Analyse de la RSA avec la Multiplication de Montgomery

1er décembre 2023

Table des matières

Introduction

La sécurité des communications numériques est devenue une préoccupation cruciale à l'ère de l'information, où les échanges électroniques de données sensibles sont omniprésents. La cryptographie joue un rôle essentiel dans la protection de ces données, et parmi les algorithmes cryptographiques les plus fondamentaux et répandus, la RSA (Rivest-Shamir-Adleman) occupe une place centrale. La RSA repose sur la difficulté de factoriser de grands nombres premiers, un problème considéré comme intraitable en temps polynomial avec les ressources informatiques actuelles.

Cependant, la performance de la RSA est étroitement liée à la rapidité de ses opérations arithmétiques, en particulier la multiplication de grands entiers. La multiplication de Montgomery, introduite par Peter L. Montgomery dans les années 1980, offre une approche novatrice pour accélérer ces opérations, en particulier dans le contexte de la RSA. Cette méthode s'est avérée particulièrement efficace pour réduire le coût computationnel de la RSA, offrant ainsi des gains significatifs en termes de temps d'exécution.

Ce rapport vise à explorer en profondeur la relation entre la RSA et la multiplication de Montgomery, en mettant en lumière les avantages, les défis et les considérations de sécurité associés à cette approche. Nous examinerons les principes fondamentaux de la RSA et de la multiplication de Montgomery, discuterons des scénarios d'application appropriés, et évaluerons l'impact de cette méthode sur la sécurité globale des systèmes cryptographiques.

À travers cette analyse approfondie, nous chercherons à fournir des insights pertinents pour les professionnels de la sécurité informatique, les chercheurs en cryptographie et toute personne intéressée par les avancements actuels dans le domaine de la protection des données numériques.

RSA Optimisé avec Montgomery

Introduction au RSA avec la multiplication de Montgomery

Principe du fonctionnement (Algorithme)

Introduction

La sécurité des systèmes d'information repose de plus en plus sur des méthodes de cryptographie robustes, et parmi celles-ci, l'algorithme RSA occupe une place prépondérante. L'efficacité de RSA dépend en grande partie de l'opération de réduction modulo, une étape cruciale dans le processus de chiffrement et de déchiffrement. L'une des techniques les plus avancées pour améliorer cette réduction est la Réduction de Montgomery.

Cette section se consacre à une exploration approfondie de la Réduction de Montgomery appliquée à l'algorithme RSA. Nous plongerons dans les fondements théoriques de cette méthode, mettant en lumière son impact sur l'accélération des opérations modulo. En comprenant les nuances de la Réduction de Montgomery, nous pourrons apprécier son rôle essentiel dans l'optimisation des performances de l'algorithme RSA.

Algorithme de Réduction de Montgomery

Présentation du problème, notation

Étant donnés trois nombres entiers a, b, n, le problème est de calculer le plus rapidement possible $a \times b \mod n$. L'algorithme de Réduction de Montgomery se distingue particulièrement lorsqu'il est appliqué à des chaînes de multiplications, ce qui est fréquent dans le cas de l'exponentiation modulaire.

En conclusion, il faut considérer l'algorithme de Montgomery comme s'insérant dans l'algorithme d'exponentiation modulaire du calcul de $a^n \mod n$.

La transformation de Montgomery

2.1 Définition de la transformation - lien avec le produit

Soit n le modulo intervenant dans l'opération. On supposera désormais que n est un nombre impair. Le nombre de bits de n est l'entier k tel que :

$$2^{k-1} < n < 2^k$$

On appellera r le nombre 2^k. Comme n est impair, r est premier avec n et donc inversible modulo n. On notera r^{-1} l'inverse de r modulo n.

Soit Φ (transformation de Montgomery) l'application de $I_n = \{0, 1, \ldots, n-1\}$ dans lui-même définie par :

$$\Phi(a) = a \times r \mod n$$

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 :

$$a = \Phi(a) \times r^{-1} \mod n$$

Le théorème qui suit nous indique comment s'exprime le transformé d'un produit de deux termes en fonction des transformés de chacun de ces deux termes.

