Report Project 2

Carl-Anton Grandelius, cagra@kth.se & Suhaib Abdi Muhummed, muhummed@kth.se

Course code: IX1500_HT2024
Date: 2024-10-02

1 Sammanfattning med Resultat

I detta projekt har vi implementerat två huvudsakliga uppgifter. I Task 1 har vi dekrypterat det meddelande som Alice skickade till Bob enligt RSA med hjälp av enbart en offentlig nyckel. RSA-krypteringsalgoritmen bygger på den matematiska svårigheten att faktorisera stora tal i deras primtalskomponenter. I denna lösning kan man dock inte använda ”trial division” för att faktorisera modulen $nBob$, vilket gör det möjligt att härleda den hemliga nyckeln och använda den för dekryptering. Detta eftersom p och q i vårt specifika fall är för stora primtal för datorn att beräkna på en rimlig tid. Därför har vi använt oss av sympy-bibliotekets mycket effektiva faktoriseringsfunktion $factorint$.
Siffrorna i listan (chiffertexten) var av samma storlek eftersom meddelandet har delats upp i block av samma storlek inom RSA-kryptering. Detta gör att varje block ligger inom $0 < m < n$.

I Task 2 har vi undersökt hur lång tid det tar att knäcka RSA-nycklar av olika storlekar (100–200 bitar) och extrapolerat resultaten för att förutse tiden att knäcka större nyckelstorlekar (1024, 2048 och 4096 bitar). Resultaten visar att tiden ökar exponentiellt med större nyckelstorlekar, och extrapoleringen visar att det skulle vara extremt tidskrävande att knäcka nycklar över 1024 bitar.

1.1 Resultat Task 1

Resultatet efter dekryptering av den krypterade texten blev ”RSA Cryptography is the world’s most widely used public-key cryptography method for securing communication on the Internet. Instrumental to the growth of e-commerce, RSA is used in almost all Internet-based transactions to safeguard sensitive data such as credit card numbers. Introduced in 1977 by MIT colleagues Ron Rivest, Adi Shamir, and Leonard Adleman, RSA -its name derived from the initials of their surnames- is a specific type of public-key cryptography, or PKC, innovated in 1976 by Whitfield Diffie, Martin Hellman, and Ralph Merkle. Intrigued by their research, Rivest, with Shamir and Adleman, developed cryptographic algorithms and techniques to practically enable secure encoding and decoding of messages between communicating parties”.

1.2 Resultat 2

Figur 1 visar tiden att knäcka RSA-nycklar för storlekar mellan 100 och 200 bitar. Figur 2 visar extrapoleringen för nycklar upp till 4096 bitar.

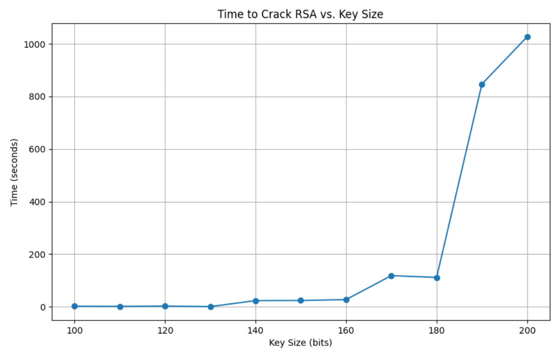

Figure 1: Tid att knäcka RSA för nyckelstorlekar 100-200 bitar

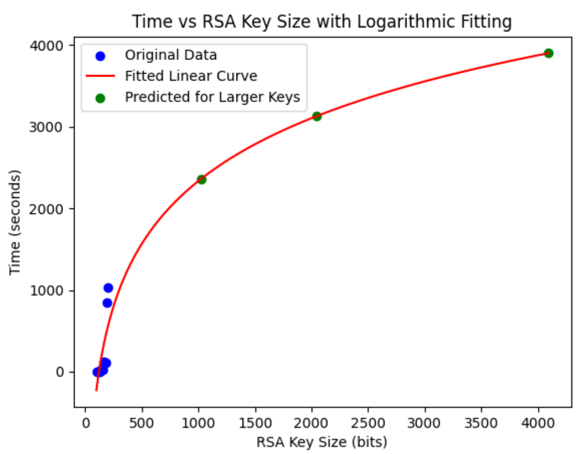

Figure 2: Extrapolering för större nyckelstorlekar (1024, 2048, och 4096 bitar)

Med hänsyn till de små RSA-nyckelstorlekarna ökar tiden för att knäcka krypteringen markant, men inte explosivt. Den logaritmiska funktionen modellerar en stabil ökning som stämmer väl överens med de observerade data för mindre nycklar och ger realistiska uppskattningar för större nycklar. Den växer långsamt över tid och undviker extrem överestimation.

