Projet Scilab - Cryptographie

Samuel Abergel & Yaniv Ohnona

9 janvier 2007

1 Introduction

Ce projet avait pour but de programmer des algorithmes de cryptographie, on a ainsi considéré les 3 méthodes notamment étudiés en module d'arithmétiques :
- RSA
- Le protocole d'échange de clés de Diffie-Hellman
- L'algorithme de El-Gamal.

1.1 Mise en route

En cryptographie, les notions reviennent (Euler, primalité, bezout, pgcd, modulo, ..) ; par souci d'efficacité il nous semblait important de diviser les programmes en sous-programmes indépendants ainsi que de faire des déclarations globales pour les variables principales. Nous allons d'abord mettre en place les fonctions qui nous serviront tout au long du projet.
- isprime : un test de primalité qui n'est pas défini à l'origine dans scilab, l'algorithme « brut »est ici efficace.
- fonction d'euler : l'algorithme est simple, par souci d'efficacité, il est utile de faire un test de primalité en début de fonction.
- isgenerator : soit $\mathbb{F}_q$ et k, on test si k est un générateur de $\mathbb{F}_q$. Pour cela on verifie si il existe $i \in [1, q-2]$ tel que $k^i = 1$. Si un tel i existe c'est que k ne genere pas $\mathbb{F}_q$ et reciproquement. (on suppose q premier)
- fonction modulo : Ce sont celles qui nous ont posé le plus de problèmes ! on pouvait s'en douter, un algorithme RSA manipule de grands nombres, les calculs deviennent alors vite impossible si les implémentations ne sont pas définies de facon éfficaces. La fonction définie à la base dans scilab est tout à fait inneficace, dans un premier temps on a définit un algorithme qui décomposait les éléments en puissances de 10 déja un peu mieux mais insatisfaisant pour de grands chiffres. Le dernier algortihme est bien plus rapide.

fonction modulo - code retenu	Explication

function y=pmod(ii,oo,pp)
global('d')
global('e')
global('a')
l1=[ii,oo,1]
while l1(2)=0
if modulo(l1(2),2)=0 then
l1=[modulo(l1(1)^2, pp), $l1(2)/2$, $modulo(l1(3), pp)$]
else
l1=[modulo(l1(1)^2, pp), $(l1(2)-1)/2$, $modulo(l1$
(3)*l1(1),pp)]
end
end
y=l1(3)
endfunction
	Deux choses à remarquer, la fonction prend 3 paramètres (souci d'efficacité), on calcule $\mathrm{mod}(d^e, a)$
	ce qui est trés pratique puisque dans RSA et les autres algorithmes les calculs de modulos se font toujours avec cette syntaxe.
	
	Parmi les autres facons de faire, on a aussi monter une version qui à l'aide d'une double boucle enlevait $10^i * a$ avec i variant de 1 à 1000
	Des algorithmes plus longs mais combien moins efficaces.

1.2 Méthode RSA

1.2.1 Rappels

Le but du jeu est bien sûr de pouvoir transmettre un message codé que seul le récépteur officiel puisse décryper, c'est à dire qui ne puisse pas être décrypté par un tiers qui intercepterait ledit message. Nous appellerons Alice la destinatrice du message, et Bernard l'emetteur. Voici les étapes :

1. Alice génère deux gros nombres premiers p et q, ainsi qu'un gros nombre d premier avec le produit $w = (p-1)(q-1)$
2. Alice calcule $n = pq$ et e tel que $de \equiv 1\,[w]$
3. Alice diffuse n et e garde d et oublie w
4. Bernard crypte un message M par $\mathrm{M} \mapsto \mathrm{M}^e\,[n]$
5. Alice décode alors le message crypté par $\mathrm{C} \mapsto \mathrm{C}^d\,[n]$

1.2.2 Partie programmation

On définit un environnement initial qui est composé de variables globales (d, e et a) ainsi que de fonctions « primitives »(isprime, gen-prem, pmod et step1). La variable a est une matrice à deux élements, deux nombres premiers tirés aléatoirement. . Les variables e et d correspondent aux mêmes lettre e et d cités au dessus (voir rappels) . Enfin, 2 programmes finissent l'algorithme, le premier permet l'encryption (Bernard-Alice) et le second le décryptage du message (dans le sens Barnard-Alice). **Le temps de reponse ne dépasse pas la seconde**.