Théorème 2.1 Soit $c = a \times b \mod n$. Alors :

$$\Phi(c) = \Phi(a) \times \Phi(b) \times r^{-1} \mod n$$

Preuve. Calculons le second membre de l'égalité à prouver :

$$\Phi(a) \times \Phi(b) \times r^{-1} \mod n = a \times r \times b \times r \times r^{-1} \mod n;$$

$$\Phi(a) \times \Phi(b) \times r^{-1} \mod n = a \times b \times r \mod n = \Phi(c) \times r \mod n = \Phi(c)$$

Ceci nous conduit pour calculer $c = a \times b \mod n$ à calculer $\Phi(a)$ et $\Phi(b)$, à en déduire $\Phi(c)$ par la formule précédente et enfin à trouver c par la transformation de Montgomery inverse.
Comment se calcule le transformé d'un prod

On calcule une fois pour toutes r^{-1} par l'algorithme d'Euclide étendu, et pendant qu'on y est, on calcule aussi l'autre coefficient de Bézout, c'est-à-dire qu'on calcule r^{-1} et v tels que :

$$r \times r^{-1} - v \times n = 1;$$

où $0 < r^{-1} < n$ et $0 < v < r$.

Notons $\odot$ l'opération sur $\{0, 1, \ldots, n-1\}$ définie par :

$$A \odot B = A \times B \times r^{-1} \mod n;$$

Alors, nous avons montré que :

$$\Phi(a \times b \mod n) = \Phi(a) \odot \Phi(b);$$

Le calcul de $A \odot B$ peut s'effectuer de la manière suivante :

Algorithme pour Calculer $A \odot B$

1. $s = A \times B$,
2. $t = (s \times v) \mod r$,
3. $m = (s + t \times n)$,
4. $u = m/r$,
5. si $u \geq n$, alors $A \odot B = u \mod n$; sinon, $A \odot B = u$.

En effet, il existe un entier k tel que :

$$t = s \times v + k \times r;$$

On calcule alors :

$$m = (s + t \times n) = s + s \times v \times n + k \times r \times n;$$
$$m = (s + t \times n) = s + s \times (r \times r^{-1} - 1) + k \times r \times n;$$
$$m = (s + t \times n) = s \times r \times r^{-1} + k \times r \times n;$$
$$u = s \times r^{-1} + k \times n$$

Donc :

$$u \equiv A \odot B \pmod{n}$$

Mais on voit que $0 \leq t < r$, donc :

$$0 \leq m < n \times (r + n) < 2 \times r \times n;$$

et

$$0 \leq u < 2 \times n$$

Ceci prouve qu'il n'y a que deux possibilités : soit $A \odot B = u$, soit $A \odot B = u \mod n$ suivant que $0 \leq u < n$ ou $n \leq u < 2n$.

Dans cet algorithme, on ne fait pas de division par n, mais par r, ce qui change tout puisque r est une puissance de 2. Bien entendu, pour une seule opération, ceci n'est pas intéressant puisqu'il y a plusieurs calculs préparatoires à faire. Mais si on enchaîne un grand nombre d'opérations, alors on devient gagnant.

RSA avec Multiplication de Montgomery

L'algorithme RSA comprend la génération de clés, le chiffrement et le déchiffrement. Au cœur de ces opérations se trouve l'exponentiation modulaire ($X^Y \mod N$), qui peut être coûteuse en termes de calcul. Nous introduisons la Multiplication de Montgomery comme une méthode pour optimiser l'étape d'exponentiation modulaire.

- C : Ciphertext (Texte chiffré)
- P : PlainText (Texte en clair)
- e : Exposant public
- d : Exposant privé
- N : modulo

.

Les opérations de chiffrement et de déchiffrement sont définies comme suit :

$$C = P^e \mod N \quad \text{(chiffrement avec la clé publique)},$$
$$P = C^d \mod N \quad \text{(déchiffrement avec la clé privée)}.$$

La clé publique est (e, N) et la clé privée est (d, N).

Génération du Modulo (N)

Choisissez deux nombres premiers ayant une demi-longueur en bits des clés publique/privée souhaitées.

