RMIT UNIVERSITY

Security in Computing and Information Technology
COSC2550 (Semester 2, 2025)
Assignment 1
S4100564

Overview

The objective of Assignment 1 is to evaluate your knowledge of the topics covered in the lectorials and practicals from Weeks 1–4. Topics include cryptographic techniques and public-key cryptography. The task emphasises application of knowledge, critical analysis, and decision-making. Prepare your solutions and upload them as a single PDF to Canvas.

Learning Outcomes

This assessment addresses the following course learning outcomes:

CLO1: Explain the functioning of security services in computing environments and the security issues in networked applications.

CLO2: Discuss and implement various data-integrity and confidentiality mechanisms, including cryptography.

CLO3: Describe basic system-security mechanisms and protocols used in operating systems, file systems, and computer networks.

PASS (20 marks)

Q1. (10 marks) Designing Cryptographic Algorithm for Secure System

This design for a secure file access system uses a **XOR** operation. The system consists of two algorithms: one for generating a one-time Security Access Key (SAK), and another for verifying that key to grant access. The following six-digit integers will be used to demonstrate how these algorithms work:

Note: For consistency in the bitwise operations, all binary numbers are zero-padded to a length of 20 bits.

- Employee ID (EID): **100564** (00011000100011010100) - last 6 digits of my student ID
- File Number (FID): **548438** (10000101111001010110)
- ISM Code: **264327** (01000000100010000111)
- ISD Code: **314976** (01001100111001100000)
- CISO Code: **677056** (10100101010011000000)
- One-Time Pad (OTP): **897670** (11011011001010000110)

Algorithm for SAK Generation

The algorithm generates the key by XORing the six numbers together in a sequential chain; because XOR is associative and commutative, the chained result is well-defined and independent of evaluation order, and any single-bit change in any input flips the corresponding output bit, producing a different SAK.[12].

1. The Employee ID is XORed with the File Number:
   00011000100011010100
   $\oplus$ 10000101111001010110
   = 10011101011010000010 (**644738**)
2. The result is then XORed with the ISM's code:
   10011101011010000010
   $\oplus$ 01000000100010000111
   = 11011101111000000101 (**908805**)
3. This result is XORed with the ISD's code:
   11011101111000000101
   $\oplus$ 01001100111001100000
   = 10010001000001100101 (**594021**)
4. And then the CISO's code:
   10010001000001100101
   $\oplus$ 10100101010011000000
   = 00110100010010100101 (**214181**)
5. Finally, the running total is XORed with the One-Time Pad to produce the key:
   00110100010010100101
   $\oplus$ 11011011001010000110
   = 11101111011000100011 (**980515**)

The final **One-Time Security Access Key (SAK)** generated by the system is **980515**. This is the key that is sent to the employee.

Algorithm for Verification

When the employee uses their SAK (**980515**), the system must verify it by re-generating the original One-Time Pad; this works because XOR is self-inverse, so XORing the SAK with the XOR of the other inputs cancels them out and recovers the stored pad exactly if and only if the SAK was computed from those same inputs.

1. The system first calculates the XOR sum of the five known values (EID, FID, and the three Oracle codes). This value was already calculated in Step 4 of the generation process:
   `...  = 00110100010010100101` (**214181**)

2. The system then takes that result and XORs it with the SAK submitted by the employee:
   `00110100010010100101`
   $\oplus$ `11101111011000100011`
   `= 11011011001010000110` (**897670**)

The result, **897670**, is a perfect match for the original One-Time Pad stored by the system. Because the values match, access would be granted.

Q2. (10 marks) Cryptanalysis with Missing Encrypted Text

KBJQHQGQKF QZJVFFQLVZGV QX JBKZXHPBDQZL VSVBC QZWYXJBC KZW BKQXQZL UBPHPYZW VJMQGKF IYVXJQPZX. PBLKZQXKJQPZX DYXJ RKFKZGV QZZPSKJQPZ NQJM BVXUPZXQRQFQJC, VZXYBQZL JBKZXUKBVZGC, HKQBZVXX, KZW KGG-PYZJKRQFQJC QZ KFLPBQJMDX JMKJ XMKUV MYDKZ VOUVBQVZGVX KZW WVGQXQPZX. KX BDQJ XJYWVZJX QJ QX CPYB BVXUPZXQRQFQJC JP UBPGVVW NQJM GKYJQPZ

