[CS304] Introduction to Cryptography and Network Security

Course Instructor: Dr. Dibyendu Roy
Scribed by: Raghav Agiwal (202151124)

Winter 2023-2024
Lecture (Week 9)

1 Public Key Cryptography

Public-key cryptography, also known as asymmetric cryptography, involves using pairs of related keys to encrypt and decrypt data. Each key pair comprises a public key and a corresponding private key.

How It Works

1. **Key Generation**: A user generates a key pair using cryptographic algorithms. This process involves creating a public key K_{public} and a private key K_{private}. These keys are mathematically related but computationally infeasible to derive one from the other.

2. **Public Key Distribution**: The public key is made freely available to anyone who wants to send encrypted messages or verify the sender's identity. It can be shared openly without compromising security.

3. **Private Key Security**: The private key, on the other hand, is kept secret and known only to the owner. It is used for decrypting messages encrypted with the corresponding public key and for digitally signing messages.

4. **Encryption**: If someone wants to send an encrypted message to the owner of a public key, they use that public key to encrypt the message. Once encrypted, only the corresponding private key can decrypt it.

5. **Digital Signatures**: Public key cryptography is also used for creating digital signatures. A digital signature is created by encrypting a message or its hash value with the sender's private key. The recipient can then verify the signature using the sender's public key, ensuring the message hasn't been altered and originated from the claimed sender.

6. **Security Strength**: Public key cryptography relies on mathematical problems, such as factoring large numbers or calculating discrete logarithms, which are computationally difficult to solve. The security of the system depends on the complexity of these mathematical problems.

1.1 Diffie-Hellman Key Exchange Algorithm

The Diffie-Hellman key exchange algorithm is a method used to securely establish a shared secret key between two parties over an insecure communication channel. It was developed independently by Whitfield Diffie and Martin Hellman in 1976 and is one of the earliest practical examples of public-key cryptography.

How It Works

1. **Parameter Selection**: Both parties agree on two public parameters: a large prime number p and a primitive root modulo p, denoted as g. These parameters are typically chosen beforehand and shared publicly.

2. **Key Generation**:

   - Each party, let's call them Alice and Bob, generates their own private key:
     - Alice selects a secret integer a (her private key).
     - Bob selects a secret integer b (his private key).
   - Both Alice and Bob compute their corresponding public keys using the agreed-upon parameters:
     - Alice computes $A = g^a \mod p$ and sends A to Bob.
     - Bob computes $B = g^b \mod p$ and sends B to Alice.

3. **Key Exchange**:

   - Alice receives Bob's public key B and computes the shared secret key:

   $$\text{Shared Secret Key} = B^a \mod p$$

   - Bob receives Alice's public key A and computes the shared secret key:

   $$\text{Shared Secret Key} = A^b \mod p$$

4. **Agreement on Shared Secret**: Both Alice and Bob now have the same shared secret key, which can be used for symmetric encryption or other cryptographic purposes.

5. **Security**: Even though the public keys A and B are exchanged over an insecure channel, an eavesdropper cannot easily compute the shared secret key because it requires knowledge of either a or b, which are kept private by Alice and Bob, respectively. The security of the Diffie-Hellman key exchange relies on the computational difficulty of the discrete logarithm problem.

1.2 Diffie and Hellman Key Exchange Algorithm

Suppose we have a symmetric key encryption setup, meaning Alice and Bob share the same key, K.

Alice	Bob
K	K

$$C = \text{Enc}(M, K) \quad M = \text{Dec}(C, K)$$

Alice can encrypt a message and send it to Bob, who can then decrypt the ciphertext. The challenge lies in ensuring that Alice and Bob share the same secret key. Otherwise, decryption will not yield the correct plaintext. Historically, before 1976, the only way to share this secret key was through a physical meeting. In 1976, Diffie and Hellman proposed a key exchange mechanism, marking the beginning of public key cryptography. Their findings were published in IEEE Transactions on Information Theory.

