to use the large number  
    way ...')  
300  
301     M = 1  
302     for k in range(nb_eq):  
303         M *= keys[k].n  
304  
305     print(f'Is the condition good for large number  
    attack ? : {M - iroot(M, e) <= msg <= M}')  
306     if M - iroot(M, e) > msg:
```

test_attacks.py XX

```
307         print('Message is too small for the large
message attack.')
```

```
308
309         elif msg > M:
310             print('Message is too large for the large
message attack.')
```

```
311
312             t3 = dt.now()
313             ret2 = hastad_large_message(e, enc_lst, mod_lst)
314             print(f'Attack done in {dt.now() - t3}s.')
```

```
315
316             return str_diff(msg, ret2)
```

```
317
318         return str_diff(msg, ret)
```

```
319
320         #print(f'\nDecoded output :\n{dec_out}')
```

```
321
322  #-Test message size limit
```

```
323 def test_hastad_message_size(msg_size=100, e=3, size=2048):
```

test_attacks.py XXI

```
324 '''
325     Test the number size with the number of equations
326
327     - msg_size : the length of the message ;
328     - e       : the public exponent used for all the keys ;
329     - size    : the size of the modulus.
330 '''
331
332 msg = ''.join([chr(randint(65, 122)) for k in range(
333     msg_size)]) #Random chars
334 msg = int(''.join(format(ord(k), '03') for k in msg)) #
335     Encoding the message
336
337 n = math.ceil(e * math.log2(msg) / size)
338 print(f'Number of equations theoretically needed to
339     recover the message : {n}.')
```

test_attacks.py XXII

```
340     print(f'\nKey generation for Hastad\'s attack ({size}
bits) ...')
341     t0 = dt.now()
342     for k in range(n):
343         t1 = dt.now()
344         keys[k].new(size)
345         print(f'{k + 1}/{n} generated in {dt.now() - t1}s.')
346
347     print(f'Done in {dt.now() - t0}s.')
348
349     mod_lst = [keys[k].n for k in range(n)]
350     ciphers = [RSA.RSA(keys[k], 'int') for k in range(n)]
351     enc_lst = [ciphers[k].encrypt(msg) for k in range(n)]
352
353     print(f'\nHastad attack with {n} equations ...')
354     t2 = dt.now()
355     ret1 = hastad(e, enc_lst, mod_lst)
356     print(f'Attack done in {dt.now() - t2}s.')
357
```


test_attacks.py XXIII

```
358     if msg == ret1:
359         print('Attack succeeded : message correctly
recovered.')
```

```
360
361     else:
362         print('Attack failed : message NOT correctly
recovered.')
```

```
363         return False
364
365     if n - 1 == 0:
366         print('\nNot trying to with less equations than one.
')
```

```
367         return True
368
369     print(f'\nHastad attack with {n - 1} equations ...')
```

```
370     t3 = dt.now()
371     ret2 = hastad(e, enc_lst[:-1], mod_lst[:-1])
372     print(f'Attack done in {dt.now() - t3}s.')
```

```
373
```

test_attacks.py XXIV

```
374     if msg == ret2:
375         print('Attack succeeded : message correctly
recovered. So the limit is NOT correct.')
376         return False
377
378     else:
379         print('Attack failed : message not correctly
recovered. So the limit is correct.')
380         return True
381
382
383 def test_hastad_large_message(e=3, size=2048, less=0):
384     '''
385     Tests the 'hastad' function, with large message (see
Hinek's paper).
386
387     - e      : the public exponent used for all the keys ;
388     - size   : the size of the modulus ;
389     - less   : the number of equations to remove.
```

test_attacks.py XXV

```
390
391     But the problem with this is that it generates the
392     message after having M, which is not how it would be in
393     real life.
394     '''
395
396     keys = [RSA.RsaKey(e=e) for k in range(e)]
397
398     print(f'\nKey generation for Hastad\'s attack ({size}
399     bits) ...')
400     t0 = dt.now()
401     for k in range(e):
402         t1 = dt.now()
403         keys[k].new(size)
404         print(f'{k + 1}/{e} generated in {dt.now() - t1}s.')
405
406     print(f'Done in {dt.now() - t0}s.')
407
408     M = 1
```

test_attacks.py XXVI

