Assignment #2

RSA with Cipher Block Chaining (RSA-CBC)

- TCP Protocol
- Cipher Block Chaining
- RSA
- Start-up codes: TCP client-server programs

n.h.reyes@massey.ac.nz

Main components

Main task

- Implement a hybrid encryption algorithm called RSA with Cipher Block Chaining for securely communicating TCP client-server applications.

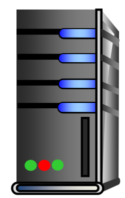
plaintext message: meet me at sunset

Encrypted message: 23938 19210 11863 9019 13976 4346 22722 4827 17725 13477 21270 11359 10558 12340 3362 3974 10785 25372 16707

Client

Decrypt the encrypted message: 23938 19210 11863 9019 13976 4346 22722 4827 17725 13477 21270 11359 10558 12340 3362 3974 10785 25372 16707

meet me at sunset



Server

RSA_CBC

Encryption

m – plaintext message

m = ‘A A A \r\n’

No regular pattern should emerge

RSA_CBC (m) → | 12589 | 50082 | 54789 | 13389 | 59510

ciphertext

Main components

Client (sender)	Server (receiver)

To reduce the complexity of the assignment, the scope is limited to one-way communication. The client is the sender of encrypted messages, and the server is the receiver

Main components

Client (sender)

Server (receiver)

- has a public key **(e,n),** and a private key **(d,n)**

- has a certificate issued by a CA. It is denoted as **dCA(e,n),** it is an encrypted public key of the server

Main components

Client (sender)	Server (receiver)
eCA – is an RSA public key **nonce** – to be used as part of Cipher Block Chaining	• has a public key **(e,n),** and a private key **(d,n)** • has a certificate issued by a CA. It is denoted as **dCA(e,n),** it is an encrypted public key of the server

Main components

Client (sender)

eCA – is an RSA public key

nonce – to be used as part of Cipher Block Chaining

Server (receiver)

- has a public key **(e,n),** and a private key **(d,n)**
- has a certificate issued by a CA.

It is denoted as **dCA(e,n),** it is an encrypted public key of the server

Care must be taken when assigning values to keys and nonce:

Client's variable:
nonce

Certification Authority's keys:
eCA, dCA, nCA

nCA > n

nonce < n

Server's keys:
e, d, n

Key exchange

Sequence of steps:

Client (sender)	Server (receiver)
	• has a public key **(e,n),** and a private key **(d,n)**
eCA – is an RSA public key **nonce** – to be used as part of Cipher Block Chaining	• has a certificate issued by a CA. It is denoted as **dCA(e,n),** it is an encrypted public key of the server

TCP connection establishment

1. send certificate **dCA(e,n)** to client

Key exchange

Sequence of steps:

Client (sender)

eCA – is an RSA public key

nonce – to be used as part of Cipher Block Chaining

2. extract the server's **public key (e,n)** using its copy of the Certification Authority's public key.

eCA (dCA(e,n)) $\rightarrow$ **(e,n)**

Server (receiver)

- has a public key **(e,n),** and a private key **(d,n)**

- has a certificate issued by a CA.

It is denoted as **dCA(e,n),** it is an encrypted public key of the server

1. send dCA(e,n) to client

Key exchange

Sequence of steps:

Client (sender)

eCA – is an RSA public key

nonce – to be used as part of Cipher Block Chaining

2. extract the server's public key **(e,n)** using its copy of the Certification Authority's public key.

eCA (dCA(e,n)) $\rightarrow$ **(e,n)**

3. Send "ACK 226 public key received".

Server (receiver)

- has a public key **(e,n),** and a private key **(d,n)**

- has a certificate issued by a CA.

It is denoted as **dCA(e,n),** it is an encrypted public key of the server

1. send dCA(e,n) to client

Key exchange

Sequence of steps:

Client (sender)

eCA – is an RSA public key

nonce – to be used as part of Cipher Block Chaining

2. extract the server's public key **(e,n)** using its copy of the Certification Authority's public key.

eCA (dCA(e,n)) $\rightarrow$ **(e,n)**

3. Send "ACK 226 public key received".

4. Send **e(nonce).**

Server (receiver)

- has a public key **(e,n),** and a private key **(d,n)**
- has a certificate issued by a CA.

It is denoted as **dCA(e,n),** it is an encrypted public key of the server

1. send dCA(e,n) to client

Key exchange

Sequence of steps:

Client (sender)

eCA – is an RSA public key

nonce – to be used as part of Cipher Block Chaining

2. extract the server's public key **(e,n)** using its copy of the Certification Authority's public key.

eCA (dCA(e,n)) → (e,n)

3. Send "ACK 226 public key received".

4. Send **e(nonce).**

Server (receiver)

- has a public key **(e,n),** and a private key **(d,n)**

- has a certificate issued by a CA.

It is denoted as **dCA(e,n),** it is an encrypted public key of the server

1. send dCA(e,n) to client

5. Extract the **nonce** using its private key, **(d,n)**. This involves the RSA decryption operation, **d(e(nonce))**. Subsequently, it sends "ACK 220 nonce ok".

Key exchange

Sequence of steps:

Client (sender) | Server (receiver)

Communication Session

Successive messages coming from the client will then be encrypted accordingly using RSA-CBC, and the server will decrypt the received messages using RSA-CBC.

Key exchange

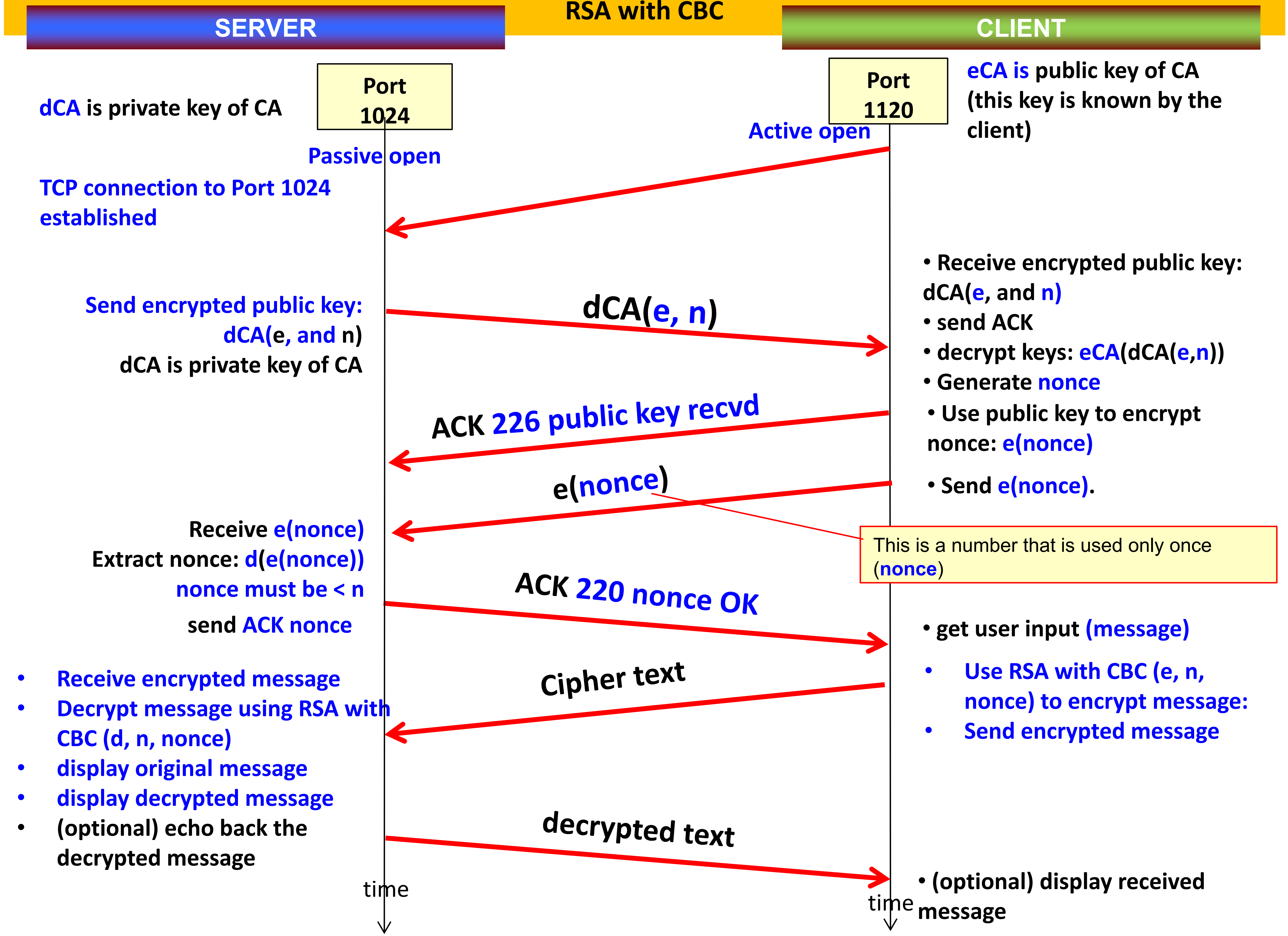
Sequence of steps:

Client (sender) Server (receiver)

Successive messages coming from the client will then be encrypted accordingly using RSA-CBC, and the server will decrypt the received messages using RSA-CBC.

