Scuola Politecnica e
delle Scienze di Base
Università degli Studi di Napoli Federico II

Elaborato di Network Security

Chosen Ciphertext Attack

Simulazione di uno Chosen Ciphertext Attack in uno scenario vulnerabile con Crittografia Textbook RSA

Anno Accademico: 2022/2023

Prof.:
Simon Pietro Romano

Studente:
Serafino Boccia
M63/952

Sommario

1. Introduzione

Lo scopo di questo elaborato è simulare uno scenario vulnerabile ad uno Chosen Ciphertext Attack. In tale scenario interagiscono tre attori: il Server, il Client legittimo e il Client malizioso. Queste entità si scambiano messaggi utilizzando il sistema crittografico RSA nella sua implementazione di base conosciuta come Textbook RSA, Naive RSA, o Naked RSA. [1]

1.1 Crittografia RSA

Il sistema di crittografia RSA è un algoritmo di crittografia asimmetrica, inventato nel 1977 da Ronald Rivest, Adi Shamir e Leonard Adleman. La crittografia RSA si basa sull'esistenza di due chiavi distinte, che vengono usate per cifrare e decifrare. Se la prima chiave viene usata per la cifratura, la seconda deve necessariamente essere utilizzata per la decifratura e viceversa. La questione fondamentale è che, nonostante le due chiavi siano fra loro dipendenti, non è possibile risalire dall'una all'altra, in modo che se anche si è a conoscenza di una delle due chiavi, non si possa risalire all'altra, garantendo in questo modo l'integrità della crittografia.

Per ottenere una discreta sicurezza è necessario utilizzare chiavi binarie di almeno 2048 bit. RSA è basato sull'elevata complessità computazionale della fattorizzazione in numeri primi.

Il suo funzionamento base è il seguente:

1. si scelgono a caso due numeri primi, **p** e **q** abbastanza grandi da garantire la sicurezza dell'algoritmo.
2. si calcola il loro prodotto **n = pq**, e il prodotto **φ(n) = (p-1)(q-1)**, dove **φ(n)** è la funzione di Eulero.
3. si considera che la fattorizzazione di **n** è segreta e solo chi sceglie i due numeri primi, **p** e **q**, la conosce.
4. si sceglie poi un numero **e** (chiamato esponente pubblico), coprimo e più piccolo di **φ(n)**.
5. si calcola il numero **d** (chiamato esponente privato) tale che il suo prodotto con **e** sia congruo a **1** modulo **φ(n)** ovvero che **ed ≡ 1 mod φ(n)**; per calcolare **d** si utilizza l'Algoritmo esteso di Euclide.

La Chiave Pubblica è rappresentata dalla coppia **(e,n)**, la Chiave Privata dalla coppia **(d,n)**.

La forza dell'algoritmo sta nel fatto che per calcolare **d** da **e** (o viceversa) non basta la conoscenza di **n** ma serve anche il numero **φ(n) = (p-1) (q-1)**; il calcolo di questo valore richiede tempi molto elevati, infatti, fattorizzare in numeri primi (cioè scomporre un numero nei suoi divisori primi) è un'operazione computazionalmente costosa.

1.2 Textbook RSA

L'implementazione più elementare del sistema di crittografia RSA è conosciuta come Textbook RSA, Naive RSA o Naked RSA. In questa implementazione, il processo di cifratura e decifratura viene eseguito direttamente utilizzando l'esponenziazione modulare, senza l'aggiunta di schemi di padding. [1]

Cifratura con Chiave Pubblica (e,n):

$C = Encrypt(M) = M^e \bmod n$

Decifratura con Chiave Privata (d,n):

$M = Decrypt(C) = C^d \bmod n$

Le tecniche di padding permettono di introdurre un livello di complessità e casualità al processo di crittografia, rendendo l'algoritmo più resistente ad attacchi e vulnerabilità. Tale operazione assicura che il messaggio cifrato non riveli informazioni indesiderate sulla lunghezza o sulla struttura del messaggio originale, contribuendo a proteggere la riservatezza e l'integrità dei dati scambiati. [2]