1.3 Sammanfattning av kodimplementation för Task 1

Den huvudsakliga dekrypteringsprocessen utförs i funktionen `dekryptera_rsa`. För att dekryptera meddelandet måste vi känna till primtalsfaktorerna p och q för n, som används för att beräkna den hemliga nyckeln.

- Först faktoreras modulen $nBob$ för att få fram primtalen p och q. Om modulen inte är produkten av exakt två primtal, genererar funktionen ett fel. När de två primtalen har hittats returneras dessa.

$$p, q = \texttt{faktorisera\_modul}(nBob)$$

- Därefter beräknas $\varphi(n)$, som är Eulers totientfunktion, med formeln $(p-1) \times (q-1)$. Den hemliga nyckeln d beräknas sedan genom att ta den modulära inversen av den offentliga exponenten e modulo $\varphi(n)$. Funktionen `modul_inv` använder den utökade Euklides algoritmen för att beräkna denna invers:

$$\texttt{hemlig\_nyckel} = \texttt{modul\_inv}(eBob, \varphi(n))$$

- Den utökade Euklides algoritmen beräknar den största gemensamma delaren (gcd) mellan två heltal och hittar koefficienterna x och y som uppfyller ekvationen:

  $$a \times x + b \times y = \gcd(a, b)$$

  Denna algoritm är viktig eftersom den används för att beräkna den modulära inversen.

- Chiffertexten består av flera block, och dessa bearbetas i omvänd ordning. Detta görs eftersom blocken kan ha sparats baklänges under krypteringen. Varje block dekrypteras genom att upphöja blocket med den hemliga nyckeln modulo $nBob$, vilket utförs med funktionen `pow()`:

  $$\texttt{meddelande} = \texttt{pow}(chiffer, hemlig\_nyckel, nBob)$$

  Här sker den faktiska RSA-dekrypteringen, där $chiffer$ är varje block av chiffertexten och $meddelande$ är det dekrypterade resultatet.

- När ett heltal har dekrypterats måste det omvandlas till en byte-sekvens för att kunna avkodas som text. Funktionen `heltal_till_bytes` beräknar hur många bytes som behövs för att representera det dekrypterade heltalet och returnerar motsvarande byte-sekvens. Denna sekvens kan sedan omvandlas till en sträng för att få det ursprungliga meddelandet:

$$\texttt{dekrypterad\_text} = \texttt{''.join}(\texttt{heltal\_till\_bytes}(m).decode('utf{-}8', errors = 'ignore')$$
$$\text{for } m \text{ in } \texttt{dekrypterade\_meddelanden})$$

- Efter att alla chifferblock har dekrypterats och konverterats till text omvänds hela den dekrypterade texten med hjälp av `[::-1]`. Detta steg är avgörande eftersom texten kan ha kodats baklänges under krypteringen, och omvändningen ger tillbaka texten i rätt ordning:

$$\texttt{dekrypterad\_text} = \texttt{dekrypterad\_text[::-1]}$$

1.4 Sammanfattning av kodimplementation för Task 2

I Task 2 använde vi liknande kod som i Task 1, men med justeringar för att knäcka RSA-nycklar av olika storlekar (100–200 bitar) och mätte tiden för varje försök. Genom att generera slumpmässiga primtal för varje nyckelstorlek och sedan beräkna den publika och privata nyckeln, kunde vi dekryptera ett krypterat meddelande och logga tiden det tog att knäcka nyckeln.

Efter att ha samlat data för nyckelstorlekar mellan 100 och 200 bitar, extrapolerade vi resultaten för större nycklar (1024, 2048 och 4096 bitar) med hjälp av en logaritmisk modell.

2 Formler och ekvationer

RSA-algoritmen möjliggör både kryptering och dekryptering. Även om uppgiften specifikt anger att vi ska dekryptera en viss sträng, blir det enklare att förstå dekrypteringsprocessen om vi först tittar på hur krypteringen och skapandet av nycklarna fungerar. I uppgiften har vi enbart den offentliga nyckeln (n, e), ej den hemliga nyckeln d för att dekryptera chiffertexten.