The client-server programs must be able to show the original, encrypted and decrypted messages for verification purposes.

RSA with CBC


SERVER
CLIENT
Port 1024
Port 1120
dCA is private key of CA
eCA is public key of CA (this key is known by the client)
Passive open
Active open
TCP connection to Port 1024 established
Send encrypted public key: dCA(e, and n)
dCA is private key of CA
dCA(e, n)
• Receive encrypted public key: dCA(e, and n)
• send ACK
• decrypt keys: eCA(dCA(e,n))
• Generate nonce
• Use public key to encrypt nonce: e(nonce)
ACK 226 public key recvd
e(nonce)
• Send e(nonce).
This is a number that is used only once (nonce)
Receive e(nonce)
Extract nonce: d(e(nonce))
nonce must be < n
send ACK nonce
ACK 220 nonce OK
• get user input (message)
• Use RSA with CBC (e, n, nonce) to encrypt message:
• Send encrypted message
Cipher text
• Receive encrypted message
• Decrypt message using RSA with CBC (d, n, nonce)
• display original message
• display decrypted message
• (optional) echo back the decrypted message
decrypted text
• (optional) display received message
time
time

Demonstration

CLIENT

```
===<<< SECURE TCP CLIENT >>>===
===<< by: n.h.reyes@massey.ac.nz >>===

The Winsock 2.2 dll was initialised.
USAGE: ClientWindows IP-address [port]
Default portNum = 1234
Using default settings, IP:localhost, Port:1234

Connected to <<<SERVER>>> with IP address: localhost, IPv6 at port: 1234

Connected to <<<SERVER>>> extracted IP address: ::1, IPv6 at port: 1234

Received Server's Certificate: PUBLIC_KEY 51177, 42697, scannedItems = 2
Decrypted Server's Public key: [e = 737, n = 67637]
---> Sending reply to SERVER: ACK 226 Public key received

---> Sending Nonce to SERVER: NONCE 16150

Received packet: ACK 220 nonce ok, scannedItems = 1
nonce ACKed by Server

--------------------------------------------
you may now start sending commands to the <<<SERVER>>>.
===========================================
Enter message to send:
===========================================
```

SERVER

```
===<<< SECURE TCP SERVER >>>===
===<< by: n.h.reyes@massey.ac.nz >>===

The Winsock 2.2 dll was initialised.

Using DEFAULT_PORT = 1234

getaddrinfo response 1
        Flags: 0x0
        Family: AF_INET6 (IPv6)
        IPv6 address [::]:1234
        Socket type: SOCK_STREAM (stream)
        Protocol: IPPROTO_TCP (TCP)
        Length of this sockaddr: 28
        Canonical name: (null)

<<<SERVER>>> is listening at PORT: 1234

A <<<CLIENT>>> has been accepted.

Connected to <<<Client>>> with IP address:::1, at Port:55362
Sending packet: PUBLIC_KEY 51177, 42697

Received packet: ACK 226 Public key received, scannedItems = 1
Received packet: NONCE 16150, scannedItems = 1
After decryption, received nonce = 1234

nonce = 1234
Sending packet: ACK 220 nonce ok


--------------------------------------------
the <<<SERVER>>> is waiting to receive commands.
```

Decrypted server's public key
(e,n)=(737, 67637)

Encrypted server's public key
dCA(e,n)=(51177, 42697)

Demonstration

CLIENT

```
===<<< SECURE TCP CLIENT >>>===
===<< by: n.h.reyes@massey.ac.nz >>===

The Winsock 2.2 dll was initialised.
USAGE: ClientWindows IP-address [port]
Default portNum = 1234
Using default settings, IP:localhost, Port:1234

Connected to <<<SERVER>>> with IP address: localhost, IPv6 at port: 1234

Connected to <<<SERVER>>> extracted IP address: ::1, IPv6 at port: 1234

Received Server's Certificate: PUBLIC_KEY 51177, 42697, scannedItems = 2
Decrypted Server's Public key: [e = 737, n = 67637]
---> Sending reply to SERVER: ACK 226 Public key received

---> Sending Nonce to SERVER: NONCE 16150

Received packet: ACK 220 nonce ok, scannedItems = 1
nonce ACKed by Server

--------------------------------------------
you may now start sending commands to the <<<SERVER>>>.
===========================================
Enter message to send:
===========================================
```

SERVER

```
===<<< SECURE TCP SERVER >>>===
===<< by: n.h.reyes@massey.ac.nz >>===

The Winsock 2.2 dll was initialised.

Using DEFAULT_PORT = 1234

getaddrinfo response 1
        Flags: 0x0
        Family: AF_INET6 (IPv6)
        IPv6 address [::]:1234
        Socket type: SOCK_STREAM (stream)
        Protocol: IPPROTO_TCP (TCP)
        Length of this sockaddr: 28
        Canonical name: (null)

<<<SERVER>>> is listening at PORT: 1234

A <<<CLIENT>>> has been accepted.

Connected to <<<Client>>> with IP address:::1, at Port:55362
Sending packet: PUBLIC_KEY 51177, 42697

Received packet: ACK 226 Public key received, scannedItems = 1
Received packet: NONCE 16150, scannedItems = 1
After decryption, received nonce = 1234

nonce = 1234
Sending packet: ACK 220 nonce ok


--------------------------------------------
the <<<SERVER>>> is waiting to receive commands.
```

Encrypted nonce:
e(nonce) = 16150

decrypted nonce
nonce=1234

Demonstration

CLIENT

C:\WINDOWS\system32\cmd.exe - RSA_CBC_client\RSAxClient localhost 1235 1 0

```
===<<< SECURE TCP CLIENT >>>===
===<< by: n.h.reyes@massey.ac.nz >>===

The Winsock 2.2 dll was initialised.
USAGE: ClientWindows IP-address [port]
Default portNum = 1234
Using default settings, IP:localhost, Port:1234

Connected to <<<SERVER>>> with IP address: localhost, IPv6 at port: 1234

Connected to <<<SERVER>>> extracted IP address: ::1, IPv6 at port: 1234

Received Server's Certificate: PUBLIC_KEY 51177, 42697, scannedItems = 2
Decrypted Server's Public key: [e = 737, n = 67637]
---> Sending reply to SERVER: ACK 226 Public key received

---> Sending Nonce to SERVER: NONCE 16150

Received packet: ACK 220 nonce ok, scannedItems = 1
nonce ACKed by Server

--------------------------------------------
you may now start sending commands to the <<<SERVER>>>.
===========================================
Enter message to send:
===========================================
```

SERVER

C:\WINDOWS\system32\cmd.exe - RSA_CBC_server\RSAxServer 1235 1 0

```
Using DEFAULT_PORT = 1234

getaddrinfo response 1
        Flags: 0x0
        Family: AF_INET6 (IPv6)
        IPv6 address [::]:1234
        Socket type: SOCK_STREAM (stream)
        Protocol: IPPROTO_TCP (TCP)
        Length of this sockaddr: 28
        Canonical name: (null)

<<<SERVER>>> is listening at PORT: 1234

A <<<CLIENT>>> has been accepted.

Connected to <<<Client>>> with IP address:::1, at Port:55362
Sending packet: PUBLIC_KEY 51177, 42697

Received packet: ACK 226 Public key received, scannedItems = 1
Received packet: NONCE 16150, scannedItems = 1
After decryption, received nonce = 1234

nonce = 1234
Sending packet: ACK 220 nonce ok


--------------------------------------------
the <<<SERVER>>> is waiting to receive commands.
```

RSA: Choosing keys

1. Choose two large prime numbers **p**, **q**. (e.g., 1024 bits each)

2. Compute **n** = **pq**, z = (**p** -1)(**q** -1)

3. Choose **e** (with **e** **<** **n**) that has no common factors with z. (e, z are 'relatively prime').

4. Choose **d** such that **ed** -1 is exactly divisible by z. (in other words: ed mod z = 1).

5. Public key is (**e, n**). Private key is (**d, n**).

In mathematics, a **prime number** (or a prime) is a natural number that has exactly two (distinct) natural number divisors, which are 1 and the prime number itself. The first 30 prime numbers are 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, and 113

RSA: Encryption, decryption

0. Given a public key (***e, n***) and a private key (***d, n***)

1. <u>To **encrypt** bit pattern, *m*, compute</u>

   $c = m^e \bmod n$ (i.e., remainder when m^e is divided by n)

2. <u>To **decrypt** received bit pattern, *c*, compute</u>

   $m = c^d \bmod n$ (i.e., remainder when c^d is divided by n)

Magic happens! $m = (m^e \bmod n)^d \bmod n$

RSA example:

Bob chooses $p=5$, $q=7$. Then $n=35$, $z=24$.