- p : nombre premier
- q : nombre premier
- $N = p \times q$

Génération de l'Exposant Public (e)

L'exposant public (e) est un petit entier. Des choix valides sont 3, 5, 7, 17, 257 ou 65537. Avec les clés RSA, l'exposant public est généralement 65537.

Les exigences pour e sont :

- $1 < e < \phi(N) = (p-1) \times (q-1)$
- e et la fonction ϕ d'Euler de N sont premiers entre eux.

Génération de l'Exposant Privé (d)

L'exposant privé est généré à partir des nombres premiers p et q et de l'exposant public.

$$d = e^{-1} \mod (p-1) \times (q-1)$$

Exponentiation Modulaire Efficace (en utilisant la Multiplication de Montgomery

L'algorithme de multiplication et carré « square and multiply » pour l'exponentiation modulaire nécessite une multiplication modulaire sous la forme :

— $Z = X \times X \mod N$
— $Z = X \times Y \mod N$

.

Cet algorithme consiste à décomposer l'exposant en une séquence de carrés et de multiplications. En intégrant la multiplication de Montgomery, cet algorithme devient encore plus efficace.

L'algorithme de multiplication et carré

```
fonction exponentiation_modulaire_montgomery(base, exposant, n):
    si n == 1:
        retourner 0

    resultat = 1
    base = base % n

    tant_que exposant > 0:
        si exposant % 2 == 1:
            resultat = multiplication_montgomery(resultat, base, n)
        exposant = exposant >> 1
        base = multiplication_montgomery(base, base, n)

    retourner resultat
```

Avantages et Limitations

Avantages

L'un des principaux avantages de la Réduction de Montgomery réside dans son impact sur les opérations de multiplication modulaire, qui sont au cœur de l'algorithme RSA. En remplaçant les multiplications modulaires standard par la multiplication de Montgomery,

l'algorithme devient plus efficace, réduisant ainsi la charge computationnelle associée aux opérations de chiffrement et de déchiffrement.

1 Réduction des Opérations de Division

La Réduction de Montgomery permet également de réduire le nombre d'opérations de division, qui sont généralement coûteuses en termes de calcul. En transformant les opérations de division en des opérations plus légères, l'algorithme devient plus rapide et plus adapté à des environnements où les ressources sont limitées.

2 Amélioration de la Sécurité

Bien que la Réduction de Montgomery soit principalement introduite pour ses avantages en termes de performance, elle contribue également à renforcer la sécurité de l'algorithme RSA. En optimisant les opérations de base, l'algorithme devient plus résistant aux attaques par canal auxiliaire, améliorant ainsi sa robustesse face à diverses menaces potentielles.

3 Adaptabilité aux Contraintes de Puissance de Calcul

La Réduction de Montgomery rend l'algorithme RSA plus adaptable aux contraintes de puissance de calcul, ce qui le rend particulièrement attrayant pour les systèmes embarqués et les environnements où les ressources sont limitées. Cette adaptabilité en fait un choix viable pour des applications variées, allant des dispositifs IoT aux serveurs de calcul intensif.

Limitations

Complexité de mise en œuvre : La mise en œuvre correcte de la réduction de Montgomery peut être complexe, nécessitant une compréhension approfondie des mathématiques sous-jacentes. Des erreurs d'implémentation pourraient compromettre la sécurité globale du système.

Possibilité d'erreurs : Une mauvaise utilisation de la réduction de Montgomery peut entraîner des erreurs de calcul. Ces erreurs pourraient être exploitées par des attaquants pour compromettre la confidentialité ou l'intégrité des données.

Sensibilité aux attaques latérales : Bien que la réduction de Montgomery puisse aider à réduire la vulnérabilité aux attaques temporelles, elle n'élimine pas complètement la sensibilité aux attaques latérales. Des précautions supplémentaires peuvent être nécessaires pour garantir la sécurité contre ces types d'attaques.

Conclusion

En conclusion, l'utilisation de la multiplication de Montgomery dans l'exponentiation modulaire, notamment avec l'algorithme « square and multiply », peut considérablement optimiser les opérations modulo et améliorer les performances globales de l'algorithme.

