2EL1740 : Algèbre & Cryptologie

CentraleSupélec - 2A

Challenge 3

27 septembre 2024

CentraleSupélec

Raphaël PAIN DIT HERMIER
Alexis LOMBARD-GAILLARD
Edward LUCYSZYN

Table des matières

1 Question 1

Dans cette question, notre tâche est de trouver un groupe cyclique fini $G = \langle g \rangle$ sur lequel le problème du logarithme discret est difficile avec une sécurité d'au moins 64 bits.

Premièrement, la condition sur la sécurité nous impose que $|G| > 2^{64}$. Dans un premier temps, on pourrait considérer le groupe cyclique $(F_p^n)^\times$ où p est petit (par exemple $p = 2$) et $n > 64$. Cependant cette méthode ne marche pas : un article publié en 2014 [1] révèle que le problème du logarithme discret est cassé sur les groupes obtenus par corps finis de petite caractéristique.

Une autre piste qui semble bien meilleure est celle du logarithme discret dans $\mathbb{F}_p^\times$ où $p > 2^64$ est un nombre premier. Ici la difficulté ne vient pas dans la sécurité mais dans le choix d'un générateur : en effet, on n'a a priori aucune méthode pour s'assurer que $x \in \mathbb{F}_p^\times$ est bien un générateur de ce dernier.

Notre approche est alors la suivante : En s'inspirant du cours, on cherche $q > 2^{64}$ un premier de Sophie-Germain, c'est-à-dire un nombre premier tel que $2q + 1$ est aussi premier. Alors pour n'importe quel $x \in \mathbb{F}_p^\times \backslash \{\pm 1\}$, $G = \langle x^2 \rangle$ est un groupe cyclique fini de cardinal q, dans lequel le problème du logarithme discret est difficile avec une sécurité d'au moins $\lfloor \log_2 q \rfloor$ bits. (On remarquera que le G obtenu ne dépend pas de l'élément x choisi)

Il nous reste alors à générer un premier de Sophie-Germain de taille satisfaisante. Rappelons d'abord le théorème de Pocklington :

Soit n un entier. S'il existe des entiers a et q tels que :

1. *q est premier, $q|(n-1)$, et $q > \sqrt{n} - 1$;*
2. *$a^{n-1} = 1 \ mod \ n$;*
3. *$\gcd(a^{(n-1)/q} - 1, n) = 1$;*

alors n est premier.

On propose alors l'algorithme suivant :

1. On commence avec $a = 2$, $R = 2$, q un petit nombre premier arbitraire.
2. On calcule $n = qR + 1$.
3. Si (a, q, n) vérifie Pocklington, alors n est premier, $q \leftarrow n$. Sinon, $R \leftarrow R + 1$ et revenir à l'étape 2.
4. Si q est plus grand que la taille désirée (2^{64}) et $R = 2$, alors l'algorithme s'arrête. Sinon, $R \leftarrow 2$ et on revient à l'étape 2.

L'algorithme n'est pas garanti de s'arrêter, mais s'il s'arrête alors on a bien obtenu un nombre q premier de Sophie-Germain de taille convenable.
Voici ce que l'algorithme fournit, lorsque codé et implémenté avec $q = 3$: (avertissement pour les âmes sensibles avec une phobie des grands nombres)

— 64822267375873812414684644006656820041194610312722752106927357865862090777037931543235985767564057913012 59 est premier (a=2, q=3241113368793690620734232200332841002059730515636137605346367893293104538851896577161799288378202895650629) ;

— 3241113368793690620734232200332841002059730515636137605346367893293104538851896577161799288378202895650629 est premier (a=2, q=4071750463308656558711346985342765077964485572407208046917547604639578566396854996434421216555531275943) ;

— 407175046330865655871134698534276507796448557240720804691754760463957856639685499643442121655553127 5943 est premier (a=2, q=2262083590727031421506303880745980598869158651337337803843082002577543647998252775796900675864184044219) ;

— 22620835907270314215063038807459805988691586513373378038430820025775436479982527757969006758641840 4219 est premier (a=2, q=1300048040647719207762243609624126780959286581228355059679932185389392901148421135515460158542634507) ;

— 13000480406477192077622436096241267809592865812283550596799321853893929011484211355154601585426345 07 est premier (a=2, q=722248911470955115423468672013403767199603656237975033155517880771884945082456186397477865857019 17) ;

