À LA RECHERCHE DE CLÉS SECRÈTES VULNÉRABLES

SUJET PROPOSÉ PAR BRUNO SALVY
VERSION 1.2
DIFFICULTÉ : MOYENNE

Résumé. Une bonne partie de la cryptographie actuelle repose sur des clés publiques qui sont des produits de (grands) nombres premiers, auxquelles correspond une clé secrète qui est *grosso modo* la factorisation. Être capable de trouver ces factorisations révèle le secret. Ce sujet explore des algorithmes ayant récemment permis de montrer la faiblesse de nombreuses clés utilisées par des sites web ou des routeurs.

Lorsque votre navigateur web arrive sur un site qui utilise de la cryptographie, il vous présente un petit cadenas devant l'url du site pour vous en informer. Cliquer sur ce cadenas fait apparaître de nombreuses informations comme les certificats qui permettent d'assurer que le site auquel vous accédez est bien celui qu'il prétend être. Une bonne partie de ces sites (ainsi que les connections entre machines que vous effectuez avec la commande `ssh`, ou les puces de vos cartes de paiement) utilisent au moins en partie de la cryptographie à clé publique à l'aide du cryptosystème RSA ou DSA (voir ci-dessous). Le principe est qu'une *clé publique*, qui est un produit de deux grands nombres premiers, vous est envoyée pour que vous puissiez l'utiliser pour transmettre au site des informations cryptées, que lui seul peut décrypter à l'aide de sa *clé privée* (facile à calculer avec les deux facteurs). La sécurité de ces systèmes repose sur le fait qu'il est facile de produire des nombres premiers de grande taille et de les multiplier, mais actuellement encore très difficile de factoriser des entiers de quelques centaines de chiffres, et c'est même impossible pour des entiers de mille chiffres. Bien entendu, si deux sites utilisent la même clé, l'un peut se faire passer pour l'autre. Pire, si les clés de deux sites ont un facteur premier commun, alors un simple calcul de pgcd (opération rapide) permet de récupérer les clés privées de chacun d'entre eux et donc de briser leur sécurité. Cette observation simple, jointe au très grand nombre de sites utilisant de la cryptographie, est à la base d'une expérience intéressante menée il y a peu par un petit groupe de cryptographes. Ils ont commencé par collecter un très grand nombre de clés publiques, avant de rechercher si des facteurs premiers répétés pouvaient être détectés dans cet ensemble. Ce sujet de projet propose d'implanter la partie la plus algorithmique de cette expérience.

1. Mise en place : une implantation jouet de RSA

L'algorithme RSA est désormais classique. Son nom est composé des initiales de ses trois découvreurs en 1978 : Rivest, Shamir et Adleman. Une clé publique est composée de deux éléments : un entier $N = pq$ produit de deux nombres premiers p et q et un exposant e inférieur à $(p-1)(q-1)$ et premier avec $(p-1)(q-1)$. Le système repose sur un theorème d'Euler qui affirme que pour tout m premier avec N, l'équation $m^{(p-1)(q-1)} \equiv 1 \bmod N$ est satisfaite.

Les messages à transmettre[1] sont d'abord découpés en suites d'entiers entre 0 et $N-1$. Un tel entier $m \in \{0, \dots, N-1\}$ est crypté à l'aide de la clé publique en

$$m \mapsto y = m^e \bmod N.$$

La clé privée permet facilement de calculer une fois pour toutes

$$d = e^{-1} \bmod (p-1)(q-1).$$

1. En pratique, ces messages sont souvent des clés privées qui seront utilisées pendant le reste de la transaction, mais nous ignorerons ce point ici.

À l'aide de ce nombre on peut retrouver le message parce que

$$y^d = m^{ed} = m^{1+k(p-1)(q-1)} = m \bmod N.$$

Le décryptage consiste donc simplement en l'opération $y \mapsto y^d \bmod N$.