Av den anledningen har detta avsnitt delats upp i fyra underrubriker: Nyckelgenerering, kryptering, dekryptering och dekryptering med offentlig nyckel.

2.1 Nyckelgenerering

1. Välj (slumpmässigt) två stora primtal p och q.
2. Beräkna modulus n:
$$n = p \times q$$
3. Beräkna Eulers totientfunktion $\varphi(n)$. Ifall p och q är två stora primtal är det mycket svårt att få fram dem utifrån enbart n.
$$\varphi(n) = (p - 1)(q - 1)$$
4. Välj en offentlig heltalsexponent e inom intervallet $1 < e < \varphi(n)$ och att $\gcd(e, \varphi(n)) = 1$, vilket matematiskt kallas att de är ”relativt prima”.
5. Beräkna den hemliga exponenten d: - Beräkna d som den modulära inversen av e modulo $\varphi(n)$:
$$d \equiv e^{-1} \mod \varphi(n)$$

$$e \times d \equiv 1 \mod \varphi(n)$$

6. Generera nyckelpar: Offentlig nyckel: (n, e). Hemlig nyckel: (n, d).

2.2 Kryptering

1. Konvertera klartext till heltal m där $0 < m < n$.
2. Kryptera meddelandet med offentlig nyckel (n, e) för att beräkna chiffertext c:
$$c \equiv m^e \mod n$$

2.3 Dekryptering

1. Dekryptera chiffertexten c med hemlig nyckel (n, d) för att få m:
$$m \equiv c^d \mod n$$
2. Konvertera heltal m till klartext.

2.4 Dekryptering med offentlig nyckel

Dekryptering av chiffertexten med offentlig nyckel (n, e) genom att rekonstruera den hemliga nyckeln. Detta görs genom att faktorisera n för att hitta p och q, och sedan beräkna d.

1. **Faktorisera modulus n för att hitta p och q:** $n = p \times q$
2. **Beräkna Eulers totientfunktion $\varphi(n)$:** $\varphi(n) = (p - 1)(q - 1)$
3. **Beräkna den hemliga exponenten d:** $e \times d \equiv 1 \mod \varphi(n)$
4. **Dekryptera chiffertexten c med d:** $m \equiv c^d \mod n$
5. **Konvertera heltalet m till klartext.**

2.5 Extrapolering av knäckningstider för större nycklar

Logaritmisk funktion användes för att förutsäga knäckningstiden för större nyckelstorlekar (1024, 2048 och 4096 bitar). Den logaritmiska modellen tar formen:

$$y = a \cdot \log(x) + b$$

Där y representerar tiden att knäcka en RSA-nyckel, x är nyckelstorleken, och a och b är anpassningskoefficienter som beräknas från de observerade data.

3 Diskussion

Den centrala tanken med RSA-kryptering är dess beroende av den matematiska svårigheten att faktorisera stora tal. I denna lösning faktoriseras modulen n i dess primtalsfaktorer p och q, vilket gör det möjligt att beräkna den hemliga nyckeln och använda den för dekryptering.

3.1 Angående storleken på siffrorna i listan

Siffrorna i chiffertextlistan är alla av liknande storlek på grund av hur RSA-kryptering fungerar. RSA arbetar med block av klartext, och varje block omvandlas till ett stort tal med hjälp av den offentliga nyckeln och modulen. Modulen n bestämmer den övre gränsen för storleken på dessa tal. Eftersom klartexten delas upp i block av lika storlek, tenderar även de resulterande chiffertexttalen att vara av liknande storlek.

Detta beteende är förväntat eftersom RSA säkerställer att varje chiffertextblock är mindre än modulen n. Därför kommer chiffertexttalen att vara relativt lika i storlek, begränsade av värdet av n.

3.2 Tidsåtgång vid knäckning av RSA-kryptering

Resultaten från våra tester visar att tiden det tar att knäcka RSA-krypteringen ökar snabbt med nyckelstorleken. För nycklar mellan 100 och 200 bitar tog det från bråkdelar av en sekund till flera minuter(20 minut) att knäcka nyckeln. Medan dessa tider är hanterbara för små nycklar, blir tiden snabbt ohanterlig när nyckelstorleken ökar.