2. Chosen Ciphertext Attack

Un attacco con testo cifrato scelto (Chosen Ciphertext Attack, CCA) è un tipo di attacco di natura matematica che mira a ottenere il testo in chiaro manipolando i dati cifrati intercettati in una comunicazione e osservando le risposte del sistema. L'attaccante crea testi cifrati arbitrari a partire dal messaggio intercettato e li invia alla parte responsabile della decifratura RSA. In questo modo, l'attaccante ha la possibilità di analizzare come il sistema risponde in base agli input da lui forniti. L'attaccante può utilizzare le informazioni ottenute dalle risposte del sistema per dedurre informazioni sulla Chiave Privata del destinatario o ancora più semplicemente sui dati originali. Questo tipo di attacco è basato sull'assunzione teorica che l'attaccante abbia accesso a un dispositivo di decifratura che restituisca la decifratura completa per un testo cifrato scelto. [3] [4]

2.1 Scenario

In questo elaborato si presenta come caso di studio uno scenario in cui sono presenti tre entità: un Server, un Client legittimo e un Client malizioso. L'obiettivo del Client malizioso è quello di risalire al testo in chiaro eseguendo operazioni matematiche su un messaggio cifrato, precedentemente intercettato, inviato da un client lecito.

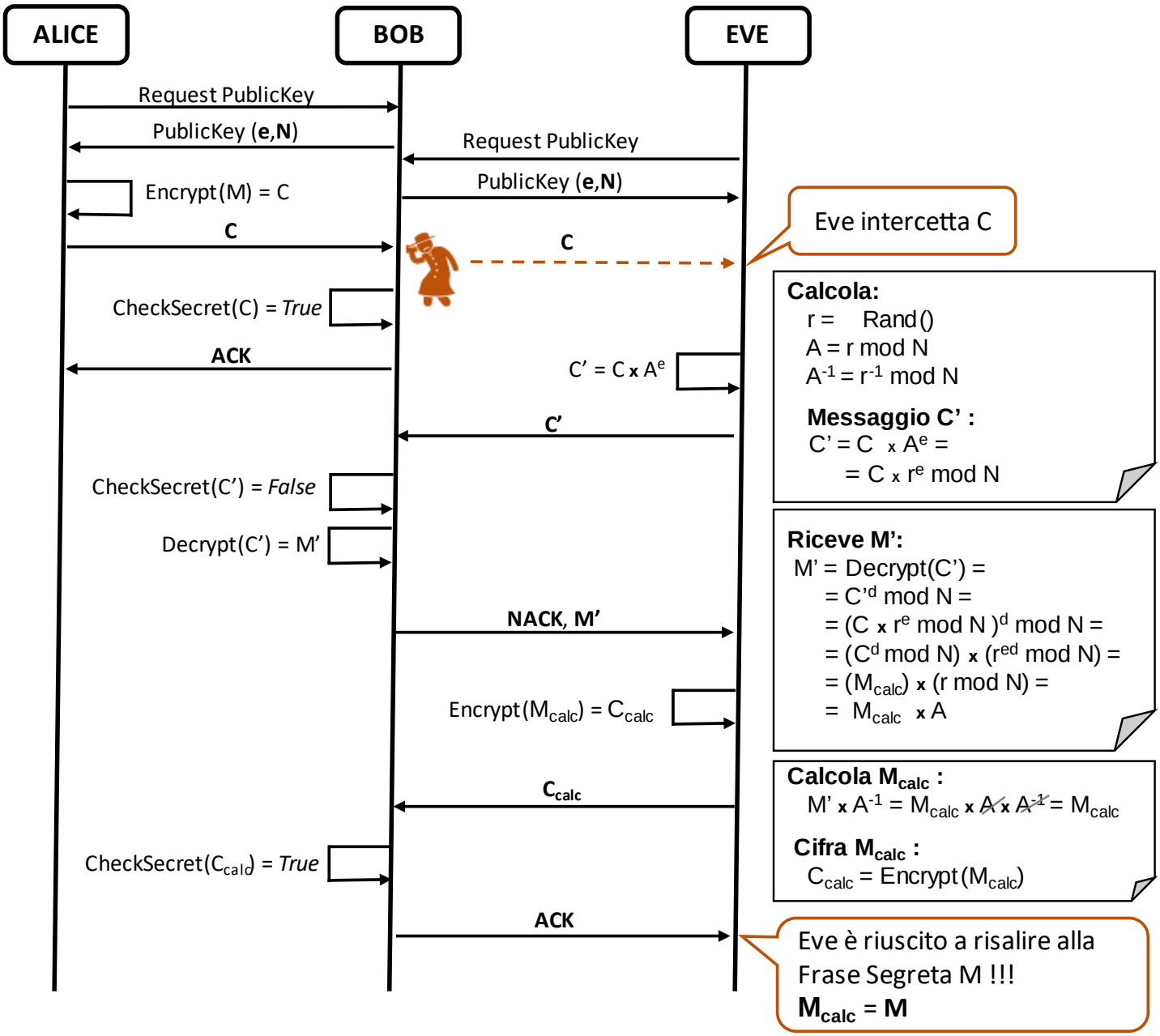
In particolare:

Il **Server (Bob)** offre servizi che permettono di condividere la propria Chiave Pubblica e di trasmettere e verificare una "Frase Segreta" che i Client legittimi conoscono. In particolare, il Server risponde con un messaggio di ACK se la Frase Segreta che ha ricevuto è corretta, altrimenti risponde con NACK e il messaggio decifrato.