1.1. **Exponentiation binaire.** Pour calculer une puissance k^e, une approche récursive permet de passer de k à moins de $2\log k$ multiplications grâce à l'observation suivante :

$$x^k = \begin{cases} (x^{k/2})^2, & \text{si } k \text{ est pair,} \\ x(x^{(k-1)/2})^2, & \text{sinon.} \end{cases}$$

1.2. **Algorithme d'Euclide et inversion modulaire.** L'algorithme d'Euclide pour calculer le pgcd G de deux entiers A et B est sans doute l'un des plus vieux algorithmes mathématiques encore en usage. Une variante, appelée algorithme d'Euclide étendu, permet de calculer, en même temps que le pgcd, deux cofacteurs U et V tels que $UA + VB = G$. En appliquant cet algorithme à $A = e$ et $B = (p-1)(q-1)$, on obtient ainsi l'inverse d dont on a besoin pour décrypter. Le pseudo-code est le suivant :

procédure ExtendedGCD(A, B)
Entrée : deux entiers positifs A et B ;
Sortie : Un pgcd G de A et B, et deux entiers U et V tels que $UA + VB = G$
$R_0 := A; U_0 := 1; V_0 := 0; R_1 := B; U_1 := 0; V_1; = 1; i := 1;$
tant que $R_i \neq 0$
faire $\begin{cases} (Q_i, R_{i+1}) := \text{DivisionEuclidienne}(R_{i-1}, R_i); \\ U_{i+1} := U_{i-1} - Q_i U_i; V_{i+1} := V_{i-1} - Q_i V_i; \\ i := i + 1 \end{cases}$
renvoyer $R_{i-1}, U_{i-1}, V_{i-1}$

1.3. **Implantation.** Le fichier `minisecret.txt` disponible sur le site du sujet contient un message crypté à l'aide d'une petite clé publique (de 256 bits) fournie dans le fichier `minicle.pub`, dont le contenu est :

```
-----BEGIN PUBLIC KEY-----
MDwwDQYJKoZIhvcNAQEBBQADKwAwKAIhAMPvQ2oTQEGVpQOO4D7OuANlMsrse9pa
iYfGU6fobbqDAgMBAAE=
-----END PUBLIC KEY-----
```

La commande unix

```
openssl rsa -pubin -text < minicle.pub
```

permet de récupérer l'entier N et l'exposant e. S'ils sont en hexadécimal, il vous faudra les convertir en décimal. Vous pouvez ensuite profiter de la petite taille de N pour le factoriser [2]. Ensuite, écrire un petit programme qui étant donnés les facteurs de N, retrouve le message secret. (Des indications sur l'usage d'`openssl` sont données sur le site du sujet.)

2. Recherche naïve : algorithme d'Euclide et exploration paire par paire

Le principe de cette recherche de clés est de calculer les pgcds nontriviaux entre de nombreuses clés publiques. Une première approche consiste donc à rechercher ces pgcds en effectuant une recherche exhaustive par l'algorithme d'Euclide (le même que ci-dessus, sauf qu'il n'est pas nécessaire de calculer les cofacteurs U et V).

Implantation. Le fichier `keys100.txt` disponible sur le site du sujet contient 100 clés publiques, en décimal, avec 9 facteurs premiers qui apparaissent deux fois. Votre programme doit les retrouver. Le résultat sera la liste des clés vulnérables et leurs facteurs découverts, écrits dans un fichier. En effectuant le même calcul sur les 50 premières clés et en comparant les temps de calculs, estimer la complexité de cette méthode.

2. Pas à la main ! Utilisez un système de calcul formel. La commande est `ifactor` en Maple, `FactorInteger` en Mathematica, et `factor` en Sage. Ce dernier est libre et peut aussi s'utiliser sans installation, via une version en ligne à l'adresse https://cloud.sagemath.com/).

3. Arbres des produits et des restes

Lorsque le nombre N de clés publiques devient grand, la méthode naïve demande $O(N^2)$ calculs de pgcd, ce qui devient rapidement trop coûteux. Il est possible de tirer parti d'une multiplication rapide pour abaisser cette complexité en procédant en deux temps.