E	T	A	O	I	N	S	H	R	D	L	C	U	M	W	F	G	Y	P	B	V	K	J	X	Q	Z
12.7	9.1	8.2	7.5	7.0	6.7	6.3	6.1	6.0	4.3	4.0	2.8	2.8	2.4	2.4	2.2	2.0	2.0	1.9	1.5	1.0	0.8	0.15	0.15	0.10	0.07

Figure 1: English Alphabet Frequency Count

Q	Z	X	J	K	V	B	P	G	Y	F	W	C	M	U	L	D	H	R	N	S	I	O	A	E	T
37	31	26	25	24	22	19	17	11	10	9	8	7	7	7	6	5	4	4	2	2	1	1	0	0	0
13.0	10.9	9.1	8.8	8.4	7.7	6.7	6.0	3.9	3.5	3.2	2.8	2.5	2.5	2.5	2.1	1.8	1.4	1.4	0.7	0.7	0.4	0.4	0.0	0.0	0.0

Figure 2: Ciphertext Frequency Count [5]

Step 1: Initial Frequency-Based Guesses
The most common letters in the new cipher text are “Q” (13.0%), “Z” (10.9%), and “X” (9.1%). Because Q is the start of many two letter words in the cipher text, which may be “is” or “in”, we’ll guess that Q is i. To match this, we’ll let Z be n. Also, X is the final letter in many words in the cipher text, so we’ll guess that X is s.

Starting substitutions:

- Q → i
- Z → n
- X → s

Applying these to the ciphertext fragment, with solved letters in lowercase:

KBJiHiGiKF inJVFFiLVnGV is JBKnsHPBDinL VSVBC inWYsJBC KnW BKisinL UBPHPYnW VJMiGKF IYVsJiPns. PBLKnisKJiPns DYsJ RKFKnGV innPSKJiPn NiJM BVsUPnsiRiFiJC, Vn-sYBinL JBKnsUKBVnGC, HKiBnVss, KnW KGGPYnJKRiFiJC in KFLPBiJMDs JMKJ sMKUV MYDKn VOUVBiVnGVs KnW WVGisiPns. Ks BDiJ sJYWVnJs iJ is CPYB BVsUPnsiRiFiJC JP UBPGVVW NiJM GKYJiPn

Step 2: Identifying Common Words
Now, the second word in the cipher is 11 letters long, begins with “in” and the 5th and 6th letter are the same. A good guess for this word would be “intelligence” giving:

- J→ t
- V → e
- F → l
- L → g
- G → c

Applying these new substitutions yields:

> KBtiHiciKl intelligence is tBKnsHPBDing eSeBC inWYstBC KnW BKising UBPHPYnW etMicKl IYestiPns. PBgKnisKtiPns DYst RKlKnce innPSKtiPn NitM BesUPnsiRilitC, ensYBing tBKnsUK-BencC, HKiBness, KnW KccPYntKRilitC in KlgPBitMDs tMKt sMKUe MYDKn eOUeBiences KnW WecisiPns. Ks BDit stYWents it is CPYB BesUPnsiRilitC tP UBPceeW NitM cKYtiPn

Step 3: Contextual Substitutions
Now, certain words become obvious, such as "stYWents" representing "students" giving Y → u and W → d. The phrase "BesUPnsiRilitC" appears twice, and based off the context, we can guess that this is "responsibility", giving:

- B→ r
- U → p
- P → o
- R → b
- C → y

Updated ciphertext:

> KrtiHiciKl intelligence is trKnsHorDing eSery industry Knd rKising proHound etMicKl Iuestions. orgKnisKtions Dust RKlKnce innoSKtion NitM responsiRility, ensuring trKnspKrency, HKirness, Knd KccountKRility in KlgoritMDs tMKt sMKpe MuDKn eOperiences Knd decisions. Ks rDit students it is your responsiRility to proceed NitM cKution

Step 4: Decrypted Ciphertext
At this point, the text is almost fully decrypted. The remaining letters can be found by filling in the gaps of recognizable words.

