Kryptoanaliza stosowana 2025

Lista zadań nr 4: RSA

Na zajęcia 3 listopada 2025

Liczby całkowite (typu `int`) w Pythonie mają nieograniczony rozmiar. Wbudowana trójargumentowa funkcja `pow`(b, e, n) wyznacza b^e mod n za pomocą algorytmu szybkiego potęgowania. Wprost z pudełka są więc dostępne w Pythonie podstawowe narzędzia do zabawy kryptografią asymetryczną. Nie są one jednak zbyt wydajne. Napisana w C biblioteka *GNU Multiple Precision Arithmetic Library* (GMP),[1] wykorzystywana m. in. przez różne biblioteki kryptograficzne (np. GNU TLS) i dostępna w Pythonie w postaci pakietu `gmpy2`[2] jest dużo bardziej efektywna. Korzystanie z niej nie jest jednak tak wygodne, jak z wbudowanego typu `int`, dlatego warto z niej korzystać *tylko* w razie konieczności, gdy program korzystający z pythonowej arytmetyki działałby zbyt długo. W poniższych zadaniach korzystanie z pakietu `gmpy2` nie jest zalecane.

Teksty z Zadania 1 i liczby z Zadania 7 są dołączone do niniejszego pliku jako załącznik.

W poniższych zadaniach używamy konwersji pomiędzy napisami i ciągami bajtów a liczbami w reprezentacji *big endian*. Odpowiednie funkcje są dla ustalenia konwencji przedstawione poniżej.

```
def int2bytes(n):
    return n.to_bytes((n.bit_length()+7)//8, 'big')

def int2str(n):
    return int2bytes(n).decode()
```

```
def bytes2int(b):
    return int.from_bytes(b, 'big')

def str2int(s):
    return bytes2int(s.encode())
```

Zadanie 1 (1 pkt). Oto klucz RSA zapisany[3] w znakowym formacie PEM:[4]

```
-----BEGIN PRIVATE KEY-----
MIIEvgIBADANBgkqhkiG9w0BAQEFAASCBKgwggSkAgEAAoIBAQC3LUP+867qYKlL
dmoCAi0UYepf69SG3yYITluGd6CS5LtPULw3xt14rOcHud1OGQYnkidW9YhbzxxJ
PCHAUbNNGYgy7dxsQS30eePkz/ydeBQ3x5J2kSwgg79wJtqgtRC9lpg4Kmw0W6+i
PC+RxRnLX6DQUw3vn+TSbEidjQBKq6N9nX7spAA20/cLCp/5y++mpPkD7LTi2Ic0
HvXFSFFAQRCTwC6e/mJeN9j4e/3uTdgdMOzLkROhBMXh5ArbjkTqKrrEqfAC0WBl
jIKCqOs4rGClvzsqYH36tgasQA/ZuC2EI2HrMBJmQflbJtL8L8KVdekD04l1RxBp
jd3mCpwbAgMBAAECggEBAIE6Ov0zt1P8i5MEYDc4h8sYZPv06UOZTECR3zoXTT40
hfynzZFgB8Dp4jduJbXCK7hhAMX1hJSYobD87ia6vM8I0zBy5I2u0hfmoiM7pFEH
dOQBNM/XDRWHPNiHYLf95WiLRKpjK2Eoydifs0OYlOEIfvBZl7xUxl65K+WSowEJ
sZRJeFYuttGpXiTyFH5Epjw1WshTy0hrepbvG8OOeotOVe5mP7QDvDh4D6RK4jNO
U7A0ZCthJhiy/KPGorQPvAeR2ogAHU8Tk3EOv1cynIzNXObuOxvjPhyv1KtnDnPj
5bbLfClUhfcCrP6WnCnGnFb28CFfS1lTk4Lb/7/lXckCgYEA59xbMiADvAZLor91
KCYLwEDaqbkJdQatP84+3pKNfPq0vLcaUme+ghqMZIwd5Xn8YaRFyF/UBV6L7Kdw
gyoIoEMZaBncEeawKu1OxWc74pHMpA2fbzQkNNCTOUMNj1zZQGWd7iVogp05fSnc
mOo7UI5xsdJQszdDowEJAO+RcI8CgYEAyj9do04B2Wb55I7pZPo+IBmQzDUblZe6
dsDG3bRb7PZIONZtzcNSA4Lk910+SHsMVvsaNxZNSNowZVu4n20pIbJsr/AHYi/Q
bBOx6vyHtOMJ6y4rM9yY+VmhznkEjqg2Qy/eDfA4y7mjDdcNHfqpiR/SBGsD6lUT
Ow2Sc/MvCbUCgYEApjyrdgjWJJ0u5YT37s04Z6MYI8/zI+Chnrm1Acd5gj4OD7AI
Q0T2pjl6pyx+0uTfOvKYxc1PKnpbQFXcqW4duSnisWy5CGypql1yuL4Hha6bVpQd
15E+1E8m/0tAEaW0biNtVC2l/N/VNBQrgceOt3cptSDhk1zwNZKsbetWD6kCgYAL
gsO/kacFSpkaaw/ZkWyLwyi8lqE1SGdybtJit+srd3Dw2zNwrJc0X/TMJ2BuG6K+
KJh7XieT5+ctPaUqXbojktYgxVaGPhZssEpGB41pvCpKWlxXHUOPlAxjhfu+atB7
znOi83MoFS60kPLMLAIOLeN0IOseIS1xBPq64rJgqQKBgFNoGn6byhDoZfSE2X/a
```