```
406     for k in range(e):
407         M *= keys[k].n
408
409     msg = randint(M - iroot(M, e), M)
410     print('len(str(msg)) :', len(str(msg)), 'log2(M) :',
411           math.log2(M))
411     print(f'Number of equations actually needed to recover
412           the message (without large message idea) : {math.ceil(e
413           * math.log2(msg) / size)}.')
412     #print(f'msg : {msg}')
413
414     mod_lst = [keys[k].n for k in range(e - less)]
415     ciphers = [RSA.RSA(keys[k], 'int') for k in range(e -
416               less)]
416     enc_lst = [ciphers[k].encrypt(msg) for k in range(e -
417               less)]
417
418     print('\nHastad attack ...')
419     t2 = dt.now()
```

test_attacks.py XXVII

```
420     ret = hastad_large_message(e, enc_lst, mod_lst)
421     print(f'Attack done in {dt.now() - t2}s.')
```

```
422
423     if str_diff(str(msg), str(ret)):
424         return True
425
426     else:
427         return False
428
429
430 ##-Wiener
431 def test_wiener(size=2048, large=False,
432               not_in_good_condition=False):
433     '''
434     Test Wiener's attack.
435
436     - size           : The RSA key size ;
437     - large          : if True, generates a large
438     private exponent ;
```

test_attacks.py XXVIII

```
437     - not_in_good_condition : Do not try to generate a key  
438     that is breakable with this attack.  
439     '''  
440     key = RSA.RsaKey()  
441  
442     print(f'Key generation for Wiener\'s attack ({size} bits  
443           ) ...')  
444     t0 = dt.now()  
445     if not_in_good_condition:  
446         key.new()  
447  
448     elif large:  
449         key.new_wiener_large(size)  
450  
451     else:  
452         key.new_wiener(size)  
453  
454     print(f'Key generated in {dt.now() - t0}s.')
```

test_attacks.py XXIX

```
454
455     pb = key.public()
456
457     t1 = dt.now()
458     try:
459         recovered_key = wiener(pb.e, pb.n)
460
461     except ValueError as err:
462         print(f'Wiener\'s attack finished in {dt.now() - t1}
463 s.')
```

```
463         print(err)
464         return False
465
466     print(f'Wiener\'s attack finished in {dt.now() - t1}s.')
```

```
467
468     if recovered_key == key:
469         print('Correct result !')
470         return True
471
```

test_attacks.py XXX

```
472     else:
473         print('Incorrect result !')
474         return False
475
476 if __name__ == '__main__':
477     # test_wiener(large=True)
478     test_multiplicative_attack()
```


tests.py |

```
1 |#!/bin/python3
2 |# -*- coding: utf-8 -*-
3 |
4 |'''Tests'''
5 |
6 |##-Import
7 |try:
8 |    from base import *
9 |    from arithmetic import *
10 |    from RSA_attacks import *
11 |    from RSA import *
12 |    import test_attacks
13 |
14 |except ModuleNotFoundError:
15 |    from modules.base import *
16 |    from modules.arithmetic import *
17 |    from modules.RSA_attacks import *
18 |    from modules.RSA import *
19 |    import modules.test_attacks as test_attacks
```

tests.py II

```
20
21 from datetime import datetime as dt
22
23
24 ##-Test function
25 def tester(func_name, assertion):
26     '''Print what is tested and fail if the assertion failed
27     .'''
28
29     if assertion:
30         print(f'Testing {func_name}: passed')
31         return True
32
33     else:
34         print(f'Testing {func_name}: failed')
35         raise AssertionError
36
37 ##-Base
```

tests.py III

```
38 def test_base():
39     tester(
40         'base: split',
41         split('azertyuiopqsdfghjklmwxcvbn', 3) == ['aze', 'rty', 'uio', 'pqs', 'dfg', 'hjk', 'lmw', 'xcv', 'bn']
42     )
43     tester(
44         'base: split',
45         split('azertyuiopqsdfghjklmwxcvbn', 3, '0') == ['aze', 'rty', 'uio', 'pqs', 'dfg', 'hjk', 'lmw', 'xcv', 'bn0']
46     )
47
48
49 ##-Arithmetic
50 def test_arith(size=2048):
51     tester(
52         'arithmetic: mult_inverse',
```

tests.py IV