- The partially decrypted word "KrtiHiciKl" becomes "artificial", giving K → a and H → f
- Next, the phrase "etMical Iuestions" becomes "ethical questions" becomes "artificial", giving M → h and I → q
- "Dust Ralance innoSation Nith" becomes "must balance innovation with", giving D → m, R → b, S → v and N → w
- The final encrypted word remaining is "eOperiences", so naturally, O → x

After these changes, the text is fully deciphered:

> artificial intelligence is transforming every industry and raising profound ethical questions. organisations must balance innovation with responsibility, ensuring transparency, fairness, and accountability in algorithms that shape human experiences and decisions. as rmit students it is your responsibility to proceed with caution

Q	Z	X	J	K	V	B	P	G	Y	F	W	C	M	U	L	D	H	R	N	S	I	O	A	E	T
37	31	26	25	24	22	19	17	11	10	9	8	7	7	7	6	5	4	4	2	2	1	1	0	0	0
13.0	10.9	9.1	8.8	8.4	7.7	6.7	6.0	3.9	3.5	3.2	2.8	2.5	2.5	2.5	2.1	1.8	1.4	1.4	0.7	0.7	0.4	0.4	0.0	0.0	0.0
i	n	s	t	a	e	r	o	c	u	l	d	y	h	p	g	m	f	b	w	v	q	x			

Figure 3: Letter Pairings [5]

CREDIT (4 marks)

Q3. (4 marks) Breaking the RSA Key

a) To break the RSA encryption and find the private key, we must first find the two prime numbers, p and q, that were used to generate the public key parameter n. The security of RSA relies on the difficulty of factoring this large number n. Given the public key parameters $n = 10009999019$ and $e = 65537$, the first step is to perform prime factorization on n.

Step 1: Find Prime Numbers p and q

Since n is a very large number, it is not practical to factor it manually. I will use an online prime factorization tool to find the two prime factors. The process involves testing for divisibility against a list of prime numbers until the factors are found.

Using a python script I wrote, the prime factors of $n = 10009999019$ are found to be: - $p = 99991$ - $q = 100109$

```python
import math

# Function to generate the first ~10,000 primes using Sieve of Eratosthenes
def generate_primes(limit=105000):
    sieve = [True] * (limit + 1)
    sieve[0] = sieve[1] = False
    for i in range(2, int(math.sqrt(limit)) + 1):
        if sieve[i]:
            for j in range(i * i, limit + 1, i):
                sieve[j] = False
    return [i for i, prime in enumerate(sieve) if prime]

# Generate the list of primes
primes = generate_primes()

# The number n from the problem
n = 10009999019

# Perform trial division to find factors
found = False
for p in primes:
    if p * p > n:  # No need to check beyond sqrt(n)
        break
    if n % p == 0:
        q = n // p
        print(f"Factors: p={p}, q={q}")
        found = True
        break

if not found:
    print("No factors found within the prime list.")
```

Figure 4: Python Prime Factorization

We can verify this by multiplying them:

$$99991 \times 100109 = 10009999019$$

which matches the given value of n.

Step 2: Compute the Private Key d

With the prime numbers p and q identified, we can now calculate the private key, d. This involves two parts as described in the lecture materials [6].

First, we calculate r using the formula

$$r = (p-1)(q-1)$$

$$r = (99991-1) \times (100109-1)$$

$$r = 99990 \times 100108$$

$$r = 10009798920$$

Next, we need to find the private key d, which is the multiplicative inverse of e modulo r. The relationship is defined as:

$$d \equiv e^{-1} \pmod{r}$$

Substituting the known values:

$$d \equiv 65537^{-1} \pmod{10009798920}$$

$$d = 6509721833$$

Using Wolfram Alpha [11]:

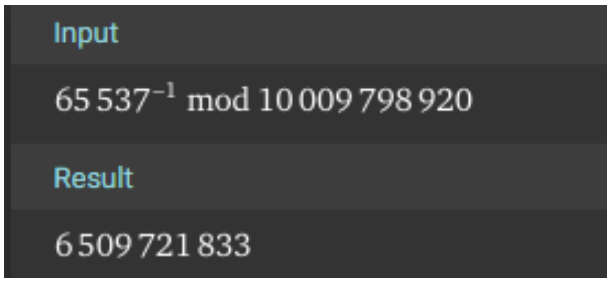

Figure 5: Calculation of secret key d

We can also confirm that our secret key, d, is correct:

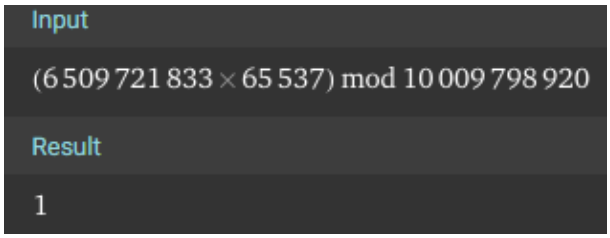

Figure 6: Calculation of secret key d

It is confirmed that $(d \times e) \bmod r = 1$. This verification demonstrates that the computed private key d is indeed the correct modular inverse of e modulo r, ensuring the functionality and security of RSA decryption as expected.