[1] `https://gmplib.org/` *«Arithmetic without limitations»*

[2] `https://pypi.org/project/gmpy2/`

[3] W załączniku `lista_04.txt` umieszczono ten klucz w formacie tekstowym.

[4] PEM to skrót od *Privacy Enhanced Mail* (RFC 989 Feb 1987, 1040 Jan 1988, 1113–1115 Aug 1989, 1421–1424 Feb. 1993), obecnie już martwego standardu szyfrowania poczty, który został wyparty przez OpenPGP i S/MIME. Format PEM znakowej reprezentacji materiału kryptograficznego pozostał natomiast popularny do dziś i przejęty przez inne protokoły.

```
NzIKzNq8y6Wcb0t1Cu4PrUIF8KIM+fp8vQ7mqTEhpSxzqNkyKzmNnaKZ321jJFZv
lcbo1djDwk/2jEHLGba7sOiuOZJ1WpZnVaAiW2TrkTopQf8fkU/SpA/Zm4XvLzfb
jELxdU7HXEKc9qZitRb3kQ2D
-----END PRIVATE KEY-----
```

Format PEM enkapsuluje binarny format DER (*Distinguished Encoding Rules*), który jest sposobem reprezentacji danych serializowanych za pomocą ASN.1 (*Abstract Syntax Notation One*). Plik DER ma strukturę rekurencyjną — zawiera elementy, które zawierają dalsze elementy. Każdy element pliku ma swój typ i składa się z nagłówka i treści o określonych długościach.

Z pliku PEM można wyekstrahować binarny plik DER (jest on w nim zapisany jako `OCTET STREAM`) poleceniem:

```
openssl pkey -inform PEM -outform DER -in key.pem -out key.der
```

Zawartość binarnego pliku DER można ujawnić poleceniem:

```
openssl pkey -inform DER -in rsakey.der -text
```

natomiast jego strukturę ASN.1 poleceniem:

```
openssl asn1parse -in key.der -inform DER
```

Dowiadujemy się, że plik `key.der` na poziomie zagnieżdżenia `d` $= 0$ zawiera ciąg (`cons: SEQUENCE`) o 4-bajtowym nagłówku i długości 1188 bajtów. Składa się nań 9 liczb całkowitych (`prim: INTEGER`), wyświetlonych dalej w postaci szesnastkowej. Są to w kolejności (plik DER nie zawiera tych informacji, bo znaczenie wartości nie jest częścią protokołu serializacji; znaczenie określa standard PKCS#1, zob.: J. Jonsson, B. Kaliski, *Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1*, RFC 3447, Feb. 2003):

1. wersja, dla PKCS#1 ver. 2.1 równa `0` (można używać, co tu pomijamy, więcej niż dwóch liczb pierwszych; wówczas wersja jest równa `1`);
2. moduł $n = pq$;
3. wykładnik publiczny e;
4. wykładnik prywatny $d = e^{-1} \bmod \operatorname{lcm}(p-1, q-1)$;[5]
5. liczba pierwsza p;
6. liczba pierwsza q;
7. wykładnik $d_p = e^{-1} \bmod (p-1)$;
8. wykładnik $d_q = e^{-1} \bmod (q-1)$;
9. współczynnik CRT $q_{inv} = q^{-1} \bmod p$.

