Démonstrateur d'attaque temporelle sur le RSA

TER Project

UVSQ - M1 SECRET

Sofiane Hamad, Shuyang Zhou, Linda Bedjaoui, Dylan Haral, Allaye Diallo

Month Day, 2022

Université Paris-Saclay

Contents

I - Introduction

1.1) Présentation du sujet

En cryptanalyse, une attaque temporelle consiste à estimer et analyser le temps mis pour effectuer certaines opérations cryptographiques dans le but de découvrir des informations secrètes. Certaines opérations peuvent prendre plus de temps que d'autres. La mise en œuvre de ce genre d'attaque dépend de la conception du système cryptographique, du processeur exécutant le système, des algorithmes utilisés, des divers détails de mise en œuvre, de la précision des mesures temporelles, etc...
L'objectif de ce projet est de mettre au point une démonstration la plus réaliste possible de l'attaque temporelle sur le cryptosystème RSA. L'attaque peut se faire au travers d'un protocole de communication réseau (en considérant SSL/TLS) entre deux ordinateurs. Dans ce projet, le cryptosystème RSA devra implémenter l'algorithme de Montgomery pour la multiplication modulaire ainsi qu'un outil automatique de chiffrement RSA pour l'attaque temporelle.

1.2) Présentation du chiffrement RSA

Contexte :

Inventé par Ron Rivest, Adi Shamir et Len Adleman, le système RSA fut présenté pour la première fois en août 1977, dans la chronique mathématique de Martin Gardner de la revue Scientific American.
Ces trois auteurs avaient décidé de travailler ensemble pour démontrer l'impossibilité logique des systèmes cryptographiques "à clé publique". Ils échouèrent donc en découvrant un système de cryptographie à clé publique, le système RSA.
Aujourd'hui son efficacité est reconnue mondialement, il est devenu universel servant dans une multitude d'applications. Il est très utilisé pour le commerce électronique ou encore pour les transactions sécurisées via internet qui en fait un usage systématique pour assurer la confidentialité du courrier électronique ou pour authentifier les utilisateurs. De nombreux protocoles, tels que SSH, OpenPGP, S/MIME et SSL/TLS reposent sur RSA pour leurs fonctions de chiffrement et de signature numérique.

Fonctionnement :

Le chiffrement RSA est un type chiffrement asymétrique c'est à dire qu'il utilise une paire de clés composée d'une clé publique pour chiffrer et d'une clé privée pour déchiffrer.
La sécurité de RSA repose sur la difficulté que représente la factorisation de grands nombres. Il est très difficile de factoriser de grands nombres entiers qui sont eux-mêmes le produit de deux grands nombres premiers. Multiplier ces deux nombres premiers est facile, mais déterminer les nombres premiers d'origines à partir du produit est considéré comme une opération 'impossible', étant donné qu'il n'existe pas à l'heure actuelle d'algorithme assez efficace, capable de trouver une solution en un temps raisonnable, même avec les super-calculateurs actuels. Cependant, à mesure que la puissance de traitement augmente, il devient possible de factoriser des nombres de plus en plus grands. C'est pour cette raison que la norme actuelle est d'adopter des clés d'une longueur minimale de 2048 bits, voir même de 3072 bits.

L'algorithme de génération des clés est la partie la plus complexe du chiffrement RSA. Deux grands nombres premiers, que l'on nomme p et q, sont générés à l'aide du test de primalité de Rabin-Miller ou du test de primalité de Solovay-Strassen. Le module de chiffrement n est calculé en multipliant p et q. Ce nombre est utilisé par les deux clés et constitue le lien qui les relient. La longueur du module n est exprimée en bits et équivaut à la longueur de la clé publique et de la clé privée.
La clé publique se compose du module n ainsi que d'un exposant public que l'on nomme e, de manière standard, il est défini à 65537. Pour le bon fonctionnement du système RSA, les exposants doivent valider certaines conditions. Pour l'exposant public, il doit être un nombre positif strictement supérieur à 1 et strictement inférieur à l'indicatrice d'Euler de n, que l'on nomme Phi de n ou encore $\varphi(n)$, et doit aussi être premier avec celle-ci, c'est à dire que le plus grand diviseur commun entre e et $\varphi(n)$ doit être égale à 1. Il n'est pas nécessaire que le nombre e soit secrètement choisi, car il s'agit d'une clé publique qui sera partagée par tous.
La clé privée se compose du module n et d'un exposant privé que l'on nomme d. Il est calculé à l'aide de l'algorithme d'Euclide étendu afin de trouver l'inverse modulaire de e par rapport à $\varphi(n)$, c'est à dire que le calcul se fait modulo $\varphi(n)$.