Let's first revisit the concept of a group and a cyclic group before delving further into the Diffie-Hellman Key Exchange Algorithm. A set G along with a binary operation, denoted as $(G, *)$, is termed a group if it satisfies the following properties:

1. Closure: If $a, b \in G$, then $a * b \in G$. 2. Associativity: $a * (b * c) = (a * b) * c$ for all $a, b, c \in G$. 3. Identity Element: There exists an element $e \in G$ (identity element) such that $a * e = a = e * a$ for all $a \in G$. 4. Inverse Element: For each $a \in G$, there exists an element $a^{-1} \in G$ (inverse of a) such that $a * a^{-1} = e = a^{-1} * a$ for all $a \in G$.

A cyclic group $(G, *)$ is one where every element in G can be generated using a single element g in G. Formally, for every $a \in G$, there exists $g \in G$ such that $a = g^i$, where $i \in \mathbb{Z}$. Here, g is known as the generator of G and is denoted as $G = \langle g \rangle$.

According to the Diffie-Hellman Key Exchange Algorithm, Alice and Bob agree on a cyclic group $(G, *)$ over a public channel. This means they both agree to use the cyclic group $(G, *)$ with generator g.

Alice	Bob
$k_a = g^a$	$k_b = g^b$
$0 < a < n$	$0 < b < n$
$k_a = g^a$	$k_b = g^b$
$k_b = g^b$	$k_a = g^a$
$(G, *) = \langle g \rangle, \lvert G \rvert = n$	

The part written in green in the figure above is public, while the part written in red is secret to the respective person. Alice selects a number a such that $0 < a < n$ and keeps it secret. Similarly, Bob selects a number b such that $0 < b < n$ and keeps it secret. Alice computes $k_a = g^a$ and sends it to Bob, and Bob computes $k_b = g^b$ and sends it to Alice over the public channel. Thus, Alice makes k_a public and Bob makes k_b public.

Now, both Alice and Bob have the following data:

Alice	Bob
a	b
g	g
$k_b = g^b$	$k_a = g^a$

Since g^a and g^b belong to the same cyclic group, given g^b, if we have a, we can compute $(g^b)^a$. Therefore, Alice computes $(g^b)^a$ and Bob computes $(g^a)^b$. Hence, Alice now has g^{ba} and Bob has g^{ab}. Since a and b are integers, we have:

$$a \cdot b = b \cdot a \Rightarrow g^{ba} = g^{ab}$$

Hence, Alice and Bob have the same element g^{ab}, which they can use as their secret key for communication using symmetric key encryption algorithms. The key generated by both Alice and Bob, i.e., g^{ab}, is called the shared secret key.

We observe that Alice and Bob are exchanging data over the public channel and are able to establish a secret key. Since a and b are only known to Alice and Bob respectively (i.e., they are not shared publicly), they are known as secret keys. However, $k_a = g^a$ and $k_b = g^b$ are shared by Alice and Bob with each other over the public channel. Hence, they are made public and are known as public keys. Both parties have two keys: one public key and one secret key.

Note that there may be some techniques to compute $a(b)$ from $g^a(g^b)$, but since Alice (Bob) is

not sharing $a(b)$, we assume it to be the secret key for Alice (Bob).

Now, if we don't know Alice's secret key a, even if we know g^a and g^b, we will not be able to compute g^{ab}, which is the key used for communication between Alice and Bob. That means, without knowing Alice's or Bob's secret key, we will not get the shared secret key used for communication between them.
Now, we have g^x as the public key and x as the secret key. If we have a very good cyclic group $(G, *)$, based on the properties of the group, finding x from g^x is computationally difficult. It is possible theoretically, but the amount of time it requires is exponential.

This hard problem is known as the Discrete Log Problem. Since finding x from g^x for a good group is computationally hard, and we are using this group to establish the shared secret key, the key establishment mechanism will be secure. Therefore, the security of the Diffie-Hellman Key Exchange Algorithm relies on the fact that the Discrete Log Problem is hard for certain groups.

One straightforward approach to determining x from g^x, given knowledge of G, g, and g^x, is to compute g^i for $1 \leq i \leq n-1$ and return i if we find $g^i = g^x$.