Do szyfrowania wystarczą n i e, do deszyfrowania p, q i e. Pozostałe liczby są wyliczone w czasie generowania klucza w celu przyspieszenia korzystania z klucza.[6]

Sprawdź, że faktycznie w przykładowym kluczu $n = pq$, $d = e^{-1} \bmod \operatorname{lcm}(p-1, q-1)$, $d_p = e^{-1} \bmod (p-1)$, $d_q = e^{-1} \bmod (q-1)$, zaś $q_{inv} = q^{-1} \bmod p$.

Oto szyfrogram (szesnastkowy zapis liczby $c = \mathrm{RSA}_{(n,e)}(m) = m^e \bmod n$ w konwencji *big endian*) zaszyfrowany za pomocą powyższego klucza (n, e):

```
59dc3b0327e09b6b762385ac3fa1724a5c3761338018c1c69d691d563be751377119397a86bd0873
b9dd4259b94cdc7398fecbe9c252a4ef682bb09c17afad343e4c38f264b12abfff5aef54f830a08b
8c30e334bbd94a003310b5678a238135b1b89c324246129a6d8fcc34aa18386444ae3bf1d4bb7adb
603da50b44683e024e32dcee966531d1204d34a4cd51cf53431acb3044588a4de8ca124154308f55
7796873763fd625767b4296266e2c7ce07f891b8c6716deb78de0b84c6883fc42633a6ceee2b4118
e3c105c55de092a61fbca5cc4ec1f9a29c45a688d18d49ae5296363668e30c611e4ecbaacdfc4af9
458110a80ecab25a6e2b19745271c1be
```

Odszyfruj go.[7] Klucz prywatny (n, d) odczytaj z pliku DER ręcznie za pomocą polecenia `openssl` (jak powyżej) lub (lepiej) używając jednej z licznych pythonowych bibliotek do serializacji danych w formacie ASN.1. Tekst jawny jest napisem w kodowaniu UTF-8. Skąd pochodzi ten cytat?[8]

[5] Zob. Zadanie 2.
[6] Zob. Zadanie 3.
[7] Kolejny przykład kryptoanalizy albańskiej.
[8] Warto przeczytać tę pracę!

Zadanie 2 (1 pkt). W praktyce zamiast funkcji Eulera ϕ do wyznaczenia pary (e, d) w szyfrowaniu RSA używa się funkcji Carmichaela:

$$\psi(n) \quad = \quad \min\{m \in \mathbb{N}^+ \mid \forall a \in \mathbb{N}^+ \; a \perp n \Rightarrow a^m \equiv 1 \bmod n\}.$$

Jeśli $n = pq$ dla p i q pierwszych, to

$$\begin{aligned} \phi(n) &= (p-1)(q-1), \\ \psi(n) &= \operatorname{lcm}(p-1, q-1) = \phi(n)/\gcd(p-1, q-1). \end{aligned}$$

Użycie ψ pozwala znaleźć wykładnik

$$d \quad \equiv \quad e^{-1} \pmod{\psi(n)}$$

mniejszy niż przy użyciu ϕ, a równie dobry do szyfrowania — udowodnij, że jeśli $e \in \mathbb{Z}^*_{\psi(n)}$ i $d \equiv e^{-1} \bmod \psi(n)$, to $m^{ed} \equiv m \bmod n$ dla dowolnego $m \in \mathbb{Z}_n$.

Większość oprogramowania akceptuje klucze wyliczone za pomocą ϕ, ale większość też (zgodnie z PKCS#1) generuje klucze używając ψ.