Le chiffrement et le déchiffrement d'un message est simple à réaliser, puisqu'il s'agit d'effectuer une exponentiation modulaire.
Si un utilisateur désire envoyer un message que l'on nomme m à une personne, il lui suffit à

l'aide de la clé publique de cette dernière, de calculer le message chiffré que l'on nomme c. L'opération est définie par l'équation : $c \equiv m^{\,e} \, mod \, n$.
Pour déchiffrer le message chiffré, le receveur retrouve m à l'aide de sa propre clé privée. L'opération est définie par l'équation : $m \equiv c^{\,d} \, mod \, n$.
Cependant, même si cette exponentiation modulaire est simple à réalisé, elle est néanmoins coûteuse à effectuer avec des grands nombres comme c'est le cas dans RSA. Le système RSA utilise donc une méthode d'exponentiation rapide modulaire que ce soit à l'aide de l'algorithme 'square and multiply' ou bien à l'aide de la réduction de Montgomery.

Le système RSA étant basé sur un type de chiffrement asymétrique, il permet d'assurer l'authenticité et l'intégrité d'un message en effectuant une signature numérique. Pour signer un message, l'utilisateur utilise sa propre clé privée avec le hash du message, afin de calculer le signé du message que l'on nomme s. L'opération de signature s'effectue de la même manière que pour chiffrer ou déchiffrer un message. Elle est définie par l'équation : $s \equiv H(m)^{\,d} \, mod \, n$.
Lors de la réception du message et du signé, le receveur vérifie l'authenticité et l'intégrité du message à l'aide de sa clé publique. L'opération est définie par l'équation : $H(m) \equiv s^{\,e} \, mod \, n$.
Si le receveur ne retrouve pas le hash de m lors de la vérification de s, alors la signature est dite invalide, et il sait que le message a été intercepté puis modifié.

1.3) Réduction de Montgomery

Notre algorithme précédent a résolu de nombreux problèmes, mais il utilisait des instructions de division pour les opérations modulo, dont nous savons qu'elles prennent beaucoup de temps dans les systèmes informatiques ; nous pouvons donc l'améliorer. L'algorithme de Montgomery a été développé par le mathématicien américain Peter L. Montgomery en 1985 dans son article "Modular Multiplication Without Trial Division" pour montrer comment réaliser des calculs modulo multiplicatifs rapides sans utiliser la division. Nous utiliserons cet algorithme pour réaliser plus rapidement des opérations modulo en puissance. Afin d'éviter la division, l'idée naturelle est d'utiliser une autre opération à la place. Si nous considérons faire des boucles pour soustraire le nombre de modulo, comme on peut imaginer, lorsque le nombre à calculer est très grand, l'opération modulo effectuera trop de cycles de soustraction et la consommation sera similaire à celle de la division. La direction que recherche l'algorithme de Montgomery est donc la possibilité d'utiliser l'addition pour faire le calcule de modulo. Dans ce projet, nous avons étudié cet algorithme et profité ce changement de mode de pensée, que je vais illustrer dans la suite.

Forme de Montgomery:

Prenons comme exemple $43 \times 56 \bmod 97$ que l'on souhaite calculer, comme les deux valeurs données ne sont pas très grandes, on calcule le produit rapidement $43 \times 56 = 2408$. On peut alors calculer le résultat du mode 97 en utilisant la division ou la soustraction. Cela est le calcul normal. Mais au lieu de calculer le module, nous utilisons le produit plus n fois le module 97 de sorte que le 2408 est converti en un nombre qui est plus pratique pour le calcul dans notre esprit humain. Par exemple, un état où le chiffre a la fin est tout à fait nul. A ce point, si on nous demande de calculer $5900 \div 100$, c'est très facile, est équivalent à 59. Malheureusement 59 n'est pas le résultat que nous demandons cet équation: $43 \times 56 \bmod 97$. Mais nous reconnaissons qu'à ce moment-là, notre calcul aurait été beaucoup plus facile si le diviseur avait été 100 plutôt que 97. Ainsi, puisqu'il n'existe pas d'environnement de calcul idéal dans nos connaissances déjà connues, pensons à en créer un. Par exemple, lorsqu'un système de coordonnées cartésiennes ne nous aide pas à modéliser efficacement, nous considérons un système de coordonnées polaires pour modéliser la situation. Les problèmes qui ne peuvent être résolus dans une structure algébrique peuvent être très simples à résoudre dans une autre. L'algorithme de Montgomery crée un tel espace, où 'divise 97' dans le domaine de calcul ordinaire est converti en 'divise 100' dans ce structure algébrique de Montgomery. Donc la difficulté des opérations modulo diminue fortement.

La forme de Montgomery est une manière différente d'exprimer les éléments de l'anneau dans laquelle les produits modulaires peuvent être calculés sans divisions coûteuses. Bien que les divisions soient toujours nécessaires, elles peuvent être effectuées par rapport à un diviseur différent R. Ce diviseur peut être choisi comme étant une puissance de 2, pour laquelle la division peut être remplacée par un décalage à droite bit à bit, ou alors, un nombre entier de mots machine pour lequel la division peut être remplacée par l'omission de mots. Ces divisions sont rapides, de sorte que la majeure partie du coût du calcul des produits modulaires à l'aide de la forme de Montgomery correspond au coût du calcul des produits ordinaires.

Le module auxiliaire R doit être un entier positif tel que $pgcd(R, N) = 1$, soit N le modulo intervenant dans l'opération. Pour des raisons de calcul, il est également nécessaire que la division et la réduction modulo R soient peu coûteuses, le module n'est pas utile pour la multiplication modulaire à moins que $R > N$.

Pour transférer un nombre du domaine de calcul ordinaire de la forme de Montgomery, nous commençons par calculer R. On supposera désormais que N est un nombre impair. (Dans

RSA, le fait que N soit un grand nombre impair crée des problèmes). b est le binaire de calcul, pour notre commodité nous prenons b égal à 10, mais en donnant le calcul à l'ordinateur nous prenons b égal à 2. On calcul R tel que :

$$R \geq N, R = b^m$$

m est le plus petit entier qui satisfait l'équation ci-dessus.
Par exemple, on calcule $43 \times 65 \bmod 97$. $N = 97$. Nous calculons, en décimal, que b = 10 et que le plus petit m qui satisfait à la condition est 2. Alors $R = 100$.

Comme N est impair, R est premier avec N et donc inversible modulo N. On notera R^{-1} l'inverse de $R \, mod \, N$.

Soit la transformation de Montgomery ϕ l'application de $I_n = \{0, 1, ..., n-1\}$ dans lui-même définie par:

$$\phi(a) = a \cdot R \bmod N$$

Cette application ϕ (multiplication par R modulo N) est une bijection de I_n dans lui-même puisque R est inversible modulo N et on peut écrire :

$$a = \phi(a) \cdot R^{-1} \bmod N$$

Maintenant, nous avons l'application pour passer à la forme de Montgomery et en revenir. Dans notre exemple $43 \times 65 \bmod 97$:

Forme ordinaire	Forme Montgomery
43	$43 \times 100 \bmod 97 = 32$
65	$56 \times 100 \bmod 97 = 71$

L'addition et la soustraction sous la forme de Montgomery sont les mêmes que l'addition et la soustraction modulaires ordinaires car $aR \pm bR = (a \pm b)R$. Ceci est une conséquence du fait que, comme $pgcd(R, N) = 1$, la multiplication par R est un isomorphisme sur le groupe additif Z/NZ.