— 7222489114709551154234686720134037671996036562379750331555178807718849450824561863974778658570191 7 est premier (a=2, q=54715826626587508744202172122227558121182095169543563117842263694839768566852741393748323170986 3) ;

— 547158266265875087442021721222275581211820951695435631178422636948397685668527413937483231709863 est premier (a=2, q=57354115960783552142769572455165155263293600806649437230442624418071036233598261419023399550 3) ;

— 573541159607835521427695724551651552632936008066494372304426244180710362335982614190233995503 est premier (a=2, q=26309227504946583551729161676681263882244771012224512491028726797280291850274431843588715 39) ;

— 2630922750494658355172916167668126388224477101222451249102872679728029185027443184358871539 est premier (a=2, q=18792305360676131108377972626200902773031979294446080350734804855200208464481737031134796 7) ;

— 187923053606761311083779726262009027730319792944460803507348048552002084644817370311347967 est premier (a=2, q=19991814213485245859976566623617981673438275845155404628441281760851285600512486203334 89) ;

— 199918142134852458599765666236179816734382758451554046284412817608512856005124862033348 9 est premier (a=2, q=41649612944760928874951180465870795152996408010740426309252670335106845001067679590281) ;

— 4164961294476092887495118046587079515299640801074042630925267033510684500106767959028 1 est premier (a=2, q=10412403236190232218737795116467698788249102002685106577313167583776711250266919897 57) ;

— 1041240323619023221873779511646769878824910200268510657731316758377671125026691989757 est premier (a=2, q=28923342322750645052049430879076941078469727785236407159203243288268642361852555271) ;

— 289233423227506450520494308790769410784697277852364071592032432882686423618525552 71 est premier (a=2, q=28923342322750645052049430879076941078469727785236407159203243288268642361852555271) ;

— 289233423227506450520494308790769410784697277852364071592032432882686423618525552 7 est premier (a=2, q=22955033589484638930197961015140429427356926813679688221589875625610033620517901) ;

— 2295503358948463893019796101514042942735692681367968822158987562561003362051790 1 est premier (a=2, q=22955033589484638930197961015140429427356926813679688221589875625610033620517901) ;

— 229550335894846389301979610151404294273569268136796882215898756256100336205179 est premier (a=2, q=38258389315807731550329935025234049045594878022799480369316459376016722700863) ;

— 382583893158077315503299350252340490455948780227994803693164593760167227008 63 est premier (a=2, q=63763982193012885917216558375390081742658130037999133948860765626694537834 77) ;

— 637639821930128859172165583753900817426581300379991339488607656266945378347 7 est premier (a=2, q=38880476946959076778790584375237854721133006120731179237110222943106425509) ;

— 388804769469590767778790584375237854721133006120731179237110222943106425509 est premier (a=2, q=20681104759020785520633289561296731234645216021665520870803310076120439 1) ;

— 206811047590207855206332895612967312346452160216655208708033100761204391 est premier (a=2, q=15908542122323681169717915047151331718957858478204246823694853904708 03) ;

— 159085421223236811697179150471513317189578584782042468236948539047080 3 est premier (a=2, q=38426430247158650168400760983457322992651832072957117931630081895 43) ;

— 384264302471586501684007609834573229926518320729571179316300818954 3 est premier (a=2, q=46861500301412988010244830467630881698355892771898924306865953531) ;

— 468615003014129880102448304676308816983558927718989243068659535 31 est premier (a=2, q=14200454636791814548559039535645721726774512961181492214201804 1) ;

— 142004546367918145485590395356457217267745129611814922142018041 est premier (a=2, q=11833712197326512123799199613038101438978760800984576845168 17) ;

— 118337121973265121237991996130381014389787608009845768451681 7 est premier (a=2, q=24653567077763566924581665860496044664539085002051201760767) ;