A brute force algorithm to find x from g^x would iterate through $i = 2$ to $N - 1$, checking if g^i is equal to g^x. If a match is found, the value of i is assigned to t, and the loop breaks. However, the computational complexity of this approach is proportional to the size of the set G, denoted as $|G|$. For instance, if our group contains $2^{512} - 1$ elements, obtaining x using this method would be practically infeasible. The loop, in this case, would never terminate within a reasonable amount of time.

To ensure the Discrete Log Problem remains challenging on a group $(G, *)$, certain properties must be observed:

1. The size of the set G, $|G|$, should be sufficiently large.

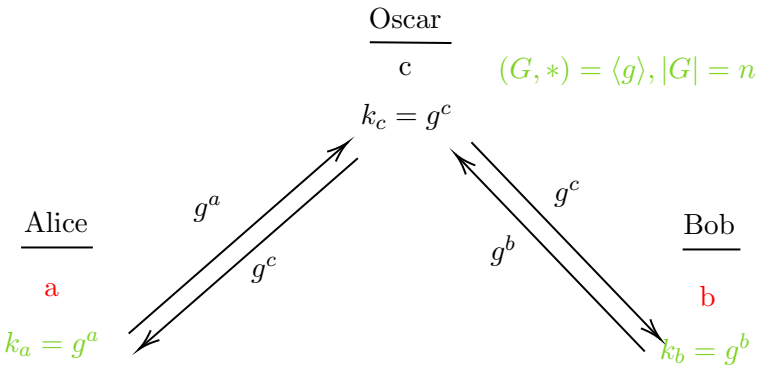
2. The group operation $*$ must be carefully selected. Even with a large group size, the properties of the group operation play a crucial role. For instance, consider the cyclic group $G = (\mathbb{Z}_p, +_p)$, where p is a large prime number and $G = \langle g \rangle$. Computing g^i in this group simplifies to $gi = i \cdot g$. Therefore, if we have the generator g of G, computing g^i becomes straightforward. Conversely, if we know g^i, finding i becomes trivial without exhaustive search. Multiplying both sides by g^{-1} yields:
$$i = g^{-1} \cdot g^i \mod p$$
We can compute g^{-1} in polynomial time using the Extended Euclidean Algorithm, which holds true because $\gcd(i, p) = 1$ since p is prime. Hence, $+_p$ is not a suitable group operation.

1.3 Man in the Middle Attack on Diffie-Hellman Key Exchange Algorithm

A Man-in-the-Middle (MITM) attack occurs when an intermediary intercepts and potentially alters communication between two parties without their knowledge. Imagine you're sending a letter to a friend through a post office. You seal the letter in an envelope and entrust it to the post office for delivery. However, if the post office itself is compromised, someone could tamper with your letter or even read its contents before delivering it to your friend. Similarly, in digital communication like messaging apps such as WhatsApp, data exchanged with the server can be intercepted and manipulated if the server is compromised.

In essence, a MITM attack involves a third party secretly eavesdropping on communications between two parties and potentially altering the messages in transit. This poses a significant security risk as the intercepted data could be manipulated or leaked without the knowledge of the communicating parties.

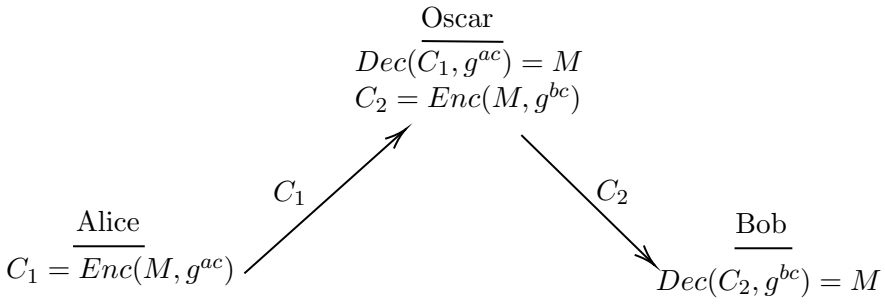
Let's try to understand the Man in the Middle Attack on the Diffie-Hellman Key Exchange Algorithm.



In this scenario, Oscar has the capability to intercept the communication between Alice and Bob. Let's say Alice sends a message to Bob, but Oscar intercepts it and replaces it with a different message. For example, if Alice sends g^a to Bob, Oscar can calculate g^c since g and G are public. Oscar then sends g^c to Bob instead of g^a. Since Bob lacks a means to verify the message's origin, he assumes it's from Alice, believing it to be her public key. However, in reality, Bob has received g^c instead of g^a.