Il **Client legittimo (Alice)** riceve la Chiave Pubblica del Server e la utilizza per cifrare la propria Frase Segreta, successivamente la invia al Server per verificarne la correttezza.

Il **Client malizioso (Eve)** tenta di risalire al messaggio originario di Alice intercettando il messaggio cifrato e sfruttando il comportamento del Server. In particolare, Eve modifica il messaggio intercettato con una operazione matematica e lo invia al Server al solo fine di studiarne la risposta contenente il messaggio decifrato. Successivamente, Eve è in grado di risalire al messaggio originario

applicando un'altra operazione matematica che concatenata all'operazione di decifratura del Server produce il messaggio originario di Alice.



2.2 Fondamenti matematici

In questa sezione vengono illustrati i passaggi matematici che permettono di eseguire lo Chosen Ciphertext Attack così come descritto in precedenza.

1) Eve individua un numero intero random minore di N e calcola **A**, $\mathbf{A^{-1}}$ e $\mathbf{A^{e}}$.

$$r = \text{Random}(2, N-1)$$
$$A = r \mod N$$
$$A^{-1} = r^{-1} \mod N$$
$$A^{e} = r^{e} \mod N$$

2) Eve può calcolare **C'** moltiplicando $\mathbf{A^{e}}$ a **C**. Eve è consapevole che questo messaggio non è quello che sta cercando e lo invia a Bob solo per sfruttarne la decifratura che viene effettuata in caso di Frase Segreta errata.

$$C' = C \mathbf{x} A^{e} =$$
$$= C \mathbf{x} r^{e} \mod N$$

3) Bob riceve e controlla **C'** che non corrisponde alla Frase Segreta, quindi, risponde con un messaggio di errore e il messaggio decifrato **M'**. Questa operazione di decifratura è fondamentale, in quanto, il server produce **M'** che corrisponde a $\mathbf{M_{calc}}$ **x A** come mostrano i seguenti passaggi matematici.

$$M' = \text{Decrypt}(C') =$$
$$= C'^{d} \mod N =$$
$$= (C \mathbf{x} A^{e})^{d} \mod N =$$
$$= (C \mathbf{x} r^{e} \mod N)^{d} \mod N =$$
$$= (C^{d} \mod N) \mathbf{x} (r^{ed} \mod N) =$$
$$= (M_{calc}) \mathbf{x} (r \mod N) = M_{calc} \mathbf{x} A$$

4) Risulta evidente che Bob ha inconsapevolmente fornito all'interno di **M'** la Frase Segreta, in quanto, moltiplicandolo per $\mathbf{A^{-1}}$ si individua $\mathbf{M_{calc}}$ che coincide con il messaggio originario **M**. Eve effettua tale operazione, cifra il risultato e lo invia a Bob per verificarne la correttezza.

$$M' \mathbf{x} A^{-1} = M_{calc} \mathbf{x} A \mathbf{x} A^{-1} =$$
$$= M_{calc}$$

$$C_{calc} = \text{Encrypt}(M_{calc})$$

In conclusione, se tutte le operazioni sono state effettuate correttamente, il messaggio cifrato calcolato $\mathbf{C}_{\mathbf{calc}}$ corrisponde a **C**, dunque $\mathbf{M}_{\mathbf{calc}}$ è proprio il messaggio **M** contenente la Frase Segreta in chiaro che si voleva scoprire. [3]

3. Implementazione

In questa sezione si offre una panoramica generale sull'implementazione dello scenario descrivendone i principali componenti:

- bob.py
- alice.py
- eve.py
- rsa_tools.py
- client_utils.py

3.1 Server: bob.py

Il Server è implementato in Python utilizzando il framework Flask: permette a un client di ottenere la Chiave Pubblica di Bob, inviare messaggi crittografati contenti la Frase Segreta e ricevere risposte riguardo la correttezza della frase inviata. Bob verifica la Frase Segreta, se è corretta risponde con un messaggio di conferma, altrimenti risponde con un avviso di errore e il messaggio decifrato.