By successfully factoring n and calculating d, we have broken the RSA key. With the private key ($d = 6509721833$) and the public parameter ($n = 10009999019$), it is now possible to decrypt any secret message 'M' that was encrypted with the corresponding public key [6].

b) To compute the plaintext secret message M from the captured ciphertext $C = 1880434495$, we use the RSA decryption [6]. With the private key $d = 6509721833$ and modulus $n = 10009999019$, the decryption is performed as follows:

$$M = C^d \bmod n$$

Substituting the values:

$$M = 1880434495^{6509721833} \bmod 10009999019$$

Using Wolfram Alpha [11] again:

Figure 7: Calculation of M

The result is:

$$M = 12345$$

This recovers the original secret message M.

c) To verify that the decrypted plaintext secret message $M = 12345$ is correct, we re-encrypt it using the public key ($n = 10009999019, e = 65537$) and check if it matches the original ciphertext $C = 1880434495$. The encryption formula is:

$$C_{\text{check}} = M^e \bmod n$$

Substituting the values:

$$C_{\text{check}} = 12345^{65537} \bmod 10009999019 = 1880434495$$

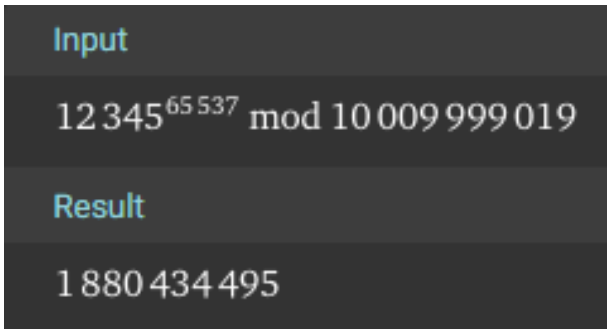

Figure 8: Verifying M

Since C_{check} equals the original C, the plaintext M is confirmed to be correct.

d) In principle, using 10 computers would accelerate the process of finding the private key. The main computational challenge is factoring the modulus n through trial division—a task that is easily parallelized. We could split the list of primes (up to $\sqrt{n} \approx 100050$) into 10 smaller chunks, with each computer searching its assigned portion. When one machine finds a factor, like $p = 99991$, the search is complete. This divide-and-conquer approach could theoretically cut the factoring time by a factor of ten, although some minor overhead for coordinating the machines is expected. Since factoring n is the most time-consuming part, this parallel effort is where the real speedup would occur [1].

However, in this particular case, the practical benefit would be minimal. Factoring n by testing just 10,000 primes is already incredibly fast on a single modern computer, likely finishing in under a second. The overhead costs of distributing the task, communicating between machines, and synchronizing the results would likely outweigh the small computational gain. For such a trivial task, a parallel approach isn't worth the effort and might even be slower. This illustrates a key principle described by Amdahl's Law: the speedup from parallelization is limited by the parts of the task that cannot be parallelized, which in this case includes the setup and coordination overhead [2][3].

DISTINCTION (4 marks)

Q4. (4 marks) Designing Secure Online Bidding System for an Online Freelancing Platform using Hash Algorithm

a) **Mechanism Overview**

The proposed blind bidding mechanism for the "GetExarts.com" platform uses the SHA-256 cryptographic hash function to ensure the authenticity of bids. The core principle is that freelancers submit a commitment to their bid (the hash) without revealing the actual bid amount. This prevents other bidders from gaining an advantage and ensures a fair selection process based on the lowest price.

The bidding process is conducted in the following phases:

i. **Bidding Phase:** Each freelancer determines their bid amount. Instead of submitting the actual value (e.g., $250), they compute the SHA-256 hash of the value. This hash is submitted to the platform.

ii. **Bidding Period Ends:** Once the deadline for submissions passes, no more bids are accepted. The platform has a list of hash commitments from all participating freelancers.

iii. **Reveal Phase:** The system prompts all bidders to submit their cleartext bid amounts.

iv. **Winner Determination:** For each freelancer, the platform computes the SHA-256 hash of the revealed bid amount and compares it to the hash submitted during the Bidding Phase.

- If the hashes match, the bid is considered valid.
- If they do not match, the bid is disqualified, as this indicates tampering or a false reveal.

Among all valid bids, the freelancer with the lowest numerical bid amount is declared the winner. In the event of a tie, the profiles of the tied freelancers are presented to the client, who then makes the final selection based on qualifications and work history. This allows the client to choose the best candidate for the project.