1.2.3 Mise en route

Nous allons proceder en deux étapes ; dans un premier temps on initialise scilab avec l'environnement de départ puis nous testerons deux nombres et par la même occasion nous essairons de trouver les limites de notre l'algorithme. Voir transparents page ci-aprés.

RSA - Environnement initial

```
-->global('d')
-->global('e')
-->global('a')
```

Déclarations des variables g^ales

```
-->
-->exec('C:\Documents and Settings\Samuel Abergel\Bureau\Projet Scilab\Nouveau dossier\Nouveau dossier\gen_d(… conserver
).sci');disp('exec done');
 exec done

-->exec('C:\Documents and Settings\Samuel Abergel\Bureau\Projet Scilab\Nouveau dossier\Nouveau dossier\gen_prem.sci');di
sp('exec done');
 exec done

-->exec('C:\Documents and Settings\Samuel Abergel\Bureau\Projet Scilab\Nouveau dossier\Nouveau dossier\isprime.sci');dis
p('exec done');
 exec done

-->exec('C:\Documents and Settings\Samuel Abergel\Bureau\Projet Scilab\Nouveau dossier\Nouveau dossier\gen_e.sci');disp(
'exec done');

 -->exec('C:\Documents and Settings\Samuel Abergel\Bureau\Projet Scilab\Nouveau dossier\Nouveau dossier\pmod.sci');disp('
 exec done');
```

Fonctions « primitives »

1. gen_d
2. gen_prem
3. isprime
4. gen_e
5. pmod

RSA – Cryptage/décryptage – 1/2

```
-->exec('C:\Documents and Settings\Samuel Abergel\Bureau\Projet Scilab\Nouveau dossier\Nouveau dossier\step1.sci');disp(
'exec done');

 exec done

-->step1
 ans  =

    467.    29.

-->a
 a  =

    467.    29.

-->gen_e
 ans  =

    3.

-->gen_d
 ans  =

    8699.

-->exec('C:\Documents and Settings\Samuel Abergel\Bureau\Projet Scilab\Nouveau dossier\Nouveau dossier\bernard_alice.sci
');disp('exec done');

 exec done

-->exec('C:\Documents and Settings\Samuel Abergel\Bureau\Projet Scilab\Nouveau dossier\Nouveau dossier\bernard_alice_d.s
ci');disp('exec done');

 exec done

-->bernard_alice(12)
 ans  =

    1728.

-->bernard_alice(1728)
 ans  =

    5.160D+09

-->bernard_alice_dec(1728)
 ans  =

    12.
```

On génère les deux nombres premiers à l'aide de la fonction 'step1' qui sont stockés automatiquement dans la variable globale a.

On génère les 2 autres variables globales qui correspondent aux nombres premiers trouvés

TEST: le message à envoyer est 12 – après cryptage il envoie 1728

Décryptage: Alice reçoit le message M=12

RSA – Cryptage/décryptage – 2/2

```
-->bernard_alice_dec(bernard_alice(11))
 ans  =

    11.

-->bernard_alice_dec(bernard_alice(34))
 ans  =

    34.

-->bernard_alice_dec(bernard_alice(56))
 ans  =

    56.

-->bernard_alice_dec(bernard_alice(6898765))
 ans  =

    0.

-->(bernard_alice(6898765))
 ans  =

    3.283D+20
```

On a bien une application identité: $id(M) = f^{-1}(f) = M$

Etonnant!