Exemple illustratif (Chiffrement et Déchiffrement)

La génération de clés RSA est une étape cruciale dans la mise en place d'un système de chiffrement robuste. Dans ce rapport, nous détaillerons le processus de génération de clés RSA en utilisant la multiplication de Montgomery pour accélérer les calculs modulaires. Pour illustrer chaque étape, nous prendrons des valeurs spécifiques

1. Choix des Nombres Premiers
   Nous avons choisi deux nombres premiers, $p = 17$ et $q = 19$, pour générer notre modulo $N = pq$ Ainsi $N = 19 \times 17 = 323$
2. Calcul de $\Phi(N)$
   Le calcul de $\Phi(N)$ est essentiel , $\Phi(N) = (p-1)(q-1) = 16 \times 18 = 288$
3. Choix de la Clé Publique
   Nous avons opté pour e=5 comme clé publique.
4. Calcul de la Clé Privée avec la Multiplication de Montgomery

   (a) Initialisation de Montgomery
   Nous avons $2^8(256)$ comme une puissance de 2 plus grande que N , et calculé $R^{-1}modN = 137$

   (b) Conversion en Représentation de Montgomery
   eR mod N) 1275 et $\Phi(N)$R mod N) 8

   (c) Multiplication
   t =221 résulte de eR . $\Phi(N)$R mod N

   (d) Conversion de Retour
   Le résultat final en représentation normale est 25
5. Clé Publique et Privée RSA
   La clé publique est $(N, e) = (323, 5)$, et la clé privée est $(N, d) = (323, 25)$

6. Chiffrement RSA
   Maintenant que nous avons généré avec succès notre paire de clés RSA, nous pouvons procéder au chiffrement d'un message. Supposons que notre message clair M soit 42
   (a) Conversion en Représentation de Montgomery
   Calculer $MR \mod N = 42 \times 256 \mod 323 = 280$
   (b) Chiffrement
   Le message chiffré C est obtenu en calculant $C \equiv MR^e \mod N$. Dans notre cas $C \equiv 280^5 \mod 323$
7. déchiffrement RSA
   Passons maintenant au processus de déchiffrement du message chiffré C.
   (a) Conversion en Représentation de Montgomery
   Convertir C en représentation de Montgomery $CR \mod N = 280 \times 256 \mod 323 = 140$
   (b) Déchiffrement
   Calculons $M \equiv CR^d \mod N$ ou $d = 25$. $M \equiv 140^{25} \mod 323$ Cependant, effectuer ce calcul directement pourrait être fastidieux. Utilisons l'exponentiation modulaire pour simplifier le processus. $M \equiv (140^5)5 \mod 323$ calculons $M \equiv 140^5 \mod 323$ en utilisant la multiplication de Montgomery.
   (c) Initialisation :
   — Choisissons $R = 2^8 et R^{-1} \mod 137$
   — Convertissons 140 en représentation de Montgomery , $140 \times 256 \mod 323 = 252$
   (d) Exponentiation modulaire :
   — $140^2 \mod 323 = 64$(Calcul en représentation de Montgomery)
   — $140^4 \mod 323 = (64)^2 \mod 323 = 1$(Calcul en représentation de Montgomery)
   Ainsi , $140^5 \mod 323 = 140 \times (140)^4 \mod 323 = 140 \times 1 \mod 323 = 140$ Par conséquent $M \equiv (140)^5 \mod 323$
8. Conversion de Retour
   Si nécessaire, convertissez le résultat en représentation normale en multipliant par $R^{-1} \mod N$. Dans notre exemple, cela pourrait être $140 \times 137 \mod 323 = 295$.

La multiplication de Montgomery a été utilisée avec succès pour accélérer les calculs modulaires dans le processus de génération de clés RSA. Cette méthode permet d'optimiser les performances tout en garantissant la sécurité du système de chiffrement.

.

