SAE Crypto - Défi 2 : Logarithme discret et attaque Meet-in the-Middle

Ronceray Maxime, Antonin Reydet
IUT Informatique, IUT Orléans

Table des matières

1 Introduction

Dans ce fichier, nous traiterons du problème du logarithme discret, notamment de son utilisation de cryptographie dans le processus d'échange de clé de Diffie-Hellman [1]. Puis par la suite nous traiterons l'attaque Meet-in-the-middle (MiTM) ainsi que de son utilité dans la résolution de certains problèmes.

2 Logarithme discret

2.1 Introduction

Soit G un groupe cyclique multiplicatif de n éléments. Soit b le générateur de G, alors tout élément $g\ de\ G$ de s'écrire sous la forme

$$g = b^k\ [mod\ n]\ \forall\ k\ \in R \text{ ou bien } log_b(g) = k[mod\ n]$$

En connaissant $b\ et$ k il est facile de trouver g, or l'opération inverse est beaucoup plus complexe.
On remarque que plus n est grand, plus il est complexe de trouver X.
L'un des groupes les plus populaires pour les crypto-systèmes basés sur des logarithmes discrets est le groupe Z_p^* ou p est premier [2].

2.2 Exemple

On cherche à résoudre : $log_b(g) = k[mod\ n]$ avec g = 2, n = 11, b = 9

$$g = b^k[mod\ n]$$
$$9 = 2^k[mod\ 11]$$

Méthode de résolution « Brute Force », on essaie $n = 1,2,3, \ldots$
Ce qui nous donne la solution $k = 6$.

2.3 Méthodes de résolution

Bien que résoudre un problème de logarithme discret soit complexe pour p grand. Il existe certains algorithmes permettant de résoudre ce problème.

Recherche linéaire :

Il s'agit tout simplement d'une stratégie de « brute force » qui consiste globalement à tester toutes les possibilités. Au mieux la complexité est de O(n)

Algorithme :

```
Pour i allant de 2 à p :
        si a == g^i alors :
                retourner i
```

Recherche BS-GS:

Pour aller plus loin : La recherche « *Baby-step/giant-step* » est globalement meilleure que la précédente, en utilisant notamment des tables des hachages, cependant elle nécessite que $PGCD(g,b) = 1$, cad que g et b soit premiers entre eux. Au mieux elle est de complexité $O\sqrt{n}$. [3].
Cet algorithme se base sur le fonctionnement de l'attaque MiTM (expliqué plus tard), cependant il présente un avantage non négligeable en terme de complexité.

L'algorithme BSGS utilise une des propriétés des logarithmes stipulant que : si a et b sont des nombres positifs et si x et y sont des nombres réels, alors $log_a(a^x) et\ log_a\left(\frac{a^x}{a^y}\right) = x - y$

Pour utiliser l'algorithme BSGS, vous devez connaître la valeur de g, de p et de $g^x (mod\ p)$, où g est la base du logarithme, p est le module et x est l'exposant que vous souhaitez résoudre. L'objectif de l'algorithme BSGS est de trouver la valeur de x en utilisant ces données.

L'algorithme BSGS fonctionne en deux étapes :

La première étape consiste à calculer la valeur $g^{m^2} (mod\ p)$ pour différentes valeurs de m. Cette étape est appelée l'étape des "pas de géants".

La seconde étape consiste à calculer la valeur de $g^m (mod\ p)$ pour différentes valeurs de m et à comparer ces valeurs à celles obtenues lors de la première étape. Cette étape est appelée l'étape des "pas de bébés".

La première étape peut être effectuée rapidement en utilisant des techniques d'exponentiation rapide.

Lors de l'étape des "pas de bébés", vous calculez la valeur de $g^m (mod\ p)$ pour différentes valeurs de m et vous les comparez à celles obtenues lors de la première étape. Si vous trouvez une valeur de $g^m (mod\ p)$ qui correspond à une valeur de $g^{m^2} (mod\ p)$ calculée précédemment, cela signifie que vous avez trouvé la valeur de x.

L'algorithme BSGS est avantageux par rapport à l'attaque Meet-In-the-Middle en raison de sa complexité inférieure. La complexité de l'algorithme BSGS est environ $O(\sqrt{p} * \log(p))$, alors que la complexité de l'attaque MiTM est environ $O(nk^2)$, où n est la longueur du message à casser et k est le nombre de clés possibles. EN conclusion, bien que l'algorithme BSGS implémente le fonctionnement d'une attaque MiTM, ce dernier gagne en rapidité grâce à l'utilisation de propriétés mathématiques des logarithmes.