When Bob shares his public key g^b with Alice, Oscar repeats the interception, sending g^c to Alice. Once again, Alice cannot verify the source of the message and accepts it.

Using the intercepted data, Alice can calculate $(g^c)^a = g^{ac}$, Bob can calculate $(g^c)^b = g^{bc}$, and Oscar can calculate $(g^a)^c = g^{ac}$ and $(g^b)^c = g^{bc}$ respectively. Consequently, Oscar possesses two shared secret keys: one identical to Alice's and the other to Bob's. Unaware of Oscar's interference, Alice and Bob continue their communication. Suppose Alice sends a message to Bob under the assumption that their communication remains secure.



Alice encrypts the message using g^{ac} and sends it. Oscar intercepts this message and decrypts it using g^{ac}, gaining access to the original message sent by Alice. Oscar then encrypts this message using g^{bc} and sends it to Bob. Upon receiving the message, Bob decrypts it using g^{bc} and perceives it as the original message sent by Alice. Unbeknownst to both Alice and Bob, Oscar has intercepted

and manipulated their communication. This scenario represents a Man-in-the-Middle Attack on the Diffie-Hellman Key Exchange Algorithm.

In messaging platforms like WhatsApp or Telegram, Diffie-Hellman Key Exchange is utilized. However, this exchange occurs through the intermediary service provider (e.g., WhatsApp or Telegram), introducing potential vulnerabilities. While these service providers can manipulate the exchanged data if they choose, they typically implement security measures to safeguard the communication.

To compute the shared secret key g^{ab}, it is necessary to compute g^a and g^b in a large group G, with both a and b being sufficiently large. Since selecting any pseudo-random number between 1 and $(n-1)$ is highly likely to fall within the middle range, if $|G| = 2^{512}$, then a and b would typically be around 256 bits. The challenge lies in efficiently computing g^a and g^b. Traditional methods such as using loops and direct multiplication by g in each iteration are impractical. Instead, the Square and Multiply Algorithm is employed. In this algorithm, to compute x^c, where c is represented in binary as $c_{l-1} \dots c_1 c_0$, a more efficient approach is utilized.Then,
$c = \Sigma_{i=0}^{l-1} c_i \cdot 2^i$

$x^c = x^{\Sigma_{i=0}^{l-1} c_i \cdot 2^i} = \Pi_{i=0}^{l-1} x^{c_i \cdot 2^i}$

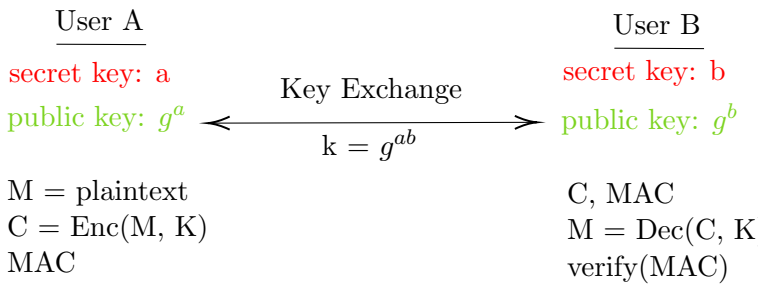
$x^c = x^{c_0 \cdot 2^0} \dots x^{c_{l-1} \cdot 2^{l-1}}$

These are just $log(c)$ multiplications, hence, we can compute x^c in logarithmic time of c.

Square and Multiply Algorithm to find x^c **Input:** x and c
Z = 1;
$i = l - 1$ to $N0$ $Z = Z^2$;
$c_i == 1$ Z = Z * x;
return Z;

Let's illustrate an example and compute 3^5. Here, $x = 3$ and $c = 5$, represented as 101_2 in binary. Initially, $Z = 1$ and $i = 2$ (since $l = 3$). In the first iteration of the loop, $Z = Z^2 = 1$, since $c_2 = 1$, $Z = Z * x = 3$. In the second iteration, $Z = 3$, $i = 1$, $Z = Z^2 = 9$, as $c_1 = 0$, Z remains unchanged. In the last iteration, $Z = 9$, $i = 0$, $Z = Z^2 = 81$, since $c_0 = 1$, $Z = Z * x = 81 * 3 = 243$. Thus, $Z = 243 = 3^5$.