Cryptanalyse

La cryptanalyse est définie comme le déchiffrement de messages chiffrés dont on ne connaît pas le code. C'est une analyse qui consiste à retrouver le clair d'un message chiffré au moyen

d'algorithmes. Elle est la technique qui consiste à déduire un texte en clair d'un texte chiffré sans posséder la clé de chiffrement2. En d'autres termes, c'est l'ensemble des techniques mises en œuvre pour tenter de déchiffrer un message codé dont on ne connaît pas la clé.
La sécurité du RSA repose sur la difficulté de factoriser de grands nombres. Cependant, la cryptanalyse joue un rôle important en cherchant à exploiter les éventuelles faiblesses du cryptosystème RSA. Par exemple, si les paramètres de sécurité sont mal choisis ou s'ils vérifient certaines relations, un attaquant pourrait en tirer profit pour casser le cryptosystème.

La cryptanalyse de RSA avec la multiplication de Montgomery présente des défis uniques. La multiplication de Montgomery est une méthode qui accélère les opérations de chiffrement et de déchiffrement dans RSA en transformant les nombres dans un espace appelé le "Monde de Montgomery", où les multiplications modulaires peuvent être effectuées plus rapidement. Cependant, cette optimisation peut également offrir de nouvelles opportunités aux attaquants.

Si un attaquant peut déterminer que la multiplication de Montgomery est utilisée, il peut essayer d'exploiter les caractéristiques spécifiques de cette méthode pour obtenir des informations sur la clé privée. Par exemple, l'attaquant pourrait analyser le temps nécessaire pour effectuer certaines opérations ou les variations de consommation d'énergie associées à ces opérations. Ces informations pourraient potentiellement être utilisées pour déduire des informations sur la clé privée.

En outre, il est important de noter que toutes les attaques courantes contre RSA, telles que l'attaque par force brute ou l'attaque par factorisation, restent applicables même lorsque la multiplication de Montgomery est utilisée. Par conséquent, la sécurité globale du système dépend toujours de la taille de la clé et de la sécurité de sa mise en œuvre.

Il est donc crucial de bien comprendre ces défis lors de l'implémentation de RSA avec la multiplication de Montgomery et de prendre les mesures appropriées pour atténuer ces risques. Cela peut inclure l'utilisation de contre-mesures pour prévenir les attaques par canal auxiliaire et s'assurer que l'implémentation de la multiplication de Montgomery est correcte et ne laisse pas de traces exploitables.

En fin de compte, l'objectif de la cryptanalyse est de tester la résistance d'un cryptosystème à diverses attaques, d'identifier et de corriger les faiblesses, afin d'améliorer la sécurité globale du système. C'est un aspect crucial pour préserver l'intégrité et la confidentialité des données dans notre monde numérique.

Classification des attaques :

- **Attaque sur texte chiffré seul (ciphertext-only) :** le cryptanalyste possède des exemplaires chiffrés des messages, il peut faire des hypothèses sur les messages originaux qu'il ne possède pas. La cryptanalyse est plus ardue de par le manque d'informations à disposition.
- **Attaque à texte clair connu (known-plaintextattack) :** le cryptanalyste possède des messages ou des parties de messages en clair ainsi que les versions chiffrées. La cryptanalyse linéaire fait partie de cette catégorie.
- **Attaque à texte clair choisi (chosen-plaintextattack) :** le cryptanalyste possède des messages en clair, il peut générer les versions chiffrées de ces messages avec l'algorithme que l'on peut dès lors considérer comme une boîte noire. La cryptanalyse différentielle est un exemple d'attaque à texte clair choisi.
- **Attaque à texte chiffré choisi (chosen-ciphertextattack) :** le cryptanalyste possède des messages chiffrés et demande la version en clair de certains de ces messages pour mener l'attaque.

Cryptanalyse de RSA connaissant (N) :

Proposition :
Soit N un module RSA. Si on connaît $\phi(N)$, alors on peut factoriser N.

Démonstration :
Supposons que $\phi(N)$ est connu. Ainsi, on dispose d'un système de deux équations en p et q :

$$\begin{aligned} pq &= N, \\ p + q &= N + 1 - \phi(N), \end{aligned}$$

Qui donnent l'équation en p :

$$p^2 - (N + 1 - \phi(N))p + N = 0.$$

On obtient ainsi :

$$\begin{aligned} p &= \frac{N + 1 - \phi(N) + \sqrt{(N + 1 - \phi(N))^2 - 4N}}{2}, \\ q &= \frac{N + 1 - \phi(N) - \sqrt{(N + 1 - \phi(N))^2 - 4N}}{2}. \end{aligned}$$

Cryptanalyse de RSA dans le cas de l'Utilisation du même module et deux exposants différents :

Proposition :
Si un message clair M est chiffré avec (N, e_1) et (N, e_2) avec $\text{pgcd}(e_1, e_2) = 1$, alors on peut calculer M.