La multiplication sous la forme de Montgomery, cependant, est apparemment plus compliquée. Le produit habituel de aR et de bR ne représente pas le produit de a et de b car il comporte un facteur supplémentaire de R :

$$(aR \bmod N)(bR \bmod N) = abR^2 \bmod N$$

Comme la division par R est peu coûteuse, le produit intermédiaire $(aR \bmod N)(bR \bmod N)$ n'est pas divisible par R car l'opération modulo a annulé cette propriété. Enlever le facteur supplémentaire de R peut être fait en multipliant par un entier R' tel que $RR' \equiv 1(\bmod N)$. L'entier R' existe car R et N sont premier entre eux. Il peut être construit en utilisant l'algorithme euclidien étendu qui détermine les entiers R' et N' qui satisfont le théorème de Bézout. Pour $0 < R' < N, 0 < N' < R$, on a $RR' - NN' = 1$.

Soit $c = a \cdot b \bmod n$, alors:

$$\phi(c) = \phi(a) \cdot \phi(b) \cdot r^{-1}$$

Ceci nous conduit pour calculer $c = a \cdot b \bmod n$ à calculer $\phi(a)$ et $\phi(b)$ pour en déduire $\phi(c)$ par la formule précédente et enfin à trouver c par la transformation de Montgomery inverse.

Montgomery reduction:

La réduction de Montgomery, également connue sous le nom de 'REDC', est un algorithme qui calcule simultanément le produit par R' et réduit modulo N plus rapidement que la méthode naïve. À la différence de la réduction modulaire classique, qui vise à rendre le nombre plus petit que N, la réduction de Montgomery vise à rendre le nombre plus divisible par R. Pour ce faire, elle ajoute un petit multiple de N qui est choisi pour annuler le résidu modulo R. En divisant le résultat par R, on obtient un nombre beaucoup plus petit. Ce nombre est tellement plus petit qu'il est presque la réduction modulo N, et le calcul de la réduction modulo N ne nécessite qu'une soustraction conditionnelle finale. Étant donné que tous les calculs sont effectués en utilisant uniquement des réductions et des divisions par rapport à R, et non à N, l'algorithme s'exécute plus rapidement qu'une simple réduction modulaire par division.

Dans notre exemple précédent $43 \times 65 \bmod 97$, R = 100.

Forme ordinaire	Forme Montgomery
$a = 43$	$a' = 32$
$b = 65$	$b' = 71$

On calcule c' sous la forme Montgomery. Tout d'abord, $c_1 = a' \cdot b' = 32 \cdot 71 = 2272$. D'aprés $a \equiv (a + np) \bmod p, n \in Z$, on a $c_2 = c_1 + 24N = 2272 + 24 \times 97 = 4600$ Au final, on obtient $c' = c_2/R = 4600/100 = 46$

Forme ordinaire	Forme Montgomery
$a = 43$	$a' = 32$
$b = 65$	$b' = 71$
c	$c' = 46$

Afin d'obtenir c de la domaine réaliste, nous devons également effectuer une transformation inverse. Selon notre algorithme précédent:

$$a = \phi(a) \cdot R^{-1} \bmod N$$

Donc $c = c' \cdot R^{-1} \bmod N$. Comme nous l'avons mentionné précédemment, selon le théorème de Bézout, nous avons $RR' - NN' = 1$ tel que $0 < R' < N, 0 < N' < R$. On calcule R' par l'algorithme d'Euclide étendu. Dans notre exemple, $100 \times 65 - 97 \times 67 = 1$. $R^{-1} = 65$.

Donc $a \cdot b \bmod N = a \cdot R^{-1} \bmod N = 46 \times 65 \bmod 97 = 80 = 43 \times 56 \bmod 97$.

Implémentation de RSA avec Montgomery exponentitaion :

L'algorithme du produit de Montgomery est plus approprié lorsque plusieurs multiplications modulaires par rapport au même modulo sont nécessaires. C'est le cas lorsqu'on doit calculer une exponentiation modulaire par exemple dans RSA, c'est-à-dire le calcul de

$$x^e \bmod N$$

L'algorithme d'exponentiation utilise la méthode binaire. L'opération d'exponentiation est remplacée par une suite d'opérations d'élévation au carré et de multiplication modulo n comme le square and multiply sauf qu'au lieu de faire un carré ou une multiplication ordinaire c'est là que l'opération du produit de Montgomery trouve sa meilleure utilisation. Dans la section 'Les algorithmes (pseudo-code) ci-dessous, nous présentons l'opération d'exponentiation modulaire qui utilise la fonction de produit de Montgomery MontgomeryProduct.