Algorithme (python) [4] :

```
def bsgs(g, a, p):
    # To solve g^e mod p = a and find e
    m = ceil(sqrt(p-1))
    # Baby Step
    lookup_table = {pow(g, i, p): i for i in range(m)}
    # Giant Step Precomputation c = g^(-m) mod p
    c = pow(g, m*(p-2), p)
    # Giant Step
    for j in range(m):
        x = (a*pow(c, j, p)) % p
        if x in lookup_table:
            return j*m + lookup_table[x]
    return None
```

3 Protocole d'échange de Diffie-Hellman

Le protocole d'échange de Diffie-Hellman se base sur les propriétés des logarithmes discrets énoncés ci-dessus. Il permet à deux utilisateurs d'avoir une même clé sans jamais que cette dernière soit publique.

3.1 Processus

ALICE	Public	BOB
Choisit α un entier quelconque	Alice et Bob choisissent : - n un entier quelconque - p un nombre premier	Choisit β un entier quelconque
Calcule $n^{\alpha}[mod\ p] = R_a$		Calcule $n^{b}[mod\ p] = R_b$
	Alice et Bob s'échangent $R_a\ et\ R_b$	
Calcule $(R_b)^{\alpha}[mod\ p] = K_a$		Calcule $(R_a)^{\beta}[mod\ p] = K_b$

3.2 Démonstration

$Soit\ (n, \alpha, \beta) \in N^3, et\ p\ un\ grand\ nombre\ premier$
$n^{\alpha} = R_a$
$n^{\beta} = R_b$
$(n^{\alpha})^{\beta} = K_b = n^{\alpha\beta}\ d'après\ les\ propriétés\ des\ exponentielles$
$(n^{\beta})^{\alpha} = K_a = n^{\beta\alpha}$
$Donc\ K_a = K_b$

Alice et Bob possèdent donc bien la même clé.
Etant donné que $n, p, R_a\ et\ R_b$ sont publiques, il est théoriquement possible de retrouver la valeur de $\alpha\ et\ \beta$, en résolvant $n^{\alpha}[mod\ p] = R_a$. Cependant, comme énoncé dans la **partie 2,** pour p grand, il est très difficile de résoudre ce genre de problèmes.
L'algorithme d'échange de Diffie-Hellman assure la confidentialité persistante (PFS : Perfect Forward Security). « La confidentialité persistante est assurée lorsque la découverte des clés privées d'un correspondant ne compromet pas les communications passée » [5].

Confidentialité qui n'est pas respecté dans le cadre d'une transmission avec RSA. Si quelqu'un venait à surveiller et intercepter les messages il pourrait alors remonter jusqu'à la clé utilisée.
Cependant, l'échange de Diffie-Hellman possède plusieurs limites. En effet, il ne s'agit que d'un échange de clé, suivant la méthode de chiffrement utilisée par la suite, il est possible de réaliser une attaque par cryptanalyse.
De plus, Alice n'est pas assurée de communiquer avec Bob, un troisième personnage (Eve), pourrait se placer entre les deux et se faire passer pour Alice ou Bob, ruinant ainsi l'échange [6].
Ce protocole est donc vulnérable aux attaques de type Man-in-the-Middle [7].

3.3 Implémentation Python

L'implémentation du processus d'échange de clé de Diffie-Hellman a été faite dans le fichier DH_Key_exchange.py. Le programme permet de procéder à un échange de clé pour des valeurs ayant jusqu'à 200 caractères dans un temps très raisonnable.

Temps d'exécution du protocole de Diffie-Hellman en fonction de la longueur du modulo p

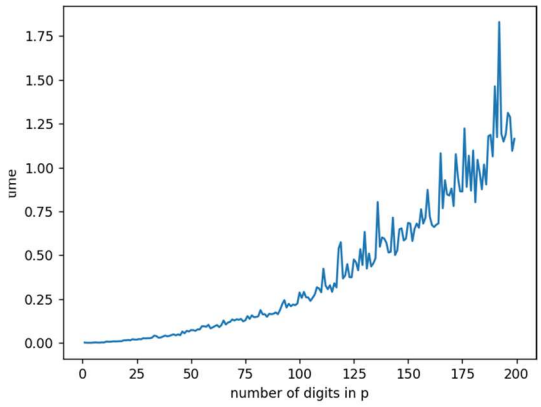

Fig 1. DH computation time

A l'aide du graphique ci-dessus nous pouvons observer que le temps d'exécution du protocole de Diffie-Hellman dépend en grande partie de la longueur de la clé utilisée. Plus la clé est longue, plus il faudra de temps pour exécuter le protocole.