Démonstration :
Si $\text{pgcd}(e_1, e_2) = 1$, alors $e_1x_1 - e_2x_2 = 1$. Supposons :

$$C_1 \equiv M^{e_1} \pmod{N},$$
$$C_2 \equiv M^{e_2} \pmod{N},$$

Alors :

$$C_1^{x_1} \cdot C_2^{-x_2} \equiv M^{e_1x_1} \cdot M^{-e_2x_2} \equiv M^{e_1x_1 - e_2x_2} \equiv M \pmod{N}.$$

Cryptanalyse de RSA si $|p - q| < cN^{1/4}$) : Méthode de Fermat

Dans cette partie, nous supposons que les nombres premiers p et q qui forment le module RSA $N = pq$ sont très proches, plus précisément $|p - q| < cN^{1/4}$ où c est une constante fixe, assez petite.

Théorème : Soit $N = pq$ un module RSA où les nombres premiers p et q vérifient $|p - q| < cN^{1/4}$ où c est une constante assez petite. Alors on peut factoriser N en temps polynomial dépendant de c.

Démonstration. La méthode de Fermat consiste à la recherche de deux nombres entiers x et y tels que

$$4N = x^2 - y^2 = (x + y)(x - y).$$

Si en plus $x - y \neq 2$, alors on obtient la factorisation de N en posant

$$p = \frac{x + y}{2}, \quad q = \frac{x - y}{2}.$$

Pour déterminer x et y, on prend pour x les valeurs $x_0 = 2\sqrt{N}$, $x_1 = 2\sqrt{N}+1$, $x_2 = 2\sqrt{N}+2$, ..., et on teste si $4N - x^2$ est un carré parfait. On désigne alors par k le nombre entier pour lequel $x_k = 2\sqrt{N} + k$ donne la factorisation de N.

Alors $x_k = p + q$ et on a, en supposant que $|p - q| < cN^{1/4}$:

$$\begin{aligned} k &= x_k - 2\sqrt{N} \\ &= p + q - 2\sqrt{N} \\ &< p + q - 2\sqrt{N} + 1 \\ &= \frac{(p+q)^2 - 4N}{p + q + 2\sqrt{N}} + 1 \\ &= \frac{(p-q)^2}{p + q + 2\sqrt{N}} + 1 \\ &< \frac{c^2\sqrt{N}}{2\sqrt{N}} + 1 \\ &< \frac{c^2}{2} + 1 \end{aligned}$$

Il en résulte que le nombre de tests est assez petit si c est une constante qui ne dépend pas de N.

Mesures de sécurité :

- **Choix des clés :** Assurez-vous que les clés privées sont suffisamment longues pour résister aux attaques par force brute. De plus, évitez d'utiliser des nombres premiers proches pour le module RSA, car cela pourrait faciliter l'attaque par factorisation.
- **Sécurité de l'implémentation :** Assurez-vous que l'implémentation de la multiplication de Montgomery est correcte et ne laisse pas de traces exploitables. Par exemple, elle ne devrait pas varier en temps ou en consommation d'énergie en fonction des données traitées.
- **Utilisation de contre-mesures contre les attaques par canal auxiliaire :** Ces attaques exploitent les informations obtenues à partir de l'implémentation physique du cryptosystème, comme le temps d'exécution ou la consommation d'énergie. Des techniques comme le brouillage peuvent aider à prévenir ces attaques.
- **Mises à jour régulières :** Assurez-vous que le système est régulièrement mis à jour pour protéger contre les nouvelles vulnérabilités découvertes.
- **Tests de pénétration :** Effectuez régulièrement des tests de pénétration pour évaluer la résistance du système aux attaques.