Zadanie 3 (1 pkt). Aby przyspieszyć potęgowanie $c = m^e \bmod n$ podczas szyfrowania, wykładnik e zwykle dobiera się tak, by miał w rozwinięciu binarnym tylko dwie jedynki. Aby e było odwracalne modulo $\psi(n) = \operatorname{lcm}(p-1, q-1)$ tj. aby $e \perp \psi(n)$, e musi być nieparzyste, a zatem musi być postaci $2^i + 1$. Dla liczby $e = 2^i + 1$ obliczenie $m^e \bmod n$ wymaga $i + 1$ mnożeń liczb rzędu n (używając algorytmu szybkiego potęgowania potrzebujemy i razy podnieść m do kwadratu, a następnie pomnożyć przez m). Aby zwiększyć prawdopodobieństwo, że $e \perp \psi(n)$, jako e wybieramy nieparzystą liczbą pierwszą.[9] Jeśli liczba $2^i + 1$ jest pierwsza, to $i = 2^j$ dla pewnego j, tj. liczba ta jest liczbą pierwszą Fermata.[10] Pierwsze pięć liczb Fermata to liczby pierwsze:

$$\begin{aligned} 3 &= 2^1 + 2^0 = 2^{2^0} + 1 = F_0, \\ 5 &= 2^2 + 2^0 = 2^{2^1} + 1 = F_1, \\ 17 &= 2^4 + 2^0 = 2^{2^2} + 1 = F_2, \\ 257 &= 2^8 + 2^0 = 2^{2^3} + 1 = F_3, \\ 65537 &= 2^{16} + 2^0 = 2^{2^4} + 1 = F_4. \end{aligned}$$

Wiadomo że F_j, gdy $5 \le j \le 32$, są złożone, zatem nasz wybór jest ograniczony do pięciu powyższych liczb. Przeważnie używa się $e = 65537$ (wymaga 17 mnożeń), choć dawniej używano powszechnie $e = 3$ (wymaga 2 mnożeń). Ze względu na pewien atak na mały wykładnik e, który zbadamy za tydzień, używanie $e = 3$ wymaga ostrożności. Wartości $e = 5$, 17 oraz 257 nigdy nie były popularne.

Niestety wykładnik $d = e^{-1} \bmod \operatorname{lcm}(p-1, q-1)$ ma zwykle bardzo dużo jedynek. Zatem np. w protokole TLS powyższa procedura pozwala obniżyć złożoność obliczeń u klienta, ale złożoność obliczeń na serwerze pozostaje duża. Wolelibyśmy odwrotnie — przenieść koszt zestawienia połączenia na klienta.[11] Dowcipne pytanie: czemu nie można zatem wybrać najpierw d tak, żeby miało tylko dwie jedynki kosztem tego, że e będzie miało ich tysiące?

Aby przyspieszyć potęgowanie podczas deszyfrowania, można jednak skorzystać z chińskiego twierdzenia o resztach (CRT). Niech

$$\begin{aligned} d_p &= e^{-1} \pmod{p-1}, \\ d_q &= e^{-1} \pmod{q-1}, \\ p_{inv} &= p^{-1} \pmod{q}, \\ q_{inv} &= q^{-1} \pmod{p}. \end{aligned}$$

Obliczenie

$$\begin{aligned} m_p &= c^{d_p} \pmod{p}, \\ m_q &= c^{d_q} \pmod{q} \end{aligned}$$

jest szybsze, niż obliczenie $c^d \bmod n$ (jak bardzo?). Na mocy CRT mamy wówczas

$$c^d \bmod n \quad = \quad qq_{inv}m_p + pp_{inv}m_q \pmod{n}.$$

[9] Jeśli okaże się, że dla ustalonego e nie zachodzi $e \perp \psi(n)$, to ponownie losujemy p i q.

[10] Zob. odpowiednie ćwiczenie z Matematyki Dyskretnej.

[11] Kiedyś moc obliczeniowa klientów była znikoma i wówczas $e = 3$ ułatwiało osiągnąć akceptowalną szybkość zestawienia połączenia pod warunkiem, że serwer TLS był dostatecznie wydajny. Przyjęcie $e = 3$ pozwalało zestawić połączenie TLS *jedynie* 400 razy wolniej, niż nieszyfrowane połączenie TCP. Obecnie klient da radę wykonać szybko bardziej skomplikowane obliczenia, więc po co płacić za moc obliczeniową serwerów?