Example

Let's consider a scenario where three freelancers are bidding for a project.

1. Bidding Phase The freelancers decide on their bids and compute the corresponding SHA-256 hashes. They submit only the hash values to "GetExarts.com".

Freelancer	Bid Amount	Submitted SHA-256 Hash
Freelancer A	$250	`1e472b...f67b`
Freelancer B	$300	`983bd6...58389`
Freelancer C	$200	`27badc...addf`

Full hash values for reference:

- **Hash of "250":** `1e472b39b105d349bcd069c4a711b44a2fffb8e274714bb07ecfff69a9a7f67b`
- **Hash of "300":** `983bd614bb5afece5ab3b6023f71147cd7b6bc2314f9d27af7422541c6558389`
- **Hash of "200":** `27badc983df1780b60c2b3fa9d3a19a00e46aac798451f0febdca52920faaddf`

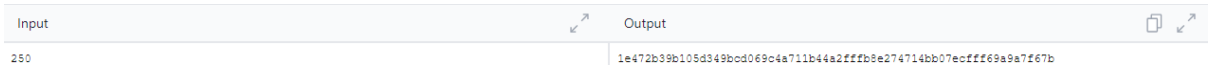

Figure 9: Hash of 250 [4]

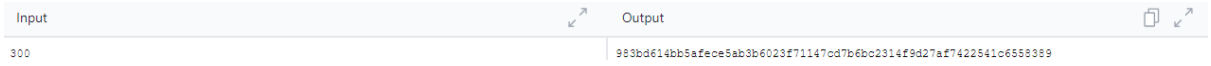

Figure 10: Hash of 300 [4]

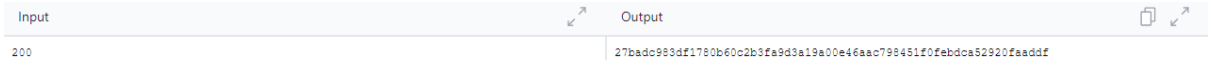

Figure 11: Hash of 200 [4]

2. Verification Phase After the bidding period closes, the freelancers submit their clear-text bids: \$250, \$300, and \$200. The system verifies each bid by hashing these amounts and confirming they match the stored hashes.

3. Winner Determination All three bids are verified successfully. The system compares the valid numerical bids: \$250, \$300, and \$200. Freelancer C, with the lowest bid of **\$200**, is selected as the winner. If another freelancer had also bid \$200, the client would be prompted to choose between them.

b) **Retrieving Plaintext Bids from Hashes**

A smart bidder can't directly retrieve the plaintext bid amounts from the hashed values. The security of the system relies on the *preimage resistance* of the SHA-256 algorithm [9]. This property means that given a hash output, it is computationally infeasible to reverse the function to find the original input.

However, a smart bidder can attempt to cheat the system using brute force or a dictionary attack [8]. This method does not reverse the hash (which is impossible) but instead relies on guessing the original bid. Since bid amounts are often predictable (e.g., round numbers, specific price points), an attacker can pre-compute the hashes for a range of likely bids and compare them against the hashes submitted to the platform.

Example of a Dictionary Attack

Let's assume the following three hashes have been submitted to "GetExarts.com" for a project:

- **Hash 1:** `9ae2bdd7beedc2e766c6b76585530e16925115707dc7a06ab5ee4aa2776b2c7b`
- **Hash 2:** `deeeb5df3f2cee6bf4e597a8a3a878a6ce49b932b9e90b416922d4499f54fae6`
- **Hash 3:** `0604cd3138feed202ef293e062da2f4720f77a05d25ee036a7a01c9cfcdd1f0a`

A bidder, let's call him Bob, wants to discover the bid amounts of his competitors. He assumes the bids are likely between \$100 and \$500.

1. Attack Preparation Bob creates a "dictionary" by generating the SHA-256 hashes for every possible bid amount in his suspected range. For simplicity, let's assume he checks every $50 increment.

Bob's Guessed Bid	Pre-computed SHA-256 Hash
$100	`ad5736...d306`
$150	`9ae2bd...2c7b`
$200	`27badc...addf`
$250	`1e472b...f67b`
$300	`983bd6...8389`
$350	`deeeb5...fae6`
$400	`e36928...469a`
$450	`d15197...6f1f`
$500	`0604cd...1f0a`

2. Comparison Bob compares his pre-computed list of hashes with the hashes submitted to the platform.

- He finds that his hash for $150 matches **Hash 1**.
- He finds that his hash for $350 matches **Hash 2**.
- He finds that his hash for $500 matches **Hash 3**.