Le chiffres sont trop grands pour être considérés comme entiers
Ils sont « approximés » d'où les erreurs.

1.3 Diffie-Hellman : Un protocole d'échange de clés

1.3.1 Rappels

Soit $\mathbb{F}_q$, un corps finis à q éléments (ici on simplifie, q est premier) et soit g un générateur de $\mathbb{F}_q$. La clé publique est alors $(\mathbb{F}_q, g)$. Le protocole est le suivant :
– Alice choisit un entier a vérifiant $1 < a < q - 1$ et transmet sa clé publique g^a à Bernard
– Bernard choisit un entier vérifiant $1 < b < q - 1$ et transmet sa clé publique g^b à Alice
– Alice élève g^b à la puissance a, obtenant $\gamma = g^{ab}$ qui sera leur clé secréte commune
Alice et Bernard sont donc les seuls à connaître γ

1.3.2 Mise en route

La seule variable globale correspond à la clé publique (voir rappel), ie $(\mathbb{F}_q, g)$.
En ce qui concerne les fonctions utilisées, on utilisera la plupart des primitives de RSA (isprime, gen_, euler,...) auquel on ajoute les fonctions suivantes :
– 'generateur' : aprés avoir généré un nombre premier q – aléatoirement – cette fonction trouve un générateur de $\mathbb{F}_q$; en utilise la primitive 'isgenerator' qu'on a définit en §1
– 'dh_base' : soit q un nombre premier, cette fonction trouve un generateur de $(\mathbb{F}_q, g)$.
Alice puis Bernard choisissent deux nombres aléatoire a et b La clé secrete est $\gamma = g^a * g^b$
Notre algorithme est donc défini !

1.3.3 El Gamal

Le fonctionnement est identique à l'algorithme de Diffie-Hellman.

2 Conclusion

Sur la dizaine de séance sur Scilab qui se sont tenues ce semestre, environ la moitié a été nécessaire pour nous permettre une prise en main correct pour une utilisation de « base »du logiciel. Pour l'autre moitié du semestre, on devait réaliser un projet sur un thème au choix. On a préféré éviter les domaines qui nous étaient inconnus comme les statistiques, l'image numérique pour une raison simple : on voulait passer le plus de temps à réaliser quelque chose de complet plutôt que de passer du temps sur des notions de cours. Dans ce sens notre choix s'est porté sur la cryptographie (on possédait des notions de Terminale). La tache s'est avérée difficile, Matlab (Matrix laboratory) est un logiciel d'approximation numérique qui n'est pas commode à utiliser avec de grands chiffres comme on en trouve en cryptographie. De plus, les primitives implantées à l'origine dans Scilab sont difficilement manipulables [1] et pas du tout efficaces pour de grands nombres, on a cependant utilisé le maximum de fonctions prédéfinies dans Scilab.
Une organisation solide était obligatoire pour aboutir, on a donc scindé le projet en 2 parties

1. la partie programmation " théorique " (ie en considérant les primitives efficaces et prédéfinies) : Les notions reviennent à chaque bout de code en arithmétique, c'est donc par souci d'efficacité que les programmes ont été écrits et pensés le plus indépendamment possible, les uns de autres. On compte plus d'une vingtaine de sous-programmes réalisés avec une moyenne de 12 lignes.
2. la programmation des primitives : La programmation des primitives devait se faire de façon efficace, c'est ainsi que pour une même primitive on avait rédigé 5 versions toutes différentes cependant toutes inefficaces.

Dans ce compte-rendu, on a volontairement omis d'approfondir chaque fonction il s'agit à l'origine d'une unité de maths et pas d'info. Cette ue hormis l'aspect mathèmatiques nous a permit d'approfondir nos connaissances en ce qui concerne la suite LaTeX

[1] On a testé un calcul de modulo entre deux nombres d'une vingtaine de chiffres chacun, on a du attendre le lendemain pour obtenir une réponse... fausse