TP4 - Signature électronique RSA

(SignRSA) **Exercise 1.** *Signature RSA*

RSA est un algorithme asymétrique de chiffrement, qui permet par ailleurs de signer ses échanges. Publié en 1977 par Ronald Rivest, Adi Shamir et Leonard Adelman, il est aujourd'hui le standard de chiffrement à clé publique, se faisant peu à peu remplacer par les standards post-quantiques (que vous découvrirez l'an prochain). La sécurité de ce crypto-système repose entièrement sur la difficulté du problème de factorisation, que l'on sait à ce jour sensible aux attaques quantiques.

Tout comme les différents systèmes que vous avez pu traiter au cours des TP précédents, l'implémentation de RSA est documentée consciencieusement dans la RFC 8017. Cette dernière fait état des systèmes de Chiffrements (*RSAES-OEAP* et *RSAES-PKCS1-v1_5*) et de Signatures (*RSASSA-PSS* et *RSASSA-PKCS1-v1_5*), lequel nous intéresse ici. Les systèmes décrits diffèrent dans la méthode d'encodage du message. On considère ici la Signature RSASSA-PSS, qui bénéficie d'un encodage dit *EMSA-PSS*.

D'un point de vue implémentation, le chiffrement et la signature RSA opèrent tous deux sur des messages sous forme de suite d'octets au format *little endian*. Ainsi, un message (entier) m sera vu comme une suite d'octet $m_0, ...m_n$. Cette conversion est décrite dans la section 4.

1. Écrire une fonction `I2OSP` qui prend en entrée un entier x et une taille n et qui retourne la représentation hexadécimale de x sur n octets au format little endian. Vérifiez que $\texttt{I2OSP}(x = 928735817, n = 7) = \texttt{0x 49 62 5b 37 00 00 00}$.

2. Écrire une fonction `O2ISP` qui prend en entrée une suite d'octets X au format litlle endian et qui retourne l'entier x décrit par la représentation hexadécimale X. Vérifiez votre fonction avec le test précédent.

Afin de tester les fonctions suivantes, générez une paire de clés publique/privée $((N, e), (p, q, d))$ avec le module RSA de PyCryptodome, dont vous trouverez un exemple simple de génération dans la documentation.

Le schéma de Signature est implémenté sur les primitives de Signature et de Vérification décrites en section 5.2. Un utilisateur souhaitant signer un message (entier) m utilise sa clé privée. Il possède deux manières d'utiliser sa clé :

- (n, d) - il signe comme s'il effectuait un déchiffrement,
- $(p, q, d_p, d_q, inv\_q, inv\_p)$ - il recompose la signature avec le théorème des restes chinois : $s = s_p \cdot q \cdot (inv\_q \pmod p) + s_q \cdot d \cdot (inv\_p \pmod p)$ où $s_p = m^{d_p} \pmod p$ (et de même pour s_q).

3. Implémentez la primitive de signature `RSASP(k,m)` qui prend en entrée une clé privée k et un message (entier) m et qui retourne la signature de ce message m sous la clé k. Cet algorithme doit laisser le choix à l'utilisateur d'utiliser les deux versions de sa clés privées (N, d) ou p, q, inv_p, d_d, d_q comme vu en TD récemment.

4. Implémentez la primitive de vérification `RSAVP(K,s,m)` qui prend en entrée une clé publique K et une signature s et qui retourne m le message sous-jacent.