Ostatnie obliczenie można zastąpić jeszcze bardziej efektywnym *algorytmem Garnera* (H. Garner, The Residue Number System, *IRE Transactions on Electronic Computers* EC-**8**(6):140–147, June 1959; zob. też: A. Menezes, P. van Oorschot, S. Vanstone, *Handbook of Applied Cryptography*, CRC Press 1996, Algorytm 14.71, str. 612 oraz J.-J. Quisquater, C. Couvreur, Fast Decipherment Algorithm for RSA Public-Key Cryptosystem, *Electronics Letters* **18**(21):905–907, Oct 1982):

$$\begin{aligned} h &= q_{inv}(m_p - m_q) \pmod p, \\ c^d \bmod n &= m_q + hq. \end{aligned}$$

Do wyznaczenia $c^d \bmod n$ potrzebujemy p, q, d_p, d_q oraz q_{inv}, dlatego znajdują się one w kluczu prywatnym. Wartość n nie jest potrzebna. Przechowuje się ją w kluczu na potrzeby tych, którzy nie mają zaimplementowanego powyższego usprawnienia i obliczają $c^d \bmod n$ wprost z definicji (np. nas w poprzednim zadaniu).

Zaimplementuj tę wersję obliczania $c^d \bmod n$ i porównaj jej efektywność ze zwykłym wywołaniem $\texttt{pow}(c, d, n)$.

Zadanie 4 (1 pkt). Clifford Cocks w notatce z 20 listopada 1973 roku (pięć lat przed RSA) zdefiniował szyfr implementujący ideę kryptografii asymetrycznej, opisanej przez J. H. Ellisa w notatce ze stycznia 1970 roku (pięć lat przed Merklem, sześć lat przed Diffie i Hellmannem). Wylosuj takie dwie liczby pierwsze p i q, żeby $p \nmid (q-1)$ oraz $q \nmid (p-1)$. Oblicz klucz publiczny $n = pq$. Klucz prywatny: za pomocą algorytmu Euklidesa wyznacz $p_{inv} = p^{-1} \bmod q$ oraz $q_{inv} = q^{-1} \bmod p$. Szyfrowanie: dla danej wiadomości $m < n$ oblicz $c = m^n \bmod n$. Deszyfrowanie: dla danego szyfrogramu c korzystając z chińskiego twierdzenia o resztach rozwiąż układ równań:

$$\begin{cases} m' &\equiv c^{p_{inv}} \pmod q \\ m' &\equiv c^{q_{inv}} \pmod p. \end{cases}$$

Udowodnij poprawność tego algorytmu, tj. pokaż, że dla dowolnego $m < n$ jest $m' = m$. Czym różni się ten algorytm od RSA?

Zadanie 5 (2 pkt). Do generowania wielkich liczb pierwszych stosuje się przeważnie test Rabina-Millera. Jeśli $\tilde{p}$ jest nieparzystą liczbą pierwszą, $\tilde{p} - 1 = 2^s d$, gdzie d jest nieparzyste, to dla dowolnego $a = \mathbb{Z}^*_{\tilde{p}}$ albo $a^d \equiv 1 \bmod \tilde{p}$, albo $a^{2^k d} \equiv -1 \bmod \tilde{p}$ dla pewnego $k = 0, \ldots, s-1$. Przeciwna implikacja nie zawsze jest prawdziwa, ale zachodzi dla co najmniej $\frac{3}{4}\phi(\tilde{p})$ wartości a. Aby zweryfikować, czy liczba $\tilde{p}$ jest złożona, można wybrać liczbę $a \in \{2, \ldots, \tilde{p} - 2\}$ i sprawdzić, czy

$$a^d \not\equiv 1 \bmod \tilde{p} \quad \wedge \quad \bigwedge_{k=0}^{s-1} a^{2^k d} \not\equiv -1 \bmod \tilde{p}.$$

False positive nie jest możliwy, ale test może dać *false negative*. Liczba a która nie przechodzi powyższego testu dla złożonej liczby $\tilde{p}$ nazywa się *silnym kłamcą* dla $\tilde{p}$.