In particolare:

- Si genera e memorizza una coppia di chiavi crittografiche (Public Key e Private Key) di dimensione 2048 bit con un esponente pubblico fisso (65537).
- Si carica la coppia di chiavi memorizzate
- Si definisce e memorizza una "Frase Segreta" che viene cifrata con la Public Key di Bob.
- Si definisce una applicazione web Flask con tre endpoint:
    - '/' restituisce una pagina HTML di presentazione.
    - '/get_publickey' restituisce la chiave pubblica di Bob in formato JSON tramite il metodo GET.
    - '/post_message' riceve messaggi crittografati inviati tramite il metodo POST, decifra la Frase Segreta utilizzando la chiave privata di Bob e restituisce una risposta.

bob.py

```
#Genera PublicKey, PrivateKey
print(f"[ BOB ]:\t Genero la mia Public Key e la mia Private Key")
bob_e,bob_N,bob_d = generate_store_bob_keys(size = 2048, public_exponent=65537)

#LOAD Public Key Private Key
bob_e,bob_N = load_bob_publickey()
bob_d,bob_N = load_bob_privatekey()
print(f"[ BOB ]:\t Ho caricato la mia Public Key e la mia Private Key")

#Definizione e cifratura Frase Segreta
print(f"[ BOB ]:\t Definisco e cifro la Frase Segreta")
encrypted_frase_segreta = define_encrypt_secret(bob_e,bob_N)

app = Flask(__name__)

# Endpoint per restituire una pagina HTML
@app.route('/')
def index():
    return send_file('utils/index.html')

# Endpoint per restituire la Public Key di Bob tramite il metodo GET
@app.route('/get_publickey', methods=['GET'])
def get_publickey():
    data = {'e': bob_e, 'N': bob_N}
    return jsonify(data)

# Endpoint per ricevere messaggi tramite il metodo POST
@app.route('/post_message', methods=['POST'])
def post_data():
    data = request.json
    encrypted = int(data['message'])
    if encrypted == encrypted_frase_segreta:
        response = "ACK"
        print("\n[ BOB ]:\t La Frase Segreta ricevuta dal Client è CORRETTA.")
    else:
        response = decifratura_textbook(encrypted,bob_d,bob_N)
        print("\n[ BOB ]:\t La Frase Segreta ricevuta dal Client è ERRATA.")
    return jsonify(response)
```

3.2 Client legittimo: alice.py

Il Client legittimo è implementato in Python, interagisce con il Server, ottiene la Public Key di Bob e invia due Frasi Segrete a Bob in due tentativi separati.

alice.py

```
# GET Bob Public Key
bob_e,bob_N = get_bob_publickey()
print(f"[ALICE]:\t Ho ricevuto la Public Key del server Bob")

# -> INVIO n.1 (caso errato)
input("\n\t\t Premi Invio per provare la prima Frase Segreta.")
print("\n[ALICE]:\t *** Invio ***  Tentativo n.1")
parola_segreta_to_send = "GASPERINI Allenatore"
print(f"[ALICE]:\t Cifro e Invio la mia Frase Segreta a Bob:\t{parola_segreta_to_send}")
send_parola_segreta(parola_segreta_to_send,bob_e,bob_N)

# -> INVIO n.2 (caso corretto)
input("\n\t\t Premi Invio per provare la seconda Frase Segreta.")
print("\n[ALICE]:\t *** Invio ***  Tentativo n.2")
parola_segreta_to_send = "SSC Napoli CAMPIONE DI ITALIA 2023"
print(f"[ALICE]:\t Cifro e Invio la mia Frase Segreta a Bob:\t{parola_segreta_to_send}")
send_parola_segreta(parola_segreta_to_send,bob_e,bob_N)
```

3.3 Client malizioso: eve.py

Il Client malizioso ottiene la Chiave Pubblica di Bob, intercetta un messaggio cifrato trasmesso da Alice e poi procede a manipolarlo prima di inviarlo a Bob. Successivamente, Eve acquisisce la risposta di Bob contenente il messaggio decifrato, e grazie ad una ulteriore manipolazione matematica, riesce a risalire al messaggio originario di Alice.