```
53     [mult_inverse(k, 7) for k in range(1, 7)] == [1, 4,
54     5, 2, 3, 6]
55     )
56     tester(
57         'arithmetic: max_parity',
58         max_parity(256) == (8, 1) and max_parity(123) == (0,
59         123) and max_parity(8 * 5) == (3, 5)
60     )
61     tester(
62         'arithmetic: isSurelyPrime',
63         (not isSurelyPrime(1)) and isSurelyPrime(2) and
64         isSurelyPrime(11) and isSurelyPrime(97) and (not
65         isSurelyPrime(561))
66     )
67     tester(
68         'arithmetic: iroot',
69         iroot(2, 2) == 1 and iroot(27, 3) == 3
70     )
71     print('Testing fermat_factor :')
```

tests.py ✓

```
68     tester(  
69         'arithmetic: fermat_factor',  
70         test_attacks.test_fermat_factor(size, size // 4)  
71     )  
72  
73  
74 ## - RSA  
75 def test_OAEP(size=2048):  
76     '''  
77     Test the OAEP padding scheme.  
78  
79     - size : the RSA key's size. The OAEP size is 'size // 8  
80     - 1'.  
81     '''  
82     # Using the LICENCE file as test file  
83     try:  
84         with open('LICENCE') as f:  
85             m = f.read()
```

tests.py VI

```
86
87     except FileNotFoundError:
88         with open('../LICENCE') as f:
89             m = f.read()
90
91     C = OAEP(size // 8 - 1)
92     e = C.encode(m)
93
94     tester('RSA: OAEP', m.encode() == C.decode(e))
95
96
97 def test_RSA(k=None, pad='raw', size=2048):
98     '''Test RSA encryption / decryption'''
99
100    print(f'Testing RSA (padding : {pad}).')
101
102    if k is None:
103        print('Generating a key ...', end=' ')
104        k = RsaKey()
```

tests.py VII

```
105         k.new(size=size)
106         print('Done.')
```

107

```
108     else:
109         size = k.size
```

110

```
111     C = RSA(k, pad)
```

112

```
113     if pad.lower() == 'int':
114         m = randint(0, k.n - 1)
```

115

```
116     else:
117         print('Reading file ...', end=' ')
118         # Using the LICENCE file as test file
119         try:
120             with open('LICENCE') as f:
121                 m = f.read()
```

122

```
123     except FileNotFoundError:
```

tests.py VIII

```
124         with open('../LICENCE') as f:
125             m = f.read()
126
127         print('Done.')
```



```
128
129     print('Encrypting ...', end=' ')
130     enc = C.encrypt(m)
131     print('Done.\nDecrypting ...', end=' ')
132     dec = C.decrypt(enc)
133
134     if pad.lower() == 'oaep':
135         # print(dec)
136         dec = dec.decode()
137
138     print('Done.')
```



```
139
140     tester(f'RSA: RSA (padding : {pad})', dec == m)
141
142
```


tests.py IX

```
143  ##-Run tests function
144  def run_tests(size=2048):
145      '''Run all the tests'''
146
147      t0 = dt.now()
148      test_base()
149      print(f'\n--- {dt.now() - t0}s elapsed.\n')
150      test_arith(size=size)
151      print(f'\n--- {dt.now() - t0}s elapsed.\n')
152
153      test_OAEP(size)
154      print(f'\n--- {dt.now() - t0}s elapsed.\n')
155
156      test_RSA(pad='int', size=size)
157      print(f'\n--- {dt.now() - t0}s elapsed.\n')
158      test_RSA(pad='raw', size=size)
159      print(f'\n--- {dt.now() - t0}s elapsed.\n')
160      test_RSA(pad='oaep', size=size)
161      print(f'\n--- {dt.now() - t0}s elapsed.\n')
```

tests.py X

```
162
163     print('All tests passed.') #Otherwise the function '
    tester' in the tests would have raised an AssertionError
    .
164
165
166 ##-Main
167 if __name__ == '__main__':
168     from sys import argv
169     from sys import exit as sysexit
170
171     if len(argv) == 1:
172         size = 2048
173
174     else:
175         try:
176             size = int(argv[1])
177
178         except:
```

tests.py XI

```
179 |         print(f'Wrong argument at position 1 : should be  
    |         the RSA key size (in bits).\nExample : "{argv[0]}"  
    |         2048".')
```

```
180 |         sys.exit()
```

```
181 |
```

```
182 | run_tests(size=size)
```