- 24653567077763566924581665860496044664539085002051201760767 est premier (a=2, q=152182512825701030398652258398123732497154845691674084943) ;
- 15218251282570103039865225839812373249715484569167408494 3 est premier (a=2, q=110277183207029732172936419129075168476199163544691365 9) ;
- 1102771832070297321729364191290751684761991635446913659 est premier (a=2, q=183795305345049553621560698548458614126998605907818943) ;
- 18379530534504955362156069854845861412699860590781894 3 est premier (a=2, q=113453892188302193593555986758307786498147287597419 1) ;
- 11345389218830219359355598675830778649814728759741 91 est premier (a=2, q=113453892188302193593555986758307786498147287597419) ;
- 1134538921883021935935559867583077864981472875974 19 est premier (a=2, q=110578842288793561007364509511021234403652327093) ;
- 110578842288793561007364509511021234403652327093 est premier (a=2, q=162615944542343472069653690457384168240665 1869) ;
- 16261594454234347206965369045738416824066518 69 est premier (a=2, q=689050612467557085040905468039763424748581 3) ;
- 68905061246755708504090546803976342474858 13 est premier (a=2, q=246089504452698958943180524299915508838779) ;
- 24608950445269895894318052429991550883877 9 est premier (a=2, q=315499364682947383260487851666558344665 1) ;
- 31549936468294738326048785166655834466 51 est premier (a=2, q=63099872936589476652097570333311668933) ;
- 6309987293658947665209757033331166893 3 est premier (a=2, q=157749682341473691630243925833279172 33) ;
- 15774968234147369163024392583327917233 est premier (a=2, q=32864517154473685756300817881933160 9) ;
- 3286451715447368575630081788193316 09 est premier (a=2, q=14165740152790381791508973224971 19) ;
- 1416574015279038179150897322497119 est premier (a=2, q=23609566921317302985848288708285 3) ;
- 236095669213173029858482887082853 est premier (a=2, q=31065219633312240770853011458 27) ;
- 3106521963331224077085301145827 est premier (a=2, q=17258455351840133761585006365 7) ;
- 172584553518401337615850063657 est premier (a=2, q=10272890090381032000943456 17) ;
- 1027289009038103200094345617 est premier (a=2, q=21401854354960483335298867) ;
- 214018543549604833352988 67 est premier (a=2, q=11889919086089157408499 37) ;
- 1188991908608915740849937 est premier (a=2, q=5716307252927479523317) ;
- 571630725292747952331 7 est premier (a=2, q=476358937743956626943) ;
- 476358937743956626943 est premier (a=2, q=21652678988361664861) ;
- 21652678988361664861 est premier (a=2, q=360877983139361081) ;
- 360877983139361081 est premier (a=2, q=9021949578484027) ;
- 9021949578484027 est premier (a=2, q=1503658263080671) ;
- 1503658263080671 est premier (a=2, q=50121942102689) ;
- 50121942102689 est premier (a=2, q=1566310690709) ;
- 1566310690709 est premier (a=2, q=391577672677) ;
- 391577672677 est premier (a=2, q=32631472723) ;
- 32631472723 est premier (a=2, q=5438578787) ;
- 5438578787 est premier (a=2, q=2719289393) ;
- 2719289393 est premier (a=2, q=169955587) ;
- 169955587 est premier (a=2, q=9441977) ;
- 9441977 est premier (a=2, q=1180247) ;
- 1180247 est premier (a=2, q=590123) ;
- 590123 est premier (a=2, q=22697) ;

— 22697 est premier (a=2, q=2837) ;
— 2837 est premier (a=2, q=709) ;
— 709 est premier (a=2, q=59) ;
— 59 est premier (a=2, q=29) ;
— 29 est premier (a=2, q=7) ;
— 7 est premier (a=2, q=3) ;

Donc $q = 3241113368793690620734232200332841002059730515636137605346367893293104538851896577161799288-378202895650629$ est premier de Sophie-Germain, et en notant $p = 2q + 1$, G tel que décrit précédemment est un groupe cyclique fini de cardinal q et de sécurité d'environ 350 bits.

2 Question 2

L'objectif de cette question de reconstituer le message d'origine d'un texte fragmenté, codé à partir du code de Reed-Solomon.

L'idée est que, à chaque groupe de 8 octets $(m_i)_{0\leq i\leq 7}$ du message transmis, il existe un UNIQUE polynôme interpolateur de degré au plus 3, à coefficients dans $\mathbb{F}_2[X]/(X^8+X^4+X^3+X^2+1)$, tel que :

$$\forall i \in [\![0,3]\!],\ L(X^i) = m_i$$

et

$$\forall i \in [\![4,7]\!],\ L(X^{i+1}) = m_i,$$

où les m_i sont sous la forme de polynômes (*i.e.* si $m_i = abcdefgh_2$, alors le polynôme associé est alors $m_i(X) = aX^7+bX^6+cX^5+dX^4+eX^3+fX^2+gX+h$).