Using the Square and Multiply Algorithm, g^a and g^b can be efficiently computed in $O(\log(a))$ and $O(\log(b))$ time complexity, respectively.



In the event of a Man-in-the-Middle Attack, it's essential to establish a communication setup that incorporates Message Authentication Codes (MAC). A MAC serves the dual purpose of verifying

the message source and ensuring message integrity. By implementing MAC, we authenticate the source of the message, thereby mitigating the risk of a Man-in-the-Middle Attack.

Explanation:

A Man-in-the-Middle Attack occurs when an unauthorized third party intercepts and potentially alters communication between two parties. By introducing a Message Authentication Code (MAC) into the communication setup, we add an extra layer of security. The MAC acts as a cryptographic checksum, ensuring that the message originates from the expected source and hasn't been tampered with during transmission. Consequently, the inclusion of MAC prevents unauthorized intermediaries from intercepting and manipulating the communication undetected.

1.4 RSA (Rivest Shamir Adleman) Encryption

RSA (Rivest-Shamir-Adleman) Encryption is a fundamental public-key encryption algorithm. Before delving into RSA encryption, it's crucial to review some foundational concepts:

- The Euler's Totient Function $\phi(n)$ determines the count of integers less than n that are coprime to n. In other words, it represents the number of x such that $\gcd(x, n) = 1$, where $1 \leq x \leq n - 1$.

- Let's consider a set $S = \{x \mod m\}$, where $|S| = m$. All elements in set S are unique, typically ranging from 0 to $m - 1$. Assume an integer a such that $\gcd(a, m) = 1$. Now, suppose there exists another set S_1 defined as:

  $$S_1 = \{ar_1 \mod m, ar_2 \mod m, \ldots, ar_m \mod m\}$$

  Since the elements $\{r_1, r_2, \ldots, r_m\}$ are distinct, and $\gcd(a, m) = 1$, it can be inferred that $\{ar_1 \mod m, ar_2 \mod m, \ldots, ar_m \mod m\}$ will also consist of m unique elements. This can be proven using contradiction. Suppose $ar_i = ar_j$ for $r_i \neq r_j$. Therefore:

  $$ar_i \equiv ar_j \mod m$$

  As $\gcd(a, m) = 1$, there exists an integer b such that $ab \equiv 1 \mod m$ (from Bezout's Identity). This b is termed the multiplicative inverse of a, calculable using the Extended Euclidean Algorithm. Therefore, multiplying both sides of the equation by b yields:

  $$b \cdot a \cdot r_i \equiv b \cdot a \cdot r_j \mod m$$

  $$r_i \equiv r_j \mod m \quad (\because ab \equiv 1 \mod m)$$

  Hence, it contradicts the initial assumption that $r_i \neq r_j$. Thus, the elements in set S_1 are unique if and only if $\gcd(a, m) = 1$.

2 Euler's Theorem

If $\gcd(a, m) = 1$, then $a^{\phi(m)} \equiv 1 \pmod{m}$.

Let's consider a set S such that

$$S = \{x \mid \gcd(x, m) = 1\}$$

$$S = \{s_1, s_2, s_3, s_4, \ldots, s_{\phi(m)}\}$$

Now, assuming $\gcd(a, m) = 1$, let's create another set S_1 such that

$$S_1 = \{as_1, as_2, as_3, \ldots, as_{\phi(m)}\}$$

If $as_i \equiv as_j \pmod{m}$,

$$\Rightarrow s_i \equiv s_j \pmod{m}$$

Given that $\gcd(a, m) = 1$ and $b \cdot a \equiv 1 \pmod{m}$,

$$|S| = \phi(m)$$

$$|S_1| = \phi(m)$$

Since a is coprime with m and s_i is also coprime with m, there must be some correspondence between elements of S and S_1.