När vi extrapolerade resultaten för nyckelstorlekar på 1024, 2048 och 4096 bitar, blev det tydligt att knäckningstiden skulle vara orealistiskt lång. Detta bekräftar varför moderna RSA-system använder nyckelstorlekar på minst 2048 bitar för att garantera säkerhet. Den logaritmiska modellen som användes för att förutsäga tiderna för större nycklar fungerade bra för att ge en övergripande förståelse, men exakt tidsberäkning för mycket stora nycklar kräver mer avancerade metoder.

RSA-metoden är robust när det gäller att skydda information med stora nycklar, och vårt experiment bekräftar detta. Samtidigt visade experimentet på de svagheter som finns vid användning av små nyckelstorlekar, där moderna datorer kan knäcka dessa på bara några sekunder.

4 Kod

Här nedan följer den fullständiga koden (2 sidor). Den finns också tillgänglig på Github: https://github.com/F7pvqc/Diskret-matte-398476.git

4.1 Task 1

```
def faktorisera_modul(nBob):
    for kandidat in range(2, int(nBob**0.5) + 1):
        if nBob % kandidat == 0:
            return kandidat, nBob // kandidat
    raise ValueError("Inga faktorer hittades")

def utokad_euklides(a, b):
    if a == 0:
        return b, 0, 1
    gcd, x1, y1 = utokad_euklides(b % a, a)
    x = y1 - (b // a) * x1
    y = x1
    return gcd, x, y

def modul_inv(e, phi):
    gcd, x, y = utokad_euklides(e, phi)
    if gcd != 1:
        raise ValueError("Modulär invers existerar inte")
    else:
        return x % phi

def heltal_till_bytes(meddelande):
    byte_langd = (meddelande.bit_length() + 7) // 8
    return meddelande.to_bytes(byte_langd, byteorder='big')

def dekryptera_rsa(chiffertext, nBob, eBob):
    p, q = faktorisera_modul(nBob)
    phi_n = (p - 1) * (q - 1)
    hemlig_nyckel = modul_inv(eBob, phi_n)

    dekrypterade_meddelanden = []
    for chiffer in reversed(chiffertext):
        meddelande = pow(chiffer, hemlig_nyckel, nBob)
        dekrypterade_meddelanden.append(meddelande)

    dekrypterad_text = ''.join(heltal_till_bytes(m).decode('utf-8', errors='ignore')
    for m in dekrypterade_meddelanden)

    dekrypterad_text = dekrypterad_text[::-1]
```

```
    return dekrypterad_text

nBob = 126456119090476383371855906671054993650778797793018127
eBob = 7937

chiffertext = [
    71813256693940924296894077934214561172810879712474411,
    9448822287828090646994864850737396938193829207476291,
    88668970435389288697377439396925326741948237682465270,
    86506877126882849406016686638047102838609248170576618,
    16709897999784737136957685475437549241701090506782283,
    112082150953644879808862406205324790087623126644040573,
    101300870021945928543132671557050279918096489651239300,
    32937734818698596498554567892462717857351635451752837,
    103250795649561696933993996191026658588156558009063626,
    99446883997415524776158640105790361842457834118830 57,
    119023583366882743798043931890543936842945498718476068,
    80592157601474287498990443067778705256100803395677817,
    102380508653117028882619903124827386257126674349361274,
    29811966446563529123471275226007901312118773446042793,
    92762330649448230399110375463713210616117461806861915,
    52785580108219931044518308758110100269607232524031605,
    96630768452430661169900035905564353166088443035700946,
    104675165348205433706999623683285417639543643952502324,
    40951632727878687548912007343839372258522783062745255,
    34396485788419603319314775862549362529268070611841 28,
    92627296356479225180584868594165589614134261562166537,
    17702026915107984931197975326852130481340232863388490,
    35046202376732485019333999169687110582305137751148612,
    77294680692381954105730803435472597801358963832333113,
    58483888921987241464318109604079587034521404720554634,
    36276638436400152414964124035009391520390748123243684,
    51639523466890776909441678913110130114797820309131676,
    88728872239148972759884018820709080618086999507011767,
    45676147252256101340528647372987947783315245082701701,
    23720650117296688653687823869949231140410366974406435,
    116873909796842028543216809278057888647421675552833624,
    48366928605018172969920839968881382332820063246862564,
    35425491594411738404916586785616696655411948001887947,
    40450505118769549506412191479348341185611602935328569,
    107418270783831708663699380219027152916779513788697702,
    101200673359310801145084267798164209444861857835311695,
    65616489296627251359500608540483019164860755372512518,
    11847413450576524199351895472796724862275584777010578,
```

```
    2731217915540071371661447436484606877270200777923464,
    10599418784042349226543806726994624123223235946860821
]

dekrypterat_meddelande = dekryptera_rsa(chiffertext, nBob, eBob)
print(dekrypterat_meddelande)
```