$e=5$ (so e, z relatively prime).
$d=29$ (so $ed-1$ exactly divisible by z)
(equivalently, ed mod z = 1).

encrypt:

letter	m	m^e	$c = m^e \bmod n$
ℓ	12	248832	17

---

decrypt:

c	c^d	$m = c^d \bmod n$	letter
17		12	ℓ

$\underline{c}^d$ = 481968572106750915091411825223072000 - **too big !!** (**int** type)

How to solve this problem:

Repeated Squaring: calculate $y = x^{e} \bmod n$

```
long repeatSquare(long x, long e, long n) {

y=1;//initialize y to 1, very important
while (e >  0) {
        if (( e % 2 ) == 0) {
                x = (x*x) % n;
                e = e/2;
        }
        else {
                y = (x*y) % n;
                e = e-1;
        }
}
return y; //the result is stored in y
}
```

$$c = m^{e} \bmod n$$

$$m = c^{d} \bmod n$$

Use **repeatSquare** for both encryption and decryption. You might want to change the datatypes used in this function to accommodate bigger numbers in the encryption scheme.

Summary of Techniques

RSA public key (***e, n***) and a private key (***d, n***)

Check if **e** & **z** are coprimes.

Euclidean Algorithm gcd(e,z):

Solve for **d**

Extended Euclidean Algorithm

$zx + ed = gcd(z,e)=1$

Bézout's identity.

It's not very difficult to implement both Algorithms for automatically generating the RSA keys.

Example of RSA keys

1024-bit RSA encryption key (in hex format):

n=
A9E167983F39D55FF2A093415EA6798985C8355D9A915BFB1D01DA197026170F
BDA522D035856D7A986614415CCFB7B7083B09C991B81969376DF9651E7BD9A9
3324A37F3BBBAF460186363432CB07035952FC858B3104B8CC18081448E64F1C
FB5D60C4E05C1F53D37F53D86901F105F87A70D1BE83C65F38CF1C2CAA6AA7EB

e=010001

d=
67CD484C9A0D8F98C21B65FF22839C6DF0A6061DBCEDA7038894F21C6B0F8B35
DE0E827830CBE7BA6A56AD77C6EB517970790AA0F4FE45E0A9B2F419DA8798D6
308474E4FC596CC1C677DCA991D07C30A0A2C5085E217143FC0D073DF0FA6D14
9E4E63F01758791C4B981C3D3DB01BDFFA253BA3C02C9805F61009D887DB0319

https://www.boost.org/

http://www.di-mgt.com.au/rsa_alg.html#realexample

You need a big number library, like **Boost** to implement values at this range

Sample Small Keys

p=257 and q=293

e=529;
d=24305;
n=75301;

Use keys at least around the following range of values. Remember, the bigger the keys, the better! However, you need to make sure that the data type you are using can accommodate the range of values for which the algorithm will be operating on.

Review of CBC

- Given the binary representation of a plain text message (m), compute for its cipher text using CBC. Assume that the message is processed in blocks of **3 bits**, and that the initial random number is equal to **010**. Fill-up the following table with the details of your solution.

m = 110110110

1) ENCRYPTION

i	1	2	3
m(i)			
r(i)	010		
c(i)			
INFO. TO SEND			

Table representing $K_s()$

input	output	input	output
000	110	100	011
001	111	101	010
010	101	110	000
011	100	111	001

Put initial random number as the value for **r(1)**

Review of CBC

- Given the binary representation of a plain text message (m), compute for its cipher text using CBC. Assume that the message is processed in blocks of **3 bits**, and that the initial random number is equal to **010**. Fill-up the following table with the details of your solution.

m = 110110110

1) ENCRYPTION

i	1	2	3
m(i)	110	110	110
r(i)			
c(i)			
INFO. TO SEND			

Table representing $K_s()$

input	output	input	output
000	110	100	011
001	111	101	010
010	101	110	000
011	100	111	001

Split the original plaintext message into blocks of 3-bits

Review of CBC

- Given the binary representation of a plain text message (m), compute for its cipher text using CBC. Assume that the message is processed in blocks of **3 bits**, and that the initial random number is equal to **010**. Fill-up the following table with the details of your solution.

m = 110110110

1) ENCRYPTION

i	1	2	3
m(i)	110	110	110
r(i)	010		
c(i)	011		
INFO. TO SEND			

Table representing $K_s()$

input	output	input	output
000	110	100	011
001	111	101	010
010	101	110	000
011	100	111	001

Solve for c(1):

$K_s(m(1) \oplus r(1)) = K_s(110 \oplus 010) = K_s(100) = 011$

Review of CBC

- Given the binary representation of a plain text message (m), compute for its cipher text using CBC. Assume that the message is processed in blocks of **3 bits**, and that the initial random number is equal to **010**. Fill-up the following table with the details of your solution.

m = 110110110

1) ENCRYPTION

i	1	2	3
m(i)	110	110	110
r(i)	010	011	
c(i)	011		
INFO. TO SEND			

Table representing $K_s()$

input	output	input	output
000	110	100	011
001	111	101	010
010	101	110	000
011	100	111	001

The next random number block is equal to the previous cipherblock, C(1). Therefore, copy c(1) onto r(2).

Review of CBC

- Given the binary representation of a plain text message (m), compute for its cipher text using CBC. Assume that the message is processed in blocks of **3 bits**, and that the initial random number is equal to **010**. Fill-up the following table with the details of your solution.

m = 110110110

1) ENCRYPTION

i	1	2	3
m(i)	110	110	110
r(i)	010	011	
c(i)	011	010	
INFO. TO SEND			

Table representing $K_s()$

input	output	input	output
000	110	100	011
001	111	101	010
010	101	110	000
011	100	111	001

Compute for c(2).

$K_s(m(2) \oplus r(2)) = K_s(110 \oplus 011) = K_s(101) = 010$

Review of CBC

- Given the binary representation of a plain text message (m), compute for its cipher text using CBC. Assume that the message is processed in blocks of **3 bits**, and that the initial random number is equal to **010**. Fill-up the following table with the details of your solution.

m = 110110110

1) ENCRYPTION

i	1	2	3
m(i)	110	110	110
r(i)	010	011	010
c(i)	011	010	011
INFO. TO SEND			

Table representing $K_s()$

input	output	input	output
000	110	100	011
001	111	101	010
010	101	110	000
011	100	111	001

Continue until we have computed all the **cipher blocks**.

Message to send: = 010 011 010 011

Review of CBC

2) DECRYPTION

Using your answer in Item#1 (information to send), decrypt the message using CBC. Fill-up the table with the details of your solution.

Table representing $K_s()$

input	output	input	output
000	110	100	011
001	111	101	010
010	101	110	000
011	100	111	001

Encrypted message = 010 011 010 011

i	1	2	3
c(i)	011	110	110
r(i)	010	011	110
Ks(c(i))			
m(i)			

Review of CBC

2) DECRYPTION

Using your answer in Item#1 (information to send), decrypt the message using CBC. Fill-up the table with the details of your solution.

Table representing $K_s()$

input	output	input	output
000	110	100	011
001	111	101	010
010	101	110	000
011	100	111	001

Encrypted message = 010 011 010 011

i	1	2	3
c(i)	011	110	110
r(i)	010	011	110
Ks(c(i))	100		
m(i)	110		

Compute for m(1).

Ks(011) = 100

Ks(011) ⊕ 010 = 100 ⊕ 010 = 110

Review of CBC

2) DECRYPTION

Using your answer in Item#1 (information to send), decrypt the message using CBC. Fill-up the table with the details of your solution.

Table representing $K_s()$

input	output	input	output
000	110	100	011
001	111	101	010
010	101	110	000
011	100	111	001

Encrypted message = 010 011 010 011

i	1	2	3
c(i)	011	110	110
r(i)	010	011	110
Ks(c(i))	100	101	100
m(i)	110	110	110

Compute for m(2) and m(3) following a similar strategy.

Cryptography

RSA with

Cipher Block Chaining

- Combination of Public key and Symmetric key cryptography
- Avoid generating the same ciphertext for identical plaintext blocks
- Avoid sending twice the number of ciphertext bits by applying a formula to calculate the sequence of random numbers automatically, except for the very first random number.
- No need for any mini-table mapping of plaintext to ciphertext.