Wśród liczb od 2 do 9999 mamy 1229 liczb pierwszych, które oczywiście nie przechodzą testu na złożoność. Spośród pozostałych 8769 liczb złożonych 2554 liczby mają co najmniej jednego silnego kłamcę. Rekordzistkami są 1891, mająca 23.73% silnych kłamców (wiemy, że nie może być ich więcej niż 25%) oraz w liczbach bezwzględnych: 8911 mająca 1780 silnych kłamców (19.98%). Dla większych liczb silni kłamcy się zdarzają, ale są proporcjonalnie nieliczni (np. 1 000 001 ma ich 3748, ale to tylko 0.37%). Jeśli będziemy losować wartości a i powtarzać test k-krotnie, to prawdopodobieństwo błędu wyniesie 4^{-k}. W praktyce, jeśli $\tilde{p}$ jest duża i wylosowana (a nie złośliwie przygotowana przez atakującego), to prawdopodobieństwo błędu dla losowo wybranego a jest znacznie mniejsze niż 1/4 (zob. I. Damgård, P. Landrock, C. Pomerance, Average case error estimates for the strong probable prime test, *Mathematics of Computation* **61**(203):177–194).

Zaimplementuj ten test. Wylicz statystyki silnych kłamców dla rozsądnych zakresów liczb.

Zaprogramuj następnie funkcję znajdującą najmniejszą liczbę pierwszą nie mniejszą niż podana wartość n. Postępuj tak: jeśli n jest parzysta, zwiększ ją o 1. Następnie testuj liczby $n + 2i$, dla $i = 0, 1, 2 \ldots$, aż znajdziesz liczbę pierwszą. Do testowania użyj algorytmu Rabina-Millera, powtarzając go ustaloną liczbę razy z losowo wygenerowanymi wartościami a.[12]

Testowanie na pierwszość możesz znacznie przyspieszyć dzieląc najpierw testowaną liczbę przez małe liczby pierwsze. Np. najmniejszą liczbą pierwszą większą niż $x = 2^{2047} + 2^{1023}$ jest $x + 805$. Spośród 402 liczb złożonych $x + 2i + 1$ (dla $i = 0, \ldots, 401$), które należało sprawdzić, 350 miało najmniejszy czynnik pierwszy mniejszy od 10 000, 360 — mniejszy od 100 000, 366 — mniejszy od miliona, a tylko 36 liczb miało najmniejszy czynnik większy od miliona. Jeśli więc będziemy najpierw próbować dzielić przez liczby pierwsze mniejsze od miliona, to test Rabina-Millera będzie uruchomiony jedynie 37 razy na 403 próby (ostatnia zakończy się powodzeniem).

[12] Dla dużych n zwykle już pierwsza próba wskaże liczbę złożoną. W praktyce powtarzanie testu będzie więc wykonywane tylko dla ostatniej liczby, która z dużym prawdopodobieństwem okaże się pierwsza. Liczbę iteracji dobierz tak, by nie czekać zbyt długo na wynik.

Liczb pierwszych mniejszych od 10 000 jest 1229, mniejszych od 100 000 jest 9592, zaś mniejszych od miliona jest 78498. Nie warto więc wyliczać ich zbyt wiele. Dobierz optymalną wartość.

Zaprogramuj następnie funkcję generującą klucz RSA. Należy wylosować liczby pierwsze p i q tak, by $n = pq$ miało zadaną liczbę bitów k (np. $k = 4096$). Jest wiele sposobów na wylosowanie takich liczb. Możemy np. wylosować dwie liczby $k/2$-bitowe. W tym celu losujemy dwa ciągi $k/2 - 2$ losowych bitów i dodajemy do nich z przodu i z tyłu jedynkę. W ten sposób wylosowaliśmy liczby nieparzyste z przedziału $[2^{k/2}, 2^{k/2+1})$. Znajdujemy najmniejsze liczby pierwsze p i q nie mniejsze od tak wylosowanych liczb. Z prawdopodobieństwem bliskim pewności obie liczby pierwsze też będą $k/2$-bitowe, tj. będą należeć do przedziału $[2^{k/2}, 2^{k/2+1})$. Ich iloczyn będzie więc należeć do przedziału $[2^k, 2^{k+2})$, będzie więc liczbą k- lub $k+1$-bitową. Jeśli zależy nam, by iloczyn był dokładnie k-bitowy, powyższą procedurę można powtarzać.