4.2 Task 2

```
import time
import sympy
import numpy as np
import matplotlib.pyplot as plt
from sympy import mod_inverse

# Faktorisera n till två primtal p och q
def faktorisera_n(n):
    faktorer = sympy.factorint(n)
    if len(faktorer) != 2:
        raise ValueError("Inga faktorer hittades eller n är inte produkten av två primtal")
    return tuple(faktorer.keys())

# RSA-avkodningsmetod
def knäck_RSA(chiffer, n, e):
    p, q = faktorisera_n(n)

    phi_n = (p - 1) * (q - 1)

    # Beräkna den privata nyckeln d
    d = mod_inverse(e, phi_n)

    dekrypterat_meddelande = pow(chiffer, d, n)

    return dekrypterat_meddelande

# Testa hur lång tid det tar att knäcka RSA för olika nyckelstorlekar
def tid_att_knäcka_rsa(start_bitstorlek, slut_bitstorlek, meddelande):
    tider = []
    bitstorlekar = list(range(start_bitstorlek, slut_bitstorlek + 1, 10))

    for bits in bitstorlekar:
        # Generera två slumpmässiga primtal p och q av lämplig bitstorlek
        p = sympy.randprime(2**(bits//2 - 1), 2**(bits//2))
        q = sympy.randprime(2**(bits//2 - 1), 2**(bits//2))
        n = p * q
        e = 65537  # Vanlig offentlig exponent
```

```
        # Kryptera meddelandet
        chiffer = pow(meddelande, e, n)

        # Mät tiden det tar att knäcka
        start_tid = time.time()
        knäck_RSA(chiffer, n, e)
        elapsed_tid = time.time() - start_tid

        tider.append(elapsed_tid)
        print(f"Knäckte {bits}-bitars RSA på {elapsed_tid:.4f} sekunder.")

    return bitstorlekar, tider

# Visualisera resultaten
def plot_rsa_knäckningstider(bitstorlekar, tider):
    plt.figure(figsize=(10, 6))
    plt.plot(bitstorlekar, tider, marker='o')
    plt.title('Tid att knäcka RSA vs. Nyckelstorlek')
    plt.xlabel('Nyckelstorlek (bitar)')
    plt.ylabel('Tid (sekunder)')
    plt.grid(True)
    plt.show()

# Meddelandet som ska krypteras och knäckas
meddelande = int.from_bytes(b"GO DOWN DEEP ENOUGH INTO ANYTHING AND YOU WILL FIND MATHEMATIC

# Knäck RSA för nyckelstorlekar från 100 till 200 bitar
bitstorlekar, tider = tid_att_knäcka_rsa(100, 200, meddelande)

# Plotta resultaten
plot_rsa_knäckningstider(bitstorlekar, tider)

# Extrapolera för att förutse knäckningstiden för större nyckelstorlekar
def förutse_tid_for_stora_nycklar(bitstorlekar, tider, stora_nyckelstorlekar):
    log_tider = np.log(tider)
    koeffs = np.polyfit(bitstorlekar, log_tider, 1)
    förutsagda_tider = np.exp(np.polyval(koeffs, stora_nyckelstorlekar))

    plt.figure(figsize=(10, 6))
    plt.plot(bitstorlekar, tider, marker='o', label='Observerade tider')
    plt.plot(stora_nyckelstorlekar, förutsagda_tider, marker='x', linestyle='--', label='För
    plt.title('Förutsagd tid att knäcka RSA för större nyckelstorlekar')
    plt.xlabel('Nyckelstorlek (bitar)')
    plt.ylabel('Tid (sekunder)')
    plt.grid(True)
```

```python
    plt.legend()
    plt.show()

# Förutse tid för 1024, 2048, och 4096-bitars RSA-nycklar
stora_nyckelstorlekar = [1024, 2048, 4096]
förutse_tid_for_stora_nycklar(bitstorlekar, tider, stora_nyckelstorlekar)
```