Commençons par analyser le message. En découpant le message codé en groupes de 8 octets et en effectuant une simple analyse de fréquence, nous observons qu'il y a exactement 23 fois 2 octets manquants, 43 fois 3 octets manquants, 36 fois 4 octets manquants, 7 fois 5 octets manquants, et une seule fois 6 octets manquants. Sachant qu'il faut au minimum 4 octets pour avoir le polynôme interpolateur, cela signifie que nous pourrons décrypter intégralement 102/110 groupes de 8 octets. Nous ferons les autres avec le contexte de la phrase. Le code utilisé pour cette analyse est le suivant.

```
message = message.split()

groups = []
for i in range(0, len(message), 8):
    groups.append(message[i:i+8])

missing_bytes = []
for group in groups:
    missing_bytes.append(group.count("??"))

missing_bytes_count = {}
for missing_byte in missing_bytes:
    if missing_byte not in missing_bytes_count:
        missing_bytes_count[missing_byte] = 0
    missing_bytes_count[missing_byte] += 1

print(missing_bytes_count)
```

Pour chaque groupe de 8 octets $(m_0,...,m_7)$ possédant au moins 4 octets sans "??", on note $\mathbb{I}$ l'ensemble des indices des 4 premiers octets sans "??", incrémenté de 1 si l'indice est supérieur à 4. Pour calculer le polynôme interpolateur L du groupe de 8 octets, il faut d'abord, pour chaque $i \in \mathbb{I}$, calculer un polynôme ℓ_i, tel que :

$$\forall j \in \mathbb{I},\ \ell_i(X^j) = \delta_{ij},$$

où δ est le symbole de Kronecker. Ces polynômes peuvent s'exprimer avec la formule suivante, modulo $X^8+X^4+X^3+X^2+1$:

$$\forall T \in \mathbb{F}_{256},\quad \ell_i(T) = \prod_{\substack{j\in\mathbb{I},\\ i\neq j}} \frac{T-X^j}{X^i-X^j}.$$

Petit aparté : on peut noter que "$-X^j$" revient à faire "$+X^j$" comme on travaille dans $\mathbb{F}_2[X]$.
On aura alors,

$$\forall T \in \mathbb{F}_{256},\quad L(T) = \sum_{i\in\mathbb{I}} m_i(X)\ell_i(T).$$

On reconstitue enfin octets manquants dans le groupe de 8 octets par la formule

$$\forall i \in [\![0,3]\!],\ L(X^i) = m_i,$$

et

$$\forall i \in [\![4,7]\!],\ L(X^{i+1}) = m_i.$$

Numériquement, nous notons les polynômes sous forme de liste. Par exemple, $[a_2, a_1, a_0]$ représente le polynôme $a_2X^2 + a_1X + a_0$. La difficulté majeure de ce code a été de réaliser les opérations élémentaires sur les polynômes, telles que l'addition, la multiplication, et surtout la division. Pour cette dernière, nous utilisé le fait que $\mathbb{F}_{256}$ est un corps, et que tout élément non nul admet un inverse. Le code utilisé a été le suivant.

```
def poly_add(poly1, poly2):
    if len(poly1) < len(poly2):
        poly = poly2.copy()
        for i in range(len(poly1)):
            poly[-i-1] = (poly1[-i-1] + poly2[-i-1])%2
    else:
        poly = poly1.copy()
        for i in range(len(poly2)):
            poly[-i-1] = (poly1[-i-1] + poly2[-i-1])%2
    while poly and poly[0] == 0:
        del poly[0]
    return poly

def poly_mod(dividend, divisor):
    while dividend and dividend[0] == 0:
        del dividend[0]
    while divisor and divisor[0] == 0:
        del divisor[0]
    while len(dividend) >= len(divisor):
        ratio = dividend[0] / divisor[0]
        for i in range(1, len(divisor)):
            dividend[i] -= ratio * divisor[i]
        del dividend[0]
    for i in range(len(dividend)):
        dividend[i] = int(dividend[i])%2
    while dividend and dividend[0] == 0:
        del dividend[0]
    return dividend

def poly_multiply(poly1, poly2, modulus_poly=modulus_poly):
    product = [0] * (len(poly1) + len(poly2) - 1)
    for i in range(len(poly1)):
        for j in range(len(poly2)):
            product[i + j] = (product[i + j] + poly1[i]*poly2[j])%2
    return poly_mod(product, modulus_poly)

def poly_power(poly,k,modulus_poly=modulus_poly):
    result = [1]
    n = k
    polycopy = poly.copy()
    while n > 0:
        if n % 2 == 1:
            result = poly_multiply(result,polycopy,modulus_poly)
        polycopy = poly_multiply(polycopy,polycopy,modulus_poly)
        n //= 2
    return result

def poly_pow(poly,k,modulus_poly=modulus_poly):
    if k==0:
        return [1]
```

```
    return poly_multiply(poly,poly_pow(poly,k-1,modulus_poly),modulus_poly)


list_pow = [[1]]
for k in range(254):
    list_pow.append(poly_multiply(list_pow[-1], [1,0], modulus_poly))

def poly_inv(poly):
    k = 0
    while poly and poly[0] == 0:
        del poly[0]
    while k < 255:
        if poly == list_pow[k]:
            return list_pow[-k]
        k+=1
    return None

def ell(i, T, I):
    result = [1]
    for j in I:
        if j!=i:
            U = poly_multiply(poly_add(T, list_pow[j]), \\
            poly_inv(poly_add(list_pow[i], list_pow[j])), modulus_poly)
            result = poly_multiply(result, U, modulus_poly)
    return result

def L(T, M ,I):
    result = [0]*8
    for k in range(4):
        result = poly_add(result,poly_multiply(ell(I[k], T, I), M[k], modulus_poly))
    return result

def eval(M, I):
    return [L(list_pow[i], M, I) for i in [0, 1, 2, 3, 5, 6, 7, 8]]
```