Dokończ generowanie klucza wyznaczając dla podanego wykładnika e jego element odwrotny modulo $\phi(n)$ lub lepiej $\psi(n)$ (użyj uogólnionego algorytmu Euklidesa). Zauważ, że odwracanie e nie powiedzie się, jeśli $\gcd(e, \psi(n)) \neq 1$ (algorytm Euklidesa odwracając e przy okazji wylicza tę wartość). Jeśli zależy nam na konkretnej wartości e, to generowanie liczb pierwszych należy powtórzyć. Jeśli e jest liczbą pierwszą, to prawdopodobieństwo, że przydarzy się nam taki kłopot wynosi $1/e$, więc jest dla $e = 65537$ dosyć małe, ale dla $e = 3$ sprawia problemy. Wygenerowane liczby p i q sprawdza się również pod kątem różnych podatności. Na przykład $p - 1$ i $q - 1$ powinny mieć duże czynniki pierwsze itd.

Oszacuj ile bitów losowych potrzeba do wygenerowania klucza o danej długości k. Sprawdź oszacowanie generując kilka kluczy.

Losowość pobierałem w Pythonie za pomocą funkcji `os.urandom`, choć od wersji 3.6 można też używać modułu `secrets.SystemRandom`, oferującego bardziej wysokopoziomowy interfejs. Oczywiście nie należy używać modułu `random`!

Zadanie 6 (1 pkt). Zajrzyj do kodu jakiegoś programu lub biblioteki implementujących szyfr RSA (OpenSSL, GnuPG, GnuTLS itp.). Jak tam generowane są liczby p i q oraz e i d? Czy stosują jakieś inne metody, niż opisane w poprzednim zadaniu?

Zadanie 7 (2+5 pkt). Szyfrowanie RSA staje się podatne, jeśli jego podstawowe parametry (p, q i e) nie są właściwie dobrane. Pewne ograniczenia należy nałożyć nawet na szyfrowaną wiadomość m! Opracuj ataki na RSA w przypadku, gdy:

- n nie jest dostatecznie duże;[13]
- p lub q nie jest dostatecznie duże;
- różnica $|p - q|$ nie jest dostatecznie duża. Wówczas można zastosować algorytm faktoryzacji wynaleziony przez Fermata. Zauważmy, że $(p+q)^2 - (p-q)^2 = 4n$, więc jeśli $|p-q|$ jest małe, to $(p+q)^2$ nieznacznie przekracza $4n$, a więc $x = (p+q)/2$ nieznacznie przekracza $\lfloor\sqrt{n}\rfloor$.[14] Szukamy takiego $x > \sqrt{n}$, żeby $x^2 - n$ było kwadratem pewnej liczby y. Wtedy $n = x^2 - y^2$. Jak bardzo ten algorytm jest szybszy od zwykłego wyszukiwania dzielnika n większego niż $\sqrt{n}$?
- m nie jest dostatecznie duże lub jego entropia jest mała (np. wiemy, że wiadomością jest „`Niniejszym zaświadcza się, że liczba punktów przyznawanych za to zadanie wynosi ...`”, tylko nie wiemy, ile). Dlatego tak ważny jest *padding* (najlepiej zrandomizowany), którym zajmiemy się później.

Złam następnie poniższe szyfrogramy (po jednym punkcie za każdy). We wszystkich przypadkach przekonwertowano napis UTF-8 na liczbę i zaszyfrowano ją za pomocą klucza publicznego (n, e) otrzymując szyfrogram c. Trójki (n, e, c) podano w zapisie dziesiętnym.[15]

Szyfrogram 1.

$$
\begin{aligned}
n &= 95039348686345519775275209679375306731181331400983736849866628265918539882878\\
&\quad 43158456473141002826917471097524750757360944895306548727093962803321388723504\\
&\quad 28019836476714894596975619694031094375247839653280895061312326129464129035577\\
&\quad 11709359854561944488361470216531319834712066071233170587785911540042579698907\\
e &= 3\\
c &= 76174535651052895376915037596288799975843956798412006031780298843292094616237\\
&\quad 72036630671643624768579726075728065229831029416930275008703956342076605609592\\
&\quad 88145870240629551430894558842601937622628283783834950684553257470744888739701\\
&\quad 78318524145259063288118659857483469331707700010203651514076827618723762827292
\end{aligned}
$$