eve.py

```
# GET Bob Public Key
bob_e,bob_N = get_bob_publickey()
print(f"[ EVE ]:\t Ho ricevuto la Public Key del server Bob")

input("\n\t\t Premi Invio per caricare il messaggio cifrato C (intercettato tra Alice e Bob).")
#LOAD Messaggio (Chiave Segreta) cifrato inviato da Alice a Bob
intercepted_message = load_message_alice_to_bob()
formatted_intercepted_message = "{:.30s}".format(str(intercepted_message))
print(f"\n[ EVE ]:\t Ho intercettato la seguente comunicazione cifrata tra Alice e Bob:\n\t\t C
= {formatted_intercepted_message}... (continua)")

input("\n\t\t Premi Invio per calcolare il nuovo messaggio cifrato C'")
```

```
#CALCOLO di un nuovo messaggio: (MSG_cifrato_intercettato x RAND^e)mod_N
print("[ EVE ]:\t Calcolo un nuovo messaggio come segue: C' = (C x R^e)mod_N")
rand_value = random.randint(2, 99)
tampered_message_from_alice = (intercepted_message * pow(rand_value,bob_e,bob_N) ) % bob_N
formatted_tamp_msg = "{:.30s}".format(str(tampered_message_from_alice))
print(f"[ EVE ]:\t Il nuovo messaggio 'esca' calcolato è il seguente:\n\t\t C' =
{formatted_tamp_msg}... (continua)")

#Invio n.1 (C')
input("\n\t\t Premi Invio per inviare al server C'")
print(f"\n[ EVE ]:\t Invio il messaggio calcolato (non corrisponde sicuramente alla Frase
Segreta)")
resp_from_bob = send_to_bob(tampered_message_from_alice)
check_from_bob = check_bob_res(resp_from_bob)

#Calcolare inverso modulare
input("\n\t\t Premi Invio per risalire matematicamente alla Frase Segreta")
print(f"\n[ EVE ]:\t Utilizzo la risposta M' di Bob per calcolare la Frase Segreta calcolando M'
x R^-1 mod_N")
inv_resp_from_bob =  (resp_from_bob * libnum.invmod(rand_value,bob_N)) % bob_N
maybe_the_secret = intero_a_testo(inv_resp_from_bob)
print(f"[ EVE ]:\t Credo che la frase segreta sia:\t M* = {maybe_the_secret}\n\t\t Invio a Bob
per la verifica ...\n")

#Cifro con Public Key di BOB
encr_inv_resp_from_bob = cifratura_textbook(inv_resp_from_bob,bob_e,bob_N)

#POST n.2 (M)
resp_from_bob = send_to_bob(encr_inv_resp_from_bob)
check_from_bob = check_bob_res(resp_from_bob)
if check_from_bob == True:
    print(f"\n[ EVE ]:\t Sono riuscito a risalire alla FRASE SEGRETA!!!!\n\t\t M =
{maybe_the_secret}")
else:
    print(f"\n[ EVE ]:\t Non sono stato in grado di risalire alla FRASE SEGRETA")
```

3.3 Funzioni per la crittografia: rsa_tools.py

Il file **rsa_tools.py** contiene l'implementazione delle funzioni fondamentali per generare coppie di chiavi, e cifrare e decifrare i messaggi. Di seguito vengono illustrate le funzioni sviluppate.

- **generate_key_pair(size, public_exponent)**: Genera una coppia di chiavi RSA, composta da una chiave privata e una chiave pubblica, con la dimensione specificata (size) e l'esponente pubblico (public_exponent) specificato.
- **publickey_parameters(public_key)**: Estrae i parametri (e, N) dalla chiave pubblica RSA fornita come input.
- **privatekey_parameters(private_key)**: Estrae i parametri (d, N) dalla chiave privata RSA fornita come input.
- **encrypt_textbookRSA(string, e, N)**: Cifra una stringa di testo utilizzando l'algoritmo RSA "Textbook" con la chiave pubblica specificata (e, N) e restituisce il testo cifrato.
- **decrypt_textbookRSA(int_number, d, N)**: Decifra un numero intero (testo cifrato) utilizzando l'algoritmo RSA "Textbook" con la chiave privata specificata (d, N) e restituisce il testo decifrato.
- **cifratura_textbook(M, e, N)**: Esegue l'operazione di cifratura RSA "Textbook" su un intero M utilizzando la chiave pubblica specificata (e, N) e restituisce il testo cifrato.
- **decifratura_textbook(C, d, N)**: Esegue l'operazione di decifratura RSA "Textbook" su un intero C utilizzando la chiave privata specificata (d, N) e restituisce il testo decifrato.
- **testo_a_intero(testo)**: Converte una stringa di testo in un numero intero utilizzando la codifica base64. Questa funzione è utilizzata per preparare il testo per la cifratura.
- **intero_a_testo(intero)**: Converte un numero intero in una stringa di testo utilizzando la decodifica base64. Questa funzione viene utilizzata per ottenere il testo decifrato.