Il est important de noter que ces temps d'exécution sont indicatifs et peuvent varier en fonction de plusieurs facteurs.

Notamment l'ordinateur utilisé, mais également l'algorithme de génération de nombre premier (afin de trouver p). Ou bien l'algorithme utilisé pour effectuer l'exponentiation modulaire.

Cependant, comme énoncé plus haut, dans un problème de logarithme discret, plus p est grand, plus ce dernier est difficile à résoudre et plus l'échange est sécurisé.

Les meilleurs algorithmes d'échange de clé de Diffie-Hellman sont de l'ordre de $O(\log(p))$, cependant dans notre cas, il est au mieux d'une complexité de $O(p^2)$.

4 Meet-In-The-Middle

L'attaque Meet-In-the-Middle (MiTM), est introduite par Diffie-Hellman en 1997.
Il s'agit d'une technique utilisée en cryptanalyse pour casser certains types de chiffrements symétriques à clé unique. Cette attaque consiste à diviser le message à casser en deux parties égales, de les chiffrer séparément en utilisant toutes les clés possibles, puis de comparer les deux parties chiffrées pour voir si elles correspondent. Si elles correspondent, cela signifie que la même clé a été utilisée pour chiffrer les deux parties, et donc que la clé a été trouvée.

4.1 Fonctionnement

Le fonctionnement de l'attaque Meet-in-the-middle consiste en plusieurs étapes :

1. Diviser le message à casser en deux parties égales, A et B.
2. Chiffrer la partie A en utilisant toutes les clés possibles, et stocker les résultats dans une table.
3. Chiffrer la partie B en utilisant toutes les clés possibles, et stocker les résultats dans une autre table.
4. Parcourir les lignes des deux tables en les comparant l'une à l'autre. Si on trouve une ligne dans la table A qui correspond à une ligne dans la table B, cela signifie que les deux parties du message ont été chiffrées avec la même clé, et donc que la clé a été trouvée.

4.2 Equations

Pour réduire le nombre de cas à étudier lors d'une recherche en force brute, il est possible d'utiliser une stratégie de recherche binaire. Cela consiste à diviser l'ensemble des valeurs possibles de la clé en deux parties égales et à choisir la partie qui contient la clé à chaque étape. Cela permet de réduire le nombre de cas à étudier à chaque étape, ce qui accélère considérablement la recherche.

Pour trouver un logarithme discret, il faut d'abord écrire les équations de déchiffrement. Dans le cas où la clé K1 est utilisée en premier pour chiffrer le message M, cela donne :

M = decode(K1, encode(K1, M))

En simplifiant, on obtient :

M = decode(K1, C1)

Où C1 est le chiffré obtenu en utilisant la clé K1. De même, lorsque la clé K2 est utilisée en second pour chiffrer le message chiffré C1, on obtient :

C1 = encode(K2, decode(K2, C))

En simplifiant, on obtient :

C1 = encode(K2, M)

Pour trouver les valeurs des clés K1 et K2, il suffit donc de tester toutes les valeurs possibles de K1 et K2 et de vérifier si elles satisfont les deux équations ci-dessus. Si une paire de valeurs (K1, K2) satisfait les deux équations, alors on a trouvé les valeurs des clés et on peut déchiffrer le message chiffré.

Cette méthode est beaucoup plus efficace que la recherche en force brute, car elle ne nécessite que de tester les valeurs des clés K1 et K2 une seule fois chacune, plutôt que de tester toutes les valeurs possibles de la clé K1 et K2 séparément.

4.3 Exemple

Pour illustrer ces étapes, supposons que nous avons un message à casser qui est divisé en deux parties, A et B, et que nous souhaitons trouver la clé utilisée pour chiffrer ce message. Nous procéderions comme suit :

Nous divisons le message en deux parties égales, A et B. Par exemple, si le message est "Hello World!", nous pourrions diviser le message en "Hello" et "World!".

Nous chiffrons la partie A en utilisant toutes les clés possibles, et nous stockons les résultats dans une table. Par exemple, si la partie A est "Hello" et que nous avons 100 clés possibles, notre table contiendrait 100 lignes, chacune contenant le résultat du chiffrement de "Hello" avec une des 100 clés.

Nous chiffrons la partie B en utilisant toutes les clés possibles, et nous stockons les résultats dans une autre table. Par exemple, si la partie B est "World!" et que nous avons toujours 100 clés possibles, notre deuxième table contiendrait également 100 lignes, chacune contenant le résultat du chiffrement de "World!" avec une des 100 clés.