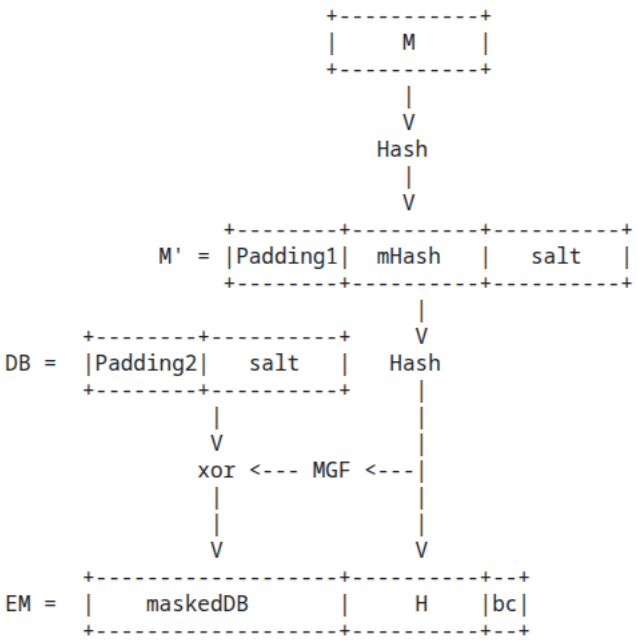

Figure 1: EMSA-PSS Encoding Operation

Testez vos fonctions en générant des messages aléatoires entre 0 et $N-1$ ou N vaut votre clé publique générée précédemment. Vérifiez ainsi que

$$\texttt{RSAVP}((N,e),\texttt{Sign}((N,d),m)) == m \text{ et } \texttt{RSAVP}((N,e),\texttt{Sign}((p,q,inv\_p,d_p,d_q),m)) == m$$

Maintenant que les primitives et les conversions sont implémentées, il nous faut encoder le message M de la bonne manière. Pour un message M, vu comme une suite d'octet, la figure précédente décrit l'encodage *EMSA-PSS* expliqué en section 9.1.1 de M avant conversion vers un entier.

Dans le but du bon fonctionnement de votre encodage, vous appelerez les librairies suivantes :

- `hashlib` pour les fonctions de hachage (vous utiliserez `sha256`),
- `PyCryptodome` pour la fonction de génération de masque (`MGF`).
- `PyCryptodome` pour le sel `salt` random.

5. Écrire une fonction d'encodage `EMSA_PSS_Encode(`M`,emBits)` qui pour une suite d'octets M et une taille maximale d'entiers (en bits) `emBits` retourne EM l'encodage *EMSA-PSS* du message M.

Dès lors, vous pouvez implémenter la signature d'un message (suite d'octet) M pour une clé privée k, comme décrit en section 8.1.1.

6. Implémentez la fonction de signature `RSASSA_PSS_Sign(`k`,`M`)` qui prend en entrée un message M (suite d'octets) et une clé privée k et qui renvoie la signature sous la clé k du message encodé sous *EMSA-PSS* EM puis convertie en entier em.

Afin de vérifier la signature générée précédemment, il faut maintenant pouvoir attester que l'encodage EM est valide pour le message M après application de la primitive de vérification. La description de l'algorithme d'encodage pour la vérification se trouve à la suite de l'algorithme d'encodage pour la signature, en section 9.1.2.

7. Écrire une fonction d'encodage pour la vérification `EMSA_PSS_Verify(`M`,`EM`,emBits)` qui pour une suite d'octets M, un encodage EM et une taille maximale d'entiers (en bits) `emBits` retourne un booléen caractérisant la consistence de l'encodage EM au regard de M.

8. Implémentez la fonction de vérification `RSASSA_PSS_Verify(`(N,e)`,`M`,`s`)` qui prend en entrée un message M (suite d'octets), une clé publique (N,e) et une signature s et qui retourne un booléen `True` si et seulement si la signature s est valide pour le message M.

Générez des suites d'octets aléatoires M afin de vérifier que

$$\texttt{RSASSA\_PSS\_Verify}((N,e), M, \texttt{RSASSA\_PSS\_Sign}((N,d), M)) == \texttt{True}$$
$$\texttt{RSASSA\_PSS\_Verify}((N,e), M, \texttt{RSASSA\_PSS\_Sign}((p,q,inv\_p,d_p,d_q), M)) == \texttt{True}$$