$$s_i \equiv as_j \pmod{m}$$

Let's now take the product on both sides:

$$\prod_{i=1}^{\phi(m)} s_i \equiv \prod_{j=1}^{\phi(m)} as_j \pmod{m}$$

$$\Rightarrow \prod_{i=1}^{\phi(m)} s_i \equiv a^{\phi(m)} \cdot \prod_{j=1}^{\phi(m)} s_j \pmod{m}$$

Since $\gcd(s_i, m) = 1$, each s_i will have a multiplicative inverse under mod m. After simplification, we get:

$$a^{\phi(m)} \equiv 1 \pmod{m}$$

3 Fermat's Theorem

If P is a prime number and P does not divide a (meaning that P is coprime to a), then

$$a^{P-1} \equiv 1 \pmod{P}$$

Using Fermat's theorem,

$$a^P \equiv a \pmod{P}$$

Note: If $P \mid a$ (where P divides a), then

$$a \equiv 0 \pmod{P}$$

$$a^P \equiv 0 \pmod{P}$$

$$a^P \equiv a \pmod{P}$$

However, Fermat's theorem will not hold when P divides a.

4 RSA Cryptosystem

Facts:

- If $\gcd(a, m) = 1$, then $a^{\phi(m)} \equiv 1 \pmod{m}$.
- If P is a prime number, then $a^{P-1} \equiv 1 \pmod{P}$.

Now, let's understand the components of RSA:

1. $n = pq$, where p and q are prime numbers.
2. Plaintext space: $\mathbb{Z}_n$, Ciphertext space: $\mathbb{Z}_n$.
3. Key space: $\{K = (n, p, q, e, d) \mid ed \equiv 1 \pmod{\phi(n)}\}$.
4. Encryption:
$$E(x, K) = c, \quad c = E(x, K) = x^e \pmod{n}$$
5. Decryption:
$$\text{Dec}(c, K) = x, \quad c = \text{Dec}(c, K) = c^d \pmod{n}$$

We know that e and d are related as:

$$ed \equiv 1 \pmod{\phi(n)} \Rightarrow ed - 1 = t \cdot \phi(n) \Rightarrow 1 = ed + t \cdot \phi(n)$$

Encryption:

$$c = x^e \pmod{n}$$

Decryption:

$$x = c^d \pmod{n}$$
$$c^d = (x^e)^d \mod n$$
$$c^d = x^{ed} \mod n$$

Now using $ed = 1 + t \cdot \phi(n)$ from above:

$$c^d = x^{1+t\cdot\phi(n)} \mod n$$

$$c^d = x \cdot x^{t\cdot\phi(n)} \mod n$$

Since p and q are primes and $n = pq$, then $\phi(n) = (p-1)(q-1)$:

$$c^d = x \cdot x^{t[(p-1)(q-1)]} \mod (pq)$$

Now let us simplify the part $x^{t[(p-1)(q-1)]} \mod (pq)$, where $x \in \mathbb{Z}_n$:
We check $x^{t[(p-1)(q-1)]} \mod p$:

$$\equiv x^{p-1} \mod p \equiv 1 \pmod{p}$$

[As $x^{p-1} \equiv 1 \pmod{p}$]
We check $x^{t[(p-1)(q-1)]} \mod q$:

$$\equiv x^{q-1} \mod q \equiv 1 \pmod{q}$$

[As $x^{q-1} \equiv 1 \pmod{q}$]

Finally, we have:

$$x^{t[(p-1)(q-1)]} \equiv 1 \pmod{p}$$
$$x^{t[(p-1)(q-1)]} \equiv 1 \pmod{q}$$
$$\Rightarrow x^{t[(p-1)(q-1)]} \equiv 1 \pmod{pq}$$

Substituting the above result:

$$c^d = x \cdot 1 \mod (pq)$$
$$c^d = x \mod (pq)$$

Hence, the decryption process is successful. Now, let's consider a scenario where Alice intends to communicate with Bob. In this scenario, two keys play pivotal roles: one is the public key, and the other is the secret key. Bob encrypts the message and sends it to Alice, who possesses both keys. While Bob is aware of the public key, he doesn't have access to the secret key.