Nous parcourons les lignes des deux tables en les comparant l'une à l'autre. Si nous trouvons une ligne dans la table A qui correspond à une ligne dans la table B, il s'agit de la valeur de la clé recherché.

4.4 Complexité

La complexité d'un algorithme utilisant l'attaque Meet-in-the-Middle dépend de plusieurs facteurs, tels que la longueur du message à casser, le nombre de clés possibles, et la complexité du chiffrement utilisé. Dans l'exemple que nous avons donné, la complexité serait en grande partie déterminée par le nombre de clés possibles.

Si nous supposons que le message à casser a une longueur de n caractères et que le nombre de clés possibles est k, alors la complexité de l'algorithme serait environ $O(nk^2)$, car il faut d'abord chiffrer chaque partie du message avec toutes les clés possibles (ce qui prend un temps $O(nk)$,), puis comparer les résultats obtenus (ce qui prend un temps $O(k^2)$,).

Dans l'exemple que vous avez donné, où le message est "Hello World!" et où il y a 100 clés possibles, la complexité serait environ $O(11 * 100^2) = O(110\,000)$, c'est-à-dire que l'algorithme prendrait environ 110000 opérations pour trouver la clé. Cette complexité est assez élevée, mais elle peut être réduite en utilisant des optimisations spécifiques, telles que l'utilisation de hash tables pour stocker les résultats des chiffrements.

En comparaison, si l'on devait brute force afin de trouver la solution la complexité serait de $O(k^n)$. Toujours avec le même exemple, $O(100^{11}) = O(100\,000\,000\,000\,000\,000\,000)$, ce qui donne un quintilions d'opération afin de trouver la solution.

L'attaque MiTM est grandement avantageuse, de par sa rapidité, mais également sa simplicité de mise en œuvre. Cependant cette dernière est limitée par le stockage des résultats ainsi que par le temps pour trouver l'intersection des deux tableaux. Beaucoup d'attaques modernes se basent sur cette dernière dans leur fonctionnement.

4.5 Fonctionnement de l'algorithme

Dans l'algorithme présent dans le fichier Python DLP.py, nous simulons l'attaque d'un échange de Diffie-Hellman. Dans cet échange, les messages intermédiaires d'Alice et Bob sont interceptés par Eve. Cette dernière connaît donc R_a *et* R_b, n *et* p. Afin de trouver la clé partagée d'Alice et Bob elle décide d'utiliser une attaque de type Meet-In-The-Middle. Voici la simulation utilisée.

1. Génère un nombre premier p (6 caractères dans notre cas)
2. Génère les clés privées d'Alice et Bob, a et b
3. Génère un nombre aléatoire n
4. Calcule les clés publiques d'Alice et Bob, Ra et Rb
5. Eve effectue une attaque par force brute pour trouver les clés privées d'Alice et Bob possibles.
6. Eve calcule les clés partagées secrètes possibles pour Alice et Bob
7. Eve trouve l'intersection des deux listes de clés partagées secrètes
8. Affiche la clé partagée secrète commune

Ou bien de façon plus concrète en pseudo-code :

```
p = generatePrime(10**6)
a = generateRandomInteger(2, p - 1)
b = generateRandomInteger(2, p - 1)
g = generateRandomInteger(2, p / 2)
A = fast_exp(g, a, p)
B = fast_exp(g, b, p)
listA = []
listB = []
for i in range(1, p):
    temp = fast_exp(g, i, p)
    if temp == A:
        listA.append(i)
    if temp == B:
        listB.append(i)
resA = map(lambda x: fast_exp(B, x, p), listA) -> On applique l'exponentiation modulaire a tous les éléments de listA
resB = map(lambda x: fast_exp(A, x, p), listB) -> On applique l'exponentiation modulaire a tous les éléments de listB
intersection = intersection(resA, resB)
```

Une attaque MITM consiste à utiliser une force brute pour calculer toutes les valeurs possibles pour une des parties du problème, puis à utiliser ces valeurs pour calculer toutes les valeurs possibles pour l'autre partie du problème. L'intersection des deux ensembles de valeurs est la solution du problème.

Dans le cas de l'algorithme présenté ici, l'attaquant utilise une force brute pour calculer toutes les valeurs possibles pour les clés privées d'Alice et Bob. Il utilise ensuite ces clés privées pour calculer toutes les valeurs possibles pour les clés partagées secrètes d'Alice et Bob. L'intersection des deux ensembles de valeurs est la clé partagée secrète commune, qui permet à l'attaquant de déchiffrer les communications entre Alice et Bob. Cette technique est appelée "Meet-in-the-middle" car les valeurs calculées pour les deux parties du problème se rencontrent au milieu de l'algorithme.