Puis, il a fallu décrypter le message. Seuls les 4 premiers octets du groupe seront ensuite gardés et convertis en ASCII pour reconstituer cette portion du message.
Si plus de 4 octets sont manquants dans le groupe considéré, alors on mettra des points d'interrogation pour les parties de messages qui n'ont pas pu être restaurées.

```
def decode(smalllist):
    I = []
    i = 0
    while len(I) < 4 and i < 8:
        if smalllist[i] != '??':
            if i < 4:
                I.append(i)
            else:
                I.append(i + 1)
        i += 1
    if len(I) != 4:
        list = smalllist
        for i in range(len(list)):
            if list[i] == '??':
                list[i] = '?'
        return list
    else:
        M = []
```

```
        for i in range(4):
            if I[i] >= 4:
                M.append(number_to_binary(int(smalllist[I[i] - 1],16)))
            else:
                M.append(number_to_binary(int(smalllist[I[i]],16)))
        List = eval(M, I)
        return [chr(binary_to_number(List[i])) for i in range(len(List))]

decrypted = []
for group in groups:
    decrypted.append(decode(group))

def merge(lists):
    result = []
    for l in lists:
        result += l[:4]
    # Join the strings in the list
    return ''.join(result)

print(merge(decrypted))
```

Nous obtenons le message suivant :

"*People of Earth, your attention, please. Thi? ? ?s Prostetnic Vogo? ?J ?ltz of the Galactic Hyperspace Planning Council. As you will no doubt be aware, the plans for developmen? ?of the outlyin? ?r ?gion? ?of the Galaxy require the building of a hyperspatial express route through you? ?star system. And regre? ?a ?ly, your planet is one of those schedule? ?for demolition. The process will take slightly less than two of your Earth minutes. Thank you.*"

En devinant les morceaux manquants, cela se traduirait par : *"Habitants de la Terre, votre attention s'il-vous-plaît. Nous sommes Prostetnic Vogon Jeltz du Conseil de Planification de l'Hyperespace Galactique. Comme vous le savez sans doute, les plans de développement des régions extérieures de la Galaxie nécessitent la construction d'une voie express hyperspatiale à travers votre système solaire. Et malheureusement, votre planète fait partie de celles programmées pour la démolition. Le processus prendra légèrement moins de deux de vos minutes terrestres. Merci."*

Merci, merci, c'est dommage ... ! Mais bon, vu le temps que nous avons passé à décrypter ce message (bien plus que 2 minutes) et que nous sommes toujours vivants, il se peut qu'il existe des petits farceurs au sein de l'Hyperespace Galactique, comme décrits dans le Guide...

Références

[1] Razvan Barbulescu et al. *A Heuristic Quasi-Polynomial Algorithm for Discrete Logarithm in Finite Fields of Small Characteristic*. Eurocrypt, 2014, p. 1-16. URL : https://link.springer.com/chapter/10.1007/978-3-642-55220-5_1.