3. Result By using this dictionary attack, Bob has successfully discovered the exact bid amounts of all his competitors. He did not break the SHA-256 algorithm; he simply guessed the inputs correctly and verified his guesses. This vulnerability exists because the set of possible inputs (the bid amounts) is relatively small and predictable. If a bidder were to use an unpredictable bid (e.g., $251.37), it would be much harder for an attacker to guess.

c) **Preventing Cheating in the Designed System**

The designed system can be cheated using a dictionary attack, as demonstrated in part (b). The main vulnerability is that the bid amount is predictable, allowing an attacker to pre-compute hashes of likely bids and match them to the bids submitted to the platform.

A robust and effective method to prevent this is to introduce a *cryptographic nonce* into the hashing process [7]. This technique, commonly referred to as *salting*, involves adding a unique, random value to each input before it is hashed [10]. By combining the bid with a secret nonce, the input to the hash function becomes unpredictable, rendering dictionary attacks computationally infeasible.

Improved Bidding Mechanism with a Salted Hash

This enhanced mechanism requires each bidder to generate and manage a secret random value for each bid.

i. **Bidding Phase:** Each freelancer generates a random nonce (e.g., a long, unpredictable string like `R4nDoM5aLt!123`). They then concatenate their bid amount with this nonce (e.g., `"250|R4nDoM5aLt!123"`) to create a unique input string. The SHA-256 hash of this

combined string is what they submit to the platform. The nonce must be kept secret until the reveal phase.

ii. **Reveal Phase:** After the bidding period ends, each freelancer submits both their original cleartext bid and the secret nonce they used to generate their commitment hash.

iii. **Winner Determination:** The platform verifies each submission by concatenating the revealed bid and nonce in the same manner and re-computing the SHA-256 hash. If it matches the hash submitted during the bidding phase, the bid is considered valid. The lowest valid bid wins.

Example: How Salting Protects Against an Attacker

Let's assume an attacker, Bob, is attempting a dictionary attack.

1. Submission

- Freelancer A's Bid: $250
- Freelancer A's Secret Nonce: `aBcDeF12345`
- **Value to be Hashed:** `"250|aBcDeF12345"`
- **Submitted Hash:** The resulting SHA-256 hash of the concatenated string = `14c77276741d7043cf69f7c96f1c6983df5a8fcc38bcd79df5af646826b596b0`

2. How the Attack Fails Bob sees the submitted hash, but his dictionary attack is now impossible.

- To test if the bid was $250, Bob would need to compute the hash of `"250|nonce"`. However, he does not know Freelancer A's secret nonce (aBcDeF12345).
- Guessing the nonce is infeasible if it is sufficiently long and random. For every single bid guess (e.g., $250), he would have to try billions or trillions of possible nonces, a computationally prohibitive task.

By adding a secret, random nonce to each bid before hashing, the input space for the hash function becomes enormous. This effectively neutralizes dictionary attacks and secures the integrity of the blind bidding process.

HIGH DISTINCTION (7 marks)

Q5. (7 marks) Application of Public-Key Cryptography

a) RSA Public-Key Cryptography Algorithm

The login system uses the RSA algorithm for encryption and decryption. I will use my student ID M = 410056.

i. Web Application Key Generation

The server generates its public and private keys using the given prime numbers, p = 3217 and q = 3041.

- **Step 1: Calculate the modulus (n)**
  The modulus 'n' is the product of the two primes 'p' and 'q'. This value forms part of both the public and private keys.

$$n = p \times q = 3217 \times 3041 = 9,782,897$$

- **Step 2: Calculate Euler's Totient Function ($\phi(n)$)**
  The totient '$\phi(n)$' represents the number of positive integers up to 'n' that are relatively prime to 'n'. For two primes 'p' and 'q', it is calculated as:

$$\phi(n) = (p-1) \times (q-1) = (3216) \times (3040) = 9,776,640$$

- **Step 3: Validate the Public Key Exponent (e)**
  The public exponent 'e' is given as 773. A valid choice for 'e' must satisfy two conditions: $1 < e < \phi(n)$ and 'e' must be coprime to '$\phi(n)$' (i.e., $\gcd(e, \phi(n)) = 1$). We can verify this:

$$\gcd(773, 9776640) = 1$$

  Since 773 is a prime number and does not divide 9776640, the condition holds. This makes 'e=773' a valid choice.

