Algorithme RSA

1. Préparation
2. Cryptage, décryptage
3. Une attaque
4. Algorithmique
5. Test probabiliste de primalité
6. Miller-Rabin
7. Génération
8. Classe `BigInteger`

1. Préparation

Algorithme de Rivest, Shamir et Adleman.

FAIT *Soient* $p \neq q$ *deux nombres premiers,* $n = pq$*, et soit* e *un entier premier à* $(p-1)(q-1)$*. Alors il existe* d *tel que* $ed \equiv 1 \bmod (p-1)(q-1)$*, et*

$$a^{ed} \equiv a \bmod n$$

Le système RSA travaille avec l'hypothèse qu'il est difficile d'obtenir p et q à partir de n.

Génération de clés

Chaque destinataire (Bob) prépare son système comme suit.

- **Il choisit deux grands nombres premiers p et q, et calcule $n = pq$.**
- **Il choisit e premier à $(p-1)(q-1)$ et calcule d tel que $ed \equiv 1 \bmod ((p-1)(q-1)$.**

(n, e) est la *clé publique* de Bob.

d est sa *clé secrète.*

Les nombres p et q sont oubliés.

2. Cryptage, décryptage

Alice
crypte **un message $m < n$ pour Bob avec la clé publique de Bob.**

$$c = E_e(m) \equiv m^e \bmod n$$

et envoit c à Bob.

Bob
reçoit c et *décrypte*, grâce à sa clé secrète

$$m = D_d(c) \equiv c^d \bmod n$$

Comme $c^d \equiv m^{ed} \equiv m$, il obtient le texte en clair.

Il est *difficile* de calculer d (sauf si on connaît p et q), et il est *difficile* de factoriser n.
La fonction $E : m \mapsto m^e$ est *à sens unique* et d est une *trappe*.

Implantations

Il existe des chips pour RSA. Sur ces chips, RSA est environ 1000 fois plus lent que DES.

Une attaque

L'attaque porte sur la communication de clés.

- **Alice demande la clé publique à Bob.**
- **Bob envoit e à Alice.**
- **Mallory intercepte le message de Bob et envoit sa clé e'.**
- **Alice crypte avec e'.**
- **Mallory intercepte le message c' d'Alice.**
- **Il décrypte avec sa clé secrète.**
- **Mallory change le message d'Alice et le crypte avec e.**

Alice doit être sûre que e est bien la clé de Bob.

Algorithmique

Bezout

Donnée a, b positifs.

Résultat (d, x, y) tels que $d = \mathrm{pgcd(a, b)}$ et $ax + by = d$.

- Si $b = 0$: retourner $(d, x, y) = (a, 1, 0)$.
- $(x, y) = (1, 0)$, $(x', y') = (0, 1)$
- tantque $b > 0$
  - $q = \lfloor a/b \rfloor$, $r = a - bq$, $(x'', y'') = (x - qx', y - qy')$; $(a, b) = (b, r)$, $(x, y) = (x', y')$; $(x', y') = (x'', y'')$;
- retourner (a, x, y)

Le temps est $O(\max(\|a\|, \|b\|)^2)$, où $\|a\|$ est le nombre de bits de a.
Si a et b ont 512 bits (150 décimales), env 1 million d'opérations élémentaires.

Exponentiation modulo n, par l'algorithme dichotomique.

Donnée $a < n$ **et** k.
Résultat $a^k \equiv \bmod n$

- $b = 1$, $y = a$;
- **tantque** $k \neq 0$

  si k **impair** $b \equiv y * b \bmod n$

  $k = \lfloor k/2 \rfloor$; $y = y^2 \bmod n$.
- **retourner** b.

Temps $O(\|n\|^3)$.

Tout ce qui est polynomial en $\|n\|$ **est "faisable".**

Test probabiliste de primalité

Algorithme qui donne une information partielle de primalité.

Soit $n > 1$ impair. Soit $W(n) \subset Z_n$ tel que

- **pour $a \in Z_n$, il est facile de vérifier si $a \in W(n)$;**
- **si n est premier, alors $W(n)$ est vide;**
- **si n est composé, alors Card(W(n)) $\geq$ n/2.**

Si n est composé, $W(n)$ est l'ensemble des *témoins* de n, et $L(n) = Z_n - W(n)$ est l'ensemble des *menteurs*.