Szyfrogram 2.

n = 16984672065805841868743142368918158329967383083904040727042534398673406377566144115573679021027432950560441632760706567627811075996000308483786790405894324915178712730193426943242173992540478206366058381059724840768305105444663251985009342938280931630396405369426679132010890098494905351274669158752714623197900 = 3

c = 37675579701159906245501206415596555240149102832105763736083378600977411624967257526032556649820858744741731006616680765006221591658850229788324509534372067253342069914381960782334314885677548034670016093712209217259500726870368385455441122538949740859384811682193117974262341818602594113385283674778199344605

Szyfrogram 3.

n = 10224450755003945665664517319057070894333419768588437138319271292355834244580732268997325471016619094452787910204679700308119769485402098138200517613986716814651959577428175515562986129151369882216487607837588051636091917997936812491695299201012052851829036946408875803854799145859435415188332070223074140702100 = 3

c = 62057984030869688007344897545444088014053229140520739023398036549804513554187302368646898163319719487662967854883843708191719446586622128218614823062462620494799562276323459706109603433640902011749478795426087706463227112410824687632610734979910849764856022166871507194913981458497804940452952622970258764222

Szyfrogram 4.

n = 67191827727309137341573532809939931380368665038072407741205201371709602251718509563825029610455534376814205366654071169399146227164775602405456829446027676217016558670696590007846444036399968841672658346467031116895663131732332376643140855204552240829424609061279841648093389752079212672599318672005545019803 = 3

c = 34992590276975757936895571599234534565326134075413688554464523215675901020385043951376837452801562839043039923369378002695152625257193357761990656530962177360586924191104275120676906877547308719976541991630899117412671607365533984715202035150784585550518496497194286828836614399861356101284369850299936777 8

Szyfrogram 5.

n = 188623968747499

e = 3

c = 109688584900477

Uwaga: żaden punkt tego zadania nie wymaga dużej mocy obliczeniowej. We wszystkich przypadkach używałem niezbyt optymalnego kodu w Pythonie, a Maleństwo z łatwością dało radę. Użyj sprytu, nie siły!

Zadanie 8 (1 pkt). Rivest, Shamir i Adleman sugerowali w oryginalnej pracy,[16] by p i q różniły się o kilka rzędów wielkości („should differ in length by a few digits"). Nic jednak nie stoi na przeszkodzie, by obie miały dokładnie tyle samo bitów, byle by ich różnica była duża! Przypuśćmy, że losujemy dwie dokładnie k-bitowe liczby n i m. Jaka jest wartość oczekiwana $|n-m|$? Jakie jest prawdopodobieństwo, że $|n-m| \geq 2^{k-i}$ dla $i = 2, \ldots, k$? Jeśli wylosujemy dwie dokładnie k-bitowe liczby n i m i znajdziemy najmniejsze liczby pierwsze p i q nie mniejsze od nich, to dla dużych k będzie $|n-m| \approx |p-q|$. Czy na potrzeby RSA dla tak wygenerowanych liczb pierwszych różnica $|p-q|$ będzie dostatecznie duża, żeby wyeliminować atak z poprzedniego zadania?

Zadanie 9 (1 pkt). Zaprogramuj protokół łamigłówek Merkle'a. Użyj DES-a[17] i przyjmij, że $n = 20$. Zaatakuj protokół i oszacuj czas łamania klucza.

[13] Ponoć NSA radzi sobie z faktoryzacją $n \sim 2^{1024}$, mogliby więc siłowo złamać *wszystkie* szyfrogramy z tego zadania!

[14] Zauważ, że pierwiastkowanie w $\mathbb{Z}$ jest łatwe — wystarczy użyć bisekcji i potęgowania. Niestety nie przenosi się to na $\mathbb{Z}_n^*$, gdyż bisekcja wymaga liniowego porządku, a potęgowanie musi być funkcją rosnącą...

[15] W załączniku `lista_04.txt` zapisano poniższe liczby w formacie tekstowym.

[16] R.L. Rivest, A. Shamir, L. Adleman, *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems*, CACM **21**(2):120–126 Feb 1978.

[17] Np. implementacji w Pythonie dodanej jako załącznik `des.tar.xz` do niniejszego pliku.