RSA with Cipher Block Chaining

Encryption Example

Plaintext message

m = 'AAA\n'

Server's PUBLIC_KEY: (3, 25777)

Initial NONCE: 1234

RSA_CBA Encryption

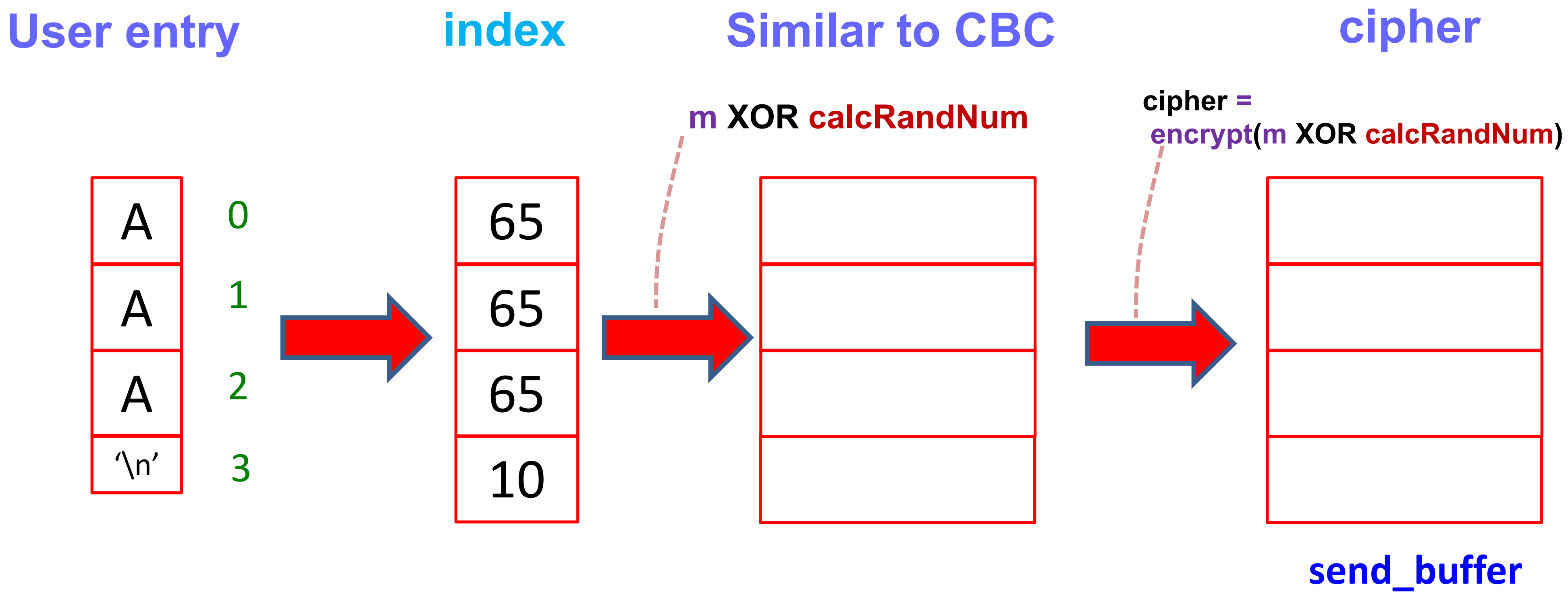
CLIENT

Encryption is performed on the index number of the character XORed with a random number.

ENCRYPTION: $c = m^{e} \bmod n$

NONCE: 1234

PUBLIC_KEY: (3, 25777)



RSA_CBA Encryption

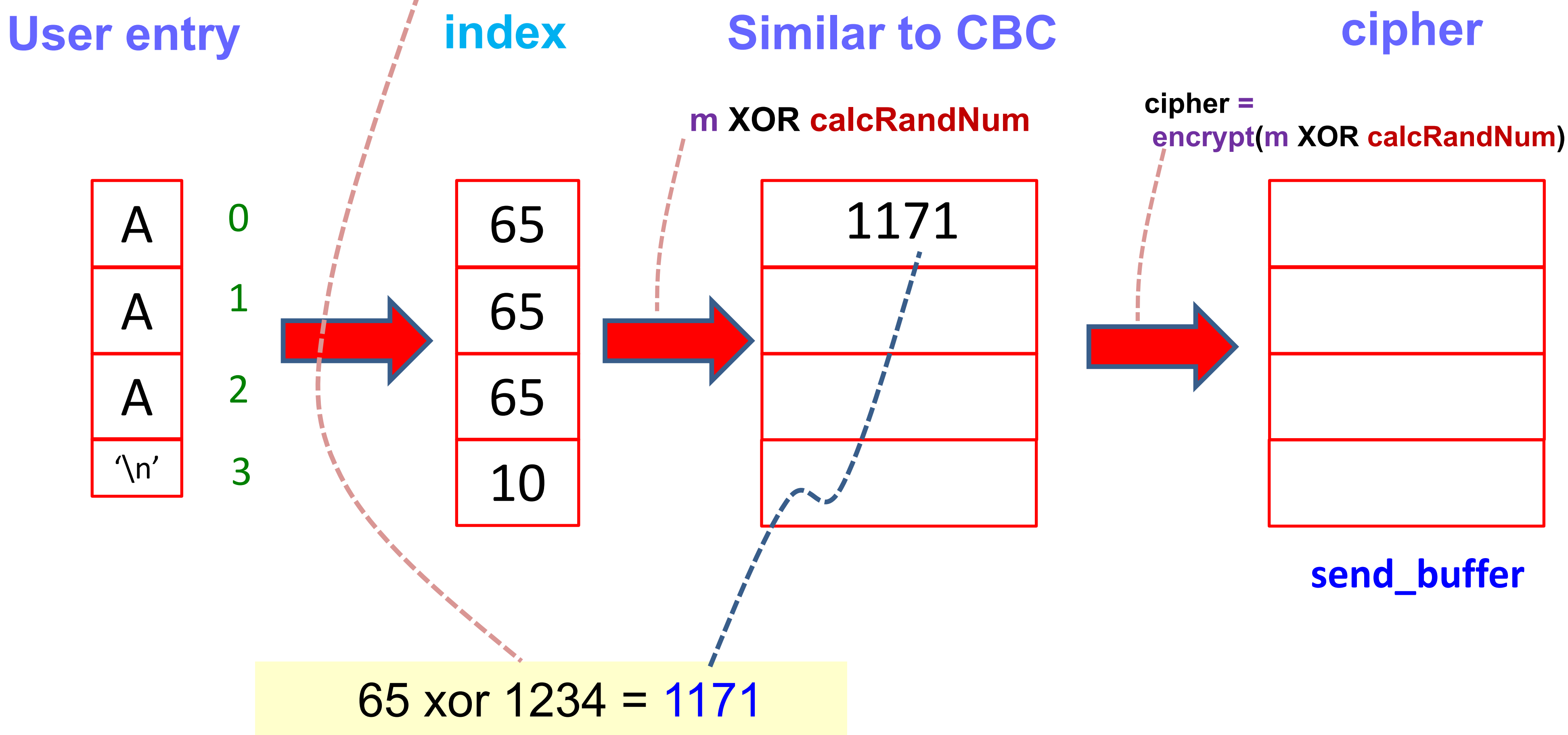
CLIENT

Encryption is performed on the index number of the character XORed with a random number.

ENCRYPTION: $c = m^{e} \bmod n$

NONCE: 1234

PUBLIC_KEY: (3, 25777)