rsa_tools.py

```
'''
    Funzioni per PublicKey e Private Key
'''
#Genera coppia di chiavi
def generate_key_pair(size, public_exponent):
    private_key = rsa.generate_private_key(
        public_exponent=public_exponent,
        key_size=size,
        backend=default_backend()
    )
    public_key = private_key.public_key()
    return private_key, public_key

#Estrazione (e,N) da Public Key
def publickey_parameters(public_key):
    public_numbers = public_key.public_numbers()
    e = public_numbers.e
    n = public_numbers.n
    return e, n

#Estrazione (d,N) da Private Key
def privatekey_parameters(private_key):
    private_numbers = private_key.private_numbers()
    d = private_numbers.d
    n = private_numbers.public_numbers.n
    return d, n

'''
    Funzioni per Cifrare e Decifrare
    RSA Textbook
'''
# Cifra testo con Textbook RSA
def encrypt_textbookRSA(string,e,N):
    int_string = testo_a_intero(string)                 # Converte testo -> intero
    encrypted  = cifratura_textbook(int_string,e,N)     # Cifra intero
    return encrypted

# Decifra testo cifrato con Textbook RSA
def decrypt_textbookRSA(int_number,d,N):
    string_int = intero_a_testo(int_number)             # Converte intero -> testo
    decrypted = decifratura_textbook(string_int,d,N)    # Decifra intero
    return decrypted

#Encrypt - Operazione
def cifratura_textbook(M, e, N):
    cipher = M ** e % N         # operazione di cifratura: (M^e) mod_N
    return cipher
```

```python
#Decrypt - Operazione
def decifratura_textbook(C,d,N):
    decipher = pow(C, d , N)    # operazione di decifratura: (C^d) mod_N
    return decipher

#Conversione Testo->Intero
def testo_a_intero(testo):
    testo_codificato = base64.b64encode(testo.encode('utf-8'))
# Converte la stringa in una sequenza di byte codificata in base64.
    intero = int.from_bytes(testo_codificato, byteorder='big')
# Converte la sequenza di byte codificata in base64 in un intero
    return intero

#Conversione Intero->Testo
def intero_a_testo(intero):
    testo_codificato = intero.to_bytes((intero.bit_length() + 7) // 8, byteorder='big')
# Calcola il numero di byte necessari e converte l'intero in una sequenza di byte
    testo = base64.b64decode(testo_codificato).decode('utf-8')
# Converte la sequenza di byte in una stringa
    return testo
```

3.4 Funzioni per i client: client_utils.py

Le funzioni realizzate per il Client sono state implementate in **client_utils.py** e sono illustrate di seguito.

- **get_bob_publickey()**: Effettua una richiesta GET al server per ottenere la chiave pubblica di Bob. Estrae i valori dell'esponente bob_e e del modulo bob_N dalla risposta del server.
- **send_parola_segreta(frase_to_send, e_val, N_val)**: Cifra una frase segreta frase_to_send utilizzando la chiave pubblica di Bob (definita da e_val e N_val) e la invia al server. La funzione memorizza anche il messaggio su un file, invia il messaggio al server e verifica la risposta da Bob.
- **send_to_bob(my_secret_encrypted)**: Effettua una richiesta POST al server Bob, inviando un messaggio cifrato my_secret_encrypted. Restituisce la risposta ricevuta dal server.
- **check_bob_res(resp_from_bob)**: Verifica la risposta ricevuta da Bob. Se la risposta è "ACK," la funzione indica che la frase segreta è stata correttamente ricevuta e decifrata dal server. In caso contrario, segnala un errore.
- **store_message_alice_to_bob(message)**: Memorizza un messaggio su un file specifico.
- **load_message_alice_to_bob()**: Carica un messaggio precedentemente memorizzato da un file.