II - Conception

2.1) choix de programmation

Au cours de ce projet, nous avons eu à faire face à une multitude de choix de programmation que nous allons détailler.

choix du langage :

Le premier choix a était de déterminer le langage de programmation avec lequel nous allions implémenter notre projet. Notre choix se divisait entre le Java, le C, le C++ et le Python. Après une brève concertation des membres du groupe, nous avons choisit d'adopter le langage C. Les critères de choix ont été : la rapidité d'exécution, l'éventail des libraries cryptographiques et mathématiques disponibles, ainsi que notre maîtrise du langage.
En effet, le langage C contient tous les outils pour nous permettre d'implémenter un cryptosystème RSA efficace. Ce langage possède de nombreuses libraries mathématiques et cryptographiques qui permettent la manipulation des grands nombres, la difficulté étant de pouvoir les stocker et d'appliquer des opérations sur ceux-ci.
Le langage C est également un langage dont tous les membres du groupe ont une maîtrise correcte, ou au moins, dont chacun a déjà eu l'occasion d'utiliser auparavant.
Le Concept Orienté Objet ainsi que la Programmation Orientée Objet n'ont pas été retenu, n'étant pas nécessaire pour la réalisation de ce projet, que ce soit pour l'implémentation du module RSA ou pour l'attaque temporelle. C'est pour cette raison que nous avons écarter les autres langages que sont C++, Java et Python, qui intègrent tous ce concept.
Cette sobriété apporté par le langage C, permettra lors de la réalisation de l'attaque temporelle sur RSA de gagner un temps considérable.

choix des libraries :

Le deuxième choix que nous avions eu à faire, a été le choix des libraries.
Pour la manipulation des grands nombres, le choix c'est porté sur 'GNU Multiple Precision Arithmetic Library' qui est disponible en C sous 'gmplib'. C'est une library sûre, efficace et simple à utiliser. Elle possède notamment une documentation très riche.
Pour utiliser la fonction de hachage présente dans PKCS#1 v1.5, ainsi que pour utiliser le

protocole de communication SSL/TLS, nous avons choisi d'intégrer OpenSSL qui est une des libraries les plus utilisées en cryptographie aujourd'hui et qui dispose d'une documentation riche. Elle est divisée en deux parties, avec une partie orientée cryptographie apportée par la library 'libcrypto' qui comporte des algorithmes cryptographiques comme des fonctions de hachage cryptographique, et d'une autre partie orientée uniquement sur le protocole de communication SSL/TLS apportée par la library 'libssl'.

choix des algorithmes :

Le troisième choix à été de choisir les différents algorithmes à utiliser pour l'implémentation de RSA.
Tout d'abord, nous avons choisi d'implémenter l'algorithme de Miller-Rabin pour la génération des nombres premiers à la place de l'algorithme de Solovay-Strassen. L'algorithme de Miller-Rabin à été d'une part plus facile à comprendre, et d'autre part plus simple à implémenter que celui de Solovay-Strassen.
Ensuite, nous avons choisi d'utiliser le standard PCKS#1 v1.5 pour l'utilisation d'un padding lors du chiffrement et lors de la signature d'un message. Bien que nous sachions que cette version du standard n'est plus sûre d'un point de vue sécurité et qu'il existe une attaque cryptographique possible comme celle de Daniel Bleichenbacher appelée aussi 'million message attack' découverte en 1998, nous avons choisi ce standard car il est très simple a implémenter. Néanmoins, nous sommes conscients de la faille que présente cette version du standard et nous pouvons éventuellement passer à une version supérieure du standard comme celle de PKCS#1 v2.2 qui intègre le padding OAEP 'Optimal Asymmetric Encryption Padding' qui est beaucoup plus sécurisé.
Enfin, pour le hachage du message utilisé dans PKCS#1 v1.5 pour la signature, nous avons décider d'utiliser la fonction de hachage SHA-256 car elle est plus sûre que les fonctions de hachage SHA-1 ou SHA-2 qui présente des failles de sécurité qui peuvent être exploitées pour déjouer l'authentification.