RSA_CBA Encryption

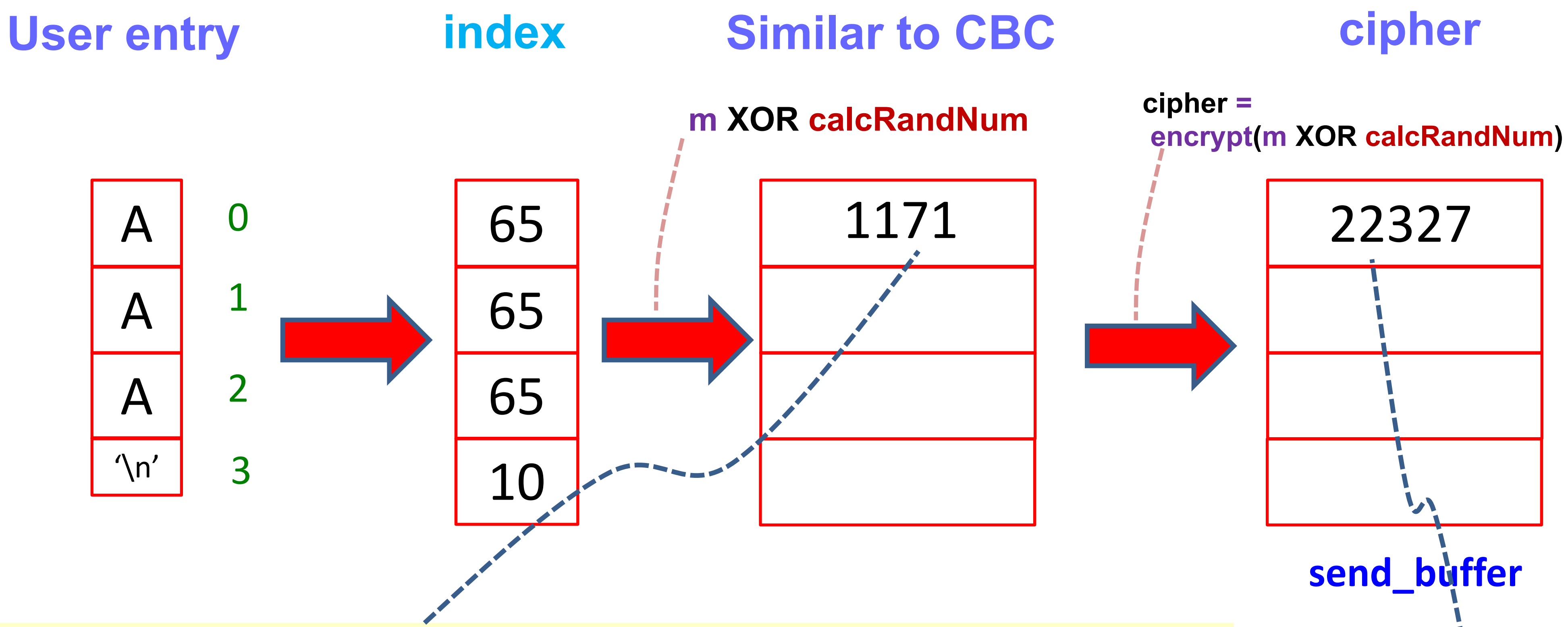
CLIENT

Encryption is performed on the index number of the character XORed with a random number.

ENCRYPTION: $c = m^{e} \bmod n$

NONCE: 1234

PUBLIC_KEY: (3, 25777)



encrypt(1171) using the PUBLIC_KEY: (3, 25777)

This is implemented by calling repeatSquare(1171, 3, 25777) = 22327

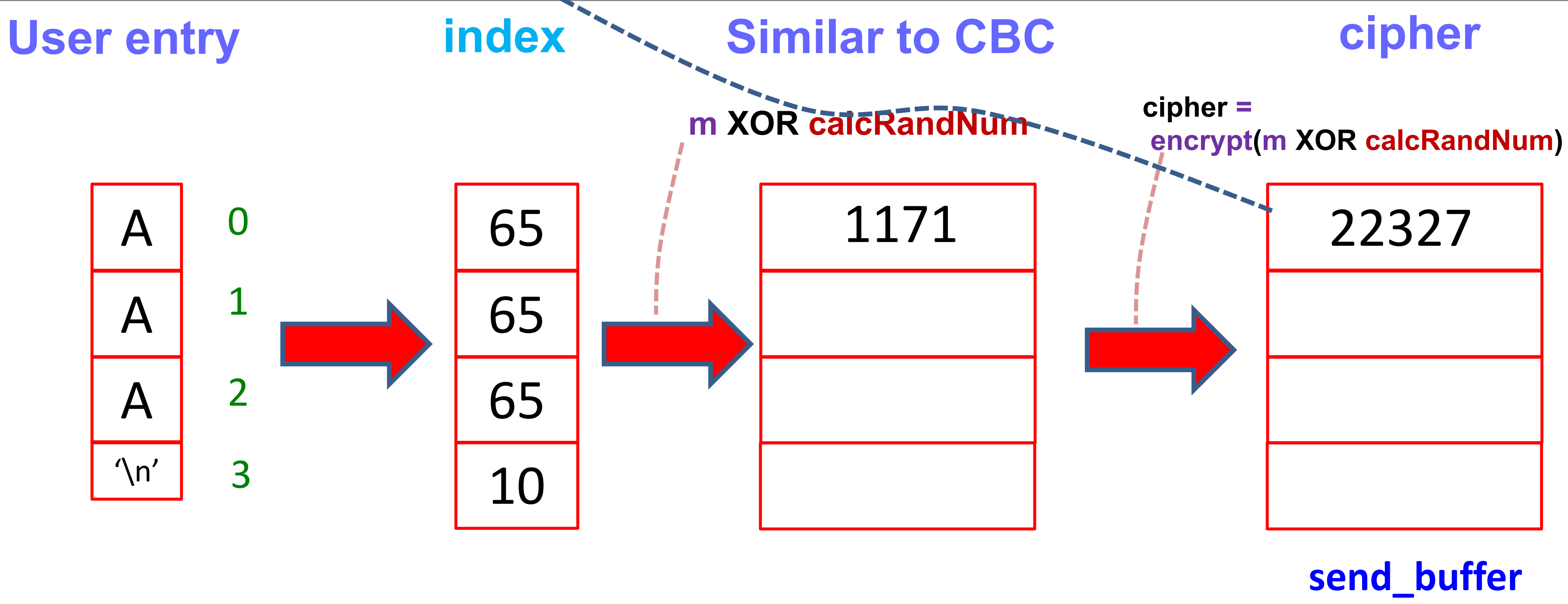
RSA_CBA Encryption

CLIENT

Encryption is performed on the index number of the character XORed with a random number.

ENCRYPTION: $c = m^{e} \bmod n$

NONCE: 22327 PUBLIC_KEY: (3, 25777)



NONCE is updated to 22327 (the most recent cipher block computed).

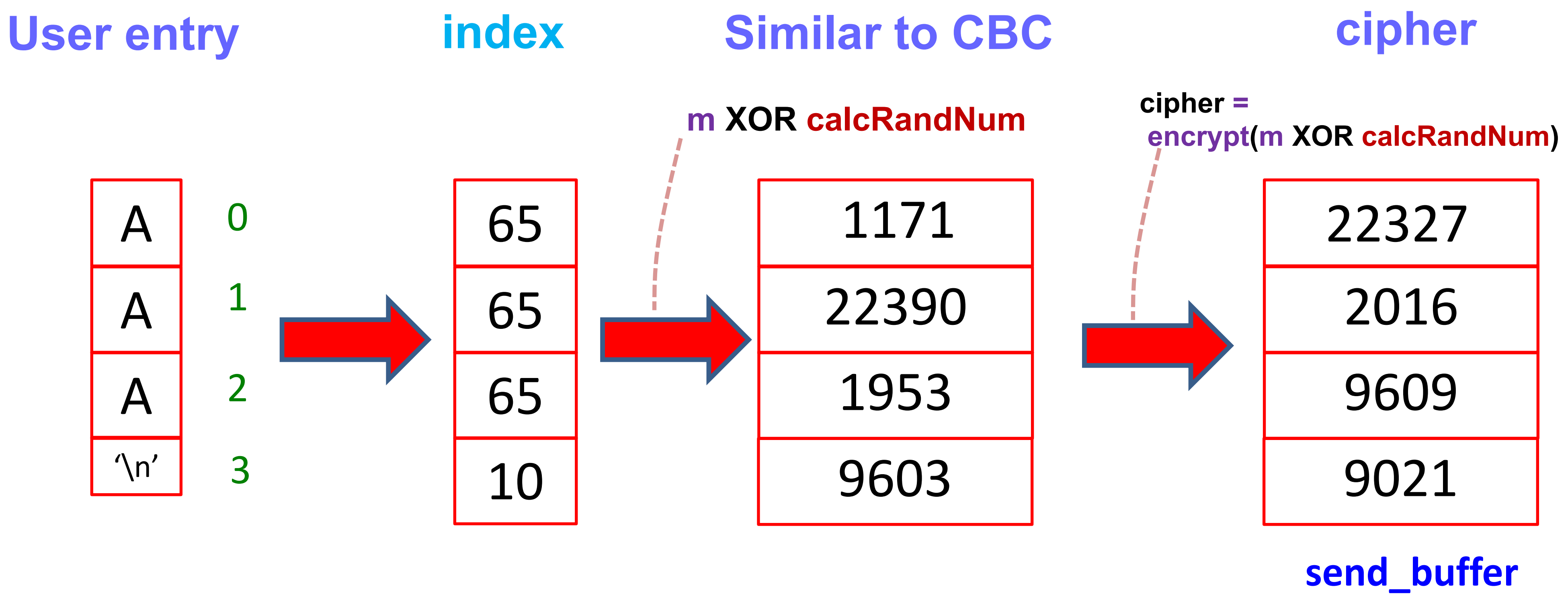
RSA_CBA Encryption

CLIENT

Encryption is performed on the index number of the character XORed with a random number.

ENCRYPTION: $c = m^e \bmod n$

NONCE: 22327 PUBLIC_KEY: (3, 25777)



Continuing with the same sequence of operations, we get the complete ciphertext.

RSA_CBA Encryption

CLIENT

Encryption is performed on the index number of the character XORed with a random number.