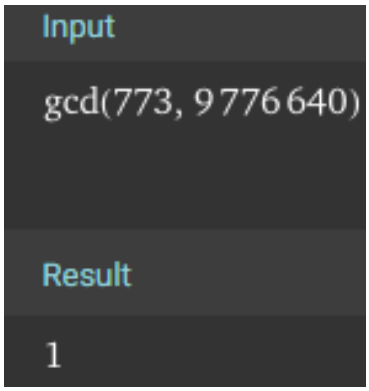

Figure 12: Calculating Public Key Exponent e

- **Step 4: Calculate the Private Key Exponent (d)**
  Set $r = \phi(n) = 9{,}776{,}640$. Then compute the modular inverse:

$$d = e^{-1} \bmod r$$

$$= 773^{-1} \bmod 9{,}776{,}640 = 5{,}147{,}597$$

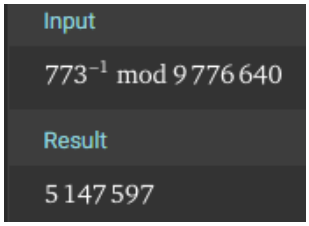

Figure 13: Calculating Private Key Exponent d

- **Step 5: Define the Public and Private Keys** The keys are now defined as follows:
  - Web Application's **Public Key**: $= \{e, n\} = \{773, 9782897\}$
  - Web Application's **Private Key**: $= \{d\} = \{5147597\}$

ii. **Encryption of Account ID (M) to produce C**

The web application encrypts the user's account ID M = 410056 using its public key to generate the secret text C. The message M must be an integer less than n, which is true here ($410056 < 9782897$).

$$C = M^e \pmod{n}$$
$$C = 410056^{773} \pmod{9782897} = 5,944,003$$

The secret text C = 5944003 is sent to the user's mobile phone via SMS.

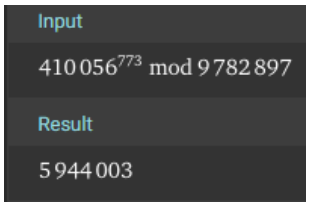

Figure 14: Calculating the Secret Text c

iii. **Decryption of C to produce M'**

At login, the user provides their account ID (M = 410056) and the secret text (C = 5944003). The web application uses its private key to decrypt C and produce M'.

$$M' = C^d \pmod{n}$$
$$M' = 5944003^{5147597} \pmod{9782897} = 410,056$$

The system then verifies that M' equals the user-provided M (410056 = 410056). Since they match, the user is successfully logged in.

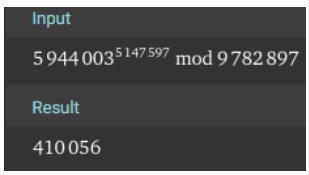

Figure 15: Producing M'

b) **ElGamal Public-Key Cryptography Algorithm**

The login system uses the ElGamal algorithm for encryption and decryption. Again, I will use my student ID M = 410056.

i. **Web Application Key Generation**

- **Step 1: Validating the Parameters** We verify the required conditions for ElGamal over $\mathbb{Z}_p^*$:
  - *Prime modulus:* $p = 8{,}902{,}967$ is prime.
  - *Generator in the group:* $\gcd(g, p) = \gcd(3707, 8{,}902{,}967) = 1$, hence $g \in \mathbb{Z}_p^*$.
  - *Secret exponent range:* $g \leq x \leq p - 2$); here $3707 \leq 7841 \leq 8{,}902{,}965$.
- **Step 2: Calculate the Public Key (y)**
  The public key component y is computed from g and x as:

$$y = g^x \bmod p = 3707^{7841} \bmod 8{,}902{,}967 = 795{,}883$$

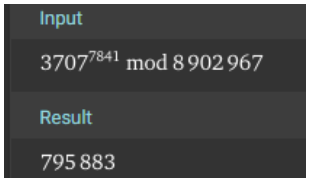

Figure 16: Calculating the Public Key

- **Step 3: Define the Public and Private Keys**
  - Web Application's **Public Key**: $(p, g, y) = (8{,}902{,}967, 3707, 795{,}883)$
  - Web Application's **Private Key**: $x = 7841$

ii. **Encryption of Account ID (M) to produce C**

The web application encrypts the user's account ID $M = 410{,}056$ using its public key and the given random $r = 16{,}889$.

$$C_1 = g^r \bmod p = 3707^{16{,}889} \bmod 8{,}902{,}967 = 3{,}079{,}929$$

$$k = y^r \bmod p = 795{,}883^{16{,}889} \bmod 8{,}902{,}967 = 2{,}508{,}500$$

$$C_2 = (M \cdot k) \bmod p = (410{,}056 \cdot 2{,}508{,}500) \bmod 8{,}902{,}967 = 3{,}377{,}721$$

The secret text is the pair $C = (C_1, C_2) = (3{,}079{,}929,\ 3{,}377{,}721)$.

