TD 4: hachage et protocoles

1 El Gamal

On étudie le fonctionnement d'El Gamal dans une structure additive (comme ce serait le cas par exemple pour l'ensemble des points rationnels d'une courbe elliptique). Pour p premier et $\alpha \in \mathbb{Z}_p$, on calcule $\beta \equiv \alpha.a \mod p$. On signe un message M par (γ, δ):

$$\begin{cases} \gamma \equiv k.\alpha \mod p & \text{pour } k \text{ une valeur aléatoire} \\ \delta \equiv (M - a.\gamma).k^{-1} \mod p \end{cases}$$

(1) Quels sont les paramètres publics et privés?

(2) Montrez que $\gamma.\beta + \gamma.\delta \equiv \alpha.M \mod p$ valide la signature.

(3) Décrivez une attaque simple contre ce procédé de signature.

(4) Avec $p = 37$, $\alpha = 2$, $\beta = 14$ vérifiez la validité de $(M, \gamma, \delta) = (8, 10, 32)$ et retrouvez le paramètre privé.

2 Hachage de Bernstein

Dan Bernstein est le créateur de la fonction de hachage suivante, aussi connue sous le nom de Chris Torek Hash.

```
def djb2(key):
  hash = 5381
  for c in key:
    hash = ((hash * 33) + ord(c)) % (2**8)
  return hash
```

1. Calculez l'empreinte du mot "`doc`" codé en ASCII ou en utf-8 (`a`=97 sur un format fixe de un octet) par la fonction de hachage ci-dessus (à la main ou avec Python).
2. Pourriez-vous trouver une collision au mot "`doc`"? En déduire que la fonction `djb2` ne satisfait pas la faible résistance.
3. Appliquez le paradoxe des anniversaires à cette fonction de hachage pour savoir combien il faut engendrer de paires pour avoir une probabilité de trouver une collision supérieure à 3/4.
4. En utilisant le paradoxe des anniversaires avec probabilité 3/4, trouvez deux mots de longueur 5 distincts qui donnent la même empreinte (vous pourrez faire appel à un dictionnaire `fr_dic`).

Remarque: La fonction d'origine travaille modulo 2^{32} et non 2^8. Réflechissez à ce que cela change dans la recherche de collisions.

3 Signature DSA avec hachage en utilisant `Cryptography`

Avec la documentation officielle, générez une clé DSA de 1024 bits puis signez l'empreinte par `SHA256` d'un petit message au format `bytestream`.
Vous écrirez quatre fonctions:

- `ecrirecle` qui prend en entrée la taille de la clé à engendrer et qui retourne deux fichiers au format `pem`: `cle_privee.pem` et `cle_publique.pem`;
- `lirecle` qui va lire les fichiers de clé publique et privée et retourne les variables `publique, privee`;
- `signe` qui prend en entrée un message au format bytestream et une clé (privée) et qui retournera la signature au format bytestream;
- `verifie` qui prend en entrée le message, la signature et une clé (publique) et qui retourne `verification OK` si la signature est valide et `invalide` sinon.