- **Formation et sensibilisation :** Assurez-vous que tous les utilisateurs du système sont bien formés et conscients des risques de sécurité. Ils devraient savoir comment utiliser le système de manière sûre et être capables de reconnaître les signes d'une éventuelle attaque.

Ces mesures peuvent aider à renforcer la sécurité de RSA avec la multiplication de Montgomery, mais il est important de noter qu'aucun système n'est totalement à l'abri des attaques. Par conséquent, une approche de sécurité en profondeur, qui utilise plusieurs couches de défense, est souvent la meilleure stratégie.

Implémentation de l'algorithme RSA en python

Introduction

Nous avons choisi d'implémenté l'algorithme de réduction de Montgomery en python pour accélérer les opérations de multiplication modulaire. Cet algorithme est particulièrement efficace lorsqu'il est appliqué à des chaînes de multiplications, comme dans le cas de l'exponentiation modulaire.
L'objectif principal de l'algorithme de réduction de Montgomery est de calculer efficacement $(c \equiv a \times b \mod n)$ pour des valeurs données de (a), (b) , et (n).en mettant l'accent sur la transformation des nombres dans une forme spéciale appelée la forme de Montgomery.

Étapes du Calcul

Étape 1 : Choix de (r)

1. Choisissez un entier (r) tel que $(r > n)$ et $(\text{pgcd}(r, n) = 1)$.
2. Calculez $(v = N^{-1} = \frac{r(r^{-1} \mod n)-1}{n})$.

Étape 2 : Transformation en Forme de Montgomery

1. Convertissez les nombres d'entrée (a) et (b) en forme de Montgomery : $(\bar{a} = (ar \mod n))$ et $(\bar{b} = (br \mod n))$.

Étape 3 : Réduction de Montgomery

$$s = \bar{a}\bar{b}$$

$$t = (s.v) \mod r$$

$$m = (s + t.n)$$

$$u = m/r$$

La valeur de (u) est égale à (u) si $(u < n)$, sinon elle est égale à $(u - n)$.

$$c = (u \cdot r^{-1}) \mod n$$

Voici L'implémentation de la classe Montgomery :

```
class Montgomery:
    def __init__(self):
        self.base = base
        self.exponent = exponent
        self.n = n

    @staticmethod
    def choose_r(n):
        # Ensure n is odd and greater than 3
        if n % 2 == 0 or n <= 3:
            raise ValueError("Modulus n must be odd and greater than 3.")

        # Find the smallest power of 2 greater than n
        r = 2
        while r <= n:
            r *= 2

        return r
    @staticmethod
    def extended_gcd(a, b):
        x0, x1, y0, y1 = 1, 0, 0, 1
        while b != 0:
            q, a, b = a // b, b, a % b
            x0, x1 = x1, x0 - q * x1
            y0, y1 = y1, y0 - q * y1
        return a, x0, y0

    @staticmethod
    def modinv(R, N):
        gcd, x, y =Montgomery.extended_gcd(R, N)
        if gcd != 1:
            raise ValueError(f"The modular inverse does not exist for {R} mod {N}.")
        return x % N
```

```
    @staticmethod
    def montgomery_multiply(a,b,n):
        r=choose_r(n)
        r_inv=modinv(r,n)
        n_inv=(r*(r_inv%n)-1)//n
        a_bar=(a*r)%n
        b_bar=(b*r)%n
        s=a_bar*b_bar
        t=(s*n_inv)%r
        m=s+(t*n)
        u=m//r
        c_bar=u if u<n else u-n
        c=(c_bar*r_inv) % n
        return c


    @staticmethod
    def montgomery_power(base,exponent,n):
        if n == 1:
            return 0
        result = 1
        base = base % n
        while exponent > 0:
            if exponent % 2 == 1:
                result = Montgomery.montgomery_multiply(result , base , n)
            exponent = exponent >> 1
            base = Montgomery.montgomery_multiply(base , base,n )
        return result
```

RSA Avec Montgomery

Les opérations RSA pour le chiffrement et le déchiffrement impliquent l'exponentiation modulaire : $(X^Y \mod M)$.

