**RSA Algorithm**

* the RSA operation can't handle messages longer than the modulus size.

That means that if you have a 2048 bit RSA key, you would be unable to directly sign any messages longer than 256 bytes long

* In contrast, a cryptographical hash can take an arbitrarily long message, and 'compress' it into a short string, in such a way that we cannot find two messages that hash to the same value.

Hence, signing the hash is just as good as signing the original message; without the length restrictions we would have if we didn't use a hash.

Note :

* If an attacker tries to forge a signature for a different message M**′** without knowing the private key:
  + They can compute the padded hash of M**′** as Pad(Hash(M**′**))
  + However, they would need to find a value S**′** such that S**′e**≡Pad(Hash(M′))(modN).
* Without knowledge of the private key d, which is required for the modular exponentiation operation (S**′e**)d≡S**′**(modN), forging a valid signature is infeasible.
* **The process of creating a digital signature involves the following steps:**
* **Hashing**:
* The first step in creating a digital signature is to create a hash of the message or document that needs to be signed.
* This is done using a hash function, which produces a fixed-length output (the hash value) from an input of any size.
* **Padding**:
* The hash value is then padded to ensure its length matches the modulus size required by the RSA algorithm.
* Padding schemes like PKCS#1 (v1.5) or PSS are commonly used for this purpose.
* **Signing**:
* The padded hash value is then signed using the sender's private key.
* This involves raising the padded hash to the power of the private exponent modulo the public modulus
* the signature is typically applied to a digest of the original message rather than directly to the message itself
* **Verification**:

Upon receiving the signed message, the recipient can verify the signature's authenticity by:

* Recomputing the hash value of the original message.
* Verifying that the decrypted signature (using the signer's public key) matches this hash value.

**Overall**

* RSA is well-suited for digital signatures
* because it provides strong security and efficient performance. The security of RSA is based on the difficulty of factoring large prime numbers. In RSA, the private key is a pair of prime numbers, and the public key is a product of these primes. Because factoring the public key into its prime factors is considered a computationally difficult problem, it is infeasible for an attacker to deduce the private key from the public key.
* Furthermore, RSA is efficient because the signing process only involves modular exponentiation, which is a relatively fast operation. This makes it suitable for use in a wide range of applications, including digital certificates, secure email, and electronic commerce.
* Attaching the signature to the digest of the original message rather than the message itself offers several advantages:
* It allows for more efficient processing, as the hash value is typically much smaller than the original message.
* It ensures that the signature remains valid even if the message is too large to be efficiently processed or transmitted.
* It provides a consistent and standardized representation of the message for signature verification, regardless of the message's format or encoding.

**Salt Length**

* The salt length, in the context of RSA signatures, typically refers to the length of the random value (salt) added during the padding process before applying the signature.
* if SHA-256 is used as the hash function, then the salt length should be 32 bytes
* Current security standards typically recommend salt lengths equal to the output size of the hash function used in the signature scheme (e.g., 32 bytes for SHA-256)

**RSA-PSS**

PSS is randomized and will produce a different signature value each time (unless you use a zero-length salt).

**Why we cant extract PSS digest**

This is because PKCSV1\_5 uses a fixed padding format that allows you to identify the hash function and the digest value after decrypting the signature with the public key.

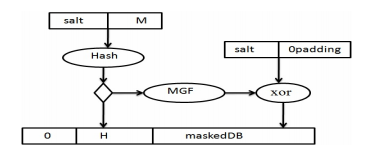
PSS,on the other hand, uses a variable padding format that depends on some parameters, such as the salt length and the mask generation function. These parameters need to be known beforehand to verify the signature, and they cannot be deduced from the signature value alone.

Even if I extracted the digest and tried dicitionariy attack by using know messages and apply hash on it and compare it to this digest , still the original message is padded with salt length

**The mask Generation function MGF**

* The mask has the same length as the RSA modulus minus the length of the hash.

If we are using (sha256 ) and key size 2048

And Since the mask needs to have the same length as the modulus minus the hash, it will be 2048 bits - 256 bits = 1792 bits (224 bytes).

RSA modulus length: 2048 bits (256 bytes)

Hash length (SHA256 output): 32 bytes

Mask length: 224 bytes

generates a pseudo random mask that is XORed with the hashed message to introduce randomness and enhance the security of the signature scheme.

**Salt**: This is a random value added during the padding process before signing. It's unique for each signature, adding randomness and preventing pre-computed attacks.

**Seed**: This is another random value used specifically by the MGF

**Importance of MGF:**

Prevents precomputed attacks: Without MGF, attackers could create tables of signature values for common messages. Adding randomness through MGF makes this impractical.

Strengthens against chosen-message attacks: By masking the digest, attackers cannot easily extract information about the private key even if they can feed specific messages to the signing process.

**Digital Signature Flow**