D'abord, les clés, vues ici comme des entiers $k_1, \ldots, k_N$ sont multipliées entre elles dans un arbre binaire appelé *l'arbre des produits* : la racine contient le produit des k_i pour $i = 1, \ldots, N$, le sous-arbre gauche est obtenu à partir des clés $k_1, \ldots, k_{\lceil N/2 \rceil}$ et le sous-arbre droit avec les clés restantes. Vous estimerez la complexité de la construction de cet arbre lorsque la multiplication de deux entiers d'au plus m bits a complexité $O(m^2)$, et lorsqu'elle a complexité $O(m \log m \log\log m)$.

Ensuite, les pgcds sont construits récursivement à partir de cet arbre par un *arbre des restes* : si $K_0 = K_1^{(1)} K_1^{(2)}$ est le produit à la racine de l'arbre des produits, $K_1^{(1)}$ et $K_1^{(2)}$ étant les produits de ses sous-arbres gauche et droit, on calcule $R_0 := K_0 \bmod K_0^2$ (qui vaut K_0) et on fait un appel récursif pour calculer $R_1^{(1)} := R_0 \bmod (K_1^{(1)})^2$ et $R_1^{(2)} = R_0 \bmod (K_1^{(2)})^2$. Les feuilles de l'arbre binaire ainsi construit contiennent $r_i := K_0 \bmod k_i^2$, pour $i = 1, \ldots, N$. Par construction, r_i est un multiple de k_i, et il ne reste plus qu'à calculer le pgcd de r_i/k_i et de k_i pour détecter les facteurs communs éventuels. Vous estimerez la complexité du calcul de cet arbre lorsque la division eucilidienne de deux entiers d'au plus m bits a complexité $O(m^2)$, et lorsqu'elle a complexité $O(m \log m \log\log m)$.

Implantation. Implanter cet algorithme et vérifier que vous parvenez bien à retrouver les facteurs communs du fichier `keys100.txt`. Essayer aussi d'attrapper les facteurs communs des fichiers plus longs `keys1000.txt`,. . .,`keys100000.txt`.

Quelques optimisations sont possibles : par exemple, il n'est pas nécessaire de stocker tout l'arbre des restes en mémoire. Vous pouvez aussi (une fois toutes les questions traitées), tenter de paralléliser une partie du programme.

4. Multiplication et pgcd rapides

L'efficacité de l'algorithme à base d'arbres de produits et de restes dépend de manière cruciale de l'efficacité de la multiplication d'entiers et de la division qui s'en déduit, ainsi que, dans une moindre mesure, de celle du pgcd. La classe `java.math.BigInteger` permet de manipuler des entiers de taille arbitraire, mais l'algorithme utilisé pour la multiplication (au moins jusque dans les toutes dernières versions de la bibliothèque Java) est la méthode naïve en complexité quadratique, qui ne permet pas de calculer efficacement au-delà de quelques centaines de chiffres décimaux. Une bonne implantation d'algorithmes de multiplication rapide constituerait un projet en soi, et ne fait donc pas partie de ce projet, qui reposera sur une version plus moderne de `java.math.BigInteger` [3].

Implantation. En utilisant cette implantation de l'arithmétique des entiers, vérifier que la complexité observée lors de l'exécution de votre programme correspond à la complexité théorique que vous aviez obtenue. Parmi les clés du fichier `keys100000.txt`, l'une d'entre elles a servi à produire le fichier crypté `bigsecret.txt` qu'ils vous est demandé de décrypter.

Le jour de la soutenance, un nouveau fichier de 10000 clés publiques et un nouveau message secret encrypté avec l'une d'elles seront fournis.

5. Page Web

La page web
http://perso.ens-lyon.fr/bruno.salvy/INF421/Projets/Projet-cles-secretes.html
contient ce sujet, des références bibliographiques, des pointeurs vers des données avec lesquelles vous pourrez expérimenter. Elle évoluera en fonction de vos questions ; il sera donc judicieux de la consulter régulièrement.

3. On en trouve une à l'adresse https://github.com/tbuktu/bigint.git.