III - Réalisation de la Timing Attack

Avec la découverte de fuites d'informations d'algorithmes de chiffrement et de déchiffrement, des attaques par canaux secrets sont apparues.
En effet, les signaux physiques peuvent fournir des informations sur le programme. Les exemples incluent la température, l'éclairage, la consommation d'énergie et le **temps d'exécution**.
Le principe est de mesurer ces fuites et de les analyser, à partir de ces analyses on peut calculer et reconstituer la clé de chiffrement puis attaquer le système d'information.

L'attaque que nous avons réalisé est ce qu'on appelle une "timing attack" (attaque temporelle), c'est une attaque très efficace qui a causé beaucoup d'inquiétude à l'époque, avant que les mathématiciens, informaticiens et cryptologues ne trouvent des contre-mesures efficace pour protéger les systèmes d'information contre ce type d'attaque particulier.
En 1996, Kocher a montré qu'on pouvait casser un algorithme RSA à partir du temps d'exécution enregistré lors du calculs d'exponentiation modulaire du RSA.
Brumley et Tuveri ont proposés une attaque temporelle sur l'algorithme de Montgomery. La vulnérabilité exploitée se concentre sur la réduction supplémentaire qui se trouve dans la réduction modulaire de Montgomery

Dans cette partie de notre projet, nous allons expliquer le fonctionnement de l'attaque temporelle sur le cryptosystème RSA que nous avons implémenté et qui utilise l'exponentiation de Montgomery avec réduction.
Enfin, nous allons donner quelques contre-mesures contre cette attaque.

3.1) Fonctionnement de la Timing Attack sur le modèle RSA implémenté

Cette attaque est réalisable dans l'hypothèse où l'attaquant a à sa disposition le matériel et les outils nécessaire pour mesurer les opérations du cryptosystème lors du déchiffrement et où il connaît l'algorithme est utilisé.
On suppose dans notre cas qu'on sait que l'algorithme est utilisé RSA avec la réduction de Montgomery. L'exponentiation de Montgomery contient la fonction Montgomery_product qui est la cible de notre attaque.

Voici le code de la fonction :

```
if((mpz_cmp(t, n) == 0) || (mpz_cmp(t,n) > 0)) // t >= n
{
   mpz_sub(t, t, n); // t = t - n

   nanosleep(&time_if, NULL);
}
```

Cette fonction permet d'effectuer la réduction de Montgomery. Si le résultat de la multiplication est supérieur ou égal à la valeur du module n alors la condition du if est vrai et l'on effectue la réduction, sinon on ne fait rien.

C'est cette instruction que nous avons mesuré pour effectué l'attaque temporelle. En effet, si la réduction est faite alors le temps mesuré est supérieur ou égale au temps passé dans le if sinon le temps mesuré est strictement inférieur au temps passé dans le if.
Ici nous avons volontairement ajouté une fonctions qui ralentit le programme si l'instruction if est effectué. Dans notre cas, nous définissons un temps minimum de 4 microsecondes. Ainsi si le temps mesuré est d'au minimum 4 microsecondes nous savons que la réduction a été effectuée.

Montgomery_product est utilisée dans la fonction Montgomery_Exponentiation qui effectue l'exponentiation modulaire de Montgomery. Elle contient une boucle dans laquelle la fonction Montgomery_product est effectue une fois pour l'élévation au carré et une autre fois pour la multiplication. Il s'agit d'une boucle de k itérations, avec k la taille de l'exposant secret d que l'on souhaite récupérer). Comme la multiplication dépend de la valeur du bit de l'exposant secret, si la valeur du bit de $d_k - i$ vaut 1 alors le deuxième Montgomery_product sera exécuté. Si la valeur du bit de $d_k - i$ vaut 0 alors le deuxième Montgomery_product ne sera pas exécuté.

Nous nous concentrons sur le deuxième Montgomery_product car le temps de la multiplication de Montgomery est plus remarquable puisque que cette multiplication consiste à multiplier deux nombres différents donc la réduction à une chance plus grande de se réaliser.

L'attaque se déroule comme suit :

- Tout d'abord, on définit deux ensembles A et B. L'ensemble A, où seront stockés tous les temps mesurés inférieures au temps de l'exécution du if et l'ensemble B où seront stockés tous les temps qui seront supérieure au temps d'exécution du if.