4.6 Temps de résolution d'un problème de logarithme discret

Temps de résolution d'un problème de logarithme discret selon la méthode d'approche et la longueur de p

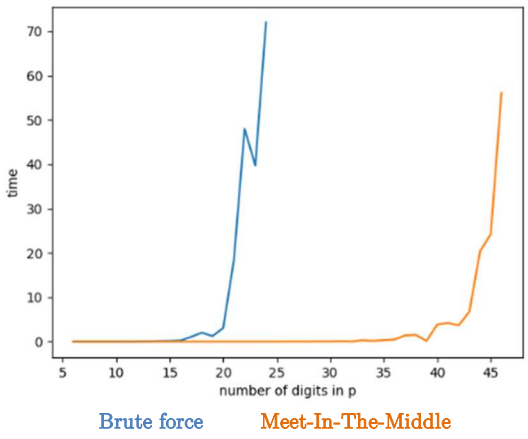

Fig 2. MITM vs Brute Force

Le graphique ci-dessus permet de comparer le temps de résolution d'un problème de logarithme discret selon la méthode utilisée.

On peut observer que l'attaque « Brute Force » est beaucoup moins efficace que l'attaque MiTM. Cependant les deux attaques se retrouvent très rapidement inefficaces pour des clés de grande taille.

Dans le cadre de la simulation, nous n'utilisons que des clés de 6 chiffres maximum afin que le temps de calcul ne soit pas trop lent.

Vous pouvez retrouver tout le code ayant permis de réaliser les différents graphiques utilisés dans le fichier plot.py

5 Conclusion

En conclusion, nous avons étudié les logarithmes discrets et leur utilisation dans le protocole d'échange de clés de Diffie-Hellman. Nous avons également présenté l'attaque Meet-in-the-middle, une technique utilisée pour résoudre des problèmes de cryptographie à chiffrements multiples. Nous avons montré comment cette attaque peut être utilisée pour déterminer la clé partagée secrète dans le protocole de Diffie-Hellman.

En résumé, les logarithmes discrets et les attaques de type Meet-in-the-middle jouent un rôle important dans la cryptographie moderne. Ils sont utilisés dans de nombreux protocoles de cryptographie à clé publique, tels que le protocole de Diffie-Hellman et les courbes éliptiques. Les dérivés de ces attaques, tels que les attaques de type partial matching et differential-MITM, sont également des outils importants pour la sécurité des systèmes de cryptographie. Dans l'avenir, il est important de continuer à étudier ces techniques afin de garantir la sécurités des systèmes d'échange d'informations.

6 Références :

[1]. Changyu Dong (S.D). Math in Network Security: A Crash Course [En-ligne]. Disponible : https://www.doc.ic.ac.uk/~mrh/330tutor/index.html

[2]. Ginni(2021). What is Discrete Logarithmic Problem in Information Security? [En-ligne]. Disponible : https://www.tutorialspoint.com/what-is-discrete-logarithmic-problem-in-information-security

[3]. Henri Cohen (S.D) A Course in Computational Algebraic Number Theory.

[4]. Ashutosh Ahelleya (S.D) DLPcand Baby Step Giant Step Algorithm [En-ligne]. Disponible : https://masterpessimistaa.wordpress.com/2018/01/14/dlp-and-baby-step-giant-step-algorithm/

[5]. OpenSSL (2021). Diffie-Hellman [En-ligne]. Disponible : https://wiki.openssl.org/index.php/Diffie_Hellman

[6]. Thomas Pornin (2013). Diffie-Hellman and its TLS/SSL usage [En-ligne]. Disponible : https://security.stackexchange.com/questions/41205/diffie-hellman-and-its-tls-ssl-usage

[7]. Art of the Problem (2012). Public key cryptography - Diffie-Hellman Key Exchange (full version)[En-ligne]. Disponible : https://www.youtube.com/watch?v=YEBfamv-_do&ab_channel=ArtoftheProblem

[8] . Christina Boura , Nicolas David , Rachelle Heim Boissier , and María Naya-Plasencia .(2022). Better Steady than Speedy: Full break of SPEEDY-7-192

[9]. Nicolas David . (2022). Differential Meet-In-The-Middle Cryptanalysis

[10]. Chris Kowalczyk (S.D). Meet In The Middle Attack. [En-ligne]. Disponible : http://www.crypto-it.net/eng/attacks/meet-in-the-middle.html
[X]. Nicolas David, doctorant en cryptographique a l'INRIA : nicolas.david@inria.fr