client_utils.py

```
#Get
def get_bob_publickey():
    response = requests.get(URL_server_publickey)
    if response.status_code == 200:
        data = response.json()
        bob_e = data['e']
        bob_N = data['N']
    else:
        print('Errore nella richiesta al server.')
    return bob_e,bob_N
def send_parola_segreta(frase_to_send,e_val,N_val): #Post
    my_secret_encrypted  = encrypt_textbookRSA(frase_to_send,e_val,N_val)
    # POST encrypted secret
    store_message_alice_to_bob(my_secret_encrypted)     # Memorizza su file
    resp_from_bob = send_to_bob(my_secret_encrypted)    # Effettua Post
    check_bob_res(resp_from_bob)                      # Legge e verifica risposta
    return resp_from_bob

def send_to_bob(my_secret_encrypted): #INVIO msg a Server
    payload = {'message': int(my_secret_encrypted)}   # Creazione messaggio da inviare a Bob
    response = requests.post(URL_server_message, json=payload)
    if response.status_code == 200:  #RICEZIONE msg da Server
    # Controlla se la richiesta ha avuto successo (codice di stato 200)
        data_from_bob = response.json()
     # Ricevi il valore come JSON e memorizzalo in una variabile
        return data_from_bob
    else:
        print("Errore nella richiesta al server Bob:", response.status_code)
        return ""
# Verifica risposta di Bob
def check_bob_res(resp_from_bob):
    if resp_from_bob == "ACK":
        print(f"[ BOB ]:\t FRASE SEGRETA CORRETTA. -  Il Server ha risposto: \t{resp_from_bob}")
        return True
    else:
        formatted_resp_from_bob = "{:.30s}".format(str(resp_from_bob))
        print(f"[ BOB ]:\t FRASE SEGRETA ERRATA!   -  Il Server ha decifrato:
\t{formatted_resp_from_bob}... (continua)")
        return False
# Memorizza e Carica messaggi su/da file
def store_message_alice_to_bob(message):
    with open(PATH_communications_a_b, 'w') as file:
        file.write(str(message))
def load_message_alice_to_bob():
    with open(PATH_communications_a_b, 'r') as file:
        stored = int(file.read())
    return stored
```

4. Simulazione

In questa sezione si procede con la simulazione dello scenario descritto.
Il Server Bob si avvia, crea una coppia di chiavi e cifra la Frase Segreta, Alice usa la Chiave Pubblica di Bob per cifrare e inviare un messaggio cifrato diverso dalla Frase Segreta cifrata, ricevendo come risposta un errore, seguito dal messaggio decifrato.

Alice

```
PS C:\Users\seraf\Desktop\Simulazione_ChosenCiphertextAttack> python .\alice.py

* * * * * CLIENT STARTED * * * * *

                Premi Invio per chiedere a Bob la PublicKey.
[ALICE]:        Ho ricevuto la Public Key del server Bob

                Premi Invio per provare la prima Frase Segreta.

[ALICE]:        *** Invio ***  Tentativo n.1
[ALICE]:        Cifro e Invio la mia Frase Segreta a Bob:      GASPERINI Allenatore
[ BOB ]:        FRASE SEGRETA ERRATA!   -  Il Server ha decifrato:     865546698654002280433363280804... (continua)

                Premi Invio per provare la seconda Frase Segreta.
```

Bob

```
PS C:\Users\seraf\Desktop\Simulazione_ChosenCiphertextAttack> python .\bob.py

* * * * * SERVER STARTING * * * * *
[ BOB ]:        INIZIALIZZAZIONE ...
[ BOB ]:        Genero la mia Public Key e la mia Private Key
[ BOB ]:        Ho caricato la mia Public Key e la mia Private Key
[ BOB ]:        Definisco e cifro la Frase Segreta
[ BOB ]:        Starting service...
 * Serving Flask app 'bob'
 * Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
127.0.0.1 - - [03/Sep/2023 18:33:24] "GET /get_publickey HTTP/1.1" 200 -

[ BOB ]:        La Frase Segreta ricevuta dal Client è ERRATA.
127.0.0.1 - - [03/Sep/2023 18:33:30] "POST /post_message HTTP/1.1" 200 -
```

Alice procede con un secondo invio, questa volta trasmettendo a Bob il messaggio cifrato che contiene la Frase Segreta corretta. Bob risponde con un messaggio di conferma.