- On mesure le temps d'exécution du if dans la fonction Montgomery_product pour chaque $d_k - iavec1 < i < k - 1$.

- On répartie les temps selon une limite de répartition de 4 microsecondes qui est la durée du temps ajouté dans l'instruction de la réduction de Montgomery. Tous les temps inférieurs à 4 microsecondes seront ajouter dans l'ensemble A et tous les temps supérieur ou égale seront ajouter dans l'ensemble B. Cependant, pour réduire le temps aléatoire supplémentaire qui se rajoute aux 4 microsecondes nous devons effectuer un grand nombre de mesure avec plusieurs message qui ciblerait chaque bit de l'exposant secret d. Il y a toujours du bruit dans un programme, et ce bruit est dû à des instructions aléatoires qui sont exécutées en parallèle lors de la mesure du temps. La même instruction peut avoir des intervalles de temps différents à chaque fois qu'elle est exécutée. Pour résoudre ce problème et réduire le temps du bruit à une valeur négligeable, plusieurs clairs-chiffrés doivent être utilisés (par exemple 100 itérations ou plus).

- Ensuite, on calcule le temps moyen de l'ensemble A que l'on nomme T_a et de l'ensemble B que l'on nomme T_b. - On calcul la valeur absolue de la soustraction entre ces deux moyennes $|T_b$ - $T_a|$. Si la valeur de cette soustraction est strictement inférieure à la valeur absolue de la soustraction du temps total des ensembles A et B, on en déduit que le bit de l'exposant secret $d_k - i$ vaut 1 sinon le bit vaut 0.
Nous remplissons un tableau préalablement initialiser à 0 sauf la première case qui représente la valeur du bit $d_k - 1$ de l'exposant secret d dont on connaît la valeur qui est 1.
Ainsi si la condition ci-dessus est vérifiée on change la valeur 0 du bit concerné à 1 et on passe au bit suivant, sinon on garde 0.

- Enfin en appliquant la stratégie que nous venons d'expliquer à tous les bits de la clé, nous finirons par trouver une représentation binaire de la clé, puis de la clé elle-même. Ce qui signifie que nous avons réussi à attaquer l'algorithme RSA de manière indirecte.

3.2) Les algorithmes (pseudo-code)

Sans l'attaque :

- MongomeryProduct

function MonPro($\bar{a}, \bar{b}$)
Step 1. $t := \bar{a} \cdot \bar{b} \bmod r$
Step 2. $m := t \cdot n' \bmod r$
Step 3. $u := (\bar{a} \cdot \bar{b} + m \cdot n)/r$
Step 4. **if** $u \geq n$ **then return** $u - n$ **else return** u

- MongomeryExponenetiation

function ModExp(a, e, n) { n is an odd number }
Step 1. Compute n' using the extended Euclid algorithm.
Step 2. $\bar{a} := a \cdot r \bmod n$
Step 3. $\bar{x} := 1 \cdot r \bmod n$
Step 4. **for** $i = k - 1$ **down to** 0 **do**
Step 5. $\bar{x} := \text{MonPro}(\bar{x}, \bar{x})$
Step 6. **if** $e_i = 1$ **then** $\bar{x} := \text{MonPro}(\bar{a}, \bar{x})$
Step 7. $x := \text{MonPro}(\bar{x}, 1)$
Step 8. **return** x

Avec l'attaque :

- MongomeryProduct

```
//############################-TIMING ATTACK-#################################//
   double tta = 0.0, tta_cpu = 0.0;
   clock_t tta_cpu_deb = 0, tta_cpu_fin = 0;
   struct timespec tta_deb = {0,0}, tta_fin = {0,0};

   struct timespec time_if = {0,2000}; //2 micro secondes

   if(TIMING_ATTACK_CONFIRMED && DECRYPT)
      debut_chrono(&tta_cpu_deb,&tta_deb);
//#############################################################################//

if((mpz_cmp(t, n) == 0) || (mpz_cmp(t,n) > 0))
{
   mpz_sub(t, t, n); // t = t - n

   nanosleep(&time_if, NULL); //attend 2 micro secondes
}

//############################-TIMING ATTACK-#################################//
   if(TIMING_ATTACK_CONFIRMED && DECRYPT)
   {

      fin_chrono(&tta_cpu,tta_cpu_deb,tta_cpu_fin,&tta,tta_deb,tta_fin);

      ELEMENT* elem = initialiser_element(tta_cpu);

      if(elem->temps < LIMITE){ //si pas de if et si le nanosleep n'a pas été fait
         ajouter_element(elem, &A, target_bit);
      }
      else{
         ajouter_element(elem, &B, target_bit);
      }

      calculer_temps_moyen(&A);  //calcul Ta
      calculer_temps_moyen(&B);  //calcul Tb

      //met le résultat de la difference dans la variable globale T
      calculer_difference_temps_moyen(&A, &B);
   }
//#############################################################################//
```