ENCRYPTION: $c = m^{e} \bmod n$

NONCE: 22327

PUBLIC_KEY: (3, 25777)

User entry		index	Similar to CBC	cipher
			m XOR calcRandNum	cipher = encrypt(m XOR calcRandNum)
				22327
A	0	65	1171	2016
A	1	65	22390	9609
A	2	65	1953	9021
'\n'	3	10	9603	\r
				\n

The client then sends the following ciphertext concatenated with some delimeters to the server.

send_buffer

RSA with Cipher Block Chaining

Decryption Example

Server's PRIVATE_KEY: (16971, 25777)

Cipher

Initial NONCE: 1234

c = 22327 2016 9606 9021 \r\n

RSA_CBA Decryption

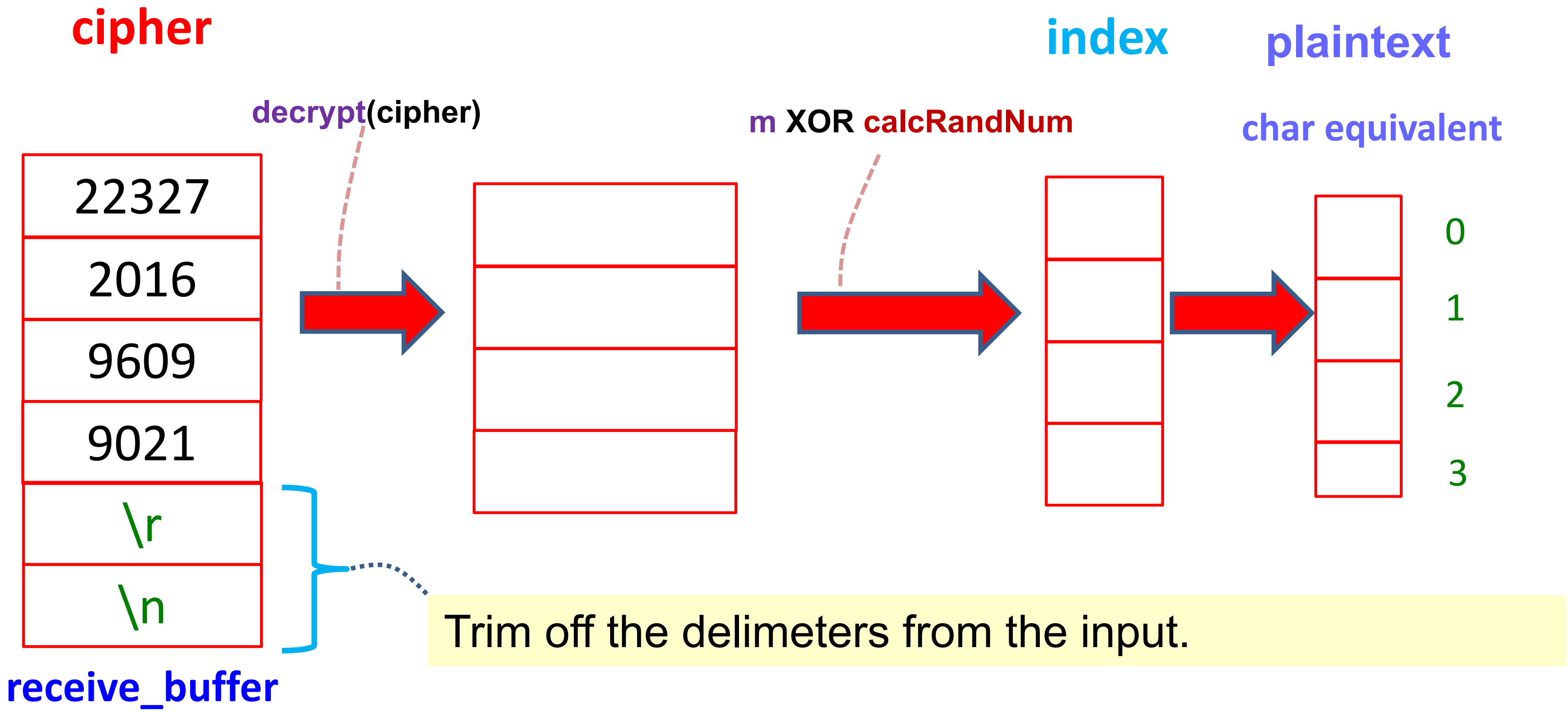
SERVER

Decryption is performed on the received cipher, then the result is XORed with a random number.

DECRYPTION: $m = c^{d} \bmod n$

NONCE: 1234

PRIVATE_KEY: (16971, 25777)



RSA_CBA Decryption

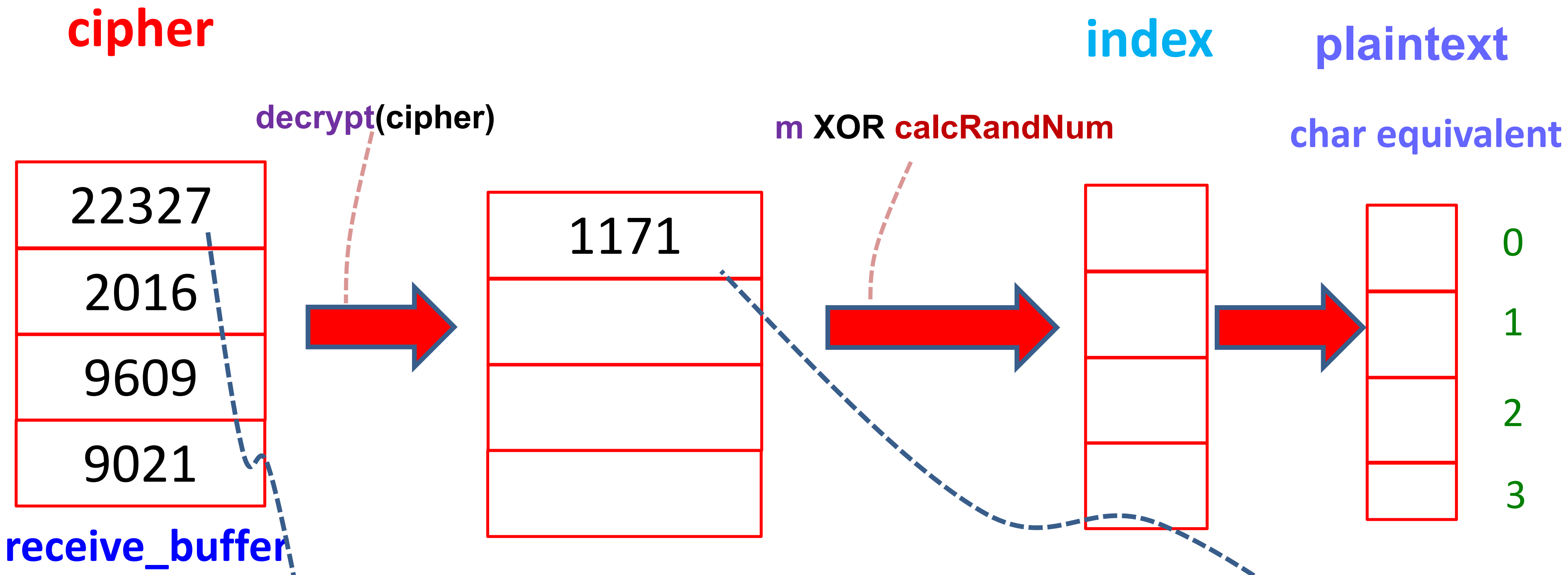
SERVER

Decryption is performed on the received cipher, then the result is XORed with a random number.

DECRYPTION: $m = c^{d} \bmod n$

NONCE: 1234

PRIVATE_KEY: (16971, 25777)



Decrypt(22327) using the Server's PRIVATE_KEY: (16971, 25777)