- C : Ciphertext (Texte chiffré)
- P : PlainText (Texte en clair)
- e : Exposant public
- d : Exposant privé
- M : modulo
- $C = Pe \ mod \ M$ (chiffrement avec la clé publique)
- $P = Cd \ mod \ M$ (déchiffrement avec la clé privée)

La clé publique est (e, M) et la clé privée est (d, M).

Génération du Modulo (M)

Choisissez deux nombres premiers ayant une demi-longueur en bits des clés publique/privée souhaitées.

- p : nombre premier
- q : nombre premier
- $M = p\ x\ q$

Génération de l'Exposant Public (e)

L'exposant public (e) est un petit entier. Des choix valides sont 3, 5, 7, 17, 257 ou 65537. Avec les clés RSA, l'exposant public est généralement 65537.

Les exigences pour e sont :

- $1 < e <$ phi(M) = (p - 1) * (q - 1)
- e et la fonction phi d'Euler de M sont premiers entre eux.

Génération de l'Exposant Privé (d)

L'exposant privé est généré à partir des nombres premiers p et q et de l'exposant public.

- $d = e$-1 mod (p - 1) * (q - 1)

Exponentiation Modulaire Efficace (en utilisant la Multiplication de Montgomery)

L'algorithme de multiplication et carré pour l'exponentiation modulaire nécessite une multiplication modulaire sous la forme :

- $Z = X * X\ mod\ M$
- $Z = X * Y\ mod\ M$

Cette opération nécessite des opérations coûteuses de multiplication et de division. La multiplication de Montgomery convertit les paramètres en résidus modulaires où la multiplication devient une addition et la division devient un décalage bit à bit.

```
from sympy import randprime, mod_inverse

#  Choix de deux nombres premiers et calcul de n
p = randprime(1000, 2000)
q = randprime(3000, 4000)
n = p * q
```

```
#  Calcul de la fonction totient
phi_n = (p - 1) * (q - 1)

# Choix de e
e = 65537  # Il est courant d'utiliser 65537 comme e dans RSA

# Step 1.5: Calcul d
d = mod_inverse(e, phi_n)
print("e = ",e,"d = ",d)
print("p = ",p,"q = ",q)



def encrypt(m, e, n):
    return Montgomery.montgomery_power(m, e, n) #  Calcule M ^(e) utilisant l'algorithm

def decrypt(c, d, n):
    return Montgomery.montgomery_power(c, d, n) ##  Calcule C ^(d) utilisant l'algorith

message=15
C=encrypt(message,e,n)
M=decrypt(C,d,n)
print("message ",message)
print("message  Encrypt ",C)
print("message  Decrypt ",M)
```

```
e =  65537 d =  2789225
p =  1783 q =  3557
message  15
message  Encrypt  3690277
message  Decrypt  15
```

Comparaison des Performances

```
def regular_power(base, exponent, modulus):
    """
    Computes (base^exponent) % modulus using repeated multiplication.

    Parameters:
    - base: Base of the multiplication.
```

```
    - exponent: Number of times to multiply.
    - modulus: Modulus.

    Returns:
    - Result of (base^exponent) % modulus.
    """
    result = 1

    for _ in range(exponent):
        result = (result * base) % modulus

    return result


import timeit


# Test parameters
a = 12953
b = 6789
n = 9985


# Time the Montgomery multiplication function
montgomery_time = timeit.timeit(
    lambda:Montgomery.montgomery_power(a, b, n),
    number=10000  # Adjust the number of repetitions as needed
)



# Time the regular modular multiplication function
modular_time = timeit.timeit(
    lambda: regular_power(a, b, n),
    number=10000  # Adjust the number of repetitions as needed
)


# Display the results
print(f"Montgomery Power Time: {montgomery_time:.6f} seconds")
```

```
print(f"Regular Power Time: {modular_time:.6f} seconds")


# Compare the performance
if montgomery_time < modular_time:
    print("Montgomery  faster.")
elif montgomery_time > modular_time:
    print("Regular modular Power is faster.")
else:
    print("Both functions have similar performance.")
```

```
Montgomery Power Time: 2.754705 seconds
Regular Power Time: 11.401858 seconds
Montgomery  faster.
```