Alice

```
[ALICE]:        *** Invio ***  Tentativo n.2
[ALICE]:        Cifro e Invio la mia Frase Segreta a Bob:      SSC Napoli CAMPIONE DI ITALIA 2023
[ BOB ]:        FRASE SEGRETA CORRETTA. -  Il Server ha risposto:      ACK
* * * * * CLIENT ENDED * * * * *

Premi Invio per uscire...
```

Bob

```
[ BOB ]:        La Frase Segreta ricevuta dal Client è CORRETTA.
127.0.0.1 - - [03/Sep/2023 18:41:35] "POST /post_message HTTP/1.1" 200 -
```

Eve è in grado di intercettare il messaggio cifrato che Alice invia a Bob ed effettua su di esso le manipolazioni aritmetiche precedentemente illustrate. Successivamente, Eve trasmette il messaggio calcolato a Bob che risponde con un avviso di errore e il messaggio decifrato.

Eve

```
PS C:\Users\seraf\Desktop\Simulazione_ChosenCiphertextAttack> python .\eve.py

* * * * * EVE STARTED * * * * *

                Premi Invio per chiedere a Bob la PublicKey.
[ EVE ]:        Ho ricevuto la Public Key del server Bob

                Premi Invio per caricare il messaggio cifrato C (intercettato tra Alice e Bob).

[ EVE ]:        Ho intercettato la seguente comunicazione cifrata tra Alice e Bob:
                C = 9065505019079975296193870846O9... (continua)

                Premi Invio per calcolare il nuovo messaggio cifrato C'
[ EVE ]:        Calcolo un nuovo messaggio come segue: C' = (C x R^e)mod_N
[ EVE ]:        Il nuovo messaggio 'esca' calcolato è il seguente:
                C' = 45224814226378082374071633995O... (continua)

                Premi Invio per inviare al server C'

[ EVE ]:        Invio il messaggio calcolato (non corrisponde sicuramente alla Frase Segreta)
[ BOB ]:        FRASE SEGRETA ERRATA!   -  Il Server ha decifrato:     970313256261694485266737461588... (continua)

                Premi Invio per risalire matematicamente alla Frase Segreta
```

Bob

```
127.0.0.1 - - [03/Sep/2023 18:45:41] "GET /get_publickey HTTP/1.1" 200 -

[ BOB ]:        La Frase Segreta ricevuta dal Client è ERRATA.
127.0.0.1 - - [03/Sep/2023 18:45:57] "POST /post_message HTTP/1.1" 200 -
```

Eve sfrutta il messaggio decifrato che ha ricevuto da Bob per eseguire ulteriori operazioni aritmetiche che generano il testo originale di Alice. In seguito, Eve cifra e invia questo messaggio, che Bob verifica con esito positivo.

Eve

```
                Premi Invio per risalire matematicamente alla Frase Segreta

[ EVE ]:        Utilizzo la risposta M' di Bob per calcolare la Frase Segreta calcolando M' x R^-1 mod_N
[ EVE ]:        Credo che la frase segreta sia:         M* = SSC Napoli CAMPIONE DI ITALIA 2023
                Invio a Bob per la verifica ...

[ BOB ]:        FRASE SEGRETA CORRETTA. -  Il Server ha risposto:      ACK

[ EVE ]:        Sono riuscito a risalire alla FRASE SEGRETA!!!!
                M = SSC Napoli CAMPIONE DI ITALIA 2023
* * * * * EVE ENDED * * * * *

Premi Invio per uscire...
```

Bob

```
[ BOB ]:        La Frase Segreta ricevuta dal Client è CORRETTA.
127.0.0.1 - - [03/Sep/2023 18:48:20] "POST /post_message HTTP/1.1" 200 -
```

Eve è riuscito a individuare la Frase Segreta di Bob, inizialmente crittografata da Alice, pur non avendo mai avuto conoscenza della Private Key. Questo risultato è stato reso possibile grazie alle operazioni matematiche descritte nelle sezioni precedenti e alle risposte incaute fornite da Bob.

Bibliografia

[1] N. P. Smart, “The “Naive” RSA Algorithm,” in *Cryptography Made Simple. Information Security and Cryptography*, Springer, 2016, p. 295–311.

[2] D. Bleichenbacher, “Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS #1,” in *vances in Cryptology - CRYPTO '98, 18th Annual International Cryptology Conference, Santa Barbara, California, USA, August 23-27, 1998, Proceedings*, vol. 1462, Springer, 1998, pp. 1-12.

[3] I. &. A. D. &. O. S. Salah, “Mathematical Attacks on RSA Cryptosystem,” *Journal of Computer Science, Vol.2,* pp. 665-671, 2006.

[4] Y. L. Jonathan Katz, “3.7 Chosen-Ciphertext Attacks,” in *Introduction to Modern Cryptography*, Chapman and Hall/CRC, 2014, pp. 96-98.