Alice:

- $n = pq$, where p and q are large prime numbers.
- $ed \equiv 1 \pmod{\phi(n)}$
- Alice selects e.
- Public key of Alice: (n, e)
- She computes d using the extended Euclidean algorithm.
- Secret key of Alice: (p, q, d)

Bob:

- Bob chooses a message x from $\mathbb{Z}_n$.
- He knows n and e for Alice, allowing him to encrypt: $y = x^e \pmod{n}$
- Bob sends the encrypted message y to Alice.

Alice:

- Alice decrypts the message using her secret key: $x = y^d \pmod{n}$

If we are given n, how can we find p and q in polynomial time? We can iterate a loop from 2 to $\sqrt{n}$, and for each iteration, check if $n\%i$ equals zero. If it does, it means we have found a factor. Then, we verify if the factor is prime. If it is, we divide n by p to obtain q. However, finding the prime factors of a large number n is a computationally hard problem.

Note: If we can compute p and q from n, we can also compute $\phi(n)$. Since we already have e, we can find d using the extended Euclidean algorithm, thus compromising the security of RSA. Hence, RSA relies on the difficulty of the factorization problem.

5 RSA Problem :-

The RSA problem involves the challenge of deducing the plaintext x given the public key (n, e) and the ciphertext c (where $c = x^e$). If one could efficiently derive x from this information, the security of RSA would be compromised.

There's a misconception that there exists an algorithm capable of solving the RSA problem without requiring factorization. However, this is not true. It's essential to note that even if RSA can be broken, there's no guarantee that its factors can be determined. Conversely, if the factors are known, RSA can be broken, but the reverse isn't always valids.
So RSA is secure under two assumptions:
• factorization is hard
• decryption is hard
It's noteworthy that public key encryption tends to be resource-intensive due to the exponentiation operations involved in x^e and c^d. Consequently, such operations are typically avoided when possible.

Factorization Problem in RSA

The factorization problem in the context of RSA encryption involves the challenge of finding the prime factors of a large composite number n when only n is known. In RSA, the security of the encryption scheme relies on the difficulty of factoring the product of two large prime numbers.

RSA encryption relies on the principle that it's easy to multiply two large prime numbers to obtain a large composite number, but it's computationally difficult to factorize the resulting composite number back into its prime factors. Specifically, given a composite number n that is the product of two large primes p and q ($n = p \times q$), the factorization problem involves finding p and q when only n is known.

The security of RSA encryption hinges on the assumption that factoring large composite numbers into their prime factors is a computationally intensive task, especially as the size of the primes increases. If an efficient algorithm were to be developed that could quickly factorize large composite numbers, it would render RSA encryption insecure, as an adversary could derive the private key d from the public key (n, e) and decrypt encrypted messages without authorization.

Therefore, the factorization problem of RSA refers to the difficulty of factoring a large composite number n into its prime factors p and q, which forms the basis of the security of the RSA encryption algorithm.

Digital Signature Algorithms

Digital Signature Algorithms (DSA) play a crucial role in digitally signing documents and messages. When employing a DSA, only the signer is authorized to sign a document, but anyone can verify the signature. Formally, a DSA is defined as a tuple $(P, S, K, \text{Sign}, \text{V})$, where:

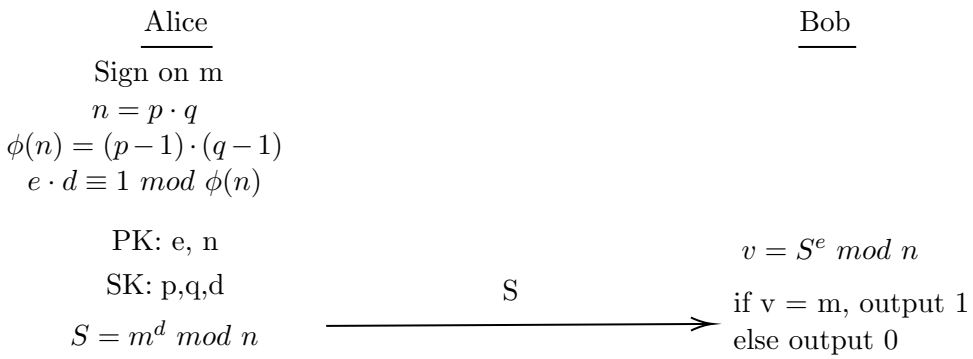
- P: Plaintext
- S: Signature Text
- K: Key Space
- Sign: Signing Algorithm

- V: Verification Algorithm

The signing algorithm, denoted as Sign(p, k), takes plaintext p and a key k from the key space to generate a signature s. On the other hand, the verification algorithm V takes plaintext p, signature s, and key k as inputs, and outputs 1 if s matches the signature generated for p with key k, and 0 otherwise.