- MongomeryExponenetiation

```
    if(!(mpz_cmp_ui(andr, 1)))
    {
        //##########################-TIMING ATTACK-################################//
        TIMING_ATTACK_CONFIRMED = 1;  //active le timing attack

        if(k <= taille - 1 && DECRYPT)  //on commence à dk-2
            target_bit = k - 1;
        //#########################################################################//

        Montgomery_product(v, a_bar, x_bar, n, x_bar, N_SIZE); // multiply

        //##########################-TIMING ATTACK-################################//
        TIMING_ATTACK_CONFIRMED = 0;  //désactive le timing attack
        //#########################################################################//
    }
}
```

3.3) Comparaisons entre différentes implémentations RSA

Type RSA	Nbr max des msg par bit	Nbr total des msg	Temps d'exécution réel
1024 bits sans padding	≈ 50	≈ 28000	réel = 100.00 s
1024 bits avec padding	≈ 50	≈ 28000	réel = 105.00 s

Nous constatons qu'avec le même nombre d'itérations (nombre de messages) le temps d'exécution est différents selon le nombre de bits et si le padding est fait ou pas.

IV - Conclusion

4.1) Bilan

Ce projet comporte 2 versions pour la timing attack. La première consiste à effectuer l'attaque sur tout les bits de l'exposant secret d en même temps, la seconde effectue l'attaque sur l'exposant d bit à bit. Les deux versions marchent sauf que la seconde est beaucoup plus longue en terme d'execution puisqu'elle fait l'attaque sur chaque bit. Cette réalisation a été une expérience très enrichissante et il nous a vraiment amené à découvrir des aspects fondamentaux très utiles de la sécurité informatique qui sont de plus en plus importants pour nous en tant qu'informaticiens. Nous avons examiner l'une des cyber-attaques les plus dangereuses et l'avons implémenter et montrer comment récupérer l'exposant secret d. Toutefois, il existe des contre-mesures pour garantir la sûreté de l'algorithme contre de telles attaques.

4.2) Améliorations possibles

Nous pouvons améliorer notre projet pour rendre la sécurité de notre RSA encore plus efficace, pour cela nous pouvons améliorer la version du padding PCKS1 v1.5 en le remplaçant par le padding PCKS1 v2.0 ou v2.2 qui intègre comme mentionné précédemment le schéma de padding OAEP (Optimal Asymmetric Encryption Padding) qui évite l'attaque sensible de Daniel Bleichenbache sur PCKS1 v1.5. Par manque de temps nous n'avons pas pu réalisé l'attaque temporelle sur RSA en utilisant le protocole SSL/TLS. Il est donc possible d'intégrer cette fonctionnalité dans notre programme.

V - Bibliographie

References

[1] **Chiffement RSA :**
https://en.wikipedia.org/wiki/RSA_(cryptosystem)

[2] **Description Montgomery :**
https://en.wikipedia.org/wiki/Montgomery_modular_multiplication

[3] **Fonctionnement Montgomery :**
http://koclab.cs.ucsb.edu/teaching/cs154/docx/Notes7-Montgomery.pdf

[4] **Padding OAEP :**
https://en.wikipedia.org/wiki/Optimal_Asymmetric_Encryption_Padding

[5] **Test de primalité Miller-Rabin :**
https://en.wikipedia.org/wiki/Miller\OT1\textendashRabin_primality_test

[6] **Factorisation de Fermat :**
https://en.wikipedia.org/wiki/Fermat's_factorization_method

[7] **RSA version 2.2 :**
https://www.rfc-editor.org/rfc/rfc8017