This is implemented by calling repeatSquare(22327, 16971, 25777) = 1171

RSA_CBA Decryption

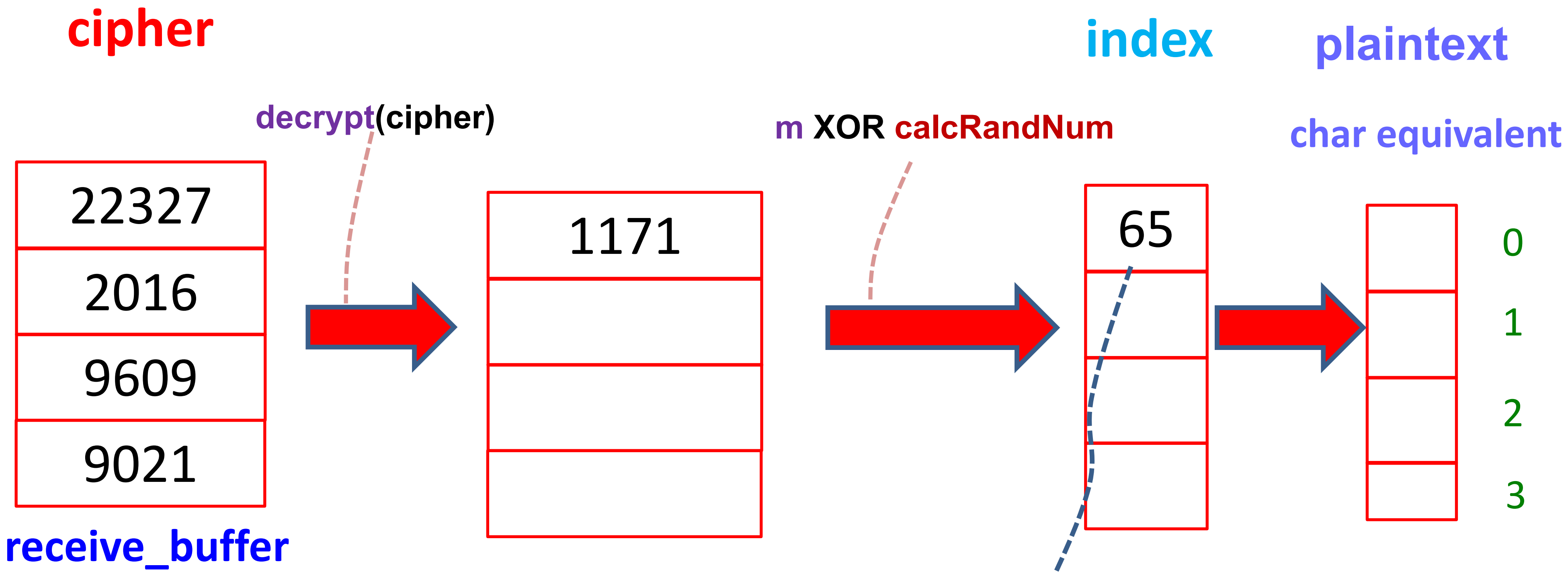
SERVER

Decryption is performed on the received cipher, then the result is XORed with a random number.

DECRYPTION: $m = c^{d} \bmod n$

NONCE: 1234

PRIVATE_KEY: (16971, 25777)



1171 xor NONCE = 1171 xor 1234 = 65

RSA_CBA Decryption

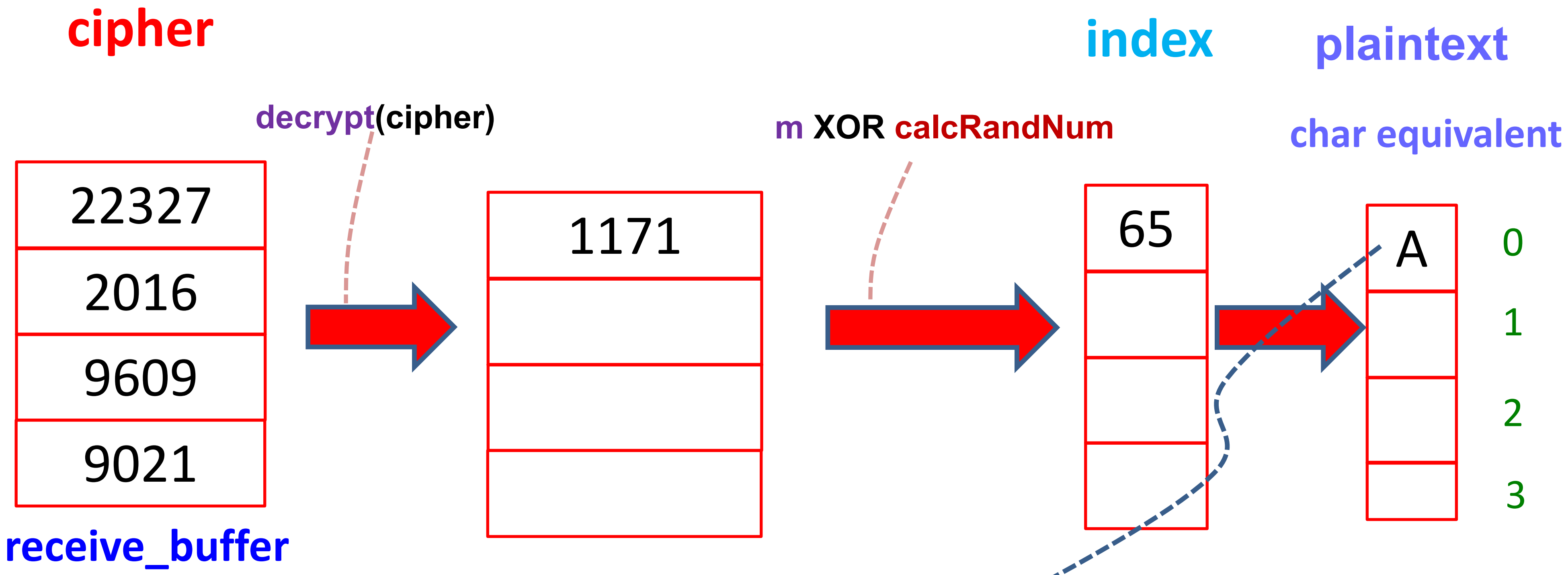
SERVER

Decryption is performed on the received cipher, then the result is XORed with a random number.

DECRYPTION: $m = c^d \bmod n$

NONCE: 1234

PRIVATE_KEY: (16971, 25777)



65 is the ASCII code for the letter 'A'.

RSA_CBA Decryption

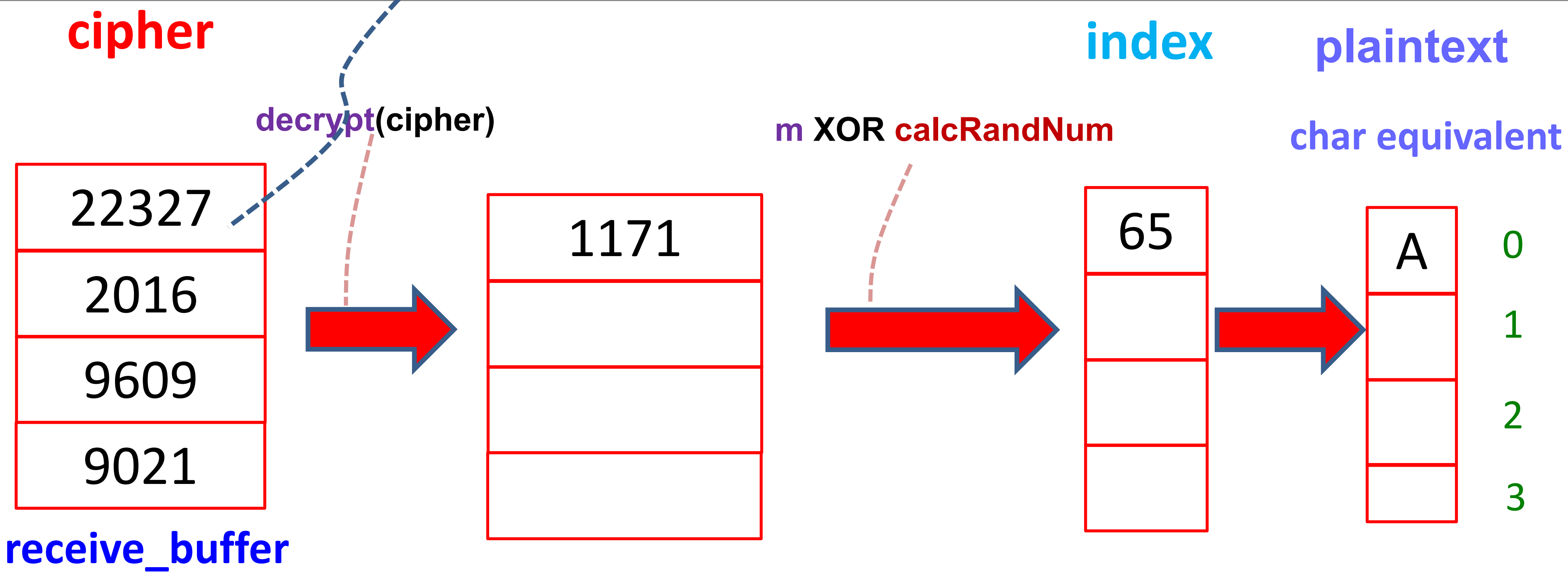
SERVER

Decryption is performed on the received cipher, then the result is XORed with a random number.

DECRYPTION: $m = c^{d} \bmod n$

NONCE: 22327

PRIVATE_KEY: (16971, 25777)



NONCE is then set to 22327 (the previous cipher block).

This will ensure that any repeating characters in the input will not generate any repeating patterns in the ciphertext.