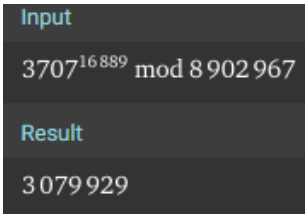

Figure 17: Calculating C_1

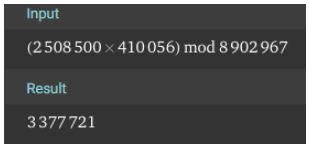

Figure 18: Calculating C_2

iii. **Decryption of C to Produce M'**

At login, the user provides M and the secret text C. The server decrypts C using its private key to recover M'.

$$k' = C_1^x \bmod p = 3{,}079{,}929^{7841} \bmod 8{,}902{,}967 = 2{,}508{,}500$$

$$(k')^{-1} \bmod p = 1{,}654{,}589$$

$$M' = (C_2 \cdot (k')^{-1}) \bmod p = (3{,}377{,}721 \cdot 1{,}654{,}589) \bmod 8{,}902{,}967 = 410{,}056$$

The system verifies that $M' = M$ (i.e., $410{,}056 = 410{,}056$), and the user is logged in successfully.

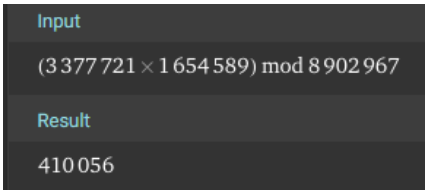

Figure 19: Decrypting C to Recover M'

References

[1] Gene Amdahl. *Validity of the single-processor approach to achieving large scale computing capabilities*. AFIPS Conference Proceedings. Presented at the AFIPS Spring Joint Computer Conference, 1967 [Accessed: 6 August 2025]. 1967.

[2] *Amdahl's law - Wikipedia*. [Accessed: 6 August 2025]. URL: `https://en.wikipedia.org/wiki/Amdahl's_law`.

[3] *Amdahl's Law: Understanding the Basics*. [Accessed: 6 August 2025]. URL: `https://www.splunk.com/en_us/blog/learn/amdahls-law.html`.

[4] Yi-Cyuan Chen. *SHA256 - Online Tools*. Online tool. Available at: `https://emn178.github.io/online-tools/sha256.html` [Accessed: 10 August 2025]. 2022.

[5] Interactive Maths. *Frequency Analysis: Breaking the Code*. Online resource. [Accessed: 31 July 2025]. 2023. URL: `https://crypto.interactive-maths.com/frequency-analysis-breaking-the-code.html`.

[6] RMIT University. *Lecture 2: More concepts on cryptography*. PDF file. Security in Computing & Information Technology (COSC2536) lecture notes. [Accessed: 6 August 2025]. 2025.

[7] Wikipedia contributors. *Cryptographic nonce — Wikipedia, The Free Encyclopedia*. Available at: `https://en.wikipedia.org/w/index.php?title=Cryptographic_nonce&oldid=1238421257` [Accessed: 10 August 2025]. 2024.

[8] Wikipedia contributors. *Dictionary attack — Wikipedia, The Free Encyclopedia*. Available at: `https://en.wikipedia.org/w/index.php?title=Dictionary_attack&oldid=1232810978` [Accessed: 10 August 2025]. 2024.

[9] Wikipedia contributors. *Preimage attack — Wikipedia, The Free Encyclopedia*. Available at: `https://en.wikipedia.org/w/index.php?title=Preimage_attack&oldid=1179477583` [Accessed: 10 August 2025]. 2023.

[10] Wikipedia contributors. *Salt (cryptography) — Wikipedia, The Free Encyclopedia*. Available at: `https://en.wikipedia.org/w/index.php?title=Salt_(cryptography)&oldid=1236357577` [Accessed: 10 August 2025]. 2024.

[11] *Wolfram—Alpha: Making the world's knowledge computable*. Online computational tool. [Accessed: 6 August 2025].

[12] XOR.pw. *XOR Calculator Online*. Online tool. Available at: `https://xor.pw/` [Accessed: 4 August 2025]. 2025.