Usage

On veut tester si n est premier.

- **on choisit $a \in Z_n$ au hasard;**
- **on teste si $a \in W(n)$;**
- **si "oui", alors n est composé;**
- **si "non", alors n *a passé le test pour* a.**

Si n est composé et n a passé le test pour a, alors $a \in L(n)$.

On a moins d'une chance sur 2 de passer le test.

Si on fait t tests, et si n passe ces tests, on a moins de $(1/2)^t$ probabilité que n est composé.

3. Miller-Rabin

Le plus couramment utilisé.

FAIT *Soit* n *premier impair, et* $n - 1 = 2^e r$ *avec* r *impair. Pour tout entier* a *tel que* $\mathrm{pgcd(a, n)} = 1$, *on a*

$$a^r \equiv 1 \bmod n$$

ou il existe j *avec* $0 \leq j \leq e - 1$ *tel que*

$$a^{2^j r} \equiv -1 \bmod n$$

Exemple: $n = 91$. **Alors** $n - 1 = 2 \cdot 45$, **donc** $e = 1$, $r = 45$. **On a** $9^{45} \equiv 1 \bmod 91$

Soit n composé et $n-1 = 2^e r$ avec r impair.

Un *témoin* est un entier a avec $2 \leq a \leq n-2$ tel que

$$a^r \not\equiv 1 \bmod n \quad \text{et} \quad a^{2^j r} \equiv -1 \bmod n$$

pou tous $j = 0, \ldots, e-1$.

Un *menteur* est un nombre a tel que $a^r \equiv 1 \bmod n$ ou il existe j avec $0 \leq j \leq e-1$ tel que $a^{2^j r} \equiv -1 \bmod n$.

FAIT *Soit* n *composé impair. Alors au plus* $1/4$ *des nombres entre* 1 *et* $n-1$ *sont des menteurs.*

Algorithme

Donnée **Un entier $n \geq 3$ impair et un paramètre de sécurité t.**
Résultat **n est premier avec probabilité $1 - 1/2^k$ ou n est sûrement composé.**

1. Calculer e, r tels que $n - 1 = 2^e r$.

2. Faire t fois

   - choisir a avec $2 \leq a \leq n-2$;
   - calculer $y =\equiv a^r \bmod n$
   - si $y = 1$ ou $y = n-1$ continuer 2.
   - Faire e fois

     $y = y^2$;

     si $y = -1$ continuer 2;
   - retourner “compose”

3. retourner “probablement premier”.

Génération

Génération est différente de test.

Donnée **Un nombre k de bits et un paramètre de sécurité t.**
Résultat **Un nombre à k bits probablement premier.**

- **Engendre un entier n impair à k bits au hasard.**
- **Eliminer n s'il a un "petit" diviseur.**
- **Eliminer n si le test Miller-Rabin(n, t) échoue, et recommencer.**
- **retourner n.**

L'expérience montre

- **le test par de petits nombres premiers élimine 80% des candidats;**
- **commener Miller-Rabin avec $a = 2$ permet un calcul des puissances par décalage, et élimine la plupart des candidats;**
- **$t = 3$ suffit jusqu'à 1000 bits.**

4. La classe `BigInteger`

Le package `java.math` fournit deux classes `BigDecimal` et `BigInteger`.

La classe `BigInteger` possède de nombreuses primitives utiles pour la cryptographie, par exemple;

`BigInteger modPow(BigInteger e, BigInteger n`

retourne l'entier $\texttt{this}^e \bmod n$

`int getLowestBit()`

retourne le nombre de bits nuls à la fin de l'entier

`BigInteger shiftRight(int)`

décale comme indiqué.

Par exemple, le début de Miller-Rabin s'écrit:

```
int e = n;getLowestBit();
r = n.shiftRight(e);
```

De fait, la classe contient un constructeur

```
BigInteger(int bitLength, int certainty,
Random rnd)
```

qui construit un entier au hasard (avec le générateur `rnd`**) qui possède** `bitLength` **bits et qui est probablement premier avec certitude ($1/2^c$), où** c= `certainty`.