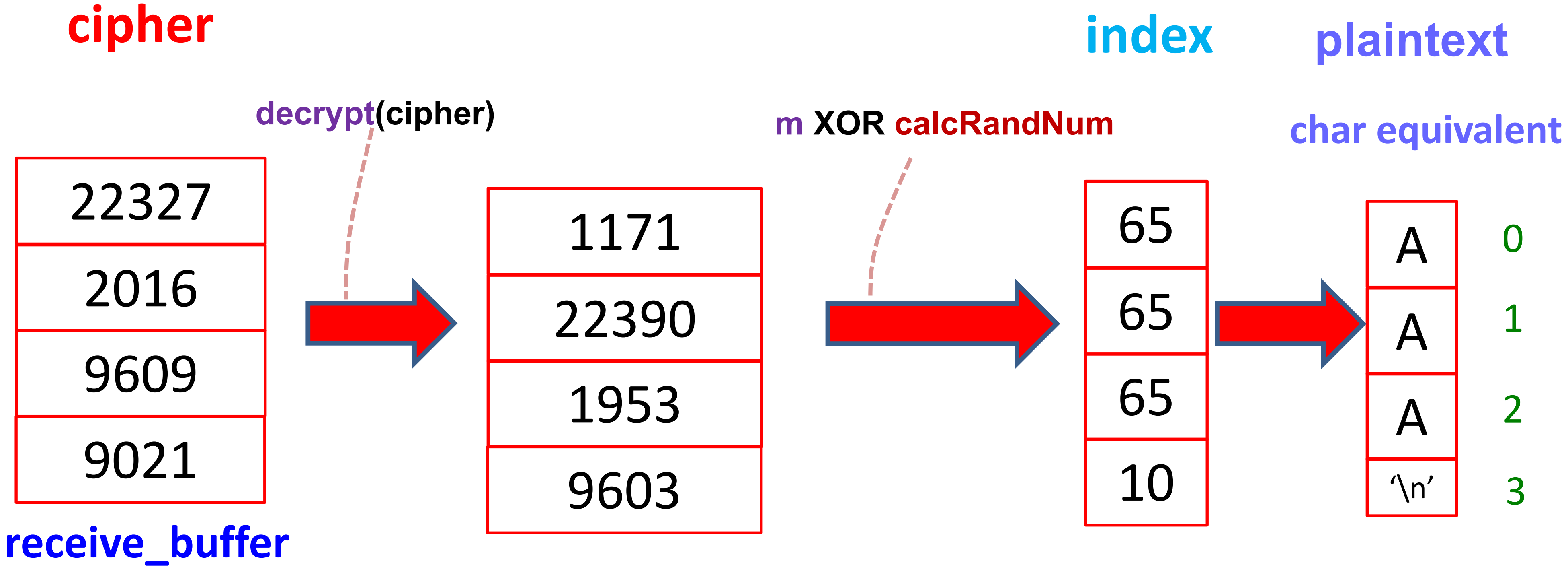
RSA_CBA Decryption

SERVER

Decryption is performed on the received cipher, then the result is XORed with a random number.

DECRYPTION: $m = c^{d} \bmod n$

NONCE: 1234 PRIVATE_KEY: (16971, 25777)



Continue with the decryption process until all plaintext characters have been extracted.

Checklist

<table>
<tr><td colspan="5">Checklist (Include this in your submission)
Please accomplish the following check list in order to allow for accurate marking of your assignment.</td></tr>
<tr><td></td><td>Item</td><td colspan="2">your assignment details</td><td>Comments</td></tr>
<tr><td>1</td><td>Names and ID numbers of Group Members</td><td colspan="2"></td><td>(Maximum of 3 members in a group)</td></tr>
<tr><td>2</td><td>Operating System(s) used for testing your client and server codes</td><td colspan="2">Windows 10</td><td></td></tr>
<tr><td>3</td><td>Compiler used</td><td colspan="2">g++ 11.2.0</td><td>g++ 11.2.0 is required</td></tr>
<tr><td>4</td><td>IDE used</td><td colspan="3"></td></tr>
<tr><td rowspan="3">5</td><td rowspan="3">Required Functionalities</td><td>Sending encrypted RSA public key to the client, and decryption of the public key by the client</td><td>full/partial/none</td><td rowspan="3">Indicate ‘full’, if you have completed the implementation of the required command, ‘partial’, if you are only submitting a partial implementation, or ‘none’, if not accomplished.</td></tr>
<tr><td>Sending encrypted nonce to the server, and decryption of nonce by client</td><td>full/partial/none</td></tr>
<tr><td>RSA with Cipher Block Chaining</td><td>full/partial/none</td></tr>
<tr><td rowspan="4">6</td><td>Snap shots of sample interactions, encryption and decryption: one for each required test input.</td><td colspan="3">Indicate ‘full’, if your codes can encrypt the following test inputs and then decrypt them correctly, snap shots must be included; ‘partial’, if your codes can only solve them partially (not all characters can be encrypted, then decrypted back to their original form), snap shots must be included; write ‘none’, if they are not solvable by your codes.</td></tr>
<tr><td>Input containing all letters of the alphabet</td><td>Test Input #1:
the quick brown fox jumps over the lazy dog</td><td>full/partial/none</td><td>Must include snap shots</td></tr>
<tr><td>Input with repeating characters</td><td>Test Input #2:
AAA</td><td>full/partial/none</td><td>The encrypted message must not exhibit any repeating pattern as cipher block chaining addresses this problem. Show snapshots.</td></tr>
<tr><td>Input with repeating characters</td><td>Test Input #3:
555</td><td>full/partial/none</td><td>The encrypted message must not exhibit any repeating pattern as cipher block chaining addresses this problem. Show snapshots.</td></tr>
</table>

Checklist

Optional

7 Extra work done (Max. bonus of 2 marks)	Program uses the Boost library's big number facility. Very large public key and private keys are being used.	Yes/No
	Include instructions on how to install the big number library you used and the compilation instructions.	
	Write CA's large Public key:	
	Write CA's large Private key:	
	Write Server's large Public key:	
	Write Server's large Private key:	

Other useful notes

Reading characters from stdin

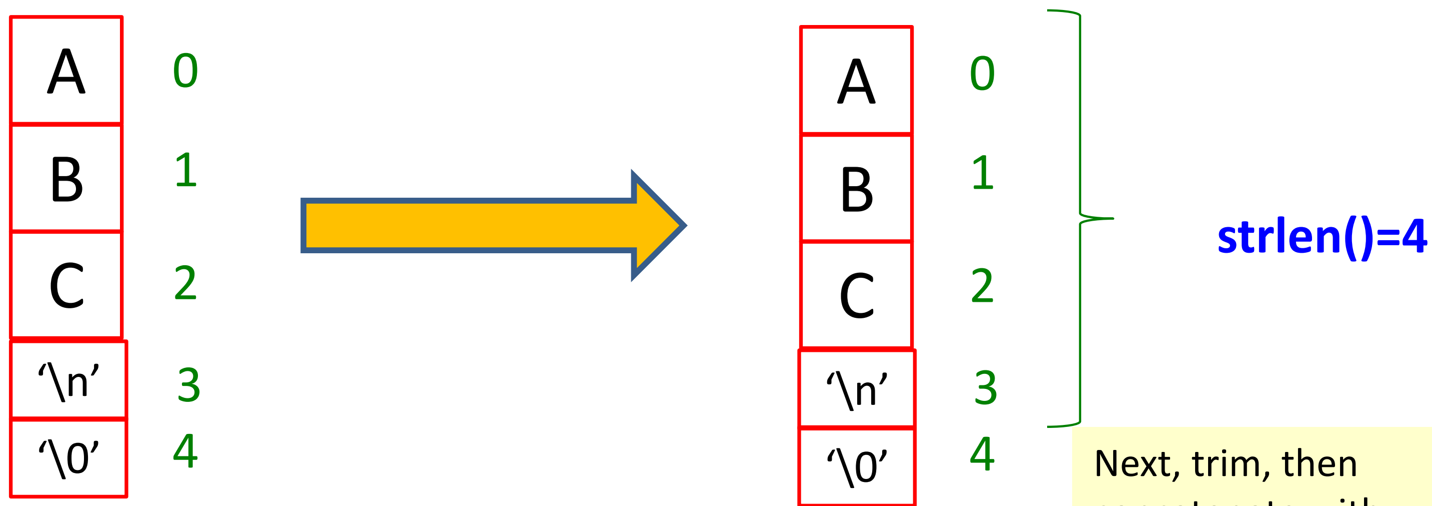
get string from stream

CLIENT

char* fgets(char* send_buffer, int num, FILE *stream)

- **reads characters from stream until (num-1) characters have been read or until it encounters:**
  - a **new line character** (copied into send_buffer)
  - **a NULL-termination character ('\0') is automatically appended**
  - if an error occurs, a NULL pointer is returned

strlen() – counts the number of characters excluding the NULL-character



User entry from keyboard **send_buffer**

Next, trim, then concatenate with \r\n

String Tokenizer

Example

```c
/* strtok example */
#include <stdio.h>
#include <string.h>

int main ()
{
  char str[] ="- This, a sample string.";
  char * pch;
  printf ("Splitting string \"%s\" into tokens:\n",str);
  pch = strtok (str," ,.-");
  while (pch != NULL)
  {
    printf ("%s\n",pch);
    pch = strtok (NULL, " ,.-");
  }
  return 0;
}
```

The End

Advanced Reading

- http://www.di-mgt.com.au/rsa_alg.html#KALI93
- The Handbook of Applied Cryptography
  - http://cacr.uwaterloo.ca/hac/