One commonly used algorithm for digital signatures is the RSA Signature Algorithm.

5.1 RSA Signature Algorithm



Alice wishes to digitally sign a message m, and Bob needs to verify Alice's digital signature. Alice begins by generating two large prime numbers, p and q, which she will use to compute $n = p \cdot q$ and $\phi(n) = (p-1) \cdot (q-1)$. Next, Alice selects an integer e and computes d such that $e \cdot d \equiv 1 \mod \phi(n)$. Using these parameters, Alice signs the message in a way that others can verify. Specifically, Alice computes:

$$\text{Signing Algorithm: } S \equiv m^d \mod n$$

Since d is Alice's secret key, only Alice can perform this computation. She then sends S to Bob. Bob computes $v \equiv S^e \mod n$ using e, which is Alice's public key. If $v = m$, the output of the verification algorithm will be 1, indicating that the signature is verified. Otherwise, it will output 0. Note that Bob must know the message m for verification, which can be transferred using any encryption algorithm.

In RSA, during encryption, we compute $y \equiv x^e \mod n$, using the public key of the receiver, while during decryption, we compute $x \equiv y^d \mod n$, using the secret key of the receiver. However, in the RSA Signature Algorithm, during signing, we use the secret key of the signer (the person who signs), while during verification, we use the public key of the signer. This means that in the RSA Signature Algorithm, the computations are opposite to those in RSA encryption and decryption. In every Digital Signature Algorithm (DSA), signing is done using the secret key of the signer, and verification is done using the public key of the signer. .

5.2 RSA Encryption :-

RSA encryption is a widely used cryptographic algorithm for securing digital communication. Let's delve into RSA encryption once more.

In RSA encryption, there are two sets of keys: the public key (PK) and the secret key (SK). The public key consists of n and e, while the secret key consists of p, q, and d. These values are computed as follows:

$$n = p \cdot q$$

$$\phi(n) = (p-1) \cdot (q-1)$$

$$e \cdot d \equiv 1 \mod \phi(n)$$

The encryption process involves raising the message m to the power of e modulo n, resulting in the ciphertext c:

$$c = m^e \mod n$$

When encrypting two different messages, m_1 and m_2, with RSA, we obtain two ciphertexts, c_1 and c_2. Multiplying these ciphertexts together yields a ciphertext for the product of the original messages:

$$c_1 \cdot c_2 = (m_1 \cdot m_2)^e \mod n$$

This property allows for computation on encrypted data, enabling operations on the encrypted form of the messages without decryption.

However, addition of ciphertexts does not result in meaningful computation. But if a constant a is multiplied to the message m (with corresponding ciphertext c), the ciphertext for $a \cdot m$ can be obtained without actual encryption. This multiplication is achieved by multiplying c with a^e.

While RSA encryption does not support addition homomorphically, it facilitates multiplication homomorphically. Fully Homomorphic Encryption (FHE) algorithms enable both addition and multiplication operations on ciphertexts.

In the RSA Signature Algorithm, signatures on messages m_1 and m_2 are generated using the secret key:

$$s_1 = m_1^d \mod n$$
$$s_2 = m_2^d \mod n$$

Multiplying these signatures produces a signature for the product of the original messages:

$$s_1 \cdot s_2 = (m_1 \cdot m_2)^d \mod n$$

This operation allows for computation on authenticated data, where the signature on the product of messages can be verified by others.

To prevent signature forgery, signatures are often computed on the hash of the message instead of the message itself. This ensures that forging a signature on a product of messages is not possible without knowing the secret key. The verification process involves comparing the computed value with the hashed message. Only with the public key of the signer can the message be verified.